Thinking in SwiftUI

By Chris Eidhof and Florian Kugler

Thinking in SwiftUI

Chris Eidhof

Florian Kugler

objc.io

Thinking in SwiftUI

Thinking in SwiftUI

Introduction

SwiftUI is a radical departure from UIKit, AppKit, and other object-oriented UI frameworks.

In SwiftUI, views are values instead of objects. Compared to how they're handled in object-oriented frameworks, view construction and view updates are expressed in an entirely different, declarative way. While this eliminates a whole category of bugs (views getting out of sync with the application's state), it also means you have to think differently about how to translate an idea into working SwiftUI code. The primary goal of this book is to help you develop and hone your intuition of SwiftUI and the new approach it entails.

SwiftUI also comes with its own layout system that fits its declarative nature. The layout system is simple at its core, but it can appear complicated at first. To help break this down, we explain the layout behavior of elementary views and view containers and how they can be composed. We also show advanced techniques for creating complex custom layouts.

Finally, this book covers animations. Like all view updates in SwiftUI, animations are triggered by state changes. We use several examples – ranging from implicit animations to custom ones – to show how to work with this new animation system.

What's Not in This Book

Since SwiftUI is a young framework, this book is not a reference of all the (platform-specific) Swift APIs. For example, we won't discuss how to use a navigation view on iOS, a split view on macOS, or a carousel on watchOS — especially since specific APIs will change and develop over the coming years. Instead, this book focuses on the concepts behind SwiftUI that we believe are essential to understand and which will prepare you for the next decade of SwiftUI development.

Acknowledgments

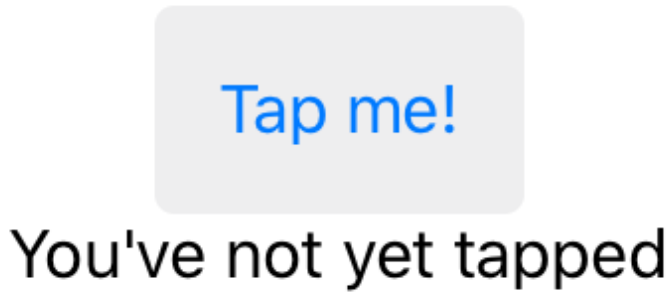
Thanks to Javier Nigro, Matt Gallagher, and Ole Begemann for your invaluable feedback on our book. Thanks to Natalye Childress for copy editing. Chris would like to thank Erni and Martina for providing a good place to write.

Overview

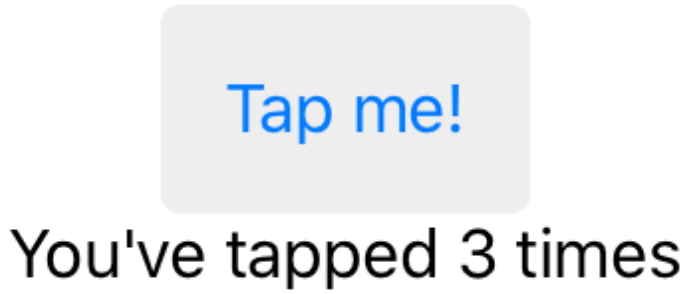
In this chapter, we'll give you an overview of how SwiftUI works and how it works differently from frameworks like UIKit. SwiftUI is a radical conceptual departure from the previous way of developing apps on Apple's platforms, and it requires you to rethink how to translate an idea you have in mind into working code.

We'll walk through a simple SwiftUI application and explore how views are constructed, laid out, and updated. Hopefully this will give you a first look at the new mental model that's required for working with SwiftUI. In subsequent chapters, we'll dive into more detail on each of the aspects described in this chapter.

We'll build a simple counter for our sample application. The app has a button to increase the counter, and below the button is a label. The label shows either the number of times the counter was tapped, or a placeholder if the button hasn't been tapped yet:



The example app in its launch state...



...and after tapping a few times

We strongly recommend following along by running and modifying the code yourself. Consider the following quote:

> *The only way to learn a new programming language is by writing programs in it. — Dennis Ritchie*

We believe this advice applies not just to programming languages, but also to complicated frameworks such as SwiftUI. And as a matter of fact, this describes our experience with learning SwiftUI.

View Construction

To construct views in SwiftUI, you create a tree of view values that describe what should be onscreen. To change what's onscreen, you modify state, and a new tree of view values is computed. SwiftUI then updates the screen to reflect these new view values. For example, when the user taps the counter button, we should increment our state and let SwiftUI rerender the view tree.

> *Note: At the time of writing, Xcode's built-in previews for SwiftUI, Playgrounds, and the simulator don't always work. When you see unexpected behavior, make sure to doublecheck on a real device.*

Here's the entire SwiftUI code for the counter application:

```
import SwiftUI

struct ContentView: View {
    @State var counter = 0
    var body: some View {
        VStack {
            Button(action: { self.counter += 1 }, label: {
                Text("Tap me!")
                    .padding()
                    .background(Color(.tertiarySystemFill))
                    .cornerRadius(5)
            })

            if counter > 0 {
                Text("You've tapped \(counter) times")
            } else {
                Text("You've not yet tapped")
            }
        }
    }
}
```

The ContentView contains a vertical stack with two nested views: a button, which increments the counter property when it's tapped, and a text label that shows either the number of taps or a placeholder text.

Note that the button's action closure does not change the tap count Text view directly. The closure doesn't capture a reference to the Text view, but even if it did, modifying regular properties of a SwiftUI view after it is presented onscreen will not change the onscreen presentation. Instead, we must modify the state (in this case, the counter property), which

causes SwiftUI to call the view's body, generating a new description of the view with the new value of counter.

Looking at the type of the view's body property, some View, doesn't tell us much about the view tree that's being constructed. It only says that whatever the exact type of the body might be, this type definitely conforms to the View protocol. The real type of the body looks like this:

```
VStack<
    TupleView<
        (
            Button<
                ModifiedContent<
                    ModifiedContent<
                        ModifiedContent<
                            Text,
                            _PaddingLayout
                        >,
                        _BackgroundModifier<Color>
                    >,
                    _ClipEffect<RoundedRectangle>
                >
            >,
            _ConditionalContent<Text, Text>
        )
    >
>
```

That's a huge type with lots of generic parameters — and it immediately explains why a construct like some View (an opaque type) is required for abstracting away these complicated view types. However, for learning purposes, it's instructive to look at this type in more detail.

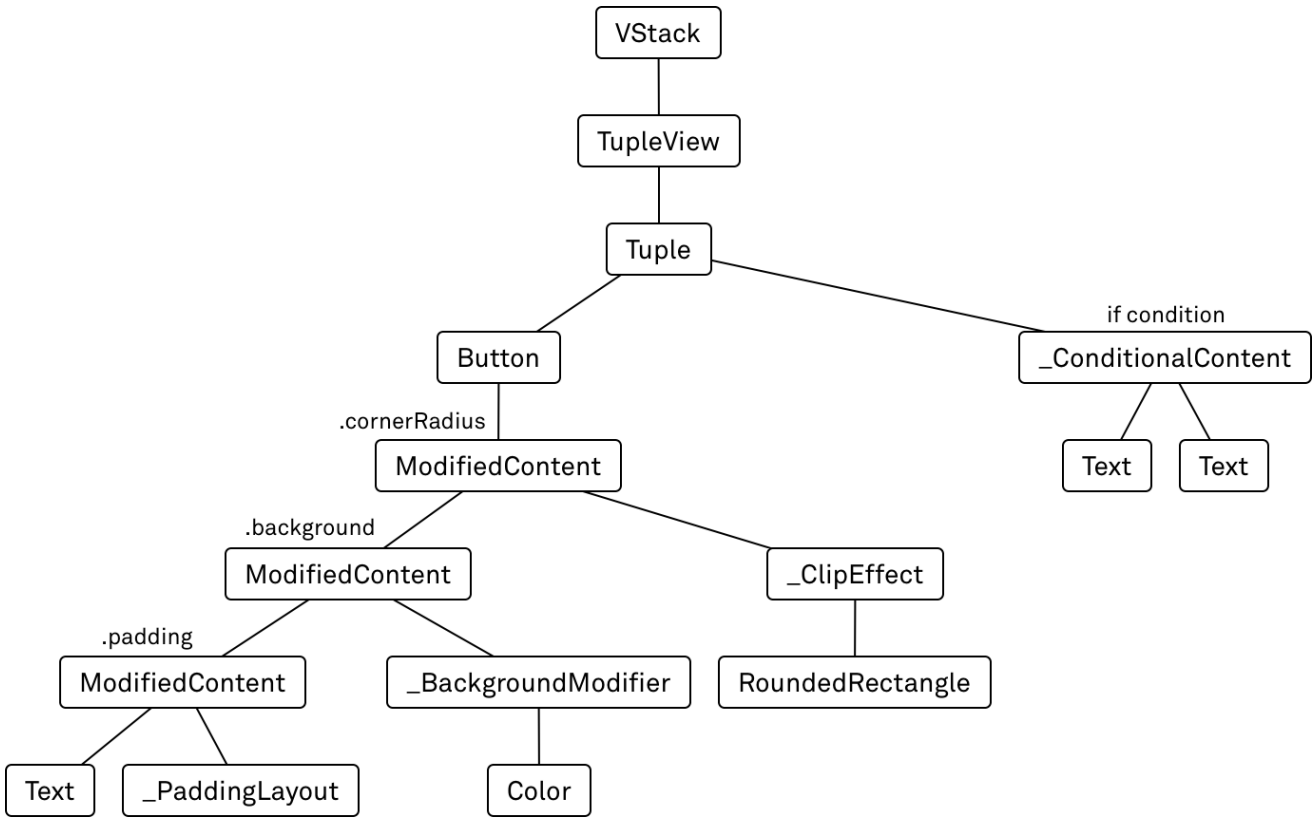
To inspect the underlying type of the body, we created the following helper function, which uses Swift's Mirror API:

```
extension View {
    func debug() -> Self {
        print(Mirror(reflecting: self).subjectType)
        return self
    }
}
```

The function is used like this to print out the view's type when body gets executed:

```
var body: some View {
    VStack { /*... */ }.debug()
}
```

Here's the same type visualized as a tree diagram:



The first thing to notice is that the type of the view constructed in the body property contains the structure of the entire view tree — not just the part that's onscreen at the moment, but all views that could ever be onscreen during the app's lifecycle. The if statement has been encoded as a value of type _ConditionalContent, which contains the type of both branches. You might wonder how this is even possible. Isn't if a language-level construct that's being evaluated at runtime?

To make this possible, SwiftUI leverages a Swift feature called function builders. As an example, the trailing closure after VStack is not a normal Swift function; it's a ViewBuilder (which is implemented using Swift's function builders feature). In view builders, you can only write a very limited subset of Swift: for example, you cannot write loops, guards, or if lets. However, you can write simple Boolean if statements to construct a view tree that's dependent on the app's current state — like the counter variable in the example above (see the section below for more details about view builders).

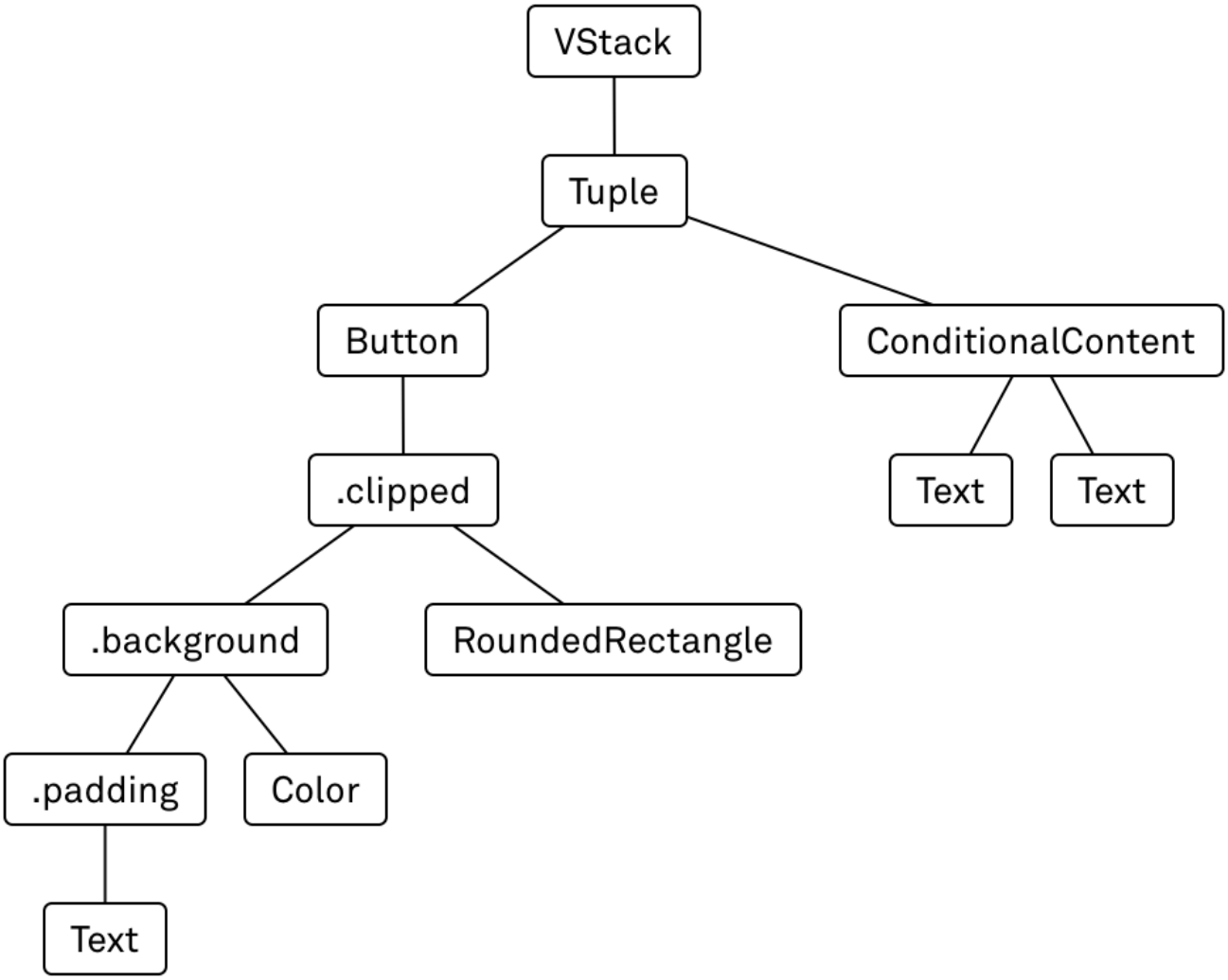
The advantage of the view tree containing the entire structure instead of just the currently visible structure is that it's more efficient for SwiftUI to figure out what has changed after a view update — but we'll get to view updates later in this chapter.

The second feature to highlight in this type is the deep nesting of ModifiedContent values. The padding, background, and cornerRadius APIs we're using on the button are not simply changing properties on the button. Rather, each of these method calls creates another layer in the view tree. Calling .padding() on the button wraps the button in a value of type ModifiedContent, which contains the information about the padding that should be applied. Calling .background on this value in turn creates another ModifiedContent value around the existing one, this time

adding on the information about the background color. Note that .cornerRadius is implemented by clipping the view with a rounded rectangle, which is also reflected in the type.

Since all these modifiers create new layers in the view tree, their sequence often matters. Calling .padding().background(...) is different than calling .background(...).padding(). In the former case, the background will extend to the outer edge of the padding; the background will only appear within the padding in the latter case.

In the rest of this book, we'll simplify the diagrams for readability, leaving out things like ModifiedContent. For example, here's the previous diagram, simplified:



View Builders

As mentioned above, SwiftUI relies heavily on view builders to construct the view tree. A view builder looks similar to a regular Swift closure expression, but it only supports a very limited syntax. While you can write any kind of *expression* that returns a View, there are very few *statements* you can write. The following example contains almost all possible statements in a view builder:

```
VStack {
    Text("Hello")
    if true {
        Image(systemName: "circle")
    }
    if false {
        Image(systemName: "square")
    } else {
        Divider()
    }
    Button(action: {}, label: {
        Text("Hi")
    })
}
```

The type of the view above is:

```
VStack<
    TupleView<(
        Text,
        Optional<Image>,
        _ConditionalContent<Image, Divider>,
        Button<Text>
    )>
>
```

Each statement in a view builder gets translated into a different type:

- A view builder with a single statement (for example, the button's label) evaluates to the type of that statement (in this case, a `Text`).
- An `if` statement without an `else` inside a view builder becomes an optional. For example, the `if true { Image(...) }` gets the type `Optional<Image>`.
- An `if/else` statement becomes a `_ConditionalContent`. For example, the `if/else` above gets translated into a `_ConditionalContent<Image, Divider>`. Note that inside view builders, it is perfectly fine to have different types for the branches of an `if/else` statement (whereas you can't return different types from the branches of an `if` statement outside of a view builder).
- Multiple statements get translated into a `TupleView` with a tuple that has one element for every statement. For example, the view builder we pass to the `VStack` contains four statements, and its type is:

  ```
  TupleView<(
      Text,
      Optional<Image>,
      _ConditionalContent<Image, Divider>,
      Button<Text>
  )>
  ```

At the moment, it is not possible to write loops or switches, declare variables, or use syntax such as `if let`. Most of these statements will be supported in the future, but at the time of writing, this support hadn't been implemented.

Compared to UIKit

When we talk about views or view controllers in UIKit, we refer to *instances* of the `UIView` or `UIViewController` *classes*. View construction in UIKit means building up a tree of view controllers and view *objects*, which can be modified later on to update the contents of the screen.

View construction in SwiftUI refers to an entirely different process, because there are no instances of view classes in SwiftUI. When we talk about views, we're talking about *values* conforming to the `View` *protocol*. These values describe what should be onscreen, but they do not have a one-to-one relationship to what you see onscreen like UIKit views do: view values in SwiftUI are transient and can be recreated at any time.

Another big difference is that in UIKit, view construction for the counter app would only be one part of the necessary code; you'd also have to implement an event handler for the button that modifies the counter, which in turn would need to trigger an update to the text label. View construction and view updates are two different code paths in UIKit.

In the SwiftUI example above, these two code paths are unified: there is no extra code we have to write in order to update the text label onscreen. Whenever the state changes, the view tree gets reconstructed, and SwiftUI takes over the responsibility of making sure that the screen reflects the description in the view tree.

View Layout

SwiftUI's layout system is a marked departure from UIKit's constraint- or frame-based system. In this section, we'll walk you through the basics, and we'll expand on the topic in the view layout chapter later in the book.

SwiftUI starts the layout process at the outermost view. In our case, that's the `ContentView` containing a single `VStack`. The layout system offers the `ContentView` the entire available space, since it's the root view in the hierarchy. The `ContentView` then offers the same space to the `VStack` to lay itself out. The `VStack` divides the available space by the number of its children, and it offers this space to each child (this is an oversimplification of how stacks divide up the available space between their children, but we'll come back to this in the layout chapter). In our example, the vertical stack will consult the button (wrapped in several modifiers) and the conditional text label below it.

The first child of the stack (the button) is wrapped in three modifiers: the first (`cornerRadius`) takes care of clipping the rounded corners, the second (`background`) applies a background color,

and the third (padding) adds padding. The first two modifiers don't modify the proposed size. However, the padding modifier will take the space it's offered by its parent, subtract the padding, and offer the now slightly reduced space to the button. The button in turn offers this space to its label, which responds with the size it really needs based on the text. The button takes on the size of the text label, the padding modifier takes on the size of the button plus the padding, the two other modifiers just take on the size of their children, and the final size is communicated up to the vertical stack.

After the vertical stack has gone through the same process with its second child, the conditional text label, it can determine its own size, which it reports back to its parent. Recall that the vertical stack was offered the entire available space by the ContentView, but since the stack needs much less space, the layout algorithm centers it onscreen by default.

At first, laying out views in SwiftUI feels a bit like doing it in UIKit: setting frames and working with stack views. However, we're never setting a frame property of a view in SwiftUI, since we're just describing what should be onscreen. For example, adding this to the vertical stack looks like we're setting a frame, but we're not:

```
VStack {
    // ...
}.frame(width: 200, height: 200)
```

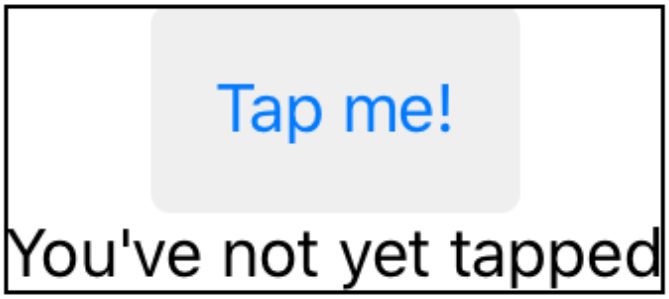
When calling .frame, all we're doing is wrapping the vertical stack in another modifier (which itself conforms to the View protocol). The type of the view's body now has changed to:

```
ModifiedContent<VStack<...>, _FrameLayout>
```

This time, the entire space onscreen will be offered to the frame modifier, which in turn will offer its (200, 200) space to the vertical stack. The vertical stack will still end up the same size as before, being centered within the (200, 200) frame modifier by default. It's important to keep in mind that calling APIs like .frame and .offset does not modify properties of the view, but rather wraps the view in a modifier. This really makes a difference when you try to combine these calls with other things like backgrounds or borders.

Let's say we want to add a background to the (200, 200) frame we've specified on the vertical stack. At first, we might try something like this:

```
VStack {
    // ...
}
.border(Color.black)
.frame(width: 200, height: 200)
```



Perhaps surprisingly, the border will only show up around the minimal area of the vertical stack instead of in the 200-by-200 area we've specified. The reason is that the .border call added an overlay modifier around the vertical stack, which just takes on the size of its children. If we want to draw the border around the entire (200, 200) area, we'd have to reverse the calls:

```
VStack {
    // ...
}
.frame(width: 200, height: 200)
.border(Color.black)
```

This time, the frame modifier gets wrapped in an overlay modifier, so the overlay (the border) now takes on the size of its child: the frame modifier with the fixed 200-by-200 size we've specified. This demonstrates how, while it might seem like a theoretical issue, the order of modifiers quickly becomes important to anyone who writes more than a short example.

> *Putting borders around views can be a helpful debugging technique to visualize the views' frames.*

In SwiftUI, you never force a view to take on a particular size directly. You can only wrap it in a frame modifier, whose available space will then be offered to its child. As we'll see in the view layout chapter, views can define their ideal size (similar to UIKit's `sizeThatFits` method), and you can force a view to become its ideal size.

Implementing layouts where the layout of a parent is dependent on the size of its children (for example, if you wanted to reimplement `VStack`) is a bit more complicated and requires the use of geometry readers and preferences, which we'll cover later in this book.

View Updates

Now that the views have been constructed and laid out, SwiftUI displays them onscreen and waits for any state changes that affect the view tree. In our example, tapping the button triggers such a state change, since this modifies the `@State` property `counter`.

Properties that need to trigger view updates are marked with the `@State`, `@ObservedObject`, or `@EnvironmentObject` property attributes (among others we'll discuss in the next chapter). For now, it's enough to know that changes to properties marked with any of these attributes will cause the view tree to be reevaluated.

When the `counter` property in our example is changed, SwiftUI will access the `body` property of the content view again to retrieve the view tree for the new state. Note that the type of the view tree (the complicated type hidden behind the `some View` discussed above) does not change. In fact, it cannot change, since the type is fixed at compile time. Therefore, the only things that can change are properties of the views (like the text of the label showing the number of taps) and which branch of the `if` statement is taken. The static encoding of the view tree's structure in the view type has important performance advantages, which we'll discuss in detail in the next chapter.

It's important to keep in mind that changing state properties is the *only* way to trigger a view update in SwiftUI. We cannot do what is common in UIKit, i.e. modify the view tree in an event handler. This new way of doing things eliminates a whole category of common bugs — views getting out of sync with the application's state — but it requires us to think differently: we have to model the application state explicitly and describe to SwiftUI what should be onscreen for each given state.

Takeaways

- SwiftUI views are values, not objects: they are immutable, transient descriptions of what should be onscreen.
- Almost all methods we call on a view (like `frame` or `background`) wrap the view in a modifier. Therefore, the sequence of these calls matters, unlike with most `UIView` properties.
- Layouts proceed top down: parent views offer their available space to their children, which decide their size based on that.
- We can't update what's onscreen directly. Instead, we have to modify state properties (e.g. `@State` or `@ObservedObject`) and let SwiftUI figure out how the view tree has changed.

View Updates

In the first chapter, we looked at how the view tree gets constructed in SwiftUI and how it's updated in response to state changes. In this chapter, we'll go into detail about the view update process and explain what you need to know to write clean and efficient view update code.

Updating the View Tree

In most object-oriented GUI applications — such as UIKit apps and the Document Object Model (DOM) apps in a browser — there are two view-related code paths: one path handles the initial construction of the views, and the other path updates the views when events happen. Due to the separation of these code paths and the manual updating involved, it's easy to make mistakes: we might update the views in response to an event but forget to update the model, or vice versa. In either case, the view gets out of sync with the model and the app might exhibit undefined behavior, be stuck at a dead end, or even crash. It's possible to avoid these bugs with discipline and testing, but it's not easy.

In AppKit and UIKit programming, there have been a number of techniques that try to solve this problem. AppKit uses the Cocoa bindings technology, a two-way layer to keep the models and views in sync. In UIKit, people use techniques like reactive programming to (mostly) unify both code paths.

SwiftUI has been designed to entirely avoid this category of problems. First, there is a single code path that constructs the initial views and is also used for all subsequent updates: the view's `body` property. Second, SwiftUI makes it impossible to bypass the normal view update cycle and to modify the view tree directly. Triggering a reevaluation of the `body` property is the only way to update what's onscreen in SwiftUI.

When the framework needs to render an updated view tree, the simplest implementation could just throw everything away and redraw the screen from scratch. However, this is inefficient, as underlying view objects (for example, a `UITableView`) might be expensive to recreate. Even worse, recreating these view objects could mean losing view state such as the scroll position, the current selection, and so on.

To solve this problem, SwiftUI needs to know which of the underlying view objects need to be changed, added, or removed. In other words: SwiftUI needs to compare the previous view tree value (the result of evaluating `body`) with the current view tree value (the result of reevaluating `body` after the state changes). SwiftUI has a bunch of tricks up its sleeve to optimize this process so that even changes in large view trees can be performed efficiently.

In an ideal world, we wouldn't have to know anything about this process to work with SwiftUI. However, these implementation details always bleed through in one form or another. And it's helpful to have a basic understanding of how SwiftUI performs view updates so that, if nothing else, our code does not get in the way of SwiftUI doing its job efficiently. To explore this process, we'll start with a slightly simplified version of the example we used in the previous chapter:

```
struct ContentView: View {
    @State var counter = 0
    var body: some View {
        VStack {
            Button("Tap Me") { self.counter += 1 }
            if counter > 0 {
                Text("You've tapped\(counter)times")
            }
        }
    }
}
```

The first time our view gets rendered, the ContentView's body property is executed, which results in a value conforming to the View protocol (as the type some View of body indicates). However, looking at the exact type, we can see that all the views in the tree — whether they're currently visible or not — have been encoded in the type of the value:

```
VStack<TupleView<(Button<Text>, Text?)>>
```

Although the text label within the if condition is not yet onscreen (since counter is still zero), it is present from the beginning in the type of the view tree as Text?. When the counter property changes, the body will be executed again, resulting in a new view tree value that has the exact same type. The Text? value will always be there — it can either be nil if the label should not be onscreen, or it can contain a Text view. SwiftUI relies on the elements in the view tree being the same from update to update, and since the structure of the view tree is encoded within the type system, this invariant is guaranteed by the compiler.

Why does the view tree have to have the same structure every time? Isn't it wasteful to encode the entire structure of all possible view trees of the program in this value each time?

Each time the application's state changes and the view tree gets recomputed, SwiftUI has to figure out what has changed between the previous tree and the new one, in order to efficiently update the display without reconstructing and redrawing everything from scratch. If the old tree and the new tree are guaranteed to have the same structure, this task is much easier and more efficient.

Tree diffing algorithms, i.e. algorithms that can compare two trees with different structures, have a complexity in the order of O(n^3). This means that if we'd have to perform 1,000 operations to diff a 10-element tree, we'd have to perform 1,000,000 operations to diff a 100-element tree. To manage this complexity, frameworks like React use a heuristic diffing algorithm with O(n) complexity, which trades precision of the diff for performance: the

algorithm might cause larger parts of the tree than are strictly necessary to be recreated, and the developer might have to provide hints about which parts of the tree are stable from update to update to counter this effect.

SwiftUI takes a different approach to this problem: since the structure of the view tree is always the same across updates, it doesn't need to perform a full tree diff to begin with. SwiftUI can simply walk the old tree and the new tree in unison, knowing that the structure is still the same. For example, even when our `counter` property is still zero, SwiftUI knows by the `Text?` element in the tree that there might be a text label in a different state. When we increase the counter from zero to one, it can compare the items in the tree and will notice that a `Text` view is present where there was previously a `nil`. When `counter` changes from one to two, a `Text` view will be present in both trees, and SwiftUI can compare the text properties of both to determine whether or not the contents have to be rerendered.

Similarly, SwiftUI knows there will always be a `VStack` at the top level and a `TupleView` with two elements inside. It doesn't need to account for the case that an entirely different view might show up at the root level — it only needs to compare the properties of the old and the new `VStack` that could have changed (e.g. the alignment or spacing of the stack).

At this point you might wonder again: even if the view tree comparison is much faster than performing a full tree diff, isn't it still wasteful to recreate and compare the entire view tree value each time?

It turns out that SwiftUI is smart about this as well. To investigate when a view body is executed, we can use the `debug` helper method from the first chapter or simply insert a print statement. Each time the `counter` variable changes in our example, the `ContentView`'s body is executed, since `counter` is declared with the `@State` attribute (we'll look at the `@State` and other view update-triggering attributes in more detail below). If we split the content view in our example into two views and pass the value of `counter` from the `ContentView` to the new `LabelView`, the entire view tree still gets recomputed on each update:

```
struct LabelView: View {
    let number: Int
    var body: some View {
        print("LabelView")
        return Group {
            if number > 0 {
                Text("You've tapped\(number)times")
            }
        }
    }
}

struct ContentView: View {
    @State var counter = 0
    var body: some View {
        print("ContentView")
        return VStack {
```

```
            Button("Tap me!") { self.counter += 1 }
            LabelView(number: counter)
        }
    }
}

// Console output on each update:
// ContentView
// LabelView
```

SwiftUI keeps track of which views use which state variables: it knows that the content view's body uses the @State variable counter during its construction, since we access counter to pass it as a parameter to LabelView (in the section on property wrappers below, we'll talk more about how dependency tracking works). Because of this, it reexecutes the content view's body when counter changes.

If we divide the view in a way so that the counter property is only used in a subview, the situation changes:

```
struct LabelView: View {
    @State var counter = 0
    var body: some View {
        print("LabelView")
        return VStack {
            Button("Tap me!") { self.counter += 1 }
            if counter > 0 {
                Text("You've tapped\(counter)times")
            }
        }
    }
}

struct ContentView: View {
    var body: some View {
        print("ContentView")
        return LabelView()
    }
}

// Initial console output:
// ContentView
// LabelView

// Console output on each update:
// LabelView
```

SwiftUI only reexecutes the body of a view that *uses* a @State property (the same holds true for the other property wrappers, such as @ObservedObject and @Environment). Therefore, only the label view's body is executed when counter changes. In theory, this also invalidates the

entire subtree of the label view, but SwiftUI optimizes this process as well: it avoids reexecuting a subview's body when it knows the subview hasn't changed.

We can achieve a similar effect while declaring the @State property in the content view by using a binding in the label view. When counter changes, the LabelView's body gets reexecuted, but the ContentView's body does not:

```
struct LabelView: View {
    @Binding var number: Int
    var body: some View {
        print("LabelView")
        return Group {
            if number > 0 {
                Text("You've tapped\(number)times")
            }
        }
    }
}

struct ContentView: View {
    @State var counter = 0
    var body: some View {
        print("ContentView")
        return VStack {
            Button("Tap me!") { self.counter += 1 }
            LabelView(number: $counter)
        }
    }
}

// Initial console output:
// ContentView
// LabelView

// Console output on each update:
// LabelView
```

A binding is essentially a getter and a setter for a captured variable. SwiftUI's property wrappers (@State, @ObservedObject, etc.) have a corresponding binding that you can access by using the $ prefix. (In property wrapper terminology, the binding is called a *projected value*.) As in the previous example, SwiftUI keeps track of which views use which state variables: it knows that ContentView doesn't use counter when rendering the body, but that LabelView does (indirectly through the binding). Therefore, changes to the counter property only trigger a reevaluation of the LabelView's body.

> *SwiftUI's bindings and Cocoa bindings are different technologies that serve a similar purpose. Both SwiftUI and Cocoa provide two-way bindings, but their implementations are very different.*

Dynamic View Trees

If SwiftUI encodes the structure of the view tree in the type of the root view, how can we build view trees that are not static? We clearly need ways to dynamically swap out parts of the view tree or insert views that we didn't know about at compile time.

SwiftUI has three different mechanisms for building a view tree with dynamic parts:

1. if/else conditions in view builders
2. ForEach
3. AnyView

Each of these mechanisms has specific properties and capabilities, so let's go through them one by one.

if/else conditions in view builders are the most restrictive option for dynamically changing what's onscreen at runtime. The branches of an if/else are fully encoded in the type of the view (as _ConditionalContent): it's clear at compile time that the view onscreen will come from either the if branch or the else branch. In other words, if/else conditions allow us to hide or show views at runtime based on a condition, but we have to decide the types of the views at compile time. Likewise, an if without an else is encoded as an optional view that only displays when the condition is true.

Within ForEach, the number of views can change, but they all need to have the same type. ForEach is most commonly used with Lists (similar to a table view in UIKit). The number of items in a list is often based upon model data and cannot always be known at compile time:

```
struct ContentView: View {
    var body: some View {
        ForEach(1...3, id: \.self) { x in
            Text("Item\(x)")
        }
    }
}

// Type: ForEach<ClosedRange<Int>, Int, Text>
```

ForEach has three parameters that correspond directly to three generic parameters in its type. We'll cover the most verbose initializer, but there are two other convenience initializers as well.

The first parameter of ForEach is the collection of data that's displayed. The type of the first parameter is also the first generic parameter of ForEach — in our case, that's ClosedRange<Int>, but it could be any RandomAccessCollection.

The second parameter is a keypath that specifies which property should be used to identify an element (the collection's elements have to either conform to the Identifiable protocol, or we have to specify a keypath to an identifier). We use the element itself as the identifier by specifying the identity keypath, \.self. Therefore, ForEach's second generic parameter — the type of the identifier — is Int.

The third parameter of ForEach constructs a view from an element in the collection. The type of this view is the third generic parameter of ForEach (in our case, we render a Text).

Since ForEach requires each element to be identifiable, it can figure out at runtime (by computing a diff) which views have been added or removed since the last view update. While this is more work than evaluating if/else conditions for dynamic subtrees, it still allows SwiftUI to be smart about updating what's onscreen. Also, uniquely identifying elements helps with animations, even when the properties of an element change.

Lastly, AnyView is a view that can be initialized with any view to erase the wrapped view's type. This means that an AnyView can contain completely arbitrary view trees with no requirement that their type be statically fixed at compile time. While this gives us a lot of freedom, AnyView should be something we only use as a last resort. This is because using AnyView takes away essential static type information about the view tree that otherwise helps SwiftUI perform efficient updates. We'll look at this in more detail in the next section.

Efficient View Trees

SwiftUI relies on static information about the structure of the view tree to perform efficient comparisons of view tree values between updates. View builders help us construct these trees and capture the static structure in the type system. However, sometimes we're not inside a view builder. Consider the following example, which will not compile:

```
struct LabelView: View {
    @Binding var counter: Int
    var body: some View {
        if counter > 0 {
            Text("You've tapped\(counter)times")
        }
    }
}
```

The body property of a View doesn't use the view builder syntax, and the if condition is a normal Swift if condition. Swift doesn't know what the type of the view above is: if the condition is true, it'll render a text, but what should it render if the condition is false? What we really want is to display the Text only when the condition is true. We can wrap our view inside a Group (which takes a view builder as the trailing closure):

```
struct LabelView: View {
    @Binding var counter: Int
```

```
    var body: some View {
        Group {
            if counter > 0 {
                Text("You've tapped\(counter)times")
            }
        }
    }
}
```

The if condition inside the view builder now gets translated into an optional Text view, and the compiler is satisfied. A similar problem happens when we have an if/else condition with different types. In normal Swift code, this is not allowed:

```
var body: some View {
    if counter > 0 {
        return Text("You've tapped\(counter)times")
    } else {
        return Image(systemName: "lightbulb")
    }
}
// error: Function declares an opaque return type, but the return statements
// in its body do not have matching underlying types
```

Although both Text and Image conform to the View protocol, we can't return values of different concrete types from different branches in the body. We can return anything that conforms to View, but then we need to return the same thing from all branches. Just like before, we can solve this by wrapping the entire body in a group. Since the parameter of the group's initializer is a view builder closure, the if/else condition gets encoded as part of the view tree. The resulting type is:

```
Group<_ConditionalContent<Text, Image>>
```

Using the group's view builder to encapsulate the conditional returning different types is a good solution for this problem, as it preserves all the information about possible view trees: SwiftUI now knows that there will be either a text label or an image.

> *Instead of using a Group, you can also apply the @ViewBuilder attribute to the computed body property. However, at the time of writing (Xcode 11.3), this didn't yet work consistently.*

Here's a bad solution for this kind of problem:

```
var body: some View {
    if counter > 0 {
        return AnyView(Text("You've tapped\(counter)times"))
    } else {
        return AnyView(Image(systemName: "lightbulb"))
    }
}
```

By wrapping the returned views in an AnyView, we satisfied the requirement of the compiler to return the same concrete type from all branches. However, we also erased all information from the type system about what kind of views can be present in this place. SwiftUI now has to do more work to determine how to update the views onscreen when the counter property changes. Since an AnyView could literally be any kind of view, rather than having to check a flag on the _ConditionalContent value, SwiftUI has to compare the actual values within the AnyView to determine what has changed.

Another possible performance concern has to do with how and where we use state properties that trigger view updates. We saw in the previous section that SwiftUI knows where @State properties (or one of the other state property wrappers we'll discuss below in more detail) are being used: once we factored out the LabelView and passed the counter as a binding, instead of reconstructing the entire view tree, SwiftUI only executed the LabelView's body when the counter changed. In general terms: SwiftUI tracks which views use which state properties, and on view updates, SwiftUI only executes the bodies of the views that could actually have changed.

We should take advantage of this when building our views because it matters where we place state properties and how we use them. We can best make use of SwiftUI's smart view tree updates when we place state properties as locally as possible. Conversely, it's the worst possible option to represent all model state with one state property on the root view and pass all data down the view tree in the form of simple parameters, as this will cause many more views to be needlessly reconstructed.

State Property Attributes

The only way to cause SwiftUI to update the views is to change the source of truth, i.e. state properties like the ones declared with @State in our examples. All the property wrappers SwiftUI uses for triggering view updates conform to the DynamicProperty protocol. Looking up this protocol in the documentation reveals the following conforming types:

- Binding
- Environment
- EnvironmentObject
- FetchRequest
- GestureState
- ObservedObject
- State

Below we'll go into more detail about the most common wrappers: @State, @Binding, and @ObservedObject. We'll discuss the environment-based wrappers in the environment chapter.

Property Wrappers

When SwiftUI was released, two new features were added to Swift in order to enable concise and readable SwiftUI programs: function builders and property wrappers. Specifically, SwiftUI uses view builders (discussed in the previous chapter) and a set of built-in property wrappers, e.g. @State, @Environment, and @Binding.

While property wrappers are a huge topic on their own, we'll show how they help make SwiftUI more readable. Consider the example from earlier in this chapter that uses the @State property wrapper:

```
struct ContentView: View {
    @State var counter = 0
    var body: some View {
        return VStack {
            Button("Tap me!") { self.counter += 1 }
            LabelView(number: $counter)
        }
    }
}
```

If we remove the @State prefix, it's no longer possible to mutate counter inside body, as body is not a mutating function or property. To better understand what property wrappers do, let's revisit the above example, this time without using property wrapper syntax:

```
struct ContentView: View {
    var counter = State(initialValue: 0)
    var body: some View {
        return VStack {
            Button("Tap me!") { self.counter.wrappedValue += 1 }
            LabelView(number: counter.projectedValue)
        }
    }
}
```

First, we have to now initialize our counter with an explicit initializer on State, which is a struct defined in SwiftUI and marked as @propertyWrapper. Second, it's still not possible to modify the counter variable itself inside the body (because body is non-mutating). However, State does define a wrappedValue property that's marked as nonmutating set, which means we're allowed to modify that property inside a method or property that isn't mutating, such as body. Finally, instead of passing $counter to our LabelView, we now pass counter.projectedValue, which is of type Binding<Int>.

When we run both the code snippets above, they're equivalent to one another. However, the example with property wrappers has less visual noise: we can directly initialize and modify the counter variable as if it were an Int (under the hood, the wrappedValue is modified, but we

don't have to worry about spelling that out). Likewise, we can use `$counter` instead of accessing the binding ourselves.

The `State` type also enables dependency tracking. When a view's `body` accesses the `wrappedValue` of a `State` variable, a dependency is added between that view and the `State` variable. This means SwiftUI knows which views to update when the `wrappedValue` changes. Dependency tracking is not part of property wrappers themselves, but the simplified syntax of property wrappers makes it invisible to the programmer.

While this section used `State` as the example, the other property wrappers in SwiftUI have the same syntactic benefits. Each of them also provides dependency tracking.

For more information on property wrappers, the Swift Evolution proposal on this topic contains the full specification, and WWDC 2019 Session 415 shows how to use property wrappers when designing your own APIs.

State and Binding

Of all the property wrappers, `@State` is the easiest to use when experimenting with SwiftUI: we simply write `@State` in front of a property and use it as normal. Each time we change the property, a view update will be triggered. `@State` is great for representing local view state, such as state we might have to track in a small component.

As an example, we'll build a simple circular knob, similar to those used in audio programs. To start, here's a simple knob component that contains a regular property for the current value (we left out the `KnobShape` definition; it can be found in the accompanying code). When the value is `0`, the knob is displayed as a circle with a small pointer pointing to the top. As the value gets closer to `1`, we rotate the knob by almost a full turn:

```
struct Knob: View {
    // should be between 0 and 1
    var value: Double
    var body: some View {
        KnobShape()
            .fill(Color.gray)
            .rotationEffect(Angle(degrees: value * 330))
    }
}
```

In another view, we can now use our knob. For example, we could have some local state that's controlled by a slider. As the slider changes, the knob automatically gets rerendered:

```
struct ContentView: View {
    @State var volume: Double = 0.5
    var body: some View {
        VStack {
```

```
            Knob(value: volume)
                .frame(width: 100, height: 100)
            Slider(value: $volume, in: (0...1))
        }
    }
}
```

Unlike the knob (which uses volume directly), the slider is configured with $volume, which is of type Binding<Double>. When the slider first draws itself, it needs an initial value (provided by the binding). Once the user interacts with the slider, it also needs to write the changed value back. The slider itself doesn't really care about where that Double comes from, where it's stored, or what it represents: it just needs a way to read and write a Double, which is exactly what a Binding<Double> provides.

Let's add a simple interaction to our knob. When the knob is tapped, we want to toggle the value between 0 and 1. This means we need a way to communicate our new value back to the content view. While we could use a binding here, we'll first experiment with recreating the functionality of a binding with a simple callback function:

```
struct Knob: View {
    var value: Double // should be between 0 and 1
    var valueChanged: (Double) -> ()

    var body: some View {
        KnobShape()
            .fill(Color.gray)
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture {
                self.valueChanged(self.value < 0.5 ? 1 : 0)
            }
    }
}
```

When we configure the knob, we now need to pass in a callback function that sets self.volume to the new value. The state change will trigger a rerendering, and both the slider and knob will display the new value:

```
Knob(value: volume, valueChanged: { newValue in
    self.volume = newValue
})
```

While this works (and is a good exercise to understand what bindings do behind the scenes), using a binding is clearly more elegant:

```
struct Knob: View {
    @Binding var value: Double // should be between 0 and 1

    var body: some View {
        KnobShape()
            .fill(Color.gray)
```

```
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture {
                self.value = self.value < 0.5 ? 1 : 0
            }
    }
}
```

With a binding, it's easier to create the knob:

```
Knob(value: $volume)
```

We typically start out writing a custom control using @State variables so that it's quicker to prototype. However, once we need to store and observe that state outside of the control, we simply change @State to @Binding, which is an action that doesn't require any other changes to the code.

Note that we don't need to create a binding to the entire state value; we can also create a binding to a property of a state value. If we had a state struct that contained multiple properties, we could initialize the knob as Knob(value: $state.volume). Or if we had an array of volumes, we could write Knob(value: $volumes[0]).

Similar to the @State property wrapper, there's also @GestureState, which is a special variant for gesture recognizers. Gesture state properties are initialized with an initial value and then get updated while the gesture is ongoing. Once the gesture has finished, the gesture state property is automatically reset to its initial value.

ObservedObject

In almost all real-world applications, we want to use our own model objects. To make our model object observable from SwiftUI, its class needs to conform to ObservableObject (a protocol defined in the Combine framework).

For example, let's consider building a simple clock. We need some way to tell SwiftUI to rerender our view tree every second. To do this, we can create a CurrentTime class that creates and schedules a timer. We'll also expose a single property — now — that contains the current date and time:

```
final class CurrentTime: ObservableObject {
    @Published var now: Date = Date()
    let interval: TimeInterval = 1
    private var timer: Timer? = nil

    // ...
}
```

The only requirement of the ObservableObject protocol is an implementation of objectWillChange, which is a publisher that emits before the object changes. By using

@Published in front of the now property, the framework will create an implementation of objectWillChange for us that emits whenever now changes.

To start the timer, we'll add a start method to CurrentTime (for the sake of brevity, we'll leave out the stop method and deinit):

```
// ...
func start() {
    guard timer == nil else { return }
    now = Date()
    timer = Timer.scheduledTimer(withTimeInterval: interval, repeats: true) {
        [weak self] _ in
        self?.now = Date()
    }
}
// ...
```

To use our new class, we have to create a property and mark it as @ObservedObject:

```
struct TimerView: View {
    @ObservedObject var date = CurrentTime()

    var body: some View {
        Text("\(date.now)")
    }
}
```

Now we still need a way to call start. It's tempting to just call start in the initializer of CurrentTime, but in general, it's a bad idea to start expensive or long-running work in an ObservableObject before having any observers. In a simple app like this, this works fine: when the view tree is created, our CurrentTime is also created, and it automatically starts updating itself. However, consider the following example:

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            NavigationLink(destination: TimerView(), label: {
                Text("Go to timer")
            })
        }
    }
}
```

The app above displays a root view with a button that says "Go to timer," and once the user taps the button, it navigates to our TimerView. However, during initial construction, the entire view tree is constructed, including the TimerView. This means that our CurrentTime value is also constructed, and if we call start from the initializer, the timer will start updating even if we don't display the view at all.

The simplest way around this is calling start when the TimerView appears:

```
struct TimerView: View {
    @ObservedObject var date = CurrentTime()

    var body: some View {
        Text("\(date.now)")
            .onAppear { self.date.start() }
            .onDisappear { self.date.stop() }
    }
}
```

When creating other observable objects, keep in mind that they will be created on view construction. For example, when a screen in an app has to load a lot of images from the network, we typically want to start these requests when the images are needed and not when the view tree is constructed.

> *If we forget the @ObservedObject prefix, our code will mostly still work: we can access the object's properties and call methods on it. However, SwiftUI will not subscribe to the objectWillChange notifications, and views will not get rerendered when the state changes.*

Other Dynamic View Properties

So far, we have looked at @State, @GestureState, @Binding, and @ObservedObject. There are a few more though. First, there's @Environment and @EnvironmentObject. These property wrappers are used to observe values or objects from the environment. The next chapter goes into detail about SwiftUI's concept of the environment, and we will look at these two property wrappers in detail there.

There's also a property wrapper specifically for observing the results of a Core Data fetch request: @FetchRequest. We can initialize this with an NSFetchRequest, and it automatically updates the view when the data changes. Think of it as SwiftUI's version of NSFetchedResultsController.

Takeaways

- With SwiftUI, we can't manipulate the view tree directly. Instead, we change state, which causes the view tree to be reevaluated.
- The most commonly used state property wrappers are @State, @Binding, and @ObservedObject. They're used to trigger view updates in response to state changes.
- We need to access state properties within a view's body to specify the view tree that matches the current state.

- Large parts of the view tree, and not just the parts that are currently onscreen, are constructed upfront. Therefore, we must avoid doing expensive or unnecessary work on view construction.

Exercises

To practice the concepts from this chapter, you'll build a simple app that displays a list of photo metadata, and when you tap on a list item, it should display the corresponding photo. There are a few challenges to this exercise:

- You have to load the data from the network in such a way that SwiftUI updates the view tree once the data has loaded.
- For both screens (the list view and the photo view), you have to implement two states — one for while the data is loading, and one for when the data is present.
- Because the entire view tree is constructed up front, you need to start loading the data once a view appears onscreen, and not once it's constructed.

You can find the solution in the solutions appendix at the end of this book, or you can download the full project from GitHub.

Step 1: Load the Metadata

You will load the data from https://picsum.photos/v2/list. So the first step is to create a new `Photo` struct that matches the response of this API. You'll need fields for the ID, author, width, height, URL, and download URL. The struct needs to conform to `Codable` so that you can use `JSONDecoder` later on. Here's a sample of what the JSON looks like:

```
[
   {
      "id": "0",
      "author": "Alejandro Escamilla",
      "width": 5616,
      "height": 3744,
      "url": "https://unsplash.com/photos/yC-Yzbqy7PY",
      "download_url": "https://picsum.photos/id/0/5616/3744"
   },
   ...
]
```

Step 2: Create an ObservableObject

Create a new class called `Remote` that conforms to the `ObservableObject` protocol. This will store the URL, along with a function to turn the data from the network into the expected type,

e.g. an array of photos (this means Remote needs to be generic over this type). This class also has a property to store the loaded data, which is nil initially. Finally, add a load method that loads the data from the network using URLSession.

Step 3: Display the List

Create a view that displays the author names in a List. For each Photo value, display the author as the list item's text. If you didn't already do so, make your Photo struct conform to Identifiable so that you can use it with ForEach. Use the Remote object to load the array of photos from the network, and display a placeholder while the data is loading.

Step 4: Display the Image

Wrap the list view in a NavigationView, and wrap the author name in a NavigationLink. As the destination of the navigation link, create a new PhotoView that's initialized with the photo's download_url. In the PhotoView, display the photo (or a placeholder while the photo's loading). By combining resizable() and aspectRatio, you can make the image fit the screen (if you need some help with this, the view layout chapter explains how resizable and aspectRatio work).

One common mistake is to start loading the image when the view is being constructed. Make sure that it only loads when the view is onscreen!

> *Note that SwiftUI previews don't work correctly when loading content asynchronously from the network. If you want to use previews, the simplest option is to, for the time being, insert some static content instead of the dynamically loaded content. For example, you could put a Rectangle with a specific aspect ratio into the PhotoView in place of the image view.*
>
> *Alternatively, you could implement a more sophisticated approach where you abstract away the resource loading with a protocol and inject either the network loader or a static file loader into the views, in order to make them work more smoothly with previews.*

Bonus Exercises

You can expand this exercise in multiple ways. For example:

- Wrap UIActivityIndicatorView to display a native activity indicator instead of a static placeholder.
- Make sure the placeholder in PhotoView has the same aspect ratio as the image.
- Show image thumbnails in the list. Avoid loading the same thumbnail image multiple times when the list's cells are reconstructed.

- Instead of manually triggering the resource loading in `onAppear`, move that logic to the `ObservableObject`. You need to find a way to get notified when SwiftUI subscribes to the `objectWillChange` publisher.
- Display proper errors when `Remote` contains an error instead of the loaded data.

Environment

The environment is an important piece of the puzzle for understanding how SwiftUI functions. In short, it is the mechanism SwiftUI uses to propagate values down the view tree, i.e. from a parent view to its contained subview tree. SwiftUI makes extensive use of the environment, but we can also leverage it for our own purposes.

In the first part of this chapter, we'll explore how the environment works and how SwiftUI uses it. In the second part, we'll read from the environment to customize view drawing. We'll also use the environment to store custom values in a way similar to how SwiftUI uses the environment for built-in views. Lastly, we'll look at environment objects, which allow dependency injection through a special mechanism built on top of the environment.

How the Environment Works

In SwiftUI, there are a lot of methods available on views that don't seem to fit the view at hand: methods like `font`, `foregroundColor`, and `lineSpacing` are available on all view types. For example, we can set the font on a `VStack` or the line spacing on a `Color`. How does this make sense?

To explore what's going on here, we'll again look at the type of the view we're building up. Let's start with a simple `VStack` containing a text label:

```
var body: some View {
    VStack {
        Text("Hello World!")
    }.debug()
}

// VStack<Text>
```

We're using the `debug` helper function from the first chapter to print out the concrete type of the view. In this case, it's `VStack<Text>`. Now let's call the `font` method on the stack and see how the type changes:

```
var body: some View {
    VStack {
        Text("Hello World!")
    }
    .font(Font.headline)
    .debug()
}
```

```
/*
ModifiedContent<
VStack<Text>,
_EnvironmentKeyWritingModifier<Optional<Font>>
>
*/
```

The type tells us that the .font call has wrapped the vertical stack in another view called ModifiedContent, which has two generic parameters: the first one is the type of the content itself, and the second one is the modifier that's being applied to this content. In this case, it's the private _EnvironmentKeyWritingModifier, which — as the name suggests — writes a value to the environment. For a .font call, an optional Font value is written to the environment. Since the environment is passed down the view tree, the text label within the stack can read the font value from the environment.

Even though setting a font on a vertical stack doesn't make immediate sense, the font setting is not lost; it's preserved via the environment for any child view in the tree that might actually be interested in it. To verify that the font value is available to the text label, we can print out the part of the environment we're interested in. (Usually we would use the @Environment property wrapper to read a specific value from the environment, but for debugging purposes, we can use transformEnvironment):

```
var body: some View {
    VStack {
        Text("Hello, world!")
            .transformEnvironment(\.font) { dump($0) }
    }
    .font(Font.headline)
}

// ...
// - style: SwiftUI.Font.TextStyle.headline
```

We can look up all the publicly available environment properties on the EnvironmentValues type. However, SwiftUI stores more than just these public properties in the environment. To see everything that's in there, we can use the transformEnvironment modifier and pass \.self as the key path.

Since font is a public property on EnvironmentValues, we can also use it to set the font in the environment instead of calling the font method:

```
var body: some View {
    VStack {
        Text("Hello World!")
    }
    .environment(\.font, Font.headline)
    .debug()
}

/*
```

```
ModifiedContent<
VStack<Text>,
_EnvironmentKeyWritingModifier<Optional<Font>>
>
*/
```

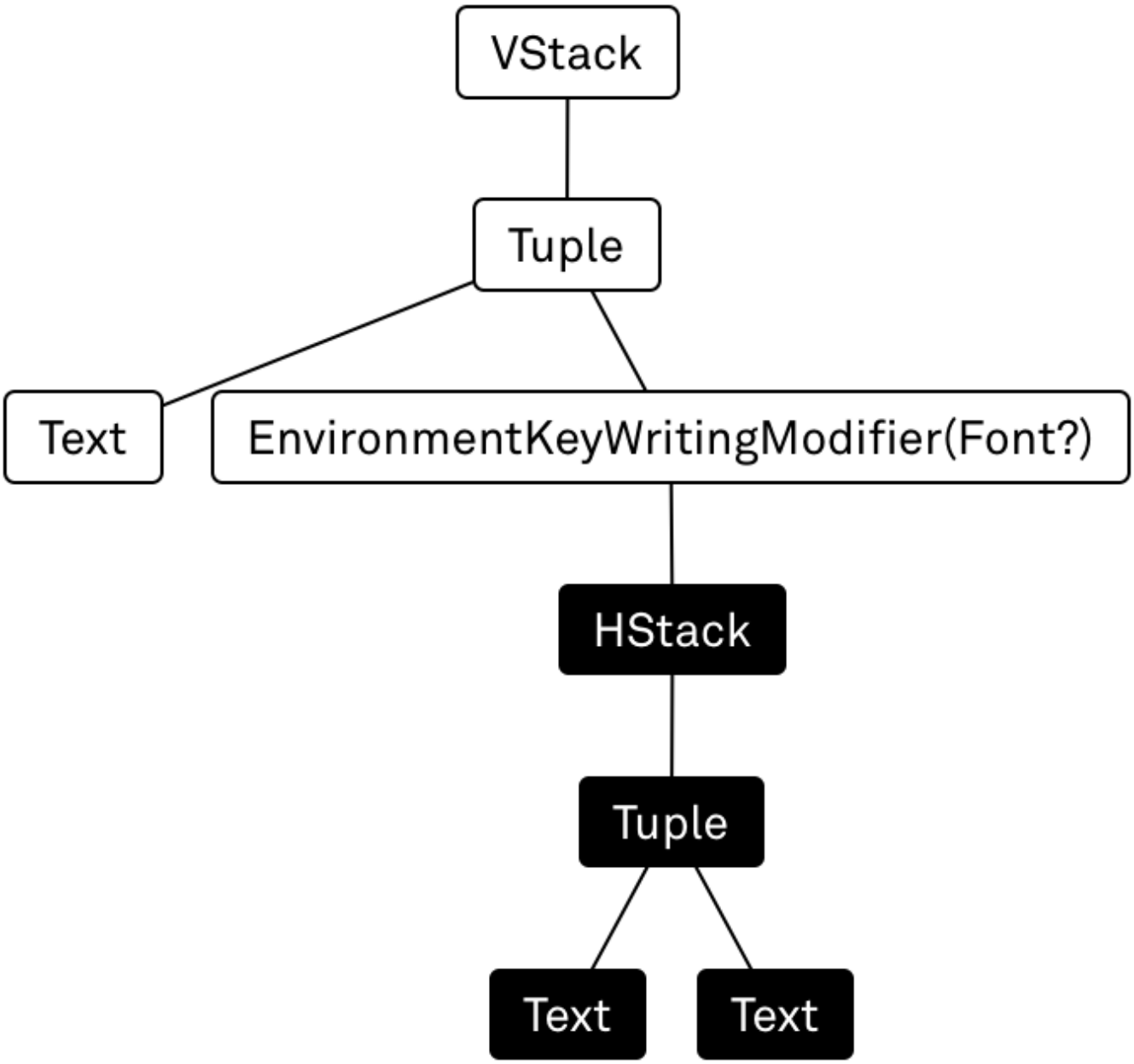
Calling .environment(\.font, ...) results in the exact same type as calling .font(...) does. There's no good reason to do this, but it demonstrates that the font method we call on e.g. a VStack is just a simple wrapper around that method.

> *When we call .font directly on Text, the return type is Text. In other words, the font is stored in the Text instead of being written to the environment. Meanwhile, the .font method called on VStack is a completely different (overloaded) method. This pattern is used throughout SwiftUI.*

The environment becomes especially useful when we want to modify multiple views at once. For example, in the following case, the first label is rendered in the default font, and the two nested labels are rendered using a .largeTitle font:

```
VStack {
    Text("Text 1")
    HStack {
        Text("Text 2")
        Text("Text 3")
    }.font(.largeTitle)
}
```

The environment modifiers always only modify the environment of their direct subview trees and never the environment of sibling or parent views. We can also see this when we visualize the structure of the code above — only the black nodes receive the changed environment containing the .largeTitle font:



Using the Environment

In this section, we'll first use an existing environment value provided by SwiftUI to adapt the appearance of a custom view, and then we'll introduce custom environment values that allow configuration of the custom view in the same way we can configure SwiftUI's built-in views.

Using Existing Environment Values

For this example, we'll come back to the knob control from the previous chapter:

```
struct Knob: View {
    @Binding var value: Double // should be between 0 and 1
```

```
    var body: some View {
        KnobShape()
            .fill(Color.gray)
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture { ... }
    }
}
```

For the sake of brevity, the code for the actual shape of the knob is not included here, but it's available in the sample code.

To make this custom control work nicely with both light appearance and dark appearance, we'll use SwiftUI's colorScheme environment value to read the current color scheme setting from the environment and set the knob's color accordingly:

```
struct Knob: View {
    @Binding var value: Double // should be between 0 and 1
    @Environment(\.colorScheme) var colorScheme

    var body: some View {
        KnobShape()
            .fill(colorScheme == .dark ? Color.white : Color.black)
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture { ... }
    }
}
```

The @Environment property wrapper takes the key path of the value we want to read from the environment, which we can then use like a normal property. When filling the knob shape, we can read this colorScheme property and choose different colors based on its value. The same thing can be achieved by using a color from an asset catalog, but by using the environment value, we can do much more than just adjust the colors; we can also adjust the way we draw the knob, e.g. we might want to adapt stroke widths to go well with the colors.

When the color scheme value in the environment changes, the knob view will automatically be rerendered. In this regard, @Environment properties act just like @State properties: when the value changes, a view update is triggered. Note however that the environment is not just a global dictionary of values — the environment of one view subtree can be different from that of another subtree, i.e. values in the environment are strictly propagated down in the view tree (from parents to their children).

Custom Environment Values

The KnobShape we're using to draw the actual knob has a parameter that allows the size of the pointer on the knob to be configured. We want to expose this customization option to the outside, and we want to do it in a way that allows us to specify the pointer size anywhere in the view tree upstream from the knob view (similar to how we set the font on the vertical stack above, which then got picked up by the text label within the stack).

Of course, it doesn't make sense to expose all view customizations in this way. Oftentimes, it's fine to expose them directly via the view's initializer. However, if we were writing e.g. an audio application, it might make sense to be able to control knob styles on a higher level.

Let's get started with the simplest option, which is that of exposing the configuration via the initializer:

```
struct Knob: View {
    @Binding var value: Double // should be between 0 and 1
    var pointerSize: CGFloat = 0.1
    @Environment(\.colorScheme) var colorScheme

    var body: some View {
        KnobShape(pointerSize: pointerSize)
            .fill(colorScheme == .dark ? Color.white : Color.black)
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture { ... }
    }
}
```

By specifying a default value for the pointer size, we don't have to provide a value during construction. Ideally, we would also be able to specify the pointer size later on in the view hierarchy, for example, like this:

```
Knob(value: $value)
    .knobPointerSize(0.2)
```

However, the following should also work:

```
Knob(value: $value)
    .frame(width: 100, height: 100)
    .knobPointerSize(0.2)
```

By calling .frame on the knob, we're no longer dealing with a view of type Knob. The frame modifier has wrapped the knob in a ModifiedContent struct, which means we have to make knobPointerSize available on any type that conforms to View (not just on Knob):

```
extension View {
    func knobPointerSize(_ size: CGFloat) -> some View {
        // ...
    }
}
```

The environment allows us to pass the pointer size value from where knobPointerSize is called, down to the actual knob view. The implementation of knobPointerSize should just call .environment on self, specifying some key path and the size value. To make this work, we first have to define a new type conforming to the EnvironmentKey protocol:

```
fileprivate struct PointerSizeKey: EnvironmentKey {
    static let defaultValue: CGFloat = 0.1
}
```

The EnvironmentKey protocol's only requirement is a static defaultValue property. Whenever we try to read the pointer size value from an environment that does not contain a value for pointer size, this default value will be returned. (Note that we've implicitly specified CGFloat as the protocol's associated Value type — the compiler infers this from the declaration of the defaultValue property.)

Since the .environment API takes a key path from EnvironmentValues to the type of the value, we have to add a property to EnvironmentValues so that we can use it as a key path:

```
extension EnvironmentValues {
    var knobPointerSize: CGFloat {
        get { self[PointerSizeKey.self] }
        set { self[PointerSizeKey.self] = newValue }
    }
}
```

The getter uses the existing subscript on EnvironmentValues to retrieve the value using the custom PointerSizeKey, and the setter sets the new value using the same subscript. With this property in place, we can come back to the implementation of knobPointerSize. Note that environment is a method of self and returns self with the modified environment:

```
extension View {
    func knobPointerSize(_ size: CGFloat) -> some View {
        environment(\.knobPointerSize, size)
    }
}
```

The only thing left to do is to actually use the value from the environment in the knob view itself. We change our pointerSize property to be optional and use the @Environment property wrapper to read the value from the environment (which will be the default value if we didn't call knobPointerSize somewhere up the view tree). We can then use either the pointerSize provided in the initializer, or if that is nil, the pointer size from the environment:

```
struct Knob: View {
    @Binding var value: Double // should be between 0 and 1
    var pointerSize: CGFloat? = nil
    @Environment(\.knobPointerSize) var envPointerSize
    @Environment(\.colorScheme) var colorScheme

    var body: some View {
        KnobShape(pointerSize: pointerSize ?? envPointerSize)
            .fill(colorScheme == .dark ? Color.white : Color.black)
            .rotationEffect(Angle(degrees: value * 330))
            .onTapGesture { ... }
    }
}
```

Now we can customize the appearance of the knob like this:

```
Knob(value: $value)
```

```
    .frame(width: 100, height: 100)
    .knobPointerSize(0.2)
```

Or even like this for multiple knobs at once:

```
HStack {
    VStack {
        Text("Volume")
        Knob(value: $volume)
            .frame(width: 100, height: 100)
    }
    VStack {
        Text("Balance")
        Knob(value: $balance)
            .frame(width: 100, height: 100)
    }
}
.knobPointerSize(0.2)
```

The environment is a powerful tool for passing data down the view tree. However, it also decouples reading a configuration value from setting that value. As a result, it's very easy to forget to set an environment value or to set it on the wrong view subtree. We suggest always starting by exposing customization options as simple view parameters, and then when noticing that a more decoupled API would be useful, it's easy to change the implementation to use an environment value as well.

Dependency Injection

We can see the environment as a form of dependency injection; setting an environment value is the same as injecting a dependency, and reading the environment value is the same as receiving a dependency.

However, the environment is usually used with value types: a view that depends on an environment value via an `@Environment` property is only invalidated when a new environment value is set for the key in question. If we store an object in the environment and observe it with `@Environment`, the view will not be invalidated if a property of the object changes — this only occurs if an entirely different object is set. However, that's usually not the behavior we want when working with object dependencies.

SwiftUI's environment system provides specific support for injecting objects using the `environmentObject(_:)` modifier. This method takes an `ObservableObject` and passes it down the view tree. It works without specifying an environment key because the type of the object is automatically used as the key. To observe an environment object in a view, we use the `@EnvironmentObject` property wrapper. This will cause the view to be invalidated when the `objectWillChange` publisher of the observed object is triggered.

For example, we can use this mechanism to pass a database connection throughout our code. As a first step, we create a view that uses a DatabaseConnection from the environment:

```
struct MyView: View {
    @EnvironmentObject var connection: DatabaseConnection
    var body: some View {
        VStack {
            if connection.isConnected {
                Text("Connected")
            }
        }
    }
}
```

To inject the database connection, we have to provide an instance of DatabaseConnection to the environment somewhere in an ancestor of MyView. If we forget to do this, our code will crash. For example, we could provide it like this:

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            MyView()
        }.environmentObject(DatabaseConnection())
    }
}
```

If we pull the call to environmentObject out of the content view, we can now easily swap out the implementation of DatabaseConnection for a subclass that's used for testing.

While the @EnvironmentObject pattern is useful, we suggest using @Environment with a key if it's possible to get away with passing a value type, because it's the safer mechanism: EnvironmentKey requires us to provide a default value. With @EnvironmentObject, it is almost too easy to end up with a hard crash if we forget to inject the object.

Preferences

While the environment allows us to implicitly pass values from a view to its children, the preference system allows us to implicitly pass values from child views to a superview.

As an example of how SwiftUI uses preferences, let's look at SwiftUI's NavigationView, which is similar to UIKit's UINavigationViewController, in that individual views can define a navigationBarTitle, navigationBarItems, and so on through modifiers. For example, this is how to create a navigation view with a single root view and a navigation title:

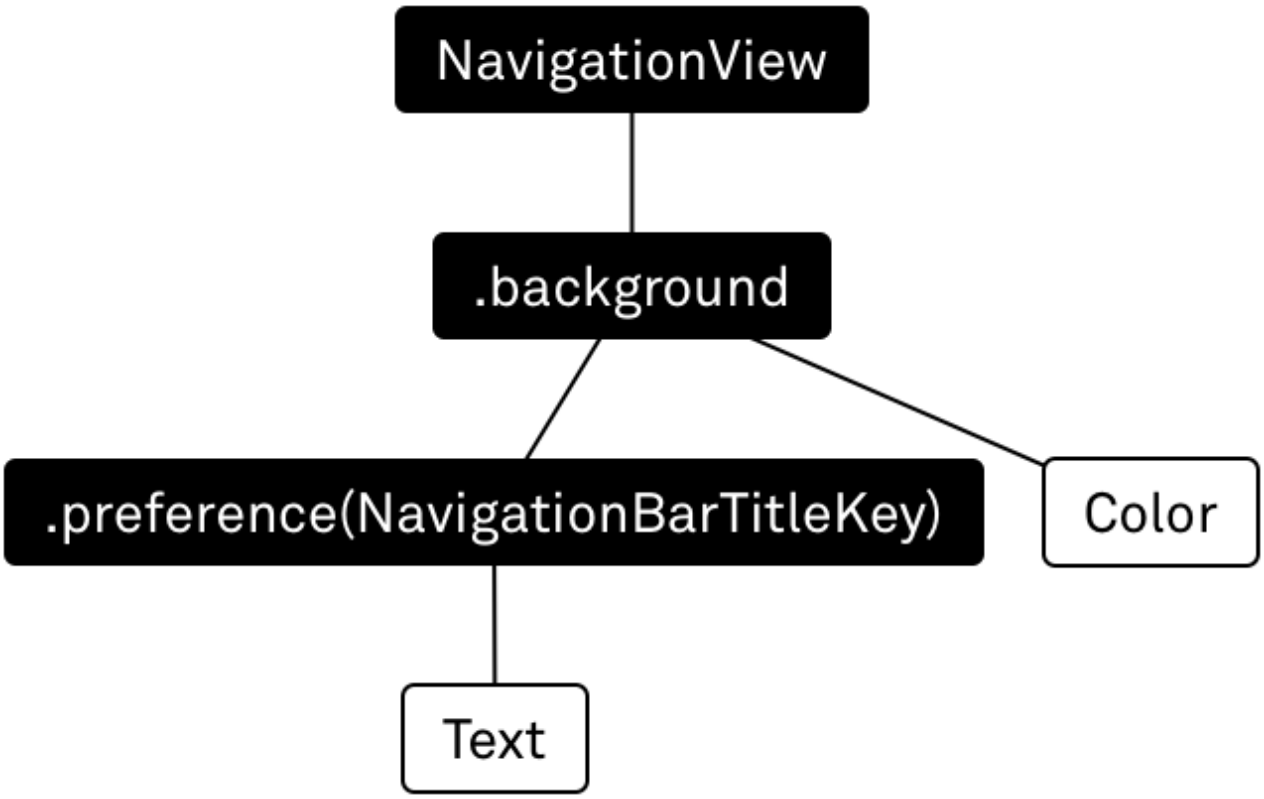
```
NavigationView {
    Text("Hello")
        .navigationBarTitle("Root View")
```

```
}
```

Earlier in this chapter, we looked at modifiers such as .font and .foregroundColor, which modify the environment for their respective view subtrees. However, a .navigationBarTitle is the opposite: the Text isn't interested in the title, but the parent view is — and it's not always the direct parent either. For example, we could write our code like this:

```
NavigationView {
    Text("Hello")
        .navigationBarTitle("Root View")
        .background(Color.gray)
}
```

When we look at a graph of the (simplified) type of this view tree, we can see that the preference is a child of the background modifier, which is in turn a child of the navigation view. Every ancestor (colored black in the diagram) of the .preference can read out the value:



The navigationBarTitle defines the preference, and the value gets propagated up the tree, through the .background modifier, all the way to the navigation view, which can read out the value.

Just like with the environment, a preference consists of a key (represented as a type), an associated type for the value (in the case of the navigation title, this is a private type containing a Text view), and a default value (in this case, probably nil). Unlike an EnvironmentKey, a PreferenceKey also needs a way to combine values (in case multiple view subtrees define the same preference).

As an example, we'll recreate a very small part of the NavigationView. As a first step, we'll create a new PreferenceKey. For the associated type, we choose String?. The reduce function is necessary for the case where multiple subtrees define a navigation title key; in our implementation, we simply choose the first non-nil value we see:

```
struct MyNavigationTitleKey: PreferenceKey {
    static var defaultValue: String? = nil
    static func reduce(value: inout String?, nextValue: () -> String?) {
        value = value ?? nextValue()
    }
}
```

As the second step, we need a way to define navigation titles on any view. We can do this with a method on View that sets the preference:

```
extension View {
    func myNavigationTitle(_ title: String) -> some View {
        preference(key: MyNavigationTitleKey.self, value: title)
    }
}
```

Finally, we need to read out the preference in our MyNavigationView. To use the value, we need to store it in a @State variable:

```
struct MyNavigationView<Content>: View where Content: View {
    let content: Content
    @State private var title: String? = nil
    var body: some View {
        VStack {
            Text(title ?? "")
                .font(Font.largeTitle)
            content.onPreferenceChange(MyNavigationTitleKey.self) { title in
                self.title = title
            }
        }
    }
}
```

The first time the view gets rendered, title is nil. As the child view (content) gets rendered, it will propagate its value for MyNavigationTitleKey up the tree, and the onPreferenceChange closure will be called. This changes the title property, and, since title is a @State property, the body of MyNavigationView is executed again.

We can use this new MyNavigationView in combination with the myNavigationTitle modifier just like we use a normal navigation view (except for the view builder syntax):

```
MyNavigationView(content:
    Text("Hello")
        .myNavigationTitle("Root View")
        .background(Color.gray)
```

)

We mentioned above that multiple view subtrees can set a value for the same preference key. Where we just picked the first non-nil value in the reduce method of the preference key for the navigation title, we could also collect the preference values from all subtrees. For example, this would be useful for implementing a tab view instead of a navigation view: the latter only displays the title of the topmost view on the navigation stack, whereas a tab view displays the titles of all its children. The preference key for the tab item titles could look something like this:

```
struct TabItemKey: PreferenceKey {
    static let defaultValue: [String] = []
    static func reduce(value: inout [String], nextValue: () -> [String]) {
        value.append(contentsOf: nextValue())
    }
}
```

This preference key uses [String] as its associated type so that we can collect all tab item titles in the reduce method. The tab view itself could read and use this preference similar to how the navigation view does it in the example above:

```
struct MyTabView: View {
    @State var titles: [String] = []
    var body: some View {
        VStack {
            // ...
            HStack {
                ForEach(Array(titles.enumerated()), id: \.offset) { item in
                    Text(item.element)
                }
            }
        }.onPreferenceChange(TabItemKey.self) { self.titles = $0 }
    }
}
```

In our experience with SwiftUI thus far, we've used preferences almost exclusively to work with the layout system (we'll show this in the chapter on custom layouts). However, there are other cases (such as in a navigation view) where this technique is also useful.

Takeaways

- The environment is used throughout all of SwiftUI, and it enables us to write concise code. For example, if we set the font on a VStack, all children inherit that font.
- @EnvironmentObject is a built-in way to perform dependency injection (i.e. to use the environment with objects and not just values). But we need to be careful to not forget setting the dependencies.
- The counterpart to the environment is the preference system: it's used to implicitly communicate values up the view tree.

Exercises

Configurable Knob Color

In this exercise, you'll practice using the environment to make a custom view configurable just like SwiftUI's built-in views are. Take the knob from the example above (you can download the full code on GitHub), and add a way to configure its color.

Since the knob already contains logic for using different colors based on the environment's color scheme (light appearance or dark appearance), the environment-based color configuration will be an optional override of this default behavior.

Step 1: Create an Environment Key

Create a new environment key named `ColorKey` with an optional color value, and add a property for this key in an extension on `EnvironmentValues`. Then create a convenience method `knobColor` on `View` to set the knob color in the environment.

Step 2: Create an @Environment Property

Create an `@Environment` property for the knob color in the `Knob` view and, if it's not `nil`, use it instead of the color scheme-based fill color.

Step 3: Control the Color with a Slider

Create a slider to control the color (for example, you can use the slider to control the hue value) and a toggle to control whether this custom color or the default color of the knob should be used.

Layout

The view layout process has the task of assigning each view in the view tree a position and a size. In SwiftUI, the algorithm for this is simple in principle: for each view in the hierarchy, SwiftUI proposes a size (the available space). The view lays itself out within that available space and reports back with its actual size. The system (by default) then centers the view in the available space. While there's no public API available for this, imagine that each View has the following method implemented:

```
struct ProposedSize {
    var width, height: CGFloat?
}
extension View {
    func layout(in: ProposedSize) -> CGSize {
        // ...
        for child in children {
            child.layout(in: ...)
        }
        // ...
    }
}
```

To explain the layout behavior of individual views, we'll pretend that the above method exists.

What makes layouts in SwiftUI complex is that each view (or view modifier) behaves differently when it comes to determining the actual size from the proposed size. For example, a shape always fits itself into the proposed size; a horizontal stack takes on the size that's needed for its children (up to the proposed size); and a text view takes on the size needed to render its text, unless its size exceeds the proposed size, in which case the text gets clipped.

Typically, both dimensions of the proposed size will have a non-nil value. A nil value for a dimension means that the view can become its ideal size in that dimension. We'll look at ideal sizes later in this chapter.

While the layout process assigns a size and a position to each view (in other words: a rectangle), the view doesn't always draw itself within those bounds. This is especially useful during animations: we might want to keep a view's layout position (so that other views stay in place as well) but draw the view with an offset or rotation.

In the rest of this chapter, we'll explore SwiftUI's layout process. We'll start with the layout behavior of some elementary views, like Text and Image. Then we'll look at several layout modifiers, like .frame and .offset. Finally, we'll talk about stack views.

Elementary Views

Let's take a more detailed look at the layout behavior of some commonly used views: shapes, images, and text views.

Since the exact layout behavior of SwiftUI's views is not documented, we have to determine it by experimentation. A simple way of doing this is to wrap any view within a frame. The frame size is controlled by two sliders, which makes it easy to experiment:

```
struct MeasureBehavior<Content: View>: View {
 @State private var width: CGFloat = 100
 @State private var height: CGFloat = 100
 var content: Content

 var body: some View {
     VStack {
         content
             .border(Color.gray)
             .frame(width: width, height: height)
             .border(Color.black)
         Slider(value: $width, in: 0...500)
         Slider(value: $height, in: 0...200)
     }
 }
}
```

We're drawing two borders to show the actual sizes: the gray border is drawn around the view, and the black border is drawn around the frame. Sometimes the view chooses to be smaller than the proposed size, in which case the gray border will be smaller than the black border. When the view is larger than the proposed size (i.e. draws out of bounds) the gray border will be larger than the black border.

If we just want to visualize the actual size of a view, the easiest way is by putting a border around it.

Paths

A Path is a list of 2D drawing instructions (similar to a CGPath in Cocoa). When the layout method on Path is called, it always returns the proposed size as its actual size. If any of the proposed dimensions is nil, it returns a default value of 10. For example, here's a Path that draws a triangle in the top-left corner:

```
Path { p in
    p.move(to: CGPoint(x: 50, y: 0))
    p.addLines([
        CGPoint(x: 100, y: 75),
```

```
            CGPoint(x: 0, y: 75),
            CGPoint(x: 50, y: 0)
        ])
}
```

In the example above, the bounding rectangle of the path has an origin of 0, a width of 100, and a height of 75. While the path itself is drawn within the rectangle, the layout method ignores the bounding rectangle; it still returns the proposed size.

Shapes

Often, we want a Path to fit or fill the proposed size. We can achieve this using the Shape protocol. Here's the full definition of Shape:

```
protocol Shape : Animatable, View {
    func path(in rect: CGRect) -> Path
}
```

Shapes and paths have some of the most predictable layout behavior. Like Path, the layout method on Shape always returns the proposed size as its actual size. Similar to Path, Shape chooses a default value of 10 when a proposed dimension is nil. During the layout process, a Shape receives a call to path(in:), and the rect parameter contains the proposed size as its size. This makes it possible to draw a Path that's dependent on the proposed size.

Built-in shapes such as Rectangle, Circle, Ellipse, and Capsule draw themselves within the proposed size. Shapes without constrained aspect ratios, like Rectangle, draw themselves by filling up the entire available space, whereas shapes like Circle draw themselves to fit into the available space. When creating custom shapes, it's good practice to stick with the same behavior and take the available space into account. For example, here's another triangle, but it's defined as a Shape that fills the proposed size:

```
struct Triangle: Shape {
    func path(in rect: CGRect) -> Path {
        return Path { p in
            p.move(to: CGPoint(x: rect.midX, y: rect.minY))
            p.addLines([
                CGPoint(x: rect.maxX, y: rect.maxY),
                CGPoint(x: rect.minX, y: rect.maxY),
                CGPoint(x: rect.midX, y: rect.minY)
            ])
        }
    }
}
```

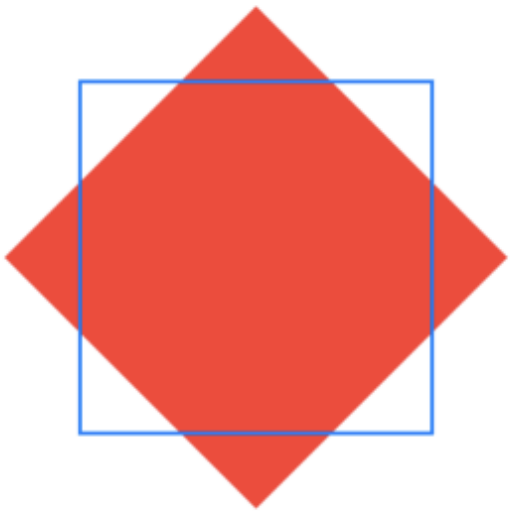
A Shape can have modifiers. For example, we could create a rotated Rectangle:

```
Rectangle()
    .rotation(.degrees(45))
```

```
    .fill(Color.red)
```

The rotation modifier returns a RotatedShape<Rectangle>. In RotatedShape's layout method, it passes the proposed size on to its child, unchanged, and it also returns that same size. In other words, the rotated rectangle above draws outside of its bounds. To visualize this, we can add a border around the rectangle:

```
Rectangle()
    .rotation(.degrees(45))
    .fill(Color.red)
    .border(Color.blue)
    .frame(width: 100, height: 100)
```



The blue border visualizes the size as far the layout system is concerned, yet the rectangle draws itself out of bounds. The offset modifier on shapes exhibits the same behavior: it doesn't change the layout; rather it draws the shape in a different position (as an OffsetShape).

Images

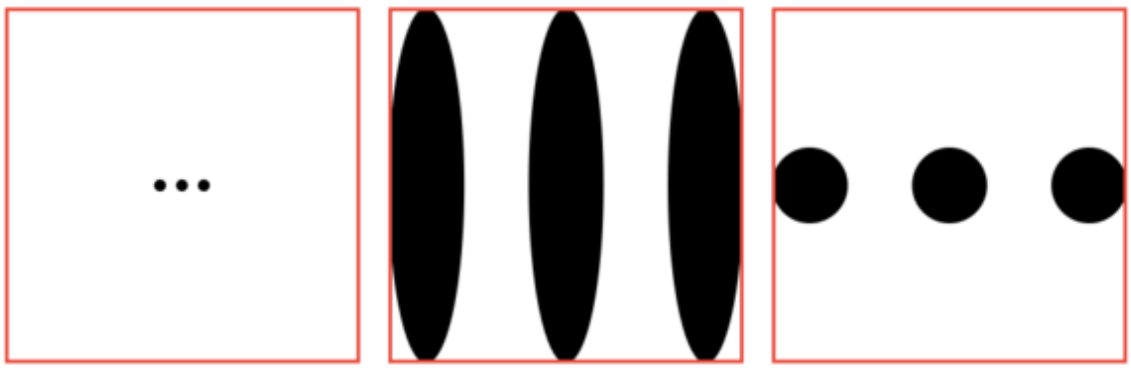
By default, Image views have a fixed size — namely the size of the image in points. This means that the image view's layout method ignores the size proposed by the layout system and always returns the image's size.

To make an image view flexible, i.e. to make it accept the proposed size and fit the image within this space, we can call .resizable on it. By default, this stretches the image to fill the proposed size (we can also configure it to tile, or to only stretch part of an image). Since most images are meant to be displayed at a fixed aspect ratio, .resizable is commonly combined with a call to .aspectRatio directly after.

The .aspectRatio modifier takes the proposed size and creates a new size that best fills the proposed size, taking the given aspect ratio into account. It then proposes this size to the resizable image (which fills up this size) and returns it to the parent. We can choose to either fit or fill the proposed size, and we can either specify a fixed aspect ratio or leave out the aspect ratio and let the child view determine it.

Here is the same image displayed three different ways. The first image is unchanged, the second image is resizable (filling up the proposed size), and the third image also has an aspect ratio. Because we only defined the content mode, the aspect ratio modifier takes on the aspect ratio of its child (the image):

```
let image = Image(systemName: "ellipsis")
HStack {
    image
    image.resizable()
    image.resizable().aspectRatio(contentMode: .fit)
}
```



(For the image above, we have added a red border and a 100-by-100 frame to each view in the HStack.)

Text

A Text view's layout method always tries to fit its contents in the proposed size, and it returns the bounding box of the rendered text as its result. If the underlying string does not contain newlines, it attempts to render the entire text on a single line (given there is enough horizontal space). If there's not enough horizontal space available, Text looks at the available vertical space. If there's enough vertical space, it will break the text into multiple lines (line wrapping) and try to fit the entire text inside the proposed size. If there's not enough vertical space either, the text will be truncated.

We can customize most of this behavior by using view modifiers:

- fixedSize proposes nil as the size, and the text view becomes its ideal size. More importantly, this prevents line wrapping. This does mean that the text may draw outside

of the size proposed to fixedSize.

- lineLimit specifies the maximum number of lines. If the text contains newlines and the line limit is less than the number of lines in the text, the text will show truncation marks at the end of the last line.
- minimumScaleFactor allows Text to be rendered at a smaller font size (if the text doesn't fit).
- truncationMode determines the truncation behavior, i.e. whether the beginning, middle, or end is truncated.

Layout Modifiers

The layout of views is influenced by modifiers: these are wrapper views that change layout behavior, although they're typically defined as extension methods on the View protocol.

Frame

There are two versions of the .frame modifier: one to specify fixed-size frames, and the other to specify flexible frames.

For the fixed-size .frame call, we specify a width or a height (or both) and an alignment. By default, the width and height parameters are nil, and the alignment is .center.

When we specify a parameter such as width, the frame's layout method will propose this fixed width to its child view (likewise for the height). The layout method also returns the fixed size as its size. If only one dimension is fixed, the returned size uses the return value from the child's layout method for the other dimension. For example, consider applying a frame with only a fixed width to a text view:

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

This means that the text's layout method will always receive a proposed size that's 100 points wide, regardless of the width proposed to the frame modifier. Because only the width is specified, the .frame modifier's layout method will return a size that's 100 points wide, and it uses the height of the Text. In code, it would look something like this:

```
func layout(in proposedSize: ProposedSize) -> CGSize {
    let proposedChildSize = ProposedSize(width: 100,
        height: proposedSize.height)
    let childSize = child.layout(proposedSize: proposedChildSize)
    return CGSize(width: 100, height: childSize.height)
}
```

The frame's alignment is used to position the child in case the child's size is different than the frame's size. For example, if we shorten the text in the snippet above so that it is narrower than 100 points, there is some horizontal space left. By default, the alignment is .center, and the text view is centered inside the 100 points. We can also align views at the leading edge or trailing edge, or even using a custom alignment guide. Note that the alignment isn't just useful when the child is smaller than the fixed dimension: it's also used when the child is larger than the fixed dimension.

Flexible frames work in a similar way. Instead of a fixed size, we give a minimum, ideal, and maximum width and height (similar to fixed-size frames, we can leave out any of the parameters and by default they will be nil). The minimum and maximum dimensions are used to *clamp* both the proposed and the returned size. For example, when we configure a maximum width, the .frame modifier's layout implementation looks at the proposed width, and if the proposed width is larger than the maximum width, it only proposes the maximum width to its child. Likewise, if the child returns a size that's wider than the maximum width, that result is clamped.

The ideal dimensions are used when a proposed dimension is nil. For example, when the frame gets proposed a nil width but an ideal width is specified, the frame modifier proposes that ideal width to its child. Likewise, when the proposed width is nil, the frame modifier also returns the ideal width as its width, and it disregards the child's width. Of course, the same logic applies to the height dimension. To propose a nil dimension, we use the .fixedSize() modifier.

As an example, we can wrap the KnobShape from the view updates chapter in a view that behaves almost like an Image: by default, the knob has a fixed size, but you can call .resizable() on it to create a resizable knob. Note that resizable() is not a built-in method, but rather it's defined by us:

```
struct Knob: View {
    var body: some View {
        KnobShape().frame(width: 32, height: 32)
    }

    func resizable() -> some View {
        KnobShape()
            .aspectRatio(1, contentMode: .fit)
            .frame(idealWidth: 32, idealHeight: 32)
    }
}
```

Knob().resizable() creates a knob view that fits itself into the proposed size (since it has a flexible frame). The call to aspectRatio ensures that the view's width and height are identical, in order to prevent distortion. When we call .fixedSize() on the resizable knob, its proposed size will be nil in both dimensions, and so the knob takes on its ideal size, which we have specified as 32 points.

> *Note that calling fixedSize() on stacks often causes unexpected behavior, because stack views lay out their content in two passes (we'll talk about how stack views work at the end of this*

chapter).

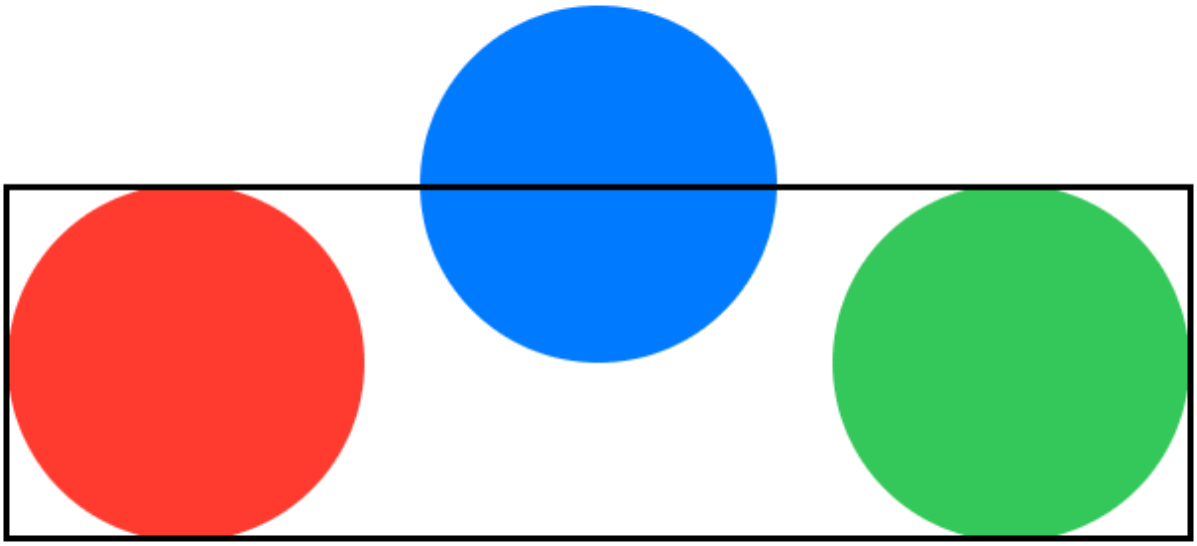
Offset

The offset modifier simply forwards the proposed size to its child, and it reports back the child's size as its own size. In other words, it does not affect the layout. However, it does draw the child at a different position (given by the horizontal and vertical offset).

We found `offset` to be especially useful during animations and interactions. For example, when we display a list of draggable items, we can use `offset` to move the dragged item to the drag position while still maintaining its space in the list.

In the following example, we give the middle circle a y offset of -30. This draws the view at a different position, yet it does not affect the layout:

```
HStack {
    Circle().fill(Color.red)
    Circle().fill(Color.blue).offset(y: -30)
    Circle().fill(Color.green)
}
.frame(width: 200, height: 60)
.border(Color.black)
```



Padding

The padding modifier is one of the simplest around. The full version of this modifier takes `EdgeInsets` as its parameter. Or, in other words: we can specify the padding per edge (top,

bottom, leading, and trailing). There are also convenience variants. For example, we can write .padding() without any arguments in order to add the default system padding for each edge.

In its layout method, the padding modifier subtracts the padding from the proposed size and proposes this new size to its child. It then returns the size of the child with added padding. It positions the child accordingly (in effect, it offsets the child using the leading and top edges of the inserts).

Note that we can also use negative padding, which will propose to the child a size that's larger than the size proposed to the padding.

Overlay and Background

The overlay and background modifiers are important parts of the layout system as well. When we write content.overlay(other), the system creates an overlay modifier with two children: content and other.

When the overlay modifier is laid out, the proposed size is passed to content. Then the reported size from content is passed to other as the proposed size. The overlay modifier reports back the size of content as its own size: in other words, the reported size of other is ignored. The implementation of layout would look something like this:

```
extension Overlay {
    func layout(in proposedSize: ProposedSize) -> CGSize {
      let backgroundSize = background.layout(proposedSize: proposedSize)
      _ = foreground.layout(proposedSize: ProposedSize(
          width: backgroundSize.width, height: backgroundSize.height))
      // ...
      return backgroundSize
    }
}
```

For content.background(other), the process is much the same, except that other is now drawn behind content. It's important to note that content.overlay(other) is *not* the same as other.background(content): in the former case, the size of content is returned as the layout size, whereas in the latter case, the size of other is returned.

Overlays and backgrounds are often useful when combined with shapes. For example, here's an overlay that draws a capsule shape around a Text without influencing the layout:

```
Text("Hello").background(
    Capsule()
        .stroke()
        .padding(-5)
)
```

Let's consider drawing a circular button, similar to the buttons in iOS's built-in stopwatch. As a first attempt, we might draw the circle as the .background of the text:

```
Text("Hello, World!")
    .foregroundColor(.white)
    .background(Circle().fill(Color.blue))
```

When we run the code above, we'll see a tiny circle that's as high as the text. While it's not the result we hoped for, the behavior makes sense: first the text is sized, and then the circle is drawn inside the size of the text. By default, the circle fits itself into the available space (taking the minimum of the width and height as its diameter).

Instead, we can draw the text as an overlay on the circle, and we can give the circle an explicit width and height:

```
Circle()
    .fill(Color.blue)
    .overlay(Text("Start").foregroundColor(.white))
    .frame(width: 75, height: 75)
```

Ideally, the button would also become larger if the text doesn't fit, but unfortunately, we'll need some advanced techniques for that (which we'll show in the next chapter).

We can also combine multiple overlays in useful ways. For example, we could draw the same button with a slightly smaller circle inside that's inset by a few points:

```
Circle()
    .fill(Color.blue)
    .overlay(Circle().strokeBorder(Color.white).padding(3))
    .overlay(Text("Start").foregroundColor(.white))
    .frame(width: 75, height: 75)
```

Clipping and Masking

As a final modifier, clipping and masking can be helpful in combination with the modifiers above. Neither one affects the layout, but they do influence what's drawn on the screen. When we write .clipped(), the view is clipped to its bounding rectangle. In other words, anything the view draws outside of that rectangle won't be visible. As an example, we can use the rotated rectangle from before:

```
Rectangle()
    .rotation(.degrees(45))
    .fill(Color.red)
```

By default, the rectangle will draw partly out of bounds due to the rotation. However, if we add the .clipped modifier, only the parts of the rotated rectangle within the parent's bounding box will be visible:

```
Rectangle()
    .rotation(.degrees(45))
    .fill(Color.red)
    .clipped()
    .frame(width: 100, height: 100)
```

There is also clipShape, which works like clipped but takes a Shape instead of clipping to the bounding rect. Fun fact: rounded corners (using .cornerRadius) are implemented by calling clipShape with a RoundedRectangle.

Finally, we can provide a mask using .mask. What sets mask apart from clipped is that mask takes any view and uses it to mask the underlying view.

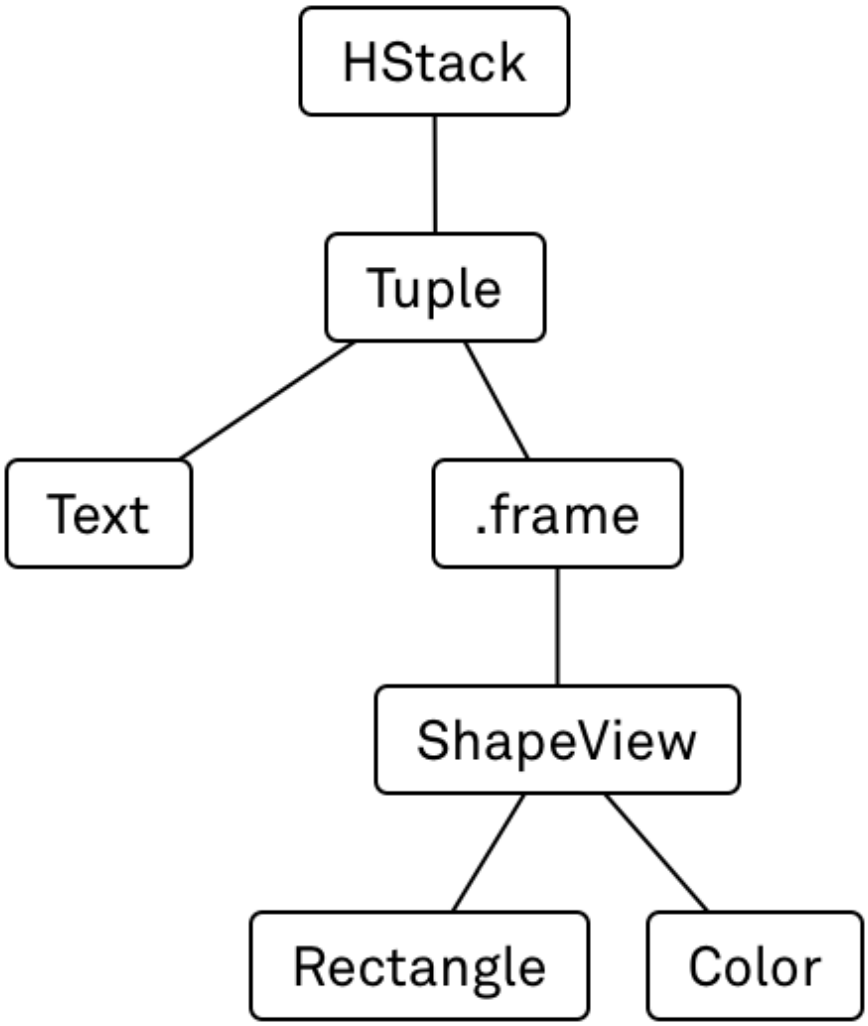
Stack Views

In SwiftUI, stack views are an essential mechanism for building up complex layouts from individual views. Stack views come in three different flavors: they can lay out views either horizontally (HStack), vertically (VStack), or on top of each other (ZStack). In this section, we'll focus on HStack, but the same rules apply to the other stack types as well.

Consider the following example for a horizontal stack:

```
HStack {
    Text("Hello, World")
    Rectangle().fill(Color.red).frame(minWidth: 200)
}
```

It's helpful to visualize the view tree structure as well:



It's interesting to wrap this view in a MeasureBehavior view (defined at the beginning of the chapter), slowly decrease the width, and observe what happens: as long as the width is large enough, the text will fit and the rectangle will fill up the remaining space.

In our setup, the text's ideal size has a width of 93 points. When the HStack's layout method receives a proposed size that's less than 301 (93 + 8 + 200) points in width, the layout no longer fits. What happens then is interesting: at first, the text is *not* broken into multiple lines, and the red rectangle doesn't shrink either. Instead, the HStack draws itself wider than the proposed size (if we use MeasureBehavior, we'll see that the gray border is wider than the black border). To understand why this happens, let's look at the algorithm in detail.

The HStack's layout method works in two passes. In the first pass, the HStack figures out which children have a fixed width, and during the second pass, the flexible space is divided among children with a flexible width.

During the first pass, HStack's layout takes the proposed horizontal space, subtracts the spacing, and divides up the remaining space into equal parts. It then proposes the parts to the children. For example, we have two views, and the default spacing between children is 8 points, so the width that's proposed to each child is (proposedSize.width - 8) / 2.

The first child, a Text, will try to lay itself out without line wrapping. If the proposed size is large enough, it reports back the size of the bounding box (the width and height of the laid-out glyphs) as its size. However, if there's not enough space to render the text in one line, Text can compress itself horizontally: depending on the configuration, it can choose to either lay itself out over multiple lines or truncate itself. The Text will never grow bigger than its bounding box.

The second child in the HStack is a frame layout containing a rectangle. A rectangle will always fit itself exactly into the proposed size. However, the frame layout is configured to have a minimum width of 200. When the HStack's layout method proposes a size that's less wide than 200, it ignores the proposed width and proposes 200 to its child (the Rectangle). The return value of the frame's layout method also depends on the proposed width: if the proposed width is less than 200, the frame's layout returns 200. Otherwise, it returns the proposed width.

After the first pass, the HStack knows which elements have a flexible width and which ones have a fixed width. It also knows the actual widths of the fixed elements. An element can be either fixed size, flexible with a maximum width, or flexible with a minimum width. In our case, the Text will report back that it has a fixed width and the rectangle reports back that it is flexible (with a minimum width of 200). Note that while the same Text can return different widths for different proposed sizes, it always reports back as being fixed width.

> *Note: We're not completely sure how "flexibility" is implemented under the hood. Does the layout system propose different sizes and look at how the view behaves? Or do views themselves report back a valid range for each dimension? Regardless, for our purposes, it doesn't matter too much, as we're mostly interested in how the views behave.*

Now the second pass of the HStack begins. It takes all the fixed-size elements and deducts their widths from the proposed width. It also deducts any spacing, which leaves us with the flexible space. It then evenly distributes that flexible space to all the flexible children.

In the example above, given that there is enough horizontal space, the HStack will first take the width of the Text (which is fixed) and the spacing in between elements and deduct that from the proposed size. Because the rectangle has a flexible maximum width, it will fill up the remainder of the proposed width.

Consider that the Text view renders at 93 points wide. If the HStack's layout method gets offered 250 points, during the first pass, it will first subtract the space (8 points) and then propose 121 points to each of the children (calculated as (250-8)/2). When Text gets proposed 121 points, it is fine with this: it only needs 93 points, and it reports that as its fixed size. The rectangle (or to be more precise, the .frame modifier) wants to be at least 200 points wide, and it is larger than the proposed size. That means the Text will draw itself at 93 points, the Rectangle will be 200 points, and with 8 points of space in between, the entire HStack will be 301 points wide. In other words, the HStack will be wider than the proposed size, and its content will be out of bounds.

Layout Priorities

There is one more way to control the layout of a stack: through layout priorities. For example, when we want to display a file path, the last path component (the file name) is typically the most important. To display a long path, we could separate it into the base path and the file name and then construct our views like this:

```
HStack(spacing: 0) {
    Text(longPath).truncationMode(.middle).lineLimit(1)
    Text("chapter1.md").layoutPriority(1)
}
```

When there is limited space available, the HStack will first offer the available space to Text("chapter1.md"). Any remaining space is then offered to Text(longPath), which automatically gets truncated in the middle when it doesn't fit inside the remaining space.

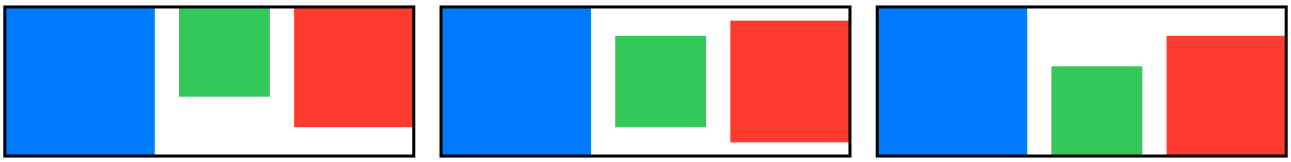
Remember the description of a stack's layout process: in the first pass, the stack's layout method determines for each child whether it has a fixed or a flexible size. In the second pass, the flexible space is divided among all the flexible children. When layout priorities are set, the second pass is a bit more complicated. The elements are grouped by layout priority: first the group with the highest layout priority gets offered the flexible space, then the group with the second-highest layout priority, and so on.

If the group with the highest layout priority contains items that are truly flexible (without any constraints), this means the other groups won't get offered any space at all. However, layout priorities are most useful when dealing with elements that have a maximum width (either through a frame, or intrinsically, as is the case with a Text).

Stack Alignment

We discussed above how stack views lay out their children along their primary axes. However, the views also need to be laid out along the other axes: vertically for an HStack, horizontally for a VStack, and horizontally and vertically for a ZStack. This is done using alignment guides: every stack view has a single alignment guide (or two in the case of a ZStack — one guide per axis). The default alignment is center (for both axes). Next, we'll show how alignment guides work in detail for an HStack, but of course, it works the same for a VStack and ZStack, just with different axes.

In an HStack, the specified alignment guide is used to ask each child for its value. For example, when we specify .center, each child view is asked for its vertical center (in terms of its own coordinate system). All the centers of the child view are then placed on a single (imaginary) line. Instead of .center, we could have chosen .top or .bottom, and the views would be top-aligned or bottom-aligned. Finally, we could have chosen to align the views along the text baseline (this is useful when working with text that has different font sizes). Here are three HStacks with top, center, and bottom alignments:



Custom Alignment

We can also create custom alignment guides or modify the behavior of alignment guides for individual views. An alignment guide is, in essence, very simple: it's a way to compute a specific point in the view. For example, in the case of vertical alignment, the alignment guide gives us a point on the vertical axis. Each alignment guide is implemented as a function from ViewDimensions to CGFloat. (The ViewDimensions struct is similar to a CGSize, but it has a few added helpers.)

For example, the .center alignment guide could be reimplemented in two parts. First, we need a custom identifier type, which has a default implementation that's used to compute the vertical alignment in a view:

```
enum MyCenterID: AlignmentID {
    static func defaultValue(in context: ViewDimensions) -> CGFloat {
        return context.height / 2
    }
}
```

Second, we can add an extension to VerticalAlignment that makes it easier to discover our custom alignment:

```
extension VerticalAlignment {
    static let myCenter: VerticalAlignment = VerticalAlignment(MyCenterID.self)
}
```

Finally, we can use the new layout:

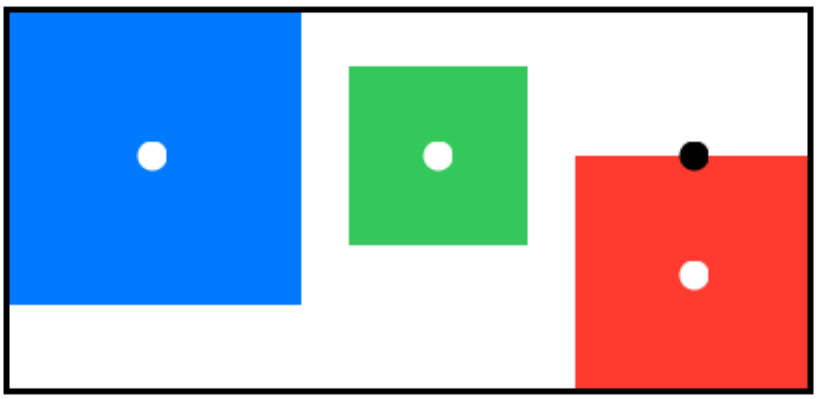
```
HStack(alignment: .myCenter) {
    Rectangle().fill(Color.blue).frame(width: 50, height: 50)
    Rectangle().fill(Color.green).frame(width: 30, height: 30)
}
```

Once the HStack knows the sizes of each child, it can use the defaultValue(in:) method to compute the center of each view. This method is called once per child, and it receives the child's size as the view dimensions. Note that the children can only specify the position in terms of their own sizes; they have no knowledge of either the siblings or the parent.

We can also override the alignment guide for a specific child. For example, if we wanted to center all our views but have one view that's slightly offset, we could do the following:

```
HStack(alignment: .myCenter) {
```

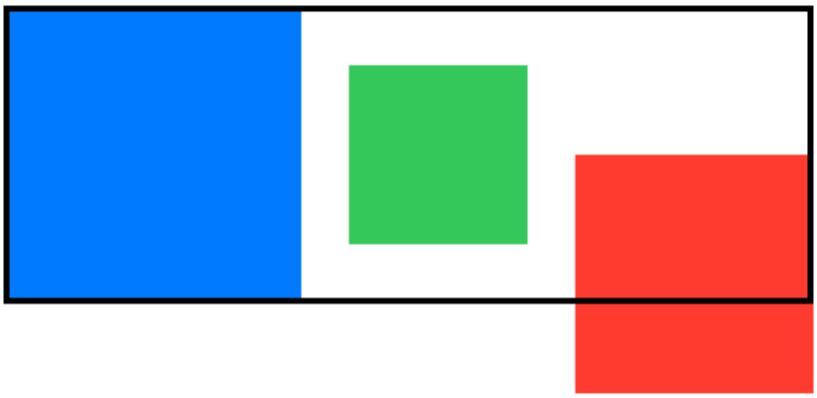
```
    Rectangle().fill(Color.blue)
        .frame(width: 50, height: 50)
    Rectangle().fill(Color.green)
        .frame(width: 30, height: 30)
    Rectangle().fill(Color.red)
        .frame(width: 40, height: 40)
        .alignmentGuide(.myCenter, computeValue: { dim in
            return dim[.myCenter] - 20
        })
}.border(Color.black)
```



We have visualized the centers of the rectangles as white dots and the custom alignment value as a black dot. For the first two rectangles, the default value is used. In the case of the first rectangle, the view dimensions have a height of 50, so the returned value is 25. For the second rectangle (with a height of 30), the returned value is 15. For the last rectangle, we provide a custom value: we take the center, which is 20, and subtract 20, returning 0. This means that the center of the first view, the center of the second view, and the top of the third view are on a single imaginary horizontal line. Note that if we override an alignment like this, the first argument to alignmentGuide (.myCenter) has to match the alignment of the HStack.

Instead of overriding the alignment guide, we could have also chosen to use .offset to move the third view. However, there's a big difference between the two: an offset does not change the layout (it only draws the view in a different place), whereas an alignment guide does change the layout of the surrounding stack view: after computing all the sizes and horizontal and vertical alignments (using the alignment guide), the stack view computes a bounding box around all its children and reports that back as its final size.

Here's what the same stack view looks like with an offset on the third view instead of a custom alignment value. Notice that the reported size is less tall than the version with a custom alignment value, and the third view draws out of bounds:



Organizing Layout Code

There are many options for organizing view layout code: we could create new View structs, write extensions on View, or create a ViewModifier. Which technique we use is mostly a matter of style. For example, we could write our blue circular button from before as an extension on View so that we can take any view and put a blue circle behind it. Note the use of self (previously, we had the Text("Hello") in that position):

```
extension View {
    func circle(foreground: Color = .white, background: Color = .blue)
        -> some View {
        Circle()
            .fill(background)
            .overlay(Circle().strokeBorder(foreground).padding(3))
            .overlay(self.foregroundColor(foreground))
            .frame(width: 75, height: 75)
    }
}
```

Now we're able to add a circular background to any view:

```
Text("Hello")
    .circle(foreground: .white, background: .gray)
```

We also could have written this as a custom View struct instead. This takes quite a bit more code. When creating container views, it's common practice to take a view builder instead of a regular view:

```
struct CircleWrapper<Content: View>: View {
    var foreground, background: Color
    var content: Content
    init(foreground: Color = .white, background: Color = .blue,
        @ViewBuilder content: () -> Content) {
        self.foreground = foreground
```

```
        self.background = background
        self.content = content()
    }

    var body: some View {
        Circle()
            .fill(background)
            .overlay(Circle().strokeBorder(foreground).padding(3))
            .overlay(content.foregroundColor(foreground))
            .frame(width: 75, height: 75)
    }
}
```

While it renders the same way as the other two approaches, the syntax for creating a CircleWrapper is a bit different:

```
CircleWrapper {
    Text("Hello")
}
```

There is a third option as well: we can create a ViewModifier. This is often useful for views that wrap other views or lay out a view differently:

```
struct CircleModifier: ViewModifier {
    var foreground = Color.white
    var background = Color.blue
    func body(content: Content) -> some View {
        Circle()
            .fill(background)
            .overlay(Circle().strokeBorder(foreground).padding(3))
            .overlay(content.foregroundColor(foreground))
            .frame(width: 75, height: 75)
    }
}
```

To create a button using our modifier, we can use the modifier method on View. In the code below, the type will be ModifiedContent<Text, CircleModifier>:

```
Text("Hello").modifier(CircleModifier())
```

In the chapter on animations, we'll look at more uses for modifiers.

We can achieve the same outcome with each of the three approaches (an extension, a custom view, and a modifier). Note that properties such as @State cannot be used with a plain extension; they need to be part of a custom view or modifier. However, if we choose to create a custom view or modifier, we can still make it available through an extension. SwiftUI itself does this; almost every method on View returns either a custom View or a view modifier. The decision of which to use generally comes down to personal preference.

Styling Buttons

While we have called our blue round button a button, it is really just text with a circle behind it. It's not interactive; nothing happens when we tap the view, and there is no visual cue when the user taps or clicks the button.

To style buttons, we use the ButtonStyle protocol. This is similar to a ViewModifier, but instead of getting the content as a parameter, we get a struct that has both the label of the button and the button's pressed state. We can then use that information to style the button:

```
struct CircleStyle: ButtonStyle {
    var foreground = Color.white
    var background = Color.blue

    func makeBody(configuration: ButtonStyleConfiguration) -> some View {
        Circle()
            .fill(background.opacity(configuration.isPressed ? 0.8 : 1))
            .overlay(Circle().strokeBorder(foreground).padding(3))
            .overlay(configuration.label.foregroundColor(foreground))
            .frame(width: 75, height: 75)
    }
}
```

To create a button with this button style, we can call buttonStyle on the button itself:

```
Button("Button", action: {})
    .buttonStyle(CircleStyle())
```

Another advantage of buttonStyle is that we can style multiple buttons at once. The buttonStyle modifier is defined on View and modifies the environment. For example, we can use it to style multiple buttons in an HStack at the same time:

```
HStack {
    Button("One", action: {})
    Button("Two", action: {})
    Button("Three", action: {})
}.buttonStyle(CircleStyle())
```

When styling buttons, a custom ButtonStyle is almost always the best approach.

Takeaways

- SwiftUI's layout process operates top-down: the parent view proposes a size to its children, which in turn lay themselves out within this proposed size and return the actual size they need.
- The relationship between the proposed size and the actual size differs between views. For example, shapes and image views always use the entire proposed size, whereas text views only take as much space as they need to fit the content.
- Layout modifiers like .frame and .padding can be used to adjust the layout. Other modifiers, like .offset and .rotation, affect the drawing of the views but not the layout.
- Stack views are a common method for building complex layouts from individual views. We can customize their behavior with layout priorities and alignment guides.
- We should factor out common view code in extensions, custom views, or modifiers.

Exercises

You can find the solutions in the solutions appendix at the end of this book, or you can download the full projects from GitHub.

Collapsible HStack

Write a collapsible wrapper view around HStack. Here's what the interface should look like:

```
struct Collapsible<Element, Content: View>: View {
    var data: [Element]
```

```
    var expanded: Bool = false
    var content: (Element) -> Content
    var body: some View { ... }
}
```

When expanded is true, the view should display just like a regular HStack, but when expanded is false, the items should be drawn on top of each other. The last item should be completely visible. Both states are in the following screenshot:

When animating expanded from true to false, the items should animate between the two states. Note that we added a black border around the Collapsible view to show that the content should not be drawing out of bounds.

Bonus Exercise

Make the Collapsible stack more configurable — especially the vertical alignment and the spacing.

Badge View

Write a view extension that displays a badge at the top-right corner of a view without influencing the layout of the views around it. This is how the API should work:

```
struct ContentView: View {
    var body: some View {
        Text("Hello")
            .padding(10)
            .background(Color.gray)
            .badge(count: 5)
    }
}
```

If `count` is `0`, the badge should be hidden. Alternatively, the `badge` could take an optional `Int` and hide itself when `count` is `nil`.

Bonus Exercises

- Ensure your badge works correctly in a right-to-left layout direction.
- Add configuration options for the position of the badge relative to the view (trailing or leading), the size and color, etc.

Custom Layout

In the previous chapter, we looked at the built-in ways of specifying layouts. In many cases, using the views and modifiers from the view layout chapter allows us to express a layout, but sometimes we need more customization. For example, what if we wanted to display completely different views depending on the available width? Or how do we implement something like a flow layout, in which items are put on a horizontal line until they don't fit anymore, after which a new line is started (similar to word wrapping when rendering text)?

In this chapter, we'll build upon the foundations of the previous chapter and show some techniques that will help you hook into (and customize) the layout process. We'll discuss geometry readers (which allow you to receive the proposed layout size), preferences (which allow you to communicate up the view tree), and anchors. Finally, we'll show how to build a custom layout by combining all these techniques.

> *Many of the techniques in this chapter feel like workarounds for the current limitations of SwiftUI. We believe that in future versions, SwiftUI will provide us with APIs that make a lot of the code in this chapter simpler or even unnecessary. That said, while some layouts are currently impossible without using these techniques, we recommend applying them cautiously.*

Geometry Readers

We can hook into the layout process by using a GeometryReader. Most importantly, we can use it to receive the proposed layout size for a view. A GeometryReader is configured with a ViewBuilder (just like any other container view), but unlike other containers, the view builder for a geometry reader receives a parameter: the GeometryProxy. The proxy has a property for the view's proposed layout size and a subscript to resolve anchors. Inside the ViewBuilder, we can use this information to lay out our subviews.

For example, if we want to draw a Rectangle with a width that's a third of the proposed size, we can use a GeometryReader:

```
GeometryReader { proxy in
    Rectangle()
        .fill(Color.red)
        .frame(width: proxy.size.width/3)
}
```

An important caveat when working with GeometryReader is that it reports its proposed size back as the actual size. Because of this sizing behavior, geometry readers are often especially useful when used as the background or overlay of another view: they become the exact size of

the view. We can use this size either to draw something in the bounds of the view or to measure the size of the view (or both).

In the previous chapter, we built a small rounded button similar to the one in iOS's built-in stopwatch app:

```
Circle()
    .fill(Color.blue)
    .overlay(Circle().strokeBorder(Color.white).padding(3))
    .overlay(Text("Start").foregroundColor(.white))
    .frame(width: 75, height: 75)
```

Unfortunately, we had to give the button a fixed size. To make it fit the text automatically, we could resort to a simple workaround: we could place a geometry reader inside the text's background and use that to draw the circle. By putting some padding around the text, the circle would be slightly wider than the text:

```
Text("Start")
  .foregroundColor(.white)
  .padding(10)
  .background(
      GeometryReader { proxy in
          Circle()
              .fill(Color.blue)
              .frame(width: proxy.size.width,
                     height: proxy.size.width)
  })
```

One problem with the approach above is that the size of the entire view will still be the size of the text plus the padding; the height of the circle will be ignored. For example, if we clip the view using .clipped(), it will simply show a blue rectangle as the background. Likewise, if we put multiple buttons like these in a vertical stack, the circles will overlap each other:

To fix this problem, we could put a frame around the entire view with the same width and height as the text's width (plus padding). Unfortunately, we won't know the width of the text until the views are laid out. In the next section, we'll show how to use preferences to deal with this.

Preferences and GeometryReaders

Using preferences, we can communicate values up the view tree, from views to their ancestors. For example, if we have a way to measure the size of a view, we can communicate this size back to a parent view using a preference. Preferences are set using keys and values, and views can set a value for a specific preference key.

To solve the problem from the previous solution, our approach is as follows: using a `GeometryReader`, we measure the size of the `Text` inside our button. We then use a preference to communicate that value up the tree and add a frame around the entire view, with the width and height equal to the width of the text. (We'll assume that the text is always wider than it is tall.)

Using the `PreferenceKey` protocol, we can define our own preference keys. The protocol has one associated type (for the values) and two further requirements: a default value (for when there are no views that define a preference), and a way to combine two values. The latter requirement is necessary because a parent view can have more than one child, and each child could define its own preference. As such, the parent needs a way to combine all the preferences of the children into a single preference.

Going back to the example of our button, we want to collect the width of the Text view. We'll use a CGFloat? as the value, where nil means that we haven't seen any Text. For now, we're not interested in combining values, so we simply take the first non-nil value we see:

```
struct WidthKey: PreferenceKey {
    static let defaultValue: CGFloat? = nil
    static func reduce(value: inout CGFloat?,
        nextValue: () -> CGFloat?) {
        value = value ?? nextValue()
    }
}
```

We are now ready to propagate our new preference. Like before, we start with the text and add a GeometryReader — which will receive the text's size as its proposed size — as the background. To set the preference, we have to call .preference(...) on some view within the geometry reader's closure. Since we don't want to draw anything, we'll use a Color.clear view:

```
Text("Hello, world")
    .background(GeometryReader { proxy in
        Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
    })
```

We have tried to use EmptyView() instead of Color.clear to propagate the preference, but surprisingly, it didn't work. We're not sure if it's just a bug or if there's a deeper reason for the different behavior of EmptyView() and Color.clear.

Any parent view of this tree can now read out the preference using .onPreferenceChange. For example, we can store the preference in a @State variable:

```
struct TextWithCircle: View {
    @State private var width: CGFloat? = nil
    var body: some View {
        Text("Hello, world")
            .background(GeometryReader { proxy in
                Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
            })
            .onPreferenceChange(WidthKey.self) {
                self.width = $0
            }
    }
}
```

To display a circle around the text, we'll add a frame around the text that uses the width property for both its width and height. Then we'll add a Circle as the background view:

```
struct TextWithCircle: View {
    @State private var width: CGFloat? = nil
    var body: some View {
        Text("Hello, world")
            .background(GeometryReader { proxy in
```

```
                Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
            })
            .onPreferenceChange(WidthKey.self) {
                self.width = $0
            }
            .frame(width: width, height: width)
            .background(Circle().fill(Color.blue))

    }
}
```

Here's a VStack containing three TextWithCircles:

When rendering the TextWithCircle view, the layout engine first proposes a size to the .frame modifier. Because the initial value of the TextWithCircle's width property is nil, the frame passes the proposed size down the view hierarchy, all the way to the Text view. After Text has laid itself out, the geometry reader in the text's background is laid out using the exact same size as the text. Inside, we use .preference to communicate the text's size up the view tree. Then

.onPreferenceChange happens, the view's state changes, and the view construction and layout process are triggered again. This time, self.width contains the text's width, and the view is laid out correctly. All of this happens during a single rendering pass (i.e. in between screen refreshes).

When dealing with preferences, it's important to keep in mind that changing state during .onPreferenceChange is allowed, but we have to be careful that we don't create an infinite loop: if the state change causes a new layout pass, which causes a preference change, and so on, the view update system will get stuck in a loop.

It's easy to make other mistakes also. For example, if we want to have a padding of 10 points around the text (so that it doesn't hit the edge of the circle), we need to add a call to .padding(10) somewhere. If we do this after the .onPreferenceChange and before the .frame, we'll get stuck in a loop as well (try it out and see the results). Instead, we should add the .padding modifier directly to the text so that it's included in the measured size.

Anchors

Anchors are helpful when passing points and rectangles between different parts of the layout hierarchy. An anchor is a wrapper around a value (for example, a point) that can be resolved inside the coordinate system of a different view somewhere else in the view hierarchy. We can think of anchors as a safer alternative to UIView's convert(_:from:) methods in UIKit.

For example, let's consider a simple tab bar component in which we might want to display a number of tabs (represented as a text). The selected tab should have an underline. When the user taps a new tab, we want to animate both the position and the width of the underline. This means the underline won't be part of the tab item, but rather drawn somewhere else in the view hierarchy. We can use an anchor to communicate the position of the selected tab (via a preference) and resolve the anchor in a different part of the view hierarchy.

We will render the tabs in an HStack using a ForEach. The selected item will propagate its bounds (as an Anchor<CGRect>) using a preference. Further up in the view hierarchy, we can read out the anchor of the selected item and draw a line at that position. As a first step, we need a custom preference key. BoundsKey is similar to our WidthKey above, and it returns the first non-nil value:

```
struct BoundsKey: PreferenceKey {
    static var defaultValue: Anchor<CGRect>? = nil
    static func reduce(value: inout Anchor<CGRect>?,
        nextValue: () -> Anchor<CGRect>?) {
        value = value ?? nextValue()
    }
}
```

Here's the code that draws the items in an HStack and provides the anchor. We add an anchorPreference to our button and use .bounds as the value. The anchorPreference modifier

also takes a transform function that allows us to transform the anchor into something different. In our case, we transform it into an optional anchor (it will be nil unless the current item is selected):

```
struct ContentView: View {
    let tabs: [Text] = [
        Text("World Clock"),
        Text("Alarm"),
        Text("Bedtime")
    ]

    @State var selectedTabIndex = 0

    var body: some View {
        HStack {
            ForEach(tabs.indices) { tabIndex in
                Button(action: {
                    self.selectedTabIndex = tabIndex
                }, label: { self.tabs[tabIndex] })
                .anchorPreference(key: BoundsKey.self, value: .bounds, transform: {
                    anchor in
                    self.selectedTabIndex == tabIndex ? anchor : nil
                })
            }
        }
    }
}
```

Previously we've used .onPreferenceChange to read out preference values, but .onPreferenceChange has the requirement that the value conforms to Equatable. Unfortunately, Anchor does not conform to Equatable. Instead, we'll need to use either .overlayPreference or .backgroundPreference. These modifiers receive the preference value and add an overlay or background, respectively.

To resolve an anchor in the coordinate system of another view, we need to use a GeometryReader. Within the geometry reader's view builder closure, we can use a subscript on the geometry proxy to resolve the anchor in the geometry reader's coordinate system (which is the same as the view we're overlaying the geometry reader on). As a first attempt, we might try to add a modifier to our HStack that adds a Rectangle with the width of the anchor and an offset based on the anchor's minX:

```
.overlayPreferenceValue(BoundsKey.self, { anchor in
    GeometryReader { proxy in
        Rectangle()
            .fill(Color.accentColor)
            .frame(width: proxy[anchor!].width, height: 2)
            .offset(x: proxy[anchor!].minX)
    }
})
```

While the width of our rectangle changes correctly with the selection, the position is strange: it's both vertically and horizontally centered when the first item is selected. The geometry reader has the size of the HStack, but our Rectangle() is much smaller and is centered by default. To solve this, we'll add a frame that's the same size as the geometry reader's proposed size, and we'll use .bottomLeading for the alignment. Finally, we'll add an .animation(.default) to animate any changes (in the next chapter, we'll look at animations in detail):

```
// ...
Rectangle()
    .fill(Color.blue)
    .frame(width: proxy[anchor!].width, height: 2)
    .offset(x: proxy[anchor!].minX)
    .frame(
        width: proxy.size.width,
        height: proxy.size.height,
        alignment: .bottomLeading
    )
    .animation(.default)
```

Now our indicator displays correctly, using the correct width and offset.

Note that the force-unwrapping of the anchor is safe in this specific example. In general, it is only safe as long as there is at least one tab item, or to be more precise, if there is a valid selection. In a real tab bar, we would need to either enforce this, or first check that the anchor isn't nil before force-unwrapping it.

Thus far, we've used two different ways of propagating a view's geometry up the view hierarchy. In the example above, we used .anchorPreference to propagate an anchor. In the previous section, we used a geometry reader to retrieve the view's size and set it as a preference directly.

We can achieve the same outcomes with both approaches, but each approach has its unique tradeoffs. Anchor preferences are advantageous if we need to convert the coordinates from one view into the coordinate system of another view without having to do error-prone manual frame calculations. However, it's more cumbersome to store an anchor's value in a state

property, because we cannot use onPreferenceChange to observe an anchor preference, and we're not allowed to alter the state in overlayPreferenceValue or backgroundPreferenceValue.

On the other hand, if we propagate a simple CGSize or CGPoint value as a preference, we can observe it with onPreferenceChange and store it in a state variable. However, SwiftUI doesn't help us with converting plain geometry values between coordinate systems. But if we just want to propagate a view's size, we don't need to convert between coordinate spaces, and as such, we mostly use a simple CGSize preference value.

Custom Layouts

With geometry readers, anchors, and preferences, we have all the building blocks in place to create more complex custom layouts. As an example, we'll build a custom layout container — a stack that can lay itself out either horizontally or vertically. Since we want to animate the items when the stack switches between the horizontal and vertical layout, we cannot just wrap an HStack and a VStack. Instead, we use a ZStack and lay out the children ourselves. To calculate the layout, we collect the sizes of the children in a private state variable using a geometry reader and the preference system. To position the children, we override their alignment guides in both axes (for the horizontal and vertical layouts).

As a first step, we define a preference key for collecting the sizes of the children. This preference value is a dictionary from the child's index to its size. We start out with an empty dictionary and merge the two dictionaries in the reduce method. Since we don't expect duplicate keys during the merge, it doesn't matter which value we choose in case of a duplicate key, so we use { $1 } to always pick the second value:

```
struct CollectSizePreference: PreferenceKey {
    static let defaultValue: [Int: CGSize] = [:]
    static func reduce(value: inout Value, nextValue: () -> Value) {
        value.merge(nextValue(), uniquingKeysWith: { $1 })
    }
}
```

Next, we create a view modifier that propagates the view's size and index using this preference key. We use the same technique as before — a geometry reader in the background and a Color.clear with a preference:

```
struct CollectSize: ViewModifier {
    var index: Int
    func body(content: Content) -> some View {
        content.background(GeometryReader { proxy in
            Color.clear.preference(key: CollectSizePreference.self,
                value: [self.index: proxy.size])
        })
    }
}
```

With the preference key and the view modifier from above at hand, we can start to build the stack view itself. SwiftUI's built-in stacks take a view builder and figure out whether there's a single view, a tuple view, or a ForEach inside. Since we can't easily do this, our stack view will take an array of elements instead (similar to a ForEach), along with a function to turn an element into a view. We also add parameters for the spacing, the axis, and the alignment, along with a private @State variable to store the computed offsets as CGPoints:

```
struct Stack<Element, Content: View>: View {
    var elements: [Element]
    var spacing: CGFloat = 8
    var axis: Axis = .horizontal
    var alignment: Alignment = .topLeading
    var content: (Element) -> Content
    @State private var offsets: [CGPoint] = []
}
```

Now we have to compute the offsets of the children based on their sizes. Since the sizes are propagated as a preference value, we use onPreferenceChange to update the layout:

```
struct Stack<Element, Content: View>: View {
    // ...
    var body: some View {
        ZStack(alignment: alignment) {
            // ...
        }
        .onPreferenceChange(CollectSizePreference.self,
            perform: self.computeOffsets)
    }
}
```

In the computeOffsets method, we can compute the horizontal and vertical offsets for all children in a single pass. The first view has a .zero offset. For each subsequent view, we take the last offset and add the spacing plus the view's width or height. Note that this implementation expects the sizes dictionary to either be empty (during the first layout pass) or be fully populated for all elements. If this condition is not fulfilled, we simply crash (we could also handle this gracefully with a warning):

```
private func computeOffsets(sizes: [Int: CGSize]) {
    guard !sizes.isEmpty else { return }

    var offsets: [CGPoint] = [.zero]
    for idx in 0..<self.elements.count {
        guard let size = sizes[idx] else { fatalError() }
        var newPoint = offsets.last!
        newPoint.x += size.width + self.spacing
        newPoint.y += size.height + self.spacing
        offsets.append(newPoint)
    }
    self.offsets = offsets
}
```

To get the offset for a specific child view, we implement a private helper method that looks up the offset in the offsets array. During the very first layout pass, the array isn't yet populated, and the helper returns .zero:

```
private func offset(at index: Int) -> CGPoint {
    guard index < offsets.endIndex else { return .zero }
    return offsets[index]
}
```

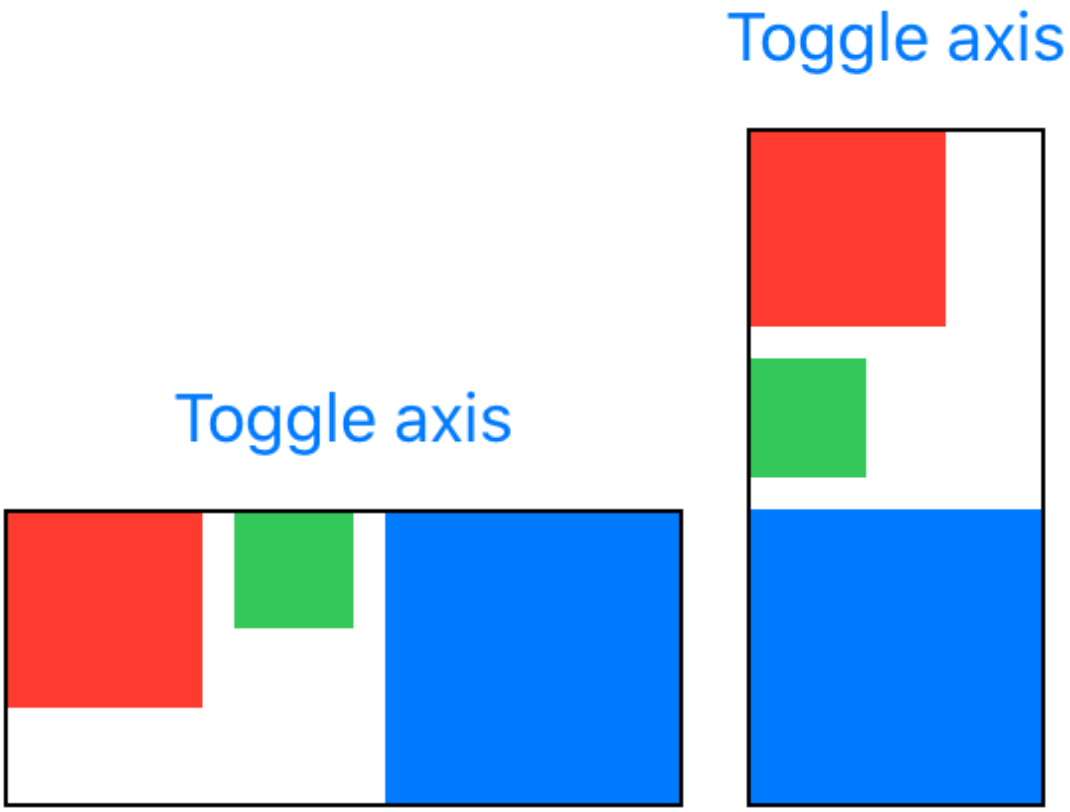
We're now finally ready to write the body of our stack view. We create a ZStack with the given alignment and loop over the indices of the elements array. For each element, we create a view using the content function, apply our CollectSize modifier to measure its size, and add an alignment guide for each axis. For the horizontal axis, we use the horizontal offset we computed when the view is in horizontal mode, or the default alignment when the view is in vertical mode. For the vertical axis, this logic is reversed:

```
var body: some View {
    ZStack(alignment: alignment) {
        ForEach(elements.indices, content: { idx in
            self.content(self.elements[idx])
                .modifier(CollectSize(index: idx))
                .alignmentGuide(self.alignment.horizontal, computeValue: {
                    self.axis == .horizontal
                        ? -self.offset(at: idx).x
                        : $0[self.alignment.horizontal]
                })
                .alignmentGuide(self.alignment.vertical, computeValue: {
                    self.axis == .vertical
                        ? -self.offset(at: idx).y
                        : $0[self.alignment.vertical]
                })
        })
    }
    .onPreferenceChange(CollectSizePreference.self, perform: self.computeOffsets)
}
```

Here's an example using our custom Stack view and displaying three rectangles with different sizes. We can fluidly animate between the horizontal and vertical layout. The black border around the stack shows that its size changes when switching from the horizontal to the vertical layout:

```
struct ContentView: View {
    let colors: [(Color, CGFloat)] = [(.red, 50), (.green, 30), (.blue, 75)]
    @State var horizontal: Bool = true
    var body: some View {
        VStack {
            Button(action: {
                withAnimation(.default) {
                    self.horizontal.toggle()
                }
```

```
            }) { Text("Toggle axis") }
            Stack(elements: colors, axis: horizontal ? .horizontal : .vertical) { item in
                Rectangle()
                    .fill(item.0)
                    .frame(width: item.1, height: item.1)
            }
            .border(Color.black)
        }
    }
}
```



Takeaways

- Use a GeometryReader in any view's background to hook into the layout process and read the proposed size of the view.
- To propagate this size up the view hierarchy, set a preference value using Color.clear.preference(...).
- To modify a view's state based on this preference value, use onPreferenceChange to observe it.
- To read a view's geometry data within another view's coordinate system, set an anchor preference via the anchorPreference API.
- Anchor preferences can be read using the overlayPreferenceValue or backgroundPreferenceValue methods, within which we create a geometry reader to

resolve the anchor in the target coordinate system.

Exercises

Create a Table View

In this exercise, you'll create a view to display tabular data, i.e. multiple rows with multiple columns. Each column should adapt its width to the widest item in the column. For example, here's a sample table:

	Monday	Tuesday	Wednesday
Berlin	Cloudy	Mostly Sunny	Sunny
London	Heavy Rain	Cloudy	Sunny

It's generated using the following code:

```
struct ContentView: View {
    var cells = [
        [Text(""), Text("Monday").bold(), Text("Tuesday").bold(),
            Text("Wednesday").bold()],
        [Text("Berlin").bold(), Text("Cloudy"), Text("Mostly\nSunny"), Text("Sunny")],
        [Text("London").bold(), Text("Heavy Rain"), Text("Cloudy"), Text("Sunny")],
    ]
    var body: some View {
        Table(cells: cells)
            .font(Font.system(.body, design: .serif))
    }
}
```

Note that each column should be aligned to the leading edge, and each row should be aligned to the top.

Step 1: Measure the Cells

Create a new preference key that you can use to store the maximum width per column. Add a helper method to View that reads the view's size and stores it in the WidthPreference for a specific column.

Step 2: Lay out the Table

Arrange the cells using SwiftUI's built-in stacks. Apply the width measurement to each cell and set a frame based on the maximum width measured for the cell's column.

Bonus Exercise

Once your table is working, you can make the table cells selectable. Here's the table from above with the cell at row one and column one selected:

	Monday	Tuesday	Wednesday
Berlin	Cloudy	Mostly Sunny	Sunny
London	Heavy Rain	Cloudy	Sunny

It's relatively easy to add support for selecting items by adding a border around the item that got tapped. However, showing a selection rectangle that animates from cell to cell when the selection changes requires some more effort. We encourage you to try both kinds of solutions!

Animations

Since the very beginning of iOS, animations have been a key part of the user experience. Scroll views animate their content fluidly and have a bounce animation when they reach the end. Likewise, when you tap an app icon on your home screen, the app animates in from the icon's position. These animations aren't just ornamental; they provide the user with context.

SwiftUI has animations built into it from the start. You can use implicit animations and explicit animations, and you can even take full manual control of how things move over time onscreen. First, we'll show some examples of what you can do with basic implicit animations, and then we'll examine how animations work under the hood. Finally, we'll show how you can create custom animations.

Implicit Animations

An implicit animation is part of the view tree: by adding an .animation modifier to a view, any change to that view is automatically animated. As a first example, we'll create a rounded button that changes its color and size when it's tapped:

```
struct ContentView: View {
    @State var selected: Bool = false
    var body: some View {
        Button(action: { self.selected.toggle() }) {
            RoundedRectangle(cornerRadius: 10)
                .fill(selected ? Color.red : .green)
                .frame(width: selected ? 100 : 50, height: selected ? 100 : 50)
        }.animation(.default)
    }
}
```

When tapped, the button rectangle fluidly animates its size between (50, 50) and (100, 100) points, and the color changes from red to green. Creating animations in this way feels a bit like Magic Move in Apple's Keynote application: we define the starting point and the end point, and the software figures out how to animate between the two.

Animations in SwiftUI are part of the view update system, which we looked at in chapter two. Like other view updates, animations can only be triggered through state changes (in the example above, by toggling the selected property). When we add a modifier like .animation(.default) to a view tree, SwiftUI animates the changes between the old and the new view tree on view updates.

Because animations are driven by state changes, some animations require us to get creative. For example, what if we want to build something like iOS's built-in activity indicator, in which an image rotates infinitely? We can solve this through the use of a few tricks. First of all, we need some state that we can change to trigger the animation, so we'll use a Boolean property, which we immediately set to true when the view appears. Second, we need to repeat the animation indefinitely by adding repeatForever to a linear animation. By default, a repeating animation reverses itself every other

repeat, but we don't want that (it would cause the indicator to rotate a full turn and then rotate backward), so we specify autoreverses to be false:

```
struct LoadingIndicator: View {
    @State private var animating = false
    var body: some View {
        Image(systemName: "rays")
            .rotationEffect(animating ? Angle.degrees(360) : .zero)
            .animation(Animation
                .linear(duration: 2)
                .repeatForever(autoreverses: false)
            )
            .onAppear { self.animating = true }
    }
}
```

While the solution above does not feel very clean due to the use of onAppear and the Boolean state property, we were at least able to hide all the implementation details. Using this loading indicator is as simple as writing LoadingIndicator().

Transitions

The animations we've looked at so far animate a view that's onscreen from one state to another. But sometimes we might want to animate the insertion of a new view or the removal of an existing view. SwiftUI has a specific construct for this purpose: transitions. For example, here's a transition that animates a rectangle on and off the screen using a slide animation. When the rectangle gets inserted into the view tree, it animates in from the left, and when it gets removed from the view tree, it animates to the right:

```
struct ContentView: View {
    @State var visible = false
    var body: some View {
        VStack {
            Button("Toggle") { self.visible.toggle() }
            if visible {
                Rectangle()
                    .fill(Color.blue)
                    .frame(width: 100, height: 100)
                    .transition(.slide)
                    .animation(.default)
            }
        }
    }
}
```

Note that transitions don't animate by themselves — we still have to enable animations. Just like before, we use .animation(.default) for this. We can also combine transitions. For example, AnyTransition.move(edge: .leading).combined(with: .opacity) moves the view from and to the leading edge and performs a fade at the same time. To further customize transitions, we can use .asymmetric, which lets us specify one transition for insertion and another for removal of the view.

How Animations Work

Let's consider our animated rounded rectangle, but with just its width animated. We'll also change the animation to be five seconds long and linear:

```
struct AnimatedButton: View {
    @State var selected: Bool = false
    var body: some View {
        Button(action: { self.selected.toggle() }) {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.green)
                .frame(width: selected ? 100 : 50, height: 50)
        }.animation(.linear(duration: 5))
    }
}
```

To animate the rounded rectangle's width value from 50 to 100, SwiftUI needs to interpolate between those two values. We know the animation takes five seconds to complete, and at any point in time, the value of width has a linear relation to the time: at the beginning it will be 50, after one second it will be 60, after three seconds it will be 80, and after five or more seconds it will be 100.

During the animation, the width value is computed by SwiftUI in two parts. First, SwiftUI computes the animation's *progress* using the Animation value we specified (this is .linear in the example above; we'll look at animation curves in more detail later). Typically, the progress of an animation is a value between 0 and 1, where 0 is the start of the animation, and 1 is the end of the animation (however, some animations can "bounce" and have values that are greater than 1 or less than 0).

Given the animation's progress, SwiftUI then interpolates between the start and end values of the properties that have changed, e.g. the fill color and the size in the example at the beginning of this chapter, or just the rectangle's width in the simplified version above.

To animate values, SwiftUI uses the Animatable protocol, which has only one requirement: the animatableData property with a type that conforms to the VectorArithmetic protocol. VectorArithmetic types can be added and subtracted, or multiplied with a Double. With these operations, SwiftUI can compute the difference between the start and end values of the animation and multiply it by the current progress. Taking the rectangle's width animation from 50 to 100 as an example, SwiftUI calculates the current width as 50 + (100 - 50) * progress. Or, when animating from green to red, we could write it as .green + (.red - .green) * progress. In general terms, SwiftUI computes the current value of an animatableData property as startValue + (endValue - startValue) * progress.

When specifying an implicit animation (like .animation(.default)) for a view subtree, all the animatable properties within that subtree will get animated. However, we can also call .animation(nil), which disables all animations (even explicit ones) within that subtree. In theory, we can turn animations on and off for specific subtrees by nesting calls that enable and disable animations in between view modifiers.

We say “in theory” because in practice it’s not obvious what enabling and disabling animations at certain points of the view tree will do. Here’s the example from beginning of this chapter with an added .animation(nil) line:

```
struct ContentView: View {
    @State var selected: Bool = false
    var body: some View {
        Button(action: { self.selected.toggle() }) {
            RoundedRectangle(cornerRadius: 10)
                .fill(selected ? Color.red : .green)
                .animation(nil)
                .frame(width: selected ? 100 : 50, height: 50)
        }.animation(.linear(duration: 5))
    }
}
```

We might expect that the frame will be animated but the color won’t (since it is within an .animation(nil) call). What happens though is that neither the frame nor the color gets animated. The frame modifier just passes on the correct frame for the current state, but it doesn’t get animated itself. The actual frame animation happens on the level of the rounded rectangle shape, where we’ve already disabled animations.

Let’s take the same example and add a rotation effect after the frame modifier:

```
Button(action: { self.selected.toggle() }) {
    RoundedRectangle(cornerRadius: 10)
        .fill(selected ? Color.red : .green)
        .animation(nil)
        .frame(width: selected ? 100 : 50, height: 50)
        .rotationEffect(Angle.degrees(selected ? 45 : 0))
}.animation(..linear(duration: 5))
```

The frame and the color still don’t animate, just like before. However, the rotation animates smoothly between 0 and 45 degrees. The rotation effect doesn’t just pass the information about the rotation on to the shape; it performs the animation itself.

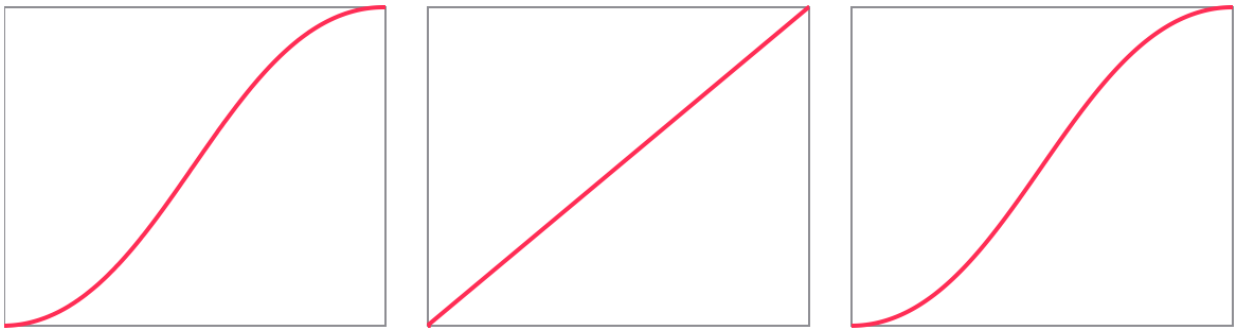
When trying to control animations this way, the behavior depends on opaque implementation details and is difficult to predict. Therefore, we strongly recommend using explicit animations (which we’ll look at below) instead.

Animation Curves

In the example above, we’ve specified a .linear(duration: 5) animation, which means the animation will run at constant speed for five seconds. In other words, the .linear(duration: 5) Animation value provides an animation curve that interpolates the progress evenly from zero to one over the course of five seconds. Since linear animations often look unnatural when run at normal speed, we can use other built-in animation curves — e.g. an ease-in/ease-out curve that specifies .easeInOut(duration: 0.35) to start the movement slow, then pick up speed, and finally slow down again.

An animation curve can be seen as a function that takes time as the input and returns how far the animation has progressed at that point in time. Here’s what some of the built-in animation curves

look like when we visualize them. The horizontal axis shows time, and the vertical axis shows how far the animation has progressed:

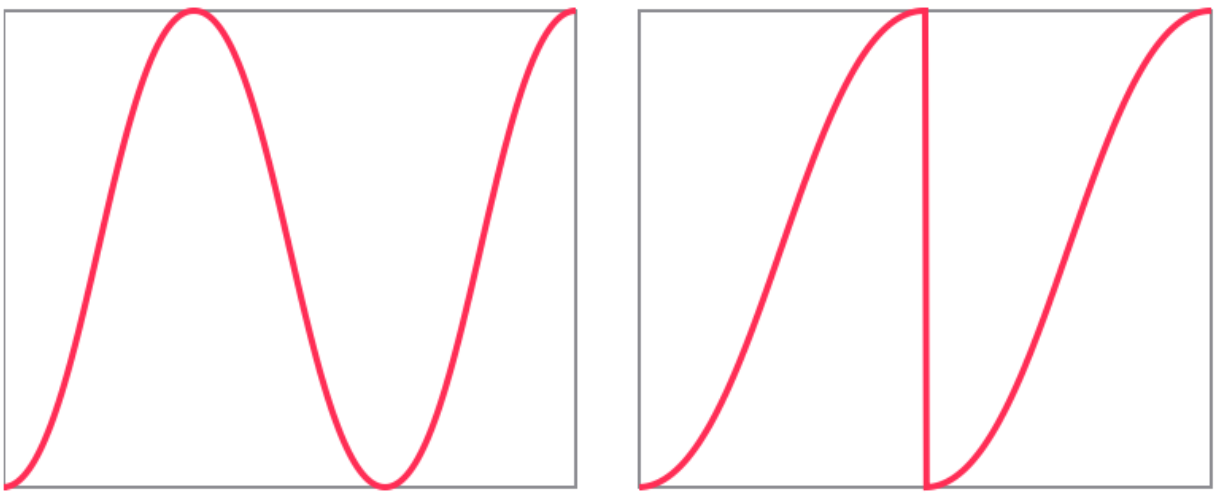


.default, .linear and .easeInOut

We can change Animation values with a few modifiers — for example, we can slow down an animation ten times by calling .speed(0.1), or we can delay an animation for two seconds by calling .delay(2). Another interesting modifier is repeatCount, which lets us repeat an animation multiple times. For example, here's an animation that starts green and pulses between green and red before finally settling on red:

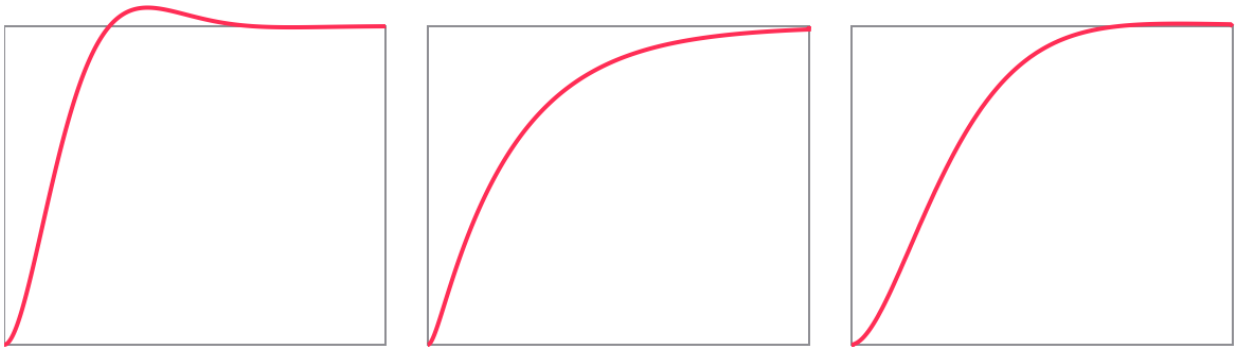
```
Rectangle()
    .fill(selected ? Color.red : Color.green)
    .animation(Animation.default.repeatCount(3))
```

When repeating an animation, the first repeat animates in the forward direction, the second repeat in the reverse direction, and so on. We can turn this behavior off by passing false to the the autoreverses parameter of repeatCount. Visualized, the animation curves of repeated animations look like this:



.default.repeatCount(3) and .default.repeatCount(2, autoreverses: false)

There are a number of other built-in animation curves, most notably for spring animations and custom animation curves (through control points). Note that it's possible for animations to overshoot. For example, for some spring animations, a view might move from its start position, to just beyond its end position, and finally to the end position. We can see this in some of the animation curves as well:



.interpolatingSpring, .interactiveSpring and .spring()

One interesting property of the way SwiftUI implements animations is that additive animations are supported by design. When the state changes while we're in the middle of an animation, the new animation starts from the current state of the previous animation.

Explicit Animations

The implicit animations we've used so far are defined at the view level: we create an animatable view by calling .animation on a view. When we create an implicit animation, anytime the view tree is recomputed, changes of animatable properties in this view tree are animated. Although this is convenient, implicit animations sometimes produce surprising effects. As an example, consider this alternative version of a loading indicator, which animates a small dot along a circle:

```
struct LoadingIndicator: View {
    @State var appeared = false
    let animation = Animation
        .linear(duration: 2)
        .repeatForever(autoreverses: false)
    var body: some View {
        Circle()
            .fill(Color.accentColor)
            .frame(width: 5, height: 5)
            .offset(y: -20)
            .rotationEffect(appeared ? Angle.degrees(360) : .zero)
            .animation(animation)
            .onAppear { self.appeared = true }
    }
}
```

When we run the above code on iOS, at first, the animation seems to work as expected. But the moment we rotate the simulator (or device), the dot follows a strange path: it definitely doesn't animate in a circle anymore. The change in device orientation causes this problem. Because we used an implicit animation, SwiftUI will animate the frame change of the dot (resulting from the device rotation) using the same infinite animation.

The solution to this problem is to use an explicit animation — we only want to animate the changes in the view tree that are caused by the change of the `self.appeared` state property. To do so, we remove the implicit animation and instead wrap our state change in a call to `withAnimation` (an explicit animation):

```
Circle()
    .fill(Color.accentColor)
    .frame(width: 5, height: 5)
    .offset(y: -20)
    .rotationEffect(appeared ? Angle.degrees(360) : .zero)
    .onAppear {
        withAnimation(self.animation) {
            self.appeared = true
        }
    }
```

Now the circle keeps animating correctly when the device is rotated. In talking with other SwiftUI developers, we've noticed that many prefer to use explicit animations, as implicit animations often have unexpected side effects like the one above.

Custom Animations

Through a creative use of built-in view modifiers, it's often possible to build the animation we want. However, sometimes implementing a custom animatable view modifier is the only way to achieve certain animations.

For example, consider an animation that shakes a view to bring it to the user's attention. When the animation starts, the view should move to the right of its original position, then move to the left of its original position, and finally move back to the starting position. We can't achieve this directly with the `offset` modifier: if we specify `offset(x: animating ? 20 : 0)` with a repeated animation, the view ends up 20 points to the right of its original position at the end of the animation, since that is the new state of the view tree once the animation has been triggered.

Luckily, we don't have to rely on the `offset` modifier for this animation; we can implement our own custom view modifier and conform it to the `AnimatableModifier` protocol (which inherits from both `Animatable` and `ViewModifier`). Our modifier will have a property, `times`, that, when incremented, will shake the view. Our goal is to shake the view once when the property gets increased by 1, twice when the property gets increased by 2, and so on:

```
struct Shake: AnimatableModifier {
    var times: CGFloat = 0
    let amplitude: CGFloat = 10
    var animatableData: CGFloat {
        get { times }
        set { times = newValue }
    }
    func body(content: Content) -> some View {
        return content.offset(x: sin(times * .pi * 2) * amplitude)
    }
}
```

To satisfy the requirements of `AnimatableModifier`, we implement the `animatableData` property, which simply gets and sets the `times` property. Its type, `CGFloat`, already conforms to `VectorArithmetic`. In the implementation of the `body` method, we use `sin` to create a smooth curve for

the shake: if times is a round number, the offset will be 0. In between round numbers (for example, in between 0 and 1), the offset will animate from 0 to 10 to -10 and back to 0.

To expose two properties as animatable, we can wrap them in an AnimatablePair. We can nest AnimatablePairs inside each other to support any number of properties.

To help with the discoverability and readability of our new animation, we add a convenience method to View:

```
extension View {
    func shake(times: Int) -> some View {
        return modifier(Shake(times: CGFloat(times)))
    }
}
```

To test this animation, we'll create a button that shakes each time it gets tapped. To achieve this, we store the number of total taps (starting at 0) in a state property, and we increase it on each tap. We then add our Shake modifier to the button and multiply the number of taps by three:

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

    var body: some View {
        Button("Hello") {
            withAnimation(.linear(duration: 0.5)) {
                self.taps += 1
            }
        }
        .shake(times: taps * 3)
    }
}
```

When we run the code, initially the number of taps will be zero, and there will be no animation. Once we tap the button, the taps state changes from 0 to 1, and the view is rerendered. This time, SwiftUI will notice that the animatableData of the Shake modifier changed from 0 to 3 in the new view tree, and it will animate that change. We use a linear animation curve so that the sine curve of the shake movement doesn't get distorted.

To examine how SwiftUI interpolates between different values during an animation, we can add log statements to the setter of animatableData or the body method of the animatable modifier.

When our animatable modifier animates something that can be expressed as an affine transformation (either in 2D or 3D), we can also use GeometryEffect instead of AnimatableModifier. For example, here's the same modifier written as a GeometryEffect instead:

```
struct ShakeEffect: GeometryEffect {
    var times: CGFloat = 0
    let amplitude: CGFloat = 10
    var animatableData: CGFloat {
        get { times }
        set { times = newValue }
    }
```

```
    func effectValue(size: CGSize) -> ProjectionTransform {
        ProjectionTransform(CGAffineTransform(
            translationX: sin(times * .pi * 2) * amplitude,
            y: 0
        ))
    }
}
```

Unfortunately, there's no AnimatableView protocol (and conforming our Views to Animatable doesn't work either). The only way we know of to write custom view animations is through AnimatableModifier or its more specialized version, GeometryEffect. However, Shapes *are* animatable: we can animate properties such as the corner radius of a RoundedRectangle or properties of our own custom shapes.

> *In recent versions of Xcode, AnimatableModifier has been the cause of some bugs. For more details on the problems you might encounter, read the Dancing with Versions section in the Advanced SwiftUI Animations – Part 3 article.*

Custom Transitions

To customize view insertion and removal animations, we can create a custom transition using the AnyTransition.modifier(active:identity:) method. This method takes two view modifiers: one for when the transition is active, and one for when the transition has completed. When a view gets inserted with a transition, the active modifier is applied at the moment of insertion and animates toward the state in which the identity modifier is applied. During removal, the animation happens in reverse: from the identity modifier to the active modifier. As an example, we'll create a custom blur transition. When the view is inserted, it appears blurred and with zero opacity, and when it is removed, it fades out and blurs.

As a first step, we'll create a custom ViewModifier. When active is true, the content is blurred and transparent, and when active is false, the blur radius is set to 0 and the content is opaque:

```
struct Blur: ViewModifier {
    var active: Bool
    func body(content: Content) -> some View {
        return content
            .blur(radius: active ? 50 : 0)
            .opacity(active ? 0 : 1)
    }
}
```

As a second step, we can add a static property to AnyTransition, which makes it easier to discover our new transition:

```
extension AnyTransition {
    static var blur: AnyTransition {
        .modifier(active: Blur(active: true),
                  identity: Blur(active: false))
    }
}
```

Finally, we can take the previous example and replace our .transition(.slide) with a .transition(.blur). Now the view will fade in from a blurred state when it is inserted, and fade out to a blurred state when it is removed.

Takeaways

- Animations, like view updates, are triggered by state changes.
- Implicit animations (.animation(...)) animate changes to all animatable properties in the view subtree.
- Using explicit animations (withAnimation { ... }) allows for more control and can prevent unwanted side effects.
- Use transitions to animate the insertion and removal of views. It's possible to combine transitions, use different transitions for insertion and removal, and create custom transitions.
- To build custom animations, implement an animatable view modifier (conforming to the AnimatableModifier protocol) or a GeometryEffect and expose the animatable properties as animatableData.

Exercises

Bounce Animation

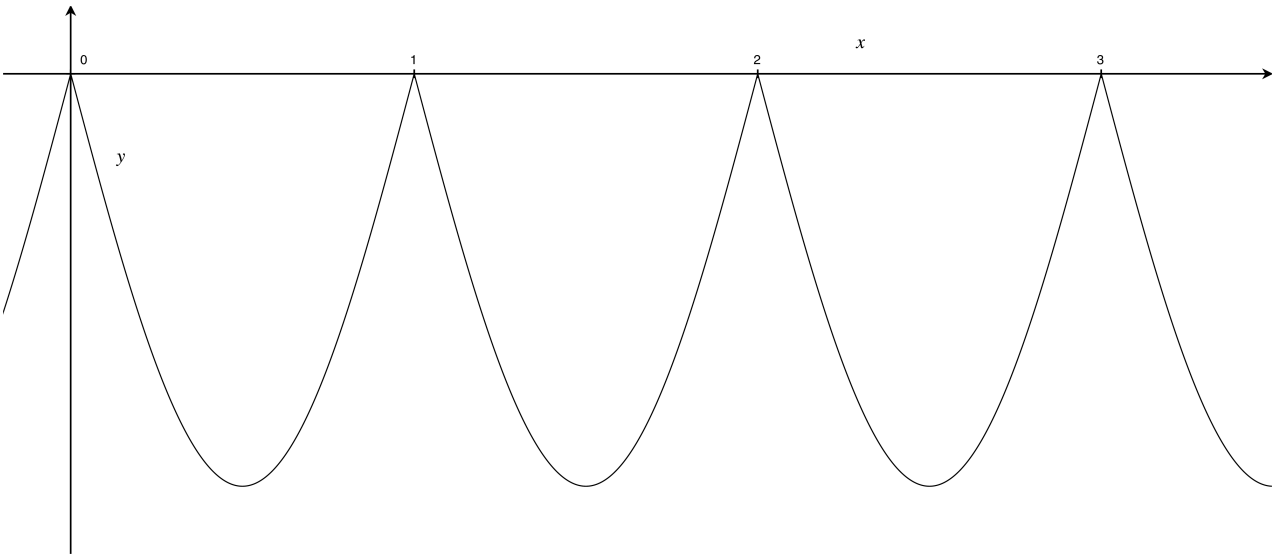
A bounce animation, like the shake animation above, is characterized by the start and end points of the animation being equal. In both cases, we cannot achieve this with a built-in repeating animation. Therefore, you can use the code from the shake animation above as a starting point for building a bounce animation. After a few renamings, this is the skeleton for the bounce modifier:

```
struct Bounce: AnimatableModifier {
    var times: CGFloat = 0
    var amplitude: CGFloat = 10
    var animatableData: CGFloat {
        get { times }
        set { times = newValue }
    }
    func body(content: Content) -> some View {
        // ...
    }
}
```

Step 1: Implement the Animation Curve

Implement the body method so that the animated view bounces once per increment of times. One bounce consists of the view "jumping" straight up and falling back down to its original position.

Here's how the animation curve should look (roughly):



Step 2: View Extension

Expose the new bounce animation via an extension on View, taking the number of bounces as its parameter.

Bonus Exercise

- Add some elasticity to the bounce, i.e. the animated view should bounce a few times in a dampened fashion after it has fallen down onto the "ground." You can make all the parameters of this animation configurable.
- Use the bounce modifier to create a custom bounce transition.

Path Animations

In this exercise, you'll draw a line graph based on some data points. The line graph should be drawn as a shape, so that it can fit itself into any proposed size. The graph should take an array of CGFloats as its input, with numbers in the range of zero to one. Numbers outside of that range are drawn out of bounds. We expect the API to work like this:

```
let sampleData: [CGFloat] = [0.1, 0.7, 0.3, 0.6, 0.45, 1.1]

LineGraph(dataPoints: sampleData)
    .stroke(Color.red, lineWidth: 2)
    .border(Color.gray, width: 1)
```

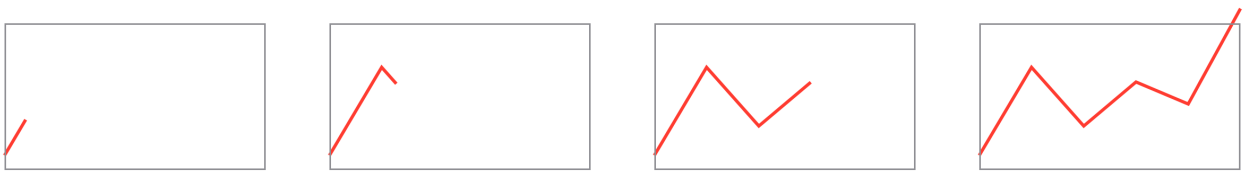
It should draw like this:

Step 1: Implement the Shape

As a first step, implement the shape that draws the graph.

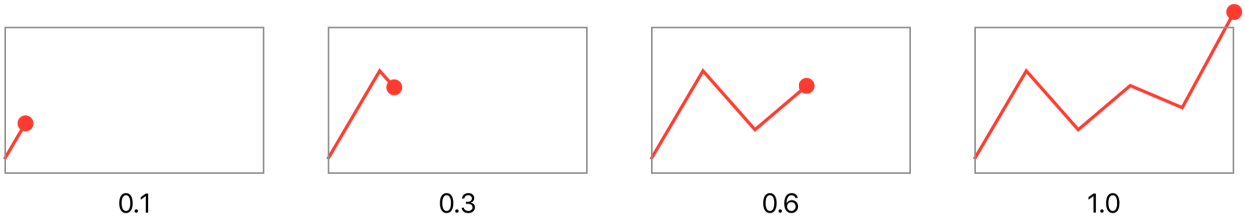
Step 2: Animate the Shape

Animate the path of the shape from an empty path to the entire path. For example, here's the previous graph drawn at different stages of the animation:



Bonus: Add a Leading Dot

Add a leading dot that follows the tip of the graph as it animates:



To achieve this, you can create a GeometryEffect named PositionOnShapeEffect which moves a view (e.g. the dot) to a certain offset on a Path. For example, at offset 1 it is at the end of the path, and at offset 0 it is at the beginning of the path. Combine this with a custom extension on View to make sure you can use it with a Shape (filling the proposed width) rather than a Path. The code for the dot should look something like this at the call site:

```
Circle()
    .fill(Color.red)
    .frame(width: 10, height: 10)
    .position(on: LineGraph(dataPoints: sampleData), at: visible ? 1 : 0)
```

Conclusion

We hope this book gave you a good overview of the mental model of SwiftUI. If you haven't already done so, we definitely recommend going back and doing all the exercises.

This book is not a complete reference, and there are a number of concepts you can explore from here. For example, there are many platform-specific topics we haven't covered, ranging from scroll views to integration with AppKit and UIKit. We expect these aspects to evolve significantly over the next few releases of SwiftUI.

Likewise, with the exception of its most essential parts, we have not discussed Combine. Because SwiftUI's integration with model objects is built upon Combine, reading up on it is a great idea.

We hope to update this book as the platform evolves.

For further reading, here are a number of resources we found helpful:

- https://swiftui-lab.com/
- https://netsplit.com/category/swiftui/
- https://troz.net/post/2019/swiftui-for-mac-1/
- https://www.bigmountainstudio.com/swiftui-views-book
- https://www.hackingwithswift.com/quick-start/swiftui/

Florian and Chris

March 2020

Exercise Solutions

The solutions for the exercises are also available on GitHub.

Chapter 2: View Updates

Step 1: Load the Metadata

The Photo struct simply matches the fields of the JSON API and can be conformed to Codable automatically:

```
struct Photo: Codable, Identifiable {
    var id: String
    var author: String
    var width, height: Int
    var url, download_url: URL
}
```

Step 2: Create an ObservableObject

Here's our implementation of Remote, which stores the loaded data as an optional result:

- When the value is nil, it means it's not loaded yet.
- When the value is .failure, it means something went wrong during the loading.
- When the value is .success, it means the loading worked.

We need to mark our storage as @Published so that SwiftUI will trigger objectWillChange when the property changes. For simplicity's sake, we also expose a computed property value that combines the error state and unloaded state into one:

```
final class Remote<A>: ObservableObject {
    @Published var result: Result<A, Error>? = nil // nil means not loaded yet
    var value: A? { try? result?.get() }

    let url: URL
    let transform: (Data) -> A?

    init(url: URL, transform: @escaping (Data) -> A?) {
        self.url = url
```

```
        self.transform = transform
    }

    func load() {
        URLSession.shared.dataTask(with: url) { data, _, _ in
            DispatchQueue.main.async {
                if let d = data, let v = self.transform(d) {
                    self.result = .success(v)
                } else {
                    self.result = .failure(LoadingError())
                }
            }
        }.resume()
    }
}
```

Note that we also need to switch to the main thread before setting the value. SwiftUI expects all state changes to happen on the main thread.

Step 3: Display the List

First, we added an @ObservedObject property to our ContentView. If you forget to mark the property as @ObservedObject, the code will compile, but SwiftUI won't observe the changes, and your interface won't update.

The second step is the if condition to check whether or not the data has loaded. When the data is available, we show it in a List; otherwise, we display a Text. When the Text first appears onscreen, we start loading the data. This ensures that the data really loads when needed, and not when the view is constructed:

```
struct ContentView: View {
    @ObservedObject var items = Remote(
        url: URL(string: "https://picsum.photos/v2/list")!,
        transform: { try? JSONDecoder().decode([Photo].self, from: $0) }
    )

    var body: some View {
        NavigationView {
            if items.value == nil {
                Text("Loading...")
                    .onAppear { self.items.load() }
            } else {
                List {
                    ForEach(items.value!) { photo in
                        Text(photo.author)
                    }
                }
            }
        }
```

```
    }
}
```

Step 4: Display the Image

As a first step, we wrap the Text of each list item inside a NavigationLink:

```
ForEach(items.value!) { photo in
    NavigationLink(destination: PhotoView(photo.download_url), label: {
        Text(photo.author)
    })
}
```

The photo view uses the download URL to construct a Remote value. When the view first appears, it starts loading the image. Once the image has loaded, the Remote value will fire an objectWillChange and the view renders the image instead:

```
struct PhotoView: View {
    @ObservedObject var image: Remote<UIImage>

    init(_ url: URL) {
        image = Remote(url: url, transform: { UIImage(data: $0) })
    }

    var body: some View {
        Group {
            if image.value == nil {
                Text("Loading...")
                    .onAppear { self.image.load() }
            } else {
                Image(uiImage: image.value!)
                    .resizable()
                    .aspectRatio(image.value!.size, contentMode: .fit)
            }
        }
    }
}
```

Chapter 3: Environment

Configurable Knob Color

Step 1: Create an Environment Key

We create a new type named ColorKey that conforms to the EnvironmentKey protocol:

```
fileprivate struct ColorKey: EnvironmentKey {
    static let defaultValue: Color? = nil
}
```

Note that we're using an optional Color. If the color value in the environment is nil, the knob view will use the default colors based on the current color scheme. To provide a keypath for this key, we also have to add a knobColor property to EnvironmentValues:

```
extension EnvironmentValues {
    var knobColor: Color? {
        get { self[ColorKey.self] }
        set { self[ColorKey.self] = newValue }
    }
}
```

Step 2: Create an @Environment Property

In the knob view, we create a fillColor property that determines whether or not a color value is present in the environment, and if not, it substitutes the default color:

```
struct Knob: View {
    // ...
    @Environment(\.knobColor) var envColor

    private var fillColor: Color {
        envColor ?? (colorScheme == .dark ? Color.white : Color.black)
    }

    var body: some View {
         KnobShape(pointerSize: pointerSize ?? envPointerSize)
            .fill(fillColor)
            // ...
    }
}
```

Step 3: Control the Color with a Slider

In the content view, we create a toggle to specify whether or not the default color should be used, along with a slider to control the hue of the custom color. Then we use our custom knobColor method on View to provide the color value to the knob via the environment:

```
struct ContentView: View {
    // ...
    @State var useDefaultColor = true
    @State var hue: Double = 0
```

```
    var body: some View {
        VStack {
            Knob(value: $value)
                .frame(width: 100, height: 100)
                .knobPointerSize(knobSize)
                .knobColor(useDefaultColor
                    ? nil
                    : Color(hue: hue, saturation: 1, brightness: 1)
                )
            // ...
            HStack {
                Text("Color")
                Slider(value: $hue, in: 0...1)
            }
            Toggle(isOn: $useDefaultColor) {
                Text("Default Color")
            }
            // ...
        }
    }
}
```

Chapter 4: View Layout

Collapsible HStack

To create the collapsible HStack, we use a regular HStack with a particular frame modifier on each of its children. When the stack is expanded, we set a nil width for the item (i.e. we don't interfere with the width of the item). When the stack is collapsed, we set a fixed-width frame (to collapsedWidth) on each child except for the last one. It's also important to explicitly set the horizontal alignment to .leading. Otherwise, the children are centered in their frames by default:

```
struct Collapsible<Element, Content: View>: View {
    var data: [Element]
    var expanded: Bool = false
    var spacing: CGFloat? = 8
    var alignment: VerticalAlignment = .center
    var collapsedWidth: CGFloat = 10
    var content: (Element) -> Content

    func child(at index: Int) -> some View {
        let showExpanded = expanded || index == self.data.endIndex - 1
        return content(data[index])
           .frame(width: showExpanded ? nil : collapsedWidth,
                  alignment: Alignment(horizontal: .leading, vertical: alignment))
```

```
    }

    var body: some View {
        HStack(alignment: alignment, spacing: expanded ? spacing : 0) {
            ForEach(data.indices, content: { self.child(at: $0) })
        }
    }
}
```

Badge View

We start by creating a view for the badge itself, not worrying about the positioning to begin with. The badge is composed of a circle and a text label arranged on top of each other using a ZStack (we could also put the text in an overlay on the circle instead). We wrap both views in an if condition to hide them when the badge count is zero:

```
ZStack {
    if count != 0 {
    Circle()
        .fill(Color.red)
    Text("\(count)")
        .foregroundColor(.white)
        .font(.caption)
    }
}
```

To position the badge, we wrap the ZStack with an offset and a frame modifier. This subtree is put inside an overlay modifier that's aligned with a .topTrailing position. The frame sets a fixed size for the badge. Finally, to center the badge at the corner, we offset it by half of its size:

```
extension View {
    func badge(count: Int) -> some View {
        overlay(
            ZStack {
                if count != 0 {
                    Circle()
                        .fill(Color.red)
                    Text("\(count)")
                        .foregroundColor(.white)
                        .font(.caption)
                }
            }
            .offset(x: 12, y: -12)
            .frame(width: 24, height: 24)
        , alignment: .topTrailing)
    }
}
```

Chapter 5: Custom Layout

Create a Table View

Step 1: Measure the Cells

We create a width preference key that stores a dictionary that maps columns to their maximum widths. As we don't need the widths of all the cells within the same column, we merge the two dictionaries in the key's reduce method by taking the maximum value for a given key:

```
struct WidthPreference: PreferenceKey {
    static let defaultValue: [Int:CGFloat] = [:]
    static func reduce(value: inout Value, nextValue: () -> Value) {
        value.merge(nextValue(), uniquingKeysWith: max)
    }
}
```

To measure the width of a cell, we create a helper on View that takes the column index and stores the view's size as a preference using the key from above:

```
extension View {
    func widthPreference(column: Int) -> some View {
        background(GeometryReader { proxy in
            Color.clear.preference(key: WidthPreference.self,
                value: [column: proxy.size.width])
        })
    }
}
```

Step 2: Lay out the Table

The table itself is a VStack of HStacks. For each cell, we measure the width using the widthPreference helper and then use that width to set a frame. During the first layout pass, the columnWidths dictionary is still empty and the frame modifier receives a nil width. During the second pass, the widths of the cells have been propagated and the actual width for the column is used:

```
struct Table<Cell: View>: View {
    var cells: [[Cell]]
    let padding: CGFloat = 5
    @State private var columnWidths: [Int: CGFloat] = [:]
```

```
    func cellFor(row: Int, column: Int) -> some View {
        cells[row][column]
            .widthPreference(column: column)
            .frame(width: columnWidths[column], alignment: .leading)
            .padding(padding)
    }

    var body: some View {
        VStack(alignment: .leading) {
            ForEach(cells.indices) { row in
                HStack(alignment: .top) {
                    ForEach(self.cells[row].indices) { column in
                        self.cellFor(row: row, column: column)
                    }
                }
                .background(row.isMultiple(of: 2) ?
                    Color(.secondarySystemBackground) : Color(.systemBackground)
                )
            }
        }
        .onPreferenceChange(WidthPreference.self) { self.columnWidths = $0 }
    }
}
```

Bonus Exercise

The key to the more difficult solution of this bonus exercise, i.e. a selection rectangle that animates from cell to cell when the selection changes, is to draw the rectangle not as a border on the cell, but as a separate rectangle outside of the stacks.

The first step is to propagate not just the widths, but also the heights of the cells. Then we set both the width and the height on the cells. This is how the adapted cellFor method looks:

```
func cellFor(row: Int, column: Int) -> some View {
    cells[row][column]
        .sizePreference(row: row, column: column)
        .frame(width: columnWidths[column], height: columnHeights[row],
            alignment: .topLeading)
        .padding(padding)
}
```

sizePreference sets the preference values for the width and the height.

In addition, we create a new preference that contains the bounds anchor for the currently selected cell, so that we later know where to draw the selection rectangle:

```
self.cellFor(row: row, column: column)
    .anchorPreference(key: SelectionPreference.self, value: .bounds, transform: {
        self.isSelected(row: row, column: column) ? $0 : nil
```

```
    })
    // ...
```

To know which cell has been selected, we add a tap gesture to each cell and set the selection as a (row, column) value on a state property:

```
self.cellFor(row: row, column: column)
    // ...
    .onTapGesture {
        withAnimation(.default) {
            self.selection = (row: row, column: column)
        }
    }
```

Finally, we add an overlayPreferenceValue on the outer VStack to read out the bounds anchor of the selected cell, and we use a GeometryReader to resolve the anchor and draw a rectangle:

```
struct Table<Cell: View>: View {
    // ...
    var body: some View {
        VStack(alignment: .leading) { /* ... */ }
        .overlayPreferenceValue(SelectionPreference.self) {
            SelectionRectangle(anchor: $0)
        }
    }
}

struct SelectionRectangle: View {
    let anchor: Anchor<CGRect>?
    var body: some View {
        GeometryReader { proxy in
            ifLet(self.anchor.map { proxy[$0] }) { rect in
                Rectangle()
                    .fill(Color.clear)
                    .border(Color.blue, width: 2)
                    .offset(x: rect.minX, y: rect.minY)
                    .frame(width: rect.width, height: rect.height)
            }
        }
    }
}
```

The ifLet helper abstracts away the force-unwrapping of the optional anchor after we've checked for nil using an if statement. The full source code of this solution is available on GitHub.

Chapter 6: Animations

Bounce Animation

Step 1: Implement the Animation Curve

For the bounce animation, we use a sine curve, just like we did for the shake animation in this chapter. However, we take the absolute value of the sine function and negate it, since we only want to "jump" the view up:

```
struct Bounce: AnimatableModifier {
    var times: CGFloat = 0
    let amplitude: CGFloat = 30
    var animatableData: CGFloat {
        get { times }
        set { times = newValue }
    }
    func body(content: Content) -> some View {
        return content.offset(y: -abs(sin(times * .pi)) * amplitude)
    }
}
```

Step 2: View Extension

The view extension simply wraps a modifier call:

```
extension View {
    func bounce(times: Int) -> some View {
        return modifier(Bounce(times: CGFloat(times)))
    }
}
```

Bonus Exercise

To implement a dampened bouncing movement, the equation to calculate the view's y coordinate has to be modified. For some ideas, you can look up dampened sine wave or harmonic oscillator.

Path Animations

Step 1: Implement the Shape

Our implementation only works for graphs that have at least two data points. We start by moving the path to the first data point, and we add a line to all the other points. In SwiftUI, the y axis has 0 at the top, and in a line graph, the y axis is typically drawn with 0 at the bottom, so we flip the coordinate system by always using (1-point) instead of point. The x and y coordinates are then scaled by the shape's width and height:

```
struct LineGraph: Shape {
    var dataPoints: [CGFloat]
    func path(in rect: CGRect) -> Path {
        Path { p in
            guard dataPoints.count > 1 else { return }
            let start = dataPoints[0]
            p.move(to: CGPoint(x: 0, y: (1-start) * rect.height))
            for (offset, point) in dataPoints.enumerated() {
                let x = rect.width * CGFloat(offset) / CGFloat(dataPoints.count - 1)
                let y = (1-point) * rect.height
                p.addLine(to: CGPoint(x: x, y: y))
            }
        }
    }
}
```

Step 2: Animate the Shape

To animate the path of a shape, we can use the trim modifier on Shape. This takes to and from parameters, which are Doubles in the range of 0 to 1. The trim gives us back only part of the underlying shape. Both to and from are animatable, which means that animating our line graph can be done using just the trim modifier:

```
struct ContentView: View {
    @State var visible = false
    var body: some View {
        VStack {
            LineGraph(dataPoints: sampleData)
                .trim(from: 0, to: visible ? 1 : 0)
                .stroke(Color.red, lineWidth: 2)
                .aspectRatio(16/9, contentMode: .fit)
                .border(Color.gray, width: 1)
                .padding()
            Button(action: {
                withAnimation(Animation.easeInOut(duration: 2)) {
                    self.on.toggle()
                }
            }) { Text("Animate") }
        }
    }
}
```

Bonus Exercise

Our solution is to create a custom method on `View` called `position(on:at:)`, which takes a `Shape` and an amount. It is implemented using a `GeometryReader` (which, just like a shape, becomes its proposed size). The geometry reader uses its size to turn the shape into a path, and it then applies a custom `PositionOnShapeEffect` geometry effect. The effect exposes the amount as animatable data so that the position on the path will be animated.

Table of Contents