

HACKING WITH SWIFT



iOS

SWIFTUI EDITION

Learn to make iOS
apps with real-world
Swift projects

Paul Hudson

Hacking with iOS: SwiftUI Edition

Paul Hudson

Contents

Project 1: WeSplit	10
WeSplit: Introduction	
Understanding the basic structure of a SwiftUI app	
Creating a form	
Adding a navigation bar	
Modifying program state	
Binding state to user interface controls	
Creating views in a loop	
Reading text from the user with TextField	
Creating pickers in a form	
Adding a segmented control for tip percentages	
Calculating the total per person	
Hiding the keyboard	
WeSplit: Wrap up	
Project 2: Guess the Flag	55
Guess the Flag: Introduction	
Using stacks to arrange views	
Colors and frames	
Gradients	
Buttons and images	
Showing alert messages	
Stacking up buttons	
Showing the player's score with an alert	
Styling our flags	
Upgrading our design	
Guess the Flag: Wrap up	
Project 3: Views and Modifiers	91
Views and modifiers: Introduction	
Why does SwiftUI use structs for views?	
What is behind the main SwiftUI view?	
Why modifier order matters	
Why does SwiftUI use “some View” for its view type?	
Conditional modifiers	
Environment modifiers	
Views as properties	

View composition
Custom modifiers
Custom containers
Views and modifiers: Wrap up

Project 4: BetterRest

122

BetterRest: Introduction
Entering numbers with Stepper
Selecting dates and times with DatePicker
Working with dates
Training a model with Create ML
Building a basic layout
Connecting SwiftUI to Core ML
Cleaning up the user interface
BetterRest: Wrap up

Project 5: Word Scramble

152

Word Scramble: Introduction
Introducing List, your best friend
Loading resources from your app bundle
Working with strings
Adding to a list of words
Running code when our app launches
Validating words with UITextChecker
Word Scramble: Wrap up

Project 6: Animation

178

Animation: Introduction
Creating implicit animations
Customizing animations in SwiftUI
Animating bindings
Creating explicit animations
Controlling the animation stack
Animating gestures
Showing and hiding views with transitions
Building custom transitions using ViewModifier
Animation: Wrap up

Project 7: iExpense

210

iExpense: Introduction
Why @State only works with structs

Sharing SwiftUI state with `@StateObject`
Showing and hiding views
Deleting items using `onDelete()`
Storing user settings with `Userdefaults`
Archiving Swift objects with `Codable`
Building a list we can delete from
Working with Identifiable items in SwiftUI
Sharing an observed object with a new view
Making changes permanent with `Userdefaults`
Final polish
iExpense: Wrap up

Project 8: Moonshot 254

Moonshot: Introduction
Resizing images to fit the screen using `GeometryReader`
How `ScrollView` lets us work with scrolling data
Pushing new views onto the stack using `NavigationLink`
Working with hierarchical `Codable` data
How to lay out views in a scrolling grid
Loading a specific kind of `Codable` data
Using generics to load any kind of `Codable` data
Formatting our mission view
Showing mission details with `ScrollView` and `GeometryReader`
Merging `Codable` structs
Finishing up with one last view
Moonshot: Wrap up

Project 9: Drawing 300

Drawing: Introduction
Creating custom paths with SwiftUI
Paths vs shapes in SwiftUI
Adding `strokeBorder()` support with `InsettableShape`
Transforming shapes using `CGAffineTransform` and even-odd fills
Creative borders and fills using `ImagePaint`
Enabling high-performance Metal rendering with `drawingGroup()`
Special effects in SwiftUI: blurs, blending, and more
Animating simple shapes with `animatableData`
Animating complex shapes with `AnimatablePair`
Creating a spirograph with SwiftUI
Drawing: Wrap up

Project 10: Cupcake Corner 344

Cupcake Corner: Introduction
Adding Codable conformance for @Published properties
Sending and receiving Codable data with URLSession and SwiftUI
Loading an image from a remote server
Validating and disabling forms
Taking basic order details
Checking for a valid address
Preparing for checkout
Encoding an ObservableObject class
Sending and receiving orders over the internet
Cupcake Corner: Wrap up

Project 11: Bookworm 384

Bookworm: Introduction
Creating a custom component with @Binding
Accepting multi-line text input with TextEditor
How to combine Core Data and SwiftUI
Creating books with Core Data
Adding a custom star rating component
Building a list with @FetchRequest
Showing book details
Sorting fetch requests with SortDescriptor
Deleting from a Core Data fetch request
Using an alert to pop a NavigationLink programmatically
Bookworm: Wrap up

Project 12: Core Data 426

Core Data: Introduction
Why does \.self work for ForEach?
Creating NSManagedObject subclasses
Conditional saving of NSManagedObjectContext
Ensuring Core Data objects are unique using constraints
Filtering @FetchRequest using NSPredicate
Dynamically filtering @FetchRequest with SwiftUI
One-to-many relationships with Core Data, SwiftUI, and @FetchRequest
Core Data: Wrap up

Project 13: Instafilter 459

Instafilter: Introduction
How property wrappers become structs
Responding to state changes using onChange()
Showing multiple options with confirmationDialog()

Integrating Core Image with SwiftUI
Wrapping a UIViewController in a SwiftUI view
Using coordinators to manage SwiftUI view controllers
How to save images to the user's photo library
Building our basic UI
Importing an image into SwiftUI using PHPickerViewController
Basic image filtering using Core Image
Customizing our filter using confirmationDialog()
Saving the filtered image using UIImageWriteToSavedPhotosAlbum()
Instafilter: Wrap up

Project 14: Bucket List 518

Bucket List: Introduction
Adding conformance to Comparable for custom types
Writing data to the documents directory
Switching view states with enums
Integrating MapKit with SwiftUI
Using Touch ID and Face ID with SwiftUI
Adding user locations to a map
Improving our map annotations
Selecting and editing map annotations
Downloading data from Wikipedia
Sorting Wikipedia results
Introducing MVVM into your SwiftUI project
Locking our UI behind Face ID
Bucket List: Wrap up

Project 15: Accessibility 575

Accessibility: Introduction
Identifying views with useful labels
Hiding and grouping accessibility data
Reading the value of controls
Fixing Guess the Flag
Fixing Word Scramble
Fixing Bookworm
Accessibility: Wrap up

Project 16: Hot Prospects 596

Hot Prospects: Introduction
Reading custom values from the environment with @EnvironmentObject
Creating tabs with TabView and tabItem()
Manually publishing ObservableObject changes

Understanding Swift's Result type
Controlling image interpolation in SwiftUI
Creating context menus
Adding custom row swipe actions to a List
Scheduling local notifications
Adding Swift package dependencies in Xcode
Building our tab bar
Sharing data across tabs using @EnvironmentObject
Dynamically filtering a SwiftUI List
Generating and scaling up a QR code
Scanning QR codes with SwiftUI
Adding options with swipe actions
Saving and loading data with UserDefaults
Adding a context menu to an image
Posting notifications to the lock screen
Hot Prospects: Wrap up

Project 17: Flashzilla

661

Flashzilla: Introduction
How to use gestures in SwiftUI
Making vibrations with UINotificationFeedbackGenerator and Core Haptics
Disabling user interactivity with allowsHitTesting()
Triggering events repeatedly using a timer
How to be notified when your SwiftUI app moves to the background
Supporting specific accessibility needs with SwiftUI
Designing a single card view
Building a stack of cards
Moving views with DragGesture and offset()
Coloring views as we swipe
Counting down with a Timer
Ending the app with allowsHitTesting()
Making iPhones vibrate with UINotificationFeedbackGenerator
Fixing the bugs
Adding and deleting cards
Flashzilla: Wrap up

Project 18: Layout and Geometry

729

Layout and geometry: Introduction
How layout works in SwiftUI
Alignment and alignment guides
How to create a custom alignment guide
Absolute positioning for SwiftUI views
Understanding frames and coordinates inside GeometryReader

ScrollView effects using GeometryReader
Layout and geometry: Wrap up

Project 19: SnowSeeker

759

SnowSeeker: Introduction
Working with two side by side views in SwiftUI
Using alert() and sheet() with optionals
Using groups as transparent layout containers
Making a SwiftUI view searchable
Building a primary list of items
Making NavigationView work in landscape
Creating a secondary view for NavigationView
Searching for data in a List
Changing a view's layout in response to size classes
Binding an alert to an optional string
Letting the user mark favorites
SnowSeeker: Wrap up

Project 1

WeSplit

Learn the basics of SwiftUI with your first project

WeSplit: Introduction

In this project we're going to be building a check-splitting app that you might use after eating at a restaurant – you enter the cost of your food, select how much of a tip you want to leave, and how many people you're with, and it will tell you how much each person needs to pay.

This project isn't trying to build anything complicated, because its real purpose is to teach you the basics of SwiftUI in a useful way while also giving you a real-world project you can expand on further if you want.

You'll learn the basics of UI design, how to let users enter values and select from options, and how to track program state. As this is the first project, we'll be going nice and slow and explaining everything along the way – subsequent projects will slowly increase the speed, but for now we're taking it easy.

This project – like all the projects that involve building a complete app – is broken down into three stages:

1. A hands-on introduction to all the techniques you'll be learning.
2. A step-by-step guide to build the project.
3. Challenges for you to complete on your own, to take the project further.

Each of those are important, so don't try to rush past any of them.

In the first step I'll be teaching you each of the individual new components in isolation, so you can understand how they work individually. There will be lots of code, but also some explanation so you can see how everything works just by itself. This step is an *overview*: here are the things we're going to be using, here is how they work, and here is how you use them.

In the second step we'll be taking those concepts and applying them in a real project. This is where you'll see how things work in practice, but you'll also get more context – here's *why* we want to use them, and here's how they fit together with other components.

In the final step we'll be summarizing what you learned, and then you'll be given a short test

Project 1: WeSplit

to make sure you've really understood what was covered. You'll also be given three challenges: three wholly new tasks that you need to complete yourself, to be sure you're able to apply the skills you've learned. We don't provide solutions for these challenges (so please don't write an email asking for them!), because they are there to test *you* rather than following along with a solution.

Anyway, enough chat: it's time to begin the first project. We're going to look at the techniques required to build our check-sharing app, then use those in a real project.

So, launch Xcode now, and choose Create A New Xcode Project. You'll be shown a list of options, and I'd like you to choose iOS and App, then press Next. On the subsequent screen you need to do the following:

- For Product Name please enter “WeSplit”.
- For Organization Identifier you can enter whatever you want, but if you have a website you should enter it with the components reversed: “hackingwithswift.com” would be “com.hackingwithswift”. If you don't have a domain, make one up – “me.yourlastname.yourfirstname” is perfectly fine.
- For Interface please select SwiftUI.
- For Language please make sure you have Swift selected.
- Make sure all the checkboxes at the bottom are *not* checked.

In case you were curious about the organization identifier, you should look at the text just below: “Bundle Identifier”. Apple needs to make sure all apps can be identified uniquely, and so it combines the organization identifier – your website domain name in reverse – with the name of the project. So, Apple might have the organization identifier of “com.apple”, so Apple's Keynote app might have the bundle identifier “com.apple.keynote”.

When you're ready, click Next, then choose somewhere to save your project and click Create. Xcode will think for a second or two, then create your project and open some code ready for you to edit.

Later on we're going to be using this project to build our check-splitting app, but for now

we're going to use it as a sandbox where we can try out some code.

Important: All the projects we're building in this series target iOS 15.0 or later, but by default Xcode creates new projects to target iOS 14.0. To fix this you need to select "WeSplit" in the project navigator in the left-hand side of Xcode – it's at the very top of the sidebar – then select WeSplit under the Targets list, and finally change "iOS 14.0" to "iOS 15.0" in the Deployment Info section.

Repeat: Yes, I'm repeating this because it's so important: for this project and all projects in this book, you need to update your target's deployment info option so that it uses iOS 15.0 rather than iOS 14.0.

Okay, let's get to it!

Understanding the basic structure of a SwiftUI app

When you create a new SwiftUI app, you'll get a selection of files and maybe 20 lines of code in total.

Inside Xcode you should see the following files in the space on the left, which is called the project navigator:

- WeSplitApp.swift contains code for launching your app. If you create something when the app launches and keep it alive the entire time, you'll put it here.
- ContentView.swift contains the initial user interface (UI) for your program, and is where we'll be doing all the work in this project.
- Assets.xcassets is an *asset catalog* – a collection of pictures that you want to use in your app. You can also add colors here, along with app icons, iMessage stickers, and more.
- Preview Content is a group, with Preview Assets.xcassets inside – this is another asset catalog, this time specifically for example images you want to use when you're designing your user interfaces, to give you an idea of how they might look when the program is running.

Tip: Depending on Xcode's configuration, you may or may not see file extensions in your project navigator. You can control this by going to Xcode's preferences, choosing the General tab, then adjusting the File Extensions option.

All our work for this project will take place in ContentView.swift, which Xcode will already have opened for you. It has some comments at the top – those things marked with two slashes at the start – and they are ignored by Swift, so you can use them to add explanations about how your code works.

Below the comments are ten or so lines of code:

```
import SwiftUI
```

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, world!")  
            .padding()  
    }  
}  
  
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}
```

Before we start writing our own code, it's worth going over what all that does, because a couple of things will be new.

First, **import SwiftUI** tells Swift that we want to use all the functionality given to us by the SwiftUI framework. Apple provides us with many frameworks for things like machine learning, audio playback, image processing, and more, so rather than assume our program wants to use everything ever we instead say which parts we want to use so they can be loaded.

Second, **struct ContentView: View** creates a new struct called **ContentView**, saying that it conforms to the **View** protocol. **View** comes from SwiftUI, and is the basic protocol that must be adopted by anything you want to draw on the screen – all text, buttons, images, and more are all views, including your own layouts that combine other views.

Third, **var body: some View** defines a new computed property called **body**, which has an interesting type: **some View**. This means it will return something that conforms to the **View** protocol, which is our layout. Behind the scenes this will actually result in a very complicated data type being returned based on all the things in our layout, but **some View** means we don't need to worry about that.

Project 1: WeSplit

The **View** protocol has only one requirement, which is that you have a computed property called **body** that returns **some View**. You can (and will) add more properties and methods to your view structs, but **body** is the only thing that's required.

Fourth, **Text("Hello, world!")** creates a text view using the string “Hello, world!” Text views are simple pieces of static text that get drawn onto the screen, and will automatically wrap across multiple lines as needed.

Fifth, you'll see the **padding()** method is being called on the text view. This is what Swift calls a *modifier*, which are regular methods with one small difference: they always return a new view that contains both your original data, plus the extra modification you asked for. In our case that means **body** will return a padded text view, not just a regular text view.

Below the **ContentView** struct you'll see a **ContentView_Previews** struct, which conforms to the **PreviewProvider** protocol. This piece of code won't actually form part of your final app that goes to the App Store, but is instead specifically for Xcode to use so it can show a preview of your UI design alongside your code.

These previews use an Xcode feature called the canvas, which is usually visible directly to the right of your code. You can customize the preview code if you want, and they will only affect the way the canvas shows your layouts – it won't change the actual app that gets run.

The canvas will automatically preview using one specific Apple device, such as the iPhone 13 Pro or an iPod touch. To change this, look at the top center of your Xcode window for the current device, then click on it and select an alternative. This will also affect how your code is run in the virtual iOS simulator later on.

Important: If you don't see the canvas in your Xcode window, go to the Editor menu and select Canvas.

Very often you'll find that an error in your code stops Xcode's canvas from updating – you'll see something like “Automatic preview updating paused”, and can press Resume to fix it. As you'll be doing this a lot, let me recommend an important shortcut: Option+Cmd+P does the same as clicking Resume.

Understanding the basic structure of a SwiftUI app

Creating a form

Many apps require users to enter some sort of input – it might be asking them to set some preferences, it might be asking them to confirm where they want a car to pick them up, it might be to order food from a menu, or anything similar.

SwiftUI gives us a dedicated view type for this purpose, called **Form**. Forms are scrolling lists of static controls like text and images, but can also include user interactive controls like text fields, toggle switches, buttons, and more.

You can create a basic form just by wrapping the default text view inside **Form**, like this:

```
var body: some View {
    Form {
        Text("Hello, world!")
            .padding()
    }
}
```

If you're using Xcode's canvas, you'll see it change quite dramatically: before Hello World was centered on a white screen, but now the screen is a light gray, and Hello World appears in the top left in white.

You can remove the padding too – we'll come back to that later:

```
Form {
    Text("Hello, world!")
}
```

What you're seeing here is the beginnings of a list of data, just like you'd see in the Settings app. We have one row in our data, which is the Hello World text, but we can add more freely and have them appear in our form immediately:

```
Form {
```

```
Text("Hello, world!")
Text("Hello, world!")
Text("Hello, world!")
}
}
```

In fact, you can have as many things inside a form as you want, although if you intend to add more than 10 SwiftUI requires that you place things in groups to avoid problems.

For example, this code shows ten rows of text just fine:

```
Form {
    Text("Hello, world!")
    Text("Hello, world!")
}
```

But this attempts to show 11, which is not allowed:

```
Form {
    Text("Hello, world!")
    Text("Hello, world!")
}
```

Project 1: WeSplit

```
Text("Hello, world!")
Text("Hello, world!")
Text("Hello, world!")
Text("Hello, world!")
}
}
```

Tip: In case you were curious why 10 rows are allowed but 11 is not, this is a limitation in SwiftUI: it was coded to understand how to add one thing to a form, how to add two things to a form, how to add three things, four things, five things, and more, all the way up to 10, but not beyond – they needed to draw a line somewhere. This limit of 10 children inside a parent actually applies everywhere in SwiftUI.

If you wanted to have 11 things inside the form you should put some rows inside a **Group**:

```
Form {
    Group {
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
    }

    Group {
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
        Text("Hello, world!")
    }
}
```

Groups don't actually change the way your user interface looks, they just let us work around SwiftUI's limitation of ten child views inside a parent – that's text views inside a form, in this instance.

If you *want* your form to look different when splitting items into chunks, you should use the **Section** view instead. This splits your form into discrete visual groups, just like the Settings app does:

```
Form {  
    Section {  
        Text("Hello, world!")  
    }  
  
    Section {  
        Text("Hello, world!")  
        Text("Hello, world!")  
    }  
}
```

There's no hard and fast rule when you should split a form into sections – it's just there to group related items visually.

Adding a navigation bar

If we ask for it, iOS allows us to place content anywhere on the screen, including under the system clock at the top and the home indicator at the bottom. This doesn't look great, which is why by default SwiftUI ensures components are placed in an area where they can't be covered up by system UI or device rounded corners – an area known as the *safe area*.

On an iPhone 13, the safe area spans the space from just below the notch down to just above the home indicator. You can see it in action with a user interface like this one:

```
struct ContentView: View {  
    var body: some View {  
        Form {  
            Section {  
                Text("Hello, world!")  
            }  
        }  
    }  
}
```

Try running that in the iOS simulator – press the Play button in the top-left corner of Xcode's window, or press Cmd+R.

You'll see that the form starts below the notch, so by default the row in our form is fully visible. However, forms can also scroll, so if you swipe around in the simulator you'll find you can move the row up so it goes under the clock, making them both hard to read.

A common way of fixing this is by placing a navigation bar at the top of the screen. Navigation bars can have titles and buttons, and in SwiftUI they also give us the ability to display new views when the user performs an action.

We'll get to buttons and new views in a later project, but I do at least want to show you how to add a navigation bar and give it a title, because it makes our form look better when it scrolls.

You've seen that we can place a text view inside a section by adding **Section** around the text view, and that we can place the section inside a **Form** in a similar way. Well, we add a navigation bar in just the same way, except here it's called **NavigationView**.

```
var body: some View {
    NavigationView {
        Form {
            Section {
                Text("Hello, world!")
            }
        }
    }
}
```

When you see that code in Xcode's canvas, you'll notice there's a large gray space at the top of your UI. Well, that's our navigation bar in action, and if you run your code in the simulator you'll see the form slides under the bar as it moves to the top of the screen.

You'll usually want to put some sort of title in the navigation bar, and you can do that by attaching a *modifier* to whatever you've placed inside.

Let's try adding a modifier to set the navigation title for our form:

```
NavigationView {
    Form {
        Section {
            Text("Hello, world!")
        }
    }
    .navigationTitle("SwiftUI")
}
```

When we attach the **.navigationTitle()** modifier to our form, Swift actually creates a new form

Project 1: WeSplit

that has a navigation title plus all the existing contents you provided.

When you add a title to a navigation bar, you'll notice it uses a large font for that title. You can get a small font by adding another modifier:

```
.navigationBarTitleDisplayMode(.inline)
```

You can see how Apple uses these large and small titles in the Settings app: the first screen says “Settings” in large text, and subsequent screens show their titles in small text.

Modifying program state

There's a saying among SwiftUI developers that our "views are a function of their state," but while that's only a handful of words it might be quite meaningless to you at first.

If you were playing a fighting game, you might have lost a few lives, scored some points, collected some treasure, and perhaps picked up some powerful weapons. In programming, we call these things *state* – the active collection of settings that describe how the game is right now.

When we say SwiftUI's views are a function of their state, we mean that the way your user interface looks – the things people can see and what they can interact with – are determined by the state of your program. For example, they can't tap Continue until they have entered their name in a text field.

Let's put this into practice with a button, which in SwiftUI can be created with a title string and an action closure that gets run when the button is tapped:

```
struct ContentView: View {
    var tapCount = 0

    var body: some View {
        Button("Tap Count: \((tapCount)") {
            tapCount += 1
        }
    }
}
```

That code looks reasonable enough: create a button that says "Tap Count" plus the number of times the button has been tapped, then add 1 to **tapCount** whenever the button is tapped.

However, it won't build; that's not valid Swift code. You see, **ContentView** is a struct, which might be created as a constant. If you think back to when you learned about structs, that means it's *immutable* – we can't change its values freely.

Project 1: WeSplit

When creating struct methods that want to change properties, we need to add the **mutating** keyword: **mutating func doSomeWork()**, for example. However, Swift doesn't let us make mutating computed properties, which means we can't write **mutating var body: some View** – it just isn't allowed.

This might seem like we're stuck at an impasse: we want to be able to change values while our program runs, but Swift won't let us because our views are structs.

Fortunately, Swift gives us a special solution called a *property wrapper*: a special attribute we can place before our properties that effectively gives them super-powers. In the case of storing simple program state like the number of times a button was tapped, we can use a property wrapper from SwiftUI called **@State**, like this:

```
struct ContentView: View {  
    @State var tapCount = 0  
  
    var body: some View {  
        Button("Tap Count: \(tapCount)") {  
            self.tapCount += 1  
        }  
    }  
}
```

That small change is enough to make our program work, so you can now build it and try it out.

@State allows us to work around the limitation of structs: we know we can't change their properties because structs are fixed, but **@State** allows that value to be stored separately by SwiftUI in a place that *can* be modified.

Yes, it feels a bit like a cheat, and you might wonder why we don't use classes instead – they *can* be modified freely. But trust me, it's worthwhile: as you progress you'll learn that SwiftUI destroys and recreates your structs frequently, so keeping them small and simple is important for performance.

Tip: There are several ways of storing program state in SwiftUI, and you'll learn all of them. `@State` is specifically designed for simple properties that are stored in one view. As a result, Apple recommends we add **private** access control to those properties, like this: `@State private var tapCount = 0`.

Binding state to user interface controls

SwiftUI's `@State` property wrapper lets us modify our view structs freely, which means as our program changes we can update our view properties to match.

However, things are a little more complex with user interface controls. For example, if you wanted to create an editable text box that users can type into, you might create a SwiftUI view like this one:

```
struct ContentView: View {  
    var body: some View {  
        Form {  
            TextField("Enter your name")  
            Text("Hello, world!")  
        }  
    }  
}
```

That tries to create a form containing a text field and a text view. However, that code won't compile because SwiftUI wants to know where to store the text in the text field.

Remember, views are a function of their state – that text field can only show something if it reflects a value stored in your program. What SwiftUI wants is a string property in our struct that can be shown inside the text field, and will also store whatever the user types in the text field.

So, we could change the code to this:

```
struct ContentView: View {  
    var name = ""  
  
    var body: some View {
```

```

Form {
    TextField("Enter your name", text: name)
    Text("Hello, world!")
}
}
}

```

That adds a **name** property, then uses it to create the text field. However, that code *still* won't work because Swift needs to be able to update the **name** property to match whatever the user types into the text field, so you might use **@State** like this:

```
@State private var name = ""
```

But that still isn't enough, and our code still won't compile.

The problem is that Swift differentiates between “show the value of this property here” and “show the value of this property here, *but write any changes back to the property.*”

In the case of our text field, Swift needs to make sure whatever is in the text is also in the **name** property, so that it can fulfill its promise that our views are a function of their state – that everything the user can see is just the visible representation of the structs and properties in our code.

This is what's called a *two-way binding*: we bind the text field so that it shows the value of our property, but we also bind it so that any changes to the text field also update the property.

In Swift, we mark these two-way bindings with a special symbol so they stand out: we write a dollar sign before them. This tells Swift that it should read the value of the property but also write it back as any changes happen.

So, the correct version of our struct is this:

```

struct ContentView: View {
    @State private var name = ""

```

Project 1: WeSplit

```
var body: some View {
    Form {
        TextField("Enter your name", text: $name)
        Text("Hello, world!")
    }
}
```

Try running that code now – you should find you can tap on the text field and enter your name, as expected.

Before we move on, let's modify the text view so that it shows the user's name directly below their text field:

```
Text("Your name is \(name)")
```

Notice how that uses **name** rather than **\$name**? That's because we don't want a two-way binding here – we want to *read* the value, yes, but we don't want to write it back somehow, because that text view won't change.

So, when you see a dollar sign before a property name, remember that it creates a two-way binding: the value of the property is read, but also written.

Creating views in a loop

It's common to want to create several SwiftUI views inside a loop. For example, we might want to loop over an array of names and have each one be a text view, or loop over an array of menu items and have each one be shown as an image.

SwiftUI gives us a dedicated view type for this purpose, called **ForEach**. This can loop over arrays and ranges, creating as many views as needed. Even better, **ForEach** doesn't get hit by the 10-view limit that would affect us if we had typed the views by hand.

ForEach will run a closure once for every item it loops over, passing in the current loop item. For example, if we looped from 0 to 100 it would pass in 0, then 1, then 2, and so on.

For example, this creates a form with 100 rows:

```
Form {
    ForEach(0 ..< 100) { number in
        Text("Row \($number)")
    }
}
```

Because **ForEach** passes in a closure, we can use shorthand syntax for the parameter name, like this:

```
Form {
    ForEach(0 ..< 100) {
        Text("Row \($0)")
    }
}
```

ForEach is particularly useful when working with SwiftUI's **Picker** view, which lets us show various options for users to select from.

To demonstrate this, we're going to define a view that:

Project 1: WeSplit

1. Has an array of possible student names.
2. Has an **@State** property storing the currently selected student.
3. Creates a **Picker** view asking users to select their favorite, using a two-way binding to the **@State** property.
4. Uses **ForEach** to loop over all possible student names, turning them into a text view.

Here's the code for that:

```
struct ContentView: View {  
    let students = ["Harry", "Hermione", "Ron"]  
    @State private var selectedStudent = "Harry"  
  
    var body: some View {  
        NavigationView {  
            Form {  
                Picker("Select your student", selection:  
$selectedStudent) {  
                    ForEach(students, id: \.self) {  
                        Text($0)  
                    }  
                }  
            }  
        }  
    }  
}
```

There's not a lot of code in there, but it's worth clarifying a few things:

1. The **students** array doesn't need to be marked with **@State** because it's a constant; it isn't going to change.
2. The **selectedStudent** property starts with the value "Harry" but can change, which is why it's marked with **@State**.

3. The **Picker** has a label, “Select your student”, which tells users what it does and also provides something descriptive for screen readers to read aloud.
4. The **Picker** has a two-way binding to **selectedStudent**, which means it will start showing a selection of “Harry” but update the property when the user selects something else.
5. Inside the **ForEach** we loop over all the students.
6. For each student we create one text view, showing that student’s name.

The only confusing part in there is this: **ForEach(students, id: \.self)**. That loops over the **students** array so we can create a text view for each one, but **the id: \.self part is important**. **This exists because SwiftUI needs to be able to identify every view on the screen uniquely, so it can detect when things change**.

For example, if we rearranged our array so that Ron came first, SwiftUI would move its text view at the same time. So, we need to tell SwiftUI how it can identify each item in our string array uniquely – what about each string makes it unique? If we had an array of structs we might say “oh, my struct has a **title** string that is always unique,” or “my struct has an **id** integer that is always unique.” Here, though, we just have an array of simple strings, and the only thing unique about the string is the string itself: each string in our array is different, so the strings are naturally unique.

So, when we’re using **ForEach** to create many views and SwiftUI asks us what identifier makes each item in our string array unique, our answer is **\.self**, which means “the strings themselves are unique.” This does of course mean that if you added duplicate strings to the **students** array you might hit problems, but here it’s just fine.

Anyway, we’ll look at other ways to use **ForEach** in the future, but that’s enough for this project.

This is the final part of the overview for this project, so it’s almost time to get started with the real code. If you want to save the examples you’ve programmed you should copy your project directory somewhere else.

When you’re ready, put `ContentView.swift` back to the way it started when you first made the

Project 1: WeSplit

project, so we have a clean slate to work from:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Reading text from the user with TextField

We're building a check-splitting app, which means users need to be able to enter the cost of their check, how many people are sharing the cost, and how much tip they want to leave.

Hopefully already you can see that means we need to add three `@State` properties, because there are three pieces of data we're expecting users to enter into our app.

So, start by adding these three properties to our `ContentView` struct:

```
@State private var checkAmount = 0.0  
@State private var numberOfPeople = 2  
@State private var tipPercentage = 20
```

As you can see, that gives us a default of 0.0 for the check amount, a default value of 2 for the number of people, and a default value of 2 for the tip percentage. Each of these properties have a sensible default: we don't know how much the check will come to, but assuming two people and a 20% tip both seem like good starting points for the app.

Of course, some people prefer to leave a different percentage of tip, so we're going to let them select values from a predetermined array of tip sizes. We need to store the list of possible tip sizes somewhere, so please add this fourth property beneath the previous three:

```
let tipPercentages = [10, 15, 20, 25, 0]
```

We're going to build up the form step by step, starting with a text field where users can enter the value of their check. We'll start with what you know already, but as you'll see that won't quite work right.

Modify the `body` property to this:

```
Form {
```

Project 1: WeSplit

```
Section {  
    TextField("Amount", text: $checkAmount)  
}  
}
```

That isn't going to work, and that's okay. The problem is that SwiftUI likes **TextField** to be used for entering text – strings, that is. We *could* allow that here, but it would mean users could enter any kind of text, and we'd need to carefully convert that string to a number we can work with.

Fortunately, we can do better: we can pass our **Double** to **TextField** and ask it to treat the input as a currency, like this:

```
TextField("Amount", value: $checkAmount,  
format: .currency(code: "USD"))
```

That's an improvement, but we can do even better. You see, that tells SwiftUI we want the currency formatted as US dollars, or USD for short, but given that over 95% of the world's population don't use US dollars as their currency we should probably not force "USD" on them.

A better solution is to ask iOS if it can give us the currency code for the current user, if there is one. This might be USD, but it might also be CAD (Canadian dollars), AUD (Australian dollars), JPY (Japanese Yen) and more – or it might not currently have a value, if the user hasn't set one.

So, a better format to use is this:

```
.currency(code: Locale.current.currencyCode ?? "USD"))
```

Locale is a massive struct built into iOS that is responsible for storing all the user's region settings – what calendar they use, how they separate thousands digits in numbers, whether they use the metric system, and more. In our case, we're asking whether the user has a preferred

Reading text from the user with TextField

currency code, and if they don't we'll fall back to "USD" so at least we have *something*.

So far our code creates a scrolling entry form of one section, which in turn contains one row: our text field. When you create text fields in forms, the first parameter is a string that gets used as the *placeholder* – gray text shown inside the text field, giving users an idea of what should be in there. The second parameter is the two-way binding to our **checkAmount** property, which means as the user types that property will be updated. The third parameter here is the one that controls the way the text is formatted, making it a currency.

One of the great things about the **@State** property wrapper is that it automatically watches for changes, and when something happens it will automatically re-invoke the **body** property. That's a fancy way of saying it will reload your UI to reflect the changed state, and it's a fundamental feature of the way SwiftUI works.

To demonstrate this, we could add a second section with a text view showing the value of **checkAmount**, like this:

```
Form {
    Section {
        TextField("Amount", value: $checkAmount,
format: .currency(code: Locale.current.currencyCode ?? "USD"))
    }

    Section {
        Text(checkAmount, format: .currency(code:
Locale.current.currencyCode ?? "USD"))
    }
}
```

That does almost exactly the same thing as our **TextField**: it asks SwiftUI to format the number as a currency, using either the system default or USD if nothing else is available. Later on in this project we'll be using a different format style to show percentages – these text formatters are really helpful!

Project 1: WeSplit

We'll be making that show something else later on, but for now please run the app in the simulator so you can try it yourself.

Tap on the check amount text field, then enter an example amount such as 50. What you'll see is that as you type the text view in the second section automatically and immediately reflects your actions.

This synchronization happens because:

1. Our text field has a two-way binding to the **checkAmount** property.
2. The **checkAmount** property is marked with **@State**, which automatically watches for changes in the value.
3. When an **@State** property changes SwiftUI will re-invoke the **body** property (i.e., reload our UI)
4. Therefore the text view will get the updated value of **checkAmount**.

The final project won't show **checkAmount** in that text view, but it's good enough for now. Before we move on, though, I want to address one important problem: when you tap to enter text into our text field, users see a regular alphabetical keyboard. Sure, they can tap a button on the keyboard to get to the numbers screen, but it's annoying and not really necessary.

Fortunately, text fields have a modifier that lets us force a different kind of keyboard: **keyboardType()**. We can give this a parameter specifying the kind of keyboard we want, and in this instance either **.numberPad** or **.decimalPad** are good choices. Both of those keyboards will show the digits 0 through 9 for users to tap on, but **.decimalPad** also shows a decimal point so users can enter check amount like \$32.50 rather than just whole numbers.

So, modify your text field to this:

```
TextField("Amount", value: $checkAmount,  
format: .currency(code: Locale.current.currencyCode ?? "USD"))  
.keyboardType(.decimalPad)
```

You'll notice I added a line break before **.keyboardType** and also indented it one level deeper

Reading text from the user with `TextField`

than **TextField** – that isn't required, but it can help you keep track of which modifiers apply to which views.

Go ahead and run the app now and you should find you can now only type numbers into the text field.

Tip: The `.numberPad` and `.decimalPad` keyboard types tell SwiftUI to show the digits 0 through 9 and optionally also the decimal point, but that doesn't stop users from *entering* other values. For example, if they have a hardware keyboard they can type what they like, and if they copy some text from elsewhere they'll be able to paste that into the text field no matter what is inside that text. That's OK, though – the text field will automatically filter out bad values when they hit Return.

Creating pickers in a form

SwiftUI's pickers serve multiple purposes, and exactly how they look depends on which device you're using and the context where the picker is used.

In our project we have a form asking users to enter how much their check came to, and we want to add a picker to that so they can select how many people will share the check.

Pickers, like text fields, need a two-way binding to a property so they can track their value. We already made an `@State` property for this purpose, called `numberOfPeople`, so our next job is to loop over all the numbers from 2 through to 99 and show them inside a picker.

Modify the first section in your form to include a picker, like this:

```
Section {
    TextField("Amount", value: $checkAmount,
        format: .currency(code: Locale.current.currencyCode ?? "USD"))
        .keyboardType(.decimalPad)

    Picker("Number of people", selection: $numberOfPeople) {
        ForEach(2 ..< 100) {
            Text("\($0) people")
        }
    }
}
```

Now run the program in the simulator and try it out – what do you notice?

Hopefully you spot several things:

1. There's a new row that says “Number of people” on the left and “4 people” on the right.
2. There's a gray disclosure indicator on the right edge, which is the iOS way of signaling that tapping the row shows another screen.
3. Tapping the row *doesn't* show another screen.

4. The row says “4 people”, but we gave our **numberOfPeople** property a default value of 2.

So, it’s a bit of “two steps forward, two steps back” – we have a nice result, but it doesn’t work and doesn’t show the right information!

We’ll fix both of those, starting with the easy one: why does it say 4 people when we gave **numberOfPeople** the default value of 2? Well, when creating the picker we used a **ForEach** view like this:

```
ForEach(2 ..< 100) {
```

That counts from 2 up to 100, creating rows. What that means is that our 0th row – the first that is created – contains “2 People”, so when we gave **numberOfPeople** the value of 2 we were actually setting it to the *third* row, which is “4 People”.

So, although it’s a bit brain-bending, the fact that our UI shows “4 people” rather than “2 people” isn’t a bug. But there is still a large bug in our code: why does tapping on the row do nothing?

If you create a picker by itself, outside a form, iOS will show a spinning wheel of options. Here, though, we’ve told SwiftUI that this is a form for user input, and so it has automatically changed the way our picker looks so that it doesn’t take up so much space.

What SwiftUI *wants* to do – which is also why it’s added the gray disclosure indicator on the right edge of the row – is show a new view with the options from our picker. To do that, we need to add a navigation view, which does two things: gives us some space across the top to place a title, and also lets iOS slide in new views as needed.

So, directly before the form add **NavigationView** {}, and after the form’s closing brace add another closing brace. If you got it right, your code should look like this:

```
var body: some View {
    NavigationView {
        Form {
```

Project 1: WeSplit

```
// everything inside your form
}
}
}
```

If you run the program again you'll see a large gray space at the top, which is where iOS is giving us room to place a title. We'll do that in a moment, but first try tapping on the Number Of People row and you should see a new screen slide in with all the other possible options to choose from.

You should see that “4 People” has a checkmark next to it because it’s the selected value, but you can also tap a different number instead – the screen will automatically slide away again, taking the user back to the previous screen with their new selection.

What you’re seeing here is the importance of what’s called *declarative user interface design*. This means we say *what* we want rather than say *how* it should be done. We said we wanted a picker with some values inside, but it was down to SwiftUI to decide whether a wheel picker or the sliding view approach is better. It’s choosing the sliding view approach because the picker is inside a form, but on other platforms and environments it could choose something else.

Before we’re done with this step, let’s add a title to that new navigation bar. Give the form this modifier:

```
.navigationTitle("WeSplit")
```

Tip: It’s tempting to think that modifier should be attached to the end of the **NavigationView**, but it needs to be attached to the end of the **Form** instead. The reason is that navigation views are capable of showing many views as your program runs, so by attaching the title to the thing *inside* the navigation view we’re allowing iOS to change titles freely.

Adding a segmented control for tip percentages

Now we're going to add a second picker view to our app, but this time we want something slightly different: we want a *segmented control*. This is a specialized kind of picker that shows a handful of options in a horizontal list, and it works great when you have only a small selection to choose from.

Our form already has two sections: one for the amount and number of people, and one where we'll show the final result – it's just showing **checkAmount** for now, but we're going to fix it soon.

In the middle of those two sections I'd like you to add a third to show tip percentages:

```
Section {  
    Picker("Tip percentage", selection: $tipPercentage) {  
        ForEach(tipPercentages, id: \.self) {  
            Text($0, format: .percent)  
        }  
    }  
}
```

That loops over all the options in our **tipPercentages** array, converting each one into a text view with the **.percent** format. Just like the previous picker, SwiftUI will convert that to a single row in our list, and slide a new screen of options in when it's tapped.

Here, though, I want to show you how to use a segmented control instead, because it looks much better. So, please add this modifier to the tip picker:

```
.pickerStyle(.segmented)
```

That should go at the end of the picker's closing brace, like this:

Project 1: WeSplit

```
Section {
    Picker("Tip percentage", selection: $tipPercentage) {
        ForEach(tipPercentages, id: \.self) {
            Text($0, format: .percent)
        }
    }
    .pickerStyle(.segmented)
}
```

If you run the program now you'll see things are starting to come together: users can now enter the amount on their check, select the number of people, and select how much tip they want to leave – not bad!

But things aren't *quite* what you think. One problem app developers face is that we take for granted that our app does what we intended it to do – we designed it to solve a particular problem, so we automatically know what everything means.

Try to look at our user interface with fresh eyes, if you can:

- “Amount” makes sense – it's a box users can type a number into.
- “Number of people” is also pretty self-explanatory.
- The label at the bottom is where we'll show the total, so right now we can ignore that.
- That middle section, though – what are those percentages for?

Yes, *we* know they are to select how much tip to leave, but that isn't obvious on the screen. We can – and *should* do better.

One option is to add another text view directly before the segmented control, which we could do like this:

```
Section {
    Text("How much tip do you want to leave?")
}
```

Adding a segmented control for tip percentages

```
Picker("Tip percentage", selection: $tipPercentage) {
    ForEach(tipPercentages, id: \.self) {
        Text($0, format: .percent)
    }
}
.pickerStyle(.segmented)
```

That works OK, but it doesn't look great – it looks like it's an item all by itself, rather than a label for the segmented control.

A much better idea is to modify the section itself: SwiftUI lets us add views to the header and footer of a section, which in this instance we can use to add a small explanatory prompt. In fact, we can use the same text view we just created, just moved to be the section header rather than a loose label inside it.

Here's how that looks in code:

```
Section {
    Picker("Tip percentage", selection: $tipPercentage) {
        ForEach(tipPercentages, id: \.self) {
            Text($0, format: .percent)
        }
    }
    .pickerStyle(.segmented)
} header: {
    Text("How much tip do you want to leave?")
}
```

That uses multiple trailing closures to specify both the section body (the first closure) and the second header (the second closure).

It's a small change to the code, but I think the end result looks a lot better – the text now looks

Project 1: WeSplit

like a prompt for the segmented control directly below it.

Calculating the total per person

So far the final section in our form has shown a simple text view with whatever check amount the user entered, but now it's time for the important part of this project: we want that text view to show how much each person needs to contribute to the payment.

There are a few ways we could solve this, but the easiest one also happens to be the *cleanest* one, by which I mean it gives us code that is clear and easy to understand: we're going to add a computed property that calculates the total.

This needs to do a small amount of mathematics: the total amount payable per person is equal to the value of the order, plus the tip percentage, divided by the number of people.

But before we can get to that point, we first need to pull out the values for how many people there are, what the tip percentage is, and the value of the order. That might sound easy, but as you've already seen, **numberOfPeople** is off by 2 – when it stores the value 3 it means 5 people.

So, we're going to create a new computed property called **totalPerPerson** that will be a **Double**, and it will start off by getting the input data ready: what is the correct number of people, and how much tip do they want to leave?

First, add the computed property itself, just before the **body** property:

```
var totalPerPerson: Double {
    // calculate the total per person here
    return 0
}
```

That sends back 0 so your code doesn't break, but we're going to replace the **// calculate the total per person here** comment with our calculations.

Next, we can figure out how many people there are by reading **numberOfPeople** and adding 2 to it. Remember, this thing has the range 2 to 100, but it *counts* from 0, which is why we need

Project 1: WeSplit

to add the 2.

So, start by replacing **// calculate the total per person here** with this:

```
let peopleCount = Double(numberOfPeople + 2)
```

You'll notice that converts the resulting value to a **Double** because it needs to be used alongside the **checkAmount**.

For the same reason, we also need to convert our tip percentage into a **Double**:

```
let tipSelection = Double(tipPercentage)
```

Now that we have our input values, it's time do our mathematics. This takes another three steps:

- We can calculate the tip value by dividing **checkAmount** by 100 and multiplying by **tipSelection**.
- We can calculate the grand total of the check by adding the tip value to **checkAmount**.
- We can figure out the amount per person by dividing the grand total by **peopleCount**.

Once that's done, we can return the amount per person and we're done.

Replace **return 0** in the property with this:

```
let tipValue = checkAmount / 100 * tipSelection
let grandTotal = checkAmount + tipValue
let amountPerPerson = grandTotal / peopleCount

return amountPerPerson
```

If you've followed everything correctly your code should look like this:

```
var totalPerPerson: Double {
```

Calculating the total per person

```
let peopleCount = Double(numberOfPeople + 2)
let tipSelection = Double(tipPercentage)

let tipValue = checkAmount / 100 * tipSelection
let grandTotal = checkAmount + tipValue
let amountPerPerson = grandTotal / peopleCount

return amountPerPerson
}
```

Now that **totalPerPerson** gives us the correct value, we can change the final section in our table so it shows the correct text.

Replace this:

```
Section {
    Text(checkAmount, format: .currency(code:
Locale.current.currencyCode ?? "USD"))
}
```

With this:

```
Section {
    Text(totalPerPerson, format: .currency(code:
Locale.current.currencyCode ?? "USD"))
}
```

Try running the app now, and see what you think. You should find that because all the values that make up our total are marked with **@State**, changing any of them will cause the total to be recalculated automatically.

Hopefully you're now seeing for yourself what it means that SwiftUI's views are a function of their state – when the state changes, the views automatically update to match.

Hiding the keyboard

We're now almost at the end of our project, but you might have spotted an annoyance: once the keyboard appears for the check amount entry, it never goes away!

This is a problem with the decimal and number keypads, because the regular alphabetic keyboard has a return key on there to dismiss the keyboard. However, with a little extra work we can fix this:

1. We need to give SwiftUI some way of determining whether the check amount box should currently have *focus* – should be receiving text input from the user.
2. We need to add some kind of button to remove that focus when the user wants, which will in turn cause the keyboard to go away.

To solve the first one you need to meet your second property wrapper: **@FocusState**. This is exactly like a regular **@State** property, except it's specifically designed to handle input focus in our UI.

Add this new property to **ContentView**:

```
@FocusState private var amountIsFocused: Bool
```

Now we can attach that to our text field, so that when the text field is focused **amountIsFocused** is true, otherwise it's false. Add this modifier to your **TextField**:

```
.focused($amountIsFocused)
```

That's the first part of our problem solved: although we can't see anything different on the screen, SwiftUI is at least silently aware of whether the text field should have focus or not.

The second part of our solution is to add a toolbar to the keyboard when it appears, so we can place a Done button in there. To make this work really well you need to meet several new SwiftUI views, so I think the best thing to do is show you the code then explain what it does.

Add this new modifier to your form, below the existing **navigationTitle()** modifier:

```
.toolbar {
    ToolbarItemGroup(placement: .keyboard) {
        Button("Done") {
            amountIsFocused = false
        }
    }
}
```

Yes, that's quite a lot of code, so let's break it down:

1. The **toolbar()** modifier lets us specify toolbar items for a view. These toolbar items might appear in various places on the screen – in the navigation bar at the top, in a special toolbar area at the bottom, and so on.
2. **ToolbarItemGroup** lets us place one or more buttons in a specific location, and this is where we get to specify we want a *keyboard* toolbar – a toolbar that is attached to the keyboard, so it will automatically appear and disappear with the keyboard.
3. The **Button** view we're using here displays some tappable text, which in our case is “Done”. We also need to provide it with some code to run when the button is pressed, which in our case sets **amountIsFocused** to false so that the keyboard is dismissed.

You'll meet these more in the future, but for now I recommend you run the program and try it out – it's a big improvement!

Before we're done, there's one last small change I want to make: I'd like you to modify the **ToolbarItemGroup** to this:

```
ToolbarItemGroup(placement: .keyboard) {
    Spacer()

    Button("Done") {
        amountIsFocused = false
    }
}
```

Project 1: WeSplit

```
    }  
}
```

That adds one small but important new view before the button, called **Spacer**. This is a *flexible space* by default – wherever you place a spacer it will automatically push other views to one side. That might mean pushing them up, down, left, or right depending on where it's used, but by placing it first in our toolbar it will cause our button to be pushed to the right.

If you run the app again you'll see the difference – it's really minor, but having the Done button on the right of the keyboard is also the same thing other iOS apps do, and it's good to make our own code adopt those conventions.

Anyway, that was the last step in this project – pat yourself on the back, because we're finished!

WeSplit: Wrap up

You've reached the end of your first SwiftUI app: good job! We've covered a lot of ground, but I've also tried to go nice and slowly to make sure it all sinks in – we've got lots more to cover in future projects, so taking a little extra time now is OK.

In this project you learn about the basic structure of SwiftUI apps, how to build forms and sections, creating navigation views and navigation bar titles, how to store program state with the `@State` and `@FocusState` property wrappers, how to create user interface controls like `TextField` and `Picker`, and how to create views in a loop using `ForEach`. Even better, you have a real project to show off for your efforts.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on:

1. Add a header to the third section, saying “Amount per person”
2. Add another section showing the total amount for the check – i.e., the original amount plus tip value, without dividing by the number of people.
3. Change the tip percentage picker to show a new screen rather than using a segmented control, and give it a wider range of options – everything from 0% to 100%. Tip: use the range `0..101` for your range rather than a fixed array.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Project 1: WeSplit

Solution to WeSplit. If you don't already subscribe, you can start a free trial today.

And if you found those easy, here's a tough one for you: rather than having to type `.currency(code: Locale.current.currencyCode ?? "USD")` in two places, can you make a new property to store the currency formatter? You'll need to give your property a specific return type in order to keep the rest of your code happy:

`FloatingPointFormatStyle<Double>.Currency`.

You can find that for yourself using Xcode's Quick Help window – open up the right-hand navigator, then select the Quick Help inspector, and finally click select the `.currency` code. You'll see Xcode displays `<Value>` rather than `<Double>`, because this thing is able to display other kinds of floating-point numbers too, but here we need `Double`.

Project 2

Guess the Flag

Build a game with stacks, images, and alerts

Guess the Flag: Introduction

In this second SwiftUI project we're going to be building a guessing game that helps users learn some of the many flags of the world.

This project is still going to be nice and easy, but gives me chance to introduce you to whole range of new SwiftUI functionality: stacks, buttons, images, alerts, asset catalogs, and more.

Our first app used a completely standard iOS look and feel, but here we're going to make something more customized so you can see how easy it is with SwiftUI.

You're going to need to download some files for this project, which you can do from GitHub: <https://github.com/twostraws/HackingWithSwift> – make sure you look in the SwiftUI section of the files.

Once you have those, go ahead and create a new App project in Xcode called GuessTheFlag, remembering to change your deployment target to iOS 15 or later. As before we're going to be starting with an overview of the various SwiftUI technologies required to build the app, so let's get into it...

Using stacks to arrange views

When we return **some View** for our body, SwiftUI expects to receive back some kind of view that can be displayed on the screen. That might be a navigation view, a form, a text view, a picker, or something else entirely, but it must conform to the **View** protocol so that it can be drawn on the screen.

If we want to return *multiple* things we have various options, but three are particularly useful. They are **HStack**, **VStack**, and **ZStack**, which handle horizontal, vertical, and, er, zdepth.

Let's try it out now. Our default template looks like this:

```
var body: some View {
    Text("Hello, world!")
        .padding()
}
```

That returns precisely one kind of view, which is a text view. If we wanted to return two text views, this kind of code just won't work the way you expect:

```
var body: some View {
    Text("Hello, world!")
    Text("This is another text view")
}
```

If you're using the SwiftUI canvas in Xcode, you'll now see *two* screens appear, because that's how SwiftUI interprets us sending back two independent text views.

We need to make sure SwiftUI gets exactly one kind of view back, and that's where stacks come in: they allow us to say "here are two text views, and I want them to be positioned like this..."

So, for **VStack** – a vertical stack of views – the two text views would be placed one above the other, like this:

Project 2: Guess the Flag

```
var body: some View {
    VStack {
        Text("Hello, world!")
        Text("This is inside a stack")
    }
}
```

By default **VStack** places some automatic amount of spacing between the two views, but we can control the spacing by providing a parameter when we create the stack, like this:

```
VStack(spacing: 20) {
    Text("Hello, world!")
    Text("This is inside a stack")
}
```

Just like SwiftUI's other views, **VStack** can have a maximum of 10 children – if you want to add more, you should wrap them inside a **Group**.

By default, **VStack** aligns its views so they are centered, but you can control that with its **alignment** property. For example, this aligns the text views to their leading edge, which in a left-to-right language such as English will cause them to be aligned to the left:

```
VStack(alignment: .leading) {
    Text("Hello, world!")
    Text("This is inside a stack")
}
```

Alongside **VStack** we have **HStack** for arranging things horizontally. This has the same syntax as **VStack**, including the ability to add spacing and alignment:

```
HStack(spacing: 20) {
    Text("Hello, world!")
    Text("This is inside a stack")
```

```
}
```

Vertical and horizontal stacks automatically fit their content, and prefer to align themselves to the center of the available space. If you want to change that you can use one or more **Spacer** views to push the contents of your stack to one side. These automatically take up all remaining space, so if you add one at the end a **VStack** it will push all your views to the top of the screen:

```
VStack {
    Text("First")
    Text("Second")
    Text("Third")
    Spacer()
}
```

If you add more than one spacer they will divide the available space between them. So, for example we could have one third of the space at the top and two thirds at the bottom, like this:

```
VStack {
    Spacer()
    Text("First")
    Text("Second")
    Text("Third")
    Spacer()
    Spacer()
}
```

We also have **ZStack** for arranging things by depth – it makes views that overlap. In the case of our two text views, this will make things rather hard to read:

```
ZStack {
    Text("Hello, world!")
    Text("This is inside a stack")
}
```

Project 2: Guess the Flag

ZStack doesn't have the concept of spacing because the views overlap, but it *does* have alignment. So, if you have one large thing and one small thing inside your **ZStack**, you can make both views align to the top like this: **ZStack(alignment: .top) {**

ZStack draws its contents from top to bottom, back to front. This means if you have an image then some text **ZStack** will draw them in that order, placing the text on top of the image.

Try placing several horizontal stacks inside a single vertical stack – can you make a 3x3 grid?

Colors and frames

SwiftUI gives us a range of functionality to render colors, and manages to be both simple and powerful – a difficult combination, but one they really pulled off.

To try this out, let's create a **ZStack** with a single text label:

```
ZStack {  
    Text("Your content")  
}
```

If we want to put something behind the text, we need to place it above it in the **ZStack**. But what if we wanted to put some red behind there – how would we do that?

One option is to use the **background()** modifier, which can be given a color to draw like this:

```
ZStack {  
    Text("Your content")  
}  
.background(.red)
```

That *might* have done what you expected, but there's a good chance it was a surprise: only the text view had a background color, even though we've asked the whole **ZStack** to have it.

In fact, there's no difference between that code and this:

```
ZStack {  
    Text("Your content")  
.background(.red)  
}
```

If you want to fill in red the whole area behind the text, you should place the color into the **ZStack** – treat it as a whole view, all by itself:

Project 2: Guess the Flag

```
ZStack {  
    Color.red  
    Text("Your content")  
}
```

In fact, **Color.red** is a view in its own right, which is why it can be used like shapes and text.

Tip: When we were using the **background()** modifier, SwiftUI was able to figure out that **.red** actually meant **Color.red**. When we're using the color as a free-standing view Swift has no context to help it figure out what **.red** means so we need to be specific that we mean **Color.red**.

Colors automatically take up all the space available, but you can also use the **frame()** modifier to ask for specific sizes. For example, we could ask for a 200x200 red square like this:

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

You can also specify minimum and maximum widths and heights, depending on the layout you want. For example, we could say we want a color that is no more than 200 points high, but for its width must be at least 200 points wide but can stretch to fill all the available width that's not used by other stuff:

```
Color.red  
.frame(minWidth: 200, maxWidth: .infinity, maxHeight: 200)
```

SwiftUI gives us a number of built-in colors to work with, such as **Color.blue**, **Color.green**, **Color.indigo**, and more. We also have some *semantic* colors: colors that don't say what hue they contain, but instead describe their purpose.

For example, **Color.primary** is the default color of text in SwiftUI, and will be either black or white depending on whether the user's device is running in light mode or dark mode. There's also **Color.secondary**, which is also black or white depending on the device, but now has

slight transparency so that a little of the color behind it shines through.

If you need something specific, you can create custom colors by passing in values between 0 and 1 for red, green, and blue, like this:

```
Color(red: 1, green: 0.8, blue: 0)
```

Even when taking up the full screen, you'll see that using **Color.red** will leave some space white.

How much space is white depends on your device, but on iPhones with Face ID – iPhone 13, for example – you'll find that both the status bar (the clock area at the top) and the home indicator (the horizontal stripe at the bottom) are left uncolored.

This space is left intentionally blank, because Apple doesn't want important content to get obscured by other UI features or by any rounded corners on your device. So, the remaining part – that whole middle space – is called *the safe area*, and you can draw into it freely without worrying that it might be clipped by the notch on an iPhone.

If you *want* your content to go under the safe area, you can use the **.ignoresSafeArea()** modifier to specify which screen edges you want to run up to, or specify nothing to automatically go edge to edge. For example, this creates a **ZStack** which fills the screen edge to edge with red then draws some text on top:

```
ZStack {
    Color.red
    Text("Your content")
}
.ignoresSafeArea()
```

It is *critically important* that no important content be placed outside the safe area, because it might be hard if not impossible for the user to see. Some views, such as **List**, allow content to scroll outside the safe area but then add extra insets so the user can scroll things into view.

Project 2: Guess the Flag

If your content is just decorative – like our background color here – then extending it outside the safe area is OK.

Before we're done, there's one more thing I want to mention: as well as using fixed colors such as `.red` and `.green`, the `background()` modifier can also accept *materials*. These apply a frosted glass effect over whatever comes below them, which allows us to create some beautiful depth effects.

To see this in action, we could build up our `ZStack` so that it has two colors inside a `VStack`, so they split the available space between them. Then, we'll attach a couple of modifiers to our text view so that it has a gray color, with an ultra thin material behind it:

```
ZStack {
    VStack(spacing: 0) {
        Color.red
        Color.blue
    }

    Text("Your content")
        .foregroundColor(.secondary)
        .padding(50)
        .background(.ultraThinMaterial)
    }
    .ignoresSafeArea()
```

That uses the thinnest material, which means we're letting a lot of the background colors shine through our frosted glass effect. iOS automatically adapts the effect based on whether the user has light or dark mode enabled – our material will either be light-colored or dark-colored, as appropriate.

There are other material thicknesses available depending on what effect you want, but there's something even neater I want to show to you. It's subtle, though, so I'd like you to click the tiny magnifying glass icon at the bottom of your SwiftUI preview so you can get a super close-

up look at the “Your content” text.

Right now you’ll see “Your content” is written in gray, because we’re using a secondary foreground color. However, SwiftUI gives us an alternative that provides a very slightly different effect: change the **foregroundColor()** modifier to **foregroundStyle()** – do you see the difference?

You should be able to see that the text is no longer just gray, but instead allows a little of the red and blue background colors to come through. It’s not a lot, just a hint, but when used effectively this provides a really beautiful effect to make sure text stands out regardless of the background behind it. iOS calls this effect *vibrancy*, and it’s used a lot throughout the system.

Gradients

SwiftUI gives us three kinds of gradients to work with, and like colors they are also views that can be drawn in our UI.

Gradients are made up of several components:

- An array of colors to show
- Size and direction information
- The type of gradient to use

For example, a linear gradient goes in one direction, so we provide it with a start and end point like this:

```
LinearGradient(gradient: Gradient(colors: [.white, .black]),  
startPoint: .top, endPoint: .bottom)
```

The inner **Gradient** type used there can also be provided with gradient stops, which let you specify both a color and how far along the gradient the color should be used. For example, we could specify that our gradient should be white from the start up to 45% of the available space, then black from 55% of the available space onwards:

```
LinearGradient(gradient: Gradient(stops: [  
    Gradient.Stop(color: .white, location: 0.45),  
    Gradient.Stop(color: .black, location: 0.55),  
]), startPoint: .top, endPoint: .bottom)
```

That will create a much sharper gradient – it will be compressed into a small space in the center.

Tip: Swift knows we're creating gradient stops here, so as a shortcut we can just write **.init** rather than **Gradient.Stop**, like this:

```
LinearGradient(gradient: Gradient(stops: [
```

```
.init(color: .white, location: 0.45),  
.init(color: .black, location: 0.55),  
]), startPoint: .top, endPoint: .bottom)
```

As an alternative, radial gradients move outward in a circle shape, so instead of specifying a direction we specify a start and end radius – how far from the center of the circle the color should start and stop changing. For example:

```
RadialGradient(gradient: Gradient(colors: [.blue, .black]),  
center: .center, startRadius: 20, endRadius: 200)
```

The last gradient type is called an angular gradient, although you might have heard it referred to elsewhere as a conic or conical gradient. This cycles colors around a circle rather than radiating outward, and can create some beautiful effects.

For example, this cycles through a range of colors in a single gradient, centered on the middle of the gradient:

```
AngularGradient(gradient: Gradient(colors:  
[.red, .yellow, .green, .blue, .purple, .red]),  
center: .center)
```

All of these gradient types can have stops provided rather than simple colors. Plus, they can also work as standalone views in your layouts, or be used as part of a modifier – you can use them as the background for a text view, for example.

Buttons and images

We've looked at SwiftUI's buttons briefly previously, but they are remarkably flexible and can adapt to a huge range of use cases.

The simplest way to make a button is one we've looked at previously: when it just contains some text you pass in the title of the button, along with a closure that should be run when the button is tapped:

```
Button("Delete selection") {  
    print("Now deleting...")  
}
```

Of course, that could be any function rather than just a closure, so this kind of thing is fine:

```
struct ContentView: View {  
    var body: some View {  
        Button("Delete selection", action: executeDelete)  
    }  
  
    func executeDelete() {  
        print("Now deleting...")  
    }  
}
```

There are few different ways we can customize the way buttons look. First, we can attach a *role* to the button, which iOS can use to adjust its appearance both visually and for screen readers. For example, we could say that our Delete button has a destructive role like this:

```
Button("Delete selection", role: .destructive, action:  
executeDelete)
```

Second, we can use one of the built-in styles for buttons: **.bordered** and **.borderedProminent**.

These can be used by themselves, or in combination with a role:

```
 VStack {
    Button("Button 1") { }
        .buttonStyle(.bordered)
    Button("Button 2", role: .destructive) { }
        .buttonStyle(.bordered)
    Button("Button 3") { }
        .buttonStyle(.borderedProminent)
    Button("Button 4", role: .destructive) { }
        .buttonStyle(.borderedProminent)
}
```

If you want to customize the colors used for a bordered button, use the `tint()` modifier like this:

```
 Button("Button 3") { }
    .buttonStyle(.borderedProminent)
    .tint(.mint)
```

Important: Apple explicitly recommends against using too many prominent buttons, because when everything is prominent nothing is.

If you want something completely custom, you can pass a custom label using a second trailing closure:

```
 Button {
    print("Button was tapped")
} label: {
    Text("Tap me!")
    .padding()
    .foregroundColor(.white)
    .background(.red)
}
```

Project 2: Guess the Flag

This is particularly common when you want to incorporate images into your buttons.

SwiftUI has a dedicated **Image** type for handling pictures in your apps, and there are three main ways you will create them:

- **Image("pencil")** will load an image called “Pencil” that you have added to your project.
- **Image(decorative: "pencil")** will load the same image, but won’t read it out for users who have enabled the screen reader. This is useful for images that don’t convey additional important information.
- **Image(systemName: "pencil")** will load the pencil icon that is built into iOS. This uses Apple’s SF Symbols icon collection, and you can search for icons you like – download Apple’s free SF Symbols app from the web to see the full set.

By default the screen reader will read your image name if it is enabled, so make sure you give your images clear names if you want to avoid confusing the user. Or, if they don’t actually add information that isn’t already elsewhere on the screen, use the **Image(decorative:)** initializer.

Because the longer form of buttons can have any kind of views inside them, you can use images like this:

```
Button {  
    print("Edit button was tapped")  
} label: {  
    Image(systemName: "pencil")  
}
```

If you want both text *and* image at the same time, SwiftUI has a dedicated type called **Label**.

```
Button {  
    print("Edit button was tapped")  
} label: {  
    Label("Edit", systemImage: "pencil")  
}
```

```
}
```

That will show both a pencil icon and the word “Edit” side by side, which on the surface sounds exactly the same as what we’d get by using a simple **HStack**. However, SwiftUI is really smart: when we use a label it will automatically decide whether to show the icon, the text, or both depending on how they are being used in our layout. This makes **Label** a fantastic choice in many situations, as you’ll see.

Tip: If you find that your images have become filled in with a color, for example showing as solid blue rather than your actual picture, this is probably SwiftUI coloring them to show that they are tappable. To fix the problem, use the **renderingMode(.original)** modifier to force SwiftUI to show the original image rather than the recolored version.

Showing alert messages

If something important happens, a common way of notifying the user is using an alert – a pop up window that contains a title, message, and one or two buttons depending on what you need.

But think about it: *when* should an alert be shown and *how*? Views are a function of our program state, and alerts aren't an exception to that. So, rather than saying “show the alert”, we instead create our alert and set the conditions under which it should be shown.

A basic SwiftUI alert has a title and a button that dismisses it, but the more interesting part is how we present that alert: we don't assign the alert to a variable then write something like **myAlert.show()**, because that would be back to the old “series of events” way of thinking.

Instead, we create some state that tracks whether our alert is showing, like this:

```
@State private var showingAlert = false
```

We then attach our alert somewhere to our user interface, telling it to use that state to determine whether the alert is presented or not. SwiftUI will watch **showingAlert**, and as soon as it becomes true it will show the alert.

Putting that all together, here's some example code that shows an alert when a button is tapped:

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

    var body: some View {
        Button("Show Alert") {
            showingAlert = true
        }
        .alert("Important message", isPresented: $showingAlert) {
            Button("OK") { }
        }
    }
}
```

```

    }
}
}
```

That attaches the alert to the button, but honestly it doesn't matter where the `alert()` modifier is used – all we're doing is saying that an alert exists and is shown when `showingAlert` is true.

Take a close look at the `alert()` modifier:

```
alert("Important message", isPresented: $showingAlert)
```

The first part is the alert title, which is straightforward enough, but there's also another two-way data binding because SwiftUI will automatically set `showingAlert` back to false when the alert is dismissed.

Now look at the button:

```
Button("OK") { }
```

That's an empty closure, meaning that we aren't assigning any functionality to run when the button is pressed. That doesn't matter, though, because *any* button inside an alert will automatically dismiss the alert – that closure is there to let us add any extra functionality beyond just dismissing the alert.

You can add more buttons to your alert, and this is a particularly good place to add roles to make sure it's clear what each button does:

```
.alert("Important message", isPresented: $showingAlert) {
    Button("Delete", role: .destructive) { }
    Button("Cancel", role: .cancel) { }
}
```

And finally, you can add message text to go alongside your title with a second trailing closure, like this:

Project 2: Guess the Flag

```
Button("Show Alert") {
    showingAlert = true
}
.alert("Important message", isPresented: $showingAlert) {
    Button("OK", role: .cancel) { }
} message: {
    Text("Please read this.")
}
```

This is the final part of the overview for this project, so it's almost time to get started with the real code. If you want to save the examples you've programmed you should copy your project directory somewhere else.

When you're ready, put ContentView.swift back to the way it started when you first made the project, so we have a clean slate to work from.

Stacking up buttons

We're going to start our app by building the basic UI structure, which will be two labels telling the user what to do, then three image buttons showing three world flags.

First, find the assets for this project and drag them into your asset catalog. That means opening Assets.xcassets in Xcode, then dragging in the flag images from the project2-files folder. You'll notice that the images are named after their country, along with either @2x or @3x – these are images at double resolution and triple resolution to handle different types of iPhone screen.

Next, we need two properties to store our game data: an array of all the country images we want to show in the game, plus an integer storing which country image is correct.

```
var countries = [ "Estonia", "France", "Germany", "Ireland",
  "Italy", "Nigeria", "Poland", "Russia", "Spain", "UK", "US"]
var correctAnswer = Int.random(in: 0...2)
```

The **Int.random(in:)** method automatically picks a random number, which is perfect here – we'll be using that to decide which country flag should be tapped.

Inside our body, we need to lay out our game prompt in a vertical stack, so let's start with that:

```
var body: some View {
    VStack {
        Text("Tap the flag of")
        Text(countries[correctAnswer])
    }
}
```

Below there we want to have our tappable flag buttons, and while we *could* just add them to the same **VStack** we can actually create a *second* **VStack** so that we have more control over the spacing.

Project 2: Guess the Flag

The **VStack** we just created above holds two text views and has no spacing, but the flags are going to have 30 points of spacing between them so it looks better.

So, start by adding this **ForEach** loop directly below the end of the **VStack** we just created:

```
ForEach(0..<3) { number in
    Button {
        // flag was tapped
    } label: {
        Image(countries[number])
            .renderingMode(.original)
    }
}
```

The **renderingMode(.original)** modifier tells SwiftUI to render the original image pixels rather than trying to recolor them as a button.

And now we have a problem: our **body** property is trying to send back two views, a **VStack** and a **ForEach**, but that won't work correctly. This is where our second **VStack** will come in: I'd like you to wrap the original **VStack** and the **ForEach** below in a new **VStack**, this time with a spacing of 30 points.

So your code should look like this:

```
var body: some View {
    VStack(spacing: 30) {
        VStack {
            Text("Tap the flag of")
            // etc
        }

        ForEach(0..<3) { number in
            // etc
        }
    }
}
```

```

    }
}
}
```

Having two vertical stacks like this allows us to position things more precisely: the outer stack will space its views out by 30 points each, whereas the inner stack has no spacing.

That's enough to give you a basic idea of our user interface, and already you'll see it doesn't look great – some flags have white in them, which blends into the background, and all the flags are centered vertically on the screen.

We'll come back to polish the UI later, but for now let's put in a blue background color to make the flags easier to see. Because this means putting something behind our outer **VStack**, we need to use a **ZStack** as well. Yes, we'll have a **VStack** inside another **VStack** inside a **ZStack**, and that is perfectly normal.

Start by putting a **ZStack** around your outer **VStack**, like this:

```
var body: some View {
    ZStack {
        // previous VStack code
    }
}
```

Now put this just inside the **ZStack**, so it goes behind the outer **VStack**:

```
Color.blue
    .ignoresSafeArea()
```

That **.ignoresSafeArea()** modifier ensures the color goes right to the edge of the screen.

Now that we have a darker background color, we should give the text something brighter so that it stands out better:

```
Text("Tap the flag of")
```

Project 2: Guess the Flag

```
.foregroundColor(.white)  
  
Text(countries[correctAnswer])  
.foregroundColor(.white)
```

This design is not going to set the world alight, but it's a solid start!

Showing the player's score with an alert

In order for this game to be fun, we need to randomize the order in which flags are shown, trigger an alert telling them whether they were right or wrong whenever they tap a flag, then reshuffle the flags.

We already set **correctAnswer** to a random integer, but the flags always start in the *same* order. To fix that we need to shuffle the **countries** array when the game starts, so modify the property to this:

```
var countries = [ "Estonia", "France", "Germany", "Ireland",
  "Italy", "Nigeria", "Poland", "Russia", "Spain", "UK",
  "US" ].shuffled()
```

As you can see, the **shuffled()** method automatically takes care of randomizing the array order for us.

Now for the more interesting part: when a flag has been tapped, what should we do? We need to replace the // **flag was tapped** comment with some code that determines whether they tapped the correct flag or not, and the best way of doing *that* is with a new method that accepts the integer of the button and checks whether that matches our **correctAnswer** property.

Regardless of whether they were correct, we want to show the user an alert saying what happened so they can track their progress. So, add this property to store whether the alert is showing or not:

```
@State private var showingScore = false
```

And add this property to store the title that will be shown inside the alert:

```
@State private var scoreTitle = ""
```

Project 2: Guess the Flag

So, whatever method we write will accept the number of the button that was tapped, compare that against the correct answer, then set those two new properties so we can show a meaningful alert.

Add this directly after the **body** property:

```
func flagTapped(_ number: Int) {
    if number == correctAnswer {
        scoreTitle = "Correct"
    } else {
        scoreTitle = "Wrong"
    }

    showingScore = true
}
```

We can now call that by replacing the **// flag was tapped** comment with this:

```
flagTapped(number)
```

We already have **number** because it's given to us by **ForEach**, so it's just a matter of passing that on to **flagTapped()**.

Before we show the alert, we need to think about what happens when the alert is dismissed. Obviously the game shouldn't be over, otherwise the whole thing would be over immediately.

Instead we're going to write an **askQuestion()** method that resets the game by shuffling up the countries and picking a new correct answer:

```
func askQuestion() {
    countries.shuffle()
    correctAnswer = Int.random(in: 0...2)
}
```

Showing the player's score with an alert

That code won't compile, and hopefully you'll see why pretty quickly: we're trying to change properties of our view that haven't been marked with `@State`, which isn't allowed. So, go to where `countries` and `correctAnswer` are declared, and put `@State private` before them, like this:

```
@State private var countries = ["Estonia", "France", "Germany",
"Ireland", "Italy", "Nigeria", "Poland", "Russia", "Spain",
"UK", "US"].shuffled()
@State private var correctAnswer = Int.random(in: 0...2)
```

And now we're ready to show the alert. This needs to:

1. Use the `alert()` modifier so the alert gets presented when `showingScore` is true.
2. Show the title we set in `scoreTitle`.
3. Have a dismiss button that calls `askQuestion()` when tapped.

So, put this at the end of the `ZStack` in the `body` property:

```
.alert(scoreTitle, isPresented: $showingScore) {
    Button("Continue", action: askQuestion)
} message: {
    Text("Your score is ???")
}
```

Yes, there are three question marks that should hold a score value – you'll be completing that part soon!

Styling our flags

Our game now works, although it doesn't look great. Fortunately, we can make a few small tweaks to our design to make the whole thing look better.

First, let's replace the solid blue background color with a linear gradient from blue to black, which ensures that even if a flag has a similar blue stripe it will still stand out against the background.

So, find this line:

```
Color.blue  
.ignoresSafeArea()
```

And replace it with this:

```
LinearGradient(gradient: Gradient(colors: [.blue, .black]),  
startPoint: .top, endPoint: .bottom)  
.ignoresSafeArea()
```

It still ignores the safe area, ensuring that the background goes edge to edge.

Now let's adjust the fonts we're using just a little, so that the country name – the part they need to guess – is the most prominent piece of text on the screen, while the “Tap the flag of” text is smaller and bold.

We can control the size and style of text using the **font()** modifier, which lets us select from one of the built-in font sizes on iOS. As for adjusting the *weight* of fonts – whether we want super-thin text, slightly bold text, etc – we can get fine-grained control over that by adding a **weight()** modifier to whatever font we ask for.

Let's use both of these here, so you can see them in action. Add this directly after the “Tap the flag of” text:

```
.font(.subheadline.weight(.heavy))
```

And put this modifiers directly after the `Text(countries[correctAnswer])` view:

```
.font(.largeTitle.weight(.semibold))
```

“Large title” is the largest built-in font size iOS offers us, and automatically scales up or down depending on what setting the user has for their fonts – a feature known as *Dynamic Type*. We’re overriding the weight of the font so it’s a little bolder, but it will still scale up or down as needed.

Finally, let’s jazz up those flag images a little. SwiftUI gives us a number of modifiers to affect the way views are presented, and we’re going to use two here: one to change the shape of flags, and one to add a shadow.

There are four built-in shapes in Swift: rectangle, rounded rectangle, circle, and capsule. We’ll be using capsule here: it ensures the corners of the shortest edges are fully rounded, while the longest edges remain straight – it looks great for buttons. Making our image capsule shaped is as easy as adding the `.clipShape(Capsule())` modifier, like this:

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

And finally we want to apply a shadow effect around each flag, making them really stand out from the background. This is done using `shadow()`, which takes the color, radius, X, and Y offset of the shadow, but if you skip the color we get a translucent black, and if we skip X and Y it assumes 0 for them – all sensible defaults.

So, add this last modifier below the previous two:

```
.shadow(radius: 5)
```

So, our finished flag image looks like this:

```
Image(countries[number])
```

Project 2: Guess the Flag

```
.renderingMode(.original)  
.clipShape(Capsule())  
.shadow(radius: 5)
```

SwiftUI has so many modifiers that help us adjust the way fonts and images are rendered. They all do exactly one thing, so it's common to stack them up as you can see above.

Upgrading our design

At this point we've built the app and it works well, but with all the SwiftUI skills you've learned so far we can actually take what we've built and re-skin it – produce a different UI for the project we've currently built. This won't affect the logic at all; we're just trying out some different UI to see what you can do with your current knowledge.

Experimenting with designs like this is a lot of fun, but I do want to add one word of caution: at the very least, make sure you run your code on all sizes of iOS device, from the tiny iPod touch up to an iPhone 13 Pro Max. Finding something that works well on that wide range of screen sizes takes some thinking!

Let's start off with the blue-black gradient we have behind our flags. It was okay to get us going, but now I want to try something a little fancier: a radial gradient with custom stops.

Previously I showed you how we can use very precise gradient stop locations to adjust the way our gradient is drawn. Well, if we create two stops that are *identical* to each other then the gradient goes away entirely – the color just switches from one to the other directly. Let's try it out with our current design:

```
RadialGradient(stops: [
    .init(color: .blue, location: 0.3),
    .init(color: .red, location: 0.3),
], center: .top, startRadius: 200, endRadius: 700)
    .ignoresSafeArea()
```

That's an interesting effect, I think – like we have a blue circle overlaid on top of a red background. That said, it's also *ugly*: those red and blue colors together are much too bright.

So, we can send in toned-down versions of those same colors to get something looking more harmonious – shades that are much more common in flags:

```
RadialGradient(stops: [
    .init(color: Color(red: 0.1, green: 0.2, blue: 0.45),
```

Project 2: Guess the Flag

```
location: 0.3),  
    .init(color: Color(red: 0.76, green: 0.15, blue: 0.26),  
location: 0.3),  
], center: .top, startRadius: 200, endRadius: 400)  
.ignoresSafeArea()
```

Next, right now we have a **VStack** with spacing of 30 to place the question area and the flags, but I'd like to reduce that down to 15:

```
VStack(spacing: 15) {
```

Why? Well, because we're going to make that whole area into a visual element in our UI, making it a colored rounded rectangle so that part of the game stands out on the screen.

To do that, add these modifiers to the end of the same **VStack**:

```
.frame(maxWidth: .infinity)  
.padding(.vertical, 20)  
.background(.regularMaterial)  
.clipShape(RoundedRectangle(cornerRadius: 20))
```

That lets it resize to take up all the horizontal space it needs, adds a little vertical padding, applies a background material so that it stands out from the red-blue gradient the background, and finally clips the whole thing into the shape of a rounded rectangle.

I think that's already looking a lot better, but let's keep pressing on!

Our next step is to add a title before our main box, and a score placeholder after. This means *another* **VStack** around what we have so far, because the existing **VStack(spacing: 15)** we have is where we apply the material effect.

So, wrap your current **VStack** in a new one with a title at the top, like this:

```
VStack {
```

```

Text("Guess the Flag")
    .font(.largeTitle.weight(.bold))
    .foregroundColor(.white)

// current VStack(spacing: 15) code
}

```

Tip: Asking for bold fonts is so common there's actually a small shortcut: `.font(.largeTitle.bold())`.

That adds a new title at the top, but we can also slot in a score label at the bottom of that new **VStack**, like this:

```

Text("Score: ???")
    .foregroundColor(.white)
    .font(.title.bold())

```

Both the “Guess the Flag” title and score label look great with white text, but the text inside our box *doesn't* – we made it white because it was sitting on top of a dark background originally, but now it's really hard to read.

To fix this, we can delete the **foregroundColor()** modifier for `Text(countries[correctAnswer])` so that it defaults to using the primary color for the system – black in light mode, and white in dark mode.

As for the white “Tap the flag of”, we can have that use the iOS vibrancy effect to let a little of the background color shine through. Change its **foregroundColor()** modifier to this:

```
.foregroundStyle(.secondary)
```

At this point our UI more or less works, but I think it's a little too squished up – if you're on a larger device you'll see the content all sits in the center of the screen with lots of space above and below, and the white box in the middle runs right to the edges of the screen.

Project 2: Guess the Flag

To fix this we're going to do two things: add a little padding to our outermost **VStack**, then add some **Spacer()** views to force the UI elements apart. On larger devices these spacers will split up the available space between them, but on small devices they'll practically disappear – it's a great way to make our UI work well on all screen sizes.

There are four spacers I'd like you to add:

- One directly before the “Guess the Flag” title.
- *Two* (yes, two) directly before the “Score: ????” text.
- And one directly *after* the “Score: ????” text.

Remember, when you have multiple spacers like this they will automatically divide the available space equally – having two spacers together will make them take up twice as much space as a single spacer.

And now all that remains is to add a little padding around the outermost **VStack**, with this:

```
.padding()
```

And that's our refreshed design complete! Having all those spacers means that on small devices such as the iPod touch, while also scaling up smoothly to look good even on Pro Max iPhones.

However, this is only *one* possible design for our app – maybe you prefer the old design over this one, or maybe you want to try something else. The point is, you've seen how even with the handful of SwiftUI skills you already have it's possible to build very different designs, and if you have the time I would encourage you to have a play around and see where you end up!

Guess the Flag: Wrap up

That's another SwiftUI app completed, including lots of important new techniques. You'll use **VStack**, **HStack**, and **ZStack** in almost every project you make, and you'll see find you can quickly build complex layouts by combining them together.

Many people find SwiftUI's way of showing alerts a little odd at first: creating it, adding a condition, then simply triggering that condition at some point in the future seems like a lot more work than just asking the alert to show itself. But like I said, it's important that our views always be a reflection of our program state, and that rules out us just showing alerts whenever we want to.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on:

1. Add an **@State** property to store the user's score, modify it when they get an answer right or wrong, then display it in the alert and in the score label.
2. When someone chooses the wrong flag, tell them their mistake in your alert message – something like “Wrong! That’s the flag of France,” for example.
3. Make the game show only 8 questions, at which point they see a final alert judging their score and can restart the game.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Project 2: Guess the Flag

Solution to Guess the Flag. If you don't already subscribe, you can start a free trial today.

Note: That last one takes a little more thinking than the others. A good place to start would be to add a second **alert()** modifier watching a different Boolean property, then connect its button to a **reset()** method to set the game back to its initial state.

Project 3

Views and Modifiers

Dive deep into Swift's rendering system

Views and modifiers: Introduction

This third SwiftUI project is actually our first *technique project* – a change in pace as we explore certain SwiftUI features in depth, looking at how they work in detail along with *why* they work that way.

In this technique project we’re going to take a close look at views and view modifiers, and hopefully answer some of the most common questions folks have at this point – why does SwiftUI use structs for its views? Why does it use **some View** so much? *How do modifiers really work?* My hope is that by the end of this project you’ll have a thorough understanding of what makes SwiftUI tick.

As with the other days it’s a good idea to work in an Xcode project so you can see your code in action, so please create a new App project called `ViewsAndModifiers`, making sure to mark it as using an iOS 15.0 deployment target.

Why does SwiftUI use structs for views?

If you ever programmed for UIKit or AppKit (Apple's original user interface frameworks for iOS and macOS) you'll know that they use *classes* for views rather than structs. SwiftUI does not: we prefer to use structs for views across the board, and there are a couple of reasons why.

First, there is an element of performance: structs are simpler and faster than classes. I say *an element* of performance because lots of people think this is the primary reason SwiftUI uses structs, when really it's just one part of the bigger picture.

In UIKit, every view descended from a class called **UIView** that had many properties and methods – a background color, constraints that determined how it was positioned, a layer for rendering its contents into, and more. There were *lots* of these, and every **UIView** and **UIView** subclass had to have them, because that's how inheritance works.

In SwiftUI, all our views are trivial structs and are almost free to create. Think about it: if you make a struct that holds a single integer, the entire size of your struct is... that one integer. Nothing else. No surprise extra values inherited from parent classes, or grandparent classes, or great-grandparent classes, etc – they contain exactly what you can see and nothing more.

Thanks to the power of modern iPhones, I wouldn't think twice about creating 1000 integers or even 100,000 integers – it would happen in the blink of an eye. The same is true of 1000 SwiftUI views or even 100,000 SwiftUI views; they are so fast it stops being worth thinking about.

However, even though performance is important there's something much more important about views as structs: it forces us to think about isolating state in a clean way. You see, classes are able to change their values freely, which can lead to messier code – how would SwiftUI be able to know when a value changed in order to update the UI?

By producing views that don't mutate over time, SwiftUI encourages us to move to a more functional design approach: our views become simple, inert things that convert data into UI,

Project 3: Views and Modifiers

rather than intelligent things that can grow out of control.

You can see this in action when you look at the kinds of things that can be a view. We already used **Color.red** and **LinearGradient** as views – trivial types that hold very little data. In fact, you can't get a great deal simpler than using **Color.red** as a view: it holds no information other than “fill my space with red”.

In comparison, Apple's **documentation for UIView** lists about 200 properties and methods that **UIView** has, all of which get passed on to its subclasses whether they need them or not.

Tip: If you use a class for your view you might find your code either doesn't compile or crashes at runtime. Trust me on this: use a struct.

What is behind the main SwiftUI view?

When you're just starting out with SwiftUI, you get this code:

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, world!")  
            .padding()  
    }  
}
```

It's common to then modify that text view with a background color and expect it to fill the screen:

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, world!")  
            .padding()  
            .background(.red)  
    }  
}
```

However, that doesn't happen. Instead, we get a small red text view in the center of the screen, and a sea of white beyond it.

This confuses people, and usually leads to the question – “how do I make what's behind the view turn red?”

Let me say this as clearly as I can: **for SwiftUI developers, there is nothing behind our view.** You shouldn't try to make that white space turn red with weird hacks or workarounds, and you certainly shouldn't try to reach outside of SwiftUI to do it.

Project 3: Views and Modifiers

Now, right now at least there is something behind our content view called a **UIHostingController**: it is the bridge between UIKit (Apple's original iOS UI framework) and SwiftUI. However, if you start trying to modify that you'll find that your code no longer works on Apple's other platforms, and in fact might stop working entirely on iOS at some point in the future.

Instead, you should try to get into the mindset that there is nothing behind our view – that what you see is all we have.

Once you're in that mindset, the correct solution is to make the text view take up more space; to allow it to fill the screen rather than being sized precisely around its content. We can do that by using the **frame()** modifier, passing in **.infinity** for both its maximum width and maximum height.

```
Text("Hello, world!")
    .frame(maxWidth: .infinity, maxHeight: .infinity)
    .background(.red)
```

Using **maxWidth** and **maxHeight** is different from using **width** and **height** – we're not saying the text view *must* take up all that space, only that it *can*. If you have other views around, SwiftUI will make sure they all get enough space.

Why modifier order matters

Whenever we apply a modifier to a SwiftUI view, we actually create a new view with that change applied – we don't just modify the existing view in place. If you think about it, this behavior makes sense: our views only hold the exact properties we give them, so if we set the background color or font size there is no place to store that data.

We're going to look at *why* this happens shortly, but first I want to look at the practical implications of this behavior. Take a look at this code:

```
Button("Hello, world!") {  
    // do nothing  
}  
.background(.red)  
.frame(width: 200, height: 200)
```

What do you think that will look like when it runs?

Chances are you guessed wrong: you *won't* see a 200x200 red button with "Hello, world!" in the middle. Instead, you'll see a 200x200 empty square, with "Hello, world!" in the middle and with a red rectangle directly around "Hello, world!".

You can understand what's happening here if you think about the way modifiers work: each one creates a new struct with that modifier applied, rather than just setting a property on the view.

You can peek into the underbelly of SwiftUI by asking for the type of our view's body. Modify the button to this:

```
Button("Hello, world!") {  
    print(type(of: self.body))  
}  
.background(.red)  
.frame(width: 200, height: 200)
```

Project 3: Views and Modifiers

Swift's **type(of:)** method prints the exact type of a particular value, and in this instance it will print the following: **ModifiedContent<ModifiedContent<Button<Text>, _BackgroundStyleModifier<Color>>, _FrameLayout>**

You can see two things here:

- Every time we modify a view SwiftUI applies that modifier by using generics: **ModifiedContent<OurThing, OurModifier>**.
- When we apply multiple modifiers, they just stack up:
ModifiedContent<ModifiedContent<...>, _FrameLayout>

To read what the type is, start from the innermost type and work your way out:

- The innermost type is **ModifiedContent<Button<Text>, _BackgroundStyleModifier<Color>**: our button has some text with a background color applied.
- Around that we have **ModifiedContent<..., _FrameLayout>**, which takes our first view (button + background color) and gives it a larger frame.

As you can see, we end with **ModifiedContent** types stacking up – each one takes a view to transform plus the actual change to make, rather than modifying the view directly.

What this means is that the order of your modifiers matter. If we rewrite our code to apply the background color *after* the frame, then you might get the result you expected:

```
Button("Hello, world!") {
    print(type(of: self.body))
}
.frame(width: 200, height: 200)
.background(.red)
```

The best way to think about it for now is to imagine that SwiftUI renders your view after every

single modifier. So, as soon as you say **.background(.red)** it colors the background in red, regardless of what frame you give it. If you then later expand the frame, it won't magically redraw the background – that was already applied.

Of course, this isn't *actually* how SwiftUI works, because if it did it would be a performance nightmare, but it's a neat mental shortcut to use while you're learning.

An important side effect of using modifiers is that we can apply the same effect multiple times: each one simply adds to whatever was there before.

For example, SwiftUI gives us the **padding()** modifier, which adds a little space around a view so that it doesn't push up against other views or the edge of the screen. If we apply padding then a background color, then more padding and a different background color, we can give a view multiple borders, like this:

```
Text("Hello, world!")
    .padding()
    .background(.red)
    .padding()
    .background(.blue)
    .padding()
    .background(.green)
    .padding()
    .background(.yellow)
```

Why does SwiftUI use “some View” for its view type?

SwiftUI relies very heavily on a Swift power feature called “opaque return types”, which you can see in action every time you write **some View**. This means “one object that conforms to the **View** protocol, but we don’t want to say what.”

Returning **some View** means even though we don’t know what view type is going back, the compiler does. That might sound small, but it has important implications.

First, using **some View** is important for performance: SwiftUI needs to be able to look at the views we are showing and understand how they change, so it can correctly update the user interface. If SwiftUI didn’t have this extra information, it would be really slow for SwiftUI to figure out exactly what changed – it would pretty much need to ditch everything and start again after every small change.

The second difference is important because of the way SwiftUI builds up its data using **ModifiedContent**. Previously I showed you this code:

```
Button("Hello World") {  
    print(type(of: self.body))  
}  
.frame(width: 200, height: 200)  
.background(.red)
```

That creates a simple button then makes it print its exact Swift type, and gives some long output with a couple of instances of **ModifiedContent**.

The **View** protocol has an associated type attached to it, which is Swift’s way of saying that **View** by itself doesn’t mean anything – we need to say exactly what kind of view it is. It effectively has a hole in it, in a similar way to how Swift doesn’t let us say “this variable is an array” and instead requires that we say what’s *in* the array: “this variable is a string array.”

Why does SwiftUI use “some View” for its view type?

So, while it’s not allowed to write a view like this:

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

It is perfectly legal to write a view like this:

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

Returning **View** makes no sense, because Swift wants to know what’s inside the view – it has a big hole that must be filled. On the other hand, returning **Text** is fine, because we’ve filled the hole; Swift knows what the view is.

Now let’s return to our code from earlier:

```
Button("Hello World") {  
    print(type(of: self.body))  
}  
.frame(width: 200, height: 200)  
.background(.red)
```

If we want to return one of those from our **body** property, what should we write? While you could try to figure out the exact combination of **ModifiedContent** structs to use, it’s hideously painful and the simple truth is that we don’t care because it’s all internal SwiftUI stuff.

What **some View** lets us do is say “this will a view, such as **Button** or **Text**, but I don’t want to say what.” So, the hole that **View** has will be filled by a real view object, but we aren’t

Project 3: Views and Modifiers

required to write out the exact long type.

There are two places where it gets a bit more complicated:

1. How does **VStack** work – it conforms to the **View** protocol, but how does it fill the “what kind of content does it have?” hole if it can contain lots of different things inside it?
2. What happens if we send back two views directly from our **body** property, without wrapping them in a stack?

To answer the first question first, if you create a **VStack** with two text views inside, SwiftUI silently creates a **TupleView** to contain those two views – a special type of view that holds exactly two views inside it. So, the **VStack** fills the “what kind of view is this?” with the answer “it’s a **TupleView** containing two text views.”

And what if you have three text views inside the **VStack**? Then it’s a **TupleView** containing three views. Or four views. Or eight views, or even ten views – there is literally a version of **TupleView** that tracks ten different kinds of content:

```
TupleView<(C0, C1, C2, C3, C4, C5, C6, C7, C8, C9)>
```

And that’s why SwiftUI doesn’t allow more than 10 views inside a parent: they wrote versions of **TupleView** that handle 2 views through 10, but no more.

As for the second question, Swift silently applies a special attribute to the **body** property called **@ViewBuilder**. This has the effect of silently wrapping multiple views in one of those **TupleView** containers, so that even though it looks like we’re sending back multiple views they get combined into one **TupleView**.

This behavior isn’t magic: if you right-click on the **View** protocol and choose “Jump to Definition”, you’ll see the requirement for the **body** property and also see that it’s marked with the **@ViewBuilder** attribute:

```
@ViewBuilder var body: Self.Body { get }
```

Why does SwiftUI use “some View” for its view type?

Of course, how SwiftUI interprets multiple views going back without a stack around them isn’t specifically defined anywhere, but as you’ll learn later on that’s actually helpful.

Conditional modifiers

It's common to want modifiers that apply only when a certain condition is met, and in SwiftUI the easiest way to do that is with the ternary conditional operator.

As a reminder, to use the ternary operator you write your condition first, then a question mark and what should be used if the condition is true, then a colon followed by what should be used if the condition is false. If you forget this order a lot, remember **Scott Michaud's helpful mnemonic**: What do you want to check, True, False, or "WTF" for short.

For example, if you had a property that could be either true or false, you could use that to control the foreground color of a button like this:

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

    var body: some View {
        Button("Hello World") {
            // flip the Boolean between true and false
            useRedText.toggle()
        }
        .foregroundColor(useRedText ? .red : .blue)
    }
}
```

So, when **useRedText** is true the modifier effectively reads **.foregroundColor(.red)**, and when it's false the modifier becomes **.foregroundColor(.blue)**. Because SwiftUI watches for changes in our **@State** properties and re-invokes our **body** property, whenever that property changes the color will immediately update.

You can often use regular **if** conditions to return different views based on some state, but this actually creates more work for SwiftUI – rather than seeing one **Button** being used with different colors, it now sees two different **Button** views, and when we flip the Boolean

condition it will destroy one to create the other rather than just recolor what it has.

So, this kind of code might *look* the same, but it's actually less efficient:

```
var body: some View {
    if useRedText {
        Button("Hello World") {
            useRedText.toggle()
        }
        .foregroundColor(.red)
    } else {
        Button("Hello World") {
            useRedText.toggle()
        }
        .foregroundColor(.blue)
    }
}
```

Sometimes using **if** statements are unavoidable, but where possible prefer to use the ternary operator instead.

Environment modifiers

Many modifiers can be applied to containers, which allows us to apply the same modifier to many views at the same time.

For example, if we have four text views in a **VStack** and want to give them all the same font modifier, we could apply the modifier to the **VStack** directly and have that change apply to all four text views:

```
VStack {  
    Text("Gryffindor")  
    Text("Hufflepuff")  
    Text("Ravenclaw")  
    Text("Slytherin")  
}  
.font(.title)
```

This is called an environment modifier, and is different from a regular modifier that is applied to a view.

From a coding perspective these modifiers are used exactly the same way as regular modifiers. However, they behave subtly differently because if any of those child views override the same modifier, the child's version takes priority.

As an example, this shows our four text views with the title font, but one has a large title:

```
VStack {  
    Text("Gryffindor")  
    .font(.largeTitle)  
    Text("Hufflepuff")  
    Text("Ravenclaw")  
    Text("Slytherin")  
}  
.font(.title)
```

There, **font()** is an environment modifier, which means the Gryffindor text view can override it with a custom font.

However, this applies a blur effect to the **VStack** then attempts to disable blurring on one of the text views:

```
 VStack {  
     Text("Gryffindor")  
     .blur(radius: 0)  
     Text("Hufflepuff")  
     Text("Ravenclaw")  
     Text("Slytherin")  
 }  
 .blur(radius: 5)
```

That won't work the same way: **blur()** is a regular modifier, so any blurs applied to child views are *added* to the **VStack** blur rather than replacing it.

To the best of my knowledge there is no way of knowing ahead of time which modifiers are environment modifiers and which are regular modifiers other than reading the individual documentation for each modifier and hope it's mentioned. Still, I'd rather have them than not: being able to apply one modifier everywhere is much better than copying and pasting the same thing into multiple places.

Views as properties

There are lots of ways to make it easier to use complex view hierarchies in SwiftUI, and one option is to use properties – to create a view as a property of your own view, then use that property inside your layouts.

For example, we could create two text views like this as properties, then use them inside a `VStack`:

```
struct ContentView: View {
    let motto1 = Text("Draco dormiens")
    let motto2 = Text("nunquam titillandus")

    var body: some View {
        VStack {
            motto1
            motto2
        }
    }
}
```

You can even apply modifiers directly to those properties as they are being used, like this:

```
VStack {
    motto1
        .foregroundColor(.red)
    motto2
        .foregroundColor(.blue)
}
```

Creating views as properties can be helpful to keep your `body` code clearer – not only does it help avoid repetition, but it can also get more complex code out of the `body` property.

Swift doesn't let us create one stored property that refers to other stored properties, because it would cause problems when the object is created. This means trying to create a **TextField** bound to a local property will cause problems.

However, you can create *computed* properties if you want, like this:

```
var motto1: some View {
    Text("Draco dormiens")
}
```

This is often a great way to carve up your complex views into smaller chunks, but be careful: unlike the **body** property, Swift won't automatically apply the **@ViewBuilder** attribute here, so if you want to send multiple views back you have three options.

First, you can place them in a stack, like this:

```
var spells: some View {
    VStack {
        Text("Lumos")
        Text("Obliviate")
    }
}
```

If you don't specifically want to organize them in a stack, you can also send back a **Group**. When this happens, the arrangement of your views is determined by how you use them elsewhere in your code:

```
var spells: some View {
    Group {
        Text("Lumos")
        Text("Obliviate")
    }
}
```

Project 3: Views and Modifiers

The third option is to add the **@ViewBuilder** attribute yourself, like this:

```
@ViewBuilder var spells: some View {  
    Text("Lumos")  
    Text("Oblivate")  
}
```

Of them all, I prefer to use **@ViewBuilder** because it mimics the way **body** works, however I'm also wary when I see folks cram lots of functionality into their properties – it's usually a sign that their views are getting a bit too complex, and need to be broken up. Speaking of which, let's tackle that next...

View composition

SwiftUI lets us break complex views down into smaller views without incurring much if any performance impact. This means that we can split up one large view into multiple smaller views, and SwiftUI takes care of reassembling them for us.

For example, in this view we have a particular way of styling text views – they have a large font, some padding, foreground and background colors, plus a capsule shape:

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 10) {
            Text("First")
                .font(.largeTitle)
                .padding()
                .foregroundColor(.white)
                .background(.blue)
                .clipShape(Capsule())
            Text("Second")
                .font(.largeTitle)
                .padding()
                .foregroundColor(.white)
                .background(.blue)
                .clipShape(Capsule())
        }
    }
}
```

Because those two text views are identical apart from their text, we can wrap them up in a new custom view, like this:

```
struct CapsuleText: View {
```

Project 3: Views and Modifiers

```
var text: String

var body: some View {
    Text(text)
        .font(.largeTitle)
        .padding()
        .foregroundColor(.white)
        .background(.blue)
        .clipShape(Capsule())
}
```

We can then use that **CapsuleText** view inside our original view, like this:

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 10) {
            CapsuleText(text: "First")
            CapsuleText(text: "Second")
        }
    }
}
```

Of course, we can also store some modifiers in the view and customize others when we use them. For example, if we removed **foregroundColor** from **CapsuleText**, we could then apply custom colors when creating instances of that view like this:

```
VStack(spacing: 10) {
    CapsuleText(text: "First")
        .foregroundColor(.white)
    CapsuleText(text: "Second")
        .foregroundColor(.yellow)
}
```

[View composition](#)

Don't worry about performance issues here – it's extremely efficient to break up SwiftUI views in this way.

Custom modifiers

SwiftUI gives us a range of built-in modifiers, such as `font()`, `background()`, and `clipShape()`. However, it's also possible to create custom modifiers that do something specific.

To create a custom modifier, create a new struct that conforms to the **ViewModifier** protocol. This has only one requirement, which is a method called **body** that accepts whatever content it's being given to work with, and must return **some View**.

For example, we might say that all titles in our app should have a particular style, so first we need to create a custom **ViewModifier** struct that does what we want:

```
struct Title: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.largeTitle)
            .foregroundColor(.white)
            .padding()
            .background(.blue)
            .clipShape(RoundedRectangle(cornerRadius: 10))
    }
}
```

We can now use that with the `modifier()` modifier – yes, it's a modifier called “modifier”, but it lets us apply any sort of modifier to a view, like this:

```
Text("Hello World")
    .modifier(Title())
```

When working with custom modifiers, it's usually a smart idea to create extensions on **View** that make them easier to use. For example, we might wrap the **Title** modifier in an extension such as this:

```
extension View {
```

```
func titleStyle() -> some View {
    modifier>Title()
}
```

We can now use the modifier like this:

```
Text("Hello World")
    .titleStyle()
```

Custom modifiers can do much more than just apply other existing modifiers – they can also create new view structure, as needed. Remember, modifiers return new objects rather than modifying existing ones, so we could create one that embeds the view in a stack and adds another view:

```
struct Watermark: ViewModifier {
    var text: String

    func body(content: Content) -> some View {
        ZStack(alignment: .bottomTrailing) {
            content
            Text(text)
                .font(.caption)
                .foregroundColor(.white)
                .padding(5)
                .background(.black)
        }
    }
}

extension View {
    func watermarked(with text: String) -> some View {
        modifier(Watermark(text: text))
    }
}
```

Project 3: Views and Modifiers

```
    }  
}
```

With that in place, we can now add a watermark to any view like this:

```
Color.blue  
.frame(width: 300, height: 200)  
.watermarked(with: "Hacking with Swift")
```

Tip: Often folks wonder when it's better to add a custom view modifier versus just adding a new method to **View**, and really it comes down to one main reason: custom view modifiers can have their own stored properties, whereas extensions to **View** cannot.

Custom containers

Although it's not something you're likely to do often, I want to at least show you that it's perfectly possible to create custom containers in your SwiftUI apps. This takes more advanced Swift knowledge because it leverages some of Swift's power features, so it's OK to skip this if you find it too much.

To try it out, we're going to make a new type of stack called a **GridStack**, which will let us create any number of views inside a fixed grid. What we want to say is that there is a new struct called **GridStack** that conforms to the **View** protocol and has a set number of rows and columns, and that inside the grid will be lots of content cells that themselves must conform to the **View** protocol.

In Swift we'd write this:

```
struct GridStack<Content: View>: View {  
    let rows: Int  
    let columns: Int  
    let content: (Int, Int) -> Content  
  
    var body: some View {  
        // more to come  
    }  
}
```

The first line – **struct GridStack<Content: View>: View** – uses a more advanced feature of Swift called *generics*, which in this case means “you can provide any kind of content you like, but whatever it is it must conform to the **View** protocol.” After the colon we repeat **View** again to say that **GridStack** itself also conforms to the **View** protocol.

Take particular note of the **let content** line – that defines a closure that must be able to accept two integers and return some sort of content we can show.

We need to complete the **body** property with something that combines multiple vertical and

Project 3: Views and Modifiers

horizontal stacks to create as many cells as was requested. We don't need to say what's *in* each cell, because we can get that by calling our **content** closure with the appropriate row and column.

So, we might fill it in like this:

```
var body: some View {
    VStack {
        ForEach(0..
```

Tip: When looping over ranges, SwiftUI can use the range directly only if we know for sure the values in the range won't change over time. Here we're using **ForEach** with **0..<rows** and **0..<columns**, both of which are values that *can* change over time – we might add more rows, for example. In this situation, we need to add a second parameter to **ForEach**, **id: \.self**, to tell SwiftUI how it can identify each view in the loop. We'll go into more detail on this in project 5.

Now that we have a custom container, we can write a view using it like this:

```
struct ContentView: View {
    var body: some View {
        GridStack(rows: 4, columns: 4) { row, col in
            Text("R\((row)) C\((col))")
        }
    }
}
```

```
}
```

Our **GridStack** is capable of accepting any kind of cell content, as long as it conforms to the **View** protocol. So, we could give cells a stack of their own if we wanted:

```
GridStack(rows: 4, columns: 4) { row, col in
    HStack {
        Image(systemName: "\((row * 4 + col).circle")
        Text("R\((row) C\((col)")
    }
}
```

For more flexibility we could leverage the same **@ViewBuilder** attribute used by SwiftUI for the **body** property of its views. Modify the **content** property of **GridStack** to this:

```
@ViewBuilder let content: (Int, Int) -> Content
```

With that in place SwiftUI will now automatically create an implicit horizontal stack inside our cell closure:

```
GridStack(rows: 4, columns: 4) { row, col in
    Image(systemName: "\((row * 4 + col).circle")
    Text("R\((row) C\((col)")
}
```

Both options work, so do whichever you prefer.

Views and modifiers: Wrap up

These technique projects are designed to dive deep into specific SwiftUI topics, and I hope you've learned a lot about views and modifiers here – why SwiftUI uses structs for views, why **some View** is so useful, how modifier order matters, and much more.

Views and modifiers are the fundamental building blocks of any SwiftUI app, which is why I wanted to focus on them so early in this course. View composition is particularly key, as it allows to build small, reusable views that can be assembled like bricks into larger user interfaces.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on:

1. Go back to project 1 and use a conditional modifier to change the total amount text view to red if the user selects a 0% tip.
2. Go back to project 2 and replace the **Image** view used for flags with a new **FlagImage()** view that renders one flag image using the specific set of modifiers we had.
3. Create a custom **ViewModifier** (and accompanying **View** extension) that makes a view have a large, blue font suitable for prominent titles in a view.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here: **Solution to Views and Modifiers**. If you don't already subscribe, you can start a free trial

today.

Project 4

BetterRest

Use machine learning to improve sleep

BetterRest: Introduction

This SwiftUI project is another forms-based app that will ask the user to enter information and convert that all into an alert, which might sound dull – you've done this already, right?

Well, yes, but practice is never a bad thing. However, the reason we have a fairly simple project is because I want to introduce you to one of the true power features of iOS development: machine learning (ML).

All iPhones come with a technology called Core ML built right in, which allows us to write code that makes predictions about new data based on previous data it has seen. We'll start with some raw data, give that to our Mac as training data, then use the results to build an app able to make accurate estimates about new data – all on device, and with complete privacy for users.

The actual app we're building is called BetterRest, and it's designed to help coffee drinkers get a good night's sleep by asking them three questions:

1. When do they want to wake up?
2. Roughly how many hours of sleep do they want?
3. How many cups of coffee do they drink per day?

Once we have those three values, we'll feed them into Core ML to get a result telling us when they ought to go to bed. If you think about it, there are billions of possible answers – all the various wake times multiplied by all the number of sleep hours, multiplied again by the full range of coffee amounts.

That's where machine learning comes in: using a technique called *regression analysis* we can ask the computer to come up with an algorithm able to represent all our data. This in turn allows it to apply the algorithm to fresh data it hasn't seen before, and get accurate results.

You're going to need to download some files for this project, which you can do from GitHub: <https://github.com/twostraws/HackingWithSwift> – make sure you look in the SwiftUI section of the files.

Project 4: BetterRest

Once you have those, go ahead and create a new App project in Xcode called BetterRest, making sure to target iOS 15 or later. As before we're going to be starting with an overview of the various technologies required to build the app, so let's get into it...

Entering numbers with Stepper

SwiftUI has two ways of letting users enter numbers, and the one we'll be using here is **Stepper**: a simple - and + button that can be tapped to select a precise number. The other option is **Slider**, which we'll be using later on – it also lets us select from a range of values, but less precisely.

Steppers are smart enough to work with any kind of number type you like, so you can bind them to **Int**, **Double**, and more, and it will automatically adapt. For example, we might create a property like this:

```
@State private var sleepAmount = 8.0
```

We could then bind that to a stepper so that it showed the current value, like this:

```
Stepper("$(sleepAmount) hours", value: $sleepAmount)
```

When that code runs you'll see 8.000000 hours, and you can tap the - and + to step downwards to 7, 6, 5 and into negative numbers, or step upwards to 9, 10, 11, and so on.

By default steppers are limited only by the range of their storage. We're using a **Double** in this example, which means the maximum value of the slider will be absolutely massive.

Now, as a father of two kids I can't tell you how much I love to sleep, but even *I* can't sleep that much. Fortunately, **Stepper** lets us limit the values we want to accept by providing an **in** range, like this:

```
Stepper("$(sleepAmount) hours", value: $sleepAmount, in:  
4...12)
```

With that change, the stepper will start at 8, then allow the user to move between 4 and 12 inclusive, but not beyond. This allows us to control the sleep range so that users can't try to sleep for 24 hours, but it also lets us reject impossible values – you can't sleep for -1 hours, for example.

Project 4: BetterRest

There's a fourth useful parameter for **Stepper**, which is a **step** value – how far to move the value each time - or + is tapped. Again, this can be any sort of number, but it does need to match the type used for the binding. So, if you are binding to an integer you can't then use a **Double** for the step value.

In this instance, we might say that users can select any sleep value between 4 and 12, moving in 15 minute increments:

```
Stepper("\($sleepAmount) hours", value: $sleepAmount, in:  
4...12, step: 0.25)
```

That's starting to look useful – we have a precise range of reasonable values, a sensible step increment, and users can see exactly what they have chosen each time.

Before we move on, though, let's fix that text: it says 8.000000 right now, which is accurate but a little *too* accurate. To fix this, we can just ask Swift to format the **Double** using **formatted()**:

```
Stepper("\($sleepAmount.formatted()) hours", value:  
$sleepAmount, in: 4...12, step: 0.25)
```

Perfect!

Selecting dates and times with DatePicker

SwiftUI gives us a dedicated picker type called **DatePicker** that can be bound to a date property. Yes, Swift has a dedicated type for working with dates, and it's called – unsurprisingly – **Date**.

So, to use it you'd start with an **@State** property such as this:

```
@State private var wakeUp = Date.now
```

You could then bind that to a date picker like this:

```
DatePicker("Please enter a date", selection: $wakeUp)
```

Try running that in the simulator so you can see how it looks. You should see a tappable options to control days and times, plus the “Please enter a date” label on the left.

Now, you might think that label looks ugly, and try replacing it with this:

```
DatePicker("", selection: $wakeUp)
```

But if you do that you now have *two* problems: the date picker still makes space for a label even though it's empty, and now users with the screen reader active (more familiar to us as VoiceOver) won't have any idea what the date picker is for.

A better alternative is to use the **labelsHidden()** modifier, like this:

```
DatePicker("Please enter a date", selection: $wakeUp)  
.labelsHidden()
```

That still includes the original label so screen readers can use it for VoiceOver, but now they aren't visible onscreen any more – the date picker won't be pushed to one side by some empty

Project 4: BetterRest

text.

Date pickers provide us with a couple of configuration options that control how they work. First, we can use **displayedComponents** to decide what kind of options users should see:

- If you don't provide this parameter, users see a day, hour, and minute.
- If you use **.date** users see month, day, and year.
- If you use **.hourAndMinute** users see just the hour and minute components.

So, we can select a precise time like this:

```
DatePicker("Please enter a time", selection: $wakeUp,  
displayedComponents: .hourAndMinute)
```

Finally, there's an **in** parameter that works just the same as with **Stepper**: we can provide it with a date range, and the date picker will ensure the user can't select beyond it.

Now, we've been using ranges for a while now, and you're used to seeing things like **1...5** or **0..<10**, but we can also use Swift dates with ranges. For example:

```
func exampleDates() {  
    // create a second Date instance set to one day in seconds  
    from now  
    let tomorrow = Date.now.addingTimeInterval(86400)  
  
    // create a range from those two  
    let range = Date.now...tomorrow  
}
```

That's really useful with **DatePicker**, but there's something even better: Swift lets us form *one-sided ranges* – ranges where we specify either the start or end but not both, leaving Swift to infer the other side.

For example, we could create a date picker like this:

Selecting dates and times with DatePicker

```
DatePicker("Please enter a date", selection: $wakeUp, in:  
Date.now...)
```

That will allow all dates in the future, but none in the past – read it as “from the current date up to anything.”

Working with dates

Having users enter dates is as easy as binding an `@State` property of type `Date` to a `DatePicker` SwiftUI control, but things get a little woolier afterwards.

You see, working with dates is hard. Like, *really* hard – way harder than you think. Way harder than *I* think, and I've been working with dates for years.

Take a look at this trivial example:

```
let now = Date.now
let tomorrow = Date.now.addingTimeInterval(86400)
let range = now...tomorrow
```

That creates a range from now to the same time tomorrow (86400 is the number of seconds in a day).

That might seem easy enough, but do all days have 86,400 seconds? If they did, a lot of people would be out of jobs! Think about daylight savings time: sometimes clocks go forward (losing an hour) and sometimes go backwards (gaining an hour), meaning that we might have 23 or 25 hours in those days. Then there are leap seconds: times that get added to the clocks in order to adjust for the Earth's slowing rotation.

If you think that's hard, try running this from your Mac's terminal: `cal`. This prints a simple calendar for the current month, showing you the days of the week. Now try running `cal 9 1752`, which shows you the calendar for September 1752 – you'll notice 12 whole days are missing, thanks to the calendar moving from Julian to Gregorian.

Now, the reason I'm saying all this isn't to scare you off – dates are inevitable in our programs, after all. Instead, I want you to understand that for anything significant – any usage of dates that actually matters in our code – we should rely on Apple's frameworks for calculations and formatting.

In the project we're making we'll be using dates in three ways:

1. Choosing a sensible default “wake up” time.
2. Reading the hour and minute they want to wake up.
3. Showing their suggested bedtime neatly formatted.

We could, if we wanted, do all that by hand, but then you’re into the realm of daylight savings, leap seconds, and Gregorian calendars.

Much better is to have iOS do all that hard work for us: it’s much less work, and it’s guaranteed to be correct regardless of the user’s region settings.

Let’s tackle each of those individually, starting with choosing a sensible wake up time.

As you’ve seen, Swift gives us **Date** for working with dates, and that encapsulates the year, month, date, hour, minute, second, timezone, and more. However, we don’t want to think about most of that – we want to say “give me an 8am wake up time, regardless of what day it is today.”

Swift has a slightly different type for that purpose, called **DateComponents**, which lets us read or write specific parts of a date rather than the whole thing.

So, if we wanted a date that represented 8am today, we could write code like this:

```
var components = DateComponents()
components.hour = 8
components.minute = 0
let date = Calendar.current.date(from: components)
```

Now, because of difficulties around date validation, that **date(from:)** method actually returns an optional date, so it’s a good idea to use nil coalescing to say “if that fails, just give me back the current date”, like this:

```
let date = Calendar.current.date(from: components) ?? Date.now
```

The second challenge is how we could read the hour they want to wake up. Remember,

Project 4: BetterRest

DatePicker is bound to a **Date** giving us lots of information, so we need to find a way to pull out just the hour and minute components.

Again, **DateComponents** comes to the rescue: we can ask iOS to provide specific components from a date, then read those back out. One hiccup is that there's a disconnect between the values we *request* and the values we *get* thanks to the way **DateComponents** works: we can ask for the hour and minute, but we'll be handed back a **DateComponents** instance with optional values for all its properties. Yes, we know hour and minute will be there because those are the ones we asked for, but we still need to unwrap the optionals or provide default values.

So, we might write code like this:

```
let components =
    Calendar.current.dateComponents([.hour, .minute], from:
someDate)
let hour = components.hour ?? 0
let minute = components.minute ?? 0
```

The last challenge is how we can format dates and times, and here we have two options.

First is to rely on the **format** parameter that has worked so well for us in the past, and here we can ask for whichever parts of the date we want to show.

For example, if we just wanted the time from a date we would write this:

```
Text(Date.now, format: .dateTime.hour().minute())
```

Or if we wanted the day, month, and year, we would write this:

```
Text(Date.now, format: .dateTime.day().month().year())
```

You might wonder how that adapts to handling different date formats – for example, here in the UK we use day/month/year, but in some other countries they use month/day/year. Well, the

magic is that we don't need to worry about this: when we write `day().month().year()` we're asking for that data, not *arranging* it, and iOS will automatically format that data using the user's preferences.

As an alternative, we can use the `formatted()` method directly on dates, passing in configuration options for how we want both the date and the time to be formatted, like this:

```
Text(Date.now.formatted(date: .long, time: .shortened))
```

The point is that dates *are* hard, but Apple has provided us with stacks of helpers to make them *less* hard. If you learn to use them well you'll write less code, and write better code too!

Training a model with Create ML

On-device machine learning went from “extremely hard to do” to “quite possible, and surprisingly powerful” in iOS 11, all thanks to one Apple framework: Core ML. A year later, Apple introduced a second framework called Create ML, which added “easy to do” to the list, and then a second year later Apple introduced a Create ML app that made the whole process drag and drop. As a result of all this work, it’s now within the reach of anyone to add machine learning to their app.

Core ML is capable of handling a variety of training tasks, such as recognizing images, sounds, and even motion, but in this instance we’re going to look at tabular regression. That’s a fancy name, which is common in machine learning, but all it really means is that we can throw a load of spreadsheet-like data at Create ML and ask it to figure out the relationship between various values.

Machine learning is done in two steps: we train the model, then we ask the model to make predictions. Training is the process of the computer looking at all our data to figure out the relationship between all the values we have, and in large data sets it can take a long time – easily hours, potentially much longer. Prediction is done on device: we feed it the trained model, and it will use previous results to make estimates about new data.

Let’s start the training process now: please open the Create ML app on your Mac. If you don’t know where this is, you can launch it from Xcode by going to the Xcode menu and choosing Open Developer Tool > Create ML.

The first thing the Create ML app will do is ask you to create a project or open a previous one – please click New Document to get started. You’ll see there are lots of templates to choose from, but if you scroll down to the bottom you’ll see Tabular Regression; please choose that and press Next. For the project name please enter BetterRest, then press Next, select your desktop, then press Create.

This is where Create ML can seem a little tricky at first, because you’ll see a screen with quite a few options. Don’t worry, though – once I walk you through it isn’t so hard.

The first step is to provide Create ML with some training data. This is the raw statistics for it to look at, which in our case consists of four values: when someone wanted to wake up, how much sleep they thought they liked to have, how much coffee they drink per day, and how much sleep they *actually* need.

I've provided this data for you in BetterRest.csv, which is in the project files for this project. This is a comma-separated values data set that Create ML can work with, and our first job is to import that.

So, in Create ML look under Data and select “Select...” under the Training Data title. When you press “Select...” again it will open a file selection window, and you should choose BetterRest.csv.

Important: This CSV file contains sample data for the purpose of this project, and should not be used for actual health-related work.

The next job is to decide the target, which is the value we want the computer to learn to predict, and the features, which are the values we want the computer to inspect in order to predict the target. For example, if we chose how much sleep someone thought they needed and how much sleep they *actually* needed as features, we could train the computer to predict how much coffee they drink.

In this instance, I'd like you to choose “actualSleep” for the target, which means we want the computer to learn how to predict how much sleep they actually need. Now press Choose Features, and select all three options: wake, estimatedSleep, and coffee – we want the computer to take all three of those into account when producing its predictions.

Below the Select Features button is a dropdown button for the algorithm, and there are five options: Automatic, Random Forest, Boosted Tree, Decision Tree, and Linear Regression. Each takes a different approach to analyzing data, but helpfully there is an Automatic option that attempts to choose the best algorithm automatically. It's not always correct, and in fact it does limit the options we have quite dramatically, but for this project it's more than good enough.

Project 4: BetterRest

Tip: If you want an overview of what the various algorithms do, I have a talk just for you called Create ML for Everyone – it's on YouTube at <https://youtu.be/a905KIBw1hs>

When you're ready, click the Train button in the window title bar. After a couple of seconds – our data is pretty small! – it will complete, and you'll see a big checkmark telling you that everything went to plan.

To see how the training went, select the Evaluation tab then choose Validation to see some result metrics. The value we care about is called Root Mean Squared Error, and you should get a value around about 170. This means on average the model was able to predict suggested accurate sleep time with an error of only 170 seconds, or three minutes.

Tip: Create ML provides us with both Training and Validation statistics, and both are important. When we asked it to train using our data, it automatically split the data up: some to use for training its machine learning model, but then it held back a chunk for validation. This validation data is then used to check its model: it makes a prediction based on the input, then checks how far that prediction was off the real value that came from the data.

Even better, if you go to the Output tab you'll see our finished model has a file size of 544 bytes or so. Create ML has taken 180KB of data, and condensed it down to just 544 bytes – almost nothing.

Now, 544 bytes sounds tiny, I know, but it's worth adding that almost all of those bytes are metadata: the author name is in there, along with the names of all the fields: wake, estimatedSleep, coffee, and actualSleep.

The actual amount of space taken up by the hard data – how to predict the amount of required sleep based on our three variables – is well under 100 bytes. This is possible because Create ML doesn't actually care what the values are, it only cares what the relationships are. So, it spent a couple of billion CPU cycles trying out various combinations of weights for each of the features to see which ones produce the closest value to the actual target, and once it knows the best algorithm it simply stores that.

Now that our model is trained, I'd like you to press the Get button to export it to your desktop,

so we can use it in code.

Tip: If you want to try training again – perhaps to experiment with the various algorithms available to us – right-click on your model source in the left-hand window, then select Duplicate.

Building a basic layout

This app is going to allow user input with a date picker and two steppers, which combined will tell us when they want to wake up, how much sleep they usually like, and how much coffee they drink.

So, please start by adding three properties that let us store the information for those controls:

```
@State private var wakeUp = Date.now  
@State private var sleepAmount = 8.0  
@State private var coffeeAmount = 1
```

Inside our **body** we're going to place three sets of components wrapped in a **VStack** and a **NavigationView**, so let's start with the wake up time. Replace the default "Hello World" text view with this:

```
NavigationView {  
    VStack {  
        Text("When do you want to wake up?")  
            .font(.headline)  
  
        DatePicker("Please enter a time", selection: $wakeUp,  
displayedComponents: .hourAndMinute)  
            .labelsHidden()  
  
        // more to come  
    }  
}
```

We've asked for **.hourAndMinute** configuration because we care about the time someone wants to wake up and not the day, and with the **labelsHidden()** modifier we don't get a second label for the picker – the one above is more than enough.

Next we're going to add a stepper to let users choose roughly how much sleep they want. By giving this thing an **in** range of **4...12** and a step of 0.25 we can be sure they'll enter sensible values, but we can combine that with the **formatted()** method so we see numbers like "8" and not "8.000000".

Add this code in place of the **// more to come** comment"

```
Text("Desired amount of sleep")
    .font(.headline)

Stepper("\(sleepAmount.formatted()) hours", value:
$sleepAmount, in: 4...12, step: 0.25)
```

Finally we'll add one last stepper and label to handle how much coffee they drink. This time we'll use the range of 1 through 20 (because surely 20 coffees a day is enough for anyone?), but we'll also display one of two labels inside the stepper to handle pluralization better. If the user has set a **coffeeAmount** of exactly 1 we'll show "1 cup", otherwise we'll use that amount plus "cups", all decided using the ternary conditional operator.

Add these inside the **VStack**, below the previous views:

```
Text("Daily coffee intake")
    .font(.headline)

Stepper(coffeeAmount == 1 ? "1 cup" : "\(coffeeAmount) cups",
value: $coffeeAmount, in: 1...20)
```

The final thing we need is a button to let users calculate the best time they should go to sleep. We could do that with a simple button at the end of the **VStack**, but to spice up this project a little I want to try something new: we're going to add a button directly to the navigation bar.

First we need a method for the button to call, so add an empty **calculateBedtime()** method like this:

Project 4: BetterRest

```
func calculateBedtime() {  
}
```

Now we need to use the **toolbar()** modifier to add a trailing button to the navigation view. We used this previously along with **ToolbarItemGroup** to place a button next to the keyboard, but here our needs are much simpler: we just want a single button in the navigation bar, which can be done by adding a button directly to the toolbar.

While we're here, we might as well also use **navigationTitle()** to put some text at the top.

So, add these modifiers to the **VStack**:

```
.navigationTitle("BetterRest")  
.toolbar {  
    Button("Calculate", action: calculateBedtime)  
}
```

Tip: Our button will automatically be placed in the top-right corner for left-to-right languages such as English, but will automatically move to the other side for right-to-left languages.

That won't do anything yet because **calculateBedtime()** is empty, but at least our UI is good enough for the time being.

Connecting SwiftUI to Core ML

In the same way that SwiftUI makes user interface development easy, Core ML makes machine learning easy. How easy? Well, once you have a trained model you can get predictions in just two lines of code – you just need to send in the values that should be used as input, then read what comes back.

In our case, we already made a Core ML model using Xcode’s Create ML app, so we’re going to use that. You should have saved it on your desktop, so please now drag it into the project navigator in Xcode. When Xcode prompts you to “Copy items if needed”, please make sure that box is checked.

When you add an .mlmodel file to Xcode, it will automatically create a Swift class of the same name. You can’t see the class, and don’t need to – it’s generated automatically as part of the build process. However, it *does* mean that if your model file is named oddly then the auto-generated class name will also be named oddly.

No matter what name your model file has, please rename it to be “SleepCalculator.mlmodel”, thus making the auto-generated class be called `SleepCalculator`.

How can we be sure that’s the class name? Well, just select the model file itself and Xcode will show you more information. You’ll see it knows our author, the name of the Swift class that gets made, plus a list of inputs and their types, and an output plus type too – these were encoded in the model file, which is why it was (comparatively!) so big.

We’re going to start filling in `calculateBedtime()` in just a moment, but before that can start we need to add an import for CoreML because we’re using functionality outside of SwiftUI.

So, scroll to the top of ContentView.swift and add this before the `import` line for SwiftUI:

```
import CoreML
```

Tip: You don’t strictly need to add CoreML before SwiftUI, but keeping your imports in

Project 4: BetterRest

alphabetical order makes them easier to check later on.

Okay, now we can turn to **calculateBedtime()**. First, we need to create an instance of the **SleepCalculator** class, like this:

```
do {
    let config = MLModelConfiguration()
    let model = try SleepCalculator(configuration: config)

    // more code here
} catch {
    // something went wrong!
}
```

That model instance is the thing that reads in all our data, and will output a prediction. The configuration is there in case you need to enable a handful of what are fairly obscure options – perhaps folks working in machine learning full time need these, but honestly I’d guess only 1 in 1000 folks actually use these.

I *do* want you to focus on the **do/catch** blocks, because using Core ML can throw errors in two places: loading the model as seen above, but also when we ask for predictions. Honestly, I can’t think I’ve ever had a prediction fail in my life, but there’s no harm being safe!

Anyway, we trained our model with a CSV file containing the following fields:

- “wake”: when the user wants to wake up. This is expressed as the number of seconds from midnight, so 8am would be 8 hours multiplied by 60 multiplied by 60, giving 28800.
- “estimatedSleep”: roughly how much sleep the user wants to have, stored as values from 4 through 12 in quarter increments.
- “coffee”: roughly how many cups of coffee the user drinks per day.

So, in order to get a prediction out of our model, we need to fill in those values.

We already have two of them, because our **sleepAmount** and **coffeeAmount** properties are

mostly good enough – we just need to convert **coffeeAmount** from an integer to a **Double** so that Swift is happy.

But figuring out the wake time requires more thinking, because our **wakeUp** property is a **Date** not a **Double** representing the number of seconds. Helpfully, this is where Swift's **DateComponents** type comes in: it stores all the parts required to represent a date as individual values, meaning that we can read the hour and minute components and ignore the rest. All we then need to do is multiply the minute by 60 (to get seconds rather than minutes), and the hour by 60 and 60 (to get seconds rather than hours).

We can get a **DateComponents** instance from a **Date** with a very specific method call: **Calendar.current.dateComponents()**. We can then request the hour and minute components, and pass in our wake up date. The **DateComponents** instance that comes back has properties for all its components – year, month, day, timezone, etc – but most of them won't be set. The ones we asked for – hour and minute – *will* be set, but will be optional, so we need to unwrap them carefully.

So, put this in place of the // **more code here** comment in **calculateBedtime()**:

```
let components =
    Calendar.current.dateComponents([.hour, .minute], from: wakeUp)
let hour = (components.hour ?? 0) * 60 * 60
let minute = (components.minute ?? 0) * 60
```

That code uses 0 if either the hour or minute can't be read, but realistically that's never going to happen so it will result in **hour** and **minute** being set to those values in seconds.

The next step is to feed our values into Core ML and see what comes out. This is done using the **prediction()** method of our model, which wants the wake time, estimated sleep, and coffee amount values required to make a prediction, all provided as **Double** values. We just calculated our **hour** and **minute** as seconds, so we'll add those together before sending them in.

Please add this just below the previous code:

Project 4: BetterRest

```
let prediction = try model.prediction(wake: Double(hour +  
minute), estimatedSleep: sleepAmount, coffee:  
Double(coffeeAmount))  
  
// more code here
```

With that in place, **prediction** now contains how much sleep they actually need. This almost certainly wasn't part of the training data our model saw, but was instead computed dynamically by the Core ML algorithm.

However, it's not a helpful value for users – it will be some number in seconds. What we want is to convert that into the time they should go to bed, which means we need to subtract that value in seconds from the time they need to wake up.

Thanks to Apple's powerful APIs, that's just one line of code – you can subtract a value in seconds directly from a **Date**, and you'll get back a new **Date!** So, add this line of code after the prediction:

```
let sleepTime = wakeUp - prediction.actualSleep
```

And now we know exactly when they should go to sleep. Our final challenge, for now at least, is to show that to the user. We'll be doing this with an alert, because you've already learned how to do that and could use the practice.

So, start by adding three properties that determine the title and message of the alert, and whether or not it's showing:

```
@State private var alertTitle = ""  
@State private var alertMessage = ""  
@State private var showingAlert = false
```

We can immediately use those values in **calculateBedtime()**. If our calculation goes wrong – if reading a prediction throws an error – we can replace the // **something went wrong**

comment with some code that sets up a useful error message:

```
alertTitle = "Error"
alertMessage = "Sorry, there was a problem calculating your
bedtime."
```

And regardless of whether or not the prediction worked, we should show the alert. It might contain the results of their prediction or it might contain the error message, but it's still useful. So, put this at the end of `calculateBedtime()`, *after* the `catch` block:

```
showingAlert = true
```

If the prediction worked we create a constant called `sleepTime` that contains the time they need to go to bed. But this is a **Date** rather than a neatly formatted string, so we'll pass it through the `formatted()` method to make sure it's human-readable, then assign it to `alertMessage`.

So, put these final lines of code into `calculateBedtime()`, directly after where we set the `sleepTime` constant:

```
alertTitle = "Your ideal bedtime is..."
alertMessage = sleepTime.formatted(date: .omitted,
time: .shortened)
```

To wrap up this stage of the app, we just need to add an `alert()` modifier that shows `alertTitle` and `alertMessage` when `showingAlert` becomes true.

Please add this modifier to our `VStack`:

```
.alert(alertTitle, isPresented: $showingAlert) {
    Button("OK") { }
} message: {
    Text(alertMessage)
}
```

Project 4: BetterRest

Now go ahead and run the app – it works! It doesn't look *great*, but it works.

Cleaning up the user interface

Although our app works right now, it's not something you'd want to ship on the App Store – it has at least one major usability problem, and the design is... well... let's say "substandard".

Let's look at the usability problem first, because it's possible it hasn't occurred to you. When you read **Date.now** it is automatically set to the current date and time. So, when we create our **wakeUp** property with a new date, the default wake up time will be whatever time it is right now.

Although the app needs to be able to handle any sort of times – we don't want to exclude folks on night shift, for example – I think it's safe to say that a default wake up time somewhere between 6am and 8am is going to be more useful to the vast majority of users.

To fix this we're going to add a computed property to our **ContentView** struct that contains a **Date** value referencing 7am of the current day. This is surprisingly easy: we can just create a new **DateComponents** of our own, and use **Calendar.current.date(from:)** to convert those components into a full date.

So, add this property to **ContentView** now:

```
var defaultWakeTime: Date {
    var components = DateComponents()
    components.hour = 7
    components.minute = 0
    return Calendar.current.date(from: components) ?? Date.now
}
```

And now we can use that for the default value of **wakeUp** in place of **Date.now**:

```
@State private var wakeUp = defaultWakeTime
```

If you try compiling that code you'll see it fails, and the reason is that we're accessing one property from inside another – Swift doesn't know which order the properties will be created

Project 4: BetterRest

in, so this isn't allowed.

The fix here is simple: we can make **defaultWakeTime** a static variable, which means it belongs to the **ContentView** struct itself rather than a single instance of that struct. This in turn means **defaultWakeTime** can be read whenever we want, because it doesn't rely on the existence of any other properties.

So, change the property definition to this:

```
static var defaultWakeTime: Date {
```

That fixes our usability problem, because the majority of users will find the default wake up time is close to what they want to choose.

As for our styling, this requires more effort. A simple change to make is to switch to a **Form** rather than a **VStack**. So, find this:

```
NavigationView {  
    VStack {
```

And replace it with this:

```
NavigationView {  
    Form {
```

That immediately makes the UI look better – we get a clearly segmented table of inputs, rather than some controls centered in a white space.

There's still an annoyance in our form: every view inside the form is treated as a row in the list, when really all the text views form part of the same logical form section.

We *could* use **Section** views here, with our text views as titles – you'll get to experiment with that in the challenges. Instead, we're going to wrap each pair of text view and control with a **VStack** so they are seen as a single row each.

Go ahead and wrap each of the pairs in a **VStack** now, using **.leading** for the alignment and 0 for spacing. For example, you'd take these two views:

```
Text("Desired amount of sleep")
    .font(.headline)

Stepper("\(sleepAmount.formatted()) hours", value:
$sleepAmount, in: 4...12, step: 0.25)
```

And wrap them in a **VStack** like this:

```
VStack(alignment: .leading, spacing: 0) {
    Text("Desired amount of sleep")
        .font(.headline)

    Stepper("\(sleepAmount.formatted()) hours", value:
$sleepAmount, in: 4...12, step: 0.25)
}
```

And now run the app one last time, because it's done – good job!

BetterRest: Wrap up

This project gave you the chance to get some practice with forms and bindings, while also introducing you to **DatePicker**, **Stepper**, **Date**, **DateComponents**, and more, while also seeing how to place buttons into the navigation bar – these are things you’ll be using time and time again, so I wanted to get them in nice and early.

Of course, I also took the chance to give you a glimpse of some of the incredible things we can build using Apple’s frameworks, all thanks to Create ML and Core ML. As you saw, these frameworks allow us to take advantage of decades of research and development in machine learning, all using a drag and drop user interface and a couple of lines of code – it really couldn’t be any easier.

The truly fascinating thing about machine learning is that it doesn’t need big or clever scenarios to be used. You could use machine learning to predict used car prices, to figure out user handwriting, or even detect faces in images. And most importantly of all, the entire process happens on the user’s device, in complete privacy.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It’s my job to make sure you take as much from these tutorials as possible, so I’ve prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what’s going on:

1. Replace each **VStack** in our form with a **Section**, where the text view is the title of the section. Do you prefer this layout or the **VStack** layout? It’s your app – you choose!
2. Replace the “Number of cups” stepper with a **Picker** showing the same range of values.

3. Change the user interface so that it always shows their recommended bedtime using a nice and large font. You should be able to remove the “Calculate” button entirely.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to BetterRest. If you don't already subscribe, you can start a free trial today.

Project 5

Word Scramble

Build a letter rearranging game with List

Word Scramble: Introduction

This project will be another game, although really it's just a sneaky way for me to introduce more Swift and SwiftUI knowledge! The game will show players a random eight-letter word, and ask them to make words out of it. For example, if the starter word is “alarming” they might spell “alarm”, “ring”, “main”, and so on.

Along the way you'll meet **List**, **onAppear()**, **Bundle**, **fatalError()**, and more – all useful skills that you'll use for years to come. You'll also get some practice with **@State**, **NavigationView**, and more, which you should enjoy while you can – this is our last easy project!

To get started, please create a new App project called WordScramble, making sure to set iOS 15 as your deployment target. You'll need to download the files for this project, as it contains a file called “start.txt” that you'll be needing later on.

OK, let's get into some code...

Introducing List, your best friend

Of all SwiftUI's view types, **List** is the one you'll rely on the most. That doesn't mean you'll *use* it the most – I'm sure **Text** or **VStack** will claim that crown – more that it's such a workhorse that you'll come back to it time and time again. And this isn't new: the equivalent of **List** in UIKit was **UITableView**, and it got used just as much.

The job of **List** is to provide a scrolling table of data. In fact, it's pretty much identical to **Form**, except it's used for presentation of data rather than requesting user input. Don't get me wrong: you'll use **Form** quite a lot too, but really it's just a specialized type of **List**.

Just like **Form**, you can provide **List** a selection of static views to have them rendered in individual rows:

```
List {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

We can also switch to **ForEach** in order to create rows dynamically from an array or range:

```
List {  
    ForEach(0..<5) {  
        Text("Dynamic row \($0)")  
    }  
}
```

Where things get more interesting is the way you can mix static and dynamic rows:

```
List {  
    Text("Static row 1")  
    Text("Static row 2")
```

```
ForEach(0..<5) {
    Text("Dynamic row \($0)")
}

Text("Static row 3")
Text("Static row 4")
}
```

And of course we can combine that with sections, to make our list easier to read:

```
List {
    Section("Section 1") {
        Text("Static row 1")
        Text("Static row 2")
    }

    Section("Section 2") {
        ForEach(0..<5) {
            Text("Dynamic row \($0)")
        }
    }

    Section("Section 3") {
        Text("Static row 3")
        Text("Static row 4")
    }
}
```

Tip: As you can see, if your section header is just some text you can pass it in directly as a string – it's a helpful shortcut for times you don't need anything more advanced.

Being able to have both static and dynamic content side by side lets us recreate something like the Wi-Fi screen in Apple's Settings app – a toggle to enable Wi-Fi system-wide, then a

Project 5: Word Scramble

dynamic list of nearby networks, then some more static cells with options to auto-join hotspots and so on.

You'll notice that this list looks similar to the form we had previously, but we can adjust how the list looks using the **listStyle()** modifier, like this:

```
.listStyle(.grouped)
```

Now, everything you've seen so far works fine with **Form** as well as **List** – even the dynamic content. But one thing **List** can do that **Form** can't is to generate its rows entirely from dynamic content without needing a **ForEach**.

So, if your entire list is made up of dynamic rows, you can simply write this:

```
List(0..<5) {  
    Text("Dynamic row \($0)")  
}
```

This allows us to create lists really quickly, which is helpful given how common they are.

In this project we're going to use **List** slightly differently, because we'll be making it loop over an array of strings. We've used **ForEach** with ranges a lot, either hard-coded (`0..<5`) or relying on variable data (`0..<students.count`), and that works great because SwiftUI can identify each row uniquely based on its position in the range.

When working with an array of data, SwiftUI still needs to know how to identify each row uniquely, so if one gets removed it can simply remove that one rather than having to redraw the whole list. This is where the **id** parameter comes in, and it works identically in both **List** and **ForEach** – it lets us tell SwiftUI exactly what makes each item in the array unique.

When working with arrays of strings and numbers, the only thing that makes those values unique is the values themselves. That is, if we had the array `[2, 4, 6, 8, 10]`, then those numbers themselves are themselves the unique identifiers. After all, we don't have anything else to work with!

When working with this kind of list data, we use **id: \.self** like this:

```
struct ContentView: View {  
    let people = ["Finn", "Leia", "Luke", "Rey"]  
  
    var body: some View {  
        List(people, id: \.self) {  
            Text($0)  
        }  
    }  
}
```

That works just the same with **ForEach**, so if we wanted to mix static and dynamic rows we could have written this instead:

```
List {  
    Text("Static Row")  
  
    ForEach(people, id: \.self) {  
        Text($0)  
    }  
  
    Text("Static Row")  
}
```

Loading resources from your app bundle

When we use **Image** views, SwiftUI knows to look in your app's asset catalog to find the artwork, and it even automatically adjusts the artwork so it loads the correct picture for the current screen resolution – that's the @2x and @3x stuff we looked at earlier.

For other data, such as text files, we need to do more work. This also applies if you have specific data formats such as XML or JSON – it takes the same work regardless of what file types you're loading.

When Xcode builds your iOS app, it creates something called a “bundle”. This happens on all of Apple's platforms, including macOS, and it allows the system to store all the files for a single app in one place – the binary code (the actual compiled Swift stuff we wrote), all the artwork, plus any extra files we need all in one place.

In the future, as your skills grow, you'll learn how you can actually include multiple bundles in a single app, allowing you to write things like Siri extensions, iMessage apps, widgets, and more, all inside a single iOS app bundle. Although these get included with our app's download from the App Store, these other bundles are stored separately from our main app bundle – our main iOS app code and resources.

All this matters because it's common to want to look in a bundle for a file you placed there. This uses a new data type called **URL**, which stores pretty much exactly what you think: a URL such as <https://www.hackingwithswift.com>. However, URLs are a bit more powerful than just storing web addresses – they can also store the locations of files, which is why they are useful here.

Let's start writing some code. If we want to read the URL for a file in our main app bundle, we use **Bundle.main.url()**. If the file exists it will be sent back to us, otherwise we'll get back **nil**, so this is an optional **URL**. That means we need to unwrap it like this:

```
if let fileURL = Bundle.main.url(forResource: "some-file",
```

Loading resources from your app bundle

```
withExtension: "txt") {  
    // we found the file in our bundle!  
}
```

What's inside the **URL** doesn't really matter, because iOS uses paths that are impossible to guess – our app lives in its own sandbox, and we shouldn't try to read outside of it.

Once we have a URL, we can load it into a string with a special initializer:

String(contentsOf:). We give this a file URL, and it will send back a string containing the contents of that file if it can be loaded. If it *can't* be loaded it throws an error, so you need to call this using **try** or **try?** like this:

```
if let fileContents = try? String(contentsOf: fileURL) {  
    // we loaded the file into a string!  
}
```

Once you have the contents of the file, you can do with it whatever you want – it's just a regular string.

Working with strings

iOS gives us some really powerful APIs for working with strings, including the ability to split them into an array, remove whitespace, and even check spellings. We've looked at some of these previously, but there's at least one major addition I want to look at.

In this app, we're going to be loading a file from our app bundle that contains over 10,000 eight-letter words, each of which can be used to start the game. These words are stored one per line, so what we really want is to split that string up into an array of strings in order that we can pick one randomly.

Swift gives us a method called **components(separatedBy:)** that can converts a single string into an array of strings by breaking it up wherever another string is found. For example, this will create the array ["a", "b", "c"]:

```
let input = "a b c"  
let letters = input.components(separatedBy: " ")
```

We have a string where words are separated by line breaks, so to convert that into a string array we need to split on that.

In programming – almost universally, I think – we use a special character sequence to represent line breaks: `\n`. So, we would write code like this:

```
let input = """  
a  
b  
c  
"""  
  
let letters = input.components(separatedBy: "\n")
```

Regardless of what string we split on, the result will be an array of strings. From there we can read individual values by indexing into the array, such as `letters[0]` or `letters[2]`, but Swift

gives us a useful other option: the **randomElement()** method returns one random item from the array.

For example, this will read a random letter from our array:

```
let letter = letters.randomElement()
```

Now, although we can see that the letters array will contain three items, Swift doesn't know that – perhaps we tried to split up an empty string, for example. As a result, the **randomElement()** method returns an optional string, which we must either unwrap or use with nil coalescing.

Another useful string method is **trimmingCharacters(in:)**, which asks Swift to remove certain kinds of characters from the start and end of a string. This uses a new type called **CharacterSet**, but most of the time we want one particular behavior: removing whitespace and new lines – this refers to spaces, tabs, and line breaks, all at once.

This behavior is so common it's built right into the **CharacterSet** struct, so we can ask Swift to trim all whitespace at the start and end of a string like this:

```
let trimmed =
letter?.trimmingCharacters(in: .whitespacesAndNewlines)
```

There's one last piece of string functionality I'd like to cover before we dive into the main project, and that is the ability to check for misspelled words.

This functionality is provided through the class **UITextChecker**. You might not realize this, but the “UI” part of that name carries two additional meanings with it:

1. This class comes from UIKit. That doesn't mean we're loading all the old user interface framework, though; we actually get it automatically through SwiftUI.
2. It's written using Apple's older language, Objective-C. We don't need to write Objective-C to use it, but there is a slightly unwieldy API for Swift users.

Project 5: Word Scramble

Checking a string for misspelled words takes four steps in total. First, we create a word to check and an instance of **UITextChecker** that we can use to check that string:

```
let word = "swift"  
let checker = UITextChecker()
```

Second, we need to tell the checker how much of our string we want to check. If you imagine a spellchecker in a word processing app, you might want to check only the text the user selected rather than the entire document.

However, there's a catch: Swift uses a very clever, very advanced way of working with strings, which allows it to use complex characters such as emoji in exactly the same way that it uses the English alphabet. However, Objective-C does *not* use this method of storing letters, which means we need to ask Swift to create an Objective-C string range using the entire length of all our characters, like this:

```
let range = NSRange(location: 0, length: word.utf16.count)
```

UTF-16 is what's called a *character encoding* – a way of storing letters in a string. We use it here so that Objective-C can understand how Swift's strings are stored; it's a nice bridging format for us to connect the two.

Third, we can ask our text checker to report where it found any misspellings in our word, passing in the range to check, a position to start within the range (so we can do things like “Find Next”), whether it should wrap around once it reaches the end, and what language to use for the dictionary:

```
let misspelledRange = checker.rangeOfMisspelledWord(in: word,  
range: range, startingAt: 0, wrap: false, language: "en")
```

That sends back another Objective-C string range, telling us where the misspelling was found. Even then, there's still one complexity here: Objective-C didn't have any concept of optionals, so instead relied on special values to represent missing data.

In this instance, if the Objective-C range comes back as empty – i.e., if there was no spelling mistake because the string was spelled correctly – then we get back the special value **NSNotFound**.

So, we could check our spelling result to see whether there was a mistake or not like this:

```
let allGood = misspelledRange.location == NSNotFound
```

OK, that's enough API exploration – let's get into our actual project...

Adding to a list of words

The user interface for this app will be made up of three main SwiftUI views: a **NavigationView** showing the word they are spelling from, a **TextField** where they can enter one answer, and a **List** showing all the words they have entered previously.

For now, every time users enter a word into the text field, we'll automatically add it to the list of used words. Later, though, we'll add some validation to make sure the word hasn't been used before, can actually be produced from the root word they've been given, and is a real word and not just some random letters.

Let's start with the basics: we need an array of words they have already used, a root word for them to spell other words from, and a string we can bind to a text field. So, add these three properties to **ContentView** now:

```
@State private var usedWords = [String]()
@State private var rootWord = ""
@State private var newWord = ""
```

As for the body of the view, we're going to start off as simple as possible: a **NavigationView** with **rootWord** for its title, then a couple of sections inside a list:

```
var body: some View {
    NavigationView {
        List {
            Section {
                TextField("Enter your word", text: $newWord)
            }

            Section {
                ForEach(usedWords, id: \.self) { word in
                    Text(word)
                }
            }
        }
    }
}
```

```

        }
    }
    .navigationTitle(rootWord)
}
}

```

Note: Using **id: \self** would cause problems if there were lots of duplicates in **usedWords**, but soon enough we'll be disallowing that so it's not a problem.

Now, our text view has a problem: although we can type into the text box, we can't submit anything from there – there's no way of adding our entry to the list of used words.

To fix that we're going to write a new method called **addNewWord()** that will:

1. Lowercase **newWord** and remove any whitespace
2. Check that it has at least 1 character otherwise exit
3. Insert that word at position 0 in the **usedWords** array
4. Set **newWord** back to be an empty string

Later on we'll add some extra validation between steps 2 and 3 to make sure the word is allowable, but for now this method is straightforward:

```

func addNewWord() {
    // lowercase and trim the word, to make sure we don't add
    // duplicate words with case differences
    let answer =
        newWord.lowercased().trimmingCharacters(in: .whitespacesAndNewl
        ines)

    // exit if the remaining string is empty
    guard answer.count > 0 else { return }

    // extra validation to come
}

```

Project 5: Word Scramble

```
    usedWords.insert(answer, at: 0)
    newWord = ""
}
```

We want to call **addNewWord()** when the user presses return on the keyboard, and in SwiftUI we can do that by adding an **onSubmit()** modifier somewhere in our view hierarchy – it could be directly on the button, but it can be anywhere else in the view because it will be triggered when *any* text is submitted.

onSubmit() needs to be given a function that accepts no parameters and returns nothing, which exactly matches the **addNewWord()** method we just wrote. So, we can pass that in directly by adding this modifier below **navigationTitle()**:

```
.onSubmit(addNewWord)
```

Run the app now and you'll see that things are starting to come together already: we can now type words into the text field, press return, and see them appear in the list.

Inside **addNewWord()** we used **usedWords.insert(answer, at: 0)** for a reason: if we had used **append(answer)** the new words would have appeared at the end of the list where they would probably be off screen, but by inserting words at the start of the array they automatically appear at the top of the list – much better.

Before we put a title up in the navigation view, I'm going to make two small changes to our layout.

First, when we call **addNewWord()** it lowercases the word the user entered, which is helpful because it means the user can't add "car", "Car", and "CAR". However, it looks odd in practice: the text field automatically capitalizes the first letter of whatever the user types, so when they submit "Car" what they see in the list is "car".

To fix this, we can disable capitalization for the text field with another modifier: **autocapitalization()**. Please add this to the text field now:

```
.autocapitalization(.none)
```

The second thing we'll change, just because we can, is to use Apple's SF Symbols icons to show the length of each word next to the text. SF Symbols provides numbers in circles from 0 through 50, all named using the format "x.circle.fill" – so 1.circle.fill, 20.circle.fill.

In this program we'll be showing eight-letter words to users, so if they rearrange all those letters to make a new word the longest it will be is also eight letters. As a result, we can use those SF Symbols number circles just fine – we know that all possible word lengths are covered.

So, we can wrap our word text in a **HStack**, and place an SF Symbol next to it using `Image(systemName:)`` like this:

```
ForEach(usedWords, id: \.self) { word in
    HStack {
        Image(systemName: "\((word.count).circle")
        Text(word)
    }
}
```

If you run the app now you'll see you can type words in the text field, press return, then see them appear in the list with their length icon to the side. Nice!

Now, if you wanted to we could add one sneaky little extra tweak in here. When we submit our text field right now, the text just appears in the list immediately, but we could animate that by modifying the `insert()` call inside `addNewWord()` to this:

```
withAnimation {
    usedWords.insert(answer, at: 0)
}
```

We haven't looked at animations just yet, and we're going to look at them much more shortly,

Project 5: Word Scramble

but that change alone will make our new words slide in much more nicely – I think it's a big improvement!

Running code when our app launches

When Xcode builds an iOS project, it puts your compiled program, your asset catalog, and any other assets into a single directory called a *bundle*, then gives that bundle the name YourAppName.app. This “.app” extension is automatically recognized by iOS and Apple’s other platforms, which is why if you double-click something like Notes.app on macOS it knows to launch the program inside the bundle.

In our game, we’re going to include a file called “start.txt”, which includes over 10,000 eight-letter words that will be randomly selected for the player to work with. This was included in the files for this project that you should have downloaded from GitHub, so please drag start.txt into your project now.

We already defined a property called **rootWord**, which will contain the word we want the player to spell from. What we need to do now is write a new method called **startGame()** that will:

1. Find start.txt in our bundle
2. Load it into a string
3. Split that string into array of strings, with each element being one word
4. Pick one random word from there to be assigned to **rootWord**, or use a sensible default if the array is empty.

Each of those four tasks corresponds to one line of code, but there’s a twist: what if we can’t locate start.txt in our app bundle, or if we can *locate* it but we can’t *load* it? In that case we have a serious problem, because our app is really broken – either we forgot to include the file somehow (in which case our game won’t work), or we included it but for some reason iOS refused to let us read it (in which case our game won’t work, and our app is broken).

Regardless of what caused it, this is a situation that never ought to happen, and Swift gives us a function called **fatalError()** that lets us respond to unresolvable problems really clearly. When

Project 5: Word Scramble

we call **fatalError()** it will – unconditionally and always – cause our app to crash. It will just die. Not “might die” or “maybe die”: it will always just terminate straight away.

I realize that sounds bad, but what it lets us do is important: for problems like this one, such as if we forget to include a file in our project, there is no point trying to make our app struggle on in a broken state. It’s much better to terminate immediately and give us a clear explanation of what went wrong so we can correct the problem, and that’s exactly what **fatalError()** does.

Anyway, let’s take a look at the code – I’ve added comments matching the numbers above:

```
func startGame() {
    // 1. Find the URL for start.txt in our app bundle
    if let startWordsURL = Bundle.main.url(forResource: "start",
withExtension: "txt") {
        // 2. Load start.txt into a string
        if let startWords = try? String(contentsOf: startWordsURL)
{
            // 3. Split the string up into an array of strings,
            splitting on line breaks
            let allWords = startWords.components(separatedBy: "\n")

            // 4. Pick one random word, or use "silkworm" as a
            sensible default
            rootWord = allWords.randomElement() ?? "silkworm"

            // If we are here everything has worked, so we can exit
            return
        }
    }

    // If we are *here* then there was a problem – trigger a
    // crash and report the error
    fatalError("Could not load start.txt from bundle.")
}
```

Running code when our app launches

}

Now that we have a method to load everything for the game, we need to actually call that thing when our view is shown. SwiftUI gives us a dedicated view modifier for running a closure when a view is shown, so we can use that to call `startGame()` and get things moving – add this modifier after `onSubmit()`:

```
.onAppear(perform: startGame)
```

If you run the game now you should see a random eight-letter word at the top of the navigation view. It doesn't really *mean* anything yet, because players can still enter whatever words they want. Let's fix that next...

Validating words with UITextChecker

Now that our game is all set up, the last part of this project is to make sure the user can't enter invalid words. We're going to implement this as four small methods, each of which perform exactly one check: is the word original (it hasn't been used already), is the word possible (they aren't trying to spell "car" from "silkworm"), and is the word real (it's an actual English word).

If you were paying attention you'll have noticed that was only *three* methods – that's because the fourth method will be there to make showing error messages easier.

Anyway, let's start with the first method: this will accept a string as its only parameter, and return true or false depending on whether the word has been used before or not. We already have a **usedWords** array, so we can pass the word into its **contains()** method and send the result back like this:

```
func isOriginal(word: String) -> Bool {  
    !usedWords.contains(word)  
}
```

That's one method down!

The next one is slightly trickier: how can we check whether a random word can be made out of the letters from another random word?

There are a couple of ways we could tackle this, but the easiest one is this: if we create a variable copy of the root word, we can then loop over each letter of the user's input word to see if that letter exists in our copy. If it does, we remove it from the copy (so it can't be used twice), then continue. If we make it to the end of the user's word successfully then the word is good, otherwise there's a mistake and we return false.

So, here's our second method:

```

func isPossible(word: String) -> Bool {
    var tempWord = rootWord

    for letter in word {
        if let pos = tempWord.firstIndex(of: letter) {
            tempWord.remove(at: pos)
        } else {
            return false
        }
    }

    return true
}

```

The final method is harder, because we need to use **UITextChecker** from UIKit. In order to bridge Swift strings to Objective-C strings safely, we need to create an instance of **NSRange** using the UTF-16 count of our Swift string. This isn't nice, I know, but I'm afraid it's unavoidable until Apple cleans up these APIs.

So, our last method will make an instance of **UITextChecker**, which is responsible for scanning strings for misspelled words. We'll then create an **NSRange** to scan the entire length of our string, then call **rangeOfMisspelledWord()** on our text checker so that it looks for wrong words. When that finishes we'll get back *another* **NSRange** telling us where the misspelled word was found, but if the word was OK the location for that range will be the special value **NSNotFound**.

So, here's our final method:

```

func isReal(word: String) -> Bool {
    let checker = UITextChecker()
    let range = NSRange(location: 0, length: word.utf16.count)
    let misspelledRange = checker.rangeOfMisspelledWord(in: word,
        range: range, startingAt: 0, wrap: false, language: "en")

```

Project 5: Word Scramble

```
    return misspelledRange.location == NSNotFound  
}
```

Before we can use those three, I want to add some code to make showing error alerts easier. First, we need some properties to control our alerts:

```
@State private var errorTitle = ""  
@State private var errorMessage = ""  
@State private var showingError = false
```

Now we can add a method that sets the title and message based on the parameters it receives, then flips the **showingError** Boolean to true:

```
func wordError(title: String, message: String) {  
    errorTitle = title  
    errorMessage = message  
    showingError = true  
}
```

We can then pass those directly on to SwiftUI by adding an **alert()** modifier below **.onAppear()**:

```
.alert(errorTitle, isPresented: $showingError) {  
    Button("OK", role: .cancel) {}  
} message: {  
    Text(errorMessage)  
}
```

We've done that several times now, so hopefully it's becoming second nature!

At long last it's time to finish our game: replace the **// extra validation to come** comment in **addNewWord()** with this:

```
guard isOriginal(word: answer) else {
    wordError(title: "Word used already", message: "Be more
original")
    return
}

guard isPossible(word: answer) else {
    wordError(title: "Word not possible", message: "You can't
spell that word from '\(rootWord)'!")
    return
}

guard isReal(word: answer) else {
    wordError(title: "Word not recognized", message: "You can't
just make them up, you know!")
    return
}
```

If you run the app now you should find that it will refuse to let you use words if they fail our tests – trying a duplicate word won’t work, words that can’t be spelled from the root word won’t work, and gibberish words won’t work either.

That’s another app done – good job!

Word Scramble: Wrap up

This project was a last chance to review the fundamentals of SwiftUI before we move on to greater things with the next app. Still, we managed to cover some useful new things, not least **List**, **onAppear**, **Bundle**, **fatalError()**, **UITextChecker**, and more, and you have another app you can extend if you want to.

One thing I want to pick out before we finish is my use of **fatalError()**. If you read code from my own projects on GitHub, or read some of my more advanced tutorials, you'll see that I rely on **fatalError()** a *lot* as a way of forcing code to shut down when something impossible has happened. The key to this technique – the thing that stops it from being recklessly dangerous – is *knowing* when a specific condition ought to be impossible. That comes with time and practice: there is no one quick list of all the places it's a good idea to use **fatalError()**, and instead you'll figure it out with experience.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on:

1. Disallow answers that are shorter than three letters or are just our start word.
2. Add a toolbar button that calls **startGame()**, so users can restart with a new word whenever they want to.
3. Put a text view somewhere so you can track and show the player's score for a given root word. How you calculate score is down to you, but something involving number of words

Word Scramble: Wrap up

and their letter count would be reasonable.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Word Scramble. If you don't already subscribe, you can start a free trial today.

Project 6

Animation

Spruce up your UI with springs, bounces, and more

Animation: Introduction

We're back to another technique project, and this time we're going to be looking at something fast, beautiful, and really under-valued: animations.

Animations are there for a few reasons, of which one definitely is to make our user interfaces look better. However, they are also there to help users understand what's going on with our program: when one window disappears and another slides in, it's clear to the user where the other window has gone to, which means it's also clear where they can look to get it back.

In this technique project we're going to look at a range of animations and transitions with SwiftUI. Some are easy – in fact, you'll be able to get great results almost immediately! – but some require more thinking. *All* will be useful, though, particularly as you work to make sure your apps are attractive and help guide the user's eyes as best as you can.

As with the other days it's a good idea to work in an Xcode project so you can see your code in action, so please create a new App project called Animations.

Creating implicit animations

In SwiftUI, the simplest type of animation is an *implicit* one: we tell our views ahead of time “if someone wants to animate you, here’s how you should respond”, and nothing more.

SwiftUI will then take care of making sure any changes that *do* occur follow the animation you requested. In practice this makes animation trivial – it literally could not be any easier.

Let’s start with an example. This code shows a simple red button with no action, using 50 points of padding and a circular clip shape:

```
Button("Tap Me") {
    // do nothing
}
.padding(50)
.background(.red)
.foregroundColor(.white)
.clipShape(Circle())
```

What we want is for that button to get bigger every time it’s tapped, and we can do that with a new modifier called **scaleEffect()**. You provide this with a value from 0 up, and it will be drawn at that size – a value of 1.0 is equivalent to 100%, i.e. the button’s normal size.

Because we want to change the scale effect value every time the button is tapped, we need to use an **@State** property that will store a **Double**. So, please add this property to your view now:

```
@State private var animationAmount = 1.0
```

Now we can make the button use that for its scale effect, by adding this modifier:

```
.scaleEffect(animationAmount)
```

Finally, when the button is tapped we want to increase the animation amount by 1, so use this for the button’s action:

```
animationAmount += 1
```

If you run that code you'll see that you can tap the button repeatedly to have it scale up and up. It won't get redrawn at increasingly high resolutions, so as the button gets bigger you'll see it gets a bit blurry, but that's OK.

Now, the human eye is highly sensitive to movement – we're extremely good at detecting when things move or change their appearance, which is what makes animation both so important and so pleasing. So, we can ask SwiftUI to create an implicit animation for our changes so that all the scaling happens smoothly by adding an **animation()** modifier to the button:

```
.animation(.default, value: animationAmount)
```

That asks SwiftUI to apply a default animation whenever the value of **animationAmount** changes, and immediately you'll see that tapping the button now causes it to scale up with an animation.

That implicit animation takes effect on all properties of the view that change, meaning that if we attach more animating modifiers to the view then they will all change together. For example, we could add a second new modifier to the button, **.blur()**, which lets us add a Gaussian blur with a special radius – add this *before* the **animation()** modifier:

```
.blur(radius: (animationAmount - 1) * 3)
```

A radius of **(animationAmount - 1) * 3** means the blur radius will start at 0 (no blur), but then move to 3 points, 6 points, 9 points, and beyond as you tap the button.

If you run the app again you'll see that it now scales and blurs smoothly.

The point is that nowhere have we said what each frame of the animation should look like, and we haven't even said when SwiftUI should start and finish the animation. Instead, our animation becomes a function of our state just like the views themselves.

Project 6: Animation

Customizing animations in SwiftUI

When we attach the `animation()` modifier to a view, SwiftUI will automatically animate any changes that happen to that view using whatever is the default system animation, whenever the value we're watching changes. In practice, that is an “ease in, ease out” animation, which means iOS will start the animation slow, make it pick up speed, then slow down as it approaches its end.

We can control the type of animation used by passing in different values to the modifier. For example, we could use `.easeOut` to make the animation start fast then slow down to a smooth stop:

```
.animation(.easeOut, value: animationAmount)
```

Tip: If you were curious, implicit animations always need to watch a particular value otherwise animations would be triggered for every small change – even rotating the device from portrait to landscape would trigger the animation, which would look strange.

There are even spring animations, that cause the movement to overshoot then return to settle at its target. You can control the initial stiffness of the spring (which sets its initial velocity when the animation starts), and also how fast the animation should be “damped” – lower values cause the spring to bounce back and forth for longer.

For example, this makes our button scale up quickly then bounce:

```
.animation(.interpolatingSpring(stiffness: 50, damping: 1),
value: animationAmount)
```

For more precise control, we can customize the animation with a duration specified as a number of seconds. So, we could get an ease-in-out animation that lasts for two seconds like this:

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

Project 6: Animation

```
var body: some View {
    Button("Tap Me") {
        animationAmount += 1
    }
    .padding(50)
    .background(.red)
    .foregroundColor(.white)
    .clipShape(Circle())
    .scaleEffect(animationAmount)
    .animation(.easeInOut(duration: 2), value: animationAmount)
}
}
```

When we say **.easeInOut(duration: 2)** we're actually creating an instance of an **Animation** struct that has its own set of modifiers. So, we can attach modifiers directly to the animation to add a delay like this:

```
.animation(
    .easeInOut(duration: 2)
    .delay(1),
    value: animationAmount
)
```

With that in place, tapping the button will now wait for a second before executing a two-second animation.

We can also ask the animation to repeat a certain number of times, and even make it bounce back and forward by setting **autoreverses** to true. This creates a one-second animation that will bounce up and down before reaching its final size:

```
.animation(
    .easeInOut(duration: 1)
```

```
.repeatCount(3, autoreverses: true),  
value: animationAmount  
)
```

If we had set repeat count to 2 then the button would scale up then down again, then jump immediately back up to its larger scale. This is because ultimately the button must match the state of our program, regardless of what animations we apply – when the animation finishes the button must have whatever value is set in **animationAmount**.

For continuous animations, there is a **repeatForever()** modifier that can be used like this:

```
.animation(  
    .easeInOut(duration: 1)  
        .repeatForever(autoreverses: true),  
    value: animationAmount  
)
```

We can use these **repeatForever()** animations in combination with **onAppear()** to make animations that start immediately and continue animating for the life of the view.

To demonstrate this, we're going to remove the animation from the button itself and instead apply it an overlay to make a sort of pulsating circle around the button. Overlays are created using an **overlay()** modifier, which lets us create new views at the same size and position as the view we're overlaying.

So, first add this **overlay()** modifier to the button before the **animation()** modifier:

```
.overlay(  
    Circle()  
        .stroke(.red)  
        .scaleEffect(animationAmount)  
        .opacity(2 - animationAmount)  
)
```

Project 6: Animation

That makes a stroked red circle over our button, using an opacity value of **2 - animationAmount** so that when **animationAmount** is 1 the opacity is 1 (it's opaque) and when **animationAmount** is 2 the opacity is 0 (it's transparent).

Next, remove the **scaleEffect()** and **blur()** modifiers from the button and comment out the **animationAmount += 1** action part too, because we don't want that to change any more, and move its animation modifier up to the circle inside the overlay:

```
.overlay(  
    Circle()  
        .stroke(.red)  
        .scaleEffect(animationAmount)  
        .opacity(2 - animationAmount)  
        .animation(  
            .easeOut(duration: 1)  
                .repeatForever(autoreverses: false),  
            value: animationAmount  
        )  
)
```

I've switched **autoreverses** to false, but otherwise it's the same animation.

Finally, add an **onAppear()** modifier to the button, which will set **animationAmount** to 2:

```
.onAppear {  
    animationAmount = 2  
}
```

Because the overlay circle uses that for a “repeat forever” animation without autoreversing, you'll see the overlay circle scale up and fade out continuously.

Your finished code should look like this:

```
Button("Tap Me") {
```

```
// animationAmount += 1
}
.padding(50)
.background(.red)
.foregroundColor(.white)
.clipShape(Circle())
.overlay(
    Circle()
        .stroke(.red)
        .scaleEffect(animationAmount)
        .opacity(2 - animationAmount)
        .animation(
            .easeInOut(duration: 1)
                .repeatForever(autoreverses: false),
            value: animationAmount
        )
)
.onAppear {
    animationAmount = 2
}
```

Given how little work that involves, it creates a remarkably attractive effect!

Animating bindings

The `animation()` modifier can be applied to any SwiftUI binding, which causes the value to animate between its current and new value. This even works if the data in question isn't really something that sounds like it can be animated, such as a Boolean – you can mentally imagine animating from 1.0 to 2.0 because we could do 1.05, 1.1, 1.15, and so on, but going from “false” to “true” sounds like there’s no room for in between values.

This is best explained with some working code to look at, so here’s a view with a **VStack**, a **Stepper**, and a **Button**:

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

    var body: some View {
        VStack {
            Stepper("Scale amount", value:
$animationAmount.animation(), in: 1...10)

            Spacer()

            Button("Tap Me") {
                animationAmount += 1
            }
            .padding(40)
            .background(.red)
            .foregroundColor(.white)
            .clipShape(Circle())
            .scaleEffect(animationAmount)
        }
    }
}
```

As you can see, the stepper can move **animationAmount** up and down, and tapping the button will add 1 to it – they are both tied to the same data, which in turn causes the size of the button to change. However, tapping the button changes **animationCount** immediately, so the button will just jump up to its larger size. In contrast, the stepper is bound to **\$animationAmount.animation()**, which means SwiftUI will automatically animate its changes.

Now, as an experiment I'd like you to change the start of the **body** to this:

```
var body: some View {
    print(animationAmount)

    return VStack {
```

Because we have some non-view code in there, we need to add **return** before the **VStack** so Swift understands which part is the view that is being sent back. But adding **print(animationAmount)** is important, and to see why I'd like you to run the program again and try manipulating the stepper.

What you should see is that it prints out 2.0, 3.0, 4.0, and so on. At the same time, the button is scaling up or down smoothly – it doesn't just jump straight to scale 2, 3, and 4. What's *actually* happening here is that SwiftUI is examining the state of our view before the binding changes, examining the target state of our views *after* the binding changes, then applying an animation to get from point A to point B.

This is why we can animate a Boolean changing: Swift isn't somehow inventing new values between false and true, but just animating the view changes that occur as a result of the change.

These binding animations use a similar **animation()** modifier that we use on views, so you can go to town with animation modifiers if you want to:

```
Stepper("Scale amount", value: $animationAmount.animation()
    .easeInOut(duration: 1)
```

Project 6: Animation

```
.repeatCount(3, autoreverses: true)  
, in: 1...10)
```

Tip: With this variant of the **animation()** modifier, we don't need to specify which value we're watching for changes – it's literally attached to the value it should watch!

These binding animations effectively turn the tables on implicit animations: rather than setting the animation on a view and implicitly animating it with a state change, we now set nothing on the view and explicitly animate it with a state change. In the former, the state change has no idea it will trigger an animation, and in the latter the view has no idea it will be animated – both work and both are important.

Creating explicit animations

You've seen how SwiftUI lets us create implicit animations by attaching the `animation()` modifier to a view, and how it also lets us create animated binding changes by adding the `animation()` modifier to a binding, but there's a third useful way we can create animations: explicitly asking SwiftUI to animate changes occurring as the result of a state change.

This still doesn't mean we create each frame of the animation by hand – that remains SwiftUI's job, and it continues to figure out the animation by looking at the state of our views before and after the state change was applied.

Now, though, we're being explicit that we want an animation to occur when some arbitrary state change occurs: it's not attached to a binding, and it's not attached to a view, it's just us explicitly asking for a particular animation to occur because of a state change.

To demonstrate this, let's return to a simple button example again:

```
struct ContentView: View {
    var body: some View {
        Button("Tap Me") {
            // do nothing
        }
        .padding(50)
        .background(.red)
        .foregroundColor(.white)
        .clipShape(Circle())
    }
}
```

When that button is tapped, we're going to make it spin around with a 3D effect. This requires another new modifier, `rotation3DEffect()`, which can be given a rotation amount in degrees as well as an axis that determines how the view rotates. Think of this axis like a skewer through your view:

Project 6: Animation

- If we skewer the view through the X axis (horizontally) then it will be able to spin forwards and backwards.
- If we skewer the view through the Y axis (vertically) then it will be able to spin left and right.
- If we skewer the view through the Z axis (depth) then it will be able to rotate left and right.

Making this work requires some state we can modify, and rotation degrees are specified as a **Double**. So, please add this property now:

```
@State private var animationAmount = 0.0
```

Next, we're going to ask the button to rotate by **animationAmount** degrees along its Y axis, which means it will spin left and right. Add this modifier to the button now:

```
.rotation3DEffect(.degrees(animationAmount), axis: (x: 0, y: 1, z: 0))
```

Now for the important part: we're going to add some code to the button's action so that it adds 360 to **animationAmount** every time it's tapped.

If we just write **animationAmount += 360** then the change will happen immediately, because there is no animation modifier attached to the button. This is where explicit animations come in: if we use a **withAnimation()** closure then SwiftUI will ensure any changes resulting from the new state will automatically be animated.

So, put this in the button's action now:

```
withAnimation {
    animationAmount += 360
}
```

Run that code now and I think you'll be impressed by how good it looks – every time you tap the button it spins around in 3D space, and it was so easy to write. If you have time,

experiment a little with the axes so you can really understand how they work. In case you were curious, you can use more than one axis at once.

withAnimation() can be given an animation parameter, using all the same animations you can use elsewhere in SwiftUI. For example, we could make our rotation effect use a spring animation using a **withAnimation()** call like this:

```
withAnimation(.interpolatingSpring(stiffness: 5, damping: 1)) {  
    animationAmount += 360  
}
```

Controlling the animation stack

At this point, I want to put together two things that you already understand individually, but together might hurt your head a little.

Previously we looked at how the order of modifiers matters. So, if we wrote code like this:

```
Button("Tap Me") {  
    // do nothing  
}  
.background(.blue)  
.frame(width: 200, height: 200)  
.foregroundColor(.white)
```

The result would look different from code like this:

```
Button("Tap Me") {  
    // do nothing  
}  
.frame(width: 200, height: 200)  
.background(.blue)  
.foregroundColor(.white)
```

This is because if we color the background before adjusting the frame, only the original space is colored rather than the expanded space. If you recall, the underlying reason for this is the way SwiftUI wraps views with modifiers, allowing us to apply the same modifier multiple times – we repeated **background()** and **padding()** several times to create a striped border effect.

That's concept one: modifier order matters, because SwiftUI wraps views with modifiers in the order they are applied.

Concept two is that we can apply an **animation()** modifier to a view in order to have it implicitly animate changes.

To demonstrate this, we could modify our button code so that it shows different colors depending on some state. First, we define the state:

```
@State private var enabled = false
```

We can toggle that between true and false inside our button's action:

```
enabled.toggle()
```

Then we can use a conditional value inside the **background()** modifier so the button is either blue or red:

```
.background(enabled ? .blue : .red)
```

Finally, we add the **animation()** modifier to the button to make those changes animate:

```
.animation(.default, value: enabled)
```

If you run the code you'll see that tapping the button animates its color between blue and red.

So: order modifier matters and we can attach one modifier several times to a view, and we can cause implicit animations to occur with the **animation()** modifier. All clear so far?

Right. Brace yourself, because this might hurt.

You can attach the `animation()` modifier several times, and the order in which you use it matters.

To demonstrate this, I'd like you to add this modifier to your button, after all the other modifiers:

```
.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))
```

That will cause the button to move between a square and a rounded rectangle depending on the

Project 6: Animation

state of the **enabled** Boolean.

When you run the program, you'll see that tapping the button causes it to animate between red and blue, but *jump* between square and rounded rectangle – that part *doesn't* animate.

Hopefully you can see where we're going next: I'd like you to move the **clipShape()** modifier before the animation, like this:

```
.frame(width: 200, height: 200)
.background(enabled ? .blue : .red)
.foregroundColor(.white)
.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))
.animation(.default, value: enabled)
```

And now when you run the code both the background color and clip shape animate.

So, the order in which we apply animations matters: only changes that occur *before* the **animation()** modifier get animated.

Now for the fun part: if we apply multiple **animation()** modifiers, each one controls everything before it up to the next animation. This allows us to animate state changes in all sorts of different ways rather than uniformly for all properties.

For example, we could make the color change happen with the default animation, but use an interpolating spring for the clip shape:

```
Button("Tap Me") {
    enabled.toggle()
}
.frame(width: 200, height: 200)
.background(enabled ? .blue : .red)
.animation(.default, value: enabled)
.foregroundColor(.white)
.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))
```

```
.animation(.interpolatingSpring(stiffness: 10, damping: 1),  
value: enabled)
```

You can have as many **animation()** modifiers as you need to construct your design, which lets us split one state change into as many segments as we need.

For even more control, it's possible to disable animations entirely by passing **nil** to the modifier. For example, you might want the color change to happen immediately but the clip shape to retain its animation, in which case you'd write this:

```
Button("Tap Me") {  
    enabled.toggle()  
}  
.frame(width: 200, height: 200)  
.background(enabled ? .blue : .red)  
.animation(nil, value: enabled)  
.foregroundColor(.white)  
.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))  
.animation(.interpolatingSpring(stiffness: 10, damping: 1),  
value: enabled)
```

That kind of control wouldn't be possible without multiple **animation()** modifiers – if you tried to move **background()** after the animation you'd find that it would just undo the work of **clipShape()**.

Animating gestures

SwiftUI lets us attach gestures to any views, and the effects of those gestures can also be animated. We get a range of gestures to work with, such as tap gestures to let any view respond to taps, drag gestures that respond to us dragging a finger over a view, and more.

We'll be looking at gestures in more detail later on, but for now let's try something relatively simple: a card that we can drag around the screen, but when we let go it snaps back into its original location.

First, our initial layout:

```
struct ContentView: View {
    var body: some View {
        LinearGradient(gradient: Gradient(colors: [.yellow, .red]),
                      startPoint: .topLeading, endPoint: .bottomTrailing)
            .frame(width: 300, height: 200)
            .clipShape(RoundedRectangle(cornerRadius: 10))
    }
}
```

That draws a card-like view in the center of the screen. We want to move that around the screen based on the location of our finger, and that requires three steps.

First, we need some state to store the amount of their drag:

```
@State private var dragAmount = CGSize.zero
```

Second, we want to use that size to influence the card's position on-screen. SwiftUI gives us a dedicated modifier for this called **offset()**, which lets us adjust the X and Y coordinate of a view without moving other views around it. You can pass in discrete X and Y coordinates if you want to, but – by no mere coincidence – **offset()** can also take a **CGSize** directly.

So, step two is to add this modifier to the card gradient:

```
.offset(dragAmount)
```

Now comes the important part: we can create a **DragGesture** and attach it to the card. Drag gestures have two extra modifiers that are useful to us here: **onChanged()** lets us run a closure whenever the user moves their finger, and **onEnded()** lets us run a closure when the user lifts their finger off the screen, ending the drag.

Both of those closures are given a single parameter, which describes the drag operation – where it started, where it is currently, how far it moved, and so on. For our **onChanged()** modifier we're going to read the *translation* of the drag, which tells us how far it's moved from the start point – we can assign that directly to **dragAmount** so that our view moves along with the gesture. For **onEnded()** we're going to ignore the input entirely, because we'll be setting **dragAmount** back to zero.

So, add this modifier to the linear gradient now:

```
.gesture(  
    DragGesture()  
        .onChanged { dragAmount = $0.translation }  
        .onEnded { _ in dragAmount = .zero }  
)
```

If you run the code you'll see you can now drag the gradient card around, and when you release the drag it will jump back to the center. The card has its offset determined by **dragAmount**, which in turn is being set by the drag gesture.

Now that everything works we can bring that movement to life with some animation, and we have two options: add an implicit animation that will animate the drag *and* the release, or add an explicit animation to animate just the release.

To see the former in action, add this modifier to the linear gradient:

```
.animation(.spring(), value: dragAmount)
```

Project 6: Animation

As you drag around, the card will move to the drag location with a slight delay because of the spring animation, but it will also gently overshoot if you make sudden movements.

To see *explicit* animations in action, remove that **animation()** modifier and change your existing **onEnded()** drag gesture code to this:

```
.onEnded { _ in
    withAnimation(.spring()) {
        dragAmount = .zero
    }
}
```

Now the card will follow your drag immediately (because that's not being animated), but when you release it *will* animate.

If we combine offset animations with drag gestures and a little delay, we can create remarkably fun animations without a lot of code.

To demonstrate this, we could write the text “Hello SwiftUI” as a series of individual letters, each one with a background color and offset that is controlled by some state. Strings are just slightly fancy arrays of characters, so we can get a *real* array from a string like this:

Array("Hello SwiftUI").

Anyway, try this out and see what you think:

```
struct ContentView: View {
    let letters = Array("Hello SwiftUI")
    @State private var enabled = false
    @State private var dragAmount = CGSize.zero

    var body: some View {
        HStack(spacing: 0) {
            ForEach(0 ..<letters.count) { num in
```

```
Text(String(letters[num]))
    .padding(5)
    .font(.title)
    .background(enabled ? .blue : .red)
    .offset(dragAmount)
    .animation(.default.delay(Double(num) / 20), value:
dragAmount)
}
}
.gesture(
    DragGesture()
        .onChanged { dragAmount = $0.translation }
        .onEnded { _ in
            dragAmount = .zero
            enabled.toggle()
        }
    )
}
}
```

If you run that code you'll see that any letter can be dragged around to have the whole string follow suit, with a brief delay causing a snake-like effect. SwiftUI will also add in color changing as you release the drag, animating between blue and red even as the letters move back to the center.

Showing and hiding views with transitions

One of the most powerful features of SwiftUI is the ability to customize the way views are shown and hidden. Previously you've seen how we can use regular `if` conditions to include views conditionally, which means when that condition changes we can insert or remove views from our view hierarchy.

Transitions control how this insertion and removal takes place, and we can work with the built-in transitions, combine them in different ways, or even create wholly custom transitions.

To demonstrate this, here's a `VStack` with a button and a rectangle:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Button("Tap Me") {
                // do nothing
            }

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

We can make the rectangle appear only when a certain condition is satisfied. First, we add some state we can manipulate:

```
@State private var isShowingRed = false
```

Next we use that state as a condition for showing our rectangle:

```
if isShowingRed {
    Rectangle()
        .fill(.red)
        .frame(width: 200, height: 200)
}
```

Finally we can toggle **isShowingRed** between true and false in the button's action:

```
isShowingRed.toggle()
```

If you run the program, you'll see that pressing the button shows and hides the red square. There's no animation; it just appears and disappears abruptly.

We can get SwiftUI's default view transition by wrapping the state change using **withAnimation()**, like this:

```
withAnimation {
    isShowingRed.toggle()
}
```

With that small change, the app now fades the red rectangle in and out, while also moving the button up to make space. It looks OK, but we can do better with the **transition()** modifier.

For example, we could have the rectangle scale up and down as it is shown just by adding the **transition()** modifier to it:

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

Now tapping the button looks much better: the rectangle scales up as the button makes space,

Project 6: Animation

then scales down when tapped again.

There are a handful of other transitions you can try if you want to experiment. A useful one is **.asymmetric**, which lets us use one transition when the view is being shown and another when it's disappearing. To try it out, replace the rectangle's existing transition with this:

```
.transition(.asymmetric(insertion: .scale, removal: .opacity))
```

Building custom transitions using ViewModifier

It's possible – and actually surprisingly easy – to create wholly new transitions for SwiftUI, allowing us to add and remove views using entirely custom animations.

This functionality is made possible by the **.modifier** transition, which accepts any view modifier we want. The catch is that we need to be able to instantiate the modifier, which means it needs to be one we create ourselves.

To try this out, we could write a view modifier that lets us mimic the Pivot animation in Keynote – it causes a new slide to rotate in from its top-left corner. In SwiftUI-speak, that means creating a view modifier that causes our view to rotate in from one corner, without escaping the bounds it's supposed to be in. SwiftUI actually gives us modifiers to do just that: **rotationEffect()** lets us rotate a view in 2D space, and **clipped()** stops the view from being drawn outside of its rectangular space.

rotationEffect() is similar to **rotation3DEffect()**, except it always rotates around the Z axis. However, it also gives us the ability to control the *anchor point* of the rotation – which part of the view should be fixed in place as the center of the rotation. SwiftUI gives us a **UnitPoint** type for controlling the anchor, which lets us specify an exact X/Y point for the rotation or use one of the many built-in options – **.topLeading**, **.bottomTrailing**, **.center**, and so on.

Let's put all this into code by creating a **CornerRotateModifier** struct that has an anchor point to control where the rotation should take place, and an amount to control how much rotation should be applied:

```
struct CornerRotateModifier: ViewModifier {
    let amount: Double
    let anchor: UnitPoint

    func body(content: Content) -> some View {
        content
```

Project 6: Animation

```
.rotationEffect(.degrees(amount), anchor: anchor)
.clipped()
}
}
```

The addition of **clipped()** there means that when the view rotates the parts that are lying outside its natural rectangle don't get drawn.

We can try that straight away using the **.modifier** transition, but it's a little unwieldy. A *better* idea is to wrap that in an extension to **AnyTransition**, making it rotate from -90 to 0 on its top leading corner:

```
extension AnyTransition {
    static var pivot: AnyTransition {
        .modifier(
            active: CornerRotateModifier(amount: -90,
                anchor: .topLeading),
            identity: CornerRotateModifier(amount: 0,
                anchor: .topLeading)
        )
    }
}
```

With that in place we now attach the pivot animation to any view using this:

```
.transition(.pivot)
```

For example, we could use the **onTapGesture()** modifier to make a red rectangle pivot onto the screen, like this:

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

Building custom transitions using ViewModifier

```
var body: some View {
    ZStack {
        Rectangle()
            .fill(.blue)
            .frame(width: 200, height: 200)

        if isShowingRed {
            Rectangle()
                .fill(.red)
                .frame(width: 200, height: 200)
                .transition(.pivot)
        }
    }
    .onTapGesture {
        withAnimation {
            isShowingRed.toggle()
        }
    }
}
}
```

Animation: Wrap up

This technique project started off easier, took a few twists and turns, and progressed into more advanced animations, but I hope it's given you an idea of just how powerful – and how flexible! – SwiftUI's animation system is.

As I've said previously, animation is about both making your app look great and also adding extra meaning. So, rather than making a view disappear abruptly, can you add a transition to help the user understand something is changing?

Also, don't forget what it looks like to be *playful* in your user interface. My all-time #1 favorite iOS animation is one that Apple ditched when they moved to iOS 7, and it was the animation for deleting passes in the Wallet app – a metal shredder appeared and cut your pass into a dozen strips that then dropped away. It only took a fraction of a second more than the current animation, but it was beautiful and fun too!

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

Go back to the Guess the Flag project and add some animation:

1. When you tap a flag, make it spin around 360 degrees on the Y axis.
2. Make the other two buttons fade out to 25% opacity.
3. Add a third effect of your choosing to the two flags the user didn't choose – maybe make

them scale down? Or flip in a different direction? Experiment!

These challenges aren't easy. They take only a few lines of code, but you'll need to think carefully about what modifiers you use to get the exact animations you want. Try adding an `@State` property to track which flag the user tapped on, then using that inside conditional modifiers for rotation, fading, and whatever you decide for the third challenge.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Animations. If you don't already subscribe, you can start a free trial today.

Project 7

iExpense

Bring in a second view with this expense tracking app

iExpense: Introduction

Our next two projects are going to start pushing your SwiftUI skills beyond the basics, as we explore apps that have multiple screens, that load and save user data, and have more complex user interfaces.

In this project we're going to build iExpense, which is an expense tracker that separates personal costs from business costs. At its core this is an app with a form (how much did you spend?) and a list (here are the amounts you spent), but in order to accomplish those two things you're going to need to learn how to:

- Present and dismiss a second screen of data.
- Delete rows from a list
- Save and load user data

...and more.

There's lots to do, so let's get started: create a new iOS app using the App template, naming it "iExpense", then making sure to select iOS 15 as your deployment target. We'll be using that for the main project, but first lets take a closer look at the new techniques required for this project...

Why @State only works with structs

SwiftUI's **State** property wrapper is designed for simple data that is local to the current view, but as soon as you want to share data between views it stops being useful.

Let's break this down with some code – here's a struct to store a user's first and last name:

```
struct User {  
    var firstName = "Bilbo"  
    var lastName = "Baggins"  
}
```

We can now use that in a SwiftUI view by creating an **@State** property and attaching things to **\$user.firstName** and **\$user.lastName**, like this:

```
struct ContentView: View {  
    @State private var user = User()  
  
    var body: some View {  
        VStack {  
            Text("Your name is \(user.firstName) \(user.lastName).")  
  
            TextField("First name", text: $user.firstName)  
            TextField("Last name", text: $user.lastName)  
        }  
    }  
}
```

That all works: SwiftUI is smart enough to understand that one object contains all our data, and will update the UI when either value changes. Behind the scenes, what's actually happening is that each time a value inside our struct changes the *whole* struct changes – it's

like a new user every time we type a key for the first or last name. That might sound wasteful, but it's actually extremely fast.

Previously we looked at the differences between classes and structs, and there were two important differences I mentioned. First, that structs always have unique owners, whereas with classes multiple things can point to the same value. And second, that classes don't need the **mutating** keyword before methods that change their properties, because you *can* change properties of constant classes.

In practice, what this means is that if we have two SwiftUI views and we send them both the same struct to work with, they actually each have a unique copy of that struct; if one changes it, the other won't see that change. On the other hand, if we create an instance of a *class* and send that to both views, they *will* share changes.

For SwiftUI developers, what this means is that if we want to share data between multiple views – if we want two or more views to point to the same data so that when one changes they all get those changes – we need to use classes rather than structs.

So, please change the **User** struct to be a class. From this:

```
struct User {
```

To this:

```
class User {
```

Now run the program again and see what you think.

Spoiler: it doesn't work any more. Sure, we can type into the text fields just like before, but the text view above doesn't change.

When we use **@State**, we're asking SwiftUI to watch a property for changes. So, if we change a string, flip a Boolean, add to an array, and so on, the property has changed and SwiftUI will re-invoke the **body** property of the view.

Project 7: iExpense

When **User** was a struct, every time we modified a property of that struct Swift was actually creating a new instance of the struct. **@State** was able to spot that change, and automatically reloaded our view. Now that we have a class, that behavior no longer happens: Swift can just modify the value directly.

Remember how we had to use the **mutating** keyword for struct methods that modify properties? This is because if we create the struct's properties as variable but the struct itself is constant, we can't change the properties – Swift needs to be able to destroy and recreate the whole struct when a property changes, and that isn't possible for constant structs. Classes *don't* need the **mutating** keyword, because even if the class instance is marked as constant Swift can still modify variable properties.

I know that all sounds terribly theoretical, but here's the twist: now that **User** is a class the property itself isn't changing, so **@State** doesn't notice anything and can't reload the view. Yes, the values *inside* the class are changing, but **@State** doesn't monitor those, so effectively what's happening is that the values inside our class are being changed but the view isn't being reloaded to reflect that change.

To fix this, it's time to leave **@State** behind. Instead we need a more powerful property wrapper called **@StateObject** – let's look at that now...

Sharing SwiftUI state with `@StateObject`

If you want to use a class with your SwiftUI data – which you *will* want to do if that data should be shared across more than one view – then SwiftUI gives us three property wrappers that are useful: `@StateObject`, `@ObservedObject`, and `@EnvironmentObject`. We'll be looking at environment objects later on, but for now let's focus on the first two.

Here's some code that creates a `User` class, and shows that user data in a view:

```
class User {
    var firstName = "Bilbo"
    var lastName = "Baggins"
}

struct ContentView: View {
    @State private var user = User()

    var body: some View {
        VStack {
            Text("Your name is \u{20}(\u{20}user.firstName) \u{20}(\u{20}user.lastName).\u{20}")

            TextField("First name", text: $user.firstName)
            TextField("Last name", text: $user.lastName)
        }
    }
}
```

However, that code won't work as intended: we've marked the `user` property with `@State`, which is designed to track local structs rather than external classes. As a result, we can type into the text fields but the text view above won't be updated.

Project 7: iExpense

To fix this, we need to tell SwiftUI when interesting parts of our class have changed. By “interesting parts” I mean parts that should cause SwiftUI to reload any views that are watching our class – it’s possible you might have lots of properties inside your class, but only a few should be exposed to the wider world in this way.

Our **User** class has two properties: **firstName** and **lastName**. Whenever either of those two changes, we want to notify any views that are watching our class that a change has happened so they can be reloaded. We can do this using the **@Published** property observer, like this:

```
class User {  
    @Published var firstName = "Bilbo"  
    @Published var lastName = "Baggins"  
}
```

@Published is more or less half of **@State**: it tells Swift that whenever either of those two properties changes, it should send an announcement out to any SwiftUI views that are watching that they should reload.

How do those views know which classes might send out these notifications? That’s another property wrapper, **@StateObject**, which is the other half of **@State** – it tells SwiftUI that we’re creating a new class instance that should be watched for any change announcements.

So, change the **user** property to this:

```
@StateObject var user = User()
```

I removed the **private** access control there, but whether or not you use it depends on your usage – if you’re intending to share that object with other views then marking it as **private** will just cause confusion.

Now that we’re using **@StateObject**, our code will no longer compile. It’s not a problem, and in fact it’s expected and easy to fix: the **@StateObject** property wrapper can only be used on types that conform to the **ObservableObject** protocol. This protocol has no requirements, and really all it means is “we want other things to be able to monitor this for changes.”

So, modify the `User` class to this:

```
class User: ObservableObject {  
    @Published var firstName = "Bilbo"  
    @Published var lastName = "Baggins"  
}
```

Our code will now compile again, and, even better, it will now actually *work* again – you can run the app and see the text view update when either text field is changed.

As you've seen, rather than just using `@State` to declare local state, we now take three steps:

- Make a class that conforms to the `ObservableObject` protocol.
- Mark some properties with `@Published` so that any views using the class get updated when they change.
- Create an instance of our class using the `@StateObject` property wrapper.

The end result is that we can have our state stored in an external object, and, even better, we can now use that object in multiple views and have them all point to the same values.

However, there is a catch. Like I said earlier, `@StateObject` tells SwiftUI that we're creating a new class instance that should be watched for any change announcements, but that should only be used when you're *creating* the object like we are with our `User` instance.

When you want to use a class instance elsewhere – when you've created it in view A using `@StateObject` and want to use that same object in view B – you use a slightly different property wrapper called `@ObservedObject`. That's the only difference: when *creating* the shared data use `@StateObject`, but when you're just *using* it in a different view you should use `@ObservedObject` instead.

Showing and hiding views

There are several ways of showing views in SwiftUI, and one of the most basic is a *sheet*: a new view presented on top of our existing one. On iOS this automatically gives us a card-like presentation where the current view slides away into the distance a little and the new view animates in on top.

Sheets work much like alerts, in that we don't present them directly with code such as `mySheet.present()` or similar. Instead, we define the *conditions* under which a sheet should be shown, and when those conditions become true or false the sheet will either be presented or dismissed respectively.

Let's start with a simple example, which will be showing one view from another using a sheet. First, we create the view we want to show inside a sheet, like this:

```
struct SecondView: View {  
    var body: some View {  
        Text("Second View")  
    }  
}
```

There's nothing special about that view at all – it doesn't know it's going to be shown in a sheet, and doesn't *need* to know it's going to be shown in a sheet.

Next we create our initial view, which will show the second view. We'll make it simple, then add to it:

```
struct ContentView: View {  
    var body: some View {  
        Button("Show Sheet") {  
            // show the sheet  
        }  
    }  
}
```

Filling that in requires four steps, and we'll tackle them individually.

First, we need some state to track whether the sheet is showing. Just as with alerts, this can be a simple Boolean, so add this property to **ContentView** now:

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

Second, we need to toggle that when our button is tapped, so replace the `// show the sheet` comment with this:

```
showingSheet.toggle()
```

Third, we need to attach our sheet somewhere to our view hierarchy. If you remember, we show alerts using **isPresented** with a two-way binding to our state property, and we use something almost identical here: **sheet(isPresented:)**.

sheet() is a modifier just like **alert()**, so please add this modifier to our button now:

```
.sheet(isPresented: $showingSheet) {
    // contents of the sheet
}
```

Fourth, we need to decide what should actually be in the sheet. In our case, we already know exactly what we want: we want to create and show an instance of **SecondView**. In code that means writing **SecondView()**, then... er... well, that's it.

So, the finished **ContentView** struct should look like this:

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

    var body: some View {
        Button("Show Sheet") {

```

Project 7: iExpense

```
        showingSheet.toggle()
    }
    .sheet(isPresented: $showingSheet) {
        SecondView()
    }
}
```

If you run the program now you'll see you can tap the button to have our second view slide upwards from the bottom, and you can then drag that down to dismiss it.

When you create a view like this, you can pass in any parameters it needs to work. For example, we could require that **SecondView** be sent a name it can display, like this:

```
struct SecondView: View {
    let name: String

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

And now just using **SecondView()** in our sheet isn't good enough – we need to pass in a name string to be shown. For example, we could pass in my Twitter username like this:

```
.sheet(isPresented: $showingSheet) {
    SecondView(name: "@twostraws")
}
```

Now the sheet will show “Hello, @twostraws”.

Swift is doing a ton of work on our behalf here: as soon as we said that **SecondView** has a name property, Swift ensured that our code wouldn't even build until all instances of

SecondView() became **SecondView(name: "some name")**, which eliminates a whole range of possible errors.

Before we move on, there's one more thing I want to demonstrate, which is how to make a view dismiss itself. Yes, you've seen that the user can just swipe downwards, but sometimes you will want to dismiss views programmatically – to make the view go away because a button was pressed, for example.

To dismiss another view we need another property wrapper – and yes, I realize that so often the solution to a problem in SwiftUI is to use another property wrapper.

Anyway, this new one is called **@Environment**, and it allows us to create properties that store values provided to us externally. Is the user in light mode or dark mode? Have they asked for smaller or larger fonts? What timezone are they on? All these and more are values that come from the environment, and in this instance we're going to ask the environment to dismiss our view.

Yes, we need to ask the environment to dismiss our view, because it might have been presented in any number of different ways. So, we're effectively saying “hey, figure out how my view was presented, then dismiss it appropriately.”

To try it out add this property to **SecondView**, which creates a property called **dismiss** based on a value from the environment:

```
@Environment(\.dismiss) var dismiss
```

Now replace the text view in **SecondView** with this button:

```
Button("Dismiss") {
    dismiss()
}
```

Anyway, with that button in place, you should now find you can show and hide the sheet using button presses.

Project 7: iExpense

Deleting items using onDelete()

SwiftUI gives us the `onDelete()` modifier for us to use to control how objects should be deleted from a collection. In practice, this is almost exclusively used with **List** and **ForEach**: we create a list of rows that are shown using **ForEach**, then attach `onDelete()` to that **ForEach** so the user can remove rows they don't want.

This is another place where SwiftUI does a heck of a lot of work on our behalf, but it does have a few interesting quirks as you'll see.

First, let's construct an example we can work with: a list that shows numbers, and every time we tap the button a new number appears. Here's the code for that:

```
struct ContentView: View {
    @State private var numbers = [Int]()
    @State private var currentNumber = 1

    var body: some View {
        VStack {
            List {
                ForEach(numbers, id: \.self) {
                    Text("Row \($0)")
                }
            }

            Button("Add Number") {
                numbers.append(currentNumber)
                currentNumber += 1
            }
        }
    }
}
```

Project 7: iExpense

Now, you might think that the **ForEach** isn't needed – the list is made up of entirely dynamic rows, so we could write this instead:

```
List(numbers, id: \.self) {  
    Text("Row \($0)")  
}
```

That would also work, but here's our first quirk: the **onDelete()** modifier only exists on **ForEach**, so if we want users to delete items from a list we must put the items inside a **ForEach**. This does mean a small amount of extra code for the times when we have only dynamic rows, but on the flip side it means it's easier to create lists where only some rows can be deleted.

In order to make **onDelete()** work, we need to implement a method that will receive a single parameter of type **IndexSet**. This is a bit like a set of integers, except it's sorted, and it's just telling us the positions of all the items in the **ForEach** that should be removed.

Because our **ForEach** was created entirely from a single array, we can actually just pass that index set straight to our **numbers** array – it has a special **remove(atOffsets:)** method that accepts an index set.

So, add this method to **ContentView** now:

```
func removeRows(at offsets: IndexSet) {  
    numbers.remove(atOffsets: offsets)  
}
```

Finally, we can tell SwiftUI to call that method when it wants to delete data from the **ForEach**, by modifying it to this:

```
ForEach(numbers, id: \.self) {  
    Text("Row \($0)")  
}  
.onDelete(perform: removeRows)
```

Now go ahead and run your app, then add a few numbers. When you're ready, swipe from right to left across any of the rows in your list, and you should find a delete button appears. You can tap that, or you can also use iOS's swipe to delete functionality by swiping further.

Given how easy that was, I think the result works really well. But SwiftUI has another trick up its sleeve: we can add an Edit/Done button to the navigation bar, that lets users delete several rows more easily.

First, wrap your **VStack** in a **NavigationView**, then add this modifier to the **VStack**:

```
.toolbar {  
    EditButton()  
}
```

That's literally all it takes – if you run the app you'll see you can add some numbers, then tap Edit to start deleting those rows. When you're ready, tap Done to exit editing mode. Not bad, given how little code it took!

Storing user settings with UserDefaults

Most users pretty much *expect* apps to store their data so they can create more customized experiences, and as such it's no surprise that iOS gives us several ways to read and write user data.

One common way to store a small amount of data is called **UserDefaults**, and it's great for simple user preferences. There is no specific number attached to "a small amount", but everything you store in **UserDefaults** will automatically be loaded when your app launches – if you store a lot in there your app launch will slow down. To give you at least an *idea*, you should aim to store no more than 512KB in there.

Tip: If you're thinking "512KB? How much is *that*?" then let me give you a rough estimate: it's about as much text as all the chapters you've read in this book so far.

UserDefaults is perfect for storing things like when the user last launched the app, which news story they last read, or other passively collected information. Even better, SwiftUI can often wrap up **UserDefaults** inside a nice and simple property wrapper called **@AppStorage** – it only supports a subset of functionality right now, but it can be really helpful.

Enough chat – let's look at some code. Here's a view with a button that shows a tap count, and increments that count every time the button is tapped:

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

    var body: some View {
        Button("Tap count: \(tapCount)") {
            tapCount += 1
        }
    }
}
```

As this is clearly A Very Important App, we want to save the number of taps that the user made, so when they come back to the app in the future they can pick up where they left off.

To make that happen, we need to write to **UserDefaults** inside our button's action closure. So, add this after the **tapCount += 1** line:

```
UserDefaults.standard.set(self.tapCount, forKey: "Tap")
```

In just that single line of code you can see three things in action:

1. We need to use **UserDefaults.standard**. This is the built-in instance of **UserDefaults** that is attached to our app, but in more advanced apps you can create your own instances. For example, if you want to share defaults across several app extensions you might create your own **UserDefaults** instance.
2. There is a single **set()** method that accepts any kind of data – integers, Booleans, strings, and more.
3. We attach a string name to this data, in our case it's the key “Tap”. This key is case-sensitive, just like regular Swift strings, and it's important – we need to use the same key to read the data back out of **UserDefaults**.

Speaking of reading the data back, rather than start with **tapCount** set to 0 we should instead make it read the value back from **UserDefaults** like this:

```
@State private var tapCount =  
UserDefaults.standard.integer(forKey: "Tap")
```

Notice how that uses exactly the same key name, which ensures it reads the same integer value.

Go ahead and give the app a try and see what you think – you ought to be able tap the button a few times, go back to Xcode, run the app again, and see the number exactly where you left it.

There are two things you can't see in that code, but both matter. First, what happens if we

Project 7: iExpense

don't have the "Tap" key set? This will be the case the very first time the app is run, but as you just saw it works fine – if the key can't be found it just sends back 0.

Sometimes having a default value like 0 is helpful, but other times it can be confusing. With Booleans, for example, you get back false if **boolean(forKey:)** can't find the key you asked for, but is that false a value you set yourself, or does it mean there was no value at all?

Second, it takes iOS a little time to write your data to permanent storage – to actually save that change to the device. They don't write updates immediately because you might make several back to back, so instead they wait some time then write out all the changes at once. How much time is another number we don't know, but a couple of seconds ought to do it.

As a result of this, if you tap the button then quickly relaunch the app from Xcode, you'll find your most recent tap count wasn't saved. There used to be a way of forcing updates to be written immediately, but at this point it's worthless – even if the user immediately started the process of terminating your app after making a choice, your defaults data would be written immediately so nothing will be lost.

Now, I mentioned that SwiftUI provides an **@AppStorage** property wrapper around **UserDefaults**, and in simple situations like this one it's really helpful. What it does is let us effectively ignore **UserDefaults** entirely, and just use **@AppStorage** rather than **@State**, like this:

```
struct ContentView: View {
    @AppStorage("tapCount") private var tapCount = 0

    var body: some View {
        Button("Tap count: \(tapCount)") {
            tapCount += 1
        }
    }
}
```

Again, there are three things I want to point out in there:

1. Our access to the **UserDefaults** system is through the **@AppStorage** property wrapper.
This works like **@State**: when the value changes, it will reinvoked the **body** property so our UI reflects the new data.
2. We attach a string name, which is the **UserDefaults** key where we want to store the data.
I've used "tapCount", but it can be anything at all – it doesn't need to match the property name.
3. The rest of the property is declared as normal, including providing a default value of 0. That will be used if there is existing value saved inside **UserDefaults**.

Clearly using **@AppStorage** is easier than **UserDefaults**: it's one line of code rather than two, and it also means we don't have to repeat the key name each time. However, right now at least **@AppStorage** doesn't make it easy to handle storing complex objects such as Swift structs – perhaps because Apple wants us to remember that storing lots of data in there is a bad idea!

Archiving Swift objects with Codable

@AppStorage is great for storing simple settings such as integers and Booleans, but when it comes to complex data – custom Swift types, for example – we need to do a little more work. This is where we need to poke around directly with **UserDefaults** itself, rather than going through the **@AppStorage** property wrapper.

Here's a simple **User** data structure we can work with:

```
struct User {  
    let firstName: String  
    let lastName: String  
}
```

That has two strings, but those aren't special – they are just pieces of text. The same goes for integer (plain old numbers), Boolean (true or false), and **Double** (plain old numbers, just with a dot somewhere in there). Even arrays and dictionaries of those values are easy to think about: there's one string, then another, then a third, and so on.

When working with data like this, Swift gives us a fantastic protocol called **Codable**: a protocol specifically for *archiving* and *unarchiving* data, which is a fancy way of saying “converting objects into plain text and back again.”

We're going to be looking at **Codable** much more in future projects, but for now we're going to keep it as simple as possible: we want to archive a custom type so we can put it into **UserDefaults**, then unarchive it when it comes back *out* from **UserDefaults**.

When working with a type that only has simple properties – strings, integers, Booleans, arrays of strings, and so on – the only thing we need to do to support archiving and unarchiving is add a conformance to **Codable**, like this:

```
struct User: Codable {
```

```

let firstName: String
let lastName: String
}

```

Swift will automatically generate some code for us that will archive and unarchive **User** instances for us as needed, but we still need to tell Swift *when* to archive and what to do with the data.

This part of the process is powered by a new type called **JSONEncoder**. Its job is to take something that conforms to **Codable** and send back that object in JavaScript Object Notation (JSON) – the name implies it's specific to JavaScript, but in practice we all use it because it's so fast and simple.

The **Codable** protocol doesn't require that we use JSON, and in fact other formats are available, but it is by far the most common. In this instance, we don't actually care *what* sort of data is used, because it's just going to be stored in **UserDefaults**.

To convert our **user** data into JSON data, we need to call the **encode()** method on a **JSONEncoder**. This might throw errors, so it should be called with **try** or **try?** to handle errors neatly. For example, if we had a property to store a **User** instance, like this:

```
@State private var user = User(firstName: "Taylor", lastName: "Swift")
```

Then we could create a button that archives the user and save it to **UserDefaults** like this:

```

Button("Save User") {
    let encoder = JSONEncoder()

    if let data = try? encoder.encode(user) {
        UserDefaults.standard.set(data, forKey: "UserData")
    }
}

```

Project 7: iExpense

That accesses **UserDefaults** directly rather than going through **@AppStorage**, because the **@AppStorage** property wrapper just doesn't work here.

That **data** constant is a new data type called, perhaps confusingly, **Data**. It's designed to store any kind of data you can think of, such as strings, images, zip files, and more. Here, though, all we care about is that it's one of the types of data we can write straight into **UserDefaults**.

When we're coming back the other way – when we have JSON data and we want to convert it to Swift **Codable** types – we should use **JSONDecoder** rather than **JSONEncoder()**, but the process is much the same.

That brings us to the end of our project overview, so go ahead and reset your project to its initial state ready to build on.

Building a list we can delete from

In this project we want a list that can show some expenses, and previously we would have done this using an `@State` array of objects. Here, though, we’re going to take a different approach: we’re going to create an **Expenses** class that will be attached to our list using `@StateObject`.

This might sound like we’re over-complicating things a little, but it actually makes things much easier because we can make the **Expenses** class load and save itself seamlessly – it will be almost invisible, as you’ll see.

First, we need to decide what an expense *is* – what do we want it to store? In this instance it will be three things: the name of the item, whether it’s business or personal, and its cost as a **Double**.

We’ll add more to this later, but for now we can represent all that using a single **ExpenseItem** struct. You can put this into a new Swift file called `ExpenseItem.swift`, but you don’t need to – you can just put this into `ContentView.swift` if you like, as long as you don’t put it *inside* the **ContentView** struct itself.

Regardless of where you put it, this is the code to use:

```
struct ExpenseItem {  
    let name: String  
    let type: String  
    let amount: Double  
}
```

Now that we have something that represents a single expense, the next step is to create something to store an array of those expense items inside a single object. This needs to conform to the **ObservableObject** protocol, and we’re also going to use `@Published` to make sure change announcements get sent whenever the **items** array gets modified.

As with the **ExpenseItem** struct, this will start off simple and we’ll add to it later, so add this

Project 7: iExpense

new class now:

```
class Expenses: ObservableObject {  
    @Published var items = [ExpenseItem]()  
}
```

That finishes all the data required for our main view: we have a struct to represent a single item of expense, and a class to store an array of all those items.

Let's now put that into *action* with our SwiftUI view, so we can actually see our data on the screen. Most of our view will just be a **List** showing the items in our expenses, but because we want users to delete items they no longer want we can't just use a simple **List** – we need to use a **ForEach** *inside* the list, so we get access to the **onDelete()** modifier.

First, we need to add an **@StateObject** property in our view, that will create an instance of our **Expenses** class:

```
@StateObject var expenses = Expenses()
```

Remember, using **@StateObject** here asks SwiftUI to watch the object for any change announcements, so any time one of our **@Published** properties changes the view will refresh its body. It's only used when *creating* a class instance – all other times you'll use **@ObservedObject** instead.

Second, we can use that **Expenses** object with a **NavigationView**, a **List**, and a **ForEach**, to create our basic layout:

```
NavigationView {  
    List {  
        ForEach(expenses.items, id: \.name) { item in  
            Text(item.name)  
        }  
    }.navigationTitle("iExpense")
```

```
}
```

That tells the **ForEach** to identify each expense item uniquely by its name, then prints the name out as the list row.

We're going to add two more things to our simple layout before we're done: the ability to add new items for testing purposes, and the ability to delete items with a swipe.

We're going to let users add their own items soon, but it's important to check that our list actually works well before we continue. So, we're going to add a toolbar button that adds example **ExpenseItem** instances for us to work with – add this modifier to the **List** now:

```
.toolbar {
    Button {
        let expense = ExpenseItem(name: "Test", type: "Personal",
amount: 5)
        expenses.items.append(expense)
    } label: {
        Image(systemName: "plus")
    }
}
```

That brings our app to life: you can launch it now, then press the + button repeatedly to add lots of testing expenses.

Now that we can *add* expenses, we can also add code to remove them. This means adding a method capable of deleting an **IndexSet** of list items, then passing that directly on to our **expenses** array:

```
func removeItems(at offsets: IndexSet) {
    expenses.items.remove(atOffsets: offsets)
}
```

Project 7: iExpense

And to attach that to SwiftUI, we add an **onDelete()** modifier to our **ForEach**, like this:

```
ForEach(expenses.items, id: \.name) { item in
    Text(item.name)
}
.onDelete(perform: removeItems)
```

Go ahead and run the app now, press + a few times, then swipe to delete the rows.

Now, remember: when we say **id: \.name** we're saying we can identify each item uniquely by its name, which isn't true here – we have the same name multiple times, and we can't guarantee our expenses will be unique either.

Often this will Just Work, but sometimes it will cause bizarre, broken animations in your project, so let's look at a better solution next.

Working with Identifiable items in SwiftUI

When we create static views in SwiftUI – when we hard-code a **VStack**, then a **TextField**, then a **Button**, and so on – SwiftUI can see exactly which views we have, and is able to control them, animate them, and more. But when we use **List** or **ForEach** to make dynamic views, SwiftUI needs to know how it can identify each item uniquely otherwise it will struggle to compare view hierarchies to figure out what has changed.

In our current code, we have this:

```
ForEach(expenses.items, id: \.name) { item in
    Text(item.name)
}
.onDelete(perform: removeItems)
```

In English, that means “create a new row for every item in the expense items, identified uniquely by its name, showing that name in the row, and calling the **removeItems()** method to delete it.”

Then, later, we have this code:

```
Button {
    let expense = ExpenseItem(name: "Test", type: "Personal",
amount: 5)
    expenses.items.append(expense)
} label: {
    Image(systemName: "plus")
}
```

Every time that button is pressed, it adds a test expense to our list, so we can make sure adding and deleting works.

Project 7: iExpense

Can you see the problem?

Every time we create an example expense item we're using the name "Test", but we've also told SwiftUI that it can use the expense name as a unique identifier. So, when our code runs and we delete an item, SwiftUI looks at the array beforehand – "Test", "Test", "Test", "Test" – then looks at the array *afterwards* – "Test", "Test", "Test" – and can't easily tell what changed. *Something* has changed, because one item has disappeared, but SwiftUI can't be sure which.

In this situation we're lucky, because **List** knows exactly which row we were swiping on, but in many other places that extra information won't be available and our app will start to behave strangely.

This is a logic error on our behalf: our code is fine, and it doesn't crash at runtime, but we've applied the wrong logic to get to that end result – we've told SwiftUI that something will be a unique identifier, when it isn't unique at all.

To fix this we need to think more about our **ExpenseItem** struct. Right now it has three properties: **name**, **type**, and **amount**. The name by itself *might* be unique in practice, but it's also likely not to be – as soon as the user enters "Lunch" twice we'll start hitting problems. We could perhaps try to combine the name, type and amount into a new computed property, but even then we're just delaying the inevitable; it's still not really unique.

The smart solution here is to add something to **ExpenseItem** that *is* unique, such as an ID number that we assign by hand. That would work, but it does mean tracking the last number we assigned so we don't use duplicates there either.

There is in fact an easier solution, and it's called **UUID** – short for "universally unique identifier", and if *that* doesn't sound unique I'm not sure what does.

UUIDs are long hexadecimal strings such as this one: 08B15DB4-2F02-4AB8-A965-67A9C90D8A44. So, that's eight digits, four digits, four digits, four digits, then twelve digits, of which the only requirement is that there's a 4 in the first number of the third block. If we subtract the fixed 4, we end up with 31 digits, each of which can be one of 16 values – if

we generated 1 UUID every second for a billion years, we might begin to have the slightest chance of generating a duplicate.

Now, we could update **ExpenseItem** to have a **UUID** property like this:

```
struct ExpenseItem {  
    let id: UUID  
    let name: String  
    let type: String  
    let amount: Int  
}
```

And that would work. However, it would also mean we need to generate a UUID by hand, then load and save the UUID along with our other data. So, in this instance we're going to ask Swift to generate a **UUID** automatically for us like this:

```
struct ExpenseItem {  
    let id = UUID()  
    let name: String  
    let type: String  
    let amount: Int  
}
```

Now we don't need to worry about the **id** value of our expense items – Swift will make sure they are always unique.

With that in place we can now fix our **ForEach**, like this:

```
ForEach(expenses.items, id: \.id) { item in  
    Text(item.name)  
}
```

If you run the app now you'll see our problem is fixed: SwiftUI can now see exactly which expense item got deleted, and will animate everything correctly.

Project 7: iExpense

We're not done with this step quite yet, though. Instead, I'd like you to modify the **ExpenseItem** to make it conform to a new protocol called **Identifiable**, like this:

```
struct ExpenseItem: Identifiable {
    let id = UUID()
    let name: String
    let type: String
    let amount: Int
}
```

All we've done is add **Identifiable** to the list of protocol conformances, nothing more. This is one of the protocols built into Swift, and means "this type can be identified uniquely." It has only one requirement, which is that there must be a property called **id** that contains a unique identifier. We just added that, so we don't need to do any extra work – our type conforms to **Identifiable** just fine.

Now, you might wonder why we added that, because our code was working fine before. Well, because our expense items are now guaranteed to be identifiable uniquely, we no longer need to tell **ForEach** which property to use for the identifier – it knows there will be an **id** property and that it will be unique, because that's the point of the **Identifiable** protocol.

So, as a result of this change we can modify the **ForEach** again, to this:

```
ForEach(expenses.items) { item in
    Text(item.name)
}
```

Much better!

Sharing an observed object with a new view

Classes that conform to **ObservableObject** can be used in more than one SwiftUI view, and all of those views will be updated when the published properties of the class change.

In this app, we're going to design a view specially for adding new expense items. When the user is ready, we'll add that to our **Expenses** class, which will automatically cause the original view to refresh its data so the expense item can be shown.

To make a new SwiftUI view you can either press Cmd+N or go to the File menu and choose New > File. Either way, you should select “SwiftUI View” under the User Interface category, then name the file AddView.swift. Xcode will ask you where to save the file, so make sure you see a folder icon next to “iExpense”, then click Create to have Xcode show you the new view, ready to edit.

As with our other views, our first pass at **AddView** will be simple and we'll add to it. That means we're going to add text fields for the expense name and amount, plus a picker for the type, all wrapped up in a form and a navigation view.

This should all be old news to you by now, so let's get into the code:

```
struct AddView: View {
    @State private var name = ""
    @State private var type = "Personal"
    @State private var amount = 0.0

    let types = [ "Business", "Personal" ]

    var body: some View {
        NavigationView {
            Form {
                TextField("Name", text: $name)
```

Project 7: iExpense

```
Picker("Type", selection: $type) {
    ForEach(types, id: \.self) {
        Text($0)
    }
}

TextField("Amount", value: $amount,
format: .currency(code: "USD"))
    .keyboardType(.decimalPad)
}
.navigationTitle("Add new expense")
}
}
```

Yes, that always uses US dollars for the currency type – you'll need to make that smarter in the challenges for this project.

We'll come back to the rest of that code in a moment, but first let's add some code to **ContentView** so we can show **AddView** when the + button is tapped.

In order to present **AddView** as a new view, we need to make three changes to **ContentView**. First, we need some state to track whether or not **AddView** is being shown, so add this as a property now:

```
@State private var showingAddExpense = false
```

Next, we need to tell SwiftUI to use that Boolean as a condition for showing a sheet – a pop-up window. This is done by attaching the **sheet()** modifier somewhere to our view hierarchy. You can use the **List** if you want, but the **NavigationView** works just as well. Either way, add this code as a modifier to one of the views in **ContentView**:

Sharing an observed object with a new view

```
.sheet(isPresented: $showingAddExpense) {  
    // show an AddView here  
}
```

The third step is to put something inside the sheet. Often that will just be an instance of the view type you want to show, like this:

```
.sheet(isPresented: $showingAddExpense) {  
    AddView()  
}
```

Here, though, we need something more. You see, we already have the **expenses** property in our content view, and inside **AddView** we're going to be writing code to add expense items. We don't want to create a *second* instance of the **Expenses** class in **AddView**, but instead want it to share the existing instance from **ContentView**.

So, what we're going to do is add a property to **AddView** to store an **Expenses** object. It won't create the object there, which means we need to use **@ObservedObject** rather than **@StateObject**.

Please add this property to **AddView**:

```
@ObservedObject var expenses: Expenses
```

And now we can pass our existing **Expenses** object from one view to another – they will both share the same object, and will both monitor it for changes. Modify your **sheet()** modifier in **ContentView** to this:

```
.sheet(isPresented: $showingAddExpense) {  
    AddView(expenses: expenses)  
}
```

We're not quite done with this step yet, for two reasons: our code won't compile, and even if it

Project 7: iExpense

did compile it wouldn't work because our button doesn't trigger the sheet.

The compilation failure happens because when we made the new SwiftUI view, Xcode also added a preview provider so we can look at the design of the view while we were coding. If you find that down at the bottom of AddView.swift, you'll see that it tries to create an **AddView** instance without providing a value for the **expenses** property.

That isn't allowed any more, but we can just pass in a dummy value instead, like this:

```
struct AddView_Previews: PreviewProvider {
    static var previews: some View {
        AddView(expenses: Expenses())
    }
}
```

The second problem is that we don't actually have any code to *show* the sheet, because right now the + button in **ContentView** adds test expenses. Fortunately, the fix is trivial – just replace the existing action with code to toggle our **showingAddExpense** Boolean, like this:

```
Button {
    showingAddExpense = true
} label: {
    Image(systemName: "plus")
}
```

If you run the app now the whole sheet should be working as intended – you start with **ContentView**, tap the + button to bring up an **AddView** where you can type in the various fields, then can swipe to dismiss.

Making changes permanent with UserDefaults

At this point, our app's user interface is functional: you've seen we can add and delete items, and now we have a sheet showing a user interface to create new expenses. However, the app is far from working: any data placed into **AddView** is completely ignored, and even if it *weren't* ignored then it still wouldn't be saved for future times the app is run.

We'll tackle those problems in order, starting with actually *doing* something with the data from **AddView**. We already have properties that store the values from our form, and previously we added a property to store an **Expenses** object passed in from the **ContentView**.

We need to put those two things together: we need a button that, when tapped, creates an **ExpenseItem** out of our properties and adds it to the **expenses** items.

Add this modifier below **navigationTitle()** in **AddView**:

```
.toolbar {
    Button("Save") {
        let item = ExpenseItem(name: name, type: type, amount: amount)
        expenses.items.append(item)
    }
}
```

Although we have more work to do, I encourage you to run the app now because it's actually coming together – you can now show the add view, enter some details, press “Save”, then swipe to dismiss, and you'll see your new item in the list. That means our data synchronization is working perfectly: both the SwiftUI views are reading from the same list of expense items.

Now try launching the app again, and you'll immediately hit our second problem: any data you add isn't stored, meaning that everything starts blank every time you relaunch the app.

Project 7: iExpense

This is obviously a pretty terrible user experience, but thanks to the way we have **Expense** as a separate class it's actually not that hard to fix.

We're going to leverage four important technologies to help us save and load data in a clean way:

- The **Codable** protocol, which will allow us to archive all the existing expense items ready to be stored.
- **User Defaults**, which will let us save and load that archived data.
- A custom initializer for the **Expenses** class, so that when we make an instance of it we load any saved data from **User Defaults**
- A **didSet** property observer on the **items** property of **Expenses**, so that whenever an item gets added or removed we'll write out changes.

Let's tackle *writing* the data first. We already have this property in the **Expenses** class:

```
@Published var items = [ExpenseItem]()
```

That's where we store all the expense item structs that have been created, and that's also where we're going to attach our property observer to write out changes as they happen.

This takes four steps in total: we need to create an instance of **JSONEncoder** that will do the work of converting our data to JSON, we ask that to try encoding our **items** array, and then we can write that to **User Defaults** using the key "Items".

Modify the **items** property to this:

```
@Published var items = [ExpenseItem]() {
    didSet {
        if let encoded = try? JSONEncoder().encode(items) {
            UserDefaults.standard.set(encoded, forKey: "Items")
        }
    }
}
```

Tip: Using `JSONEncoder().encode()` means “create an encoder and use it to encode something,” all in one step, rather than creating the encoder first then using it later.

Now, if you’re following along you’ll notice that code doesn’t actually compile. And if you’re following along *closely* you’ll have noticed that I said this process takes *four* steps while only listing three.

The problem is that the `encode()` method can only archive objects that conform to the **Codable** protocol. Remember, conforming to **Codable** is what asks the compiler to generate code for us able to handle archiving and unarchiving objects, and if we don’t add a conformance for that then our code won’t build.

Helpfully, we don’t need to do any work other than add **Codable** to **ExpenseItem**, like this:

```
struct ExpenseItem: Identifiable, Codable {
    let id = UUID()
    let name: String
    let type: String
    let amount: Int
}
```

Swift already includes **Codable** conformances for the **UUID**, **String**, and **Int** properties of **ExpenseItem**, and so it’s able to make **ExpenseItem** conform automatically as soon as we ask for it.

However, you *will* see a warning that **id** will not be decoded because we made it a constant and assigned a default value. This is actually the behavior we want, but Swift is trying to be helpful because it’s possible you *did* plan to decode this value from JSON. To make the warning go away, make the property variable like this:

```
var id = UUID()
```

With that change, we’ve written all the code needed to make sure our items are saved when the

Project 7: iExpense

user adds them. However, it's not effective by itself: the data might be saved, but it isn't loaded again when the app relaunches.

To solve that – and also to make our code compile again – we need to implement a custom initializer. That will:

1. Attempt to read the “Items” key from **UserDefaults**.
2. Create an instance of **JSONDecoder**, which is the counterpart of **JSONEncoder** that lets us go from JSON data to Swift objects.
3. Ask the decoder to convert the data we received from **UserDefaults** into an array of **ExpenseItem** objects.
4. If that worked, assign the resulting array to **items** and exit.
5. Otherwise, set **items** to be an empty array.

Add this initializer to the **Expenses** class now:

```
init() {
    if let savedItems = UserDefaults.standard.data(forKey: "Items") {
        if let decodedItems = try? JSONDecoder().decode([ExpenseItem].self, from: savedItems) {
            items = decodedItems
            return
        }
    }

    items = []
}
```

The two key parts of that code are the **data(forKey: "Items")** line, which attempts to read whatever is in “Items” as a **Data** object, and **try?**

JSONDecoder().decode([ExpenseItem].self, from: savedItems), which does the actual job of unarchiving the **Data** object into an array of **ExpenseItem** objects.

Making changes permanent with UserDefaults

It's common to do a bit of a double take when you first see **[ExpenseItem].self** – what does the **.self** mean? Well, if we had just used **[ExpenseItem]**, Swift would want to know what we meant – are we trying to make a copy of the class? Were we planning to reference a static property or method? Did we perhaps mean to create an instance of the class? To avoid confusion – to say that we mean we're referring to the type itself, known as the *type object* – we write **.self** after it.

Now that we have both loading and saving in place, you should be able to use the app. It's not finished quite yet, though – let's add some final polish!

Final polish

If you try using the app, you'll soon see it has two problems:

1. Adding an expense doesn't dismiss **AddView**; it just stays there.
2. When you add an expense, you can't actually see any details about it.

Before we wrap up this project, let's fix those to make the whole thing feel a little more polished.

First, dismissing **AddView** is done by calling **dismiss()** on the environment when the time is right. This is controlled by the view's environment, and links to the **isPresented** parameter for our sheet – that Boolean gets set to true by us to show **AddView**, but will be flipped back to false by the environment when we call **dismiss()**.

Start by adding this property to **AddView**:

```
@Environment(\.dismiss) var dismiss
```

You'll notice we don't specify a type for that – Swift can figure it out thanks to the **@Environment** property wrapper.

Next, we need to call **dismiss()** when we want the view to dismiss itself. This causes the **showingAddExpense** Boolean in **ContentView** to go back to false, and hides the **AddView**. We already have a Save button in **AddView** that creates a new expense item and appends it to our existing expenses, so add this on the line directly after:

```
dismiss()
```

That solves the first problem, which just leaves the second: we show the name of each expense item but nothing more. This is because the **ForEach** for our list is trivial:

```
ForEach(expenses.items) { item in
    Text(item.name)
```

```
}
```

We're going to replace that with a stack within another stack, to make sure all the information looks good on screen. The inner stack will be a **VStack** showing the expense name and type, then around that will be a **HStack** with the **VStack** on the left, then a spacer, then the expense amount. This kind of layout is common on iOS: title and subtitle on the left, and more information on the right.

Replace the existing **ForEach** in **ContentView** with this:

```
ForEach(expenses.items) { item in
    HStack {
        VStack(alignment: .leading) {
            Text(item.name)
                .font(.headline)
            Text(item.type)
        }
        Spacer()
        Text(item.amount, format: .currency(code: "USD"))
    }
}
```

Now run the program one last time and try it out – we're done!

iExpense: Wrap up

Although the app we were building wasn't itself difficult, *getting there* took quite a bit of learning – we had to cover **UserDefault**s, **Codable**, **sheet()**, **onDelete()**, **@StateObject**, and more. Some of those things – particularly **UserDefault**s and **Codable** – have only been introduced at a high level in this project, which is intentional; we'll be getting into them more later on.

Now that your SwiftUI skills are coming along, I hope you noticed we were able to skip past some parts with very little explanation. For example, you know very well how to create forms with text fields and pickers because we've already covered that extensively, which means we can spend our time focusing on new stuff.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Use the user's preferred currency, rather than always using US dollars.
2. Modify the expense amounts in **ContentView** to contain some styling depending on their value – expenses under \$10 should have one style, expenses under \$100 another, and expenses over \$100 a third style. What those styles are depend on you.
3. For a bigger challenge, try splitting the expenses list into two sections: one for personal expenses, and one for business expenses. This is tricky for a few reasons, not least because it means being careful about how items are deleted!

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to iExpense. If you don't already subscribe, you can start a free trial today.

Project 8

Moonshot

Teach users about space history with scroll views, Codable, and more

Moonshot: Introduction

In this project we’re going to build an app that lets users learn about the missions and astronauts that formed NASA’s Apollo space program. You’ll get more experience with **Codable**, but more importantly you’ll also work with scroll views, navigation, and much more interesting layouts.

Yes, you’ll get some practice time with **List**, **Text**, and more, but you’ll also start to solve important SwiftUI problems – how can you make an image fit its space correctly? How can we clean up code using computed properties? How can we compose smaller views into larger ones to help keep our project organized?

As always there’s lots to do, so let’s get started: create a new iOS app using the App template, naming it “Moonshot”. We’ll be using that for the project, but first lets take a closer look at the new techniques you’ll need to become familiar with…

Resizing images to fit the screen using GeometryReader

When we create an **Image** view in SwiftUI, it will automatically size itself according to the dimensions of its contents. So, if the picture is 1000x500, the **Image** view will also be 1000x500. This is sometimes what you want, but mostly you'll want to show the image at a lower size, and I want to show you how that can be done, but also how we can make an image fit some amount of the user's screen width using a new view type called **GeometryReader**.

First, add some sort of image to your project. It doesn't matter what it is, as long as it's wider than the screen. I called mine "Example", but obviously you should substitute your image name in the code below.

Now let's draw that image on the screen:

```
struct ContentView: View {  
    var body: some View {  
        Image("Example")  
    }  
}
```

Even in the preview you can see that's way too big for the available space. Images have the same **frame()** modifier as other views, so you might try to scale it down like this:

```
Image("Example")  
.frame(width: 300, height: 300)
```

However, that won't work – your image will still appear to be its full size. If you want to know *why*, take a close look at the preview window: you'll see your image is full size, but there's now a box that's 300x300, sat in the middle. The *image view's* frame has been set correctly, but the *content* of the image is still shown as its original size.

Try changing the image to this:

Resizing images to fit the screen using GeometryReader

```
Image("Example")
    .frame(width: 300, height: 300)
    .clipped()
```

Now you'll see things more clearly: our image view is indeed 300x300, but that's not really what we wanted.

If you want the image *contents* to be resized too, we need to use the **resizable()** modifier like this:

```
Image("Example")
    .resizable()
    .frame(width: 300, height: 300)
```

That's better, but only just. Yes, the image is now being resized correctly, but it's probably looking squashed. My image was not square, so it looks distorted now that it's been resized into a square shape.

To fix this we need to ask the image to resize itself proportionally, which can be done using the **scaledToFit()** and **scaledToFill()** modifiers. The first of these means the entire image will fit inside the container even if that means leaving some parts of the view empty, and the second means the view will have no empty parts even if that means some of our image lies outside the container.

Try them both to see the difference for yourself. Here is **.fit** mode applied:

```
Image("Example")
    .resizable()
    .scaledToFit()
    .frame(width: 300, height: 300)
```

And here is **scaledToFill()**:

Project 8: Moonshot

```
Image("Example")
    .resizable()
    .scaledToFill()
    .frame(width: 300, height: 300)
```

All this works great if we want fixed-sized images, but very often you want images that automatically scale up to fill more of the screen in one or both dimensions. That is, rather than hard-coding a width of 300, what you *really* want to say is “make this image fill 80% of the width of the screen.”

SwiftUI gives us a dedicated type for this called **GeometryReader**, and it’s remarkably powerful. Yes, I know lots of SwiftUI is powerful, but honestly: what you can do with **GeometryReader** will blow you away.

We’ll go into much more detail on **GeometryReader** in project 15, but for now we’re going to use it for one job: to make sure our image fills the full width of its container view.

GeometryReader is a view just like the others we’ve used, except when we create it we’ll be handed a **GeometryProxy** object to use. This lets us query the environment: how big is the container? What position is our view? Are there any safe area insets? And so on.

In principle that seems simple enough, but in practice you need to use **GeometryReader** carefully because it automatically expands to take up available space in your layout, then positions its own content aligned to the top-left corner.

For example, we could make an image that’s 80% the width of the screen, with a fixed height of 300:

```
GeometryReader { geo in
    Image("Example")
        .resizable()
        .scaledToFit()
        .frame(width: geo.size.width * 0.8, height: 300)
}
```

Resizing images to fit the screen using GeometryReader

You can even remove the **height** from the image, like this:

```
GeometryReader { geo in
    Image("Example")
        .resizable()
        .scaledToFit()
        .frame(width: geo.size.width * 0.8)
}
```

We've given SwiftUI enough information that it can automatically figure out the height: it knows the original width, it knows our target width, and it knows our content mode, so it understands how the target height of the image will be proportional to the target width.

Tip: If you ever want to center a view inside a **GeometryReader**, rather than aligning to the top-left corner, add a second frame that makes it fill the full space of the container, like this:

```
GeometryReader { geo in
    Image("Example")
        .resizable()
        .scaledToFit()
        .frame(width: geo.size.width * 0.8)
        .frame(width: geo.size.width, height: geo.size.height)
}
```

How ScrollView lets us work with scrolling data

You've seen how **List** and **Form** let us create scrolling tables of data, but for times when we want to scroll *arbitrary* data – i.e., just some views we've created by hand – we need to turn to SwiftUI's **ScrollView**.

Scroll views can scroll horizontally, vertically, or in both directions, and you can also control whether the system should show scroll indicators next to them – those are the little scroll bars that appear to give users a sense of how big the content is. When we place views inside scroll views, they automatically figure out the size of that content so users can scroll from one edge to the other.

As an example, we could create a scrolling list of 100 text views like this:

```
ScrollView {  
    VStack(spacing: 10) {  
        ForEach(0..<100) {  
            Text("Item \($0)")  
                .font(.title)  
        }  
    }  
}
```

If you run that back in the simulator you'll see that you can drag the scroll view around freely, and if you scroll to the bottom you'll also see that **ScrollView** treats the safe area just like **List** and **Form** – their content goes *under* the home indicator, but they add some extra padding so the final views are fully visible.

You might also notice that it's a bit annoying having to tap directly in the center – it's more common to have the whole area scrollable. To get *that* behavior, we should make the **VStack** take up more space while leaving the default centre alignment intact, like this:

How ScrollView lets us work with scrolling data

```
ScrollView {  
    VStack(spacing: 10) {  
        ForEach(0..<100) {  
            Text("Item \($0)")  
                .font(.title)  
        }  
    }  
    .frame(maxWidth: .infinity)  
}
```

Now you can tap and drag anywhere on the screen, which is much more user-friendly.

This all seems really straightforward, however there's an important catch that you need to be aware of: when we add views to a scroll view they get created immediately. To demonstrate this, we can create a simple wrapper around a regular text view, like this:

```
struct CustomText: View {  
    let text: String  
  
    var body: some View {  
        Text(text)  
    }  
  
    init(_ text: String) {  
        print("Creating a new CustomText")  
        self.text = text  
    }  
}
```

Now we can use that inside our **ForEach**:

```
ForEach(0..<100) {  
    CustomText("Item \($0)")
```

Project 8: Moonshot

```
.font(.title)  
}
```

The result will look identical, but now when you run the app you'll see "Creating a new **CustomText**" printed a hundred times in Xcode's log – SwiftUI won't wait until you scroll down to see them, it will just create them immediately.

If you want to avoid this happening, there's an alternative for both **VStack** and **HStack** called **LazyVStack** and **LazyHStack** respectively. These can be used in exactly the same way as regular stacks but will load their content on-demand – they won't create views until they are actually shown, and so minimize the amount of system resources being used.

So, in this situation we could swap our **VStack** for a **LazyVStack** like this:

```
LazyVStack(spacing: 10) {  
    ForEach(0 ..< 100) {  
        CustomText("Item \($0)")  
            .font(.title)  
    }  
}.frame(maxWidth: .infinity)
```

Literally all it takes is to add "Lazy" before "VStack" to have our code run more efficiently – it will now only create the **CustomText** structs when they are actually needed.

Although the *code* to use regular and lazy stacks is the same, there is one important layout difference: lazy stacks always take up as much as room as is available in our layouts, whereas regular stacks take up only as much space as is needed. This is intentional, because it stops lazy stacks having to adjust their size if a new view is loaded that wants more space.

One last thing: you can make horizontal scrollviews by passing **.horizontal** as a parameter when you make your **ScrollView**. Once that's done, make sure you create a *horizontal* stack or lazy stack, so your content is laid out as you expect:

How ScrollView lets us work with scrolling data

```
ScrollView(.horizontal) {
    LazyHStack(spacing: 10) {
        ForEach(0..<100) {
            CustomText("Item \($0)")
                .font(.title)
        }
    }
}
```

Pushing new views onto the stack using NavigationLink

SwiftUI's **NavigationView** shows a navigation bar at the top of our views, but also does something else: it lets us push views onto a view stack. In fact, this is really the most fundamental form of iOS navigation – you can see it in Settings when you tap Wi-Fi or General, or in Messages whenever you tap someone's name.

This view stack system is very different from the sheets we've used previously. Yes, both show some sort of new view, but there's a difference in the *way* they are presented that affects the way users think about them.

Let's start by looking at some code so you can see for yourself. If we wrap the default text view with a navigation view and give it a title, we get this:

```
struct ContentView: View {  
    var body: some View {  
        NavigationView {  
            Text("Hello, world!")  
                .padding()  
                .navigationTitle("SwiftUI")  
        }  
    }  
}
```

That text view is just static text; it's not a button with any sort of action attached to it. We're going to make it so that when the user taps on "Hello, world!" we present them with a new view, and that's done using **NavigationLink**: give this a destination and something that can be tapped, and it will take care of the rest.

One of the many things I love about SwiftUI is that we can use **NavigationLink** with any kind of destination view. Yes, we can design a custom view to push to, but we can also push

Pushing new views onto the stack using NavigationLink

straight to some text.

To try this out, change your view to this:

```
NavigationView {
    NavigationLink {
        Text("Detail View")
    } label: {
        Text("Hello, world!")
            .padding()
    }
    .navigationTitle("SwiftUI")
}
```

Now run the code and see what you think. You will see that “Hello, world!” now looks like a button, and tapping it makes a new view slide in from the right saying “Detail View”. Even better, you’ll see that the “SwiftUI” title animates down to become a back button, and you can tap that or swipe from the left edge to go back.

So, both **sheet()** and **NavigationLink** allow us to show a new view from the current one, but the *way* they do it is different and you should choose them carefully:

- **NavigationLink** is for showing details about the user’s selection, like you’re digging deeper into a topic.
- **sheet()** is for showing unrelated content, such as settings or a compose window.

The most common place you see **NavigationLink** is with a list, and there Swift UI does something quite marvelous.

Try modifying your code to this:

```
NavigationView {
    List(0..<100) { row in
        NavigationLink {
```

Project 8: Moonshot

```
    Text("Detail \u202a(\u202arow\u202a)\u202a")
} label: {
    Text("Row \u202a(\u202arow\u202a)\u202a")
}
}
.navigationTitle("SwiftUI")
}
```

When you run the app now you'll see 100 list rows that can be tapped to show a detail view, but you'll also see gray disclosure indicators on the right edge. This is the standard iOS way of telling users another screen is going to slide in from the right when the row is tapped, and SwiftUI is smart enough to add it automatically here. If those rows weren't navigation links – if you comment out the **NavigationLink** line and its closing brace – you'll see the indicators disappear.

Working with hierarchical Codable data

The **Codable** protocol makes it trivial to decode flat data: if you’re decoding a single instance of a type, or an array or dictionary of those instances, then things Just Work. However, in this project we’re going to be decoding slightly more complex JSON: there will be an array inside another array, using different data types.

If you want to decode this kind of hierarchical data, the key is to create separate types for each level you have. As long as the data matches the hierarchy you’ve asked for, **Codable** is capable of decoding everything with no further work from us.

To demonstrate this, put this button in to your content view:

```
Button("Decode JSON") {
    let input = """
    {
        "name": "Taylor Swift",
        "address": {
            "street": "555, Taylor Swift Avenue",
            "city": "Nashville"
        }
    }
    """
}

// more code to come
}
```

That creates a string of JSON in code. In case you aren’t too familiar with JSON, it’s probably best to look at the Swift structs that match it – you can put these directly into the button action or outside of the **ContentView** struct, it doesn’t matter:

Project 8: Moonshot

```
struct User: Codable {
    let name: String
    let address: Address
}

struct Address: Codable {
    let street: String
    let city: String
}
```

Hopefully you can now see what the JSON contains: a user has a name string and an address, and addresses are a street string and a city string.

Now for the best part: we can convert our JSON string to the **Data** type (which is what **Codable** works with), then decode that into a **User** instance:

```
let data = Data(input.utf8)
let decoder = JSONDecoder()
if let user = try? decoder.decode(User.self, from: data) {
    print(user.address.street)
}
```

If you run that program and tap the button you should see the address printed out – although just for the avoidance of doubt I should say that it's not her actual address!

There's no limit to the number of levels **Codable** will go through – all that matters is that the structs you define match your JSON string.

How to lay out views in a scrolling grid

SwiftUI's **List** view is a great way to show scrolling rows of data, but sometimes you also want *columns* of data – a grid of information, that is able to adapt to show more data on larger screens.

In SwiftUI this is accomplished with two views: **LazyHGrid** for showing horizontal data, and **LazyVGrid** for showing vertical data. Just like with lazy stacks, the “lazy” part of the name is there because SwiftUI will automatically delay loading the views it contains until the moment they are needed, meaning that we can display more data without chewing through a lot of system resources.

Creating a grid is done in two steps. First, we need to define the rows or columns we want – we only define one of the two, depending on which kind of grid we want.

For example, if we have a vertically scrolling grid then we might say we want our data laid out in three columns exactly 80 points wide by adding this property to our view:

```
let layout = [
    GridItem(.fixed(80)),
    GridItem(.fixed(80)),
    GridItem(.fixed(80))
]
```

Once you have your layout defined, you should place your grid inside a **ScrollView**, along with as many items as you want. Each item you create inside the grid is automatically assigned a column in the same way that rows inside a list automatically get placed inside their parent.

For example, we could render 1000 items inside our three-column grid like this:

```
ScrollView {
    LazyVGrid(columns: layout) {
```

Project 8: Moonshot

```
ForEach(0..<1000) {
    Text("Item \($0)")
}
```

That works for some situations, but the best part of grids is their ability to work across a variety of screen sizes. This can be done with a different column layout using *adaptive* sizes, like this:

```
let layout = [
    GridItem(.adaptive(minimum: 80)),
]
```

That tells SwiftUI we're happy to fit in as many columns as possible, as long as they are at least 80 points in width. You can also specify a maximum range for even more control:

```
let layout = [
    GridItem(.adaptive(minimum: 80, maximum: 120)),
]
```

I tend to rely on these adaptive layouts the most, because they allow grids that make maximum use of available screen space.

Before we're done, I want to briefly show you how to make *horizontal* grids. The process is almost identical, you just need to make your **ScrollView** work horizontally, then create a **LazyHGrid** using rows rather than columns:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: layout) {
        ForEach(0..<1000) {
            Text("Item \($0)")
        }
    }
}
```

How to lay out views in a scrolling grid

```
    }  
}
```

That brings us to the end of the overview for this project, so please go ahead and reset ContentView.swift to its original state.

Loading a specific kind of Codable data

In this app we’re going to load two different kinds of JSON into Swift structs: one for astronauts, and one for missions. Making this happen in a way that is easy to maintain and doesn’t clutter our code takes some thinking, but we’ll work towards it step by step.

First, drag in the two JSON files for this project. These are available in the GitHub repository for this book, under “project8-files” – look for astronauts.json and missions.json, then drag them into your project navigator. While we’re adding assets, you should also copy all the images into your asset catalog – these are in the “Images” subfolder. The images of astronauts and mission badges were all created by NASA, so under Title 17, Chapter 1, Section 105 of the US Code they are available for us to use under a public domain license.

If you look in astronauts.json, you’ll see each astronaut is defined by three fields: an ID (“grissom”, “white”, “chaffee”, etc), their name (“Virgil I. "Gus" Grissom”, etc), and a short description that has been copied from Wikipedia. If you intend to use the text in your own shipping projects, it’s important that you give credit to Wikipedia and its authors and make it clear that the work is licensed under CC-BY-SA available from here: <https://creativecommons.org/licenses/by-sa/3.0>.

Let’s convert that astronaut data into a Swift struct now – press Cmd+N to make a new file, choose Swift file, then name it Astronaut.swift. Give it this code:

```
struct Astronaut: Codable, Identifiable {
    let id: String
    let name: String
    let description: String
}
```

As you can see, I’ve made that conform to **Codable** so we can create instances of this struct straight from JSON, but also **Identifiable** so we can use arrays of astronauts inside **ForEach**

and more – that **id** field will do just fine.

Next we want to convert astronauts.json into a dictionary of **Astronaut** instances, which means we need to use **Bundle** to find the path to the file, load that into an instance of **Data**, and pass it through a **JSONDecoder**. Previously we put this into a method on **ContentView**, but here I'd like to show you a better way: we're going to write an extension on **Bundle** to do it all in one centralized place.

Create another new Swift file, this time called Bundle-Decodable.swift. This will mostly use code you've seen before, but there's one small difference: previously we used **String(contentsOf:)** to load files into a string, but because **Codable** uses **Data** we are instead going to use **Data(contentsOf:)**. It works in just the same way as **String(contentsOf:)**: give it a file URL to load, and it either returns its contents or throws an error.

Add this to Bundle-Decodable.swift now:

```
extension Bundle {
    func decode(_ file: String) -> [String: Astronaut] {
        guard let url = self.url(forResource: file, withExtension:
nil) else {
            fatalError("Failed to locate \(file) in bundle.")
        }

        guard let data = try? Data(contentsOf: url) else {
            fatalError("Failed to load \(file) from bundle.")
        }

        let decoder = JSONDecoder()

        guard let loaded = try? decoder.decode([String:
Astronaut].self, from: data) else {
            fatalError("Failed to decode \(file) from bundle.")
        }
    }
}
```

Project 8: Moonshot

```
    return loaded
}
}
```

As you can see, that makes liberal use of `fatalError()`: if the file can't be found, loaded, or decoded the app will crash. As before, though, this will never actually happen unless you've made a mistake, for example if you forgot to copy the JSON file into your project.

Now, you might wonder why we used an extension here rather than a method, but the reason is about to become clear as we load that JSON into our content view. Add this property to the `ContentView` struct now:

```
let astronauts = Bundle.main.decode("astronauts.json")
```

Yes, that's all it takes. Sure, all we've done is just moved code out of `ContentView` and into an extension, but there's nothing wrong with that – anything we can do to help our views stay small and focused is a good thing.

If you want to double check that your JSON is loaded correctly, modify the default text view to this:

```
Text("\(astronauts.count)")
```

That should display 32 rather than "Hello World".

Using generics to load any kind of Codable data

We added a **Bundle** extension for loading one specific type of JSON data from our app bundle, but now we have a second type: missions.json. This contains slightly more complex JSON:

- Every mission has an ID number, which means we can use **Identifiable** easily.
- Every mission has a description, which is a free text string taken from Wikipedia.
- Every mission has an array of crew, where each crew member has a name and role.
- All but one missions has a launch date. Sadly, Apollo 1 never launched because a launch rehearsal cabin fire destroyed the command module and killed the crew.

Let's start converting that to code. Crew roles need to be represented as their own struct, storing the name string and role string. So, create a new Swift file called Mission.swift and give it this code:

```
struct CrewRole: Codable {  
    let name: String  
    let role: String  
}
```

As for the missions, this will be an ID integer, an array of **CrewRole**, and a description string. But what about the launch date – we might have one, but we also might not have one. What should *that* be?

Well, think about it: how does Swift represent this “maybe, maybe not” elsewhere? How would we store “might be a string, might be nothing at all”? I hope the answer is clear: we use optionals. In fact, if we mark a property as optional **Codable** will automatically skip over it if the value is missing from our input JSON.

So, add this second struct to Mission.swift now:

```
struct Mission: Codable, Identifiable {
```

Project 8: Moonshot

```
let id: Int
let launchDate: String?
let crew: [CrewRole]
let description: String
}
```

Before we look at how to load JSON into that, I want to demonstrate one more thing: our **CrewRole** struct was made specifically to hold data about missions, and as a result we can actually put the **CrewRole** struct *inside* the **Mission** struct like this:

```
struct Mission: Codable, Identifiable {
    struct CrewRole: Codable {
        let name: String
        let role: String
    }

    let id: Int
    let launchDate: String?
    let crew: [CrewRole]
    let description: String
}
```

This is called a *nested struct*, and is simply one struct placed inside of another. This won't affect our code in this project, but elsewhere it's useful to help keep your code organized: rather than saying **CrewRole** you'd write **Mission.CrewRole**. If you can imagine a project with several hundred custom types, adding this extra context can really help!

Now let's think about how we can load missions.json into an array of **Mission** structs. We already added a **Bundle** extension that loads some JSON file into a dictionary of **Astronaut** structs, so we could very easily copy and paste that, then tweak it so it loads missions rather than astronauts. However, there's a better solution: we can leverage Swift's generics system, which is an advanced feature we touched on lightly back in project 3.

Using generics to load any kind of Codable data

Generics allow us to write code that is capable of working with a variety of different types. In this project, we wrote the **Bundle** extension to work with dictionary of astronauts, but really we want to be able to handle dictionaries of astronauts, arrays of missions, or potentially lots of other things.

To make a method generic, we give it a placeholder for certain types. This is written in angle brackets (< and >) after the method name but before its parameters, like this:

```
func decode<T>(_ file: String) -> [String: Astronaut] {
```

We can use anything for that placeholder – we could have written “Type”, “TypeOfThing”, or even “Fish”; it doesn’t matter. “T” is a bit of a convention in coding, as a short-hand placeholder for “type”.

Inside the method, we can now use “T” everywhere we would use **[String: Astronaut]** – it is literally a placeholder for the type we want to work with. So, rather than returning **[String: Astronaut]** we would use this:

```
func decode<T>(_ file: String) -> T {
```

Be very careful: There is a big difference between **T** and **[T]**. Remember, **T** is a placeholder for whatever type we ask for, so if we say “decode our dictionary of astronauts,” then **T** becomes **[String: Astronaut]**. If we attempt to return **[T]** from **decode()** then we would actually be returning **[[String: Astronaut]]** – an array of dictionaries of astronauts!

Towards the end of the **decode()** method there’s another place where **[String: Astronaut]** is used:

```
guard let loaded = try? decoder.decode([String:  
Astronaut].self, from: data) else {
```

Again, please change that to **T**, like this:

```
guard let loaded = try? decoder.decode(T.self, from: data) else
```

Project 8: Moonshot

```
{
```

So, what we've said is that **decode()** will be used with some sort of type, such as **[String: Astronaut]**, and it should attempt to decode the file it has loaded to be that type.

If you try compiling this code, you'll see an error in Xcode: "Instance method 'decode(_:from:)' requires that 'T' conform to 'Decodable'". What it means is that **T** could be anything: it could be a dictionary of astronauts, or it could be a dictionary of something else entirely. The problem is that Swift can't be sure the type we're working with conforms to the **Codable** protocol, so rather than take a risk it's refusing to build our code.

Fortunately we can fix this with a *constraint*: we can tell Swift that **T** can be whatever we want, as long as that thing conforms to **Codable**. That way Swift knows it's safe to use, and will make sure we don't try to use the method with a type that *doesn't* conform to **Codable**.

To add the constraint, change the method signature to this:

```
func decode<T: Codable>(_ file: String) -> T {
```

If you try compiling again, you'll see that things *still* aren't working, but now it's for a different reason: "Generic parameter 'T' could not be inferred", over in the **astronauts** property of **ContentView**. This line worked fine before, but there has been an important change now: before **decode()** would always return a dictionary of astronauts, but now it returns anything we want as long as it conforms to **Codable**.

We know it will *still* return a dictionary of astronauts because the actual underlying data hasn't changed, but Swift *doesn't* know that. Our problem is that **decode()** can return any type that conforms to **Codable**, but Swift needs more information – it wants to know exactly what type it will be.

So, to fix this we need to use a type annotation so Swift knows exactly what **astronauts** will be:

```
let astronauts: [String: Astronaut] =
```

Using generics to load any kind of Codable data

```
Bundle.main.decode("astronauts.json")
```

Finally – after all that work! – we can now also load mission.json into another property in **ContentView**. Please add this below **astronauts**:

```
let missions: [Mission] = Bundle.main.decode("missions.json")
```

And *that* is the power of generics: we can use the same **decode()** method to load any JSON from our bundle into any Swift type that conforms to **Codable** – we don't need half a dozen variants of the same method.

Before we're done, there's one last thing I'd like to explain. Earlier you saw the message "Instance method 'decode(_:from:)' requires that 'T' conform to 'Decodable'", and you might have wondered what **Decodable** was – after all, we've been using **Codable** everywhere. Well, behind the scenes, **Codable** is just an alias for two separate protocols: **Encodable** and **Decodable**. You can use **Codable** if you want, or you can use **Encodable** and **Decodable** if you prefer being specific – it's down to you.

Formatting our mission view

Now that we have all our data in place, we can look at the design for our first screen: a grid of all the missions, next to their mission badges.

The assets we added earlier contain pictures named “apollo1@2x.png” and similar, which means they are accessible in the asset catalog as “apollo1”, “apollo12”, and so on. Our **Mission** struct has an **id** integer providing the number part, so we could use string interpolation such as "**apollo\\$(mission.id)**" to get our image name and "**Apollo \\$(mission.id)**" to get the formatted display name of the mission.

Here, though, we’re going to take a different approach: we’re going to add some computed properties to the **Mission** struct to send that same data back. The result will be the same – “apollo1” and “Apollo 1” – but now the code is in one place: our **Mission** struct. This means any other views can use the same data without having to repeat our string interpolation code, which in turn means if we change the way these things are formatted – i.e., we change the image names to “apollo-1” or something – then we can just change the property in **Mission** and have all our code update.

So, please add these two properties to the **Mission** struct now:

```
var displayName: String {  
    "Apollo \$(id)"  
}  
  
var image: String {  
    "apollo\$(id)"  
}
```

With those two in place we can now take a first pass at filling in **ContentView**: it will have a **NavigationView** with a title, a **LazyVGrid** using our **missions** array as input, and each row inside there will be a **NavigationLink** containing the image, name, and launch date of the mission. The only small complexity in there is that our launch date is an optional string, so we

need to use nil coalescing to make sure there's a value for the text view to display.

First, add this property to **ContentView** to define an adaptive column layout:

```
let columns = [  
    GridItem(.adaptive(minimum: 150))  
]
```

And now replace your existing body property with this:

```
NavigationView {  
    ScrollView {  
        LazyVGrid(columns: columns) {  
            ForEach(missions) { mission in  
                NavigationLink {  
                    Text("Detail view")  
                } label: {  
                    VStack {  
                        Image(mission.image)  
                            .resizable()  
                            .scaledToFit()  
                            .frame(width: 100, height: 100)  
  
                        VStack {  
                            Text(mission.displayName)  
                                .font(.headline)  
                            Text(mission.launchDate ?? "N/A")  
                                .font(.caption)  
                        }  
                        .frame(maxWidth: .infinity)  
                }  
            }  
        }  
    }  
}
```

Project 8: Moonshot

```
        }
    }
    .navigationTitle("Moonshot")
}
```

I know it looks pretty ugly, but we'll fix it right up in just a moment. First, let's focus on what we have so far: a scrolling, vertical grid that uses **resizable()**, **scaledToFit()**, and **frame()** to make the image occupy a 100x100 space while also maintaining its original aspect ratio.

Run the program now, and apart from the scrappy layout changes you'll notice the dates aren't great – although we can look at "1968-12-21" and understand it's the 21st of December 1968, it's still an unnatural date format for almost everyone. We can do better than this!

Swift's **JSONDecoder** type has a property called **dateDecodingStrategy**, which determines how it should decode dates. We can provide that with a **DateFormatter** instance that describes how our dates are formatted. In this instance, our dates are written as year-month-day, which in the world of **DateFormat** is written as "y-MM-dd" – that means "a year, then a dash, then a zero-padded month, then a dash, then a zero-padded day", with "zero-padded" meaning that January is written as "01" rather than "1".

Warning: Date formats are case sensitive! **mm** means "zero-padded minute" and **MM** means "zero-padded month."

So, open `Bundle-Decodable.swift` and add this code directly after `let decoder = JSONDecoder()`:

```
let formatter = DateFormatter()
formatter.dateFormat = "y-MM-dd"
decoder.dateDecodingStrategy = .formatted(formatter)
```

That tells the decoder to parse dates in the exact format we expect.

Tip: When working with dates it is often a good idea to be specific about your time zone, otherwise the user's own time zone is used when parsing the date and time. However, we're

also going to be *displaying* the date in the user's time zone, so there's no problem here.

If you run the code now... things will look exactly the same. Yes, nothing has changed, but that's OK: nothing has changed because Swift doesn't realize that **launchDate** is a date. After all, we declared it like this:

```
let launchDate: String?
```

Now that our decoding code understands how our dates are formatted, we can change that property to be an optional **Date**:

```
let launchDate: Date?
```

...and now our code won't even compile!

The problem *now* is this line of code in **ContentView.swift**:

```
Text(mission.launchDate ?? "N/A")
```

That attempts to use an optional **Date** inside a text view, or replace it with "N/A" if the date is empty. This is another place where a computed property works better: we can ask the mission itself to provide a formatted launch date that converts the optional date into a neatly formatted string or sends back "N/A" for missing dates.

This uses the same **formatted()** method we've used previously, so this should be somewhat familiar for you. Add this computed property to **Mission** now:

```
var formattedLaunchDate: String {
    launchDate?.formatted(date: .abbreviated, time: .omitted) ??
    "N/A"
}
```

And now replace the broken text view in **ContentView** with this:

Project 8: Moonshot

```
Text(mission.formattedLaunchDate)
```

With that change our dates will be rendered in a much more natural way, and, even better, will be rendered in whatever way is region-appropriate for the user – what you see isn’t necessarily what I see.

Now let’s focus on the bigger problem: our layout is pretty dull!

To spruce it up a little, I want to introduce you to two useful features: how to share custom app colors easily, and how to force a dark theme for our app.

First, colors. There are two ways to do this, and both are useful: you can add colors to your asset catalog with specific names, or you can add them as Swift extensions. They both have their advantages – using the asset catalog lets you work visually, but using code makes it easier to monitor changes using something like GitHub.

I’m not a big fan of the way asset catalogs force us to use strings for color names, just like we do with image names, so we’re going to take the alternative approach and place our color names into Swift as extensions.

If we make these extensions on **Color** we can use them in a handful of places in SwiftUI, but Swift gives us a better option with only a little more code. You see, **Color** conforms to a bigger protocol called **ShapeStyle** that is what lets us use colors, gradients, materials, and more as if they were the same thing.

This **ShapeStyle** protocol is what the **background()** modifier uses, so what we really want to do is extend **Color** but do so in a way that all the SwiftUI modifiers using **ShapeStyle** work too. This can be done with a very precise extension that literally says “we want to add functionality to **ShapeStyle**, but only for times when it’s being used as a color.”

To try this out, make a new Swift file called `Color-Theme.swift`, and give it this code:

```
extension ShapeStyle where Self == Color {
    static var darkBackground: Color {
```

```

    Color(red: 0.1, green: 0.1, blue: 0.2)
}

static var lightBackground: Color {
    Color(red: 0.2, green: 0.2, blue: 0.3)
}
}

```

That adds two new colors called **darkBackground** and **lightBackground**, each with precise values for red, green, and blue. But more importantly they place those inside a very specific extension that allows us to use those colors everywhere SwiftUI expects a **ShapeStyle**.

I want to put those new colors into action immediately. First, find the **VStack** containing the mission name and launch date – it should already have **.frame(maxWidth: .infinity)** on there, but I'd like you to change its modifier order to this:

```

.padding(.vertical)
.frame(maxWidth: .infinity)
.background(.lightBackground)

```

Next, I want to make the *outer* **VStack** – the one that is the whole label for our **NavLink** – look more like a box in our grid, which means drawing a line around it and clipping the shape just a little. To get that effect, add these modifiers to the end of it:

```

.clipShape(RoundedRectangle(cornerRadius: 10))
.overlay(
    RoundedRectangle(cornerRadius: 10)
        .stroke(.lightBackground)
)

```

Third, we need to add a little padding to get things away from their edges just a touch. That means adding some simple padding to the mission images, directly after their 100x100 frame:

Project 8: Moonshot

```
.padding()
```

Then also adding some horizontal and bottom padding to the grid:

```
.padding([.horizontal, .bottom])
```

Important: This should be added to the **LazyVGrid**, *not* to the **ScrollView**. If you pad the scroll view you're also padding its scrollbars, which will look odd.

Now we can replace the white background with the custom background color we created earlier – add this modifier to the **ScrollView**, after its **navigationTitle()** modifier:

```
.background(.darkBackground)
```

At this point our custom layout is almost done, but to finish up we're going to look at the remaining colors we have – the light blue color used for our mission text isn't great, and the "Moonshot" title is black at the top, which is impossible to read against our dark blue background.

We can fix the first of these by assigning specific colors to those two text fields:

```
 VStack {  
     Text(mission.displayName)  
         .font(.headline)  
         .foregroundColor(.white)  
     Text(mission.formattedLaunchDate)  
         .font(.caption)  
         .foregroundColor(.white.opacity(0.5))  
 }
```

Using a translucent white for the foreground color allows just a hint of the color behind to come through.

As for the Moonshot title, that belongs to our **NavigationView**, and will appear either black or

Formatting our mission view

white depending on whether the user is in light mode or dark mode. To fix this, we can tell SwiftUI our view prefers to be in dark mode *always* – this will cause the title to be in white no matter what, and will also darken other colors such as navigation bar backgrounds.

So, to finish up the design for this view please add this final modifier to the **ScrollView**, below its background color:

```
.preferredColorScheme(.dark)
```

If you run the app now you'll see we have a beautifully scrolling grid of mission data that will smoothly adapt to a wide range of device sizes, we have bright white navigation text and a dark navigation background no matter what appearance the user has enabled, and tapping any of our missions will bring in a temporary detail view. A great start!

Showing mission details with ScrollView and GeometryReader

When the user selects one of the Apollo missions from our main list, we want to show information about the mission: its mission badge, its mission description, and all the astronauts that were on the crew along with their roles. The first two of those aren't too hard, but the third requires a little more work because we need to match up crew IDs with crew details across our two JSON files.

Let's start simple and work our way up: make a new SwiftUI view called `MissionView.swift`. Initially this will just have a `mission` property so that we can show the mission badge and description, but shortly we'll add more to it.

In terms of layout, this thing needs to have a scrolling `VStack` with a resizable image for the mission badge, then a text view. We'll use `GeometryReader` to set the maximum width of the mission image, although through some trial and error I found that the mission badge worked best when it wasn't full width – somewhere between 50% and 70% width looked better, to avoid it becoming weirdly big on the screen.

Put this code into `MissionView.swift` now:

```
struct MissionView: View {
    let mission: Mission

    var body: some View {
        GeometryReader { geometry in
            ScrollView {
                VStack {
                    Image(mission.image)
                        .resizable()
                        .scaledToFit()
                        .frame(maxWidth: geometry.size.width * 0.6)
                }
            }
        }
    }
}
```

Showing mission details with ScrollView and GeometryReader

```
.padding(.top)

VStack(alignment: .leading) {
    Text("Mission Highlights")
        .font(.title.bold())
        .padding(.bottom, 5)

    Text(mission.description)
}
    .padding(.horizontal)
}
    .padding(.bottom)
}
}
.navigationTitle(mission.displayName)
.navigationBarTitleDisplayMode(.inline)
.background(.darkBackground)
}
}
```

Placing a **VStack** inside another **VStack** allows us to control alignment for one specific part of our view – our main mission image can be centered, while the mission details can be aligned to the leading edge.

Anyway, with that new view in place the code will no longer build, all because of the previews struct below it – that thing needs a **Mission** object passed in so it has something to render.

Fortunately, our **Bundle** extension is available here as well:

```
struct MissionView_Previews: PreviewProvider {
    static let missions: [Mission] =
        Bundle.main.decode("missions.json")

    static var previews: some View {
```

Project 8: Moonshot

```
    MissionView(mission: missions[0])
        .preferredColorScheme(.dark)
    }
}
```

Tip: This view will automatically have a dark color scheme because it's applied to the **NavigationView** in **ContentView**, but the **MissionView** preview doesn't know that so we need to enable it by hand.

If you look in the preview you'll see that's a good start, but the next part is trickier: we want to show the list of astronauts who took part in the mission below the description. Let's tackle that next...

Merging Codable structs

Below our mission description we want to show the pictures, names, and roles of each crew member, which means matching up data that came from two different JSON files.

If you remember, our JSON data is split across missions.json and astronauts.json. This eliminates duplication in our data, because some astronauts took part in multiple missions, but it also means we need to write some code to join our data together – to resolve “armstrong” to “Neil A. Armstrong”, for example. You see, on one side we have missions that know crew member “armstrong” had the role “Commander”, but has no idea who “armstrong” is, and on the other side we have “Neil A. Armstrong” and a description of him, but no concept that he was the commander on Apollo 11.

So, what we need to do is make our **MissionView** accept the mission that got tapped, along with our full astronauts dictionary, then have it figure out which astronauts actually took part in the launch.

Add this nested struct inside **MissionView** now:

```
struct CrewMember {  
    let role: String  
    let astronaut: Astronaut  
}
```

Now for the tricky part: we need to add a property to **MissionView** that stores an array of **CrewMember** objects – these are the fully resolved role / astronaut pairings. At first that’s as simple as adding another property:

```
let crew: [CrewMember]
```

But then how do we *set* that property? Well, think about it: if we make this view be handed its mission and all astronauts, we can loop over the mission crew, then for each crew member look in the dictionary to find the one that has a matching ID. When we find one we can convert

Project 8: Moonshot

that and their role into a **CrewMember** object, but if we don't it means somehow we have a crew role with an invalid or unknown name.

That latter case should never happen. To be clear, if you've added some JSON to your project that points to missing data in your app, you've made a fundamental mistake – it's not the kind of thing you should try to write error handling for at runtime, because it should never be allowed to happen in the first place. So, this is a great example of where **fatalError()** is useful: if we can't find an astronaut using their ID, we should exit immediately and complain loudly.

Let's put all that into code, using a custom initializer for **MissionView**. Like I said, this will accept the mission it represents along with all the astronauts, and its job is to store the mission away then figure out the array of resolved astronauts.

Here's the code:

```
init(mission: Mission, astronauts: [String: Astronaut]) {
    self.mission = mission

    self.crew = mission.crew.map { member in
        if let astronaut = astronauts[member.name] {
            return CrewMember(role: member.role, astronaut:
astronaut)
        } else {
            fatalError("Missing \(member.name)")
        }
    }
}
```

As soon as that code is in, our preview struct will stop working again because it needs more information. So, add a second call to **decode()** there so it loads all the astronauts, then passes those in too:

```

struct MissionView_Previews: PreviewProvider {
    static let missions: [Mission] =
    Bundle.main.decode("missions.json")
    static let astronauts: [String: Astronaut] =
    Bundle.main.decode("astronauts.json")

    static var previews: some View {
        MissionView(mission: missions[0], astronauts: astronauts)
        .preferredColorScheme(.dark)
    }
}

```

Now that we have all our astronaut data, we can show this directly below the mission description using a horizontal scroll view. We're also going to add a little extra styling to the astronaut pictures to make them look better, using a capsule clip shape and overlay.

Add this code just after the **VStack(alignment: .leading)**:

```

ScrollView(.horizontal, showsIndicators: false) {
    HStack {
        ForEach(crew, id: \.role) { crewMember in
            NavigationLink {
                Text("Astronaut details")
            } label: {
                HStack {
                    Image(crewMember.astronaut.id)
                    .resizable()
                    .frame(width: 104, height: 72)
                    .clipShape(Capsule())
                    .overlay(
                        Capsule()
                        .strokeBorder(.white, lineWidth: 1)
                    )
                }
            }
        }
    }
}

```

Project 8: Moonshot

```
    VStack(alignment: .leading) {
        Text(crewMember.astronaut.name)
            .foregroundColor(.white)
            .font(.headline)
        Text(crewMember.role)
            .foregroundColor(.secondary)
    }
}
.padding(.horizontal)
}
}
}
```

Why *after* the **VStack** rather than inside? Because scroll views work best when they take full advantage of the available screen space, which means they should scroll edge to edge. If we put this inside our **VStack** it would have the same padding as the rest of our text, which means it would scroll strangely – the crew would get clipped as it hit the leading edge of our **VStack**, which looks odd.

We'll make that **NavLink** do something more useful shortly, but first we need to modify the **NavLink** in **ContentView** – it pushes to **Text("Detail View")** right now, but please replace it with this:

```
MissionView(mission: mission, astronauts: astronauts)
```

Now go ahead and run the app in the simulator – it's starting to become useful!

Before you move on, try spending a few minutes customizing the way the astronauts are shown – I've used a capsule clip shape and overlay, but you could try circles or rounded rectangles, you could use different fonts or larger images, or even add some way of marking who the mission commander was.

In my project, I think it would be useful to add a little visual separation in our mission view, so that the mission badge, description, and crew are more clearly split up.

SwiftUI does provide a dedicated **Divider** view for creating a visual divide in your layout, but it's not customizable – it's always just a skinny line. So, to get something a little more useful, I'm going to draw a custom divider to break up our view.

First, place this directly before the “Mission Highlights” text:

```
Rectangle()  
    .frame(height: 2)  
    .foregroundColor(.lightBackground)  
    .padding(.vertical)
```

Now place another one of those – the same code – directly after the **mission.description** text.

Much better!

To finish up this view, I'm going to add a title before our crew, but this needs to be done carefully. You see, although this relates to the scroll view, it needs to have the same padding as the rest of our text. So, the best place for this is inside the **VStack**, directly after the previous rectangle:

```
Text("Crew")  
    .font(.title.bold())  
    .padding(.bottom, 5)
```

You don't need to put it there – if you wanted we could move it outside the **VStack** then apply padding individually to that text view. However, if you do that make sure you apply the same amount of padding to keep everything neatly aligned.

Finishing up with one last view

To finish this program we're going to make a third and final view to display astronaut details, which will be reached by tapping one of the astronauts in the mission view. This should mostly just be practice for you, but I hope it also shows you the importance of **NavigationView** – we're digging deeper into our app's information, and the presentation of views sliding in and out really drives that home to the user.

Start by making a new SwiftUI view called **AstronautView**. This will have a single **Astronaut** property so it knows what to show, then it will lay that out using a similar **ScrollView/VStack** combination as we had in **MissionView**. Give it this code:

```
struct AstronautView: View {
    let astronaut: Astronaut

    var body: some View {
        ScrollView {
            VStack {
                Image(astronaut.id)
                    .resizable()
                    .scaledToFit()

                Text(astronaut.description)
                    .padding()
            }
        }
        .background(.darkBackground)
        .navigationTitle(astronaut.name)
        .navigationBarTitleDisplayMode(.inline)
    }
}
```

Once again we need to update the preview so that it creates its view with some data:

```
struct AstronautView_Previews: PreviewProvider {
    static let astronauts: [String: Astronaut] =
    Bundle.main.decode("astronauts.json")

    static var previews: some View {
        AstronautView(astronaut: astronauts["aldrin"]!)
            .preferredColorScheme(.dark)
    }
}
```

Now we can present that from the **NavigationLink** inside **MissionView**. This points to **Text("Astronaut details")** right now, but we can update it to point to our new **AstronautView** instead:

```
AstronautView(astronaut: crewMember.astronaut)
```

That was easy, right? But if you run the app now you'll see how natural it makes our user interface feel – we start at the broadest level of information, showing all our missions, then tap to select one specific mission, then tap to select one specific astronaut. iOS takes care of animating in the new views, but also providing back buttons and swipes to return to previous views.

Moonshot: Wrap up

This app is the most complex one we've built so far. Yes, there are multiple views, but we also strayed away from lists and forms and into our own scrolling layouts, using **GeometryReader** to get precise sizes to make the most of our space.

But this was also the most complex *Swift* code we've written so far – generics are an incredibly powerful feature, and once you add in constraints you open up a huge range of functionality that lets you save time while also gaining flexibility.

You're also now starting to see how useful **Codable** is: its ability to decode a hierarchy of data in one pass is invaluable, which is why it's central to so many Swift apps.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Add the launch date to **MissionView**, below the mission badge. You might choose to format this differently given that more space is available, but it's down to you.
2. Extract one or two pieces of view code into their own new SwiftUI views – the horizontal scroll view in **MissionView** is a great candidate, but if you followed my styling then you could also move the **Rectangle** dividers out too.
3. For a tough challenge, add a toolbar item to **ContentView** that toggles between showing missions as a grid and as a list.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here: [Solution to Moonshot](#). If you don't already subscribe, you can start a free trial today.

Tip: For that last one, your best bet is to make all your grid code and all your list code two separate views, and switch between them using an `if` condition in `ContentView`. You can't attach SwiftUI modifiers to an `if` condition, but you can wrap that condition in a `Group` then attach modifiers to there, like this:

```
Group {
    if showingGrid {
        GridLayout(astronauts: astronauts, missions: missions)
    } else {
        ListLayout(astronauts: astronauts, missions: missions)
    }
}.navigationTitle("My Group")
```

You might hit some speed bumps trying to style your list, because they have a particular look and feel on iOS by default. Try attaching `.listStyle(.plain)` to your list, then something like `.listRowBackground(Color.darkBackground)` to the contents of your list row – that should get you a long way towards your goal.

Project 9

Drawing

Use shapes, paths, colors, and more to create custom art for your app

Drawing: Introduction

In this technique project we're going to take a close look at drawing in SwiftUI, including creating custom paths and shapes, animating your changes, solving performance problems, and more – it's a really big topic, and deserves close attention.

Behind the scenes, SwiftUI uses the same drawing system that we have on the rest of Apple's frameworks: Core Animation and Metal. Most of the time Core Animation is responsible for our drawing, whether that's custom paths and shapes or UI elements such as **TextField**, but when things really get complex we can move down to Metal – Apple's low-level framework that's optimized for complex drawing. One of the neat features of SwiftUI is that these two are almost interchangeable: we can move from Core Animation to Metal with one small change.

Anyway, we have lots to cover so please create a new App project called Drawing and let's dive in...

Creating custom paths with SwiftUI

SwiftUI gives us a dedicated **Path** type for drawing custom shapes. It's very low level, by which I mean you will usually want to wrap it in something else in order for it to be more useful, but as it's the building block that underlies other work we'll do we're going to start there.

Just like colors, gradients, and shapes, paths are views in their own right. This means we can use them just like text views and images, although as you'll see it's a bit clumsy.

Let's start with a simple shape: drawing a triangle. There are a few ways of creating paths, including one that accepts a closure of drawing instructions. This closure must accept a single parameter, which is the path to draw into. I realize this can be a bit brain-bending at first, because we're creating a path and inside the initializer for the path we're getting passed the path to draw into, but think of it like this: SwiftUI is creating an empty path for us, then giving us the chance to add to it as much as we want.

Paths have lots of methods for creating shapes with squares, circles, arcs, and lines. For our triangle we need to move to a starting position, then add three lines like this:

```
Path { path in
    path.move(to: CGPoint(x: 200, y: 100))
    path.addLine(to: CGPoint(x: 100, y: 300))
    path.addLine(to: CGPoint(x: 300, y: 300))
    path.addLine(to: CGPoint(x: 200, y: 100))
}
```

We haven't used **CGPoint** before, but I did sneak in a quick reference to **CGSize** back in project 6. "CG" is short for Core Graphics, which provides a selection of basic types that lets us reference X/Y coordinates (**CGPoint**), widths and heights (**CGSize**), and rectangular frames (**CGRect**).

When our triangle code runs, you'll see a large black triangle. *Where* you see it relative to your screen depends on what simulator you are using, which is part of the problem of these raw paths: we need to use exact coordinates, so if you want to use a path by itself you either need to accept that sizing across all devices or use something like **GeometryReader** to scale them relative to their container.

We'll look at a better option shortly, but first let's look at coloring our path. One option is to use the **fill()** modifier, like this:

```
Path { path in
    path.move(to: CGPoint(x: 200, y: 100))
    path.addLine(to: CGPoint(x: 100, y: 300))
    path.addLine(to: CGPoint(x: 300, y: 300))
    path.addLine(to: CGPoint(x: 200, y: 100))
}
.fill(.blue)
```

We can also use the **stroke()** modifier to draw around the path rather than filling it in:

```
.stroke(.blue, lineWidth: 10)
```

That doesn't look quite right, though – the bottom corners of our triangle are nice and sharp, but the top corner is broken. This happens because SwiftUI makes sure lines connect up neatly with what comes before and after rather than just being a series of individual lines, but our last line has nothing after it so there's no way to make a connection.

One way to fix this is ask SwiftUI to close the subpath, which is the shape we've drawn inside our path:

```
Path { path in
    path.move(to: CGPoint(x: 200, y: 100))
    path.addLine(to: CGPoint(x: 100, y: 300))
    path.addLine(to: CGPoint(x: 300, y: 300))
}
```

Project 9: Drawing

```
    path.addLine(to: CGPoint(x: 200, y: 100))
    path.closeSubpath()
}
.stroke(.blue, lineWidth: 10)
```

An alternative is to use SwiftUI's dedicated **StrokeStyle** struct, which gives us control over how every line should be connected to the line after it (line join) and how every line should be drawn when it ends without a connection after it (line cap). This is particularly useful because one of the options for join and cap is **.round**, which creates gently rounded shapes:

```
.stroke(.blue, style: StrokeStyle(lineWidth: 10,
lineCap: .round, lineJoin: .round))
```

With that in place you can remove the call to **path.closeSubpath()**, because it's no longer needed.

Using rounded corners solves the problem of our rough edges, but it doesn't solve the problem of fixed coordinates. For that we need to move on from paths and look at something more complex: *shapes*.

Paths vs shapes in SwiftUI

SwiftUI enables custom drawing with two subtly different types: paths and shapes. A path is a series of drawing instructions such as “start here, draw a line to here, then add a circle there”, all using absolute coordinates. In contrast, a shape has no idea where it will be used or how big it will be used, but instead will be asked to draw itself inside a given rectangle.

Helpfully, shapes are built using paths, so once you understand paths shapes are easy. Also, just like paths, colors, and gradients, shapes are views, which means we can use them alongside text views, images, and so on.

SwiftUI implements **Shape** as a protocol with a single required method: given the following rectangle, what path do you want to draw? This will still create and return a path just like using a raw path directly, but because we’re handed the size the shape will be used at we know exactly how big to draw our path – we no longer need to rely on fixed coordinates.

For example, previously we created a triangle using a **Path**, but we could wrap that in a shape to make sure it automatically takes up all the space available like this:

```
struct Triangle: Shape {
    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: rect.midX, y: rect.minY))
        path.addLine(to: CGPoint(x: rect minX, y: rect maxY))
        path.addLine(to: CGPoint(x: rect maxX, y: rect maxY))
        path.addLine(to: CGPoint(x: rect.midX, y: rect.minY))

        return path
    }
}
```

That job is made much easier by **CGRect**, which provides helpful properties such as **minX**

Project 9: Drawing

(the smallest X value in the rectangle), **maxX** (the largest X value in the rectangle), and **midX** (the mid-point between **minX** and **maxX**).

We could then create a red triangle at a precise size like this:

```
Triangle()  
    .fill(.red)  
    .frame(width: 300, height: 300)
```

Shapes also support the same **StrokeStyle** parameter for creating more advanced strokes:

```
Triangle()  
    .stroke(.red, style: StrokeStyle(lineWidth: 10,  
lineCap: .round, lineJoin: .round))  
    .frame(width: 300, height: 300)
```

The key to understanding the difference between **Path** and **Shape** is reusability: paths are designed to do one specific thing, whereas shapes have the flexibility of drawing space and can also accept parameters to let us customize them further.

To demonstrate this, we could create an **Arc** shape that accepts three parameters: start angle, end angle, and whether to draw the arc clockwise or not. This might seem simple enough, particularly because **Path** has an **addArc()** method, but as you'll see it has a couple of interesting quirks.

Let's start with the simplest version of an arc shape:

```
struct Arc: Shape {  
    var startAngle: Angle  
    var endAngle: Angle  
    var clockwise: Bool  
  
    func path(in rect: CGRect) -> Path {  
        var path = Path()
```

```

    path.addArc(center: CGPoint(x: rect.midX, y: rect.midY),
radius: rect.width / 2, startAngle: startAngle, endAngle:
endAngle, clockwise: clockwise)

    return path
}

}

```

We can now create an arc like this:

```

Arc(startAngle: .degrees(0), endAngle: .degrees(110),
clockwise: true)
.stroke(.blue, lineWidth: 10)
.frame(width: 300, height: 300)

```

If you look at the preview of our arc, chances are it looks nothing like you expect. We asked for an arc from 0 degrees to 110 degrees with a clockwise rotation, but we appear to have been given an arc from 90 degrees to 200 degrees with a counterclockwise rotation.

What's happening here is two-fold:

1. In the eyes of SwiftUI 0 degrees is not straight upwards, but instead directly to the right.
2. Shapes measure their coordinates from the bottom-left corner rather than the top-left corner, which means SwiftUI goes the other way around from one angle to the other. This is, in my not very humble opinion, extremely alien.

We can fix both of those problems with a new **path(in:)** method that subtracts 90 degrees from the start and end angles, and also flips the direction so SwiftUI behaves the way nature intended:

```

func path(in rect: CGRect) -> Path {
    let rotationAdjustment = Angle.degrees(90)
    let modifiedStart = startAngle - rotationAdjustment

```

Project 9: Drawing

```
let modifiedEnd = endAngle - rotationAdjustment

var path = Path()
path.addArc(center: CGPoint(x: rect.midX, y: rect.midY),
radius: rect.width / 2, startAngle: modifiedStart, endAngle:
modifiedEnd, clockwise: !clockwise)

return path
}
```

Run that code and see what you think – to me it produces a much more natural way of working, and neatly isolates SwiftUI's drawing behavior.

Adding strokeBorder() support with InsettableShape

If you create a shape without a specific size, it will automatically expand to occupy all available space. For example, this will create a circle that fills our view, giving it a 40-point blue border:

```
struct ContentView: View {  
    var body: some View {  
        Circle()  
            .stroke(.blue, lineWidth: 40)  
    }  
}
```

Take a close look at the left and right edges of the border – do you notice how they are cut off?

What you’re seeing here is a side effect of the way SwiftUI draws borders around shapes. If you handed someone a pencil outline of a circle and asked them to draw over that circle with a thick pen, they would trace the exact line of the circle – about half the pen would be inside the line, and half outside. This is what SwiftUI is doing for us, but where our shapes go to the edge of the screen it means the outside part of the border ends up beyond our screen edges.

Now try using this circle instead:

```
Circle()  
    .strokeBorder(.blue, lineWidth: 40)
```

That changes **stroke()** to **strokeBorder()** and now we get a better result: all our border is visible, because Swift strokes the inside of the circle rather than centering on the line.

Previously we built an **Arc** shape like this:

```
struct Arc: Shape {
```

Project 9: Drawing

```
var startAngle: Angle
var endAngle: Angle
var clockwise: Bool

func path(in rect: CGRect) -> Path {
    let rotationAdjustment = Angle.degrees(90)
    let modifiedStart = startAngle - rotationAdjustment
    let modifiedEnd = endAngle - rotationAdjustment

    var path = Path()
    path.addArc(center: CGPoint(x: rect.midX, y: rect.midY),
    radius: rect.width / 2, startAngle: modifiedStart, endAngle:
    modifiedEnd, clockwise: !clockwise)

    return path
}

}
```

Just like **Circle**, that automatically takes up all available space. However, this kind of code won't work:

```
Arc(startAngle: .degrees(-90), endAngle: .degrees(90),
clockwise: true)
    .strokeBorder(.blue, lineWidth: 40)
```

If you open Xcode's error message you'll see it says "Value of type 'Arc' has no member 'strokeBorder'" – that is, the **strokeBorder()** modifier just doesn't exist on **Arc**.

There is a small but important difference between SwiftUI's **Circle** and our **Arc**: both conform to the **Shape** protocol, but **Circle** *also* conforms to a second protocol called **InsettableShape**. This is a shape that can be inset – reduced inwards – by a certain amount to produce another shape. The inset shape it produces can be any other kind of insettable shape, but realistically it should be the same shape just in a smaller rectangle.

Adding strokeBorder() support with InsettableShape

To make **Arc** conform to **InsettableShape** we need to add one extra method to it: **inset(by:)**. This will be given the inset amount (half the line width of our stroke), and should return a new kind of insettable shape – in our instance that means we should create an inset arc. The problem is, we don't *know* the arc's actual size, because **path(in:)** hasn't been called yet.

It turns out the solution is pretty simple: if we give our **Arc** shape a new **insetAmount** property that defaults to 0, we can just add to that whenever **inset(by:)** is called. Adding to the inset allows us to call **inset(by:)** multiple times if needed, for example if we wanted to call it once by hand then use **strokeBorder()**.

First, add this new property to **Arc**:

```
var insetAmount = 0.0
```

Now give it this **inset(by:)** method:

```
func inset(by amount: CGFloat) -> some InsettableShape {
    var arc = self
    arc.insetAmount += amount
    return arc
}
```

Important: This is one of the very few places where we need to use **CGFloat**, which is an archaic form of **Double** that, somewhat bizarrely, wormed its way into SwiftUI. It gets used in many other places too, but mostly Swift lets us use **Double** instead!

The **amount** parameter being passed in should be applied to all edges, which in the case of arcs means we should use it to reduce our draw radius. So, change the **addArc()** call inside **path(in:)** to be this:

```
path.addArc(center: CGPoint(x: rect.midX, y: rect.midY),
            radius: rect.width / 2 - insetAmount, startAngle:
            modifiedStart, endAngle: modifiedEnd, clockwise: !clockwise)
```

Project 9: Drawing

With that change we can now make **Arc** conform to **InsettableShape** like this:

```
struct Arc: InsettableShape {
```

Note: **InsettableShape** actually builds upon **Shape**, so there's no need to add both there.

Transforming shapes using CGAffineTransform and even-odd fills

When you move beyond simple shapes and paths, two useful features of SwiftUI come together to create some beautiful effects with remarkably little work. The first is **CGAffineTransform**, which describes how a path or view should be rotated, scaled, or sheared; and the second is even-odd fills, which allow us to control how overlapping shapes should be rendered.

To demonstrate both of these, we're going to create a flower shape out of several rotated ellipse petals, with each ellipse positioned around a circle. The mathematics behind this is relatively straightforward, with one catch: **CGAffineTransform** measures angles in radians rather than degrees. If it's been a while since you were at school, the least you need to know is this: 3.141 radians is equal to 180 degrees, so 3.141 radians multiplied by 2 is equal to 360 degrees. And the 3.141 isn't a coincidence: the actual value is the mathematical constant pi.

So, what we're going to do is as follows:

- Create a new empty path.
- Count from 0 up to pi multiplied by 2 (360 degrees in radians), counting in one eighth of pi each time – this will give us 16 petals.
- Create a rotation transform equal to the current number.
- Add to that rotation a movement equal to half the width and height of our draw space, so each petal is centered in our shape.
- Create a *new* path for a petal, equal to an ellipse of a specific size.
- Apply our transform to that ellipse so it's moved into position.
- Add that petal's path to our main path.

This will make more sense once you see the code running, but first I want to add three more small things:

Project 9: Drawing

1. Rotating something then moving it does not produce the same result as moving then rotating, because when you rotate it first the direction it moves will be different from if it were not rotated.
2. To really help you understand what's going on, we'll be making our petal ellipses use a couple of properties we can pass in externally.
3. Ranges such as `1...5` are great if you want to count through numbers one at a time, but if you want to count in 2s, or in our case count in “`pi/8`’s, you should use `stride(from:to:by:)` instead.

Alright, enough talk – add this shape to your project now:

```
struct Flower: Shape {  
    // How much to move this petal away from the center  
    var petalOffset: Double = -20  
  
    // How wide to make each petal  
    var petalWidth: Double = 100  
  
    func path(in rect: CGRect) -> Path {  
        // The path that will hold all petals  
        var path = Path()  
  
        // Count from 0 up to pi * 2, moving up pi / 8 each time  
        for number in stride(from: 0, to: Double.pi * 2, by:  
Double.pi / 8) {  
            // rotate the petal by the current value of our loop  
            let rotation = CGAffineTransform(rotationAngle: number)  
  
            // move the petal to be at the center of our view  
            let position =  
rotation.concatenating(CGAffineTransform(translationX:  
rect.width / 2, y: rect.height / 2))
```

Transforming shapes using CGAffineTransform and even-odd fills

```
// create a path for this petal using our properties plus
a fixed Y and height
let originalPetal = Path(ellipseIn: CGRect(x:
petalOffset, y: 0, width: petalWidth, height: rect.width / 2))

// apply our rotation/position transformation to the
petal
let rotatedPetal = originalPetal.applying(position)

// add it to our main path
path.addPath(rotatedPetal)

}

// now send the main path back
return path
}

}
```

I realize that's quite a lot of code, but hopefully it will become clearer when you try it out. Modify your **ContentView** to this:

```
struct ContentView: View {
    @State private var petalOffset = -20.0
    @State private var petalWidth = 100.0

    var body: some View {
        VStack {
            Flower(petalOffset: petalOffset, petalWidth: petalWidth)
                .stroke(.red, lineWidth: 1)

            Text("Offset")
            Slider(value: $petalOffset, in: -40...40)
        }
    }
}
```

Project 9: Drawing

```
    .padding([.horizontal, .bottom])  
  
    Text("Width")  
    Slider(value: $petalWidth, in: 0...100)  
    .padding(.horizontal)  
}  
}  
}
```

Now try that out. You should be able to see exactly how the code works once you start dragging the offset and width sliders around – it's just a series of rotated ellipses, placed in a circular formation.

That in itself is interesting, but with one small change we can go from *interesting* to *sublime*. If you look at the way our ellipses are being drawn, they overlap frequently – sometimes one ellipse is drawn over another, and sometimes over several others.

If we fill our path using a solid color, we get a fairly unimpressive result. Try it like this:

```
Flower(petalOffset: petalOffset, petalWidth: petalWidth)  
    .fill(.red)
```

But as an alternative, we can fill the shape using the even-odd rule, which decides whether part of a path should be colored depending on the overlaps it contains. It works like this:

- If a path has no overlaps it will be filled.
- If another path overlaps it, the overlapping part won't be filled.
- If a third path overlaps the previous two, then it *will* be filled.
- ...and so on.

Only the parts that actually overlap are affected by this rule, and it creates some remarkably beautiful results. Even better, Swift UI makes it trivial to use, because whenever we call `fill()` on a shape we can pass a `FillStyle` struct that asks for the even-odd rule to be enabled.

Transforming shapes using CGAffineTransform and even-odd fills

Try it out with this:

```
Flower(petalOffset: petalOffset, petalWidth: petalWidth)  
    .fill(.red, style: FillStyle(eoFill: true))
```

Now run the program and play – honestly, given how little work we've done the results are quite entrancing!

Creative borders and fills using ImagePaint

SwiftUI relies heavily on protocols, which can be a bit confusing when working with drawing. For example, we can use **Color** as a view, but it also conforms to **ShapeStyle** – a different protocol used for fills, strokes, and borders.

In practice, this means we can modify the default text view so that it has a red background:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .background(.red)
```

Or a red border:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .border(.red, width: 30)
```

In contrast, we can use an image for the background:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .background(Image("Example"))
```

But using the same image as a border won't work:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .border(Image("Example"), width: 30)
```

This makes sense if you think about it – unless the image is the exact right size, you have very little control over how it should look.

To resolve this, SwiftUI gives us a dedicated type that wraps images in a way that we have complete control over how they should be rendered, which in turn means we can use them for borders and fills without problem.

The type is called **ImagePaint**, and it's created using one to three parameters. At the very least you need to give it an **Image** to work with as its first parameter, but you can also provide a rectangle within that image to use as the source of your drawing specified in the range of 0 to 1 (the second parameter), and a scale for that image (the third parameter). Those second and third parameters have sensible default values of “the whole image” and “100% scale”, so you can sometimes ignore them.

As an example, we could render an example image using a scale of 0.2, which means it's shown at 1/5th the normal size:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .border(ImagePaint(image: Image("Example"), scale: 0.2),
width: 30)
```

If you want to try using the **sourceRect** parameter, make sure you pass in a **CGRect** of relative sizes and positions: 0 means “start” and 1 means “end”. For example, this will show the entire width of our example image, but only the middle half:

```
Text("Hello World")
    .frame(width: 300, height: 300)
    .border(ImagePaint(image: Image("Example"), sourceRect:
CGRect(x: 0, y: 0.25, width: 1, height: 0.5), scale: 0.1),
width: 30)
```

It's worth adding that **ImagePaint** can be used for view backgrounds and also shape strokes. For example, we could create a capsule with our example image tiled as its stroke:

```
Capsule()
```

Project 9: Drawing

```
.strokeBorder(ImagePaint(image: Image("Example"), scale:  
0.1), lineWidth: 20)  
.frame(width: 300, height: 200)
```

ImagePaint will automatically keep tiling its image until it has filled its area – it can work with backgrounds, strokes, borders, and fills of any size.

Enabling high-performance Metal rendering with drawingGroup()

SwiftUI uses Core Animation for its rendering by default, which offers great performance out of the box. However, for complex rendering you might find your code starts to slow down – anything below 60 frames per second (FPS) is a problem, but really you ought to aim higher because many iOS devices now render at 120fps.

To demonstrate this, let's look at some example code. We're going to create a color-cycling view that renders concentric circles in a range of colors. The result will look like a radial gradient, but we're going to add two properties to make it more customizable: one to control how many circles should be drawn, and one to control the color cycle – it will be able to move the gradient start and end colors around.

We can get a color cycling effect by using the **Color(hue:saturation:brightness:)** initializer: hue is a value from 0 to 1 controlling the kind of color we see – red is both 0 and 1, with all other hues in between. To figure out the hue for a particular circle we can take our circle number (e.g. 25), divide that by how many circles there are (e.g. 100), then add our color cycle amount (e.g. 0.5). So, if we were circle 25 of 100 with a cycle amount of 0.5, our hue would be 0.75.

One small complexity here is that hues don't automatically wrap after we reach 1.0, which means a hue of 1.0 is equal to a hue of 0.0, but a hue of 1.2 is *not* equal to a hue of 0.2. As a result, we're going to wrap the hue by hand: if it's over 1.0 we'll subtract 1.0, to make sure it always lies in the range of 0.0 to 1.0.

Here's the code:

```
struct ColorCyclingCircle: View {  
    var amount = 0.0  
    var steps = 100  
  
    var body: some View {
```

Project 9: Drawing

```
    ZStack {
        ForEach(0..
```

We can now use that in a layout, binding its color cycle to a local property controlled by a slider:

```
struct ContentView: View {
    @State private var colorCycle = 0.0

    var body: some View {
        VStack {
            ColorCyclingCircle(amount: colorCycle)
                .frame(width: 300, height: 300)
```

Enabling high-performance Metal rendering with drawingGroup()

```
    Slider(value: $colorCycle)
}
}
}
```

If you run the app you'll see we have a neat color wave effect controlled entirely by dragging around the slider, and it works really smoothly.

What you're seeing right now is powered by Core Animation, which means it will turn our 100 circles into 100 individual views being drawn onto the screen. This is computationally expensive, but as you can see it works well enough – we get smooth performance.

However, if we increase the complexity a little we'll find things aren't quite so rosy. Replace the existing **strokeBorder()** modifier with this one:

```
.strokeBorder(
    LinearGradient(
        gradient: Gradient(colors: [
            color(for: value, brightness: 1),
            color(for: value, brightness: 0.5)
        ]),
        startPoint: .top,
        endPoint: .bottom
    ),
    lineWidth: 2
)
```

That now renders a gentle gradient, showing bright colors at the top of the circle down to darker colors at the bottom. And *now* when you run the app you'll find it runs much slower – SwiftUI is struggling to render 100 gradients as part of 100 separate views.

We can fix this by applying one new modifier, called **drawingGroup()**. This tells SwiftUI it should render the contents of the view into an off-screen image before putting it back onto the

Project 9: Drawing

screen as a single rendered output, which is significantly faster. Behind the scenes this is powered by Metal, which is Apple's framework for working directly with the GPU for extremely fast graphics.

So, modify the **ColorCyclingCircle** body to this:

```
var body: some View {
    ZStack {
        // existing code...
    }
    .drawingGroup()
}
```

Now run it again – with that one tiny addition you'll now find we get everything rendered correctly and we're also back at full speed even with the gradients.

Important: The **drawingGroup()** modifier is helpful to know about and to keep in your arsenal as a way to solve performance problems when you hit them, but you should *not* use it that often. Adding the off-screen render pass might slow down SwiftUI for simple drawing, so you should wait until you have an actual performance problem before trying to bring in **drawingGroup()**.

Special effects in SwiftUI: blurs, blending, and more

SwiftUI gives us extraordinary control over how views are rendered, including the ability to apply real-time blurs, blend modes, saturation adjustment, and more.

Blend modes allow us to control the way one view is rendered on top of another. The default mode is `.normal`, which just draws the pixels from the new view onto whatever is behind, but there are lots of options for controlling color and opacity.

As an example, we could draw an image inside a `ZStack`, then add a red rectangle on top that is drawn with the *multiply* blend mode:

```
ZStack {  
    Image("PaulHudson")  
  
    Rectangle()  
        .fill(.red)  
        .blendMode(.multiply)  
    }  
    .frame(width: 400, height: 500)  
    .clipped()
```

“Multiply” is so named because it multiplies each source pixel color with the destination pixel color – in our case, each pixel of the image and each pixel of the rectangle on top. Each pixel has color values for RGBA, ranging from 0 (none of that color) through to 1 (all of that color), so the highest resulting color will be 1x1, and the lowest will be 0x0.

Using multiply with a solid color applies a really common tint effect: blacks stay black (because they have the color value of 0, so regardless of what you put on top multiplying by 0 will produce 0), whereas lighter colors become various shades of the tint.

In fact, multiply is so common that there’s a shortcut modifier that means we can avoid using a

Project 9: Drawing

ZStack:

```
var body: some View {
    Image("PaulHudson")
        .colorMultiply(.red)
}
```

There are lots of other blend modes to choose from, and it's worth spending some time experimenting to see how they work. Another popular effect is called *screen*, which does the opposite of multiply: it inverts the colors, performs a multiply, then inverts them again, resulting in a brighter image rather than a darker image.

As an example, we could render three circles at various positions inside a **ZStack**, then use a slider to control their size and overlap:

```
struct ContentView: View {
    @State private var amount = 0.0

    var body: some View {
        VStack {
            ZStack {
                Circle()
                    .fill(.red)
                    .frame(width: 200 * amount)
                    .offset(x: -50, y: -80)
                    .blendMode(.screen)

                Circle()
                    .fill(.green)
                    .frame(width: 200 * amount)
                    .offset(x: 50, y: -80)
                    .blendMode(.screen)
            }
        }
    }
}
```

```

        Circle()
            .fill(.blue)
            .frame(width: 200 * amount)
            .blendMode(.screen)
    }
    .frame(width: 300, height: 300)

    Slider(value: $amount)
        .padding()
    }
    .frame(maxWidth: .infinity, maxHeight: .infinity)
    .background(.black)
    .ignoresSafeArea()
}
}

```

If you're particularly observant, you might notice that the fully blended color in the center isn't quite white – it's a very pale lilac color. The reason for this is that **Color.red**, **Color.green**, and **Color.blue** aren't fully those colors; you're not seeing pure red when you use **Color.red**. Instead, you're seeing SwiftUI's adaptive colors that are designed to look good in both dark mode and light mode, so they are a custom blend of red, green, and blue rather than pure shades.

If you want to see the full effect of blending red, green, and blue, you should use custom colors like these three:

```

.fill(Color(red: 1, green: 0, blue: 0))
.fill(Color(red: 0, green: 1, blue: 0))
.fill(Color(red: 0, green: 0, blue: 1))

```

There are a host of other real-time effects we can apply, and we already looked at **blur()** back in project 3. So, let's look at just one more before we move on: **saturation()**, which adjusts how much color is used inside a view. Give this a value between 0 (no color, just grayscale)

Project 9: Drawing

and 1 (full color).

We could write a little code to demonstrate both **blur()** and **saturation()** in the same view, like this:

```
Image("PaulHudson")
    .resizable()
    .scaledToFit()
    .frame(width: 200, height: 200)
    .saturation(amount)
    .blur(radius: (1 - amount) * 20)
```

With that code, having the slider at 0 means the image is blurred and colorless, but as you move the slider to the right it gains color and becomes sharp – all rendered at lightning-fast speed.

Animating simple shapes with animatableData

We've now covered a variety of drawing-related tasks, and back in project 6 we looked at animation, so now I want to look at putting those two things together.

First, let's build a custom shape we can use for an example – here's the code for a trapezoid shape, which is a four-sided shape with straight sides where one pair of opposite sides are parallel:

```
struct Trapezoid: Shape {
    var insetAmount: Double

    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: 0, y: rect.maxY))
        path.addLine(to: CGPoint(x: insetAmount, y: rect.minY))
        path.addLine(to: CGPoint(x: rect.maxX - insetAmount, y:
rect.minY))
        path.addLine(to: CGPoint(x: rect.maxX, y: rect.maxY))
        path.addLine(to: CGPoint(x: 0, y: rect.maxY))

        return path
    }
}
```

We can now use that inside a view, passing in some local state for its inset amount so we can modify the value at runtime:

```
struct ContentView: View {
    @State private var insetAmount = 50.0
```

Project 9: Drawing

```
var body: some View {
    Trapezoid(insetAmount: insetAmount)
        .frame(width: 200, height: 100)
        .onTapGesture {
            insetAmount = Double.random(in: 10...90)
        }
}
```

Every time you tap the trapezoid, **insetAmount** gets set to a new value, causing the shape to be redrawn.

Wouldn't it be nice if we could animate the change in inset? Sure it would – try changing the **onTapGesture()** closure to this:

```
.onTapGesture {
    withAnimation {
        insetAmount = Double.random(in: 10...90)
    }
}
```

Now run it again, and... nothing has changed. We've asked for animation, but we aren't getting animation – what gives?

When looking at animations previously, I asked you to add a call to **print()** inside the **body** property, then said this:

"What you should see is that it prints out 2.0, 3.0, 4.0, and so on. At the same time, the button is scaling up or down smoothly – it doesn't just jump straight to scale 2, 3, and 4. What's actually happening here is that SwiftUI is examining the state of our view before the binding changes, examining the target state of our views after the binding changes, then applying an animation to get from point A to point B."

So, as soon as **insetAmount** is set to a new random value, it will immediately jump to that value and pass it directly into **Trapezoid** – it won’t pass in lots of intermediate values as the animation happens. This is why our trapezoid jumps from inset to inset; it has no idea an animation is even happening.

We can fix this in only four lines of code, one of which is just a closing brace. However, even though this code is simple, *the way it works* might bend your brain.

First, the code – add this new computed property to the **Trapezoid** struct now:

```
var animatableData: Double {
    get { insetAmount }
    set { insetAmount = newValue }
}
```

You can now run the app again and see our trapezoid changing shape with a smooth animation.

What’s happening here is quite complex: when we use **withAnimation()**, SwiftUI immediately changes our state property to its new value, but behind the scenes it’s also keeping track of the changing value over time as part of the animation. As the animation progresses, SwiftUI will set the **animatableData** property of our shape to the latest value, and it’s down to us to decide what that means – in our case we assign it directly to **insetAmount**, because that’s the thing we want to animate.

Remember, SwiftUI evaluates our view state before an animation was applied and then again after. It can see we originally had code that evaluated to **Trapezoid(insetAmount: 50)**, but then after a random number was chosen we ended up with (for example) **Trapezoid(insetAmount: 62)**. So, it will interpolate between 50 and 62 over the length of our animation, each time setting the **animatableData** property of our shape to be that latest interpolated value – 51, 52, 53, and so on, until 62 is reached.

Animating complex shapes with AnimatablePair

SwiftUI uses an **animatableData** property to let us animate changes to shapes, but what happens when we want two, three, four, or more properties to animate? **animatableData** is a property, which means it must always be one value, however we get to decide *what* type of value it is: it might be a single **Double**, or it might be two values contained in a special wrapper called **AnimatablePair**.

To try this out, let's look at a new shape called **Checkerboard**, which must be created with some number of rows and columns:

```
struct Checkerboard: Shape {
    var rows: Int
    var columns: Int

    func path(in rect: CGRect) -> Path {
        var path = Path()

        // figure out how big each row/column needs to be
        let rowSize = rect.height / Double(rows)
        let columnSize = rect.width / Double(columns)

        // loop over all rows and columns, making alternating
        squares colored
        for row in 0..
```

```

        let rect = CGRect(x: startX, y: startY, width:
columnSize, height: rowSize)
        path.addRect(rect)
    }
}

return path
}
}
}

```

We can now create a 4x4 checkerboard in a SwiftUI view, using some state properties that we can change using a tap gesture:

```

struct ContentView: View {
    @State private var rows = 4
    @State private var columns = 4

    var body: some View {
        Checkerboard(rows: rows, columns: columns)
            .onTapGesture {
                withAnimation(.linear(duration: 3)) {
                    rows = 8
                    columns = 16
                }
            }
    }
}

```

When that runs you should be able to tap on the black squares to see the checkerboard jump from being 4x4 to 8x16, without animation even though the change is inside a

Project 9: Drawing

withAnimation() block.

As with simpler shapes, the solution here is to implement an **animatableData** property that will be set with intermediate values as the animation progresses. Here, though, there are two catches:

1. We have two properties that we want to animate, not one.
2. Our **row** and **column** properties are integers, and SwiftUI can't interpolate integers.

To resolve the first problem we're going to use a new type called **AnimatablePair**. As its name suggests, this contains a pair of animatable values, and because both its values can be animated the **AnimatablePair** can itself be animated. We can read individual values from the pair using **.first** and **.second**.

To resolve the *second* problem we're just going to do some type conversion: we can convert a **Double** to an **Int** just by using **Int(someDouble)**, and go the other way by using **Double(someInt)**.

So, to make our checkerboard animate changes in the number of rows and columns, add this property:

```
var animatableData: AnimatablePair<Double, Double> {
    get {
        AnimatablePair(Double(rows), Double(columns))
    }

    set {
        rows = Int(newValue.first)
        columns = Int(newValue.second)
    }
}
```

Now when you run the app you should find the change happens smoothly – or as smoothly as

Animating complex shapes with AnimatablePair

you would expect given that we're rounding numbers to integers.

Of course, the next question is: how do we animate three properties? Or four?

To answer that, let me show you the **animatableData** property for SwiftUI's **EdgeInsets** type:

```
AnimatablePair<CGFloat, AnimatablePair<CGFloat,  
AnimatablePair<CGFloat, CGFloat>>>
```

Yes, they use three separate animatable pairs, then just dig through them using code such as **newValue.second.second.first**.

I'm not going to claim this is the most elegant of solutions, but I hope you can understand why it exists: because SwiftUI can read and write the animatable data for a shape regardless of what that data is or what it means, it doesn't need to re-invoke the **body** property of our views 60 or even 120 times a second during an animation – it just changes the parts that actually are changing.

Creating a spirograph with SwiftUI

To finish off with something that really goes to town with drawing, I'm going to walk you through creating a simple spirograph with SwiftUI. "Spirograph" is the trademarked name for a toy where you place a pencil inside a circle and spin it around the circumference of another circle, creating various geometric patterns that are known as *roulettes* – like the casino game.

This code involves a very specific equation. I'm going to explain it, but it's totally OK to skip this chapter if you're not interested – this is just for fun, and no new Swift or SwiftUI is covered here.

Our algorithm has four inputs:

- The radius of the inner circle.
- The radius of the outer circle.
- The distance of the virtual pen from the center of the outer circle.
- What amount of the roulette to draw. This is optional, but I think it really helps show what's happening as the algorithm works.

So, let's start with that:

```
struct Spirograph: Shape {  
    let innerRadius: Int  
    let outerRadius: Int  
    let distance: Int  
    let amount: Double  
}
```

We then prepare three values from that data, starting with the greatest common divisor (GCD) of the inner radius and outer radius. Calculating the GCD of two numbers is usually done with Euclid's algorithm, which in a slightly simplified form looks like this:

```
func gcd(_ a: Int, _ b: Int) -> Int {
```

```

var a = a
var b = b

while b != 0 {
    let temp = b
    b = a % b
    a = temp
}

return a
}

```

Please add that method to the **Spirograph** struct.

The other two values are the difference between the inner radius and outer radius, and how many steps we need to perform to draw the roulette – this is 360 degrees multiplied by the outer radius divided by the greatest common divisor, multiplied by our amount input. All our inputs work best when provided as integers, but when it comes to *drawing* the roulette we need to use **Double**, so we’re also going to create **Double** copies of our inputs.

Add this **path(in:)** method to the **Spirograph** struct now:

```

func path(in rect: CGRect) -> Path {
    let divisor = gcd(innerRadius, outerRadius)
    let outerRadius = Double(self.outerRadius)
    let innerRadius = Double(self.innerRadius)
    let distance = Double(self.distance)
    let difference = innerRadius - outerRadius
    let endPoint = ceil(2 * Double.pi * outerRadius /
Double(divisor)) * amount

    // more code to come
}

```

Project 9: Drawing

Finally we can draw the roulette itself by looping from 0 to our end point, and placing points at precise X/Y coordinates. Calculating the X/Y coordinates for a given point in that loop (known as “theta”) is where the real mathematics comes in, but honestly I just converted the standard equation to Swift from Wikipedia – this is not something I would dream of memorizing!

- X is equal to the radius difference multiplied by the cosine of theta, added to the distance multiplied by the cosine of the radius difference divided by the outer radius multiplied by theta.
- Y is equal to the radius difference multiplied by the sine of theta, subtracting the distance multiplied by the sine of the radius difference divided by the outer radius multiplied by theta.

That’s the core algorithm, but we’re going to make two small changes: we’re going to add to X and Y half the width or height of our drawing rectangle respectively so that it’s centered in our drawing space, and if theta is 0 – i.e., if this is the first point in our roulette being drawn – we’ll call **move(to:)** rather than **addLine(to:)** for our path.

Here’s the final code for the **path(in:)** method – replace the **// more code to come** comment with this:

```
var path = Path()

for theta in stride(from: 0, through: endPoint, by: 0.01) {
    var x = difference * cos(theta) + distance * cos(difference / outerRadius * theta)
    var y = difference * sin(theta) - distance * sin(difference / outerRadius * theta)

    x += rect.width / 2
    y += rect.height / 2

    if theta == 0 {
```

```

        path.move(to: CGPoint(x: x, y: y))
    } else {
        path.addLine(to: CGPoint(x: x, y: y))
    }
}

return path

```

I realize that was a lot of heavy mathematics, but the pay off is about to come: we can now use that shape in a view, adding various sliders to control the inner radius, outer radius, distance, amount, and even color:

```

struct ContentView: View {
    @State private var innerRadius = 125.0
    @State private var outerRadius = 75.0
    @State private var distance = 25.0
    @State private var amount = 1.0
    @State private var hue = 0.6

    var body: some View {
        VStack(spacing: 0) {
            Spacer()

            Spirograph(innerRadius: Int(innerRadius), outerRadius:
Int(outerRadius), distance: Int(distance), amount: amount)
                .stroke(Color(hue: hue, saturation: 1, brightness: 1),
lineWidth: 1)
                .frame(width: 300, height: 300)

            Spacer()
        }

        Group {

```

Project 9: Drawing

```
Text("Inner radius: \(Int(innerRadius))")
Slider(value: $innerRadius, in: 10...150, step: 1)
.padding([.horizontal, .bottom])

Text("Outer radius: \(Int(outerRadius))")
Slider(value: $outerRadius, in: 10...150, step: 1)
.padding([.horizontal, .bottom])

Text("Distance: \(Int(distance))")
Slider(value: $distance, in: 1...150, step: 1)
.padding([.horizontal, .bottom])

Text("Amount: \(amount,
format: .number.precision(.fractionLength(2)))")
Slider(value: $amount)
.padding([.horizontal, .bottom])

Text("Color")
Slider(value: $hue)
.padding(.horizontal)
}

}
}

}
```

That was a lot of code, but I hope you take the time to run the app and appreciate just how beautiful roulettes are. What you're seeing is actually only one form of a roulette, known as a hypotrochoid – with small adjustments to the algorithm you can generate epitrochoids and more, which are beautiful in different ways.

Before I finish, I'd like to remind you that the parametric equations used here are mathematical standards rather than things I just invented – I literally went to Wikipedia's page on

Creating a spirograph with SwiftUI

hypotrochoids (<https://en.wikipedia.org/wiki/Hypotrochoid>) and converted them to Swift.

Drawing: Wrap up

We covered a *huge* amount of ground in this project, and you've learned about paths, shapes, strokes, transforms, drawing groups, animating values, and more. Not everyone will want to use all those features, and that's OK – hopefully you have a clear idea of which parts interested you the most, and have some good coding experience with them.

If you combine your new-found drawing abilities with all the animation functionality we covered back in project 6, I hope you're starting to realize just how much power and flexibility SwiftUI gives us. Yes, you can of course create whole apps using **List**, **NavigationView**, and similar, but you can also build completely custom user interfaces that look fantastic and are just as fast.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Create an **Arrow** shape – having it point straight up is fine. This could be a rectangle/triangle-style arrow, or perhaps three lines, or maybe something else depending on what kind of arrow you want to draw.
2. Make the line thickness of your **Arrow** shape animatable.
3. Create a **ColorCyclingRectangle** shape that is the rectangular cousin of **ColorCyclingCircle**, allowing us to control the position of the gradient using one or more properties.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Drawing. If you don't already subscribe, you can start a free trial today.

Tip: Gradient positions like `.top` and `.bottom` are actually instances of `UnitPoint`, and you can create your own `UnitPoint` instances with X/Y values ranging from 0 to 1.

Project 10

Cupcake Corner

Build an app that sends and receives JSON from the internet

Cupcake Corner: Introduction

In this project we're going to build a multi-screen app for ordering cupcakes. This will use a couple of forms, which are old news for you, but you're also going to learn how to make classes conform to **Codable** when they have **@Published** properties, how to send and receive the order data from the internet, how to validate forms, and more.

As we continue to dig deeper and deeper into **Codable**, I hope you'll continue to be impressed by how flexible and safe it is. In particular, I'd like you to keep in mind how very different it is from the much older **User Defaults** API – it's so nice not having to worry about typing strings exactly correctly!

Anyway, we have lots to get through so let's get started: create a new iOS app using the App template, and name it CupcakeCorner. If you haven't already downloaded the project files for this book, please fetch them now: <https://github.com/twostraws/HackingWithSwift>

As always we're going to start with the new techniques you'll need for the project...

Adding Codable conformance for @Published properties

If all the properties of a type already conform to **Codable**, then the type itself can conform to **Codable** with no extra work – Swift will synthesize the code required to archive and unarchive your type as needed. However, this *doesn't* work when we use property wrappers such as **@Published**, which means conforming to **Codable** requires some extra work on our behalf.

To fix this, we need to implement **Codable** conformance ourself. This will fix the **@Published** encoding problem, but is also a valuable skill to have elsewhere too because it lets us control exactly what data is saved and how it happens.

First let's create a simple type that recreates the problem. Add this class to `ContentView.swift`:

```
class User: ObservableObject, Codable {
    var name = "Paul Hudson"
}
```

That will compile just fine, because **String** conforms to **Codable** out of the box. However, if we make it **@Published** then the code no longer compiles:

```
class User: ObservableObject, Codable {
    @Published var name = "Paul Hudson"
}
```

The **@Published** property wrapper isn't magic – the name *property wrapper* comes from the fact that our **name** property is automatically wrapped inside another type that adds some additional functionality. In the case of **@Published** that's a struct called **Published** that can store any kind of value.

Previously we looked at how we can write generic methods that work with any kind of value, and the **Published** struct takes that a step further: the whole type itself is generic, meaning that you can't make an instance of **Published** all by itself, but instead make an instance of

Published<String> – a publishable object that contains a string.

If that sounds confusing, back up: it's actually a fairly fundamental principle of Swift, and one you've been working with for some time. Think about it – we can't say **var names: Set**, can we? Swift doesn't allow it; Swift wants to know what's *in* the set. This is because **Set** is also a generic type: you must make an instance of **Set<String>**. The same is also true of arrays and dictionaries: we always make them have something specific inside.

Swift already has rules in place that say if an array contains **Codable** types then the whole array is **Codable**, and the same for dictionaries and sets. However, SwiftUI *doesn't* provide the same functionality for its **Published** struct – it has no rule saying “if the published object is **Codable**, then the published struct itself is also **Codable**.”

As a result, we need to make the type conform ourselves: we need to tell Swift which properties should be loaded and saved, and how to do both of those actions.

None of those steps are terribly hard, so let's just dive in with the first one: telling Swift which properties should be loaded and saved. This is done using an enum that conforms to a special protocol called **CodingKey**, which means that every case in our enum is the name of a property we want to load and save. This enum is conventionally called **CodingKeys**, with an S on the end, but you can call it something else if you want.

So, our first step is to create a **CodingKeys** enum that conforms to **CodingKey**, listing all the properties we want to archive and unarchive. Add this inside the **User** class now:

```
enum CodingKeys: CodingKey {
    case name
}
```

The next task is to create a custom initializer that will be given some sort of container, and use that to read values for all our properties. This will involve learning a few new things, but let's look at the code first – add this initializer to **User** now:

```
required init(from decoder: Decoder) throws {
```

Project 10: Cupcake Corner

```
let container = try decoder.container(keyedBy:  
CodingKeys.self)  
name = try container.decode(String.self, forKey: .name)  
}
```

Even though that isn't much code, there are at least four new things in there.

First, this initializer is handed an instance of a new type called **Decoder**. This contains all our data, but it's down to us to figure out how to read it.

Second, anyone who subclasses our **User** class must override this initializer with a custom implementation to make sure they add their own values. We mark this using the **required** keyword: **required init**. An alternative is to mark this class as **final** so that subclassing isn't allowed, in which case we'd write **final class User** and drop the **required** keyword entirely.

Third, inside the method we ask our **Decoder** instance for a container matching all the coding keys we already set in our **CodingKey** struct by writing **decoder.container(keyedBy: CodingKeys.self)**. This means "this data should have a container where the keys match whatever cases we have in our **CodingKeys** enum. This is a throwing call, because it's possible those keys don't exist.

Finally, we can read values directly from that container by referencing cases in our enum – **container.decode(String.self, forKey: .name)**. This provides really strong safety in two ways: we're making it clear we expect to read a string, so if **name** gets changed to an integer the code will stop compiling; and we're also using a case in our **CodingKeys** enum rather than a string, so there's no chance of typos.

There's one more task we need to complete before the **User** class conforms to **Codable**: we've made an initializer so that Swift can *decode* data into this type, but now we need to tell Swift how to *encode* this type – how to archive it ready to write to JSON.

This step is pretty much the reverse of the initializer we just wrote: we get handed an **Encoder** instance to write to, ask it to make a container using our **CodingKeys** enum for keys, then write our values attached to each key.

Adding Codable conformance for @Published properties

Add this method to the **User** class now:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(name, forKey: .name)
}
```

And now our code compiles: Swift knows what data we want to write, knows how to convert some encoded data into our object's properties, and knows how to convert our object's properties into some encoded data.

I hope you're able to see some real advantages here compared to the stringly typed API of **UserDefaults** – it's much harder to make a mistake with **Codable** because we don't use strings, and it automatically checks our data types are correct.

Sending and receiving Codable data with URLSession and SwiftUI

iOS gives us built-in tools for sending and receiving data from the internet, and if we combine it with **Codable** support then it's possible to convert Swift objects to JSON for sending, then receive back JSON to be converted back to Swift objects. Even better, when the request completes we can immediately assign its data to properties in SwiftUI views, causing our user interface to update.

To demonstrate this we can load some example music JSON data from Apple's iTunes API, and show it all in a SwiftUI **List**. Apple's data includes lots of information, but we're going to whittle it down to just two types: a **Result** will store a track ID, its name, and the album it belongs to, and a **Response** will store an array of results.

So, start with this code:

```
struct Response: Codable {
    var results: [Result]
}

struct Result: Codable {
    var trackId: Int
    var trackName: String
    var collectionName: String
}
```

We can now write a simple **ContentView** that shows an array of results:

```
struct ContentView: View {
    @State private var results = [Result]()

    var body: some View {
```

Sending and receiving Codable data with URLSession and SwiftUI

```
List(results, id: \.trackId) { item in
    VStack(alignment: .leading) {
        Text(item.trackName)
            .font(.headline)
        Text(item.collectionName)
    }
}
```

That won't show anything at first, because the **results** array is empty. This is where our networking call comes in: we're going to ask the iTunes API to send us a list of all the songs by Taylor Swift, then use **JSONDecoder** to convert those results into an array of **Result** instances.

However, doing this means you need to meet two important Swift keywords: **async** and **await**. You see, any iPhone capable of running SwiftUI can perform billions of operations every second – it's so fast that it completes most work before we even realized it started it. On the flip side, networking – downloading data from the internet – might take several hundreds milliseconds or more to come, which is extremely slow for a computer that's used to doing literally a billion other things in that time.

Rather than forcing our entire progress to stop while the networking happens, Swift gives us the ability to say “this work will take some time, so please wait for it to complete while the rest of the app carries on running as usual.”

This functionality – this ability to leave some code running while our main app code carries on working – is called an *asynchronous* function. A synchronous function is one that runs fully before returning a value as needed, but an asynchronous function is one that is able to go to sleep for a while, so that it can wait for some other work to complete before continuing. In our case, that means going to sleep while our networking code happens, so that the rest of our app doesn't freeze up for several seconds.

Project 10: Cupcake Corner

To make this easier to understand, let's write it in a few stages. First, here's the basic method stub – please add this to the **ContentView** struct:

```
func loadData() async {  
}
```

Notice the new **async** keyword in there – we're telling Swift this function might want to go to sleep in order to complete its work.

We want that to be run as soon as our **List** is shown, but we can't just use **onAppear()** here because that doesn't know how to handle sleeping functions – it expects its function to be synchronous.

SwiftUI provides a different modifier for these kinds of tasks, helpfully called just **task()**. This *can* call functions that might go to sleep for a while; all Swift asks us to do is mark those functions with a second keyword, **await**, so we're explicitly acknowledging that a sleep might happen.

Add this modifier to the **List** now:

```
.task {  
    await loadData()  
}
```

Tip: Think of **await** as being like **try** – we're saying we understand a sleep might happen, in the same way **try** says we acknowledge an error might be thrown.

*Inside **loadData()*** we have three steps we need to complete:

1. Creating the URL we want to read.
2. Fetching the data for that URL.
3. Decoding the result of that data into a **Response** struct.

We'll add those step by step, starting with the URL. This needs to have a precise format: "itunes.apple.com" followed by a series of parameters – you can find the full set of parameters if you do a web search for "iTunes Search API". In our case we'll be using the search term "Taylor Swift" and the entity "song", so add this to **loadData()** now:

```
guard let url = URL(string: "https://itunes.apple.com/search?term=taylor+swift&entity=song") else {
    print("Invalid URL")
    return
}
```

Step 2 is to fetch the data from that URL, which is where our sleep is likely to happen. I say "likely" because it might not – iOS will do a little caching of data, so if the URL is fetched twice back to back then the data will get sent back immediately rather than triggering a sleep.

Regardless, a sleep *is* possible here, and every time a sleep is possible we need to use the **await** keyword with the code we want to run. Just as importantly, an error might also be thrown here – maybe the user isn't currently connected to the internet, for example.

So, we need to use both **try** *and* **await** at the same time. Please add this code directly after the previous code:

```
do {
    let (data, _) = try await URLSession.shared.data(from: url)

    // more code to come
} catch {
    print("Invalid data")
}
```

That introduced three important things, so let's break it down:

1. Our work is being done by the **data(from:)** method, which takes a URL and returns the

Project 10: Cupcake Corner

Data object at that URL. This method belongs to the **URLSession** class, which you can create and configure by hand if you want, but you can also use a shared instance that comes with sensible defaults.

2. The return value from **data(from:)** is a tuple containing the data at the URL and some metadata describing how the request went. We don't use the metadata, but we *do* want the URL's data, hence the underscore – we create a new local constant for the data, and toss the metadata away.
3. When using both **try** and **await** at the same time, we must write **try await** – using **await try** is not allowed. There's no special reason for this, but they had to pick one so they went with the one that reads more naturally.

So, if our download succeeds our **data** constant will be set to whatever data was sent back from the URL, but if it fails for any reason our code prints “Invalid data” and does nothing else.

The last part of this method is to convert the **Data** object into a **Response** object using **JSONDecoder**, then assign the array inside to our **results** property. This is exactly what we've used before, so this shouldn't be a surprise – add this last code in place of the **// more code to come** comment now:

```
if let decodedResponse = try?  
JSONDecoder().decode(Response.self, from: data) {  
    results = decodedResponse.results  
}
```

If you run the code now you should see a list of Taylor Swift songs appear after a short pause – it really isn't a lot of code given how well the end result works.

All this only handles *downloading* data. Later on in this project we're going to look at how to adopt a slightly different approach so you can *send* **Codable** data, but that's enough for now.

Loading an image from a remote server

SwiftUI's **Image** view works great with images in your app bundle, but if you want to load a *remote* image from the internet you need to use **AsyncImage** instead. These are created using an image URL rather than a simple asset name, but SwiftUI takes care of all the rest for us – it downloads the image, caches the download, and displays it automatically.

So, the simplest image we can create looks like this:

```
AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))
```

I created that picture to be 1200 pixels high, but when it displays you'll see it's much bigger. This gets straight to one of the fundamental complexities of using **AsyncImage**: SwiftUI knows nothing about the image until our code is run and the image is downloaded, and so it isn't able to size it appropriately ahead of time.

If I were to include that 1200px image in my project, I'd actually name it logo@3x.png, then also add an 800px image that was logo@2x.png. SwiftUI would then take care of loading the correct image for us, and making sure it appeared nice and sharp, and at the correct size too. As it is, SwiftUI loads that image as if it were designed to be shown at 1200 pixels high – it will be much bigger than our screen, and will look a bit blurry too.

To fix this, we can tell SwiftUI ahead of time that we're trying to load a 3x scale image, like this:

```
AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"),  
scale: 3)
```

When you run the code now you'll see the resulting image is a much more reasonable size.

And if you wanted to give it a *precise* size? Well, then you might start by trying this:

Project 10: Cupcake Corner

```
AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))
    .frame(width: 200, height: 200)
```

That won't work, but perhaps that won't even surprise you because it wouldn't work with a regular **Image** either. So you might try to make it resizable, like this:

```
AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))
    .resizable()
    .frame(width: 200, height: 200)
```

...except that won't work either, and in fact it's worse because now our code won't even compile. You see, the modifiers we're applying here don't apply directly to the image that SwiftUI downloads – they can't, because SwiftUI can't know how to apply them until it has actually fetched the image data.

Instead, we're applying modifiers to a wrapper *around* the image, which is the **AsyncImage** view. That will ultimately contain our finished image, but it will also contain a *placeholder* that gets used while the image is loading. You can actually see the placeholder just briefly when your app runs – that 200x200 gray square is it, and it will automatically go away once loading finishes.

To adjust our image, you need to use a more advanced form of **AsyncImage** that passes us the final image view once it's ready, which we can then customize as needed. As a bonus, this also gives us a second closure to customize the placeholder as needed.

For example, we could make the finished image view be both resizable and scaled to fit, and use **Color.red** as the placeholder so it's more obvious while you're learning.

```
AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))
{ image in
    image
        .resizable()
        .scaledToFit()
```

Loading an image from a remote server

```
    } placeholder: {
        Color.red
    }
    .frame(width: 200, height: 200)
```

A resizable image and **Color.red** both automatically take up all available space, which means the **frame()** modifier actually works now.

The placeholder view can be whatever you want. For example, if you replace **Color.red** with **ProgressView()** – just that – then you’ll get a little spinner activity indicator instead of a solid color.

If you want *complete* control over your remote image, there’s a third way of creating **AsyncImage** that tells us whether the image was loaded, hit an error, or hasn’t finished yet. This is particularly useful for times when you want to show a dedicated view when the download fails – if the URL doesn’t exist, or the user was offline, etc.

Here’s how that looks:

```
AsyncImage(url: URL(string: "https://hws.dev/img/bad.png"))
{ phase in
    if let image = phase.image {
        image
            .resizable()
            .scaledToFit()
    } else if phase.error != nil {
        Text("There was an error loading the image.")
    } else {
        ProgressView()
    }
}
.frame(width: 200, height: 200)
```

Project 10: Cupcake Corner

So, that will show our image if it can, an error message if the download failed for any reason, or a spinning activity indicator while the download is still in progress.

Validating and disabling forms

SwiftUI's **Form** view lets us store user input in a really fast and convenient way, but sometimes it's important to go a step further – to *check* that input to make sure it's valid before we proceed.

Well, we have a modifier just for that purpose: **disabled()**. This takes a condition to check, and if the condition is true then whatever it's attached to won't respond to user input – buttons can't be tapped, sliders can't be dragged, and so on. You can use simple properties here, but any condition will do: reading a computed property, calling a method, and so on,

To demonstrate this, here's a form that accepts a username and email address:

```
struct ContentView: View {
    @State private var username = ""
    @State private var email = ""

    var body: some View {
        Form {
            Section {
                TextField("Username", text: $username)
                TextField("Email", text: $email)
            }

            Section {
                Button("Create account") {
                    print("Creating account...")
                }
            }
        }
    }
}
```

Project 10: Cupcake Corner

In this example, we don't want users to create an account unless both fields have been filled in, so we can disable the form section containing the Create Account button by adding the `disabled()` modifier like this:

```
Section {  
    Button("Create account") {  
        print("Creating account...")  
    }  
}  
.disabled(username.isEmpty || email.isEmpty)
```

That means “this section is disabled if username is empty or email is empty,” which is exactly what we want.

You might find that it's worth spinning out your conditions into a separate computed property, such as this:

```
var disableForm: Bool {  
    username.count < 5 || email.count < 5  
}
```

Now you can just reference that in your modifier:

```
.disabled(disableForm)
```

Regardless of how you do it, I hope you try running the app and seeing how SwiftUI handles a disabled button – when our test fails the button's text goes gray, but as soon as the test passes the button lights up blue.

That brings us to the end of the overview for this project, so please put `ContentView.swift` back to its original state so we can begin building the main project.

Taking basic order details

The first step in this project will be to create an ordering screen that takes the basic details of an order: how many cupcakes they want, what kind they want, and whether there are any special customizations.

Before we get into the UI, we need to start by defining the data model. Previously we've used **@State** for simple value types and **@StateObject** for reference types, and we've looked at how it's possible to have an **ObservableObject** class containing structs inside it so that we get the benefits of both.

Here we're going to take a different solution: we're going to have a single class that stores all our data, which will be passed from screen to screen. This means all screens in our app share the same data, which will work really well as you'll see.

For now this class won't need many properties:

- The type of cakes, plus a static array of all possible options.
- How many cakes the user wants to order.
- Whether the user wants to make special requests, which will show or hide extra options in our UI.
- Whether the user wants extra frosting on their cakes.
- Whether the user wants to add sprinkles on their cakes.

Each of those need to update the UI when changed, which means we need to mark them with **@Published** and make the whole class conform to **ObservableObject**.

So, please make a new Swift file called Order.swift, change its Foundation import for SwiftUI, and give it this code:

```
class Order: ObservableObject {  
    static let types = [ "Vanilla", "Strawberry", "Chocolate",  
    "Rainbow" ]
```

Project 10: Cupcake Corner

```
@Published var type = 0
@Published var quantity = 3

@Published var specialRequestEnabled = false
@Published var extraFrosting = false
@Published var addSprinkles = false
}
```

We can now create a single instance of that inside **ContentView** by adding this property:

```
@StateObject var order = Order()
```

That's the only place the order will be created – every other screen in our app will be passed that property so they all work with the same data.

We're going to build the UI for this screen in three sections, starting with cupcake type and quantity. This first section will show a picker letting users choose from Vanilla, Strawberry, Chocolate and Rainbow cakes, then a stepper with the range 3 through 20 to choose the amount. All that will be wrapped inside a form, which is itself inside a navigation view so we can set a title.

There's a small speed bump here: our cupcake topping list is an array of strings, but we're storing the user's selection as an integer – how can we match the two? One easy solution is to use the **indices** property of the array, which gives us a position of each item that we can then use with as an array index. This is a bad idea for mutable arrays because the order of your array can change at any time, but here our array order won't ever change so it's safe.

Put this into the body of **ContentView** now:

```
NavigationView {
    Form {
        Section {
            Picker("Select your cake type", selection: $order.type) {
```

```

        ForEach(Order.types.indices) {
            Text(Order.types[$0])
        }
    }

    Stepper("Number of cakes: \($order.quantity)", value:
$order.quantity, in: 3...20)
}
}

.navigationTitle("Cupcake Corner")
}

```

The second section of our form will hold three toggle switches bound to **specialRequestEnabled**, **extraFrosting**, and **addSprinkles** respectively. However, the second and third switches should only be visible when the first one is enabled, so we'll wrap them in a condition.

Add this second section now:

```

Section {
    Toggle("Any special requests?", isOn:
$order.specialRequestEnabled.animation())

    if order.specialRequestEnabled {
        Toggle("Add extra frosting", isOn: $order.extraFrosting)

        Toggle("Add extra sprinkles", isOn: $order.addSprinkles)
    }
}

```

Go ahead and run the app again, and try it out – notice how I bound the first toggle with an **animation()** modifier attached, so that the second and third toggles slide in and out smoothly.

Project 10: Cupcake Corner

However, there's another bug, and this time it's one of our own making: if we enable special requests then enable one or both of "extra frosting" and "extra sprinkles", then *disable* the special requests, our previous special request selection stays active. This means if we re-enable special requests, the previous special requests are still active.

This kind of problem isn't hard to work around if every layer of your code is aware of it – if the app, your server, your database, and so on are all programmed to ignore the values of **extraFrosting** and **addSprinkles** when **specialRequestEnabled** is set to false. However, a better idea – a *safer* idea – is to make sure that both **extraFrosting** and **addSprinkles** are reset to false when **specialRequestEnabled** is set to false.

We can make this happen by adding a **didSet** property observer to **specialRequestEnabled**. Add this now:

```
@Published var specialRequestEnabled = false {
    didSet {
        if specialRequestEnabled == false {
            extraFrosting = false
            addSprinkles = false
        }
    }
}
```

Our third section is the easiest, because it's just going to be a **NavLink** pointing to the next screen. We don't have a second screen, but we can add it quickly enough: create a new SwiftUI view called "AddressView", and give it an **order** observed object property like this:

```
struct AddressView: View {
    @ObservedObject var order: Order

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

```
}
```



```
struct AddressView_Previews: PreviewProvider {
    static var previews: some View {
        AddressView(order: Order())
    }
}
```

We'll make that more useful shortly, but for now it means we can return to ContentView.swift and add the final section for our form. This will create a **NavLink** that points to an **AddressView**, passing in the current order object.

Please add this final section now:

```
Section {
    NavLink {
        AddressView(order: order)
    } label: {
        Text("Delivery details")
    }
}
```

That completes our first screen, so give it a try one last time before we move on – you should be able to select your cake type, choose a quantity, and toggle all the switches just fine.

Checking for a valid address

The second step in our project will be to let the user enter their address into a form, but as part of that we're going to add some validation – we only want to proceed to the third step if their address looks good.

We can accomplish this by adding a **Form** view to the **AddressView** struct we made previously, which will contain four text fields: name, street address, city, and zip. We can then add a **NavLink** to move to the next screen, which is where the user will see their final price and can check out.

To make this easier to follow, we're going to start by adding a new view called **CheckoutView**, which is where this address view will push to once the user is ready. This just avoids us having to put a placeholder in now then remember to come back later.

So, create a new SwiftUI view called **CheckoutView** and give it the same **Order** observed object property and preview that **AddressView** has:

```
struct CheckoutView: View {
    @ObservedObject var order: Order

    var body: some View {
        Text("Hello, World!")
    }
}

struct CheckoutView_Previews: PreviewProvider {
    static var previews: some View {
        CheckoutView(order: Order())
    }
}
```

Again, we'll come back to that later, but first let's implement **AddressView**. Like I said, this

needs to have a form with four text fields bound to four properties from our **Order** object, plus a **NavLink** passing control off to our check out view.

First, we need four new **@Published** properties in **Order** to store delivery details:

```
@Published var name = ""
@Published var streetAddress = ""
@Published var city = ""
@Published var zip = ""
```

Now replace the existing **body** of **AddressView** with this:

```
Form {
    Section {
        TextField("Name", text: $order.name)
        TextField("Street Address", text: $order.streetAddress)
        TextField("City", text: $order.city)
        TextField("Zip", text: $order.zip)
    }

    Section {
        NavigationLink {
            CheckoutView(order: order)
        } label: {
            Text("Check out")
        }
    }
}

.navigationTitle("Delivery details")
.navigationBarTitleDisplayMode(.inline)
```

As you can see, that passes our **order** object on one level deeper, to **CheckoutView**, which means we now have three views pointing to the same data.

Project 10: Cupcake Corner

Go ahead and run the app again, because I want you to see why all this matters. Enter some data on the first screen, enter some data on the second screen, then try navigating back to the beginning then forward to the end – that is, go back to the first screen, then click the bottom button twice to get to the checkout view again.

What you should see is that all the data you entered stays saved no matter what screen you’re on. Yes, this is the natural side effect of using a class for our data, but it’s an instant feature in our app without having to do any work – if we had used a struct, then any address details we had entered would disappear if we moved back to the original view. If you really wanted to use a struct for your data, you should follow the same struct inside class approach we used back in project 7; it’s certainly worth keeping it in mind when you evaluate your options.

Now that **AddressView** works, it’s time to stop the user progressing to the checkout unless some condition is satisfied. What condition? Well, that’s down to us to decide. Although we *could* write length checks for each of our four text fields, this often trips people up – some names are only four or five letters, so if you try to add length validation you might accidentally exclude people.

So, instead we’re just going to check that the **name**, **streetAddress**, **city**, and **zip** properties of our order aren’t empty. I prefer adding this kind of complex check inside my data, which means you need to add a new computed property to **Order** like this one:

```
var hasValidAddress: Bool {
    if name.isEmpty || streetAddress.isEmpty || city.isEmpty ||
zip.isEmpty {
        return false
    }

    return true
}
```

We can now use that condition in conjunction with SwiftUI’s **disabled()** modifier – attach that to any view along with a condition to check, and the view will stop responding to user

interaction if the condition is true.

In our case, the condition we want to check is the computed property we just wrote, **isValidAddress**. If that is false, then the form section containing our **NavLink** ought to be disabled, because we need users to fill in their delivery details first.

So, add this modifier to the end of the second section in **AddressView**:

```
.disabled(order.isValidAddress == false)
```

The code should look like this:

```
Section {  
    NavigationLink {  
        CheckoutView(order: order)  
    } label: {  
        Text("Check out")  
    }  
}  
.disabled(order.isValidAddress == false)
```

Now if you run the app you'll see that all four address fields must contain at least one character in order to continue. Even better, SwiftUI automatically grays out the button when the condition isn't true, giving the user really clear feedback when it is and isn't interactive.

Preparing for checkout

The final screen in our app is **CheckoutView**, and it's really a tale of two halves: the first half is the basic user interface, which should provide little real challenge for you; but the second half is all new: we need to encode our **Order** class to JSON, send it over the internet, and get a response.

We're going to look at the whole encoding and transferring chunk of work soon enough, but first let's tackle the easy part: giving **CheckoutView** a user interface. More specifically, we're going to create a **ScrollView** with an image, the total price of their order, and a Place Order button to kick off the networking.

For the image, I've uploaded a cupcake image to my server that we'll load remotely with **AsyncImage** – we *could* store it in the app, but having a remote image means we can dynamically switch it out for seasonal alternatives and promotions.

As for the order cost, we don't actually have any pricing for our cupcakes in our data, so we can just invent one – it's not like we're actually going to be charging people here. The pricing we're going to use is as follows:

- There's a base cost of \$2 per cupcake.
- We'll add a little to the cost for more complicated cakes.
- Extra frosting will cost \$1 per cake.
- Adding sprinkles will be another 50 cents per cake.

We can wrap all that logic up in a new computed property for **Order**, like this:

```
var cost: Double {  
    // $2 per cake  
    var cost = Double(quantity) * 2  
  
    // complicated cakes cost more  
    cost += (Double(type) / 2)
```

```

// $1/cake for extra frosting
if extraFrosting {
    cost += Double(quantity)
}

// $0.50/cake for sprinkles
if addSprinkles {
    cost += Double(quantity) / 2
}

return cost
}

```

The actual view itself is straightforward: we'll use a **VStack** inside a vertical **ScrollView**, then our image, the cost text, and button to place the order.

We'll be filling in the button's action in a minute, but first let's get the basic layout done – replace the existing **body** of **CheckoutView** with this:

```

ScrollView {
    VStack {
        AsyncImage(url: URL(string: "https://hws.dev/img/
cupcakes@3x.jpg"), scale: 3) { image in
            image
                .resizable()
                .scaledToFit()
        } placeholder: {
            ProgressView()
        }
        .frame(height: 233)

        Text("Your total is \(order.cost, format: .currency(code:

```

Project 10: Cupcake Corner

```
"USD" ) )")
    .font(.title)

Button("Place Order", action: { })
    .padding()
}

.navigationTitle("Check out")
.navigationBarTitleDisplayMode(.inline)
```

That should all be old news for you by now. But the tricky part comes next...

Encoding an ObservableObject class

We've organized our code so that we have one **Order** object that gets shared between all our screens, which has the advantage that we can move back and forward between those screens without losing data. However, this approach comes with a cost: we've had to use the **@Published** property wrapper for the properties in the class, and as soon we did that we lost support for automatic **Codable** conformance.

If you don't believe me, just try modifying the definition of **Order** to include **Codable**, like this:

```
class Order: ObservableObject, Codable {
```

The build will now fail, because Swift doesn't understand how to encode and decode published properties. This is a problem, because we want to submit the user's order to an internet server, which means we need it as JSON – we *need* the **Codable** protocol to work.

The fix here is to add **Codable** conformance by hand, which means telling Swift what should be encoded, how it should be encoded, and also how it should be *decoded* – converted back from JSON to Swift data.

That first step means adding an enum that conforms to **CodingKey**, listing all the properties we want to save. In our **Order** class that's almost everything – the only thing we don't need is the static **types** property.

So, add this enum to **Order** now:

```
enum CodingKeys: CodingKey {
    case type, quantity, extraFrosting, addSprinkles, name,
    streetAddress, city, zip
}
```

Project 10: Cupcake Corner

The second step requires us to write an `encode(to:)` method that creates a container using the coding keys enum we just created, then writes out all the properties attached to their respective key. This is just a matter of calling `encode(_:forKey:)` repeatedly, each time passing in a different property and coding key.

Add this method to **Order** now:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)

    try container.encode(type, forKey: .type)
    try container.encode(quantity, forKey: .quantity)

    try container.encode(extraFrosting, forKey: .extraFrosting)
    try container.encode(addSprinkles, forKey: .addSprinkles)

    try container.encode(name, forKey: .name)
    try container.encode(streetAddress, forKey: .streetAddress)
    try container.encode(city, forKey: .city)
    try container.encode(zip, forKey: .zip)
}
```

Because that method is marked with `throws`, we don't need to worry about catching any of the errors that are thrown inside – we can just use `try` without adding `catch`, knowing that any problems will automatically propagate upwards and be handled elsewhere.

Our final step is to implement a required initializer to decode an instance of **Order** from some archived data. This is pretty much the reverse of encoding, and even benefits from the same `throws` functionality:

```
required init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy:
CodingKeys.self)
```

```

type = try container.decode(Int.self, forKey: .type)
quantity = try container.decode(Int.self, forKey: .quantity)

extraFrosting = try container.decode(Bool.self,
forKey: .extraFrosting)
addSprinkles = try container.decode(Bool.self,
forKey: .addSprinkles)

name = try container.decode(String.self, forKey: .name)
streetAddress = try container.decode(String.self,
forKey: .streetAddress)
city = try container.decode(String.self, forKey: .city)
zip = try container.decode(String.self, forKey: .zip)
}

```

It's worth adding here that you can encode your data in any order you want – you don't need to match the order in which properties are declared in your object.

That makes our code fully **Codable** compliant: we've effectively bypassed the **@Published** property wrapper, reading and writing the values directly. However, it doesn't make our code compile – in fact, we now get a completely different error back in `ContentView.swift`.

The problem *now* is that we just created a custom initializer for our **Order** class, **init(from:)**, and Swift wants us to use it everywhere – even in places where we just want to create a new empty order because the app just started.

Fortunately, Swift lets us add multiple initializers to a class, so that we can create it in any number of different ways. In this situation, that means we need to write a new initializer that can create an order without any data whatsoever – it will rely entirely on the default property values we assigned.

So, add this new initializer to **Order** now:

Project 10: Cupcake Corner

```
init() { }
```

Now our code is back to compiling, and our **Codable** conformance is complete. This means we're ready for the final step: sending and receiving **Order** objects over the network.

Sending and receiving orders over the internet

iOS comes with some fantastic functionality for handling networking, and in particular the **URLSession** class makes it surprisingly easy to send and receive data. If we combine that with **Codable** to convert Swift objects to and from JSON, we can use a new **URLRequest** struct to configure exactly how data should be sent, accomplishing great things in about 20 lines of code.

First, let's create a method we can call from our Place Order button – add this to **CheckoutView**:

```
func placeOrder() async {  
}
```

Just like when we were downloading data using **URLSession**, uploading is also done asynchronously.

Now modify the Place Order button to this:

```
Button("Place Order") {  
    placeOrder()  
}  
.padding()
```

That code won't work, and Swift will be fairly clear why: it calls an asynchronous function from a function that does not support concurrency. What it means is that our button expects to be able to run its action immediately, and doesn't understand how to wait for something – even if wrote **await placeOrder()** it still wouldn't work, because the button doesn't want to wait.

Previously I mentioned that **onAppear()** didn't work with these asynchronous functions, and we needed to use the **task()** modifier instead. That isn't an option here because we're executing an action rather than just attaching modifiers, but Swift provides an alternative: we

Project 10: Cupcake Corner

can create a new task out of thin air, and just like the **task()** modifier this will run any kind of asynchronous code we want.

In fact, all it takes is placing our **await** call inside a task, like this:

```
Button("Place Order") {
    Task {
        await placeOrder()
    }
}
```

And now we're all set – that code will call **placeOrder()** asynchronously just fine. Of course, that function doesn't actually do anything just yet, so let's fix that now.

Inside **placeOrder()** we need to do three things:

1. Convert our current **order** object into some JSON data that can be sent.
2. Tell Swift how to send that data over a network call.
3. Run that request and process the response.

The first of those is straightforward, so let's get it out of the way. We've made the **Order** class conform to **Codable**, which means we can use **JSONEncoder** to archive it by adding this code to **placeOrder()**:

```
guard let encoded = try? JSONEncoder().encode(order) else {
    print("Failed to encode order")
    return
}
```

The second step means using a new type called **URLRequest**, which is like a **URL** except it gives us options to add extra information such as the type of request, user data, and more.

We need to attach the data in a very specific way so that the server can process it correctly, which means we need to provide two extra pieces of data beyond just our order:

1. The HTTP method of a request determines how data should be sent. There are several HTTP methods, but in practice only GET (“I want to read data”) and POST (“I want to write data”) are used much. We want to write data here, so we’ll be using POST.
2. The content type of a request determines what kind of data is being sent, which affects the way the server treats our data. This is specified in what’s called a MIME type, which was originally made for sending attachments in emails, and it has several thousand highly specific options.

So, the next code for **placeOrder()** will be to create a **URLRequest** object, then configure it to send JSON data using a HTTP POST request. We can then use that to upload our data using **URLSession**, and handle whatever comes back.

Of course, the *real* question is *where* to send our request, and I don’t think you really want to set up your own web server in order to follow this tutorial. So, instead we’re going to use a really helpful website called <https://reqres.in> – it lets us send any data we want, and will automatically send it back. This is a great way of prototyping network code, because you’ll get real data back from whatever you send.

Add this code to **placeOrder()** now:

```
let url = URL(string: "https://reqres.in/api/cupcakes")!
var request = URLRequest(url: url)
request.setValue("application/json", forHTTPHeaderField:
"Content-Type")
request.httpMethod = "POST"
```

That first line contains a force unwrap for the **URL(string:)** initializer, which means “this returns an optional URL, but please force it to be nonoptional.” Creating URLs from strings might fail because you inserted some gibberish, but here I hand-typed the URL so I can see it’s always going to be correct – there are no string interpolations in there that might cause problems.

At this point we’re all set to make our network request, which we’ll do using a new method

Project 10: Cupcake Corner

called **URLSession.shared.upload()** and the URL request we just made. So, go ahead and add this to **placeOrder()**:

```
do {
    let (data, _) = try await URLSession.shared.upload(for:
request, from: encoded)
    // handle the result
} catch {
    print("Checkout failed.")
}
```

Now for the important work: we need to read the result of our request for times when everything has worked correctly. If something went wrong – perhaps because there was no internet connection – then our **catch** block will be run, so we don't have to worry about that here.

Because we're using the ReqRes.in, we'll actually get back the same order we sent, which means we can use **JSONDecoder** to convert that back from JSON to an object.

To confirm everything worked correctly we're going to show an alert containing some details of our order, but we're going to use the *decoded* order we got back from ReqRes.in. Yes, this ought to be identical to the one we sent, so if it *isn't* it means we made a mistake in our coding.

Showing an alert requires properties to store the message and whether it's visible or not, so please add these two new properties to **CheckoutView** now:

```
@State private var confirmationMessage = ""
@State private var showingConfirmation = false
```

We also need to attach an **alert()** modifier to watch that Boolean, and show an alert as soon as its true. Add this modifier below the navigation title modifiers in **CheckoutView**:

```
.alert("Thank you!", isPresented: $showingConfirmation) {
    Button("OK") { }
```

```
} message: {
    Text(confirmationMessage)
}
```

And now we can finish off our networking code: we'll decode the data that came back, use it to set our confirmation message property, then set **showingConfirmation** to true so the alert appears. If the decoding fails – if the server sent back something that wasn't an order for some reason – we'll just print an error message.

Add this final code to **placeOrder()**, replacing the // **handle the result** comment:

```
let decodedOrder = try JSONDecoder().decode(Order.self, from:
data)
confirmationMessage = "Your order for \(decodedOrder.quantity)x
\(Order.types[decodedOrder.type].lowercased()) cupcakes is on
its way!"
showingConfirmation = true
```

With that final code in place our networking code is complete, and in fact our app is complete too. If you try running it now you should be able to select the exact cakes you want, enter your delivery information, then press Place Order to see an alert appear!

We're done! Well, I'm done – you still have some challenges to complete!

Cupcake Corner: Wrap up

Hopefully this project has shown you how to take the skills you know – SwiftUI's forms, pickers, steppers, and navigation – and build them into an app that sends all the user's data off to a server and processes the response.

You might not realize this yet, but the skills you learned in this project are the most important skills for the vast majority of iOS developers: take user data, send it to a server, and process the response probably accounts for half the non-trivial apps in existence. Yes, what data gets sent and how it's used to update the UI varies massively, but the concepts are identical.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Our address fields are currently considered valid if they contain anything, even if it's just only whitespace. Improve the validation to make sure a string of pure whitespace is invalid.
2. If our call to `placeOrder()` fails – for example if there is no internet connection – show an informative alert for the user. To test this, try commenting out the `request.httpMethod = "POST"` line in your code, which should force the request to fail.
3. For a more challenging task, see if you can convert our data model from a class to a struct, then create an **ObservableObject** class wrapper around it that gets passed around. This will result in your class having one **@Published** property, which is the data struct inside it, and should make supporting **Codable** on the struct much easier.

Cupcake Corner: Wrap up

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Cupcake Corner. If you don't already subscribe, you can start a free trial today.

Project 11

Bookworm

Use Core Data to build an app that tracks books you like

Bookworm: Introduction

In this project we're going to build an app to track which books you've read and what you thought of them, and it's going to follow a similar theme to project 10: let's take all the skills you've already mastered, then add some bonus new skills that take them all to a new level.

This time you're going to meet Core Data, which is Apple's battle-hardened framework for working with databases. This project will serve as an introduction for Core Data, but we'll be going into much more detail soon.

At the same time, we're also going to build our first custom user interface component – a star rating widget where the user can tap to leave a score for each book. This will mean introducing you to *another* property wrapper, called **@Binding** – trust me, it will all make sense.

As usual we're going to start with a walkthrough of all the new techniques you need for this project, so please create a new iOS app called Bookworm, using the App template.

Important: I know it's tempting, but please *don't* check the box marked **Use Core Data**. It adds a whole bunch of unhelpful code to your project, and you'll just need to delete it in order to follow along.

Creating a custom component with **@Binding**

You've already seen how SwiftUI's **@State** property wrapper lets us work with local value types, and how **@StateObject** lets us work with shareable reference types. Well, there's a third option, called **@Binding**, which lets us connect an **@State** property of one view to some underlying model data.

Think about it: when we create a toggle switch we send in some sort of Boolean property that can be changed, like this:

```
@State private var rememberMe = false

var body: some View {
    Toggle("Remember Me", isOn: $rememberMe)
}
```

So, the toggle needs to change our Boolean when the user interacts with it, but how does it remember what value it should change?

That's where **@Binding** comes in: it lets us store a mutable value in a view that actually points to some other value from elsewhere. In the case of **Toggle**, the switch changes its own local binding to a Boolean, but behind the scenes that's actually manipulating the **@State** property in our view.

This makes **@Binding** extremely important for whenever you want to create a custom user interface component. At their core, UI components are just SwiftUI views like everything else, but **@Binding** is what sets them apart: while they might have their local **@State** properties, they also expose **@Binding** properties that let them interface directly with other views.

To demonstrate this, we're going to look at the code it takes to create a custom button that stays down when pressed. Our basic implementation will all be stuff you've seen before: a button with some padding, a linear gradient for the background, a **Capsule** clip shape, and so

Creating a custom component with @Binding

on – add this to ContentView.swift now:

```
struct PushButton: View {
    let title: String
    @State var isOn: Bool

    var onColors = [Color.red, Color.yellow]
    var offColors = [Color(white: 0.6), Color(white: 0.4)]

    var body: some View {
        Button(title) {
            isOn.toggle()
        }
        .padding()
        .background(LinearGradient(gradient: Gradient(colors:
            isOn ? onColors : offColors), startPoint: .top,
            endPoint: .bottom))
        .foregroundColor(.white)
        .clipShape(Capsule())
        .shadow(radius: isOn ? 0 : 5)
    }
}
```

The only vaguely exciting thing in there is that I used properties for the two gradient colors so they can be customized by whatever creates the button.

We can now create one of those buttons as part of our main user interface, like this:

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

    var body: some View {
        VStack {
```

Project 11: Bookworm

```
    PushButton(title: "Remember Me", isOn: rememberMe)
    Text(rememberMe ? "On" : "Off")
}
}
}
```

That has a text view below the button so we can track the state of the button – try running your code and see how it works.

What you'll find is that tapping the button does indeed affect the way it appears, but our text view doesn't reflect that change – it always says "Off". Clearly *something* is changing because the button's appearance changes when it's pressed, but that change isn't being reflected in **ContentView**.

What's happening here is that we've defined a one-way flow of data: **ContentView** has its **rememberMe** Boolean, which gets used to create a **PushButton** – the button has an initial value provided by **ContentView**. However, once the button was created it takes over control of the value: it toggles the **isOn** property between true or false internally to the button, but doesn't pass that change back on to **ContentView**.

This is a problem, because we now have two sources of truth: **ContentView** is storing one value, and **PushButton** another. Fortunately, this is where **@Binding** comes in: it allows us to create a two-way connection between **PushButton** and whatever is using it, so that when one value changes the other does too.

To switch over to **@Binding** we need to make just two changes. First, in **PushButton** change its **isOn** property to this:

```
@Binding var isOn: Bool
```

And second, in **ContentView** change the way we create the button to this:

```
PushButton(title: "Remember Me", isOn: $rememberMe)
```

Creating a custom component with @Binding

That adds a dollar sign before **rememberMe** – we’re passing in the binding itself, not the Boolean inside it.

Now run the code again, and you’ll find that everything works as expected: toggling the button now correctly updates the text view as well.

This is the power of **@Binding**: as far as the button is concerned it’s just toggling a Boolean – it has no idea that something else is monitoring that Boolean and acting upon changes.

Accepting multi-line text input with TextEditor

We've used SwiftUI's **TextField** view several times already, and it's great for times when the user wants to enter short pieces of text. However, for *longer* pieces of text you should switch over to using a **TextEditor** view instead: it also expects to be given a two-way binding to a text string, but it has the additional benefit of allowing multiple lines of text – it's much better for accepting longer strings from the user.

Mostly because it has nothing special in the way of configuration options, using **TextEditor** is actually easier than using **TextField** – you can't adjust its style or add placeholder text, you just bind it to a string. However, you *do* need to be careful to make sure it doesn't go outside the safe area, otherwise typing will be tricky; embed it in a **NavigationView**, a **Form**, or similar.

For example, we could create the world's simplest notes app by combining **TextEditor** with **@AppStorage**, like this:

```
struct ContentView: View {
    @AppStorage("notes") private var notes = ""

    var body: some View {
        NavigationView {
            TextEditor(text: $notes)
                .navigationTitle("Notes")
                .padding()
        }
    }
}
```

Tip: **@AppStorage** is not designed to store secure information, so never use it for anything private.

Accepting multi-line text input with TextEditor

How to combine Core Data and SwiftUI

SwiftUI and Core Data were introduced almost exactly a decade apart – SwiftUI with iOS 13, and Core Data with iPhoneOS 3; so long ago it wasn't even called iOS because the iPad wasn't released yet. Despite their distance in time, Apple put in a ton of work to make sure these two powerhouse technologies work beautifully alongside each other, meaning that Core Data integrates into SwiftUI as if it were always designed that way.

First, the basics: Core Data is an object graph and persistence framework, which is a fancy way of saying it lets us define objects and properties of those objects, then lets us read and write them from permanent storage.

On the surface this sounds like using **Codable** and **UserDefault**s, but it's much more advanced than that: Core Data is capable of sorting and filtering of our data, and can work with much larger data – there's effectively no limit to how much data it can store. Even better, Core Data implements all sorts of more advanced functionality for when you really need to lean on it: data validation, lazy loading of data, undo and redo, and much more.

In this project we're going to be using only a small amount of Core Data's power, but that will expand soon enough – I just want to give you a taste of it at first.

When you created your Xcode project I asked you to *not* check the Use Core Data box, because although it gets some of the boring set up code out of the way it also adds a whole bunch of extra example code that is just pointless and just needs to be deleted.

So, instead you're going to learn how to set up Core Data by hand. It takes three steps, starting with us defining the data we want to use in our app.

Previously we described data like this:

```
struct Student {  
    var id: UUID
```

```
    var name: String  
}
```

However, Core Data doesn't work like that. You see, Core Data needs to know ahead of time what all our data types look like, what it contains, and how it relates to each other.

This is all contained in a new file type called Data Model, which has the file extension “xcdatamodeld”. Let's create one now: press Cmd+N to make a new file, select Data Model from the list of templates, then name your model Bookworm.xcdatamodeld.

When you press Create, Xcode will open the new file in its data model editor. Here we define our types as “entities”, then create properties in there as “attributes” – Core Data is responsible for converting that into an actual database layout it can work with at runtime.

For trial purposes, please press the Add Entity button to create a new entity, then double click on its name to rename it “Student”. Next, click the + button directly below the Attributes table to add two attributes: “id” as a UUID and “name” as a string.

That tells Core Data everything we need to know to create students and save them, so we can proceed to the second step of setting up Core Data: writing a little Swift code to load that model and prepare it for us to use.

We're going to write this in a few small pieces, so I can explain what's happening in detail. First, create a new Swift file called DataController.swift, and add this just above its **import SwiftUI** line:

```
import CoreData
```

We're going to start by creating a new class called **DataController**, making it conform to **ObservableObject** so that we can use it with the **@StateObject** property wrapper – we want to create one of these when our app launches, then keep it alive for as long as our app runs.

Inside this class we'll add a single property of the type **NSPersistentContainer**, which is the Core Data type responsible for loading a data model and giving us access to the data inside.

Project 11: Bookworm

From a modern point of view this sounds strange, but the “NS” part is short for “NeXTSTEP”, which was a huge operating system that Apple acquired when they brought Steve Jobs back into the fold in 1997 – Core Data has some really old foundations!

Anyway, start by adding this to your file:

```
class DataController: ObservableObject {
    let container = NSPersistentContainer(name: "Bookworm")
}
```

That tells Core Data we want to use the Bookworm data model. It doesn’t actually *load* it – we’ll do that in a moment – but it *does* prepare Core Data to load it. Data models don’t contain our actual data, just the definitions of properties and attributes like we defined a moment ago.

To actually load the data model we need to call **loadPersistentStores()** on our container, which tells Core Data to access our saved data according to the data model in Bookworm.xcdatamodeld. This doesn’t load all the data into memory at the same time, because that would be wasteful, but at least Core Data can see all the information we have.

It’s entirely possible that loading the saved data might go wrong, maybe the data is corrupt, for example. But honestly if it *does* go wrong there’s not a great deal you can do – the only meaningful thing you can do at this point is show an error message to the user, and hope that relaunching the app clears up the problem.

Anyway, we’re going to write a small initializer for **DataController** that loads our stored data immediately. If things go wrong – unlikely, but not impossible – we’ll print a message to the Xcode debug log.

Add this initializer to **DataController** now:

```
init() {
    container.loadPersistentStores { description, error in
        if let error = error {
            print("Core Data failed to load: \\"
```

```
(error.localizedDescription) " )  
    }  
}  
}
```

That completes **DataController**, so the final step is to create an instance of **DataController** and send it into SwiftUI's environment. You've already met **@Environment** when it came to asking SwiftUI to dismiss our view, but it also stores other useful data such as our time zone, user interface appearance, and more.

This is relevant to Core Data because most apps work with only one Core Data store at a time, so rather than every view trying to create their own store individually we instead create it once when our app starts, then store it inside the SwiftUI environment so everywhere else in our app can use it.

To do this, open BookworkApp.swift, and add this property this to the struct:

```
@StateObject private var dataController = DataController()
```

That creates our data controller, and now we can place it into SwiftUI's environment by adding a new modifier to the **ContentView()** line:

```
WindowGroup {  
    ContentView()  
        .environment(\.managedObjectContext,  
        dataController.container.viewContext)  
}
```

Tip: If you're using Xcode's SwiftUI previews, you should also inject a managed object context into your preview struct for **ContentView**.

You've already met data models, which store definitions of the entities and attributes we want to use, and **NSPersistentStoreContainer**, which handles loading the actual data we have

Project 11: Bookworm

saved to the user’s device. Well, you just met the third piece of the Core Data puzzle: managed object contexts. These are effectively the “live” version of your data – when you load objects and change them, those changes only exist in memory until you specifically save them back to the persistent store.

So, the job of the view context is to let us work with all our data in memory, which is much faster than constantly reading and writing data to disk. When we’re ready we still do need to write changes out to persistent store if we want them to be there when our app runs next, but you can also choose to discard changes if you don’t want them.

At this point we’ve created our Core Data model, we’ve loaded it, and we’ve prepared it for use with SwiftUI. There are still two important pieces of the puzzle left: reading data, and writing it too.

Retrieving information from Core Data is done using a *fetch request* – we describe what we want, how it should sorted, and whether any filters should be used, and Core Data sends back all the matching data. We need to make sure that this fetch request stays up to date over time, so that as students are created or removed our UI stays synchronized.

SwiftUI has a solution for this, and – you guessed it – it’s another property wrapper. This time it’s called **@FetchRequest** and it takes at least one parameter describing how we want the results to be sorted. It has quite a specific format, so let’s start by adding a fetch request for our students – please add this property to **ContentView** now:

```
@FetchRequest(sortDescriptors: [ ]) var students:  
FetchedResults<Student>
```

Broken down, that creates a fetch request with no sorting, and places it into a property called **students** that has the type **FetchedResults<Student>**.

From there, we can start using **students** like a regular Swift array, but there’s one catch as you’ll see. First, some code that puts the array into a **List**:

```
 VStack {
```

```

List(students) { student in
    Text(student.name ?? "Unknown")
}
}

```

Did you spot the catch? Yes, **student.name** is an optional – it might have a value or it might not. This is one area of Core Data that will annoy you greatly: it has the concept of optional data, but it's an entirely different concept to Swift's optionals. If we say to Core Data “this thing can't be optional” (which you can do inside the model editor), it will *still* generate optional Swift properties, because all Core Data cares about is that the properties have values when they are saved – they can be nil at other times.

You can run the code if you want to, but there isn't really much point – the list will be empty because we haven't added any data yet, so our database is empty. To fix that we're going to create a button below our list that adds a new random student every time it's tapped, but first we need a new property to access the managed object context we created earlier.

Let me back up a little, because this matters. When we defined the “Student” entity, what actually happened was that Core Data created a class for us that inherits from one of its own classes: **NSManagedObject**. We can't see this class in our code, because it's generated automatically when we build our project, just like Core ML's models. These objects are called *managed* because Core Data is looking after them: it loads them from the persistent container and writes their changes back too.

All our managed objects live inside a *managed object context*, one of which we created earlier. Placing it into the SwiftUI environment meant that it was automatically used for the **@FetchRequest** property wrapper – it uses whatever managed object context is available in the environment.

Anyway, when it comes to adding and saving objects, we need access to the managed object context that it is in SwiftUI's environment. This is another use for the **@Environment** property wrapper – we can ask it for the current managed object context, and assign it to a property for our use.

Project 11: Bookworm

So, add this property to **ContentView** now:

```
@Environment(\.managedObjectContext) var moc
```

With that in place, the next step is add a button that generates random students and saves them in the managed object context. To help the students stand out, we'll assign random names by creating **firstNames** and **lastNames** arrays, then using **randomElement()** to pick one of each.

Start by adding this button just below the **List**:

```
Button("Add") {
    let firstNames = ["Ginny", "Harry", "Hermione", "Luna",
    "Ron"]
    let lastNames = ["Granger", "Lovegood", "Potter", "Weasley"]

    let chosenFirstName = firstNames.randomElement()!
    let chosenLastName = lastNames.randomElement()!

    // more code to come
}
```

Note: Inevitably there are people that will complain about me force unwrapping those calls to **randomElement()**, but we literally just hand-created the arrays to have values – it will always succeed. If you desperately hate force unwraps, perhaps replace them with nil coalescing and default values.

Now for the interesting part: we're going to create a **Student** object, using the class Core Data generated for us. This needs to be attached to a managed object context, so the object knows where it should be stored. We can then assign values to it just like we normally would for a struct.

So, add these three lines to the button's action closure now:

```
let student = Student(context: moc)
```

```
student.id = UUID()
student.name = "\(chosenFirstName) \(chosenLastName)"
```

Finally we need to ask our managed object context to save itself, which means it will write its changes to the persistent store. This is a throwing function call, because in theory it might fail. In practice, nothing about what we've done has any chance of failing, so we can call this using `try?` – we don't care about catching errors.

So, add this final line to the button's action:

```
try? moc.save()
```

At last, you should now be able to run the app and try it out – click the Add button a few times to generate some random students, and you should see them slide somewhere into our list. Even better, if you relaunch the app you'll find your students are still there, because Core Data saved them.

Now, you might think this was an awful lot of learning for not a lot of result, but you now know what persistent stores and managed object contexts are, what entities and attributes are, what managed objects and fetch requests are, and you've seen how to save changes too. We'll be looking at Core Data more later on in this project, as well in the future, but for now you've come far.

This was the last part of the overview for this project, but this time I don't want you to reset your project fully. Yes, put ContentView.swift back to its original state, then delete the Student entity from Bookworm.xcdatamodeld, **but please leave BookwormApp.swift and DataController.swift alone** – we'll be using them in the real project!

Creating books with Core Data

Our first task in this project will be to design a Core Data model for our books, then creating a new view to add books to the database.

First, the model: open Bookworm.xcdatamodeld and add a new entity called “Book” – we’ll create one new object in there for each book the user has read. In terms of *what* constitutes a book, I’d like you to add the following attributes:

- id, UUID – a guaranteed unique identifier we can use to distinguish between books
- title, String – the title of the book
- author, String – the name of whoever wrote the book
- genre, String – one of several strings from the genres in our app
- review, String – a brief overview of what the user thought of the book
- rating, Integer 16 – the user’s rating for this book

Most of those should make sense, but the last one is an odd one: “integer 16”. What is the 16? And how come there are also Integer 32 and Integer 64? Well, just like **Float** and **Double** the difference is how much data they can store: Integer 16 uses 16 binary digits (“bits”) to store numbers, so it can hold values from -32,768 up to 32,767, whereas Integer 32 uses 32 bits to store numbers, so it holds values from -2,147,483,648 up to 2,147,483,647. As for Integer 64... well, that’s a really large number – about 9 quintillion.

The point is that these values aren’t interchangeable: you can’t take the value from a 64-bit number and try to store it in a 16-bit number, because you’d probably lose data. On the other hand, it’s a waste of space to use 64-bit integers for values we know will always be small. As a result, Core Data gives us the option to choose just how much storage we want.

Our next step is to write a form that can create new entries. This will combine so many of the skills you’ve learned so far: **Form**, **@State**, **@Environment**, **TextField**, **TextEditor**, **Picker**, **sheet()**, and more, plus all your new Core Data knowledge.

Start by creating a new SwiftUI view called “AddBookView”. In terms of properties, we need

an environment property to store our managed object context:

```
@Environment(\.managedObjectContext) var moc
```

As this form is going to store all the data required to make up a book, we need **@State** properties for each of the book's values except **id**, which we can generate dynamically. So, add these properties next:

```
@State private var title = ""
@State private var author = ""
@State private var rating = 3
@State private var genre = ""
@State private var review = ""
```

Finally, we need one more property to store all possible genre options, so we can make a picker using **ForEach**. Add this last property to **AddBookView** now:

```
let genres = ["Fantasy", "Horror", "Kids", "Mystery", "Poetry",
"Romance", "Thriller"]
```

We can now take a first pass at the form itself – we'll improve it soon, but this is enough for now. Replace the current **body** with this:

```
NavigationView {
    Form {
        Section {
            TextField("Name of book", text: $title)
            TextField("Author's name", text: $author)

            Picker("Genre", selection: $genre) {
                ForEach(genres, id: \.self) {
                    Text($0)
                }
            }
        }
    }
}
```

Project 11: Bookworm

```
        }
    }

Section {
    TextEditor(text: $review)

    Picker("Rating", selection: $rating) {
        ForEach(0..<6) {
            Text(String($0))
        }
    }
} header: {
    Text("Write a review")
}

Section {
    Button("Save") {
        // add the book
    }
}
.navigationBarTitle("Add Book")
}
```

When it comes to filling in the button's action, we're going to create an instance of the **Book** class using our managed object context, copy in all the values from our form (converting **rating** to an **Int16** to match Core Data), then save the managed object context.

Most of this work is just copying one value into another, with the only vaguely interesting thing being how we convert from an **Int** to an **Int16** for the rating. Even that is pretty guessable: **Int16(someInt)** does it all.

Add this code in place of the **// add the book** comment:

```

let newBook = Book(context: moc)
newBook.id = UUID()
newBook.title = title
newBook.author = author
newBook.rating = Int16(rating)
newBook.genre = genre
newBook.review = review

try? moc.save()

```

That completes the form for now, but we still need a way to show and hide it when books are being added.

Showing **AddBookView** involves returning to **ContentView.swift** and following the usual steps for a sheet:

1. Adding an **@State** property to track whether the sheet is showing.
2. Add some sort of button – in the toolbar, in this case – to toggle that property.
3. A **sheet()** modifier that shows **AddBookView** when the property becomes true.

Enough talk – let's start writing some more code. Please start by adding these three properties to **ContentView**:

```

@Environment(\.managedObjectContext) var moc
@FetchRequest(sortDescriptors: []) var books:
FetchedResults<Book>

@State private var showingAddScreen = false

```

That gives us a managed object context we can use later on to delete books, a fetch request reading all the books we have (so we can test everything worked), and a Boolean that tracks whether the add screen is showing or not.

Project 11: Bookworm

For the **ContentView** body, we're going to use a navigation view so we can add a title plus a button in its top-right corner, but otherwise it will just hold some text showing how many items we have in the **books** array – just so we can be sure everything is working. Remember, this is where we need to add our **sheet()** modifier to show an **AddBookView** as needed.

Replace the existing **body** property of **ContentView** with this:

```
NavigationView {  
    Text("Count: \(books.count)")  
    .navigationTitle("Bookworm")  
    .toolbar {  
        ToolbarItem(placement: .navigationBarTrailing) {  
            Button {  
                showingAddScreen.toggle()  
            } label: {  
                Label("Add Book", systemImage: "plus")  
            }  
        }  
    }  
    .sheet(isPresented: $showingAddScreen) {  
        AddBookView()  
    }  
}
```

Tip: That explicitly specifies a trailing navigation bar placement so that we can add a second button later.

Bear with me – we're almost done! We've now designed our Core Data model, created a form to add data, then updated **ContentView** so that it can present the form and pass in its managed object context. The final step is to make the form dismiss itself when the user adds a book.

We've done this before, so hopefully you know the drill. We need to start by adding another environment property to **AddBookView** to be able to dismiss the current view:

```
@Environment(\.dismiss) var dismiss
```

Finally, add a call to **dismiss()** to the end of your button's action closure.

You should be able to run the app now and add an example book just fine. When **AddBookView** slides away the count label should update itself to 1.

Adding a custom star rating component

SwiftUI makes it really easy to create custom UI components, because they are effectively just views that have some sort of **@Binding** exposed for us to read.

To demonstrate this, we're going to build a star rating view that lets the user enter scores between 1 and 5 by tapping images. Although we could just make this view simple enough to work for our exact use case, it's often better to add some flexibility where appropriate so it can be used elsewhere too. Here, that means we're going to make six customizable properties:

- What label should be placed before the rating (default: an empty string)
- The maximum integer rating (default: 5)
- The off and on images, which dictate the images to use when the star is highlighted or not (default: **nil** for the off image, and a filled star for the on image; if we find **nil** in the off image we'll use the on image there too)
- The off and on colors, which dictate the colors to use when the star is highlighted or not (default: gray for off, yellow for on)

We also need one extra property to store an **@Binding** integer, so we can report back the user's selection to whatever is using the star rating.

So, create a new SwiftUI view called “RatingView”, and start by giving it these properties:

```
@Binding var rating: Int  
  
var label = ""  
  
var maximumRating = 5  
  
var offImage: Image?  
var onImage = Image(systemName: "star.fill")
```

```
var offColor = Color.gray
var onColor = Color.yellow
```

Before we fill in the **body** property, please try building the code – you should find that it fails, because our **RatingView_Previews** struct doesn't pass in a binding to use for **rating**.

SwiftUI has a specific and simple solution for this called *constant bindings*. These are bindings that have fixed values, which on the one hand means they can't be changed in the UI, but also means we can create them trivially – they are perfect for previews.

So, replace the existing **previews** property with this:

```
static var previews: some View {
    RatingView(rating: .constant(4))
}
```

Now let's turn to the **body** property. This is going to be a **HStack** containing any label that was provided, plus as many stars as have been requested – although, of course, they can choose any image they want, so it might not be a star at all.

The logic for choosing which image to show is pretty simple, but it's perfect for carving off into its own method to reduce the complexity of our code. The logic is this:

- If the number that was passed in is greater than the current rating, return the off image if it was set, otherwise return the on image.
- If the number that was passed in is equal to or less than the current rating, return the on image.

We can encapsulate that in a single method, so add this to **RatingView** now:

```
func image(for number: Int) -> Image {
    if number > rating {
        return offImage ?? onImage
```

Project 11: Bookworm

```
    } else {
        return onImage
    }
}
```

And now implementing the **body** property is surprisingly easy: if the label has any text use it, then use **ForEach** to count from 1 to the maximum rating plus 1 and call **image(for:)** repeatedly. We'll also apply a foreground color depending on the rating, and add a tap gesture that adjusts the rating.

Replace your existing **body** property with this:

```
HStack {
    if label.isEmpty == false {
        Text(label)
    }

    ForEach(1..<maximumRating + 1, id: \.self) { number in
        image(for: number)
            .foregroundColor(number > rating ? offColor : onColor)
            .onTapGesture {
                rating = number
            }
    }
}
```

That completes our rating view already, so to put it into action go back to **AddBookView** and replace the second section with this:

```
Section {
    TextEditor(text: $review)
    RatingView(rating: $rating)
} header: {
```

Adding a custom star rating component

```
    Text("Write a review")  
}
```

That's all it takes – our default values are sensible, so it looks great out of the box. And the result is much nicer to use: there's no need to tap into a detail view with a picker here, because star ratings are more natural and more common.

Building a list with @FetchRequest

Right now our **ContentView** has a fetch request property like this:

```
@FetchRequest(sortDescriptors: [ ]) var books:  
FetchedResults<Book>
```

And we're using it in **body** with this simple text view:

```
Text("Count: \(books.count)")
```

To bring this screen to life, we're going to replace that text view with a **List** showing all the books that have been added, along with their rating and author.

We *could* just use the same star rating view here that we made earlier, but it's much more fun to try something else. Whereas the **RatingView** control can be used in any kind of project, we can make a new **EmojiRatingView** that displays a rating specific to this project. All it will do is show one of five different emoji depending on the rating, and it's a great example of how straightforward view composition is in SwiftUI – it's so easy to just pull out a small part of your views in this way.

So, make a new SwiftUI view called “EmojiRatingView”, and give it the following code:

```
struct EmojiRatingView: View {  
    let rating: Int16  
  
    var body: some View {  
        switch rating {  
        case 1:  
            Text("1")  
        case 2:  
            Text("2")  
        case 3:  
            Text("3")  
        case 4:  
            Text("4")  
        case 5:  
            Text("5")  
        default:  
            Text("")  
        }  
    }  
}
```

```

case 3:
    Text("3")
case 4:
    Text("4")
default:
    Text("5")
}

}

struct EmojiRatingView_Previews: PreviewProvider {
    static var previews: some View {
        EmojiRatingView(rating: 3)
    }
}

```

Tip: I used numbers in my text because emoji can cause havoc with e-readers, but you should replace those with whatever emoji you think represent the various ratings.

Notice how that specifically uses **Int16**, which makes interfacing with Core Data easier. And that's the entire view done – it really is that simple.

Now we can return to **ContentView** and do a first pass of its UI. This will replace the existing text view with a list and a **ForEach** over **books**. We don't need to provide an identifier for the **ForEach** because all Core Data's managed object class conform to **Identifiable** automatically, but things are trickier when it comes to creating views inside the **ForEach**.

You see, all the properties of our Core Data entity are optional, which means we need to make heavy use of nil coalescing in order to make our code work. We'll look at an alternative to this soon, but for now we'll just be scattering **??** around.

Inside the list we're going to have a **NavLink** that will eventually point to a detail view, and inside *that* we'll have our new **EmojiRatingView**, plus the book's title and author.

Project 11: Bookworm

So, replace the existing text view with this:

```
List {
    ForEach(books) { book in
        NavigationLink {
            Text(book.title ?? "Unknown Title")
        } label: {
            HStack {
                EmojiRatingView(rating: book.rating)
                    .font(.largeTitle)

                VStack(alignment: .leading) {
                    Text(book.title ?? "Unknown Title")
                        .font(.headline)
                    Text(book.author ?? "Unknown Author")
                        .foregroundColor(.secondary)
                }
            }
        }
    }
}
```

We'll come back to this screen soon enough, but first let's build the detail view...

Showing book details

When the user taps a book in **ContentView** we're going to present a detail view with some more information – the genre of the book, their brief review, and more. We're also going to reuse our new **RatingView**, and even customize it so you can see just how flexible SwiftUI is.

To make this screen more interesting, we're going to add some artwork that represents each category in our app. I've picked out some artwork already from Unsplash, and placed it into the project11-files folder for this book – if you haven't downloaded them, please do so now and then drag them into your asset catalog.

Unsplash has a license that allows us to use pictures commercially or non-commercially, with or without attribution, although attribution is appreciated. The pictures I've added are by Ryan Wallace, Eugene Triguba, Jamie Street, Alvaro Serrano, Joao Silas, David Dilbert, and Casey Horner – you can get the originals from <https://unsplash.com> if you want.

Next, create a new SwiftUI view called “DetailView”. This only needs one property, which is the book it should show, so please add that now:

```
let book: Book
```

Even just having that property is enough to break the preview code at the bottom of `DetailView.swift`. Previously this was easy to fix because we just sent in an example object, but with Core Data involved things are messier: creating a new book also means having a managed object context to create it inside.

To fix this, we can update our preview code to create a temporary managed object context, then use *that* to create our book. Once that's done we can pass in some example data to make our preview look good, then use the test book to create a detail view preview.

Creating a managed object context means we need to start by importing Core Data. Add this line near the top of `DetailView.swift`, next to the existing **import**:

```
import CoreData
```

Project 11: Bookworm

As for the previous code itself, replace whatever you have now with this:

```
struct DetailView_Previews: PreviewProvider {
    static let moc =
    NSManagedObjectContext(concurrencyType: .mainQueueConcurrencyTy
pe)

    static var previews: some View {
        let book = Book(context: moc)
        book.title = "Test book"
        book.author = "Test author"
        book.genre = "Fantasy"
        book.rating = 4
        book.review = "This was a great book; I really enjoyed it."

        return NavigationView {
            DetailView(book: book)
        }
    }
}
```

As you can see, creating a managed object context involves telling the system what concurrency type we want to use. This is another way of saying “which thread do you plan to access your data using?” For our example, using the main queue – that’s the one the app was launched using – is perfectly fine.

With that done we can turn our attention to more interesting problems, namely designing the view itself. To start with, we’re going to place the category image and genre inside a **ZStack**, so we can put one on top of the other nicely. I’ve picked out some styling that I think looks good, but you’re welcome to experiment with the styling all you want – the only thing I’d recommend you definitely keep is the **ScrollView**, which ensures our review will fit fully onto the screen no matter how long it is, what device the user has, or whether they have adjusted

their font sizes or not.

Replace the current **body** property with this:

```
ScrollView {
    ZStack(alignment: .bottomTrailing) {
        Image(book.genre ?? "Fantasy")
            .resizable()
            .scaledToFit()

        Text(book.genre?.uppercased() ?? "FANTASY")
            .font(.caption)
            .fontWeight(.black)
            .padding(8)
            .foregroundColor(.white)
            .background(.black.opacity(0.75))
            .clipShape(Capsule())
            .offset(x: -5, y: -5)
    }
}

.navigationBarTitle(book.title ?? "Unknown Book")
.navigationBarTitleDisplayMode(.inline)
```

That places the genre name in the bottom-right corner of the **ZStack**, with a background color, bold font, and a little padding to help it stand out.

Below that **ZStack** we're going to add the author, review, and rating. We don't want users to be able to adjust the rating here, so instead we can use another constant binding to turn this into a simple read-only view. Even better, because we used SF Symbols to create the rating image, we can scale them up seamlessly with a simple **font()** modifier, to make better use of all the space we have.

So, add these views directly below the previous **ZStack**:

Project 11: Bookworm

```
Text(book.author ?? "Unknown author")
    .font(.title)
    .foregroundColor(.secondary)

Text(book.review ?? "No review")
    .padding()

RatingView(rating: .constant(Int(book.rating)))
    .font(.largeTitle)
```

That completes **DetailView**, so we can head back to `ContentView.swift` to change the navigation link so it points to the correct thing:

```
NavigationLink {
    DetailView(book: book)
} label: {
```

Now run the app again, because you should be able to tap any of the books you've entered to show them in our new detail view.

Sorting fetch requests with SortDescriptor

When you use SwiftUI's `@FetchRequest` property wrapper to pull objects out of Core Data, you get to specify how you want the data to be sorted – should it be alphabetically by one of the fields? Or numerically with the highest numbers first? We specified an empty array, which might work OK for a handful of items but after 20 or so will just annoy the user.

In this project we have various fields that might be useful for sorting purposes: the title of the book, the author, or the rating are all sensible and would be good choices, but I suspect title is probably the most common so let's use that.

Fetch request sorting is performed using a new type called **SortDescriptor**, and we can create them from either one or two values: the attribute we want to sort on, and optionally whether it should be reversed or not. For example, we can alphabetically sort on the title attribute like this:

```
@FetchRequest(sortDescriptors: [  
    SortDescriptor(\.title)  
) var books: FetchedResults<Book>
```

Sorting is done in ascending order by default, meaning alphabetical order for text, but if you wanted to *reverse* the sort order you'd use this instead:

```
SortDescriptor(\.title, order: .reverse)
```

You can specify more than one sort descriptor, and they will be applied in the order you provide them. For example, if the user added the book "Forever" by Pete Hamill, then added "Forever" by Judy Blume – an entirely different book that just happens to have the same title – then specifying a second sort field is helpful.

So, we might ask for book title to be sorted ascending first, followed by book author ascending second, like this:

Project 11: Bookworm

```
@FetchRequest(sortDescriptors: [  
    SortDescriptor(\.title),  
    SortDescriptor(\.author)  
]) var books: FetchedResults<Book>
```

Having a second or even third sort field has little to no performance impact unless you have lots of data with similar values. With our books data, for example, almost every book will have a unique title, so having a secondary sort field is more or less irrelevant in terms of performance.

Deleting from a Core Data fetch request

We already used `@FetchRequest` to place Core Data objects into a SwiftUI `List`, and with only a little more work we can enable both swipe to delete and a dedicated Edit/Done button.

Just as with regular arrays of data, most of the work is done by attaching an `onDelete(perform:)` modifier to `ForEach`, but rather than just removing items from an array we instead need to find the requested object in our fetch request then use it to call `delete()` on our managed object context. Once all the objects are deleted we can trigger another save of the context; without that the changes won't actually be written out to disk.

So, start by adding this method to `ContentView`:

```
func deleteBooks(at offsets: IndexSet) {
    for offset in offsets {
        // find this book in our fetch request
        let book = books[offset]

        // delete it from the context
        moc.delete(book)
    }

    // save the context
    try? moc.save()
}
```

We can trigger that by adding an `onDelete(perform:)` modifier to the `ForEach` of `ContentView`, but remember: it needs to go on the `ForEach` and *not* the `List`.

Add this modifier now:

```
.onDelete(perform: deleteBooks)
```

Project 11: Bookworm

That gets us swipe to delete, and we can go one better by adding an Edit/Done button too. Find the **toolbar()** modifier in **ContentView**, and add another **ToolbarItem**:

```
ToolbarItem(placement: .navigationBarLeading) {
    EditButton()
}
```

That completes **ContentView**, so try running the app – you should be able to add and delete books freely now, and can delete by using swipe to delete or using the edit button.

Using an alert to pop a NavigationLink programmatically

You've already seen how **NavLink** lets us push to a detail screen, which might be a custom view or one of SwiftUI's built-in types such as **Text** or **Image**. Because we're inside a **NavigationView**, iOS automatically provides a "Back" button to let users get back to the previous screen, and they can also swipe from the left edge to go back. However, sometimes it's useful to programmatically go back – i.e., to move back to the previous screen when we want rather than when the user swipes.

To demonstrate this, we're going to add one last feature to our app that deletes whatever book the user is currently looking at. To do this we need to show an alert asking the user if they really want to delete the book, then delete the book from the current managed object context if that's what they want. Once that's done, there's no point staying on the current screen because its associated book doesn't exist any more, so we're going to pop the current view – remove it from the top of the **NavigationView** stack, so we move back to the previous screen.

First, we need three new properties in our **DetailView** struct: one to hold our Core Data managed object context (so we can delete stuff), one to hold our dismiss action (so we can pop the view off the navigation stack), and one to control whether we're showing the delete confirmation alert or not.

So, start by adding these three new properties to **DetailView**:

```
@Environment(\.managedObjectContext) var moc  
@Environment(\.dismiss) var dismiss  
@State private var showingDeleteAlert = false
```

The second step is writing a method that deletes the current book from our managed object context, and dismisses the current view. It doesn't matter that this view is being shown using a navigation link rather than a sheet – we still use the same **dismiss()** code.

Project 11: Bookworm

Add this method to **DetailView** now:

```
func deleteBook() {
    moc.delete(book)

    // uncomment this line if you want to make the deletion
    permanent
    // try? moc.save()
    dismiss()
}
```

The third step is to add an **alert()** modifier that watches **showingDeleteAlert**, and asks the user to confirm the action. So far we've been using simple alerts with a dismiss button, but here we need two buttons: one button to delete the book, and another to cancel. Both of these have specific button roles that automatically make them look correct, so we'll use those.

Apple provides very clear guidance on how we should label alert text, but it comes down to this: if it's a simple "I understand" acceptance then "OK" is good, but if you want users to make a choice then you should avoid titles like "Yes" and "No" and instead use verbs such as "Ignore", "Reply", and "Confirm".

In this instance, we're going to use "Delete" for the destructive button, then provide a cancel button next to it so users can back out of deleting if they want. So, add this modifier to the **ScrollView** in **DetailView**:

```
.alert("Delete book", isPresented: $showingDeleteAlert) {
    Button("Delete", role: .destructive, action: deleteBook)
    Button("Cancel", role: .cancel) { }
} message: {
    Text("Are you sure?")
}
```

The final step is to add a toolbar item that starts the deletion process – this just needs to flip the

Using an alert to pop a NavigationLink programmatically

showingDeleteAlert Boolean, because our **alert()** modifier is already watching it. So, add this one last modifier to the **ScrollView**:

```
.toolbar {
    Button {
        showingDeleteAlert = true
    } label: {
        Label("Delete this book", systemImage: "trash")
    }
}
```

You can now delete books in **ContentView** using swipe to delete or the edit button, or navigate into **DetailView** then tap the dedicated delete button in there – it should delete the book, update the list in **ContentView**, then automatically dismiss the detail view.

That's another app complete – good job!

Bookworm: Wrap up

Congratulations on finishing another SwiftUI project! With technologies like Core Data at your side, you're now capable of building some serious apps that interact with the user and – most importantly – *remember* what they entered. Although we only scratched the surface of Core Data, it's capable of a *lot* more and I expect Apple to keep expanding the link between Core Data and SwiftUI in future updates. In the meantime, the very next project focuses deeply on Core Data – there's lots to explore!

As for the other things you learned, you've now even more of SwiftUI's property wrappers, and I hope you're getting a sense for which one to choose and when. **@Binding** is particularly useful when building custom UI components, because its ability to share data between views is just so useful.

There's one last thing I'd like to leave you with, and it's something you might not even have noticed. When we built a star rating component, we created something that became a user-interactive control just like **Button** and **Slider**. However, we *didn't* stop to consider how it works with accessibility and that's a problem: **Button**, **Slider**, and others work great out of the box, but as soon as we start creating our *own* components we need to step in and do that work ourselves.

Building apps that are accessible for everyone is something everyone needs to take seriously, which is why I've dedicated a whole technique project to it in the future – we're going to be looking back at the previous projects we've made and seeing how we can improve them.

Anyway, first things first – you have a new review and some challenges. Good luck!

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Right now it's possible to select no title, author, or genre for books, which causes a problem for the detail view. Please fix this, either by forcing defaults, validating the form, or showing a default picture for unknown genres – you can choose.
2. Modify **ContentView** so that books rated as 1 star are highlighted somehow, such as having their name shown in red.
3. Add a new “date” attribute to the Book entity, assigning **Date.now** to it so it gets the current date and time, then format that nicely somewhere in **DetailView**.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Bookworm. If you don't already subscribe, you can start a free trial today.

Project 12

Core Data

Take an in-depth tour of how SwiftUI and Core Data work together

Core Data: Introduction

This technique project is going to explore Core Data in more detail, starting with a summary of some basic techniques then building up to tackling some more complex problems.

When you're working with Core Data, please try to keep in mind that it has been around for a long time – it was designed way before Swift existed, never mind SwiftUI, so occasionally you'll meet parts that don't work quite as well in Swift as we might hope. Hopefully we'll see this improve over the years ahead, but in the meantime be patient!

We have lots of Core Data to explore, so please create a fresh project where we can try it out. Call it “CoreDataProject” and *not* just “CoreData” because that will cause Xcode to get confused.

Make sure you do not check the “Use Core Data” box. Instead, please copy `DataController.swift` from your Bookworm project, make a new, empty Data Model called `CoreDataProject.xcdatamodeld`, then update `CoreDataProjectApp.swift` to create a **DataController** instance and inject it into the SwiftUI environment. This will leave you where we started in the early stages of the Bookworm project.

Important: As you're copying **DataController** from Bookworm to this new project, the data model file has changed – make sure you change **NSPersistentContainer** initializer to refer to the new data model file rather than Bookworm.

As you progress, you might sometimes find that Xcode will ignore changes made in the Core Data editor, so I like to drive the point home by pressing Cmd+S before going to another file. Failing that, restart Xcode!

All set? Let's go!

Tip: Sometimes you'll see a heading titled “Want to go further?” This contains some bonus examples that help take your knowledge further, but you don't need to follow here unless you want to – think of it as extra credit.

Why does `.self` work for `ForEach`?

Previously we looked at the various ways `ForEach` can be used to create dynamic views, but they all had one thing in common: SwiftUI needs to know how to identify each dynamic view uniquely so that it can animate changes correctly.

If an object conforms to the **Identifiable** protocol, then SwiftUI will automatically use its **id** property for unquing. If we don't use **Identifiable**, then we can use a keypath for a property we know to be unique, such as a book's ISBN number. But if we don't conform to **Identifiable** and don't have a keypath that is unique, we can often use `\.self`.

Previously we used `\.self` for primitive types such as **Int** and **String**, like this:

```
List {  
    ForEach([2, 4, 6, 8, 10], id: \.self) {  
        Text("\($0) is even")  
    }  
}
```

With Core Data we can use a unique identifier if we want, but we can also use `\.self` and also have something that works well.

When we use `\.self` as an identifier, we mean “the whole object”, but in practice that doesn’t mean much – a struct is a struct, so it doesn’t have any sort of specific identifying information other than its contents. So what actually happens is that Swift computes the *hash value* of the struct, which is a way of representing complex data in fixed-size values, then uses that hash as an identifier.

Hash values can be generated in any number of ways, but the concept is identical for all hash-generating functions:

1. Regardless of the input size, the output should be the same fixed size.
2. Calculating the same hash for an object twice in a row should return the same value.

Why does `.self` work for `ForEach`?

Those two sound simple, but think about it: if we get the hash of “Hello World” and the hash of the complete works of Shakespeare, both will end up being the same size. This means it’s not possible to convert the hash back into its original value – we can’t convert 40 seemingly random hexadecimal letters and numbers into the complete works of Shakespeare.

Hashes are commonly used for things like data verification. For example, if you download a 8GB zip file, you can check that it’s correct by comparing your local hash of that file against the server’s – if they match, it means the zip file is identical. Hashes are also used with dictionary keys and sets; that’s how they get their fast look up.

All this matters because when Xcode generates a class for our managed objects, it makes that class conform to **Hashable**, which is a protocol that means Swift can generate hash values for it, which in turn means we can use `\.self` for the identifier. This is also why **String** and **Int** work with `\.self`: they also conform to **Hashable**.

Hashable is a bit like **Codable**: if we want to make a custom type conform to **Hashable**, then as long as everything it contains also conforms to **Hashable** then we don’t need to do any work. To demonstrate this, we could create a custom struct that conforms to **Hashable** rather than **Identifiable**, and use `\.self` to identify it:

```
struct Student: Hashable {
    let name: String
}

struct ContentView: View {
    let students = [Student(name: "Harry Potter"), Student(name: "Hermione Granger")]

    var body: some View {
        List(students, id: \.self) { student in
            Text(student.name)
        }
    }
}
```

Project 12: Core Data

}

We can make **Student** conform to **Hashable** because all its properties already conform to **Hashable**, so Swift will calculate the hash values of each property then combine them into one hash that represents the whole struct. Of course, if we end up with two students that have the same name we'll hit problems, just like if we had an array of strings with two identical strings.

Now, you might think this leads to a problem: if we create two Core Data objects with the same values, they'll generate the same hash, and we'll hit animation problems. However, Core Data does something really smart here: the objects it creates for us actually have a selection of other properties beyond those we defined in our data model, including one called the object ID – an identifier that is unique to that object, regardless of what properties it contains. These IDs are similar to **UUID**, although Core Data generates them sequentially as we create objects.

So, `\.self` works for anything that conforms to **Hashable**, because Swift will generate the hash value for the object and use that to uniquely identify it. This also works for Core Data's objects because they already conform to **Hashable**. So, if you want to use a specific identifier that's awesome, but you don't need to because `\.self` is also an option.

Warning: Although calculating the same hash for an object twice in a row should return the same value, calculating it between two runs of your app – i.e., calculating the hash, quitting the app, relaunching, then calculating the hash again – can return different values.

Creating NSManagedObject subclasses

When we create a new Core Data entity, Xcode automatically generates a managed object class for us when we build our code. We can then use that in a SwiftUI `@FetchRequest` to show data in our user interface, but as you've seen it's quite painful: there are lots of optionals to unwrap, so you need to scatter nil coalescing around in order to make your code work.

There are two solutions to this: a fast and easy one that can sometimes end up being problematic, or a slightly slower solution that works better in the long term.

First, let's create an entity to work with: open your data model and create an entity called Movie with the attributes "title" (string), "director" (string), and "year" (an integer 16). Before you leave the data model editor, I'd like you to go to the View menu and choose Inspectors > Show Data Model Inspector, which brings up a pane on the right of Xcode containing more information about whatever you have selected right now.

When you select Movie you'll see a variety of data model options for that entity, but there's one in particular I'd like you to look at: "Codegen". This controls how Xcode generates the entity as a managed object class when we build our project, and by default it will be Class Definition. I'd like to change that to Manual/None, which gives us full control over how the class is made.

Now that Xcode is no longer generating a **Movie** class for us to use in code, we can't use it in code unless we actually make the class with some real Swift code. To do that, go to the Editor menu and choose Create NSManagedObject Subclass, make sure "CoreDataProject" is selected then press Next, then make sure Movie is selected and press Next again. You'll be asked where Xcode should save its code, so please make sure you choose "CoreDataProject" with a yellow folder icon on its left, and select the CoreDataProject folder too. When you're ready, press Create to finish the process.

What we just did was ask Xcode to convert its generated code into actual Swift files that we

Project 12: Core Data

can see and change, although keep in mind if you change the files Xcode generated for us then re-generate those files, your changes will be lost.

Xcode will have generated two files for us, but we only care about one of them: Movie+CoreDataProperties.swift. Inside there you'll see these three lines of code:

```
@NSManaged public var title: String?  
@NSManaged public var director: String?  
@NSManaged public var year: Int16
```

In that tiny slice of code you can see three things:

1. This is where our optional problem stems from.
2. **year** is not optional, which means Core Data will assume a default value for us.
3. It uses **@NSManaged** on all three properties.

@NSManaged is *not* a property wrapper – this is much older than property wrappers in SwiftUI. Instead, this reveals a little of how Core Data works internally: rather than those values actually existing as properties in the class, they are really just there to read and write from a dictionary that Core Data uses to store its information. When we read or write the value of a property that is **@NSManaged**, Core Data catches that and handles it internally – it's far from a simple Swift string.

Now, you might look at that code and think “I don't want optionals there,” and change it to this:

```
@NSManaged public var title: String  
@NSManaged public var director: String  
@NSManaged public var year: Int16
```

And you know what? That will absolutely work. You can make **Movie** objects with just the same code as before, use fetch requests to query them, save their managed object contexts, and more, all with no problems.

However, you might notice something strange: even though our properties aren't optional any more, it's still possible to create an instance of the **Movie** class without providing those values. This *ought* to be impossible: these properties aren't optional, which means they must have values all the time, and yet we can create them without values.

What's happening here is a little of that **@NSManaged** magic leaking out – remember, these aren't real properties, and as a result **@NSManaged** is letting us do things that ought not to work. The fact that it *does* work is neat, and for small Core Data projects and/or learners I think removing the optionality is a great idea. However, there's a deeper problem: Core Data is lazy.

Remember Swift's **lazy** keyword, and how it lets us delay work until we actually need it? Core Data does much the same thing, except with data: sometimes it looks like some data has been loaded when it really hasn't been because Core Data is trying to minimize its memory impact. Core Data calls these *faults*, in the sense of a “fault line” – a line between where something exists and where something is just a placeholder.

We don't need to do any special work to handle these faults, because as soon as we try to read them Core Data transparently fetches the real data and sends it back – another benefit of **@NSManaged**. However, when we start futzing with the types of Core Data's properties we risk exposing its peculiar underbelly. This thing specifically does not work the way Swift expects, and if we try to circumvent that then we're pretty much inviting problems – values we've said definitely won't be nil might suddenly be nil at any point.

Instead, you might want to consider adding computed properties that help us access the optional values safely, while also letting us store your nil coalescing code all in one place. For example, adding this as a property on **Movie** ensures that we always have a valid title string to work with:

```
public var wrappedTitle: String {
    title ?? "Unknown Title"
}
```

Project 12: Core Data

This way the whole rest of your code doesn't have to worry about Core Data's optionality, and if you want to make changes to default values you can do it in a single file.

Conditional saving of NSManagedObjectContext

We've been using the `save()` method of `NSManagedObjectContext` to flush out all unsaved changes to permanent storage, but what we *haven't* done is check whether any changes actually *need* to be saved. This is often OK, because it's common to place `save()` calls only after we specifically made a change such as inserting or deleting data.

However, it's also common to bulk your saves together so that you save everything at once, which has a lower performance impact. In fact, Apple specifically states that we should always check for uncommitted changes before calling `save()`, to avoid making Core Data do work that isn't required.

Fortunately, we can check for changes in two ways. First, every managed object is given a `hasChanges` property, that is true when the object has unsaved changes. And, the entire context also contains a `hasChanges` property that checks whether any object owned by the context has changes.

So, rather than call `save()` directly you should always wrap it in a check first, like this:

```
if moc.hasChanges {  
    try? moc.save()  
}
```

It's a small change to make, but it matters – there's no point doing work that isn't needed, no matter how small, particularly because if you do enough small work you realize you've stacked up some big work.

Ensuring Core Data objects are unique using constraints

By default Core Data will add any object you want, but this can get messy very quickly, particularly if you know two or more objects don't make sense at the same time. For example, if you stored details of contacts using their email address, it wouldn't make sense to have two or three different contacts attached to the same email address.

To help resolve this, Core Data gives us *constraints*: we can make one attribute constrained so that it must always be unique. We can then go ahead and make as many objects as we want, unique or otherwise, but as soon as we ask Core Data to save those objects it will resolve duplicates so that only one piece of data gets written. Even better, if there was some data already written that clashes with our constraint, we can choose how it should handle merging the data.

To try this out, create a new entity called Wizard, with one string attribute called "name". Now select the Wizard entity, look in the data model inspector for Constraints, and press the + button directly below. You should see "comma-separated,properties" appear, giving us an example to work from. Select that and press enter to rename it, and give it the text "name" instead – that makes our name attribute unique. Remember to press Cmd+S to save your change!

Now go over to ContentView.swift and give it this code:

```
struct ContentView: View {
    @Environment(\.managedObjectContext) var moc

    @FetchRequest(sortDescriptors: [ ]) var wizards:
    FetchedResults<Wizard>

    var body: some View {
        VStack {
```

Ensuring Core Data objects are unique using constraints

```
List(wizards, id: \.self) { wizard in
    Text(wizard.name ?? "Unknown")
}

Button("Add") {
    let wizard = Wizard(context: moc)
    wizard.name = "Harry Potter"
}

Button("Save") {
    do {
        try moc.save()
    } catch {
        print(error.localizedDescription)
    }
}
```

You can see that has a list for showing wizards, one button for adding wizards, and a second button for saving. When you run the app you'll find that you can press Add multiple times to see "Harry Potter" slide into the table, but when you press "Save" we get an error instead – Core Data has detected the collision and is refusing to save the changes.

If you *want* Core Data to write the changes, you need to open DataController.swift and adjust the loadPersistentStores() completion handler to specify how data should be merged in this situation:

```
container.loadPersistentStores { description, error in
    if let error = error {
        print("Core Data failed to load: \
(error.localizedDescription)")
```

Project 12: Core Data

```
        return  
    }  
  
    self.container.viewContext.mergePolicy =  
NSMergePolicy.mergeByPropertyObjectTrump  
}
```

That asks Core Data to merge duplicate objects based on their properties – it tries to intelligently overwrite the version in its database using properties from the new version. If you run the code again you'll see something quite brilliant: you can press Add as many times as you want, but when you press Save it will all collapse down into a single row because Core Data strips out the duplicates.

Filtering @FetchRequest using NSPredicate

When we use SwiftUI's **@FetchRequest** property wrapper, we can provide an array of sort descriptors to control the ordering of results, but we can also provide an **NSPredicate** to control *which* results should be shown. Predicates are simple tests, and the test will be applied to each object in our Core Data entity – only objects that pass the test will be included in the resulting array.

The syntax for **NSPredicate** isn't something you can guess easily, but realistically you're only ever going to want a few types of predicate so it's not as bad as you think.

To try out some predicates, create a new entity called Ship with two string attributes: "name" and "universe".

Now modify ContentView.swift to this:

```
import CoreData
import SwiftUI

struct ContentView: View {
    @Environment(\.managedObjectContext) var moc
    @FetchRequest(sortDescriptors: [], predicate: nil) var ships:
    FetchedResults<Ship>

    var body: some View {
        VStack {
            List(ships, id: \.self) { ship in
                Text(ship.name ?? "Unknown name")
            }
        }

        Button("Add Examples") {

```

Project 12: Core Data

```
let ship1 = Ship(context: moc)
ship1.name = "Enterprise"
ship1.universe = "Star Trek"

let ship2 = Ship(context: moc)
ship2.name = "Defiant"
ship2.universe = "Star Trek"

let ship3 = Ship(context: moc)
ship3.name = "Millennium Falcon"
ship3.universe = "Star Wars"

let ship4 = Ship(context: moc)
ship4.name = "Executor"
ship4.universe = "Star Wars"

try? moc.save()
}

}

}

}
```

We can now press the button to inject some sample data into Core Data, but right now we don't have a predicate. To fix that we need to replace the `nil` predicate value with some sort of test that can be applied to our objects.

For example, we could ask for ships that are from Star Wars, like this:

```
@FetchRequest(sortDescriptors: [], predicate:
NSPredicate(format: "universe == 'Star Wars'")) var ships:
FetchedResults<Ship>
```

That gets complicated if your data includes quote marks, so it's more common to use special

Filtering @FetchRequest using NSPredicate

syntax instead: `%@` means “insert some data here”, and allows us to provide that data as a parameter to the predicate rather than inline.

So, we could instead write this:

```
NSPredicate(format: "universe == %@", "Star Wars"))
```

As well as `==`, we can also use comparisons such as `<` and `>` to filter our objects. For example this will return Defiant, Enterprise, and Executor:

```
NSPredicate(format: "name < %@", "F"))
```

`%@` is doing a lot of work behind the scenes, particularly when it comes to converting native Swift types to their Core Data equivalents. For example, we could use an **IN** predicate to check whether the universe is one of three options from an array, like this:

```
NSPredicate(format: "universe IN %@", ["Aliens", "Firefly",  
"Star Trek"])
```

We can also use predicates to examine part of a string, using operators such as **BEGINSWITH** and **CONTAINS**. For example, this will return all ships that start with a capital E:

```
NSPredicate(format: "name BEGINSWITH %@", "E"))
```

That predicate is case-sensitive; if you want to ignore case you need to modify it to this:

```
NSPredicate(format: "name BEGINSWITH[c] %@", "e"))
```

CONTAINS[c] works similarly, except rather than starting with your substring it can be anywhere inside the attribute.

Finally, you can flip predicates around using **NOT**, to get the inverse of their regular behavior. For example, this finds all ships that don’t start with an E:

Project 12: Core Data

```
NSPredicate(format: "NOT name BEGINSWITH[c] %@", "e") )
```

If you need more complicated predicates, join them using **AND** to build up as much precision as you need, or add an import for Core Data and take a look at **NSCompoundPredicate** – it lets you build one predicate out of several smaller ones.

Dynamically filtering @FetchRequest with SwiftUI

One of the SwiftUI questions I've been asked more than any other is this: how can I dynamically change a Core Data **@FetchRequest** to use a different predicate or sort order? The question arises because fetch requests are created as a property, so if you try to make them reference another property Swift will refuse.

There is a simple solution here, and it is usually pretty obvious in retrospect because it's exactly how everything else works: we should carve off the functionality we want into a separate view, then inject values into it.

I want to demonstrate this with some real code, so I've put together the simplest possible example: it adds three singers to Core Data, then uses two buttons to show either singers whose last name ends in A or S.

Start by creating a new Core Data entity called Singer and give it two string attributes: "firstName" and "lastName". Use the data model inspector to change its Codegen to Manual/None, then go to the Editor menu and select Create NSManagedObject Subclass so we can get a **Singer** class we can customize.

Once Xcode has generated files for us, open Singer+CoreDataProperties.swift and add these two properties that make the class easier to use with SwiftUI:

```
var wrappedFirstName: String {
    firstName ?? "Unknown"
}

var wrappedLastName: String {
    lastName ?? "Unknown"
}
```

OK, now onto the real work.

Project 12: Core Data

The first step is to design a view that will host our information. Like I said, this is also going to have two buttons that lets us change the way the view is filtered, and we're going to have an extra button to insert some testing data so you can see how it works.

First, add two properties to your **ContentView** struct so that we have a managed object context we can save objects to, and some state we can use as a filter:

```
@Environment(\.managedObjectContext) var moc
@State private var lastNameFilter = "A"
```

For the body of the view, we're going to use a **VStack** with three buttons, plus a comment for where we want the **List** to show matching singers:

```
VStack {
    // list of matching singers

    Button("Add Examples") {
        let taylor = Singer(context: moc)
        taylor.firstName = "Taylor"
        taylor.lastName = "Swift"

        let ed = Singer(context: moc)
        ed.firstName = "Ed"
        ed.lastName = "Sheeran"

        let adele = Singer(context: moc)
        adele.firstName = "Adele"
        adele.lastName = "Adkins"

        try? moc.save()
    }
}
```

Dynamically filtering @FetchRequest with SwiftUI

```
Button("Show A") {
    lastNameFilter = "A"
}

Button("Show S") {
    lastNameFilter = "S"
}
```

So far, so easy. Now for the interesting part: we need to replace that `// list of matching singers` comment with something real. This *isn't* going to use `@FetchRequest` because we want to be able to create a custom fetch request inside an initializer, but the code we'll be using instead is almost identical.

Create a new SwiftUI view called “`FilteredList`”, and give it this property:

```
@FetchRequest var fetchRequest: FetchedResults<Singer>
```

That will store our fetch request, so that we can loop over it inside the `body`. However, we don't *create* the fetch request here, because we still don't know what we're searching for. Instead, we're going to create a custom initializer that accepts a filter string and uses that to set the `fetchRequest` property.

Add this initializer now:

```
init(filter: String) {
    _fetchRequest = FetchRequest<Singer>(sortDescriptors: [],
predicate: NSPredicate(format: "lastName BEGINSWITH %@", filter))
}
```

That will run a fetch request using the current managed object context. Because this view will be used inside `ContentView`, we don't even need to inject a managed object context into the

Project 12: Core Data

environment – it will inherit the context from **ContentView**.

Did you notice how there's an underscore at the start of `_fetchRequest`? That's intentional. You see, we're not writing to the fetched results object inside our fetch request, but instead writing a wholly new fetch request.

To understand this, think about the `@State` property wrapper. Behind the scenes this is implemented as a struct called `State`, which contains whatever value we put inside – an integer, for example. If we have an `@State` property called `score` and assign the value 10 to it, we mean to put 10 into the integer inside the `State` property wrapper. However, we can also assign a value to `_score` – we can write a wholly new `State` struct in there, if needed.

So, by assigning to `_fetchRequest`, we aren't trying to say "here's a whole bunch of new results we want to you to use," but instead we're telling Swift we want to change the whole fetch request itself.

All that remains is to write the body of the view, so give the view this body:

```
var body: some View {
    List(fetchRequest, id: \.self) { singer in
        Text("\(singer.wrappedFirstName) \
(singer.wrappedLastName)")
    }
}
```

As for the preview struct for `FilteredList`, you can remove it safely.

Now that the view is complete, we can return to **ContentView** and replace the comment with some actual code that passes our filter into `FilteredList`:

```
FilteredList(filter: lastNameFilter)
```

Now run the program to give it a try: tap the Add Examples button first to create three singer objects, then tap either "Show A" or "Show S" to toggle between surname letters. You should

see our **List** dynamically update with different data, depending on which button you press.

So, it took a *little* new knowledge to make this work, but it really wasn't that hard – as long as you think like SwiftUI, the solution is right there.

Tip: You might look at our code and think that every time the view is recreated – which is every time any state changes in our container view – we're also recreating the fetch request, which in turn means reading from the database when nothing else has changed.

That might seem terribly wasteful, and it *would* be terribly wasteful if it actually happened. Fortunately, Core Data won't do anything like this: it will only actually re-run the database query when the filter string changes, even if the view is recreated.

Want to go further?

For more flexibility, we could improve our **FilteredList** view so that it works with any kind of entity, and can filter on any field. To make this work properly, we need to make a few changes:

1. Rather than specifically referencing the **Singer** class, we're going to use generics with a constraint that whatever is passed in must be an **NSManagedObject**.
2. We need to accept a second parameter to decide which key name we want to filter on, because we might be using an entity that doesn't have a **lastName** attribute.
3. Because we don't know ahead of time what each entity will contain, we're going to let our containing view decide. So, rather than just using a text view of a singer's name, we're instead going to ask for a closure that can be run to configure the view however they want.

There are two complex parts in there. The first is the closure that decides the content of each list row, because it needs to use two important pieces of syntax. We looked at these towards the end of our earlier technique project on views and modifiers, but if you missed them:

- **@ViewBuilder** lets our containing view (whatever is using the list) send in multiple views if they want.
- **@escaping** says the closure will be stored away and used later, which means Swift needs

Project 12: Core Data

to take care of its memory.

The second complex part is how we let our container view customize the search key. Previously we controlled the filter value like this:

```
NSPredicate(format: "lastName BEGINSWITH %@", filter)
```

So you *might* take an educated guess and write code like this:

```
NSPredicate(format: "%@ BEGINSWITH %@", keyName, filter)
```

However, that won't work. You see, when we write `%@` Core Data automatically inserts quote marks for us so that the predicate reads correctly. This is helpful, because if our string contains quote marks it will automatically make sure they don't clash with the quote marks it adds.

This means when we use `%@` for the attribute name we might end up with a predicate like this:

```
NSPredicate(format: "'lastName' BEGINSWITH 'S'")
```

And that's not correct: the attribute name should *not* be in quote marks.

To resolve this, **NSPredicate** has a special symbol that can be used to replace attribute names: `%K`, for “key”. This will insert values we provide, but won't add quote marks around them. The correct predicate is this:

```
NSPredicate(format: "%K BEGINSWITH %@", filterKey, filterValue)
```

So, add an import for CoreData so we can reference **NSManagedObject**, then replace your current **FilteredList** struct with this:

```
struct FilteredList<T: NSManagedObject, Content: View>: View {
    @FetchRequest var fetchRequest: FetchedResults<T>
```

Dynamically filtering @FetchRequest with SwiftUI

```
// this is our content closure; we'll call this once for each
// item in the list
let content: (T) -> Content

var body: some View {
    List(fetchRequest, id: \.self) { singer in
        self.content(singer)
    }
}

init(filterKey: String, filterValue: String, @ViewBuilder
content: @escaping (T) -> Content) {
    _fetchRequest = FetchRequest<T>(sortDescriptors: [],
predicate: NSPredicate(format: "%K BEGINSWITH %@", filterKey,
filterValue))
    self.content = content
}
}
```

We can now use that new filtered list by upgrading **ContentView** like this:

```
FilteredList(filterKey: "lastName", filterValue:
lastNameFilter) { (singer: Singer) in
    Text("\(singer.wrappedFirstName) \(singer.wrappedLastName)")
}
```

Notice how I've specifically used **(singer: Singer)** as the closure's parameter – this is required so that Swift understands how **FilteredList** is being used. Remember, we said it could be any type of **NSManagedObject**, but in order for Swift to know exactly *what* type of managed object it is we need to be explicit.

Anyway, with that change in place we now use our list with any kind of filter key and any kind

Project 12: Core Data

of entity – it's much more useful!

One-to-many relationships with Core Data, SwiftUI, and `@FetchRequest`

Core Data allows us to link entities together using relationships, and when we use `@FetchRequest` Core Data sends all that data back to us for use. However, this is one area where Core Data shows its age a little: to get relationships to work well we need to make a custom `NSManagedObject` subclass that provides wrappers that are more friendly to SwiftUI.

To demonstrate this, we're going to build two Core Data entities: one to track candy bars, and one to track countries where those bars come from.

Relationships come in four forms:

- A one to one relationship means that one object in an entity links to exactly one object in another entity. In our example, this would mean that each type of candy has one country of origin, and each country could make only one type of candy.
- A one to many relationship means that one object in an entity links to many objects in another entity. In our example, this would mean that one type of candy could have been introduced simultaneously in many countries, but that each country still could only make one type of candy.
- A many to one relationship means that many objects in an entity link to one object in another entity. In our example, this would mean that each type of candy has one country of origin, and that each country can make many types of candy.
- A many to many relationship means that many objects in an entity link to many objects in another entity. In our example, this would mean that one type of candy had been introduced simultaneously in many countries, and each country can make many types of candy.

All of those are used at different times, but in our candy example the many to one relationship makes the most sense – each type of candy was invented in a single country, but each country

Project 12: Core Data

can have invented many types of candy.

So, open your data model and add two entities: Candy, with a string attribute called “name”, and Country, with string attributes called “fullName” and “shortName”. Although some types of candy have the same name – see “Smarties” in the US and the UK – countries are definitely unique, so please add a constraint for “shortName”.

Tip: Don’t worry if you’ve forgotten how to add constraints: select the Country entity, go to the View menu and choose Inspectors > Show Data Model Inspector, click the + button under Constraints, and rename the example to “shortName”.

Before we’re done with this data model, we need to tell Core Data there’s a one-to-many relationship between Candy and Country:

- With Country selected, press + under the Relationships table. Call the relationship “candy”, change its destination to Candy, then over in the data model inspector change Type to To Many.
- Now select Candy, and add another relationship there. Call the relationship “origin”, change its destination to “Country”, then set its inverse to “candy” so Core Data understands the link goes both ways.

That completes our entities, the next step is to take a look at the code Xcode generates for us. **Remember to press Cmd+S to force Xcode to save your changes.**

Select both Candy and Country and set their Codegen to Manual/None, then go to the Editor menu and choose Create NSManagedObject Subclass to create code for both our entities – remember to save them in the CoreDataProject group and folder.

As we chose two entities, Xcode will generate four Swift files for us.

Candy+CoreDataProperties.swift will be pretty much exactly what you expect, although notice how **origin** is now a **Country**. Country+CoreDataProperties.swift is more complex, because Xcode also generated some methods for us to use.

Previously we looked at how to clean up Core Data’s optionals using **NSManagedObject**

One-to-many relationships with Core Data, SwiftUI, and @FetchRequest

subclasses, but here there's a bonus complexity: the **Country** class has a **candy** property that is an **NSSet**. This is the older, Objective-C data type that is equivalent to Swift's **Set**, but we can't use it with SwiftUI's **ForEach**.

To fix this we need to modify the files Xcode generated for us, adding convenience wrappers that make SwiftUI work well. For the **Candy** class this is as easy as just wrapping the **name** property so that it always returns a string:

```
public var wrappedName: String {
    name ?? "Unknown Candy"
}
```

For the **Country** class we can create the same string wrappers around **shortName** and **fullName**, like this:

```
public var wrappedShortName: String {
    shortName ?? "Unknown Country"
}

public var wrappedFullName: String {
    fullName ?? "Unknown Country"
}
```

However, things are more complicated when it comes to **candy**. This is an **NSSet**, which could contain anything at all, because Core Data hasn't restricted it to just instances of **Candy**.

So, to get this thing into a useful form for SwiftUI we need to:

1. Convert it from an **NSSet** to a **Set<Candy>** – a Swift-native type where we know the types of its contents.
2. Convert that **Set<Candy>** into an array, so that **ForEach** can read individual values from there.
3. Sort that array, so the candy bars come in a sensible order.

Project 12: Core Data

Swift actually lets us perform steps 2 and 3 in one, because sorting a set automatically returns an array. However, sorting the array is harder than you might think: this is an array of custom types, so we can't just use `sorted()` and let Swift figure it out. Instead, we need to provide a closure that accepts two candy bars and returns true if the first candy should be sorted before the second.

So, please add this computed property to `Country` now:

```
public var candyArray: [Candy] {
    let set = candy as? Set<Candy> ?? []
    return set.sorted {
        $0.wrappedName < $1.wrappedName
    }
}
```

That completes our Core Data classes, so now we can write some SwiftUI code to make all this work.

Open `ContentView.swift` and give it these two properties:

```
@Environment(\.managedObjectContext) var moc
@FetchRequest(sortDescriptors: []) var countries:
FetchedResults<Country>
```

Notice how we don't need to specify anything about the relationships in our fetch request – Core Data understands the entities are linked, so it will just fetch them all as needed.

As for the body of the view, we're going to use a `List` with two `ForEach` views inside it: one to create a section for each country, and one to create the candy inside each country. This `List` will in turn go inside a `VStack` so we can add a button below to generate some sample data:

```
VStack {
    List {
```

One-to-many relationships with Core Data, SwiftUI, and @FetchRequest

```
ForEach(countries, id: \.self) { country in
    Section(country.wrappedFullName) {
        ForEach(country.candyArray, id: \.self) { candy in
            Text(candy.wrappedName)
        }
    }
}

Button("Add") {
    let candy1 = Candy(context: moc)
    candy1.name = "Mars"
    candy1.origin = Country(context: moc)
    candy1.origin?.shortName = "UK"
    candy1.origin?.fullName = "United Kingdom"

    let candy2 = Candy(context: moc)
    candy2.name = "KitKat"
    candy2.origin = Country(context: moc)
    candy2.origin?.shortName = "UK"
    candy2.origin?.fullName = "United Kingdom"

    let candy3 = Candy(context: moc)
    candy3.name = "Twix"
    candy3.origin = Country(context: moc)
    candy3.origin?.shortName = "UK"
    candy3.origin?.fullName = "United Kingdom"

    let candy4 = Candy(context: moc)
    candy4.name = "Toblerone"
    candy4.origin = Country(context: moc)
    candy4.origin?.shortName = "CH"
```

Project 12: Core Data

```
candy4.origin?.fullName = "Switzerland"

try? moc.save()
}

}
```

Make sure you run that code, because it works really well – all our candy bars are automatically sorted into sections when the Add button is tapped. Even better, because we did all the heavy lifting inside our **NSManagedObject** subclasses, the resulting SwiftUI code is actually remarkably straightforward – it has no idea that an **NSSet** is behind the scenes, and is much easier to understand as a result.

Tip: If you don't see your candy bars sorted into sections after pressing Add, make sure you haven't removed the **mergePolicy** change from the **DataController** class. As a reminder, it should set to **NSMergePolicy.mergeByPropertyObjectTrump**.

Core Data: Wrap up

Core Data might seem like a dry topic at first, but it's so useful when building apps – you've seen how it can add, delete, sort, filter, and more, all with relatively simple code. Yes, a few parts are a little murky in Swift – **NSPredicate**, for example, could do with some refinement, and **NSSet** is never pleasant to deal with – but with a little work on our behalf this stops being a problem.

Perhaps the most important thing about Core Data is that it's guaranteed to be there for all apps, on all of Apple's platforms. This means you can use it regardless of your needs: maybe it's for saving important data, maybe it's just a cache of content you downloaded; it doesn't matter, because Core Data will do a great job of managing it for you.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

All three of these tasks require you to modify the **FilteredList** view we made:

1. Make it accept a string parameter that controls which predicate is applied. You can use Swift's string interpolation to place this in the predicate.
2. Modify the predicate string parameter to be an enum such as **.beginsWith**, then make that enum get resolved to a string inside the initializer.
3. Make **FilteredList** accept an array of **SortDescriptor** objects to get used in its fetch request.

Project 12: Core Data

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Core Data. If you don't already subscribe, you can start a free trial today.

Tip: If you're using the generic version of **FilteredList**, your sort descriptors are of type **SortDescriptor<T>**. If you're using the simpler, non-generic version, your sort descriptors are of type **SortDescriptor<Singer>**.

Project 13

Instafilter

Learn to link SwiftUI, UIKit, and Core Image in one app

Instafilter: Introduction

In this project we’re going to build an app that lets the user import photos from their library, then modify them using various image effects. We’ll cover a number of new techniques, but at the center of it all are one useful app development skill – using Apple’s Core Image framework – and one important SwiftUI skill – integrating with UIKit. There are other things too, but those two are the big takeaways.

Core Image is Apple’s high-performance framework for manipulating images, and it’s extraordinarily powerful. Apple have designed dozens of example image filters for us, providing things like blurs, color shifts, pixellation, and more, and all are optimized to take full advantage of the graphics processing unit (GPU) on iOS devices.

Tip: Although you *can* run your Core Image app in the simulator, don’t be surprised if most things are really slow – you’ll only get great performance when you run on a physical device.

As for integrating with UIKit, you might wonder why this is needed – after all, SwiftUI is designed to replace UIKit, right? Well, sort of. Before SwiftUI launched, almost every iOS app was built with UIKit, which means that there are probably several billion lines of UIKit code out there. So, if you want to integrate SwiftUI into an existing project you’ll need to learn how to make the two work well together.

But there’s another reason, and I’m hoping it won’t always be a reason: many parts of Apple’s frameworks don’t have SwiftUI wrappers yet, which means if you want to integrate MapKit, Safari, or other important APIs, you need to know how to wrap their code for use with SwiftUI. I’ll be honest, the code required to make this work isn’t pretty, but at this point in your SwiftUI career you’re more than ready for it.

As always we have some techniques to cover before we get into the project, so please create a new iOS app using the App template, naming it “Instafilter”.

How property wrappers become structs

You've seen how SwiftUI lets us store changing data in our structs by using the `@State` property wrapper, how we can bind that state to the value of a UI control using `$`, and how changes to that state automatically cause SwiftUI to reinvoke the `body` property of our struct.

All that combined lets us write code such as this:

```
struct ContentView: View {
    @State private var blurAmount = 0.0

    var body: some View {
        VStack {
            Text("Hello, World!")
                .blur(radius: blurAmount)

            Slider(value: $blurAmount, in: 0...20)

            Button("Random Blur") {
                blurAmount = Double.random(in: 0...20)
            }
        }
    }
}
```

If you run that, you'll find that dragging the slider left and right adjusts the blur amount for the text label, exactly as you would expect, and tapping the button immediately jumps to a random blur amount.

Now, let's say we want that binding to do *more* than just handle the radius of the blur effect. Perhaps we want to run a method, or just print out the value for debugging purposes. You

Project 13: Instafilter

might try updating the property like this:

```
@State private var blurAmount = 0.0 {  
    didSet {  
        print("New value is \(blurAmount)")  
    }  
}
```

If you run that code, you'll be disappointed: as you drag the slider around you'll see the blur amount change, but you won't see our `print()` statement being triggered – in fact, nothing will be output at all. But if you try pressing the *button* you *will* see a message printed.

To understand what's happening here, I want you to think about when we looked at Core Data: we used the `@FetchRequest` property wrapper to query our data, but I also showed you how to use the `FetchRequest` struct directly so that we had more control over how it was created.

Property wrappers have that name because they wrap our property inside another struct. What this means is that when we use `@State` to wrap a string, the actual type of property we end up with is a `State<String>`. Similarly, when we use `@Environment` and others we end up with a struct of type `Environment` that contains some other value inside it.

Previously I explained that we can't modify properties in our views because they are structs, and are therefore fixed. However, *now* you know that `@State itself` produces a struct, so we have a conundrum: how come *that* struct can be modified?

Xcode has a really helpful command called “Open Quickly” (accessed using Cmd+Shift+O), which lets you find any file or type in your project or any of the frameworks you have imported. Activate it now, and type “State” – hopefully the first result says SwiftUI below it, but if not please find that and select it.

You'll be taken to a generated interface for SwiftUI, which is essentially all the parts that SwiftUI exposes to us. There's no implementation code in there, just lots of definitions for protocols, structs, modifiers, and such.

How property wrappers become structs

We asked to see **State**, so you should have been taken to this line:

```
@propertyWrapper public struct State<Value> : DynamicProperty {
```

That **@propertyWrapper** attribute is what makes this into **@State** for us to use.

Now look a few lines further down, and you should see this:

```
public var wrappedValue: Value { get nonmutating set }
```

That wrapped value is the actual value we're trying to store, such as a string. What this generated interface is telling us is that the property can be read (**get**), and written (**set**), but that when we set the value it won't actually change the struct itself. Behind the scenes, it sends that value off to SwiftUI for storage in a place where it can be modified freely, so it's true that the struct itself never changes.

Now you know all that, let's circle back to our problematic code:

```
@State private var blurAmount = 0.0 {  
    didSet {  
        print("New value is \(blurAmount)")  
    }  
}
```

On the surface, that states “when **blurAmount** changes, print out its new value.” However, because **@State** actually wraps its contents, what it’s *actually* saying is that when the **State** struct that wraps **blurAmount** changes, print out the new blur amount.

Still with me? Now let’s go a stage further: you’ve just seen how **State** wraps its value using a non-mutating setter, which means neither **blurAmount** or the **State** struct wrapping it are changing – our binding is directly changing the internally stored value, which means the property observer is never being triggered.

So, changing the property directly using a button works fine, because it goes through the

Project 13: Instafilter

nonmutating setter and triggers the **didSet** observer, but using a binding *doesn't* because it bypasses the setter and adjusts the value directly.

How then can we solve this – how can we ensure some code is run whenever a binding is changed, no matter how that change happens? Well, there's a modifier just for that purpose...

Responding to state changes using onChange()

Because of the way SwiftUI sends binding updates to property wrappers, assigning property observers used with property wrappers often won't work, which means this kind of code won't print anything even as the blur radius changes:

```
struct ContentView: View {
    @State private var blurAmount: CGFloat = 0.0 {
        didSet {
            print("New value is \(blurAmount)")
        }
    }

    var body: some View {
        VStack {
            Text("Hello, World!")
                .blur(radius: blurAmount)

            Slider(value: $blurAmount, in: 0...20)
        }
    }
}
```

To fix this we need to use the **onChange()** modifier, which tells SwiftUI to run a function of our choosing when a particular value changes. SwiftUI will automatically pass in the new value to whatever function you attach, or you can just read the original property if you prefer:

```
struct ContentView: View {
    @State private var blurAmount = 0.0

    var body: some View {

```

Project 13: Instafilter

```
    VStack {
        Text("Hello, World!")
            .blur(radius: blurAmount)

        Slider(value: $blurAmount, in: 0...20)
            .onChange(of: blurAmount) { newValue in
                print("New value is \(newValue)")
            }
    }
}
```

Now that code will correctly print out values as the slider changes, because **onChange()** is watching it. Notice how most other things have stayed the same: we still use **@State private var** to declare the **blurAmount** property, and we still use **blur(radius: blurAmount)** as the modifier for our text view.

What all this means is that you can do whatever you want inside the **onChange()** function: you can call methods, run an algorithm to figure out how to apply the change, or whatever else you might need.

Showing multiple options with confirmationDialog()

SwiftUI gives us **alert()** for presenting important announcements with one or two buttons, and **sheet()** for presenting whole views on top of the current view, but it also gives us **confirmationDialog()**: an alternative to **alert()** that lets us add many buttons.

Visually alerts and confirmation dialogs are very different: on iPhones, alerts appear in the center of the screen and must actively be dismissed by choosing a button, whereas confirmation dialogs slide up from the bottom, can contain multiple buttons, and can be dismissed by tapping on Cancel or by tapping outside of the options.

Although they *look* very different, confirmation dialogs and alerts are created almost identically:

- Both are created by attaching a modifier to our view hierarchy – **alert()** for alerts and **confirmationDialog()** for confirmation dialogs.
- Both get shown automatically by SwiftUI when a condition is true.
- Both can be filled with buttons to take various actions.
- Both can have a second closure attached to provide an extra message.

To demonstrate confirmation dialogs being used, we first need a basic view that toggles some sort of condition. For example, this shows some text, and tapping the text changes a Boolean:

```
struct ContentView: View {  
    @State private var showingConfirmation = false  
    @State private var backgroundColor = Color.white  
  
    var body: some View {  
        Text("Hello, World!")  
            .frame(width: 300, height: 300)  
            .background(backgroundColor)
```

Project 13: Instafilter

```
.onTapGesture {
    showingConfirmation = true
}
}

}
```

Now for the important part: we need to add another modifier to the text, creating and showing a confirmation dialog when we're ready.

Just like **alert()**, we have a **confirmationDialog()** modifier that accepts two parameters: a binding that decides whether the dialog is currently presented or not, and a closure that provides the buttons that should be shown – usually provided as a trailing closure.

We provide our confirmation dialog with a title and optionally also a message, then an array of buttons. These are stacked up vertically on the screen in the order you provide, and it's generally a good idea to include a cancel button at the end – yes, you *can* cancel by tapping elsewhere on the screen, but it's much better to give users the explicit option.

So, add this modifier to your text view:

```
.confirmationDialog("Change background", isPresented:
$showingConfirmation) {
    Button("Red") { backgroundColor = .red }
    Button("Green") { backgroundColor = .green }
    Button("Blue") { backgroundColor = .blue }
    Button("Cancel", role: .cancel) { }
} message: {
    Text("Select a new color")
}
```

When you run the app, you should find that tapping the text causes the confirmation dialog to slide over, and tapping its options should cause the text's background color to change.

Integrating Core Image with SwiftUI

Just like Core Data is Apple's built-in framework for manipulating data, Core Image is their framework for manipulating images. This isn't *drawing*, or at least for the most part it isn't drawing, but instead it's about changing existing images: applying sharpening, blurs, vignettes, pixellation, and more. If you ever used all the various photo effects available in Apple's Photo Booth app, that should give you a good idea of what Core Image is good for!

However, Core Image doesn't integrate into SwiftUI very well. In fact, I wouldn't even say it integrates into UIKit very well – Apple did some work to provide helpers, but it still takes quite a bit of thinking. Stick with me, though: the results are quite brilliant once you understand how it all works, and you'll find it opens up a whole range of functionality for your apps in the future.

First, we're going to put in some code to give us a basic image. I'm going to structure this in a slightly odd way, but it will make sense once we mix in Core Image: we're going to create the **Image** view as an optional `@State` property, force it to be the same width as the screen, then add an `onAppear()` modifier to actually load the image.

Add an example image to your asset catalog, then modify your **ContentView** struct to this:

```
struct ContentView: View {
    @State private var image: Image?

    var body: some View {
        VStack {
            image?
                .resizable()
                .scaledToFit()
        }
        .onAppear(perform: loadImage)
    }
}
```

Project 13: Instafilter

```
}

func loadImage() {
    image = Image("Example")
}

}
```

First, notice how smoothly SwiftUI handles optional views – it just works! However, notice how I attached the `onAppear()` modifier to a `VStack` around the image, because if the optional image is `nil` then it won't trigger the `onAppear()` function.

Anyway, when that code runs it should show the example image you added, neatly scaled to fit the screen.

Now for the complex part: what actually *is* an **Image**? As you know, it's a *view*, which means it's something we can position and size inside our SwiftUI view hierarchy. It also handles loading images from our asset catalog and SF Symbols, and it's capable of loading from a handful of other sources too. However, ultimately it is something that gets displayed – we can't write its contents to disk or otherwise transform them beyond applying a few simple SwiftUI filters.

If we want to use Core Image, SwiftUI's **Image** view is a great end point, but it's not useful to use elsewhere. That is, if we want to create images dynamically, apply Core Image filters, save them to the user's photo library, and so on, then SwiftUI's images aren't up to the job.

Apple gives us three other image types to work with, and cunningly we need to use all three if we want to work with Core Image. They might sound similar, but there is some subtle distinction between them, and it's important that you use them correctly if you want to get anything meaningful out of Core Image.

Apart from SwiftUI's **Image** view, the three other image types are:

- **UIImage**, which comes from UIKit. This is an extremely powerful image type capable of working with a variety of image types, including bitmaps (like PNG), vectors (like SVG),

and even sequences that form an animation. **UIImage** is the standard image type for UIKit, and of the three it's closest to SwiftUI's **Image** type.

- **CGImage**, which comes from Core Graphics. This is a simpler image type that is really just a two-dimensional array of pixels.
- **CIIImage**, which comes from Core Image. This stores all the information required to produce an image but doesn't actually turn that into pixels unless it's asked to. Apple calls **CIIImage** "an image recipe" rather than an actual image.

There is some interoperability between the various image types:

- We can create a **UIImage** from a **CGImage**, and create a **CGImage** from a **UIImage**.
- We can create a **CIIImage** from a **UIImage** and from a **CGImage**, and can create a **CGImage** from a **CIIImage**.
- We can create a SwiftUI **Image** from both a **UIImage** and a **CGImage**.

I know, I know: it's confusing, but hopefully once you see the code you feel better. What matters is that these image types are pure *data* – we can't place them into a SwiftUI view hierarchy, but we can manipulate them freely then present the results in a SwiftUI **Image**.

We're going to change **loadImage()** so that it creates a **UIImage** from our example image, then manipulate it using Core Image. More specifically, we'll start with two tasks:

1. We need to load our example image into a **UIImage**, which has an initializer called **UIImage(named:)** to load images from our asset catalog. It returns an optional **UIImage** because we might have specified an image that doesn't exist.
2. We'll convert that into a **CIIImage**, which is what Core Image wants to work with.

So, start by replacing your current **loadImage()** implementation with this:

```
func loadImage() {
    guard let inputImage = UIImage(named: "Example") else
    { return }
    let beginImage = CIIImage(image: inputImage)
```

Project 13: Instafilter

```
// more code to come  
}
```

The next step will be to create a Core Image context and a Core Image filter. Filters are the things that do the actual work of transforming image data somehow, such as blurring it, sharpening it, adjusting the colors, and so on, and contexts handle converting that processed data into a **CGImage** we can work with.

Both of these data types come from Core Image, so you'll need to add two imports to make them available to us. So please start by adding these near the top of ContentView.swift:

```
import CoreImage  
import CoreImage.CIFilterBuiltins
```

Next we'll create the context and filter. For this example we're going to use a sepia tone filter, which applies a brown tone that makes a photo look like it was taken a long time ago.

So, replace the **// more code to come** comment with this:

```
let context = CIContext()  
let currentFilter = CIFilter.sepiaTone()
```

We can now customize our filter to change the way it works. Sepia is a simple filter, so it only has two interesting properties: **inputImage** is the image we want to change, and **intensity** is how strongly the sepia effect should be applied, specified in the range 0 (original image) and 1 (full sepia).

So, add these two lines of code below the previous two:

```
currentFilter.inputImage = beginImage  
currentFilter.intensity = 1
```

None of this is terribly hard, but here's where that changes: we need to convert the output from

our filter to a SwiftUI **Image** that we can display in our view. This is where we need to lean on all four image types at once, because the easiest thing to do is:

- Read the output image from our filter, which will be a **CIIImage**. This might fail, so it returns an optional.
- Ask our context to create a **CGImage** from that output image. This also might fail, so again it returns an optional.
- Convert that **CGImage** into a **UIImage**.
- Convert that **UIImage** into a SwiftUI **Image**.

You *can* go direct from a **CGImage** to a SwiftUI **Image** but it requires extra parameters and it just adds even more complexity!

Here's the final code for **loadImage()**:

```
// get a CIIImage from our filter or exit if that fails
guard let outputImage = currentFilter.outputImage else
{ return }

// attempt to get a CGImage from our CIIImage
if let cgimg = context.createCGImage(outputImage, from:
outputImage.extent) {
    // convert that to a UIImage
    let uiImage = UIImage(cgImage: cgimg)

    // and convert that to a SwiftUI image
    image = Image(uiImage: uiImage)
}
```

If you run the app again you should see your example image now has a sepia effect applied, all thanks to Core Image.

Now, you might well think that was a heck of a lot of work just to get a fairly simple result, but

Project 13: Instafilter

now that you have all the basics of Core Image in place it's relatively easy to switch to different filters.

That being said, Core Image is a little bit... well... let's say "creative". It was introduced way back in iOS 5.0, and by that point Swift was already being developed inside Apple, but you really wouldn't know it – for the longest time its API was the least Swifty thing you could imagine, and although Apple has slowly chipped away at its crust sometimes you have no choice but to dig into its underbelly.

First, let's look at the modern API – we could replace our sepia tone with a pixellation filter like this:

```
let currentFilter = CIFilter.pixellate()
currentFilter.inputImage = beginImage
currentFilter.scale = 100
```

When that runs you'll see our image looks pixellated. A scale of 100 should mean the pixels are 100 points across, but because my image is so big the pixels are relatively small.

Now let's try a crystal effect like this:

```
let currentFilter = CIFilter.crystallize()
currentFilter.inputImage = beginImage
currentFilter.radius = 200
```

Or we could add a twirl distortion filter like this:

```
let currentFilter = CIFilter.twirlDistortion()
currentFilter.inputImage = beginImage
currentFilter.radius = 1000
currentFilter.center = CGPoint(x: inputImage.size.width / 2, y:
inputImage.size.height / 2)
```

So, there's a lot we can do using only the modern API. But for this project we're going to use

the older API for setting values such as **radius** and **scale** because it lets us set values dynamically – we can literally ask the current filter what values it supports, then send them on in.

Here's how that looks:

```
let currentFilter = CIFilter.twirlDistortion()
currentFilter.inputImage = beginImage

let amount = 1.0

let inputKeys = currentFilter.inputKeys

if inputKeys.contains(kCIIInputIntensityKey) {
    currentFilter.setValue(amount, forKey:
    kCIIInputIntensityKey)
}
if inputKeys.contains(kCIIInputRadiusKey)
{ currentFilter.setValue(amount * 200, forKey:
kCIIInputRadiusKey)
}
if inputKeys.contains(kCIIInputScaleKey)
{ currentFilter.setValue(amount * 10, forKey:
kCIIInputScaleKey) }
```

With that in place, you can now change the twirl distortion to any other filter and the code will carry on working – each of the adjustment values are sent in only if they are supported.

Notice how that relies on setting values for keys, which might remind you of the way **User Defaults** works. In fact, all those **kCIIInput** keys are all implemented as strings behind the scenes, so it's even more similar than you might have realized!

If you're implementing precise Core Image adjustments you should definitely be using the newer API that uses exact property names and types, but in this project the older API comes in useful because it lets us send in adjustments regardless of what filter is actually used.

Wrapping a UIViewController in a SwiftUI view

SwiftUI is a really fantastic framework for building apps, but right now it's far from complete – there are many things it just can't do, so you need to learn to talk to UIKit if you want to add more advanced functionality. Sometimes this will be to integrate existing code you wrote for UIKit (for example, if you work for a company with an existing UIKit app), but other times it will be because UIKit and Apple's other frameworks provide us with useful code we want to show inside a SwiftUI layout.

In this project we're going to ask users to import a picture from their photo library. Apple's APIs come with dedicated code for doing just this, but that hasn't been ported to SwiftUI and so we need to write that bridge ourself. Instead, it's built into a separate framework called PhotosUI, which was designed to work with UIKit and so requires us to look at the way UIKit works.

Before we write the code, there are three things you need to know, all of which are a bit like UIKit 101 but if you've only used SwiftUI they will be new to you:

1. UIKit has a class called **UIView**, which is the parent class of all views in the layouts. So, labels, buttons, text fields, sliders, and so on – those are all views.
2. UIKit has a class called **UIViewController**, which is designed to hold all the code to bring views to life. Just like **UIView**, **UIViewController** has many subclasses that do different kinds of work.
3. UIKit uses a design pattern called *delegation* to decide where work happens. So, when it came to deciding how to respond to a text field changing, we'd create a custom class with our functionality and make that the delegate of our text field.

Wrapping a UIViewController in a SwiftUI view

All this matters because asking the user to select a photo from their library uses a view controller called **PHPickerViewController**, and the delegate protocol **PHPickerViewControllerDelegate**. SwiftUI can't use these two directly, so we need to wrap them.

We're going to start simple and work our way up. Wrapping a UIKit view controller requires us to create a struct that conforms to the **UIViewControllerRepresentable** protocol.

So, press Cmd+N to make a new file, choose Swift File, and name it ImagePicker.swift. Add **import PhotosUI** and **import SwiftUI** to the top of the new file, then give it this code:

```
struct ImagePicker: UIViewControllerRepresentable {  
}
```

That protocol builds on **View**, which means the struct we're defining can be used inside a SwiftUI view hierarchy, however we don't provide a **body** property because the view's body is the view controller itself – it just shows whatever UIKit sends back.

Conforming to **UIViewControllerRepresentable** does require us to fill in that struct with two methods: one called **makeUIViewController()**, which is responsible for creating the initial view controller, and another called **updateUIViewController()**, which is designed to let us update the view controller when some SwiftUI state changes.

These methods have really precise signatures, so I'm going to show you a neat shortcut. The reason the methods are long is because SwiftUI needs to know what type of view controller our struct is wrapping, so if we just straight up tell Swift that type Xcode will help us do the rest.

Add this code to the struct now:

```
typealias UIViewControllerType = PHPickerViewController
```

That isn't enough code to compile correctly, but when Xcode shows an error saying "Type

Project 13: Instafilter

ImagePicker does not conform to protocol `UIViewControllerRepresentable`”, please click the red and white circle to the left of the error and select “Fix”. This will make Xcode write the two methods we actually need, and in fact those methods are actually enough for Swift to figure out the view controller type so you can delete the `typealias` line.

You should have a struct like this:

```
struct ImagePicker: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) ->
    PHPickerViewController {
        code
    }

    func updateUIViewController(_ uiViewController:
    PHPickerViewController, context: Context) {
        code
    }
}
```

We aren’t going to be using `updateUIViewController()`, so you can just delete the “code” line from there so that the method is empty.

However, the `makeUIViewController()` method is important, so please replace its existing “code” line with this:

```
var config = PHPickerConfiguration()
config.filter = .images

let picker = PHPickerViewController(configuration: config)
return picker
```

That creates a new photo picker configuration, asking it to provide us only images, then uses that to create and return a `PHPickerViewController` that does the actual work of selecting an

image.

We'll add some more code to that shortly, but that's actually all we need to wrap a basic view controller.

Our **ImagePicker** struct is a valid SwiftUI view, which means we can now show it in a sheet just like any other SwiftUI view. This particular struct is designed to show an image, so we need an optional **Image** view to hold the selected image, plus some state that determines whether the sheet is showing or not.

Replace your current **ContentView** struct with this:

```
struct ContentView: View {
    @State private var image: Image?
    @State private var showingImagePicker = false

    var body: some View {
        VStack {
            image?
                .resizable()
                .scaledToFit()

            Button("Select Image") {
                showingImagePicker = true
            }
        }
        .sheet(isPresented: $showingImagePicker) {
            ImagePicker()
        }
    }
}
```

Go ahead and run that, either in the simulator or on your real device. When you tap the button

Project 13: Instafilter

the default iOS image picker should slide up letting you browse through all your photos.

However, nothing will happen when an image is selected, and the Cancel button won't do anything either. You see, even though we've created and presented a valid **PHPickerViewController**, we haven't actually told it how to respond to user interactions.

To make *that* happens requires a wholly new concept: *coordinators*.

Using coordinators to manage SwiftUI view controllers

Previously we looked at how we can use **UIViewControllerRepresentable** to wrap a UIKit view controller so that it can be used inside SwiftUI, in particular focusing on **PHPickerViewController**. However, we hit a problem: although we could show the image picker, we weren't able to respond to the user selecting an image or pressing cancel.

SwiftUI's solution to this is called *coordinators*, which is a bit confusing for folks coming from a UIKit background because there we had a design pattern also called coordinators that performed an entirely different role. To be clear, SwiftUI's coordinators are nothing like the coordinator pattern many developers used with UIKit, so if you've used that pattern previously please jettison it from your brain to avoid confusion!

SwiftUI's coordinators are designed to act as delegates for UIKit view controllers. Remember, “delegates” are objects that respond to events that occur elsewhere. For example, UIKit lets us attach a delegate object to its text field view, and that delegate will be notified when the user types anything, when they press return, and so on. This meant that UIKit developers could modify the way their text field behaved without having to create a custom text field type of their own.

Using coordinators in SwiftUI requires you to learn a little about the way UIKit works, which is no surprise given that we're literally integrating UIKit's view controllers. So, to demonstrate this we're going to upgrade our **ImagePicker** view so that it can report back when the user selects an image or presses Cancel.

As a reminder, here's the code we have right now:

```
struct ImagePicker: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) ->
        PHPickerViewController {
        var config = PHPickerConfiguration()
        config.filter = .images
```

Project 13: Instafilter

```
let picker = PHPickerViewController(configuration: config)
return picker
}

func updateUIViewController(_ uiViewController:
PHPickerViewController, context: Context) {

}
}
```

We're going to take it step by step, because there's a lot to take in here – don't feel bad if it takes you some time to understand, because coordinators really aren't easy the first time you encounter them.

First, add this nested class inside the **ImagePicker** struct:

```
class Coordinator {
```

Yes, it needs to be a class as you'll see in a moment. It *doesn't* need to be a nested class, although it's a good idea because it neatly encapsulates the functionality – without a nested class it would be confusing if you had lots of view controllers and coordinators all mixed up.

Even though that class is inside a **UIViewControllerRepresentable** struct, SwiftUI won't automatically use it for the view's coordinator. Instead, we need to add a new method called **makeCoordinator()**, which SwiftUI will automatically call if we implement it. All this needs to do is create and configure an instance of our **Coordinator** class, then send it back.

Right now our **Coordinator** class doesn't do anything special, so we can just send one back by adding this method to the **ImagePicker** struct:

```
func makeCoordinator() -> Coordinator {
    Coordinator()
}
```

```
}
```

What we've done so far is create an **ImagePicker** struct that knows how to create a **PHPickerViewController**, and now we just told **ImagePicker** that it should have a coordinator to handle communication from that **PHPickerViewController**.

The next step is to tell the **PHPickerViewController** that when something happens it should tell our coordinator. This takes just one line of code in **makeUIViewController()**, so add this directly before the **return picker** line:

```
picker.delegate = context.coordinator
```

That code won't compile, but before we fix it I want to spend just a moment digging in to what just happened.

We don't call **makeCoordinator()** ourselves; SwiftUI calls it automatically when an instance of **ImagePicker** is created. Even better, SwiftUI automatically associates the coordinator it created with our **ImagePicker** struct, which means when it calls **makeUIViewController()** and **updateUIViewController()** it will automatically pass that coordinator object to us.

So, the line of code we just wrote tells Swift to use the coordinator that just got made as the delegate for the **PHPickerViewController**. This means any time something happens inside the photo picker controller – i.e., when the user selects an image or presses Cancel – it will report that action to our coordinator.

The reason our code doesn't compile is that Swift is checking that our coordinator class is *capable* of acting as a delegate for **PHPickerViewController**, finding that it isn't, and so is refusing to build our code any further. To fix this we need to modify our **Coordinator** class from this:

```
class Coordinator {
```

To this:

Project 13: Instafilter

```
class Coordinator: NSObject, PHPickerViewControllerDelegate {
```

That does three things:

1. It makes the class inherit from **NSObject**, which is the parent class for almost everything in UIKit. **NSObject** allows Objective-C to ask the object what functionality it supports at runtime, which means the photo picker can say things like “hey, the user selected an image, what do you want to do?”
2. It makes the class conform to the **PHPickerViewControllerDelegate** protocol, which is what adds functionality for detecting when the user selects an image. (**NSObject** lets Objective-C *check* for the functionality; this protocol is what actually provides it.)
3. It stops our code from compiling, because we’ve said that class conforms to **PHPickerViewControllerDelegate** but we haven’t implemented the one method required by that protocol.

Still, at least now you can see why we needed to use a class for **Coordinator**: we need to inherit from **NSObject** so that Objective-C can query our coordinator to see what functionality it supports.

At this point we have an **ImagePicker** struct that wraps a **PHPickerViewController**, and we’ve configured that image picker controller to talk to our **Coordinator** class when something interesting happens.

The last step is to implement the one required method of the **PHPickerViewControllerDelegate** protocol, which will be called when the user has finished making their selection. That might mean we have an image, or that the user pressed cancel, so we need to respond appropriately.

If we put UIKit to one side for a second and think in pure functionality, what we want is for our **ImagePicker** to report back that image to whatever used the picker in the first place. We’re presenting **ImagePicker** inside a sheet in our **ContentView** struct, so we want that to be given whatever image was selected, then dismiss the sheet.

Using coordinators to manage SwiftUI view controllers

What we need here is SwiftUI's **@Binding** property wrapper, which allows us to create a binding from **ImagePicker** up to whatever created it. This means we can set the binding value in our image picker and have it actually update a value being stored somewhere else – in **ContentView**, for example.

So, add this property to **ImagePicker**:

```
@Binding var image: UIImage?
```

Now, we just added that property to **ImagePicker**, but we need to *access* it inside our **Coordinator** class because that's the one that will be informed when an image was selected.

Rather than just pass the data down one level, a better idea is to tell the coordinator what its parent is, so it can modify values there directly. That means adding an **ImagePicker** property and associated initializer to the **Coordinator** class, like this:

```
var parent: ImagePicker

init(_ parent: ImagePicker) {
    self.parent = parent
}
```

And now we can modify **makeCoordinator()** so that it passes the **ImagePicker** struct into the coordinator, like this:

```
func makeCoordinator() -> Coordinator {
    Coordinator(self)
}
```

At this point your entire **ImagePicker** struct should look like this:

```
struct ImagePicker: UIViewControllerRepresentable {
    class Coordinator: NSObject, PHPickerViewControllerDelegate {
        var parent: ImagePicker
```

Project 13: Instafilter

```
init(_ parent: ImagePicker) {
    self.parent = parent
}

@Binding var image: UIImage?

func makeUIViewController(context: Context) ->
PHPickerViewController {
    var config = PHPickerConfiguration(photoLibrary:
PHPPhotoLibrary.shared())
    config.filter = .images

    let picker = PHPickerViewController(configuration: config)
    picker.delegate = context.coordinator
    return picker
}

func updateUIViewController(_ uiViewController:
PHPickerViewController, context: Context) {

}

func makeCoordinator() -> Coordinator {
    Coordinator(self)
}
}
```

At long last we're ready to actually read the response from our **PHPickerViewController**, which is done by implementing a method with a very specific name. Swift will look for this method in our **Coordinator** class, as it's the delegate of the image picker, so make sure and

add it there.

The method name is long and needs to be exactly right in order for UIKit to find it, but helpfully Xcode can help us out with autocomplete. So, click on the red hexagon on the error line, then click “Fix” to add this method stub:

```
func picker(_ picker: PHPickerViewController, didFinishPicking
results: [PHPickerResult]) {
    code
}
```

That method receives two objects we care about: the picker view controller that the user was interacting with, plus an array of the users selections because it’s possible to let the user select multiple images at once.

It’s our job to do three things:

1. Tell the picker to dismiss itself.
2. Exit if the user made no selection – if they tapped Cancel.
3. Otherwise, see if the user’s results includes a **UIImage** we can actually load, and if so place it into the **parent.image** property.

So, replace the “code” placeholder with this:

```
// Tell the picker to go away
picker.dismiss(animated: true)

// Exit if no selection was made
guard let provider = results.first?.itemProvider else
{ return }

// If this has an image we can use, use it
if provider.canLoadObject(ofClass: UIImage.self) {
    provider.loadObject(ofClass: UIImage.self) { image, _ in
```

Project 13: Instafilter

```
    self.parent.image = image as? UIImage
}
}
```

Notice how we need the typecast for **UIImage** – that's because the data we're provided could in theory be anything. Yes, I know we specifically asked for photos, but **PHPickerViewControllerDelegate** calls this same method for any kind of media, which is why we need to be careful.

At this point I bet you're *really* missing the beautiful simplicity of SwiftUI, so you'll be glad to know that we're finally done with the **ImagePicker** struct – it does everything we need now.

So, at last we can return to `ContentView.swift`. Here's how we left it from earlier:

```
struct ContentView: View {
    @State private var image: Image?
    @State private var showingImagePicker = false

    var body: some View {
        VStack {
            image?
                .resizable()
                .scaledToFit()

            Button("Select Image") {
                showingImagePicker = true
            }
        }
        .sheet(isPresented: $showingImagePicker) {
            ImagePicker()
        }
    }
}
```

To integrate our **ImagePicker** view into that we need to start by adding another **@State** image property that can be passed into the picker:

```
@State private var inputImage: UIImage?
```

We can now change our **sheet()** modifier to pass that property into our image picker, so it will be updated when the image is selected:

```
ImagePicker(image: $inputImage)
```

Next, we need a method we can call when that property changes. Remember, we can't use a plain property observer here because Swift will ignore changes to the binding, so instead we'll write a method that checks whether **inputImage** has a value, and if it does uses it to assign a new **Image** view to the **image** property.

Add this method to **ContentView** now:

```
func loadImage() {
    guard let inputImage = inputImage else { return }
    image = Image(uiImage: inputImage)
}
```

And now we can use an **onChange()** modifier to call **loadImage()** whenever a new image is chosen:

```
.onChange(of: inputImage) { _ in loadImage() }
```

And we're done! Go ahead and run the app and try it out – you should be able to tap the button, browse through your photo library, and select a picture. When that happens the photo picker should disappear, and your selected image will be shown below.

I realize at this point you're probably sick of UIKit and coordinators, but before we move on I want to sum up the complete process:

Project 13: Instafilter

- We created a SwiftUI view that conforms to **UIViewControllerRepresentable**.
- We gave it a **makeUIViewController()** method that created some sort of **UIViewController**, which in our example was a **PHPickerViewController**.
- We added a nested **Coordinator** class to act as a bridge between the UIKit view controller and our SwiftUI view.
- We gave that coordinator a **didFinishPicking** method, which will be triggered by iOS when an image was selected.
- Finally, we gave our **ImagePicker** an **@Binding** property so that it can send changes back to a parent view.

For what it's worth, after you've used coordinators once, the second and subsequent times are easier, but I wouldn't blame you if you found the whole system quite baffling for now.

Don't worry too much – we'll be coming back to this again soon, so you'll have more than enough time to practice. It also means you shouldn't delete your `ImagePicker.swift` file, because that's another useful component you'll use in this project and in others you make.

How to save images to the user's photo library

Before we're done with the techniques for this project, there's one last piece of UIKit joy we need to tackle: once we've processed the user's image we'll get a **UIImage** back, but we need a way to save that processed image to the user's photo library. This uses a UIKit function called **UIImageWriteToSavedPhotosAlbum()**, which in its simplest form is trivial to use, but in order to make it work *usefully* you need to wade back into UIKit. At the very least it will make you really appreciate how much better SwiftUI is!

Before we write any code, we need to do something new: we need to add a configuration option for our project. Every project we build has a whole bunch of these baked right in, describing which interface orientations we support, the version number of our app, and other fixed pieces of data. This *isn't code*: these options must all be declared ahead of time, in a separate file, so the system can read them without having to run our app.

These options all live in a particular place in Xcode, and it's bizarrely hard to find unless you know what you're doing:

1. In the Project Navigator, select the top item in the tree. It will have your project name, Instafilter.
2. You'll see Instafilter listed under both PROJECT and TARGETS. Please select it under TARGETS.
3. Now you'll see a bunch of tabs across the top, including General, Signing & Capabilities, and more – select Info from there.

This is where you can add a whole range of configuration options for your project, but right now there's one specific option we need. You see, writing to the photo library is a protected operation, which means we can't do it without explicit permission from the user. iOS will take care of asking for permission and checking the response, but we need to provide a short string explaining to users why we want to write images in the first place.

Project 13: Instafilter

To add your permission string, right-click on any of the existing options then choose Add Row. You'll see a dropdown list of options to choose from – I'd like you to scroll down and select “Privacy - Photo Library Additions Usage Description”. For the value on its right, please enter the text “We want to save the filtered photo.”

With that done, we can now use the **UIImageWriteToSavedPhotosAlbum()** method to write out a picture. We already have this **loadImage()** method from our previous work:

```
func loadImage() {  
    guard let inputImage = inputImage else { return }  
    image = Image(uiImage: inputImage)  
}
```

We could modify that so it immediately saves the image that got loaded, effectively creating a duplicate. Add this line to the end of the method:

```
UIImageWriteToSavedPhotosAlbum(inputImage, nil, nil, nil)
```

And that's it – every time you import an image, our app will save it back to the photo library. The first time you try it, iOS will automatically prompt the user for permission to write the photo and show the string we added to the configuration options.

Now, you might look at that and think “that was easy!” And you'd be right. But the reason it's easy is because we did the least possible work: we provided the image to save as the first parameter to **UIImageWriteToSavedPhotosAlbum()**, then provided **nil** as the other three.

Those **nil** parameters *matter*, or at least the first two do: they tell Swift what method should be called when saving completes, which in turn will tell us whether the save operation succeeded or failed. If you don't care about that then you're done – passing **nil** for all three is fine. But remember: users can deny access to their photo library, so if you don't catch the save error they'll wonder why your app isn't working properly.

The reason it takes UIKit *two* parameters to know which function to call is because this code is *old* – way older than Swift, and in fact so old it even pre-dates Objective-C's equivalent of

closures. So instead, this uses a completely different way of referring to functions: in place of the first **nil** we should point to an object, and in place of the second **nil** we should point to the name of the method that should be called.

If that sounds bad, I'm afraid you only know half the story. You see, both of those two parameters have their own complexities:

- The object we provide must be a class, and it must inherit from **NSObject**. This means we can't point to a SwiftUI view struct.
- The method is provided as a method *name*, not an actual method. This method name was used by Objective-C to find the actual code at runtime, which could then be run. That method needs to have a specific signature (list of parameters) otherwise our code just won't work.

But wait: there's more! For performance reasons, Swift prefers not to generate code in a way that Objective-C can read – that whole “look up methods at runtime” thing was really neat, but also really slow. And so, when it comes to writing the method name we need to do two things:

1. Mark the method using a special compiler directive called **#selector**, which asks Swift to make sure the method name exists where we say it does.
2. Add an attribute called **@objc** to the method, which tells Swift to generate code that can be read by Objective-C.

You know, I wrote UIKit code for over a decade before I switched to SwiftUI, and already writing out all this explanation makes this old API seem like a crime against humanity. Sadly it is what it is, and we're stuck with it.

Before I show you the code, I want to mention the fourth parameter. So, the first one is the image to save, the second one is an object that should be notified about the result of the save, the third one is the method on the object that should be run, and then there's the fourth one. We aren't going to be using it here, but you need to be *aware* of what it does: we can provide any sort of data here, and it will be passed *back* to us when our completion method is called.

Project 13: Instafilter

This is what UIKit calls “context”, and it helps you identify one image save operation from another. You can provide literally anything you want here, so UIKit uses the most hands-off type you can imagine: a raw chunk of memory that Swift makes no guarantees about whatsoever. This has its own special type name in Swift: **UnsafeRawPointer**. Honestly, if it weren’t for the fact that we *had* to use it here I simply wouldn’t even tell you it existed, because it’s not really useful at this point in your app development career.

Anyway, that’s more than enough talk. Before you decide to throw this project away and go straight to the next one, let’s get this over and done with.

As I’ve said, to write an image to the photo library and read the response, we need some sort of class that inherits from **NSObject**. Inside there we need a method with a precise signature that’s marked with **@objc**, and we can then call that from **UIImageWriteToSavedPhotosAlbum()**.

Putting all that together, please add this class somewhere outside of **ContentView**:

```
class ImageSaver: NSObject {
    func writeToPhotoAlbum(image: UIImage) {
        UIImageWriteToSavedPhotosAlbum(image, self,
#selector(saveCompleted), nil)
    }

    @objc func saveCompleted(_ image: UIImage,
didFinishSavingWithError error: Error?, contextInfo:
UnsafeRawPointer) {
        print("Save finished!")
    }
}
```

With that in place we can now use it from SwiftUI, like this:

```
Button("Save Image") {
    guard let inputImage = inputImage else { return }
}
```

```
let imageSaver = ImageSaver()  
imageSaver.writeToPhotoAlbum(image: inputImage)  
}
```

If you run the code now, you should see the “Save finished!” message output as soon as you select an image, but you’ll also see the system prompt you for permission to write to the photo library.

Yes, that is remarkably little code given how much explanation it needed, but on the bright side that completes the overview for this project so at long (long, long!) last we can get into the actual implementation.

Please go ahead and put your project back to its default state so we have a clean slate to work from, but I’d like you to keep **ImagePicker** and **ImageSaver** – both of those will be used later in this project, and they are also useful in other projects you might create in the future.

Building our basic UI

The first step in our project is to build the basic user interface, which for this app will be:

1. A **NavigationView** so we can show our app's name at the top.
2. A large gray box saying “Tap to select a picture”, over which we'll place their imported picture.
3. An “Intensity” slider that will affect how strongly we apply our Core Image filters, stored as a value from 0.0 to 1.0.
4. A “Save” button to write out the modified image to the user's photo library.

Initially the user won't have selected an image, so we'll represent that using an **@State** optional image property.

First add these two properties to **ContentView**:

```
@State private var image: Image?  
@State private var filterIntensity = 0.5
```

Now modify the contents of its **body** property to this:

```
NavigationView {  
    VStack {  
        ZStack {  
            Rectangle()  
                .fill(.secondary)  
  
            Text("Tap to select a picture")  
                .foregroundColor(.white)  
                .font(.headline)  
  
            image?  
                .resizable()  
        }  
    }  
}
```

```
        .scaledToFit()
    }
    .onTapGesture {
        // select an image
    }

HStack {
    Text("Intensity")
    Slider(value: $filterIntensity)
}
.padding(.vertical)

HStack {
    Button("Change Filter") {
        // change filter
    }

Spacer()

Button("Save") {
    // save the picture
}
}

.padding([.horizontal, .bottom])
.navigationTitle("Instafilter")
}
```

I love being able to place optional views right inside a SwiftUI layout, and I think it works particularly well with **ZStack** because the text below our picture will automatically be obscured when a picture has been loaded by the user.

Project 13: Instafilter

Now, as our code was fairly easy here, I want to just briefly explore what it looks like to clean up our **body** property a little – we have lots of layout code in there, but as you can see we also have a couple of button closures in there too.

For very small pieces of code I'm usually happy enough to have button actions specified as closures, but that Save button is going to have quite a few lines in there when we're done so I would suggest spinning it out into its own function.

Right now that just means adding an empty **save()** method to **ContentView**, like this:

```
func save() {  
}
```

We would then call that from the button like so:

```
Button("Save", action: save)
```

When you're learning it's very common to write button actions and similar directly inside your views, but once you get onto real projects it's a good idea to spend extra time keeping your code cleaned up – it makes your life easier in the long term, trust me!

I'll be adding more little cleanup tips like this going forward, so as you start to approach the end of the course you feel increasingly confident that your code is in good shape.

Importing an image into SwiftUI using PHPickerViewController

In order to bring this project to life, we need to let the user select a photo from their library, then display it in **ContentView**. I've already shown you how this all works, so if you followed the introductory chapters you'll already have most of the code you need.

If you missed those chapters, it's not too late: create a new Swift file called `ImagePicker.swift`, then replace its code with this:

```
import PhotosUI
import SwiftUI

struct ImagePicker: UIViewControllerRepresentable {
    @Binding var image: UIImage?

    func makeUIViewController(context: Context) ->
        PHPickerViewController {
        var config = PHPickerConfiguration()
        config.filter = .images
        let picker = PHPickerViewController(configuration: config)
        picker.delegate = context.coordinator
        return picker
    }

    func updateUIViewController(_ uiViewController:
        PHPickerViewController, context: Context) {
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }
}
```

Project 13: Instafilter

```
}

class Coordinator: NSObject, PHPickerViewControllerDelegate {
    let parent: ImagePicker

    init(_ parent: ImagePicker) {
        self.parent = parent
    }

    func picker(_ picker: PHPickerViewController,
    didFinishPicking results: [PHPickerResult]) {
        picker.dismiss(animated: true)

        guard let provider = results.first?.itemProvider else
        { return }

        if provider.canLoadObject(ofClass: UIImage.self) {
            provider.loadObject(ofClass: UIImage.self) { image, _
            in
                self.parent.image = image as? UIImage
            }
        }
    }
}
```

That's all code we've looked at before, so I'm not going to re-explain what it all does. Instead, I want to head back to `ContentView.swift` so we can make use of it.

First we need an `@State` Boolean to track whether our image picker is being shown or not, so start by adding this to `ContentView`:

```
@State private var showingImagePicker = false
```

Importing an image into SwiftUI using PHPickerViewController

Second, we need to set that Boolean to true when the big gray rectangle is tapped, so replace the `// select an image` comment with this:

```
showingImagePicker = true
```

Third, we need a property that will store the image the user selected. We gave the **ImagePickerController** struct an `@Binding` property attached to a **UIImage**, which means when we create the image picker we need to pass in a **UIImage** for it to link to. When the `@Binding` property changes, the external value changes as well, which lets us read the value.

So, add this property to **ContentView**:

```
@State private var inputImage: UIImage?
```

Fourth, we need a method that will be called when the **ImagePickerController** view has been dismissed. For now this will just place the selected image directly into the UI, so please add this method to **ContentView** now:

```
func loadImage() {
    guard let inputImage = inputImage else { return }
    image = Image(uiImage: inputImage)
}
```

We can then call that whenever our `inputImage` value changes, by attaching an `onChange()` modifier somewhere in **ContentView** – it really doesn't matter where, but after `navigationTitle()` would seem sensible.

```
.onChange(of: inputImage) { _ in loadImage() }
```

And finally, we need to add a `sheet()` modifier somewhere in **ContentView**. This will use `showingImagePicker` as its condition, and present an **ImagePickerController** bound to `inputImage` as its contents.

Project 13: Instafilter

So, add this directly below the existing **navigationTitle()** modifier:

```
.sheet(isPresented: $showingImagePicker) {  
    ImagePicker(image: $inputImage)  
}
```

That completes all the steps required to wrap a UIKit view controller for use inside SwiftUI. We went over it a little faster this time but hopefully it still all made sense!

Go ahead and run the app again, and you should be able to tap the gray rectangle to import a picture, and when you've found one it will appear inside our UI.

Tip: The **ImagePicker** view we just made is completely reusable – you can put that Swift file to one side and use it on other projects easily. If you think about it, all the complexity of wrapping the view is contained inside `ImagePicker.swift`, which means if you *do* choose to use it elsewhere it's just a matter of showing a sheet and binding an image.

Basic image filtering using Core Image

Now that our project has an image the user selected, the next step is to let the user apply varying Core Image filters to it. To start with we're just going to work with a single filter, but shortly we'll extend that using a confirmation dialog.

If we want to use Core Image in our apps, we first need to add two imports to the top of `ContentView.swift`:

```
import CoreImage
import CoreImage.CIFilterBuiltins
```

Next we need both a context and a filter. A Core Image context is an object that's responsible for rendering a **CILImage** to a **CGImage**, or in more practical terms an object for converting the recipe for an image into an actual series of pixels we can work with.

Contexts are expensive to create, so if you intend to render many images it's a good idea to create a context once and keep it alive. As for the filter, we'll be using **CIFilter.sepiaTone()** as our default but because we'll make it flexible later we'll make the filter use **@State** so it can be changed.

So, add these two properties to **ContentView**:

```
@State private var currentFilter = CIFilter.sepiaTone()
let context = CIContext()
```

With those two in place we can now write a method that will process whatever image was imported – that means it will set our sepia filter's intensity based on the value in **filterIntensity**, read the output image back from the filter, ask our **CIContext** to render it, then place the result into our **image** property so it's visible on-screen.

```
func applyProcessing() {
```

Project 13: Instafilter

```
currentFilter.intensity = Float(filterIntensity)

guard let outputImage = currentFilter.outputImage else
{ return }

if let cgimg = context.createCGImage(outputImage, from:
outputImage.extent) {
    let uiImage = UIImage(cgImage: cgimg)
    image = Image(uiImage: uiImage)
}
}
```

Tip: Sadly the Core Image behind the sepia tone filter wants **Float** rather than **Double** for its values. This makes Core Image feel even older, I know, but don't worry – we'll make it go away soon!

The next job is to change the way **loadImage()** works. Right now *that* assigns to the **image** property, but we don't want that any more. Instead, it should send whatever image was chosen into the sepia tone filter, then call **applyProcessing()** to make the magic happen.

Core Image filters have a dedicated **inputImage** property that lets us send in a **CIIImage** for the filter to work with, but often this is thoroughly broken and will cause your app to crash – it's much safer to use the filter's **setValue()** method with the key **kCIIInputImageKey**.

So, replace your existing **loadImage()** method with this:

```
func loadImage() {
    guard let inputImage = inputImage else { return }

    let beginImage = CIIImage(image: inputImage)
    currentFilter.setValue(beginImage, forKey: kCIIInputImageKey)
    applyProcessing()
}
```

If you run the code now you'll see our basic app flow works great: we can select an image, then see it with a sepia effect applied. But that intensity slider we added doesn't do anything, even though it's bound to the same **filterIntensity** value that our filter is reading from.

What's happening here ought not to be too surprising: even though the slider is changing the value of **filterIntensity**, changing that property won't automatically trigger our **applyProcessing()** method again. Instead, we need to do that by hand by telling SwiftUI to watch **filterIntensity** with **onChange()**.

Again, these **onChange()** modifiers can go anywhere in our SwiftUI view hierarchy, but in this situation I do something different: when one specific view is responsible for changing a value I usually add **onChange()** directly to that view so it's clear to me later on that adjusting the view triggers a side effect. If multiple views adjust the same value, or if it's not quite so specific what is changing the value, then I'd add the modifier at the end of the view.

Anyway, here **filterIntensity** is being changed by the slider, so let's add **onChange()** there:

```
Slider(value: $filterIntensity)
    .onChange(of: filterIntensity) { _ in
        applyProcessing()
    }
```

You can go ahead and run the app now, but be warned: even though Core Image is extremely fast on all iPhones, it's usually extremely *slow* in the simulator. That means you can try it out to make sure everything works, but don't be surprised if your code runs about as fast as an asthmatic ant carrying a heavy bag of shopping.

Customizing our filter using confirmationDialog()

So far we've brought together SwiftUI, **PHPickerViewController**, and Core Image, but the app still isn't terribly useful – after all, the sepia tone effect isn't *that* interesting.

To make the whole app better, we're going to let users customize the filter they want to apply, and we'll accomplish that using a confirmation dialog. On iPhone this is a list of buttons that slides up from the bottom of the screen, and you can add as many as you want – it can even scroll if you really need it to.

First we need a property that will store whether the confirmation dialog should be showing or not, so add this to **ContentView**:

```
@State private var showingFilterSheet = false
```

Now we can add our buttons using the **confirmationDialog()** modifier. This works identically to **alert()**: we provide a title and condition to monitor, and as soon as the condition becomes true the confirmation dialog will be shown.

Start by adding this modifier below the **sheet()**:

```
.confirmationDialog("Select a filter", isPresented:  
$showingFilterSheet) {  
    // dialog here  
}
```

Now replace the **// change filter** button action with this:

```
showingFilterSheet = true
```

In terms of *what* to show in the confirmation dialog, we can an array of buttons to show and an optional message. These buttons work just like with **alert()**: we provide a text title and an

Customizing our filter using confirmationDialog()

action to run when it's selected.

For the confirmation dialog in this app, we want users to select from a range of different Core Image filters, and when they choose one it should be activated and immediately applied. To make this work we're going to write a method that modifies **currentFilter** to whatever new filter they chose, then calls **loadImage()** straight away.

There is a wrinkle in our plan, and it's a result of the way Apple wrapped the Core Image APIs to make them more Swift-friendly. You see, the underlying Core Image API is entirely stringly typed, so rather than invent all new classes for us to use Apple instead created a series of protocols.

When we assign **CIFilter.sepiaTone()** to a property, we get an object of the class **CIFilter** that happens to conform to a protocol called **CISepiaTone**. That protocol then exposes the **intensity** parameter we've been using, but internally it will just map it to a call to **setValue(_:_forKey:)**.

This flexibility actually works in our favor because it means we can write code that works across all filters, as long as we're careful not to send in an invalid value.

So, let's start solving the problem. Please change your **currentFilter** property to this:

```
@State private var currentFilter: CIFilter =  
CIFilter.sepiaTone()
```

So, again, **CIFilter.sepiaTone()** returns a **CIFilter** object that conforms to the **CISepiaTone** protocol. Adding that explicit type annotation means we're throwing away some data: we're saying that the filter must be a **CIFilter** but doesn't have to conform to **CISepiaTone** any more.

As a result of this change we lose access to the **intensity** property, which means this code won't work any more:

```
currentFilter.intensity = filterIntensity
```

Project 13: Instafilter

Instead, we need to replace that with a call to `setValue(_:forKey:)`. This is all the protocol was doing anyway, but it did provide valuable extra type safety.

Replace that broken line of code with this:

```
currentFilter.setValue(filterIntensity, forKey:  
kCIIInputIntensityKey)
```

`kCIIInputIntensityKey` is another Core Image constant value, and it has the same effect as setting the `intensity` parameter of the sepia tone filter.

With that change, we can return to our confirmation dialog: we want to be able to change that filter to something else, then call `loadImage()`. So, add this method to `ContentView`:

```
func setFilter(_ filter: CIFilter) {  
    currentFilter = filter  
    loadImage()  
}
```

With that in place we can now replace the `// dialog here` comment with a series of buttons that try out various Core Image filters.

Put this in its place:

```
Button("Crystallize") { setFilter(CIFilter.crystallize()) }  
Button("Edges") { setFilter(CIFilter.edges()) }  
Button("Gaussian Blur") { setFilter(CIFilter.gaussianBlur()) }  
Button("Pixelate") { setFilter(CIFilter.pixelate()) }  
Button("Sepia Tone") { setFilter(CIFilter.sepiaTone()) }  
Button("Unsharp Mask") { setFilter(CIFilter.unsharpMask()) }  
Button("Vignette") { setFilter(CIFilter.vignette()) }  
Button("Cancel", role: .cancel) { }
```

Customizing our filter using confirmationDialog()

I picked out those from the vast range of Core Image filters, but you're welcome to try using code completion to try something else – type **CIFilter**, and see what comes up!

Go ahead and run the app, select a picture, then try changing Sepia Tone to Vignette – this applies a darkening effect around the edges of your photo. (If you're using the simulator, remember to give it a little time because it's slow!)

Now try changing it to Gaussian Blur, which *ought* to blur the image, but will instead cause our app to crash. By jettisoning the **CISepiaTone** restriction for our filter, we're now forced to send values in using **setValue(_ :forKey:)**, which provides no safety at all. In this case, the Gaussian Blur filter doesn't have an intensity value, so the app just crashes.

To fix this – and also to make our single slider do much more work – we're going to add some more code that reads all the valid keys we can use with **setValue(_ :forKey:)**, and only sets the intensity key if it's supported by the current filter. Using this approach we can actually query as many keys as we want, and set all the ones that are supported. So, for sepia tone this will set intensity, but for Gaussian blur it will set the radius (size of the blur), and so on.

This conditional approach will work with any filters you choose to apply, which means you can experiment with others safely. The only thing you need be careful with is to make sure you scale up **filterIntensity** by a number that makes sense – a 1-pixel blur is pretty much invisible, for example, so I'm going to multiply that by 200 to make it more pronounced.

Replace this line:

```
currentFilter.setValue(filterIntensity, forKey:  
kCIIInputIntensityKey)
```

With this:

```
let inputKeys = currentFilter.inputKeys  
  
if inputKeys.contains(kCIIInputIntensityKey)  
{ currentFilter.setValue(filterIntensity, forKey:
```

Project 13: Instafilter

```
kCIIInputIntensityKey) }  
if inputKeys.contains(kCIIInputRadiusKey)  
{ currentFilter.setValue(filterIntensity * 200, forKey:  
kCIIInputRadiusKey) }  
if inputKeys.contains(kCIIInputScaleKey)  
{ currentFilter.setValue(filterIntensity * 10, forKey:  
kCIIInputScaleKey) }
```

And with that in place you can now run the app safely, import a picture of your choosing, then try out all the various filters – nothing should crash any more. Try experimenting with different filters and keys to see what you can find!

Saving the filtered image using `UIImageWriteToSavedPhotosAlbum()`

To complete this project we're going to make that Save button do something useful: save the filtered photo to the user's photo library, so they can edit it further, share it, and so on.

As I explained previously, the `UIImageWriteToSavedPhotosAlbum()` function does everything we need, but it has the catch that it needs to be used with some code that really doesn't fit well with SwiftUI: it needs to be a class that inherits from `NSObject`, have a callback method that is marked with `@objc`, then point to that method using the `#selector` compiler directive.

Like I showed you in an earlier part of this tutorial, the best way to approach this is to isolate the whole image saving functionality in a separate, reusable class. You should already have `ImageSaver.swift` in place from a previous stage of this project, but if not create a new Swift file with that name now and give it this code:

```
import UIKit

class ImageSaver: NSObject {
    var successHandler: ((() -> Void)?
    var errorHandler: ((Error) -> Void)?

    func writeToPhotoAlbum(image: UIImage) {
        UIImageWriteToSavedPhotosAlbum(image, self,
#selector(saveComplete), nil)
    }

    @objc func saveComplete(_ image: UIImage,
didFinishSavingWithError error: Error?, contextInfo:
```

Project 13: Instafilter

```
UnsafeRawPointer) {
    if let error = error {
        errorHandler?(error)
    } else {
        successHandler?()
    }
}
```

We're going to return back to that in a moment to make it more useful, but first we need to make sure we request photo saving permission from the user correctly: we need to add a permission request string to our project's configuration options.

If you deleted the one you added earlier, please re-add it now:

- Open your target settings
- Select the Info tab
- Right-click on an existing option
- Choose Add Row
- Select “Privacy - Photo Library Additions Usage Description” for the key name.
- Enter “We want to save the filtered photo.” as the value.

With that in place, we can now think about how to save an image using the **ImageSaver** class. Right now we're setting our **image** property like this:

```
if let cgimg = context.createCGImage(outputImage, from:
outputImage.extent) {
    let uiImage = UIImage(cgImage: cgimg)
    image = Image(uiImage: uiImage)
}
```

You can actually go straight from a **CGImage** to a SwiftUI **Image** view, and previously I said we're going via **UIImage** because the **CGImage** equivalent requires some extra parameters.

Saving the filtered image using `UIImageWriteToSavedPhotosAlbum()`

That's all true, but there's an important second reason that now becomes important: we need a **UIImage** to send to our **ImageSaver** class, and this is the perfect place to create it.

So, add a new property to **ContentView** that will store this intermediate **UIImage**:

```
@State private var processedImage: UIImage?
```

And now we can modify the `applyProcessing()` method so that our **UIImage** gets stashed away for later:

```
if let cgimg = context.createCGImage(outputImage, from: outputImage.extent) {
    let uiImage = UIImage(cgImage: cgimg)
    image = Image(uiImage: uiImage)
    processedImage = uiImage
}
```

And now filling in the `save()` method is almost trivial:

```
func save() {
    guard let processedImage = processedImage else { return }

    let imageSaver = ImageSaver()
    imageSaver.writeToPhotoAlbum(image: processedImage)
}
```

Now, we *could* leave it there, but the whole reason we made **ImageSaver** into its own class was so that we could read whether the save was successful or not. Right now this gets reported back to us in a method in **ImageSaver**:

```
@objc func saveComplete(_ image: UIImage,
didFinishSavingWithError error: Error?, contextInfo:
UnsafeRawPointer) {
    // save complete
```

Project 13: Instafilter

}

In order for that result to be useful we need to make it propagate upwards so that our **ContentView** can use it. However, I don't want the horrors of `@objc` to escape our little class, so instead we're going to isolate that mess where it is and instead report back success or failure using closures – a much friendlier solution for Swift developers.

First add these two properties to the **ImageSaver** class, to represent closures handling success and failure:

```
var successHandler: ((() -> Void)?  
var errorHandler: ((Error) -> Void)?
```

Second, fill in the **didFinishSavingWithError** method so that it checks whether an error was provided, and calls one of those two closures:

```
if let error = error {  
    errorHandler?(error)  
} else {  
    successHandler?()  
}
```

And now we can – if we want to – provide one or both of those closures when using the **ImageSaver** class, like this:

```
let imageSaver = ImageSaver()  
  
imageSaver.successHandler = {  
    print("Success!")  
}  
  
imageSaver.errorHandler = {  
    print("Oops: \"\($0.localizedDescription)\"")  
}
```

```
Saving the filtered image using UIImageWriteToSavedPhotosAlbum()  
}  
  
imageSaver.writeToPhotoAlbum(image: processedImage)
```

Although the code is very different, the concept here is identical to what we did with **ImagePicker**: we wrapped up some UIKit functionality in such a way that we get all the behavior we want, just in a nicer, more SwiftUI-friendly way. Even better, this gives us another reusable piece of code that we can put into other projects in the future – we’re slowly building a library!

That final step completes our app, so go ahead and run it again and try it from end to end – import a picture, apply a filter, then save it to your photo library. Well done!

Instafilter: Wrap up

We covered a lot of ground in this tutorial, and we'll be coming back to a lot of it in the very next project – working with UIKit isn't a “nice to have” for most apps, so it's best you get used to it and start building up your library of functionality wrappers.

Still, we also learned some great SwiftUI stuff, including confirmation dialogs and `onChange()`, both of which are super common and will continue to be useful for years to come.

And there's Core Image. This is another one Apple's extremely powerful frameworks that never quite made the smooth leap to Swift – you need to know its quirks if you want to make the most of it. Still, you're through the worst of it now, so hopefully you can try using it in your own code!

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Try making the Save button disabled if there is no image in the image view.
2. Experiment with having more than one slider, to control each of the input keys you care about. For example, you might have one for radius and one for intensity.
3. Explore the range of available Core Image filters, and add any three of your choosing to the app.

Tip: That last one might be a little trickier than you expect. Why? Maybe have a think about it for 10 seconds!

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Instafilter. If you don't already subscribe, you can start a free trial today.

Project 14

Bucket List

Embed maps and make network calls in this life goals app

Bucket List: Introduction

In this project we're going to build an app that lets the user build a private list of places on the map that they intend to visit one day, add a description for that place, look up interesting places that are nearby, and save it all to the iOS storage for later.

To make all that work will mean leveraging some skills you've met already, such as forms, sheets, **Codable**, and **URLSession**, but also teach you some new skills: how to embed maps in a SwiftUI app, how to store private data safely so that only an authenticated user can access it, how to load and save data outside of **UserDefault**s, and more.

So, lots to learn and another great app to make! Anyway, let's get started with our techniques: create a new iOS project using the App template, and name it BucketList. And now on to our techniques...

Adding conformance to Comparable for custom types

When you think about it, we take a heck of a lot of stuff for granted when we write Swift code. For example, if we write `4 < 5`, we expect that to return true – the developers of Swift (and LLVM, larger compiler project that sits behind Swift) have already done all the hard work of checking whether that calculation is actually true, so we don't have to worry about it.

But what Swift does *really* well is extend functionality into lots of places using protocols and protocol extensions. For example, we know that `4 < 5` is true because we're able to compare two integers and decide whether the first one comes before or after the second. Swift extends that functionality to arrays of integers: we can compare all the integers in an array to decide whether each one should come before or after the others. Swift then uses that result to sort the array.

So, in Swift we expect this kind of code to Just Work:

```
struct ContentView: View {  
    let values = [1, 5, 3, 6, 2, 9].sorted()  
  
    var body: some View {  
        List(values, id: \.self) {  
            Text(String($0))  
        }  
    }  
}
```

We don't need to tell `sorted()` how it should work, because it understands how arrays of integers work.

Now consider a struct like this one:

```
struct User: Identifiable {
```

Adding conformance to Comparable for custom types

```
let id = UUID()
let firstName: String
let lastName: String
}
```

We could make an array of those users, and use them inside a **List** like this:

```
struct ContentView: View {
    let users = [
        User(firstName: "Arnold", lastName: "Rimmer"),
        User(firstName: "Kristine", lastName: "Kochanski"),
        User(firstName: "David", lastName: "Lister"),
    ]

    var body: some View {
        List(users) { user in
            Text("\(user.lastName), \(user.firstName)")
        }
    }
}
```

That will work just fine, because we made the **User** struct conform to **Identifiable**.

But how about if we wanted to show those users in a sorted order? If we modify the code to this it won't work:

```
let users = [
    User(firstName: "Arnold", lastName: "Rimmer"),
    User(firstName: "Kristine", lastName: "Kochanski"),
    User(firstName: "David", lastName: "Lister"),
].sorted()
```

Swift doesn't understand what **sorted()** means here, because it doesn't know whether to sort

Project 14: Bucket List

by first name, last name, both, or something else.

Previously I showed you how we could provide a closure to `sorted()` to do the sorting ourselves, and we could use the same here:

```
let users = [
    User(firstName: "Arnold", lastName: "Rimmer"),
    User(firstName: "Kristine", lastName: "Kochanski"),
    User(firstName: "David", lastName: "Lister"),
].sorted {
    $0.lastName < $1.lastName
}
```

That absolutely works, but it's not an ideal solution for two reasons.

First, this is *model* data, by which I mean that it's affecting the way we work with the `User` struct. That struct and its properties are our data model, and in a well-developed application we don't really want to tell the model how it should behave inside our SwiftUI code. SwiftUI represents our *view*, i.e. our layout, and if we put model code in there then things get confused.

Second, what happens if we want to sort `User` arrays in multiple places? You might copy and paste the closure once or twice, before realizing you're just creating a problem for yourself: if you end up changing your sorting logic so that you also use `firstName` if the last name is the same, then you need to search through all your code to make sure all the closures get updated.

Swift has a better solution. Arrays of integers get a simple `sorted()` method with no parameters because Swift understands how to compare two integers. In coding terms, `Int` conforms to the **Comparable** protocol, which means it defines a function that takes two integers and returns true if the first should be sorted before the second.

We can make our own types conform to **Comparable**, and when we do so we *also* get a `sorted()` method with no parameters. This takes two steps:

1. Add the **Comparable** conformance to the definition of `User`.

Adding conformance to Comparable for custom types

2. Add a method called `<` that takes two users and returns true if the first should be sorted before the second.

Here's how that looks in code:

```
struct User: Identifiable, Comparable {  
    let id = UUID()  
    let firstName: String  
    let lastName: String  
  
    static func <(lhs: User, rhs: User) -> Bool {  
        lhs.lastName < rhs.lastName  
    }  
}
```

There's not a lot of code in there, but there is still a lot to unpack.

First, yes the method is just called `<`, which is the “less than” operator. It’s the job of the method to decide whether one user is “less than” (in a sorting sense) another, so we’re adding functionality to an existing operator. This is called *operator overloading*, and it can be both a blessing and a curse.

Second, **lhs** and **rhs** are coding conventions short for “left-hand side” and “right-hand side”, and they are used because the `<` operator has one operand on its left and one on its right.

Third, this method must return a Boolean, which means we *must* decide whether one object should be sorted before another. There is no room for “they are the same” here – that’s handled by another protocol called **Equatable**.

Fourth, the method must be marked as **static**, which means it’s called on the **User** struct directly rather than a single instance of the struct.

Finally, our logic here is pretty simple: we’re just passing on the comparison to one of our properties, asking Swift to use `<` for the two last name strings. You can add as much logic as

Project 14: Bucket List

you want, comparing as many properties as needed, but ultimately you need to return true or false.

Tip: One thing you *can't* see in that code is that conforming to **Comparable** also gives us access to the `>` operator – greater than. This is the opposite of `<`, so Swift creates it for us by using `<` and flipping the Boolean between true and false.

Now that our **User** struct conforms to **Comparable**, we automatically get access to the parameter-less version of **sorted()**, which means this kind of code works now:

```
let users = [
    User(firstName: "Arnold", lastName: "Rimmer"),
    User(firstName: "Kristine", lastName: "Kochanski"),
    User(firstName: "David", lastName: "Lister"),
].sorted()
```

This resolves the problems we had before: we now isolate our model functionality in the struct itself, and we no longer need to copy and paste code around – we can use **sorted()** everywhere, safe in the knowledge that if we ever change the algorithm then all our code will adapt.

Writing data to the documents directory

Previously we looked at how to read and write data to **UserDefaults**, which works great for user settings or small amounts of JSON. However, it's generally not a great place to store data, particularly if you think you'll start storing more in the future.

In this app we're going to be letting users create as much data as they want, which means we want a better storage solution than just throwing things into **UserDefaults** and hoping for the best. Fortunately, iOS makes it very easy to read and write data from device storage, and in fact all apps get a directory for storing any kind of documents we want. Files here are automatically synchronized with iCloud backups, so if the user gets a new device then our data will be restored along with all the other system data – we don't even need to think about it.

There is a catch – isn't there always? – and it's that all iOS apps are *sandboxed*, which means they run in their own container with a hard to guess directory name. As a result, we can't – and shouldn't try to – guess the directory where our app is installed, and instead need to rely on Apple's API for finding our app's documents directory.

There is no nice way of doing this, so I nearly always just copy and paste the same helper method into my projects, and we're going to do exactly the same thing now. This uses a new class called **FileManager**, which can provide us with the document directory for the current user. In theory this can return several path URLs, but we only ever care about the first one.

So, add this method to **ContentView**:

```
func getDocumentsDirectory() -> URL {
    // find all possible documents directories for this user
    let paths = FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)

    // just send back the first one, which ought to be the only
    one
```

Project 14: Bucket List

```
    return paths[0]  
}
```

That documents directory is ours to do with as we please, and because it belongs to the app it will automatically get deleted if the app itself gets deleted. Other than physical device limitations there is no limit to how much we can store, although remember that users can use the Settings app to see how much storage your app takes up – be respectful!

Now that we have a directory to work with, we can read and write files there freely. You already met **String(contentsOf:)** and **Data(contentsOf:)** for *reading* data, but for writing data we need to use the **write(to:)** method. When used with strings this takes three parameters:

1. A **URL** to write to.
2. Whether to make the write *atomic*, which means “all at once”.
3. What character encoding to use.

The first of those can be created by combining the documents directory URL with a filename, such as myfile.txt.

The second should nearly always be set to true. If this is set to false and we try to write a big file, it’s possible that another part of our app might try and read the file while it’s still being written. This shouldn’t cause a crash or anything, but it *does* mean that it’s going to read only part of the data, because the other part hasn’t been written yet. *Atomic* writing causes the system to write our full file to a temporary filename (not the one we asked for), and when that’s finished it does a simple rename to our target filename. This means either the whole file is there or nothing is.

The third parameter is something we looked at briefly in project 5, because we had to use a Swift string with an Objective-C API. Back then we used the character encoding UTF-16, which is what Objective-C uses, but Swift’s native encoding is UTF-8, so we’re going to use that instead.

To put all this code into action, we’re going to modify the default text view of our template so

Writing data to the documents directory

that it writes a test string to a file in the documents directory, reads it back into a new string, then prints it out – the complete cycle of reading and writing data.

Change the **body** property of **ContentView** to this:

```
Text("Hello World")
    .onTapGesture {
        let str = "Test Message"
        let url =
            getDocumentsDirectory().appendingPathComponent("message.txt")

        do {
            try str.write(to: url, atomically: true, encoding: .utf8)
            let input = try String(contentsOf: url)
            print(input)
        } catch {
            print(error.localizedDescription)
        }
    }
}
```

When that runs you should be able to tap the label to see “Test message” printed to Xcode’s debug output area.

Before we move on, here’s a small challenge for you: back in project 8 we looked at how to create a generic extension on **Bundle** that let us find, load, and decode any **Codable** data from our app bundle. Can you write something similar for the documents directory, perhaps making it an extension on **FileManager**?

Switching view states with enums

You've seen how we can use regular Swift conditions to present one type of view or the other, and we looked at code along the lines of this:

```
if Bool.random() {
    Rectangle()
} else {
    Circle()
}
```

Tip: When returning different kinds of view, make sure you're either inside the **body** property or using something like **@ViewBuilder** or **Group**.

Where conditional views are particularly useful is when we want to show one of several different states, and if we plan it correctly we can keep our view code small and also easy to maintain – it's a great way to start training your brain to think about SwiftUI architecture.

There are two parts to this solution. The first is to define an enum for the various view states you want to represent. For example, you might define this as a nested enum:

```
enum LoadingState {
    case loading, success, failed
}
```

Next, create individual views for those states. I'm just going to use simple text views here, but they could hold anything:

```
struct LoadingView: View {
    var body: some View {
        Text("Loading...")
    }
}
```

```

struct SuccessView: View {
    var body: some View {
        Text("Success!")
    }
}

struct FailedView: View {
    var body: some View {
        Text("Failed.")
    }
}

```

Those views could be nested if you want, but they don't have to be – it really depends on whether you plan to use them elsewhere and the size of your app.

With those two parts in place, we now effectively use **ContentView** as a simple wrapper that tracks the current app state and shows the relevant child view. That means giving it a property to store the current **LoadingState** value:

```
var loadingState = LoadingState.loading
```

Then filling in its **body** property with code that shows the correct view based on the enum value, like this:

```

if loadingState == .loading {
    LoadingView()
} else if loadingState == .success {
    SuccessView()
} else if loadingState == .failed {
    FailedView()
}

```

Using this approach our **ContentView** doesn't spiral out of control as more and more code

Project 14: Bucket List

gets added to the views, and in fact has no idea what loading, success, or failure even look like.

Integrating MapKit with SwiftUI

Maps have been a core feature of iPhone since the very first device shipped way back in 2007, and the underlying framework has been available to developers for almost as long. Even better, Apple provides a SwiftUI **Map** view that wraps up the underlying map framework beautifully, letting us place maps, annotations, and more alongside the rest of our SwiftUI view hierarchy.

Let's start with something simple: showing a map means creating some program state that stores the map's current center coordinate and zoom level, which is handled through a dedicated type called **MKCoordinateRegion**. The "MK" in that name means this comes from Apple's MapKit framework, so our first step is to import that framework:

```
import MapKit
```

Now we can make a property such as this one:

```
@State private var mapRegion = MKCoordinateRegion(center:  
CLLocationCoordinate2D(latitude: 51.5, longitude: -0.12), span:  
MKCoordinateSpan(latitudeDelta: 0.2, longitudeDelta: 0.2))
```

That centers on the city of London. Both sets of latitude and longitude are measured in degrees, but in practice longitude changes in its underlying value as you move further away from the equator so it might take a little experimentation to find a starting value you like.

Finally, we can add a map view like this:

```
Map(coordinateRegion: $mapRegion)
```

That has a two-way binding to the region so it can be updated as the user moves around the map, and when the app runs you should see London right there on your map.

There are a variety of extra options we can use when creating maps, but by far the most important is the ability to add annotations to the map – markers that represent various places of our choosing.

Project 14: Bucket List

To do this takes at least three steps depending on your goal: defining a new data type that contains your location, creating an array of those containing *all* your locations, then adding them as annotations in the map. Whatever new data type you create to store locations, it *must* conform to the **Identifiable** protocol so that SwiftUI can identify each map marker uniquely.

For example, we might start with this kind of **Location** struct:

```
struct Location: Identifiable {
    let id = UUID()
    let name: String
    let coordinate: CLLocationCoordinate2D
}
```

Now we can go ahead and define an array of locations, wherever we want map annotations to appear:

```
let locations = [
    Location(name: "Buckingham Palace", coordinate:
CLLocationCoordinate2D(latitude: 51.501, longitude: -0.141)),
    Location(name: "Tower of London", coordinate:
CLLocationCoordinate2D(latitude: 51.508, longitude: -0.076))
]
```

Step three is the important part: we can feed that array of locations into the **Map** view, as well as providing a function that transforms one location into a visible annotation on the map. SwiftUI provides us with a couple of different annotation types, but the simplest is **MapMarker**: a simple balloon with a latitude/longitude coordinate attached.

For example, we could place markers at both our locations like so:

```
Map(coordinateRegion: $mapRegion, annotationItems: locations)
{ location in
    MapMarker(coordinate: location.coordinate)
```

```
}
```

When that runs you'll see two red balloons on the map, although they don't show any useful information – our locations don't have their name visible, for example. To add that extra information we need to create a wholly custom view using a different annotation type, helpfully just called **MapAnnotation**. This accepts the same coordinate as **MapMarker**, except rather than just showing a system-style balloon we instead get to pass in whatever custom SwiftUI views we want.

So, we could replace the balloons with stroked red circles like this:

```
Map(coordinateRegion: $mapRegion, annotationItems: locations)
{ location in
    MapAnnotation(coordinate: location.coordinate) {
        Circle()
            .stroke(.red, lineWidth: 3)
            .frame(width: 44, height: 44)
    }
}
```

Once you're using **MapAnnotation** you can pass in any SwiftUI views you want – it's a great customization point, and can include any interactivity you want.

For example, we could add a tap gesture to our annotations like this:

```
MapAnnotation(coordinate: location.coordinate) {
    Circle()
        .stroke(.red, lineWidth: 3)
        .frame(width: 44, height: 44)
        .onTapGesture {
            print("Tapped on \(location.name)")
        }
}
```

Project 14: Bucket List

We could even place a **NavLink** into our map annotation, directing the user to a different view when the annotation was tapped:

```
NavigationView {
    Map(coordinateRegion: $mapRegion, annotationItems: locations)
    { location in
        MapAnnotation(coordinate: location.coordinate) {
            NavigationLink {
                Text(location.name)
            } label: {
                Circle()
                    .stroke(.red, lineWidth: 3)
                    .frame(width: 44, height: 44)
            }
        }
    }
    .navigationTitle("London Explorer")
}
```

The point is that you get to decide whether you want something simple or something more advanced, then add any interactivity using all the SwiftUI tools and techniques you already know.

Using Touch ID and Face ID with SwiftUI

The vast majority of Apple's devices come with biometric authentication as standard, which means they use fingerprint and facial recognition to unlock. This functionality is available to us too, which means we can make sure that sensitive data can only be read when unlocked by a valid user.

This is another Objective-C API, but it's only a *little* bit unpleasant to use with SwiftUI, which is better than we've had with some other frameworks we've looked at so far.

Before we write any code, you need to add a new key to your project options, explaining to the user why you want access to Face ID. For reasons known only to Apple, we pass the Touch ID request reason in code, and the Face ID request reason in project options.

So, select your current target, go to the Info tab, right-click on an existing key, then choose Add Row. Scroll through the list of keys until you find “Privacy - Face ID Usage Description” and give it the value “We need to unlock your data.”

Now head back to ContentView.swift, and add this import near the top of the file:

```
import LocalAuthentication
```

And with that, we're all set to write some biometrics code.

I mentioned earlier this was “only a *little* bit unpleasant”, and here’s where it comes in: Swift developers use the **Error** protocol for representing errors that occur at runtime, but Objective-C uses a special class called **NSError**. We need to be able to pass that into the function and have it changed *inside* the function rather than returning a new value – although this was the standard in Objective-C, it’s quite an alien way of working in Swift so we need to mark this behavior specially by using **&**.

We’re going to write an **authenticate()** method that isolates all the biometric functionality in a

Project 14: Bucket List

single place. To make that happen requires four steps:

1. Create instance of **LAContext**, which allows us to query biometric status and perform the authentication check.
2. Ask that context whether it's capable of performing biometric authentication – this is important because iPod touch has neither Touch ID nor Face ID.
3. If biometrics are possible, then we kick off the actual request for authentication, passing in a closure to run when authentication completes.
4. When the user has either been authenticated or not, our completion closure will be called and tell us whether it worked or not, and if not what the error was.

Please go ahead and add this method to **ContentView**:

```
func authenticate() {  
    let context = LAContext()  
    var error: NSError?  
  
    // check whether biometric authentication is possible  
    if  
        context.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, error: &error) {  
            // it's possible, so go ahead and use it  
            let reason = "We need to unlock your data."  
  
            context.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, localizedReason: reason) { success, authenticationError in  
                // authentication has now completed  
                if success {  
                    // authenticated successfully  
                } else {  
                    // there was a problem  
                }  
            }  
        }  
    }  
}
```

```

        }
    } else {
        // no biometrics
    }
}

```

That method by itself won't do anything, because it's not connected to SwiftUI at all. To fix that we need to do add some state we can adjust when authentication is successful, and also an **onAppear()** modifier to trigger authentication.

So, first add this property to **ContentView**:

```
@State private var isUnlocked = false
```

That simple Boolean will store whether the app is showing its protected data or not, so we'll flip that to true when authentication succeeds. Replace the **// authenticated successfully** comment with this:

```
isUnlocked = true
```

Finally, we can show the current authentication state and begin the authentication process inside the **body** property, like this:

```

VStack {
    if isUnlocked {
        Text("Unlocked")
    } else {
        Text("Locked")
    }
}.onAppear(perform: authenticate)

```

If you run the app there's a good chance you just see "Locked" and nothing else. This is

Project 14: Bucket List

because the simulator isn't opted in to biometrics by default, and we didn't provide any error messages, so it fails silently.

To take Face ID for a test drive, go to the Features menu and choose Face ID > Enrolled, then launch the app again. This time you should see the Face ID prompt appear, and you can trigger successful or failed authentication by going back to the Features menu and choosing Face ID > Matching Face or Non-matching Face.

All being well you should see the Face ID prompt go away, and underneath it will be the “Unlocked” text view – our app has detected the authentication, and is now open to use.

Important: When working with biometric authentication, you should always look for a backup plan that lets users authenticate without biometrics. Think about folks who might be wearing a face mask, or perhaps are wearing gloves right now – if you try to force them to use biometrics all the time you'll just have angry users. So, consider adding a screen that prompts for a passcode, then provide that as a fallback if biometrics fail.

Adding user locations to a map

This project is going to be based around a map view, asking users to add places to the map that they want to visit. To do that we need to place a **Map** so that it takes up our whole view, track its center coordinate, and then also whether or not the user is viewing place details, what annotations they have, and more.

We're going to start with a full-screen **Map** view, then place a translucent circle on top to represent the center point. Although this view will have a binding to track the center coordinate, we don't need to use that to place the circle – a simple **ZStack** will make sure the circle always stays in the center of the map.

First, add an extra **import** line so we get access to MapKit's data types:

```
import MapKit
```

Second, add a property inside **ContentView** that will store the current state of the map. Later on we're going to use this to add a place mark:

```
@State private var mapRegion = MKCoordinateRegion(center:  
CLLocationCoordinate2D(latitude: 50, longitude: 0), span:  
MKCoordinateSpan(latitudeDelta: 25, longitudeDelta: 25))
```

That starts the map so that most of Western Europe and North Africa are visible.

And now we can fill in the **body** property

```
ZStack {  
    Map(coordinateRegion: $mapRegion)  
        .ignoresSafeArea()  
    Circle()  
        .fill(.blue)  
        .opacity(0.3)  
        .frame(width: 32, height: 32)
```

Project 14: Bucket List

}

If you run the app now you'll see you can move the map around freely, but there's always a blue circle showing exactly where the center is.

All this work by itself isn't terribly interesting, so the next step is to add a button in the bottom-right that lets us add place marks to the map. We're already inside a **ZStack**, so the easiest way to align this button is to place it inside a **VStack** and a **HStack** with spacers before it each time. Both those spacers end up occupying the full vertical and horizontal space that's left over, making whatever comes at the end sit comfortably in the bottom-right corner.

We'll add some functionality for the button soon, but first let's get it in place and add some basic styling to make it look good.

Please add this **VStack** below the **Circle**:

```
VStack {  
    Spacer()  
    HStack {  
        Spacer()  
        Button {  
            // create a new location  
        } label: {  
            Image(systemName: "plus")  
        }  
        .padding()  
        .background(.black.opacity(0.75))  
        .foregroundColor(.white)  
        .font(.title)  
        .clipShape(Circle())  
        .padding(.trailing)  
    }  
}
```

Notice how I added the **padding()** modifier twice there – once is to make sure the button is bigger before we add a background color, and the second time to push it away from the trailing edge.

Where things get *interesting* is how we place locations on the map. We've bound the location of the map to a property in **ContentView**, but now we need to send in an array of locations we want to show.

This takes a few steps, starting with a basic definition of the type of locations we're creating in our app. This needs to conform to a few protocols:

- **Identifiable**, so we can create many location markers in our map.
- **Codable**, so we can load and save map data easily.
- **Equatable**, so we can find one particular location in an array of locations.

In terms of the data it will contain, we'll give each location a name and description, plus a latitude and longitude. We'll also need to add a unique identifier so SwiftUI is happy to create them from dynamic data.

So, create a new Swift file called `Location.swift`, giving it this code:

```
struct Location: Identifiable, Codable, Equatable {
    let id: UUID
    var name: String
    var description: String
    let latitude: Double
    let longitude: Double
}
```

Storing latitude and longitude separately gives us **Codable** conformance out of the box, which is always nice to have. We'll add a little more to that shortly, but it's enough to get us moving.

Now that we have a data type where we can store an individual location, we need an array of

Project 14: Bucket List

those to store all the places the user wants to visit. We'll put this into **ContentView** for now just we can get moving, but again we'll return to it shortly to add more.

So, start by adding this property to **ContentView**:

```
@State private var locations = [Location]()
```

Next, we want to add a location to that whenever the + button is tapped, so replace the *// create a new location* comment with this:

```
let newLocation = Location(id: UUID(), name: "New location",
description: "", latitude: mapRegion.center.latitude,
longitude: mapRegion.center.longitude)
locations.append(newLocation)
```

Finally, update **ContentView** so that it sends in the **locations** array to be converted into annotations:

```
Map(coordinateRegion: $mapRegion, annotationItems: locations)
{ location in
    MapMarker(coordinate: CLLocationCoordinate2D(latitude:
location.latitude, longitude: location.longitude))
}
.ignoresSafeArea()
```

That's enough map work for now, so go ahead and run your app again – you should be able to move around as much as you need, then press the + button to add locations.

I know it took a fair amount of work to get set up, but at least you can see the basics of the app coming together!

Improving our map annotations

Right now we're using **MapMarker** to place locations in our **Map** view, but SwiftUI lets us place any kind of view on top of our map so we can have complete customizability. So, we're going to use that to show a custom SwiftUI view containing an icon and some text to show the location's name, then take a look at the underlying data type to see what improvements can be made there.

Thanks to the brilliance of SwiftUI, this takes hardly any code at all – replace your existing **MapMarker** code with this:

```
MapAnnotation(coordinate: CLLocationCoordinate2D(latitude:  
location.latitude, longitude: location.longitude)) {  
    VStack {  
        Image(systemName: "star.circle")  
            .resizable()  
            .foregroundColor(.red)  
            .frame(width: 44, height: 44)  
            .background(.white)  
            .clipShape(Circle())  
  
        Text(location.name)  
    }  
}
```

That's already an immediate improvement, because now it's clear what each marker represents – the location name appears directly underneath. However, I want to look beyond just the SwiftUI view: I want to look at the **Location** struct itself, and apply a few improvements that make it better.

First, I don't particularly like having to make a **CLLocationCoordinate2D** inside our SwiftUI view, and I'd much prefer to move that kind of logic inside our **Location** struct. So, we can move that into a computed property to clean up our code. First, add an import for MapKit into

Project 14: Bucket List

Location.swift, then add this to Location`:

```
var coordinate: CLLocationCoordinate2D {
    CLLocationCoordinate2D(latitude: latitude, longitude:
longitude)
}
```

Now our **ContentView** code is simpler:

```
MapAnnotation(coordinate: location.coordinate) {
```

The second change I want to make is one I encourage everyone to make when building custom data types for use with SwiftUI: add an example! This makes previewing significantly easier, so where possible I would encourage you to add a static **example** property to your types containing some sample data that can be previewed well.

So, add this second property to **Location** now:

```
static let example = Location(id: UUID(), name: "Buckingham
Palace", description: "Where Queen Elizabeth lives with her
dorgis.", latitude: 51.501, longitude: -0.141)
```

The last change I'd like to make here is to add a custom == function to the struct. We already made **Location** conform to **Equatable**, which means we can already compare one location to another using ==. Behind the scenes, Swift will write this function for us by comparing every property against every other property, which is rather wasteful – all our locations already have a unique identifier, so if two locations have the same identifier then we can be sure they are the same without also checking the other properties.

So, we can save a bunch of work by writing our own == function to **Location**, which compares two identifiers and nothing else:

```
static func ==(lhs: Location, rhs: Location) -> Bool {
    lhs.id == rhs.id
```

}

I'm a huge fan of making structs conform to **Equatable** as standard, even if you can't use an optimized comparison function like above – structs are simple values like strings and integers, and I think we should extend that same status to our own custom structs too.

With that in place the next step of our project is complete, so please run it now – you should be able to drop a marker and see our custom annotation, but now behind the scenes know that our code is a little bit neater too!

Selecting and editing map annotations

Users can now drop markers onto our SwiftUI **Map**, but they can't do anything with them – they can't attach their own name and description. Fixing this requires a few steps, and learning a couple of things along the way, but it really brings the whole app together as you'll see.

First, we want to show some kind of sheet when the user selects a map annotation, giving them the chance to view or edit details about a location.

The way we've tackled sheets previously has meant creating a Boolean that determines whether the sheet is visible, then sending in some other data for the sheet to present or edit. This time, though, we're going to take a different approach: we're going to handle it all with one piece property.

So, add this to **ContentView** now:

```
@State private var selectedPlace: Location?
```

What we're saying is that we might have a selected location, or we might not – and that's all SwiftUI needs to know in order to present a sheet. As soon as we place a value into that optional we're telling SwiftUI to show the sheet, and the value will automatically be set back to **nil** when the sheet is dismissed. Even better, SwiftUI automatically unwraps the optional for us, so when we're creating the contents of our sheet we can be sure we have a real value to work with.

To try it out, attach this modifier to the **ZStack** in **ContentView**:

```
.sheet(item: $selectedPlace) { place in
    Text(place.name)
}
```

As you can see, it takes an optional binding, but also a function that will receive the

unwrapped optional when it has a value set. So, inside there our sheet can refer to **place.name** directly rather than needing to unwrap the optional or use nil coalescing.

Now to bring the whole thing to life, we just need to give **selectedPlace** a value by adding a tap gesture to the **VStack** in our map annotation:

```
.onTapGesture {
    selectedPlace = location
}
```

That's it! We can now present a sheet showing the selected location's name, and it only took a small amount of code. This kind of optional binding isn't always possible, but I think where it *is* possible it makes for much more natural code – SwiftUI's behavior of unwrapping the optional automatically is really helpful.

Of course, just showing the place's name isn't too useful, so the next step here is to create a detail view where used can see and adjust a location's name and description. This needs to receive a location to edit, allow the user to adjust the two values for that location, then will send back a new location with that tweaked data – it will effectively work like a function, receiving data and sending back something transformed.

As always we're going to start small and work our way up, so please create a new SwiftUI view called "EditView" then give it this code:

```
struct EditView: View {
    @Environment(\.dismiss) var dismiss
    var location: Location

    @State private var name: String
    @State private var description: String

    var body: some View {
        NavigationView {
```

Project 14: Bucket List

```
Form {
    Section {
        TextField("Place name", text: $name)
        TextField("Description", text: $description)
    }
}
.navigationBar("Place details")
.toolbar {
    Button("Save") {
        dismiss()
    }
}
}
```

That code won't compile, because we have a conundrum: what initial values should we use for the **name** and **description** properties? Previously we've used **@State** with initial values, but we can't do that here – their initial values should come from what location is being passed in, so the user sees the saved data.

The solution is to create a new initializer that accepts a location, and uses that to create **State** structs using the location's data. This uses the same underscore approach we used when creating a fetch request inside an initializer, which allows us to create an instance of the property wrapper not the data inside the wrapper.

So, to solve our problem we need to add this initializer to **EditView**:

```
init(location: Location) {
    self.location = location

    _name = State(initialValue: location.name)
    _description = State(initialValue: location.description)
```

}

You'll need to modify your preview code to use that initializer:

```
static var previews: some View {
    EditView(location: Location.example)
}
```

That makes the code compile, but we have a second problem: when we're done editing the location, how can we pass the new location data back? We *could* use something like **@Binding** to pass in a remote value, but that creates problems with our optional in **ContentView** – we want **EditView** to be bound to a real value rather than an optional value, because otherwise it would get confusing.

We're going to take simplest solution we can: we'll require a function to call where we can pass back whatever new location we want. This means any other SwiftUI can send us some data, and get back some new data to process however we want.

Start by adding this property to **EditView**:

```
var onSave: (Location) -> Void
```

That asks for a function that accepts a single location and returns nothing, which is perfect for our usage. We need to accept that in our initializer, like this:

```
init(location: Location, onSave: @escaping (Location) -> Void)
{
    self.location = location
    self.onSave = onSave

    _name = State(initialValue: location.name)
    _description = State(initialValue: location.description)
}
```

Project 14: Bucket List

Remember, **@escaping** means the function is being stashed away for user later on, rather than being called immediately, and it's needed here because the **onSave** function will get called only when the user presses Save.

Speaking of which, we need to update that Save button to create a new location with the modified details, and send it back with **onSave()**:

```
Button("Save") {
    var newLocation = location
    newLocation.name = name
    newLocation.description = description

    onSave(newLocation)
    dismiss()
}
```

By taking a variable copy of the original location, we get access to its existing data – it's identifier, latitude, and longitude.

Don't forget to update your preview code too – just passing in a placeholder closure is fine here:

```
EditView(location: Location.example) { newLocation in }
```

That completes **EditView** for now, but there's still some work to do back in **ContentView** because we need to present the new UI in our sheet, send in the location that was selected, and also handle updating changes.

Well, thanks to the way we've built our code this only takes a handful of lines of code – place this into the **sheet()** modifier in **ContentView**:

```
EditView(location: place) { newLocation in
    if let index = locations.firstIndex(of: place) {
```

```

    locations[index] = newLocation
}
}
}

```

So, that passes the location into **EditView**, and also passes in a closure to run when the Save button is pressed. That accepts the new location, then looks up where the *current* location is and replaces it in the array. This will cause our map to update immediately with the new data.

Go ahead and give the app a try – see if you spot a problem with our code. Hopefully it's rather glaring: renaming doesn't actually work!

The problem here is that we told SwiftUI that two places were identical if their IDs were identical, and that isn't true any more – when we update a marker so it has a different name, SwiftUI will compare the old marker and new one, see that their IDs are the same, and therefore not bother to change the map.

The fix here is to make the **id** property mutable, like this:

```
var id: UUID
```

And now we can adjust that when we create new locations:

```

var newLocation = location
newLocation.id = UUID()
newLocation.name = name
newLocation.description = description

```

There is no hard and fast rule for when it's better to make a wholly new object from scratch, or just copy an existing one and change the bits you want like we're doing here; I encourage you to experiment and find an approach you like.

Anyway, with that you can now run your code again. Sure, it doesn't save any data yet, but you can now add as many locations as you want and give them meaningful names.

Project 14: Bucket List

There is one last thing, though, and it's entirely possible this might not exist in a future SwiftUI update so try it for yourself: right now I find that giving a location a short name such as "Home", then changing it to have a long name such as "This is my home", will cause its label to be clipped until you interact with the map.

We can fix this with a new modifier called **fixedSize()**, which forces any view to be given its natural size rather than try to accommodate the amount of space offered by its parent. In this case, the **MapAnnotation** doesn't do a great job of handling resizing children, which causes the clipping, but **fixedSize()** lets us bypass that so the text automatically grows into as much space as needed.

So, to finish up this step please modify your map annotation content to this:

```
Text(location.name)
    .fixedSize()
```

It's a small change, and again hopefully it will get resolved in a future SwiftUI release, but it solves our problem for now.

Downloading data from Wikipedia

To make this whole app more useful, we’re going to modify our **EditView** screen so that it shows interesting places. After all, if visiting London is on your bucket list, you’d probably want some suggestions for things to see nearby. This might sound hard to do, but actually we can query Wikipedia using GPS coordinates, and it will send back a list of places that are nearby.

Wikipedia’s API sends back JSON data in a precise format, so we need to do a little work to define **Codable** structs capable of storing it all. The structure is this:

- The main result contains the result of our query in a key called “query”.
- Inside the query is a “pages” dictionary, with page IDs as the key and the Wikipedia pages themselves as values.
- Each page has a lot of information, including its coordinates, title, terms, and more.

We can represent that using three linked structs, so create a new Swift file called `Result.swift` and give it this content:

```
struct Result: Codable {
    let query: Query
}

struct Query: Codable {
    let pages: [Int: Page]
}

struct Page: Codable {
    let pageid: Int
    let title: String
    let terms: [String: [String]]?
}
```

Project 14: Bucket List

We're going to use that to store data we fetch from Wikipedia, then show it immediately in our UI. However, we need something to show while the fetch is happening – a text view saying “Loading” or similar ought to do the trick.

This means conditionally showing different UI depending on the current load state, and that means defining an enum that actually *stores* the current load state otherwise we don't know what to show.

Start by adding this nested enum to **EditView**:

```
enum LoadingState {
    case loading, loaded, failed
}
```

Those cover are all the states we need to represent our network request.

Next we're going to add two properties to **EditView**: one to represent the loading state, and one to store an array of Wikipedia pages once the fetch has completed. So, add these two now:

```
@State private var loadingState = LoadingState.loading
@State private var pages = [Page]()
```

Before we tackle the network request itself, we have one last easy job to do: adding to our **Form** a new section to show pages if they have loaded, or status text views otherwise. We can put these **if/else if** conditions or a **switch** statement right into the **Section** and SwiftUI will figure it out.

So, put this section below the existing one:

```
Section("Nearby...") {
    switch loadingState {
        case .loaded:
            ForEach(pages, id: \.pageid) { page in
                Text(page.title)
            }
    }
}
```

```

        .font(.headline)
        + Text(": ") +
    Text("Page description here")
        .italic()
    }

case .loading:
    Text("Loading...")
case .failed:
    Text("Please try again later.")
}

}

```

Tip: Notice how we can use `+` to add text views together? This lets us create larger text views that mix and match different kinds of formatting. That “Page description here” is just temporary – we’ll replace it soon.

Now for the part that really brings all this together: we need to fetch some data from Wikipedia, decode it into a **Result**, assign its pages to our **pages** property, then set **loadingState** to **.loaded**. If the fetch fails, we’ll set **loadingState** to **.failed**, and SwiftUI will load the appropriate UI.

Warning: The Wikipedia URL we need to load is really long, so rather than try to type it in you might want to copy and paste from the text or from my GitHub gist: <http://bit.ly/swiftwiki>.

Add this method to **EditView**:

```

func fetchNearbyPlaces() async {
    let urlString = "https://en.wikipedia.org/w/api.php?
ggscoord=\(location.latitude)%7C\
\(location.longitude)&action=query&prop=coordinates%7Cpageimages
%7Cpageterms&colimit=50&prop=thumbnail&thumbheight=500&prop=&
t=50&wbptterms=description&generator=geosearch&ggsradius=10000&

```

Project 14: Bucket List

```
ggslimit=50&format=json"

guard let url = URL(string: urlString) else {
    print("Bad URL: \(urlString)")
    return
}

do {
    let (data, _) = try await URLSession.shared.data(from: url)

    // we got some data back!
    let items = try JSONDecoder().decode(Result.self, from:
data)

    // success – convert the array values to our pages array
    pages = items.query.pages.values.sorted { $0.title <
$1.title }
    loadingState = .loaded
} catch {
    // if we're still here it means the request failed somehow
    loadingState = .failed
}
}
```

That request should begin as soon as the view appears, so add this **task()** modifier after the existing **toolbar()** modifier:

```
.task {
    await fetchNearbyPlaces()
}
```

Now go ahead and run the app again – you'll find that as you drop a pin our **EditView** screen

Downloading data from Wikipedia

will slide up and show you all the places nearby. Nice!

Sorting Wikipedia results

Wikipedia's results come back to us in an order that probably seems random, but it's actually sorted according to their internal page ID. That doesn't help *us* though, which is why we're sorting results using a custom closure.

There are lots of times when using a custom sorting function is exactly what you need, but more often than not there is one natural order to your data – maybe showing news stories newest first, or contacts last name first, etc. So, rather than just provide an inline closure to `sorted()` we are instead going to make our **Page** struct conform to **Comparable**. This is actually pretty easy to do, because we already have the sorting code written – it's just a matter of moving it across to our **Page** struct.

So, start by modifying the definition of the **Page** struct to this:

```
struct Page: Codable, Comparable {
```

If you recall, conforming to **Comparable** has only a single requirement: we must implement a `<` function that accepts two parameters of the type of our struct, and returns true if the first should be sorted before the second. In this case we can just pass the test directly onto the `title` strings, so add this method to the **Page** struct now:

```
static func <(lhs: Page, rhs: Page) -> Bool {
    lhs.title < rhs.title
}
```

Now that Swift understands how to sort pages, it will automatically give us a parameter-less `sorted()` method on page arrays. This means when we set `self.pages` in `fetchNearbyPlaces()` we can now add `sorted()` to the end, like this:

```
pages = items.query.pages.values.sorted()
```

Before we're done with this screen, we need to replace the `Text("Page description here")`

view with something real. Wikipedia's JSON data does contain a description, but it's buried: the **terms** dictionary might not be there, and if it is there it might not have a **description** key, and if it *has* a **description** key it might be an empty array rather than an array with some text inside.

We don't want this mess to plague our SwiftUI code, so again the best thing to do is make a computed property that returns the description if it exists, or a fixed string otherwise. Add this to the **Page** struct to finish it off:

```
var description: String {
    terms?[ "description" ]?.first ?? "No further information"
}
```

With that done you can replace **Text("Page description here")** with this:

```
Text(page.description)
```

That completes **EditView** – it lets us edit the two properties of our annotation views, it downloads and sorts data from Wikipedia, it shows different UI depending on how the network request is going, and it even carefully looks through the Wikipedia content to decide what can be shown.

Introducing MVVM into your SwiftUI project

So far I've introduced you to a range of concepts across Swift and SwiftUI, and I've also dropped a few tips on ways to organize your code better. Well, here I want to explore that latter part a bit further: we're going to look at what is commonly called a *software architecture*, or the more grandiose name an *architectural design pattern* – really it's just a particular way of structuring your code.

The pattern we're going to look at is called MVVM, which is an acronym standing for Model View View-Model. This is a terrifically bad name, and thoroughly confuses people, but I'm afraid we're rather stuck with it at this point. There is no single definition of what is MVVM, and you'll find all sorts of people arguing about it online, but that's okay – here we're going to keep it simple, and use MVVM as a way of getting some of our program state and logic out of our view structs. We are, in effect, separating logic from layout.

We'll explore that definition as we go, but for now let's start with the big stuff: make a new Swift file called `ContentView-ViewModel.swift`, then give it an extra import for MapKit. We're going to use this to create a new class that manages our data, and manipulates it on behalf of the **ContentView** struct so that our view doesn't really care how the underlying data system works.

We're going to start with three trivial things, then build our way up from there. First, create a new class that conforms to the **ObservableObject** protocol, so we're able to report changes back to any SwiftUI view that's watching:

```
class ViewModel: ObservableObject {  
}
```

Second, I want you to place that new class *inside* an extension on **ContentView**, like this:

```
extension ContentView {  
    class ViewModel: ObservableObject {
```

```

    }
}
}
```

Now we're saying this isn't just *any* view model, it's the view model for **ContentView**. Later on it will be your job to add a second view model to handle **EditView**, so you can try seeing how the concepts map elsewhere.

The final small change I'd like you to make is to add a new attribute, **@MainActor**, to the whole class, like this:

```
extension ContentView {
    @MainActor class ViewModel: ObservableObject {
    }
}
```

The main actor is responsible for running all user interface updates, and adding that attribute to the class means we want all its code – any time it runs anything, unless we specifically ask otherwise – to run on that main actor. This is important because it's responsible for making UI updates, and those must happen on the main actor. In practice this isn't quite so easy, but we'll come to that later on.

Now, we've used **ObservableObject** classes before, but *didn't* have **@MainActor** – how come they worked? Well, behind the scenes whenever we use **@StateObject** or **@ObservedObject** Swift was silently inferring the **@MainActor** attribute for us – it knows that both those mean a SwiftUI view is relying on an external object to trigger its UI updates, and so it will make sure all the work automatically happens on the main actor without us asking for it.

However, that doesn't provide 100% safety. Yes, Swift will infer this when used from a SwiftUI view, but what if you access your class from somewhere else – from another class, for example? Then the code could run anywhere, which isn't safe. So, by adding the **@MainActor** attribute here we're taking a “belt and braces” approach: we're telling Swift every part of this class should run on the main actor, so it's safe to update the UI, no matter

Project 14: Bucket List

where it's used.

Now that we have our class in place, we get to choose which pieces of state from our view should be moved into the view model. Some people will tell you to move all of it, others will be more selective, and that's okay – again, there is no single of what MVVM looks like, so I'm going to provide you with the tools and knowledge to experiment yourself.

Let's start with the easy stuff: move all three `@State` properties in **ContentView** over to its view model, switching `@State private` for just `@Published` – they can't be private any more, because they explicitly need to be shared with **ContentView**:

```
extension ContentView {  
    @MainActor class ViewModel: ObservableObject {  
        @Published var mapRegion = MKCoordinateRegion(center:  
            CLLocationCoordinate2D(latitude: 50, longitude: 0), span:  
            MKCoordinateSpan(latitudeDelta: 25, longitudeDelta: 25))  
        @Published var locations = [Location]()  
        @Published var selectedPlace: Location?  
    }  
}
```

And now we can replace all those properties in **ContentView** with a single one:

```
@StateObject private var viewModel = ViewModel()
```

That will of course break a lot of code, but the fixes are easy – just add `viewModel` in various places. So, `$mapRegion` becomes `$viewModel.mapRegion`, `locations` becomes `viewModel.locations`, and so on.

Once you've added that everywhere it's needed your code will compile again, but you might wonder how this has helped – haven't we just moved our code from one place to another? Well, yes, but there is an important distinction that will become clearer as your skills grow: having all this functionality in a separate class makes it much easier to write tests for your

code.

Views work best when they handle presentation of data, meaning that *manipulation* of data is a great candidate for code to move into a view model. With that in mind, if you have a look through your **ContentView** code you might notice two places our view does more work than it ought to: adding a new location and updating an existing location, both of which root around inside the internal data of our view model.

Reading data from a view model's properties is usually fine, but *writing* it isn't because the whole point of this exercise is to separate logic from layout. You can find these two places immediately if we clamp down on writing view model data – modify the **locations** property in your view model to this:

```
@Published private(set) var locations = [Location]()
```

Now we've said that reading locations is fine, but only the class itself can *write* locations. Immediately Xcode will point out the two places where we need to get code out of the view: adding a new location, and updating an existing one.

So, we can start by adding a new method to the view model to handle adding a new location:

```
func addLocation() {
    let newLocation = Location(id: UUID(), name: "New location",
description: "", latitude: mapRegion.center.latitude,
longitude: mapRegion.center.longitude)
    locations.append(newLocation)
}
```

That can then be used from the + button in **ContentView**:

```
Button {
    viewModel.addLocation()
} label: {
    Image(systemName: "plus")
```

Project 14: Bucket List

}

The second problematic place is updating a location, so I want you to cut that whole **if let index** check to your clipboard, then paste it into a new method in the view model, adding in a check that we have a selected place to work with:

```
func update(location: Location) {
    guard let selectedPlace = selectedPlace else { return }

    if let index = locations.firstIndex(of: selectedPlace) {
        locations[index] = location
    }
}
```

Make sure and remove the two **viewModel** references from there – they aren’t needed any more.

Now the **EditView** sheet in **ContentView** can just pass its data onto the view model:

```
EditView(location: place) {
    viewModel.update(location: $0)
}
```

At this point the view model has taken over all aspects of **ContentView**, which is great: the view is there to present data, and the view model is there to *manage* data. The split isn’t always quite as clean as that, despite what you might hear elsewhere online, and again that’s okay – once you move into more advanced projects you’ll find that “one size fits all” approaches usually fit nobody, so we just do our best with what we have.

Anyway, in this case now that we have our view model all set up, we can upgrade it to support loading and saving of data. This will look in the documents directory for a particular file, then use either **JSONEncoder** or **JSONDecoder** to convert it ready for use.

Previously I showed you how to find our app's documents directory with a reusable function, but here we're going to package it up as an extension on **FileManager** for easier access in any project.

Create a new Swift file called FileManager-DocumentsDirectory.swift, then give it this code:

```
extension FileManager {
    static var documentsDirectory: URL {
        let paths =
FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)
        return paths[0]
    }
}
```

Now we can create a URL to a file in our documents directory wherever we want, however I *don't* want to do that when both loading and saving files because it means if we ever change our save location we need to remember to update both places.

So, a better idea is to add a new property to our view model to store the location we're saving to:

```
let savePath =
FileManager.documentsDirectory.appendingPathComponent("SavedPlaces")
```

And with that in place we can create a new initializer and a new **save()** method that makes sure our data is persisted automatically. Start by adding this to the view model:

```
init() {
    do {
        let data = try Data(contentsOf: savePath)
        locations = try JSONDecoder().decode([Location].self, from:
data)
```

Project 14: Bucket List

```
    } catch {
        locations = []
    }
}
```

As for saving, previously I showed you how to write a string to disk, but the **Data** version is even better because it lets us do something quite amazing in just one line of code: we can ask iOS to ensure the file is written with encryption so that it can only be read once the user has unlocked their device. This is in *addition* to requesting atomic writes – iOS does almost all the work for us.

Add this method to the view model now:

```
func save() {
    do {
        let data = try JSONEncoder().encode(locations)
        try data.write(to: savePath, options:
            [.atomic, .completeFileProtection])
    } catch {
        print("Unable to save data.")
    }
}
```

Yes, all it takes to ensure that the file is stored with strong encryption is to add **.completeFileProtection** to the data writing options.

Using this approach we can write any amount of data in any number of files – it's much more flexible than **UserDefaults**, and if we need it also allows us to load and save data as needed rather than immediately when the app launches as with **UserDefaults**.

Before we're done with this step, we need to make a handful of small changes to our view model so that uses the code we just wrote.

First, the **locations** array no longer needs to be initialized to an empty array, because that's handled by the initializer. Change it to this:

```
@Published private(set) var locations: [Location]
```

And second, we need to call the **save()** method after adding a new location or after updating an existing one, so add **save()** to the end of both those methods.

Go ahead and run the app now, and you should find that you can add items freely, then relaunch the app to see them restored just as they were.

That took quite a bit of code in total, but the end result is that we have loading and saving done really well:

- All the logic is handled outside the view, so later on when you learn to write tests you'll find the view model is much easier to work with.
- When we write data we're making iOS encrypt it so the file can't be read or written until the user unlocks their device.
- The load and save process is almost transparent – we added one modifier and changed another, and that's all it took.

Of course, our app isn't truly secure yet: we've ensured our data file is saved out using encryption so that it can only be read once the device has been unlocked, but there's nothing stopping someone else from reading the data afterwards.

Locking our UI behind Face ID

To finish off our app, we’re going to make one last important change: we’re going to require the user to authenticate themselves using either Touch ID or Face ID in order to see all the places they have marked on the app. After all, this is their private data and we should be respectful of that, and of course it gives me a chance to let you use an important skill in a practical context!

First we need some new state in our view model that tracks whether the app is unlocked or not. So, start by adding this new property:

```
@Published var isUnlocked = false
```

Second, we need to add the Face ID permission request key to our project configuration options, explaining to the user why we want to use Face ID. If you haven’t added this already, go to your target options now, select the Info tab, then right-click on any existing row and add the “Privacy - Face ID Usage Description” key there. You can enter what you like, but “Please authenticate yourself to unlock your places” seems like a good choice.

Third, we need to add **import LocalAuthentication** to the top of your view model’s file, so we have access to Apple’s authentication framework.

And now for the hard part. If you recall, the code for biometric authentication was a teensy bit unpleasant because of its Objective-C roots, so it’s always a good idea to get it far away from the neatness of SwiftUI. So, we’re going to write a dedicated **authenticate()** method that handles all the biometric work:

1. Creating an **LAContext** so we have something that can check and perform biometric authentication.
2. Ask it whether the current device is capable of biometric authentication.
3. If it is, start the request and provide a closure to run when it completes.
4. When the request finishes, check the result.
5. If it was successful, we’ll set **isUnlocked** to true so we can run our app as normal.

Add this method to your view model now:

```
func authenticate() {
    let context = LAContext()
    var error: NSError?

    if context.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, error: &error) {
        let reason = "Please authenticate yourself to unlock your places."
        context.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, localizedReason: reason) { success, authenticationError in
            if success {
                self.isUnlocked = true
            } else {
                // error
            }
        }
    } else {
        // no biometrics
    }
}
```

Remember, the string in our code is used for Touch ID, whereas the string in Info.plist is used for Face ID.

And now we need to make an adjustment that is in reality very small, but can be hard to visualize if you're reading this rather than watching the video. Everything inside the **ZStack**

Project 14: Bucket List

needs to be indented in by one level, and have this placed before it:

```
if viewModel.isUnlocked {
```

Just before the end of the **ZStack** add this:

```
} else {
    // button here
}
```

So, it should look something like this:

```
ZStack {
    if viewModel.isUnlocked {
        MapView...
        Circle...
        VStack...
    } else {
        // button here
    }
}
.sheet(item: $viewModel.selectedPlace) { place in
```

So now all we need to do is fill in the **// button here** comment with an actual button that triggers the **authenticate()** method. You can design whatever you want, but something like this ought to be enough:

```
Button("Unlock Places") {
    viewModel.authenticate()
}
.padding()
.background(.blue)
.foregroundColor(.white)
```

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

You can now go ahead and run the app again, because our code is almost done. If this is the first time you've used Face ID in the simulator you'll need to go to the Features menu and choose Face ID > Enrolled, but once you relaunch the app you can authenticate using Features > Face ID > Matching Face.

However, when it runs you might notice a problem: the app will seem to work just fine, but Xcode is likely to show a warning message in its debug output. More importantly, it will also show a purple warning, which is Xcode's issue of flagging up *runtime issues* - when our code does something it really ought not to.

In this instead, it should point at this line in our view model:

```
self.isUnlocked = true
```

Next to that it should say “publishing changes from background threads is not allowed”, which translated means “you’re trying to change the UI but you’re not doing it from the main actor and that’s going to cause problems.”

Now, This might be confusing given that earlier on we specifically added the **@MainActor** attribute to our whole class, which I said means all the code from the class will be run on the main actor and therefore be safe for UI updates. However, I added an important proviso there: “unless we specifically request otherwise.”

In this instance we *did* request otherwise, but it might not be obvious: when we asked Face ID to do the work of authenticating the user, this happens outside of our program – it’s not us doing the actual face check, it’s Apple. When that process completes Apple will call our completion closure to say whether it succeeded or not, but that *won’t* be called on the main actor despite our **@MainActor** attribute.

The solution here is to make sure we change the **isUnlocked** property on the main actor. This can be done by starting a new task, then calling **await MainActor.run()** inside there, like this:

Project 14: Bucket List

```
if success {
    Task {
        await MainActor.run {
            self.isUnlocked = true
        }
    }
} else {
    // error
}
```

That effectively means “start a new background task, then immediately use that background task to queue up some work on the main actor.”

That works, but we can do better: we can tell Swift that our task’s code needs to run directly on the main actor, by giving the closure itself the **@MainActor** attribute. So, rather than bouncing to a background task then back to the main actor, the new task will immediately start running on the main actor:

```
if success {
    Task { @MainActor in
        self.isUnlocked = true
    }
} else {
    // error
}
```

And with that our code is done, and that’s another app complete – good job!

Bucket List: Wrap up

This was our biggest project yet, but we've covered a huge amount of ground: adding **Comparable** to custom types, finding the documents directory, integrating MapKit, using biometric authentication, secure **Data** writing, and much more. And of course you have another real app, and hopefully you're able to complete the challenges below to take it further.

Although this exact project we made places maps at the center of its existence, they also extremely useful as little bonuses elsewhere – showing where a meeting takes place, or the location of a friend, etc, can add just that extra bit of useful detail to your other projects.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Our + button is rather hard to tap. Try moving all its modifiers to the image inside the button – what difference does it make, and can you think why?
2. Our app silently fails when errors occur during biometric authentication, so add code to show those errors in an alert.
3. Create another view model, this time for **EditView**. What you put in the view model is down to you, but I would recommend leaving **dismiss** and **onSave** in the view itself – the former uses the environment, which can only be read by the view, and the latter doesn't really add anything when moved into the model.

Project 14: Bucket List

Tip: That last challenge will require you to make **StateObject** instance in your **EditView** initializer – remember to use an underscore with the property name!

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here: [Solution to Bucket List](#). If you don't already subscribe, you can start a free trial today.

Project 15

Accessibility

Learn how to make your apps available to everyone

Accessibility: Introduction

Making your app accessible means taking steps to ensure that everyone can use it fully regardless of their individual needs. For example, if they are blind then your app should work well with the system's VoiceOver system to ensure your UI can be read smoothly.

SwiftUI gives us a huge amount of functionality for free, because its layout system of **VStack** and **HStack** naturally forms a flow of views. However, it isn't perfect, and any time you can add some extra information to help out the iOS accessibility system it's likely to help.

Usually the best way to test out your app is to enable VoiceOver support and run the app on real device – if your app works great with VoiceOver, there's a good chance you're already far ahead of the average for iOS apps.

Anyway, in this technique project we're going to look at a handful of accessibility techniques, then look at some of the previous projects we made to see how they might get upgraded.

For now, please create a new iOS app using the App template. You should run this project on a real device, so you can enable VoiceOver for real.

Acknowledgements: I'm grateful for the help of Robin Kipp in preparing this chapter – he wrote in with some detailed suggestions for things he'd like to see with regards to accessibility, giving me some great examples of how it would affect his own personal use.

Identifying views with useful labels

In the files for this project I have placed four pictures downloaded from Unsplash. Unsplash filenames are made up of a picture ID and the photographer's name, so if you drag them into your asset catalog you'll see they have names such as "ales-krivec-15949" and so on. That in itself isn't a problem, and in fact I think it can be a helpful way of remembering where assets came from. However, it does present a problem for screen readers.

To get started with VoiceOver, we're going to create a simple view that cycles randomly through the four pictures in our asset catalog. Modify the **ContentView** struct to this:

```
struct ContentView: View {
    let pictures = [
        "ales-krivec-15949",
        "galina-n-189483",
        "kevin-horstmann-141705",
        "nicolas-tissot-335096"
    ]

    @State private var selectedPicture = Int.random(in: 0...3)

    var body: some View {
        Image(pictures[selectedPicture])
            .resizable()
            .scaledToFit()
            .onTapGesture {
                selectedPicture = Int.random(in: 0...3)
            }
    }
}
```

There's nothing complicated there, but it already helps to illustrate two serious problems.

Project 15: Accessibility

If you haven't already enabled VoiceOver in the Settings app on your iOS device, please do so now: Settings > Accessibility > VoiceOver, then toggle it on. Alternatively, you can activate Siri at any time and ask to enable or disable VoiceOver.

Important: Immediately below the VoiceOver toggle is instructions for how to use it. The regular taps and swipes you're used to no longer function the same way, so read those instructions!

Now launch our app on your device, and try tapping once on the picture to activate it. If you listen carefully to VoiceOver you should hear two problems:

1. Reading out "Kevin Horstmann one four one seven zero five" is not only unhelpful for the user because it doesn't describe the picture at all, but it's actually confusing – the long string of numbers does more harm than good.
2. After reading the above string, VoiceOver then says "image". This is true, it *is* an image, but it's also acting as a button because we added an `onTapGesture()` modifier.

The first of those problems is a side effect of SwiftUI trying to give us sensible behavior out of the box: when given an image, it automatically uses the image's filename as the text to read out.

We can control what VoiceOver reads for a given view by attaching two modifiers: `.accessibilityLabel()` and `.accessibilityHint()`. They both take text containing anything we want, but they serve different purposes:

- The *label* is read immediately, and should be a short piece of text that gets right to the point. If this view deletes an item from the user's data, it might say "Delete".
- The *hint* is read after a short delay, and should provide more details on what the view is there for. It might say "Deletes an email from your inbox", for example.

The label is exactly what we need to solve the first of our problems, because it means we can leave the image name as it is while still having VoiceOver read out something that helps users.

First, add this second array of image descriptions as a property for **ContentView**:

```
let labels = [  
    "Tulips",  
    "Frozen tree buds",  
    "Sunflowers",  
    "Fireworks",  
]
```

And now attach this modifier to the image:

```
.accessibilityLabel(labels[selectedPicture])
```

This allows VoiceOver to read the correct label no matter what image is present. Of course, if your image wasn't randomly changing you could just type your label directly into the modifier.

The second problem is that the image is identified as an image. This is self-evidently true, but it's also not helpful because we've attached a tap gesture to it so it's effectively a button.

We can fix this second problem using another modifier, **.accessibilityAddTraits()**. This lets us provide some extra behind the scenes information to VoiceOver that describes how the view works, and in our case we can tell it that our image is also a button by adding this modifier:

```
.accessibilityAddTraits(.isButton)
```

If you wanted, you could remove the image trait as well, because it isn't really adding much:

```
.accessibilityRemoveTraits(.isImage)
```

With these changes in place our UI works much better: VoiceOver now reads a useful description of the image's contents, and also makes users aware the image is also a button.

Hiding and grouping accessibility data

If you spend even a few minutes with an active VoiceOver user, you'll learn two things very quickly: they are remarkably adept at navigating around user interfaces, and they also often set reading speed extremely fast – way faster than you or I would use.

It's important to take both of those things into account when we're designing our UI: these users aren't just trying VoiceOver out of curiosity, but are instead VoiceOver power users who rely on it to access your app. As a result, it's important we ensure our UI removes as much clutter as possible so that users can navigate through it quickly and not have to listen to VoiceOver reading unhelpful descriptions.

Beyond setting labels and hints, there are several ways we can control what VoiceOver reads out. There are three in particular I want to focus on:

- Marking images as being unimportant for VoiceOver.
- Hiding views from the accessibility system.
- Grouping several views as one.

All of these are simple changes to make, but they result in a big improvement.

For example, we can tell SwiftUI that a particular image is just there to make the UI look better by using **Image(decorative:)**. Whether it's a simple bullet point or an animation of your app's mascot character running around, it doesn't actually convey any information and so **Image(decorative:)** tells SwiftUI it should be ignored by VoiceOver.

Use it like this:

```
Image(decorative: "character")
```

This leaves the image as being accessible to VoiceOver if it has some important traits, such as **.isButton** – it will say “button” when it's highlighted, and if we attach a tap gesture that

works – but it *doesn't* read out the image's filename as the automatic VoiceOver label. If you then add a label or a hint that *will* be read.

If you want to go a step further, you can use the `.accessibilityHidden()` modifier, which makes any view completely invisible to the accessibility system:

```
Image(decorative: "character")
    .accessibilityHidden(true)
```

With that modifier the image becomes invisible to VoiceOver regardless of what traits it has. Obviously you should only use this if the view in question really does add nothing – if you had placed a view offscreen so that it wasn't currently visible to users, you should mark it inaccessible to VoiceOver too.

The last way to hide content from VoiceOver is through *grouping*, which lets us control how the system reads several views that are related. As an example, consider this layout:

```
VStack {
    Text("Your score is")
    Text("1000")
    .font(.title)
}
```

VoiceOver sees that as two unrelated text views, and so it will either read “Your score is” or “1000” depending on what the user has selected. Both of those are unhelpful, which is where the `.accessibilityElement(children:)` modifier comes in: we can apply it to a parent view, and ask it to combine children into a single accessibility element.

For example, this will cause both text views to be read together:

```
VStack {
    Text("Your score is")
    Text("1000")
    .font(.title)
```

Project 15: Accessibility

```
}
```

```
.accessibilityElement(children: .combine)
```

That works really well when the child views contain separate information, but in our case the children really should be read as a single entity. So, the better solution here is to use `.accessibilityElement(children: .ignore)` so the child views are invisible to VoiceOver, then provide a custom label to the parent, like this:

```
VStack {
    Text("Your score is")
    Text("1000")
        .font(.title)
}
.accessibilityElement(children: .ignore)
.accessibilityLabel("Your score is 1000")
```

It's worth trying both of these to see how they differ in practice. Using `.combine` adds a pause between the two pieces of text, because they aren't necessarily designed to be read together. Using `.ignore` and a custom label means the text is read all at once, and is much more natural.

Tip: `.ignore` is the default parameter for `children`, so you can get the same results as `.accessibilityElement(children: .ignore)` just by saying `.accessibilityElement()`.

Reading the value of controls

By default SwiftUI provides VoiceOver readouts for its user interface controls, and although these are often good sometimes they just don't fit with what you need. In these situations we can use the **accessibilityValue()** modifier to separate a control's *value* from its *label*, but we can also specify custom swipe actions using **accessibilityAdjustableAction()**.

For example, you might build a view that shows some kind of input controlled by various buttons, like a custom stepper:

```
struct ContentView: View {
    @State private var value = 10

    var body: some View {
        VStack {
            Text("Value: \(value)")

            Button("Increment") {
                value += 1
            }

            Button("Decrement") {
                value -= 1
            }
        }
    }
}
```

That might work just the way you want with tap interactions, but it's not a great experience with VoiceOver because all users will hear is "Increment" or "Decrement" every time they tap one of the buttons.

To fix this we can give iOS specific instructions for how to handle adjustment, by grouping

Project 15: Accessibility

our **VStack** together using **accessibilityElement()** and **accessibilityLabel()**, then by adding the **accessibilityValue()** and **accessibilityAdjustableAction()** modifiers to respond to swipes with custom code.

Adjustable actions hand us the direction the user swiped, and we can respond however we want. There is one proviso: yes, we can choose between increment and decrement swipes, but we also need a special default case to handle unknown future values – Apple has reserved the right to add other kinds of adjustments in the future.

Here's how it looks in code:

```
 VStack {
    Text("Value: \(value)")

    Button("Increment") {
        value += 1
    }

    Button("Decrement") {
        value -= 1
    }
}

.accessibilityElement()
.accessibilityLabel("Value")
.accessibilityValue(String(value))
.accessibilityAdjustableAction { direction in
    switch direction {
        case .increment:
            value += 1
        case .decrement:
            value -= 1
        default:
            print("Not handled.")
    }
}
```

```
    }  
}
```

That lets users select the whole **VStack** to have “Value: 10” read out, but then they can swipe up or down to manipulate the value and have just the numbers read out – it’s a much more natural way of working.

Fixing Guess the Flag

Way back in project 2 we made Guess the Flag, which showed three flag pictures and asked the users to guess which was which. Well, based on what you now know about VoiceOver, can you spot the fatal flaw in our game?

That's right: SwiftUI's default behavior is to read out the image names as their VoiceOver label, which means anyone using VoiceOver can just move over our three flags to have the system announce which one is correct.

To fix this we need to add text descriptions for each of our flags, describing them in enough detail that they can be guessed correctly by someone who has learned them, but of course without actually giving away the name of the country.

If you open your copy of this project, you'll see it was written to use an array of country names, like this:

```
@State private var countries = ["Estonia", "France", "Germany",
"Ireland", "Italy", "Nigeria", "Poland", "Russia", "Spain",
"UK", "US"].shuffled()
```

So, the easiest way to attach labels there – the way that doesn't require us to change any of our code – is to create a dictionary with country names as keys and accessibility labels as values, like this. Please add this to **ContentView**:

```
let labels = [
    "Estonia": "Flag with three horizontal stripes of equal size.
Top stripe blue, middle stripe black, bottom stripe white",
    "France": "Flag with three vertical stripes of equal size.
Left stripe blue, middle stripe white, right stripe red",
    "Germany": "Flag with three horizontal stripes of equal size.
Top stripe black, middle stripe red, bottom stripe gold",
    "Ireland": "Flag with three vertical stripes of equal size.
Left stripe green, middle stripe white, right stripe orange",
```

```

    "Italy": "Flag with three vertical stripes of equal size.  

Left stripe green, middle stripe white, right stripe red",  

    "Nigeria": "Flag with three vertical stripes of equal size.  

Left stripe green, middle stripe white, right stripe green",  

    "Poland": "Flag with two horizontal stripes of equal size.  

Top stripe white, bottom stripe red",  

    "Russia": "Flag with three horizontal stripes of equal size.  

Top stripe white, middle stripe blue, bottom stripe red",  

    "Spain": "Flag with three horizontal stripes. Top thin stripe  

red, middle thick stripe gold with a crest on the left, bottom  

thin stripe red",  

    "UK": "Flag with overlapping red and white crosses, both  

straight and diagonally, on a blue background",  

    "US": "Flag with red and white stripes of equal size, with  

white stars on a blue background in the top-left corner"  

]

```

And now all we need to do is add the **accessibilityLabel()** modifier to the flag images. I realize that sounds simple, but the code has to do three things:

1. Use **countries[number]** to get the name of the country for the current flag.
2. Use that name as the key for **labels**.
3. Provide a string to use as a default if somehow the country name doesn't exist in the dictionary. (This should never happen, but there's no harm being safe!)

Putting all that together, put this modifier directly below the rest of the modifiers for the flag images:

```
.accessibilityLabel(labels[countries[number]], default: "Unknown  

flag")
```

And now if you run the game again you'll see it actually *is* a game, regardless of whether you use VoiceOver or not. This gets right to the core of accessibility: everyone can have fun

Project 15: Accessibility

playing this game now, regardless of their access needs.

Fixing Word Scramble

In project 5 we built Word Scramble, a game where users were given a random eight-letter word and had to produce new words using its letters. This mostly works great with VoiceOver: no parts of the app are *inaccessible*, although that doesn't mean we can't do better.

To see an obvious pain point, try adding a word. You'll see it slide into the table underneath the prompt, but if you tap into it with VoiceOver you'll realize it isn't read well: the letter count is read as "five circle", and the text is a separate element.

There are a few ways of improving this, but probably the best is to make both those items a single group where the children are ignored by VoiceOver, then add a label for the whole group that contains a much more natural description.

Our current code looks like this:

```
List(usedWords, id: \.self) { word in
    HStack {
        Image(systemName: "\((word.count).circle)")
        Text(word)
    }
}
```

To fix this we need to group the elements inside the **HStack** together so we can apply our VoiceOver customization:

```
List(usedWords, id: \.self) { word in
    HStack {
        Image(systemName: "\((word.count).circle)")
        Text(word)
    }
    .accessibilityElement(children: .ignore)
    .accessibilityLabel("\((word), \((word.count) letters")
}
```

Project 15: Accessibility

Alternatively, you could break that text up to have a hint as well as a label, like this:

```
HStack {  
    Image(systemName: "\((word.count).circle")  
    Text(word)  
}  
.accessibilityElement(children: .ignore)  
.accessibilityLabel(word)  
.accessibilityHint("\((word.count) letters")
```

Regardless of which you choose, if you try the game again you'll hear it now reads “spill, five letters”, which is much better.

Fixing Bookworm

In project 11 we built Bookworm, an app that lets users store ratings and descriptions for books they had read, and we also introduced a custom **RatingView** UI component that showed star ratings from 1 to 5.

Again, most of the app does well with VoiceOver, but that rating control is a hard fail – it uses tap gestures to add functionality, so users won’t realize they are buttons, and it doesn’t even convey the fact that they are supposed to represent ratings. For example, if I tap one of the gray stars, VoiceOver reads out to me, “star, fill, image, possibly airplane, star” – it’s really not useful at all.

That in itself is a problem, but it’s *extra* problematic because our **RatingView** is designed to be reusable – it’s the kind of thing you can take from this project and use in a dozen other projects, and that just means you end polluting many apps with poor accessibility.

We’re going to tackle this one in an unusual way: first with a simpler set of modifiers that do an okay job, but then by seeing how we can use **accessibilityAdjustableAction()** to get a more optimal result.

Our initial approach will use three modifiers, each added below the current **tapGesture()** modifier inside **RatingView**. First, we need to add one that provides a meaningful label for each star, like this:

```
.accessibilityLabel(" \(number == 1 ? "1 star" : "\ \(number) stars")")
```

Second, we can remove the **.isImage** trait, because it really doesn’t matter that these are images:

```
.accessibilityRemoveTraits(.isImage)
```

And finally, we should tell the system that each star is actually a button, so users know it can be tapped. While we’re here, we can make VoiceOver do an even better job by adding a

Project 15: Accessibility

second trait, `.isSelected`, if the star is already highlighted.

So, add this final modifier beneath the previous two:

```
.accessibilityAddTraits(number > rating ? .isButton :  
[.isButton, .isSelected])
```

It only took three small changes, but this improved component is much better than what we had before.

This initial approach works well enough, and it's certainly the easiest to take because it builds on all the skills you've used elsewhere. However, there's a second approach that I want to look at, because I think it yields a far more *useful* result – it works more efficiently for folks relying on VoiceOver and other tools.

First, remove the three modifiers we just added, and instead add these four to the `HStack`:

```
.accessibilityElement()  
.accessibilityLabel(label)  
.accessibilityValue(rating == 1 ? "1 star" : "\u2022(\rating) stars")  
.accessibilityAdjustableAction { direction in  
    switch direction {  
        case .increment:  
            if rating < maximumRating { rating += 1 }  
        case .decrement:  
            if rating > 1 { rating -= 1 }  
        default:  
            break  
    }  
}
```

That groups all its children together, applies the label “Rating”, but then adds a value based on the current stars. It also allows that rating value to be adjusted up or down using swipes, which is much better than trying to work through lots of individual images.

Accessibility: Wrap up

Accessibility isn't something that's "nice to have" – it should be regarded as a fundamental part of your application design, and considered from the very beginning onwards. SwiftUI didn't get its excellent accessibility support because Apple thought about it at the last minute, but instead because it got baked in from the start – every part of SwiftUI was crafted with accessibility in mind, and we'd be doing a great disservice to our users if we didn't match that same standard.

What's more, I hope you can agree that adding extra accessibility is surprisingly easy – some special values here, a little grouping there, and some bonus traits are all simple things that take only minutes to add, but are the difference between "opaque" and "easy to use" for millions of people around the world.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. The check out view in Cupcake Corner uses an image and loading spinner that don't add anything to the UI, so find a way to make the screenreader not read them out.
2. Fix the list rows in iExpense so they read out the name and value in one single VoiceOver label, and their type in a hint.
3. Do a full accessibility review of Moonshot – what changes do you need to make so that it's fully accessible?

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Accessibility. If you don't already subscribe, you can start a free trial today.

Project 16

Hot Prospects

Build an app for conferences with tabs, context menus, and more.

Hot Prospects: Introduction

In this project we're going to build Hot Prospects, which is an app to track who you meet at conferences. You've probably seen apps like it before: it will show a QR code that stores your attendee information, then others can scan that code to add you to their list of possible leads for later follow up.

That might sound easy enough, but along the way we're going to cover stacks of really important new techniques: creating tab bars and context menus, sharing custom data using the environment, sending custom change notifications, and more. The resulting app is awesome, but what you learn along the way will be particularly useful!

As always we have lots of techniques to cover before we get into the implementation of our project, so please start by creating a new iOS project using the App template, naming it HotProspects.

Let's get to it!

Reading custom values from the environment with `@EnvironmentObject`

You've seen how `@State` is used to work with state that is local to a single view, and how `@ObservedObject` lets us pass one object from view to view so we can share it. Well, `@EnvironmentObject` takes that one step further: we can place an object into the environment so that any child view can automatically have access to it.

Imagine we had multiple views in an app, all lined up in a chain: view A shows view B, view B shows view C, C shows D, and D shows E. View A and E both want to access the same object, but to get from A to E you need to go through B, C, and D, and they *don't* care about that object. If we were using `@ObservedObject` we'd need to pass our object from each view to the next until it finally reached view E where it could be used, which is annoying because B, C, and D don't care about it. With `@EnvironmentObject` view A can put the object into the environment, view E can read the object out from the environment, and views B, C, and D don't have to know anything happened – it's much nicer.

There's one last thing before I show you some code: environment objects use the same `ObservableObject` protocol you've already learned, and SwiftUI will automatically make sure all views that share the same environment object get updated when it changes.

OK, let's look at some code that shows how to share data between two views using environment objects. First, here's some basic data we can work with:

```
@MainActor class User: ObservableObject {
    @Published var name = "Taylor Swift"
}
```

As you can see, that uses `@MainActor`, `ObservableObject`, and `@Published` just like we've learned previously – all that knowledge you built up continues to pay off.

Reading custom values from the environment with `@EnvironmentObject`

Next we can define two SwiftUI views to use our new class. These will use the `@EnvironmentObject` property wrapper to say that the value of this data comes from the environment rather than being created locally:

```
struct EditView: View {
    @EnvironmentObject var user: User

    var body: some View {
        TextField("Name", text: $user.name)
    }
}

struct DisplayView: View {
    @EnvironmentObject var user: User

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

That `@EnvironmentObject` property wrapper will automatically look for a `User` instance in the environment, and place whatever it finds into the `user` property. **If it can't find a `User` in the environment your code will just crash, so please be careful.**

We can now bring those two views together in one place, and send in a `User` instance for them to work with:

```
struct ContentView: View {
    @StateObject private var user = User()

    var body: some View {
        VStack {
            EditView().environmentObject(user)
        }
    }
}
```

Project 16: Hot Prospects

```
        DisplayView().environmentObject(user)
    }
}
}
```

And that's all it takes to get our code working – you can run the app now and change the textfield to see its value appear in the text view below. Of course, we could have represented this in a single view, but this way you can see exactly how seamless the communication is when using environment objects.

Now, here's the clever part. Try rewriting the **body** property of **ContentView** to this”

```
 VStack {
    EditView()
    DisplayView()
}
.environmentObject(user)
```

What you'll find is that it works *identically*. We're now placing **user** into the environment of **ContentView**, but because both **EditView** and **DisplayView** are children of **ContentView** they inherit its environment automatically.

Tip: Given that we are explicitly sharing our **User** instance with other views, I would personally be inclined to remove the **private** access control because it's not accurate.

Now, you might wonder how SwiftUI makes the connection between **.environmentObject(user)** and **@EnvironmentObject var user: User** – how does it know to place that object into the correct property?

Well, you've seen how dictionaries let us use one type for the key and another for the value. The environment effectively lets us use data types themselves for the key, and instances of the type as the value. This is a bit mind bending at first, but imagine it like this: the keys are things like **Int**, **String**, and **Bool**, with the values being things like 5, “Hello”, and true, which means

Reading custom values from the environment with `@EnvironmentObject`

we can say “give me the `Int`” and we’d get back 5.

Creating tabs with TabView and tabItem()

Navigation views are great for letting us create hierarchical stacks of views that let users drill down into data, but they don't work so well for showing unrelated data. For that we need to use SwiftUI's **TabView**, which creates a button strip across the bottom of the screen, where tapping each button shows a different view.

Placing tabs inside a **TabView** is as simple as listing them out one by one, like this:

```
TabView {  
    Text("Tab 1")  
    Text("Tab 2")  
}
```

However, in practice you will always want to customize the way the tabs are shown – in the code above the tab bar will be an empty gray space. Although you *can* tap on the left and right parts of that gray space to activate the two tabs, it's a pretty terrible user experience.

Instead, it's a better idea to attach the **tabItem()** modifier to each view that's inside a **TabView**. This lets you customize the way the view is shown in the tab bar, providing an image and some text to show next to it like this:

```
TabView {  
    Text("Tab 1")  
        .tabItem {  
            Label("One", systemImage: "star")  
        }  
  
    Text("Tab 2")  
        .tabItem {  
            Label("Two", systemImage: "circle")  
        }  
}
```

```

    }
}

```

As well as letting the user switch views by tapping on their tab item, SwiftUI also allows us to control the current view programmatically using state. This takes four steps:

1. Create an **@State** property to track the tab that is currently showing.
2. Modify that property to a new value whenever we want to jump to a different tab.
3. Pass that as a binding into the **TabView**, so it will be tracked automatically.
4. Tell SwiftUI which tab should be shown for each value of that property.

The first three of those are simple enough, so let's get them out of the way. First, we need some state to track the current tab, so add this as a property to **ContentView**:

```
@State private var selectedTab = "One"
```

Second, we need to modify that somewhere, which will ask SwiftUI to switch tabs. In our little demo we could attach an **onTapGesture()** modifier to the first tab, like this:

```
Text("Tab 1")
    .onTapGesture {
        selectedTab = "Two"
    }
    .tabItem {
        Label("One", systemImage: "star")
    }
}
```

Third, we need to bind the selection of the **TabView** to **\$selectedTab**. This is passed as a parameter when we create the **TabView**, so update your code to this:

```
TabView(selection: $selectedTab) {
```

Now for the interesting part: when we say **selectedTab = "Two"** how does SwiftUI know

Project 16: Hot Prospects

which tab that represents? You might think that the tabs could be treated as an array, in which case the second tab would be at index 1, but that causes all sorts of problems: what if we move that tab to a different position in the tab view?

At a deeper level, it also breaks apart one of the core SwiftUI concepts: that we should be able to compose views freely. If tab 1 was the second item in the array, then:

1. Tab 0 is the first tab.
2. Tab 1 is the second tab.
3. Tab 0 has an **onTapGesture()** that shows tab 1.
4. Therefore tab 0 has intimate knowledge of how its parent, the **TabView**, is configured.

This is A Very Bad Idea, and so SwiftUI offers a better solution: we can attach a unique identifier to each view, and use *that* for the selected tab. These identifiers are called tags, and are attached using the **tag()** modifier like this:

```
Text("Tab 2")
    .tabItem {
        Image(systemName: "circle")
        Text("Two")
    }
    .tag("Two")
```

So, our entire view would be this:

```
struct ContentView: View {
    @State private var selectedTab = "One"

    var body: some View {
        TabView(selection: $selectedTab) {
            Text("Tab 1")
                .onTapGesture {
                    selectedTab = "Two"
                }
        }
    }
}
```

Creating tabs with TabView and tabItem()

```
        }
        .tabItem {
            Label("One", systemImage: "star")
        }
        .tag("One")

    Text("Tab 2")
    .tabItem {
        Label("Two", systemImage: "circle")
    }
    .tag("Two")
}
}
```

And now that code works: you can switch between tabs by pressing on their tab items, or by activating our tap gesture in the first tab.

Of course, just using “One” and “Two” isn’t ideal – those values are fixed and so it solves the problem of views being moved around, but they aren’t easy to remember. Fortunately, you can use whatever values you like instead: give each view a string tag that is unique and reflects its purpose, then use that for your **@State** property. This is much easier to work with in the long term, and is recommended over integers.

Tip: It’s common to want to use **NavigationView** and **TabView** at the same time, but you should be careful: **TabView** should be the parent view, with the tabs inside it having a **NavigationView** as necessary, rather than the other way around.

Manually publishing ObservableObject changes

Classes that conform to the **ObservableObject** protocol can use SwiftUI's **@Published** property wrapper to automatically announce changes to properties, so that any views using the object get their **body** property reinvoked and stay in sync with their data. That works really well a lot of the time, but sometimes you want a little more control and SwiftUI's solution is called **objectWillChange**.

Every class that conforms to **ObservableObject** automatically gains a property called **objectWillChange**. This is a *publisher*, which means it does the same job as the **@Published** property wrapper: it notifies any views that are observing that object that something important has changed. As its name implies, this publisher should be triggered immediately before we make our change, which allows SwiftUI to examine the state of our UI and prepare for animation changes.

To demonstrate this we're going to build an **ObservableObject** class that updates itself 10 times. We're going to use a method called **DispatchQueue.main.asyncAfter()**, which lets us run an attached closure after a delay of our choosing, which means we can say "do this work after 1 second" rather than "do this work now."

In this test case, we're going to use **asyncAfter()** inside a loop from 1 through 10, so we increment an integer 10 values. That integer will be wrapped using **@Published** so change announcements are sent out to any views that are watching it.

Add this class somewhere in your code:

```
@MainActor class DelayedUpdater: ObservableObject {  
    @Published var value = 0  
  
    init() {  
        for i in 1...10 {  
            DispatchQueue.main.asyncAfter(deadline: .now() +
```

Manually publishing ObservableObject changes

```
Double(i)) {
    self.value += 1
}
}
}
}
```

To use that, we just need to add an **@ObservedObject** property in **ContentView**, then show the value in our body, like this:

```
struct ContentView: View {
    @ObservedObject var updater = DelayedUpdater()

    var body: some View {
        Text("Value is: \(updater.value)")
    }
}
```

When you run that code you'll see the value counts upwards until it reaches 10, which is exactly what you'd expect.

Now, if you remove the **@Published** property wrapper you'll see the UI no longer changes. Behind the scenes all the **asyncAfter()** work is still happening, but it doesn't cause the UI to refresh any more because no change notifications are being sent out.

We can fix this by sending the change notifications manually using the **objectWillChange** property I mentioned earlier. This lets us send the change notification whenever we want, rather than relying on **@Published** to do it automatically.

Try changing the **value** property to this:

```
var value = 0 {
    willSet {
        objectWillChange.send()
    }
}
```

Project 16: Hot Prospects

```
    }  
}
```

Now you'll get the old behavior back again – the UI will count to 10 as before. Except this time we have the opportunity to add extra functionality inside that **willSet** observer. Perhaps you want to log something, perhaps you want to call another method, or perhaps you want to clamp the integer inside **value** so it never goes outside of a range – it's all under our control now.

Understanding Swift's Result type

Swift provides a special type called **Result**, that allows us to encapsulate either a successful value or some kind of error type, all in a single piece of data. So, in the same way that an optional might hold a string or might hold nothing at all, for example, **Result** might hold a string or might hold an error. The syntax for using it is a little odd at first, but it does have an important part to play in our projects.

To see **Result** in action, we could start by writing a method that downloads an array of data readings from a server, like this:

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

    var body: some View {
        Text(output)
            .task {
                await fetchReadings()
            }
    }

    func fetchReadings() async {
        do {
            let url = URL(string: "https://hws.dev/readings.json")!
            let (data, _) = try await URLSession.shared.data(from: url)
            let readings = try JSONDecoder().decode([Double].self, from: data)
            output = "Found \(readings.count) readings"
        } catch {
            print("Download error")
        }
    }
}
```

Project 16: Hot Prospects

```
}
```

That code works just fine, but it doesn't give us a lot of flexibility – what if we want to stash the work away somewhere and do something else while it's running? What if we want to read its result at some point in the future, perhaps handling any errors somewhere else entirely? Or what if just want to cancel the work because it's no longer needed?

Well, we can get all that by using **Result**, and it's actually available through an API you've met previously: **Task**. We could rewrite the above code to this:

```
func fetchReadings() async {
    let fetchTask = Task { () -> String in
        let url = URL(string: "https://hws.dev/readings.json")!
        let (data, _) = try await URLSession.shared.data(from: url)
        let readings = try JSONDecoder().decode([Double].self,
from: data)
        return "Found \(readings.count) readings"
    }
}
```

We've used **Task** before to launch pieces of work, but here we've given the **Task** object the name of **fetchTask** – that's what gives us the extra flexibility to pass it around, or cancel it if needed. And notice how our **Task** closure returns a value now? That value gets stored in our **Task** instance so that we can read it in the future when we're ready.

More importantly, that **Task** might have thrown an error if the network fetch failed, or if the data decoding failed, and that's where **Result** comes in: the result of our task might a string saying “Found 10000 readings”, but it might also contain an error. The only way to find out is to look inside – it's very similar to optionals.

To read the result from a **Task**, read its **result** property like this:

```
let result = await fetchTask.result
```

Notice how we haven't used to use `try` to read the **Result** out? That's because **Result** holds it inside itself – an error might have been thrown, but we don't have to worry about it now unless we want to.

If you look at the type of `result`, you'll see it's a `Result<String, Error>` – if it succeeded it will contain a string, but it might also have failed and will contain an error.

You can read the successful value directly from the **Result** if you want, but you'll need to make sure and handle errors appropriately, like this:

```
do {
    output = try result.get()
} catch {
    output = "Error: \(error.localizedDescription)"
}
```

Alternatively, you can **switch** on the **Result**, and write code to check for both the success and failure cases. Each of those cases have their values inside (the string for success, and an error for failure), so Swift lets us read those values out using a specially crafted **case** match:

```
switch result {
    case .success(let str):
        output = str
    case .failure(let error):
        output = "Error: \(error.localizedDescription)"
}
```

Regardless of how you handle it, the advantage of **Result** is that it lets us store the whole success or failure of some work in a single value, pass that around wherever we need, and read the error only when we're ready.

Controlling image interpolation in SwiftUI

What happens if you make a SwiftUI **Image** view that stretches its content to be larger than its original size? By default, we get *image interpolation*, which is where iOS blends the pixels so smoothly you might not even realize they have been stretched at all. There's a performance cost to this of course, but most of the time it's not worth worrying about.

However, there is *one* place where image interpolation causes a problem, and that's when you're dealing with precise pixels. As an example, the files for this project on GitHub contain a little cartoon alien image called `example@3x.png` – it's taken from the Kenney Platform Art Deluxe bundle at <https://kenney.nl/assets/platformer-art-deluxe> and is available under the public domain.

Go ahead and add that graphic to your asset catalog, then change your **ContentView** struct to this:

```
Image("example")
    .resizable()
    .scaledToFit()
    .frame(maxHeight: .infinity)
    .background(.black)
    .ignoresSafeArea()
```

That renders the alien character against a black background to make it easier to see, and because it's resizable SwiftUI will stretch it up to fill all available space.

Take a close look at the edges of the colors: they look jagged, but also blurry. The jagged part comes from the original image because it's only 66x92 pixels in size, but the *blurry* part is where SwiftUI is trying to blend the pixels as they are stretched to make the stretching less obvious.

Often this blending works great, but it struggles here because the source picture is small (and

therefore needs a *lot* of blending to be shown at the size we want), and also because the image has lots of solid colors so the blended pixels stand out quite obviously.

For situations just like this one, SwiftUI gives us the **interpolation()** modifier that lets us control how pixel blending is applied. There are multiple levels to this, but realistically we only care about one: **.none**. This turns off image interpolation entirely, so rather than blending pixels they just get scaled up with sharp edges.

So, modify your image to this:

```
Image("example")
    .interpolation(.none)
    .resizable()
    .scaledToFit()
    .frame(maxHeight: .infinity)
    .background(.black)
    .ignoresSafeArea()
```

Now you'll see the alien character retains its pixellated look, which not only is particularly popular in retro games but is also important for line art that would look wrong when blurred.

Creating context menus

When the user taps a button or a navigation link, it's pretty clear that SwiftUI should trigger the default action for those views. But what if they press and hold on something? On older iPhones users could trigger a 3D Touch by pressing hard on something, but the principle is the same: the user wants more options for whatever they are interacting with.

SwiftUI lets us attach context menus to objects to provide this extra functionality, all done using the `contextMenu()` modifier. You can pass this a selection of buttons and they'll be shown in order, so we could build a simple context menu to control a view's background color like this:

```
struct ContentView: View {
    @State private var backgroundColor = Color.red

    var body: some View {
        VStack {
            Text("Hello, World!")
                .padding()
                .background(backgroundColor)

            Text("Change Color")
                .padding()
                .contextMenu {
                    Button("Red") {
                        backgroundColor = .red
                    }

                    Button("Green") {
                        backgroundColor = .green
                    }

                    Button("Blue") {
```

```
        backgroundColor = .blue  
    }  
}  
}  
}  
}  
}  
}
```

Just like **TabView**, each item in a context menu can have text and an image attached to it using a **Label** view.

Now, there is a catch here: to keep user interfaces looking somewhat uniform across apps, iOS renders each image as a solid color where the opacity is preserved. This makes many pictures useless: if you had three photos of three different dogs, all three would be rendered as a plain black square because all the color got removed.

Instead, you should aim for line art icons such as Apple's SF Symbols, like this:

```
Button {  
    backgroundColor = .red  
}  
label: {  
    Label("Red", systemImage: "checkmark.circle.fill")  
    .foregroundColor(.red)  
}
```

When you run that you'll see the `foregroundColor()` modifier is ignored – iOS really does want our menus to look uniform, so trying to color them randomly just won't work. If you really want that item to appear red, which as you should know means *destructive*, you should use a button role instead:

```
Button(role: .destructive) {  
    backgroundColor = .red  
} label: {  
    Label("Red", systemImage: "checkmark.circle.fill")
```

Project 16: Hot Prospects

}

I have a few tips for you when working with context menus, to help ensure you give your users the best experience:

1. If you're going to use them, use them in lots of places – it can be frustrating to press and hold on something only to find nothing happens.
2. Keep your list of options as short as you can – aim for three or less.
3. Don't repeat options the user can already see elsewhere in your UI.

Remember, context menus are by their nature hidden, so please think twice before hiding important actions in a context menu.

Adding custom row swipe actions to a List

iOS apps have had “swipe to delete” functionality for as long as I can remember, but in more recent years they’ve grown in power so that list rows can have multiple buttons, often on either side of the row. We get this full functionality in SwiftUI using the **swipeActions()** modifier, which lets us register one or more buttons on one or both sides of a list row.

By default buttons will be placed on the right edge of the row, and won’t have any color, so this will show a single gray button when you swipe from right to left:

```
List {  
    Text("Taylor Swift")  
    .swipeActions {  
        Button {  
            print("Hi")  
        } label: {  
            Label("Send message", systemImage: "message")  
        }  
    }  
}
```

You can customize edge where your buttons are placed by providing an **edge** parameter to your **swipeActions()** modifier, and you can customize the color of your buttons either by adding a **tint()** modifier to them with a color of your choosing, or by attaching a button role.

So, this will display one button on either side of our row:

```
List {  
    Text("Taylor Swift")  
    .swipeActions {  
        Button(role: .destructive) {
```

Project 16: Hot Prospects

```
        print("Hi")
    } label: {
        Label("Delete", systemImage: "minus.circle")
    }
}

.swipeActions(edge: .leading) {
    Button {
        print("Hi")
    } label: {
        Label("Pin", systemImage: "pin")
    }
    .tint(.orange)
}
}
```

Like context menus, swipe actions are by their very nature hidden to the user by default, so it's important not to hide important functionality in them. We'll be using them both in this app, which should hopefully give you the chance to compare and contrast them directly!

Scheduling local notifications

iOS has a framework called UserNotifications, that does pretty much exactly what you expect: lets us create notifications to the user that can be shown on the lock screen. We have two types of notifications to work with, and they differ depending on where they were created: local notifications are ones we schedule locally, and remote notifications (commonly called *push* notifications) are sent from a server somewhere.

Remote notifications require a server to work, because you send your message to Apple's push notification service (APNS), which then forwards it to users. But local notifications are nice and easy in comparison, because we can send any message at any time as long as the user allows it.

To try this out, start by adding an extra import near the top of ContentView.swift:

```
import UserNotifications
```

Next we're going to put in some basic structure that we'll fill in with local notifications code. Using local notifications requires asking the user for permission, then actually registering the notification we want to show. We'll place each of those actions into separate buttons inside a **VStack**, so please put this inside your **ContentView** struct now:

```
VStack {
    Button("Request Permission") {
        // first
    }

    Button("Schedule Notification") {
        // second
    }
}
```

OK, that's our set up complete so let's turn our focus to the first of two important pieces of

Project 16: Hot Prospects

work: requesting authorization to show alerts. Notifications can take a variety of forms, but the most common thing to do is ask for permission to show alerts, badges, and sounds – that doesn't mean we need to use all of them at the same time, but by asking permission up front means we can be selective later on.

When we tell iOS what kinds of notifications we want, it will show a prompt to the user so they have the final say on what our app can do. When they make their choice, a closure we provide will get called and tell us whether the request was successful or not.

So, replace the `// first` comment with this:

```
UNUserNotificationCenter.current().requestAuthorization(options:  
    [.alert, .badge, .sound]) { success, error in  
    if success {  
        print("All set!")  
    } else if let error = error {  
        print(error.localizedDescription)  
    }  
}
```

If the user grants permission, then we're all clear to start scheduling notifications. Even though notifications might seem simple, Apple breaks them down into three parts to give it maximum flexibility:

- The content is what should be shown, and can be a title, subtitle, sound, image, and so on.
- The trigger determines when the notification should be shown, and can be a number of seconds from now, a date and time in the future, or a location.
- The request combines the content and trigger, but also adds a unique identifier so you can edit or remove specific alerts later on. If you don't want to edit or remove stuff, use `UUID().uuidString` to get a random identifier.

When you're just learning notifications the easiest trigger type to use is `UNTTimeIntervalNotificationTrigger`, which lets us request a notification to be shown in a

certain number of seconds from now. So, replace the **// second** comment with this:

```
let content = UNMutableNotificationContent()
content.title = "Feed the cat"
content.subtitle = "It looks hungry"
content.sound = UNNotificationSound.default

// show this notification five seconds from now
let trigger = UNTimeIntervalNotificationTrigger(timeInterval:
5, repeats: false)

// choose a random identifier
let request = UNNotificationRequest(identifier:
UUID().uuidString, content: content, trigger: trigger)

// add our notification request
UNUserNotificationCenter.current().add(request)
```

If you run your app now, press the first button to request notification permission, then press the second to add an actual notification.

Now for the important part: once your notification has been added press Cmd+L in the simulator to lock the screen. After a few seconds have passed the device should wake up with a sound, and show our message – nice!

Adding Swift package dependencies in Xcode

Everything we've been coding so far is stuff we've built from scratch, so you can see exactly how it works and apply those skills to your own projects. Sometimes, though, writing something from scratch is risky: perhaps the code is complex, perhaps it's easy to get wrong, perhaps it changes often, or any other myriad of reasons, which is why *dependencies* exist – the ability to fetch third-party code and use it in our projects.

Xcode comes with a dependency manager built in, called Swift Package Manager (SPM). You can tell Xcode the URL of some code that's stored online, and it will download it for you. You can even tell it what version to download, which means if the remote code changes sometime in the future you can be sure it won't break your existing code.

To try this out, I created a simple Swift package that you can import into any project. This adds a small extension to Swift's **Sequence** type (which **Array**, **Set**, **Dictionary**, and even ranges all conform to) that pulls out a number of random items at the same time.

Anyway, the first step is to add the package to our project: go to the File menu and choose Add Packages. For the URL enter <https://github.com/twostraws/SamplePackage>, which is where the code for my example package is stored. Xcode will fetch the package, read its configuration, and show you options asking which version you want to use. The default will be "Version – Up to Next Major", which is the most common one to use and means if the author of the package updates it in the future then as long as they don't introduce breaking changes Xcode will update the package to use the new versions.

The reason this is possible is because most developers have agreed a system of *semantic versioning* (SemVer) for their code. If you look at a version like 1.5.3, then the 1 is considered the major number, the 5 is considered the minor number, and the 3 is considered the patch number. If developers follow SemVer correctly, then they should:

- Change the patch number when fixing a bug as long as it doesn't break any APIs or add

features.

- Change the minor number when they added features that don't break any APIs.
- Change the *major* number when they *do* break APIs.

This is why “Up to Next Major” works so well, because it should mean you get new bug fixes and features over time, but won’t accidentally switch to a version that breaks your code.

Anyway, we’re done with our package, so click Finish to make Xcode add it to the project. You should see it appear in the project navigator, under “Swift Package Dependencies”.

To try it out, open **ContentView.swift** and add this import to the top:

```
import SamplePackage
```

Yes, that external dependency is now a module we can import anywhere we need it.

And now we can try it in our view. For example, we could simulate a simple lottery by making a range of numbers from 1 through 60, picking 7 of them, converting them to strings, then joining them into a single string. To be concise, this will need some code you haven’t seen before so I’m going to break it down.

First, replace your current **ContentView** with this:

```
struct ContentView: View {  
    var body: some View {  
        Text(results)  
    }  
}
```

Yes, that won’t work because it’s missing **results**, but we’re going to fill that in now.

First, creating a range of numbers from 1 through 60 can be done by adding this property to **ContentView**:

Project 16: Hot Prospects

```
let possibleNumbers = Array(1...60)
```

Second, we're going to create a computed property called **results** that picks seven numbers from there and makes them into a single string, so add this property too:

```
var results: String {  
    // more code to come  
}
```

Inside there we're going to select seven random numbers from our range, which can be done using the extension you got from my SamplePackage framework. This provides a **random()** method that accepts an integer and will return up to that number of random elements from your sequence, in a random order. Lottery numbers are usually arranged from smallest to largest, so we're going to sort them.

So, add this line of code in place of **// more code to come**:

```
let selected = possibleNumbers.random(7).sorted()
```

Next, we need to convert that array of integers into strings. This only takes one line of code in Swift, because sequences have a **map()** method that lets us convert an array of one type into an array of another type by applying a function to each element. In our case, we want to initialize a new string from each integer, so we can use **String.init** as the function we want to call.

So, add this line after the previous one:

```
let strings = selected.map(String.init)
```

At this point **strings** is an array of strings containing the seven random numbers from our range, so the last step is to join them all together with commas in between. Add this final line to the property now:

```
return strings.joined(separator: ", ")
```

Adding Swift package dependencies in Xcode

And that completes our code: the text view will show the value inside **results**, which will go ahead and pick random numbers, sort them, stringify them, then join them with commas.

PS: You can read the source code for my simple extension right inside Xcode – just open the Sources > SamplePackage group and look for SamplePackage.swift. You'll see it doesn't do much!

That finishes our final technique required for this project, so please reset your code to its original state.

Building our tab bar

This app is going to display four SwiftUI views inside a tab bar: one to show everyone that you met, one to show people you have contacted, another to show people you *haven't* contacted, and a final one showing your personal information for others to scan.

Those first three views are variations on the same concept, but the last one is quite different. As a result, we can represent all our UI with just three views: one to display people, one to show our data, and one to bring all the others together using **TabView**.

So, our first step will be to create placeholder views for our tabs that we can come back and fill in later. Press Cmd+N to make a new SwiftUI view and call it “ProspectsView”, then create another SwiftUI view called “MeView”. You can leave both of them with the default “Hello, World!” text view; it doesn’t matter for now.

For now, what matters is **ContentView**, because that’s where we’re going to store our **TabView** that contains all the other views in our UI. We’re going to add some more logic here shortly, but for now this is just going to be a **TabView** with three instances of **ProspectsView** and one **MeView**. Each of those views will have a **tabItem()** modifier with an image that I picked out from SF Symbols and some text.

Replace the body of your current **ContentView** with this:

```
TabView {
    ProspectsView()
        .tabItem {
            Label("Everyone", systemImage: "person.3")
        }
    ProspectsView()
        .tabItem {
            Label("Contacted", systemImage: "checkmark.circle")
        }
    ProspectsView()
        .tabItem {
```

```

        Label("Uncontacted", systemImage: "questionmark.diamond")
    }
MeView( )
.tabItem {
    Label("Me", systemImage: "person.crop.square")
}
}

```

If you run the app now you'll see a neat tab bar across the bottom of the screen, allowing us to tap through each of our four views.

Now, obviously creating three instances of **ProspectsView** will be weird in practice because they'll just be identical, but we can fix that by customizing each view. Remember, we want the first one to show every person you've met, the second to show people you have contacted, and the third to show people you *haven't* contacted, and we can represent that with an enum plus a property on **ProspectsView**.

So, add this enum inside **ProspectsView** now:

```

enum FilterType {
    case none, contacted, uncontacted
}

```

Now we can use that to allow each instance of **ProspectsView** to be slightly different by giving it a new property:

```
let filter: FilterType
```

This will immediately break **ContentView** and **ProspectsView_Previews** because they need to provide a value for that property when creating **ProspectsView**, but first let's use it to customize each of the three views just a little by giving them a navigation bar title.

Start by adding this computed property to **ProspectsView**:

Project 16: Hot Prospects

```
var title: String {
    switch filter {
        case .none:
            return "Everyone"
        case .contacted:
            return "Contacted people"
        case .uncontacted:
            return "Uncontacted people"
    }
}
```

Now replace the default “Hello, World!” body text with this:

```
NavigationView {
    Text("Hello, World!")
        .navigationTitle(title)
}
```

That at least makes each of the **ProspectsView** instances look slightly different so we can be sure the tabs are working correctly.

To make our code compile again we need to make sure that every **ProspectsView** initializer is called with a filter. So, in **FilteredView_Previews** change the body to this:

```
ProspectsView(filter: .none)
```

Then change the three **ProspectsView** instances in **ContentView** so they have **filter: .none**, **filter: .contacted**, and **filter: .uncontacted** respectively.

If you run the app now you’ll see it’s looking better. Now for the *real* challenge: those first three views need to work with the same data, so how can we share it all smoothly? For that we need to turn to SwiftUI’s environment...

Sharing data across tabs using {@EnvironmentObject}

SwiftUI's environment lets us share data in a really beautiful way: any view can send objects into the environment, then any child view can read those objects back out from the environment at a later date. Even better, if one view changes the object all other views automatically get updated – it's an incredibly smart way to share data in larger applications.

In our app we have a **TabView** that contains three instances of **ProspectsView**, and we want all three of those to work as different views on the same shared data. This is a great example of where SwiftUI's environment makes sense: we can define a class that stores one prospect, then place an array of those prospects into the environment so all our views can read it if needed.

So, start by making a new Swift file called `Prospect.swift`, replacing its Foundation import with SwiftUI, then giving it this code:

```
class Prospect: Identifiable, Codable {
    var id = UUID()
    var name = "Anonymous"
    var emailAddress = ""
    var isContacted = false
}
```

Yes, that's a class rather than a struct. This is intentional, because it allows us to change instances of the class directly and have it be updated in all other views at the same time. Remember, SwiftUI takes care of propagating that change to our views automatically, so there's no risk of views getting stale.

When it comes to sharing that across multiple views, one of the best things about SwiftUI's environment is that it uses the same **ObservableObject** protocol we've been using with the **@StateObject** property wrapper. This means we can mark properties that should be announced using the **@Published** property wrapper – SwiftUI takes care of most of the work

Project 16: Hot Prospects

for us.

So, add this class in Prospect.swift:

```
@MainActor class Prospects: ObservableObject {
    @Published var people: [Prospect]

    init() {
        self.people = []
    }
}
```

We'll come back to that later on, not least to make the initializer do more than just create an empty array, but it's good enough for now.

Now, we want all our **ProspectsView** instances to share a single instance of the **Prospects** class, so they are all pointing to the same data. If we were writing UIKit code here I'd go into long explanation about how difficult this is to get right and how careful we need to be to ensure all changes get propagated cleanly, but with SwiftUI it requires just three steps.

First, we need to add a property to **ContentView** that creates and stores a single instance of the **Prospects** class:

```
@StateObject var prospects = Prospects()
```

Second, we need to post that property into the SwiftUI environment, so that all child views can access it. Because tabs are considered children of the tab view they are inside, if we add it to the environment for the **TabView** then all our **ProspectsView** instances will get that object.

So, add this modifier to the **TabView** in **ContentView**:

```
.environmentObject(prospects)
```

Important: Make sure you add the same modifier to the preview struct for **ProspectsView**, so

your previews continue working.

And now we want all instances of **ProspectsView** to read that object back out of the environment when they are created. This uses a new `@EnvironmentObject` property wrapper that does all the work of finding the object, attaching it to a property, and keeping it up to date over time. So, the final step is just adding this property to **ProspectsView**:

```
@EnvironmentObject var prospects: Prospects
```

That really is all it takes – I don’t think there’s a way SwiftUI could make this any easier.

Important: When you use `@EnvironmentObject` you are explicitly telling SwiftUI that your object will exist in the environment by the time the view is created. If it isn’t present, your app will crash immediately – be careful, and treat it like an implicitly unwrapped optional.

Soon we’re going to be adding code to add prospects by scanning QR codes, but for now we’re going to add a navigation bar item that just adds test data and shows it on-screen.

Change the `body` property of **ProspectsView** to this:

```
NavigationView {
    Text("People: \(prospects.people.count)")
        .navigationTitle(title)
        .toolbar {
            Button {
                let prospect = Prospect()
                prospect.name = "Paul Hudson"
                prospect.emailAddress = "paul@hackingwithswift.com"
                prospects.people.append(prospect)
            } label: {
                Label("Scan", systemImage: "qrcode.viewfinder")
            }
        }
}
```

Project 16: Hot Prospects

Now you'll see a "Scan" button on the first three views of our tab view, and tapping it adds a person to all three simultaneously – you'll see the count increment no matter which button you tap.

Dynamically filtering a SwiftUI List

SwiftUI's **List** view likes to work with arrays of objects that conform to the **Identifiable** protocol, or at least can provide some sort of **id** parameter that is guaranteed to be unique. However, there's no reason these need to be *stored* properties of a view, and in fact if we send in a *computed* property then we're able to adjust our filtering on demand.

In our app, we have three instances of **ProspectsView** that vary only according to the **FilterType** property that gets passed in from our tab view. We're already using that to set the title of each view, but we can also use it to set the contents for a **List**.

The easiest way to do this is using Swift's **filter()** method. This runs every element in a sequence through a test you provide as a closure, and any elements that return true from the test are sent back as part of a new array. Our **ProspectsView** already has a **prospects** property being passed in with an array of people inside it, so we can either return all people, all contacted people, or all uncontacted people.

Add this property to **ProspectsView** below the previous two:

```
var filteredProspects: [Prospect] {
    switch filter {
        case .none:
            return prospects.people
        case .contacted:
            return prospects.people.filter { $0.isContacted }
        case .uncontacted:
            return prospects.people.filter { !$0.isContacted }
    }
}
```

When **filter()** runs, it passes every element in the **people** array through our test. So, **\$0.isContacted** means “does the current element have its **isContacted** property set to true?” All items in the array that pass that test – that have **isContacted** set to true – will be added to a

Project 16: Hot Prospects

new array and sent back from **filteredResults**. And when we use **!\$0.isContacted** we get the opposite: only prospects that *haven't* been contacted get included.

With that computed property in place, we can now create a **List** to loop over that array. This will show both the title and email address for each prospect using a **VStack**, and we'll also use a **ForEach** so we can add deleting later on.

Replace the existing text view in **ProspectsView** with this:

```
List {  
    ForEach(filteredProspects) { prospect in  
        VStack(alignment: .leading) {  
            Text(prospect.name)  
                .font(.headline)  
            Text(prospect.emailAddress)  
                .foregroundColor(.secondary)  
        }  
    }  
}
```

If you run the app again you'll see things are starting to look much better.

Before we move on, I want you to think about this: now that we're using a computed property, how does SwiftUI know to refresh the view when the property changed? The answer is actually quite simple: *it doesn't*.

When we added an **@EnvironmentObject** property to **ProspectsView**, we also asked SwiftUI to reinvoke the **body** property whenever that property changes. So, whenever we insert a new person into the **people** array its **@Published** property wrapper will announce the update to all views that are watching it, and SwiftUI will reinvoke the **body** property of **ProspectsView**. That in turn will calculate our computed property again, so the **List** will change.

I love the way SwiftUI transparently takes on so much work for us here, which means we can focus on how we filter and present our data rather than how to connect up all the pipes to make

Dynamically filtering a SwiftUI List

sure things are kept up to date.

Generating and scaling up a QR code

Core Image lets us generate a QR code from any input string, and do so extremely quickly. However, there's a problem: the image it generates is very small because it's only as big as the pixels required to show its data. It's trivial to make the QR code larger, but to make it look *good* we also need to adjust SwiftUI's image interpolation. So, in this step we're going to ask the user to enter their name and email address in a form, use those two pieces of information to generate a QR code identifying them, and scale up the code without making it fuzzy.

We already have a simple **MeView** struct that we made as a placeholder earlier, so our first job will be to add a couple of text fields and their string bindings.

First, add these two new pieces of state to hold a name and email address:

```
@State private var name = "Anonymous"  
@State private var emailAddress = "you@yoursite.com"
```

When it comes to the body of the view we're going to use two text fields with large fonts, but this time we're going to attach a small but useful modifier to the text fields called **textContentType()** – it tells iOS what kind of information we're asking the user for. This should allow iOS to provide autocomplete data on behalf of the user, which makes the app nicer to use.

Replace your current body with this:

```
NavigationView {  
    Form {  
        TextField("Name", text: $name)  
            .textContentType(.name)  
            .font(.title)  
  
        TextField("Email address", text: $emailAddress)
```

```

    .textContentType(.emailAddress)
    .font(.title)
}
.navigationTitle("Your code")
}

```

We're going to use the name and email address fields to generate a QR code, which is a square collection of black and white pixels that can be scanned by phones and other devices. Core Image has a filter for this built in, and as you've learned how to use Core Image filters previously you'll find this is very similar.

First, we need to bring in all the Core Image filters using a new import:

```
import CoreImage.CIFilterBuiltins
```

Second, we need two properties to store an active Core Image context and an instance of Core Image's QR code generator filter. So, add these two to **MeView**:

```

let context = CIContext()
let filter = CIFilter.qrCodeGenerator()

```

Now for the interesting part: making the QR code itself. If you remember, working with Core Image filters requires us to provide some input data, then convert the output **CIImage** into a **CGImage**, then that **CGImage** into a **UIImage**. We'll be following the same steps here, except:

1. Our input for the filter will be a string, but the input for the filter is **Data**, so we need to convert that.
2. If conversion fails for any reason we'll send back the "xmark.circle" image from SF Symbols.
3. If *that* can't be read – which is theoretically possible because SF Symbols is stringly typed – then we'll send back an empty **UIImage**.

Project 16: Hot Prospects

So, add this method to the **MeView** struct now:

```
func generateQRCode(from string: String) -> UIImage {
    filter.message = Data(string.utf8)

    if let outputImage = filter.outputImage {
        if let cgimg = context.createCGImage(outputImage, from:
outputImage.extent) {
            return UIImage(cgImage: cgimg)
        }
    }

    return UIImage(systemName: "xmark.circle") ?? UIImage()
}
```

Isolating all that functionality in a method works really well in SwiftUI, because it means the code we put in the **body** property stays as simple as possible. In fact, we can just use **Image(uiImage:)** directly with a call to **generateQRCode(from:)**, then scale it up to a sensible size onscreen – SwiftUI will make sure the method gets called every time **name** or **emailAddress** changes.

In terms of the string to pass *in* to **generateQRCode(from:)**, we'll be using the name and email address entered by the user, separated by a line break. This is a nice and simple format, and it's easy to reverse when it comes to scanning these codes later on.

Add this new **Image** view directly before the second text field:

```
Image(uiImage: generateQRCode(from: "\u{1f464} (\name)\n\(
emailAddress)") )
    .resizable()
    .scaledToFit()
    .frame(width: 200, height: 200)
```

Generating and scaling up a QR code

If you run the code you'll see it works pretty well – you'll see a default QR code, but you can also type into either of the two text fields to make the QR code change dynamically.

However, take a close look at the QR code – do you notice how it's fuzzy? This is because Core Image is generating a tiny image, and SwiftUI is trying to smooth out the pixels as we scale it up.

Line art like QR codes and bar codes is a great candidate for disabling image interpolation. Try adding this modifier to the image to see what I mean:

```
.interpolation(.none)
```

Now the QR code will be rendered nice and sharp, because SwiftUI will just repeat pixels rather than try to blend them neatly. I would imagine cameras don't care which gets used, but it looks better to users!

Scanning QR codes with SwiftUI

Scanning a QR code – or indeed any kind of visible code such as barcodes – can be done by Apple’s AVFoundation library. This doesn’t integrate into SwiftUI terribly smoothly, so to skip over a whole lot of pain I’ve packaged up a QR code reader into a Swift package that we can add and use directly inside Xcode.

My package is called **CodeScanner**, and its available on GitHub under the MIT license at <https://github.com/twostraws/CodeScanner> – you’re welcome to inspect and/or edit the source code if you want. Here, though, we’re just going to add it to Xcode by following these steps:

1. Go to File > Swift Packages > Add Package Dependency.
2. Enter **https://github.com/twostraws/CodeScanner** as the package repository URL.
3. For the version rules, leave “Up to Next Major” selected, which means you’ll get any bug fixes and additional features but *not* any breaking changes.
4. Press Finish to import the finished package into your project.

The **CodeScanner** package gives us one **CodeScanner** SwiftUI view to use, which can be presented in a sheet and handle code scanning in a clean, isolated way. I know I keep repeating myself, but I hope you can see the continuing theme: the best way to write SwiftUI is to isolate functionality in discrete methods and wrappers, so that all you expose to your SwiftUI layouts is clean, clear, and unambiguous.

We already have a “Scan” button in **ProspectsView**, and we’re going to use that trigger QR scanning. So, start by adding this new **@State** property to **ProspectsView**:

```
@State private var isShowingScanner = false
```

Earlier we added some test functionality to the “Scan” button so we could insert some sample data, but we don’t need that any more because we’re about to scan real QR codes. So, replace the action code for the toolbar button with this:

```
isShowingScanner = true
```

When it comes to handling the result of the QR scanning, I've made the **CodeScanner** package do literally all the work of figuring out what the code is and how to send it back, so all we need to do here is catch the result and process it somehow.

When the **CodeScannerView** finds a code, it will call a completion closure with a **Result** instance either containing details about the code that was found or an error saying what the problem was – perhaps the camera wasn't available, or the camera wasn't able to scan codes, for example. Regardless of what code or error comes back, we're just going to dismiss the view; we'll add more code shortly to do more work.

Start by adding this new import near the top of **ProspectsView.swift**:

```
import CodeScanner
```

Now add this method to **ProspectsView**:

```
func handleScan(result: Result<ScanResult, ScanError>) {
    isShowingScanner = false
    // more code to come
}
```

Before we show the scanner and try to handle its result, we need to ask the user for permission to use the camera:

1. Go to your target's configuration options under its Info tab.
2. Right-click on an existing key and select Add Row.
3. Select “Privacy - Camera Usage Description” for the key.
4. For the value enter “We need to scan QR codes.”

And now we're ready to scan some QR codes! We already have the **isShowingScanner** state that determines whether to show a code scanner or not, so we can now attach a **sheet()**

Project 16: Hot Prospects

modifier to present our scanner UI.

Creating a **CodeScanner** view takes three parameters:

1. An array of the types of codes we want to scan. We're only scanning QR codes in this app so `[.qr]` is fine, but iOS supports lots of other types too.
2. A string to use as simulated data. Xcode's simulator doesn't support using the camera to scan codes, so **CodeScannerView** automatically presents a replacement UI so we can still test that things work. This replacement UI will automatically send back whatever we pass in as simulated data.
3. A completion function to use. This could be a closure, but we just wrote the **handleScan()** method so we'll use that.

So, add this below the existing **toolbar()** modifier in **ProspectsView**:

```
.sheet(isPresented: $isShowingScanner) {
    CodeScannerView(codeTypes: [.qr], simulatedData: "Paul
Hudson\npaul@hackingwithswift.com", completion: handleScan)
}
```

That's enough to get most of this screen working, but there is one last step: replacing the `// more code to come` comment in **handleScan()** with some actual functionality to process the data we found.

If you recall, the QR codes we're generating are a name, then a line break, then an email address, so if our scanning result comes back successfully then we can pull apart the code string into those components and use them to create a new **Prospect**. If code scanning failed, we'll just print an error – you're welcome to show some more interesting UI if you want!

Replace the `// more code to come` comment with this:

```
switch result {
case .success(let result):
    let details = result.string.components(separatedBy: "\n")
```

```
guard details.count == 2 else { return }

let person = Prospect()
person.name = details[0]
person.emailAddress = details[1]

prospects.people.append(person)
case .failure(let error):
    print("Scanning failed: \(error.localizedDescription)")
}
```

Go ahead and run the code now. If you’re using the simulator you’ll see a test UI appear, and tapping anywhere will dismiss the view and send back our simulated data. If you’re using a real device you’ll see a permission message asking the user to allow camera use, and you grant that you’ll see a scanner view. To test out scanning on a real device, simultaneously launch the app in the simulator and switch to the Me tab – your phone should be able to scan the simulator screen on your computer.

Adding options with swipe actions

We need a way to move people between the Contacted and Uncontacted tabs, and the easiest thing to do is add a swipe action to the **VStack** in **ProspectsView**. This will allow users to swipe on any person in the list, then tap a single option to move them between the tabs.

Now, remember that this view is shared in three places, so we need to make sure the swipe actions look correct no matter where it's used. We *could* try and use a bunch of ternary conditional operators, but later on we'll add a second button so the ternary operator approach won't really help much. Instead, we'll just wrap the button inside a simple condition – add this to the **VStack** now:

```
.swipeActions {
    if prospect.isContacted {
        Button {
            prospect.isContacted.toggle()
        } label: {
            Label("Mark Uncontacted", systemImage:
"person.crop.circle.badge.xmark")
        }
        .tint(.blue)
    } else {
        Button {
            prospect.isContacted.toggle()
        } label: {
            Label("Mark Contacted", systemImage:
"person.crop.circle.fill.badge.checkmark")
        }
        .tint(.green)
    }
}
```

While the *text* for that is OK and the context menu displays correctly, the *action* doesn't do

anything. Well, that's not strictly true: it does toggle the Boolean, but it doesn't actually update the UI.

This problem occurs because the **people** array in **Prospects** is marked with **@Published**, which means if we add or remove items from that array a change notification will be sent out. However, if we quietly change an item *inside* the array then SwiftUI won't detect that change, and no views will be refreshed.

To fix this, we need to tell SwiftUI by hand that something important has changed. So, rather than flipping a Boolean in **ProspectsView**, we are instead going to call a method on the **Prospects** class to flip that same Boolean while also sending a change notification out.

Start by adding this method to the **Prospects** class:

```
func toggle(_ prospect: Prospect) {
    objectWillChange.send()
    prospect.isContacted.toggle()
}
```

Important: You should call **objectWillChange.send()** *before* changing your property, to ensure SwiftUI gets its animations correct.

Now you can replace the **prospect.isContacted.toggle()** action with this:

```
prospects.toggle(prospect)
```

If you run the app now you'll see it works much better – scan a user, then bring up the context menu and tap its action to see the user move between the Contacted and Uncontacted tabs.

We could leave it there, but there's one more change I want to make. As you saw, changing **isContacted** directly causes problems, because although the Boolean has changed internally our UI has become stale. If we leave our code as-is, it's possible we (or other developers) might forget about this problem and try to flip the Boolean directly from elsewhere, which will just cause more bugs.

Project 16: Hot Prospects

Swift can help us mitigate this problem by stopping us from modifying the Boolean outside of `Prospects.swift`. There's a specific access control option called **fileprivate**, which means "this property can only be used by code inside the current file." Of course, we still want to *read* that property, and so we can deploy another useful Swift feature: **fileprivate(set)**, which means "this property can be read from anywhere, but only *written* from the current file" – the exact combination we need to make sure the Boolean is safe to use.

So, modify the **isContacted** Boolean in **Prospect** to this:

```
fileprivate(set) var isContacted = false
```

It hasn't affected our project here, but it does help keep us safe in the future. If you were wondering why we put the **Prospect** and **Prospects** classes in the same file, now you know!

Saving and loading data with UserDefaults

This app mostly works, but it has one fatal flaw: any data we add gets wiped out when the app is relaunched, which doesn't make it much use for remembering who we met. We can fix this by making the **Prospects** initializer able to load data from **UserDefault**s, then write it back when the data changes.

This time our data is stored using a slightly easier format: although the **Prospects** class uses the **@Published** property wrapper, the **people** array inside it is simple enough that it already conforms to **Codable** just by adding the protocol conformance. So, we can get most of the way to our goal by making three small changes:

1. Updating the **Prospects** initializer so that it loads its data from **UserDefault**s where possible.
2. Adding a **save()** method to the same class, writing the current data *to* **UserDefault**s.
3. Calling **save()** when adding a prospect or toggling its **isContacted** property.

We've looked at the code to do all that previously, so let's get to it. We already have a simple initializer for **Prospects**, so we can update it to use **UserDefault**s like this:

```
init() {
    if let data = UserDefaults.standard.data(forKey: "SavedData") {
        if let decoded = try? JSONDecoder().decode([Prospect].self,
from: data) {
            people = decoded
            return
        }
    }
}

people = []
```

Project 16: Hot Prospects

```
}
```

As for the **save()** method, this will do the same thing in reverse – add this:

```
func save() {
    if let encoded = try? JSONEncoder().encode(people) {
        UserDefaults.standard.set(encoded, forKey: "SavedData")
    }
}
```

Our data is changed in two places, so we need to make both of those call **save()** to make sure the data is always written out.

The first is in the **toggle()** method of **Prospects**, so modify it to this:

```
func toggle(_ prospect: Prospect) {
    objectWillChange.send()
    prospect.isContacted.toggle()
    save()
}
```

The second is in the **handleScan(result:)** method of **ProspectsView**, where we add new prospects to the array. Find this line:

```
prospects.people.append(person)
```

And add this directly below:

```
prospects.save()
```

If you run the app now you'll see that any contacts you add will remain there even after you relaunch the app, so we could easily stop here. However, this time I want to go a stage further and fix two other problems:

1. We've had to hard-code the key name "SavedData" in two places, which again might cause problems in the future if the name changes or needs to be used in more places.
2. Having to call `save()` inside **ProspectsView** isn't good design, partly because our view really shouldn't know about the internal workings of its model, but also because if we have other views working with the data then we might forget to call `save()` there.

To fix the first problem we should create a property on **Prospects** to contain our save key, so we use that property rather than a string for **UserDefault**s.

Add this to the **Prospects** class:

```
let saveKey = "SavedData"
```

We can then use that rather than a hard-coded string, first by modifying the initializer like this:

```
if let data = UserDefaults.standard.data(forKey: saveKey) {
```

And by modifying the `save()` method to this:

```
UserDefaults.standard.set(encoded, forKey: saveKey)
```

This approach is much safer in the long term – it's far too easy to write "SaveKey" or "savedKey" by accident, and in doing so introduce all sorts of bugs.

As for the problem of calling `save()`, this is actually a deeper problem: when we write code like `prospects.people.append(person)` we're breaking a software engineering principle known as *encapsulation*. This is the idea that we should limit how much external objects can read and write values inside a class or a struct, and instead provide methods for reading (getters) and writing (setters) that data.

In practical terms, this means rather than writing `prospects.people.append(person)` we'd instead create an `add()` method on the **Prospects** class, so we could write code like this:

`prospects.add(person)`. The result would be the same – our code adds a person to the `people` array – but now the implementation is hidden away. This means that we could switch the array

Project 16: Hot Prospects

out to something else and **ProspectsView** wouldn't break, but it also means we can add extra functionality to the **add()** method.

So, to solve the second problem we're going to create an **add()** method in **Prospects** so that we can internally trigger **save()**. Add this now:

```
func add(_ prospect: Prospect) {  
    people.append(prospect)  
    save()  
}
```

Even better, we can use access control to stop external writes to the **people** array, meaning that our views must use the **add()** method to add prospects. This is done by changing the definition of the **people** property to this:

```
@Published private(set) var people: [Prospect]
```

Now that only code inside **Prospects** calls the **save()** method, we can mark *that* as being private too:

```
private func save() {
```

This helps lock down our code so that we can't make mistakes by accident – the compiler simply won't allow it. In fact, if you try building the code now you'll see exactly what I mean: **ProspectsView** tries to append to the **people** array and call **save()**, which is no longer allowed.

To fix that error and get our code compiling cleanly again, replace those two lines with this:

```
prospects.add(person)
```

Switching away from strings then using encapsulation and access control are simple ways of making our code safer, and are some great steps towards building better software.

Adding a context menu to an image

We've already written code that dynamically generates a QR code based on the user's name and email address, but with a little extra code we can also let the user *save* that QR code to their images.

Start by opening MeView.swift, and adding the **contextMenu()** modifier to the QR code image, like this:

```
Image(uiImage: generateQRCode(from: "\(name)\n\(emailAddress)"))
    .interpolation(.none)
    .resizable()
    .scaledToFit()
    .frame(width: 200, height: 200)
    .contextMenu {
        Button {
            // save my code
        } label: {
            Label("Save to Photos", systemImage:
                "square.and.arrow.down")
        }
    }
}
```

In terms of saving the image, we can use the same **ImageSaver** class we used back in project 13 (Instafilter), because that takes care of all the complex work for us. If you have **ImageSaver.swift** around from the previous project you can just drag it into your new project now, but if not here's the code again:

```
import UIKit
```

Project 16: Hot Prospects

```
class ImageSaver: NSObject {
    var successHandler: ((()) -> Void)?
    var errorHandler: ((Error) -> Void)?

    func writeToPhotoAlbum(image: UIImage) {
        UIImageWriteToSavedPhotosAlbum(image, self,
#selector(saveCompleted), nil)
    }

    @objc func saveCompleted(_ image: UIImage,
didFinishSavingWithError error: Error?, contextInfo:
UnsafeRawPointer) {
        if let error = error {
            errorHandler?(error)
        } else {
            successHandler?()
        }
    }
}
```

When it comes to *using* that we can just repeat the same code to generate our **UIImage**, then save that – replace the // **save my code** comment with this:

```
let image = generateQRCode(from: "\u{1f4bb}(\name)\n\u{1f4bd}(\emailAddress)")
let imageSaver = ImageSaver()
imageSaver.writeToPhotoAlbum(image: image)
```

And we're done!

We *could* save a little work by caching the generated QR code, however a more important side effect of that is that we wouldn't have to pass in the name and email address each time – duplicating that data means if we change one copy in the future we need to change the other too.

To add this change, first add a new `@State` property that will store the code we generate:

```
@State private var qrCode = UIImage()
```

Now modify `generateQRCode()` so that it quietly stores the new code in our cache before sending it back:

```
if let cgimg = context.createCGImage(outputImage, from: outputImage.extent) {
    qrCode = UIImage(cgImage: cgimg)
    return qrCode
}
```

And now our context menu button can use the cached code:

```
Button {
    let imageSaver = ImageSaver()
    imageSaver.writeToPhotoAlbum(image: qrCode)
} label: {
    Label("Save to Photos", systemImage: "square.and.arrow.down")
}
```

That code will compile cleanly, but I want you to run it and see what happens.

If everything has gone to plan, Xcode should show a purple warning over your code, saying that we modified our view's state during a view update, which causes undefined behavior. “Undefined behavior” is a fancy way of saying “this could behave in any number of weird ways, so don't do it.”

You see, we're telling Swift it can load our image by calling the `generateQRCode()` method, so when SwiftUI calls the `body` property it will run `generateQRCode()` as requested. However, while it's running that method, we then change our new `@State` property, even though SwiftUI hasn't actually finished updating the `body` property yet.

Project 16: Hot Prospects

This is A Very Bad Idea, which is why Xcode is flagging up a large warning. Think about it: if drawing the QR code changes our `@State` cache property, that will cause `body` to load again, which will cause the QR code to be drawn again, which will change our cache property again, and so on – it's really messy.

The smart thing to do here is tell our image to render directly from the cached `qrImage` property, then call `generateQRCode()` when the view appears and whenever either name or email address changes.

First, add this new method to **MeView**, so we can update our code from several places without having to repeat the exact string:

```
func updateCode() {  
    qrCode = generateQRCode(from: "\(name)\n\(emailAddress)")  
}
```

Second, revert the `qrCode = UIImage(cgImage: cgimg)` line in `generateQRCode()`, because that's no longer needed – you can just return the `UIImage` directly, like before.

Third, change the QR code image to this:

`Image(uiImage: qrCode)`

Finally, add these new modifiers after **navigationTitle()**:

```
.onAppear(perform: updateCode)  
.onChange(of: name) { _ in updateCode() }  
.onChange(of: emailAddress) { _ in updateCode() }
```

That will ensure the QR code is updated as soon as the view is shown, or whenever `name` or `emailAddress` get changed – perfect for our needs, and much safer than trying to change some state while SwiftUI is updating our view.

Before you try the context menu yourself, make sure you add the same project option we had.

Adding a context menu to an image

for the Instafilter project – you need to add a permission request string to your project’s configuration options.

In case you’ve forgotten how to do that, here are the steps you need:

- Open your target settings
- Select the Info tab
- Right-click on an existing option
- Choose Add Row
- Select “Privacy - Photo Library Additions Usage Description” for the key name.
- Enter “We want to save your QR code.” as the value.

And now this step is done – you should be able to run the app, switch to the Me tab, then long press the QR code to bring up your new context menu.

Posting notifications to the lock screen

For the final part of our app, we’re going to add another button to our list swipe actions, letting users opt to be reminded to contact a particular person. This will use iOS’s UserNotifications framework to create a local notification, and we’ll conditionally include it in the swipe actions as part of our existing **if** check – the button will only be shown if the user hasn’t been contacted already.

Much more interesting is how we schedule the local notifications. Remember, the first time we try this we need to use **requestAuthorization()** to explicitly ask for permission to show a notification on the lock screen, but we also need to be careful subsequent times because the user can retroactively change their mind and disable notifications.

One option is to call **requestAuthorization()** *every time we want to post a notification*, and honestly that works great: the first time it will show an alert, and all other times it will immediately return success or failure based on the previous response.

However, in the interests of completion I want to show you a more powerful alternative: we can request the current authorization settings, and use that to determine whether we should schedule a notification or request permission. The reason it’s helpful to use *this* approach rather than just requesting permission repeatedly, is that the settings object handed back to us includes properties such as **alertSetting** to check whether we can show an alert or not – the user might have restricted this so all we can do is display a numbered badge on our icon.

So, we’re going to call **getNotificationSettings()** to read whether notifications are currently allowed. If they are, we’ll show a notification. If they *aren’t*, we’ll request permissions, and if *that* comes back successfully then we’ll also show a notification. Rather than repeat the code to schedule a notification, we’ll put it inside a closure that can be called in either scenario.

Start by adding this import near the top of ProspectsView.swift:

```
import UserNotifications
```

Now add this method to the **ProspectsView** struct:

```
func addNotification(for prospect: Prospect) {
    let center = UNUserNotificationCenter.current()

    let addRequest = {
        let content = UNMutableNotificationContent()
        content.title = "Contact \(prospect.name)"
        content.subtitle = prospect.emailAddress
        content.sound = UNNotificationSound.default

        var dateComponents = DateComponents()
        dateComponents.hour = 9
        let trigger = UNCalendarNotificationTrigger(dateMatching:
dateComponents, repeats: false)

        let request = UNNotificationRequest(identifier:
UUID().uuidString, content: content, trigger: trigger)
        center.add(request)
    }

    // more code to come
}
```

That puts all the code to create a notification for the current prospect into a closure, which we can call whenever we need. Notice that I've used **UNCalendarNotificationTrigger** for the trigger, which lets us specify a custom **DateComponents** instance. I set it to have an hour component of 9, which means it will trigger the next time 9am comes about.

Tip: For testing purposes, I recommend you comment out that trigger code and replace it with the following, which shows the alert five seconds from now:

Project 16: Hot Prospects

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval:  
5, repeats: false)
```

For the second part of that method we're going to use both `getNotificationSettings()` and `requestAuthorization()` together, to make sure we only schedule notifications when allowed. This will use the `addRequest` closure we defined above, because the same code can be used if we have permission already or if we ask and have been granted permission.

Replace the `// more code to come` comment with this:

```
center.getNotificationSettings { settings in  
    if settings.authorizationStatus == .authorized {  
        addRequest()  
    } else {  
        center.requestAuthorization(options:  
[.alert, .badge, .sound]) { success, error in  
            if success {  
                addRequest()  
            } else {  
                print("D'oh")  
            }  
        }  
    }  
}
```

That's all the code we need to schedule a notification for a particular prospect, so all that remains is to add an extra button to our swipe actions – add this below the “Mark Contacted” button:

```
Button {  
    addNotification(for: prospect)  
} label: {  
    Label("Remind Me", systemImage: "bell")
```

Posting notifications to the lock screen

```
}
```

```
.tint(.orange)
```

That completes the current step, and completes our project too – try running it now and you should find that you can add new prospects, then press and hold to either mark them as contacted, or to schedule a contact reminder.

Good job!

Hot Prospects: Wrap up

This was our largest project yet, but the end result is another really useful app that could easily form the starting point for a real conference. Along the way we also learned about custom environment objects, **TabView**, **Result**, **objectWillChange**, image interpolation, context menus, local notifications, Swift package dependencies, **filter()** and **map()**, and so much more – it's been packed!

We've explored several of Apple's other frameworks now – Core ML, MapKit, Core Image, and now UserNotifications – so I hope you're getting a sense of just how much we can build just by relying on all the work Apple has already done for us.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Add an icon to the “Everyone” screen showing whether a prospect was contacted or not.
2. Use JSON and the documents directory for saving and loading our user data.
3. Use a confirmation dialog to customize the way users are sorted in each tab – by name or by most recent.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Hot Prospects. If you don't already subscribe, you can start a free trial today.

Project 17

Flashzilla

Use gestures and haptics to build a learning app.

Flashzilla: Introduction

In this project we're going to build an app that helps users learn things using flashcards – cards with one thing written on such as “to buy”, and another thing written on the other side, such as “comprar”. Of course, this is a digital app so we don't need to worry about “the other side”, and can instead just make the detail for the flash card appear when it's tapped.

The name for this project is actually the name of my first ever app for iOS – an app I shipped so long ago it was written for iPhoneOS because the iPad hadn't been announced yet. Apple actually rejected the app during app review because it had “Flash” in its product name, and at the time Apple were really keen to have Flash nowhere near their App Store! How times have changed...

Anyway, we have lots of interesting things to learn in this project, including gestures, haptics, timers, and more, so please create a new iOS project using the App template, naming it Flashzilla. As always we have some techniques to cover before we get into building the real thing, so let's get started...

How to use gestures in SwiftUI

SwiftUI gives us lots of gestures for working with views, and does a great job of taking away most of the hard work so we can focus on the parts that matter. Easily the most common is our friend `onTapGesture()`, but there are several others, and there are also interesting ways of combining gestures together that are worth trying out.

I'm going to skip past the simple `onTapGesture()` because we've covered it before so many times, but before we try bigger things I do want to add that you can pass a `count` parameter to these to make them handle double taps, triple taps, and more, like this:

```
Text("Hello, World!")
    .onTapGesture(count: 2) {
        print("Double tapped!")
    }
```

OK, let's look at something more interesting than simple taps. For handling long presses you can use `onLongPressGesture()`, like this:

```
Text("Hello, World!")
    .onLongPressGesture {
        print("Long pressed!")
    }
```

Like tap gestures, long press gestures are also customizable. For example, you can specify a minimum duration for the press, so your action closure only triggers after a specific number of seconds have passed. For example, this will trigger only after two seconds:

```
Text("Hello, World!")
    .onLongPressGesture(minimumDuration: 2) {
        print("Long pressed!")
    }
```

Project 17: Flashzilla

You can even add a second closure that triggers whenever the state of the gesture has changed. This will be given a single Boolean parameter as input, and it will work like this:

1. As soon as you press down the change closure will be called with its parameter set to true.
2. If you release before the gesture has been recognized (so, if you release after 1 second when using a 2-second recognizer), the change closure will be called with its parameter set to false.
3. If you hold down for the full length of the recognizer, then the change closure will be called with its parameter set to false (because the gesture is no longer in flight), and your completion closure will be called too.

Use code like this to try it out for yourself:

```
Text("Hello, World!")
    .onLongPressGesture(minimumDuration: 1) {
        print("Long pressed!")
    } onPressingChanged: { inProgress in
        print("In progress: \(inProgress)!")
    }
```

For more advanced gestures you should use the `gesture()` modifier with one of the gesture structs: **DragGesture**, **LongPressGesture**, **MagnificationGesture**, **RotationGesture**, and **TapGesture**. These all have special modifiers, usually `onEnded()` and often `onChanged()` too, and you can use them to take action when the gestures are in-flight (for `onChanged()`) or completed (for `onEnded()`).

As an example, we could attach a magnification gesture to a view so that pinching in and out scales the view up and down. This can be done by creating two `@State` properties to store the scale amount, using that inside a `scaleEffect()` modifier, then setting those values in the gesture, like this:

```
struct ContentView: View {
    @State private var currentAmount = 0.0
```

```

@State private var finalAmount = 1.0

var body: some View {
    Text("Hello, World!")
        .scaleEffect(finalAmount + currentAmount)
        .gesture(
            MagnificationGesture()
                .onChanged { amount in
                    currentAmount = amount - 1
                }
                .onEnded { amount in
                    finalAmount += currentAmount
                    currentAmount = 0
                }
        )
}
}

```

Exactly the same approach can be taken for rotating views using **RotationGesture**, except now we're using the **rotationEffect()** modifier:

```

struct ContentView: View {
    @State private var currentAmount = Angle.zero
    @State private var finalAmount = Angle.zero

    var body: some View {
        Text("Hello, World!")
            .rotationEffect(currentAmount + finalAmount)
            .gesture(
                RotationGesture()
                    .onChanged { angle in
                        currentAmount = angle
                    }
            )
    }
}

```

Project 17: Flashzilla

```
        .onEnded { angle in
            finalAmount += .currentAmount
            currentAmount = .zero
        }
    )
}
}
```

Where things start to get more interesting is when gestures *clash* – when you have two or more gestures that might be recognized at the same time, such as if you have one gesture attached to a view and the same gesture attached to its parent.

For example, this attaches an on **onTapGesture()** to a text view and its parent:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
                .onTapGesture {
                    print("Text tapped")
                }
            .onTapGesture {
                print("VStack tapped")
            }
        }
    }
}
```

In this situation SwiftUI will always give the child's gesture priority, which means when you tap the text view above you'll see "Text tapped". However, if you want to change that you can use the **highPriorityGesture()** modifier to force the parent's gesture to trigger instead, like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
                .onTapGesture {
                    print("Text tapped")
                }
        }
        .highPriorityGesture(
            TapGesture()
                .onEnded { _ in
                    print("VStack tapped")
                }
        )
    }
}
```

Alternatively, you can use the **simultaneousGesture()** modifier to tell SwiftUI you want both the parent and child gestures to trigger at the same time, like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
                .onTapGesture {
                    print("Text tapped")
                }
        }
        .simultaneousGesture(
            TapGesture()
                .onEnded { _ in
                    print("VStack tapped")
                }
        )
    }
}
```

Project 17: Flashzilla

```
)  
}  
}
```

That will print both “Text tapped” and “VStack tapped”.

Finally, SwiftUI lets us create gesture sequences, where one gesture will only become active if another gesture has first succeeded. This takes a little more thinking because the gestures need to be able to reference each other, so you can’t just attach them directly to a view.

Here’s an example that shows gesture sequencing, where you can drag a circle around but only if you long press on it first:

```
struct ContentView: View {  
    // how far the circle has been dragged  
    @State private var offset = CGSize.zero  
  
    // whether it is currently being dragged or not  
    @State private var isDragging = false  
  
    var body: some View {  
        // a drag gesture that updates offset and isDragging as it  
        moves around  
        let dragGesture = DragGesture()  
            .onChanged { value in offset = value.translation }  
            .onEnded { _ in  
                withAnimation {  
                    offset = .zero  
                    isDragging = false  
                }  
            }  
        }  
  
        // a long press gesture that enables isDragging
```

```
let pressGesture = LongPressGesture()
    .onEnded { value in
        withAnimation {
            isDragging = true
        }
    }

// a combined gesture that forces the user to long press
then drag
let combined = pressGesture.sequenced(before: dragGesture)

// a 64x64 circle that scales up when it's dragged, sets
its offset to whatever we had back from the drag gesture, and
uses our combined gesture
Circle()
    .fill(.red)
    .frame(width: 64, height: 64)
    .scaleEffect(isDragging ? 1.5 : 1)
    .offset(offset)
    .gesture(combined)
}

}
```

Gestures are a really great way to make fluid, interesting user interfaces, but make sure you show users how they work otherwise they can just be confusing!

Making vibrations with UINotificationFeedbackGenerator and Core Haptics

Although SwiftUI doesn't come with any haptic functionality built in, it's easy enough for us to add using UIKit and Core Haptics - two frameworks built right into the system, and available on all modern iPhones. If you haven't heard the term before, "haptics" involves small motors in the device to create sensations such as taps and vibrations.

UIKit has a very simple implementation of haptics, but that doesn't mean you should rule it out: it can be simple because it focuses on built-in haptics such as "success" or "failure", which means users can come to learn how certain things feel. That is, if you use the success haptic then some users – particularly those who rely on haptics more heavily, such as blind users – will be able to know the result of an operation without any further output from your app.

To try out UIKit's haptics, add this method to **ContentView**:

```
func simpleSuccess() {
    let generator = UINotificationFeedbackGenerator()
    generator.notificationOccurred(.success)
}
```

You can trigger that with a simple **onTapGesture()**, such as this one:

```
Text("Hello, World!")
    .onTapGesture(perform: simpleSuccess)
```

Try replacing **.success** with **.error** or **.warning** and see if you can tell the difference – **.success** and **.warning** are similar but different, I think.

For more advanced haptics, Apple provides us with a whole framework called Core Haptics.

Making vibrations with UNNotificationFeedbackGenerator and Core Haptics

This thing feels like a real labor of love by the Apple team behind it, and I think it was one of the real hidden gems introduced in iOS 13 – certainly I pounced on it as soon as I saw the release notes!

Core Haptics lets us create hugely customizable haptics by combining taps, continuous vibrations, parameter curves, and more. I don't want to go into too much depth here because it's a bit off topic, but I do at least want to give you an example so you can try it for yourself.

First add this new import near the top of ContentView.swift:

```
import CoreHaptics
```

Next, we need to create an instance of **CHHapticEngine** as a property – this is the actual object that's responsible for creating vibrations, so we need to create it up front *before* we want haptic effects.

So, add this property to **ContentView**:

```
@State private var engine: CHHapticEngine?
```

We're going to create that as soon as our main view appears. When creating the engine you can attach handlers to help resume activity if it gets stopped, such as when the app moves to the background, but here we're going to keep it simple: if the current device supports haptics we'll start the engine, and print an error if it fails.

Add this method to **ContentView**:

```
func prepareHaptics() {
    guard
        CHHapticEngine.capabilitiesForHardware().supportsHaptics else
    { return }

    do {
        engine = try CHHapticEngine()
```

Project 17: Flashzilla

```
    try engine?.start()
} catch {
    print("There was an error creating the engine: \
(error.localizedDescription)")
}
}
```

Now for the fun part: we can configure parameters that control how strong the haptic should be (`.hapticIntensity`) and how “sharp” it is (`.hapticSharpness`), then put those into a haptic event with a relative time offset. “Sharpness” is an odd term, but it will make more sense once you’ve tried it out – a sharpness value of 0 really does feel dull compared to a value of 1. As for the relative time, this lets us create lots of haptic events in a single sequence.

Add this method to **ContentView** now:

```
func complexSuccess() {
    // make sure that the device supports haptics
    guard
        CHHapticEngine.capabilitiesForHardware().supportsHaptics else
    { return }
    var events = [CHHapticEvent]()

    // create one intense, sharp tap
    let intensity =
        CHHapticEventParameter(parameterID: .hapticIntensity, value: 1)
    let sharpness =
        CHHapticEventParameter(parameterID: .hapticSharpness, value: 1)
    let event = CHHapticEvent(eventType: .hapticTransient,
parameters: [intensity, sharpness], relativeTime: 0)
    events.append(event)

    // convert those events into a pattern and play it
    immediately
```

Making vibrations with UINotificationFeedbackGenerator and Core Haptics

```
do {
    let pattern = try CHHapticPattern(events: events,
parameters: [])
    let player = try engine?.makePlayer(with: pattern)
    try player?.start(atTime: 0)
} catch {
    print("Failed to play pattern: \
(error.localizedDescription).")
}
}
```

To try out our custom haptics, modify the **body** property of **ContentView** to this:

```
Text("Hello, World!")
    .onAppear(perform: prepareHaptics)
    .onTapGesture(perform: complexSuccess)
```

That makes sure the haptics system is started so the tap gesture works correctly.

If you want to experiment with haptics further, replace the **let intensity**, **let sharpness**, and **let event** lines with whatever haptics you want. For example, if you replace those three lines with this next code you'll get several taps of increasing then decreasing intensity and sharpness:

```
for i in stride(from: 0, to: 1, by: 0.1) {
    let intensity =
CHHapticEventParameter(parameterID: .hapticIntensity, value:
Float(i))
    let sharpness =
CHHapticEventParameter(parameterID: .hapticSharpness, value:
Float(i))
    let event = CHHapticEvent(eventType: .hapticTransient,
parameters: [intensity, sharpness], relativeTime: i)
    events.append(event)
}
```

Project 17: Flashzilla

```
}
```

```
for i in stride(from: 0, to: 1, by: 0.1) {
    let intensity =
CHHapticEventParameter(parameterID: .hapticIntensity, value:
Float(1 - i))
    let sharpness =
CHHapticEventParameter(parameterID: .hapticSharpness, value:
Float(1 - i))
    let event = CHHapticEvent(eventType: .hapticTransient,
parameters: [intensity, sharpness], relativeTime: 1 + i)
    events.append(event)
}
```

Disabling user interactivity with `allowsHitTesting()`

SwiftUI has an advanced hit testing algorithm that uses both the frame of a view and often also its contents. For example, if you add a tap gesture to a text view then all parts of the text view are tappable – you can't tap through the text if you happen to press exactly where a space is. On the other hand, if you attach the same gesture to a circle then SwiftUI *will* ignore the transparent parts of the circle.

To demonstrate this, here's a circle overlapping a rectangle using a `ZStack`, both with `onTapGesture()` modifiers:

```
ZStack {
    Rectangle()
        .fill(.blue)
        .frame(width: 300, height: 300)
        .onTapGesture {
            print("Rectangle tapped!")
        }

    Circle()
        .fill(.red)
        .frame(width: 300, height: 300)
        .onTapGesture {
            print("Circle tapped!")
        }
}
```

If you try that out, you'll find that tapping inside the circle prints "Circle tapped", but on the rectangle behind the circle prints "Rectangle tapped" – even though the circle actually has the same frame as the rectangle.

Project 17: Flashzilla

SwiftUI lets us control user interactivity in two useful ways, the first of which is the **allowsHitTesting()** modifier. When this is attached to a view with its parameter set to false, the view isn't even considered tappable. That doesn't mean it's *inert*, though, just that it doesn't catch any taps – things behind the view will get tapped instead.

Try adding it to our circle like this:

```
Circle()
    .fill(.red)
    .frame(width: 300, height: 300)
    .onTapGesture {
        print("Circle tapped!")
    }
    .allowsHitTesting(false)
```

Now tapping the circle will always print “Rectangle tapped!”, because the circle will refuse to respond to taps.

The other useful way of controlling user interactivity is with the **contentShape()** modifier, which lets us specify the tappable shape for something. By default the tappable shape for a circle is a circle of the same size, but you can specify a different shape instead like this:

```
Circle()
    .fill(.red)
    .frame(width: 300, height: 300)
    .contentShape(Rectangle())
    .onTapGesture {
        print("Circle tapped!")
    }
```

Where the **contentShape()** modifier really becomes useful is when you tap actions attached to stacks with spacers, because by default SwiftUI won't trigger actions when a stack spacer is tapped.

Disabling user interactivity with allowsHitTesting()

Here's an example you can try out:

```
 VStack {
    Text("Hello")
    Spacer().frame(height: 100)
    Text("World")
}
.onTapGesture {
    print("VStack tapped!")
}
```

If you run that you'll find you can tap the "Hello" label and the "World" label, but not the space in between. However, if we use **contentShape(Rectangle())** on the **VStack** then the whole area for the stack becomes tappable, including the spacer:

```
 VStack {
    Text("Hello")
    Spacer().frame(height: 100)
    Text("World")
}
.contentShape(Rectangle())
.onTapGesture {
    print("VStack tapped!")
}
```

Triggering events repeatedly using a timer

iOS comes with a built-in **Timer** class that lets us run code on a regular basis. This uses a system of *publishers* that comes from an Apple framework called Combine. We've actually been using parts of Combine for many apps in this series, although it's unlikely you noticed it. For example, both the **@Published** property wrapper and **ObservableObject** protocols both come from Combine, but we didn't need to know that because when you import SwiftUI we also implicitly import parts of Combine.

Apple's core system library is called Foundation, and it gives us things like **Data**, **Date**, **SortDescriptor**, **User Defaults**, and much more. It also gives us the **Timer** class, which is designed to run a function after a certain number of seconds, but it can also run code repeatedly. Combine adds an extension to this so that timers can become *publishers*, which are things that announce when their value changes. This is where the **@Published** property wrapper gets its name from, and timer publishers work the same way: when your time interval is reached, Combine will send an announcement out containing the current date and time.

The code to create a timer publisher looks like this:

```
let timer = Timer.publish(every: 1, on: .main,  
in: .common).autoconnect()
```

That does several things all at once:

1. It asks the timer to fire every 1 second.
2. It says the timer should run on the main thread.
3. It says the timer should run on the common run loop, which is the one you'll want to use most of the time. (Run loops lets iOS handle running code while the user is actively doing something, such as scrolling in a list.)
4. It connects the timer immediately, which means it will start counting time.
5. It assigns the whole thing to the **timer** constant so that it stays alive.

If you remember, back in project 7 I said “`@Published` is more or less half of `@State`” – it sends change announcements that something else can monitor. In the case of regular publishers like this one, we need to catch the announcements by hand using a new modifier called `onReceive()`. This accepts a publisher as its first parameter and a function to run as its second, and it will make sure that function is called whenever the publisher sends its change notification.

For our timer example, we could receive its notifications like this:

```
Text("Hello, World!")
    .onReceive(timer) { time in
        print("The time is now \(time)")
    }
```

That will print the time every second until the timer is finally stopped.

Speaking of stopping the timer, it takes a little digging to stop the one we created. You see, the `timer` property we made is an autoconnected publisher, so we need to go to its *upstream publisher* to find the timer itself. From there we can connect to the timer publisher, and ask it to cancel itself. Honestly, if it weren’t for code completion this would be rather hard to find, but here’s how it looks in code:

```
timer.upstream.connect().cancel()
```

For example, we could update our existing example so that it fires the timer only five times, like this:

```
struct ContentView: View {
    let timer = Timer.publish(every: 1, on: .main,
    in: .common).autoconnect()
    @State private var counter = 0
```

Project 17: Flashzilla

```
var body: some View {
    Text("Hello, World!")
    .onReceive(timer) { time in
        if counter == 5 {
            timer.upstream.connect().cancel()
        } else {
            print("The time is now \(time)")
        }
        counter += 1
    }
}
```

Before we're done, there's one more important timer concept I want to show you: if you're OK with your timer having a little float, you can specify some *tolerance*. This allows iOS to perform important energy optimization, because it can fire the timer at any point between its scheduled fire time and its scheduled fire time plus the tolerance you specify. In practice this means the system can perform *timer coalescing*: it can push back your timer just a little so that it fires at the same time as one or more other timers, which means it can keep the CPU idling more and save battery power.

As an example, this adds half a second of tolerance to our timer:

```
let timer = Timer.publish(every: 1, tolerance: 0.5, on: .main,
    in: .common).autoconnect()
```

If you need to keep time strictly then leaving off the **tolerance** parameter will make your timer as accurate as possible, but please note that even without any tolerance the **Timer** class is still "best effort" – the system makes no guarantee it will execute precisely.

How to be notified when your SwiftUI app moves to the background

SwiftUI can detect when your app moves to the background (i.e., when the user returns to the home screen), and when it comes back to the foreground, and if you put those two together it allows us to make sure our app pauses and resumes work depending on whether the user can see it right now or not.

This is done using three steps:

1. Adding a new property to watch an environment value called **scenePhase**.
2. Using **onChange()** to watch for the scene phase changing.
3. Responding to the new scene phase somehow.

You might wonder why it's called *scene phase* as opposed to something to do with your current app state, but remember that on iPad the user can run multiple instances of your app at the same time – they can have multiple windows, known as *scenes*, each in a different state.

To see the various scene phases in action, try this code:

```
struct ContentView: View {  
    @Environment(\.scenePhase) var scenePhase  
  
    var body: some View {  
        Text("Hello, world!")  
            .padding()  
            .onChange(of: scenePhase) { newPhase in  
                if newPhase == .active {  
                    print("Active")  
                } else if newPhase == .inactive {  
                    print("Inactive")  
                }  
            }  
    }  
}
```

Project 17: Flashzilla

```
        print("Inactive")
    } else if newPhase == .background {
        print("Background")
    }
}
}
```

When you run that back, try going to the home screen in your simulator, locking the virtual device, and other common activities to see how the scene phase changes.

As you can see, there are three scene phases we need to care about:

- Active scenes are running right now, which on iOS means they are visible to the user. On macOS an app's window might be wholly hidden by another app's window, but that's okay – it's still considered to be active.
- Inactive scenes are running and might be visible to the user, but the user isn't able to access them. For example, if you're swiping down to partially reveal the control center then the app underneath is considered inactive.
- Background scenes are not visible to the user, which on iOS means they might be terminated at some point in the future.

Supporting specific accessibility needs with SwiftUI

SwiftUI gives us a number of environment properties that describe the user's custom accessibility settings, and it's worth taking the time to read and respect those settings.

Back in project 15 we looked at accessibility labels and hints, traits, groups, and more, but these settings are different because they are provided through the environment. This means SwiftUI automatically monitors them for changes and will reinvoke our **body** property whenever one of them changes.

For example, one of the accessibility options is “Differentiate without color”, which is helpful for the 1 in 12 men who have color blindness. When this setting is enabled, apps should try to make their UI clearer using shapes, icons, and textures rather than colors.

To use this, just add an environment property like this one:

```
@Environment(\.accessibilityDifferentiateWithoutColor) var  
differentiateWithoutColor
```

That will be either true or false, and you can adapt your UI accordingly. For example, in the code below we use a simple green background for the regular layout, but when Differentiate Without Color is enabled we use a black background and add a checkmark instead:

```
struct ContentView: View {  
    @Environment(\.accessibilityDifferentiateWithoutColor) var  
differentiateWithoutColor  
  
    var body: some View {  
        HStack {  
            if differentiateWithoutColor {  
                Image(systemName: "checkmark.circle")  
            }  
        }  
    }  
}
```

Project 17: Flashzilla

```
        Text("Success")
    }
    .padding()
    .background(differentiateWithoutColor ? .black : .green)
    .foregroundColor(.white)
    .clipShape(Capsule())
}
}
```

You can test that in the simulator by going to the Settings app and choosing Accessibility > Display & Text Size > Differentiate Without Color.

Another common option is Reduce Motion, which again is available in the simulator under Accessibility > Motion > Reduce Motion. When this is enabled, apps should limit the amount of animation that causes movement on screen. For example, the iOS app switcher makes views fade in and out rather than scale up and down.

With SwiftUI, this means we should restrict the use of **withAnimation()** when it involves movement, like this:

```
struct ContentView: View {
    @Environment(\.accessibilityReduceMotion) var reduceMotion
    @State private var scale = 1.0

    var body: some View {
        Text("Hello, World!")
            .scaleEffect(scale)
            .onTapGesture {
                if reduceMotion {
                    scale *= 1.5
                } else {
                    withAnimation {

```

I don't know about you, but I find that rather annoying to use. Fortunately we can add a little wrapper function around `withAnimation()` that uses UIKit's `UIAccessibility` data directly, allowing us to bypass animation automatically:

```
func withOptionalAnimation<Result>(_ animation: Animation?  
= .default, _ body: () throws -> Result) rethrows -> Result {  
    if UIAccessibility.isReduceMotionEnabled {  
        return try body()  
    } else {  
        return try withAnimation(animation, body)  
    }  
}
```

So, when `Reduce Motion Enabled` is true the closure code that's passed in is executed immediately, otherwise it's passed along using `withAnimation()`. The whole `throws/rethrows` thing is more advanced Swift, but it's a direct copy of the function signature for `withAnimation()` so that the two can be used interchangeably.

Use it like this:

```
struct ContentView: View {  
    @State private var scale = 1.0  
  
    var body: some View {  
        Text("Hello, World!")  
            .scaleEffect(scale)  
    }  
}
```

Project 17: Flashzilla

```
.onTapGesture {
    withOptionalAnimation {
        scale *= 1.5
    }
}
```

Using this approach you don't need to repeat your animation code every time.

One last option you should consider supporting is Reduce Transparency, and when that's enabled apps should reduce the amount of blur and translucency used in their designs to make doubly sure everything is clear.

For example, this code uses a solid black background when Reduce Transparency is enabled, otherwise using 50% transparency:

```
struct ContentView: View {
    @Environment(\.accessibilityReduceTransparency) var
reduceTransparency

    var body: some View {
        Text("Hello, World!")
            .padding()
            .background(reduceTransparency ? .black : .black.opacity(
0.5))
            .foregroundColor(.white)
            .clipShape(Capsule())
    }
}
```

That's the final technique I want you to learn ahead of building the real project, so please reset your project back to its original state so we have a clean slate to start on.

Supporting specific accessibility needs with SwiftUI

Designing a single card view

In this project we want users to see a card with some prompt text for whatever they want to learn, such as “What is the capital city of Scotland?”, and when they tap it we’ll reveal the answer, which in this case is of course Edinburgh.

A sensible place to start for most projects is to define the data model we want to work with: what does one card of information look like? If you wanted to take this app further you could store some interesting statistics such as number of times shown and number of times correct, but here we’re only going to store a string for the prompt and a string for the answer. To make our lives easier, we’re also going to add an example card as a static property, so we have some test data for previewing and prototyping.

So, create a new Swift file called Card.swift and give it this code:

```
struct Card {  
    let prompt: String  
    let answer: String  
  
    static let example = Card(prompt: "Who played the 13th Doctor  
in Doctor Who?", answer: "Jodie Whittaker")  
}
```

In terms of showing that in a SwiftUI view, we need something slightly more complicated: yes there will be two text labels shown one above the other, but we also need to show a white card behind them to bring our UI to life, then add just a touch of padding to the text so it doesn’t quite go to the edge of the card behind it. In SwiftUI terms this means a **VStack** for the two labels, inside a **ZStack** with a white **RoundedRectangle**.

I don’t know if you’ve used flashcards to learn before, but they have a very particular shape that makes them wider than they are high. This makes sense if you think about it: you’re usually only writing two or three lines of text, so it’s more natural to write long-ways than short-ways.

All our apps so far haven't really cared about device orientation, but we're going to make this one work only in landscape. This gives us more room to draw our cards, and it will also work better once we introduce gestures later on.

To force landscape mode, go to your target options in the Info tab, open the disclosure indicator for the key “Supported interface orientations (iPhone)” and delete the portrait option so it leaves just the two landscape options.

With that done we can take our first pass at a view to represent one card in our app. Create a new SwiftUI view called “CardView” and give it this code:

```
struct CardView: View {
    let card: Card

    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 25, style: .continuous)
                .fill(.white)

            VStack {
                Text(card.prompt)
                    .font(.largeTitle)
                    .foregroundColor(.black)

                Text(card.answer)
                    .font(.title)
                    .foregroundColor(.gray)
            }
            .padding(20)
            .multilineTextAlignment(.center)
        }
        .frame(width: 450, height: 250)
    }
}
```

Project 17: Flashzilla

}

Tip: A width of 450 is no accident: the smallest iPhones have a landscape width of 480 points, so this means our card will be fully visible on all devices.

That will break the **CardView_Previews** struct because it requires a **card** parameter to be passed in, but we already added a static example directly to the **Card** struct for this very purpose. So, update the **CardView_Previews** struct to this:

```
struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView(card: Card.example)
    }
}
```

If you take a look at the preview you should see our example card showing, but you can't actually see that it's a card – it has a white background, and so does it doesn't stand out against the default background of our view. This will become doubly problematic when we have a stack of cards to work through, because they'll all have white backgrounds and kind of blend into each other.

There's a simple fix for this: we can add a shadow to the **RoundedRectangle** so we get a gentle depth effect. This will help us right now by making our white card stand out from the white background, but when we start adding more cards it will look even better because the shadows will add up.

So, add this modifier below the **fill(.white)**:

```
.shadow(radius: 10)
```

Now, right now you can see both the prompt and the answer at the same time, but obviously that isn't going to help anyone learn. So, to finish this step we're going to hide the answer label by default, and toggle its visibility whenever the card is tapped.

So, start by adding this new `@State` property to `CardView`:

```
@State private var isShowingAnswer = false
```

Now wrap the answer view in a condition for that Boolean, like this:

```
if isShowingAnswer {  
    Text(card.answer)  
        .font(.title)  
        .foregroundColor(.gray)  
}
```

That simple change means it will only show the answer when `isShowingAnswer` is true.

The final step is to add an `onTapGesture()` modifier to the `ZStack`, by putting this code after the `frame()` modifier:

```
.onTapGesture {  
    isShowingAnswer.toggle()  
}
```

That's our card view done for the time being, so if you want to see it in action go back to `ContentView.swift` and replace its `body` property with this:

```
var body: some View {  
    CardView(card: Card.example)  
}
```

When you run the project you'll see the app jumps into landscape mode automatically, and our default card appears – a good start!

Building a stack of cards

Now that we've designed one card and its associated card view, the next step is to build a stack of those cards to represent the things our user is trying to learn. This stack will change as the app is used because the user will be able to remove cards, so we need to mark it with `@State`.

Right now we don't have any way of adding cards, so we're going to add a stack of 10 using our example card. Swift's arrays have a helpful initializer, `init(repeating:count:)`, which takes one value and repeats it a number of times to create the array. In our case we can use that with our example **Card** to create a simple test array.

So, start by adding this property to **ContentView**:

```
@State private var cards = [Card](repeating: Card.example, count: 10)
```

Our main **ContentView** is going to contain a number of overlapping elements inside stacks, but for now we're just going to put in a rough skeleton:

1. Our stack of cards will be placed inside a **ZStack** so we can make them partially overlap with a neat 3D effect.
2. Around that **ZStack** will be a **VStack**. Right now that **VStack** won't do much, but later on it will allow us to place a timer above our cards.
3. Around that **VStack** will be another **ZStack**, so we can place our cards and timer on top of a background.

Right now these stacks probably feel like overkill, but it will make more sense as we progress.

The only complex part of our next code is how we position the cards inside the card stack so they have slight overlapping. I've said it before, but the best way to write SwiftUI code is to carve off any messy calculations so they are handled as methods or modifiers.

In this case we're going to create a new **stacked()** modifier that takes a position in an array along with the total size of the array, and offsets a view by some amount based on those

values. This will allow us to create an attractive card stack where each card is a little further down the screen than the ones before it.

Add this extension to `ContentView.swift`, outside of the `ContentView` struct:

```
extension View {
    func stacked(at position: Int, in total: Int) -> some View {
        let offset = Double(total - position)
        return self.offset(x: 0, y: offset * 10)
    }
}
```

As you can see, that pushes views down by 10 points for each place they are in the array: 0, then 10, 20, 30, and so on.

With that simple modifier we can now build a really nice card stack effect using the layout I described earlier. Replace your current `body` property in `ContentView` with this:

```
var body: some View {
    ZStack {
        VStack {
            ZStack {
                ForEach(0..
```

When you run that back you'll see what I mean about the shadows building up as the card depth increases. It looks quite stark against a white background, but if we add a background

Project 17: Flashzilla

picture you'll see it looks better.

In the GitHub files for this project you'll see `background@2x.jpg` and `background@3x.jpg` – please drag those both into your asset catalog so we can use them.

Now add this **Image** view into **ContentView**, just inside the initial **ZStack**:

```
Image("background")
    .resizable()
    .ignoresSafeArea()
```

Adding a background image is only a small change, but I think it makes the whole app look better!

Moving views with DragGesture and offset()

SwiftUI lets us attach custom gestures to any view, then use the values created by those gestures to manipulate the rest of our views. To demonstrate this, we're going to attach a **DragGesture** to **CardView** so that it can be moved around, and we'll also use the values generated by that gesture to control the opacity and rotation of the view – it will curve away and fade out as it's dragged. This takes surprisingly little code, because SwiftUI does so much for us; I think you'll be really impressed!

First, add this new **@State** property to **CardView**, to track how far the user has dragged:

```
@State private var offset = CGSize.zero
```

Next we're going to add three modifiers to **CardView**, placed directly below the **frame()** modifier. **Remember: the order in which you apply modifiers *matters*, and nowhere is this more true than when working with offsets and rotations.**

If we rotate then offset, then the offset is applied based on the rotated axis of our view. For example, if we move something 100 pixels to its left then rotate 90 degrees, we'd end up with it being 100 pixels to the left and rotated 90 degrees. But if we rotated 90 degrees then moved it 100 pixels to its left, we'd end up with something rotated 90 degrees and moved 100 pixels directly down, because its concept of “left” got rotated.

Where things get doubly trick is when you factor in how SwiftUI creates new views by wrapping modifiers. When it comes to moving and rotating, this means if we want a view to slide directly to true west (regardless of its rotation) while also rotating it, we need to put the rotation first *then* the offset.

Now, **offset.width** will contain how far the user dragged our card, but we don't want to use that for our rotation because the card would spin too fast So, instead add this modifier below **frame()**, so we use 1/5th of the drag amount:

Project 17: Flashzilla

```
.rotationEffect(.degrees(Double(offset.width / 5)))
```

Next we're going to apply our movement, so the card slides relative to the horizontal drag amount. Again, we're not going to use the original value of **offset.width** because it would require the user to drag a long way to get any meaningful results, so instead we're going to *multiply* it by 5 so the cards can be swiped away with small gestures.

Add this modifier below the previous one:

```
.offset(x: offset.width * 5, y: 0)
```

While we're here, I want to add one more modifier based on the drag gesture: we're going to make the card fade out as it's dragged further away.

Now, the calculation for this view takes a little thinking, and I wouldn't blame you if you wanted to spin this off into a method rather than putting it inline. Here's how it works:

- We're going to take 1/50th of the drag amount, so the card doesn't fade out too quickly.
- We don't care whether they have moved to the left (negative numbers) or to the right (positive numbers), so we'll put our value through the **abs()** function. If this is given a positive number it returns the same number, but if it's given a negative number it removes the negative sign and returns the same value as a positive number.
- We then use this result to subtract from 2.

The use of 2 there is intentional, because it allows the card to stay opaque while being dragged just a little. So, if the user hasn't dragged at all the opacity is 2.0, which is identical to the opacity being 1. If they drag it 50 points left or right, we divide that by 50 to get 1, and subtract that from 2 to get 1, so the opacity is still 1 – the card is still fully opaque. But *beyond* 50 points we start to fade out the card, until at 100 points left or right the opacity is 0.

Add this modifier below the previous two:

```
.opacity(2 - Double(abs(offset.width / 50)))
```

So, we've created a property to store the drag amount, and added three modifiers that use the drag amount to change the way the view is rendered. What remains is the most important part: we need to actually attach a **DragGesture** to our card so that it updates **offset** as the user drags the card around. Drag gestures have two useful modifiers of their own, letting us attach functions to be triggered when the gesture has changed (called every time they move their finger), and when the gesture has ended (called when they lift their finger).

Both of these functions are handed the current gesture state to evaluate. In our case we'll be reading the **translation** property to see where the user has dragged to, and we'll be using that to set our **offset** property, but you can also read the start location, predicted end location, and more. When it comes to the *ended* function, we'll be checking whether the user moved it more than 100 points in either direction so we can prepare to remove the card, but if they haven't we'll set **offset** back to 0.

Add this **gesture()** modifier below the previous three:

```
.gesture(
    DragGesture()
        .onChanged { gesture in
            offset = gesture.translation
        }
        .onEnded { _ in
            if abs(offset.width) > 100 {
                // remove the card
            } else {
                offset = .zero
            }
        }
)
```

Go ahead and run the app now: you should find the cards move, rotate, and fade away as they are dragged, and if you drag more than a certain distance they *stay* away rather than jumping back to their original location.

Project 17: Flashzilla

This works well, but to really finish this step we need to fill in the `// remove the card` comment so the card actually gets removed in the parent view. Now, we *don't* want **CardView** to call up to **ContentView** and manipulate its data directly, because that causes spaghetti code. Instead, a better idea is to store a closure parameter inside **CardView** that can be filled with whatever code we want later on – it means we have the flexibility to get a callback in **ContentView** without explicitly tying the two views together.

So, add this new property to **CardView** below its existing **card** property:

```
var removal: (( ) -> Void)? = nil
```

As you can see, that's a closure that accepts no parameters and sends nothing back, defaulting to **nil** so we don't need to provide it unless it's explicitly needed.

Now we can replace `// remove the card` with a call to that closure:

```
removal?()
```

Tip: That question mark in there means the closure will only be called if it has been set.

Back in **ContentView** we can now write a method to handle removing a card, then connect it to that closure.

First, add this method that takes an index in our **cards** array and removes that item:

```
func removeCard(at index: Int) {
    cards.remove(at: index)
}
```

Finally, we can update the way we create **CardView** so that we use trailing closure syntax to remove the card when it's dragged more than 100 points. This is just a matter of calling the **removeCard(at:)** method we just wrote, but if we wrap that inside a **withAnimation()** call then the other cards will automatically slide up.

Moving views with DragGesture and offset()

Here's how your code should look:

```
ForEach(0..
```

Go ahead and run the app now – I think the result really looks great, and you can now swipe your way through all the cards in the stack until you reach the end!

Coloring views as we swipe

Users can swipe our cards left or right to mark them as being guessed correctly or not, but there's no visual distinction between the two directions. Borrowing controls from dating apps like Tinder, we'll make swiping *right* good (they guessed the answer correctly), and swiping *left* bad (they were wrong).

We'll solve this problem in two ways: for a phone with default settings we'll make the cards become colored green or red before fading away, but if the user enabled the Differentiate Without Color setting we'll leave the cards as white and instead show some extra UI over our background.

Let's start with a first pass on the cards themselves. Right now our card view is created with this background:

```
RoundedRectangle(cornerRadius: 25, style: .continuous)
    .fill(.white)
    .shadow(radius: 10)
```

We're going to replace that with some more advanced code: we'll give it a background of the same rounded rectangle except in green or red depending on the gesture movement, then we'll make the white fill from above fade out as the drag movement gets larger.

First, the background. Add this directly before the **shadow()** modifier:

```
.background(
    RoundedRectangle(cornerRadius: 25, style: .continuous)
        .fill(offset.width > 0 ? .green : .red)
)
```

As for the white fill opacity, this is going to be similar to the **opacity()** modifier we added previously except we'll use 1 minus 1/50th of the gesture width rather than 2 minus the gesture width. This creates a really nice effect: we used 2 minus earlier because it meant the card

would have to move at least 50 points before fading away, but for the card fill we're going to use 1 minus so that it starts becoming colored straight away.

Replace the existing `fill()` modifier with this:

```
.fill(
    .white
    .opacity(1 - Double(abs(offset.width / 50)))
)
```

If you run the app now you'll see that the cards blend from white to either red or green, *then* start to fade out. Awesome!

However, as nice as our code is it won't work well for folks with red/green color blindness – they will see the brightness of the cards change, but it won't be clear which side is which.

To fix this we're going to add an environment property to track whether we should be using color for this purpose or not, then disable the red/green effect when that property is true.

Start by adding this new property to **CardView**, before the existing properties:

```
@Environment(\.accessibilityDistinguishWithoutColor) var
distinguishWithoutColor
```

Now we can use that for both the fill and background for our **RoundedRectangle** to make sure we fade out the white smoothly. It's important we use it for both, because as the card fades out the background color will start to bleed through the fill.

So, replace your current **RoundedRectangle** code with this:

```
RoundedRectangle(cornerRadius: 25, style: .continuous)
.fill(
    distinguishWithoutColor
    ? .white
```

Project 17: Flashzilla

```
    : .white
    .opacity(1 - Double(abs(offset.width / 50)))

)
.background(
    differentiateWithoutColor
    ? nil
    : RoundedRectangle(cornerRadius: 25, style: .continuous)
        .fill(offset.width > 0 ? .green : .red)
)
.shadow(radius: 10)
```

So, when in a default configuration our cards will fade to green or red, but when Differentiate Without Color is enabled that won't be used. Instead we need to provide some extra UI in **ContentView** to make it clear which side is positive and which is negative.

Earlier we made a very particular structure of stacks in **ContentView**: we had a **ZStack**, then a **VStack**, then another **ZStack**. That first **ZStack**, the outermost one, allows us to have our background and card stack overlapping, and we're also going to put some buttons in that stack so users can see which side is "good".

First, add this property to **ContentView**:

```
@Environment(\.accessibilityDifferentiateWithoutColor) var
differentiateWithoutColor
```

Now add these new views directly after the **VStack**:

```
if differentiateWithoutColor {
    VStack {
        Spacer()
    }
    HStack {
```

```
Image(systemName: "xmark.circle")
    .padding()
    .background(.black.opacity(0.7))
    .clipShape(Circle())
Spacer()
Image(systemName: "checkmark.circle")
    .padding()
    .background(.black.opacity(0.7))
    .clipShape(Circle())
}
.foregroundColor(.white)
.font(.largeTitle)
.padding()
}
}
```

That creates another **VStack**, this time starting with a spacer so that the images inside the stacks are pushed to the bottom of the screen. And with that condition around them all, they'll only appear when Differentiate Without Color is enabled, so most of the time our UI stays clear.

All this extra work *matters*: it makes sure users get a great experience regardless of their accessibility needs, and that's what we should always be aiming for.

Counting down with a Timer

If we bring together Foundation, SwiftUI, and Combine, we can add a timer to our app to add a little bit of pressure to the user. A simple implementation of this doesn't take much work, but it also has a bug that requires some extra work to fix.

For our first pass of the timer, we're going to create two new properties: the timer itself, which will fire once a second, and a **timeRemaining** property, from which we'll subtract 1 every time the timer fires. This will allow us to show how many seconds remain in the current app run, which should give the user a gentle incentive to speed up.

So, start by adding these two new properties to **ContentView**:

```
@State private var timeRemaining = 100  
let timer = Timer.publish(every: 1, on: .main,  
in: .common).autoconnect()
```

That gives the user 100 seconds to start with, then creates and starts a timer that fires once a second on the main thread.

Whenever that timer fires, we want to subtract 1 from **timeRemaining** so that it counts down. We could try and do some date mathematics here by storing a start date and showing the difference between that and the current date, but there really is no need as you'll see!

Add this **onReceive()** modifier to the outermost **ZStack** in **ContentView**:

```
.onReceive(timer) { time in  
    if timeRemaining > 0 {  
        timeRemaining -= 1  
    }  
}
```

Tip: That adds a trivial condition to make sure we never stray into negative numbers.

Counting down with a Timer

That code starts our timer at 100 and makes it count down to 0, but we need to actually display it. This is as simple as adding another text view to our layout, this time with a dark background color to make sure it's clearly visible.

Put this inside the **VStack** that contains the **ZStack** for our cards:

```
Text("Time: \$(timeRemaining)")  
    .font(.largeTitle)  
    .foregroundColor(.white)  
    .padding(.horizontal, 20)  
    .padding(.vertical, 5)  
    .background(.black.opacity(0.75))  
    .clipShape(Capsule())
```

If you've placed it correctly, your layout code should look like this:

```
ZStack {  
    Image("background")  
        .resizable()  
        .ignoresSafeArea()  
  
    VStack {  
        Text("Time: \$(timeRemaining)")  
            .font(.largeTitle)  
            .foregroundColor(.white)  
            .padding(.horizontal, 20)  
            .padding(.vertical, 5)  
            .background(.black.opacity(0.75))  
            .clipShape(Capsule())  
  
        ZStack {
```

You should be able to run the app now and give it a try – it works well enough, right? Well,

Project 17: Flashzilla

there's a small problem:

1. Take a look at the current value in the timer.
2. Press Cmd+H to go back to the home screen.
3. Wait about ten seconds.
4. Now tap your app's icon to go back to the app.
5. What time is shown in the timer?

What I find is that the timer shows a value about three seconds lower than we had when we were in the app previously – the timer runs for a few seconds in the background, then pauses until the app comes back.

We can do better than this: we can detect when our app moves to the background or foreground, then pause and restart our timer appropriately.

First, add two properties to store whether the app is currently active:

```
@Environment(\.scenePhase) var scenePhase  
@State private var isActive = true
```

We have two because the environment value tells us whether the app is active or inactive in terms of its visibility, but we'll also consider the app inactive if the player has gone through their deck of flashcards – it will be active from a scene phase point of view, but we don't keep the timer ticking.

Now add this **onChange()** modifier below the existing **onReceive()** modifier:

```
.onChange(of: scenePhase) { newPhase in  
    if newPhase == .active {  
        isActive = true  
    } else {  
        isActive = false  
    }  
}
```

Finally, modify the **onReceive(timer)** function so it exits immediately if **isActive** is false, like this:

```
.onReceive(timer) { time in
    guard isActive else { return }

    if timeRemaining > 0 {
        timeRemaining -= 1
    }
}
```

And with that small change the timer will automatically pause when the app moves to the background – we no longer lose any mystery seconds.

Ending the app with `allowsHitTesting()`

SwiftUI lets us disable interactivity for a view by setting `allowsHitTesting()` to false, so in our project we can use it to disable swiping on any card when the time runs out by checking the value of `timeRemaining`.

Start by adding this modifier to the innermost `ZStack` – the one that shows our card stack:

```
.allowsHitTesting(timeRemaining > 0)
```

That enables hit testing when `timeRemaining` is 1 or greater, but sets it to false otherwise because the user is out of time.

The *other* outcome is that the user flies through all the cards correctly, and ends with none left. When the final card goes away, right now our timer slides down to the center of the screen, and carries on ticking. What we *want* to happen is for the timer to stop so users can see how fast they were, and also to show a button allowing them to reset their cards and try again.

This takes a little thinking, because just setting `isActive` to false isn't enough – if the app moves to the background and returns `isActive` will be re-enabled even though there are no cards left.

Let's tackle it piece by piece. First, we need a method to run to reset the app so the user can try again, so add this to `ContentView`:

```
func resetCards() {
    cards = [Card](repeating: Card.example, count: 10)
    timeRemaining = 100
    isActive = true
}
```

Second, we need a button to trigger that, shown only when all cards have been removed. Put

Ending the app with allowsHitTesting()

this after the innermost **ZStack**, just below the **allowsHitTesting()** modifier:

```
if cards.isEmpty {  
    Button("Start Again", action: resetCards)  
        .padding()  
        .background(.white)  
        .foregroundColor(.black)  
        .clipShape(Capsule())  
}
```

Now we have code to restart the timer when resetting the cards, but now we need to *stop* the timer when the final card is removed – and make sure it *stays* stopped when coming back to the foreground.

We can solve the first problem by adding this to the end of the **removeCard(at:)** method:

```
if cards.isEmpty {  
    isActive = false  
}
```

As for the second problem – making sure **isActive** *stays* false when returning from the background – we should just update our scene phase code so it explicitly checks for cards:

```
.onChange(of: scenePhase) { newPhase in  
    if newPhase == .active {  
        if cards.isEmpty == false {  
            isActive = true  
        }  
    } else {  
        isActive = false  
    }  
}
```

Project 17: Flashzilla

Done!

Making iPhones vibrate with **UINotificationFeedbackGenerator**

iOS comes with a number of options for generating haptic feedback, and they are all available for us to use in SwiftUI. In its simplest form, this is as simple as creating an instance of one of the subclasses of **UIFeedbackGenerator** then triggering it when you're ready, but for more precise control over feedback you should first call its **prepare()** method to give the Taptic Engine chance to warm up.

Important: Warming up the Taptic Engine helps reduce the latency between us playing the effect and it actually happening, but it also has a battery impact so the system will only stay ready for a second or two after you call **prepare()**.

There are a few different subclasses of **UIFeedbackGenerator** we could use, but the one we'll use here is **UINotificationFeedbackGenerator** because it provides success and failure haptics that are common across iOS. Now, we *could* add one central instance of **UINotificationFeedbackGenerator** to every **ContentView**, but that causes a problem: **ContentView** gets notified whenever a card has been removed, but *isn't* notified when a drag is in progress, which means we don't have the opportunity to warm up the Taptic Engine.

So, instead we're going to give each **CardView** its own instance of **UINotificationFeedbackGenerator** so they can prepare and play them as needed. The system will take care of making sure the haptics are all neatly arranged, so there's no chance of them somehow getting confused.

Add this new property to **CardView**:

```
@State private var feedback = UINotificationFeedbackGenerator()
```

Now find the **removal?()** line in the drag gesture of **CardView**, and change that whole closure to this:

```
if offset.width > 0 {
```

Project 17: Flashzilla

```
    feedback.notificationOccurred(.success)
} else {
    feedback.notificationOccurred(.error)
}

removal?()
```

That alone is enough to get haptics in our app, but there is always the risk that the haptic will be delayed because the Taptic Engine wasn't ready. In this case the haptic will still play, but it could be up to maybe half a second late – enough to feel just that little bit disconnected from our user interface.

To improve this we need to call **prepare()** on our haptic a little *before* triggering it. **It is not enough to call **prepare()** immediately before activating it:** doing so does *not* give the Taptic Engine enough time to warm up, so you won't see any reduction in latency. Instead, you should call **prepare()** as soon as you know the haptic might be needed.

Now, there are two helpful implementation details that you should be aware of.

First, it's OK to call **prepare()** then never triggering the effect – the system will keep the Taptic Engine ready for a few seconds then just power it down again. If you repeatedly call **prepare()** and never trigger it the system might start ignoring your **prepare()** calls until at least one effect has happened.

Second, it's perfectly allowable to call **prepare()** many times before triggering it once – **prepare()** doesn't pause your app while the Taptic Engine warms up, and also doesn't have any real performance cost when the system is already prepared.

Putting these two together, we're going to update our drag gesture so that **prepare()** is called whenever the gesture changes. This means it could be called a hundred times before the haptic is finally triggered, because it will get triggered every time the user moves their finger.

So, modify your **onChanged()** closure to this:

```
.onChanged { offset in
    offset = offset.translation
    feedback.prepare()
}
```

Now go ahead and give the app a try and see what you think – you should be able to feel two very different haptics depending on which direction you swipe.

Before we wrap up with haptics, there's one thing I want you to consider. Years ago PepsiCo challenged mall shoppers to the "Pepsi Challenge": drink a sip of one cola drink and a sip of another, and see which you prefer. The results found that more Americans preferred Pepsi than Coca Cola, despite Coke having a much bigger market share. However, there was a problem: people seemed to pick Pepsi in the test because Pepsi had a sweeter taste, and while that worked well in sip-size amounts it worked *less* well in the sizes of cans and bottles, where people actually preferred Coke.

The reason I'm saying this is because we added two haptic notifications to our app that will get played a *lot*. And while you're testing out in small doses these haptics probably feel great – you're making your phone buzz, and it can be really delightful. However, if you're a serious user of this app then our haptics might hit two problems:

1. The user might find them annoying, because they'll happen once every two or three seconds depending on how fast they are.
2. Worse, the user might become *desensitized* to them – they lose all usefulness either as a notification or as a little spark of delight.

So, now you've tried it for yourself I want you to think about how they should be used. If this were my app I would probably keep the failure haptic, but I think the success haptic could go – that one is likely to be triggered the most often, and it means when the failure haptic plays it feels a little more special.

Fixing the bugs

Our SwiftUI app is looking good so far: we have a stack of cards that can be dragged around to control the app, plus haptic feedback and some accessibility support. But at the same time it's also pretty full of glitches that are holding it back – some big, some small, but all worth addressing.

First, it's possible to drag cards around when they aren't at the top. This is confusing for users because they can grab a card they can't actually see, so this should never be possible.

To fix this we're going to use **allowsHitTesting()** so that only the last card – the one on top – can be dragged around. Find the **stacked()** modifier in **ContentView** and add this directly below:

```
.allowsHitTesting(index == cards.count - 1)
```

Second, our UI is a bit of a mess when used with VoiceOver. If you launch it on a real device with VoiceOver enabled, you'll find that you can tap on the background image to get "Background, image" read out, which is pointless. However, things get worse: make small swipes to the right and VoiceOver will move through all the accessibility elements – it reads out the text from all our cards, even the ones that aren't visible.

To fix the background image problem we should make it use a decorative image so it won't be read out as part of the accessibility layout. Modify the background image to this:

```
Image(decorative: "background")
```

To fix the cards, we need to use an **accessibilityHidden()** modifier with a similar condition to the **allowsHitTesting()** modifier we added a minute ago. In this case, every card that's at an index less than the top card should be hidden from the accessibility system because there's really nothing useful it can do with the card, so add this directly below the **allowsHitTesting()** modifier:

```
.accessibilityHidden(index < cards.count - 1)
```

There's a third accessibility problem with our app, and it's the direct result of using gestures to control things. Yes, gestures are great fun to use most of the time, but if you have specific accessibility needs it can be very hard to use them.

In this app our gestures are causing multiple problems: it's not apparent to VoiceOver users how they should control the app:

1. We don't say that the cards are buttons that can be tapped.
2. When the answer is revealed there is no audible notification of what it was.
3. Users have no way of swiping left or right to move through the cards.

It takes very little work to fix these problems, but the pay off is that our app is much more accessible to everyone.

First, we need to make it clear that our cards are tappable buttons. This is as simple as adding **accessibilityAddTraits()** with **.isButton** to the **ZStack** in **CardView**. Put this after its **opacity()** modifier:

```
.accessibilityAddTraits(.isButton)
```

Now the system will read “Who played the 13th Doctor in Doctor Who? Button” – an important hint to users that the card can be tapped.

Second, we need to help the system to read the answer to the cards as well as the questions. This is *possible* right now, but only if the user swipes around on the screen – it's far from obvious. So, to fix this we're going to detect whether the user has accessibility enabled on their device, and if so automatically toggle between showing the prompt and showing the answer. That is, rather than have the answer appear below the prompt we'll switch it out and just show the answer, which will cause VoiceOver to read it out immediately.

SwiftUI provides a specific environment property that tells us when VoiceOver is running, called **accessibilityVoiceOverEnabled**. So, add this new property to **CardView**:

Project 17: Flashzilla

```
@Environment(\.accessibilityVoiceOverEnabled) var  
voiceOverEnabled
```

Right now our code for displaying the prompt and answer looks like this:

```
 VStack {  
     Text(card.prompt)  
         .font(.largeTitle)  
         .foregroundColor(.black)  
  
     if isShowingAnswer {  
         Text(card.answer)  
             .font(.title)  
             .foregroundColor(.gray)  
     }  
 }
```

We're going to change that so the prompt and answer are shown in a single text view, with **accessibilityEnabled** deciding which layout is shown. Amend your code to this:

```
 VStack {  
     if voiceOverEnabled {  
         Text(isShowingAnswer ? card.answer : card.prompt)  
             .font(.largeTitle)  
             .foregroundColor(.black)  
     } else {  
         Text(card.prompt)  
             .font(.largeTitle)  
             .foregroundColor(.black)  
  
         if isShowingAnswer {  
             Text(card.answer)  
                 .font(.title)  
         }  
     }  
 }
```

```
        .foregroundColor(.gray)
    }
}
```

If you try that out with VoiceOver you'll hear that it works much better – as soon as the card is double-tapped the answer is read out.

Third, we need to make it easier for users to mark cards as correct or wrong, because right now our images just don't cut it. Not only do they stop users from interacting with our app using tap gestures, but they also get read out as their SF Symbols name – “checkmark, circle, image” – rather than anything useful.

To fix this we need to replace the images with buttons that actually remove the cards. We don't actually do anything different if the user was correct or wrong – I need to leave *something* for your challenges! – but we can at least remove the top card from the deck. At the same time, we're going to provide an accessibility label and hint so that users get a better idea of what the buttons do.

So, replace your current **HStack** with those images with this new code:

```
HStack {
    Button {
        withAnimation {
            removeCard(at: cards.count - 1)
        }
    } label: {
        Image(systemName: "xmark.circle")
            .padding()
            .background(.black.opacity(0.7))
            .clipShape(Circle())
    }
    .accessibilityLabel("Wrong")
}
```

Project 17: Flashzilla

```
.accessibilityHint("Mark your answer as being incorrect.")

Spacer()

Button {
    withAnimation {
        removeCard(at: cards.count - 1)
    }
} label: {
    Image(systemName: "checkmark.circle")
    .padding()
    .background(.black.opacity(0.7))
    .clipShape(Circle())
}
.accessibilityLabel("Correct")
.accessibilityHint("Mark your answer as being correct.")
}
```

Because those buttons remain onscreen even when the last card has been removed, we need to add a **guard** check to the start of **removeCard(at:)** to make sure we don't try to remove a card that doesn't exist. So, put this new line of code at the start of that method:

```
guard index >= 0 else { return }
```

Finally, we can make those buttons visible when either **differentiateWithoutColor** is enabled or when VoiceOver is enabled. This means adding another **accessibilityVoiceOverEnabled** property to **ContentView**:

```
@Environment(\.accessibilityVoiceOverEnabled) var
voiceOverEnabled
```

Then modifying the **if differentiateWithoutColor {** condition to this:

```
if differentiateWithoutColor || voiceOverEnabled {
```

With these accessibility changes our app works much better for everyone – good job!

Before we're done, I'd like to add one tiny extra change. Right now if you drag an image a little then let go we set its offset back to zero, which causes it to jump back into the center of the screen. If we attach a spring animation to our card, it will slide into the center, which I think is a much clearer indication to our user of what actually happened.

To make this happen, add an **animation()** modifier to the end of the **ZStack** in **CardView**, directly after the **onTapGesture()**:

```
.animation(.spring(), value: offset)
```

Much better!

Tip: If you look carefully, you might notice the card flash red if you drag it a little to the right then release. More on that later!

Adding and deleting cards

Everything we've worked on so far has used a fixed set of sample cards, but of course this app only becomes useful if users can actually customize the list of cards they see. This means adding a new view that lists all existing cards and lets the user add a new one, which is all stuff you've seen before. However, there's an interesting catch this time that will require something new to fix, so it's worth working through this.

First we need some state that controls whether our editing screen is visible. So, add this to **ContentView**:

```
@State private var showingEditScreen = false
```

Next we need to add a button to flip that Boolean when tapped, so find the **if differentateWithoutColor || accessibilityEnabled** condition and put this before it:

```
 VStack {  
     HStack {  
         Spacer()  
  
         Button {  
             showingEditScreen = true  
         } label: {  
             Image(systemName: "plus.circle")  
                 .padding()  
                 .background(.black.opacity(0.7))  
                 .clipShape(Circle())  
         }  
     }  
     Spacer()  
 }  
.foregroundColor(.white)
```

```
.font(.largeTitle)
.padding()
```

We're going to design a new **EditCards** view to encode and decode a **Card** array to **UserDefaults**, but before we do that I'd like you to make the **Card** struct conform to **Codable** like this:

```
struct Card: Codable {
```

Now create a new SwiftUI view called “EditCards”. This needs to:

1. Have its own **Card** array.
2. Be wrapped in a **NavigationView** so we can add a Done button to dismiss the view.
3. Have a list showing all existing cards.
4. Add swipe to delete for those cards.
5. Have a section at the top of the list so users can add a new card.
6. Have methods to load and save data from **UserDefaults**.

We've looked at literally all that code previously, so I'm not going to explain it again here. I hope you can stop to appreciate how far this means you have come!

Replace the template **EditCards** struct with this:

```
struct EditCards: View {
    @Environment(\.dismiss) var dismiss
    @State private var cards = [Card]()
    @State private var newPrompt = ""
    @State private var newAnswer = ""

    var body: some View {
        NavigationView {
            List {
                Section("Add new card") {
```

Project 17: Flashzilla

```
        TextField("Prompt", text: $newPrompt)
        TextField("Answer", text: $newAnswer)
        Button("Add card", action: addCard)
    }

Section {
    ForEach(0..<cards.count, id: \.self) { index in
        VStack(alignment: .leading) {
            Text(cards[index].prompt)
                .font(.headline)
            Text(cards[index].answer)
                .foregroundColor(.secondary)
        }
    }
    .onDelete(perform: removeCards)
}
}

.navigationTitle("Edit Cards")
.toolbar {
    Button("Done", action: done)
}
.listStyle(.grouped)
.onAppear(perform: loadData)
}

func done() {
    dismiss()
}

func loadData() {
    if let data = UserDefaults.standard.data(forKey: "Cards") {
```

Adding and deleting cards

```
        if let decoded = try? JSONDecoder().decode([Card].self,
from: data) {
            cards = decoded
        }
    }
}

func saveData() {
    if let data = try? JSONEncoder().encode(cards) {
        UserDefaults.standard.set(data, forKey: "Cards")
    }
}

func addCard() {
    let trimmedPrompt =
newPrompt.trimmingCharacters(in: .whitespaces)
    let trimmedAnswer =
newAnswer.trimmingCharacters(in: .whitespaces)
    guard trimmedPrompt.isEmpty == false &&
trimmedAnswer.isEmpty == false else { return }

    let card = Card(prompt: trimmedPrompt, answer:
trimmedAnswer)
    cards.insert(card, at: 0)
    saveData()
}

func removeCards(at offsets: IndexSet) {
    cards.remove(atOffsets: offsets)
    saveData()
}
}
```

Project 17: Flashzilla

That's almost all of **EditCards** complete, but before we can use it we need to add some more code to **ContentView** so that it shows the sheet on demand and calls **resetCards()** when dismissed.

We've used sheets previously, but there's one extra technique I'd like you to show you: you can attach a function to your sheet, that will automatically be run when the sheet is dismissed. This isn't helpful for times you need to pass back data from the sheet, but here we're just going to call **resetCards()** so it's perfect.

Add this **sheet()** modifier to the end of the outermost **ZStack** in **ContentView**:

```
.sheet(isPresented: $showingEditScreen, onDismiss: resetCards)
{
    EditCards()
}
```

That works, but now that you're gaining more experience in SwiftUI I want to show you an alternative way to get the same result.

When we use the **sheet()** modifier we need to give SwiftUI a function it can run that returns the view to show in the sheet. For us above that's a closure with **EditCards()** inside – that creates and returns a new view, which is what the sheet wants.

When we write **EditCards()**, we're relying on *syntactic sugar* – we're treating our view struct like a function, because Swift silently treats that as a call to the view's initializer. So, in practice we're actually writing **EditCards.init()**, just in a shorter way.

This all matters because rather than creating a closure that calls the **EditCards** initializer, we can actually pass the **EditCards** initializer directly to the sheet, like this:

```
.sheet(isPresented: $showingEditScreen, onDismiss: resetCards,
content: EditCards.init)
```

That means “when you want to read the content for the sheet, call the **EditCards** initializer and it will send you back the view to use.”

Important: This approach only works because **EditCards** has an initializer that accepts no parameters. If you need to pass in specific values you need to use the closure-based approach instead.

Anyway, as well as calling **resetCards()** when the sheet is dismissed, we also want to call it when the view first appears, so add this modifier below the previous one:

```
.onAppear(perform: resetCards)
```

So, when the view is first shown **resetCards()** is called, and when it’s shown after **EditCards** has been dismissed **resetCards()** is also called. This means we can ditch our example **cards** data and instead make it an empty array that gets filled at runtime.

So, change the **cards** property of **ContentView** to this:

```
@State private var cards = [Card]()
```

To finish up with **ContentView** we need to make it load that **cards** property on demand. This starts with the same code we just added in **EditCard**, so put this method into **ContentView** now:

```
func loadData() {
    if let data = UserDefaults.standard.data(forKey: "Cards") {
        if let decoded = try? JSONDecoder().decode([Card].self,
from: data) {
            cards = decoded
        }
    }
}
```

And now we can add a call to **loadData()** in **resetCards()**, so that we refill the **cards** property

Project 17: Flashzilla

with all saved cards when the app launches or when the user edits their cards:

```
func resetCards() {
    timeRemaining = 100
    isActive = true
    loadData()
}
```

Now go ahead and run the app. We've wiped out our default examples, so you'll need to press the + icon to add some of your own.

With that last change, our app is complete – good job!

Flashzilla: Wrap up

This was our another super-sized project, but also another one where we covered some really great techniques like gestures, haptics, Combine, and more. When these features work together we can do remarkable things in our apps, providing an experience to users that is seamless and delightful.

You also saw once again the importance of ensuring accessibility in our apps. It's easy to get carried away with cool gestures and more, but then forget that straying from standard UI also means we need to up our game when it comes to VoiceOver and more. Anyone can make a good idea, but to make a *great* app means you've taken into account the needs of everyone.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are some ways you should try extending this app to make sure you fully understand what's going on.

1. When adding a card, the textfields keep their current text – fix that so that the textfields clear themselves after a card is added.
2. If you drag a card to the right but not far enough to remove it, then release, you see it turn red as it slides back to the center. Why does this happen and how can you fix it? (Tip: think about the way we set **offset** back to 0 immediately, even though the card hasn't animated yet. You might solve this with a ternary within a ternary, but a custom modifier will be cleaner.)
3. For a harder challenge: when the users gets an answer wrong, add that card goes back into

Project 17: Flashzilla

the array so the user can try it again. Doing this successfully means rethinking the **ForEach** loop, because relying on simple integers isn't enough – your cards need to be uniquely identifiable.

Still thirsty for more? Try upgrading our loading and saving code in two ways:

1. Make it use documents JSON rather than **UserDefaults** – this is generally a good idea, so you should get practice with this.
2. Try to find a way to centralize the loading and saving code for the cards. You might need to experiment a little to find something you like!

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Solution to Flashzilla. If you don't already subscribe, you can start a free trial today.

Project 18

Layout and Geometry

Explore the inner workings of SwiftUI's layout system.

Layout and geometry: Introduction

In this technique project we're going to explore how SwiftUI handles layout. Some of these things have been explained a little already, some of them you might have figured out yourself, but many more are things you might just have taken for granted to this point, so I hope a detailed exploration will really shed some light on how SwiftUI works.

Along the way you'll also learn about creating more advanced layout alignments, building special effects using **GeometryReader**, and more – some real power features that I know you'll be keen to deploy in your own apps.

Go ahead and create a new iOS project using the App template, naming it `LayoutAndGeometry`. You'll need an image in your asset catalog in order to follow the chapter on custom alignment guides, but it can be anything you want – it's just a placeholder really.

How layout works in SwiftUI

All SwiftUI layout happens in three simple steps, and understanding these steps is the key to getting great layouts every time. The steps are:

1. A parent view proposes a size for its child.
2. Based on that information, the child then chooses its own size and the parent *must* respect that choice.
3. The parent then positions the child in its coordinate space.

Behind the scenes, SwiftUI performs a fourth step: although it *stores* positions and sizes as floating-point numbers, when it comes to rendering SwiftUI rounds off any pixels to their nearest values so our graphics remain sharp.

Those three rules might seem simple, but they allow us to create hugely complicated layouts where every view decides how and when it resizes without the parent having to get involved.

To demonstrate these rules in action, I'd like you to modify the default SwiftUI template to add a **background()** modifier, like this:

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, World!")  
            .background(.red)  
    }  
}
```

You'll see the background color sits tightly around the text itself – it takes up only enough space to fit the content we provided.

Now, think about this question: *how big is ContentView?* As you can see, the body of **ContentView** – the thing that it renders – is some text with a background color. And so the size of **ContentView** is exactly and always the size of its body, no more and no less. This is

Project 18: Layout and Geometry

called being *layout neutral*: **ContentView** doesn't have any size of its own, and instead happily adjusts to fit whatever size is needed.

Back in project 3 I explained to you that when you apply a modifier to a view we actually get back a new view type called **ModifiedContent**, which stores both our original view and its modifier. This means when we apply a modifier, the actual view that goes into the hierarchy is the modified view, not the original one.

In our simple **background()** example, that means the top-level view inside **ContentView** is the background, and inside *that* is the text. Backgrounds are layout neutral just like **ContentView**, so it will just pass on any layout information as needed – you can end up with a chain of layout information being passed around until a definitive answer comes back.

If we put this into the three-step layout system, we end up with a conversation a bit like this:

- SwiftUI: “Hey, ContentView, you have the whole screen to yourself – how much of it do you need?” (Parent view proposes a size)
- ContentView: “I don’t care; I’m layout neutral. Let me ask my child: hey, background, you have the whole screen to yourself – how much of it do you need?” (Parent view proposes a size)
- Background: “I also don’t care; I’m layout neutral too. Let me ask *my* child: hey, text, you can have the whole screen to yourself – how much of it do you need?” (Parent view proposes a size)
- Text: “Well, I have the letters ‘Hello, World’ in the default font, so I need exactly X pixels width by Y pixels height. I don’t need the whole screen, just that.” (Child chooses its size.)
- Background: “Got it. Hey, ContentView: I need X by Y pixels, please.”
- ContentView: “Right on. Hey, SwiftUI: I need X by Y pixels.”
- SwiftUI: “Nice. Well, that leaves lots of space, so I’m going to put you at your size in the center.” (Parent positions the child in its coordinate space.)

So, when we say **Text("Hello, World!").background(.red)**, the text view becomes a child of its background. SwiftUI effectively works its way from bottom to top when it comes to a view and its modifiers.

Now consider this layout:

```
Text("Hello, World!")  
    .padding(20)  
    .background(.red)
```

This time the conversation is more complicated: **padding()** no longer offers all its space to its child, because it needs to subtract 20 points from each side to make sure there's enough space for the padding. Then, when the answer comes back from the text view, **padding()** adds 20 points on each side to pad it out, as requested.

So, it's more like this:

- SwiftUI: You can have the whole screen, how much of it do you need, ContentView?
- ContentView: You can have the whole screen, how much of it do you need, background?
- Background: You can have the whole screen, how much of it do you need, padding?
- Padding: You can have the whole screen minus 20 points on each side, how much of it do you need, text?
- Text: I need X by Y.
- Padding: I need X by Y plus 20 points on each side.
- Background: I need X by Y plus 20 points on each side.
- ContentView: I need X by Y plus 20 points on each side.
- SwiftUI: OK; I'll center you.

If you remember, the order of our modifiers matters. That is, this code:

```
Text("Hello, World!")  
    .padding()  
    .background(.red)
```

And this code:

```
Text("Hello, World!")
```

Project 18: Layout and Geometry

```
.background(.red)  
.padding()
```

Yield two different results. Hopefully now you can see why: **background()** is layout neutral, so it determines how much space it needs by asking its child how much space *it* needs and using that same value. If the child of **background()** is the text view then the background will fit snugly around the text, but if the child is **padding()** then it receive back the adjusted values that including the padding amount.

There are two interesting side effects that come as a result of these layout rules.

First, if your view hierarchy is wholly layout neutral, then it will automatically take up all available space. For example, shapes and colors are layout neutral, so if your view contains a color and nothing else it will automatically fill the screen like this:

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

Remember, **Color.red** is a view in its own right, but because it is layout neutral it can be drawn at any size. When we used it inside **background()** the abridged layout conversation worked like this:

- Background: Hey text, you can have the whole screen – how much of that do you want?
- Text: I need X by Y points; I don't need the rest.
- Background: OK. Hey, Color.red: you can have X by Y points – how much of that do you want?
- Color.red: I don't care; I'm layout neutral, so X by Y points sounds good to me.

The second interesting side effect is one we faced earlier: if we use **frame()** on an image that isn't resizable, we get a larger frame without the image inside changing size. This might have been confusing before, but it makes absolute sense once you think about the frame as being the parent of the image:

- **ContentView** offers the frame the whole screen.
- The frame reports back that it wants 300x300.
- The frame then asks the image inside it what size it wants.
- The image, not being resizable, reports back a fixed size of 64x64 (for example).
- The frame then positions that image in the center of itself.

When you listen to Apple's own SwiftUI engineers talk about modifiers, you'll hear them refer them to as views – “the frame view”, “the background view”, and so on. I think that's a great mental model to help understand exactly what's going on: applying modifiers creates new views rather than just modifying existing views in-place.

Alignment and alignment guides

SwiftUI gives us a number of valuable ways of controlling the way views are aligned, and I want to walk you through each of them so you can see them in action.

The simplest alignment option is to use the **alignment** parameter of a **frame()** modifier. Remember, a text view always uses the exact width and height required to show its text, but when we place a frame around it that can be any size. As the parent doesn't have a say in the final size of the child, code like this will create a 300x300 frame with a smaller text view centered inside it:

```
Text("Live long and prosper")
    .frame(width: 300, height: 300)
```

If you don't want the text to be centered, use the **alignment** parameter of the **frame()**. For example, this code places the view in the top-left corner when running on a left-to-right environment

```
    .frame(width: 300, height: 300, alignment: .topLeading)
```

You can then use **offset(x:y:)** to move the text around inside that frame.

The next option up is to use the **alignment** parameter of a stack. For example, here are four text views of varying sizes arranged in a **HStack**:

```
HStack {
    Text("Live")
        .font(.caption)
    Text("long")
    Text("and")
        .font(.title)
    Text("prosper")
        .font(.largeTitle)
}
```

We haven't specified an alignment there, so they will be centered by default. That doesn't look great, so you might think to align them all to one edge to get a neater line, like this:

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

However, that also looks bad: because each of the text views has a different size, they also have a different *baseline* – that's the name for where letters such as “abcde” sit on a line, which excludes letters that go below the line such as “gjpy”. As a result, the bottom of the small text sits lower than the bottom of the bigger text.

Fortunately, SwiftUI has two special alignments that align text on the baseline of either the first child or the last child. This will cause all views in a stack to be aligned on a single unified baseline, regardless of their font:

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

Moving on, for more fine-grained control we can customize what “alignment” means for each individual view. To get a really good idea of how this works we're going to start with this code:

```
struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading) {
            Text("Hello, world!")
            Text("This is a longer line of text")
        }
        .background(.red)
        .frame(width: 400, height: 400)
        .background(.blue)
    }
}
```

Project 18: Layout and Geometry

When that runs you'll see the **VStack** sits tightly around its two text views with a red background. The two text views have different lengths, but because we used the **.leading** alignment they will both be aligned to their left edge in a left-to-right environment. Outside of that there's a larger frame that has a blue background. Because the frame is larger than the **VStack**, the **VStack** is centered in the middle.

Now, when the **VStack** comes to aligning each of those text views, it asks them to provide their leading edge. By default this is obvious: it uses either the left or right edge of the view, depending on the system language. But what if we wanted to change that – what if we wanted to make one view have a custom alignment?

SwiftUI provides us with the **alignmentGuide()** modifier for just this purpose. This takes two parameters: the guide we want to change, and a closure that returns a new alignment. The closure is given a **ViewDimensions** object that contains the width and height of its view, along with the ability to read its various edges.

By default, the **.leading** alignment guide for a view is its leading alignment guide – I know that sounds obvious, but its effectively equivalent to this:

```
 VStack(alignment: .leading) {
    Text("Hello, world!")
        .alignmentGuide(.leading) { d in d[.leading] }
    Text("This is a longer line of text")
}
```

We could rewrite that alignment guide to use the view's *trailing* edge for its leading alignment guide, like this:

```
 VStack(alignment: .leading) {
    Text("Hello, world!")
        .alignmentGuide(.leading) { d in d[.trailing] }
    Text("This is a longer line of text")
}
```

And now you'll see why I added colors: the first text view will move to the left so that its right edge sits directly above the left edge of the view below, the **VStack** will expand to contain it, and the whole thing will still be centered within the blue frame.

This result is *different* from using the **offset()** modifier: if you offset a text its original dimensions don't actually change, even though the resulting view is rendered in a different location. If we had offset the first text view rather than changing its alignment guide, the **VStack** wouldn't expand to contain it.

Although the alignment guide closure is passed your view's dimensions, you don't need to use them if you don't want to – you can send back a hard-coded number, or create some other calculation. For example, this creates a tiered effect for 10 text views by multiplying their position by -10:

```
var body: some View {
    VStack(alignment: .leading) {
        ForEach(0..<10) { position in
            Text("Number \(position)")
                .alignmentGuide(.leading) { _ in CGFloat(position) * -10 }
        }
        .background(.red)
        .frame(width: 400, height: 400)
        .background(.blue)
    }
}
```

For *complete* control over your alignment guides you need to create a custom alignment guide. And I think *that* deserves a mini chapter all of its own...

How to create a custom alignment guide

SwiftUI gives us alignment guides for the various edges of our views (**.leading**, **trailing**, **top**, and so on) plus **.center** and two baseline options to help with text alignment. However none of these work well when you’re working with views that are split across disparate views – if you have to make two views aligned the same when they are in entirely different parts of your user interface.

To fix this, SwiftUI lets us create custom alignment guides, and use those guides in views across our UI. It doesn’t matter what comes before or after these views; they will still line up.

For example, here’s a layout that shows my Twitter account name and my profile picture on the left, and on the right shows “Full name:” plus “Paul Hudson” in a large font:

```
struct ContentView: View {
    var body: some View {
        HStack {
            VStack {
                Text("@twostraws")
                Image("paul-hudson")
                    .resizable()
                    .frame(width: 64, height: 64)
            }
            VStack {
                Text("Full name:")
                Text("PAUL HUDSON")
                    .font(.largeTitle)
            }
        }
    }
}
```

}

If you want “@twostraws” and “Paul Hudson” to be vertically aligned together, you’ll have a hard time right now. The horizontal stack contains two vertical stacks inside it, so there’s no built-in way to get the alignment you want – things like **HStack(alignment: .top)** just won’t come close.

To fix this we need to define a custom layout guide. This should be an extension on either **VerticalAlignment** or **HorizontalAlignment**, and be a custom type that conforms to the **AlignmentID** protocol.

When I say “custom type” you might be thinking of a struct, but it’s actually a good idea to implement this as an enum instead as I’ll explain shortly. The **AlignmentID** protocol has only one requirement, which is that the conforming type must provide a static **defaultValue(in:)** method that accepts a **ViewDimensions** object and returns a **CGFloat** specifying how a view should be aligned if it doesn’t have an **alignmentGuide()** modifier. You’ll be given the existing **ViewDimensions** object for the view, so you can either pick one of those for your default or use a hard-coded value.

Let’s write out the code so you can see how it looks:

```
extension VerticalAlignment {
    struct MidAccountAndName: AlignmentID {
        static func defaultValue(in d: ViewDimensions) -> CGFloat {
            d[.top]
        }
    }

    static let midAccountAndName =
        VerticalAlignment(MidAccountAndName.self)
}
```

You can see I’ve used the **.top** view dimension by default, and I’ve also created a static

Project 18: Layout and Geometry

constant called **midAccountAndName** to make the custom alignment easier to use.

Now, I mentioned that using an enum is preferable to a struct, and here's why: we just created a new struct called **MidAccountAndName**, which means we could (if we wanted) create an instance of that struct even though doing so doesn't make sense because it doesn't have any functionality. If you replace **struct MidAccountAndName** with **enum MidAccountAndName** then you *can't* make an instance of it any more – it becomes clearer that this thing exists only to house some functionality.

Regardless of whether you choose an enum or a struct, its usage stays the same: set it as the alignment for your stack, then use **alignmentGuide()** to activate it on any views you want to align together. This is only a *guide*: it helps you align views along a single line, but doesn't say *how* they should be aligned. This means you still need to provide the closure to **alignmentGuide()** that positions the views along that guide as you want.

For example, we could update our Twitter code to use **.midAccountAndName**, then tell the account and name to use their center position for the guide. To be clear, that means “align these two views so their centers are both on the **.midAccountAndName** guide”.

Here's how that looks in code:

```
HStack(alignment: .midAccountAndName) {
    VStack {
        Text("@twostraws")
            .alignmentGuide(.midAccountAndName) { d in
                d[VerticalAlignment.center]
            }
        Image("paul-hudson")
            .resizable()
            .frame(width: 64, height: 64)
    }

    VStack {
        Text("Full name:")
    }
}
```

How to create a custom alignment guide

```
Text("PAUL HUDSON")
    .alignmentGuide(.midAccountAndName) { d in
d[VerticalAlignment.center] }
    .font(.largeTitle)
}
}
```

That will make sure they are vertically aligned regardless of what comes before or after. I suggest you try adding some more text views before and after our examples – SwiftUI will reposition everything to make sure the two we aligned stay that way.

Absolute positioning for SwiftUI views

SwiftUI gives us two ways of positioning views: absolute positions using **position()**, and relative positions using **offset()**. They might seem similar, but once you understand how SwiftUI places views inside frames the underlying differences between **position()** and **offset()** become clearer.

A simple SwiftUI view looks like this:

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello, world!")  
    }  
}
```

SwiftUI offers the full available space to **ContentView**, which in turn passes it on to the text view. The text view automatically uses only as much as space as its text needs, so it passes that back up to **ContentView**, which is always and exactly the same size as its **body** (so it directly fits around the text). As a result, SwiftUI centers **ContentView** in the available space, which from a user's perspective is what places the text in the center.

If you want to absolutely position a SwiftUI view you should use the **position()** modifier like this:

```
Text("Hello, world!")  
    .position(x: 100, y: 100)
```

That will position the text view at x:100 y:100 within its parent. Now, to *really* see what's happening here I want you to add a background color:

```
Text("Hello, world!")  
    .background(.red)
```

```
.position(x: 100, y: 100)
```

You'll see the text has a red background tightly fitted around it. Now try moving the **background()** modifier below the **position()** modifier, like this:

```
Text("Hello, world!")
    .position(x: 100, y: 100)
    .background(.red)
```

Now you'll see the text is in the same location, but the whole safe area is colored red.

To understand what's happening here you need to remember the three step layout process of SwiftUI:

1. A parent view proposes a size for its child.
2. Based on that information, the child then chooses its own size and the parent *must* respect that choice.
3. The parent then positions the child in its coordinate space.

So, the parent is responsible for positioning the child, not the child. This causes a problem, because we've just told our text view to be at an exact position – how can SwiftUI resolve this?

The answer to this is also why our **background()** color made the whole safe area red: when we use **position()** we get back a new view that takes up all available space, so it can position its child (the text) at the correct location.

When we use text, position, then background the position will take up all available space so it can position its text correctly, then the background will use that size for itself. When we use text, background, then position, the background will use the text size for its size, then the position will take up all available space and place the background in the correct location.

When discussing the **offset()** modifier earlier, I said “if you offset some text its original dimensions don't actually change, even though the resulting view is rendered in a different

Project 18: Layout and Geometry

location.” With that in mind, try running this code:

```
var body: some View {
    Text("Hello, world!")
        .offset(x: 100, y: 100)
        .background(.red)
}
```

You’ll see the text appears in one place and the background in another. I’m going to explain why that is, but first I want you to think about it yourself because if you understand *that* then you really understand how SwiftUI’s layout system works.

When we use the **offset()** modifier, we’re changing the location where a view should be rendered without actually changing its underlying geometry. This means when we apply **background()** afterwards it uses the original position of the text, *not* its offset. If you move the modifier order so that **background()** comes before **offset()** then things work more like you might have expected, showing once again that modifier order matters.

Understanding frames and coordinates inside GeometryReader

SwiftUI's **GeometryReader** allows us to use its size and coordinates to determine a child view's layout, and it's the key to creating some of the most remarkable effects in SwiftUI.

You should always keep in mind SwiftUI's three-step layout system when working with **GeometryReader**: parent proposes a size for the child, the child uses that to determine its own size, and parent uses that to position the child appropriately.

In its most basic usage, what **GeometryReader** does is let us read the size that was proposed by the parent, then use that to manipulate our view. For example, we could use **GeometryReader** to make a text view have 90% of all available width regardless of its content:

```
struct ContentView: View {
    var body: some View {
        GeometryReader { geo in
            Text("Hello, World!")
                .frame(width: geo.size.width * 0.9)
                .background(.red)
        }
    }
}
```

That **geo** parameter that comes in is a **GeometryProxy**, and it contains the proposed size, any safe area insets that have been applied, plus a method for reading frame values that we'll look at in a moment.

GeometryReader has an interesting side effect that might catch you out at first: the view that

Project 18: Layout and Geometry

gets returned has a flexible preferred size, which means it will expand to take up more space as needed. You can see this in action if you place the **GeometryReader** into a **VStack** then put some more text below it, like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            GeometryReader { geo in
                Text("Hello, World!")
                    .frame(width: geo.size.width * 0.9, height: 40)
                    .background(.red)
            }

            Text("More text")
                .background(.blue)
        }
    }
}
```

You'll see "More text" gets pushed right to the bottom of the screen, because the **GeometryReader** takes up all remaining space. To see it in action, add **background(.green)** as a modifier to the **GeometryReader** and you'll see just how big it is. Note: This is a *preferred* size, not an *absolute* size, which means it's still flexible depending on its parent.

When it comes to reading the frame of a view, **GeometryProxy** provides a **frame(in:)** method rather than simple properties. This is because the concept of a "frame" includes X and Y coordinates, which don't make any sense in isolation – do you want the view's absolute X and Y coordinates, or their X and Y coordinates compared to their parent?

SwiftUI calls these options *coordinate spaces*, and those two in particular are called the global space (measuring our view's frame relative to the whole screen), and the local space (measuring our view's frame relative to its parent). We can also create custom coordinate spaces by attaching the **coordinateSpace()** modifier to a view – any children of that can then

Understanding frames and coordinates inside GeometryReader

read its frame relative to that coordinate space.

To demonstrate how coordinate spaces work, we could create some example views in various stacks, attach a custom coordinate space to the outermost view, then add an **onTapGesture** to one of the views inside it so it can print out the frame globally, locally, and using the custom coordinate space.

Try this code:

```
struct OuterView: View {
    var body: some View {
        VStack {
            Text("Top")
            InnerView()
                .background(.green)
            Text("Bottom")
        }
    }
}

struct InnerView: View {
    var body: some View {
        HStack {
            Text("Left")
            GeometryReader { geo in
                Text("Center")
                    .background(.blue)
                    .onTapGesture {
                        print("Global center: \
(geo.frame(in: .global).midX) x \
(geo.frame(in: .global).midY)")
                        print("Custom center: \
(geo.frame(in: .named("Custom")).midX) x \
(geo.frame(in: .named("Custom")).midY)")
                    }
            }
        }
    }
}
```

Project 18: Layout and Geometry

```
(geo.frame(in: .named("Custom")).midY"))
    print("Local center: \(geo.frame(in: .local).midX)
x \(geo.frame(in: .local).midY)")
}
}
.background(.orange)
Text("Right")
}
}
}

struct ContentView: View {
var body: some View {
OuterView()
.background(.red)
.coordinateSpace(name: "Custom")
}
}
```

The output you get when that code runs depends on the device you're using, but here's what I got:

- Global center: 189.83 x 430.60
- Custom center: 189.83 x 383.60
- Local center: 152.17 x 350.96

Those sizes are mostly different, so hopefully you can see the full range of how these frame work:

- A global center X of 189 means that the center of the geometry reader is 189 points from the left edge of the screen.
- A global center Y of 430 means the center of the text view is 430 points from the top edge of the screen. This isn't dead in the center of the screen because there is more safe area at

Understanding frames and coordinates inside GeometryReader

the top than the bottom.

- A custom center X of 189 means the center of the text view is 189 points from the left edge of whichever view owns the “Custom” coordinate space, which in our case is **OuterView** because we attach it in **ContentView**. This number matches the global position because **OuterView** runs edge to edge horizontally.
- A custom center Y of 383 means the center of the text view is 383 points from the top edge of **OuterView**. This value is smaller than the global center Y because **OuterView** doesn’t extend into the safe area.
- A local center X of 152 means the center of the text view is 152 points from the left edge of its direct container, which in this case is the **GeometryReader**.
- A local center Y of 350 means the center of the text view is 350 points from the top edge of its direct container, which again is the **GeometryReader**.

Which coordinate space you want to use depends on what question you want to answer:

- Want to know where this view is on the screen? Use the global space.
- Want to know where this view is relative to its parent? Use the local space.
- What to know where this view is relative to some other view? Use a custom space.

ScrollView effects using GeometryReader

When we use the **frame(in:)** method of a **GeometryProxy**, SwiftUI will calculate the view's current position in the coordinate space we ask for. However, as the view *moves* those values will change, and SwiftUI will automatically make sure **GeometryReader** stays updated.

Previously we used **DragGesture** to store a width and height as an **@State** property, because it allowed us to adjust other properties based on the drag amount to create neat effects.

However, with **GeometryReader** we can grab values from a view's environment dynamically, feeding in its absolute or relative position into various modifiers. Even better, you can nest geometry readers if needed, so that one can read the geometry for a higher-up view and the other can read the geometry for something further down the tree.

To try some effects with **GeometryReader**, we could create a spinning helix effect by creating 50 text views in a vertical scroll view, each of which an infinite maximum width so they take up all the screen space, then apply a 3D rotation effect based on their own position.

Start by making a basic **ScrollView** of text views with varying background colors:

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

    var body: some View {
        ScrollView {
            ForEach(0..<50) { index in
                GeometryReader { geo in
                    Text("Row #\((index))")
                        .font(.title)
                        .frame(maxWidth: .infinity)
                        .background(colors[index % 7])
                }
            }
        }
    }
}
```

ScrollView effects using GeometryReader

```
        }
        .frame(height: 40)
    }
}
}
}
```

To apply a helix-style spinning effect, place this **rotation3DEffect()** directly below the **background()** modifier:

```
.rotation3DEffect(.degrees(geo.frame(in: .global).minY / 5),
axis: (x: 0, y: 1, z: 0))
```

When you run that back you'll see that text views at the bottom of the screen are flipped, those at the center are rotated about 90 degrees, and those at the very top are normal. More importantly, as you scroll around they all rotate as you move in the scroll view.

That's a neat effect, but it's also problematic because the views only reach their natural orientation when they are at the very top – it's really hard to read. To fix this, we can apply a more complex **rotation3DEffect()** that subtracts half the height of the main view, but that means using a *second GeometryReader* to get the size of the main view:

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

    var body: some View {
        GeometryReader { fullView in
            ScrollView {
                ForEach(0..<50) { index in
                    GeometryReader { geo in
                        Text("Row #\((index)")
                            .font(.title)
                    }
                }
            }
        }
    }
}
```

Project 18: Layout and Geometry

```
        .frame(maxWidth: .infinity)
        .background(colors[index % 7])
        .rotation3DEffect(.degrees(geo.frame(in: .global)
.minY - fullView.size.height / 2) / 5, axis: (x: 0, y: 1, z:
0))
    }
    .frame(height: 40)
}
}
}
}
```

With that in place, the views will reach a natural orientation nearer the center of the screen, which will look better.

We can use a similar technique to create CoverFlow-style scrolling rectangles:

```
struct ContentView: View {
var body: some View {
    ScrollView(.horizontal, showsIndicators: false) {
        HStack(spacing: 0) {
            ForEach(1..<20) { num in
                GeometryReader { geo in
                    Text("Number \\" + String(num) + "\"")
                        .font(.largeTitle)
                        .padding()
                        .background(.red)
                        .rotation3DEffect(.degrees(-
geo.frame(in: .global).minX) / 8, axis: (x: 0, y: 1, z: 0))
                        .frame(width: 200, height: 200)
                }
                .frame(width: 200, height: 200)
            }
        }
    }
}
```

ScrollView effects using GeometryReader

```
        }  
    }  
}  
}  
}
```

There are so many interesting and creative ways to make special effects with **GeometryReader** – I hope you can take the time to experiment!

Layout and geometry: Wrap up

I hope this smaller technique project proved a welcome break after our long app projects, but I hope even more that you're really starting to have a good mental model of how SwiftUI's layout system works. That three step layout system might *sound* simple, but it takes time to fully understand the ramifications it has.

As for **GeometryReader**, it's one of those things you can get by perfectly fine without even thinking about, and that's fine. But when you want to add a little pizazz to your designs – when you want to really bring something to life as the user interacts with it – **GeometryReader** is a fast and flexible fix that offers a huge amount of power in only a handful of lines of code.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three challenges for you to complete to experiment with your knowledge of **GeometryReader**.

First, put your **ContentView** back to the spinning color rows example we had:

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

    var body: some View {
        GeometryReader { fullView in
            ScrollView(.vertical) {
```

```
ForEach(0..<50) { index in
    GeometryReader { geo in
        Text("Row #\u2028(index)")
            .font(.title)
            .frame(maxWidth: .infinity)
            .background(colors[index % 7])
            .rotation3DEffect(.degrees(geo.frame(in: .global)
.minY - fullView.size.height / 2) / 5, axis: (x: 0, y: 1, z:
0))
    }
    .frame(height: 40)
}
}
```

With that done:

1. Make views near the top of the scroll view fade out to 0 opacity – I would suggest starting at about 200 points from the top.
 2. Make views adjust their scale depending on their vertical position, with views near the bottom being large and views near the top being small. I would suggest going no smaller than 50% of the regular size.
 3. For a real challenge make the views change color as you scroll. For the best effect, you should create colors using the **Color(hue:saturation:brightness:)** initializer, feeding in varying values for the hue.

Each of those will require a little trial and error from you to find values that work well. Regardless, you should use **max()** to handle the scaling so that views don't go smaller than half their size, and use **min()** with the hue so that hue values don't go beyond 1.0.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Project 18: Layout and Geometry

Solution to Layout and Geometry. If you don't already subscribe, you can start a free trial today.

Project 19

SnowSeeker

Build an app for ski resorts that works great on iPad.

SnowSeeker: Introduction

In this project we're going to create SnowSeeker: an app to let users browse ski resorts around the world, to help them find one suitable for their next holiday.

This will be the first app where we specifically aim to make something that works great on iPad by showing two views side by side, but you'll also get deep into solving problematic layouts, learn a new way to show sheets and alerts, and more.

As always we have some techniques to cover before getting into the main project, so please create a new iOS project using the App template, calling it SnowSeeker.

Let's go!

Working with two side by side views in SwiftUI

You might not have realized it, but one of the smartest, simplest ways that our apps adapt to varying screen sizes is actually baked right in to **NavigationView**.

You're already familiar with the basic usage of **NavigationView**, which allows us to create views like this one:

```
struct ContentView: View {  
    var body: some View {  
        NavigationView {  
            Text("Hello, world!")  
                .navigationTitle("Primary")  
        }  
    }  
}
```

However, what you see when that runs depends on several factors. If you're using an iPhone in portrait then you'll see the layout you've come to expect: a large "Primary" title at the top, then a small "Hello, world!" centered in the space below.

If you rotate that same phone to landscape, then you'll see one of two things: on the majority of iPhones the navigation title will shrink away to small text so that it takes up less space, but on max-sized iPhones, such as the iPhone 13 Pro Max, you'll see that our title becomes a blue button in the top-left corner, leaving the whole rest of the screen clear. *Tapping* that button makes the "Hello, world!" text slide in from the leading edge, where you'll also see the "Primary" title at the top.

On iPad things get even clever, because the system will select from three different layouts depending on the device's size and the available screen space. For example, if we were using a 12.9-inch iPad Pro in landscape, then:

Project 19: SnowSeeker

1. If our app has the whole screen to itself, you'll see the "Hello, world!" view visible on the left, with nothing on the right.
2. If the app has very little space, it will look just like a long iPhone in portrait.
3. For other sizes, you're likely to see the "Primary" button visible, which causes the "Hello, world!" text to slide in when pressed.

What you're seeing here is called *adaptive layout*, and it's used in many of Apple's apps such as Notes, Mail, and more. It allows iOS to make best use of available screen space without us needing to get involved.

What's actually happening here is that iOS is giving us a primary/secondary layout: a primary view to act as navigation, such as selecting from a list of books we've read or a list of Apollo missions, and a secondary view to act as further information, such as more details about a book or Apollo mission selected in the primary view.

In our trivial code example, SwiftUI interprets the single view inside our **NavigationView** as being the primary view in this primary/secondary layout. However, if we *do* provide two views then we get some really useful behavior out of the box. Try changing your view to this:

```
NavigationView {  
    Text("Hello, world!")  
    .navigationTitle("Primary")  
  
    Text("Secondary")  
}
```

When you launch the app what you see once again depends on your device and orientation, but on Max-sized phones and iPads you'll see "Secondary", with the Primary toolbar button bringing up the "Hello, world!" view.

SwiftUI automatically links the primary and secondary views, which means if you have a **NavigationLink** in the primary view it will automatically load its content in the secondary view:

```
NavigationView {  
    NavigationLink {  
        Text("New secondary")  
    } label: {  
        Text("Hello, World!")  
    }  
    .navigationTitle("Primary")  
  
    Text("Secondary")  
}
```

However, right now at least, all this magic has a few drawbacks that I hope will be fixed in a future SwiftUI update:

1. Detail views *always* get a navigation bar whether you want it or not, so you need to use **navigationBarHidden(true)** to hide it.
2. There's no way of making the primary view stay visible even when there is more than enough space.
3. You can't make the primary view shown in landscape by default; SwiftUI always chooses the detail.

Tip: You can even add a third view to **NavigationView**, which lets you create a sidebar. You'll see these in apps such as Notes, where you can navigate up from the list of notes to browse note folders. So, navigation links in the first view control the second view, and navigation links in the second view control the third view – it's an extra level of organization for times when you need it.

Using alert() and sheet() with optionals

SwiftUI has two ways of creating alerts and sheets, and so far we've mostly only used one: a binding to a Boolean that shows the alert or sheet when the Boolean becomes true.

The *second* option allows us to bind an optional to the alert or sheet, and we used it briefly when presenting map pins. If you remember, the key is that we use an optional **Identifiable** object as the condition for showing the sheet, and the closure hands you the non-optional value that was used for the condition, so you can use it safely.

To demonstrate this, we could create a trivial **User** struct that conforms to the **Identifiable** protocol:

```
struct User: Identifiable {
    var id = "Taylor Swift"
}
```

We could then create a property inside **ContentView** that tracks which user is selected, set to **nil** by default:

```
@State private var selectedUser: User? = nil
```

Now we can change the **body** of **ContentView** so that it sets **selectedUser** to a value when its text view is tapped, then displays a sheet when **selectedUser** is given a value:

```
Text("Hello, World!")
    .onTapGesture {
        selectedUser = User()
    }
    .sheet(item: $selectedUser) { user in
        Text(user.id)
    }
}
```

Using alert() and sheet() with optionals

With that simple code, whenever you tap “Hello, World!” an alert saying “Taylor Swift” appears. As soon as the alert is dismissed, SwiftUI sets **selectedUser** back to **nil**.

This might seem like a simple piece of functionality, but it’s simpler and safer than the alternative. If we were to rewrite the above code using the old **.alert(isPresented:)** modifier it would look like this:

```
struct ContentView: View {
    @State private var selectedUser: User? = nil
    @State private var isShowingUser = false

    var body: some View {
        Text("Hello, World!")
            .onTapGesture {
                selectedUser = User()
                isShowingUser = true
            }
            .sheet(isPresented: $isShowingUser) {
                Text(selectedUser!.id)
            }
    }
}
```

That’s another property, another value to set in the **onTapGesture()**, and a force unwrap in the **alert()** modifier – if you can avoid those things you should.

Alerts have similar functionality, although you need to pass both the Boolean and optional **Identifiable** value at the same time. This allows you to show the alert when needed, but also benefit from the same optional unwrapping behavior we have with sheets:

```
.alert("Welcome", isPresented: $isShowingUser, presenting:
selectedUser) { user in
```

Project 19: SnowSeeker

```
    Button(user.id) { }  
}
```

With that covered, you now know practically all there is to know about sheets and alerts, but there is one last thing I want to sneak in to round out your knowledge.

You see, so far we've written lots of alerts like this one:

```
.alert("Welcome", isPresented: $isShowingUser) {  
    Button("OK") { }  
}
```

That OK button works because all actions dismiss the alert they belong to when they are tapped, and we've been using this approach so far because it gives you practice creating alerts and buttons.

However, I want to show you a neat shortcut. Try this code out:

```
.alert("Welcome", isPresented: $isShowingUser) { }
```

When that runs you'll see something interesting: *exactly the same result as before*, despite not having a dedicated OK button. SwiftUI spots that we don't have any actions in the alert, so it adds a default one for us that has the title "OK" and will dismiss the alert when tapped.

Obviously this doesn't work in situations where you need other buttons alongside OK, but for simple alerts it's perfect.

Using groups as transparent layout containers

SwiftUI's **Group** view is commonly used to work around the 10-child view limit, but it also has another important behavior: it acts as a transparent layout container. This means the group doesn't actually affect our layout at all, but still gives us the ability to add SwiftUI modifiers as needed, or send back multiple views without using **@ViewBuilder**.

For example, this **UserView** has a **Group** containing three text views:

```
struct UserView: View {
    var body: some View {
        Group {
            Text("Name: Paul")
            Text("Country: England")
            Text("Pets: Luna and Arya")
        }
        .font(.title)
    }
}
```

That group contains no layout information, so we don't know whether the three text fields will be stacked vertically, horizontally, or by depth. This is where the transparent layout behavior of **Group** becomes important: whatever parent places a **UserView** gets to decide how its text views get arranged.

For example, we could create a **ContentView** like this:

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

    var body: some View {
        if layoutVertically {
            UserView()
        } else {
            UserView()
        }
    }
}
```

Project 19: SnowSeeker

```
Group {
    if layoutVertically {
        VStack {
            UserView()
        }
    } else {
        HStack {
            UserView()
        }
    }
}
.onTapGesture {
    layoutVertically.toggle()
}
```

That flips between vertical and horizontal layout every time the group is tapped, and again you see that using **Group** lets us attach the tap gesture to everything at once.

You might wonder how often you need to have alternative layouts like this, but the answer might surprise you: it's really common! You see, this is exactly what you want to happen when trying to write code that works across multiple device sizes – if we want layout to happen vertically when horizontal space is constrained, but horizontally otherwise. Apple provides a very simple solution called *size classes*, which is a thoroughly vague way of telling us how much space we have for our views.

When I say “thoroughly vague” I mean it: we have only two size classes horizontally and vertically, called “compact” and “regular”. That’s it – that covers all screen sizes from the largest iPad Pro in landscape down to the smallest iPhone in portrait. That doesn’t mean it’s useless – far from it! – just that it only lets us reason about our user interfaces in the broadest terms.

Using groups as transparent layout containers

To demonstrate size classes in action, we could create a view that has a property to track the current size class so we can switch between **VStack** and **HStack** automatically:

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

    var body: some View {
        if sizeClass == .compact {
            VStack {
                UserView()
            }
        } else {
            HStack {
                UserView()
            }
        }
    }
}
```

Tip: In situations like this, where you have only one view inside a stack and it doesn't take any parameters, you can pass the view's initializer directly to the **VStack** to make your code shorter:

```
if sizeClass == .compact {
    VStack(content: UserView.init)
} else {
    HStack(content: UserView.init)
}
```

I know short code isn't everything, but this technique is pleasingly concise when you're using this approach to grouped view layout.

What you see when that code runs depends on the device you're using. For example, an iPhone

Project 19: SnowSeeker

iPhone 13 Pro will have a compact horizontal size class in both portrait and landscape, whereas an iPhone 13 Pro Max will have a *regular* horizontal size class when in landscape.

Regardless of whether we're toggling our layout using size classes or tap gestures, the point is that **UserView** just doesn't care – its **Group** simply groups the text views together without affecting their layout at all, so the layout arrangement **UserView** is given depends entirely on how it's used.

Making a SwiftUI view searchable

iOS can add a search bar to our views using the `searchable()` modifier, and we can bind a string property to it to filter our data as the user types.

To see how this works, try this simple example:

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

    var body: some View {
        NavigationView {
            Text("Searching for \(searchText)")
                .searchable(text: $searchText, prompt: "Look for
something")
                .navigationTitle("Searching")
        }
    }
}
```

Important: You need to make sure your view is inside a `NavigationView`, otherwise iOS won't have anywhere to put the search box.

When you run that code example, you should see a search bar you can type into, and whatever you type will be shown in the view below.

In practice, `searchable()` is best used with some kind of data filtering. Remember, SwiftUI will reinvoke your body property when an `@State` property changes, so you could use a computed property to handle the actual filtering:

```
struct ContentView: View {
    @State private var searchText = ""
    let allNames = ["Subh", "Vina", "Melvin", "Stefanie"]
```

Project 19: SnowSeeker

```
var body: some View {
    NavigationView {
        List(filteredNames, id: \.self) { name in
            Text(name)
        }
        .searchable(text: $searchText, prompt: "Look for
something")
        .navigationTitle("Searching")
    }
}

var filteredNames: [String] {
    if searchText.isEmpty {
        return allNames
    } else {
        return allNames.filter { $0.contains(searchText) }
    }
}
```

When you run that, iOS will automatically hide the search bar at the very top of the list – you’ll need to pull down gently to reveal it, which matches the way other iOS apps work. iOS doesn’t *require* that we make our lists searchable, but it really makes a huge difference to users!

Tip: Rather than using `contains()` here, you probably want to use another method with a much longer name: `localizedCaseInsensitiveContains()`. That lets us check any part of the search strings, without worry about uppercase or lowercase letters.

Building a primary list of items

In this app we’re going to display two views side by side, just like Apple’s Mail and Notes apps. In SwiftUI this is done by placing two views into a **NavigationView**, then using a **NavLink** in the primary view to control what’s visible in the secondary view.

So, we’re going to start off our project by building the primary view for our app, which will show a list of all ski resorts, along with which country they are from and how many ski runs it has – how many pistes you can ski down, sometimes called “trails” or just “slopes”.

I’ve provided some assets for this project in the GitHub repository for this book, so if you haven’t already downloaded them please do so now. You should drag `resorts.json` into your project navigator, then copy all the pictures into your asset catalog. You might notice that I’ve included 2x and 3x images for the countries, but only 2x pictures for the resorts. This is intentional: those flags are going to be used for both retina and Super Retina devices, but the resort pictures are designed to fill all the space on an iPad Pro – they are more than big enough for a Super Retina iPhone even at 2x resolution.

To get our list up and running quickly, we need to define a simple **Resort** struct that can be loaded from our JSON. That means it needs to conform to **Codable**, but to make it easier to use in SwiftUI we’ll also make it conform to **Identifiable**. The actual data itself is mostly just strings and integers, but there’s also a string array called **facilities** that describe what else there is on the resort – I should add that this data is mostly fictional, so don’t try to use it in a real app!

Create a new Swift file called `Resort.swift`, then give it this code:

```
struct Resort: Codable, Identifiable {
    let id: String
    let name: String
    let country: String
    let description: String
    let imageCredit: String
```

Project 19: SnowSeeker

```
let price: Int
let size: Int
let snowDepth: Int
let elevation: Int
let runs: Int
let facilities: [String]
}
```

As usual, it's a good idea to add an example value to your model so that it's easier to show working data in your designs. This time, though, there are quite a few fields to work with and it's helpful if they have real data, so I don't really want to create one by hand.

Instead, we're going to load an array of resorts from JSON stored in our app bundle, which means we can re-use the same code we wrote for project 8 – the Bundle-Decodable.swift extension. If you have yours to hand you can drop it into your new project, but if not then create a new Swift file called Bundle-Decodable.swift and give it this code:

```
extension Bundle {
    func decode<T: Decodable>(_ file: String) -> T {
        guard let url = self.url(forResource: file, withExtension: nil) else {
            fatalError("Failed to locate \(file) in bundle.")
        }

        guard let data = try? Data(contentsOf: url) else {
            fatalError("Failed to load \(file) from bundle.")
        }

        let decoder = JSONDecoder()

        guard let loaded = try? decoder.decode(T.self, from: data)
        else {
            fatalError("Failed to decode \(file) from bundle.")
        }
    }
}
```

```

    }

    return loaded
}

}

```

With that in place, we can add some properties to **Resort** to store our example data, and there are two options here. The first option is to add two static properties: one to load all resorts into an array, and one to store the first item in that array, like this:

```

static let allResorts: [Resort] =
Bundle.main.decode("resorts.json")
static let example = allResorts[0]

```

The second is to collapse all that down to a single line of code. This requires a little bit of gentle typecasting because our **decode()** extension method needs to know what type of data it's decoding:

```

static let example = (Bundle.main.decode("resorts.json") as
[Resort])[0]

```

Of the two, I prefer the first option because it's simpler and has a little more use if we wanted to show random examples rather than the same one again and again. In case you were curious, when we use **static let** for properties, Swift automatically makes them lazy – they don't get created until they are used. This means when we try to read **Resort.example** Swift will be forced to create **Resort.allResorts** first, then send back the first item in that array for **Resort.example**. This means we can always be sure the two properties will be run in the correct order – there's no chance of **example** going missing because **allResorts** wasn't called yet.

Now that our simple **Resort** struct is done, we can also use that same **Bundle** extension to add a property to **ContentView** that loads all our resorts into a single array:

Project 19: SnowSeeker

```
let resorts: [Resort] = Bundle.main.decode("resorts.json")
```

For the body of our view, we're going to use a **NavigationView** with a **List** inside it, showing all our resorts. In each row we're going to show:

- A 40x25 flag of which country the resort is in.
- The name of the resort.
- How many runs it has.

40x25 is smaller than our flag source image, and also a different aspect ratio, but we can fix that by using **resizable()**, **scaledToFill()**, and a custom frame. To make it look a little better on the screen, we'll use a custom clip shape and a stroked overlay.

When the row is tapped we're going to push to a detail view showing more information about the resort, but we haven't built that yet so instead we'll just push to a temporary text view as a placeholder.

Replace your current **body** property with this:

```
NavigationView {
    List(resorts) { resort in
        NavigationLink {
            Text(resort.name)
        } label: {
            Image(resort.country)
                .resizable()
                .scaledToFill()
                .frame(width: 40, height: 25)
                .clipShape(
                    RoundedRectangle(cornerRadius: 5)
                )
                .overlay(
                    RoundedRectangle(cornerRadius: 5)
                )
        }
    }
}
```

Building a primary list of items

```
        .stroke(.black, lineWidth: 1)  
    )  
  
    VStack(alignment: .leading) {  
        Text(resort.name)  
            .font(.headline)  
        Text("\(resort.runs) runs")  
            .foregroundColor(.secondary)  
    }  
}  
}  
.navigationTitle("Resorts")  
}
```

Go ahead and run the app now and you should see it looks good enough, but if you rotate your iPhone to landscape you'll see the screen goes blank. This happens because SwiftUI wants to show a detail view there, but we haven't created one yet – let's fix that next.

Making NavigationView work in landscape

When we use a **NavigationView**, by default SwiftUI expects us to provide both a primary view and a secondary detail view that can be shown side by side, with the primary view shown on the left and the secondary on the right. This isn't *required* – you can force the push/pop **NavLink** behavior if you want by using the **navigationViewStyle()** modifier – but in this project we actually *want* the two-view behavior so we aren't going to use that.

On landscape iPhones that are big enough – iPhone 13 Pro Max, for example – SwiftUI's default behavior is to show the secondary view, and provide the primary view as a slide over. It's always been there, but you might not have realized until recently: try sliding from the left edge of your screen to reveal the **ContentView** we just made. If you tap rows in there you'll see the text behind **ContentView** change as the result of our **NavLink**, and if you *tap* on the text behind you can dismiss the **ContentView** slide over.

Now, there is a problem here, and it's the same problem you've had all along: it's not immediately obvious to the user that they need to slide from the left to reveal the list of options. In UIKit this can be fixed easily, but SwiftUI doesn't give us an alternative right now so we're going to work around the problem: we'll create a second view to show on the right by default, and use *that* to help the user discover the left-hand list.

First, create a new SwiftUI view called **WelcomeView**, then give it this code:

```
struct WelcomeView: View {
    var body: some View {
        VStack {
            Text("Welcome to SnowSeeker!")
                .font(.largeTitle)

            Text("Please select a resort from the left-hand menu;
swipe from the left edge to show it.")
        }
    }
}
```

```
    .foregroundColor(.secondary)
}
}
}
```

That's all just static text; it will only be shown when the app first launches, because as soon as the user taps any of our navigation links it will get replaced with whatever they were navigating to.

To put that into **ContentView** so the two parts of our UI can be used side by side, all we need to do is add a second view to our **NavigationView** like this:

```
NavigationView {
    List(resorts) { resort in
        // all the previous list code
    }
    .navigationTitle("Resorts")

    WelcomeView()
}
```

That's enough for SwiftUI to understand exactly what we want. Try running the app on several different devices, both in portrait and landscape, to see how SwiftUI responds – on an iPhone 13 Pro you'll see **ContentView** in both portrait and landscape, but on an iPhone 13 Pro Max you'll see **ContentView** in portrait and **WelcomeView** in landscape. If you're using an iPad, you might see several different things depending on the device orientation and whether the app has all the screen to itself as opposed to using split screen.

Although UIKit lets us control whether the primary view should be shown on iPad portrait, this is not yet possible in SwiftUI. However, we *can* stop iPhones from using the slide over approach if that's what you want – try it first and see what you think. If you want it gone, add this extension to your project:

Project 19: SnowSeeker

```
extension View {
    @ViewBuilder func phoneOnlyStackNavigationView() -> some View
{
    if UIDevice.current.userInterfaceIdiom == .phone {
        self.navigationViewStyle(.stack)
    } else {
        self
    }
}
```

That uses Apple's **UIDevice** class to detect whether we are currently running on a phone or a tablet, and if it's a phone enables the simpler **StackNavigationViewStyle** approach. We need to use the **@ViewBuilder** attribute here because the two returned view types are different.

Once you have that extension, simply add the **.phoneOnlyStackNavigationView()** modifier to your **NavigationView** so that iPads retain their default behavior whilst iPhones always use stack navigation. Again, give it a try and see what you think – it's your app, and it's important you like how it works.

Tip: I'm not going to be using this modifier in my own project because I prefer to use Apple's default behavior where possible, but don't let that stop you from making your own choice!

Creating a secondary view for NavigationView

Right now our **NavLink** directs the user to some sample text, which is fine for prototyping but obviously not good enough for our actual project. We're going to replace that with a new **ResortView** that shows a picture from the resort, some description text, and a list of facilities.

Important: Like I said earlier, the content in my example JSON is mostly fictional, and this includes the photos – these are just generic ski photos taken from Unsplash. Unsplash photos can be used commercially or non-commercially without attribution, but I've included the photo credit in the JSON so you can add it later on. As for the text, this is taken from Wikipedia. If you intend to use the text in your own shipping projects, it's important you give credit to Wikipedia and its authors and make it clear that the work is licensed under CC-BY-SA available from here: <https://creativecommons.org/licenses/by-sa/3.0>.

To start with, our **ResortView** layout is going to be pretty simple – not much more than a scroll view, a **VStack**, an **Image**, and some **Text**. The only interesting part is that we're going to show the resort's facilities as a single text view using **resort.facilities.joined(separator: ",")** to get a single string.

Create a new SwiftUI view called **ResortView**, and give it this code to start with:

```
struct ResortView: View {
    let resort: Resort

    var body: some View {
        ScrollView {
            VStack(alignment: .leading, spacing: 0) {
                Image(decorative: resort.id)
                    .resizable()
                    .scaledToFit()
            }
        }
    }
}
```

Project 19: SnowSeeker

```
Group {
    Text(resort.description)
        .padding(.vertical)

    Text("Facilities")
        .font(.headline)

    Text(resort.facilities.joined(separator: ", "))
        .padding(.vertical)
}
.padding(.horizontal)
}
.navigationTitle("\(resort.name), \(resort.country)")
.navigationBarTitleDisplayMode(.inline)
}
}
```

You'll also need to update **ResortView_Previews** to pass in an example resort for Xcode's preview window:

```
struct ResortView_Previews: PreviewProvider {
    static var previews: some View {
        ResortView(resort: Resort.example)
    }
}
```

And now we can update the navigation link in **ContentView** to point to our actual view, like this:

```
NavigationLink {
    ResortView(resort: resort)
```

Creating a secondary view for NavigationView

```
} label: {
```

There's nothing terribly interesting in our code so far, but that's going to change now because I want to add more details to this screen – how big the resort is, roughly how much it costs, how high it is, and how deep the snow is.

We could just put all that into a single **HStack** in **ResortView**, but that restricts what we can do in the future. So instead we're going to group them into two views: one for resort information (price and size) and one for ski information (elevation and snow depth).

The ski view is the easier of the two to implement, so we'll start there: create a new SwiftUI view called **SkiDetailsView** and give it this code:

```
struct SkiDetailsView: View {
    let resort: Resort

    var body: some View {
        Group {
            VStack {
                Text("Elevation")
                    .font(.caption.bold())
                Text("\(resort.elevation)m")
                    .font(.title3)
            }

            VStack {
                Text("Snow")
                    .font(.caption.bold())
                Text("\(resort.snowDepth)cm")
                    .font(.title3)
            }
        }
        .frame(maxWidth: .infinity)
```

Project 19: SnowSeeker

```
    }
}

struct SkiDetailsView_Previews: PreviewProvider {
    static var previews: some View {
        SkiDetailsView(resort: Resort.example)
    }
}
```

Giving the **Group** view a maximum frame width of `.infinity` doesn't actually affect the group itself, because it has no impact on layout. However, it *does* get passed down to its child views, which means they will automatically spread out horizontally.

As for the resort details, this is a little trickier because of two things:

1. The size of a resort is stored as a value from 1 to 3, but really we want to use “Small”, “Average”, and “Large” instead.
2. The price is stored as a value from 1 to 3, but we’re going to replace that with \$, \$\$, or \$\$\$.

As always, it’s a good idea to get calculations out of your SwiftUI layouts so it’s nice and clear, so we’re going to create two computed properties: **size** and **price**.

Start by creating a new SwiftUI view called **ResortDetailView**, and give it this property:

```
let resort: Resort
```

As with **ResortView**, you’ll need to update the preview struct to use some example data:

```
struct ResortDetailView_Previews: PreviewProvider {
    static var previews: some View {
        ResortDetailView(resort: Resort.example)
    }
}
```

Creating a secondary view for NavigationView

When it comes to getting the size of the resort we could just add this property to **ResortDetailsView**:

```
var size: String {  
    ["Small", "Average", "Large"][resort.size - 1]  
}
```

That works, but it would cause a crash if an invalid value was used, and it's also a bit too cryptic for my liking. Instead, it's safer and clearer to use a **switch** block like this:

```
var size: String {  
    switch resort.size {  
        case 1:  
            return "Small"  
        case 2:  
            return "Average"  
        default:  
            return "Large"  
    }  
}
```

As for the **price** property, we can leverage the same repeating/count initializer we used to create example cards in project 17: **String(repeating:count:)** creates a new string by repeating a substring a certain number of times.

So, please add this second computed property to **ResortDetailsView**:

```
var price: String {  
    String(repeating: "$", count: resort.price)  
}
```

Now what remains in the **body** property is simple, because we just use the two computed properties we wrote:

Project 19: SnowSeeker

```
var body: some View {
    Group {
        VStack {
            Text("Size")
                .font(.caption.bold())
            Text(size)
                .font(.title3)
        }

        VStack {
            Text("Price")
                .font(.caption.bold())
            Text(price)
                .font(.title3)
        }
    }
    .frame(maxWidth: .infinity)
}
```

Again, giving the whole **Group** an infinite maximum width means these views will spread out horizontally just like those from the previous view.

That completes our two mini views, so we can now drop them into **ResortView** – put this just before the group in **ResortView**:

```
HStack {
    ResortDetailsView(resort: resort)
    SkiDetailsView(resort: resort)
}
.padding(.vertical)
.background(Color.primary.opacity(0.1))
```

We're going to add to that some more in a moment, but first I want to make one small tweak:

using **joined(separator:)** does an okay job of converting a string array into a single string, but we're not here to write okay code – we're here to write *great* code.

Previously we've used the **format** parameter of **Text** to control the way numbers are formatted, but there's a format for string arrays too. This is similar to using **joined(separator:)**, but rather than sending back “A, B, C” like we have right now, we get back “A, B, *and* C” – it's more natural to read.

Replace the current facilities text view with this:

```
Text(resort.facilities, format: .list(type: .and))  
    .padding(.vertical)
```

Notice how the **.and** type is there? That's because you can also use **.or** to get “A, B, or C” if that's what you want.

Anyway, it's a tiny change but I think it's much better!

Searching for data in a List

Before our **List** view is done, we're going to add a SwiftUI modifier that makes our user's experience a whole lot better without too much work: **searchable()**. Adding this will allow users to filter the list of resorts we're showing, making it easy to find the exact thing they're looking for.

This takes only four steps, starting with a new **@State** property in **ContentView** to store the text the user is searching for:

```
@State private var searchText = ""
```

Second, we can bind that to our **List** in **ContentView** by adding this directly below the existing **navigationTitle()** modifier:

```
.searchable(text: $searchText, prompt: "Search for a resort")
```

Third, we need a computed property that will handle the filtering of our data. If our new **searchText** property is empty then we can just send back all the resorts we loaded, otherwise we'll use **localizedCaseInsensitiveContains()** to filter the array based on their search criteria:

```
var filteredResorts: [Resort] {
    if searchText.isEmpty {
        return resorts
    } else {
        return resorts.filter
    { $0.name.localizedCaseInsensitiveContains(searchText) }
    }
}
```

And the final step is to use **filteredResorts** as the data source for our list, like this:

```
List(filteredResorts) { resort in
```

Searching for data in a List

And with that we're done! If you run the app again you'll see you can drag the resort list gently down to see the search box, and entering something in there will filter the list straight away. Honestly, **searchable()** is one of the biggest "bang for buck" features in SwiftUI – it's such an important feature for users, and took us only a few minutes to implement!

Changing a view's layout in response to size classes

SwiftUI gives us two environment values to monitor the current size class of our app, which in practice means we can show one layout when space is restricted and another when space is plentiful.

For example, in our current layout we're displaying the resort details and snow details in a **HStack**, like this:

```
HStack {  
    ResortDetailsView(resort: resort)  
    SkiDetailsView(resort: resort)  
}
```

Each of those subviews are internally using a **Group** that doesn't add any of its own layout, so we end up with all four pieces of text laid out horizontally. This looks great when we have enough space, but when space is limited it would be helpful to switch to a 2x2 grid layout.

To make this happen we *could* create copies of **ResortDetailsView** and **SkiDetailsView** that handle the alternative layout, but a much smarter solution is to have both those views remain *layout neutral* – to have them automatically adapt to being placed in a **HStack** or **VStack** depending on the parent that places them.

First, add this new **@Environment** property to **ResortView**:

```
@Environment(\.horizontalSizeClass) var sizeClass
```

That will tell us whether we have a regular or compact size class. Very roughly:

- All iPhones in portrait have compact width and regular height.
- Most iPhones in landscape have compact width and compact height.
- Large iPhones (Plus-sized and Max devices) in landscape have regular width and compact

height.

- All iPads in both orientations have regular width and regular height when your app is running with the full screen.

Things get a little more complex for iPad when it comes to split view mode, which is when you have two apps running side by side – iOS will automatically downgrade our app to a compact size class at various points depending on the exact iPad model.

Fortunately, to begin with all we care about are these two horizontal options: do we have lots of horizontal space (regular) or is space restricted (compact). If we have a regular amount of space, we're going to keep the current **HStack** approach so that everything fits neatly on one line, but if space is restricted we'll ditch that and place each of the views into a **VStack**.

So, find the **HStack** that contains **ResortDetailsView** and **SkiDetailsView** and replace it with this:

```
HStack {
    if sizeClass == .compact {
        VStack(spacing: 10) { ResortDetailsView(resort: resort) }
        VStack(spacing: 10) { SkiDetailsView(resort: resort) }
    } else {
        ResortDetailsView(resort: resort)
        SkiDetailsView(resort: resort)
    }
}.padding(.vertical)
.background(Color.primary.opacity(0.1))
```

As you can see, that uses two vertical stacks placed side by side, rather than just having all four views horizontal.

Is it perfect? Well, no. Sure, there's a lot more space in compact layouts, which means the user can use larger Dynamic Type sizes without running out of space, but many users won't have

Project 19: SnowSeeker

that problem because they'll be using the default size or even smaller sizes.

To make this even better we can combine a check for the app's current horizontal size class with a check for the user's Dynamic Type setting so that we use the flat horizontal layout unless space really is tight – if the user has a compact size class and a larger Dynamic Type setting.

First add another property to read the current Dynamic Type setting:

```
@Environment(\.dynamicTypeSize) var typeSize
```

Now modify the size class check to this:

```
if sizeClass == .compact && typeSize > .large {
```

Now finally our layout should look great in both orientations: one single line of text in a regular size class, and two rows of vertical stacks in a compact size class when an increased font size is used. It took a little work, but we got there in the end!

Our solution didn't result in code duplication, which is a huge win, but it also left our two child views in a better place – they are now there just to serve up their content without specifying a layout. So, parent views can dynamically switch between **HStack** and **VStack** whenever they want, and SwiftUI will take care of the layout for us.

Before we're done, I want to show you one useful extra technique: you can limit the range of Dynamic Type sizes supported by a particular view. For example, you might have worked hard to support as wide a range of sizes as possible, but found that anything larger than the "extra extra extra large" setting just looks bad. In that situation you can use the **dynamicTypeSize()** modifier on a view, like this:

```
.dynamicTypeSize(...DynamicTypeSize.xxxLarge)
```

That's a one-sided range, meaning that any size up to and including **.xxxLarge** is fine, but nothing larger. Obviously it's best to avoid setting these limits where possible, but it's not a

Changing a view's layout in response to size classes

problem if you use it judiciously – both **TabView** and **NavigationView**, for example, limit the size of their text labels so the UI doesn't break.

Binding an alert to an optional string

SwiftUI lets us present an alert with an optional source of truth inside, but it takes a little thinking to get right as you'll see.

To demonstrate these optional alerts in action, we're going to rewrite the way our resort facilities are shown. Right now we have a plain text view generated like this:

```
Text(resort.facilities, format: .list(type: .and))  
.padding(.vertical)
```

We're going to replace that with icons that represent each facility, and when the user taps on one we'll show an alert with a description of that facility.

As usual we're going to start small then work our way up. First, we need a way to convert facility names like "Accommodation" into an icon that can be displayed. Although this will only happen in **ResortView** right now, this functionality is exactly the kind of thing that should be available elsewhere in our project. So, we're going to create a new struct to hold all this information for us.

Create a new Swift file called Facility.swift, replace its Foundation import with SwiftUI, and give it this code:

```
struct Facility: Identifiable {  
    let id = UUID()  
    var name: String  
  
    private let icons = [  
        "Accommodation": "house",  
        "Beginners": "1.circle",  
        "Cross-country": "map",  
        "Eco-friendly": "leaf.arrow.circlepath",
```

Binding an alert to an optional string

```
"Family": "person.3"
]

var icon: some View {
    if let iconName = icons[name] {
        return Image(systemName: iconName)
            .accessibilityLabel(name)
            .foregroundColor(.secondary)
    } else {
        fatalError("Unknown facility type: \(name)")
    }
}
```

As you can see, that conforms to **Identifiable** so we can loop over an array of facilities with SwiftUI, and internally it looks up a given facility name in a dictionary to return the correct icon. I've picked out various SF Symbols icons that work well for the facilities we have, and I also used an **accessibilityLabel()** modifier for the image to make sure it works well in VoiceOver.

The next step is to create **Facility** instances for every of the facilities in a **Resort**, which we can do in a computed property inside the **Resort** struct itself:

```
var facilityTypes: [Facility] {
    facilities.map(Facility.init)
}
```

We can now drop that facilities view into **ResortView** by replacing this code:

```
Text(resort.facilities, format: .list(type: .and))  
    .padding(.vertical)
```

With this:

Project 19: SnowSeeker

```
HStack {  
    ForEach(resort.facilityTypes) { facility in  
        facility.icon  
            .font(.title)  
    }  
}  
.padding(.vertical)
```

That loops over each item in the **facilities** array, converting it to an icon and placing it into a **HStack**. I used the **.font(.title)** modifier to make the images larger – using the modifier here rather than inside **Facility** allows us more flexibility if we wanted to use these icons in other places.

That was the easy part. The harder part comes next: we want to make the facility images into buttons, so that we can show an alert when they are tapped.

Using the optional form of **alert()** this starts easily enough – add two new properties to **ResortView**, one to store the currently selected facility, and one to store whether an alert should currently be shown or not:

```
@State private var selectedFacility: Facility?  
@State private var showingFacility = false
```

Now replace the previous **ForEach** loop with this:

```
ForEach(resort.facilityTypes) { facility in  
    Button {  
        selectedFacility = facility  
        showingFacility = true  
    } label: {  
        facility.icon  
            .font(.title)  
    }  
}
```

```
}
```

We can create the alert in a very similar manner as we created the icons – by adding a dictionary to the **Facility** struct containing all the keys and values we need:

```
private let descriptions = [
    "Accommodation": "This resort has popular on-site
accommodation.",
    "Beginners": "This resort has lots of ski schools.",
    "Cross-country": "This resort has many cross-country ski
routes.",
    "Eco-friendly": "This resort has won an award for
environmental friendliness.",
    "Family": "This resort is popular with families."
]
```

Then reading that inside another computed property:

```
var description: String {
    if let message = descriptions[name] {
        return message
    } else {
        fatalError("Unknown facility type: \(name)")
    }
}
```

So far this hasn't been tricky, but now comes the complex part. You see, the **selectedFacility** property is optional, so we need to handle it carefully:

- We can't use it as the only title for our alert, because we must provide a non-optional string. We can fix that with nil coalescing.
- We always want to make sure the alert reads from our optional **selectedFacility**, so it passes in the unwrapped value from there.

Project 19: SnowSeeker

- We don't need any buttons in this alert, so we can let the system provide a default OK button.
- We need to provide an alert message based on the unwrapped facility data, calling the new **message(for:)** method we just wrote.

Putting all that together, add this modifier below **navigationBarTitleDisplayMode()** in **ResortView**:

```
.alert(selectedFacility?.name ?? "More information",
isPresented: $showingFacility, presenting: selectedFacility)
{ _ in
} message: { facility in
    Text(facility.description)
}
```

Notice how we're using **_ in** for the alert's action closure because we don't actually care about getting the unwrapped **Facility** instance there, but it *is* important in the **message** closure so we can display the correct description.

Letting the user mark favorites

The final task for this project is to let the user assign favorites to resorts they like. This is mostly straightforward, using techniques we've already covered:

- Creating a new **Favorites** class that has a **Set** of resort IDs the user likes.
- Giving it **add()**, **remove()**, and **contains()** methods that manipulate the data, sending update notifications to SwiftUI while also saving any changes to **User Defaults**.
- Injecting an instance of the **Favorites** class into the environment.
- Adding some new UI to call the appropriate methods.

Swift's sets already contain methods for adding, removing, and checking for an element, but we're going to add our own around them so we can use **objectWillChange** to notify SwiftUI that changes occurred, and also call a **save()** method so the user's changes are persisted. This in turn means we can mark the favorites set using **private** access control, so we can't accidentally bypass our methods and miss out saving.

Create a new Swift file called Favorites.swift, replace its Foundation import with SwiftUI, then give it this code:

```
class Favorites: ObservableObject {  
    // the actual resorts the user has favorited  
    private var resorts: Set<String>  
  
    // the key we're using to read/write in UserDefaults  
    private let saveKey = "Favorites"  
  
    init() {  
        // load our saved data  
  
        // still here? Use an empty array  
        resorts = []  
    }  
}
```

Project 19: SnowSeeker

```
// returns true if our set contains this resort
func contains(_ resort: Resort) -> Bool {
    resorts.contains(resort.id)
}

// adds the resort to our set, updates all views, and saves
// the change
func add(_ resort: Resort) {
    objectWillChange.send()
    resorts.insert(resort.id)
    save()
}

// removes the resort from our set, updates all views, and
// saves the change
func remove(_ resort: Resort) {
    objectWillChange.send()
    resorts.remove(resort.id)
    save()
}

func save() {
    // write out our data
}
}
```

You'll notice I've missed out the actual functionality for loading and saving favorites – that will be your job to fill in shortly.

We need to create a **Favorites** instance in **ContentView** and inject it into the environment so all views can share it. So, add this new property to **ContentView**:

```
@StateObject var favorites = Favorites()
```

Now inject it into the environment by adding this modifier to the **NavigationView**:

```
.environmentObject(favorites)
```

Because that's attached to the navigation view, every view the navigation view presents will also gain that **Favorites** instance to work with. So, we can load it from inside **ResortView** by adding this new property:

```
@EnvironmentObject var favorites: Favorites
```

Tip: Make sure you modify your **ResortView** preview to inject an example **Favorites** object into the environment, so your SwiftUI preview carries on working. This will work fine: `.environmentObject(Favorites())`.

All this work hasn't really accomplished much yet – sure, the **Favorites** class gets loaded when the app starts, but it isn't actually used anywhere despite having properties to store it.

This is easy enough to fix: we're going to add a button at the end of the scrollview in **ResortView** so that users can either add or remove the resort from their favorites, then display a heart icon in **ContentView** for favorite resorts.

First, add this to the end of the scrollview in **ResortView**:

```
Button(favorites.contains(resort) ? "Remove from Favorites" :  
"Add to Favorites") {  
    if favorites.contains(resort) {  
        favorites.remove(resort)  
    } else {  
        favorites.add(resort)  
    }  
}  
.buttonStyle(.borderedProminent)
```

Project 19: SnowSeeker

```
.padding()
```

Now we can show a colored heart icon next to favorite resorts in **ContentView** by adding this to the end of the **NavLink**:

```
if favorites.contains(resort) {  
    Spacer()  
    Image(systemName: "heart.fill")  
        .accessibilityLabel("This is a favorite resort")  
        .foregroundColor(.red)  
}
```

Tip: As you can see, the **foregroundColor()** modifier works great here because our image uses SF Symbols.

That *mostly* works, but you might notice a glitch: if you favorite resorts with longer names you might find their name wraps onto two lines even though there's space for it to be all on one.

This happens because we've made an assumption in our code, and it's coming back to bite us: we were passing an **Image** and a **VStack** into the label for our **NavLink**, which SwiftUI was smart enough to arrange neatly for us, but as soon as we added a *third* view it wasn't sure how to respond.

To fix this, we need to tell SwiftUI explicitly that the content of our **NavLink** is a plain old **HStack**, so it will size everything appropriately. So, wrap the entire contents of the **NavLink** label – everything from the **Image** down to the new condition wrapping the heart icon – inside a **HStack** to fix the problem.

That should make the text layout correctly even with the spacer and heart icon – much better. And that also finishes our project, so give it one last try and see what you think. Good job!

SnowSeeker: Wrap up

This wasn't a particularly complicated project, but it still taught new skills such as split view layouts, optional alerts, layout with transparent groups, and even an improved way of formatting lists in text. It also gave you the chance to practice lots of core skills, such as handling **Codable**, creating scrolling lists, and more.

This kind of app is really good as a template, because it's the kind of thing you can use again and again in the future just by varying the kind of content you feed into it. Template apps – as well as the techniques underlying them – are the “bread and butter” of iOS apps: the kind of thing that is never going to set the world on fire, but also so fundamentally useful and common that you can't do without them.

Review what you learned

Anyone can sit through a tutorial, but it takes actual work to remember what was taught. It's my job to make sure you take as much from these tutorials as possible, so I've prepared a short review to help you check your learning.

[Click here to review what you learned in this project.](#)

Challenge

One of the best ways to learn is to write your own code as often as possible, so here are three ways you should try extending this app to make sure you fully understand what's going on.

1. Add a photo credit over the **ResortView** image. The data is already loaded from the JSON for this purpose, so you just need to make it look good in the UI.
2. Fill in the loading and saving methods for **Favorites**.
3. For a real challenge, let the user sort the resorts in **ContentView** either using the default order, alphabetical order, or country order.

Hacking with Swift+ subscribers can get a complete video solution for this checkpoint here:

Project 19: SnowSeeker

Solution to SnowSeeker. If you don't already subscribe, you can start a free trial today.