HACKING WITH SWIFT

SWIFTUI BY EXAMPLE

COMPLETE REFERENCE GUIDE

Get hands-on solutions for common problems

Paul Hudson

SwiftUI by Example

Paul Hudson

Version: 20190613

Contents

How to use Instruments to profile your SwiftUI code and identify slow layouts

Chapter 1
Introduction

A brief explanation of the basics of SwiftUI

Don’t panic!

WWDC19 contains more large announcements than any in years, so if you’re looking at all the news and code coming out from San Jose and feeling like your head is spinning, let’s start with the most important thing: don’t panic.

Yes, I know SwiftUI is a massive new thing that might seem completely alien at first, but in this guide you’ll get lots of hands-on help to guide you through understanding how SwiftUI works so you can be up and running with it *fast*.

Everything you know about UIKit is still useful, and will be for a good few years.

Everything you know about Swift hasn’t changed, so that’s all useful.

Everything you know about the way iOS apps look and feel hasn’t changed.

What *has* changed is how we make our apps. Yes, there’s a lot of new things to learn, but once you’re past the basics you’ll start to recognize lots of common ground with UIKit.

So, again: don’t panic. If you’re already proficient with UIKit you can pick up SwiftUI in less than a day. And if you’re approaching SwiftUI without any knowledge of UIKit, that’s OK too because you’ll find SwiftUI gives you an unparalleled app building experience.

So, trust me: you’ve got this.

Before we start: Please make sure you have macOS 10.15 installed alongside Xcode 11. This combination allows you to see your SwiftUI designs previewed right inside Xcode, which is significantly faster than pushing to the simulator all the time.

What is SwiftUI?

SwiftUI is a user interface toolkit that lets us design apps in a declarative way. That's a fancy way of saying that we tell SwiftUI how we want our UI to look and work, and it figures out how to make that happen as the user interacts with it.

Declarative UI is best understood in comparison to imperative UI, which is what iOS developers were doing before iOS 13. In an imperative user interface we might make a function be called when a button was clicked, and inside the function we'd read a value and show a label – we regularly modify the way the user interface looks and works based on what's happening.

Imperative UI causes all sorts of problems, most of which revolve around *state*, which is another fancy term meaning "values we store in our code". We need to track what state our code is in, and make sure our user interface correctly reflects that state.

If we have one screen with one Boolean property that affects the UI, we have two states: the Boolean might be on or off. If we have two Booleans, A and B, we now have four states:

- A is off and B is off
- A is on and B is off
- A is off and B is on
- A is on and B is on

And if we have three Booleans? Or five? Or integers, strings, dates, and more? Well, then we have lots more complexity.

If you've ever used an app that says you have 1 unread message no matter how many times you try to tell if you've read the darn thing, that's a state problem – that's an imperative UI problem.

In contrast, declarative UI lets us tell iOS about all possible states of our app at once. We might say if we're logged in show a welcome message but if we're logged out show a login button. We don't need to write code to move between those two states by hand – that's the

ugly, imperative way of working!

Instead, we let SwiftUI move between user interface layouts for us when the state changes. We already told it what to show based on whether the user was logged in or out, so when we change the authentication state SwiftUI can update the UI on our behalf.

That's what it means by declarative: we aren't making SwiftUI components show and hide by hand, we're just telling it all the rules we want it to follow and leaving SwiftUI to make sure those rules are enforced.

But SwiftUI doesn't stop there: it also acts as a cross-platform user interface layer that works across iOS, macOS, tvOS, and even watchOS. This means you can now learn one language and one layout framework, then deploy your code anywhere.

SwiftUI vs Interface Builder and storyboards

Every experienced iOS developer is familiar with Interface Builder and storyboards, and perhaps even XIBs too. They might not *like* them, but they are at least *familiar* with them. If you haven't used these before, you should just skip this bit.

Still here? OK – that means you've used IB before and are probably curious how SwiftUI is different.

Well, let me ask you this: have you ever edited a storyboard or XIB by hand?

Probably not. Well, apart from that one time once, but broadly the answer is *no* – storyboards and XIBs contain a fairly large amount of XML that isn't easy to read or easy to edit.

Worse, storyboards have a habit of growing larger and larger over time. Sure, they might *start off* small, but then you add another view controller and another, and another, and suddenly you realize that you have ten screens of data in a single file, and any source control changes you make are suddenly quite painful.

But although being a single point of failure isn't great, and it's basically impossible to see what's changed when someone opens a pull request with a storyboard modification, storyboards and XIBs have a bigger problem.

You see, Interface Builder doesn't know much about our Swift code, and our Swift code doesn't know much about Interface Builder. As a result, we end up with lots of unsafe functionality: we Ctrl-drag from IB into our code to connect something to an action, but if we then delete that action the code still compiles – IB really doesn't mind if the code it intends to call no longer exists.

Similarly, when we create view controllers from a storyboard or dequeue table view cells, we use strings to identify important objects in our code – a system so pervasive, it even has its own name: "stringly typed APIs". Even then we need to use typecasts because Swift can't

know that the table view cell it got back is actually a **MooncakeTableViewCell**.

These problems exist because IB and Swift are very separate things. This isn't a huge surprise – not only does Interface Builder date from way before the original Mac OS X was a thing, but it's also very much designed around the way Objective-C works.

SwiftUI makes a hard break from that past. It's a Swift-only framework, not because Apple has decided that it's time for Objective-C to die but because it lets SwiftUI leverage the full range of Swift's functionality – value types, opaque return types, protocol extensions, and more.

Anyway, we'll get onto exactly how SwiftUI works soon. For now, the least you need to know is that SwiftUI fixes many problems people had with the old Swift + Interface Builder approach:

- We no longer have to argue about programmatic or storyboard-based design, because SwiftUI gives us both at the same time.
- We no longer have to worry about creating source control problems when committing user interface work, because code is much easier to read and manage than storyboard XML.
- We no longer need to worry so much about stringly typed APIs – there are still some, but significantly fewer.
- We no longer need to worry about calling functions that don't exist, because our user interface gets checked by the Swift compiler.

So, I hope you'll agree there are lots of benefits to be had from moving to SwiftUI!

Frequently asked questions about SwiftUI

Lots of people are already asking me questions about SwiftUI, and I've done my best to ask other people who know much more to try to find definitive answers as appropriate.

So, here goes…

Which to learn: SwiftUI or UIKit?

Right now the overwhelming majority of iOS apps are written using UIKit. Some aren't because they use React Native or something else, but they are effectively a rounding error - as far as we're concerned, UIKit and iOS apps go hand in hand.

So, if you ignored SwiftUI for a year, two years, or perhaps even more, there would be no shortage of jobs for you. No one – not even Apple, I think! – expects the iOS community to migrate over to SwiftUI at any sort of rapid pace. There's a lot of code, a lot of time, and a lot of money invested in UIKit apps, and it has a long and happy life ahead of it.

On the other hand, if you were fresh to iOS development and ignored UIKit so you could focus on SwiftUI, chances are you'll have a very hard time finding a job because no one uses it commercially. Heck, as I write this it was only 36 hours ago the thing was actually announced!

Where can SwiftUI be used?

SwiftUI runs on iOS 13, macOS 10.15, tvOS 13, and watchOS 6, or any future later versions of those platforms. This means if you work on an app that must support iOS N-1 or even N-2 – i.e., the current version and one or two before that – then it will be a year or two before you can even think of moving to SwiftUI.

However, it's important you don't think of SwiftUI as being a multi-platform framework similar to Java's Swing or React Native. The official line seems to be that SwiftUI is not a multi-platform framework, but is instead a framework for creating apps on multiple platforms.

That might sound the same, but there's an important difference: Apple isn't saying that you can use identical SwiftUI code on every platform, because some things just aren't possible – there's no way to use the Apple Watch's digital crown on a Mac, for example, and similarly having a tab bar on a watchOS app just wouldn't work.

Does SwiftUI replace UIKit?

No. Many parts of SwiftUI directly build on top of existing UIKit components, such as **UITableView**. Of course, many other parts don't – they are new controls rendered by SwiftUI and not UIKit.

But the point isn't to what extent UIKit is involved. Instead, the point is that we don't *care*. SwiftUI more or less completely masks UIKit's behavior, so if you write your app for SwiftUI and Apple replaces UIKit with a singing elephant in two years you don't have to care – as long as Apple makes the elephant compatible with the same methods and properties that UIKit exposed to SwiftUI, your code doesn't change.

Does SwiftUI use Auto Layout?

While Auto Layout is certainly being used for some things behind the scenes, it's not exposed to us as SwiftUI designers. Instead, it uses a flexible box layout system that will be familiar to developers coming from the web.

Is SwiftUI fast?

SwiftUI is *screamingly* fast – in all my tests so far it seems to outpace UIKit. Having spoken to the team who made it I'm starting to get an idea why: first, they aggressively flatten their layer hierarchy so the system has to do less drawing, but second many operations bypass Core Animation entirely and go straight to Metal for extra speed.

So, yes: SwiftUI is incredibly fast, and all without us having to do any extra work.

Why can't I see the preview of my code?

When working with SwiftUI it's helpful to be able to see both the code for your view and a preview of your view – how it looks – side by side. If you can see the code and not the preview, chances are you have yet to upgrade to macOS 10.15; it's required to make the preview work.

How closely does the code match the preview?

When you make any change to the preview it will also update the generated code. Similarly, if you change the code it will update the user interface too. So, the code and preview are identical and always stay in sync.

Why do my colors look slightly off?

SwiftUI gives us standard system colors like red, blue, and green, but these aren't the pure red, blue, and green you might be used to from **UIColor**. Instead, these are the new style colors that automatically adapt to light and dark mode, which means they will look brighter or darker depending on your system appearance.

Is UIKit dead?

No! Apple has a massive stack of new UIKit functionality airing at this week's WWDC. If Apple are still doing WWDC talks about new features in UIKit, you're quite safe – there's no risk of them retiring it by surprise.

Can you mix views from SwiftUI and UIKit?

Yes! You can embed one inside the other and it works great.

How to follow this quick start guide

This guide is called SwiftUI by Example, because it focuses particularly on providing as many examples as possible, with each one solving real problems you'll face every day.

I have literally tried to structure this so that almost every entry starts with "How to…" because this is about giving you hands-on code you can use in your own projects immediately. That also means I've tried to get to the point as fast as possible and stay there, so if you're looking for a longer, slower introduction to SwiftUI I'm afraid this isn't it.

Already got some experience?

If you've already grabbed the basics of SwiftUI and just want code that solves your problems, by all means just jump in wherever interests you.

My code examples are specifically written for folks who are following along linearly, so if you're want to make those changes you may need to do a little light editing to make it fit your code.

Just starting out?

If you're just starting out with SwiftUI you should read this guide in a linear order – just keep reading and clicking Next until you're done. I've written the guide so that later chapters build on earlier ones, so a linear approach really is a good idea.

If this is you, you should start by creating a new iOS app using the Single View App template. It doesn't matter what you call it, but I would like you to make sure and check the "Use SwiftUI" box otherwise the rest of this guide will be very confusing indeed.

Migrating from UIKit to SwiftUI

If you've used UIKit before, many of the classes you know and love map pretty much directly to their SwiftUI equivalents just by dropping the **UI** prefix. That doesn't mean they are the same thing underneath, just that they have the same or similar functionality.

Here's a list to get you started, with UIKit class names followed by SwiftUI names:

- **UITableView**: **List**
- **UICollectionView**: No SwiftUI equivalent
- **UILabel**: **Text**
- **UITextField**: **TextField**
- **UITextField** with **isSecureTextEntry** set to true: **SecureField**
- **UITextView**: No SwiftUI equivalent
- **UISwitch**: **Toggle**
- **UISlider**: **Slider**
- **UIButton**: **Button**
- **UINavigationController**: **NavigationView**
- **UIAlertController** with style **.alert**: **Alert**
- **UIAlertController** with style **.actionSheet**: **ActionSheet**
- **UIStackView** with horizontal axis: **HStack**
- **UIStackView** with vertical axis: **VStack**
- **UIImageView**: **Image**
- **UISegmentedControl**: **SegmentedControl**
- **UIStepper**: **Stepper**
- **UIDatePicker**: **DatePicker**
- **NSAttributedString**: Incompatible with SwiftUI; use **Text** instead.

There are many other components that are exclusive to SwiftUI, such as a stack view that lets us build things by depth rather than horizontally or vertically.

Dedication

Inside Apple it took an extraordinary amount of effort to design, build, test, document, and ship SwiftUI. As third-party developers we only really see the end result – when a senior Apple staffer gets on stage at WWDC and shows it off to huge applause, when we download the new Xcode to see a huge amount of new functionality, and when we start our own journey of figuring out how to make best use of these incredible new tools.

But SwiftUI started *long* before that as a project from inside the watchOS team – about four years before, from what various folks have said.

Four years.

That's about 1500 days when Apple's engineers were working hard to build something they knew would revolutionize the way we we worked, and would be the fullest expression of what Swift is capable of for UI development. If you think how much work it took to build SwiftUI as we know it today, imagine how much change it's seen as Swift itself went from 1.0 through to 5.1!

These engineers weren't allowed to talk to the public about their work, and even inside Apple only a certain number of people were disclosed on SwiftUI's existence. In order to make SwiftUI a reality folks from the UIKit team, the Swift team, the Xcode team, the developer publications team, and more, all had to come together in secret to work on our behalf, and even today you won't find them taking credit for their incredible work.

The simple truth is that SwiftUI wouldn't have been possible without the extraordinary efforts of many, many people. I wish I could list them here and thank them personally, but the only ones I can be sure of are the people who had "SwiftUI engineer" as their job title during a WWDC session or were people I spoke to in the labs.

So, this book is dedicated to Dave Abrahams, Luca Bernardi, Kevin Cathey, Nate Cook, John Harper, Taylor Kelly, Kyle Macomber, Raj Ramamurthy, Matt Ricketson, Jacob Xiao, and all the dozens of other folks who worked so hard to make SwiftUI what it is today. We may never know how many more folks from AppKit, UIKit, WatchKit, Xcode, Swift, developer

publications, and beyond helped bring SwiftUI to life, but I hope every one of them feels just blown away by the incredibly positive reactions from our community.

I know WWDC can often be quite the “photo finish” where features land only a day or two before the keynote, but you folks pulled it off and we’re very, very grateful.

Chapter 2

Text and images

Getting started with basic controls

What's in the basic template?

Tip: You might think this chapter is totally skippable, but unless you're a Swift genius chances are you should read to the end just to be sure.

The basic Single View App template gives you the following:

1. AppDelegate.swift. This is responsible for monitoring external events, such as if another app tries to send you a file to open.
2. SceneDelegate.swift. This is responsible for managing the way your app is shown, such as letting multiple instances run at the same time or taking action when one moves to the background.
3. ContentView.swift. This is our initial piece of user interface. If this were a UIKit project, this would be the **ViewController** class that Xcode gave us.
4. Assets.xcassets. This is an asset catalog, which stores all the images and colors used in our project.
5. LaunchScreen.storyboard. This is the screen that gets shown while your app is loading.
6. Info.plist is a property list file, which in this instance is used to store system-wide settings for our app – what name should be shown below its icon on the iOS home screen, for example.
7. A group called Preview Content, which contains another asset catalog called Preview Assets.

And that's it – it's a pleasingly small amount of code and resources, which means we can build on it.

The part we really care about – in fact, here it's the only part that matters – is ContentView.swift. This is the main piece of functionality for our app, and it's where we can start trying out various SwiftUI code in just a moment.

First, though: what makes ContentView.swift get shown on the screen?

Well, if you remember I said that SceneDelegate.swift is responsible for managing the way

your app is shown. Go ahead and open SceneDelegate.swift now, and you'll see code like this in there:

```
let window = UIWindow(frame: UIScreen.main.bounds)
window.rootViewController = UIHostingController(rootView:
ContentView())
self.window = window
window.makeKeyAndVisible()
```

That code creates a new **ContentView** instance (that's the main piece of functionality we'll be looking at soon), and places it inside a window so it's visible onscreen. It's effectively bootstrapping our app by showing the first instance of **ContentView**, and from there it's over to us – what do you want to do?

Open ContentView.swift and let's look at some actual SwiftUI code. You should see code like this:

```
import SwiftUI

struct ContentView : View {
    var body: some View {
        Text("Hello World")
    }
}

#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
#endif
```

That's not a lot of code, but it does pack in a great deal.

First, notice how **ContentView** is a struct. Developers familiar with UIKit will know that this is *huge* – we get to benefit from all the immutability and simplicity of values types for our user interface! Folks who *aren't* familiar with UIKit… well, just nod and smile – you never knew the pain we used to have.

Second, **ContentView** conforms to the **View** protocol. Everything you want to show in SwiftUI needs to conform to **View**, and really that means only one thing: you need to have a property called **body** that returns some sort of **View**.

Third, the return type of **body** is **some View**. The **some** keyword is new in Swift 5.1 and is part of a feature called **opaque return types**, and in this case what it means is literally "this will return some sort of **View** but SwiftUI doesn't need to know (or care) what."

Important: Returning **some View** means that the **body** property will return something that conforms to the **View** protocol. You can't return many things or forget to return anything at all – the Swift compiler will refuse to build your code. To be clear, your view body must always return exactly one child view.

Fourth, inside the **body** property there's **Text("Hello World")**, which creates a label of the text "Hello World".

Finally, below **ContentView** is a similar-but-different struct called **ContentView_Previews**. This *doesn't* conform to the **View** protocol because it's specifically there to show view previews inside Xcode as opposed to be on-screen in a real app. This is why you'll see it inside **#if DEBUG** and **#endif** lines – this code is only built into the finished product when our app runs in a debug environment because it doesn't make sense in a production app.

We'll look at each of these components in much more detail soon enough, but first let's take a look at that **Text** component…

How to create static labels with a Text view

Text views show static text on the screen, and are equivalent to **UILabel** in UIKit. At their most basic they look like this:

```
Text("Hello World")
```

Inside the preview window for your content view you're likely to see "Automatic preview updating paused" – go ahead and press Resume to have Swift start building your code and show you a live preview of how it looks.

By default text views are a single line only – if you run out of space the characters will be clipped and replaced with "...", so if our label were longer you'd see that happen. If you want to change this behavior – if you want your text view to run over more lines, you can either do this to make it have a specific number of lines:

```
Text("Hello World")
    .lineLimit(3)
```

Tip: Notice the way **lineLimit(3)** is placed below and to the right of **Text("Hello World")**. This is *not* required, but it does make your code easier to read in the long term.

Alternatively, you can specify **nil** to the **lineLimit()** method, which allows it to run over as many lines as needed:

```
Text("Hello World")
    .lineLimit(nil)
```

Rather than adjusting the line limit, you can also adjust the way SwiftUI truncates your text. The default is to remove text from the end and show an ellipsis there instead, but you can also place the ellipsis in the middle or beginning depending on how important the various parts of your string are.

For example, this truncates your text in the middle:

```
var body: some View {
   Text("This is an extremely long string that will never fit
even the widest of Phones")
      .truncationMode(.middle)
}
```

Regardless of how you truncate the text, what you'll see is that your text view sits neatly centered in the main view. This is the default behavior of SwiftUI – unless it's told to position views somewhere else, it positions them relative to the center of the screen.

How to style text views with fonts, colors, line spacing, and more

Not only do text views give us a predictably wide range of control in terms of how they look, they are also designed to work seamlessly alongside core Apple technologies such as Dynamic Type.

By default a Text view has a "Body" Dynamic Type style, but you can select from other sizes and weights by calling **.font()** on it like this:

```
Text("This is an extremely long string that will never fit even the widest of Phones")
    .lineLimit(nil)
    .font(.largeTitle)
```

We now have two modifiers below our text view, and that's OK – you can just stack them up, and they all take effect.

In particular, now that we have multiple lines you will want to adjust their text alignment so the lines are centered, like this:

```
Text("This is an extremely long string that will never fit even the widest of Phones")
    .lineLimit(nil)
    .font(.largeTitle)
    .multilineTextAlignment(.center)
```

We can control the color of text using the **.foregroundColor()** modifier, like this:

```
Text("The best laid plans")
    .foregroundColor(Color.red)
```

You can also set the background color, but that uses **.background()** because it's possible to

use more advanced backgrounds than just a flat color. Anyway, to give our layout a yellow background color we would add this:

```
Text("The best laid plans")
    .background(Color.yellow)
    .foregroundColor(Color.red)
```

There are even more options. For example, we can adjust the line spacing in our text. The default value is 0, which means there is no extra line spacing applied, but you can also specify position values to add extra spacing between lines:

```
Text("This is an extremely long string that will never fit even the widest of Phones")
    .lineLimit(nil)
    .font(.largeTitle)
    .lineSpacing(50)
```

How to format text inside text views

SwiftUI's text views have an optional **formatter** parameter that lets us customize the way data is presented inside the label. This is important because often values are updated for us when something happens in our program, so by attaching a formatter SwiftUI can make sure our data looks right on our behalf.

For example, this defines a date formatter and uses it to make sure a task date is presented in human-readable form:

```
struct ContentView: View {
    static let taskDateFormat: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateStyle = .long
        return formatter
    }()

    var dueDate = Date()

    var body: some View {
        Text("Task due date: \(dueDate, formatter:
Self.taskDateFormat)")
    }
}
```

That will display something like "Task due date: June 5 2019".

How to draw images using Image views

Use the **Image** view to render images inside your SwiftUI layouts. These can load images from your bundle, from system icons, from a **UIImage**, and more, but those three will be the most common.

To load an image from your bundle and display it inside an image view, you'd just use this:

```
var body: some View {
    Image("example-image")
}
```

Note: I'm not going to repeat the **var body: some View {** part from now on – you get the idea. In the future I'll show it only when it really matters.

To load icons from Apple's San Francisco Symbol set, use the **Image(systemName:)** initializer, passing in the icon string to load, like this:

```
Image(systemName: "cloud.heavyrain.fill")
```

Finally, you can create an image view from an existing **UIImage**. As this requires more code, you'll need to use the **return** keyword explicitly:

```
guard let img = UIImage(named: "example-image") else {
    fatalError("Unable to load image")
}

return Image(uiImage: img)
```

If you're using the system icon set, the image you get back is scalable and colorable, which means you can tint the image using the same **foregroundColor()** modifier you've already seen:

```
Image(systemName: "cloud.heavyrain.fill")
    .foregroundColor(.red)
```

And it also means you can ask SwiftUI to scale up the image to match whatever Dynamic Type text style it accompanies, if any:

```
Image(systemName: "cloud.heavyrain.fill")
    .font(.largeTitle)
```

How to adjust the way an image is fitted to its space

SwiftUI's **Image** view has the ability to be scaled in different ways, just like the content mode of a **UIImageView**.

By default, image views automatically size themselves to their contents, which might make them go beyond the screen. If you add the **resizable()** modifier then the image will instead automatically be sized so that it fills all the available space:

```
Image("example-image")
    .resizable()
```

However, that may also cause the image to have its original aspect ratio distorted, because it will be stretched in all dimensions by whatever amount is needed to make it fill the space.

If you want to keep its aspect ratio you should add an **aspectRatio** modifier using either **.fill** or **.fit**, like this:

```
Image("example-image")
    .resizable()
    .aspectRatio(contentMode: .fill)
```

Note: In the current beta there's a bug where the **.fit** content mode still causes the image to be stretched.

How to render a gradient

SwiftUI gives us a variety of gradient options, all of which can be used in a variety of ways. For example, you could render a text view using a white to black linear gradient like this:

```
Text("Hello World")
    .padding()
    .foregroundColor(.white)
    .background(LinearGradient(gradient: Gradient(colors:
[.white, .black]), startPoint: .top, endPoint: .bottom),
cornerRadius: 0)
```

The colors are specified as an array and you can have as many as you want – by default SwiftUI will space them equally. So, we could go from white to red to black like this:

```
Text("Hello World")
    .padding()
    .foregroundColor(.white)
    .background(LinearGradient(gradient: Gradient(colors:
[.white, .red, .black]), startPoint: .top, endPoint: .bottom),
cornerRadius: 0)
```

To make a horizontal gradient rather than a vertical one, use **.leading** and **.trailing** for your start and end points:

```
Text("Hello World")
    .padding()
    .foregroundColor(.white)
    .background(LinearGradient(gradient: Gradient(colors:
[.white, .red, .black]), startPoint: .leading,
endPoint: .trailing), cornerRadius: 0)
```

How to display solid shapes

If you want simple shapes to use in your app, you can create them directly then just color and position them as needed.

For example, if you wanted a 200x200 red rectangle, you would use this:

```
Rectangle()
    .fill(Color.red)
    .frame(width: 200, height: 200)
```

Similarly, if you wanted a 50x50 blue circle you would use this:

```
Circle()
    .fill(Color.blue)
    .frame(width: 50, height: 50)
```

How to use images and other views as a backgrounds

Rather than specifying a background color, you can specify a background *image* using the same **background()** modifier.

For example, this creates a text view with a large font, then places a 100x100 image behind it:

```
Text("Hacking with Swift")
    .font(.largeTitle)
    .background(
       Image("example-image")
          .resizable()
          .frame(width: 100, height: 100))
```

However, in SwiftUI it doesn't *need* to be an image – you can actually use any kind of view for your background. For example, this creates the same text view then places a 200x200 red circle behind it:

```
Text("Hacking with Swift")
    .font(.largeTitle)
    .background(Circle()
       .fill(Color.red)
       .frame(width: 200, height: 200))
```

By default background views automatically take up as much space as they need to be fully visible, but if you want you can have them be clipped to the size of their parent view using the **clipped()** modifier:

```
Text("Hacking with Swift")
    .font(.largeTitle)
    .background(Circle()
```

```
        .fill(Color.red)
        .frame(width: 200, height: 200))
        .clipped()
```

To be clear, you can use *any* view as your background – another text view if you wanted, for example.

Chapter 3
View layout

Position views in a grid structure and more

How to create stacks using VStack and HStack

Our SwiftUI content views must return one view, which is the view we want them to show. When we want more than one view on screen at a time we need to tell SwiftUI how to arrange them and that's where *stacks* come in.

Stacks – equivalent to **UIStackView** in UIKit – come in three forms: horizontal (**HStack**), vertical (**VStack**) and depth-based (**ZStack**), with the latter being used when you want to place child views so they overlap.

Let's start with something simple. Here's one text view:

```
Text("SwiftUI")
```

If we want to place another below, we can't just write this:

```
var body: some View {
    Text("SwiftUI")
    Text("rocks")
}
```

Remember, we need to return precisely one **View**, so that code won't work.

Instead, we need to place it in a vertical stack so our text views are placed above each other:

```
VStack {
    Text("SwiftUI")
    Text("rocks")
}
```

You'll notice that the vertical stack is placed at the center of the screen, with the labels also being centered and having some automatic space between them.

If you wanted the labels side by side horizontally, replace **VStack** with **HStack** like this:

```
HStack {
    Text("SwiftUI")
    Text("rocks")
}
```

How to customize stack layouts with alignment and spacing

You can add spacing inside your SwiftUI stacks by providing a value in the initializer, like this:

```
VStack(spacing: 50) {
    Text("SwiftUI")
    Text("rocks")
}
```

Alternatively, you can create dividers between items so that SwiftUI makes a small visual distinction between each item in the stack, like this:

```
VStack {
    Text("SwiftUI")
    Divider()
    Text("rocks")
}
```

By default, items in your stacks are aligned centrally. In the case of **HStack** that means items are aligned to be vertically in the middle, so if you have two text views of different heights they would both be aligned to their vertical center. For **VStack** that means items are aligned to be horizontally in the middle, so if you have two text views of different lengths they would both be aligned to their horizontal center.

To adjust this, pass in an alignment when you create your stack, like this:

```
VStack(alignment: .leading) {
    Text("SwiftUI")
    Text("rocks")
}
```

That will align both “SwiftUI” and “rocks” to their left edge, but they will still ultimately sit in the middle of the screen because the stack takes up only as much space as it needs.

You can of course use both alignment and spacing at the same time, like this:

```
VStack(alignment: .leading, spacing: 20) {
    Text("SwiftUI")
    Text("rocks")
}
```

That will align both text views horizontally to the leading edge (that’s left for left to right languages), and place 20 points of vertical space between them.

How to control spacing around individual views using padding

SwiftUI lets us set individual padding around views using the **padding()** modifier. If you use this with no parameters you'll get system-default padding on all sides, like this:

```
VStack {
    Text("SwiftUI")
        .padding()
    Text("rocks")
}
```

But you can also customize how much padding to apply and where. So, you might want to apply system padding to only one side:

```
Text("SwiftUI")
    .padding(.bottom)
```

Or you might want to control how much padding is applied to all sides:

```
Text("SwiftUI")
    .padding(100)
```

Or you can combine the two to add a specific amount of padding to one side of the view:

```
Text("SwiftUI")
    .padding(.bottom, 100)
```

How to layer views on top of each other using ZStack

SwiftUI has a dedicated stack type for creating overlapping content, which is useful if you want to place some text over a picture for example. It's called **ZStack**, and it works identically to the other two stack types.

For example, we could place a large image underneath some text like this:

```
ZStack() {
    Image("example-image")
    Text("Hacking with Swift")
        .font(.largeTitle)
        .background(Color.black)
        .foregroundColor(.white)
}
```

Like the other stack types, **ZStack** can be created with an alignment so that it doesn't always center things inside itself:

```
ZStack(alignment: .leading) {
    Image("example-image")
    Text("Hacking with Swift")
        .font(.largeTitle)
        .background(Color.black)
        .foregroundColor(.white)
}
```

However, it doesn't have a *spacing* property because it doesn't really make sense.

How to return different view types

When we return **some View** from the body of our views, Swift understands that to mean we have one specific return type. For example, if we wanted to flip a coin and show either a "you won!" image or some text that says "Better luck next time", we can't write this:

```
var body: some View {
    if Bool.random() {
        Image("example-image")
    } else {
        Text("Better luck next time")
    }
}
```

That might return either an image or a text view, which isn't allowed – we must return precisely one type.

There are two ways you can fix this. The first option is to wrap your output in a group, so that no matter whether you send back an image or a text view they both go back in a group:

```
var body: some View {
    Group {
        if Bool.random() {
            Image("example-image")
        } else {
            Text("Better luck next time")
        }
    }
}
```

Alternatively, SwiftUI gives us a type-erased wrapper called **AnyView** that we can return:

```
var body: AnyView {
```

```
    if Bool.random() {
        return AnyView(Image("example-image"))
    } else {
        return AnyView(Text("Better luck next time"))
    }
}
```

If you haven't heard of this concept, it effectively forces Swift to forget about what specific type is inside the **AnyView**, allowing them to look like they are the same thing. This has a performance cost, though, so don't use it often.

Although both **Group** and **AnyView** achieve the same result for our layout, it's generally preferable to use **Group** because it's more efficient for SwiftUI.

How to create views in a loop using ForEach

You will commonly find that you want to loop over a sequence to create views, and in SwiftUI that's done using **ForEach**.

Important: It's easy to look at **ForEach** and think it's the same as the **forEach()** method on Swift's sequences, but this is *not* the case as you'll see.

ForEach in SwiftUI is a view struct in its own right, which means you can return it directly from your view body if you want. You provide it an array of items, and you may also need to tell SwiftUI how it how it can identify each of your items uniquely so it knows how to update them when values change. You also pass it a closure to run to create a view for each item in the loop.

For simple loops over ranges, you can pass the range directly into **ForEach**. For example, this counts from 10 down to 1 then adds a message at the end:

```
VStack(alignment: .leading) {
    ForEach((1...10).reversed()) {
        Text("\($0)…")
    }

    Text("Ready or not, here I come!")
}
```

For loops over arrays of simple types, such as strings, integers, colors, and so on, you can use **.identified(by: \.self)** on the array to have SwiftUI use the value itself as the identifier. So, if your array was **["cat", "dog", "monkey"]** then SwiftUI would use those strings themselves as the identifiers for your views.

So, this code creates an array of three colors, loops over them all, and creates text views using each color name and color value:

```
struct ContentView : View {
    let colors: [Color] = [.red, .green, .blue]

    var body: some View {
        VStack {
            ForEach(colors.identified(by: \.self)) { color in
                Text(color.description.capitalized)
                    .padding()
                    .background(color)
            }
        }
    }
}
```

If you have custom types in your array, you should use **identified(by:)** with whatever property inside your type identifies it uniquely.

For example, here's a struct to store test results like this:

```
struct Result {
    var id = UUID()
    var score: Int
}
```

That has an **id** property with a **UUID**, which mean it's guaranteed to be unique – perfect for our purposes. If we wanted to loop over an array of results, creating a text view showing each result in a **VStack**, then we'd use this:

```
struct ContentView : View {
    let results = [Result(score: 8), Result(score: 5),
Result(score: 10)]
```

```
    var body: some View {
        VStack {
            ForEach(results.identified(by: \.id)) { result in
                Text("Result: \(result.score)")
            }
        }
    }
}
```

That tells SwiftUI it can distinguish between views inside the **ForEach** by looking at their **id** property.

Tip: If you make **Result** conform to **Identifiable** protocol, you can just write **ForEach(results)**. Conforming to this protocol means adding an **id** property that uniquely identifies each object, which in our case we already have, so you can just write **struct Result: Identifiable** {!

How to create different layouts using size classes

SwiftUI supports size classes natively by exposing them in the environment for us to read. To use them, first create an **@Environment** object that will store its value, like this:

```
@Environment(\.horizontalSizeClass) var horizontalSizeClass:
UserInterfaceSizeClass?
```

Then you check the value of that property whenever you need, looking for either the **.compact** or **.regular** size class, like this:

```
if horizontalSizeClass == .compact {
   return Text("Compact")
} else {
   return Text("Regular")
}
```

Putting all that together, you might create a view like this:

```
struct ContentView : View {
   @Environment(\.horizontalSizeClass) var horizontalSizeClass:
UserInterfaceSizeClass?

   var body: some View {
      if horizontalSizeClass == .compact {
         return Text("Compact")
      } else {
         return Text("Regular")
      }
   }
}
```

How to place content outside the safe area

By default your SwiftUI views will mostly stay inside the safe area – it will go to the bottom of the screen, but it won't go near any notch at the top of the device.

If you want to change that – if you want your view to be truly full screen, even if that means being partly obscured by a notch or other hardware cut outs – then you should use the **edgesIgnoringSafeArea()** modifier.

For example, this creates a red text view that asks to fill all available space, then sets it to ignore any safe areas so that it goes truly edge to edge.

```
Text("Hello World")
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
maxHeight: .infinity)
    .background(Color.red)
    .edgesIgnoringSafeArea(.all)
```

Chapter 4
Responding to events

Respond to interaction and control your program state

Working with state

All apps change state. For example, the user might have tapped a button to reveal more information, they might have entered some text into a text box, or chosen a date from a date picker – all things that involve the app moving from one state to another.

The problem with state is that it's messy: when it changes we need to spot that change and update our layouts to match. That might sounds simple at first, but as our state grows and grows it becomes increasingly hard – it's easy to forget to update one thing, or to get the update order wrong so that the user interface state doesn't match what was expected.

SwiftUI solves this problem by removing state from our control. When we add properties to our views they are effectively inert – they have values, sure, but changing them doesn't do anything. But if we added the special **@State** attribute before them, SwiftUI will automatically watch for changes and update any parts of our views that use that state.

When it comes to *referring* to some state – for example, telling a state property to change when a toggle switch changes – we can't refer to the property directly. This is because Swift would think we're referring to the value right now rather than saying "please watch this thing." Fortunately, SwiftUI's solution is to place a dollar sign before the property name, which lets us refer to the data itself rather than its current value. I know this is a little confusing at first, but it becomes second nature after an hour or two.

Remember, SwiftUI is *declarative*, which means we tell it all layouts for all possible states up front, and let it figure out how to move between them when properties change. We call this *binding* – asking SwiftUI to synchronize changes between a UI control and an underlying property.

Working with state will cause you a few headaches at first if you're used to a more imperative style of programming, but trust me – once you're through that it's clear sailing.

How to create a toggle switch

SwiftUI's toggle lets users move between true and false states, just like **UISwitch** in UIKit.

For example, we could create a toggle that either shows a message or not depending on whether the toggle is enabled or not, but of course we *don't* want to have to track the state of the toggle by hand – we want SwiftUI to do that for us.

Instead we should define a **@State** Boolean property that will be used to store the current value of our toggle. We can then use that to show or hide other views as needed.

For example:

```
struct ContentView : View {
    @State var showGreeting = true

    var body: some View {
        VStack {
            Toggle(isOn: $showGreeting) {
                Text("Show welcome message")
            }.padding()

            if showGreeting {
                Text("Hello World!")
            }
        }
    }
}
```

I've made that code so that a text view is returned only when **showGreeting** is true, which means the **VStack** will decrease in size when **showGreeting** is false – it doesn't have a second view in its stack.

How to create a tappable button

SwiftUI's button is similar to **UIButton**, except it's more flexible in terms of what content it shows and it uses a closure for its action rather than the old target/action system.

To create a button you would start with code like this:

```
Button(action: {
    // your action here
}) {
    Text("Button title")
}
```

For example, you might make a button that shows or hides some detail text when it's tapped:

```
struct ContentView : View {
    @State var showDetails = false

    var body: some View {
        VStack {
            Button(action: {
                self.showDetails.toggle()
            }) {
                Text("Show details")
            }

            if showDetails {
                Text("You should follow me on Twitter: @twostraws")
                    .font(.largeTitle)
                    .lineLimit(nil)
            }
        }
    }
```

```
}
```

Tip: The classic thing to do when you're learning a framework is to scatter **print()** calls around so you can see when things happen. If you want to try that with your button action, you should first right-click on the play button in the preview canvas and choose "Debug Preview" so that your **print()** calls work.

The title inside the button can be any kind of view, so you can create an image button like this:

```
Button(action: {
    self.showDetails.toggle()
}) {
    Image("example-image")
}
```

How to read text from a TextField

SwiftUI's **TextField** view is similar to **UITextField**, although it looks a little different by default and relies very heavily on binding to state.

To create one, you should pass in the state value it should bind to. For example, this creates a **TextField** bound to a local string, then places a text view below it that shows the text field's output as you type:

```
struct ContentView : View {
    @State var name: String = "Tim"

    var body: some View {
        VStack {
            TextField($name)
            Text("Hello, \(name)!")
        }
    }
}
```

When that's run, you should be able to type into the text field and see a greeting appear directly below.

There are two important provisos when working with text fields. First, they don't have a border by default, so you probably won't see anything – you'll need to tap inside roughly where it is in order to activate the keyboard.

Second, you might find you can't type into the canvas preview of your layout. If you hit that problem, press Cmd+R to build and run your code in the simulator.

How to add a border to a TextField

SwiftUI's **TextField** view has no styling by default, which means it's an empty space on the screen. If that fits the style you want, great – you're done. But many of us will prefer to add a border around the text field to make it clearer.

If you want to get the "rounded rect" text field style that we're used to with **UITextField**, you should use the **.textFieldStyle(.roundedBorder)** modifier, like this:

```
TextField($yourBindingHere)
    .textFieldStyle(.roundedBorder)
```

How to create secure text fields using SecureField

SwiftUI's **SecureField** works almost identically to a regular **TextField** except the characters are masked out for privacy. The underlying value you bind it to is still a plain string, of course, so you can check it as needed.

Here's an example that creates a **SecureField** bound to a local **@State** property so we can show what they typed:

```
struct ContentView : View {
    @State private var password: String = ""

    var body: some View {
        VStack {
            SecureField($password)
            Text("You entered: \(password)")
        }
    }
}
```

How to create a Slider and read values from it

SwiftUI's **Slider** view works much like **UISlider**, although you need to bind it somewhere so you can store its value.

When you create it there are a variety of parameters you can provide, but the ones you probably care about most are:

- Value: What **Double** to bind it to.
- From and To: The range of the slider.
- By: How much to change the value when you move the slider.

For example, this code creates a slider bound to a **Celsius** property, then updates a text view as the slider moves so that it converts between Celsius and Fahrenheit:

```
struct ContentView : View {
    @State var celsius: Double = 0

    var body: some View {
        VStack {
            Slider(value: $celsius, from: -100, through: 100, by:
0.1)
            Text("\(celsius) Celsius is \(celsius * 9 / 5 + 32)
Fahrenheit")
        }
    }
}
```

How to create a picker and read values from it

SwiftUI's **Picker** view manages to combine **UIPicker** and **UITableView** in one, while also adapting to other styles on other operating systems. The great thing is that we really don't have to care how it works – SwiftUI does a good job of adapting itself automatically to its environment.

As with most other controls, you must attach your picker to some sort of state that will track the picker's selection. For example, this creates a **colors** array and an integer that stores which color was selected, then uses that with a picker and a text view so you can see values being read back:

```
struct ContentView : View {
    var colors = ["Red", "Green", "Blue", "Tartan"]
    @State private var selectedColor = 0

    var body: some View {
      VStack {
        Picker(selection: $selectedColor, label: Text("Please
choose a color")) {
          ForEach(0 ..< colors.count) {
             Text(self.colors[$0]).tag($0)
          }
        }
        Text("You selected: \(colors[selectedColor])")
      }
    }
}
```

Note: It's important that you place your picker view inside something to ensure it appears in the default wheel style on iOS.

How to create a date picker and read values from it

SwiftUI's **DatePicker** view is analogous to **UIDatePicker**, and comes with a variety options for controlling how it looks and works. Like all controls that store values, it does need to be bound to some sort of state in your app.

For example, this creates a date picker bound to a **birthDate** property, then displays the value of the date picker as it's set:

```
struct ContentView : View {
    var dateFormatter: DateFormatter {
        let formatter = DateFormatter()
        formatter.dateStyle = .long
        return formatter
    }

    @State var birthDate = Date()

    var body: some View {
        VStack {
            DatePicker(
                $birthDate,
                maximumDate: Date(),
                displayedComponents: .date
            )

            Text("Date is \(birthDate, formatter: dateFormatter)")
        }
    }
}
```

You can see I've set **displayedComponents** to **.date**, but you could also use **.hourAndMinute** to get time data instead.

I also used **maximumDate** to stop people specifying a birth date in the future, but you can specify **minimumDate** to stop people specifying a date earlier than a point of your choosing.

How to create a segmented control and read values from it

SwiftUI's **SegmentedControl** works similarly to **UISegmentedControl**, although it needs to be bound to some state and you must ensure to give each segment a tag so it can be identified. Segments can be text or pictures; anything else will silently fail.

As an example, this creates a segmented control that works with a **favoriteColor** state property, and adds a text view below that shows whichever value was selected:

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

    var body: some View {
        VStack {
            SegmentedControl(selection: $favoriteColor) {
                Text("Red").tag(0)
                Text("Green").tag(1)
                Text("Blue").tag(2)
            }
            Text("Value: \(favoriteColor)")
        }
    }
}
```

In this instance, though, it's better to create an array to store the various colors, then use **ForEach** to create the text view inside using a loop:

```
struct ContentView : View {
    @State private var favoriteColor = 0
    var colors = ["Red", "Green", "Blue"]
```

```
    var body: some View {
        VStack {
            SegmentedControl(selection: $favoriteColor) {
                ForEach(0..<colors.count) { index in
                    Text(self.colors[index]).tag(index)
                }
            }

            Text("Value: \(colors[favoriteColor])")
        }
    }
}
```

How to read tap and double-tap gestures

Any SwiftUI view can have tap actions attached, and you can specify how many taps should be received before the action is triggered.

For example, this creates a text view that will print a message when tapped:

```
Text("Tap me!")
    .tapAction {
        print("Tapped!")
    }
```

And this creates an image view that will print a message when double tapped:

```
Image("example-image")
    .tapAction(count: 2) {
        print("Double tapped!")
    }
```

How to add a gesture recognizer to a view

Any SwiftUI view can have gesture recognizers attached, and those gesture recognizers in turn can have closures attached that will be run when the recognizer activates.

There are several gesture recognizers to work with, and I'm going to provide you with code samples for several of them to help get you started – you'll see how similar they are.

First, **TapGesture**. When you create this you can specify how many taps it takes to trigger the gesture, then attach an **onEnded** closure that will be run when the gesture happens. For example, this creates an image that gets bigger every time it's tapped:

```
struct ContentView : View {
    @State private var scale: Length = 1.0

    var body: some View {
        Image("example-image")
            .scaleEffect(scale)

            .gesture(
                TapGesture()
                    .onEnded { _ in
                        self.scale += 0.1
                    }
            )
    }
}
```

Second, **LongPressGesture** recognizes when the user presses and holds on a view for at least a period of time you specify. So, this creates an image view that prints a message when its pressed for at least two seconds:

```
Image("example-image")
    .gesture(
        LongPressGesture(minimumDuration: 2)
            .onEnded { _ in
                print("Pressed!")
            }
    )
```

Finally, **DragGesture** triggers when the user presses down on a view and move at least a certain distance away. So, this creates an image with a drag gesture that triggers when they move it at least 50 points:

```
Image("example-image")
    .gesture(
        DragGesture(minimumDistance: 50)
            .onEnded { _ in
                print("Dragged!")
            }
    )
```

How to respond to view lifecycle events: appear and disappear

SwiftUI gives us equivalents to UIKit's **viewDidAppear()** and **viewDidDisappear()** in the form of **onAppear()** and **onDisappear()**. You can attach any code to these two events that you want, and SwiftUI will execute them when they occur.

Note: In the current beta **onAppear()** works great but **onDisappear()** doesn't seem to get called.

As an example, this creates two views that use **onAppear()** and **onDisappear()** to print messages, with a navigation button to move between the two:

```
struct ContentView : View {
    var body: some View {
        NavigationView {
            NavigationButton(destination: DetailView()) {
                Text("Hello World")
            }
        }.onAppear {
            print("ContentView appeared!")
        }.onDisappear {
            print("ContentView disappeared!")
        }
    }
}

struct DetailView : View {
    var body: some View {
        VStack {
            Text("Second View")
        }.onAppear {
```

```
            print("DetailView appeared!")
        }.onDisappear {
            print("DetailView disappeared!")
        }
    }
}
```

When that code runs you should be able to move between the two views and see messages printed in your Xcode debug console.

Chapter 5
Advanced state

Learn how to bind objects and query the environment

What's the difference between @ObjectBinding, @State, and @EnvironmentObject?

State is inevitable in any modern app, but with SwiftUI it's important to remember that all of our views are simply functions of their state – we don't change the views directly, but instead manipulate the state and let *that* dictate the result.

SwiftUI gives us several ways of storing state in our application, but they are subtly different and it's important to understand *how* they are different in order to use the framework properly.

In all the state examples we've so far we've use **@State** to create properties like this:

```
struct ContentView : View {
    @State var score = 0
    // more code
}
```

This creates a property inside a view, but it uses the **@State** property wrapper to ask SwiftUI to manage the memory. This *matters*: all our views are structs, which means they can't be changed, and if we couldn't add 1 to a score in a game then it isn't much of a game.

So, when we say **@State** to make a property, we hand control over it to SwiftUI so that it remains persistent in memory for as long as the view exists. When that state changes, SwiftUI knows to automatically reload the view with the latest changes so it can reflect its new information.

@State is great for simple properties that belong to a specific view and never get used outside that view, so as a result it's usually a good idea to mark those properties as being private, like this:

```
@State private var score = 0
```

This re-enforces the idea that such state is specifically designed never to escape its view.

What is @ObjectBinding?

For more complex properties – when you have a custom type you want to use that might have multiple properties and methods, or might be shared across multiple views – you should use **@ObjectBinding** instead.

This is very similar to **@State** except now we're using an external reference type rather than a simple local property like a string or an integer. You're still saying that your view depends on data that will change, except now it's data you're responsible for managing yourself – you need to create an instance of the class, create its own properties, and so on.

Whatever type you use with **@ObjectBinding** should conform to the **BindableObject** protocol, which has only one requirement: your type must implement some sort of **didChange** property that notifies the view when its data has changed.

This is what I mean when I say it's data you're responsible for managing yourself – when you set a property on your bound object you get to decide whether that should force the view to refresh or not. You usually *will*, but it's not required.

An bindable object can notify its view that important data has changed using publishers from the Combine framework. If the bindable object happens to have several views using its data, then it will automatically notify them all.

Warning: When you use a publisher to announce that your object has changed, this *must* happen on the main thread.

What is @EnvironmentObject?

You've seen how **@State** declares simple properties for a type that automatically cause a refresh of the view when it changes, and how **@ObjectBinding** declares a property for an external type that may or may not cause a refresh of the view when it changes. Both of these

two must be set by your view, but **@ObjectBinding** might be shared with other views.

There's a third type of property available to use, which is **@EnvironmentObject**. This is a value that is made available to your views through the application itself – it's shared data that every view can read if they want to. So, if your app had some important model data that all views needed to read, you could either hand it from view to view to view or just put it into the environment where every view has instant access to it.

Think of **@EnvironmentObject** as a massive convenience for times when you need to pass lots of data around your app. Because all views point to the same model, if one view changes the model all views immediately update – there's no risk of getting different parts of your app out of sync.

Summing up the differences

- Use **@State** for simple properties that belong to a single view. They should usually be marked **private**.
- Use **@ObjectBinding** for complex properties that might belong to several views. Any time you're using a reference type you should be using **@ObjectBinding** for it.
- Use **@EnvironmentObject** for properties that were created elsewhere in the app, such as shared data.

Of the three you will find that **@ObjectBinding** is both the most useful and the most commonly used, so if you're not sure which to use start there.

How to use BindableObject to create object bindings

When using object bindings there are two ever so slightly differently things we need to work with: the **BindableObject** protocol is used with some sort of class that can store data, and the **@ObjectBinding** property wrapper is used inside a view to store a bindable object instance.

As an example, here's a **UserSettings** class that conforms to **BindableObject**:

```
class UserSettings: BindableObject {
    var didChange = PassthroughSubject<Void, Never>()

    var score = 0 {
        didSet {
            didChange.send(())
        }
    }
}
```

That packs quite a lot into a small amount of space, so let me break it down.

First: **didChange** is an instance of **PassthroughSubject**. This comes from the Combine framework, you'll need to add **import Combine** to make your code compile. The job of a passthrough subject is simple: whenever we want to tell the world that our object has changed, we ask the passthrough subject to do it for us. It's called "pass through" because the value we hand it literally gets passed on to whatever views are watching for changes. **PassthroughSubject** is technically called a *publisher*, because it publishes announcements of changes to the world.

Second: **PassthroughSubject** is generic over two things: **Void** and **Never**. The first parameter, **Void**, means "I will send no value." In the case of SwiftUI we don't need a value to send because all we want is for the view to refresh – it will automatically pick up the new data from

its **@ObjectBinding** state. The second parameter, **Never**, means "I will never throw errors." If you wanted to, you could define a custom error type such as **NetworkError** and send that instead, but again it's usually find to handle errors locally inside your bindable object.

Third: we have a **didSet** property observer attached to the **age** property of **UserSettings** so that we can run code whenever that value changes. In our example code, we call **didChange.send(())** whenever **age** changes, which is what tells the **didChange** publisher to put out the news that our data has changed so that any subscribed views can refresh.

We can use that **UserSettings** class inside a view like this:

```
struct ContentView : View {
    @ObjectBinding var settings = UserSettings()

    var body: some View {
        VStack {
            Text("Your score is \(settings.score)")
            Button(action: {
                self.settings.score += 1
            }) {
                Text("Increase Score")
            }
        }
    }
}
```

As you can see, other than using the **@ObjectBinding** property wrapper with **settings**, everything else more or less looks the same – SwiftUI takes care of all the implementation details for us.

There is *one* important difference, though: the **settings** property isn't declared as private. This is because bound objects can be used by more than one view, so it's common to share it openly.

Warning: When you use a publisher to announce that your object has changed, this *must* happen on the main thread.

How to use EnvironmentObject to share data between views

For data that should be shared with all views in your entire app, SwiftUI gives us **@EnvironmentObject**. This lets us share model data anywhere it's needed, while also ensuring that our views automatically stay updated when that data changes.

Think of **@EnvironmentObject** as a smarter, simpler way of using **@ObjectBinding** on lots of views. Rather than creating some data in view A, then passing it to view B, then view C, then view D before finally using it, you can create it in view and put it into the environment so that views B, C, and D will automatically have access to it.

Note: Environment objects must be supplied by an ancestor view – if SwiftUI can't find an environment object of the correct type you'll get a crash. This applies for previews too, so be careful.

As an example, here's a bindable object that stores user settings:

```
class UserSettings: BindableObject {
    var didChange = PassthroughSubject<Void, Never>()

    var score = 0 {
        didSet {
            didChange.send(())
        }
    }
}
```

Yes, it only stores one value, but that's OK – what matters is that when the value changes the **PassthroughSubject** tells all views using it to refresh.

User settings are a sensible piece of data that we might want to share everywhere in our app, so that we no longer need to handle synchronizing it by hand.

So, when our app first launches we're going to create an instance of **UserSettings** so that shared instance is accessible everywhere in our app.

If you open SceneDelegate.swift you'll find these two lines of code inside the **scene(_:willConnectTo:options:)** method:

```
let window = UIWindow(frame: UIScreen.main.bounds)
window.rootViewController = UIHostingController(rootView:
ContentView())
```

It's that second line of code that creates our initial content view and presents it on the screen. This is where we need to pass in any environment objects that we've created, so that SwiftUI can make them available inside **ContentView** but also any other views it uses.

First, add this as a property of your **SceneDelegate**:

```
var settings = UserSettings()
```

That creates a settings instance once, and stores it safely. Now go back down to those two lines of code I showed you and change the second line so it passes our **settings** property into **ContentView** as an environment object, like this:

```
window.rootViewController = UIHostingController(rootView:
ContentView().environmentObject(settings)
```

Once that's done, the shared **UserSettings** instance is available to our content view and any other views it hosts or presents. All you need to do is create a property using the **@EnvironmentObject** property wrapper, like this:

```
@EnvironmentObject var settings: UserSettings
```

That doesn't need to be initialized with a default value, because it will automatically be read from the environment.

So, we could make a **ContentView** struct that increments our score setting, and even make it present a **DetailView** that shows the score setting, all without needing to create or pass around any local instances of **UserSettings** – it always just uses the environment.

Here's the code to make that happen:

```
struct ContentView : View {
    @EnvironmentObject var settings: UserSettings

    var body: some View {
        NavigationView {
            VStack {
                // A button that writes to the environment settings
                Button(action: {
                    self.settings.score += 1
                }) {
                    Text("Increase Score")
                }

                NavigationButton(destination: DetailView()) {
                    Text("Show Detail View")
                }
            }
        }
    }
}

struct DetailView: View {
    @EnvironmentObject var settings: UserSettings

    var body: some View {
        // A text view that reads from the environment settings
        Text("Score: \(settings.score)")
```

```
    }
}
```

So, once you've injected an object into the environment you can start using it immediately either in your top-level view or ten levels down – it doesn't matter. And most importantly, whenever any view changes the environment, all views relying on it are automatically refreshed so they stay in sync.

As you can see, we didn't need to explicitly associate the **UserSettings** instance in our scene delegate with the **settings** property in our two views – SwiftUI automatically figured out that it has a **UserSettings** instance in the environment, so that's the one that gets used.

Warning: Now that our views rely on an environment object being present, it's important that you also update your preview code to provide some example settings to use. For example, using something like **ContentView().environmentObject(UserSettings())** for your preview ought to do it.

Chapter 6

Lists

Create scrolling tables of data

Working with lists

SwiftUI's **List** view is similar to **UITableView** in that it can show static or dynamic table view cells based on your needs. However, it is significantly simpler to use: we don't need to create prototype cells in storyboards, or register them in code; we don't need to tell it how many rows there are; we don't need to dequeue and configure cells by hand, and more.

Instead, SwiftUI's lists are designed for composability – designed to be able to build bigger things from smaller things. So rather than having one large view controller that configures cells by hand, SwiftUI has us build small views that know how to configure themselves as list rows, then use those.

In terms of code size if nothing else, the difference is staggering – you can delete almost all your table view code and still get the same great look and feel you're used to.

How to create a list of static items

To create a static list of items you first need to define what each row in your list should look like. This is a view just like any other, so you might write one such as this:

```
struct RestaurantRow: View {
    var name: String

    var body: some View {
        Text("Restaurant: \(name)")
    }
}
```

Now that you've defined what each row looks like, you can create a **List** view that creates as many rows as you need, like this:

```
struct ContentView: View {
    var body: some View {
        List {
            RestaurantRow(name: "Joe's Original")
            RestaurantRow(name: "The Real Joe's Original")
            RestaurantRow(name: "Original Joe's")
        }
    }
}
```

When that code runs you'll see three rows in a table, just like you would have had with **UITableView** in UIKit.

You don't need to make each row use the same view type, so you can mix and match row views as you need.

How to create a list of dynamic items

In order to handle dynamic items, you must first tell SwiftUI how it can identify which item is which. This is done using the **Identifiable** protocol, which has only one requirement: some sort of **id** value that SwiftUI can use to see which item is which.

For example, you might create a **Restaurant** struct that says restaurants have an ID and name, with the ID being a random identifier just so that SwiftUI knows which is which:

```
struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
}
```

Next you would define what a list row looks like. In our case we're going to define a **RestaurantRow** view that stores one restaurant and prints its name in a text view:

```
struct RestaurantRow: View {
    var restaurant: Restaurant

    var body: some View {
        Text("Come and eat at \(restaurant.name)")
    }
}
```

Finally we can create a list view that shows them all. This means creating some example data, putting it into an array, then passing that into a list to be rendered:

```
struct ContentView: View {
    var body: some View {
        let first = Restaurant(name: "Joe's Original")
```

```
        let second = Restaurant(name: "The Real Joe's Original")
        let third = Restaurant(name: "Original Joe's")
        let restaurants = [first, second, third]

        return List(restaurants) { restaurant in
            RestaurantRow(restaurant: restaurant)
        }
    }
}
```

Most of that is just creating data – the last part is where the real action is:

```
return List(restaurants) { restaurant in
    RestaurantRow(restaurant: restaurant)
}
```

That creates a list from the **restaurants** array, executing the closure once for every item in the array. Each time the closure goes around the **restaurant** input will be filled with one item from the array, so we use that to create a **RestaurantRow**.

In fact, in trivial cases like this one we can make the code even shorter:

```
return List(restaurants, rowContent: RestaurantRow.init)
```

How to let users delete rows from a list

SwiftUI makes it easy to let users swipe to delete rows by attaching an **onDelete(perform:)** handler to some or all of your data. This handler needs to have a specific signature that accepts multiples indexes to delete, like this:

```
func delete(at offsets: IndexSet) {
```

Inside there you can either loop over every index in the set or just read the first one if that's the only one you want to handle. Because SwiftUI is watching your state, any changes you make will automatically be reflected in your UI.

Note: The WWDC demonstration for this feature used a non-existent method of Swift arrays called **remove(atOffsets:)** that does all this work for us – hopefully that will land in a future release.

For example, this code creates a **ContentView** struct with a list of three items, then attaches an **onDelete(perform:)** modifier that removes any item from the list:

```
struct ContentView : View {
    @State var users = ["Paul", "Taylor", "Adele"]

    var body: some View {
        NavigationView {
            List {
                ForEach(users.identified(by: \.self)) { user in
                    Text(user)
                }
                .onDelete(perform: delete)
            }
        }
```

```
    }

    func delete(at offsets: IndexSet) {
        if let first = offsets.first {
            users.remove(at: first)
        }
    }
}
```

If you run that code you'll find you can swipe to delete any row in the list.

How to let users move rows in a list

SwiftUI gives us simple hooks into lists to let us move rows around, although some of the functionality that was demonstrated at WWDC isn't actually available in the current beta so we need a workaround.

What *does* work is that we can attach an **onMove(perform:)** modifier to items in a list, and have it call a method of our choosing when a move operation happens. That method needs to accept a source **IndexSet** and a destination **Int**, like this:

```
func move(from source: IndexSet, to destination: Int) {
```

When moving several items it's always a good idea to move the later ones first so that you avoid moving other items and getting your indexes confused.

As an example, we could create a **ContentView** struct that sets up an array of three username strings, and asks SwiftUI to move them around calling a **move()** method. In order to activate moving – i.e., to make the drag handles appear – it also adds an edit button to the navigation view so the user can toggle editing mode.

Here's the code:

```
struct ContentView : View {
    @State var users = ["Paul", "Taylor", "Adele"]

    var body: some View {
        NavigationView {
            List {
                ForEach(users.identified(by: \.self)) { user in
                    Text(user)
                }
                .onMove(perform: move)
```

```
            }
            .navigationBarItems(trailing: EditButton())
        }
    }

    func move(from source: IndexSet, to destination: Int) {
        // sort the indexes low to high
        let reversedSource = source.sorted()

        // then loop from the back to avoid reordering problems
        for index in reversedSource.reversed() {
            // for each item, remove it and insert it at the
destination
            users.insert(users.remove(at: index), at: destination)
        }
    }
}
```

In the WWDC session demos their **move()** method was just one line of code because it used an extension on Swift's arrays that isn't available to us – hopefully it will arrive soon!

How to enable editing on a list using EditButton

If you have configured a SwiftUI list view to support deletion or editing of its items, you can allow the user to toggle editing mode for your list view by adding an **EditButton** somewhere.

For example, this **ContentView** struct defines an array of users, attaches an **onDelete()** method, then adds an edit button to the navigation bar:

```
struct ContentView : View {
    @State var users = ["Paul", "Taylor", "Adele"]

    var body: some View {
        NavigationView {
            List {
                ForEach(users.identified(by: \.self)) { user in
                    Text(user)
                }
                .onDelete(perform: delete)
            }
            .navigationBarItems(trailing: EditButton())
        }
    }

    func delete(at offsets: IndexSet) {
        if let first = offsets.first {
            users.remove(at: first)
        }
    }
}
```

When that is run, you'll find you can tap the edit button to enable or disable editing mode for

the items in the list.

How to add sections to a list

SwiftUI's list view has built-in support for sections and section headers, just like **UITableView** in UIKit. To add a section around some cells, start by placing a **Section** around it, optionally also adding a header and footer.

As an example, here's a row that holds task data for a reminders app:

```
struct TaskRow: View {
    var body: some View {
        Text("Task data goes here")
    }
}
```

What we want to do is create a list view that has two sections: one for important tasks and one for less important tasks. Here's how that looks:

```
struct ContentView : View {
    var body: some View {
        List {
            Section(header: Text("Important tasks")) {
                TaskRow()
                TaskRow()
                TaskRow()
            }

            Section(header: Text("Other tasks")) {
                TaskRow()
                TaskRow()
                TaskRow()
            }
        }
    }
```

```
}
```

You can also add footer text to sections, like this:

```
Section(header: Text("Other tasks"), footer: Text("End")) {
    TaskRow()
    TaskRow()
    TaskRow()
}
```

How to make a grouped list

SwiftUI's list supports grouped or plain styles, just like **UITableView**. The default is plain style, but if you want to change to grouped you should use the **.listStyle(.grouped)** modifier on your list.

For example, this defines an example row and places it inside a grouped list:

```
struct ExampleRow: View {
    var body: some View {
        Text("Example Row")
    }
}

struct ContentView : View {
    var body: some View {
        List {
            Section(header: Text("Examples")) {
                ExampleRow()
                ExampleRow()
                ExampleRow()
            }
        }.listStyle(.grouped)
    }
}
```

Working with implicit stacking

What happens if you create a dynamic list and put more than one thing in each row? SwiftUI's solution is simple, flexible, and gives us great behavior by default: it creates an implicit **HStack** to hold your items, so they automatically get laid out horizontally.

For example, if we wanted to make a row where we had a small picture on the left and the remaining space be allocated to a text field, we'd start with a struct to hold our data like this:

```
struct User: Identifiable {
    var id = UUID()
    var username = "Anonymous"
}
```

I've given both of those default values to make our example easier.

Once we have that, we could create an array of three users, and show them in a dynamic list, like this:

```
struct ContentView : View {
    let users = [User(), User(), User()]

    var body: some View {
        List(users) { user in
            Image("paul-hudson")
                .resizable()
                .frame(width: 40, height: 40)
            Text(user.username)
        }
    }
}
```

Chapter 7
Containers

Place your views inside a navigation controller

Working with containers

SwiftUI is designed to be composed right out of the box, which means you can place one view inside another as much as you need.

This is particularly useful when working with the major container views we're used to, such as navigation controllers and tab bar controllers. We can place any views we want right into another container view, and SwiftUI will adapt its layout automatically.

In this regard, SwiftUI's own containers – **NavigationView**, **TabbedView**, **Group**, and more – are no different from containers we make with our own view composition.

How to embed a view in a navigation view

SwiftUI's **NavigationView** maps more or less to UIKit's **UINavigationController** in that it presents content, it's able to handle navigation between views, and it places a navigation bar at the top of the screen.

In its simplest form you can place a text view into a navigation view like this:

```
NavigationView {
    Text("This is a great app")
}
```

However, that leaves the navigation bar at the top empty. So, you will usually use the **navigationBarTitle()** modifier on whatever you're embedding, so you can add a title at the top of your screen, like this:

```
NavigationView {
    Text("SwiftUI")
        .navigationBarTitle(Text("Welcome"))
}
```

The **navigationBarTitle()** modifier gives us some customization options. For example, by default it will inherit large title display mode from whatever view presented it, or if it's the initial view then it will use large titles. But if you'd prefer to force enable or disable large titles you should use the inline parameter like this:

```
.navigationBarTitle(Text("Welcome"), displayMode: .inline)
```

That will make small navigation titles, but you can also use **.large** to force a large title.

How to add bar items to a navigation view

The **navigationBarItems()** modifier lets us add bar button items to the leading and trailing edge of a navigation view. These might be a tappable button, but there are no restrictions – you can add any sort of view.

For example, this adds a Help button the to trailing edge of a navigation view:

```
var body: some View {
    NavigationView {
        Text("SwiftUI")
            .navigationBarTitle(Text("Welcome"))
            .navigationBarItems(trailing:
                Button(action: {
                    print("Help tapped!")
                }) {
                    Text("Help")
                })
    }
}
```

How to group views together

If you need several views to act as one – for example, to transition together – then you should use SwiftUI's **Group** view. This is particularly important because, for underlying technical reasons, you can only add up to 10 views to a parent view at a time.

To demonstrate this, here's a **VStack** with 10 pieces of text:

```
VStack {
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
    Text("Line")
}
```

That works just fine, but if you try adding an eleventh piece of text, you'll get an error like this one:

```
ambiguous reference to member 'buildBlock()'
```

…followed by a long list of errors like this:

```
SwiftUI.ViewBuilder:3:24: note: found this candidate
    public static func buildBlock<C0, C1, C2, C3, C4, C5, C6,
C7, C8, C9>(_ c0: C0, _ c1: C1, _ c2: C2, _ c3: C3, _ c4: C4, _
c5: C5, _ c6: C6, _ c7: C7, _ c8: C8, _ c9: C9) ->
TupleView<(C0, C1, C2, C3, C4, C5, C6, C7, C8, C9)> where C0 :
```

```
View, C1 : View, C2 : View, C3 : View, C4 : View, C5 : View,
C6 : View, C7 : View, C8 : View, C9 : View
```

This is because SwiftUI's view building system has various code designed to let us add 1 view, 2 views, 3 views, or 4, 5, 6, 7, 8, 9, and 10 views, but not for 11 and beyond – that doesn't work.

Fortunately, we can use a group like this:

```
var body: some View {
    VStack {
        Group {
            Text("Line")
            Text("Line")
            Text("Line")
            Text("Line")
            Text("Line")
            Text("Line")
        }

        Group {
            Text("Line")
            Text("Line")
            Text("Line")
            Text("Line")
            Text("Line")
        }
    }
}
```

That creates exactly the same result, except now we can go beyond the 10 view limit because the **VStack** contains only two views – two groups.

Chapter 8

Alerts and action sheets

Show modal notifications when something happens

Working with presentations

SwiftUI's declarative approach to programming means that we don't create and present alert and action sheets in the same way as we did in UIKit. Instead, we define the conditions in which they should be shown, tell it what they should look like, then leave it to figure the rest out for itself.

This is all accomplished using a **presentation()** modifier, which attaches new UI to our view that will be shown when a condition is satisfied. You can attach as many as you want, and they effectively lie in wait watching until their condition becomes true, at which point they show their UI. For example, you might toggle a Boolean inside a button press, which triggers an alert to show.

You can attach presentations to your main view or any of its children – even to the button that adjusts your state so the presentation triggers. It's a subtle distinction, but it's important to understand that these presentations aren't attached to the button because it's a button – i.e., that doesn't in any way make the alert be shown because the button was tapped. Instead, we're attaching it to our view hierarchy so that SwiftUI is aware that it might be shown at any point.

How to show an alert

The code to create a basic SwiftUI alert looks like this:

```
Alert(title: Text("Important message"), message: Text("Wear
sunscreen"), dismissButton: .default(Text("Got it!")))
```

That defines a title and message, like you'd see in a **UIAlertController**, then adds a dismiss button with a default style and the text "Got it!".

To *show* that alert you need to define some sort of bindable condition that determines whether the alert should be visible or not. You then attach that to your main view as a *presentation*, which presents the alert as soon as its condition becomes true.

For example, this code creates a **showingAlert** Boolean that tracks whether the sunscreen message should be shown or not, sets that Boolean to true when a button is tapped, then creates and attaches an alert view using that Boolean so it appears when the button is tapped:

```
struct ContentView : View {
    @State var showingAlert = false

    var body: some View {
        Button(action: {
            self.showingAlert = true
        }) {
            Text("Show Alert")
        }
        .presentation($showingAlert) {
            Alert(title: Text("Important message"), message:
Text("Wear sunscreen"), dismissButton: .default(Text("Got
it!")))
        }
    }
```

```
}
```

Tip: Presenting an alert like this will automatically set **showingAlert** back to false when the dismiss button is tapped.

How to add actions to alert buttons

Basic SwiftUI alerts look like this:

```
Alert(title: Text("Important message"), message: Text("Wear sunscreen"), dismissButton: .default(Text("Got it!")))
```

However, you will often want to attach actions to buttons to perform specific actions when they are tapped. To do that, attach a closure to your button that will be called when it's tapped, like this:

```
struct ContentView : View {
    @State var showingAlert = false

    var body: some View {
        Button(action: {
            self.showingAlert = true
        }) {
            Text("Show Alert")
        }
            .presentation($showingAlert) {
                Alert(title: Text("Are you sure you want to delete this?"), message: Text("There is no undo"),
primaryButton: .destructive(Text("Delete")) {
                        print("Deleting...")
                }, secondaryButton: .cancel())
            }
    }
}
```

How to show an action sheet

SwiftUI gives us the **ActionSheet** view for creating action sheets for the user to choose from. However, you do need to make sure you reset your state when it's dismissed, otherwise you won't be able to show it again.

I'll show you all the code in a moment, but I want to break it down first because it's not easy.

First, you need to define a property that will track whether to show the action sheet or not:

```
@State var showingSheet = false
```

Next, you should create a property to store your action sheet. This needs title and message text, but should also provide an array of buttons. However – and this is the important part – it should also add a trigger to reset the **showingSheet** property when the action sheet is dismissed, like this:

```
var sheet: ActionSheet {
   ActionSheet(title: Text("Action"), message: Text("Quote
mark"), buttons: [.default(Text("Show Sheet"), onTrigger: {
      self.showingSheet = false
   })])
}
```

Once that's done, you can attach the action sheet to your view using a presentation that either shows the sheet or does nothing based on the value of **showingSheet**, like this:

```
.presentation(showingSheet ? sheet : nil)
```

We can put all that together into an example view that triggers the action sheet when a button is tapped:

```
struct ContentView : View {
   @State var showingSheet = false
```

```
    var sheet: ActionSheet {
        ActionSheet(title: Text("Action"), message: Text("Quote
mark"), buttons: [.default(Text("Woo"), onTrigger: {
            self.showingSheet = false
        })])
    }

    var body: some View {
        Button(action: {
            self.showingSheet = true
        }) {
            Text("Woo")
        }
            .presentation(showingSheet ? sheet : nil)
    }
}
```

Chapter 9
Presenting views

Move your user from one view to another

How to push a new view using NavigationButton

If you have a navigation view and you want to push a new view onto the navigation stack, you should use **NavigationButton**. This take a destination as its first parameter and what to show inside the button as its second parameter (or as a trailing closure), and takes care of pushing the new view on the stack for us along with animation.

For example, if you had a detail view like this:

```
struct DetailView: View {
    var body: some View {
        Text("Detail")
    }
}
```

Then you could present it like this:

```
struct ContentView : View {
    var body: some View {
        NavigationView {
            NavigationButton(destination: DetailView()) {
                Text("Click")
            }.navigationBarTitle(Text("Navigation"))
        }
    }
}
```

How to push a new view when a list row is tapped

SwiftUI doesn't have a direct equivalent of the **didSelectRowAt** method of **UITableView**, but it doesn't need one because we can combine **NavigationButton** with a list row and get the behavior for free.

We need to put together a list with some content we can work with. First, we need some sort of data to show:

```
struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
}
```

And we need a list row view that shows one restaurant at a time:

```
struct RestaurantRow: View {
    var restaurant: Restaurant

    var body: some View {
        Text(restaurant.name)
    }
}
```

Finally, we need a view that hosts a list of available restaurants:

```
struct ContentView: View {
    var body: some View {
        let first = Restaurant(name: "Joe's Original")
        let restaurants = [first]

        return NavigationView {
```

```
            List(restaurants) { restaurant in
                RestaurantRow(restaurant: restaurant)
            }.navigationBarTitle(Text("Select a restaurant"))
        }
    }
}
```

That code shows one restaurant in a list, but it isn't selectable.

In order to make tapping a row show a detail view, we first need a detail view that can show a restaurant. For example, something like this:

```
struct RestaurantView: View {
    var restaurant: Restaurant

    var body: some View {
        Text("Come and eat at \(restaurant.name)")
            .font(.largeTitle)
    }
}
```

And with *that* in place we can now wrap our **RestaurantRow** rows in a **NavigationButton**, like this:

```
return NavigationView {
    List(restaurants) { restaurant in
        NavigationButton(destination: RestaurantView(restaurant:
restaurant)) {
            RestaurantRow(restaurant: restaurant)
        }
    }.navigationBarTitle(Text("Select a restaurant"))
}
```

As you can see, that uses **RestaurantView(restaurant: restaurant)** as the destination for the row tap event, so that will create the **RestaurantView** and pass in the restaurant that was attached to the list row in question.

Notice how we have literally put a list row inside a navigation button – SwiftUI makes it work thanks to its remarkable composition abilities.

How to present a new view using PresentationButton

SwiftUI's **PresentationButton** is used to present new view controllers modally over existing ones, like calling **present()** on a **UIViewController**. To use one, give it something to show (some text, an image, etc) plus a destination, and let SwiftUI handle the rest.

For example, if you had a detail view like this one:

```
struct DetailView: View {
    var body: some View {
        Text("Detail")
    }
}
```

Then you could present it like this:

```
struct ContentView : View {
    var body: some View {
        PresentationButton(Text("Click to show"), destination:
DetailView())
    }
}
```

Unlike navigation buttons, presentation buttons don't require a navigation view to work.

Chapter 10

Transforming views

Clip, size, scale, spin, and more

How to give a view a custom frame

By default views take up only as much space as they need, but if you want that to change you can use a **frame()** modifier to tell SwiftUI what kind of size range you want to have.

For example, you could create a button with a 200x200 tappable area like this:

```
Button(action: {
    print("Button tapped")
}) {
    Text("Welcome")
        .frame(minWidth: 0, maxWidth: 200, minHeight: 0,
maxHeight: 200)
        .font(.largeTitle)
}
```

Or you could make a text view fill the whole screen by specifying a frame with zero for its minimum width and height, and infinity for its maximum width and height, like this:

```
Text("Please log in")
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
maxHeight: .infinity)
    .font(.largeTitle)
    .foregroundColor(.white)
    .background(Color.red)
```

How to adjust the position of a view

All views have a natural position inside your hierarchy, but the **offset()** modifier lets you move them relative to that natural position.

Important: Using **offset()** will cause a view to be moved relative to its natural position, but *won't* affect the position of other views. This means you can make one view overlap another when normally it would have been positioned next to it, which may not be what you want.

For example, in this **VStack** we can use **offset()** to move the second item down by 15 points so that it begins to overlap the third item:

```
VStack {
    Text("Home")
    Text("Options")
        .offset(y: 15)
    Text("Help")
}
```

You will commonly find that using **padding()** together with **offset()** gives you the result you're looking for, as that moves one view around while also adjusting the views next to it to match.

For example, this will move the second item down by 15 points, but add 15 points of padding to its bottom edge so that it doesn't overlap the text view below:

```
VStack {
    Text("Home")
    Text("Options")
        .offset(y: 15)
        .padding(.bottom, 15)
    Text("Help")
```

```
}
```

How to color the padding around a view

The **padding()** modifier lets us add some space around a view, and the **background()** modifier lets us set a background color. However, the way you use them matters, so it's important to be clear your goal in order to get the best results.

As an example, this creates a text view with a black background and white foreground, then adds system default padding to it:

```
Text("Hacking with Swift")
    .background(Color.black)
    .foregroundColor(.white)
    .padding()
```

And this adds system default padding then sets a red background color and a white foreground:

```
Text("Hacking with Swift")
    .padding()
    .background(Color.black)
    .foregroundColor(.white)
```

Those two pieces of code might look similar, but they yield different results because the order in which you apply modifiers matters. In the second example the view is padded *then* colored, which means the padding also gets colored black. In contrast, the first example colors then pads, so the padding remains uncolored.

So, if you want some text to have a background color wider than the text itself, make sure use the second code example – pad *then* color.

How to stack modifiers to create more advanced effects

Each modifier you add to a view adjusts whatever came before it, and you're able to repeat modifiers more than once.

For example, we could add padding and a background color around a text view, then add some more padding and a different background color, then add some *more* padding and a *third* background color, all to make a particular effect:

```
Text("Forecast: Sun")
    .font(.largeTitle)
    .foregroundColor(.white)
    .padding()
    .background(Color.red)
    .padding()
    .background(Color.orange)
    .padding()
    .background(Color.yellow)
```

How to draw a border around a view

SwiftUI gives us a dedicated **border()** modifier to draw borders around views. It has a few variations depending on whether you want to specify a stroke width or a corner radius, so here are a few examples:

This adds a simple 1-point black border around a text view:

```
Text("Hacking with Swift")
    .border(Color.black)
```

If you want to make the border so that it doesn't sit right on the edges of your view, add some padding first:

```
Text("Hacking with Swift")
    .padding()
    .border(Color.black)
```

This adds a 4-point red border:

```
Text("Hacking with Swift")
    .padding()
    .border(Color.red, width: 4)
```

And this adds a 4-point blue border with 16-point rounded corners:

```
Text("Hacking with Swift")
    .padding()
    .border(Color.red, width: 4, cornerRadius: 16)
```

How to draw a shadow around a view

SwiftUI gives us a dedicated **shadow()** modifier to draw shadows around views. You can control the color, radius, and position of the shadow, and you can also control which parts of the view get shadowed by adjusting your modifier order.

In its basic form, you can add a shadow just by specifying the radius of the blur, like this:

```
Text("Hacking with Swift")
    .padding()
    .shadow(radius: 5)
    .border(Color.red, width: 4)
```

That adds a very slight shadow with a 5-point blur centered on the text.

You can also specify which color you want along with the X and Y offset from the original view. For example, this creates a strong red shadow with a 5-point blur, centered on the text:

```
Text("Hacking with Swift")
    .padding()
    .shadow(color: .red, radius: 5)
    .border(Color.red, width: 4)
```

If you want to specify offsets for the shadow, add **x** and/or **y** parameters to the modifier, like this:

```
Text("Hacking with Swift")
    .padding()
    .shadow(color: .red, radius: 5, x: 20, y: 20)
    .border(Color.red, width: 4)
```

Remember, SwiftUI applies modifiers in the order you list them, so if you want you can have

your shadow apply to the border as well just by putting the border modifier before the shadow modifier:

```
Text("Hacking with Swift")
    .padding()
    .border(Color.red, width: 4)
    .shadow(color: .red, radius: 5, x: 20, y: 20)
```

How to clip a view so only part is visible

SwiftUI lets you clip any view to control its shape, all by using the **clipShape()** modifier.

For example this creates a button using the system image "bolt.fill" (a filled lightning bolt), gives it some padding and a background color, then clips it using a circle so that we get a circular button:

```
Button(action: {
    print("Button tapped")
}) {
    Image(systemName: "bolt.fill")
        .foregroundColor(.white)
        .padding()
        .background(Color.green)
        .clipShape(Circle())
}
```

The **Circle** clip shape will always make circles from views, even if their width and height are unequal – it will just crop the larger value to match the small.

As well as **Circle** there's also **Capsule**, which crops a view to have rounded corners in a lozenge shape. For example, this creates the same button using a capsule shape:

```
Button(action: {
    print("Button tapped")
}) {
    Image(systemName: "bolt.fill")
        .foregroundColor(.white)
        .padding(EdgeInsets(top: 10, leading: 20, bottom: 10,
trailing: 20))
        .background(Color.green)
```

```
        .clipShape(Capsule())
}
```

How to rotate a view

SwiftUI's **rotationEffect()** modifier lets us rotate views freely, using either degrees or radians.

For example, if you wanted to rotate some text by -90 degrees so that it reads upwards, you would use this:

```
Text("Up we go")
    .rotationEffect(.degrees(-90))
```

If you prefer using radians, just pass in **.radians()** as your parameter, like this:

```
Text("Up we go")
    .rotationEffect(.radians(.pi))
```

View rotation is so fast that it's effectively free, so you could even make it interactive using a slider if you wanted:

```
struct ContentView: View {
    @State var rotation: Double = 0

    var body: some View {
        VStack {
            Slider(value: $rotation, from: 0.0, through: 360.0,
by: 1.0)
            Text("Up we go")
                .rotationEffect(.degrees(rotation))
        }
    }
}
```

By default views rotate around their center, but if you want to pin the rotation from a particular point you can add an extra parameter for that. For example if you wanted to make the slider

above pivoting the rotation around the view's top-left corner you'd write this:

```
struct ContentView: View {
    @State var rotation: Double = 0

    var body: some View {
        VStack {
            Slider(value: $rotation, from: 0.0, through: 360.0,
by: 1.0)
            Text("Up we go")
                .rotationEffect(.degrees(rotation), anchor:
UnitPoint(x: 0, y: 0))
        }
    }
}
```

How to rotate a view in 3D

SwiftUI's **rotation3DEffect()** modifier lets us rotate views in 3D space to create beautiful effects in almost no code.

This modifier accepts two parameters: what angle to rotate (in degrees or radians), plus a tuple containing the X, Y, and Z axis around which to perform the rotation.

Important: If you've never done 3D rotation before you should think about the X/Y/Z axes as being skewers through your views. The X axis goes horizontally, so if you rotate on the X axis it's like putting a horizontal skewer through your view – any rotation makes the top or bottom nearer or further, but won't adjust the leading and trailing edges.

So, if you wanted to rotate some text by 45 degrees around the X axis (which would cause the top of the text to look further away than the bottom), you might write this:

```
Text("EPISODE LLVM")
    .font(.largeTitle)
    .foregroundColor(.yellow)
    .rotation3DEffect(.degrees(45), axis: (x: 1, y: 0, z: 0))
```

Yes, you can make your own Star Wars crawl in SwiftUI.

How to scale a view up or down

SwiftUI's **scaleEffect()** modifier lets us increase or decrease the size of a view freely.

For example, we could make a text view five times its regular size like this:

```
Text("Up we go")
    .scaleEffect(5)
```

You can scale the X and Y dimensions independently if you want, allowing you to squash views like this:

```
Text("Up we go")
    .scaleEffect(x: 1, y: 5)
```

If you want more control, you can specify an anchor for your scaling like this:

```
Text("Up we go")
    .scaleEffect(2, anchor: UnitPoint(x: 1, y: 1))
```

That makes the text view twice its regular size, scaled from the bottom-right corner.

How to round the corners of a view

Any SwiftUI view can have its corners rounded using the **cornerRadius()** modifier. This takes a simple value in points that controls how pronounced the rounding should be.

So, you can create a text view with 25-point rounded corners like this:

```
Text("Round Me")
    .padding()
    .background(Color.red)
    .cornerRadius(25)
```

How to adjust the opacity of a view

Any SwiftUI view can be partially or wholly transparent using the **opacity()** modifier. This accepts a value between 0 (completely invisible) and 1 (fully opaque), just like the **alpha** property of **UIView** in UIKit.

For example, this creates a text view with a red background, then gives it 30% opacity:

```
Text("Now you see me")
    .padding()
    .background(Color.red)
    .opacity(0.3)
```

How to adjust the accent color of a view

iOS uses tint colors to give apps a coordinated theme, and the same functionality is available in SwiftUI under the name *accent colors*. Just like in UIKit, when you set the accent color of one view it affects all those inside it, so if you set the access color of your top-level control then everything gets colored.

For example, this creates a button inside a **VStack**, then give it an orange accent color:

```
VStack {
    Button(action: {}) {
        Text("Tap here")
    }
}.accentColor(Color.orange)
```

How to mask one view with another

SwiftUI gives us the **mask()** modifier for masking one with another, which means you can mask an image using text or an image using an image, or more.

For example, this creates a 300x300 image of stripes, then masks it using the text “SWIFT!” so that the letters act as a cut out for the image:

```
Image("stripes")
    .resizable()
    .frame(width: 300, height: 300)
    .mask(Text("SWIFT!")
        .font(Font.system(size: 72).weight(.black)))
```

How to blur a view

The **blur()** modifier lets us apply a real-time Gaussian blur to our views, at a strength of our choosing.

For example, this creates a 200x200 profile picture, then adds a 20-point Gaussian blur:

```
Image("paul-hudson")
    .resizable()
    .frame(width: 300, height: 300)
    .blur(radius: 20)
```

You can blur anything you want, including text views:

```
Text("Welcome to my SwiftUI app")
    .blur(radius: 2)
```

How to blend views together

When placing one view over another, you can control the way they overlap by using the **blendMode()** modifier. This contains a variety of ways you can mix colors together, such as using their difference or using a color burn – these will be familiar if you've used Core Graphics or something like Photoshop before.

To demonstrate this we could create a **ZStack** with two images inside, where the second has a **.multiply** blend mode so that it darkens the colors behind it:

```
ZStack {
    Image("paul-hudson")
    Image("example-image")
        .blendMode(.multiply)
}
```

How to adjust views by tinting, and desaturating, and more

SwiftUI lets us finely control the way views look by adjusting their brightness, tint, hue, saturation, and much more, all by using various modifiers.

For example, this creates an image view and tints the whole thing red:

```
Image("paul-hudson")
    .colorMultiply(.red)
```

You can adjust the saturation of views to any amount, where 0.0 is fully gray and 1.0 is its original color:

```
Image("paul-hudson")
    .saturation(0.5)
```

You can even dynamically adjust the contrast of a view by using the **contrast()** modifier. A value of 0.0 yields no contrast (a flat gray image), 1.0 gives you the original image, and everything above 1.0 *adds* contrast.

So, this will reduce the image contrast to 50%:

```
Image("paul-hudson")
    .contrast(0.5)
```

Chapter 11
Animation

Bring your views to life with movement

How to create a basic animation

SwiftUI has built-in support for animations with its **animation()** modifier. To use this modifier, place it after any other modifiers for your views, and tell it what kind of animation you want.

For example, this code creates a button that increases its scale effect by 1 each time it's pressed:

```
struct ContentView: View {
    @State var scale: Length = 1

    var body: some View {
        Button(action: {
            self.scale += 1
        }) {
            Text("Tap here")
                .scaleEffect(scale)
                .animation(.basic())
        }
    }
}
```

You can specify a precise duration for the animation if you want. For example, this animates the scale effect over three seconds:

```
Text("Tap here")
    .scaleEffect(scale)
    .animation(.basic(duration: 3))
```

You can also specify a curve, choosing between **.easeIn**, **.easeOut**, **.easeInOut**, and **.custom**, where the latter lets you specify your own control points.

For example, this animates the scale effect so that it starts slow and gets faster:

```
Text("Tap here")
    .scaleEffect(scale)
    .animation(.basic(curve: .easeIn))
```

You can animate many other modifiers, such as 2D and 3D rotation, opacity, border, and more. For example, this makes a button that spins around and increases its border every time it's tapped:

```
struct ContentView: View {
    @State var angle: Double = 0
    @State var borderThickness: Length = 1

    var body: some View {
        Button(action: {
            self.angle += 45
            self.borderThickness += 1
        }) {
            Text("Tap here")
                .padding()
                .border(Color.red, width: borderThickness)
                .rotationEffect(.degrees(angle))
                .animation(.basic())
        }
    }
}
```

How to create a spring animation

SwiftUI has built-in support for spring animations, which are animations that move to their target point, overshoot a little, then bounce back.

If you just use **.spring()** by itself, with no parameters, you get a sensible default. So, this creates a spring animation that rotates a button by 45 degrees every time it's tapped:

```
struct ContentView: View {
    @State var angle: Double = 0

    var body: some View {
        Button(action: {
            self.angle += 45
        }) {
            Text("Tap here")
                .padding()
                .rotationEffect(.degrees(angle))
                .animation(.spring())
        }
    }
}
```

If you want fine-grained control over the spring animation, send in any of the parameters that interest you: the mass of the object, how stiff the spring should be, how quickly the springiness slows down, and how fast it starts moving at launch.

For example, this creates a button with very low spring damping, which means it will bounce around for a long time before reaching its target angle:

```
Button(action: {
    self.angle += 45
}) {
```

```
    Text("Tap here")
        .padding()
        .rotationEffect(.degrees(angle))
        .animation(.spring(mass: 1, stiffness: 1, damping: 0.1,
initialVelocity: 10))
}
```

How to create an explicit animation

If you attach an animation modifier to a view, you end up with implicit animation – changing some state elsewhere in your view might use animation, even though you're just incrementing an integer or toggling a Boolean.

An alternative is to use *explicit* animation, where you don't attach modifiers to the view in question but instead ask SwiftUI to animate the precise change you want to make. To do this, wrap your changes in a call to **withAnimation()**.

For example, this uses explicit animation to make a button fade away slightly more each time it's tapped:

```
struct ContentView: View {
    @State var opacity: Double = 1

    var body: some View {
        Button(action: {
            withAnimation {
                self.opacity -= 0.2
            }
        }) {
            Text("Tap here")
                .padding()
                .opacity(opacity)
        }
    }
}
```

withAnimation() takes a parameter specifying the kind of animation you want, so you could create a three-second basic animation like this:

```
withAnimation(.basic(duration: 3)) {
    self.opacity -= 0.2
}
```

Explicit animations are often helpful because they cause every affected view to animation, not just those that have implicit animations attached. For example, if view A has to make room for view B as part of the animation, but only view B has an animation attached, then view A will jump to its new position without animating unless you use explicit animations.

How to add and remove views with a transition

You can include or exclude a view in your design just by using a regular Swift condition. For example, this adds or removes some details text when a button is tapped:

```
struct ContentView: View {
    @State var showDetails = false

    var body: some View {
        VStack {
            Button(action: {
                withAnimation {
                    self.showDetails.toggle()
                }
            }) {
                Text("Tap to show details")
            }

            if showDetails {
                Text("Details go here.")
            }
        }
    }
}
```

By default, SwiftUI uses a fade animation to insert or remove views, but you can change that if you want by attaching a **transition()** modifier to a view.

For example, we could make the details text view slide in or out from the bottom, like this:

```
Text("Details go here.")
```

```
    .transition(.move(edge: .bottom))
```

There's also the **.slide** transition, which causes a view to be animated in from its leading out and animated out on its trailing edge:

```
Text("Details go here.")
    .transition(.slide)
```

And the **.scale** transition, which causes a view to be scaled up from nothing to full size when coming in, then back down to nothing when going out:

```
Text("Details go here.")
    .transition(.scale())
```

How to combine transitions

When adding or removing a view, SwiftUI lets you combine transitions to make new animation styles using the **combined(with:)** method. For example, you can make a view move (one transition) and fade (a second transition) at the same time like this:

```
Text("Details go
here.").transition(AnyTransition.opacity.combined(with: .slide)
)
```

To make combined transitions easier to use and re-use, you can create them as extensions on **AnyTransition**, like this:

```
extension AnyTransition {
    static var moveAndScale: AnyTransition {

AnyTransition.move(edge: .bottom).combined(with: .scale())
    }
}
```

With that in place you can now make a text view be added or removed using just this:

```
Text("Details go here.").transition(.moveAndScale)
```

How to create asymmetric transitions

SwiftUI lets us specify one transition when adding a view and another when removing it, all done using the **asymmetric()** transition type.

For example, we can create a text view that uses asymmetric transitions so that it moves in from the leading edge when added and moves down to the bottom edge when being removed, like this:

```
Text("Details go
here.").transition(.asymmetric(insertion: .move(edge: .leading)
, removal: .move(edge: .bottom)))
```

Chapter 12
Composing views

Make your UI structure easier to understand

How to create and compose custom views

One of the core tenets of SwiftUI is composition, which means it's designed for us to create many small views then combine them together to create something bigger. This allows us to re-use views on a massive scale, which means less work for us. Even better, combining small subviews has virtually no runtime overhead, so we can use them freely.

The key is to start small and work your way up. For example, many apps have to work with users that look something like this:

```
struct User {
    var name: String
    var jobTitle: String
    var emailAddress: String
    var profilePicture: String
}
```

If you want to have a consistent design for user profile pictures in your app, you might create a 100x100 image view with a circular shape:

```
struct ProfilePicture: View {
    var imageName: String

    var body: some View {
        Image(imageName)
            .resizable()
            .frame(width: 100, height: 100)
            .clipShape(Circle())
    }
}
```

Your designer might tell you that whenever an email address is visible you should show a little envelope icon next to it as a visual hint, so you could make an **EmailAddress** view:

```
struct EmailAddress: View {
    var address: String

    var body: some View {
        HStack {
            Image(systemName: "envelope")
            Text(address)
        }
    }
}
```

When it comes to showing a user's details, you could create a view that has their name and job title formatted neatly, backed up by their email address using your **EmailAddress** view, like this:

```
struct UserDetails: View {
    var user: User

    var body: some View {
        VStack(alignment: .leading) {
            Text(user.name)
                .font(.largeTitle)
                .foregroundColor(.primary)
            Text(user.jobTitle)
                .foregroundColor(.secondary)
            EmailAddress(address: user.emailAddress)
        }
    }
}
```

And you could even create a larger view that puts a **ProfilePicture** next to a **UserDetails** to give a single visual representation of users, like this:

```
struct UserView: View {
    var user: User

    var body: some View {
        HStack {
            ProfilePicture(imageName: user.profilePicture)
            UserDetails(user: user)
        }
    }
}
```

With this structure we now have several ways of showing users:

- Just their picture
- Just their email address
- Just their job details
- Everything all at once

More importantly, it means that when it comes to *using* all this work, our main content views don't have to worry about what users look like or how they should be treated – all that work is baked into our smaller views.

This means we can create a **UserView** with an example user and have it just work:

```
struct ContentView: View {
    let user = User(name: "Paul Hudson", jobTitle: "Editor,
Hacking with Swift", emailAddress: "paul@hackingwithswift.com",
profilePicture: "paul-hudson")

    var body: some View {
```

```
            UserView(user: user)
        }
    }
```

How to combine text views together

SwiftUI's text view overloads the + operator so that you can combine them together to make new text views.

This is helpful for times when you need to have different formatting across your views, because you can make each text view look exactly as you want then join them together to make a single combined text view. Even better, VoiceOver automatically recognizes them as a single piece of text when it comes to reading them out.

For example, this creates three text views then uses + to join them into a single text view to be returned:

```
var body: some View {
    Text("SwiftUI ")
        .font(.largeTitle)
    + Text("is ")
        .font(.headline)
    + Text("awesome")
        .font(.footnote)
}
```

Warning: Some modifiers change a property of your view, whereas others return a different, modified view, which can cause unhelpful compiler errors. For example, this is not allowed:

```
Text("SwiftUI ")
    .foregroundColor(.red)
+ Text("is ")
    .foregroundColor(.orange)
+ Text("awesome")
    .foregroundColor(.blue)
```

The reason for this is that the **foregroundColor()** modifier returns a modified view, not a text view, so it can't be used with +. To make the code work you should use this instead:

```
Text("SwiftUI ")
    .color(.red)
+ Text("is ")
    .color(.orange)
+ Text("awesome")
    .color(.blue)
```

The **color()** modifier returns a new text view instance with the transformation applied, which is why it still works with +. This modifier only exists on text views, which is why it can return a new text view.

Tip: Combining text views like this is as close as we get to attributed strings in SwiftUI.

How to store views as properties

If you have several views nested inside another view, you might find it useful to create properties for some or all of them to make your layout code easier. You can then reference those properties inline inside your view code, helping to keep it clear.

For example, this creates two text views as properties, then places them inside a **VStack**:

```
struct ContentView : View {
    let title = Text("Paul Hudson")
                .font(.largeTitle)
    let subtitle = Text("Author")
                .foregroundColor(.secondary)

    var body: some View {
        VStack {
            title
            subtitle
        }
    }
}
```

As you can see, just writing the property names in the stack is enough to place them.

However, even better is that you can attach modifiers to those property names, like this:

```
VStack {
    title
        .color(.red)
    subtitle
}
```

That doesn't change the underlying style of **title**, only that one specific usage of it.

How to create custom modifiers

If you find yourself constantly attaching the same set of modifiers to a view – e.g., giving it a background color, some padding, a specific font, and so on – then you can avoid duplication by creating a custom view modifier that encapsulates all those changes. So, rather than say "make it red, make it use a large font" and so on, you can just say "make it look like a warning," and apply a pre-made set of modifiers.

If you want to make your own, define a struct that conforms to the **ViewModifier** protocol. This protocol requires that you accept a **body(content:)** method that transforms some sort of content however you want, returning the result.

For example, this creates a new **PrimaryButton** modifier that adds padding, a red background, white text, and a large font:

```
struct PrimaryButton: ViewModifier {
    func body(content: Content) -> some View {
        content
            .padding()
            .background(Color.red)
            .foregroundColor(Color.white)
            .font(.largeTitle)
    }
}
```

To use that in one of your views add the **.modifier(PrimaryButton())** modifier, like this:

```
struct ContentView : View {
    var body: some View {
        Text("Hello, SwiftUI")
            .modifier(PrimaryButton())
    }
}
```

Chapter 13

Tooling

Build better apps with help from Xcode

How to preview your layout at different Dynamic Type sizes

When building apps it's critical to make sure your layouts work great with all ranges of Dynamic Type. This is partly because SwiftUI natively supports it, partly because many people use *smaller* font sizes because they want a higher information density, but mostly because many people with accessibility needs rely on it.

Fortunately, all of SwiftUI's components natively adapt to Dynamic Type sizes, and it's even easy to preview your designs at various sizes by using the **\.sizeCategory** environment value in your preview

For example, if you wanted to see how a view looks with extra small text, you would add **.environment(\.sizeCategory, .extraSmall)** to your content view preview, like this:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .environment(\.sizeCategory, .extraSmall)
        }
    }
}
#endif
```

You can also send back a group of previews, all using different size categories. This allows you to see the same design at various font sizes side by side.

So, this code shows the design at extra small size, regular size, and the largest possible size:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
```

```
    static var previews: some View {
        Group {
            ContentView()
                .environment(\.sizeCategory, .extraSmall)

            ContentView()

            ContentView()
                .environment(\.sizeCategory, .accessibilityExtraExtraE
xtraLarge)

        }
    }
}
#endif
```

If your design works great across all three of those, you're good to go.

Tip: If your preview is zoomed right in, you should either scroll around or zoom out to the other previews.

How to preview your layout in light and dark mode

Most of Apple's operating systems support both light and dark mode user interfaces, so it's no surprise that SwiftUI has support for this functionality built right in.

Even better, once you've designed your interface Xcode allows you to preview your layouts in either color scheme by setting the **\.colorScheme** environment value in your preview.

For example, this shows a preview using dark mode:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .environment(\.colorScheme, .dark)
        }
    }
}
#endif
```

If you want to see both light and dark mode side by side, place multiple previews in a group, like this:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .environment(\.colorScheme, .light)

            ContentView()
```

```
            .environment(\.colorScheme, .dark)
        }
    }
}
#endif
```

Tip: If your preview is zoomed right in, you should either scroll around or zoom out to the other previews.

How to preview your layout in different devices

Xcode's SwiftUI preview lets us show our designs in multiple screen sizes at the same time using the **.previewDevice()** modifier. This needs to be provided with the exact name of a device as seen in Xcode's destination menu, e.g. "iPhone XS Max".

For example, this shows a preview on the iPhone SE:

```
ContentView()
    .previewDevice(PreviewDevice(rawValue: "iPhone SE"))
```

When using specific devices for previewing, you're likely to find it useful to add in the **.previewDisplayName()** modifier, which lets you put a name under a device in the preview window.

For example, this creates two previews for two different devices, adding the name of each to make it clear what's going on:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
      Group {
        ContentView()
          .previewDevice(PreviewDevice(rawValue: "iPhone SE"))
          .previewDisplayName("iPhone SE")

        ContentView()
          .previewDevice(PreviewDevice(rawValue: "iPhone XS
Max"))
          .previewDisplayName("iPhone XS Max")
      }
```

```
    }
}
#endif
```

How to preview your layout in a navigation view

If you've designed a view that you know will be presented as part of a navigation stack, but doesn't itself contain a navigation view, you won't see its navigation bar title or buttons by default.

Fortunately, you can add your view to a navigation view right inside your preview – this simulates having a navigation bar at the top without actually adding one for the live code, so you can see exactly how it looks.

For example, this view has no navigation view but is configured to display in a specific way when presented as part of one – i.e. being pushed from another view:

```
struct ContentView : View {
    var body: some View {
      Text("Hello World")
        .navigationBarTitle(Text("Welcome"))
    }
}
```

To preview that in a navigation view, just add a **NavigationView** around the content view in your preview, like this:

```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
      NavigationView {
        ContentView()
      }
    }
}
```

```
#endif
```

This allows you to see accurately how the view will look without having to modify the view's actual layout.

How to use Instruments to profile your SwiftUI code and identify slow layouts

Xcode's Instruments tool comes with a fantastic set of analytics for SwiftUI, allowing us to identify how often views were redrawn, how many times calculating the body of a view was slow, and even how our state has changed over time.

First, we need something that is able to provide interesting results we can look at in Instruments. So, this code creates a timer that triggers every 0.01 seconds, and has a body view that shows a random **UUID** and a button that increases the value it shows every time it's tapped:

```
import Combine
import SwiftUI

class FrequentUpdater: BindableObject {
    var didChange = PassthroughSubject<Void, Never>()
    var timer: Timer?

    init() {
        timer = Timer.scheduledTimer(
            withTimeInterval: 0.01,
            repeats: true
        ) { _ in
            self.didChange.send(())
        }
    }
}

struct ContentView : View {
```

```
    @ObjectBinding var updater = FrequentUpdater()
    @State var tapCount = 0

    var body: some View {
      VStack {
        Text("\(UUID().uuidString)")

        Button(action: {
          self.tapCount += 1
        }) {
          Text("Tap count: \(tapCount)")
        }
      }
    }
}
```

If you run that code in the simulator you'll see it's redrawing constantly because it has values that are always changing.

Note: This is a stress test specifically designed to make SwiftUI do a lot of work in order that Instruments shows us interesting data – you do *not* want to use the above code in an actual app.

Instrumenting our code

Now press Cmd+I to run the code through Instruments, and choose the SwiftUI instrument. When it appears, press the record button to make it launch the app and start watching it. Now let it run for a few seconds while you click the button ten or so times, then press stop in Instruments – we have enough data to work with.

By default the SwiftUI instrument tells us a variety of things:

1. How many views were created during that time and how long it took to create them ("View Body")

2. What the properties of the views were and how they changed over time (“View Properties”)
3. How many Core Animation commits took place (“Core Animation Commits”)
4. Exactly how much time each function call took (“Time Profiler”)

Each of these instruments can help you diagnose and solve performance problems in your SwiftUI applications, so it’s worth taking the time to try them out.

For our little stress test sandbox you will see solid walls of color for View Body, View Properties, and Core Animation Commits, which is an immediate red flag. It tells us that not only is SwiftUI having to recreate our views constantly, but that our properties are changing constantly and as a result Core Animation is having to work overtime to keep up.

Monitoring body invocations

If you select the View Body track – that’s the first row in the list of instruments – you should be able to see that Instruments breaks down the results into SwiftUI and your project, with the former being primitive types like text views and buttons, and the latter containing your custom view types. In our case, that will mean “ContentView” should appear for the custom views, because that’s the name of our view.

Now, what you *won’t* see here is a perfect one-to-one mapping of your code to SwiftUI views, because SwiftUI aggressively collapses its view hierarchy to do as little work as possible. So, don’t expect to see any **VStack** creation in the code – that is effectively free for this app.

On this screen, the numbers that matter are Count and Avg Duration – how many times each thing was created, and how long it took. Because this is a stress test you should see very high numbers for Count, but our layout is trivial so the Avg Duration is likely to be a few dozen microseconds.

Tracking state changes

Next, select the View Properties track, which is the second row in the list of instruments. This shows all the properties for all views, including both their current value and all previous values.

Our example app had a button that changes its label when tapped by adding one to a number, and that's visible right there in this instrument – look for the view type ContentView and the Property Type **State<Int>**.

Sadly, Instruments isn't (yet?) able to show us the exact property name there, which might be more confusing if you had several pieces of integer state being tracked. However, it does have a different trick up its sleeve: at the top of the recording window is an arrow marking the current view position, and if you drag that around you'll see exactly how the application state evolved over time – every time you tapped the button you'll see that state integer go up by one, and you can wind forward and backward to see it happen.

This unlocks a *huge* amount of power, because it lets us directly see when state changes caused slow redraws or other work – it's almost like being in a time machine where you can inspect the exact state of your app at every point during its run.

Identifying slow draws

Although SwiftUI is able to drop down straight to Metal for increased performance, most of the time it prefers to use Core Animation for its rendering. This means we automatically get the built-in Core Animation profiling tools from Instruments, including the ability to detect expensive commits.

Core Animation works best when multiple changes are placed together into a single group, known as a *transaction*. We effectively stack up a selection of work in one transaction, then ask CA to proceed with rendering the work – known as *committing* the transaction.

So, when Instruments shows us expensive Core Animation commits, what it's really showing

us is how many times SwiftUI was forced to redraw the pixels on our screen because of updates. In theory this should only happen when the actual state of our app resulted in a different view hierarchy, because SwiftUI should be able to compare the new output of our **body** property with the previous output.

Looking for slow function calls

The final important track is the last one, Time Profiler, which shows us exactly how much time was spent in each part of our code. This works identically to the regular time profiler in Instruments, but if you haven't tried that before here's the least you need to know:

1. The extended detail window on the right shows you the heaviest stack trace by default, which is the piece of code that took longest to run. Bright code (white or black, depending on your macOS color scheme) is code you wrote; dim code (gray) is system library code.
2. On the left you can see all the threads that were created, along with disclosure indicators letting you drill down into the functions they called and the functions *those* functions called, etc. Most of the work will happen inside "start".
3. To avoid clutter you might want to click the Call Tree button at the bottom, then choose Hide System Libraries. This will only show code that you wrote, however if your problem is that you were using the system libraries badly this might not help.
4. To get straight to specific details, you can also click Call Tree and choose Invert Call Tree to flip things around so that leaf functions – those at the end of the tree – are shown at the top, and the disclosure indicators now let you drill down (drill up?) to the functions that called them.

Although the time profiler is extremely useful for identifying performance problems, often just looking at the the heaviest stack trace will highlight the biggest problem.

Last tips

Before you charge off to profile your own code, there are a handful of things to be aware of:

1. When examining a small part of your app's performance, you should click and drag over the range in question so that you see only statistics for that part of the app. This lets you focus on performance for specific actions, such as responding to a button press.
2. Even though you see solid color bars in Instruments, they only look that way from afar – you can zoom by holding down Cmd and pressing - and + to see more detail
3. For the most accurate figures, always profile on a real device.
4. If you want to make changes as a result of profiling your code, always make one change at a time. If you make two changes it's possible one will increase your performance by 20% and the other will decrease it by 10%, but doing them together means you probably think as a whole they increased performance by 10%.
5. Instruments runs your code in release mode, which enables all of Swift's optimizations. This will also affect any debugging flags you've added to your code, so be careful.