The Pragmatic Programmers

Coming Soon

Early praise for Mastering SwiftUI

It was the best of books, it was the worst of books...

➤ **Eddy the Gerbil**
Chief Gerbil, Gerbils-r-us

We've left this page blank to make the page numbers the same in the electronic and paper books.

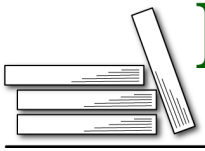
We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say “hello.”

Mastering SwiftUI

Jim Dovey

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

For Joanna, Olivia, and Jacob, who let me lock myself away for many evenings to write this book.

In Memoriam: Oli Glaser

9 March 1977 – 16 February 2020.

The crystal ship has been waiting for you, my friend.

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: Monthname xx, yyyy

- Initial beta release.

Acknowledgments

Introduction

Story Map

Why do I want to read this?

This chapter will give you a simple overview of the book's contents and principal organization. It will also introduce Declarative UI and some Swift 5.1 affordances such as Property Wrappers and Builders, which are used to implement several core SwiftUI features.

What will I learn?

A simple overview of declarative UI design and a look at the basics of how this is used by SwiftUI. You'll also learn some features of Swift 5.1 that are used by SwiftUI.

What will I be able to do that I couldn't do before?

You'll be able to discuss the merits of declarative UI coding and how it differs from the model used by UIKit and AppKit. This knowledge will help you understand the new language features which enable some of SwiftUI's capabilities and syntax.

Where are we going next, and how does this fit in?

Up next is the description of the application you'll be building during the remainder of the book. The application is a basic to-do list / GTD application; it was chosen because you can start with canned List-based interfaces, and grow it to include several customized views and interactions.

SwiftUI arrived to the surprise of many on June 3, 2019 at Apple's WorldWide Developer Conference (WWDC) in San Jose, California. There had been some rumblings suggesting a new UI framework in the prior couple of years, but most attention was focussed on the Catalyst project, which would allow iPad applications to be built for and deployed on macOS. When that was officially announced at WWDC, everyone relaxed, not expecting anything else along

those lines. Then came the big reveal: there was a new UI framework written entirely in Swift, and it worked on all of Apple's platforms.

This new framework hit all the right buttons, reflecting as it did much of the programming world's zeitgeist of the prior few years:

- The Swift Programming Language. Swift was becoming popular, and its design implemented a number of features that Apple's older Objective-C language did not, such as true generics, protocol-oriented programming, built-in block syntax, first-class functions, and high-level value types with member functions and visibility.
- Immutable Types. By making use of immutable types, a lot of the complexity of dealing with data in multi-threaded applications disappears. Data changes atomically at a larger logical scale, rather than by smaller piecemeal updates.
- Declarative Syntax. SwiftUI's users would be able to describe their interfaces in a declarative manner (one of these on top of one of those, with a blue background and a picture of a cat) rather than the usual iterative approach (create a view, set its background color, get a picture of a cat, make it this big, move it over there). The syntax was more terse, and more closely matched the logical layout of the interface, with details neatly tucked away elsewhere.
- Functional Design. Much of SwiftUI's appeal came thanks to its judicious use of functional programming conventions, using first-class methods and blocks as part of the UI definition. Amongst other things, this would enable closure-based event handling code to be written at the point of declaration, where it's easy to see and to update.

A Little UI API History

Since the advent of the GUI in the early 1980's there have been several different ways of describing your user interface, from the original Smalltalk implementation at Xerox PARC through Mac OS, Windows, and Java, up to UIKit and SwiftUI today. Smalltalk and NeXTstep embraced a fully object-oriented approach to user interface design, while the classic Macintosh and Microsoft Windows used procedural APIs implemented in C. Java, by its nature, used an object-oriented system, albeit one built on top of the procedural systems provided by the platforms on which it ran.

Working with the interface itself was different on each platform, but the Interface Builder included with NeXTstep's developer tools set the course for

the rest of the industry: drag-and-drop layout of user interface elements. Interface Builder would create serialized representations of the UI, while on other platforms the tools generally generated actual code in C or Java. Eventually the UI-as-data idea began to expand, though, and we saw Microsoft use an XML syntax to define UIs for its .NET family of languages, while Java added JavaFX.

The common thread running through all these APIs is their iterative format. Though both object-oriented and procedural APIs existed, they all used an instructional, step-by-step means of describing properties and changes. In essence the programmers using them were writing precise recipes: "take this, move it here, give it this color, rotate it, then put this other thing on top, add an image," and so on. Tools such as Interface Builder, Windows Forms, and JavaFX were providing a simpler way to encode these instructions, but anything beyond the purview of those tools would necessitate a switch to a functionally very different approach to solving the same problem.

The Declarative UI Paradigm

A primary benefit of a declarative user interface API is that the simpler, declarative model is used *everywhere*. In code, and in interface description files. In fact, with SwiftUI, only code is used, along with some new language features you'll meet later, to allow that code to visually mimic that layout to some degree in the same manner as, say, an XML-based hierarchical description format. This means that there is only one set of concepts and tools to learn, and only one way to define and interact with your interface model.

Apple illustrates the difference between the iterative and declarative paradigms using the metaphor of a food order. In iterative style, you describe the instructions on how to make the food, as if describing the entire process over the phone: "finely chop an onion, two cloves of garlic, and a stick of celery, then add to the pan with a little olive oil, sauté for five minutes; measure 250g arborio rice;" etc. You're effectively writing a recipe, and relying on the chef knowing a few things, like how to chop an onion. Declaratively, you're making an order: "a risotto with fennel and extra parmesan." The person you're talking to knows how to make a risotto, you don't need to tell them. You just lay out the specifics of your own case.

That last sounds somewhat familiar: it's the same idea behind the frameworks approach espoused by the AppKit and UIKit frameworks. You don't need to write a `main()` method that initializes different parts of the system, you don't need to write an event-processing loop, you don't even need to write a function

that would receive raw events and dispatch them. The framework handles that, and does the normal things for you—your app launches and runs without anything special on your part. What you provide is the customization: the app will ask "the last window has been closed; should I quit now?" and you will answer yes or no, but only if you have a preference. The app will ask "what items should I display in this table?" and you'll provide the data.

The declarative model used by SwiftUI, then, is a somewhat natural extension of the existing framework model. The framework knows how to put things onscreen, and how to work with them; you simply need to specify some specifics of what should go in what order. The instructions are, ultimately, still there, but they're tucked out of sight. If you need to do something very complex and specific, then (to go back to the earlier metaphor) you write a recipe and give that to the cook, then you can ask the cook to include that recipe in your dish: poached salmon, rather than grilled.

Declarative code also fits mental models quite snugly. For example, here's a list, where each item contains a stack of two text fields, one on top of the other, with these fonts:

```
List {
        VStack {
                Text("The Title").font(.title)
                Text("The Subtitle").font(.subtitle)
        }
}
```

This code defines a button with some text and a blue background. It also includes an action that plays a sound when the user clicks or taps the button:

```
Button(action: { playSound() }) {
        Text("Title")
}.backgroundColor(.blue)
```

Finally, this code display an image within a white circular border with a drop shadow behind it:

```
Image("myImage.png")
        .clipShape(Circle())
        .overlay(Circle().stroke(Color.white, lineWidth: 4))
        .shadow(radius: 10)
```

Example Application

In the remainder of this book, you're going to use SwiftUI to build an iOS application named "Do It." You'll use a small value-type data model to build the app for the iPhone initially, but then you'll move on to iPadOS and its

greater feature set, adding functionality all the while. Over the course of the book, you'll assemble this application in the following steps:

- Chapter One: Layout and Presentation. Here, you'll build a simple list-based app showing data in a master/detail format.
- Chapter Two: Application Data in SwiftUI. This chapter introduces the use of the SwiftUI `@State` type and how it's used to drive updates to your interface.
- Chapter Three: Modifying Application Data. SwiftUI's tools for handling state and data modification form the focus of this chapter as you implement editing tools for our data set.
- Chapter Four: List Mutation. Implement filtering for your to-do lists, along with an update to support delivery of predicates from higher in the view hierarchy.
- Chapter Five: Custom Views and Complex Interactions. SwiftUI provides special tools for reacting to user input and adapting your views' size and location dynamically, all in a concise declarative syntax. Here you'll learn how these work and what you can do with them.
- Chapter Six: Making the Most of the Canvas. In this chapter you'll learn how to make good use of the Xcode canvas as a development tool, using it to preview the effect various system settings have on your application.
- Chapter Seven: SwiftUI on iPadOS. In this chapter you'll investigate the additional features of iPadOS, contrasting with the similarities to the existing iPhone application. You'll work with multiple windows, keyboard and pointer support, different display options, and lay the groundwork for drag and drop ready for the next chapter.
- Chapter Eight: Implementing Drag and Drop. iPadOS and macOS both support transfer of data by dragging items into and out of your application. You'll focus on iPad as you make use of several different approaches to getting information into and out of the app, implement gestures to open new windows, and move data between those windows.
- Chapter Nine: Core Data and Combine. So far your application has been defined using only Swift structure types and a global data store. Many applications deal with a much more complex data store, so here you'll move to a more easily-modified data model defined in Core Data. You'll use SwiftUI's Core Data tools to bring in and display your new data, and

you'll see how to make good use of CoreData's predicates to drive your interface.

CHAPTER 1

Layout and Presentation

Story Map

Why do I want to read this?

This chapter will teach you how to create a new SwiftUI application, and how to perform basic layout tasks.

What will I learn?

The basics of creating a List-based interface for displaying ordered data, using text and images.

What will I be able to do that I couldn't do before?

You'll know how to start a SwiftUI application, and how to create and manipulate views.

Where are we going next, and how does this fit in?

The next chapter adds a little color to the application as you add a detail view and look at data propagation and sharing.

It's time to take your first steps into a bright new world. SwiftUI brings with it a new way of thinking about user interface development, giving you an easy-to-use syntax and flexible types that allow you to concisely express your design.

SwiftUI considers the user interface to be a function of data, and this is how you'll need to think of your views. You are no longer putting together an interface as its own object, and then wiring it into some data with the aid of controller code. Instead, the SwiftUI `View` is more akin to a *transformation* that is applied to your data and state: when the data and state change, the code you write for your views provides the instructions for SwiftUI to create an appropriate on-screen representation of that data. This is why SwiftUI views

are struct types, lightweight and all but ephemeral, which in turn makes them easy to compose together.

In this chapter, you'll start work on the application that you'll use throughout the remainder of this book. You'll see what Xcode provides to you when you create new projects and new SwiftUI views. You'll learn how to use the building blocks that SwiftUI provides as you build out this application. There's a lot to learn, but happily, the framework is concise and internally consistent, so you ought never to find yourself scratching your head in confusion.

Getting Started

The best way to learn SwiftUI is to work with it directly. In that vein, you'll immediately start building the sample application. What you create in this chapter will get modified throughout the course of this book, morphing from a simple to-do list application to a more complex and interactive project management tool, with support for every Apple platform.

The first thing to do is to create a new project. Launch Xcode and select Create a new Xcode project or choose File → New → Project.... Select iOS in the tab bar and choose the Single View App template. Click Next, and in the next section, name your project Do It, and ensure that SwiftUI is selected for the User Interface. Finally, choose a location to save your project, and click Create.

Do It: A Brief History Lesson

In the early days of the development of the Apple Macintosh computer, the familiar OK/Cancel dialogs didn't exist.[a] The OK button was felt to be too specific to the US dialect, so the designers settled on "Do It" instead. However, it didn't test well. People were clicking on the Cancel button instead, baffling the team. One day, the team learned the reason when a tester got more and more agitated, eventually asking, "Why is the software calling me a dolt?"

The team realized that with the Chicago font on the Macintosh, a serif-less capital I looks a lot like a lowercase L. With the inter-word spacing being fairly compact within the buttons, people were reading it as Dolt instead of Do It. They changed the text to OK and found that, against their suspicions, it worked perfectly well in all localizations.

a. https://www.folklore.org/StoryView.py?project=Macintosh&story=Do_It.txt

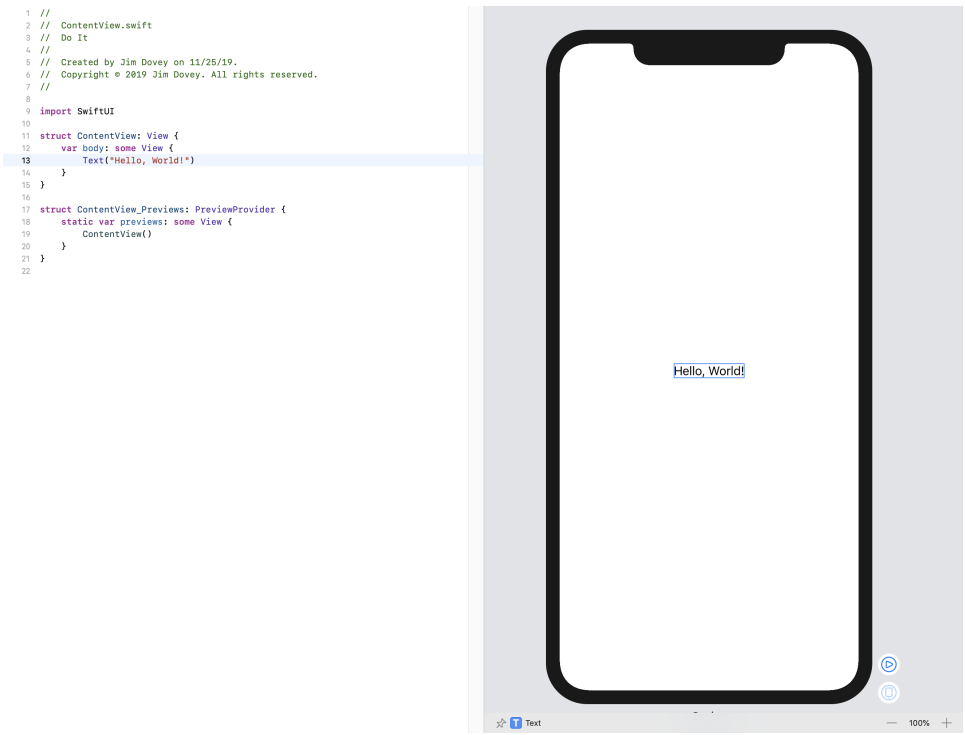
Learning the SwiftUI Basics

SwiftUI takes a layered approach to interface programming. It tries to make the most common actions simple and quick, reserving complexity for the edge cases. You'll see this as you look at the sample view created by Xcode.

From the Project Navigator, select the ContentView.swift file to open it. You'll see that by default, there's a two-pane editor rather than the single text editor of earlier Xcode versions. To the left is the source code editor; to the right lies a new pane known as the *canvas*.

The canvas allows you to immediately see the results of any changes you make to your view code in the editor. Additionally, the canvas acts rather like Interface Builder in that you can select and edit components, and you can drag and drop existing items and new ones from a palette. Initially, though, it's not running. When you make changes to a view file outside of the body property implementation, the canvas will pause. Click the Resume button at the top right of the canvas to see your view.

By default, the canvas shows an iPhone containing your view as its sole content:



Currently, there's only the "Hello, World!" text centered within the screen. Look at the editor to the left, and to the body property implementation. It contains the code Text("Hello, World!") and nothing more. Change the string value to "Hello, SwiftUI!", and notice how the canvas immediately updates to reflect your change.

View Types

Look at the type declaration for ContentView:

```
struct ContentView: View {
    «implementation»
}
```

This type formulation is a little different from UIKit and AppKit. In earlier frameworks, all views and controllers have been class types, meaning they have reference semantics. Each instance exists once in memory, and other items reference that single instance. Here, though, you have a struct type, which uses value semantics. It can be mutable or immutable, and its contents are copied into new instances on assignment, making each copy independent.

Now, this may sound counter-intuitive. After all, you might reason that a view is showing something on the display, and so you want reference semantics to make alterations to that. This is a reasonable way to do things, though it doesn't scale as well as it might. Once a view has become sufficiently complex (as most UIKit views are, for instance), there's a lot of disparate state information held by each one, and there's no simple way to guarantee ordered updates to that state.

Consider how this is affected when you introduce animation: each animatable piece of state information has both a logical value (where it's going) and concrete value (where it's at this moment in the animation). When you set a view's alpha component from 1.0 to 0.0 with a two-second animation, then the alpha may logically be 0.0, but the concrete value is changing over time, and after one second may likely be closer to 0.5.

Alternatively, the logical value might remain at 1.0 until the animation is done. How does this affect future changes that look at the alpha value to make decisions, such as whether to enable complex blend modes? The alpha will be zero, quite soon, but isn't yet. Now you have to think about whether you'll allow interaction, for example, during the animation—do you disable interaction before the animation and restore it if the animation fails to complete? Do you only disable it after the animation succeeds?

Yep, there's quite a deep rabbit-hole there, and it's not exactly Wonderland waiting at the bottom.

SwiftUI works around this by using value types to describe all views. In effect, what SwiftUI calls a *view* is more accurately a *view description*. When state associated with a view is updated (as you'll see in Dynamically Ordering List Contents, on page 42), SwiftUI asks for a new implementation of that view—a new description—which incorporates the new state. SwiftUI then ensures that all of these changes are applied together, in a single logical step. It can coalesce multiple state changes together into a single update, and it can take ownership of state change animation to directly manage the concrete values displayed on screen. One result of this is that state change animations are always reversible and always adjust correctly when state is mutated multiple times during an animation. Your description specifies the logical end state, and SwiftUI works out how it should get there.

Opaque Types and Implicit Returns

The `ContentView` conforms to the `View` protocol. As a value type, it can't inherit from a parent class, but it can take advantage of Swift's ability to define default implementations for protocol members. As a result, defaults for every method on `View` exist already, with the exception of the `body` property, which you must provide, and is present in the template:

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

Here, again, are some things that look a little different to the Swift you know. First, the code implementing the property relies on a new feature in Swift 5.1 called *implicit return statements*. For a while now, Swift's block syntax has contained a shorthand: when a block contains only one expression, then the result of that expression is implicitly returned from the block. That has enabled brief and easy-to-read code such as the following:

```
names.map { $0.uppercased() }
```

From Swift 5.1 onwards, this shorthand is available for all functions and property accessors that match the requirements: a single expression gets an implicit `return` added for them by the compiler:

```
func uppercase(name: String) -> String {
    name.uppercased()
}
```

With this facility, SwiftUI code now looks the same whether nesting via an inline block or returning from a bespoke function, keeping the syntax visually similar and code directly interchangeable between different locations.

Implicit return statements only work for single-statement implementations, though. Try adding `let x = 0` at the top of the `body` implementation and you'll see a compiler error appear. With more than one expression in the method body, you need to use an explicit `return` keyword.

Secondly, it uses an *opaque return type*, which helps deal with protocol-based types in an efficient way while maintaining a succinct declaration syntax.

Protocols in Swift are powerful. They can define lots of functionality, can contain default implementations of methods, and can define some quite complex behaviors based on associated types. This last feature, however, can cause some pain. Let's say you have the following protocol definition:

```
protocol Identifiable {
    associatedtype Identifier: Hashable
    var id: Identifier { get }
}
extension Identifiable where Identifier == String {
    var uppercaseID { id.uppercased() }
}
```

Some problems occur when you want to make use of the protocol as its own type, however:

```
func doSomething(with: Any) -> Identifiable {
    «body»
}
```

Here the compiler will complain: `Protocol 'Identifiable' can only be used as a generic constraint because it has Self or associated type requirements`. It can't determine how to build this function, because without knowing the actual type of `Identifiable.Identifier` that was provided it can't allocate enough memory, and it can't tell if initializers and destructors for the associated type are needed.

That's all very well for the compiler, but you just want to say that you want some sort of `Identifiable` to be returned. You don't care exactly what, and thus the extra work of making this function generic seems like overhead. Also, generic protocol wrapper types have some downsides in terms of size and processing overhead, so this may be unusable within performance-critical code.

For this reason, Swift 5.1 introduces *opaque return types*. Adopting them is as simple as putting the keyword `some` in front of your return type:

```
func doSomething(with: Any) -> some Identifiable {
    «body»
}
```

The compiler is now happy: it will inspect the code inside the function and determine the *actual* type being returned. The details of that type will be known to the compiler and will be used by any functions calling this one, but will be hidden from the programmer, keeping the interface simple and logical. SwiftUI uses this for its View types; many functions return some View.

Note that opaque return types only hide complexity in terms of the return type's *name*. If you actually return an _IdentifiableDictionary<String, Array<Int>> then that's the *actual* return type of your function, not 'any Identifiable.' Thus your code can only return instances of that one type. They can't optionally return one of two distinct types, each conforming to the same protocol. Thus, in SwiftUI you'll implement a body property that returns some View, but you'll *actually* return a Text, or an Image, etc.

In SwiftUI, the result of the body property says that it will always return a single particular type, but it will at least conform to the View protocol. In ContentView, the compiler infers from the body implementation that the type returned is actually a Text instance, so the compiler will create storage enough to return that. Any code that calls this method will receive a thing that implements everything in the View protocol and takes up the amount of storage needed for a Text instance. This avoids the *existential type* wrappers around a pure-protocol instance—which quickly become expensive for any types larger than 24 bytes.

This has a secondary effect of ensuring that only a single type can be returned from a method, even though your API simply indicates that it returns 'some sort of View'. To see what this means, replace the contents of body with the following:

```
if (Bool.random()) {
    return Text("Hello World!")
} else {
    return Text("Hello SwiftUI!")
}
```

This works fine, and the canvas displays one of the two text items at random. No matter which path the execution takes, the result will be the same size. Now, replace the content of the else clause with the following:

```
return Image(systemName: "iCloud")
```

This code generates an error because the compiler detects that no single type is returned:

```
var body: some View {  Function declares an opaque return type, but the return statements in its body do not have matching underlying types
    if (Bool.random()) {
        return Text("Hello World!")                    1. Return statement has underlying type 'Text'
    } else {
        return Image(systemName: "icloud")             2. Return statement has underlying type 'Image'
    }
}
```

Since the Image and Text types aren't necessarily the same size, the compiler can't provide a single return type. Remove some from the property's declaration and attempt to build the project—maybe you can return an existential 'property-only' type. Notice you get an error:

```
Protocol 'View' can only be used as a generic constraint because it has
Self or associated type requirements
```

You've seen this one before, on page 6. It means that a return type of View alone doesn't provide enough type information to the compiler. There may be methods that are available only if the instantiated view (the Self type) conforms to Equatable, for instance. Perhaps it contains a member variable of some associated type, and without knowing that type, the compiler doesn't know how much memory it will take up. Opaque types solve this problem, allowing you to specify that while you (as the programmer) only care about conformance to View, the compiler can actually see a real type, in this case Text, and can handle it appropriately. The compiler knows how big the return type is, and can signal that size to the caller.

Now revert the method back to its original format: use some View for the return type, and replace the Image with a Text. The errors should then disappear.

View Modifiers

In the world of AppKit and UIKit, adjusting a view is a matter of creating your view and then setting values for various properties. The following code likely looks familiar:

```
let label = UILabel()
label.text = "Hello World!"
label.textColor = .green
label.font = UIFont.preferredFont(forTextStyle: .largeTitle)
```

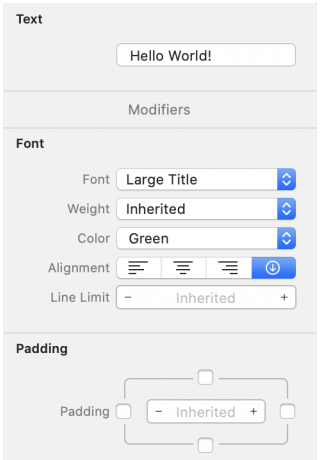
In SwiftUI, things happen differently. One of the benefits of using value types is their innate immutability, which allows for much safer code. To keep that benefit, rather than using a mutable View instance and changing its contents, you create an immutable one and call a function to obtain a new immutable view reflecting your changes. For further modification, you call a function on

that view to receive another, and so on. This is known as *chaining*, and it makes for some nicely-readable code:

```
Text("Hello World!")
    .foregroundColor(.green)
    .font(.largeTitle)
```

Make those changes to the Text view in your editor, and you'll see the changes reflected immediately on the canvas.

⌘-click on the view in the canvas, and you're presented with a pop-up menu containing numerous options. You'll explore these in full later, but for now, select the first option, Inspect..., to reveal a floating inspector palette for the view:



Here, you can change the value of the displayed text, adjust its font, weight, alignment, padding, and more. Scrolling down, you'll see a Foreground Color section containing a pop-up menu. Use this menu to select a new color. Your canvas will update to show the new color, and in the editor, you'll see that the code was altered to reflect the change. In SwiftUI, the code is the source of truth; .xib files aren't used to create and layout views now.

The inspector is also available directly from the code editor. ⌘-click on the Text initializer to see a similar pop-up menu, and again, select Inspect... to see the inspector. This time, change the font to Headline and note how the code and canvas both update to reflect the change. Open the inspector again, and this time set the font to Inherited. Your canvas updates to use the default font, and in the editor, the .font(.headline) line is removed. Do the same for the Foreground color.

Lastly, ⌘-click on the Text view and select Embed in VStack from the popup menu. Your body implementation should now look something like this:

```
VStack {
    Text("Hello World!")
        .foregroundColor(.purple)
}
```

The body property now returns a VStack view, which itself contains a Text view. The contents of the stack view are provided by a block, and in this case, it's a rather special block: a ViewBuilder. A ViewBuilder is a type of *function builder*, another new feature in Swift 5.1 that's put to heavy use by SwiftUI to provide its declarative syntax. This is what allows us to simply list out the contents of a stack view without using functions like append(view) all over the place:

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

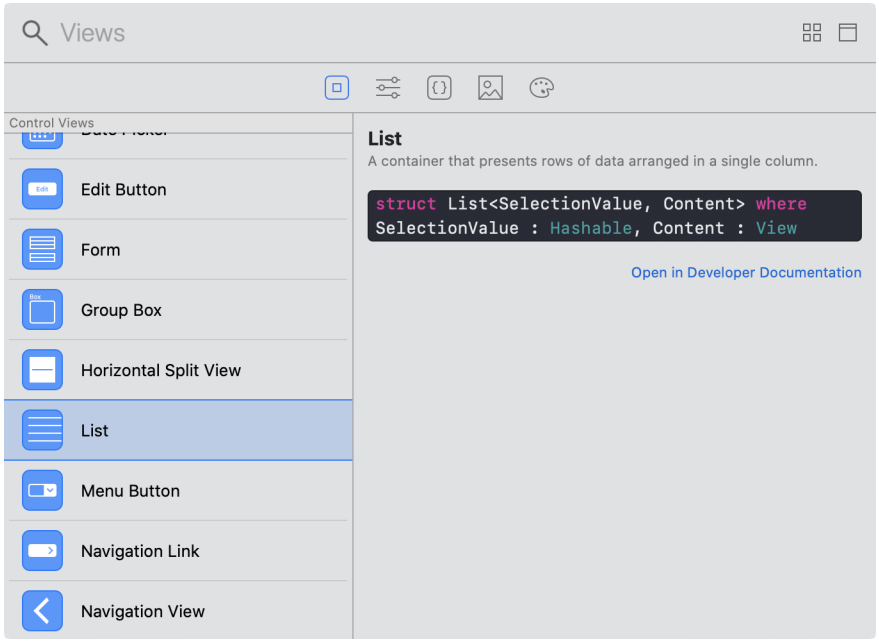
This works with a new attribute, @functionBuilder, which enables the creation of types that convert a series of inputs into a single output. The type of the block passed to the VStack in the example above has the @ViewBuilder attribute attached, meaning that an instance of the ViewBuilder function builder will be instantiated. The compiler will then collect the results of all expressions within the block which weren't otherwise assigned and will pass all those values into the builder, and the builder will, in turn, examine each one and return them as an array to the VStack initializer.

Managing Long View Lists

Function builders work by converting each statement within their braces into a parameter to a single function. Because of this, if there are 20 statements, the creator of the function builder needs to provide a function taking 20 parameters. In the case of ViewBuilder, implementations are provided for up to 10 parameters at once. If you want more subviews, you'll need to break up your content a little by assembling them into Group views, each containing ten or less subviews.

Drag & Drop Modification

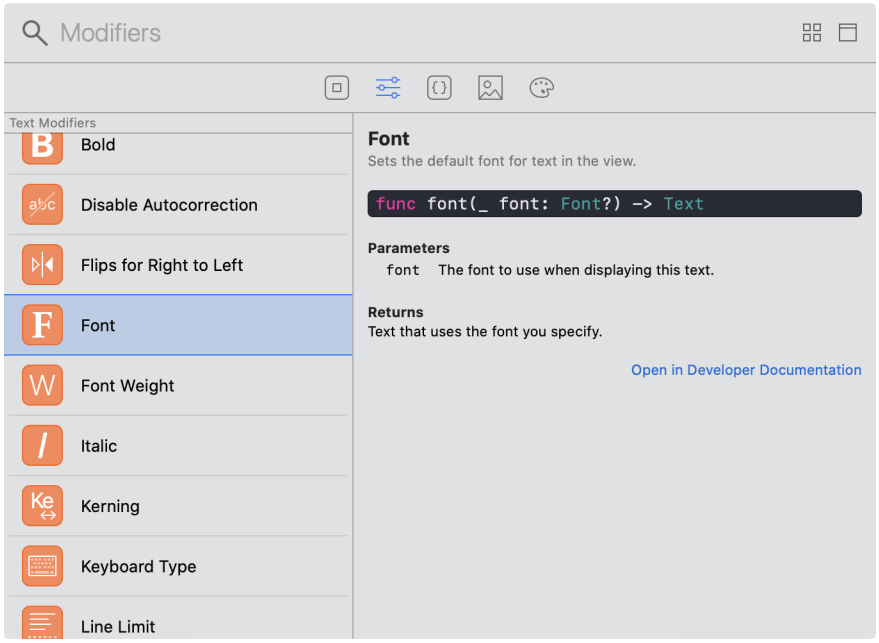
In the Xcode toolbar, click the + button to view Xcode's *Object Palette*:



If you're familiar with Interface Builder, this will look similar, but since you're working with SwiftUI, the first two tabs are different. The leftmost tab, selected by default, contains views. Double-clicking one of the views listed there will insert it at the cursor position. You can also drag views onto the canvas or the code editor to place them precisely where needed.

Drag out a new `Text` view, and place it below the existing one inside the `VStack`. If you drag it onto the canvas, a message will appear describing what the operation will do. By default, it creates a new `VStack` containing both the new item and the existing view (already a `VStack`). Dragging it into the editor will cause an empty line to appear, ready for you to drop it into place.

You now have a pair of `Text` views; let's modify their content and appearance a little. First, replace the placeholder text with `"Greetings from SwiftUI"`. Then, open the palette again, and this time, select the second tab to reveal a list of available modifiers.



Modifiers in SwiftUI are conceptually similar to properties. A modifier to a Text instance might set its font, or make its text bold. It might set a specific size for a view, or a background color, or more. Rather than modifying the content of a view class, a modifier on a SwiftUI view will yield a new view value with different properties all encoded—and all immutable, and thus thread-safe.

Scroll down to find the Font modifier, and drag one into the editor. Hover between the two Text declarations and an empty line will appear. Drop the font here, and it's appended to the first Text view. Select the default value of .title and replace it with .headline. Repeat the process, this time dropping the font onto the second text field on the canvas. Again, the editor content is changed to add a call to .font(...). This time, replace the font with .subheadline.

Working with Layout & Composition

You now have the beginnings of a nice view. Let's add some more items and adjust the layout to finish off with something that looks like this:

Currently, you have the pair of Text views vertically stacked on top of one another. To lay items to the left and right, you'll need a horizontal stack as

well. You can obtain one by wrapping the existing VStack in an HStack, either by editing the source code to to put HStack { ... } around the existing code, or by ⌘-clicking the VStack and selecting Embed in HStack.

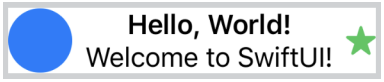
Open the views palette and drag a Circle into the HStack, above the VStack. ⌘-click on the circle you just added and set its width and height to 40. Now change its color to blue by appending .foregroundColor(.blue) in the editor:

```
Circle()
    .frame(width: 40, height: 40)
    .foregroundColor(.blue)
```

Following the VStack, add an image using one of the built-in *SF Symbols* values, and set its color to green:

```
Image(systemName: "star.fill")
    .foregroundColor(.green)
```

At this point, everything is squashed together in the center of the canvas:



To add some space between the text and the star image, SwiftUI provides the Spacer view. Drag one from the palette between the VStack and Image, or simply type Spacer() in the editor on a new line above the Image initializer.

This almost looks right, except for two things. First, the two lines of text appear to be centered with respect to one another. Second, the circle and the star are both tight against the edges of the screen on the canvas, which doesn't look very pleasing:

To solve the first problem, you need to tell the VStack how to horizontally align its contents. By default it uses .center, but you can easily change that either via the inspector or by editing the call to its initializer directly, like so:

```
VStack(alignment: .leading) {
    «content»
}
```

The two text fields are now left-aligned next to the circle. The only remaining step is to pull in your view's content from the edges of the screen. To do this, you can append .padding(.horizontal) to the HStack's declaration, after the closing brace. Alternatively, you can use the inspector, selecting the left and right

checkboxes in the Padding section, which will insert the relevant code for you.

Now, the view should look correct, and your code should look something like this:

```
HStack {
    Circle()
        .frame(width: 40.0, height: 40.0)
        .foregroundColor(.blue)

    VStack(alignment: .leading) {
        Text("Hello World!")
            .font(.headline)
        Text("Greetings from SwiftUI")
            .font(.subheadline)
    }

    Spacer()

    Image(systemName: "star.fill")
        .foregroundColor(.green)
}
.padding(.horizontal)
```

Handling Data Presentation

Your applications won't be creating user interfaces from whole cloth. You'll be working with some sort of data, and your user interface will present that data to your users. For this application, you're going to use to-do items, which you'll initially present in a list, then later in a detail view and an editor.

In the downloadable code bundle for this book[1] is a folder name Model; add this to your project, being sure to check Copy items if needed and Create groups. Inside the imported folder are four files, which you'll use throughout this book; in this chapter, you'll only need to think about two of them: TodoItem.swift and StaticData.swift.

Open TodoItem.swift to see two types defined there: TodoItem and TodoItemList. You'll use the list type in the next chapter, so for now, take a look at TodoItem; its interface looks something like this:

```
struct TodoItem: Codable, Identifiable, Hashable {
    var id: UUID
    var title: String
    var priority: Priority
    var notes: String?
    var date: Date?
```

1. https://pragprog.com/titles/jdswiftui/source_code

```
    var listID: UUID
    var completed: Date?

    enum Priority {
        « ... »
    }

    var complete: Bool {
        « ... »
    }
}
```

The structure is very simple, and you'll be working with it over the next few chapters. It contains a UUID (Universally Unique IDentifier) as an identifier, a title, and an enumeration representing its priority. Optionally, it may have associated notes a due date, and the date on which it was completed. Lastly, it contains the UUID of the TodoItemList that contains this item.

The definition takes advantage of some automatically synthesized conformances here, as well. First, since all of the properties are simple types, it gets free Codable support. Second, by conforming the Priority enumeration to the CaseIterable protocol, the compiler synthesizes an allCases static property, which will return a list of all the enumeration's values, in the order they appear in the source code. This is particulatly helpful in a UI application if you want to display a list of available priorities, for example.

Some sample data is also provided in the book's code archive, in the sample-data folder. Drag the file todo-items.json from there into your project; this might also be a good opportunity to create a separate "Resources" group to contain this and the other assets from the application, though see Info.plist Trouble, on page 15 for some important information when you do.

The JSON format of a to-do item looks like this:

```
"id": "63A3B756-BCBC-4EA3-8A35-A52462B24604",
"title": "Complete SwiftUI book sample",
"priority": "high",
"notes": "Use parts of the initial setup tutorial, to demonstrate how I plan to introduce and exp
"date": "2019-08-03T16:30:00-0500",
"listID": "B930B8BB-3804-440D-89ED-6F3E4DDCE22A",
"completed": "2019-08-05T12:14:51-0800"
```

Info.plist Trouble

In general, any file referenced in Xcodes project navigator can be moved around within that navigator and everything will work out Just Fine™. Alas, for one particular file that's not true: Info.plist, while it's handled normally as a resource file, is also ref-

erenced directly by a build setting—which includes the file's path. By moving it into a sub-folder, you'll break that build setting, and thus the build.

There are two ways around this:

1. Define the Resources group using New Group without Folder (^⌥⌘N); the group will exist in the project navigator, but the contents on disk won't move.
2. Select the Do It project in the project navigator, then the Build Settings tab, then edit the value of the Info.plist File setting to include the new folder (e.g. Do It/Resources/Info.plist).

Which approach you use is entirely up to you; Xcode will perform identically either way.

The initial view for the app will show a list of to-do items. In UIKit, you'd use a UITableView, UICollectionView, or a UIStackView to accomplish this. However, in SwiftUI's concept-based nomenclature, you have the List view, which on iOS can create views analogous to a UITableView.

Start by creating a new group named Views, and within there create a new SwiftUI View file, naming it TodoList.swift. Open this file and resume the canvas to see the familiar 'Hello World' template code. Wrap the Text declaration in a call to List, like so:

```
List {
    Text("Hello World!")
}
```

The canvas now displays a familiar table view containing a single row with some text and a number of empty cells. You can repeat this cell by passing a sequence to the List initializer. The block is then invoked once for each element in the sequence, with that element passed as a value. You can use a Range to tell the list to generate multiple rows. Change your code to match the following:

```
List(0..<5) { num in
    Text("This is row \(num)")
}
```

Your canvas should now display five rows, each announcing their row number.

This, as it turns out, is everything you need to display the to-do items. All that's left to do is pass in the todoItems list defined in StaticData.swift to the List initializer and reference each item in the supplied block:

```
List(todoItems) { item in
```

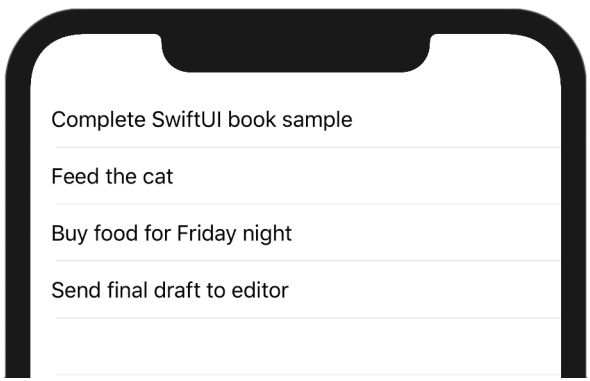
```
    Text(item.title)
}
```

Verbatim vs. Localized Strings

In AppKit and UIKit, strings were just strings. If you wanted to localize the title of a button, you had to call NSLocalizedString() to look up a localized version of that string for the user's current locale. When developing, though, that's a lot of extra typing, and it's not uncommon for developers to skip the call when starting their app or when trying things out. However, as the project gets larger, it becomes more and more difficult to remember where you need to go back and insert those localization calls, and inevitably, something slips through the net.

SwiftUI takes the opposite approach. Any inline strings (e.g. "Hello") passed to the Text initializer are automatically considered localizable. What's more, it automatically converts from Swift's string-interpolation format to the numbered-parameter format used by string localization (i.e. "Hello \(name), the current time is \(time)." becomes "Hello %$1, the current time is %$2."). String *variables* (e.g. self.name) are presented verbatim automatically. When you know a static string should not be a candidate for localization, use the Text.init(verbatim:) initializer to skip all that; this should be the default for any text that comes from your user's data. Conversely, when you want a string variable to be localized, use it to initialize a LocalizedStringKey, e.g. as Text(LocalizedStringKey(str)).

You now have a basic list of to-do items.



It's worth looking under the hood a little, though, to answer a pertinent question: how does SwiftUI know to associate a given item with a given row in the list?

The answer is in the Identifiable protocol, to which TodoItem conforms. This simple protocol has only one requirement: a property named id whose type conforms to Hashable (and thus Equatable):

```
public protocol Identifiable {
```

```
    /// A type representing the stable identity of the entity associated
    /// with `self`.
    associatedtype ID : Hashable

    /// The stable identity of the entity associated with `self`.
    var id: Self.ID { get }
}
```

If you're working with a type that doesn't conform to Identifiable, though, you can always supply a suitable identifier as a key-path; for enumerations or Strings, for example, you'd use \.self like so:

```
List(["Hello", "World"], id: \.self) { « ... » }
```

Let's make some small changes to give the view some more flavor. First, set the title's font to .headline by appending .font(.headline). Now, wrap it in a VStack(alignment: .leading). Lastly, inside the vertical stack block, check if the item's notes are non-nil, and if so, declare another Text containing those notes, using the .subheadline font. Your list content should now look like this:

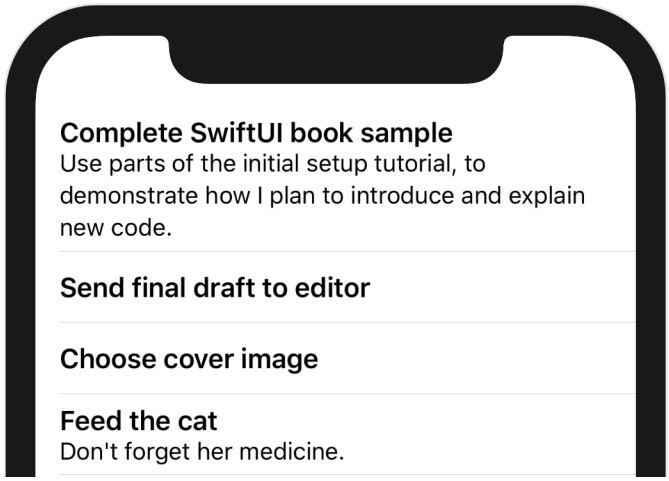
```
VStack(alignment: .leading) {
    Text(item.title)
        .font(.headline)

    if item.notes != nil {
        Text(item.notes!)
            .font(.subheadline)
    }
}
```

Author's Note

At present the allowed syntax for builder blocks does not include the if let construct, which is why you can't use a more idiomatic if let notes = item.notes. Hopefully this can be rectified with a future compiler update, but until then this check-and-explicitly-unwrap is the only straightforward option within a @ViewBuilder block.

Each row contains a nicely presented title and potentially the notes as well:



Adding Navigation

Now that you have the items displaying nicely, you want to be able to interact with them. The normal way of doing this is to place the list within a *navigation view* and to make each row tappable. Tapping on the row then presents a new view via the parent navigation view.

First, let's make the rows interactive. In AppKit or UIKit, you'd likely set a property on the view to mark it as interactive, and then you'd add a tap gesture recognizer, you'd set a target and action on the recognizer, and so on. Or, if you're using a UITableView or similar, you'd implement a method in the UITableViewDelegate that gets called when the user taps on a row. In SwiftUI, things happen differently.

As you've seen, SwiftUI prefers to describe things in a more abstract sense. Thus, for navigating, rather than setting some values on some view, you tell SwiftUI to use a NavigationLink. This view uses a view-builder block to define its content, so it can simply wrap the VStack you're already using. It exists for the simple purpose of responding to single taps by pushing some new View onto a navigation stack. It also, depending on the platform, includes some extra subviews that indicate its nature, such as the trailing-edge chevron image used by navigable UITableView rows.

Wrap your VStack in a new NavigationLink, providing a simple Text view as its destination:

```
List(todoItems) { item in
    NavigationLink(destination: Text(item.title)) {
        VStack {
            «content»
```

```
        }
    }
}
```

Your canvas will immediately reflect the change:

The trailing-edge chevron was added to each row, but for some reason the text appears to be using a more muted gray color. Launch a live preview from the canvas by clicking the play button on the lower right, then try clicking on the rows.

Nothing's happening, it seems. The reason for this is tied to the sudden color change on the text. All the NavigationLink views are in their disabled state. Glancing at the code, can you see why?

You're missing a NavigationView. The NavigationLink view must be inside a navigation view in order to function. Not finding one, it automatically adjusts its display to indicate that it's disabled and will not function. This is easy enough to remedy. Wrap the List in a new NavigationView:

```
NavigationView {
    List(todoItems) { item in
        «…»
    }
}
```

With that code, your preview comes alive. The text is in the correct color, and clicking on the rows in the live preview pushes a new view containing the item's name. You can navigate in all the usual ways, whether with buttons or by swiping backward and forward.

There's a large space at the top of the list view now, though, and the back button doesn't look very interesting. Normally you'd set a title on a view for the navigation view to use; the same is true in SwiftUI, though naturally rather than setting a property, you'll be chaining a call to your List view's declaration (the List is the root view displayed by the NavigationView). After the list's closing brace, append the text .navigationBarTitle("To-Do Items"). Your list now has a title, and when you click on a row in the live preview, you'll see that the title animates into the place of the back button in the expected manner, and animates back to title position when you back out to the list again.

It's time to clean up the UI a little by styling the list view. By default, the list uses a plain list style, described by the PlainListStyle class, but you can change this by chaining another call. Use .listStyle(GroupedListStyle()) to modify your list's appearance, and you'll get a more muted background behind the title and status bar, and any blank rows disappear from the list.

At this point, your body implementation should look like this:

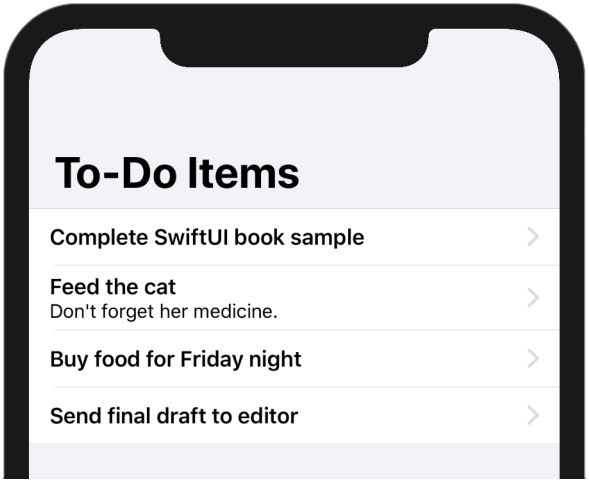
1-LayoutPresentation/Do It/Views/TodoList.swift

```
    var body: some View {
        NavigationView {
            List(defaultTodoItems) { item in
                NavigationLink(destination: Text(item.title)) {
                    VStack(alignment: .leading) {
                        Text(item.title)
                            .font(.headline)

                        if item.notes != nil {
                            Text(item.notes!)
                                .font(.subheadline)
                        }
                    }
                }
            }
            .navigationBarTitle("To-Do Items")
            .listStyle(GroupedListStyle())
        }
    }
}

struct TodoList_Previews: PreviewProvider {
    static var previews: some View {
        TodoList()
    }
}
```

The canvas should look similar to this:



Not bad for a dozen or so lines of code, I think!

What You Learned

Writing user interface code with SwiftUI is quite different than using UIKit or AppKit. By now, you should be familiar with the new building blocks that you'll be using going forward:

- The Xcode canvas and inspector, for visual feedback on your UI.
- Xcode's palette, allowing for drag-and-drop interface design.
- Modifier methods and immutable value types, rather than shared reference types with world-mutable properties.
- A clean, terse programming syntax for quickly sketching and prototyping your interfaces.

In the next chapter, you'll flesh out the application a little more by assembling a detail view for your to-do items. Along the way, you'll learn about SwiftUI's layout system, and you'll get a feel for just how much functionality SwiftUI gives you for free.

CHAPTER 2

Application Data in SwiftUI

Story Map

Why do I want to read this?

To see how you can use SwiftUI to present and interact with hierarchical data models.

What will I learn?

How data is handled within the SwiftUI ecosystem, and how changes to that data are used to update your user interface.

What will I be able to do that I couldn't do before?

You will now have a greater command of the major building blocks of user interfaces in SwiftUI. You'll be able to take a design and build it declaratively in code, and you'll be able to respond to changes in the underlying data model.

Where are we going next, and how does this fit in?

In the next chapter, you'll implement editing functionality for your application, and learn how SwiftUI manages mutable data.

In the last chapter, you created a simple to-do list application in a few lines of code. Now that you have some familiarity with the canvas and inspector, and know how to use view modifiers, it's time to look at two more involved tasks. First, you'll handle user input, making adjustments to your user interface to match. Second, you'll compose a more detailed view using stacks and more. The data model has been updated slightly, adding support for lists with icons and associated colors, letting you create a simple detail view that really *pops*. Along the way, you'll encounter some of the vagaries of SwiftUI's data and layout flow, and learn how best to deal with them.

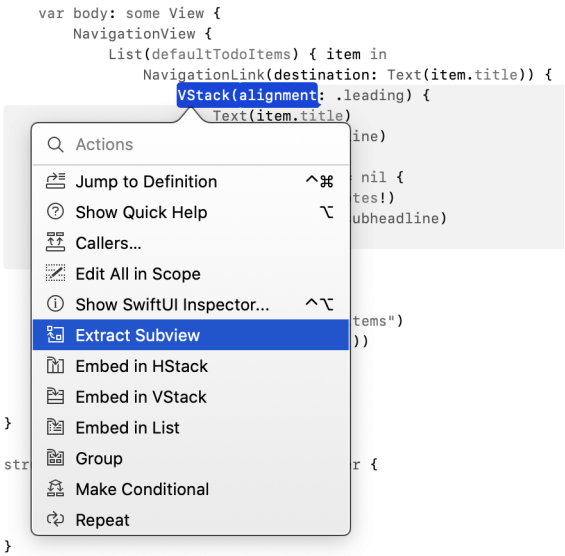
This chapter builds on the code developed during chapter 1, but makes use of some additional helpers. You can follow along by using the starter project for this chapter, which can be found in the source code download[1] in the folder 2-ApplicationData/starter. Code for the completed chapter can be found in 2-ApplicationData/final.

Interaction in Lists

Now that your data model has some extra information, let's present it to the user. You'll add a button in your item rows to toggle that item's completion state, and you'll color the button according to the color of the item's list. First, though, the code for your list view has rather a lot of levels of indentation. Let's fix that by factoring out the row content itself.

Refactoring

Open TodoList.swift and locate the VStack section inside the NavigationLink() block. ⌘-click on the VStack constructor and select Extract Subview:



This will add a new struct at the bottom of the current file and let you choose its name—use TodoItemRow. You should end up with the following definition:

```
struct TodoItemRow: View {
    var body: some View {
        VStack(alignment: .leading) {
            Text(item.title)
```

1. https://pragprog.com/titles/jdswiftui/source_code

```
                .font(.headline)
            if item.notes != nil {
                Text(item.notes!)
                    .font(.subheadline)
            }
        }
    }
}
```

The compiler is likely alerting you about the item variable used to fill out the row's content; you need to pass that into this new view, which means you'll need somewhere to store it. Add a property for this to the:

```
let item: TodoItem
```

You don't need a var here since there's no plan to modify the data. You just need an immutable copy of the data from which you'll read the details you need to provide the view when the framework requests it.

Now the editor will note that you need to pass an argument when creating the row from within your TodoList. Use the fix-it suggestion to pass in the current item:

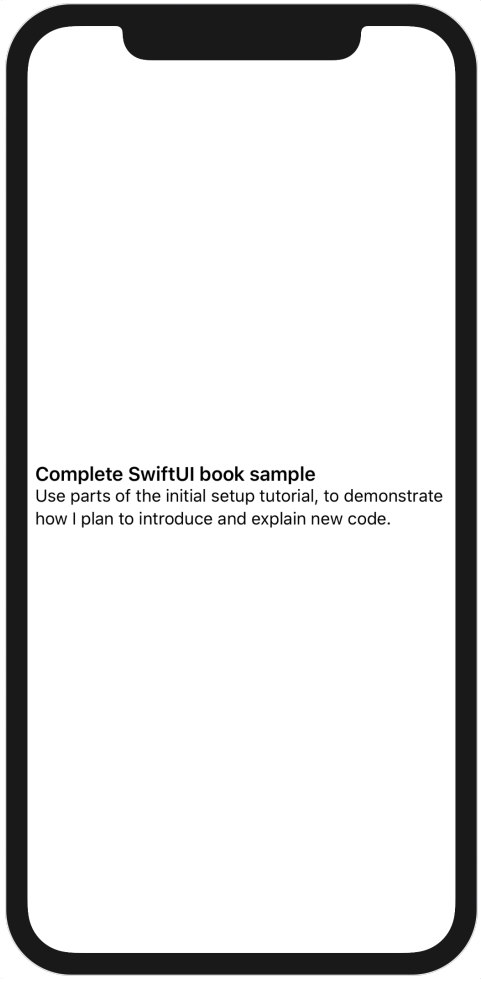
```
NavigationLink(destination: Text(item.title)) {
    TodoItemRow(item: item)
}
```

Let's neaten up the source code a little at this point. Select the entire TodoItemRow type and cut it from the document (⌘-X). Now create a new SwiftUI View file inside the Views group, named TodoItemRow.swift. Finally, replace the TodoItemRow definition in that file with the one you just cut out of TodoList.swift.

The editor will draw your attention to the preview provider at the bottom of the file. The TodoItemRow() initializer call requires a TodoItem parameter. Replace the offending line with:

```
TodoItemRow(item: defaultTodoItems[0])
```

Now you can launch the preview, and you'll see a two-line list row rendered on the canvas. Alas, it's centered in a screenful of white:



To remedy that, you'll need a customized preview layout. You do this with the previewLayout() modifier. This modifier takes a single argument of type PreviewLayout, which comes in three flavors:

- .device. This is the default, and it centers the preview within a container matching the size of the device being used on the canvas.
- .fixed(width:height:). Centers the preview in a container view with the specified dimensions.
- .sizeThatFits. Offers the preview the size of the current preview device, then fits the container to the size chosen by the preview. This will allow it to fit itself into any size that will fit on the screen of the current device, but no larger.

For this example, the sizeThatFits layout works best, because it provides a flexible width that will max out at the width of the device's display. You'll see how to alter the device used by the canvas in Using Device Previews, on page 144, but for now just append some padding with the padding() modifier then select your chosen layout.

Completion

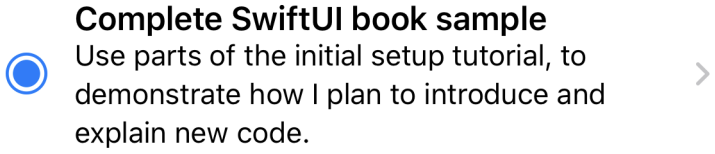
A real to-do item row would have one attribute that is currently lacking here: a button to mark the item completed. Let's follow the example of the built-in Reminders application and use a button whose icon is either an empty or a filled circle, depending on whether the item is complete.

First, you need to embed the current VStack inside an HStack, and then you can add the button at the front of the new stack:

2-ApplicationData/final/Do It/Views/TodoItemRow.swift

```
HStack {
    Button(action: {
        // « ... »
    }) {
        Image(systemName: item.complete
                ? "largecircle.fill.circle"
                : "circle")
            .imageScale(.large)
            .foregroundColor(.accentColor)
    }
    .padding(.trailing, 6)

    VStack(alignment: .leading) {
        // « ... »
    }
}
```



The button's action needs a little more work. The infrastructure to edit the item isn't yet in place, so instead, you'll pop up an alert when the button is clicked, to prove that it's working. The first part of this is to add a new Bool property to the TodoItemRow with a default value of false, marked with the @State attribute.

State Variables

The row needs to keep track of its item's completion state in order to update its appearance when it changes. SwiftUI provides support for this through a *property wrapper* named `@State`.

Swift has a few special keywords or attributes that can be applied to properties to induce certain behavior. For example, the `lazy` keyword causes a property to be initialized lazily—i.e., only when first requested. The `@NSCopying` attribute for properties of an Objective-C type will cause that type to be copied rather than retained during the assignment to the property.

Swift 5.1 democratizes these features by allowing the programmer to specify their own wrapper types for properties. These are classes or structures marked with the `@propertyWrapper` attribute and which obey certain rules. The compiler then allows the use of these as attributes on property declarations, and silently uses the wrapper type under the hood.

You'll see this a lot in SwiftUI. It's used there to define state variables (via the `@State` wrapper attribute) and bindings (`@Binding`) and more. It's also used to great effect in the *Combine* framework, which you'll encounter later in the book as you deal with more complex data flows.

When you apply the `@State` attribute to a property in a view, SwiftUI will detect when its value changes and automatically request that the view update itself by re-fetching its `body` property.

Use the button's `action` to toggle its value:

2-ApplicationData/final/Do It/Views/TodoItemRow.swift

```
@State var showAlert = false

var body: some View {
    HStack {
        Button(action: {
            self.showAlert.toggle()
        }) {
            // « ... »
        }
        .padding(.trailing, 6)
        // « ... »
    }
}
```

To make an alert appear on screen, you use the `.alert(isPresented:content:)` view modifier. The first argument takes a *binding* to a boolean property, and the second a block that returns an `Alert` instance. For the former, you'll pass `$showAlert`; you'll learn more about this in Bindings, on page 44 and Dependen-

cy Propagation, on page 60. For the latter, a simple alert with a title, a message, and a button to dismiss it:

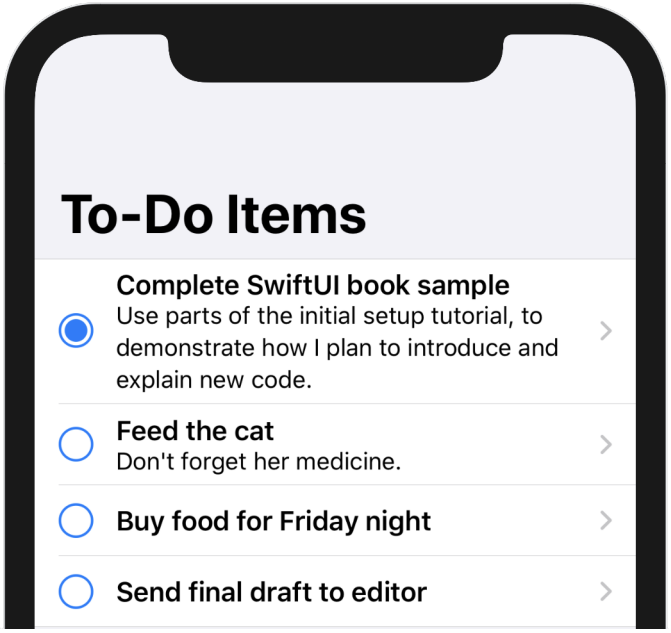
2-ApplicationData/final/Do It/Views/TodoItemRow.swift

```
Button(action: {
    // « ... »
}) {
    // « ... »
}
.padding(.trailing, 6)
.alert(isPresented: $showAlert) {
    Alert(title: Text("Complete!"),
          message: Text("This will work soon, honest."),
          dismissButton: .default(Text("OK")))
}
```

This tells SwiftUI to monitor the value of the showAlert property, and to present the alert as long as its value is true. Additionally, SwiftUI will reset the value to false when the alert is dismissed. Toggling the value to false will also cause the alert to be dismissed, should you need to do so programmatically.

Launch a live preview in the canvas and click on the button to see your alert pop up. Dismiss it and click again, and it will reappear. The button works. In later chapters, you'll hook this up to the data model directly.

Now return to TodoList.swift and fire up a live preview or launch the app in the Simulator, and try out all the functionality—everything that worked before still works now.



Unfortunately, not everything you've *added* is functional. The completion buttons aren't doing anything, even though no problems showed up when testing the item row view alone. Instead, tapping on the button merely activates the row and its navigation link, and that's definitely not what you wanted to happen; marking a task as complete or incomplete is important enough that it should be a simple one-tap task from the list. So, what's happening here?

To understand what's going on, you need to look at SwiftUI's input mechanism on iOS, the `Gesture`.

Handling User Input

In SwiftUI, you respond to input—whether directly or indirectly—through a `Gesture` of some kind. `Gesture` itself is a protocol, much of its details internal, but it requires two or three things:

1. Some `Value` type that will change as the gesture progresses, if appropriate.
2. A means to register a callback that will fire when the gesture ends successfully, and any action should be triggered.
3. If its `Value` type conforms to `Equatable`, then it should allow a callback to be registered, which will be invoked each time the value changes.

With these base requirements, then, SwiftUI provides us with several gesture types you can create and use, including `TapGesture`, `MagnificationGesture`, `LongPressGesture`, `DragGesture`, and `RotateGesture`. Internally there are many more, and they're

used as the standard means of passing around event information in SwiftUI. One internal gesture, for example, handles the normal behavior of buttons, highlighting during touch-down, unhighlighting on touch-up, or if the finger is dragged out of the button's frame. Alas, that one isn't for us to use, but it gives us a hint as to where to look for the cause of our current issue.

Gestures are attached to the view using one of several modifiers:

- gesture(_:including:) attaches a gesture to a view, making it one of the candidates for receiving events.
- highPriorityGesture(_:including:) attaches a gesture to a view but raises its priority, meaning that it has a right of first refusal to any applicable events.
- simultaneousGesture(_:including:) attaches a gesture to a view, specifically enabling it to handle events along with another gesture—for example, magnify and rotate gestures might be paired to allow both actions at once.

There are also some modifiers that install common gestures automatically. onTapGesture(count:perform:) installs a TapGesture at normal precedence that performs the provided action after the requested number of taps is recognized. Similar modifiers exist for other gestures.

Additionally, gestures can be made *exclusive*, setting an order of precedence between two gestures; a long-press might take precedence over a tap. They can also be *sequenced*, letting us require one gesture to complete before another can be recognized: so that an item may be dragged, but only after a long-press on the item in question—a 'long press and drag' similar to the iPhone's home screen.

Each of the initial three methods above takes an additional parameter, including:. This is a GestureMask, which has several values: none, gesture, subviews, and all. Passing none disables all gesture handling on this view, completely: you might use it to conditionally make a button inactive, for example. The gesture option specifies that only the gesture installed by this modifier should be recognized, and all others should be ignored. Passing subviews does the opposite: use it to disable just the gesture being passed, allowing any others on subviews to continue functioning. Lastly, all implies both gesture and subviews, and is the default behavior when no value is given.

This is all beginning to paint a picture. It seems quite possible that the gesture used to trigger the list row and its associated navigation link is installed in one of two ways: either as a high-priority gesture or with a gesture mask of gesture, disabling any gestures attached to its content. Either of these would cause the buttons in the rows to become 'untouchable.'

It seems, then, that raising the priority of the button's gesture might help here. But how to do that? We can call the API directly while adding our own gesture handler, like so:

```
Button(action: { « perform action » }) {
    « define label »
}
.highPriorityGesture(
    TapGesture(count: 1)
        .onEnded { « perform action » }
)
```

That doesn't look exactly graceful, though—the action is being defined twice, and you can't easily predict which order they might be called in, if at all. There should be a better way—and happily there is. Let's take a look at how you can begin to customize some of SwiftUI's components by looking at button styles.

Button Styles

It just so happens that Button is one of the more customizable parts of the SwiftUI toolkit. Every button looks for a *style* component in its environment, which provides the details of the UI surrounding the provided label view. These styles are provided through the .buttonStyle() modifier, and several styles are provided by the system, including DefaultButtonStyle, BorderlessButtonStyle, and PlainButtonStyle. More useful, however, are the protocols these are based on, and which enable you to create your own button styles.

SwiftUI provides two protocols for defining button styles: ButtonStyle and PrimitiveButtonStyle. Both require the definition of a makeBody(configuration:) method which takes some configuration information and returns a view used to represent the button. For ButtonStyle, the configuration contains the Label view passed to the button's constructor along with a Bool value indicating whether the button is currently 'pressed.' Using this type, you can define your own style that changes based on whether the button is currently pressed—the default style on iOS reduces the opacity a little, but you can ultimately do anything here: change scale, color, and more. For the problem in hand, you need to look at more than the *appearance* of the button, you need to define its *interaction*. The comments on the ButtonStyle type point the way:

```
/// `Button` instances built using a `ButtonStyle` will use the standard
/// button interaction behavior (defined per-platform). To create a button
/// with custom interaction behavior, use `PrimitiveButtonStyle` instead.
```

Let's follow this suggestion and look at PrimitiveButtonStyle and its associated configuration type, PrimitiveButtonConfiguration. This type also has two properties,

the first of which is the Label view provided to the button. The second, this time, is a function named trigger(); calling this will invoke the button's action. This, then, leaves your style in control of everything else: you determine the appearance and how it changes, how the interaction will work, and when the action will fire. *This* is the power tool for the job: if you own the gesture processing, then you can give it a high priority.

Raising Button Priority

In Xcode's project navigator, create a new group named Accessories, and within that create a new Swift View file named HighPriorityButtonStyle and open it. Remove the contents of the HighPriorityButtonStyle type definition and replace View with PrimitiveButtonStyle in its declaration:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
struct HighPriorityButtonStyle: PrimitiveButtonStyle {
}
```

For now, replace the contents of the previews() property in the HighPriorityButtonStyle_Previews type with an EmptyView as well:

```
struct HighPriorityButtonStyle_Previews: PreviewProvider {
    static var previews: some View {
        EmptyView()
    }
}
```

To provide a style for the button, you need to implement the makeBody(configuration:) function to return some View type. Since you're managing the press state of the button, it's best to create a custom view just for this. Add the following private view type inside the HighPriorityButtonStyle type definition:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
private struct Inner: View {
    @State var pressed = false
    let configuration: PrimitiveButtonStyle.Configuration

    var body: some View {
        // « ... »
    }
}
```

Here the view has a property used to record the press state of the button, and it also holds a copy of the configuration passed into the button style, so that it has access to both the label and trigger(). The magic is going to happen inside this view's body implementation, where you'll create a gesture and install it onto the label using the .highPriorityGesture() modifier.

The standard button gesture is not available, as it's a private API (though if you're daring, try typing _ButtonGesture in Xcode and see what the autocompletion brings up…). Instead, let's approximate it with a DragGesture. Drag gestures have one main property, which is the minimum distance required before the gesture is recognized. In this case, that distance will be zero, so the gesture is recognized as soon as the button is touched. To approximate some of the button gesture's behavior, you'll disable the 'pressed' state if the touch is dragged more than a little distance away from its starting point. The translation(_:doesNotExceed:) method in Helpers/Geometry.swift will do this for you. Go to the body property of HighPriorityButton.Inner and create the drag gesture:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
let gesture = DragGesture(minimumDistance: 0)
    .map { translation($0.translation, doesNotExceed: 15) }
```

Here you've created a new DragGesture with a minimum drag distance of zero. You're then mapping its Value type into something new, using the .map(_:). DragGesture.Value is a fairly large type, containing the start time and the starting and current locations for the drag, along with various computed properties for things like the distance moved (translation) and the predicted ending location and ending translation for the drag (to implement e.g., a 'fling' gesture where an item keeps moving after the drag ends, based on its momentum). For this button, only the touch translation is useful—you want to see if it's moved a small or large distance since it started, and adjust the pressed state property accordingly. The map function uses translation(_:doesNotExceed:) to turn the drag's translation into a Bool value.

Next, you want to update the state when the gesture's value changes. This is done with the .onChanged(_:) modifier:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
.onChanged { self.pressed = $0 }
```

This is simple enough: if the mapped value is true, the button is pressed, otherwise it's not. Now you need only respond to the end of the gesture, when the user's finger is lifted from the screen, which is done via the .onEnded(_:) modifier:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
.onEnded { _ in
    guard self.pressed else { return }
    self.pressed = false
    self.configuration.trigger()
}
```

First of all, you check whether the button is considered 'pressed' at the moment. If not, you do nothing more. If it is, then you turn off the pressed state and trigger the button's action through the configuration.

Now all that remains is to implement the appearance. You'll use the provided label as-is, but will drop the opacity while pressed, similar to a normal button. Then the crucial part: install the drag gesture with a high priority using the .highPriorityGesture(_:) modifier. Your complete body should now look like the following:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
let gesture = DragGesture(minimumDistance: 0)
    .map { translation($0.translation, doesNotExceed: 15) }
    .onChanged { self.pressed = $0 }
    .onEnded { _ in
        guard self.pressed else { return }
        self.pressed = false
        self.configuration.trigger()
    }

return configuration.label
    .opacity(pressed ? 0.5 : 1.0)
    .highPriorityGesture(gesture)
```

With the button view implemented, all that remains is to use it within the HighPriorityButtonStyle itself, implementing makeBody(configuration:) like so:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
func makeBody(configuration: Configuration) -> some View {
    Inner(configuration: configuration)
}
```

Preview and Testing

Your code should now build successfully, but it's worth testing its efficacy in a preview. Use the following code in the HighPriorityButtonStyle_Previews type to create a navigation view containing a list with a navigation link containing a button, with the new button style set:

2-ApplicationData/final/Do It/Accessories/HighPriorityButtonStyle.swift

```
static var previews: some View {
    NavigationView {
        List {
            NavigationLink(destination: Text("Hello")) {
                Button(action: { print("hello") }) {
                    Text("Button!")
                        .foregroundColor(.accentColor)
                }
                .buttonStyle(HighPriorityButtonStyle())
            }
```

```
        }
    }
}
```

Refresh the canvas and start a Live Preview. Tapping on an empty section of the link row will highlight the row and trigger the link as usual, but tapping on the button itself will not—the button should dim while touched and respond to drags as you've designed. With this, you'll have what you need to make the completion button in TodoItemRow function property while in a list. Open TodoItemRow.swift and add the following modifier to the completion button:

```
Button(action: { « ... » }) {
    « ... »
}
.padding(.trailing, 6)
.buttonStyle(HighPriorityButtonStyle())
.alert(isPresented: $showAlert) {
    « ... »
}
```

Try the button within the todo list again—now it behaves as required, and you can tap it without activating the navigation link.

Nesting Data

In Model/TodoItem.swift, you'll notice that the data model includes a second type, the TodoItemList. This is what you'll use to group sets of to-do items together in lists. Each list has an associated color and icon, which will be used to provide some visual differentiation to its contents. Regardless of their effect on the appearance of the *item* lists, though, they need a representation of their own. In this section, you'll create a view to display these lists and update the TodoList view to operate upon the lists directly.

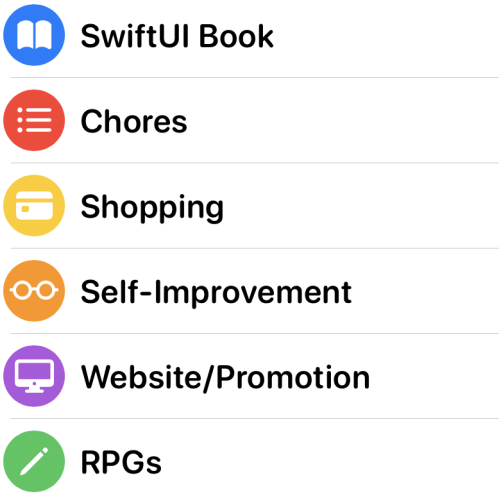
Start by creating a new SwiftUI View file inside the Views group, named Home.swift. Ignoring the body for the moment, add a private struct type inside Home named Row and conforming to the View protocol:

2-ApplicationData/final/Do It/Views/Home.swift

```
private struct Row: View {

    var body: some View {
        // « ... »
    }
}
```

The row is going to display two things: the list's icon and its title. The icon will be displayed in white, using a circle in the list's color as its background, looking like this:



This will require three properties, so add them to the `Row`:

2-ApplicationData/final/Do It/Views/Home.swift

```
var name: String
var icon: String
var color: Color
```

With these in place, you can put together the view's `body`. You'll need an `HStack` containing an `Image` and a `Text` view. The image will use the icon name to locate a *system icon*, namely a vector image from a special system font named *SF Symbols*.[2] You'll then use several modifiers to give it the required appearance. The title needs only a simple `Text` view.

Use the following code for the `Row` view's `body`:

2-ApplicationData/final/Do It/Views/Home.swift

```
Line 1  var body: some View {
2           HStack {
3               Image(systemName: icon)
4                   .foregroundColor(.white)
5                   .frame(width: 32, height: 32)
6                   .background(color)
7                   .clipShape(Circle())
8               Text(name)
9           }
10      }
```

On line 3, the system icon is obtained through the `Image(systemName:)` initializer. It's then given a white foreground color, and on line 5 it gets a fixed-size frame.

2. https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/

This helps because the different icons have slightly different dimensions—similar to letters in a variably-spaced font. Giving the image a fixed size with the .frame() modifier ensures that every row will line up, regardless the intrinsic sizes of the icons themselves. To obtain the circular background, first the background(_:) modifier is used to provide a color fill, then the .clipShape(_:) modifier, on line 7 is used to *clip* the content of the resulting view. *Clipping* means that any pixels outside the provided shape are not drawn—so providing a Circle clipping shape means that only the parts of the image and background within the circle are drawn, leaving a circular colored background.

The row view is now complete: let's put it to use. The list chooser itself will use a List view to display the available options. Since it will be the new root view for the application, this will be placed inside a NavigationView. There are two types of lists you would like to show, as well: the individual lists, showing only the items they contain, and the list of all items, regardless of their container. That split lends itself to a grouped list with two sections: one for the lists, one for "All Items." The fonts used within the list can be a little different as well, using a rounded appearance rather than the flat beveled edges of the system font's regular appearance.

Use the following code to lay the groundwork for the list:

2-ApplicationData/final/Do It/Views/Home.swift

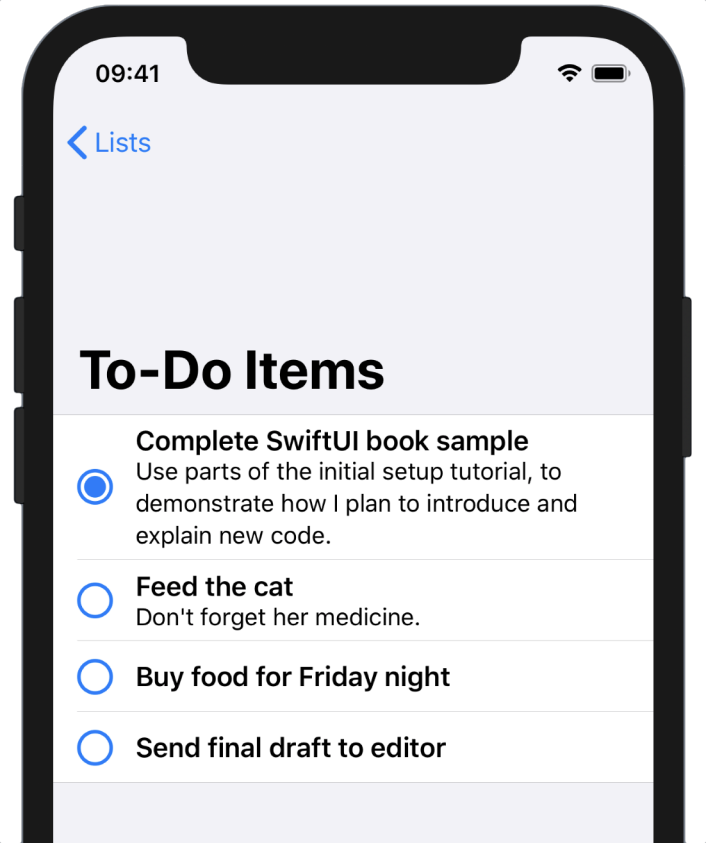
```
NavigationView {
    List {
        // « ... »
    }
    .font(.system(.headline, design: .rounded))
    .listStyle(GroupedListStyle())
    .navigationBarTitle("Lists")
}
```

The "All Items" section will appear at the top, so that's the first section to add. You create sections within List views using the Section view. This is passed a ViewBuilder block that defines the contents of the section. Here you only have a single row inside a navigation link, so the implementation is straightforward:

2-ApplicationData/final/Do It/Views/Home.swift

```
Section {
    NavigationLink(destination: TodoList()) {
        Row(name: "All Items",
            icon: "list.bullet",
            color: .gray)
    }
}
```

If you launch a live preview now and click on the row, then the familiar item list will appear, although it will look a little strange, with a giant navigation bar:



This appears because the `TodoList` view still contains its own `NavigationView`; now that there's a new root view in the navigation hierarchy, that's no longer needed. However, thinking ahead a little, there's more that needs to be changed inside `TodoList`: at present, it shows all to-do items, while you now want it to potentially show only the items within a certain list. That will need to be fixed before you can proceed.

Open `TodoList.swift`, locate the `body` implementation, and remove the `NavigationView`, leaving the `List` as the top-level view. At present, the list is iterating over `todoItems`, the global collection of all to-do items, but that needs to change. You'll potentially want to reference a single list, instead, and that means you'll need a property to hold that list. Along with that, you can add computed properties to return the view's title—either that of the list or "All Items"—and the items to display. Color-coding will depend on the type of content being

shown, as well. When showing a single list, only that list's color will be used. For "All Items," however, each item should use the color of its associated list (it's why the color is there, after all).

Add the following code above the body:

2-ApplicationData/final/Do It/Views/TodoList.swift

```
var list: TodoItemList? = nil
var items: [TodoItem] { list?.allItems ?? defaultTodoItems }
var title: String { list?.name ?? "All Items" }

func color(for item: TodoItem) -> Color {
    let list = self.list ?? item.list
    return list.color.uiColor
}
```

Temporary Code

The methods used here are defined in Helpers/StaticListAPI.swift, and are only there to keep things simple for this chapter. In the next chapter they will be replaced.

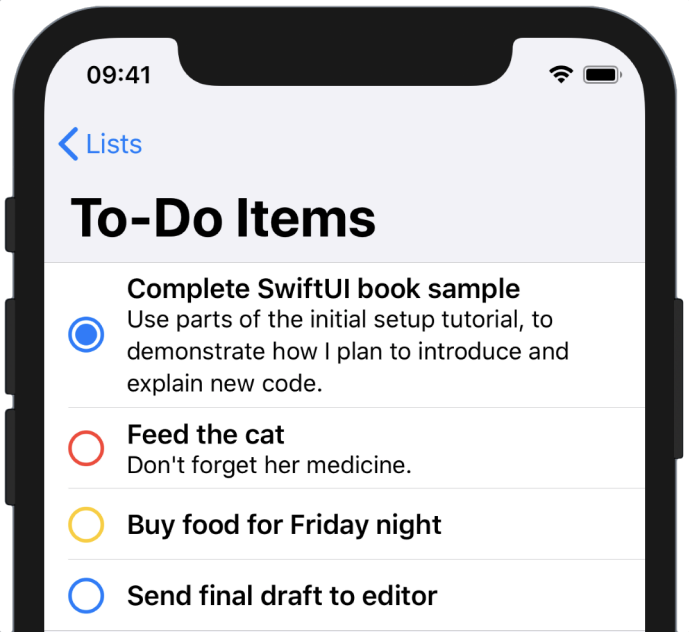
Here the list property holds either a list or nil. The items and title properties then return the name or items from that list, and use the ?? operator to return the appropriate all-items values when list is nil.

The color(for:) method will return a color to use for a given item row. If list is non-nil, the list's color will be used. Otherwise, the item will be asked for its list, and that list's color will be used instead.

To make use of these new properties, return to the body implementation and change the List view declaration like so:

```
Line 1  List(items) { item in
    2       NavigationLink(destination: Text(item.title)) {
    3           TodoItemRow(item: item)
    4               .accentColor(self.color(for: item))
    5       }
    6   }
```

Here you've changed two things: on line 1 the List now iterates over the result of the new items property; secondly, the associated list color is now being set as the accent color for the item row on line 4. Recall that the completion button in TodoItemRow has its foreground set to Color.accentColor—the .accentColor(_:) modifier sets the value for that color on the item row view and everything within it, meaning that the completion buttons will all take on the color of their respective lists:



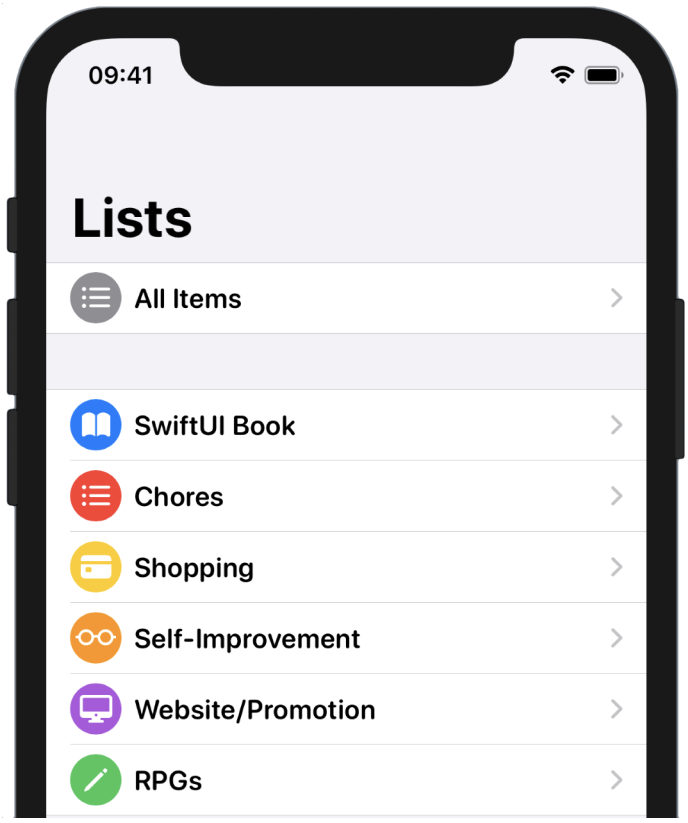
The work on TodoList is now done, so return to Home.swift and add the second section directly below the first:

2-ApplicationData/final/Do It/Views/Home.swift

```
Section {
    ForEach(defaultTodoLists) { list in
        NavigationLink(destination: TodoList(list: list)) {
            Row(name: list.name,
                icon: list.icon,
                color: list.color.uiColor)
        }
    }
}
```

This looks almost the same as the section above it, with the exception of the ForEach view iterating over the todoLists global variable. Within there, everything is recognizable: the destination for the NavigationLink is the same TodoList view, but this time it's initialized with the list to display. The content for the link is the same Row view as before, this time with the properties of the associated list passed as parameters.

All that remains is to set this as the application's root view. Open SceneDelegate.swift and replace TodoList() with Home() inside scene(_:willConnectTo:options:). Save, build, and launch your application, and try it all out!



In particular, note that every row uses a rounded headline-sized font, thanks to the .font(_:) modifier you attached to the List above. The selected font actually applies to the entire view hierarchy rooted at that view.

Warnings and Glitches

It appears that SwiftUI is doing some things that UIKit doesn't like, probably by reaching in through internal APIs that those of us on the outside can't access. This results in a few warnings appearing sometimes in the logs, but you can consider these benign.

Additionally, you might notice a stutter in the layout of the item list after you navigate in. This is nothing you can affect, it seems, and is simply a bug that will hopefully fall by the wayside.

Dynamically Ordering List Contents

The implementation of TodoList looks rather empty now, so let's make use of all the extra room and investigate the ways you can sort the list's contents.

Three methods of grouping currently present themselves: title, priority, and urgency—the items' due dates. You'll add a means to toggle this option, and to do that you need to represent the option values somehow. An enum seems like a good fit, so add this to TodoList.swift inside the definition of the TodoList type:

2-ApplicationData/final/Do It/Views/TodoList.swift
```
private enum SortOption: String, CaseIterable {
    case title = "Title"
    case priority = "Priority"
    case dueDate = "Due Date"
}
```

You'll take advantage of the ability to assign string values to enumeration cases to have each item's rawValue be usable in the UI.

Add this state variable to the properties of your TodoList:

```
@State private var sortBy: SortOption = .title
```

Sorting the list is straightforward, as the Swift standard library provides all the tools, and a simple static function in TodoList is enough to wrap the work:

2-ApplicationData/final/Do It/Views/TodoList.swift
```
private var sortedItems: [TodoItem] {
    items.sorted {
        switch sortBy {
        case .title:
            return $0.title
                .caseInsensitiveCompare($1.title) == .orderedAscending
        case .priority:
            return $0.priority > $1.priority
        case .dueDate:
            return ($0.date ?? .distantFuture) < ($1.date ?? .distantFuture)
        }
    }
}
```

Note that here you're sorting in ascending order of title or date, but descending order of priority: higher priority and closer date are most important.

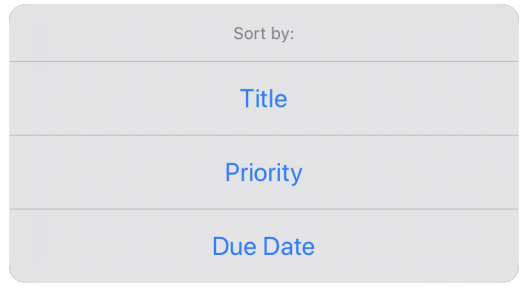
Lastly, use your new sortedItems property as input to the List view inside the body implementation:

```
List(sortedItems) { item in
    «row content»
}
```

Refresh your canvas, and you'll see that your to-do items are now sorted by title in ascending order. Try changing the value of the sortBy property in your code and refreshing the preview—does the ordering change correctly?

Changing Selections

The sort algorithm appears to be working, but your users can't very well edit the source code every time they want to change the sort order. Let's give them the means to change it via an *action sheet*, a small pop-up menu:



Action sheets and modal views in SwiftUI are implemented as view modifiers and controlled via state variables and *bindings*. The basic flow is straightforward: add a boolean-typed state variable to control whether the sheet is shown (true) or not (false); call the sheet(isPresented:content) or actionSheet(isPresented:content), passing a binding to your state variable; set the value of the state variable to show or hide the sheet.

Bindings

All property attribute types act as a wrapper for their underlying value type. For a wrapped property named foo, the compiler creates a concrete property of the wrapper type named _foo and a dynamic property of the underlying type called foo, which simply asks _foo to supply the wrapped value. Additionally, each wrapper type can provide an additional property named projectedValue which returns either the wrapped type or another wrapper. The compiler creates a new dynamic property named $foo which calls _foo.projectedValue.

SwiftUI makes use of the projected value facility to provide another wrapper type, Binding, wrapping the same underlying storage as the State property. A binding, in SwiftUI parlance, is "A value and the means to mutate it." It references some piece of state and allows other parts of your view hierarchy to observe and change that value in a thread- and type-safe manner.

First, create your state variable. Name it showingChooser:

2-ApplicationData/final/Do It/Views/TodoList.swift
```
@State private var showingChooser = false
```

Next, create your action sheet. Add a call to the .actionSheet(isPresented:content:) method onto your List view, next to the calls to .navigationBarTitle() and .listStyle(). Bind it to your showingChooser property using the $-prefixed variable $showingChooser:

2-ApplicationData/final/Do It/Views/TodoList.swift
```
.actionSheet(isPresented: $showingChooser) {
    // « ... »
}
```

The content block for the actionSheet(isPresented:content:) is expected to return an instance of ActionSheet to present, and it will be invoked whenever your showingChooser state property changes from false to true. You'll need to provide a title and a list of buttons to present. Let's start with the title:

```
ActionSheet(
    title: Text("Sort Order"),
    buttons: «…»
)
```

Since the sheet will display options derived from the list's SortOption type, you can easily determine the text for each button by mapping the enumeration's allCases to each item's rawValue, similar to this:

```
SortOption.allCases.map { Text($0.rawValue) }
```

For an action sheet, you can map these to button definitions. Action sheet buttons are all instances of Alert.Button, which provides factory functions to create three types of button: *default*, *cancel*, and *destructive*. Destructive buttons are highlighted to indicate that they will destroy some data (typically they are colored red, though this changes in some locales), while a cancel button is usually presented separately, and has a default title of "Cancel" (appropriately localized for the user's language and region). Any other type of button is created with the default type, which is what you need here.

Action buttons need two things: a label, and an optional action block. For each of your supported sorting options, you'll need to provide that option's rawValue as the button's label, and its action should set the TodoList's sortBy property to the corresponding option. Using the map() approach, that leaves the following implementation—place it in your .actionSheet() content block:

2-ApplicationData/final/Do It/Views/TodoList.swift
```
ActionSheet(
    title: Text("Sort Order"),
    buttons: SortOption.allCases.map { opt in
```

```
        ActionSheet.Button.default(Text(opt.rawValue)) {
            self.sortBy = opt
        }
})
```

Your action sheet is now ready for prime-time, but so far, there's no way for the user to invoke it. Let's use your new familiarity with buttons to create one that will present the sheet. Regular buttons in SwiftUI (i.e., Button rather than Alert.Button) are initialized with two parameters: an action block to be invoked when the button is tapped, and a content block to define its content (unlike alert buttons, you can use text, images, or both). To implement this button, the action is simple: call self.showingChooser.toggle(), or explicitly set it to true, if you prefer. For the content, let's use a large symbol image with a bold appearance. Add the following new property to TodoList:

2-ApplicationData/final/Do It/Views/TodoList.swift

```
private var sortButton: some View {
    Button(action: { self.showingChooser.toggle() }) {
        Image(systemName: "arrow.up.arrow.down.square")
            .imageScale(.large)
            .font(.system(size: 24, weight: .bold))
    }
}
```

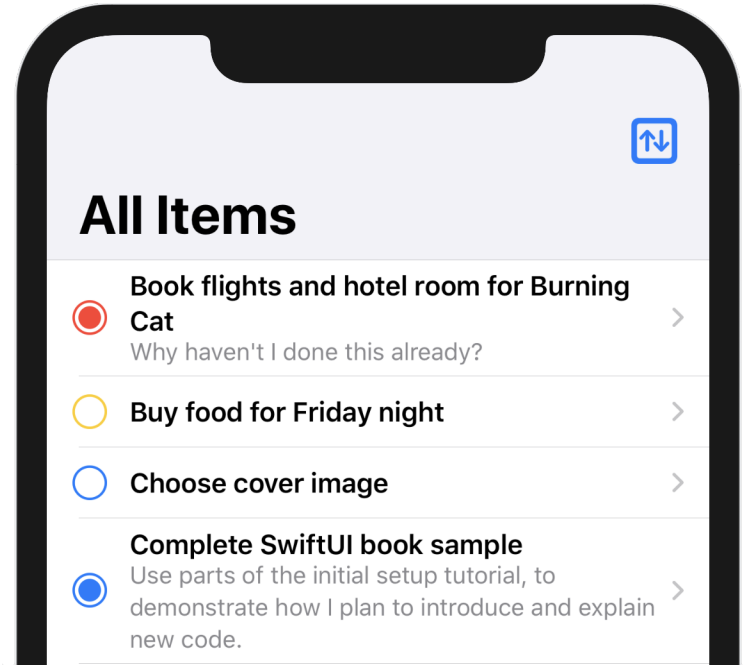
This button will fit nicely in the navigation bar, so add it to the (growing) list of modifiers on the List view you're building in your body with the navigationBarItems() method:

2-ApplicationData/final/Do It/Views/TodoList.swift

```
.navigationBarItems(trailing: sortButton)
```

Testing the Code

The code is complete, so click Resume… on the canvas to see the button presented at the top-right of the screen:



Now look to the lower right of the canvas, where the Live Preview button waits, its image is a blue 'Playback' icon. Click that icon and wait a moment while a simulator is launched inside the canvas. After a moment, the canvas will re-draw, and any selection rectangles will disappear. Click on the sort button to see the options pop up. Select each in turn (remember, it's likely set to Title already) and marvel. Not only does the list's order change, but it animates. SwiftUI provides a lot of useful features by default, and animation is just one of them.

Runtime Warnings

As noted earlier, you'll sometimes see warnings popping up in the debugger output when you run a SwiftUI application. At this point you might see a message about an invalid auto-layout constraint; consider it benign, since the application works, and you've written no auto-layout code that would need revision.

You might wonder whether the rows should display the dates or priorities of their items, and if so, you're right—they should. Right now, though, you have a simple *development interface* here, providing enough to interact with, and you'll implement a more fully-featured UI in Chapter 5, Custom Views and Complex Interactions, on page 101.

Crafting a Full-Screen View

Now that you have some List views and rows implemented, it's time to think bigger and look at how you can present all the information for a to-do item in its own view.

Create a new SwiftUI View in the Views group and name it TodoItemDetail.swift. It will need a TodoItem to operate on, so add a property for that, use it to supply the item's title to the Text view in the body, and update the TodoItemDetail() call in the preview to pass in an item:

```
struct TodoItemDetail: View {
    let item: TodoItem

    var body: some View {
        Text(item.title)
            .font(.title)
    }
}

struct TodoItemDetail_Previews: PreviewProvider {
    static var previews: some View {
        TodoItemDetail(item: defaultTodoItems[0])
    }
}
```

The detail view will present the item's title in a header section, which will make use of the to-do item's color to make the view 'pop.' Below that will be the priority, due date, and any notes.

If you think this sounds like a job for a VStack, then you're right, so that's where you'll start. Wrap the existing Text view in a VStack, then put a new Rectangle view above the title, setting its height and color:

```
VStack(alignment: .leading) {
    Rectangle()
        .fill(item.list.color.uiColor)
        .frame(height: 210)
    Text(item.title)
        .font(.title)
}
.navigationBarTitle("", displayMode: .inline)
```

Note the call to .navigationBarTitle(_:displayMode:) at the end of the VStack. If this view is displayed in a navigation view (and it will be), this call specifies the content and format of the navigation bar. Duplicating the title seems a little redundant, so an empty string will suffice, but the main reason this call is here is to specify the display mode of .inline, which will give the usual semi-transparent white bar across the top of the view, containing the back button. Without this

call, the bar would be transparent, and the title displayed in a large area below (you can see the effect on the TodoList view). Once the rectangle is extended to the top of the screen in the next section, however, any buttons on the navigation bar would become either hard to see (against a blue background) or would clash horribly. Using the standard bar provides the best of both worlds: it's white enough that the buttons look good and transparent enough that the color underneath can bleed through and give it some character.

Define a Layered Header View

This detail view doesn't look very appealing right now; what would be really nice is to have the title appear on top of the colored rectangle. SwiftUI provides two tools which can do this: first, a ZStack view will place views one on top of the other and supports both horizontal and vertical alignment types, along with combinations of both. The second tool is an *overlay* view, created by passing a view to the .overlay(_:) modifier method. An overlay view is placed over the top of an existing view as in a ZStack, but is explicitly sized to match the frame of the view below it.

In this case, a combination of the two will be useful, because there's one particular issue that can arise when placing text over a colored background: contrast. Black text works well over a light-colored background, but not so well on a darker or bolder color. Black on a mid-blue is readable, but not appealing. White text on a pale yellow is hard to read. There are ways around this. For example, you can look at the hue, saturation, and brightness of the underlying color and pick black or white text based on that, but this can be error-prone.

Instead, let's take a Gordian-knot approach and cut through the problem with a simple solution. You'll always darken the color underneath the text, then use white text. A gradient from a slightly-transparent black to fully transparent laid over the bottom part of the colored rectangle will darken the area containing text enough that you have a pleasant contrast, while still allowing the user's choice of color to shine through unchanged toward the top of the rectangle.

First, let's create the overlay view. This will contain the text of the title and a gradient providing the darker background that will give more contrast for the text above it.

In TodoItemDetail.swift, at the bottom of the TodoItemDetail definition, add a new sub-type named TitleOverlay:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
private struct TitleOverlay: View {
    let item: TodoItem
}
```

Next, define the overlay's gradient through a dynamic property:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
Line 1 var gradient: LinearGradient {
2      LinearGradient(
3          gradient: Gradient(colors: [
4              Color.black.opacity(0.6),
5              Color.black.opacity(0),
6          ]),
7          startPoint: .bottom,
8          endPoint: .init(x: 0.5, y: 0.1))
9  }
```

The linear gradient type takes a single Gradient, defined on line 3, and a start and end point which define where within the view the color should begin to change. This gradient fades between two colors: a slightly transparent black to a completely transparent one. The start point on line 7 states that the gradient should begin at the bottom of the view, while line 8 states that it should end close to the top of the view (in SwiftUI, the y-coordinate grows downwards, meaning 0.0 is at the top of the view).

Unit Points

A common concept in drawing APIs is that of a *unit point*, namely a value between zero and one that is used to define a position within some other area: the unit value is multiplied by the size of the appropriate axis within the target area to get a real location. Gradients use these extensively, and SwiftUI includes the UnitPoint type to aid in their use.

Conceptually, a unit point in a 2D coordinate system is a point with x and y coordinates, each between zero and one. A gradient is defined as a series of colors which will be interpolated with one another to create a smooth change across some region, like the content bounds of a view. When the gradient is created with some number of colors, a unit point is used to define the location at which it will place each of its constituent colors. To fade between two colors from top to bottom of an entire view, you would use y coordinate values of zero and one, respectively. To fade from the left to the right, use x coordinates of zero and one. To fade from top-left to bottom-right, both x and y values would use zero and one. The UnitPoint type includes several predefined values for the most commonly-used locations, including the leading and trailing edges, top and bottom, corners, and center.

When you only want to make use of a single dimension, as in TitleOverlay, you simply provide the same value in all points. The standard convention, however, is to use 0.5 for values on an unused axis. All of the built-in unit points provided by SwiftUI follow

this scheme, so you'll need to be aware of this if you want to pair a custom point with .bottom, as in the gradient on page 50.

Assemble the Overlay Layer

With these parts in place, you can start on the body implementation. To place the text on top of a rectangle containing the gradient, use a VStack with an alignment of .bottomLeading, so that the content is laid out from the lower-leading corner of the header view:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
var body: some View {
    ZStack(alignment: .bottomLeading) {
        Rectangle().fill(gradient)
        // « ... »
    }
}
```

Note that in this view, you don't need to specify any frame sizes; since it's used as an overlay for another view, its size will be defined by that other view.

To see this view on the canvas, you need to add it to the body of the TodoItemDetail view. At the same time, let's display any notes in the to-do item. Add a call to .overlay(TitleOverlay(item: item)) to the existing Rectangle view, and replace the title with a Text view containing the item's notes field, if it's non-nil, with a little horizontal padding. Lastly, place a Spacer view at the end to push the header to the top of the screen. Your body implementation should look something like this:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
VStack(alignment: .leading) {
    Rectangle()
        .fill(item.list.color.uiColor)
        .edgesIgnoringSafeArea(.top)
        .frame(height: 210)
        .overlay(TitleOverlay(item: item))

    if item.notes != nil {
        Text(item.notes!)
            .padding(.horizontal)
    }

    Spacer()
}
.navigationBarTitle("", displayMode: .inline)
```

Press the Resume button in the canvas, and you'll see the color rectangle taking up the top part of the device's screen, complete with a smooth gradient darkening it toward the bottom. This could look nicer, though—if you're looking at an iPhone X, XS, XR, or iPhone 11 in your canvas (this can be changed by selecting a different simulator device in Xcode's scheme switcher), then you'll notice a white strip across the top of the screen above the header. That real-estate could be a lot more use to us as part of the header, so append a call to .edgesIgnoringSafeArea(.top) to the Rectangle in TodoItemDetail's body implementation, between the .fill() and .frame() modifiers. The item's color now swoops upward to fill in the status bar area, giving the view a much more appealing aspect.

Returning to the TitleOverlay view, you can now begin to add the content. Following the Rectangle containing the gradient, add a new VStack with leading alignment and a spacing of eight points, and place the item's title in here using a Text view with a bold .title font. Your code should look like this:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
VStack(alignment: .leading, spacing: 8) {
    Text(item.title)
        .font(.title)
        .bold()
    // « ... »
}
.foregroundColor(.white)
.padding()
```

Your text should appear in large white letters over the darkened part of the item header. Next, let's add the priority and due date, if any. Start out by appending a HStack after the title:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
HStack(alignment: .firstTextBaseline) {
    // « ... »
}
.font(.subheadline)
```

All the text in this stack is going to use the same font, so that font is set on the HStack itself—this will cascade that setting to any descendant views.

Show Item Priority

For the priority, simply including the name isn't necessarily very descriptive. While "Urgent," in English at least, sounds like a call to action and can be quickly interpreted as "this is an urgent to-do item," the same can't be said for "low," "high," or "normal." "This item is low" doesn't necessarily suggest

a priority in the way "this item is urgent" does—and that's only in English. When your application is localized, there may be yet more ways that a single-word priority indicator can appear detached or even nonsensical.

For that reason, you're going to use a formatted string of "Priority: X," where "X" will be replaced with the priority name, and will additionally be rendered with bold text to indicate that this is the part of the sentence containing actionable information. Looking at the initializers for Text views, though, it doesn't seem like there's an obvious way to specify different fonts, colors, or any other attributes for parts of the text. The NSAttributedString type—the typical way to supply text with formatting information in UIKit—isn't mentioned anywhere in SwiftUI's API, in fact.

Happily, all is not lost. Text happens to implement a very Swift-y way of achieving exactly what you need:

```
extension Text {
    public static func + (lhs: Text, rhs: Text) -> Text
}
```

You can, in fact, take two separate Text instances, each with modifiers affecting their content and presentation, and add them together, resulting in a single text view with all that presentation information retained. Here, then, you can easily combine a plain "Priority:" label with a bolded label containing the priority's name (suitably capitalized), in one line of code:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
Text("Priority: ") + Text(item.priority.rawValue.capitalized).bold()
```

Your canvas now displays exactly what you defined, and not a mention of NSAttributedString.attributes(at:effectiveRange:).

There's plenty of horizontal space left over, so let's put the due date on the trailing edge of the view. First, add a Spacer view, then add a Text view containing either the due date (suitably formatted) or a static message if there is no date attached:

2-ApplicationData/final/Do It/Views/TodoItemDetail.swift

```
Spacer()
if item.date != nil {
    Text("\(item.date!, formatter: Formatters.mediumDate)")
}
else {
    Text("No Due Date")
}
```

Note that the date string uses a form of string interpolation that makes use of an optional Formatter; this uses the static formatter property you created earlier, which the string interpolation engine uses to convert the provided value (a Date in this instance) into a String, ready for display.

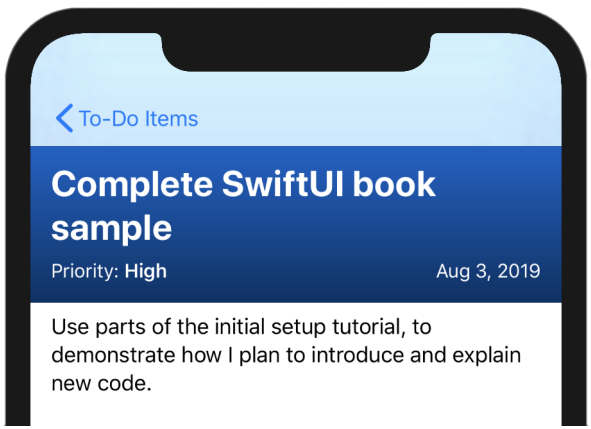
Now, your item should be displayed on the canvas with a vibrant color-filled header view drawing attention to the primary attributes of the item, with any additional notes showing below. All that remains is to tie it into the rest of the application.

Open TodoList.swift and find the NavigationLink in the view's body implementation. Replace the link's destination with a TodoItemDetail like so:

2-ApplicationData/final/Do It/Views/TodoList.swift

```
NavigationLink(destination: TodoItemDetail(item: item)) {
    TodoItemRow(item: item)
        .accentColor(self.color(for: item))
}
```

Start a live preview, or launch the application in the simulator or on a device, and navigate into a to-do item to see your new view in action:



What You Learned

This chapter has covered a lot of ground, but now you ought to have a number of useful tools on your belt:

- You can drive view content updates via state properties.
- You've seen how you can create interesting and inventive interfaces by composing several views together.
- You're able to make use of the gesture system to implement your own input controls.

- Management of deeper navigation hierarchies is now a straightforward task.
- Rich-text labels are within your grasp.
- You took your first steps into the realm of styles and modifiers in SwiftUI.
- Stack views are clearly the power tools of SwiftUI.

Next, you'll investigate the facilities provided for working with mutable data, and how to implement editing functionality in your SwiftUI application.

CHAPTER 3

Modifying Application Data

Story Map

Why do I want to read this?

User interaction is the crux of your application. Attractively presenting data is all very well, but your users will expect to be able to add, modify, and remove such data.

What will I learn?

You'll be introduced to the SwiftUI `@Environment` type and learn how to make use of standard edit-mode affordances. You'll see how to add interactive controls to your app and how to respond to user input.

What will I be able to do that I couldn't do before?

You now understand how SwiftUI's control system works for user input of various types and can safely share data and state information amongst your views.

Where are we going next, and how does this fit in?

Next, you're going to take what you've learned about state and controls and make your list view more flexible.

In this book so far you've used SwiftUI to present data, arranging it in an aesthetically pleasing manner. A real application does more than just display static data, though: it allows the user to interact and modify that data. Really good applications alert the user to changes through the use of animations that draw attention in the right direction without interrupting the application's flow and its primary purpose.

In this chapter, you'll begin to wire your data model into SwiftUI properly, providing edit controls where necessary, and reacting to changes in the data model cleanly when they happen. SwiftUI's state management tools make

these tasks simple to approach for the majority of cases while stepping out of the way when you need to take more fine-grained control of the process.

To follow along, download this book's source code bundle[1] and use the starter project found in code/3-ModifyingData/starter. An example of the complete result of this chapter's work can be found in code/3-ModifyingData/final.

Data Flow in SwiftUI

One of SwiftUI's strengths is its focus on state management. It provides types and *property wrappers* that work hand-in-hand with its layout and rendering system to ensure that any modifications to your application's state are correctly reflected in your view hierarchy. It also makes use of its central position in the procedure to ensure that only the requisite parts of the interface are updated: it is able to use its knowledge of state to determine that some subviews haven't changed without necessarily diving through the full hierarchy, comparing view contents one by one.

Property Wrappers

Swift has a few special keywords or attributes that can be applied to properties to induce certain behavior. For example, the lazy keyword causes a property to be initialized lazily—i.e. only when first requested. The @NSCopying attribute for properties of an Objective-C type will cause that type to be copied rather than retained during assignment to the property.

Swift 5.1's property wrappers allow programmers to create their own attributes via use of the @propertyWrapper attribute on a type along with some special properties. When these attributes are used, the compiler synthesizes several stored and computed properties implementing the API you'll use in your appliation.

When you declare a property using a property wrapper attribute named myProperty, the compiler creates a property of the *property wrapper's* type and names it _myProperty. It then creates a second, computed, property named myProperty whose value is fetched from within the wrapper. Further, if the wrapper offers a special *projected value*, the compiler generates a computed property named $myProperty which fetches the project value from the wrapper. The SwiftUI @State property wrapper uses this to produce a binding to its value.

State in a SwiftUI app is managed with several different tools, each with a particular purpose. There are types used to manage discrete state for one or more views, and there are ways to directly share state downwards toward child views or to pass state changes upwards to ancestor views. There are

1. https://pragprog.com/titles/jdswiftui/source_code

also ways to indirectly pass data up and down the view stack, making it available to any descendants or any ancestors that may be interested—without tying the two together directly.

State Management

State data in SwiftUI is managed by the framework. You mark certain items as state by using property wrappers, and SwiftUI then observes when these are accessed and modified. When a view's `body` implementation accesses state, this is noted by the framework. Then, when a state value is modified, any views that accessed it are recreated through their `body` property, after which SwiftUI will determine what in the view hierarchy actually changed and will merge those changes (and only those changes) into the on-screen interface.

There are two property wrapper types that you use for this:

@State

Mark a structure-type property with the `@State` attribute, and SwiftUI will keep track of it, using any changes to that value to trigger updates of the view hierarchy. This type is logically for mutable data; you can use a `class` type, but mutating the contents of an object instance won't trigger anything—only directly assigning a new (object) value will cause any change. Structure types, on the other hand, fit perfectly into `@State` properties, since any change to the contents is a change to the whole structure, triggering an update.

@ObservedObject

For class types, the `@ObservedObject` attribute is the tool of choice. It works specifically with objects that implement the `ObservableObject` protocol from the *Combine* framework. Once you have a class that conforms to this protocol, Combine's `@Published` property attribute will cause any changes to the tagged attribute to be published by the containing object. SwiftUI then uses this to observe changes to the content of an object (reference) type, obtaining the same behavior that `@State` properties provide for `struct` types.

From the point of view of the SwiftUI framework, these types both serve the same purpose: they allow the framework to react to changes of state. Additionally, they provide a second function: they can automatically provide *bindings* to themselves and their content.

Bindings are conceptually references to the `@State` or `@ObservedObject` properties, or to their contents; they read their values from the original object, and any changes made to them are written back to the original object. Here, the state properties are considered the source of truth for a given value, while a binding

is merely a reference back to that source of truth. The idea is that your application should have one source of truth for a particular piece of its state—it should live in one place exactly—while anywhere else that wishes to access it will instead bind to that value remotely.

When defining the properties of a view, you use the `@Binding` attribute to declare that your property should be a binding to some other state. When you need to create a binding, you can obtain one by using the `$` prefix on a state property, which will return a binding to itself. To get a binding to some property within a state type, you prefix the entire expression with the `$` prefix, and you'll ultimately receive a binding to the final item in the property chain:

```
@State var myState: SomeData

var body: some View {
    SomeView($myState.user.name) // receives a binding to the 'name' property
}
```

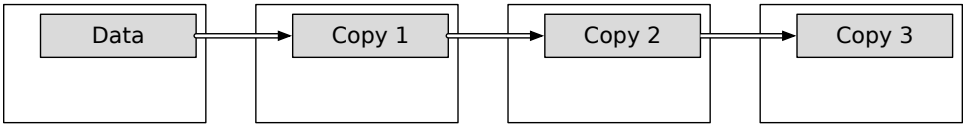
Dependency Propagation

Bindings are one of the principal tools provided for *dependency propagation* in SwiftUI. Dependency propagation refers to the way in which state data is made available to different parts of your application, from a single source of truth to various other views. You can thus link together your views in a dependency chain—one view depending on state from another—in several different ways.

In UIKit or AppKit applications, you're generally left to handle this task on your own by copying values or references around the view hierarchy, then manually notifying controllers and views when the data changes. SwiftUI sets out to handle as much of this task as possible on your behalf, distilling dependency propagation into four distinct types of direct and indirect linkage both up and down the view hierarchy:

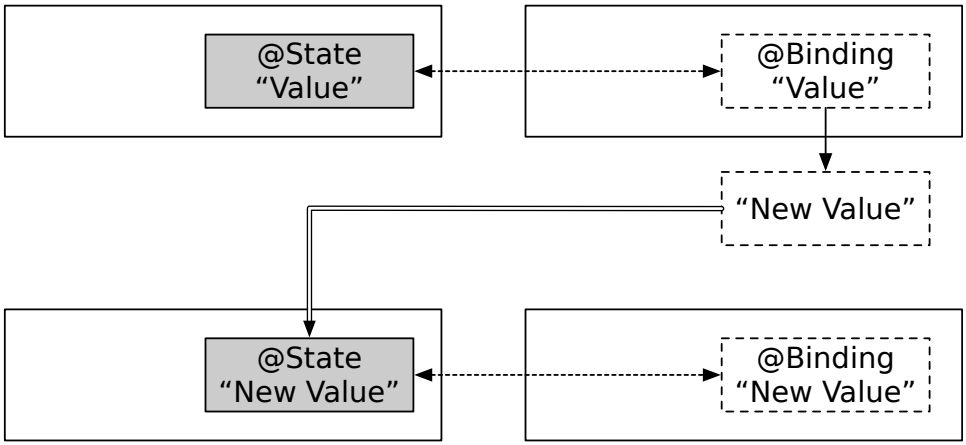
Direct, downwards

Regular Swift properties will pass an item as a parameter when creating a subview to make that data available directly. `Text` views function this way, for example; you pass them a plain `String`.



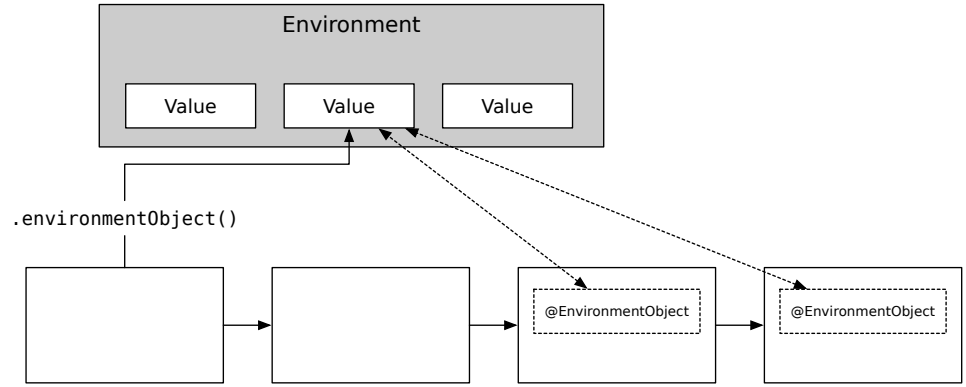
Direct, upwards

For upward state movement, the Binding type is preferred. You pass a binding to a @State or @ObservedObject property into a subview to allow that subview to both see and modify your state. Most controls use this to bind them to some data owned or observed by the ancestor that created them.



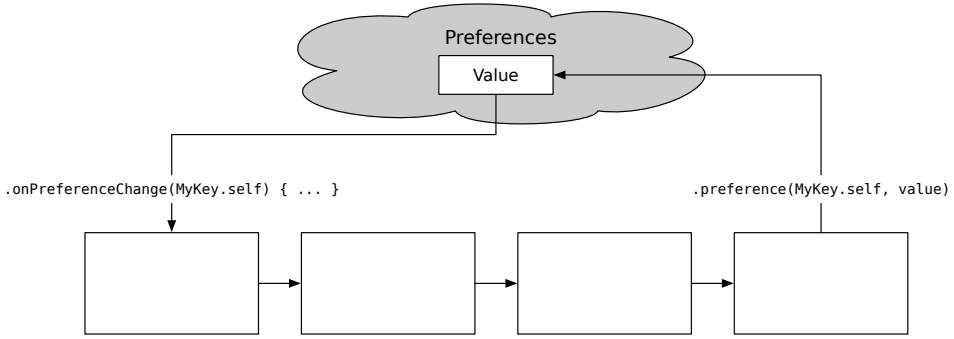
Indirect, downwards

For passing values down the view hierarchy in general, SwiftUI provides the *Environment*. Two property wrappers, @Environment and @EnvironmentObject (equivalent to @State and @ObservedObject respectively), allow child views to pluck from the environment that were placed there when some ancestor view was created. Many view modifiers use the environment behind the scenes; this is how setting a foreground color on a top-level view takes effect on its descendants.



Indirect, upwards

Here, you'll find one of the lesser-known power tools of SwiftUI: the PreferenceKey protocol. This allows you to define a type that carries some data value along with the knowledge of how to reduce multiple values down to a single one. Any descendant view can associate a value with a particular PreferenceKey type, and its ancestor can request that value via the .onPreferenceChanged() view modifier. The values from all subviews are reduced to a single value that is provided to the ancestor. This way, a subview can potentially alter many things about a parent view—a settings editor might change the background color of an application's root view, changing the values used in the body property of the root view.



Application Data in "Do It"

The application you're building involves a few items of data, which at present are being loaded from a built-in JSON file. While that may be a useful way to present some initial sample content to the user, real data generally lives elsewhere and is updated by the application in response to user input. As a precursor to building out editing support, you'll look at the final storage of your data and how it is passed down through your application's view hierarchy.

In the starter project look at Model/DataCenter.swift; this component is going to become more important as you work through this book, and you'll make use of more of its API as you continue. This defines the data source where your data will live—your application's *source of truth*. It implements the shared storage for all the lists and items you'll be using, along with automatic save and load functionality, and more. Most of its implementation is outside the bounds of a book focussed purely on SwiftUI, but there are a few lines are worth pointing out:

3-ModifyingData/final/Do It/Model/DataCenter.swift

```
Line 1 import Foundation
   2 import Combine
   3 import SwiftUI
   4
   5 final class DataCenter: ObservableObject {
   6     @Published var todoItems: [TodoItem] = []
   7     @Published var todoLists: [TodoItemList] = []
   8     @Published var defaultListID: UUID
   9 }
```

On line 2, you'll notice a new module being imported, named Combine. This framework, introduced by Apple alongside SwiftUI, underlies a lot of SwiftUI's reactive workflow. It implements a publisher-subscriber model of data flow along with a variety of tools used to transform and validate different types of data automatically. It's being imported here to tie in with SwiftUI's @ObservedObject attribute, which explicitly uses Combine's ObservableObject protocol. You'll note that DataCenter specifies conformance to that type.

Observable objects are those that *publish* information on their changes to *subscribers*, such as SwiftUI. While it's possible to implement the necessary protocol conformances manually, the simplest way to get what you need is to mark properties you wish to publish with the @Published attribute, as on line 6. This property wrapper from Combine will do everything for you so that when the property is modified (as it's a value type, that includes modification of its contents), all subscribers will be notified. SwiftUI will learn about changes by subscribing, which will happen automatically via the @ObservedObject property wrapper.

Using the Environment

Now that your data is contained and managed by an instance of the DataCenter class, you'll have to feed that instance into your application's view hierarchy. The simplest approach is to have your root view declare and initialize a property using the @ObservedObject attribute to hold it, like so:

```
@ObservedObject private var data = DataCenter()
```

That, however, presupposes that the chosen view will always be at the root of the application and that it will always pass bindings down to its children. At present, this might seem reasonable enough—your view hierarchy isn't terribly deep at present—but as your application grows, manually passing bindings around will start to become rather cumbersome. Instead, you'll use the @EnvironmentObject attribute and the .environmentObject() modifier to make it available as part of your views' environment, available to any descendants.

To install it, open SceneDelegate.swift and modify the content of scene(_:willConnectTo:options:) as follows:

3-ModifyingData/final/Do It/SceneDelegate.swift

```
let contentView = Home()
    .environmentObject(DataCenter())
```

Here, you've initialized a DataCenter instance for the scene and attached it to the environment for the root view at the same place you define that root view. Should you ever swap out the Home view for a different type, the data will still be attached to the new view in the same manner.

With the data now living in a new location, your views will need to be updated to correctly bind to the items managed by the DataCenter.

Start by opening Views/Home.swift. Here you need to add a DataCenter property with the @EnvironmentObject attribute, and then you'll need to change the the ForEach view initializer to use the todoLists property of that DataCenter. Lastly, you need to add a call to .environmentObject() in the preview provider at the bottom of the file, so that the data is available in previews as well as the real application.

Make the following updates to the file:

3-ModifyingData/final/Do It/Views/Home.swift

```
struct Home: View {
    @EnvironmentObject private var data: DataCenter

    var body: some View {
        NavigationView {
            List {
                Section {
                    // « all items »
                    }
                }
            }
            // « view modifiers »
        }
    }

    // « struct Row: View { ... } »
}

struct Home_Previews: PreviewProvider {
    static var previews: some View {
        Home()
            .environmentObject(DataCenter())
    }
}
```

Next open TodoList.swift. The changes here will be confined to three places: the items property will be rewritten to use the new DataCenter instance fetched from the environment; the environment object will be attached to the for the TodoItemDetail view; and the color(for:) method will use the data center to find the list associated with an item. The required changes are minimal:

3-ModifyingData/final/Do It/Views/TodoList.swift

```
@EnvironmentObject private var data: DataCenter

var list: TodoItemList? = nil
var items: [TodoItem] {
    guard let list = list else { return data.todoItems }
    return data.items(in: list)
}
var title: String { list?.name ?? "All Items" }

func color(for item: TodoItem) -> Color {
    let list = self.list ?? data.list(for: item)
    return list.color.uiColor
}

// « ... »

var body: some View {
    List(sortedItems) { item in
        NavigationLink(destination: TodoItemDetail(item: item)
            .environmentObject(self.data)
        ) {
            // « TodoItemRow(...) »
        }
    }
    // « modifiers »
}
```

Note that the code for creating the list content views hasn't changed, and doesn't need to; the TodoItemRow doesn't need to bind to the contents of the to-do list, as it is just reading from its contents. A copy of the value will suffice, and the potential dependency problems are nicely sidestepped. The call to .environmentObject() inside the NavigationLink is important though: the new view pushed onto the navigation stack won't automatically inherit the DataCenter, and the TodoItemDetail view will need to use it to access the list color for its TodoItem.

Don't forget to attach a DataCenter to the environment in the preview provider:

3-ModifyingData/final/Do It/Views/TodoList.swift

```
struct TodoList_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            TodoList()
        }
```

```
        .environmentObject(DataCenter())
    }
}
```

Lastly, open TodoItemDetail.swift and make the following changes to fix the errors noted by Xcode:

```
struct TodoItemDetail: View {
    let item: TodoItem
    @EnvironmentObject private var data: DataCenter

    var body: some View {
        VStack(alignment: .leading) {
            Rectangle()
                .fill(data.list(for: item).color.uiColor)
                // « modifiers »

            // « notes »
        }
        .navigationBarTitle("", displayMode: .inline)
    }

    // « private struct TitleOverlay: View { ... } »
}
```

Once again, don't forget to add an .environmentObject(DataCenter()) modifier inside the preview provider.

Building an Editor

Your next task is to implement an editor for your to-do items. This involves a few extra types added as part of this chapter's starter project, all of which reside under the Helpers group.

Create a new SwiftUI View file in the Views group and name it TodoItemEditor.swift. Edit the view and the preview provider to match the following example:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
struct TodoItemEditor: View {
    @Binding var item: TodoItem
    @EnvironmentObject private var data: DataCenter
    @State private var showTime: Bool

    init(item: Binding<TodoItem>) {
        self._item = item
        self._showTime = State(wrappedValue: false)

        if let date = item.date.wrappedValue {
            let components = Calendar.current.dateComponents(
                [.hour, .minute], from: date)
            self.showTime = components.hour! != 0 || components.minute != 0
        }
```

```
15      }
-
-       var notesEditor: some View {
-           TextView(text: self.$item.definiteNotes)
-               .padding(.horizontal)
20             .navigationBarTitle("Notes: \(item.title)")
-       }
-
-       var body: some View {
-           Form {
25             // « form contents will go here »
-           }
-           .navigationBarTitle(Text("Editing: \(item.title)"),
-                               displayMode: .inline)
-       }
30 }
-
- struct TodoItemEditor_Previews: PreviewProvider {
-     static var previews: some View {
-         NavigationView {
35             StatefulPreviewWrapper(defaultTodoItems[0]) {
-                 TodoItemEditor(item: $0)
-             }
-         }
-         .environmentObject(DataCenter())
40     }
- }
```

Two items here deserve special attention:

- Your first @Binding property appears on line 2. An editor logically operates on data owned by someone else—it provides a service. You model this by using a binding, as discussed in Dependency Propagation, on page 60. This allows the editor to view the current value and also to modify them, triggering changes to the underlying @State or @ObservedObject property that owns the value.

- It turns out that previews and @Binding types don't mix terribly well. You might normally pass in a constant binding via Binding.constant(), but that isn't useful in an editor: all changes are ignored. Declaring a real @State property is fine, but sadly fetching a binding from that outside of a View.body call isn't allowed—and SwiftUI doesn't consider PreviewProvider.previews a body method—so you can't define mutable preview state within the preview provider itself. The StatefulPreviewWrapper referenced on line 35 provides a concise workaround for this issue by accepting a value in its initializer that it stores as a @State property, then provides a binding to that state as input to the provided ViewBuilder block, allowing the preview to pass it

on. You can see find the implementation in Preview Assistants/StatefulPreviewWrapper.swift.

The view's body contains a Form view. This container view is available on all platforms, and it behaves in the appropriate manner for each. On macOS, for example, it acts more or less like a VStack, laying its contents out in a regular view. On iOS, it will instead use the appearance of a List with a grouped style. Furthermore, it alters the default appearance of certain control types, assuming the use of a navigation view to push a new view containing menus of items to choose, for example. Overall, it aims to mirror the appearance of the Preferences application, so look at that to get an idea of what you'll see.

The first thing to include in the form is the to-do item's title. For this, you'll use a TextField view, which takes a label string (amongst other things, labels serve as accessibility aids) and a binding to the value to display and edit. The title is straightforward—"Title" will suffice—and the binding is generated using the $-prefix operator on this view's item property:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift
```
TextField("Title", text: $item.title)
```

Choosing Values

Next let's provide a way to select the list to which the item will belong. For this you'll use a Picker control, which also takes a label string and a binding to the underlying value. This differs, however, in how you provide its content. Where a TextField is representing some straightforward data, a picker might display any number of options representing all sorts of things, so the picker uses a @ViewBuilder block to build its content. This picker will contain one entry for each defined TodoItemList, so to provide that you'll use a ForEach view iterating over the contents of the todoLists property of the DataCenter, providing a Text view displaying each list's name.

The picker's value itself will be bound to the listID property of the view's bound item. This presents an interesting situation: the picker is displaying Text views, each generated from a TodoItemList, but each option needs to assign a related Int value to the picker's bound property. How can SwiftUI map from a TodoItemList to an Int here?

The answer lies in the specifics of the ForEach view, or, more specifically, its output. The ViewBuilder block passed to the ForEach is invoked once for each value in the provided collection, and the resulting view is then *tagged* with that value in SwiftUI's internal data store. The associated value is then used by the picker to set the value of its binding. "That's all well and good," you

may think, "but the value doesn't match the type of the binding I'm using," which is true. In this case, you can override the tagged value using the .tag(_:) modifier, in essence supplying the value that the Picker will assign to its binding. Here the value you need is each list's id value.

The resulting code is short and sweet, belying the complexity underneath (a common occurrence in SwiftUI):

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift
```
Picker("List", selection: $item.listID) {
    ForEach(data.todoLists) { list in
        Text(list.name).tag(list.id)
    }
}
```

The next property to tackle is the item's priority, which will also use a Picker view. Here, however, the type of the bound property and the type iterated by the enclosed ForEach view already match up, so the explicit .tag(_:) call isn't needed. The TodoItem.Priority type conforms to CaseIterable, so you'll pass TodoItem.Priority.allCases as the collection the ForEach view will iterate over. Lastly, to see a properly localized representation of the priority, its rawValue will be wrapped in a LocalizedStringKey:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift
```
Picker("Priority", selection: $item.priority) {
    ForEach(TodoItem.Priority.allCases, id: \.self) {
        Text(LocalizedStringKey($0.rawValue.capitalized))
    }
}
```

Since these pickers are appearing in a Form, they will take on a particular appearance by default, that of a NavigationLink with a value displayed at its trailing edge. Tapping on the link will push a new Form view containing the available options—tapping on one of these will assign the value and pop the view from the navigation stack. If you change the Form to a List, though, you'll see that the picker changes to appear as an inline wheel control containing the possible selections. Revert back to using a Form before continuing.

Editing Optional Properties

Following the priority, you'll add the UI to select a date for your to-do item. This leads to an interesting issue, though, when no date is assigned, and the property value is nil. In logical terms, the user may not want or need to assign a date; in programming terms, the date property is an Optional. Date pickers—like the other controls you've seen so far—are bound to a concrete value, not an optional, so you'll need to lend a helping hand to make this work. You'll also

need to do something similar for the notes property, which likewise may legitimately be nil.

To make this all work, you'll take a two-pronged approach. For binding the notes and date properties, you'll create wrapper properties that transform nil values into suitable 'empty' variants. In addition, while an empty string can be easily turned into nil for the notes property, the same can't be said of the date because there isn't a suitable 'not a date' value that the user could select from a date picker. For that, then, you'll create a new property used to toggle the presence of a valid date on and off.

Add the following file-private extension for the TodoItem type inside TodoItemEditor.swift:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
Line 1  fileprivate extension TodoItem {
        var hasDueDate: Bool {
            get { date != nil }
            set {
                if newValue && date == nil {
                    date = Date()
                }
                else if !newValue {
                    date = nil
                }
            }
        }

        var definiteDate: Date {
            get { date ?? Date() }
            set { date = newValue }
        }

        var definiteNotes: String {
            get { notes ?? "" }
            set {
                if newValue.isEmpty {
                    notes = nil
                }
                else {
                    notes = newValue
                }
            }
        }
    }
```

The first property, hasDueDate, simply returns whether the date property is not nil. On line 5 in its setter, the property checks whether it needs to assign a

concrete value to date; if so, it uses the current date, since this is a reasonable initial value.

The second property is named definiteDate. It assigns any values straight through to the real date property, but in the case there currently is no concrete value, it will again return the current date, as seen on line 15.

The last property, definiteNotes, will return an empty string if there is currently no concrete value to use. When a new value is assigned, it is either saved directly to the underlying notes property, or—as you see on line 23—it will translate any empty string into a nil value.

With these new properties in place, you can continue implementing the body of the TodoItemEditor view. You'll start by adding a Toggle control bound to the hasDueDate property. Turning this on will reveal the date picker itself, and turning the toggle off will hide the picker. The picker, in turn, will be bound to the new definiteDate property.

The use of two separate controls to work on a single logical value leaves you with some nuances to consider, though. Visually the two controls are clearly related, and the animation provides a visual cue to the nature of that relation. The key word, however, is "visual," and many of your app's users will be unable to see, and will not get those cues. To VoiceOver, these are simply two separate controls, and to visually impaired users, the user interface consists almost entirely of their labels. For users able to follow the visual cues, though, the UI needn't be cluttered with two separate descriptive labels ("a picture says a thousand words," they say).

Your solution here is to provide suitably descriptive labels to the controls for VoiceOver's benefit, and then to hide those labels from the screen. You'll then add a single Text view to serve as a label for visual purposes:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
HStack {
    Text("Due Date")
    Spacer()
    Toggle("Has Due Date", isOn: $item.hasDueDate.animation())
        .labelsHidden()
}
```

Here, you've used a HStack to lay out a label followed by a Toggle control. The toggle has the accessibility-compatible label of "Has Due Date," and its selection is bound to the hasDueDate property of the to-do item being edited. You're also taking advantage of some of SwiftUI's built-in support for animations. Since bindings are frequently used for changing values, which in turn

cause views to change, they provide quick means to attach animations to all changes. Here, you use the .animation() method on Binding to get a new binding that animates its changes. As you'll see in a moment, the date picker control is added to the view when the hasDueDate property is true—using the .animation() modifier here will cause the picker to be moved into place with a suitable animation.

Now you're almost ready to implement the date picker, but there's one extra factor worthy of consideration: is the due date for this to-do item just a day, or a specific time of day? SwiftUI's DatePicker view can display either a date, a time, or both, and it would be good to let the user choose the level of granularity they want to use here (Apple's Reminders application takes a similar approach). To implement this, you'll need a new @State property that can be adjusted with another Toggle, but you'll need to do some additional set-up to have it working; you need to inspect your item's date property, if set, to see if it contains a non-zero hour or minute, and set the state appropriately. That will need to happen in the editor's initializer, which means you'll now have to implement that yourself:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
@State private var showTime: Bool

init(item: Binding<TodoItem>) {
    self._item = item
    self._showTime = State(wrappedValue: false)

    if let date = item.date.wrappedValue {
        let components = Calendar.current.dateComponents(
            [.hour, .minute], from: date)
        self.showTime = components.hour! != 0 || components.minute != 0
    }
}
```

There's some unusual activity here. There are no properties named _item or _showTime in the code, but Xcode isn't complaining at all—what's going on? The answer lies in the implementation of property wrappers, as described in Property Wrappers, on page 58. Thus, when the compiler sees @State var showTime: Bool, it effectively inserts the following code:

```
var _enabled: State<Bool> = State(initialValue: false)
var enabled: Bool {
    get { _enabled.wrappedValue }
    set { _enabled.wrappedValue = newValue }
}
var $enabled: Binding<Bool> { _enabled.projectedValue }
```

Swift normally synthesizes initializers for you and handles the details of creating a State instance from a regular value transparently. When you implement the initializer yourself, however, you can't just assign to showTime, as Xcode will complain "Variable 'self.showTime' used before being initialized." Instead, you must initialize the wrapper type itself via the synthesized _showTime property. The same, naturally, applies to the item argument: the view's item property is synthesized, and has a type of TodoItem, but the initializer has received a Binding<TodoItem> instance. That argument must therefore be assigned to _item, the Binding property itself.

Now you have everything you need to define the date picker; add the following code at the bottom of the Form content to add the new rows:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
if self.item.hasDueDate {
    Toggle("Include Time", isOn: $showTime)
    HStack {
        DatePicker("Due Date", selection: $item.definiteDate,
                   displayedComponents: showTime
                        ? [.date, .hourAndMinute]
                        : .date)
            .datePickerStyle(WheelDatePickerStyle())
            .labelsHidden()
            .frame(maxWidth: .infinity, alignment: .center)
    }
}
```

With this code, you check whether the toggle was enabled by looking at the state to which it was bound. Remember that in SwiftUI views are just a generated representation of state—so you don't check if a control is toggled on, you check if a value is set, and trust that the toggle will display that value appropriately. If the property is set, then two rows are added to the form: one toggles the showTime property, and the next displays the date picker, bound to the new definiteDate property.

As with the priority picker earlier, a Form will adopt a special layout for date pickers by default; you can see it for yourself by removing the call to .datePickerStyle(WheelDatePickerStyle()). Normally, a form will show an abbreviated representation of the date value, and when you tap on it, a wheel-style picker will animate into place below it. Tapping the row again will close the picker. However, you already have your own toggle to enable/disable the date picker, so here you explicitly request the wheel style of picker via the .datePickerStyle() modifier. As with the "Due Date" toggle above, the picker's label value is visually superfluous, so you again hide it using the .labelsHidden() modifier.

The use of .frame(maxWidth:alignment:) causes the picker to be centered onscreen without needing to resort to Spacer views on either side. The .frame() modifier takes a fairly large number of optional arguments, but here you're providing a maximum width and an alignment. What this says is that the picker's width can grow larger than its intrinsic content size, and that its actual drawn content should be centered within the view's width. Without the max-width value, the control's size would be clamped tightly to its content, and the view would be aligned to the left of the screen by the Form—which doesn't look great. Now it's still left-aligned, but it expands its bounding box to fill the screen's width, letting the picker itself center its content.

Launching a Text Editor

Last but not least, you need to add an interface for editing the notes field of your to-do items. Since this can contain arbitrary content with multiple lines, you'll use a NavigationLink to push a full-screen text view. Unfortunately, SwiftUI doesn't yet provide a TextView type. Happily, it's not too difficult to wire in a standard UITextView from UIKit, and if you look in AccessoryViews/TextView.swift you'll find just that. There's some fiddly wiring involved when it comes to correctly mapping the SwiftUI environment across, so we won't go into the details here. You'll learn about integrating views from UIKit and AppKit in later chapters, but for now, just pretend that TextView is the same as any other View type provided by SwiftUI.

Before creating the navigation link, you can make things a little easier on yourself by defining a new property on TodoItemEditor to create and return the text view. Recall that a NavigationLink takes a View as an argument to its initializer, and chaining multiple view modifier calls inside of a parameter declaration of another function call quickly becomes ungainly. Instead, create the following property in TodoItemEditor:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift

```
var notesEditor: some View {
    TextView(text: self.$item.definiteNotes)
        .padding(.horizontal)
        .navigationBarTitle("Notes: \(item.title)")
}
```

This wraps up the TextView initializer (bound to the new definiteNotes property on TodoItem) and the couple of modifier calls you need to attach a navigation bar title and a little padding on the leading and trailing edges of the screen. With that in place, you can return to the body implementation and add the final piece of the form:

3-ModifyingData/final/Do It/Views/TodoItemEditor.swift
```
NavigationLink("Notes", destination: notesEditor)
```

With that done, the editor view itself is now complete. Now, you just need to wire it into the detail view.

Presenting Modal Views

The editor you've created is designed to operate without maintaining its own state. It operates via @Binding on data owned by someone else and doesn't make assumptions about 'live' versus 'draft' data, nor about saving versus cancellation of the results. Making fewer assumptions about its use makes it a more flexible component, but it means that anyone presenting it needs to make those determinations themselves.

The detail view is going to present the editor in a modal sheet, wrapped in the NavigationView required by the editor's Form view. It will add Cancel and Done buttons to the editor's navigation bar, and those buttons will operate on the state managed by the detail view. The editor itself will be bound to a local copy of the to-do item being presented; if the user taps Cancel, the local copy is discarded, while a tap on Done saves it to the data store, effectively committing the changes.

State

The detail view will need a few new properties to maintain the state required. Open TodoItemDetail.swift and add the following properties to the TodoItemDetail view:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift
```
@State private var editingItem: TodoItem = .empty
@State private var showingEditor = false
```

The first line here defines the local copy of the item to be manipulated by the editor. It's a @State variable so you can generate a binding for the editor to use. However, since this is an implementation detail, you must provide a default value for it; otherwise, anyone creating a TodoItemDetail will be required to pass a value into its initializer. Since the initial value will be unused, you're assigning it to an 'empty' value that you'll define in a moment.

The next line is used to present the editor itself. Recall that in SwiftUI, views are a function of state, and all changes to the UI are driven by changes to state data. This continues with the concept of presenting a modal view: your state needs to include a value defining whether the modal view is shown or

not. Here is that variable, and you'll see shortly how it's used to present and dismiss the modal view containing the TodoItemEditor.

Right now you should have a compiler error. Xcode is informing you the .empty value assigned to TodoItemDetail.editingItem isn't defined anywhere. To remedy that, add the following file-private extension to the TodoItem type at the top of TodoItemDetail.swift:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
fileprivate extension TodoItem {
    static var empty = TodoItem(title: "", priority: .normal,
                                listID: .null)
}
```

Now your code compiles happily.

Presentation

Next, you need to implement the scaffolding around the editor view. As described in Presenting Modal Views, on page 75, you're going to embed the editor in a navigation view with a pair of buttons to either commit or cancel the pending edit. Let's start by creating those buttons, beginning with cancellation. Add the following property to TodoItemDetail:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
Line 1 private var cancelButton: some View {
  2     Button(action: {
  3         self.showingEditor.toggle()
  4     }) {
  5         Text("Cancel")
  6             .foregroundColor(.accentColor)
  7     }
  8 }
```

This is a straightforward button with the simple task of dismissing the editor. It accomplishes this task by modifying the showingEditor state value on line 3. This will trigger SwiftUI to update the view content, dismissing the modal sheet containing the editor.

Next comes the Done button, another property on TodoItemDetail:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
Line 1 private var doneButton: some View {
  2     Button(action: {
  3         self.data.updateTodoItem(self.editingItem)
  4         self.showingEditor.toggle()
  5     }) {
  6         Text("Done")
  7             .bold()
```

```
8               .foregroundColor(.accentColor)
9       }
10  }
```

The button's action here also toggles showingEditor to dismiss the modal sheet, but before that, it needs to save the changes. The editor has been operating on a local copy of the data, while the source of truth lives in an Array in the DataCenter within the environment. To save the to-do item, the value *in that array* needs to be updated, so you use a DataCenter API call to do that on line 3.

You likely have noted that you're not altering the value of the TodoItem being used to present the detail view itself. In fact, looking at the declaration, it's an immutable property. You wouldn't be amiss to think that when you exit the editor, you'll see the same old data you started with since that hasn't changed.

SwiftUI is way ahead of us here. By updating the array in the DataCenter, all views based on that data are recomputed. First, the TodoItemList view has its body invoked, and that then re-defines the TodoItemDetail views that are the targets of each row, including the one currently on-screen. SwiftUI maps all these values onto the existing visible view hierarchy, triggering redraws of their content, so the detail view does have its content modified, but it's SwiftUI that does the work of propagating those changes for you.

At this point, you have two buttons—let's give them somewhere to live. Define the content of the modal editor sheet with the following new property on TodoItemDetail:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
private var editor: some View {
    NavigationView {
        TodoItemEditor(item: $editingItem)
        .navigationBarItems(leading: cancelButton,
                            trailing: doneButton)
    }
}
```

With this code, you've created a TodoItemEditor, binding it to the editingItem property, and you've wrapped it in a navigation view. The two buttons are attached to the editor view as navigation bar items; Done on the trailing edge, Cancel on the leading.

Next, you need to provide the user with some way to invoke the editor, and then actually present it.

Launch the Editor

It's common to have an Edit item on the trailing edge of the navigation bar for editable views. You'll follow this convention, using the pencil-in-a-square icon commonly used to represent edit actions. Once more, you're going to place this in a new property within TodoItemDetail to allow for more brevity and clearer code within the view's body:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
private var editButton: some View {
    Button(action: {
        self.editingItem = self.item
        self.showingEditor.toggle()
    }) {
        Image(systemName: "square.and.pencil")
            .imageScale(.large)
            .foregroundColor(.accentColor)
    }
    .accessibility(label: Text("Edit"))
}
```

The button's action performs two operations: it starts by setting the value of the editingItem property that will be bound to the editor. Doing this at the last moment ensures the most up-to-date data is used. Next, the showingEditor property is toggled, informing SwiftUI that the modal sheet is to be displayed (you'll wire up that functionality momentarily).

Note also a new modifier being used on the button: .accessibility(label:). Since the edit button only uses an icon, any user with visual impairments would benefit from a more descriptive name, and this modifier attaches just that. When a user activates VoiceOver to describe what's on the screen, it will announce that this is a button labeled "edit." Without this, the response would likely be something like "a button with an image: square and pencil," which isn't particularly illuminating.

Your last task is to update the body implementation to add the edit button and present the editor itself. The edit button is added to the navigation bar using the familiar .navigationBarItems(trailing:) method, but presenting a modal view requires a new tool: .sheet(isPresented:content:).

In SwiftUI parlance, a full-screen modal view is called a *sheet*. Their presentation is—like everything in SwiftUI—tied into the values of @State properties. A couple of different options exist, but the one you'll use here simply binds to a boolean value that toggles presentation of the view on an off: while the value is true, the modal view is presented, and while false it is not. SwiftUI

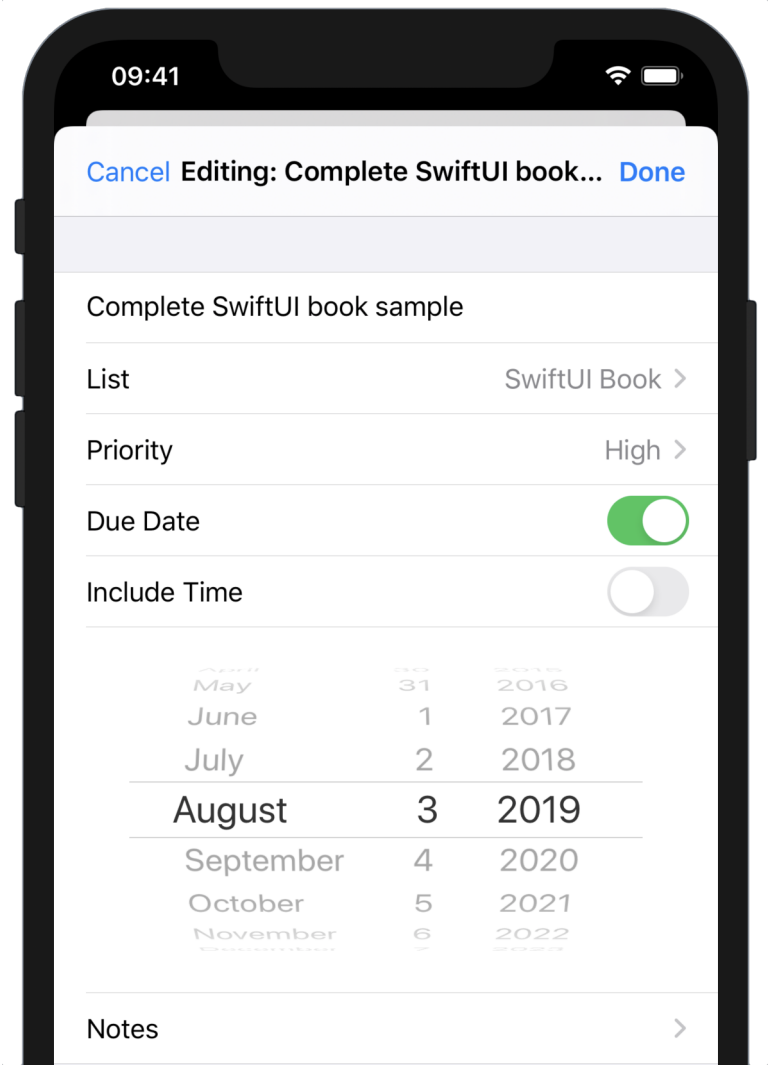
automatically uses the appropriate transition to animate its arrival and departure.

All this is done by appending two modifiers to the content of TodoItemDetail.body:

3-ModifyingData/final/Do It/Views/TodoItemDetail.swift

```
var body: some View {
    VStack(alignment: .leading) {
        // « view content »
    }
    .navigationBarTitle("", displayMode: .inline)
➤   .navigationBarItems(trailing: editButton)
➤   .sheet(isPresented: $showingEditor) {
➤       self.editor.environmentObject(self.data)
➤   }
}
```

With that in place, you're ready to roll.



Test the Code

Your application is now ready to go. Launch it in the Simulator or on an iPhone and try changing a few things. Note that edits you confirm are visible when you return to the detail and list views, and even survive after you quit and relaunch the app.

What You Learned

You've now dug deeper into the state management system of SwiftUI and are beginning to see the edges of the "view as a function of state" approach it uses. You have some further tools at your disposal:

- You know when and how to use `@State`, `@Binding`, `@ObservedObject`, and `@EnvironmentObject` to make state available across your applications.
- You understand the implementation of property wrapper types and can deal with them explicitly when necessary.
- You've worked with the `Form` type, and with several types of UI controls.
- You can present and manage modal interfaces via the `.sheet()` operator.
- When it comes to accessibility, a little can go a long way. By keeping in mind how your application might be described via VoiceOver, you can ensure you provide a good experience to all your users.

In the next chapter, you'll look at the facilities provided for editing collections of data, dynamically adding, removing, and re-ordering items, and you'll work with filtering list data at runtime.

CHAPTER 4

List Mutation

Story Map

Why do I want to read this?

While you've seen how to edit individual items, you haven't yet seen how you might perform mutations on collections themselves.

What will I learn?

You'll learn about sections and section headers. You'll see how you can easily implement adding, removing, and moving members of a collection. You'll see how to update collection views to support specific subsets of your data model.

What will I be able to do that I couldn't do before?

You'll be able to create and manage lists with all the self-confidence you have when working with `UITableView`, and you'll have some familiarity with more elaborate collection views.

Where are we going next, and how does this fit in?

Next you'll look at the SwiftUI gesture system and how you can define your own custom interactions, and how the power tools of SwiftUI let you work with geometry information.

In earlier chapters, you built a working to-do list application, complete with presentable detail views, colorful lists, and complex editors. Your users can find existing items and change them, but that's about all—there's currently no way to remove items or to add new ones. There's also something to be desired in the list presentation itself, as once a list grows, it will start to be harder to locate items quickly. While the ability to sort the list is helpful, it would be more helpful to quickly focus in on particular subsets of your data, such as which items are due today, or which are overdue. These tasks are what you'll tackle in this chapter.

To work along with this chapter, use a copy of the starter project in code/p6-starter. As in Chapter 5, Custom Views and Complex Interactions, on page 101 it contains some adjustments to tidy up the code, and it includes a few new files. If you're following along with your own project, these files have either been added or modified:

- Affordances/ItemGroups.swift
- Model/DataCenter.swift
- TodoList.swift
- TodoListChooser.swift has been renamed to Home.swift, and TodoListChooser renamed to Home to match.
- SceneDelegate.swift

Of particular note is a change in TodoList.swift which reorganizes its data storage. Rather than having an optional TodoItemList, it now uses an internal enum type to describe the type of data it represents:

p6-starter/Do It/TodoList.swift

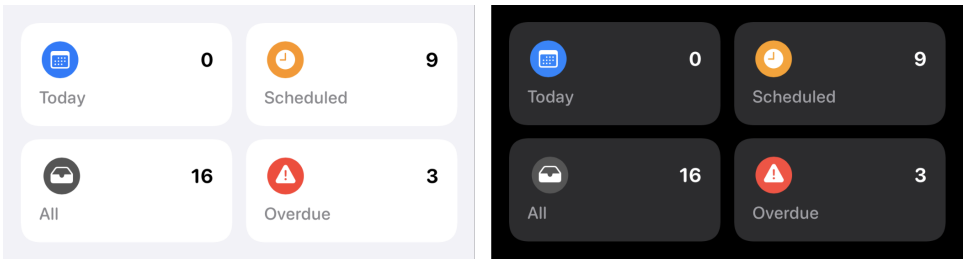
```
private enum ListData {
    case list(TodoItemList)
    case items(LocalizedStringKey, [TodoItem])
}

@State private var listData: ListData
```

You'll update this to include a new type very shortly. Also, in this file, you'll see that the numerous properties and methods are now defined mainly in extensions and that there's even one empty extension definition. You'll update all of these through the course of the chapter.

Using Sections and Header Views

When you created the original TodoListChooser, you used a pair of Section views to break up the content of the list into two parts: one contained a single static row titled "All Items," while the second contained the list rows themselves. The design of the list rows was based on the home screen for Apple's Reminders application, but if you look at that app now, you'll see that the rest of its home screen is quite different. You're going to assemble the same interface here using a *section header* view and design it to look good in both light and dark color schemes:



Each item will act as a navigation link leading to the familiar TodoList view but will refer to a dynamic set of items. To define the data for these groups, the TodoItemGroup enum type has been defined—you'll find it in Affordances/ItemGroups.swift. Open that file and look around; you'll see that it defines four groups and has properties that provide the title, color, and icon views for you to use.

Start on your headers by creating a new SwiftUI View named HomeHeader.swift. At the top of the view's definition, above the body implementation, add the following static property:

p6/Do It/HomeHeader.swift
```
static let layout: [[TodoItemGroup]] = [
    [.today, .scheduled],
    [.all, .overdue],
]
```

This is a simple two-dimensional array that you'll use to populate the four groups with the aid of a pair of ForEach views. This is a fairly straightforward task; here's the main part of the body property definition:

p6/Do It/HomeHeader.swift
```
var body: some View {
    VStack {
        ForEach(Self.layout, id: \.self) { row in
            HStack(spacing: 12) {
                ForEach(row, id: \.self) { group in
                    NavigationLink(destination: TodoList(group: group)) {
                        // « ... »
                    }
                }
            }
        }
    }
}
```

The content is going to take more than a couple of lines to define, so you'll create a private View for that; for the moment, place a Text view inside the NavigationLink:

```
NavigationLink(destination: TodoList(group: group)) {
```

```
➤       Text(group.title)
    }
```

That will fill out the preview nicely, or rather it would—except that Xcode isn't happy with what you just wrote. "There is no TodoList(group:)," it's saying, and … well … it's right. Let's fix that now.

Open TodoList.swift and look at the ListData definition. You're going to need to add a new case to that, and define a new initializer to match. Update the class with the following code:

p6/Do It/TodoList.swift

```
private enum ListData {
    case list(TodoItemList)
    case items(LocalizedStringKey, [TodoItem])
➤   case group(TodoItemGroup)
}

// « Properties »

init(list: TodoItemList) {
    self._listData = State(wrappedValue: .list(list))
}

init(title: LocalizedStringKey, items: [TodoItem]) {
    self._listData = State(wrappedValue: .items(title, items))
}

➤ init(group: TodoItemGroup) {
➤     self._listData = State(wrappedValue: .group(group))
➤ }
```

That will take care of the error in HomeHeader.swift, but it's created some more right here in TodoList.swift. Scroll down to find the lines where Xcode is reporting errors, at the tail end of the "Helper Properties" extension. There are three computed properties here that use the listData property to determine their results. Update them to return values for the new .group case:

p6/Do It/TodoList.swift

```
private var items: [TodoItem] {
    switch listData {
    case .list(let list): return data.items(in: list)
    case .items(_, let items): return items
➤   case .group(let group): return group.items(from: data)
    }
}

private var title: LocalizedStringKey {
    switch listData {
    case .list(let list): return LocalizedStringKey(list.name)
    case .items(let name, _): return name
➤   case .group(let group): return group.title
```

```
    }
}

private func color(for item: TodoItem) -> Color {
    switch listData {
    case .list(let list): return list.color.uiColor
    case .items: return data.list(for: item).color.uiColor
    case .group(let group): return group.color
    }
}
```

Xcode is now happy, and you can return to your header view definition. Open HomeHeader.swift.

Dynamic View Content

The individual items in your header will have a simple implementation, consisting of a white background with rounded corners topped with two horizontal rows containing data. The top row will contain a colored icon on the leading edge with the count of matching items on the trailing edge, in large clear text. Below that will be the name of the group, slightly muted with a secondary color. Both the counter and name will use a rounded font variant, with a bold and medium appearance, respectively.

Add a new View type inside the HomeHeader definition, below the body property:

p6/Do It/HomeHeader.swift

```
Line 1  private struct HeaderItem: View {
        -     let group: TodoItemGroup
        -     @State var itemCount: Int = 0
        -
        5     var body: some View {
        -         VStack(alignment: .leading) {
        -             HStack {
        -                 group.icon
        -                 Spacer()
        10                Text("\(itemCount)")
        -                     .foregroundColor(.primary)
        -                     .font(.system(.title, design: .rounded))
        -                     .fontWeight(.bold)
        -             }
        15
        -             Text(group.title)
        -                 .foregroundColor(.secondary)
        -                 .font(.system(.subheadline, design: .rounded))
        -                 .fontWeight(.medium)
        20        }
        -         .padding()
        -         .background(
        -             RoundedRectangle(cornerRadius: 15, style: .continuous)
```

```
                .fill(Color(.tertiarySystemBackground))
        )
    }
}
```

Everything here should look familiar by this point, with the exception perhaps of the color on line 24. Here, you're again using one of UIKit's semantic colors, specifically for something layered on top of a secondary background. The List view's grouped background is considered secondary, so the tertiary color works well here, and matches what Apple uses in their Reminders app.

There's something missing here, though: the counter text is never updated. It's stored in a state property, so the view will update when it changes, but it never actually changes. You might add a custom initializer to set its value from the input group on creation, but that requires access to a DataCenter environment object. Well, you could add one of those just as you have elsewhere, but that would reveal another problem: it wouldn't work within the view's initializer. Because the environment object is installed through a view modifier, the underlying view will already have been created *and initialized* before the .environmentObject() call happens. If you try to read from an environment object inside your initializer, you're likely to crash—certainly in the preview, and possibly elsewhere in your app, depending on how SwiftUI builds and initializes its view hierarchy. That's all out of your control, though, so you can't rely on that behavior everywhere.

Of course, if you *were* to set your itemCount when initialized, what would happen if the data changed? Let's say you changed the date on an overdue item to today? The count for the "Overdue" group should decrement, and the "Today" group's should increment, but there's no guarantee that will happen. There is a solution, however, and it'll work in every situation, meaning each HeaderItem will be entirely in control of its own dynamic content, not relying on any other view's updates, or on SwiftUI's rendering mechanisms.

Remember that DataCenter.todoItems uses the @Published property wrapper. This wrapper's projected value, $todoItems, returns a Publisher instance from the *Combine* framework. Combine is Apple's suite of reactive programming tools, implementing a *publisher-subscriber* API for passing data and events around your application. SwiftUI is using this under the covers to determine when your item lists are changing, and their related views need to be updated; there's more you can do with them, though. You can attach to publishers directly by yourself and handle the values they publish when that happens; this sounds like exactly the tool you need.

If you subscribe to the output of the $todoItems publisher, then your subscriber will immediately be given that publisher's current value. After that, whenever the list of items changes, your subscriber will receive the new set. You can use this to update your item count, recalculating its value any time any items are modified. This is such a common occurrence, in fact, that SwiftUI provides a view modifier for just this purpose: .onReceive(_:perform:). The method takes a Publisher instance and a block which will be passed the output of the publisher, handling all the subscription details internally.

To implement this approach, you'll need to read the DataCenter from the environment and respond to changes in the item list by asking your group for its matching items, then assigning the number of items within that collection to your itemCount state property. The .onReceive() modifier can simply be attached to the existing VStack:

p6/Do It/HomeHeader.swift

```
    @EnvironmentObject var data: DataCenter

    var body: some View {
        VStack(alignment: .leading) {
            // « ... »
        }
        .onReceive(data.$todoItems) { _ in
            self.itemCount = self.group.items(from: self.data).count
        }
        // « ... »
    }
}
```

Your header item view is complete—you can drop it into place inside HomeHeader.body now:

p6/Do It/HomeHeader.swift

```
NavigationLink(destination: TodoList(group: group)) {
    HeaderItem(group: group)
}
```

Previewing

If you refresh your canvas now, you'll see your header view showing up in the correct proportions, but the backgrounds won't show up—they're white on a white background. Let's fix that and take a look at how it appears in dark mode at the same time. Replace the content of the view's preview with this familiar-looking implementation:

p6/Do It/HomeHeader.swift

```
Line 1 ForEach(ColorScheme.allCases, id: \.self) { colorScheme in
  2     HomeHeader()
```

```
3           .padding()
4           .background(Color(.systemGroupedBackground))
5           .colorScheme(colorScheme)
6   }
7   .previewLayout(.sizeThatFits)
8   .environmentObject(DataCenter())
```

The most important part here is the .background() modifier on line 4: the color used here is the one used by the List view when it's using a grouped style, and by using it here you can see exactly how the header will appear when you add it to your Home view.

Modifying List Data

Open TodoList.swift and look around to familiarize yourself with its content. In the starter project for this chapter, the implementation has been shuffled around a bit in the interests of readability. The main struct definition now contains only properties and the body implementation, with everything else moved into extensions. Firstly, there's an extension marked "Helper Properties;" the sortButton implementation has been moved here, and you'll add some more shortly. The next extension, marked "Sorting," contains the sortedItems property. Below that, there's an empty extension marked "Model Manipulation," which is waiting for some content.

Right at the bottom of the file is the SortOption type. Start by adding a new case to this enumeration, named manual:

p6/Do It/TodoList.swift

```
fileprivate enum SortOption: String, CaseIterable {
    case title = "Title"
    case priority = "Priority"
    case dueDate = "Due Date"
➤   case manual = "Manual"

    var title: LocalizedStringKey { LocalizedStringKey(rawValue) }
}
```

You likely have a compiler error showing up now, pointing to the lack of a clause for the new .manual case in the sortedItems property. This is simple to fix, since the manual sort option essentially means "don't sort anything, use the items in their existing order." Scroll up to the sortedItems property implementation and update it:

p6/Do It/TodoList.swift

```
private var sortedItems: [TodoItem] {
➤   if case .manual = sortBy { return items }

    return items.sorted {
```

```
            switch sortBy {
            case .title:
                return $0.title.lowercased() < $1.title.lowercased()
            case .priority:
                return $0.priority > $1.priority
            case .dueDate:
                return ($0.date ?? .distantFuture) <
                    ($1.date ?? .distantFuture)
            case .manual:
                fatalError("unreachable")
            }
        }
    }
```

You handle the .manual case upfront by returning the input item list unchanged. In the switch statement, though, you still need a clause handling that case, so here you simply fire off a fatal error—it shouldn't be possible to reach here, and if it does, something very bad is happening, and you want it to crash and dump lots of useful info in the process.

Next, head up to the top of the file and look at the state variables. Right now you have three items there: sortBy, showingChooser, and showingListEditor. The default sort option should now be .manual, so make that change now:

p6/Do It/TodoList.swift

```
@State private var sortBy: SortOption = .manual
@State private var showingChooser: Bool = false
@Environment(\.presentationMode) private var presentationMode
```

Adding to the List

The first and easiest editing function you can implement is addition. You already have an editor view for TodoItems—all you need is the means to trigger its appearance and a way to save it. This can all function in exactly the same manner as you used in the previous chapter, with a state property used to present the editor in a modal sheet and a state variable used to hold a temporary TodoItem for the editor to use.

Scroll to the main struct definition for TodoList, and add the following properties:

p6/Do It/TodoList.swift

```
Line 1 private static let itemTemplate = TodoItem(
   2     id: Int.min, title: "New Item", priority: .normal,
   3     notes: nil, date: nil, listID: 2002, complete: false)
   4
   5 @State private var editingItem = Self.itemTemplate
```

The two @State property is familiar, but the item on line 1 is new. This provides a simple starting value for a new TodoItem instance, and is used to initialize

the editingItem property each time a new item is added. Note that its id property is set to something essentially invalid to begin with; the DataCenter will replace this when the new item is placed in the list.

Presenting the item editor will be a little different than what you've seen before, though. It seems logical to add a new .sheet(isPresented:content:) modifier next to the one used to present the list editor from the last chapter, but it so happens that the isPresented: variant only works when it's alone; if you have two of them, only the last one will function. This likely indicates that SwiftUI is using the environment internally to set this up; thus, one modifier is overriding the value placed by another.

Instead, you'll use the more flexible version, .sheet(item:content:), which accepts a binding to an optional Identifiable value that you'll use to determine what to present. When the bound value is nil, nothing will be displayed. When it's non-nil, its value will be passed to the content block, which should then return a view.

To implement your value, use an enum type with two values, conforming to Identifiable and Hashable, then create a state property to hold one of these, with a default value of nil:

p6/Do It/TodoList.swift

```
private enum EditorID: Identifiable, Hashable {
    case itemEditor
    case listEditor

    var id: EditorID { self }
}

@State private var presentedEditor: EditorID? = nil
```

Now, scroll down to the "Helper Properties" extension, and below the sortButton property implementation, add two new properties:

p6/Do It/TodoList.swift

```
private var addButton: some View {
    Button(action: {
        self.editingItem = Self.itemTemplate
        self.presentedEditor = .itemEditor
    }) {
        Image(systemName: "plus.circle.fill")
            .imageScale(.large)
            .accessibility(label: Text("Add New To-Do Item"))
    }
}

private var editorSheet: some View {
    let done = Button(action:{
        self.data.addTodoItem(self.editingItem)
```

```
            self.presentedEditor = nil
        }) {
            Text("Done")
                .bold()
        }
        let cancel = Button("Cancel") {
            self.presentedEditor = nil
        }
        return NavigationView {
            TodoItemEditor(item: $editingItem)
                .navigationBarItems(leading: cancel, trailing: done)
        }
    }
```

Here, you have a Button for the navigation bar that will present an editor for a new TodoItem instance. Its action resets the editingItem to the template created above, then it shows the editor sheet. Note the use of the .accessibility(label:) view modifier to provide a textual description to go with the image-based button.

The editor itself is defined in editorSheet; this creates the "Done" and "Cancel" buttons similar to the ones you used in Presenting Modal Views, on page 75 and attaches them to a TodoItemEditor wrapped in a NavigationView. The only difference is the action for the Done button, which now calls data.addTodoItem() to place the new TodoItem into the data store. This should all look quite familiar by this point.

The last step is to attach the new button to the navigation bar. Since the next step will be to add general editing support for the list, you can add a standard Edit button to the bar's leading edge at the same time. Find the barItems property and update it to include two more items:

p6/Do It/TodoList.swift

```
private var barItems: some View {
    HStack(spacing: 14) {
        if isList {
➤           Button(action: { self.presentedEditor = .listEditor }) {
                Image(systemName: "info.circle")
➤                   .imageScale(.large)
            }
        }
        sortButton
➤       addButton
➤       EditButton()
    }
}
```

The EditButton is a view provided by SwiftUI, which toggles the editing mode stored in the environment; you'll see its effects soon. If you refresh your Xcode

canvas, you'll see that and the new Add button, but the appearance looks a little off next to the Sort button. Find the sortButton implementation and change it to use a similar circular filled symbol:

p6/Do It/TodoList.swift

```
Button(action: { self.showingChooser.toggle() }) {
    Image(systemName: "arrow.up.arrow.down.circle.fill")
        .imageScale(.large)
        .accessibility(label: Text("Sort List"))
}
```

That looks better. Now, to present your editors when the buttons are tapped, find the existing .sheet() modifier attached to the List view in your body implementation, then replace it with the following:

p6/Do It/TodoList.swift

```
.sheet(item: $presentedEditor) { which -> AnyView in
    switch which {
    case .itemEditor:
        return AnyView(
            self.editorSheet
                .environmentObject(self.data)
        )
    case .listEditor:
        return AnyView(
            TodoListEditor(list: self.list!)
                .environmentObject(self.data)
        )
    }
}
```

Note that the block has to always return the same view type. Since you're presenting either a TodoItemEditor or a TodoListEditor, you have to wrap them in an AnyView to ensure you're always returning an instance of the same type.

If you start up a live preview now, you'll be able to call up an editor and create new items to your heart's content, while the list editor button will still function as before. All that remains is to implement support for deleting and manually reordering items, both of which come built-in.

Deleting and Moving List Items

The EditButton you added to the navigation bar provides access to controls to delete and reorder rows, but to access the functionality, you'll need to enable it through some view modifiers. The .onDelete() modifier provides a block to run in response to a deletion interaction, and it passes in an IndexSet containing the indices of all the items being removed. Similarly, the .onMove() modifier is called when the user drags to move rows from one location to another within

the list. This block receives the offsets of the items being moved and the single index to which they should go.

Happily, SwiftUI provides convenience functions on indexable Swift collections (such as Array) that take the same inputs and will perform the work for you. All you need to provide are the blocks for the two modifiers, which seems simple enough.

At least, it would be—if you weren't displaying a list that's been re-ordered, meaning the indices of the visible rows and their indices within DataCenter.todoItems are not the same. Picking a non-manual sort order, then deleting items leads to some strange behavior if you pass through the IndexSet unmodified; your humble author found himself scratching his head in confusion for some time before realizing what was going on.

Scroll down in TodoList.swift to find the empty extension titled "Model Manipulation." If you're not using the starter project directly, create a new extension on the TodoList class. Inside here, you'll implement both the methods used to translate from visible-row indices to data-list indices and the methods that will move and delete items from your data source.

Start with the translation methods:

p6/Do It/TodoList.swift

```
private var usingUnchangedList: Bool {
    sortBy == .manual
}

private func translate(offsets: IndexSet) -> IndexSet {
    guard !usingUnchangedList else { return offsets }
    let items = sortedItems // calculate this just once
    return IndexSet(offsets.map { index in
        data.todoItems.firstIndex { $0.id == items[index].id }!
    })
}

private func translate(index: Int) -> Int {
    guard !usingUnchangedList else { return index }
    return data.todoItems.firstIndex { $0.id == sortedItems[index].id }!
}
```

There are three items here. First is a property that will return true if the list is manually sorted. If that's the case, then the visual indices and the data indices match, and no translation needs to happen.

The next two methods translate either a single index or an entire IndexSet. They function by using the provided index to locate an item within the visibleItems array, then look up the item with the same id within data.todoItems.

With this in place, you can implement the delete and remove methods themselves:

p6/Do It/TodoList.swift

```
private func removeTodoItems(atOffsets offsets: IndexSet) {
    let realOffsets = translate(offsets: offsets)
    data.removeTodoItems(atOffsets: realOffsets)
}

private func moveTodoItems(fromOffsets offsets: IndexSet, to newIndex: Int) {
    let realOffsets = translate(offsets: offsets)
    let realIndex = translate(index: newIndex)
    data.moveTodoItems(fromOffsets: realOffsets, to: realIndex)
}
```

Each of these translates the supplied indices and then calls through to a method in DataCenter (provided in this chapter's starter project), which in turn simply calls the SwiftUI-provided collection methods in a thread-safe manner.

Now only one step remains—adding the view modifiers. Note, however, that you only want to support moving items while using the manual sort ordering; moving items around while sorted by date rather defeats the purpose of sorting them. As it happens, the .onMove() modifier can be passed nil to disable reordering completely, so you'll check the current sortBy value to determine whether to enable that feature, and pass nil in all other cases.

Find the ForEach view definition in TodoList.body and add these view modifiers after its closing brace:

p6/Do It/TodoList.swift

```
Line 1 .onDelete { self.removeTodoItems(atOffsets: $0) }
  2 .onMove(perform: self.sortBy == .manual
  3     ? { self.moveTodoItems(fromOffsets: $0, to: $1) }
  4     : nil)
```

In the .onMove() modifier on line 4 a ternary operator is being used to either install a handler or to set its value to nil. This is because you only want to allow the user to re-order items when in manual ordering mode. By passing nil in every other case, you disable the move functionality; by using the ternary operator to make this decision inline, you help SwiftUI detect that this particular modifier's value is driven by the sortBy state property. When in doubt, it's best to try and make these calculations happen at the same time you hand their results into SwiftUI because the framework can infer a lot of information about the view graph and its relation to the underlying data based on seeing when certain data is accessed while the view graph is being assembled.

Coming Home

Now let's do the same for the Home view. First, you'll need the usual state variables used to present the sheet. You'll need a template TodoItemList to pass into the editor, and you'll need a button used to create the new item.

Open Home.swift and update the content of the Home view:

p6/Do It/Home.swift

```
Line 1  static private let listTemplate = TodoItemList(
        -     id: Int.min, name: "New List", color: .blue, icon: "list.bullet")
        - @State private var showingEditor = false
        -
        5 var body: some View {
        -     NavigationView {
        -         // « ... »
        -     }
        -     .sheet(isPresented: $showingEditor) {
       10         TodoListEditor(list: Self.listTemplate)
        -     }
        - }
        -
        - private var addButton: some View {
       15     Button(action: { self.showingEditor.toggle() }) {
        -         Image(systemName: "plus.circle.fill")
        -             .imageScale(.large)
        -             .accessibility(label: Text("Add New List"))
        -     }
       20 }
```

This is all familiar. Note on line 10 that since the TodoListEditor doesn't operate on a binding, it is passed the listTemplate directly. Being a value type, the editor receives and operates on a copy of the template.

Implementing delete and move operations for the lists is simpler than it was for the TodoList view, since the content of the view is essentially always manually ordered. This means that you can dispense with index translations and pass the indices from SwiftUI directly into the DataCenter. Add the view modifiers to the end of the ForEach view, as before, and add navigation bar items for addButton and an EditButton:

p6/Do It/Home.swift

```
List {
    Section(header: HomeHeader().padding(.vertical)) {
        ForEach(data.todoLists) { list in
            NavigationLink(destination: TodoList(list: list)) {
                Row(name: list.name,
                    icon: list.icon,
                    color: list.color.uiColor)
            }
```

```
            }
➤           .onDelete { self.data.removeLists(atOffsets: $0) }
➤           .onMove { self.data.moveLists(fromOffsets: $0, to: $1) }
        }
    }
    .font(.system(.headline, design: .rounded))
    .listStyle(GroupedListStyle())
    .navigationBarTitle("Lists")
➤   .navigationBarItems(leading: EditButton(), trailing: addButton)
```

Manually Following Changes in Data

Following the principles described in Dynamic View Content, on page 87, the TodoList view should own the task of observing changes to its content. While its list content usually comes from DataCenter.todoItems, which is already monitored by SwiftUI, that's not always the case, and isn't the case for everything. Looking at the ListData type, for instance, one can see several cases where model value types have been copied out of the DataCenter. ListData.items contains a copied list of TodoItem instances. If one of these items is deleted from the data store, nothing happens to the copy being used by this view. Similarly, while the TodoItemList held by ListData.list will fetch items through the DataCenter, the same can't be said for the list itself: its title, icon, and color might change, leaving the TodoList with incorrect data.

The solution to this is to update the static data inside the listData state property when the underlying data store changes. In the case of ListData.group, nothing needs to be done, since everything is fetched dynamically from the DataCenter. For both .list and .items cases, however, you'll need to fetch any updates and apply them to the local values.

Start by implementing the update method. Place this inside the TodoList extension titled "Model Manipulation," just below moveTodoItems(fromOffsets:to:):

p6/Do It/TodoList.swift

```
Line 1  private func updateData() {
        -   switch listData {
        -   case let .items(title, items):
        -       let newItems = data.items(withIDs: items.map { $0.id })
        5       listData = .items(title, newItems)
        -
        -   case let .list(list):
        -       if let newList = data.todoLists.first(where: { $0.id == list.id }) {
        -           listData = .list(newList)
        10      }
        -       else {
        -           // List is gone!
        -           forciblyDismiss()
```

```
-          }
15
-      case .group:
-          break
-      }
-  }
```

On line 3 you handle the case where a specific collection of items is being used. The identifiers of the included items are used to fetch fresh copies from the DataCenter, which will leave out any that have been removed from the store. The new items are used to reset the listData property.

Meanwhile on line 7, you handle any changes to the TodoItemList copy being held. Here you fetch the list with the matching identifier and use that to reset listData. Here, however, a particular situation arises: while the view might legitimately display that a certain list has no items, it can't reasonably do the same thing for the list itself. If the list it's displaying is deleted, this view should no longer exist. It's for that reason that you call forciblyDismiss() on line 13; that method will use the current PresentationMode from the environment to pop the TodoList from the top of the navigation stack.

With all the possibilities of stale data taken care of, you now need only to call updateData() when the underlying store changes. This can be done by taking an idea from DataCenter itself, using its @Published properties to trigger an action via the .onReceive() view modifier. Attach this modifier to end of the List view in the body implementation, after the .sheet() modifier:

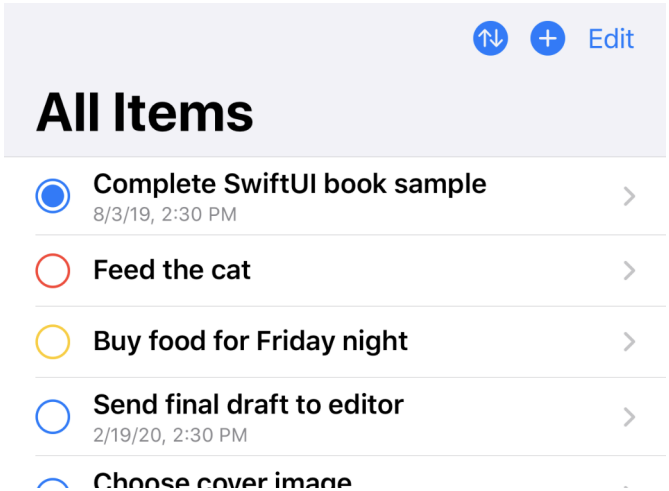
p6/Do It/TodoList.swift

```
@State private var listData: ListData
// « ... »

var body: some View {
    List {
        // « ... »
    }
    // « existing view modifiers »
➤   .onReceive(data.$todoItems.combineLatest(data.$todoLists)) { _ in
➤       self.updateData()
➤   }
}
```

Here, you've used the .combineLatest() modifier from the Combine framework to be signaled when either the todoItems or todoLists properties change. Another example of this can be found in DataCenter.swift, in the saveWhenChanged() method.

At this point, you're done—congratulations. You now have a fully editable to-do list.



What You Learned

This chapter has touched on several important items of SwiftUI design:

- You learned to manually react to changes in data with the aid of publishers.
- You worked with editable lists, adding, removing, and reordering items, correctly handling cases where on-screen and data-store locations don't match.
- The application's home screen now has a bespoke interface tailored into the List itself.
- Your list view is more flexible than ever, supporting several different types of initialization, used for several purposes.

Now that your lists are looking good, you'll spend some time working with custom views—going beyond simple stacks of other views—and how to define and handle user interactions through SwiftUI's gesture system.

CHAPTER 5

Custom Views and Complex Interactions

Story Map

Why do I want to read this?

User interaction isn't all simple pickers and text fields. You need to be able to design and implement more complex reactive interfaces. You will also need to understand the basic building blocks of custom views, like shapes and gradients, and how to put them to use.

What will I learn?

You'll see how to factor out code into reusable modules called view modifiers, and you'll make more use of the button styles used earlier in the book. Most importantly, you'll learn how to use SwiftUI's anchor system of coordinate management and translation through a new pair of controls cut from whole cloth.

What will I be able to do that I couldn't do before?

You'll be able to respond to direct user input and use that to inform the layout and appearance of your views and their underlying data in new and novel ways. You'll have an understanding of the tools necessary to implement direct feedback and absolute positioning, which worked quite differently in the iterative model.

Where are we going next, and how does this fit in?

In the next chapter you'll look at the preview functionality of the Xcode canvas and how it can help you rapidly diagnose and solve a variety of issues.

At this point, you now have a working application that displays and edits to-do items. Your detail view is presentable, and your editor makes good use of both animation and VoiceOver labels to provide important cues to your users. But it's missing something: there's no way to make changes to any of the

properties of a TodoItemList. Unlike text fields, sliders, and pickers, the editable content of a list doesn't rely much on the built-in editor controls. In this chapter, you're going to create that missing editor, providing the means to update the list's name, icon, and color. In the process, you'll build a fully-functional color picker and learn about the tools SwiftUI provides for managing coordinates systems.

This chapter comes with a starter project containing some resources and code that you'll use while building out the new functionality. You can find a complete starter project in the code archive[1] inside the p5-starter folder. Alternatively, you can follow along in your own project by importing the following new and updated files:

- Affordances/
  - Trigonometry.swift
  - HSBWeelHelpers.swift
  - Formatters.swift
- Resources/
  - list-icons.json

Creating Custom Controls

Your list editor will be modeled on the one used by Apple's Reminders application; look at the editor there to see the general layout. Yours will differ slightly, adding a top row containing done/cancel buttons, and providing a fully-featured HSB color picker as opposed to Reminders' small selection. Creating the color picker and icon chooser is your chance to look at some reasonably complex elements of the SwiftUI toolkit and will be extremely useful in the future.

The layout for your editor is going to be something like the following:

1. https://pragprog.com/titles/jdswiftui/source_code

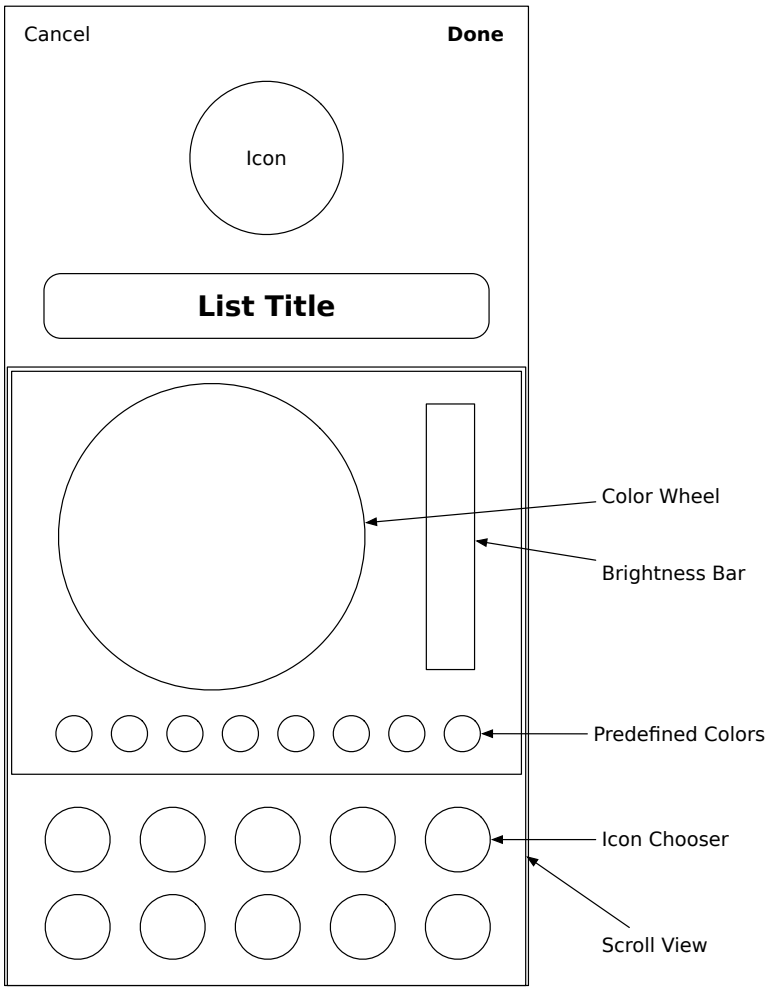


Figure 1—List Editor Layout

At the top of the editor itself are the action buttons used to commit or discard your changes. Below that, the icon is shown in a large circle colored with the list's chosen color, and below that, a text field with a gray background and rounded corners. These elements are all static within the editor.

Below the static icon and title is a `ScrollView`, which will contain two custom views you'll create. The first is a `ColorPicker`, which will let the user pick from a list of predefined colors or choose their own color from a `ColorWheel` view, which you'll also create. Below the color selector will be an `IconChooser` that will display a static list of icons with a circle highlighting the current selection. In both cases, the user will interact with a view, and you'll need to take the

location of their interaction to modify another view. For the icons, when the user selects an icon, you'll need to move the circle to surround that button alone. In the color picker, you'll change the bound color value as the user drags their finger around the well, and will display a large loupe view next to their finger as it moves around. Since SwiftUI uses immutable opaque value types for everything, the means of obtaining and using coordinates is somewhat novel but still flexible. You'll meet the `Anchor`, `GeometryReader`, and `GeometryProxy` types as you assemble this view, and learn to use SwiftUI's preferences system to pass location data around between your views.

Let's start with the color picker.

The most common interface for a color picker on Apple platforms is the HSB wheel. This uses a circular area where all colors are laid out around the circumference, and their saturation decreases closer to the center. Alongside this would be a slider controlling the brightness of the resulting color. These will both make use of some custom touch handling and some coordinate inspection, which aren't as straightforward in SwiftUI as in an iterative framework.

Your color picker will have several components, as shown in Figure 1, List Editor Layout, on page 103:

- A color wheel displaying hue and saturation.
- A bar displaying the brightness of the selected color.
- Several buttons used to select from a list of predefined colors.
- An optional element to display the selected color (since this is a component).

The intent is that the user can tap on one of the buttons to select that color, or they can drag around in the color wheel and the brightness slider to select a color of their own choice. The color wheel will need some means of indicating the location of the user's chosen color, as will the brightness bar. Additionally, while the user is dragging on the color wheel, it should display a larger loupe view following their finger, so they can clearly see which color they're selecting.

One immediate issue occurs: given a SwiftUI `Color`, how does one determine what color it is?

Describing Colors

The `Color` type in SwiftUI is described as a "late-binding token." This has a specific meaning: it doesn't have a concrete value until some time later, usually when it's about to be used. Several things may cause it to change

when it comes time to render it, for example, the color space used by a particular view hierarchy, hue rotations, inversions, and of course, the various accessibility options such as high contrast mode. By using late-binding on all its colors, SwiftUI ensures that these options are supported by all applications—you literally cannot give SwiftUI a color that will ignore these settings. However, this means that the Color type is opaque to those of us outside the library, and we can't get any useful information out of it. So, how to solve this issue?

A partial solution already exists: TodoItemList.Color. This enumerated type contains several predefined color values and one that contains separate hue, saturation, and brightness values to represent any other colors. The color picker you're designing needs three things from a color type:

- A means of reading and writing HSB values.
- A list of predefined color values to display as buttons.
- A SwiftUI Color value to use in the interface.

These requirements are all easily defined using a protocol, and one has been provided for you in Affordances/HSBWheelHelpers.swift:

p5/Do It/Affordances/HSBWheelHelpers.swift

```
protocol ColorInfo: Identifiable {
    static var predefined: [Self] { get }
    var hsb: (Double, Double, Double) { get set }
    var uiColor: SwiftUI.Color { get }
    var localizedName: LocalizedStringKey { get }
}
```

Alongside the ColorInfo type are some helper routines to fetch a color's individual hue, saturation, and brightness values, and an extension for TodoItemList.Color conforming it to the new protocol. Have a look through that implementation to see how it's able to obtain and assign HSB values for all its predefined colors (hint: it uses UIKit's UIColor to do the heavy lifting).

Now you're ready to start working on the picker view itself. This view will be generic, using as its associated value any type that conforms to ColorInfo:

p5/Do It/AccessoryViews/ColorPicker.swift

```
struct ColorPicker<Value: ColorInfo>: View {
    @Binding var selectedColor: Value

    var body: some View {
        // « ... »
    }
}
```

The picker has a lot of components to implement, but the simplest is the set of selector buttons for the different predefined colors. Let's start the view out with that: first add the VStack that will separate the color wheel from the buttons, then use an HStack and a ForEach to iterate over the predefined colors, creating a new button for each:

p5/Do It/AccessoryViews/ColorPicker.swift

```
Line 1  VStack(spacing: 16) {
2       HStack {
3           ForEach(Value.predefined) { color in
4               Button(action: { self.selectedColor = color }) {
5                   color.uiColor
6               }
7           }
8           .frame(maxHeight: 40)
9       }
10  }
```

The buttons will automatically adjust their widths to fit within the screen, but they will eat as much height as they can get. You stop them eating too much room by limiting their height with the .frame(maxHeight:) modifier on line 8.

Their current appearance leaves a little to be desired. The code you've written perfectly and concisely encapsulates their intent, though, so while you can chain on a number of modifiers to change their appearance, that would start to clutter up this clear section of the view. In addition, it would be nice to have a consistent look and feel for all of the components of the color picker, but you want to avoid duplicating code everywhere. SwiftUI provides some tools that help with this aim—one you've already met, ButtonStyle, but there's another more broadly-applicable tool you can use here: ViewModifier.

Creating View Modifiers

Let's say you want to have a drop shadow effect for the items in the color picker to give them the appearance of lifting off the page a little, which will also be useful for the loupe view in the color wheel. You can use the existing .shadow(radius:) modifier to get a gray fading ring around your view, but for a real 3D appearance, you'll need to adjust the offset of the shadow. For an even better look, a common trick is to use not one but *two* drop shadows: one darker, narrower, and close to the original object, and another lighter, further out and softer. This is entirely possible, though it means two calls to the largest of the shadow modifiers, .shadow(color:radius:x:y:). That's a lot of duplicated code, making it an ideal candidate for your first custom ViewModifier.

Create a new SwiftUI View file inside the AccessoryViews group and name it ViewModifiers.swift. Remove the ViewModifiers structure completely, and replace it with the following:

p5/Do It/AccessoryViews/ViewModifiers.swift

```
Line 1  struct DoubleShadow: ViewModifier {
2       var radius: CGFloat = 10.0
3       func body(content: Content) -> some View {
4           content
5               .shadow(color: Color.black.opacity(0.1),
6                       radius: radius, x: 0, y: radius * 1.2)
7               .shadow(color: Color.black.opacity(0.2),
8                       radius: max(radius/10, 1), x: 0, y: 1)
9       }
10  }
```

As you can see, a ViewModifier doesn't define a body property like a View would. Instead, it defines a function that is passed some kind of view, and to which it applies various other modifiers. In this case, you're taking the input view and applying two different shadows; a wider, softer shadow on line 5, and a narrower, darker shadow on line 7.

The modifier itself is designed to be tunable to a certain degree. On line 2 is a radius property with a default value, and this radius is used not only as the basis for the shadow's radius but also for its y-offset, allowing for a wider and softer or narrower and firmer shadow.

To see it in action, let's add a few previews. Inside the previews property of ViewModifiers_Previews, create a Group containing three Circle instances. Give each one a frame of 300 by 300 and a white foreground color, then apply the DoubleShadow modifier to each using the .modifier(_:) method. For the first, use DoubleShadow() unchanged, and for the next two use custom radii of 20 and 6 respectively. Lastly, set the preview layout for the Group to use a fixed area of 350 by 350, to leave some room for the shadow. Your resulting code should look something like this:

p5/Do It/AccessoryViews/ViewModifiers.swift

```
Group {
    Circle()
        .frame(width: 300, height: 300)
        .foregroundColor(.white)
        .modifier(DoubleShadow())

    Circle()
        .frame(width: 300, height: 300)
        .foregroundColor(.white)
        .modifier(DoubleShadow(radius: 20))

    Circle()
```

```
            .frame(width: 300, height: 300)
            .foregroundColor(.white)
            .modifier(DoubleShadow(radius: 6))

        Text("Start Making Things")
            .padding(.horizontal)
            .modifier(BorderedTextField())

        TextField("Title", text: sampleText)
            .modifier(BorderedTextField())
    }
    .previewLayout(.fixed(width: 350, height: 350))
```

Resume the preview in the canvas and look at each in turn to see the modifier's effect. Try commenting out then restoring the second (narrow) shadow inside DoubleShadow.body(content:) to see what a difference it makes.

Custom Button Styling

You've created a nice shadow effect for your buttons to use, but they're still a bit basic—just colored squares. It'd be better if you could create a standard appearance for these buttons and even a special interaction. The custom shadow lends itself to an inset effect on press, for example. You'll return to the ButtonStyle type to implement this, making a reusable component that can be applied to all buttons in an entire view hierarchy.

Return to ColorPicker.swift. Previously you've used a PrimitiveButtonStyle to take over handling of a button's gesture, but here you don't need to change the gesture itself, you only want to style the content. ButtonStyle enables that, giving you access to the label content and a simple isPressed property on which to operate. Aside from the shadow and its changing radius, the appearance will be straightforward: a circular clip shape and an overlay drawing a stroked circle border in white. Add this to ColorPicker.swift above the ColorPicker implementation:

p5/Do It/AccessoryViews/ColorPicker.swift

```
fileprivate struct ColorButtonStyle: ButtonStyle {
    func makeBody(configuration: Configuration) -> some View {
        configuration.label
            .clipShape(Circle())
            .overlay(Circle().stroke().foregroundColor(.white))
            .modifier(DoubleShadow(radius: configuration.isPressed ? 1 : 6))
    }
}
```

You can now apply this style to all of the buttons in the picker at once by attaching a .buttonStyle() modifier to the HStack containing the buttons:

```
HStack {
    « ... »
```

```
}
.buttonStyle(ColorButtonStyle())
```

The small shadow animation is currently the only effect these buttons would appear to have since there's no component displaying the selected color. Let's fix that.

Add a new boolean property to ColorPicker named showSelectionBar, with a default value of false. Then, within the VStack and below the HStack containing the buttons you just created, add a simple Rectangle in the selected color with a white border, double shadow, and some padding. Fix its maximum width and height to 200 and 60 points respectively:

p5/Do It/AccessoryViews/ColorPicker.swift

```
var showSelectionBar: Bool = false

var body: some View {
    VStack(spacing: 16) {
        // « ... »
        if showSelectionBar {
            Rectangle()
                .foregroundColor(selectedColor.uiColor)
                .overlay(Rectangle().stroke().foregroundColor(.white))
                .modifier(DoubleShadow(radius: 6))
                .padding()
                .frame(maxWidth: 200, maxHeight: 60)
        }
    }
}
```

Now you can create a preview and try it out. As before, use a StatefulPreviewWrapper to get a mutable binding to pass into the ColorPicker initializer:

p5/Do It/AccessoryViews/ColorPicker.swift

```
StatefulPreviewWrapper(TodoItemList.Color.blue) {
    ColorPicker(selectedColor: $0, showSelectionBar: true)
}
```

Launch a live preview in the canvas and tap on each of the buttons. Observe the animation of their shadows as they're tapped and the effect that has, and note how the content of the selection bar changes as you tap on different buttons.

Next comes the most interesting part: the color wheel, which will require the use of some new tools.

Working with Anchors

The color wheel is going to be your first entirely custom interactive control. To assemble it, you'll need to handle and track touch input, obtain coordinates based on that input, and feed those coordinates to other views, enabling them to move around. The way SwiftUI does this with its immutable value types is by necessity quite different from the inspect-and-modify approach used elsewhere, so it's an important technique to master.

Let's think about how the color wheel needs to function. In both of its subviews, it should:

- Track the user's input from touch-down to touch-up.
- Directly modify the bound color value as the user's finger moves.
- Display a small yet clear indicator of the currently selected value within the wheel and bar.

Additionally, for the hue/saturation wheel, there is another requirement:

- Display a larger 'loupe' view while the user drags their finger over the different values, changing color to indicate the current value for the user's finger location.

In all of these cases, this relies on being able to convert between HSB values and coordinate locations in two separate views. To get a color, you'll need some way of determining the location of a user's touch, and to indicate the current color, you need to be able to place the indicator at a particular location.

The use of hue, saturation, and brightness allows for some relatively simple trigonometry to convert between color components and screen coordinates. The hue and saturation are contained in the wheel; hue is represented by the angle from the horizontal plane, while saturation is the distance from the center. Brightness uses a third axis, so is represented separately:

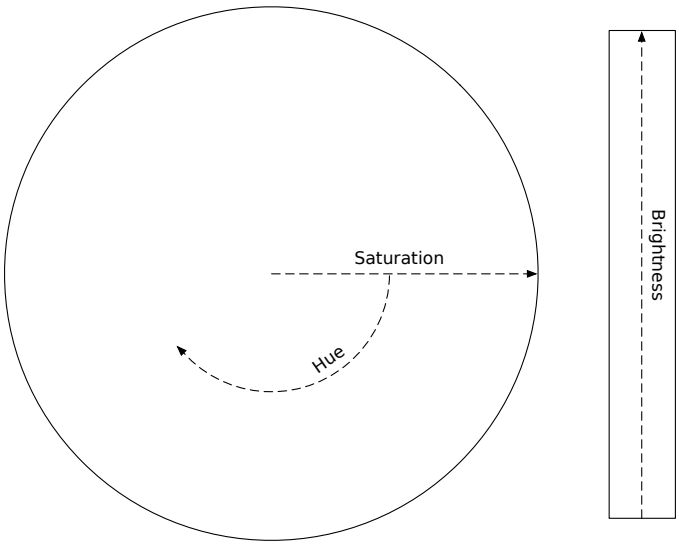


Figure 2—HSB Color Editor Layout

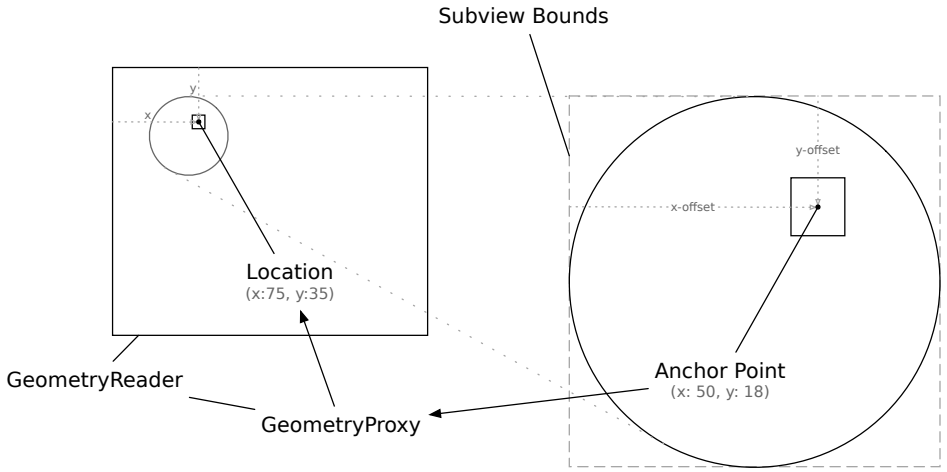
Assigning each axis a value in the range of 0 to 1 (also called *unit space*) allows you to use normal trigonometric operations to convert from angle and distance into coordinates and back again. The routines in Affordances/Trigonometry.swift provide the necessary mathematical operations for that, but still, there's a question of how to obtain a set of coordinates from the user's input or how to place something on screen at a given location. Even further, how does one translate a location from one coordinate space to another?

SwiftUI's answer to these questions is twofold. Firstly, the GeometryReader view provides its content with a GeometryProxy value which vends all sorts of coordinate information. By wrapping a view in a GeometryReader, that view is then able to set its width to exactly half the width of its parent or give itself a 20% offset above its parent's central horizontal axis. For the purposes of the color wheel and brightness bar, the item's size can be used to scale unit space coordinates to locations within the GeometryReader view's bounds, in its local coordinates.

The GeometryProxy has several more tricks up its sleeve, however. It's also the arbiter for any translation of coordinates and locations between different views and coordinate systems. The key to this is the Anchor type.

Anchor is an opaque type in SwiftUI that represents some type of coordinate or location value. This includes points, rectangles, sizes, and the like. SwiftUI can be asked to vend an anchor based on a given Anchor.Source type, for which

there are numerous static methods and properties available based on the type of data represented. An Anchor<CGRect>.Source provides a .bounds property and a .rect(_:) method, for example, while CGPoint-based anchor sources include the various unit coordinates for the edges and corners of a view, along with explicit unit points and physical coordinates. These Source values are used to obtain anchors referencing the local coordinate space, and those anchors can then be given to any GeometryProxy to obtain the same value in the coordinate space of the associated GeometryReader view:



That's almost everything you'll need, except for one wrinkle: passing the Anchor value around your view hierarchy. For this, think back to Indirect, upwards, on page 62. The PreferenceKey type exists for exactly the purpose you need here: carrying information from a subview up to its ancestors. In fact, SwiftUI's tools for obtaining and working with anchors are tied directly into the preference system: you obtain an anchor using the .anchorPreference(key:value:transform:) view modifier. This modifier takes a reference to a PreferenceKey type, an Anchor.Source, and a block used to transform the requested anchor into the value type associated with the preference key.

On top of that, there are several view modifiers that work directly with preference values to generate other views: .backgroundPreferenceValue(_:transform:) will use the current value of a preference to generate a background view, while .overlayPreferenceValue(_:transform:) will let you create an overlay view in the same way. An overlay preference is, in fact, exactly what you need to create the loupe view and have it track the user's finger as it moves.

You now have all of the pieces of the puzzle laid out. All that remains is to assemble them.

Creating and Using Gradients

The two core views of the color picker together use all three types of gradient provided by SwiftUI: linear, angular, and radial. Each has a different appearance, and each has a different role to play in the view.

First, create a new SwiftUI View in the AccessoryViews group and name it ColorWheel.swift. The color wheel will be generic, using the same ColorInfo type defined in HSBWheelHelpers.swift. It will bind to a value and will maintain two items of state for its own use: a boolean used to keep track of whether the user is currently dragging around the loupe and the location of the loupe within this view. The body will start out with an HStack to contain the wheel and the brightness bar, with a little space between them. Inside that will live the GeometryReader that you'll use to drive the wheel's interaction model.

p5/Do It/AccessoryViews/ColorWheel.swift

```
struct ColorWheel<Value: ColorInfo>: View {
    @Binding var color: Value

    @State private var dragging = false
    @State private var loupeLocation: CGPoint = .zero

    var body: some View {
        HStack(spacing: 16) {
            GeometryReader { proxy in
                // « ... »
            }
        }
    }
}
```

The wheel itself consists of two different gradients layered on top of one another. The hue is represented by an *angular gradient*, which will change colors based on the angle. The saturation will be implemented by a semitransparent *radial gradient* fading from white to clear the further it gets from the center of the view. Both will need to read the current brightness value from the bound color to obtain the correct appearance.

In both cases, the underlying gradient is defined using the Gradient type, which encapsulates the various colors that make up the gradient along with their positions in unit space (i.e., between 0 and 1). This Gradient is then used to initialize an AngularGradient or RadialGradient, which will handle the details of mapping the colors into place within a view. The approach will be similar for the brightness bar, which will use a LinearGradient to map the changes across a single axis.

Start by adding a private property to ColorWheel implementing the hue gradient:

p5/Do It/AccessoryViews/ColorWheel.swift
```
private var wheelGradient: AngularGradient {
    let (_, _, b) = color.hsb
    let stops: [Gradient.Stop] = stride(from: 0.0, through: 1.0, by: 0.01).map {
        Gradient.Stop(color: Color(hue: $0, saturation: 1, brightness: b),
                      location: CGFloat($0))
    }
    let gradient = Gradient(stops: stops)
    return AngularGradient(gradient: gradient, center: .center,
                           angle: .degrees(360))
}
```

Here, you've assigned hue values between 0 and 1 to gradient stops in the same range, using the stride(from:through:by:) method to generate 100 separate stops. At each stop, the hue is set to the stop location, the saturation is always 1, and for the brightness, you use the value from the currently selected color, i.e., the value from the brightness bar. The AngularGradient then uses that and rotates it 360° around the center of its enclosing view. The high number of gradient stops is an important requirement because the display itself uses RGB values. Each HSB value supplied to the gradient is going to be converted to RGB, then each component of the resulting RGB value will be interpolated to generate the intermediate colors of the gradient. This means that you'll see more errors appear in the wheel as the number of stops is reduced.

For saturation, the RadialGradient similarly uses a defined center point but requires a specific radius over which to change its value. You'll thus need to define a function to which you'll pass the required radius. The underlying Gradient, in this case, uses the current selection's brightness as before, but has a static hue and saturation of zero (zero saturation is white), and fades that color's opacity to zero over its range:

p5/Do It/AccessoryViews/ColorWheel.swift
```
private func fadeGradient(radius: CGFloat) -> RadialGradient {
    let (_, _, b) = color.hsb
    let fadeColor = Color(hue: 0, saturation: 0, brightness: b)
    let gradient = Gradient(colors: [fadeColor, fadeColor.opacity(0)])
    return RadialGradient(gradient: gradient, center: .center,
                          startRadius: 0, endRadius: radius)
}
```

With these done, you can create the wheel itself. Since you'll place an indicator on top of this view to highlight the currently selected color, place it in a ZStack. Then clip it into a circle shape and draw a white outline:

p5/Do It/AccessoryViews/ColorWheel.swift
```
ZStack {
    self.wheelGradient
```

```
        .overlay(self.fadeGradient(radius: proxy.size.width/2))
        .clipShape(Circle())
        .overlay(Circle().stroke(Color.white))
}
```

You'll also need to fix the aspect ratio of the GeometryReader you're using to contain the wheel so that the coordinate space being used for the gradients and the later interactions all fit into the square region bounding the circle. Along with that, you'll add the double-shadow effect to the wheel as well. Inside the body implementation, attach the following to the end of the GeometryReader:

p5/Do It/AccessoryViews/ColorWheel.swift

```
GeometryReader { proxy in
    // « ... »
}
.aspectRatio(contentMode: .fit)
.modifier(DoubleShadow())
```

Without this, the GeometryReader will take all the vertical space it's offered, and the gradient will look rather strange.

To see what you've created so far, add the following implementation to ColorWheel_Previews.previews and refresh the canvas:

p5/Do It/AccessoryViews/ColorWheel.swift

```
StatefulPreviewWrapper(TodoItemList.Color.purple) { binding in
    VStack {
        ColorWheel(color: binding)
    }
}
```

Try commenting out the .aspectRatio(contentMode:) modifier you added to the GeometryReader—what happens to the gradients? Now revisit the wheelGradient property and adjust the value of the by: parameter to the stride() function, and see how the gradient changes as the number of color stops is reduced.

Location Calculations

The next step is to look at the current color and determine where on the color wheel that color is located. For this purpose, let's use a simple 16×16 rectangle, filled with the selected color and given a white outline. Adding that to the ZStack directly will place it at the center of the wheel; to move it into the correct location, you'll use the .offset(_:) view modifier, passing in the offset calculated using the view's size. The size itself is available directly from the GeometryProxy; add this to the body implementation, just after the wheelGradient view:

p5/Do It/AccessoryViews/ColorWheel.swift
```
if !self.dragging {
    Rectangle()
        .fill(self.color.uiColor)
        .overlay(Rectangle().stroke(Color.white, lineWidth: 1))
        .frame(width: 16, height: 16)
        .offset(HSB.unitOffset(for: self.color, within: proxy.size))
}
```

Note that this is only going to be placed onscreen if the user isn't dragging the loupe around to make a selection; when that happens, this small indicator will be replaced by the larger loupe view.

Try changing the color value passed into the StatefulPreviewWrapper view in your preview provider—the color wheel will update to highlight the new value. In some cases, you'll see the brightness of the wheel change as well: green and purple both have a lower brightness value than the other predefined colors.

To make your wheel interactive, you'll need some way of setting a color value based on some location within the wheel. With the size of the wheel obtained from your GeometryProxy, it's quite straightforward to determine a unit location within those bounds and use that to update your color:

p5/Do It/AccessoryViews/ColorWheel.swift
```
private func assignColor(at location: CGPoint, in geometry: GeometryProxy) {
    let unitLocation = location.centeredUnit(within: geometry.size)
    HSB.updateColor(&color, at: unitLocation)
}
```

Responding to User Input

You can now generate colors from the coordinates of a user's actions; it remains only to enable interaction. A DragGesture, the same you used for the buttons in Raising Button Priority, on page 33, will give you what you need. The Value type for a drag gesture provides all the required information and more, though you only need the location property in this case. Attach this call to the .gesture() modifier to the end of the ZStack declaration containing the color wheel's gradient:

p5/Do It/AccessoryViews/ColorWheel.swift
```
Line 1  .gesture(
        DragGesture(minimumDistance: 0).onChanged {
            self.dragging = true
            self.loupeLocation = $0.location
5               .boundedInCircle(radius: proxy.size.width/2)
            self.assignColor(at: self.loupeLocation, in: proxy)
        }
        .onEnded { _ in
```

```
-           self.dragging = false
10          self.assignColor(at: self.loupeLocation, in: proxy)
-       }
-   )
```

Here, you've provided both .onChanged() and .onEnded() callbacks. You turn on the dragging property when the gesture's value changes on line 3, then turn it off when the gesture ends (line 9). Along with any change to the gesture, you read the new location and assign it to your loupeLocation property; note, however, that the coordinate gets clipped to the bounds of the circle, so that even if the user's finger leaves the area of the view, the location used to determine the color (and place the loupe) will remain within the wheel itself.

To see the effect in action in a live preview, you'll need to add another view to your preview that displays the currently selected color. Add the following to ColorWheel_Previews.previews, immediately following the ColorWheel:

p5/Do It/AccessoryViews/ColorWheel.swift

```
RoundedRectangle(cornerRadius: 12)
    .fill(binding.wrappedValue.uiColor)
    .overlay(RoundedRectangle(cornerRadius: 12)
        .stroke(Color.white))
    .frame(width: 300, height: 60)
    .modifier(DoubleShadow())
    .padding(.top)
    .zIndex(-1)
```

Now launch a live preview in the canvas and click and drag around the color wheel; the new feedback view at the bottom of the preview should change color as you move the mouse cursor.

Passing Data with View Preferences

Reacting to user input by toggling a boolean or changing a data value is all very well, but an interactive UI should be, well, *interactive*. It should be possible to adjust and move your views around based on a user's input. In this section, you'll learn to use the facilities SwiftUI provides for passing around coordinates, translating them, and applying them to your views in real-time.

To create a loupe view that moves around following the user's input, you'll need to create a PreferenceKey to carry anchor information. The anchor will be created using the location of the user's finger within the circle view, then resolved by the GeometryProxy to a coordinate within the GeometryReader view's bounds.

The first step is to create a preference key. To conform to the PreferenceKey protocol, a type needs to define three things:

- A Value type.
- A default value, as a static property.
- A static function named reduce(value:nextValue:), to combine multiple values from across the view tree.

The reduce() function provides the core part of the preference system in SwiftUI. While environment values are passed down the view tree to ever-larger numbers of descendants, preference resolve upwards toward a single ancestor. That means that differing values from two branches of the view hierarchy must be resolved somehow into a single value ready to be presented to a single ancestor.

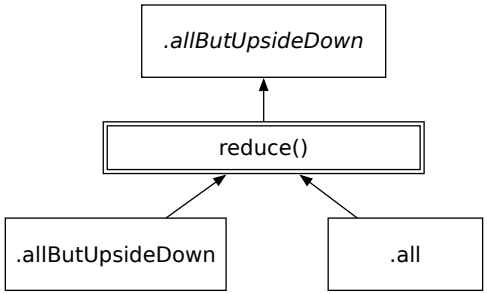
As an example, consider an OptionSet type such as UIInterfaceOrientationMask from UIKit:

```
public struct UIInterfaceOrientationMask : OptionSet {
    public init(rawValue: UInt)
    public static var portrait: UIInterfaceOrientationMask { get }
    public static var landscapeLeft: UIInterfaceOrientationMask { get }
    public static var landscapeRight: UIInterfaceOrientationMask { get }
    public static var portraitUpsideDown: UIInterfaceOrientationMask { get }
    public static var landscape: UIInterfaceOrientationMask { get }
    public static var all: UIInterfaceOrientationMask { get }
    public static var allButUpsideDown: UIInterfaceOrientationMask { get }
}
```

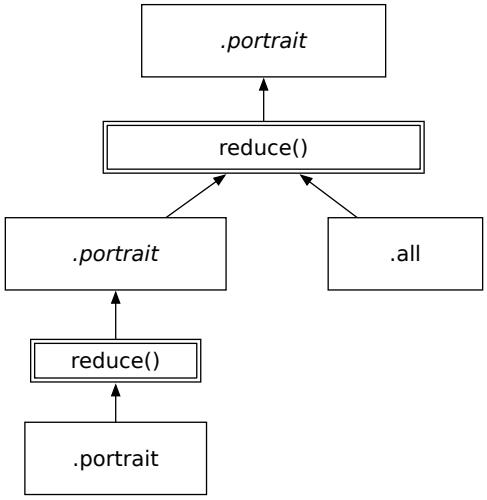
There are several different values available, and these may be combined—for instance, .landscape is the same as [.landscapeLeft, .landscapeRight]. This type is ideal for resolution through the preference system. Suppose subviews were able to declare their orientation support as a preference. One descendant supports all resolutions, while another supports everything except .portraitUpsideDown. The reduce() function could be implemented to return the smallest supported set of values, like so:

```
public static func reduce(value: inout UIInterfaceOrientationMask,
                          nextValue: () -> UIInterfaceOrientationMask) {
    // use the most restrictive set from the stack
    value.formIntersection(nextValue())
}
```

This would cause the parent view to receive a value of .allButUpsideDown:



Now consider a new subview added further down the tree, which only supports portrait orientations. This value would be reduced along with its parent's, causing a narrower set of allowed orientations to flow up the tree to the root view:



Tracking the Loupe

It's time to create your preference key. Add the following to the top of ColorWheel.swift, above the ColorWheel definition:

p5/Do It/AccessoryViews/ColorWheel.swift

```
Line 1  fileprivate struct LoupeLocationPreferenceKey: PreferenceKey {
    -       typealias Value = Anchor<CGPoint>?
    -       static var defaultValue: Anchor<CGPoint>? = nil
    -       static func reduce(
    5           value: inout Anchor<CGPoint>?,
    -           nextValue: () -> Anchor<CGPoint>?
    -       ) {
    -           if value == nil {
    -               value = nextValue()
    10          }
```

```
-     }
- }
```

On line 2 you've identified the associated value type as an Anchor<CGPoint>?—an optional anchor that resolves to a single coordinate location. The default value is nil, defined on line 3. The reduce() function is more interesting. Its two arguments are designed to be as economical as possible with memory. Instead of returning a new instance of some object of potentially large size, the current value is passed using the inout keyword, meaning it can be assigned or modified directly and in-place, avoiding costly allocations and copies. Similarly, the second argument is a block that will create/copy and return the incoming value only when requested. In the implementation you just wrote, you don't access the nextValue block at all unless the current value is nil—as on line 8.

With the preference key designed, you now need to generate an anchor value to assign to it. SwiftUI provides the .anchorPreference(key:value:transform:) view modifier for this purpose. Its first argument takes the type of PreferenceKey being used. The second takes an Anchor.Source of some kind, to request a concrete Anchor instance. The last parameter is a block to which that anchor will be provided, and which should return the Value type of the associated PreferenceKey.

In your case, you want an anchor for a single point: the value of the loupeLocation property. The value type of your LoupePreferenceKey is already an Anchor, so you won't need to do anything special in your transform block—just return the anchor as-is.

To create and assign the anchor, add the following to ColorStack.body immediately following the .gesture()() modifier:

p5/Do It/AccessoryViews/ColorWheel.swift

```
.anchorPreference(key: LoupeLocationPreferenceKey.self,
                  value: .point(self.loupeLocation),
                  transform: { $0 })
```

The loupe view will be added to the view hierarchy via the .overlayPreferenceValue(_:transform:), which can be attached to any ancestor of the ZStack to which you attached the .anchorPreference() modifier—even the same view. For the purposes of illustration, let's attach it to the GeometryReader surrounding the wheel view, immediately following the aspect ratio and double shadow modifiers:

p5/Do It/AccessoryViews/ColorWheel.swift

```
Line 1 .overlayPreferenceValue(LoupeLocationPreferenceKey.self) { anchor in
   2     GeometryReader { geometry in
   3         self.buildLoupe(geometry, anchor)
   4             .opacity(self.dragging ? 1 : 0)
```

```
5       }
6   }
```

Here, you've specified the LoupeLocationPreferenceKey as the preference to use, you've provided a block that will transform the preference's value—an Anchor<CGPoint>—into a View to be set as the overlay for the wheel. To obtain coordinates from the anchor, you use another GeometryReader view on line 2. You then pass the anchor and the provided GeometryProxy into a new method to create the loupe view itself, then use its opacity to display it only while the user is actively dragging their finger on the view.

The call to a separate buildLoupe() function is necessary here due to the nature of the ViewBuilder block passed to the GeometryReader initializer. To determine the correct offset for the loupe, you'll need to calculate a couple of values. A view builder block, unfortunately, doesn't allow for that—only statements that evaluate to View types, along with some basic if/else branches, are allowed. By moving that code to a separate function you regain the ability to use let statements.

The loupe view itself will be a circle 70 points in diameter, filled with the selected color, with a 1-point-wide white outline and a double shadow. It will be put into place using the .offset(x:y:) view modifier. Add this function to ColorWheel following the body implementation:

p5/Do It/AccessoryViews/ColorWheel.swift

```
Line 1  private func buildLoupe(
        _ geometry: GeometryProxy,
        _ anchor: Anchor<CGPoint>?
    ) -> some View {
5       let location = anchor != nil ? geometry[anchor!] : .zero
        let unitLocation = location.centeredUnit(within: geometry.size)

        return Circle()
            .fill(HSB.uiColor(at: unitLocation, basedOn: color))
10          .overlay(Circle().stroke(Color.white, lineWidth: 1))
            .frame(width: 70, height: 70)
            .modifier(DoubleShadow())
            .offset(x: location.x - 35, y: location.y - 35)
    }
```

Two locations are used here, both ultimately obtained through the anchor you've passed up through your LoupeLocationPreferenceKey. On line 5 you obtain the true screen coordinates for the anchor in the coordinate system of the overlay view. This is used on line 13 to move the loupe into the right location within its superview. The second location, defined on line 6 is in unit coordi-

nates measured from the center of the wheel, and is used on line 9 to calculate the fill color for the loupe.

The hue/saturation wheel is now complete—fire up a live preview in the canvas and try it out. You'll see the loupe appear and follow your gesture around the wheel, its color changing all the time to match the value beneath it. When you let go, the smaller indicator will reappear in the same location.

Adjusting Brightness

So far, you can select a color based on its hue and saturation values alone. To handle brightness, you need to add a new interactive view to the side of the color wheel, showing the various shades of brightness available for the current color, as shown in Figure 2, HSB Color Editor Layout, on page 111. The brightness bar will operate in a very similar manner to the color wheel. It will use a Gradient to fill an area with the chosen color at different levels of brightness, and it will use a small outlined Rectangle to indicate the currently-chosen brightness value. This will similarly use a ZStack to present the selection indicator on top of the gradient.

Your brightness bar is going to be used only along with the ColorWheel, so it can be defined as a fileprivate type within ColorWheel.swift. Add the following skeleton implementation to that file, just after the definition of LoupeLocationPreferenceKey:

p5/Do It/AccessoryViews/ColorWheel.swift

```
Line 1  fileprivate struct BrightnessBar<Value: ColorInfo>: View {
        @Binding var color: Value

        var body: some View {
5           GeometryReader { proxy in
                ZStack(alignment: .top) {
                    // « ... »
                }
            }
10      }
    }
```

The color value from the color wheel is passed on to the bar as a binding, and on line 5 you have the GeometryReader you'll use to interpret the location of the user's drag gesture within the range of brightness values represented.

Let's put the bar in the preview while you build it. First, give the bar some temporary content; add a color fill inside its body:

```
var body: some View {
    GeometryReader { proxy in
```

```
➤           Color.blue
        }
    }
```

Now, scroll down to the body implementation of your ColorWheel view and add a reference to the brightness bar following the GeometryReader there, at the end of the HStack content:

p5/Do It/AccessoryViews/ColorWheel.swift

```
Line 1  BrightnessBar(color: self.$color)
2           .padding(.vertical, 30)
3           .frame(maxWidth: 30)
4           .modifier(DoubleShadow())
5           .zIndex(-1)
```

Note here the .zIndex(-1) modifier on line 5; this causes the bar to be placed lower in the view order than the wheel and (crucially) the loupe. Try removing this line and dragging the loupe towards the brightness bar to see why that's desirable!

Refresh the preview in the canvas. The bar is stretching to fill the entire height of the device's screen, which isn't ideal. Really you want it to be of a similar height as the color wheel—a little less, in fact, since the color wheel is the primary element of this view. The reason it's growing is because its parent view, the HStack, is growing. There are a number of ways to prevent that, such as using a .frame() modifier on the stack view to set or limit its height. However, that seems a little overzealous—the wheel view shouldn't limit its size, its parent view should be able to adjust its size appropriately. Instead, set an aspect ratio, limiting the height based on its width (and vice versa). A little testing found that an aspect ratio of 1.125 worked best; apply this modifier after the closing brace of the HStack view:

p5/Do It/AccessoryViews/ColorWheel.swift

```
var body: some View {
    HStack(spacing: 16) {
        // « ... »
    }
➤   .aspectRatio(1.125, contentMode: .fit)
}
```

That looks better. Now return to BrightnessBar to add some useful properties.

First, you'll need a gradient for the current color, with its brightness ranging from 1 at the top to 0 at the bottom. This is straightforward to implement; read the hue and saturation from the the color property and create a Color instance for each end of your gradient:

p5/Do It/AccessoryViews/ColorWheel.swift
```
var gradient: Gradient {
    let (h, s, _) = color.hsb
    return Gradient(colors: [
        Color(hue: h, saturation: s, brightness: 1),
        Color(hue: h, saturation: s, brightness: 0)
    ])
}
```

Next, you'll need to be able to translate between a brightness value and a coordinate on the bar's y-axis. This is straightforward to implement using the brightness property on ColorInfo (see Affordances/HSBWheelHelpers.swift for the implementation):

p5/Do It/AccessoryViews/ColorWheel.swift
```
func selectionOffset(_ proxy: GeometryProxy) -> CGSize {
    CGSize(width: 0,
           height: CGFloat(1.0-color.brightness) * proxy.size.height - 5)
}
```

You now have everything you need to draw the bar and implement the drag gesture. Add a LinearGradient inside the body's ZStack, and give it a white border with the .border() modifier:

p5/Do It/AccessoryViews/ColorWheel.swift
```
LinearGradient(gradient: self.gradient,
               startPoint: .top,
               endPoint: .bottom)
    .border(Color.white)
```

Below that, add the location indicator: use the current color and give it a border, fix its height, and set its offset based on the current color's brightness value:

p5/Do It/AccessoryViews/ColorWheel.swift
```
self.color.uiColor
    .border(Color.white)
    .frame(height: 10)
    .offset(self.selectionOffset(proxy))
```

The final step is to add the DragGesture that will enable your users to adjust the brightness of their chosen color. No .onEnded() block is needed this time—just set the color selection as the drag's location changes:

p5/Do It/AccessoryViews/ColorWheel.swift
```
ZStack(alignment: .top) {
    // « ... »
}
.gesture(DragGesture(minimumDistance: 0).onChanged {
    let value = 1.0 - Double($0.location.y / proxy.size.height)
```

```
    self.color.brightness = min(max(0.0, value), 1.0)
})
```

Fire up the live preview and try it out. As you make adjustments on both the wheel and the bar, every view's content alters in real-time to reflect your changes.

Finalizing the Color Picker

Adding the wheel to the color picker is now a simple matter; open ColorPicker.swift and place it at the top of that view's VStack like so:

```
VStack(spacing: 16) {
    ColorWheel(color: $selectedColor)

    « ... »
}
```

Refresh the preview in your canvas and start a live preview running. The wheel works as before, this time updating the contents of the optional selection bar at the bottom of the view. Clicking the buttons for the predefined colors also immediately changes both that and the status of the color wheel and brightness bar. Note how the selection indicators in both move around as you switch between the predefined colors.

That's the first major element of the list editor complete, with only one more to go: the icon chooser. That won't be as long, I promise.

Building a Single-Choice Control

Radio buttons—a set of buttons of which only one may be selected at any time—is an interesting challenge in SwiftUI. While the Picker type is the usual way of implementing this concept on iOS, it doesn't have any representations that are ideal for selecting between a list of purely-visual icons. Instead, you'll create your own, using the same tools that you've used elsewhere in this chapter to quickly assemble a working radio-button group with a clear visual design.

Glance back at Figure 1, List Editor Layout, on page 103 and look at the "icon chooser" section. It consists of several rows of circular items, each of which will represent a single icon. All of the icons are going to be members of the SF Symbols set provided by Apple, presented in five rows of five. The full list of icons you'll support is defined in Resources/list-icons.json, and to display them, you need little more than a pair of ForEach views to iterate over the two-dimensional array.

Start by creating a new SwiftUI View in the AccessoryViews group, naming it IconChooser.swift. Give your new view a binding to a String property named selectedIcon, and implement its body using the aforementioned pair of ForEach views iterating over a global property named listIconChoices. Place the outermost ForEach within a VStack, and the innermost within an HStack, with a Button for each icon. The button should set the selectedIcon value when pressed, and its content should be an Image displaying that icon. Lastly, give both stacks a spacing of 14 points. The resulting code should look something like this:

p5/Do It/AccessoryViews/IconChooser.swift

```
Line 1  struct IconChooser: View {
        @Binding var selectedIcon: String

        var body: some View {
    5       VStack(spacing: 14) {
                ForEach(listIconChoices, id: \.self) { rowData in
                    HStack(spacing: 14) {
                        ForEach(rowData, id: \.self) { icon in
                            Button(action: { self.selectedIcon = icon }) {
   10                           Image(systemName: icon)
                            }
                        }
                    }
                }
   15       }
        }
```

Now, set up the preview using a real binding in the same manner you've used before, start a live preview, and try it out:

p5/Do It/AccessoryViews/IconChooser.swift

```
StatefulPreviewWrapper("list.bullet") {
    IconChooser(selectedIcon: $0)
}
```

The buttons aren't looking particularly special right now; they're all just accent-colored icons bunched together in the center of the screen, each row a different size, dimming slightly when tapped. Let's change that by creating a new ButtonStyle to apply to them all.

Inside the IconChooser structure, just above the body implementation, add a new private type named IconChoiceButtonStyle, and implement its makeBody(configuration:) method to give the icons a circular background, a large font, and a scale-up effect when they're pressed:

p5/Do It/AccessoryViews/IconChooser.swift

```
Line 1  private struct IconChoiceButtonStyle: ButtonStyle {
        func makeBody(configuration: Configuration) -> some View {
```

```
-         configuration.label
-             .font(.system(size: 24, weight: .bold, design: .rounded))
5             .padding(6)
-             .frame(width: 30)
-             .padding(14)
-             .background(Color(UIColor.tertiarySystemFill))
-             .clipShape(Circle())
10            .scaleEffect(configuration.isPressed ? 1.2 : 1)
-     }
- }
-
- var body: some View {
15      VStack(spacing: 14) {
-         // « ... »
-     }
-     .buttonStyle(IconChoiceButtonStyle())
- }
```

The background color on line 8 is one that hasn't been mentioned before. The `UIColor` type from UIKit makes available the same predefined colors as SwiftUI's `Color` type, but `UIColor` also vends a quite large number of *semantic* colors, named for the purpose they're intended to fulfil. By using these values, you can obtain the standard system look and feel, and your application will change to match if these colors are redefined in future OS updates. The colors will also differ appropriately when used in dark mode vs. light mode, or when the user has selected a high-contrast color scheme. SwiftUI doesn't export all of these as yet, but it's possible to initialize a SwiftUI `Color` with a `UIColor`, so you don't have to go without. Here, you've used the *tertiary system fill*; there are actually four fill colors defined, with each being a little lighter and less obtrusive than the last. The `UIColor` documentation describes their intended uses, and `tertiarySystemFill` is described as the color to use for "input fields, search bars, and buttons." These are buttons, so that's the color to use.

Try out the buttons in a live preview. They now stand apart from one another in a well-defined grid, and the bounds of each button are clearly delineated by their background. It would look nicer, though, if the currently-selected icon were highlighted in some fashion. Looking at the list editor from Apple's Reminders application (which you're emulating here), there's a darker ring around the currently selected icon. Let's do the same here.

At first, though, it seems as though this could be added directly to the `Button` itself, drawing an overlay or border if that button is selected. Doing so would likely adjust the size of each button, however, making it a little more involved to arrange them nicely. A better approach would be to display the highlight in an overlay or background layer for the entire view, moving the content to

place it behind the selected icon. You've already seen the tools for implementing this; yes, this is a job for Anchor and PreferenceKey once more.

Dealing with Multiple Preference Values

Preference values in SwiftUI are designed for collection and reduction, combining multiple inputs into a single output. This icon chooser makes use of this facility by having every button add its information to a single array, with that array being used to obtain coordinates for a particular item to highlight it.

Each button will publish the anchor for its bounds, and then you'll use that to adjust the background view's location with the aid of a GeometryProxy. However, here you have 25 subviews publishing values into the preference system, so the "first non-nil value" reduction approach used for the color wheel won't work here. Additionally, there needs to be some way to correlate the selected icon name with the location of one particular button. This means that the preference key type will need to be a little more involved.

The approach you'll take is twofold: first, you'll define a structure to serve as the preference key's Value type. This will contain an anchor for the button's bounding rectangle along with the icon name associated with that button. The preference key will then use an *array* of these types as its Value, and it will reduce values by merging together all the values into a single array. When it comes time to create the background using the final preference data, the array can be searched to locate the anchor associated with the selectedIcon property value.

First, create the value and preference key types inside the IconChooser type, above the definition of IconChoiceButtonStyle:

p5/Do It/AccessoryViews/IconChooser.swift

```
private struct IconSelectionInfo {
    let name: String
    let anchor: Anchor<CGRect>
}

private struct IconChoice: PreferenceKey {
    typealias Value = [IconSelectionInfo]
    static var defaultValue: Value = []
    static func reduce(value: inout [IconSelectionInfo],
                       nextValue: () -> [IconSelectionInfo]) {
        value.append(contentsOf: nextValue())
    }
}
```

Each button can now publish its anchor by using the .anchorPreference(key:value:transform:) view modifier:

p5/Do It/AccessoryViews/IconChooser.swift

```
Button(action: { self.selectedIcon = icon }) {
    Image(systemName: icon)
}
.anchorPreference(key: IconChoice.self, value: .bounds) {
    [IconSelectionInfo(name: icon, anchor: $0)]
}
```

The value of .bounds is an anchor source that is used to request an anchor describing the bounding rectangle of the view to which it's attached. Use it to create a circle view as the background of the outermost VStack using .backgroundPreferenceValue(_:transform:):

p5/Do It/AccessoryViews/IconChooser.swift

```
var body: some View {
    VStack(spacing: 14) {
        // « ... »
    }
    .buttonStyle(IconChoiceButtonStyle())
    .backgroundPreferenceValue(IconChoice.self) { values in
        GeometryReader { proxy in
            self.selectionCircle(for: values, in: proxy)
        }
    }
}
```

The calculations for laying out the selection circle require variables, as the color wheel's loupe view did earlier. The circle itself is again set up in a separate function.:

p5/Do It/AccessoryViews/IconChooser.swift

```
Line 1  private func selectionCircle(
        for prefs: [IconSelectionInfo],
        in proxy: GeometryProxy
    ) -> some View {
5       let p = prefs.first { $0.name == selectedIcon }
        let bounds = p != nil ? proxy[p!.anchor] : .zero

        return Circle()
            .stroke(lineWidth: 3)
10          .foregroundColor(Color(UIColor.separator))
            .frame(width: bounds.size.width + 12,
                   height: bounds.size.height + 12)
            .fixedSize()
            .offset(x: bounds.minX - 6, y: bounds.minY - 6)
15  }
```

On line 5 you search through the array of preference values to locate the one associated with the chosen icon. With that you're able to pass it into the GeometryProxy to obtain a CGRect with local coordinates matching that item's anchor. The resulting view is a circle with a frame set on line 11 to extend an extra six points outside the bounds of the button it surrounds on all sides. The .fixedSize() view modifier provides a hint to SwiftUI's layout engine that this view's size should be considered absolute. The offset from the anchor's bounding rectangle is used to position the circle within the view. Note the presence of another semantic color definition on line 10; this time, you're using the color defined for separators and thin lines, which seems appropriate in this case. Like the fill for the buttons, this is a semitransparent color that will work on top of most backgrounds.

Run the live preview again, and you'll see the final effect as the selection circle jumps around when you select each button. Its size matches the scale effect on the buttons so that it lines up with the increased size of the button as it moves; I think that's a nice effect, no?

Composing the Final Interface

It's been a long journey, but the end is in sight: now it's time to assemble all of these components into a single cohesive whole—the list editor.

Start by creating a new SwiftUI View, and name it TodoListEditor.swift. Give the new view the two properties it will need to do its job: the TodoList to operate on, and the DataCenter used to save the changes:

p5/Do It/TodoListEditor.swift

```
@EnvironmentObject var data: DataCenter
@State var list: TodoItemList
```

The list's structure, as outlined in Figure 1, List Editor Layout, on page 103, consists of five components:

- A titlebar with cancel/done buttons to either side.
- A large display of the list's selected icon.
- A text field to edit the list's name.
- The color picker you created earlier.
- The icon chooser.

The last two components are placed inside a scroll view since otherwise, the entire UI wouldn't fit on the screen; the top three items will stay in place while the color picker and icon chooser scroll beneath. If you think this looks like a job for a VStack, you'd be right. Start by laying out the structure of the view's body:

p5/Do It/TodoListEditor.swift
```
var body: some View {
    VStack {
        // « Top bar: cancel, title, done »

        // « List icon »

        // « Text field »

        VStack(spacing: 0) {
            Divider()
            ScrollView {
                // « Color picker »
                // « Icon chooser »
            }
        }
    }
}
```

Let's start at the top and work downwards.

Top Bar Layout

You've seen this particular item before when designing the todo item editor in Building an Editor, on page 66. This bar is a simple HStack containing a title and two buttons, separated by spacers:

p5/Do It/TodoListEditor.swift
```
HStack(alignment: .firstTextBaseline) {
    Button("Cancel") {
        // « dismiss sheet »
    }
    Spacer()
    Text("Name & Appearance")
        .bold()
    Spacer()
    Button(action: {
        // « save data, dismiss sheet »
    }) {
        Text("Done")
            .bold()
    }
}
.padding()
```

These buttons need to be able to do two things: the "Done" button has to save the data to the store in the DataCenter, and both buttons need to dismiss the editor sheet. Saving the data consists of locating the matching list within the DataCenter and setting it to the value of the modified list property from the editor. That's also something you've done before, and it's a straightforward matter to create a private function in TodoListEditor to implement it:

p5/Do It/TodoListEditor.swift
```
private func saveData() {
    if let idx = data.todoLists.firstIndex(where: { $0.id == list.id }) {
        data.todoLists[idx] = list
    }
}
```

When showing the TodoItemEditor (see Presentation, on page 76) you implemented the cancel/done buttons as part of the presenting TodoItemDetail view, so the buttons simply toggled the same property used to present the sheet. Here you're implementing everything from the context of the *presented* view, so you can't take that route. You could use a binding to some showEditor to enable the same approach, but SwiftUI already has your back, in the form of the PresentationMode type.

PresentationMode is a simple struct type that SwiftUI places into the environment. It provides two things:

- A boolean property, isPresented, which indicates whether the current view or one of its ancestors was presented in some reversible manner.
- A function, dismiss(), which will dismiss the topmost presented view if there is one. If isPresented is false, then the dismiss() function has no effect.

This presentation mode is wired up by SwiftUI both when a sheet is presented via the .sheet() and when a new view is pushed onto a navigation stack through a NavigationLink, and is able to dismiss both types of view. To access it, you use an @Environment attribute to fetch the presentationMode property from the environment:

p5/Do It/TodoListEditor.swift
```
@Environment(\.presentationMode) var presentation
```

The value of that property is a Binding<PresentationMode>, so you'll access its contents via the binding's wrappedValue property, as described in Property Wrappers, on page 58.

This is the last piece you'll need to implement your buttons' actions. The "Cancel" button will call dismiss() to close the editor; the "Done" button will save the changes before doing the same:

p5/Do It/TodoListEditor.swift
```
Button("Cancel") {
    self.presentation.wrappedValue.dismiss()
}
Spacer()
Text("Name & Appearance")
    .bold()
Spacer()
```

```
Button(action: {
    self.saveData()
    self.presentation.wrappedValue.dismiss()
}) {
    Text("Done")
        .bold()
}
```

Large Scale Iconography

Next up is the list icon. Since it's going to be quite large, let's add a subtle gradient to the background color to give it a little texture, so it won't seem as flat. You'll darken the list's color a little by reducing its brightness to about 70% of its current value, then fade from the original color to the darker variant across the bounds of the icon's background. To make it a little more organic, the gradient will happen at a slight angle, running from the top-left unit point to the bottom-center. Lastly, the familiar double-shadow effect will help lift it from the background.

First, add a new property to TodoListEditor to return a LinearGradient:

p5/Do It/TodoListEditor.swift

```
var iconGradient: LinearGradient {
    var (h, s, b) = list.color.hsb
    b *= 0.7

    return LinearGradient(
        gradient: Gradient(colors: [
            list.color.uiColor,
            Color(hue: h, saturation: s, brightness: b)
        ]),
        startPoint: .topLeading,
        endPoint: .bottom)
}
```

With the gradient defined, you can display the icon using a regular Image view. Increase the icon's size using a .font() modifier to specify a 56-point size and to draw the icon in a bold, rounded format. Locking a 1:1 aspect ratio, a little padding, and an explicit size round out the appearance you're looking for:

p5/Do It/TodoListEditor.swift

```
Image(systemName: list.icon)
    .font(.system(size: 56, weight: .bold, design: .rounded))
    .aspectRatio(contentMode: .fit)
    .padding(36)
    .foregroundColor(.white)
    .frame(width: 112, height: 112)
    .background(iconGradient)
    .clipShape(Circle())
    .modifier(DoubleShadow())
```

Custom Text Field Appearance

Next comes the text field. By default, a TextField view appears as plain text on an empty background. It can alternatively be given a rounded border using a style modifier and the RoundedBorderTextFieldStyle type, but that doesn't quite match the desired appearance. Sadly, though SwiftUI provides a TextFieldStyle protocol, its details are all internal, so you can't take the same approach for the text field that you have used for buttons so far. This design seems like it might be more generally useful, though, so instead, let's create a new ViewModifier to style the field.

Open AccessoryViews/ViewModifiers.swift and add the following definition below the DoubleShadow type.

p5/Do It/AccessoryViews/ViewModifiers.swift

```
Line 1  struct BorderedTextField: ViewModifier {
        func body(content: Content) -> some View {
            content
                .multilineTextAlignment(.center)
5               .padding(.vertical, 12)
                .background(
                    RoundedRectangle(cornerRadius: 10, style: .continuous)
                        .foregroundColor(Color(UIColor.tertiarySystemFill))
                )
10      }
    }
```

This modifier applies a few different effects to its view. Firstly, any text it contains is centered within the view's bounds using the .multilineTextAlignment(_:) modifier on line 4. This, like most other text-affecting modifiers, actually places its value into the environment, so its effect will cascade down the view stack.

The background applied to the view is the familiar RoundedRectangle, but on line 7 an additional argument, style, is being used to describe the type of rounding to perform on the corners. If not specified, corners are given a circular appearance—the curve of the corner is a quarter-circle. Here you've specified .continuous, which specifies the use of the same mathematical bezier curve corners used on icons and other types through iOS since version 7. When using rounded corners on larger views, the continuous arc will generally look better, but on smaller corners, the difference is less obvious, so use the default circular corners there.

Lastly, the fill color for the background set on line 8 is the same tertiarySystemFill you used in the icon chooser above.

Update ViewModifier_Previews to display a couple of different uses of this modifier:

p5/Do It/AccessoryViews/ViewModifiers.swift

```
static var _sampleText: String = "Sample Text"
static var sampleText: Binding<String> = Binding(
    get: { _sampleText }, set: { _sampleText = $0 })

static var previews: some View {
    Group {
        // « shadow previews »
        Text("Start Making Things")
            .padding(.horizontal)
            .modifier(BorderedTextField())

        TextField("Title", text: sampleText)
            .modifier(BorderedTextField())
    }
    .previewLayout(.fixed(width: 350, height: 350))
}
```

This code uses an alternate approach to providing a working Binding value for the text field—you can define a static variable on your preview provider to hold the value, then define a Binding property directly that accesses that variable. Refresh the preview in the canvas to see the effect on a regular Text view and a TextField; note that the text field uses all available width since its content isn't fixed and may grow when edited.

Now, return to TodoListEditor.swift and add the name field, giving it a 20-point semibold, rounded font, with a little padding around the outside:

p5/Do It/TodoListEditor.swift

```
TextField("List Title", text: $list.name)
    .font(.system(size: 20, weight: .semibold, design: .rounded))
    .modifier(BorderedTextField())
    .padding()
```

Only one task remains to complete the list editor. Add the color picker and icon chooser views within the ScrollView, binding them to the list's color and icon properties respectively. Note that the color picker will not turn on the optional selection view since the icon display at the top of the editor will serve that purpose nicely.

p5/Do It/TodoListEditor.swift

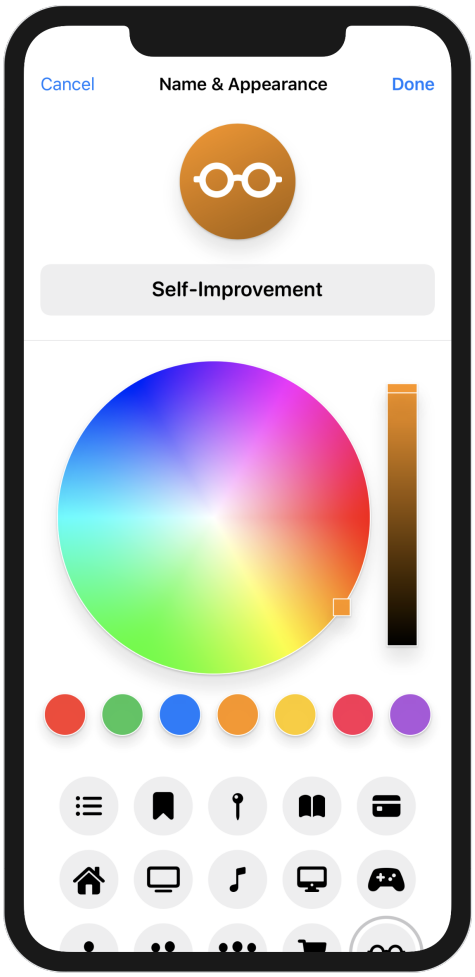
```
ScrollView {
➤     ColorPicker(selectedColor: $list.color)
➤         .padding()
➤     IconChooser(selectedIcon: $list.icon)
➤         .padding()
}
```

Finally, add a preview, not forgetting the DataCenter in the environment:

p5/Do It/TodoListEditor.swift

```
ScrollView {
    ColorPicker(selectedColor: $list.color)
        .padding()
    IconChooser(selectedIcon: $list.icon)
        .padding()
}
```

Refresh the canvas and launch a live preview. Try interacting with all the controls and observe the changes that occur, all in real-time, as you make the changes. Sit back and admire your handiwork—it's been a long road, but you've made it.



Presenting the Editor

One final task remains. At present, there is nothing in the application that will display the editor you've just assembled. Let's add an "Info" button to the navigation bar of the todo list view and have that present the editor in a sheet. Open TodoList.swift and make these changes:

p5/Do It/TodoList.swift

```
Line 1 @State private var showingListEditor: Bool = false
-
- private var barItems: some View {
-     HStack(spacing: 14) {
5         if self.list != nil {
-             Button(action: { self.showingListEditor.toggle() }) {
-                 Image(systemName: "info.circle")
-                     .imageScale(.large)
-                     .font(.system(size: 24, weight: .bold))
10            }
-         }
-         sortButton
-     }
- }
15
- var body: some View {
-     List(sortedItems) { item in
-         // « ... »
-     }
20   // « ... »
-    .navigationBarItems(trailing: barItems)
-    // « ... »
-    .sheet(isPresented: $showingListEditor) {
-        TodoListEditor(list: self.list!)
25          .environmentObject(self.data)
-    }
- }
```

This adds a property to control the display of the sheet and replaces the trailing items on the navigation bar with a new HStack containing the two buttons. It's important to note, however, the check on line 5: you can't show an editor for the "All Items" view.

Build and run your application and try everything out.

What you Learned

In this chapter, you've moved far beyond the basic UI tools of text and background colors and have learned to use the declarative, reactive tools provided by SwiftUI to create a complex and interactive interface.

- You added a good understanding of padding, backgrounds, and clipping shapes to your SwiftUI arsenal.
- You learned to look at your compositions with a critical eye for visual balance and attention to detail.
- You've seen how the use of some additional font properties can give your application a sense of identity.
- SwiftUI provides some simple tools that scale easily to implement much more complex systems, and you've seen how to use several of these in a few different ways.

The Xcode canvas is an incredibly useful tool for debugging layout issues. In the next chapter you'll learn some tips & tricks for putting it to best use as you implement support for dynamic type, localization, and right-to-left layouts.

CHAPTER 6

Making the Most of the Canvas

Story Map

Why do I want to read this?

The Xcode canvas provides a handy and flexible way of previewing your content under a wide variety of situations, and as such, is a useful prototyping and verification tool.

What will I learn?

You'll learn how to configure your previews to check your layouts on different devices and in different locales, including right-to-left layout. You'll see how you can quickly and easily view your application's views in light and dark presentation modes, and how it will render based on the user's preferred text scaling.

What will I be able to do that I couldn't do before?

You will have a veritable tool-belt of useful preview functionality at your beck and call, and you'll be able to quickly check, diagnose, and fix layout issues under a variety of real-world situations, even while prototyping your UI.

Where are we going next, and how does this fit in?

It's time to move to a larger canvas. The next chapter will bring the application to iPadOS and look at implementing support for the various features unique to that platform.

You have now assembled a working application with numerous components. At this point, you might start using the application to test it under various different conditions. Any localization work would begin, and you'd need to run through everything in your application to ensure correctly localized and translated data appears. That's potentially a lot of work, but happily the Xcode canvas provides facilities that will help a lot in that regard. The canvas is an

important tool in application development with SwiftUI; not only does it provide visual editing capabilities and immediate updates for prototyping, but it can provide customized displays so you can evaluate multiple outputs *at design time*. It has a number of tricks up its sleeve, which you'll put to good use in this chapter.

Handling Size and Appearance

When you build an application for iOS, there are several classes of devices it can run on. In size alone, you have one size for the iPhone SE, one for each of the iPhones 8S and 8S Plus, then more for the iPhone X, XS, the iPhone 11, and iPhone 11 Pro Max. In addition to these, the user can select their desired text size from very small to quite large, which will have a further effect on your app's layout. With the advent of the font-based SF Symbols for icon images in iOS 13, even standard iconography will scale with a user's chosen text size.

Alongside these sizing and layout concerns, iOS 13 brings *dark mode* support, offering a white-on-black alternative color scheme. This can directly affect some of your choices. Hard-coding your text to be a darkish gray? In dark mode, it'll be difficult to see against a now-black view background. Manually setting your background to white? Users who favor dark mode won't be happy that your app doesn't conform.

Virtually all the work in this chapter will occur inside of the preview views placed at the bottom of each of the .swift files containing your views. Let's start with the detail view.

Open TodoItemDetail.swift and scroll down to reveal the TodoItemDetail_Previews structure at the bottom of the file. This type operates in a similar manner to a View, but instead of returning a view from a body property, it uses the previews property. The content, however, is much the same, with the exception of a few extra methods that only affect the Xcode preview display.

The first thing you'll notice about the detail view on the canvas is that, unlike in the real app, there's no title bar or back button. This is expected, given that the preview is showing only this single view; however, it would be more useful to see it in context with the navigation bar displayed. To start, wrap the existing TodoItemDetail() call in the previewsproperty in a NavigationView, like so:

```
static var previews: some View {
    NavigationView {
        TodoItemDetail(item: todoItems[0])
    }
}
```

Click Resume, if necessary, to update the preview in the canvas, and you'll see… well, not a lot of difference, to be honest. The header view is tall, has content aligned at the bottom, and is stretching to the very top of the screen, so the navigation bar is fully transparent. Furthermore, it doesn't have any content, so there's really nothing to see. Let's add something there for now.

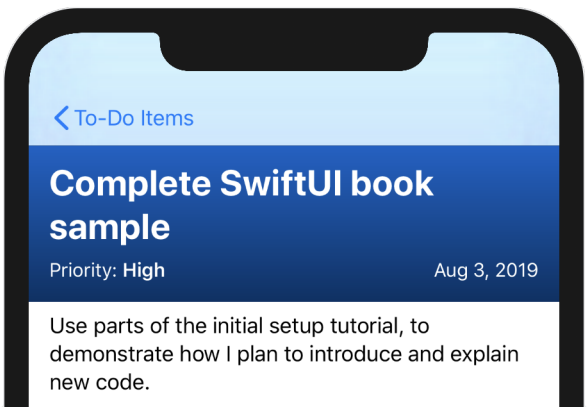
Add an imitation "Back" button by appending the following modifier to the TodoItemDetail instance you just created:

```
TodoItemDetail(item: todoItems[0])
➤     .navigationBarItems(leading: self.backButton)
```

Add a new property to the preview provider to implement the button:

```
static var backButton: some View {
    Button(action: {}) {
        HStack(spacing: 4) {
            Image(systemName: "chevron.left")
                .imageScale(.large)
                .font(.headline)
            Text("To-Do Items")
        }
    }
}
```

Now your view appears much as it does when you run the app in the simulator:



The blue coloring of the header extends up behind the navigation bar, adjusting its appearance slightly to match the rest of the content. Switch your preview to use one of the other to-do items to see what it looks like with other colors.

Using Multiple Previews

The Xcode canvas is quite flexible. One of its more useful features is the ability to display more than a single preview at a time, whether showing different data or presenting in different contexts or layouts.

Within and Without

You might want to compare the appearance of the view both when presented within a navigation stack or without. Previews enable such things via (among others) the Group view. Not so much a view in itself as an ordering mechanism, a Group view will simply render its contents within the group's enclosing view. In the case of previews, a top-level Group will cause each of its subviews to render as a separate preview.

To see your detail view with and without the navigation bar, use the following code to generate a pair of previews:

```
static var previews: some View {
    Group {
        NavigationView {
            TodoItemDetail(item: todoItems[0])
        }
        TodoItemDetail(item: todoItems[0])
    }
}
```

Now you have two previews on the canvas, and you'll note that the second one doesn't show the navigation—since there's no navigation view, there's no bar.

Multiple Items

Having multiple previews for a single view can help in other ways, too. You can quickly and easily see how several different to-do items will render in the detail view. Try it out:

```
Group {
    TodoItemDetail(item: todoItems[0])
    TodoItemDetail(item: todoItems[1])
    TodoItemDetail(item: todoItems[2])
}
```

You could wrap each of those in a NavigationView to see a navigation bar, but that quickly becomes unwieldy—as does an increasing number of items. Happily, the ForEach view comes in handy here.

ForEach operates in a manner similar to a list, in that it generates content using a block and a sequence of items. Where List creates an actual list view, however, ForEach simply creates multiple subviews and passes them all up to its enclosing view. That can be a list view itself (this is helpful if your list contains multiple types of row views, for example), or if it's a PreviewProvider, then each subview will be rendered as its own preview. To see this in action, replace the Group-based preview property with the following ForEach-based version:

p3/Do It/TodoItemDetail.swift

```
ForEach(todoItems) { item in
    TodoItemDetail(item: item)
}
```

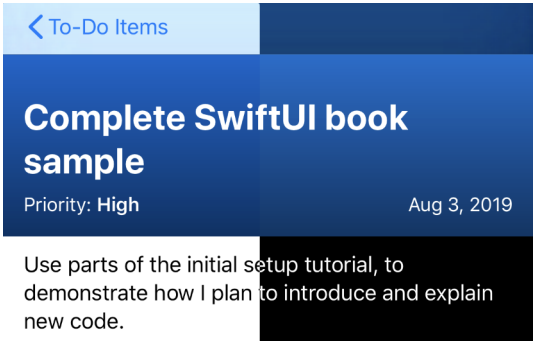
Supporting Dark Mode and Light Mode

iOS 13 brings *dark mode* to iOS for the first time. This provides a white-on-black color scheme, lighting up less of the display, and using a more muted set of standard colors. Many users will expect your app to look good in dark mode, so it's worthwhile to have the canvas show the different appearances.

SwiftUI provides the .colorScheme() modifier for views, allowing them to override the system setting. You can use this in previews to explicitly turn on dark mode:

```
NavigationView {
    TodoItemDetail(item: todoItems[0])
}
.colorScheme(.dark)
```

In the following image, note that the blue background of the header view changes slightly:



It becomes softer, a little lighter, and less vibrant so that it doesn't make such a stark contrast against the black background of the text below. Since the

TodoItem.Color type uses SwiftUI's preset colors, this happens automatically: all the system colors have variants for both light and dark mode.

Right now, there are two color schemes defined in the ColorScheme enumeration: .light and .dark. Since ColorScheme conforms to the CaseIterable protocol, it provides an allCases static property that you can use to enumerate the available values in a ForEach view. Use the following code to generate a preview for each color scheme available, including any additional ones you might encounter in the future:

p3/Do It/TodoItemDetail.swift

```
ForEach(ColorScheme.allCases, id: \.self) { scheme in
    NavigationView {
        TodoItemDetail(item: todoItems[0])
    }
    .colorScheme(scheme)
    .previewDisplayName(String(describing: scheme))
}
```

Here, you're iterating across all of the available schemes, using their enumeration value as an identifier, then setting that color scheme on the NavigationView using the .colorScheme() modifier.

Using Device Previews

The Xcode canvas can also display several different devices, which will help you to quickly discover any layout problems that might arise on the smaller screen of an iPhone SE, for example. For this purpose, SwiftUI provides a number of view modifiers specific to showing previews in the canvas:

- .previewLayout(value: PreviewLayout) enables you to define the size and shape of the preview's container (you saw this before on page 25).
- .previewDevice(value: PreviewDevice?) lets you specify a particular iOS, watchOS, or tvOS device to simulate; this is the default, with a device chosen based on the active Xcode build scheme.
- .previewDisplayName(value: String?) allows you to customize the name displayed below the preview on the canvas. This can be quite helpful when showing multiple previews.

To show multiple devices, you'll use the .previewDevice() modifier. Its argument, an instance of PreviewDevice, can be constructed using the standard (or 'marketing') name of the device in question using the PreviewDevice(rawValue: String) initializer. Handily, that value is also useful as a name for the preview, meaning that it's quite straightforward to generate multi-device previews:

p3/Do It/TodoItemDetail.swift
```
ForEach(["iPhone 11", "iPhone SE"], id: \.self) { name in
    TodoItemDetail(item: todoItems[0])
        .previewDevice(PreviewDevice(rawValue: name))
        .previewDisplayName(name)
}
```

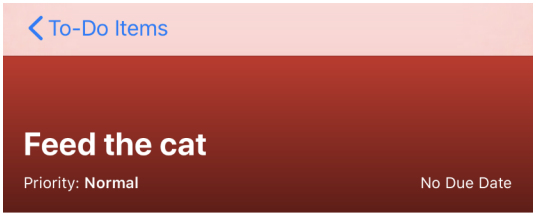
Supporting Dynamic Type

One way a user can customize the UI on their iOS devices is by changing the *dynamic type size* of text throughout all apps that correctly support it, as recommended by Apple. SwiftUI has just about everything you need for this task already implemented, so your app already supports dynamic type sizes. This leaves you with another task, though: you need to make sure that your application will behave correctly at different type sizes. Previews in the Xcode canvas make this once onerous task relatively simple and straightforward.

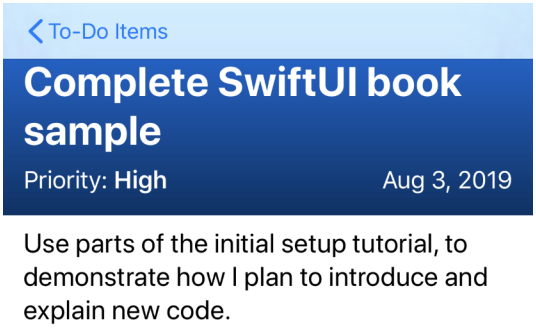
By now, you can probably guess how this preview will be implemented. The ContentSizeCategory enumeration is used to specify the size class to use, and it conforms to CaseIterable, so it has an allCases static property which can be enumerated using a ForEach view. To actually set the size class to use, you specify it using the SwiftUI environment, with the \.sizeCategory key. You can use a description of the category as the preview name since there are a lot of possible values.

p3/Do It/TodoItemDetail.swift
```
ForEach(ContentSizeCategory.allCases, id: \.self) { category in
    TodoItemDetail(item: todoItems[0])
        .environment(\.sizeCategory, category)
        .previewDisplayName(String(describing: category))
}
```

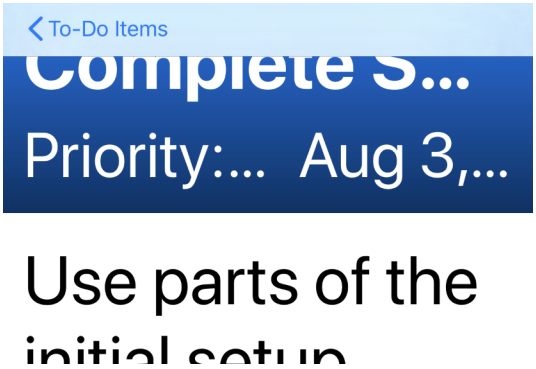
The results are quite illustrative. A single-line title on the smallest setting makes the header look far too big:



Meanwhile, a double-line title on the largest non-accessibility setting looks like it's squeezing up against the navigation bar somewhat:



Once you get to the accessibility-related size, though, things just get unbearable:



None of this is ideal. While the very largest accessibility size likely requires some special handling, the non-accessibility issues all come down to one thing: the header has a fixed 200-point height, defined by the rectangle:

```
Rectangle()
    .fill(item.color.uiColor)
    .frame(height: 210)
    .overlay(TitleOverlay(item: item))
```

This code defines an area of 210 points in height and then lays the text and gradient on top of it. A better solution is to let the text content determine the size, and have the color and gradient match.

The opposite of the .overlay() modifier is .background(). In both cases, the view provided to the modifier is forcibly sized to match the one to which the modifier is applied. So far, you've taken a fixed-size rectangle and applied the text to it. Now that the text is growing and shrinking, the rectangle should follow suit, and swapping the order of the views will fix that.

Take the code from the TitleOverlay view and move it into the appropriate places in the TodoItemDetail view:

1. Move the properties gradient and formatter to become properties of TodoItemDetail.
2. Take the content of TitleOverlay.body, and put it into TodoItemDetail.body, replacing the Rectangle() view and its modifiers.
3. Attach a .background() modifier to the VStack you just moved, filling it with a Rectangle, filled with the item's color, and with an overlay using the gradient property.

The code inside the top-level VStack of TodoItemDetail.body should now look something like this:

p3/Do It/TodoItemDetail.swift

```
VStack(alignment: .leading, spacing: 8) {
    Text(verbatim: item.title)
        .font(.title)
        .bold()
        .layoutPriority(1)

    // « … more … »
}
.foregroundColor(.white)
.padding()
.padding(.top)
    .background(ZStack {
        Rectangle()
            .fill(item.list.color.uiColor)
            .overlay(gradient)
            .edgesIgnoringSafeArea(.top)
    })

if item.notes != nil {
    Text(verbatim: self.item.notes!)
        .padding()
}
Spacer()
```

Refresh the previews in the canvas, and you'll see that the header is now growing along with the size of the text, preventing it from pushing up into the top of the screen—success!

There's one last thing you can do here, though. With short notes content such as you're currently using it isn't obvious until you scroll down to look at the accessibilityExtraExtraExtraLarge variant. There, you'll see that the notes no longer fit on the screen, and in fact, are truncated. To resolve this, you can use a ScrollView, either around the text field containing the notes:

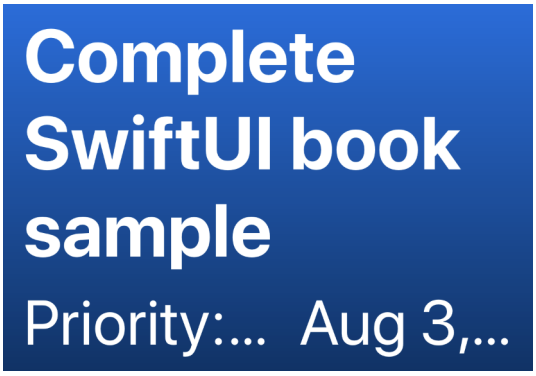
```
if item.notes != nil {
    ScrollView(.vertical) {
        Text(self.item.notes!)
```

```
                .padding()
        }
    }
```

...or around the entire view, taking care to move out the .navigationBarTitle() modifier to now sit on the scroll view:

```
    var body: some View {
        ScrollView(.vertical) {
            VStack(alignment: .leading) {
                « ... content ... »
            }
        }
        .navigationBarTitle("", displayMode: .inline)
    }
```

However, note that a scroll view, by definition, clips its contents. This means that the .edgesIgnoringSafeArea() modifier on the header background won't appear behind the navigation bar if you put the scroll view around the whole thing. If you go with the first option, however, and wrap only the notes, you'll likely notice a slight problem appear in the header at large type sizes:



This is an artifact of SwiftUI's layout system. To understand why this happens and how to fix it, you need to look at SwiftUI's layout model.

Understanding SwiftUI's Layout System

When SwiftUI looks to lay out a container's subviews, it follows a relatively simple process for each axis:

1. Create an initial budget for each subview by dividing the full amount of space equally.
2. Ask each view in turn how much space it needs, passing in its budget as an upper limit.

3. If the view doesn't use its full budget, the remaining amount is apportioned equally amongst the remaining views.

Let's look at these steps in some more detail.

SwiftUI first determines the amount available space, then it partitions that space equally between all of the available subviews, as shown here:

Figure 3—Basic Width Apportionment

The layout engine then looks at the kinds of subviews it's dealing with, separating them into two groups: *fixed* views are those whose sizes are defined either explicitly—by the `.frame()` modifier—or implicitly with an intrinsic size—for instance the bounds of an image. *Flexible* views are those that do not meet this criteria.

SwiftUI then takes each of the fixed views and informs them of their allotted space. In this example, the circle view will be offered one-third of the available width, and all of the height; the circle will reply, thanks to the explicit frame, that it requires only 18×18 plus some padding, taking somewhat less than what was offered. Once all of the fixed views have chosen their sizes, SwiftUI re-assesses the amount of space available for the flexible views to use and again apportions that equally between them. This results in the following prospective layout:

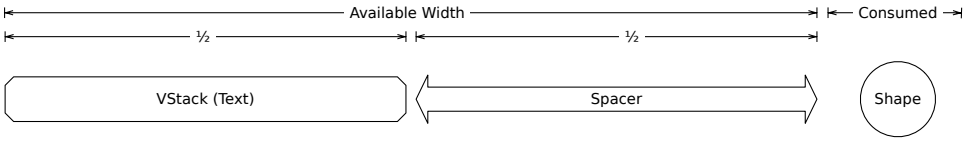


Figure 4—Fixed Widths

The layout engine hands the assigned size to each view, in turn, in layout order (leading vs. trailing, top-down vs. bottom-up, depending on the containing view's alignment settings), and each view will determine how much of that size it will use. If it uses less than that offered, then any remainder is divided among the remaining views, leaving the layout seen here:

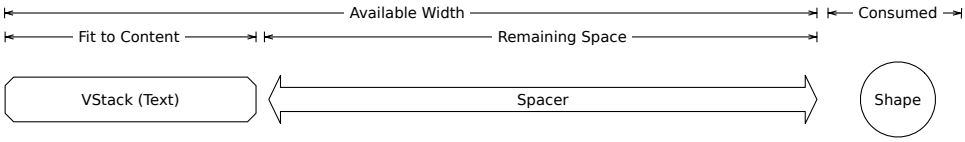


Figure 5—Variable Widths

For the header view you're working on in this chapter, the text is bumping up against the height constraint of the view. The title, priority, and date `Text` views all need to wrap their content onto at least a second line. The layout system notices that there isn't quite enough room for their ideal dimensions, so it gives precedence to the title, allowing it to expand. The priority and date fields are then forced by the height constraint to truncate themselves to a single line.

A similar thing used to happen in the first released versions of SwiftUI, where a `Spacer` view would always take all the proffered space, rather than shrinking to allow other views to grow. Thus a `Text`, `Spacer`, and `Text` in an `HStack` would end up using one third of the available width each. Apple resolved this by making implicit what here you'll have to make explicit: that one view's layout is more important than the other. For this, you use *layout priorities*.

In SwiftUI, all views have a layout priority, represented as a floating-point value. By default, this value is zero for everything, meaning all views are treated equally (it seems `Spacer` is a special case). When these values are not equal, the layout system takes an extra pass, grouping subviews by their priorities in descending order. Then, within each group, it will perform the fixed vs. flexible layout steps outlined above to apportion space, then take anything remaining on to the next-highest priority views, as shown in Figure 6, Explicit Priorities, on page 150. Note that some views have a minimum size, which is always respected—for instance, a `Text` view will not normally disappear completely: it will instead specify a minimum large enough to display a single ellipsis (…) character.

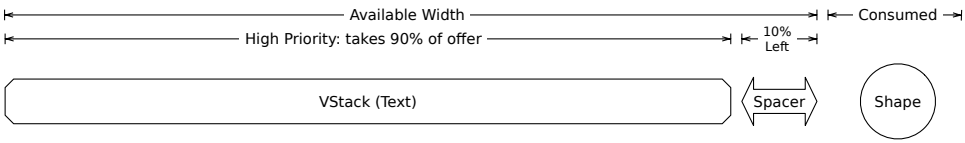


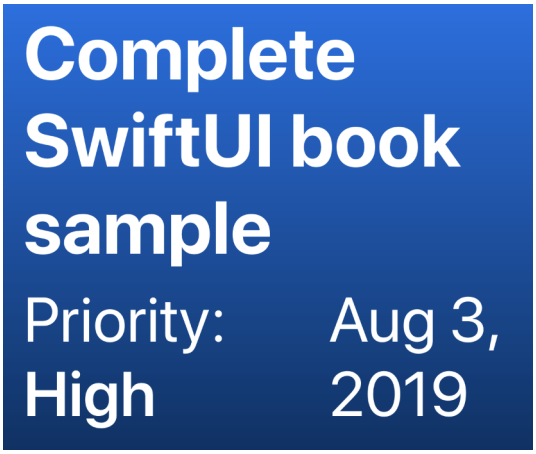
Figure 6—Explicit Priorities

In the current case, the ScrollView is accepting all the vertical space it's been offered (50% of the total), and the header view is left without enough room to display all its content. The use of named text types (headline, title, etc.) has allowed the layout system to prioritize the three-line title over the other text areas, but still, there's not quite enough room.

The solution to this is as simple as adding a new .layoutPriority() modifier to the header view, to raise its priority above that of the scroll view:

```
VStack(alignment: .leading, spacing: 8) {
    Text(item.title) {
        « ... content ... »
    }
    .foregroundColor(.white)
    .padding()
    .padding(.top)
    .background(« ... »)
➤   .layoutPriority(1)
    « ... »
}
```

Now the header renders in an expected manner:



With this step complete, let's look at how the canvas can help us preview and verify some other text-related changes: localization.

Previewing Localizations

Ideally, your application will support multiple locales and languages. The Xcode canvas offers an easy way to quickly preview how your localizations are shaping up, that translated text is appearing where it should, and to ensure your layout works for every language you support.

To use the next few examples, you need to add the localization files from the sample project to your own. You'll find them in code/p3/Do It/Resources/, where you'll see three folders: ar.lproj, en.lproj, and pl.lproj. These contain Arabic, English, and Polish versions of the Localizable.strings file containing translated versions of various words supplied by the app.

One of the localizations, Arabic, has been chosen explicitly to exercise one particular aspect of application localization: right-to-left layout. You may have noticed that alignments are named 'leading' and 'trailing' rather than 'left' or 'right.' This is because, in a right-to-left locale, the leading edge will be on the right side of the screen, and the trailing edge to the left. You'll use these three localizations in your next canvas update.

There are two parts to set related to your localization. The first part is the *locale*, the information around the language, and the common formats used for things like decimal points, currencies, times, and dates. The second part is the layout direction, either left-to-right or right-to-left. Given a list of locales and their corresponding layout directions, you can easily use a ForEach view to generate a preview using each localization and layout.

The values are passed into the preview's views using the SwiftUI environment. The .environment(_:_:) view modifier is what you'll use to pass in modified versions. This modifier takes two arguments. The first is a key-path expression, such as \.locale, used to identify which value to set. The second is an instance of the appropriate type for that value. For a locale, you use \.locale and an instance of Locale. For the layout direction, the parameters are \.layoutDirection and a case of the LayoutDirection enumeration—either .leftToRight or .rightToLeft.

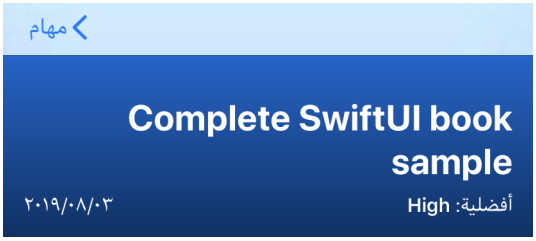
That gives you the pieces you need. Now put them together, creating an array of Locale–direction pairs to iterate over, then set the relevant environment variables on the NavigationView:

p3/Do It/TodoItemDetail.swift

```
let localePairs: [(Locale, LayoutDirection)] = [
    (Locale(identifier: "en-US"), .leftToRight),
    (Locale(identifier: "pl"), .leftToRight),
    (Locale(identifier: "ar"), .rightToLeft),
]

return ForEach(localePairs, id: \.self.0) { value in
    TodoItemDetail(item: todoItems[0])
        .environment(\.locale, value.0)
        .environment(\.layoutDirection, value.1)
        .previewDisplayName(value.0.languageCode ?? value.0.identifier)
}
```

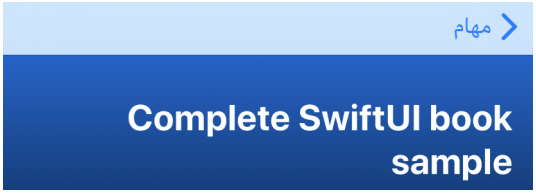
Now, the actual titles and notes from the to-do items won't be translated, but the back button title, the date, and the priority labels will be. Additionally, the Arabic locale will be laid out right-to-left, with any English text being written left-to-right but right-aligned. However, two errors immediately leap out:



The back button has been translated, but it appears on the left-to-right leading edge for some reason, and its chevron is pointing left—but in a right-to-left context, it should be pointing right.

The second error is that the priority hasn't been translated: it still reads "High," even though a translation was provided.

Happily, the first issue is a glitch with the previews and doesn't affect the real application. You can verify this by switching to TodoList.swift and applying the locale and layout direction environment changes to switch it to Arabic, then running live preview. Navigate into an item, and the back label correctly appears on the right side of the screen:



Localizing String Variables

The label is produced by the following code from TodoItemDetail.TitleOverlay.body in TodoItemDetail.swift:

```
Text("Priority: ") + Text(item.priority.rawValue.capitalized).bold()
```

The "Priority:" is being translated, so why not the other Text view's content? Neither is using the Text(verbatim:) initializer, so shouldn't this be translated automatically?

You can find the answer by looking at the initializers available on the Text structure:

```
1 public init(verbatim content: String)
2 public init<S>(_ content: S) where S : StringProtocol
3 public init(_ key: LocalizedStringKey, tableName: String? = nil,
              bundle: Bundle? = nil, comment: StaticString? = nil)
```

❶ This is the verbatim-text initializer, which does nothing special with its input.

❷ Here's the string-type initializer, which will accept values of type String or String.Substring. Note that only those two types conform to StringProtocol; the StaticString type used to represent text such as "this" in the code specifically does *not* conform to that protocol. Static inline strings will therefore not be matched against this initializer.

❸ This initializer has no argument label and takes an input of type LocalizedStringKey. Looking at that class, it conforms to ExpressibleByStringLiteral (via ExpressibleByStringInterpolation), so the use of an inline string argument without the verbatim: label will cause this initializer to be selected.

Looking at the arguments used when creating the Text views for the priority display, you'll see that the first Text instance is given a static string, so that will match the LocalizableStringKey initializer. The second takes a plain String, though, which will match against the StringProtocol initializer. Aha, that seems to be the issue then.

Test the hypothesis by modifying the code to create a new LocalizedStringKey from the priority string and use that to initialize the Text view:

p3/Do It/TodoItemDetail.swift

```
Text("Priority: ") +
    Text(LocalizedStringKey(item.priority.rawValue.capitalized)).bold()
```

Click Resume on your canvas, if necessary, and note that the priority names are now correctly translated.

Localizing the List

While we're looking at localization data, it would be useful to interact with the application in Live Preview mode to check that everything is working properly. For this, you should start at the List view.

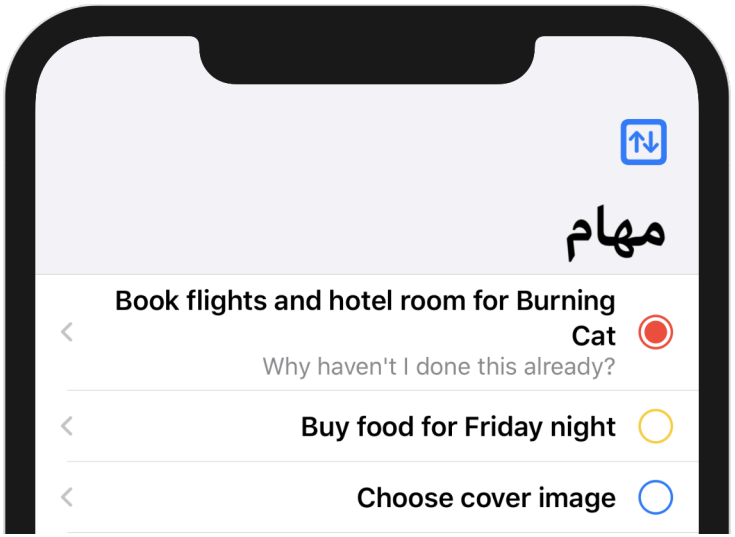
Open TodoList.swift and scroll down to the TodoList_Previews structure. Add the Arabic locale and right-to-left layout to the preview's TodoList:

p3/Do It/TodoList.swift

```
TodoList()
    .environment(\.locale, Locale(identifier: "ar"))
```

```
    .environment(\.layoutDirection, .rightToLeft)
```

Click Resume in the Xcode canvas, if necessary, to get the preview updated, and you'll immediately see the effects of the right-to-left layout.



The rows are set out correctly, with their contents' left-to-right ordering reversed. The color dot and arrow are on the left; the title and notes are on the right. The list's title is also displayed on the right.

Click the Play button at the bottom of the preview to enter Live Preview mode and select one of the rows. You'll see the detail view slide in from the left (which is now the trailing edge), and the content, including the back button you worked on earlier, all appears as expected. Click the back button to return to the list (note that the navigation bar's leading and trailing edges are now correct) and click on the Sort icon that's now at the top left of the list.

Argh, another problem! The title of the alert is translated properly, but the button titles aren't. The Localized.strings file definitely contains translations for these buttons, so what's causing this to go wrong here?

Scroll up in the editor toward the declaration for the ActionSheet and you'll see a familiar sight: the button titles are created using Text(opt.rawValue). As you saw earlier, this formulation is going to call the StringProtocol initializer for the Text view, so the solution here is the same: initialize a LocalizedStringKey from the option value, like so:

p3/Do It/TodoList.swift

```
ActionSheet.Button.default(Text(LocalizedStringKey(opt.rawValue))) {
    self.sortBy = opt
```

```
}
```

Update your live preview and try the sort button again. This time all the options should be correctly translated. The same should apply if you switch to the Polish localization as well.

The last place that needs some localization love is the TodoListChooser. Here the navigation bar title is correctly translated, but the "All Items" row title is not. The Row instance is initialized with a static string, though:

p3/Do It/TodoListChooser.swift

```
NavigationLink(destination: TodoList()) {
    Row(name: "All Items",
        icon: "list.bullet",
        color: .gray)
}
```

Scrolling down to the Row implementation, the lack of translation becomes clear: the name property has type String, so when it's passed into the Text initializer it won't be localized. There are several ways you can handle this, but the most straightforward for now is also the simplest: wrap the value passed into the Text initializer in LocalizedStringKey(), like so:

p3/Do It/TodoListChooser.swift

```
var body: some View {
    HStack {
        Image(systemName: icon)
            .foregroundColor(.white)
            .frame(width: 32, height: 32)
            .background(color)
            .clipShape(Circle())
➤       Text(LocalizedStringKey(name))
    }
}
```

With that in place, the "All Items" name will be correctly translated.

What You Learned

A lot of user interface debugging and testing takes place only by running the application and trying to get it to enter all the available states, one by one. The Xcode canvas helps out here by providing a quick and easy way to produce the same effects right next to the editor with a minimum of effort. As you've seen, it makes it easy to notice errors that you might not normally encounter at this stage of your application, and similarly easy to fix them.

Now you have the know-how to put the Xcode canvas to good use, ensuring your application's quality right from the word go. With this knowledge in

hand, it's time to move to a larger *application* canvas. In the next chapter you'll bring the app to iPadOS and look at implementing support for the unique capabilities of that platform.

CHAPTER 7

SwiftUI on iPadOS

Story Map

Why do I want to read this?

The iPad has a number of differences from the iPhone these days, beyond simply providing a larger canvas to work upon. Popovers, multiple windows, and support for keyboards and now even pointing devices. These all offer challenges to integrate successfully with SwiftUI.

What will I learn?

How to take your existing SwiftUI app and bring it to the iPad. You'll analyze the resulting UI and make some changes specific to each platform to make the app feel more at home on those devices. You'll learn to implement support for multiple windows and keyboard shortcuts, and provide pointer support in your UI.

What will I be able to do that I couldn't do before?

You'll know how the basic UI elements provided by SwiftUI adjust for different platforms, and you'll be able to provide system-specific variants of your UI while re-using as much of your existing UI code as possible. You'll also be able to wire in more features to the distinctly UIKit-based scene and window management APIs.

Where are we going next, and how does this fit in?

The iPad has supported drag & drop for a couple of versions now. It's a large topic with many nuances, so the next chapter will attempt to explore it thoroughly.

In 2019, at the same time SwiftUI was announced to the world, iOS was split in two. The iPad was growing more capable in each release, and the list of iPad-specific features was growing, as a result, the iPad operating system was officially renamed *iPadOS*. This signaled a distinct change to Apple's

approach to the device, and in early 2020 iPadOS 13.4 gained some functionality that would bring it closer in use to a laptop running macOS. Now your device could pair with and use wireless keyboards, mice, and trackpads—and the support for these would be absolutely first-class—neither an attempt to shoe-horn the macOS pointer onto a UI designed for fingers, nor a simple set of additional ways to do the same thing *Ed: That sentence is a lot to unpack; consider revising it.*. Like the windowing and scene systems announced in 2019, the iPad re-thought the purpose of a pointing device and adapted it precisely to the system.

In this chapter you'll take the application and see how to make use of iPad-specific features such as pointer support, multiple windows, drag and drop, and scene management. Some of these facilities have more support in SwiftUI than others, so you'll see how to put together the components described in iterative APIs with those in the declarative format of SwiftUI.

A certain amount of the content of this chapter deals with system-level capabilities that fall outside the purview of SwiftUI itself. The starter project for this chapter implements a lot of the work for you, such as implementing `NSItemProvider` types, `UNUserNotification` integration, and support for undo and redo via `NSUndoManager`. Rather than dive into the details of these subjects, you'll focus only on how to make use of them in SwiftUI code.

Introducing iPadOS

First things first: select the project in the Project Navigator, then select the "Do It" target, and finally the General tab. Under "Deployment Info," ensure that the target is set to "iOS 13.4," and that "iPad" is selected. Now launch the application on an iPad or iPad simulator. *Ed: Rather than use quotes for the UI elements, please use the keyword tag. You'll have to update not only this chapter but all of the chapters. You can do this on your revision pass.*

The first thing you'll notice is that the `NavigationView` now shows its root view and its descendants side-by-side. At least, it does that in Landscape mode—in Portrait, you have a blank screen and a back button in the top left that reads "Lists." Tapping that button makes the `Home` view slide in.

This isn't an ideal first-launch experience for users of your application. It would be best to have the two columns always visible next to one another, but unfortunately SwiftUI doesn't offer an easy way to do that at the moment (you'd have to use UIKit's `UISplitViewController` and a `UIViewControllerRepresentable` to create one). Aside from that, however, the detail view being empty on launch isn't good. Let's fix that.

So far your NavigationView instance contains a single List view, which makes sense. However, this isn't the only way to set up navigation: the view builder can actually accept two views to display, corresponding to a master/detail pair. The first view is a master list, while the second is the detail view that will appear to the right. The code in the Home view currently only specifies the master view, meaning the detail will initially be empty. To specify an initial state for the detail view, you can just add a TodoList instance following the initial List, and it will appear. Try adding TodoList(list: data.defaultItemList) at the end of the Home view's NavigationView view builder and relaunch the application. No empty views, even in portrait mode.

It would be useful to make these kinds of changes only while running on iPad. In the case of NavigationView, this behavior is built in, but you might want to make your own adjustments depending on whether your app is running on iPad or iPhone. The standard way to determine this in a UIKit app is for a view to fetch its attached UITraitCollection and look at the userInterfaceIdiom property. This is an enum type which tells you if your app is running on an iPhone, and iPad, an AppleTV, or in CarPlay. SwiftUI doesn't use UITraitCollection, though, so that option is unavailable. Instead, SwiftUI uses the environment to pass this type of information around.

In the starter project, open AccessoryViews/Environment.swift. You'll see an enum type here named InterfaceIdiom which acts as a wrapper for the corresponding UIKit type, UIUserInterfaceIdiom. You're going to initialize an instance of this type and write it to your root view's environment, so that views further down the hierarchy can access it with an @Environment property.

To create a new environment value, you need to create a type conforming to the EnvironmentKey protocol. This protocol has two requirements:

- It must specify a Value type. This is the type of property to be stored in the environment.
- It must have a static property of that Value type, named defaultValue. If an @Environment property attempts to retrieve a value from the environment where no value has been set, this will be returned instead.

The values you'd use for this seem straightforward: the Value type will be InterfaceIdiom, and the default value will be .unspecified. Define this type just below the InterfaceIdiom definition:

p8/Do It/AccessoryViews/Environment.swift

```
struct InterfaceIdiomEnvironmentKey: EnvironmentKey {
    typealias Value = InterfaceIdiom
    static var defaultValue: InterfaceIdiom = .unspecified
}
```

The SwiftUI environment itself is made available as an instance of the EnvironmentValues type. All the key paths you've passed into @Environment() property wrappers have actually been key paths on an EnvironmentValues type; it contains a great many extensions, each one fetching and/or storing values into some internal storage. If you look at the type's definition, you'll see that aside from its initializer and description, it contains one other API, a subscript operator:

```
public struct EnvironmentValues: CustomStringConvertible {
    public init()
    public var description: String { get }

    public subscript<K>(key: K.Type) -> K.Value where K: EnvironmentKey
}
```

This subscript provides read and write access to the underlying storage as if it were a Dictionary. Each key is the type of some EnvironmentKey (i.e. SomeEnvironmentKey.self), while the corresponding value is an instance of the same environment key's Value type. You have an environment key now, so creating your own property on EnvironmentValues is as simple as adding the following:

p8/Do It/AccessoryViews/Environment.swift

```
extension EnvironmentValues {
    var interfaceIdiom: InterfaceIdiom {
        get { self[InterfaceIdiomEnvironmentKey.self] }
        set { self[InterfaceIdiomEnvironmentKey.self] = newValue }
    }
}
```

This—a total of eight lines of code—is everything you need to do to provide your own environment values. Now you just need to set a value. Happily, UIWindowScene provides a traitCollection property that gives exactly what you need, so return to SceneDelegate.swift, navigate to the presentView() method, and make the following change to the definition of rootView:

p8/Do It/SceneDelegate.swift

```
let idiom = windowScene.traitCollection.userInterfaceIdiom
let rootView = view
    .environmentObject(sharedDataCenter)
    .environment(\.interfaceIdiom, InterfaceIdiom(idiom))
```

Now that the idiom is stored in the environment, let's update the Home view to make use of it. Open Home.swift and add a new property to the Home type:

p8/Do It/Home.swift

```
@Environment(\.interfaceIdiom) private var interfaceIdiom
```

Now scroll down to the end of the body implementation, to where you added the TodoList earlier. Replace that with the following implementation, which will check whether the app is running on an iPad before assigning the view:

p8/Do It/Home.swift
```
if interfaceIdiom == .pad {
    TodoList(list: data.defaultItemList)
}
```

You've now made your application launch experience much more pleasant for iPad users, and you now know how to pass information down the view hierarchy using custom environment values.

Popovers

If you tap on an "Add Item," "Edit," or "List Info" *Ed: Just a reminder to use the keyword tag on these UI elements rather than quotes. I won't call them all out, so keep an eye out for more instances in this chapter as well as the others.* button now, you'll see a large sheet appear from the bottom of the screen containing the appropriate editor view. It probably looks a little odd, though; it's clearly been designed around an iPhone-like view width, and it now has almost double that. Some of the interface elements, for instance the color picker, are now growing to quite large proportions to fill the available space, and that isn't ideal. On the iPad, this is solved through the use of *popovers*.

Popovers are small pop-up windows that appear to float above the main interface, and which typically are attached to the UI element that invoked them. They provide an easy way to show a small set of temporary components, and are perfect for showing the editor views that were designed for a narrower and smaller screen.

SwiftUI provides an API for displaying popovers that is very similar to the one used for displaying sheets. In fact, it's so similar that on iPhone it will just display the same content in a sheet instead. Like sheets, there are two methods available on `View`, each taking a binding to some state variable that will be used to determine whether the popover should appear. Also like sheets, one method binds to a simple `isPresented` boolean value, while the other uses a binding to some optional value, displaying when the item is non-optional. Both methods have two additional (optional) parameters though, which are unique to popovers. For instance:

```
extension View {
    public func popover<Content>(
        isPresented: Binding<Bool>,
        attachmentAnchor: PopoverAttachmentAnchor = .rect(.bounds),
        arrowEdge: Edge = .top,
        @ViewBuilder content: @escaping () -> Content
    ) -> some View where Content: View
}
```

The two new parameters are attachmentAnchor and arrowEdge. As noted above, a popover is typically attached to some other piece of UI—typically the button that caused it to appear. The attachmentAnchor specifies the location at which the popover will attach to its parent. The default value uses an Anchor.Source<CGRect> to provide the bounding rectangle of the parent view (see Working with Anchors, on page 110 for more information on anchors). Alternatively a UnitPoint can be used to indicate some location within or beyond the view's bounds, in unit-space coordinates.

The second new parameter tells the system which edge of the popover should be connected to its parent view. The default of .top specifies that the popover should appear below its parent on the screen, with its arrow pointing from its top to meet the location of the attachmentAnchor. For a button in a navigation bar, this is entirely appropriate. If your button were located at the bottom of the screen, however, you'd specify a value of .bottom instead, to show the popover above its parent. Likewise for popovers appearing close to the leading or trailing edges of the screen.

Since the popover() modifier falls back to displaying a regular sheet on iPhone, you can replace just about any use of .sheet() with a popover, safe in the knowledge that your code will work correctly everywhere. One important consideration, however, is where you'll attach that modifier. When using sheets, any view would do, as the sheet would just cover the entire screen; for popovers, the required anchor will be interpreted in terms of the view where the modifier is attached. This means that to attach a popover to a button you'll need to attach the modifier to that Button instance.

Let's start with the item detail view. Open TodoItemDetail.swift and locate the body implementation. Find the .sheet() modifier and delete it, then locate the editorButton property. Add a .popover() modifier to this button:

p8/Do It/TodoItemDetail.swift

```
var editorButton: some View {
    Button(action: {
        // «show editor»
    }) {
        // «image»
    }
    .accessibility(label: Text("Edit"))
    .popover(isPresented: $showingEditor, content: {
        self.editor
            .frame(idealWidth: 500, idealHeight: 600)
    })
}
```

Note that the editor has been given an explicit ideal width and height. The popover will be sized to fit its contents, but the List, Form, and ScrollView types will all happily shrink down to nothing—and those are what you're putting into the popover. Setting an ideal width and height here gives the system a hint as to the size it should allocate, while still allowing it to become larger or smaller under certain conditions. On the iPhone, for example, the width and height will grow to match the sheet's bounds.

Launch the application on an iPad or the iPad Simulator, then navigate into a to-do item and tap the edit button in the top right. A popover should appear containing the familiar editor interface. Tapping either the "Cancel" button or anywhere outside of the popover will dismiss it without saving any changes.

Next, open TodoList.swift. In here you previously used an enum value to specify which of two editors should be displayed by a single .sheet() modifier. Now that you're going to remove that and use two separate .popover() modifiers attached to two separate views, you no longer need that. Remove the EditorID type and the presentedEditor properties from the TodoList implementation, and replace them with two new boolean state properties:

p8/Do It/TodoList.swift
```
@State private var showItemEditor = false
@State private var showListEditor = false
```

Now find the body property and remove the .sheet() modifier. Now locate the addButton property and change it to use the new state to display a popover:

p8/Do It/TodoList.swift
```
private var addButton: some View {
    Button(action: {
        self.editingItem = Self.itemTemplate
        self.editingItem.listID = self.list?.id ?? self.data.defaultListID
        self.showItemEditor.toggle()
    }) {
        // «image»
    }
    .accessibility(label: Text("Add a new To-Do Item"))
    .popover(isPresented: $showItemEditor) {
        self.editorSheet
            .environmentObject(self.data)
            .frame(idealWidth: 500, idealHeight: 600)
    }
}
```

Next, in the editorSheet property, replace self.presentedEditor = nil in both button actions with self.showItemEditor = false. Lastly, in the barItems property definition, change the info button definition to show a popover:

p8/Do It/TodoList.swift
```
Button(action: { self.showListEditor.toggle() }) {
    // «image»
}
.popover(isPresented: $showListEditor) {
    TodoListEditor(list: self.list!)
        .environmentObject(self.data)
        .frame(idealWidth: 500, idealHeight: 600)
}
```

The last place to make changes is in Home.swift. Remove the sheet modifier from the body implementation there and replace it with a popover in the addButton property:

p8/Do It/Home.swift
```
private var addButton: some View {
    Button(action: { self.showingEditor.toggle() }) {
        // « ... »
    }
    .popover(isPresented: $showingEditor) {
        TodoListEditor(list: Self.listTemplate)
            .frame(minWidth: 500, minHeight: 600)
    }
}
```

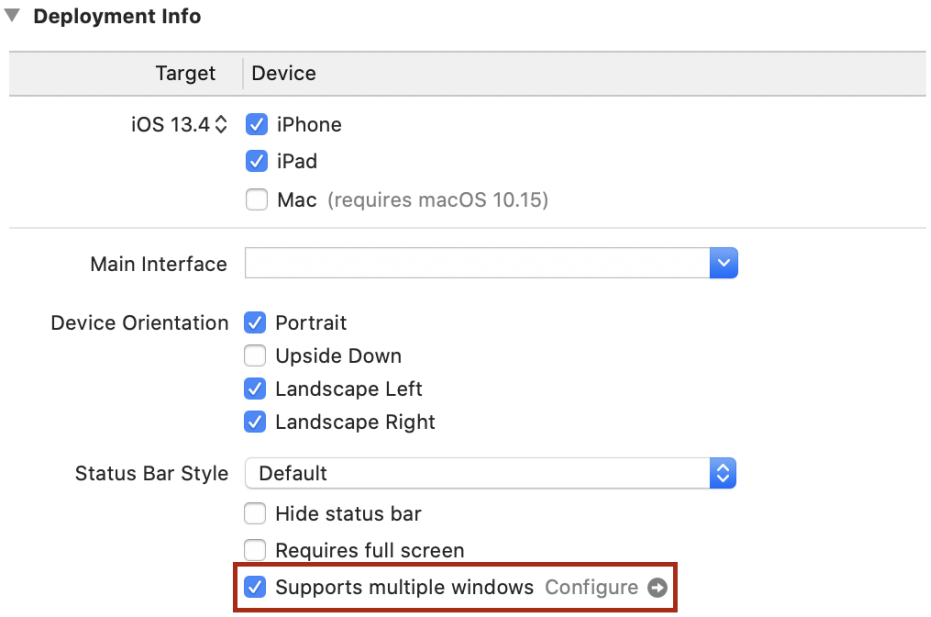
Multiple Scenes

In iPadOS, applications can be launched in a number of different ways, and can share the screen with one another. You typically pair two applications on one screen in one of two ways:

- Launch both applications, then swipe up from the bottom of the screen to enter the Dashboard. Drag one application's window on top of another to put them together on the same screen.
- While running one application, swipe up from the bottom of the screen a short way to reveal the Dock, then drag an application from the dock to the left or right of the screen.

In the first method, the two applications are always placed side-by-side in an adjustable split-screen mode. In the second, where you drop the new application makes a difference to how it opens: as you move to the right of the screen, for example, the icon will first expand to show an iPhone-sized window hovering over the top of the current application. Move a little closer to the screen edge and it will expand to fill that side of the screen, pushing the current application across into split-screen mode.

All of this works today—try it out and you'll see. What you currently *can't* do is open a second or third instance of "Do It" in this manner. To enable it, you'll need to make a small adjustment to the application's metadata. Open the project settings by selecting the project in the Project Navigator, then select the "Do It" application target in the project editor. In the Deployment Info section, select the checkbox next to "Supports multiple windows:"



With this single change, the application can be opened multiple times *Ed: There's a lot of passive voice in this chapter; although the copyeditor will likely pick it up, you may want to read through what you have to fix as many as you can. For example, you can revise this sentence to read: With this single change, you can open the application multiple times. Or if yu want to keep the reader out of it: With this single change, users can open the application multiple times. Either will work.* , allowing you to drag a second instance out of the Dock.

Each application instance on the screen is a *scene*. All scenes are part of the same running process, and share all data. Try opening two instances next to one another showing the same list, and then mark an item completed in one—the change will immediately be reflected in the other.

You'll learn more you can do with scenes in Dragging New Scenes, on page 192, but for now let's move on to look at hardware accessories.

Keyboard Commands

The iPad has worked with external keyboards for some time now, whether connected using Bluetooth or integrated with a stand or cover. Along with the ability to type on physical keys, though, this brings the capability of typing key chords to achieve effects within your application, like ⌘Z to undo, ⌘⇧Z to redo, or ⌘F to search.

UIKit provides a handy API for implementing key commands, but SwiftUI on iOS does not (though it does on macOS, so presumably official support will be forthcoming soon). Instead, you'll build a custom implementation that will serve until a first-party keyboard API for iOS arrives in a future version of SwiftUI.

Start by opening AccessoryViews/KeyCommands.swift. This file currently contains a single structure named KeyCommand, which is a SwiftUI-style wrapper for a UIKit UIKeyCommand instance. This is the type that you'll be using to define your supported key commands, and which will be used to pass the information along to UIKit in a way that it understands. In UIKit, UIKeyCommand objects are used to define the semantics of a key command while providing an Objective-C method *selector*—think of it as a 'function name' that can be applied to any object. When matching key presses are detected, that method is sent to the *responder chain*. You don't typically need to deal with responder chains when working with SwiftUI, but you can find a brief overview in A Little Responder Know-How, on page 170.

On top of this you'll build two components:

- A class named CommandRegistrar which will manage the list of installed key commands and vend Publisher instances that will deliver notifications when the user invokes a command.
- A subclass of UIHostingController to serve as the root view, which will vend a list of UIKeyCommand instances to the UIKit responder chain and provide an Objective-C method to serve as a target for all SwiftUI-based key commands.

Command Management

At the bottom of the file is a // MARK: - marker. Below that, create the CommandRegistrar class:

p8/Do It/AccessoryViews/KeyCommands.swift

```
fileprivate final class CommandRegistrar {
    typealias CommandPublisher = PassthroughSubject<KeyCommand,Never>
    var publisher = PassthroughSubject<KeyCommand,Never>()
```

```
    var commands: Set<KeyCommand> = []

    func install(command: KeyCommand) -> AnyPublisher<KeyCommand, Never> {
        // « ... »
    }

    func remove(command: KeyCommand) {
        commands.remove(command)
    }
}
```

This class is straightforward: it maintains a Set of KeyCommand instances along with a PassthroughSubject that publishes KeyCommand instances. When the user presses a key combination that triggers a command, that command will be sent to this publisher. With a little magic, the appropriate command will invoke a block provided by a view to respond to that command. The magic lives in the install(command:) method:

p8/Do It/AccessoryViews/KeyCommands.swift

```
func install(command: KeyCommand) -> AnyPublisher<KeyCommand, Never> {
    commands.insert(command)
    return publisher
        .filter { $0 == command }
        .eraseToAnyPublisher()
}
```

This is a small function, but it does a lot of work. When a KeyCommand is installed it's added to the commands set, and then a new Publisher is returned that is specific to that command. This is accomplished by a filter() modifier to the class's publisher property. This will create a new publisher that only publishes its input if it's the same KeyCommand that was passed to install(). The result is then type-erased to an AnyPublisher instance and returned.

Receiving Events

Next you need to actually send something to the root publisher. For that, you'll create a single CommandRegistrar instance, and you'll build a subclass of UIHostingController that implements the action method used by the KeyCommand type. This will also implement the keyCommands property from UIResponder to return all the commands stored by the CommandRegistrar. The event system in UIKit uses types conforming to UIResponder to deliver common events, and the *first responder* is the starting point when it looks for event receivers, and is typically a view or a control.

This all takes very little code to implement, added just below CommandRegistrar:

p8/Do It/AccessoryViews/KeyCommands.swift

```
fileprivate let keyCommander = CommandRegistrar()
```

```
final class KeyCommandHostingController<Content: View>: UIHostingController<Content> {
    override var canBecomeFirstResponder: Bool { true }

    override var keyCommands: [UIKeyCommand]? {
        keyCommander.commands.map { $0.uikit }
    }

    override func swiftUIKeyCommand(_ sender: UIKeyCommand?) {
        guard let command = KeyCommand(sender) else { return }
        keyCommander.publisher.send(command)
    }
}
```

The first item inside KeyCommandHostingController is an override of the canBecomeFirstResponder property from UIResponder. The UIHostingController doesn't implement this property, so this subclass is required in order to turn on that functionality.

A Little Responder Know-How

In UIKit, as in AppKit before it, the event system is based around the concept of *responders*, organized in a chain. In UIKit, this is represented by the UIResponder class, which defines the basic event handling semantics of the user interface, such as methods to handle touches, pressing, motion, and more. UIKit views and controls are all responders, as are components like UIScene, UIApplication, and (typically) their delegates.

The primary concept of the responder chain is that there is a single 'active' view, which is called the *first responder*. This will usually be the focused control or view. Any events that arrive are sent to that object first, but if they're not handled, then the *next responder* is queried instead. Each UIResponder identifies the next responder in sequence. For example, a UIView will indicate either its view controller (if it has one) or its superview. In this way, the event is delivered to the highest-placed responder, with the system working downwards until it finds one that accepts it.

With the view controller in the responder chain, it will now be queried by the system for the details of any key commands that it might support via the keyCommands property. This implementation returns the embedded UIKeyCommand objects held by each KeyCommand instance within the shared keyCommander.

The key commands all specify an action selector for UIResponder.swiftUIKeyCommand(_:) (defined just following the KeyCommand type itself). All objects in the responder chain will be queried to see if they respond to this method, eventually arriving at this hosting controller. The method is implemented here to wrap the UIKit class in a KeyCommand structure then send it using keyCommander's publisher.

You'll now need to use this new class as your root view controller. Open SceneDelegate.swift and locate the presentView(_:in:) method. Find the line where a UIHostingController is created and assigned to the window, and replace it with a KeyCommandHostingController:

p8/Do It/SceneDelegate.swift

```
let window = UIWindow(windowScene: windowScene)
window.rootViewController = KeyCommandHostingController(rootView: rootView)
```

Handling Key Commands

Now you have storage and management for key commands, and you've hooked them into the responder chain. It only remains to devise a suitable API so that views can register their key commands and handle them.

A simple API is the best, especially in SwiftUI. Therefore, let's use a simple method on View that takes a KeyCommand and a block to invoke. This method can then call keyCommander.install() and use the returned publisher to invoke View.onReceive(); this leaves SwiftUI in charge of the details of correctly managing threads and schedulers for the publisher.

In addition, a helper method that takes some of the component parameters of a KeyCommand would also be useful, allowing a view to just specify a command's title, input, and modifiers, for example. You can provide default values for each argument so the caller can provide as much or as little detail as necessary for their use case.

Put together, that gives you a simple API, implemented in a View extension. Return to AccessoryViews/KeyCommands.swift and place it at the end of the file:

p8/Do It/AccessoryViews/KeyCommands.swift

```
extension View {
    func onKeyCommand(
        _ command: KeyCommand,
        perform: @escaping () -> Void
    ) -> some View {
        onReceive(keyCommander.install(command: command).map { _ in () },
                  perform: perform)
    }

    func onKeyCommand(
        title: String,
        input: String,
        modifiers: KeyCommand.ModifierFlags = [],
        attributes: KeyCommand.Attributes = [],
        discoverabilityTitle: String? = nil,
        perform: @escaping () -> Void
    ) -> some View {
        let command = KeyCommand(title: title, input: input,
```

```
                                    modifierFlags: modifiers,
                                    attributes: attributes,
                                    discoverabilityTitle: discoverabilityTitle)
        return onKeyCommand(command, perform: perform)
    }
}
```

It's time to put your new API to the test. Let's add a keystroke of ⌘N to create a new to-do item. There's an existing way to do this via a button in TodoList, so by adding the key command there you can make use of the same facilities to show the item editor.

Open TodoList.swift. Somewhere above the TodoList type definition, define a KeyCommand, giving it a title and a discoverability title—the former is a short name, while the latter is more descriptive of the command's intent:

p8/Do It/TodoList.swift

```
fileprivate let newItemCommand = KeyCommand(
    title: NSLocalizedString("New Item", comment: "Key command title"),
    input: "n",
    modifierFlags: [.command],
    discoverabilityTitle: NSLocalizedString("Create a new to-do item",
                                comment: "Key command discoverability title"))
```

Now locate the TodoList view's body implementation and add a call to your new .onKeyCommand() modifier to the List view declaration:

p8/Do It/TodoList.swift

```
List(selection: $selectedItems) {
    // « ... »
}
// « view modifiers »
.onKeyCommand(newItemCommand) {
    guard !self.showItemEditor && !self.showListEditor else {
        return
    }
    self.editingItem = Self.itemTemplate
    self.editingItem.listID = self.list?.id ?? self.data.defaultListID
    self.showItemEditor.toggle()
}
```

After checking that no popovers are being presented already, the attached block performs the same steps as the "Add Item" button in the navigation bar.

Now let's add a command to pop open the list editor, as though the user had tapped on the "Info" button. Put the KeyCommand next to the previous one:

p8/Do It/TodoList.swift

```
fileprivate let listInfoCommand = KeyCommand(
```

```
    title: NSLocalizedString("List Info", comment: "Key command title"),
    input: "i",
    modifierFlags: [.command],
    discoverabilityTitle: NSLocalizedString("Show or edit list properties",
                                comment: "Key command discoverability title"))
```

Place the modifier on the List view, just as before:

p8/Do It/TodoList.swift

```
List(selection: $selectedItems) {
    // « ... »
}
// « view modifiers »
.onKeyCommand(newItemCommand) {
    // « ... »
}
.onKeyCommand(listInfoCommand) {
    guard !self.showItemEditor && !self.showListEditor, let _ = self.list else {
        return
    }

    self.showListEditor.toggle()
}
```

Since you've provided a key command to create a new item, it seems appropriate to do the same for lists. Open Home.swift and define the key command to use ⌘⇧N:

p8/Do It/Home.swift

```
fileprivate let newListCommand = KeyCommand(
    title: NSLocalizedString("New List", comment: "Key command title"),
    input: "n",
    modifierFlags: [.command, .shift],
    discoverabilityTitle: NSLocalizedString("Create a new to-do list",
                                comment: "Key command discoverability title"))
```

Attach the key command handler to the ForEach view. It only needs to toggle the showingEditor state variable on and off:

p8/Do It/Home.swift

```
ForEach(data.todoLists) { list in
    // « ... »
}
// « onDelete »
// « onMove »
.onKeyCommand(newListCommand) {
    self.showingEditor.toggle()
}
```

Launch the app on an iPad with an attached keyboard, or on the iPad Simulator. If using the simulator, make sure you turn on "Capture Keyboard" using

either the button on the toolbar (labeled ⌘) or from the menu in IO → Input → Send Cursor to Device. Press the various key combinations to see them take effect.

That's not all you can do, though. Your DataCenter has real Undo/Redo support now, and the key commands for that are built-in. Try deleting an item then pressing ⌘Z to see it reappear. Then press ⌘⇧Z to make it disappear again. You can also use the three-finger tap gesture to perform the same built-in actions. This all comes for free due to the KeyCommandHostingController class you created earlier: simply by enabling canBecomeFirstResponder on the root view your SwiftUI app gets to benefit from some of the built-in features of UIKit's event model, automatically.

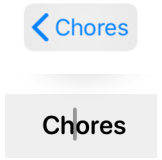
Pointing Devices

iPadOS 13.4 added first-class support for trackpads and mice, with a new cursor/pointer interaction model. SwiftUI provides some means to respond to this, though not quite as full as that provided by UIKit. You can still make good use of it to help snap the cursor to important UI elements in a similar manner to Apple's own applications, however.

SwiftUI provides two methods on View to work with pointer interaction:

- onHover(perform:) registers a block that will be run whenever the pointer enters or exits the bounds of its view. The block will receive a single boolean parameter indicating whether or not the pointer is inside the view's bounds.
- hoverEffect(_:) specifies a HoverEffect to automatically perform when the pointer is within a view's bounds. This uses the same facilities as UIKit to provide one of two standard appearances for the interaction, each using a light source effect to provide some dynamism as the pointer moves within the view's bounds:
  - .highlight will change the pointer into a platter located behind the highlighted view. This is the default effect for buttons.
  - .lift will simulate a light source casting a shadow behind the view, giving it the impression of being lifted above its surroundings. This is the effect used by icons on the iPad home screen, and is similar to the parallax effect of selectable items on tvOS.
  - A third value, .automatic, will let the system choose an appropriate value.

Many standard system components already implement pointer interactions. For example, the back button in a navigation bar already uses the highlight appearance, and any editable text morphs the pointer into a vertical bar cursor to aid precise inter-character selection:



Glitches Ahead!

At present (iPadOS 13.4) the SwiftUI implementation of hover effects leaves something to be desired. When it works correctly everything is fine, but in some cases it has some issues with its appearance. Regular buttons and text will frequently be given a solid white background, making the .highlight platter all but invisible. In contrast, when applied to a button inside a navigation bar, the effect, which takes a copy of the view to which it applies, leaves transparency in place in the copy, leaving you with one semitransparent icon floating over another. On top of this, the effect doesn't pad itself to expand beyond the content of its view, so it will be very tightly clipped.

These issues are sure to be resolved soon—maybe even by the time you read this book—but for now you should be aware that this is the case. You'll resolve the last item above with a custom view modifier in [xxx](#sec.hover.bounds.fix).

Generally speaking, you'll want to implement hover effects on any custom interface components you've created. In this application, that means the list editor with its color and icon selectors. The color wheel's touch interaction works well for pointers already—just clicking and dragging reveals and moves a loupe, and for selecting precise colors fine precision is the order of the day. Everything below that would benefit from a little help though, to snap the pointer between elements and clearly indicate which item is active.

Let's start with the color selectors. Open AccessoryViews/ColorPicker.swift. The selector buttons are all simple circles with their appearance defined by a custom button style at the top of this file. To apply the hover effect to all the buttons, you can simply add it to the button style. In ColorButtonStyle.makeBody(), add a .hoverEffect() modifier at the end of the method. If you launch the app and try it out, though, you'll see that the highlight platter fits very tightly around the button, making it less obvious and honestly not very appealing. To fix this, use this simple trick to enlarge the view's bounds for the hover effect, then reduce them back down for layout:

p8/Do It/AccessoryViews/ColorPicker.swift

```
fileprivate struct ColorButtonStyle: ButtonStyle {
    func makeBody(configuration: Configuration) -> some View {
        configuration.label
            .overlay(Circle().stroke().foregroundColor(.white))
            .modifier(DoubleShadow(configuration.isPressed ? 1 : 6))
➤           .padding(.vertical, 5)
➤           .hoverEffect()
➤           .padding(.vertical, -5)
    }
}
```

Now the selection highlight looks a lot better:

The same approach works for the icon selector. Open AccessoryViews/IconChooser.swift and add the extra modifiers to IconChoiceButtonStyle.makeBody():

p8/Do It/AccessoryViews/IconChooser.swift

```
func makeBody(configuration: Configuration) -> some View {
    configuration.label
        .font(.system(size: 24, weight: .bold, design: .rounded))
        .padding(6)
        .frame(width: 30)
        .padding(14)
        .background(background)
        .scaleEffect(configuration.isPressed ? 1.2 : 1)
➤       .padding(6)
➤       .hoverEffect()
➤       .padding(-6)
}
```

This has a similar appearance to the color selector buttons:

Fixing Hover Effect Bounds

Twice now you've used an pad-and-shrink approach to give the hover effect a larger area than the view it surrounds. If you attach a plain .hoverEffect() to any of the text buttons in your navigation bar now, you'll see that the same

problem exists there, and it appears even worse when surrounding text, because at least around a circular button there's plenty of empty space for the platter to appear. On a text view that isn't true, though, the bounds of the letters themselves reach all the edges of the view.

Rather than keep manually adding and removing padding everywhere, let's create a single ViewModifier that will apply the effect for us. It can use an API like the .padding() modifier, and will apply the requested padding, apply the hover effect, then remove the padding. There are several ways to specify padding, though:

- A simple numeric value, applied to all edges.
- An Edge.Set specifying which edges of the view should be padded, and how wide the padding should be. Both of these have default values, so this is actually the version you're using when you type .padding().
- An EdgeInsets instance giving explicit amounts of padding for the top, bottom, leading, and trailing edges of the view.

It turns out that the first two methods can be implemented in terms of the third (and this is what SwiftUI does internally, more or less). That means your modifier only needs to keep track of two things: the HoverEffect to apply, and the EdgeInsets for use for padding.

That implies the following straightforward implementation, which you should put in AccessoryViews/ViewModifiers.swift:

p8/Do It/AccessoryViews/ViewModifiers.swift

```
struct NicelyHoverable: ViewModifier {
    private let padding: EdgeInsets
    private let effect: HoverEffect

    init(_ insets: EdgeInsets, _ effect: HoverEffect = .automatic) {
        self.padding = insets
        self.effect = effect
    }

    func body(content: Content) -> some View {
        content
            .padding(padding)
            .hoverEffect(effect)
            .padding(-padding)
    }
}
```

Crafting an entire structure to specify padding is a little onerous though, so let's add a simplified initializer that will create it from a width and a set of edges:

p8/Do It/AccessoryViews/ViewModifiers.swift
```
init(_ padding: CGFloat = 8, _ edges: Edge.Set = .all,
     _ effect: HoverEffect = .automatic) {
    self.padding = EdgeInsets(
        top: edges.contains(.top) ? padding : 0,
        leading: edges.contains(.leading) ? padding : 0,
        bottom: edges.contains(.bottom) ? padding : 0,
        trailing: edges.contains(.trailing) ? padding : 0
    )
    self.effect = effect
}
```

Modifiers are typically accessed through functions on View, though, so scroll down to the View extension and add two new functions which map to the modifier's two initializers:

p8/Do It/AccessoryViews/ViewModifiers.swift
```
func niceHoverEffect(_ padding: CGFloat = 8, _ edges: Edge.Set = .all,
                     _ effect: HoverEffect = .automatic) -> some View {
    modifier(NicelyHoverable(padding, edges, effect))
}

func niceHoverEffect(_ insets: EdgeInsets, _ effect: HoverEffect = .automatic) -> some View {
    modifier(NicelyHoverable(insets, effect))
}
```

Now you're ready to put it to work. Start by opening Home.swift, where you'll update the buttons in the navigation bar. Locate the body implementation, and update the .navigationBarItems() modifier by appending .niceHoverEffect() to each button:

```
List {
    // « ... »
}
// « other modifiers »
.navigationBarItems(
    leading: EditButton().niceHoverEffect(),
    trailing: addButton.niceHoverEffect())
```

Launch your application and try it out. The platter around the "Edit" button seems correct, but the "Add" button's platter doesn't seem tall enough:

As it turns out, everything has indeed worked exactly as requested. The problem here is that images using SF Symbols draw themselves much taller than their bounds. If you create a similar button in an empty view and select it in a preview, you can see where the image view's bounds really lie:

~[](images/iPadOS/ButtonBounds.png)

To remedy this, you'll need to increase the vertical padding a little on all your icons. Since there are a lot of icons to cover, let's add another helper function the View extension in ViewModifiers.swift. This will be a version of the simple niceHoverEffect() method, but internally it will create a new EdgeInsets and add a little to the top and bottom values:

p8/Do It/AccessoryViews/ViewModifiers.swift

```
func iconHoverEffect(_ color: Color = Color(.systemGroupedBackground),
                     _ padding: CGFloat = 8,
                     _ effect: HoverEffect = .automatic) -> some View {
    let insets = EdgeInsets(top: padding+6, leading: padding,
                            bottom: padding+6, trailing: padding)
    return modifier(NicelyHoverable(insets, effect))
}
```

Hard-Coded Numbers

The 6-point increase is purely based on what happens to look "about right" in this application. Ideally a better implementation of hoverEffect() will be forthcoming in a newer iPadOS software release.

Apply this new modifier in the Home view to fix the problem on the "Add" button:

That looks much better. Now look and see what other elements might benefit from a hover effect, and see whether they need the custom niceHoverEffect() or iconHoverEffect(), or whether a plain hoverEffect() will work on its own.

What You Learned

This has been a long chapter, and you've learned a lot about the special facilities of iPadOS and how to take advantage of them. Specifically, this chapter covered:

- Displaying multiple scenes from your application side-by-side.
- Making good use of popovers rather than large screen-filling modal sheets.
- Managing the appearance of navigation views in a two-pane system.
- Providing support for the new pointer interaction model in iPadOS 13.4.
- Implementing key command handling, including the necessary wiring to plug into UIKit's APIs and event propagation system.

Right now, you might be wondering what particular utility is gained from having two instances of your application open side-by-side. In the next chapter, you'll see the answer to that, as we look at integrating drag & drop support throughout the user interface.

CHAPTER 8

Implementing Drag and Drop

Story Map

Why do I want to read this?

Drag and drop has become an important part of a good iPad app, and it's crucial to support it wherever possible.

What will I learn?

You'll learn how SwiftUI enables you to quickly and easily add drag and drop support to your application, see how to integrate the object-oriented item provider APIs with a structure-based data model, and how to drag items to create new scenes.

What will I be able to do that I couldn't do before?

You'll have a full grasp of the components of drag & drop in SwiftUI applications, and you'll be able to implement top-quality support for it in just a few lines of code.

Where are we going next, and how does this fit in?

Now you've worked with several different ways of presenting and interacting with your data. Many applications use a relational model, so in the next chapter you'll see how SwiftUI can be used with a data model defined using Core Data.

iPadOS has some first-class support for drag-and-drop, and you can already take advantage of much of it without writing any code. If you open a to-do item editor, for example, you can select and drag text to or from any text field. You can select text in another app—for instance Safari—running in split screen, and drag that directly into the text view for item's `notes` property. Many system controls and views already know how to deal with various types of data via drag & drop, in fact, including lists via the `ForEach` view. In this latter

case, however, you'll need to do a little extra work to tell SwiftUI what types of data you want to send or receive, and how to handle that data as it arrives.

Understanding Item Providers

Moving data around using drag & drop is accomplished using instances of NSItemProvider. This class encapsulates information about some piece of information and the various ways it can be encoded or decoded. Each encoded data format is represented by a *Uniform Type Identifier* or UTI, which is a string containing different identifiers in reverse-DNS format: the first element has the greatest scope, the last has the narrowest scope. These identifiers then have a hierarchy, where more narrowly-defined types conform to less narrowly-defined ones. Thus the public.text UTI contains all types and encodings of text, whether ASCII, UTF-8, UTF-32, or whether plain-text or formatted such as HTML or XML. If you only handle plain text, you'd use public.plain-text, or public.utf8-plain-text to only accept UTF-8 encoded data. If you want XML, then you'd use public.xml. See Figure 7, UTI Conformance, on page 183 for some examples of UTI conformance.

The NSItemProvider type was designed in Objective-C, and most of the API assumes it's working in terms of model *objects*. These objects would then conform to and implement protocols to make themselves usable by the item provider system. When using Swift struct types, however, it takes a little more work to set things up. For this reason an item provider has been created for you in this chapter's sample project—look in Do It/Model/ItemProvider.swift to find classes for sending and receiving data through an NSItemProvider. Most of the content of the file is out of the scope of a SwiftUI book, though your author would encourage you to look at it for some ideas on how you might implement something similar in your own application. Suffice to say that you'll use a TodoItemProvider when dragging an item or list out of the app, and an ItemReceiver when receiving something dragged from elsewhere.

At the top of the file, however, are some important definitions:

p9/Do It/Model/ItemProviders.swift

```
Line 1 let todoItemUTI = "com.pragprog.swiftui.todo.item"
     2 let todoListUTI = "com.pragprog.swiftui.todo.list"
     3 let todoItemUUIDUTI = "com.pragprog.swiftui.todo.item.uuid"
     4 let todoListUUIDUTI = "com.pragprog.swiftui.todo.list.uuid"
     5 let rawTextUTI = kUTTypeUTF8PlainText as String
     6 let jsonUTI = kUTTypeJSON as String
```

Here you have the definitions of the UTIs supported by the application for vending and receiving to-do items and lists. On lines 1 and 2 are the identifiers

for JSON-encoded items and lists. On lines 3 and 4 are identifiers for a type of data only usable within the application itself: these encode just the unique identifier (UUID) of the item being provided. Lastly, on lines 5 and 6 are some more Swift-friendly declarations of the public.utf8-plain-text and public.json identifier, found in the CoreServices framework. When using the former, the item or list is encoded as a simple formatted string, suitable for dragging into a text editor, while the latter is used only when receiving plain JSON data that may or may not contain to-do item or list data.

Since two of these type identifiers are designed to export data in a structured format, their details have been added to the project's Info.plist. If you open the project editor and select the application target, then the "Info" tab, you'll see that there are two entries in the "Exported UTIs" section corresponding to the entries on lines 1 and 2 of ItemProviders.swift. Importantly, they are declared to conform to the UTI of JSON data, public.json; this means that they might be dragged to any application that works with generic JSON data, for instance a javascript editor. The figure below shows the conformance hierarchy for the types used by "Do It:"

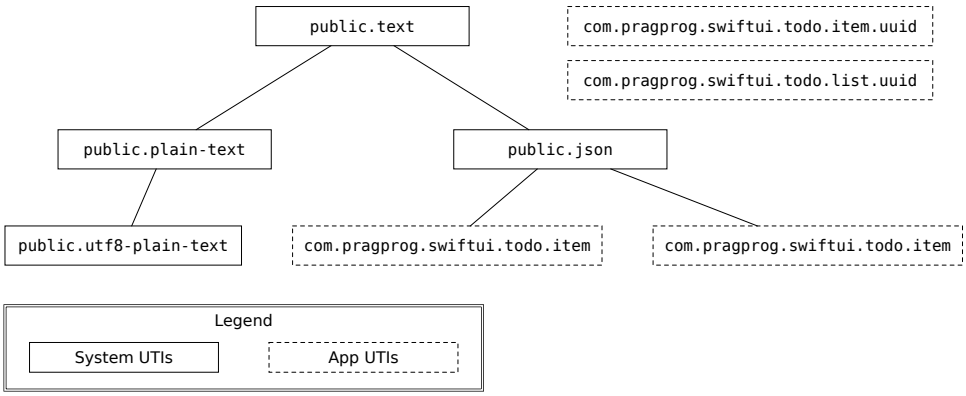


Figure 7—UTI Conformance

Dragging Out

The ItemProvider class does just about everything you need, but first you have to obtain one. Since ItemProvider needs a reference to the DataCenter, it's logical to provide API on DataCenter to provide them.

Open Model/DataCenter.swift and add an extension to define two new functions:

p9/Do It/Model/DataCenter.swift

```
// MARK: - Drag & Drop
```

```
extension DataCenter {
    func itemProvider(forList list: TodoItemList) -> NSItemProvider {
        let provider = TodoItemProvider(dataCenter: self, list: list)
        let result = NSItemProvider(object: provider)
        return result
    }

    func itemProvider(forItem item: TodoItem) -> NSItemProvider {
        let provider = TodoItemProvider(dataCenter: self, item: item)
        let result = NSItemProvider(object: provider)
        return result
    }
}
```

Now you'll use this to implement item dragging from the list view.

There are two sets of methods used to implement drag and drop in SwiftUI on iOS. One set, provided when SwiftUI launched in iOS 13.0, is tailored exclusively to lists, while the other more generic variant arrived in iOS 13.4.

The first set consists of two view modifiers:

- View.itemProvider(_:) allows you to attach a block which will vend an optional NSItemProvider instance relating to the view's content.
- DynamicViewContent.onInsert(of:perform:) will call the provided block when a drop occurs of one of a set of allowed UTIs, along with the index at which the items was dropped. This is specifically for list types, and enables animations on the list view, moving rows out of the way as an item is dragged over.

The second, more general API involves some modifiers on the View type, and which apply to any view, not only lists:

- onDrag(_:) operates in a manner similar to the older itemProvider(_:), except that the provided block is not optional, and returns a non-optional NSItemProvider.
- onDrop(of:isTargeted:perform:) and onDrop(of:delegate:) provide drop support for any view, although at this time this does *not* appear to include List, Form, or ForEach views.

The drop API is quite flexible, and in fact contains three APIs, two of which have the same selector but accept different blocks. You'll learn more about these later in this chapter.

Dropping on Lists

One drawback in SwiftUI is that in any List or Form view on iOS, this is the *only* way to receive items via drag-and-drop, and it *always* acts as though a new item will be

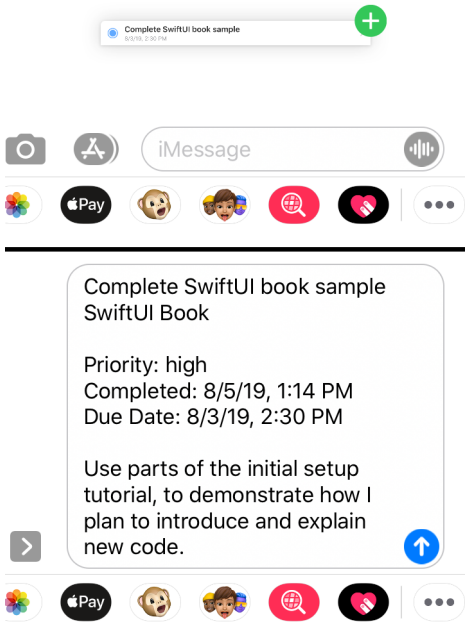
inserted into the list. None of the various onDrop() modifiers appear to work on a List or its contents at all, and any attempt to use onInsert() to add an item will cause existing list contents to move out of the way, making it impossible to, say, drag an item onto a list row to add that item to that list. Hopefully this will be resolved in a future version of SwiftUI.

Open TodoItemRow.swift and update its body property to add an onDrag(_:) modifier to the outermost HStack, returning an item provider for that row's item:

p9/Do It/TodoItemRow.swift

```
var body: some View {
    HStack {
        // « ... »
    }
    .onDrag { self.data.itemProvider(forItem: self.item) }
}
```

And… there is no step two *Ed: Not sure you need this sentence*. Launch your application on an iPad, or the iPad simulator, then open another app next to it in which you can type—Messages is available on the simulator and works nicely. Navigate into an item list and then drag the item out and onto the Messages compose window. When you drop it, the text version of your data will appear in the new message field, as shown below:

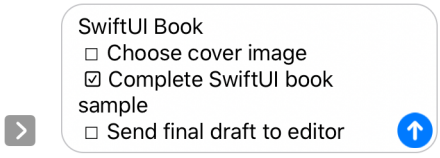


Implementing the same functionality for lists is nearly identical; this time, open Home.swift and add the onDrag(_:) modifier within the body property of the Home view itself, attached to the Row instance:

p9/Do It/Home.swift

```
ForEach(data.todoLists) { list in
    NavigationLink(destination: TodoList(list: list).font(nil),
                   tag: .list(list.id),
                   selection: self.$selection) {
        Row(name: list.name,icon: list.icon, color: list.color.uiColor)
            .onDrag { self.data.itemProvider(forList: list) }
    }
}
```

Repeat the experiment with the Messages app, this time dragging a list out of the Home view, and you'll be rewarded with something like this:



Dragging In

So far you've seen how to export items and lists via drag & drop, but that's only using the string exported. The other supported UTIs are all specific to this application, or to an application using the same data types, and they're designed to let you drag items around within the same application. For example, you might drag items into different lists rather than opening each item's editor individually and navigating the interface there to select a list.

When dragging into a list, you use the `onInsert(of:perform:)` modifier to decide what you'd like to receive and to handle any dropped items. This is the older API which is intended for inserting a new item into a list, and this is the only option available to lists (see Dropping on Lists, on page 184).

Return to `TodoList.swift`. Scroll down toward the bottom of the "Helper Properties" extension and locate the `droppableUTIs` property:

```
private var droppableUTIs: [String] {
    return []
}
```

Let's modify this to return values appropriate for the type of data being displayed. If a list is being shown, then todo items can be dropped onto it, along with plain strings (which you'll use to create a new item). If one of the group types is being displayed, then the suitable interactions are much fewer:

- "Today:" For an item to appear here, it needs to have a due date today. While you might drag an item onto this list to assign it a due date of midnight tonight (for instance), this doesn't really feel ideal. If such an operation were useful, dropping it on the button in the `HomeHeader` would seem more appropriate, if anything.
- "Scheduled:" This shows anything with a due date, across the entirety of time. It doesn't seem useful as a drop target—what date would it assign, and what would the index of its drop location mean?
- "Overdue:" Dropping here would… what? Mark an item overdue? What date would it assign? Again, what would the drop location imply?
- "All Items:" This alone seems appropriate as a drop target. Any item already in the data store wouldn't be usefully dropped (it's already there), but new items in JSON and new items from strings would be fine. You need only ensure that the item landed in the right index within the global item list, to ensure it doesn't "jump" to the end after being dropped.

With this in mind, update the definition of `droppableUTIs` to return different non-empty UTI arrays based on the type of data being displayed:

p9/Do It/TodoList.swift
```
private var droppableUTIs: [String] {
    switch listData {
    case .list: return [rawTextUTI, todoItemUUIDUTI, todoItemUTI, jsonUTI]
    case .group(.all): return [rawTextUTI, todoItemUTI, jsonUTI]
    default: return []
    }
}
```

Now scroll further down and locate a method within the "Model Manipulation" extension named handleDrop(at:providers:). This is where you'll handle any data dropped onto the list. At present it only contains some verification of the data being displayed, and the beginnings of an invocation of ItemReceiver.readFromProviders(_:completion:). Some basic error handling is there, but the rest needs needs to be implemented.

While the ItemReceiver class handles the specifics of dealing with NSItemProvider, it's still necessary to look at the returned values. Remember that multiple items can be included in a single drag operation, so the system provides one NSItemProvider instance for each item being dropped. ItemReceiver manages all of these asynchronously and hands out an array of ItemReceiver.Output instances containing the decoded data. This is an enum type, defined in Model/ItemProviders.swift:

```
enum Output {
    case item(TodoItem)
    case list(TodoItemList, [TodoItem])
    case existingItem(TodoItem)
    case existingList(TodoItemList)
    case string(String)
}
```

There is one enum entry here for each of the UTI's defined at the top of the file.

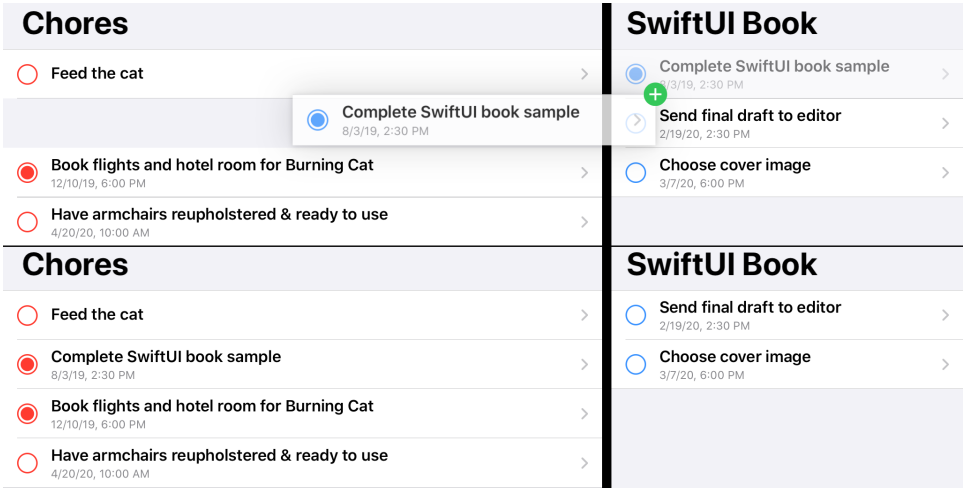
To handle these values, return to TodoList.swift and the handleDrop() method, and add a for loop to iterate over the dropped items. Start by handling just the existing-item case:

p9/Do It/TodoList.swift
```
for value in output {
    switch value {
    case .existingItem(let item):
        if item.listID == list.id {
            // just move it up or down
            let items = self.data.items(in: list)
            if let curIndex = items.firstIndex(where: {$0.id == item.id}) {
                let indices = IndexSet(integer: curIndex)
```

```
                self.data.moveTodoItems(fromOffsets: indices,
                                        to: index, within: list)
                break
            }
        }
        else {
            self.data.moveTodoItems(withIDs: [item.id],
                                    toList: list,
                                    at: index)
        }
    default:
        self.errorPublisher
            .send(ItemProviderError.unsupportedDataType)
        return
    }
}
```

The majority of the work is being done by the DataCenter here, but there is still a check to make: if the item is being dropped on the same list of which it's currently a member, you should treat it as a simple move operation, in the same manner as the onMove() modifier in this view's body. If not, then the DataCenter handles the business of moving the item from one list to another, updating the affected lists and items appropriately.

You can try this out now: launch the application on an iPad or the iPad Simulator, then open a second instance alongside it. Navigate to a different list in each scene, and drag an item out of one list and into the other. You'll see the dropped item land in the requested position, and if you navigate in each scene to the opposite scene's list you'll see that the changes are reflected on both sides.



The next type of drop you'll need to handle is a brand new item. This is handled slightly differently depending on whether the view is displaying a single list or the "All Items" group: in the former case the item is simply added and then shuffled into place within the list; in the latter the drop index means something different, so it just adds the item to the end of the list and let the data center handle the index internally.

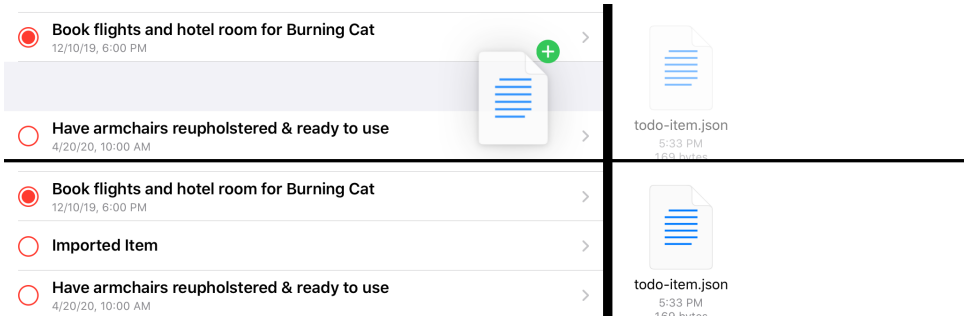
Add the following new case, after the .existingItem case:

p9/Do It/TodoList.swift

```
case .item(let item):
    // insert at end of list
    let items = self.data.items(in: list)
    var newItem = item
    newItem.listID = list.id
    if case .group(.all) = self.listData {
        self.data.addTodoItem(newItem, globalIndex: index)
    }
    else {
        self.data.undoManager?.beginUndoGrouping()
        self.data.addTodoItem(newItem)
        let offsets = IndexSet(integer: items.count)
        self.data.moveTodoItems(fromOffsets: offsets,
                                to: index, within: list)
        self.data.undoManager?.endUndoGrouping()
    }
```

Note the calls to beginUndoGrouping() and endUndoGrouping() on the DataCenter's undoManager property: most undo/redo registration happens automatically, but in this case you want the addTodoItem() and the moveTodoItems() to be a single undo/redo operation. You'll hook into the NSUndoManager later in this chapter.

Testing this is a little harder. There is a JSON file in the code download for this book, at sample-data/todo-item-drop.json. You can load this file into the Files app on your iPad or iPad Simulator and then drag it from there into a list to see it take effect:



The last type to handle is a plain string. A user can create a new to-do item by dragging some text into a list, and the new item will be created using that text as its title. This functions very similarly to the .item case, simply creating a new TodoItem rather than receiving one already made:

p9/Do It/TodoList.swift

```
case .string(let str):
    // new item with this as its title
    let items = self.data.items(in: list)
    let newItem = TodoItem(title: str, priority: .normal,
                           listID: list.id)
    if case .group(.all) = self.listData {
        self.data.addTodoItem(newItem, globalIndex: index)
    }
    else {
        self.data.undoManager?.beginUndoGrouping()
        self.data.addTodoItem(newItem)
        let offsets = IndexSet(integer: items.count)
        self.data.moveTodoItems(fromOffsets: offsets,
                                to: index, within: list)
        self.data.undoManager?.endUndoGrouping()
    }
```

Try this out by launching Safari alongside the application, selecting some text, and dragging it across.

Dropping Lists

Adding drop support to the Home view is very similar. A largely empty implementation of handleDrop(at:providers:) is already present in Home.swift in the starter project, so you need only add case statements for list and string inputs:

p9/Do It/Home.swift

```
case let .list(list, items):
    self.data.undoManager?.beginUndoGrouping()
    self.data.addList(list)
    for item in items {
        self.data.addTodoItem(item)
    }
    self.data.undoManager?.endUndoGrouping()

case let .string(str):
    let list = TodoItemList(name: str, color: .random(),
                            icon: randomIcon())
    self.data.addList(list)
```

Attaching this to the view is likewise a straightforward task; add the following after the onDelete() and onMove() modifiers in the view's body:

p9/Do It/Home.swift
```
ForEach(data.todoLists) { list in
    // « ... »
}
// « onDelete »
// « onMove »
.onInsert(of: [todoListUTI, jsonUTI, rawTextUTI],
          perform: self.handleDrop(at:providers:))
```

Dragging New Scenes

One convenient feature in iPadOS 13 is the ability to create a new scene within your application through drag & drop. For instance, dragging a tab or a URL within Safari onto the edge of the screen will open a second scene containing the item being dragged. If you're looking to move items around by dragging them from one list to another, it would be a lot simpler if there were a single gesture that would open a second list in a new scene; well, that is possible with SwiftUI as well, though much of the work takes place in UIKit and the Scene Delegate. There are some steps to take to have it all nicely integrated, though, as you'll see.

The primary means by which the system knows to make 'open a scene' available as a drag target comes from data attached to the dragged item's NSItemProvider. In this case, contained data or its UTI aren't used—instead, an NSUserActivity instance is used to determine the intent. Any application can define activity identifiers (strings in the familiar reverse-DNS format) that it will handle. If a dragged item with such an activity attached reaches the edge of the screen, iPadOS will instruct the application that responds to this activity type to create a new scene, and the activity itself will be passed into the delegate of that new scene.

The first thing, then, is to declare that your application supports activities, which is done through an entry in the app's Info.plist file. Select the project in Xcode's Project Navigator, then the "Do It" target, and finally the Info tab. Add a new row to the property list named NSUserActivityTypes and make it an Array. Next add two items within that array, and give them values of com.pragprog.swiftui.ShowTodoItem and com.pragprog.swiftui.ShowTodoList. The result should look something like this:

▼ NSUserActivityTypes	Array	(2 items)
Item 0	String	com.pragprog.swiftui.ShowTodoItem
Item 1	String	com.pragprog.swiftui.ShowTodoList

Next, open Model/ItemProviders.swift and look for the NSUserActivity section, near line 100. In here are extensions for the TodoItem and TodoItemList types, each defining their associated activity type along with a property to obtain an NSUserActivity

instance corresponding to a particular item or list. The details are fairly sparse, since all you need for the purpose of opening a new scene is some way to identify which item or list you should show. The only metadata you need alongside the activity's type identifier is the UUID of the item in question:

p9/Do It/Model/ItemProviders.swift

```
var userActivity: NSUserActivity {
    let activity = NSUserActivity(activityType: Self.activityType)
    activity.title = name
    activity.targetContentIdentifier = id.uuidString
    return activity
}
```

To attach these activities to your item providers, open Model/DataCenter.swift and find the "Drag & Drop" extension containing the two itemProvider() methods. You attach a user activity to an item provider using NSItemProvider.registerObject(_:visibility:), which adds a new type of data to those supplied by the item provider. In this case it adds data with a UTI of com.apple.uikit.useractivity. To attach the activity, add something like the following to both itemProvider() methods:

p9/Do It/Model/DataCenter.swift

```
let result = NSItemProvider(object: provider)
result.registerObject(list.userActivity, visibility: .all)
return result
```

The job of handling this activity information falls to your SceneDelegate. When a scene session is connected, your delegate will be provided with a set of UIScene.ConnectionOptions, and from there you can determine whether you've been invoked in response to a user activity.

Open SceneDelegate.swift and look at the top function: scene(_:willConnectTo:options:). The starter project has factored out the creation and assignment of the UIHostingController into a new method, presentView(_:in:), and you'll use this to put your chosen SwiftUI view on the screen.

Start by looking at any user activities in the provided connection options; you'll use only the first handled instance. Add this code just above the call to presentView(Home(), in: scene):

p9/Do It/SceneDelegate.swift

```
for activity in connectionOptions.userActivities {
    guard let contentID = activity.targetContentIdentifier else {
        continue
    }

    switch activity.activityType {
    default:
```

```
            break
        }
    }
```

You skip anything without a targetContentIdentifier, since you need that to locate any referenced list or item. Next you look at the activity type, and skip any that you don't recognize; at the moment, that's everything. Add a case for handling TodoItemList.activityType:

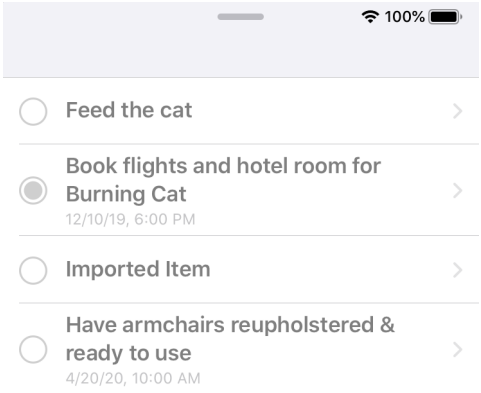
p9/Do It/SceneDelegate.swift

```
case TodoItemList.activityType:
    guard
        let uuid = UUID(uuidString: contentID),
        let list = sharedDataCenter.list(withID: uuid)
        else { break }
    let view = TodoList(list: list)
    presentView(view, in: scene)
    return
```

Here you decode the UUID from the activity's targetContentIdentifier, then locate the corresponding TodoItemList. If that all works, then you simply present that list in a new TodoList view. Fairly straightforward.

Let's test it out. Launch the app on an iPad or iPad simulator, then drag a list out towards the edge of the screen, and when the system reveals a new scene area as a drop target, let go. A new scene will appear showing your list—it worked

Well, almost:



The entire UI is disabled, and there's no title—actually, there's no navigation bar at all. In fact, that's why the list is disabled: the NavigationLink views only function within a NavigationView, and you're only displaying a TodoList, which

doesn't contain a navigation view. You'll need to add one. This could be as simple as wrapping one in the code you used above:

```
let view = NavigationView { TodoList(list: list) }
presentView(view, in: scene)
```

Try that in portrait orientation and you'll likely see something odd, though: the iPad shows the (currently empty) detail view and slides the top-level view off the scene's leading edge. Instead, you want to always use a stack format similar to the iPhone, so add a call to .navigationViewStyle() to fix that:

```
let view = NavigationView {
    TodoList(list: list)
}
.navigationViewStyle(StackNavigationViewStyle())
presentView(view, in: scene)
```

However, while you're here, let's add something else. Apple suggests that secondary scenes provide some way for the user to easily dismiss them; let's add that capability by adding a "Done" button to the leading edge of the new scene's navigation bar. The means to dismiss a scene is through the shared UIApplication instance's requestSceneSessionDestruction() method, which requires a reference to the UISceneSession being dismissed and optionally a handler for any errors that occur.

That gives you something like this:

```
let button = Button(action: {
    UIApplication.shared
        .requestSceneSessionDestruction(session, options: nil,
                                        errorHandler: nil)
}) {
    Text("Done")
        .bold()
}
let view = NavigationView {
    TodoList(list: list)
        .navigationBarItems(leading: button)
}
.navigationViewStyle(StackNavigationViewStyle())
```

Now think about what your code will look like for the other activity you need to handle: everything except TodoList(list: list) is going to be identical. So, that's 12 lines of boilerplate in 13 lines of code. This is ripe for factoring, so that's what you'll do, by creating a ViewModifier to handle the details.

Open AccessoryViews/ViewModifiers.swift, and scroll down to the View extension. Just above this, create a new ViewModifier type named SceneSessionDismissal, with properties for the UISceneSession and optional error-handler:

p9/Do It/AccessoryViews/ViewModifiers.swift
```
struct SceneSessionDismissal: ViewModifier {
    private let session: UISceneSession
    private let errorHandler: ((Error) -> Void)?

    init(_ session: UISceneSession, errorHandler: ((Error) -> Void)? = nil) {
        self.session = session
        self.errorHandler = errorHandler
    }

    func body(content: Content) -> some View {
        // « ... »
    }
}
```

The content of the body(content:) method should be familiar:

p9/Do It/AccessoryViews/ViewModifiers.swift
```
func body(content: Content) -> some View {
    let button = Button(action: {
        UIApplication.shared
            .requestSceneSessionDestruction(self.session, options: nil,
                                            errorHandler: self.errorHandler)
    }) {
        Text("Done")
            .bold()
    }

    return NavigationView {
        content.navigationBarItems(leading: button)
    }
    .navigationViewStyle(StackNavigationViewStyle())
}
```

For conciseness, add a new function to the bottom of the View extension here to wrap the modifier into a single call:

p9/Do It/AccessoryViews/ViewModifiers.swift
```
extension View {
    func dismissingSceneSession(
        _ session: UISceneSession,
        errorHandler: ((Error) -> Void)? = nil
    ) -> some View {
        modifier(SceneSessionDismissal(session, errorHandler: errorHandler))
    }
}
```

With this, your activity handler in SceneDelegate.swift can be updated with just one extra line:

p9/Do It/SceneDelegate.swift
```
let view = TodoList(list: list)
    .dismissingSceneSession(session)
presentView(view, in: scene)
```

The case for handling a single item is very similar:

p9/Do It/SceneDelegate.swift
```
case TodoItem.activityType:
    guard
        let uuid = UUID(uuidString: contentID),
        let item = sharedDataCenter.item(withID: uuid)
        else { break }
    let view = TodoItemDetail(item: item)
        .dismissingSceneSession(session)
    presentView(view, in: scene)
    return
```

Rebuild your application and drag out new scenes from lists and items. They should all appear and function correctly, and the "Done" *Ed: Just a reminder about using the keyword tag for UI elements.* button in their navigation bars should close the scene correctly.

What You Learned

This chapter has covered a lot of ground, and you've now used all of the basic drag and drop APIs provided by SwiftUI.

- Implementation of drop support for insertion into list views, each defining the specific types of data it will accept.
- Exporting and importing data in several different formats.
- Dragging and dropping items within the same application across scenes and views to make gestural modifications to your data model.
- Using drag and drop gestures to spawn new scenes displaying a subset of your app's data.

In the last chapter, we're going to look at how you might integrate your SwiftUI application with a Core Data model, such as you might encounter in a more complicated application. You'll learn the tools available from SwiftUI, build some of your own, and learn what you'll need to do differently when dealing with referential model types rather than value types.

CHAPTER 9

Core Data and Combine

Story Map

Why do I want to read this?

Your application has a very simple data model right now, but that's fairly unusual. Data is normally more complex and is frequently changing.

What will I learn?

You'll take a Core Data model, which presents a much-expanded view of your application's data, and you'll learn how to tie that into your SwiftUI views. You'll learn how to use Combine to allow your views to respond instantly to changes in your data, and to safely manage network data requests.

What will I be able to do that I couldn't do before?

You'll be able to build a SwiftUI application that uses Core Data as its backing store. You'll be able to use the facilities of the Combine framework to quickly manage data flow between the components of your application.

Where are we going next, and how does this fit in?

You're done with the book. You now know how to use SwiftUI to create well-behaved modern applications, and have a thorough grounding in the available facilities of the framework. Congratulations!

Up to this point, you've used Swift value types to represent your data models, and have used classes only sparingly. Using strictly immutable copies of value types in this way has many benefits in terms of thread safety, and can make it easier to reason about your data model, but as that model grows—particularly gaining relationships between model items—the burden of maintaining the model's internal consistency increases. Already you have helper methods in `DataCenter` to locate related lists or items, special functions to update edited copies of model data in the shared data store, and just in

the last chapter you used publishers to monitor the shared store in order to forcibly update your copies of the data should it change. The data model is only going to grow from here on, so it's worth looking at alternative ways to manage it. One way that already has support for SwiftUI is to use *Core Data*.

Core Data is Apple's framework for *object graph persistence*. Specifically, it takes on two primary roles: it maintains a graph of interrelated objects, ensuring the graph remains consistent and valid over time; and it provides serialization for that object graph in several formats. It's often thought of as a "database API," but it's really not; though one of the storage formats uses *SQLite* (pronounced *ESS-queue-ell-ite*, as if it were the name of a mineral), the vast majority of its code deals with the object graph in memory, and database support simply provides an expedient way to load and save parts of the model independently.

In this chapter, you'll be working almost entirely with the data layer of your application. The implementation of user interface features will take a back seat while you update your application to use a model implemented entirely in terms of Core Data. Much of the fine detail of Core Data itself has been provided for you in the starter project for this chapter, since it would be somewhat outside the bounds of a book on SwiftUI. However, if you'd like to learn more about Core Data, I'd heartily recommend *Core Data in Swift [Zar16]*.

The starter project for this chapter can by found in the downloadable code archive for this book[1] in the `p7-starter` folder. There are quite a few changes in here—too many to list the altered files as I've done before. Instead, here's a brief overview of what's been modified:

- First and foremost, there is now a Core Data model in `Resources/TodoItems.xcdatamodeld`.
- Files relating to the existing `struct`-based data model have been moved to `Model/OldModel`, and the type names have been changed to `TodoItemStruct` and `TodoItemListStruct`. All the UI code has been updated to refer to these new type names.
- The `Priority` type has been lifted out of `TodoItemStruct`, and `TodoItemList.Color` is now `ListColor`.
- `AppDelegate.swift` contains the necessary code to load the Core Data model (essentially the same as that provided by an Xcode template).
- Several helper routines for the Core Data models have been provided, to cut down on the code you'll need to write.

1. https://pragprog.com/titles/jdswiftui/source_code

- Two new property wrappers have been provided, @DelayedMutable and @DelayedImmutable.
- A preview-specific class, PreviewDataStore, has been added in Preview Content/PreviewDataStore.swift. This manages a CoreData stack purely for the preview system, recreating its data each time it launches.

I strongly recommend that you use the starter project while working on this chapter; if not, I suggest you use a diff tool such as Kaleidoscope[2] to determine what you'll need to add to your current project to bring it to parity before continuing.

You'll take an iterative approach while converting the application, going piece-by-piece from the smallest leaf views towards the top-level containers, getting each component working in the canvas before integrating it into its parent view.

To start with, let's look at the big picture for a moment, and see how a Core Data model is brought into the world of SwiftUI.

Integrating a Core Data Model

Open SceneDelegate.swift and locate the line where your Home view is created, then make these changes:

p7/Do It/SceneDelegate.swift

```
➤ let context = UIApplication.shared.persistentContainer.viewContext
  let contentView = Home()
➤     .environment(\.managedObjectContext, context)
```

Here you've used some convenience accessors in AppDelegate.swift to fetch a property from your AppDelegate named persistentContainer. This is an NSPersistentContainer instance which manages the Core Data "stack" on your behalf—the model description, the on-disk data store, and the in-memory data context. From this you're fetching the the *managed object context* designated for the use of the user interface, the viewContext. A managed object context provides the means of interacting with an object graph and its accompanying persistent store; and since Core Data takes a very serious attitude towards thread safety, the use of a single context used exclusively for driving the user interface (and thus synchronized to the main thread) is a particular concern. By separating out the UI's context like this, everything that happens to the context observed by the interface is guaranteed to take place on the main thread. This has two main benefits:

2. https://www.kaleidoscopeapp.com

- Any UI updates triggered by changes in the data store are guaranteed to happen on the main thread, where the UI expects them to happen.
- Any work performed by the UI's object context is likewise guaranteed to happen on the main thread, meaning it will be properly synchronized with the UI framework's ability to update the interface to match—the model can't change while it's being drawn.

This view context is passed into SwiftUI through the environment; specifically, the viewContext is assigned to the environment's managedObjectContext property. Other views and even property wrappers in SwiftUI will look for it here.

Note that you are no longer passing the DataCenter into the environment: this component is going to be replaced in every view, and by removing it from here you'll find out very quickly if a view still tries to access it—because your app will quit!

Core Data Model Objects

At the most basic level, there's not a lot of difference between a Core Data object and a value type, from SwiftUI's point of view. Both contain properties, and both can be used as a form of state information. For structures, you've used the @State property wrapper. For Core Data objects, which are class types, you use @ObservedObject instead. The ObservableObject protocol conformance is already provided by NSManagedObject, the base type in Core Data from which all model objects descend. Each property defined in the model thus has a Publisher available, and any changes made to those properties will cause SwiftUI to re-evaluate the views that use it.

The first place you'll see this is when updating the to-do item rows. Open TodoItemRow.swift and change its content:

p7/Do It/TodoItemRow.swift

```
Line 1  @ObservedObject var item: TodoItem
-
-   var body: some View {
-       HStack {
5           Button(action: {
-               self.item.complete.toggle()
-               UIApplication.shared.saveContext()
-           }) {
-               // « ... »
10          }
-           .padding(.trailing, 6)
-           .buttonStyle(HighPriorityButtonStyle())
-
-           VStack(alignment: .leading) {
15              Text(item.title ?? "")
```

```
                .font(.headline)
                .foregroundColor(.primary)
                .padding(.bottom, 2)
            // « ... »
        }
    }
}
```

On line 1 you now have all the state your row needs—a reference to the model object representing this to-do item. The DataCenter is gone now, as Core Data will handle the collection of model objects and their storage and source of truth for you. Next, the action for the "Complete" button is a lot simpler. On line 7 the lookup-and-swap process is replaced by a single call to save the model data. In fact, the toggle() call above this is all that's necessary to get the model updated, and this second line exists only to write the new values to persistent storage.

Only one other line needs to change in this view. On line 15 you're using Swift's *nil-coalescing* operator when assigning the item's title to the Text view. This is because, unlike the data model used earlier in the book, all Core Data model properties (or *attributes*, to use the correct term) are nullable by default. This applies to any type that would be represented by an object in Objective-C—so strings, dates, data, and so forth would all be nullable. Only the values of *primitive types*—numbers, booleans—are non-optional in Swift. Now, the model definition may explicitly state that the title attribute is required, not optional. That only applies to the logical models used inside of Core Data, though; it means that the object will be invalid unless it has a title. As far as the runtime is concerned, though, that value can legitimately be nil. When you first create a new model object, all its properties will be either nil or some zero/false value. The object will be invalid until you provide concrete values, but at runtime you'll need to handle invalid objects with missing values.

This ultimately leaves you with the burden of deciding what to do with nil attribute values in your UI. Here you're just using an empty string if no title attribute has been set. You'll see plenty of this pattern through the rest of this book, and you'll have some fun when it comes time to create non-optional bindings from these optional values.

Previews

That's it for the item row—almost. The row view is done, nothing else needs to change. However, the preview won't run at the moment since it's still written in terms of the old value-type data model. To update it, you'll use a

custom preview-only data store defined in Preview Content/PreviewDataStore.swift, which provides a small API for you to use in your view previews:

```
class PreviewDataStore {
    static let shared: PreviewDataStore

    let storeCoordinator: NSPersistentStoreCoordinator
    let objectModel: NSManagedObjectModel
    let viewContext: NSManagedObjectContext

    func newBackgroundContext() -> NSManagedObjectContext

    var sampleItem: TodoItem
    var sampleList: TodoItemList
}
```

The store is accessed through a single shared instance, which in turn manages a preview-specific Core Data stack. The persistent data store is deleted and recreated at each startup, so you'll always be working with the same sample data. Update your preview to use this data store and its sample TodoItem:

p7/Do It/TodoItemRow.swift

```
return TodoItemRow(item: PreviewDataStore.shared.sampleItem)
    .padding()
    .previewLayout(.sizeThatFits)
    .environment(\.managedObjectContext, PreviewDataStore.shared.viewContext)
```

Note that the call to .environmentObject() has been replaced by a .environment() call used to pass in the managed object context from PreviewDataStore.

If you try to run the preview now you'll see some compilation errors, though, since the TodoList view is still trying to pass a value-type model object into the view. For now, you can stop this error by removing the reference to TodoItemRow within that class: open TodoList.swift and find the call to initialize the TodoItemRow in that view's body implementation. Replace it with a simple Text view for now:

```
NavigationLink(destination: TodoItemDetail(item: item)) {
    Text(item.title)
        .accentColor(self.color(for: item))
}
```

Now return to TodoItemRow.swift and launch the preview. Your view renders just as it did before, looking no different. That may seem anticlimactic, but considering the changes that have occurred under the hood this is a great outcome!

Binding to Optional Properties

The next leaf component of the application is the TodoItemEditor view, which *should* be just as straightforward as the row view. Things aren't that simple,

unfortunately: TextField, Picker, DatePicker and friends all operate on bindings to state values. Specifically, bindings to String, Date, etc. Looking at the managed object class—by ^⌘-clicking on the title property referenced in TodoItemRow.swift, for instance—and you'll see again that the properties aren't quite the same: String? and Date? types won't work with the simple $-prefix syntax to create viable bindings.

You'll learn a few different ways of dealing with this type of data in this section. The item editor references plenty of optional properties, including some which are *genuinely optional* within the model, and can legitimately be nil.

To start with, open TodoItemEditor.swift and replace its properties and init(item:) method:

p7/Do It/TodoItemEditor.swift

```
@ObservedObject var item: TodoItem
@Environment(\.managedObjectContext) var objectContext
@Environment(\.presentationMode) var presentationMode
```

The showItem property remains—you'll still be using that—but the item property has now been changed from a @Binding to an @ObservedObject, and the DataCenter instance has been replaced by a reference to the managed object context from the environment, using the @Environment property wrapper. Alongside this is a reference to the editor's presentationMode; you'll take this opportunity to move the save/cancel functionality into the item editor in the same fashion you used for ListEditor in Chapter 5, Custom Views and Complex Interactions, on page 101.

Nil as an Empty Value

Below the state properties, update the notesEditor property as follows:

p7/Do It/TodoItemEditor.swift

```
Line 1 var notesEditor: some View {
2     TextView(text: Binding($item.notes, replacingNilWith: ""))
3         .padding(.horizontal)
4         .navigationBarTitle("Notes: \(item.title ?? localizedNewItemTitle)")
5 }
```

There's not a lot changed in the three lines of the property definition, but what has changed is quite meaningful. Firstly, the Binding initializer on line 2 has changed. If you scroll up to the top of this file, you'll see the extension you created in Chapter 3, Modifying Application Data, on page 57; look at the definiteNotes property there and recall its behavior. If the value stored in the model is nil, then an empty string is returned. If an empty string is set as the new value, then nil is stored in the model. The Binding initializer here is defined

in Affordances/OptionalBinding.swift, itself part of an open-source library by the author.[3] The extension provides three new initializers for SwiftUI's Binding property wrapper, all of which wrap some small yet nonzero-sized amount of boilerplate useful when binding to Optional types:

```
init(_ source: Binding<Value?>, _ defaultValue: Value)
init<T>(isNotNil source: Binding<T?>, defaultValue: T) where Value == Bool
init(_ source: Binding<Value?>, replacingNilWith nilValue: Value)
```

The first of these methods provides a version of Binding that enforces a default non-nil value for its target; this is useful for non-optional model properties that may be set to nil when first initialized. The second wraps the question "is my target binding's value nil?" You'll see this in action shortly.

The third is the one you've used when defining the notesEditor. This performs the same operations as the definiteNotes property noted above; its parameters are an underlying binding to a property of some Optional type and a suitable "empty" value. Here you've provided a binding to the notes property—a Binding<String?>—and an "empty" value of "", the empty string. The binding wrapper will silently swap the nil and empty values when accessing the underlying Binding<String?>.

The compiler will likely be complaining about line 4. Here any potential nil value from the item's title property is replaced by a reasonable (and localized) default (see *Non-Optional Optionals* below), but that default hasn't been defined. Scroll up and add the following private value above the definition of the TodoItemEditor type:

p7/Do It/TodoItemEditor.swift

```
fileprivate let localizedNewItemTitle = NSLocalizedString(
    "New Item", comment: "default title for new to-do item")
```

Notice that this uses the NSLocalizedString(_:comment:) method to obtain the localized value as a String rather than a SwiftUI LocalizedStringKey. This is necessary because the title property is an optional String, so any alternative value used with the optional-chaining operator must *also* be a String. With no public API to obtain a string value from a LocalizedStringKey, falling back to NSLocalizedString() is the only option.

Non-Optional Optionals

You might wonder why you're going to some effort to handle a nil value for a todo item's title property. After all, you're able to ensure that the value is never nil when it's

3. https://github.com/AlanQuatermain/AQUI

supplied to an editor or other view. Certainly that can't be the case for anything loaded from the data store, since title is an explicitly non-optional attribute of the data model. It would seem reasonably save to implicitly unwrap the optional value, using item.title!, since you have control over the data in every case.

Unfortunately, Core Data's memory model and SwiftUI's close observance of state property modifications sometimes find themselves at cross purposes. As an example, let's say you're creating a new to-do item, so you have the editor open. You change your mind, and click "Cancel." That button deletes the un-saved item from the object context and dismisses the editor sheet. All clear and good, right?

Alas, no—in between the delete and the dismiss, SwiftUI will detect the change of all the item's properties, and will re-render the editor view. The view will still have a reference to the TodoItem instance, but this will now be a *fault*—Core Data parlance for "model object whose values need to be fetched from the store"—so it will attempt to load its properties. You just deleted it from the store, though, so that carefully never-nil property is going to return just that, causing your implicitly unwrapped optional to crash the application.

Binding with Default Values

The next part of the view implementation to changes is the definition of the body property itself. Inside the existing Form view, update the title field and list picker like so:

p7/Do It/TodoItemEditor.swift

```
TextField("Title", text: Binding($item.title, localizedNewItemTitle))

Picker("List", selection: Binding(
    $item.list, TodoItemList.defaultList(in: self.objectContext))
) {
    ForEach(TodoItemList.allLists(in: self.objectContext)) { list in
        Text(list.name ?? "<unknown>")
    }
}
```

For both of these controls, their bindings are defined using the default-value initializer, init(_ source: Binding<Value?>, _ defaultValue: Value). While neither value *should* ever be nil, the type system doesn't know that, so you provide a non-optional default to be used if that is the case; the default value will be written to the underlying binding, in fact, making this a shorter form of object.property = value; return $object.property. The title field uses the localizedNewItemTitle value you've seen already, while the list picker uses the default list defined in the data model. Both the default list and the collection of all available lists need to be

fetched from the view's object context, so simple functions for these operations have been provided.

The priority picker is unchanged, but the "Has Due Date" Toggle control needs to use another optional-value binding:

p7/Do It/TodoItemEditor.swift

```
Toggle("Has Due Date", isOn: Binding(isNotNil: $item.date, defaultValue: Date()))
    .labelsHidden()
```

This uses the "is not nil" form of binding, which wraps a pair of operations. Firstly, its getter simply returns the result of asking "is $item.date.wrappedValue currently nil?" Its setter, on the other hand, will take a Bool value and parlay that into an appropriate concrete value for $item.date: false will set it to nil, while true will set it to the provided defaultValue—in this case, the current date. This is effectively the same as the hasDueDate property defined at the top of this file, one of whose two uses you've just replaced.

To update the date picker itself, you'll take a similar approach:

p7/Do It/TodoItemEditor.swift

```
➤ if self.item.date != nil {
    Toggle("Include Time", isOn: $item.usesTimeOfDay)
    HStack {
        Spacer()
➤       DatePicker("Due Date", selection: Binding($item.date, Date()),
                   displayedComponents: item.usesTimeOfDay
                       ? [.date, .hourAndMinute]
                       : .date)
            .datePickerStyle(WheelDatePickerStyle())
            .labelsHidden()
        Spacer()
    }
}
```

Here the use of the hasDueDate property is replaced by a simple nil check, while the binding for the picker's selection obtains a default value. The picker itself will only be displayed when the date is explicitly non-nil, but again the Swift type system doesn't know that, so a little work is needed to provide a Binding<Date> rather than a Binding<Date?>. The local showTime state property has been replaced with a binding to a custom property on TodoItem—look in Model/CoreDataHelpers.swift to find its implementation.

Safely Handling Model Updates

Core Data is designed to handle multiple discrete operations on the same data model from a variety of different locations. Data may arrive from the

network, it may be modified, normalized, or created automatically during read/write operations, or it could be updated by direct user interaction. SwiftUI's data-driven interfaces need to interact with all these changes in a safe and synchronized manner, no matter their cause.

Cancellable User Modifications

There are a few different ways to handle cancellable modifications in Core Data. The design of the framework is that every model object is part of a context, and that context loads data from and writes it to a persistent store. Thus any changes you make to a model object won't actually persist until you ask the context to `save()`. Until that point, it's possible to discard any in-memory changes and just refresh the object from the persistent store. If the object is new and has never been saved, you can determine that and just delete it from the store directly. That has some potential issues, though, including but not limited to those described in Non-Optional Optionals, on page 206.

Amongst other things, the reference-type nature of Core Data model objects means that the same instance is being used everywhere. With SwiftUI monitoring these objects for changes and performing potentially extensive view updates as a result, operating directly on the same shared instance can lead to many side-effects and additional work for the framework. It can also make things a little harder to reason about once your model gets complicated.

For example, consider a later version of the application which stores data both locally and on a cloud server; perhaps it uses Core Data's built-in iCloud support to do so. While you're editing some object—you've changed the name and due date, perhaps—a change to something else comes through from the network. There are no conflicts, so the change is imported in the background and passed to the main view context, which is then saved to disk. Now you decide to cancel your changes, so the object is reverted back to its saved form—except the entire context it was just saved by someone else, so now the new title and date are stuck in place.

Core Data provides assistance for this through the concept of hierarchical contexts. A given object context may be connected to a persistent data store, or it may be connected to another context. When this context loads, it just asks its parent context for data. When it saves, it just passes its changes along to its parent. With this facility available, it's become common practice to create a new context to associate with an editor, and constrain all changes to that context alone. Until the editor's context is saved, the changes are limited to that context alone—to discard them, you just don't save them. Once

they are saved, then you tell the parent context to do the same; generally this task is handled by whatever created the editing context.

This is all building to a single point here: this design provides a nice way to modularise save/cancel functionality into your model editor views. They simply operate in terms of the object context fetched from the SwiftUI environment, and they either save the updated object or not. This means that the "Done" and "Cancel" buttons' implementations can now be moved into `TodoItemEditor` directly.

Add the following properties above the editor's `body` implementation:

p7/Do It/TodoItemEditor.swift

```
var cancelButton: some View {
    Button(action: {
        self.presentationMode.wrappedValue.dismiss()
    }) {
        Text("Cancel")
            .foregroundColor(.accentColor)
    }
}

var doneButton: some View {
    Button(action: {
        try? self.objectContext.save()
        self.presentationMode.wrappedValue.dismiss()
    }) {
        Text("Done")
            .bold()
            .foregroundColor(.accentColor)
    }
}
```

On lines 3 and 13 you can see the same call to `PresentationMode.dismiss()` that you used in Chapter 5, Custom Views and Complex Interactions, on page 101. When the user chooses to commit their changes, the editor simply instructs the current context to `save()`, on line 12. If the current context is connected directly to a persistent store, then the data will be written out. If it's connected to another context, then the changes will be sent to that context. The view that presented the editor will know the details, and will take any necessary additional action, such as saving the underlying context.

Previewing

To make the preview work, you'll again need to take two steps. Firstly, update the preview provider to use the sample `TodoItem` and install a managed object context in the environment:

p7/Do It/TodoItemEditor.swift
```
NavigationView {
    TodoItemEditor(item: PreviewDataStore.shared.sampleItem)
}
.environment(\.managedObjectContext, PreviewDataStore.shared.viewContext)
```

Xcode will likely be complaining about lines in two other files, however: the initializer for TodoItemEditor has changed. Open TodoList.swift and locate the editorSheet property definition. At the bottom, replace the content of the NavigationView with an EmptyView. Now open TodoItemDetail.swift and do the same in the editor property there.

Now your preview should launch, and you'll be able to test-drive the interface to ensure everything still works as it did before.

Before moving on, remove the TodoItemStruct extension toward the top of the file—it isn't needed any more.

Using Editor Contexts in SwiftUI

In Cancellable User Modifications, on page 209 you learned about the use of 'editing contexts' to wall off uncommitted changes from the core data store. Setting this up is actually quite easy in SwiftUI, as you'll see: the environment provides a form of dependency injection, meaning that your editor views simply operate in terms of the current environment, allowing the parent to adjust that environment to inject values such as a new managed object context.

You'll put this into action in the item detail view; open TodoItemDetail.swift. Start by updating the state properties as before, deleting the editingItem property while you're there.

p7/Do It/TodoItemDetail.swift
```
@ObservedObject var item: TodoItem
@Environment(\.managedObjectContext) var objectContext

@State var showingEditor = false
```

While you're here, use nil-coalescing operators to fix the errors in headerBackground and body, providing default values for the list color and the item title; delete the TodoItemStruct extension at the top of the file; and find the errors identified by Xcode in TodoList.swift where it uses the old TodoItemDetail initializer, replacing them with an EmptyView. With that out of the way, you can proceed with the important task of defining and handling an editing context.

There are four operations involved in this design:

- Create a new NSManagedObjectContext, setting its parent as the context found in the environment.
- Obtain a new instance of this view's TodoItem from that new context.
- Initialize the editor with the new item, assigning the new context to the editor's environment.
- When the editor is dismissed, check if the current context has changes and save it if so.

Creating the Editor Context

The first three tasks take place in the editor property definition:

p7/Do It/TodoItemDetail.swift

```
Line 1  var editor: some View {
2           let context = self.objectContext.editingContext()
3           guard let editItem = context.realize(self.item) else {
4               preconditionFailure("Failed to get edit version of existing item")
5           }
6           return NavigationView {
7               TodoItemEditor(item: editItem)
8                   .environment(\.managedObjectContext, context)
9           }
10      }
```

On line 2 you create the editing context, using a utility method found in Model/CoreDataHelpers.swift. If you look at the implementation of editingContext() you'll see that it simply creates a new NSManagedObjectContext tied to the main thread (it's going to drive the UI, remember!), and its parent is set to the current context. That relationship will cause the new context's save() method to just propagate any changes to its parent context.

Similarly, line 3 uses another helper method to obtain a copy of the TodoItem, within the editing context; this uses the identifier of the to-do item to load an instance of it in the new context.

The remainder is almost unchanged, with the exception of line 8. This line puts the new editing context into the environment for the editor view. Now when the editor calls objectContext.save() the changes will be propagated to the item property of the detail view.

Saving Changes

The fourth operation listed above requires a change to the .sheet() view modifier in the body property definition; you'll add new parameter, providing a block to run when the sheet is dismissed:

p7/Do It/TodoItemDetail.swift
```
.sheet(isPresented: $showingEditor, onDismiss: {
    if self.item.hasPersistentChangedValues {
        UIApplication.shared.saveContext()
    }
}, content: { self.editor })
```

The only new part here is the onDismiss parameter; this looks at the view's TodoItem and asks if it contains any changes that need to be written to the data store. If it does, then you save the context using a helper method located in AppDelegate.swift. The hasPersistentChangedValues property specifically looks at the actual values that are written to permanent storage, ignoring anything that is calculated dynamically or otherwise transient in nature.

You can now try this out by updating the previews property of TodoItemDetail_Previews at the bottom of this file to use the same parameters and environment values as before:

```
TodoItemDetail(item: PreviewDataStore.shared.sampleItem)
    .environment(\.managedObjectContext, PreviewDataStore.shared.viewContext)
```

Launch a live preview and bring up the editor. Make some changes and cancel or commit them, and observe how the detail view's content changes.

Rinse and Repeat

The TodoListEditor view needs very similar changes to bring it back to working order. Open TodoListEditor.swift and replace its state properties:

p7/Do It/TodoListEditor.swift
```
@Environment(\.presentationMode) private var presentation
@Environment(\.managedObjectContext) private var objectContext
@ObservedObject var list: TodoItemList
```

Update the action for the "Done" button to save its object context:

p7/Do It/TodoListEditor.swift
```
try? self.objectContext.save()
self.presentation.wrappedValue.dismiss()
```

And lastly provide some defaults for the optional icon and name attributes:

p7/Do It/TodoListEditor.swift
```
Image(systemName: list.icon ?? "list.bullet")

TextField("List Title", text: Binding($list.name, "New List"))

IconChooser(selectedIcon: Binding($list.icon, "list.bullet"))
```

Don't forget to update the preview to try it out.

Displaying Lists

Up to now, you've only dealt with one Core Data model object at a time, and that object has always been handed into the view you're working on. To implement the TodoList, Home, and HomeHeader views, however, you need to learn how to obtain and interact with groups of objects. In TodoList you'll have the additional burden of managing sort ordering, so let's start with the more straightforward views first.

Start by opening Home.swift and adding this property near the top of the Home implementation:

```
@FetchRequest<TodoItemList>(
    sortDescriptors: [
        NSSortDescriptor(keyPath: \TodoItemList.manualSortOrder,
                         ascending: true)
    ])
var lists
```

Here you've used a new property wrapper from SwiftUI named @FetchRequest. This provides the primary means by which Core Data objects are brought into a SwiftUI interface, and its wrapped value resolves to a collection of Core Data result types, such as model objects. Each @FetchRequest ultimately wraps a Core Data NSFetchRequest, which itself encapsulates a return type, a sort order, a predicate used to select and filter objects, and more. Internally, @FetchRequest properties monitor the object context installed in the SwiftUI environment to determine when changes occur that affect the objects vended by this property. When that happens, SwiftUI is able to trigger an interface update to present the new data automatically.

Here you're using an attribute syntax you've not seen before. The FetchRequest type implementing the attribute takes initialization parameters itself, so you need to pass those in explicitly as you would for a regular type initializer. The simplest initializer it provides takes an instance of a Core Data NSFetchRequest, but here you only want to specify a type and a sort order, so creating a fetch request manually is a lot of work. Instead you're using another initializer taking only an NSSortDescriptor used to indicate the desired sort ordering, and the type of object you want to obtain is defined by the generic type parameter, <TodoItemList>.

Even this syntax is rather long, though; to fit within the 80-column space for this book it's been spread over five lines. For this reason, let's make use of a convenience initializer from Model/CoreDataHelpers.swift to reduce the amount of typing involved:

p7/Do It/Home.swift
```
@FetchRequest<TodoItemList>(ascending: \.manualSortOrder)
var lists
```

That's better. The model object is specified in the generic type parameter, and with that set Swift only needs the key subpath to build the KeyPath.

Updating List Content

Now it's time to turn your attention to the view's body. The list property works just like a regular Swift collection, so it can be passed directly into the ForEach initializer. You'll also need to provide non-nil values for the list name and icon, as with all Core Data model objects:

p7/Do It/Home.swift
```
ForEach(self.lists) { list in
    NavigationLink(destination: TodoList(list: list)) {
        Row(name: list.name ?? "<Unknown>",
            icon: list.icon ?? "list.bullet",
            color: list.color.uiColor)
    }
}
```

Xcode likely complains about the TodoList() initializer at this point; until you've updated that class as well, just replace it with an EmptyView to keep the compiler happy, and move on.

Deleting

The onDelete() and onMove() modifiers are next on the list. Deletion in Core Data is handled by passing the object to be deleted to its object context's delete() method. Once that's done, you need to call save() on the context to actually delete the object from persistent storage.

To do this, you'll need to have access to the object context; add the familiar environment property to the Home implementation to obtain it:

p7/Do It/Home.swift
```
@Environment(\.managedObjectContext) var objectContext
```

Since you're doing these two operations in a row, it's a good idea to keep them synchronized with one another; ideally you want to *only* delete this object, without saving some other change happening at the same moment. NSManagedObjectContext provides two methods to perform this sort of synchronization: perform(_:) and performAndWait(_:). These function similarly to DispatchQueue's async(_:) and sync(_:) methods—the first will run the supplied block asynchronously and return immediately, while the second will block the calling thread until

the block has finished running. The object context used in the UI is tied to the main (UI) thread, so the work will always execute on that thread, nicely synchronized with respect to any user interface updates.

In this case, you don't need to do anything immediately after the change, so you'll use perform(). Replace the existing .onDelete() implementation with this new version:

p7/Do It/Home.swift

```
.onDelete { offsets in
    self.objectContext.perform {
        for offset in offsets {
            self.objectContext.delete(self.lists[offset])
        }
        try? self.objectContext.save()
    }
}
```

Note that, since all the Core Data operations are synchronized with the UI thread, you can simply reach into the lists property by index, deleting each item. As the delete doesn't actually remove anything from memory until you save the context, neither do you need to iterate through the offsets backwards to ensure the indices remain correct. Similarly, while you're inside the perform() block, SwiftUI can't update the content of the lists property—it will have to wait until you finish iterating and deleting all the indicated objects.

Threading can be difficult at times, but once you get it right it makes things so much easier!

Reordering

Moving and ordering is a little different, however. By default, all collections in Core Data are considered *unordered*. You impose a specific ordering on them through the use of an NSSortDescriptor when fetching them from the data store (in fact, the @FetchRequest wrapper outright *requires* a sort descriptor). This means that you can't just shuffle items around inside a flat array any more; you're going to have to take a different approach.

If you've inspected the object model—or if you raised an inquisitive eyebrow at TodoItemList.manualSortOrder when defining the fetch request earlier—then you likely see how this is going to be handled. In fact, both the TodoItem and TodoItemList objects have integer attributes named manualSortOrder. This enables you to sort them easily based on that attribute's value—precisely what the list property is doing in this view. To change the order of the items, then, you need only change the numbers used to order them.

Let's look at the new .onMove() implementation:

p7/Do It/Home.swift

```
.onMove { offsets, index in
    var newOrder = Array(self.lists)
    newOrder.move(fromOffsets: offsets, toOffset: index)
    self.objectContext.perform {
        for (index, list) in newOrder.enumerated() {
            list.manualSortOrder = Int32(index)
        }
        try? self.objectContext.save()
    }
}
```

Here you've created a copy of the list property in an Array, and used SwiftUI's handy move(fromOffsets:toOffset:) method to rearrange its contents. Next, within another object context perform() block, you enumerate the contents of this array and assign each item a new manualSortOrder equal to its location in the array, thereby providing an ascending order. When the context is saved, the list property will automatically update and the view will be updated to show the lists in their new positions.

Creating New Items

Not much changes when creating new model objects with Core Data. The key point to remember is that all such objects are created *within a particular object context*. To illustrate, consider the TodoItemList.newList(in:) method in Model/CoreDataHelpers.swift:

p7/Do It/Model/CoreDataHelpers.swift

```
static func newList(in context: NSManagedObjectContext) -> TodoItemList {
    let list = TodoItemList(context: context)
    list.name = NSLocalizedString("New List", comment: "Default title for new lists")
    list.icon = "list.bullet"
    list.color = .blue
    list.manualSortOrder = Int32(listCount(in: context))
    return list
}
```

Note that the TodoItemList initializer requires a reference to the object context in which it will live. The remainder of the implementation, though is straightforward: some default values are assigned for all the non-optional (in the model-definition sense, not the language-property sense) properties.

Aside from that, not much has changed from the struct-based implementation; once the new object is created, it can be displayed in an editor. Here again you would create a child object context to place into the editor's environment, creating the new list within that context. You would also add an onDismiss

callback to the .sheet() view modifier to save the local object context when the sheet is dismissed. The result should look something like this:

p7/Do It/Home.swift

```
var body: some View {
    NavigationView {
        // « ... »
    }
    .sheet(isPresented: $showingEditor,
           onDismiss: { try? self.objectContext.save() },
           content: { self.newListEditor })
}

private var newListEditor: some View {
    let context = objectContext.editingContext()
    let list = TodoItemList.newList(in: context)
    return TodoListEditor(list: list)
        .environment(\.managedObjectContext, context)
}
```

Update the preview provider to place PreviewDataStore.shared.viewContext into the environment, and check out the results on the canvas.

Model Validation

The model definition will enforce validity when objects are saved to the persistent store, so a list with no name, for example, will only raise an error during the call to save(). The examples you've seen so far have been generally ignoring these errors for the sake of expediency, using try? and ignoring the result. In practice, this is far from ideal. Imagine the situation: one editor makes an invalid change to one object, so the save fails—but the invalid object remains in the context. From that point on, *every* call to save() that context will fail, because it will include that invalid object.

It is possible to handle these issues more gracefully than is shown in the example code. For instance, when the user clicks "Done" in an editor, the object can be explicitly validated by calling validateForUpdate()(). If the object is invalid, an error will be raised with lots of helpful information attached. That can then be used to present a dialog to the user instead of merely closing the sheet.

Aside from looking at direct validation, it's also good to catch and inspect all errors whenever they occur. Core Data's errors are particularly full of useful information, and will almost always lead you directly to a solution. While expediency (and page count!) necessitates a more cavalier approach in this book, you can find an example of some simple error handling in AppDelegate.swift, in saveContext().

Dynamically Sorting Collections

So far you've worked with individual model objects, both displaying and editing them. All these objects have either been provided as parameters or fetched from the data store using a static request and sort order. SwiftUI provides enough tooling for those operations, but stepping beyond them requires somewhat more vigilance. For instance, how does one change the sort order of a collection loaded through a @FetchRequest property? Well… one doesn't; it isn't possible to reach inside the property wrapper to update its request's parameters. Nor is it possible to keep the underlying NSFetchRequest around to mutate it on demand, because @FetchRequest copies its input rather than retaining it.

To solve this issue, you'll make use of a new @MutableFetchRequest wrapper, located in Affordances/MutableFetchRequest.swift, also a part of the author's open-source toolset[4]. This will provide you the ability to swap out the underlying NSFetchRequest dynamically, in turn driving many of the features of the TodoList view.

First, though, some book-keeping is required. The SortOption type needs to be updated to think in terms of Core Data. Since a fetch request uses an array of NSSortDescriptors to define the ordering of its result, you will need to provide that array, and the most prudent way to do that is to have the SortOption vend it directly. Open TodoList.swift and add the following computed property to SortOption:

p7/Do It/TodoList.swift

```
var sortDescriptors: [NSSortDescriptor] {
    switch self {
    case .title:
        return [NSSortDescriptor(keyPath: \TodoItem.sortingTitle,
                                 ascending: true)]
    case .priority:
        return [NSSortDescriptor(keyPath: \TodoItem.rawPriority,
                                 ascending: false)]
    case .dueDate:
        return [NSSortDescriptor(keyPath: \TodoItem.date,
                                 ascending: true)]
    case .manual:
        return [NSSortDescriptor(keyPath: \TodoItem.manualSortOrder,
                                 ascending: true)]
    }
}
```

4. https://github.com/AlanQuatermain/AQUI

Each sort option now provides a set of sort descriptors, specifying what model value should be compared to create the sort order, and whether higher or lower values should come first. Most use an ascending order—A to Z, past to future—though the priority ordering is the reverse, since the highest priority items should be most prominent.

Within the priority and dueDate cases, however, it's possible there will be duplicate values. More than one item of normal priority, for example, or two items due on the same day. In those cases, there is no clear guarantee of the relative ordering of these items; it all depends on what the underlying storage format happens to use. It would be better to provide a secondary option in those cases, so update the results for .priority and .dueDate to include the sort descriptor from .manual as a second element in the array.

Now look at the ListData type at the top of TodoList; this is still using the old data types, so you'll need to update it. At the same time, add a reference to the managed object context from the environment and declare the @MutableFetchRequest property that will provide your to-do items:

p7/Do It/TodoList.swift

```
Line 1  private enum ListData {
        -     case list(TodoItemList)
        -     case items(LocalizedStringKey, NSFetchRequest<TodoItem>)
        -     case group(TodoItemGroup)
        - }
        -
        - @Environment(\.managedObjectContext) var objectContext
       10
        - @State private var listData: ListData
        - @MutableFetchRequest<TodoItem> var items: MutableFetchedResults<TodoItem>
```

Note that the items property on line 10 isn't initialized. Unlike the @FetchRequest you used in Home.swift, this hasn't been given a fetch request or sort descriptors, and thus is only a declaration, not a definition. To give it a value, you'll need to update your initializers:

p7/Do It/TodoList.swift

```
Line 1  init(list: TodoItemList) {
        -     self._listData = State(wrappedValue: .list(list))
        -     let request = list.requestForAllItems
        -     request.sortDescriptors = SortOption.manual.sortDescriptors
        5     self._items = MutableFetchRequest(fetchRequest: request)
        - }
        -
        - init(title: LocalizedStringKey, fetchRequest: NSFetchRequest<TodoItem>) {
        -     let request = fetchRequest.copy() as! NSFetchRequest<TodoItem>
       10     request.sortDescriptors = SortOption.manual.sortDescriptors
        -     self._listData = State(wrappedValue: .items(title, request))
```

```
-       self._items = MutableFetchRequest(fetchRequest: request)
- }
-
15 init(group: TodoItemGroup) {
-       self._listData = State(wrappedValue: .group(group))
-       let request = group.fetchRequest
-       request.sortDescriptors = SortOption.manual.sortDescriptors
-       self._items = MutableFetchRequest(fetchRequest: request)
20 }
```

Each initializer still corresponds to a single form of ListData, but now these values are also used to generate the fetch request required to initialize the items property (or more properly, the _items property). In every case, the initial sort descriptors are those for SortOption.manual.

The first two initializers are quite direct: the first uses a helper property to obtain a request for all the items in the list; the second copies the input fetch request. The third is a little more involved, obtaining its fetch request from the attached TodoItemGroup. Item groups all contain quite complex sets of items, though: anything with a date; anything with a date in the past; anything due today that hasn't been completed. These are all more complex than the other requests you've used so far, so let's take a look at how they work.

Open Affordances/ItemGroups.swift and scroll to the bottom to find the fetchRequest property. This creates an NSFetchRequest for the TodoItem type, specifies that items should be loaded from storage in batches of up to 25 at a time, and then assigns a *predicate*. Predicates are instances of NSPredicate, which is a base class defining an interface whereby a single object can be inspected for conformance to a set of rules. These might be name == "Henry", identifier IN (22, 23, 29), and so on—to an SQL veteran, they should look vaguely familiar. These are both instances of *comparison predicates*, which compare some property against some value. There are also *compound predicates*, which can require all, some, or none of a set of smaller predicates to evaluate as true. By combining these types, just about any set of criteria can be expressed, including those implied by the TodoItemGroup values cited above.

Just above fetchRequest is the implementation of the fetchPredicate property, which has one case filled out, for .today. This is defined using the predicates API rather than format strings because, well, that's generally a good habit to get into (parsing the format argument is relatively expensive and should be done only when strictly necessary). The .scheduled, .overdue, and .all cases currently return nil, though; let's fix that.

Using Predicates

For the .scheduled case, you need to locate any items that have a date set. Alternatively, by turning that clause around, you arrive at "any items whose date is not nil." That can be easily expressed with a format string:

```
NSPredicate(format: "date != nil")
```

The .overdue case can be handled with a similarly straightforward clause, this time with a parameter:

```
NSPredicate(format: "date < %@", Date() as NSDate)
```

Note that the parameter is explicitly typed as an NSDate instance; trying to pass a Swift Date will raise an error in Xcode.

The .all case is the simplest, it turns out: it doesn't need to change at all, since you genuinely want all the TodoItem instances from the data store.

For extra credit, can you implement these in a similar manner to the .today case, using the API rather than a string? The final project code for this chapter includes an example of the correct way to do this.

The upshot of having your filtering and sorting take place inside the fetch request is that you no longer need to perform any sorting yourself. Return to TodoList.swift and remove the sortedItems variable—you'll replace it shortly—and replace all uses of sortedItems with items, like so:

p7/Do It/TodoList.swift

```
ForEach(items) { item in
    NavigationLink(destination: TodoItemDetail(item: item)) {
        TodoItemRow(item: item)
            .accentColor(self.color(for: item))
    }
}
```

While you're here, you'll note that you can restore the TodoItemRow initializer, since both views now use Core Data model objects.

Dynamically Updating Fetch Requests

To change the sort descriptors in your @MutableFetchRequest property, you'll need to proactively the new sort descriptors and replace them within the property wrapper. Scroll down to the now-empty extension labeled "Sorting" and add a new function:

p7/Do It/TodoList.swift

```
Line 1  private func updateSortDescriptors(_ sortOption: SortOption) {
-       let request = self._items.fetchRequest
```

```
    if case .group(.all) = self.listData, case .manual = sortOption {
        let listOrder = NSSortDescriptor(key: "list.manualSortOrder",
                                         ascending: true)
        request.sortDescriptors = [listOrder] + sortOption.sortDescriptors
    }
    else {
        request.sortDescriptors = sortOption.sortDescriptors
    }
    self._items.fetchRequest = request
}
```

This method performs three tasks:

- On line 2 it obtains the current NSFetchRequest from the _items property (recall that placing an underscore before the property name will let you access the wrapper rather than the content).
- It assigns the new sort descriptors based on the input SortOption. Note that there is an additional sort descriptor added on line 5 if the TodoList is showing all items with a manual sort order: it first orders items by their list's manual ordering, to group them together visually.
- Lastly, the updated NSFetchRequest is placed back into the @MutableFetchRequest property wrapper on line 11, which will cause the list content to change and SwiftUI to re-render the view.

Now that you have this method in place, you need to call it. Locate the .alert() modifier in the body property. Delete the assignment to self.sortBy and replace it with a call to the function you just created.

The remaining steps necessary to adapt the TodoList to use Core Data should all be familiar by this point:

1. Add an onDismiss parameter to the .sheet() modifier to save the object context when the sheet is dismissed.
2. Remove lines referencing any removed properties, such as editingItem.
3. Replace any remaining uses of the old struct-based model types with their new Core Data versions.
4. Handle optional properties within your model objects, either using implicit unwrappind or nil-coalescing operators.
5. Update the removeTodoItems() and moveTodoItems() implementations using the approaches from Deleting, on page 215 and Reordering, on page 216 respectively; the other properties and methods in the "Model Manipulation"
6. Change the implementation of presentEditor(of:) to use an editing context, similar to the implementation used in TodoItemDetail in Creating the Editor Context, on page 212.

7. Update the content of the editorSheet computed property to use an editing object context.

The editor sheet update will use methods from Model/CoreDataHelpers.swift to create a new TodoItem and to locate a suitable list. If this view is displaying a single list, then that would be used; otherwise, the default list is fetched using TodoItemList.defaultList(in:):

p7/Do It/TodoList.swift

```
Line 1  private var editorSheet: some View {
        -     let editContext = objectContext.editingContext()
        -     let editList: TodoItemList
        -     if let list = self.list {
        5         editList = editContext.realize(list)!
        -     }
        -     else {
        -         editList = TodoItemList.defaultList(in: editContext)
        -     }
        10    let editingItem = TodoItem.newTodoItem(in: editList)
        -
        -     return NavigationView {
        -         TodoItemEditor(item: editingItem)
        -             .environment(\.managedObjectContext, editContext)
        15    }
        - }
```

Dynamically Monitoring Metadata

The remaining view has a requirement that's difficult to model with the tools you've used so far. Each HeaderItem view needs to display a counter showing the number of items that match; for example, the number of items due today, or the number overdue. This could be implemented using an @FetchRequest and simply accessing the resulting collection's count property, but that involves a lot more work than is strictly necessary. If you've used SQL, for instance, you know that it's easier to request how many items there are than to fetch something from each one. The same is true in Core Data, and NSManagedObjectContext provides a helper routine for just that purpose: count(for:) takes an NSFetchRequest and returns an Int describing how many objects match, without actually loading any data.

To implement this behavior in a suitably lightweight manner, let's use the Combine framework to create an operation on top of a notification issued by Core Data. Specifically, when an object context is saved, it posts a notification named NSManagedObjectContextDidSave; you can ask the system NotificationCenter to provide a Publisher that vends these notifications when they occur, and then attach further operations to ultimately vend a counter value.

To start, you'll need to add some new state. Open HomeHeader.swift and add import statements for Combine and CoreData to the top of the file. Next add the following properties to the HeaderItem type:

p7/Do It/HomeHeader.swift

```
@Environment(\.managedObjectContext) var objectContext
@State private var countCancellable: AnyCancellable? = nil
```

The second property here is a Combine type that acts as a cancellation and continuation token for a publisher. Its purpose is twofold: firstly, it provides a way to explicitly shut down a publisher by providing a cancel() method. Secondly, its existence will keep the publisher alive until you either cancel it or discard the Cancellable instance (which will cancel on your behalf).

Watching Notifications

To start and stop monitoring the matching item count, you'll need to create a publisher and cancel it, respectively. Add these methods to HeaderItem to implement this:

p7/Do It/HomeHeader.swift

```
Line 1  private func startWatchingCount() {
        guard countCancellable == nil else { return }

        let request = group.fetchRequest
    5   countCancellable = NotificationCenter.default
            .publisher(
                for: .NSManagedObjectContextDidSave,
                object: objectContext)
            .receive(on: RunLoop.main)
   10       .compactMap { $0.object as? NSManagedObjectContext }
            .tryMap { try $0.count(for: request) }
            .replaceError(with: 0)
            .removeDuplicates()
            .assign(to: \.itemCount, on: self)
   15
        if let count = try? objectContext.count(for: request) {
            itemCount = count
        }
    }
   20
    private func stopWatchingCount() {
        countCancellable = nil
    }
```

Here you've obtained an initial publisher using NotificationCenter's publisher(for:object:) method. On line 9 you ensure that any events from the publisher are delivered on the main runloop, to properly synchronize with the user interface. Every following operation will take place on the main thread.

The following operator, .compactMap(), will attempt to fetch the managed object context from the notification. If it fails (i.e. returns nil), then nothing more will happen—the following operations are guaranteed to receive a non-nil NSManagedObjectContext instance. This is then used by the .tryMap() operator to call count(for:).

The publisher at this point might vend either an Int or an Error. To handle—well, guard against—the latter, line 12 uses the replaceError(with:) operator to catch any errors and publish an Int value instead; in this case 0. A removeDuplicates() operator then ensures that events will only be published if they actually vend a different value than their previous output.

Lastly, the assign(to:on:) operator will take that value and assign it to the itemCount state property, thus triggering SwiftUI to perform a view update. With this in place, any changes to the content of the data store will automatically trigger a view update to display the new value. There's one final importand step, however, which is (for your humble author at least) quite easy to overlook: setting the initial value, which happens on line 17. Without this, the counter would read 0 until something somewhere was modified, which certainly led to some confusion while writing this chapter…

Choosing the Right Moment

Ordinarily, SwiftUI will respond to any state variable update at any location in the view hierarchy. For instance, if you drilled down from the home view into a list, then created a new item, then upon saving that item the counter in one or more ItemHeader instances would update. SwiftUI would then redraw them, despite their being offscreen.

In this particular case the property is fairly innocuous—the state property is only used to set the content of a Text view—but other property updates might have unexpected knock-on effects. What would happen if these items were interactive only when their count was non-zero, perhaps by removing their NavigationLink, or by setting a different destination view? If the user tapped on "Overdue" and either deleted everything there or marked them complete, would the current view disappear? Be replaced by a different view? Stay on screen until the user exited? If the latter, would interactions with this view's contents still work?

Where possible, it's useful to think about when it's appropriate to update your state variables and thus trigger a view redraw. Having some parent view change further up the navigation stack in some unexpected manner might lead to some strange behavior or even bugs, so it's useful to know how to

disable these effects. To illustrate this, let's start and stop monitoring the item count changes when the view appears and disappears:

p7/Do It/HomeHeader.swift

```
var body: some View {
    VStack(alignment: .leading) {
        // « ... »
    }
    .padding()
    .background(
        RoundedRectangle(cornerRadius: 15, style: .continuous)
            .fill(Color(.tertiarySystemBackground))
    )
    .onAppear(perform: startWatchingCount)
    .onDisappear(perform: stopWatchingCount)
}
```

With these two lines, you've arranged for the publisher to only be active while the item header is actually visible. As soon as it goes offscreen—for instance when the user selects either a header item or a list—then the publisher will be cancelled. When the view returns, it will be recreated.

Finalization

You're almost done with your conversion now; only a little of the HomeHeader itself remains to be updated. You'll need to change the content of the inner ForEach class to properly define the navigation links. It turns out (whether by design or bug) that SwiftUI doesn't automatically pass on the entire environment to the destination of a NavigationLink. This works for items within the body of a List, but for this header you'll have to do it yourself. This becomes more manageable with a little refactoring.

First, add the following to the HomeHeader definition:

p7/Do It/HomeHeader.swift

```
Line 1 @Environment(\.managedObjectContext) var objectContext
   2
   3 private func linkView(for group: TodoItemGroup) -> some View {
   4     let destination = TodoList(group: group)
   5         .environment(\.managedObjectContext, objectContext)
   6     return NavigationLink(destination: destination) {
   7         HeaderItem(group: group)
   8     }
   9 }
```

That factors out all the necessary work to pass on the object context to the destination of the NavigationLink, leaving only one small change to the view's body:

p7/Do It/HomeHeader.swift

```
var body: some View {
    VStack {
        ForEach(Self.layout, id: \.self) { row in
            HStack(spacing: 12) {
                ForEach(row, id: \.self, content: self.linkView(for:))
            }
        }
    }
}
```

Now take a quick look through the application and ensure that all the navigation links are correctly updated to point to the right views once more, then launch the app and try it out. The experience should match exactly what you had at the end of the previous chapter—conversion successful!

What You Learned

There are many nuances required when dealing with Core Data in SwiftUI. Several times in this chapter you've reached out to new and updated types just to make it all a little more manageable. Now, though, you have experience, and the scars to prove it.

- You can attach a new Core Data model to an existing application.
- You know how to pass Core Data model objects through a SwiftUI view hierarchy.
- Several complex tasks have become simpler at the callsite through the use of facilities like NSSortDescriptor and NSPredicate, saving you the effort of implementing sorting and matching algorithms in multiple places.
- Similarly, implementing NSItemProvider support is now significantly easier.
- Creating bindings to optional types is actually straightforward, now that you know how it's done.
- The @FetchRequest property wrapper can significantly help to manage dynamic collection views.
- When you need a little more dynamism than @FetchRequest provides, you have the tools to go deeper, whether via notifications and publisher data flows, or through more complex tools such as @MutableFetchRequest.

It's been a long journey, but you've now worked with everything SwiftUI has to offer on iOS and iPadOS. Still more awaits in macOS, watchOS, and tvOS, but the broad strokes are the same, and it should all look quite familiar to you at this point.

You're ready to take the next steps on your own, and there's going to be plenty more for you to work with soon enough. Remember that this is only

the first public release of SwiftUI, and it will grow to encompass more possibilities as time goes by. It's all just starting, and now you get to say: you were there.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/jdswiftui

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764