THE BUILDING BLOCKS OF SWIFTUI DEVELOPMENT

SwiftUI Simplified

A Beginner's Journey

Mark Moeykens

A REFERENCE GUIDE FOR SWIFTUI DEVELOPERS

Big Mountain Studio

Version: 9-SEPTEMBER-2024

SCOPE AND PURPOSE

Who is this book for?

This beginner book is for people who:

- Want to build all or part of their app idea themselves.
- Want to explore what it is like to be a developer as a career.
- Are transitioning from other development fields.
- Stopped developing in SwiftUI and would like to come back and get a refresher.
- Want a high overview of SwiftUI to get a better idea if this is something they want to pursue.

Who is this book NOT for?

This beginner book is not for people who:

- Are experienced SwiftUI developers.
- SwiftUI developers looking for advanced topic knowledge.
- Want to learn how to build a complete app.
- Want to qualify for a junior position at a company (this is a great start though).

What is the purpose of this book?

To quickly give you a high-level understanding of SwiftUI and app development.
There are references to other resources to learn more about each topic.

BOOK QUICK LINKS

Book Conventions

Xcode

SwiftUI Concepts

SwiftUI Code

Previews

Views

Architecture

Animations

SwiftData

What is SwiftUI?

Swift

Swift is the name of a programming language used to develop apps to run on Apple products, such as the iPhone, iPads, MacBook, Watch, TV, and Vision Pro.

+

UI

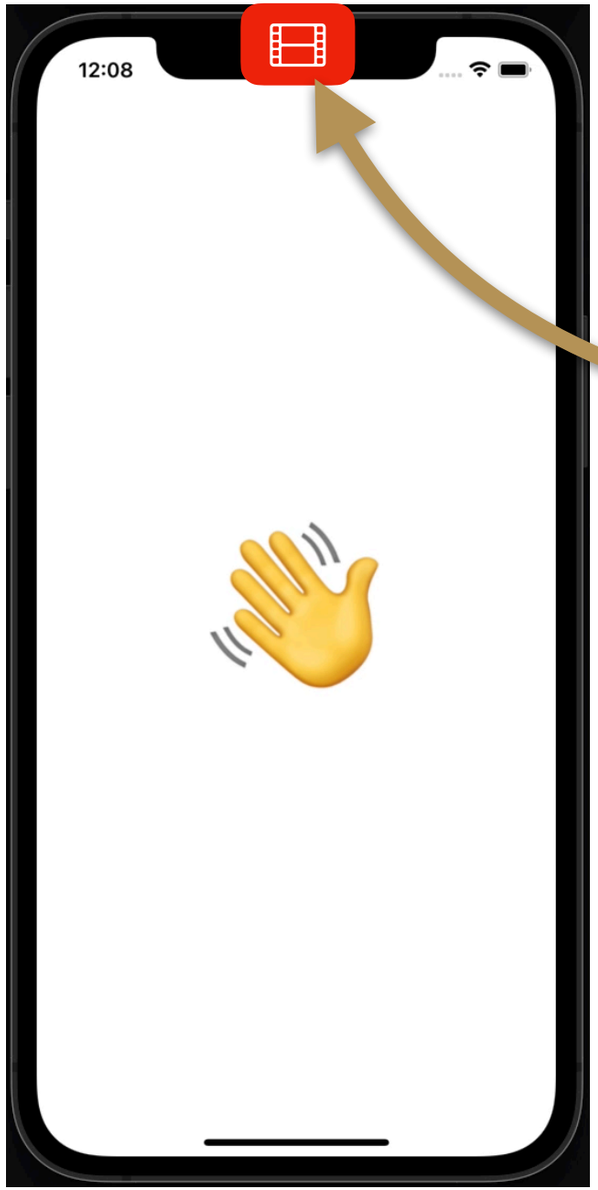
"UI" stands for "**U**ser **I**nterface".

A user interface is what the user sees and interacts with when using a device.

Note: "Swift" is also the name of a family of birds. This is where the Swift logo comes from.

BOOK CONVENTIONS

Embedded Videos



The **ePUB** version of the book supports embedded videos.

The **PDF** version does **not**.

This icon indicates that this is a playable video (only in the **ePUB** format).

But in PDF it renders as simply a screenshot.

XCODE

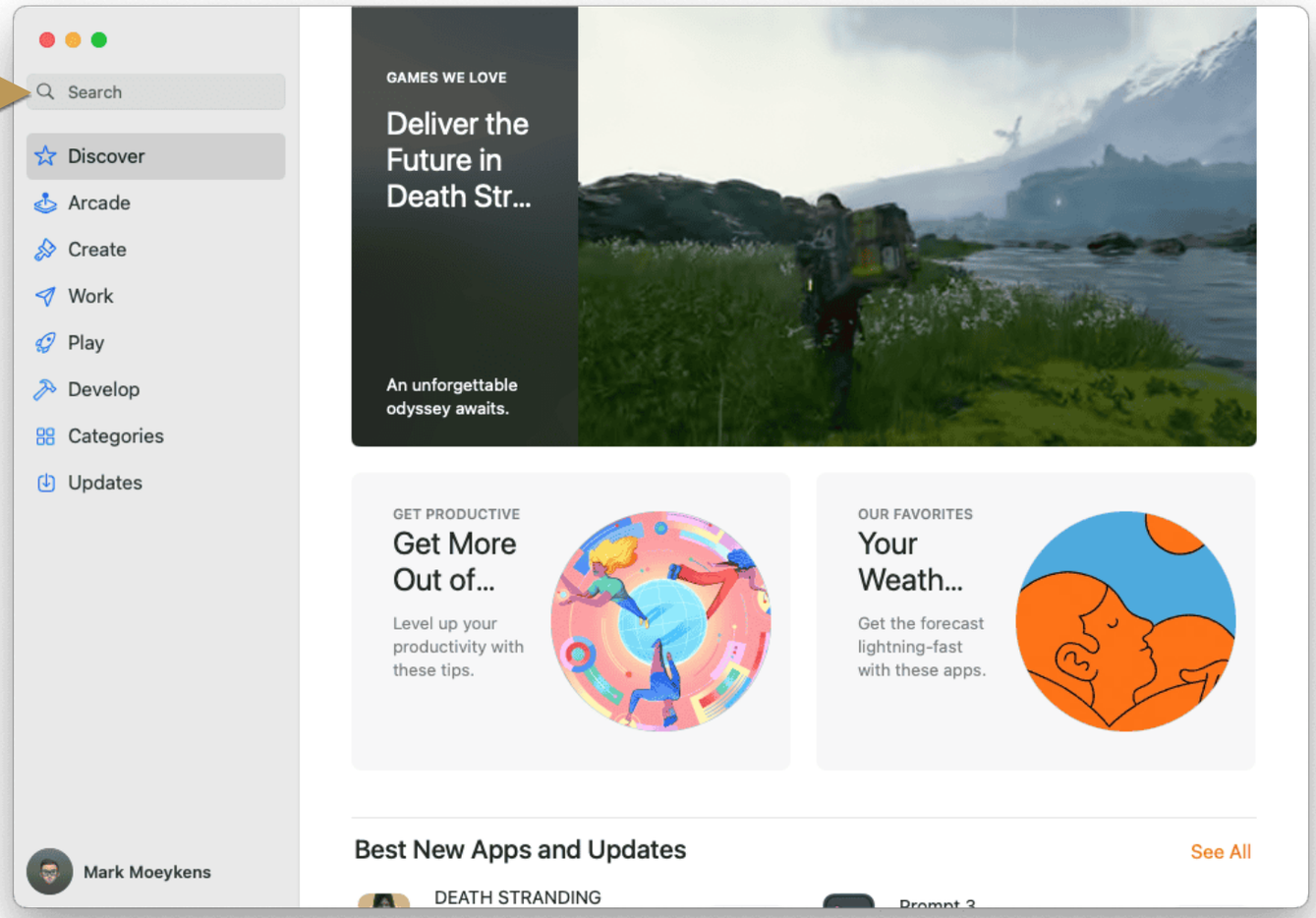
Learn how to install and get around in Xcode - the software development tool you will be using to learn and develop with SwiftUI.

If you already know Xcode, you can skip this chapter.

Note: This is a **VERY SHORT** overview to get you up and started with minimal effort.

Download & Install Xcode

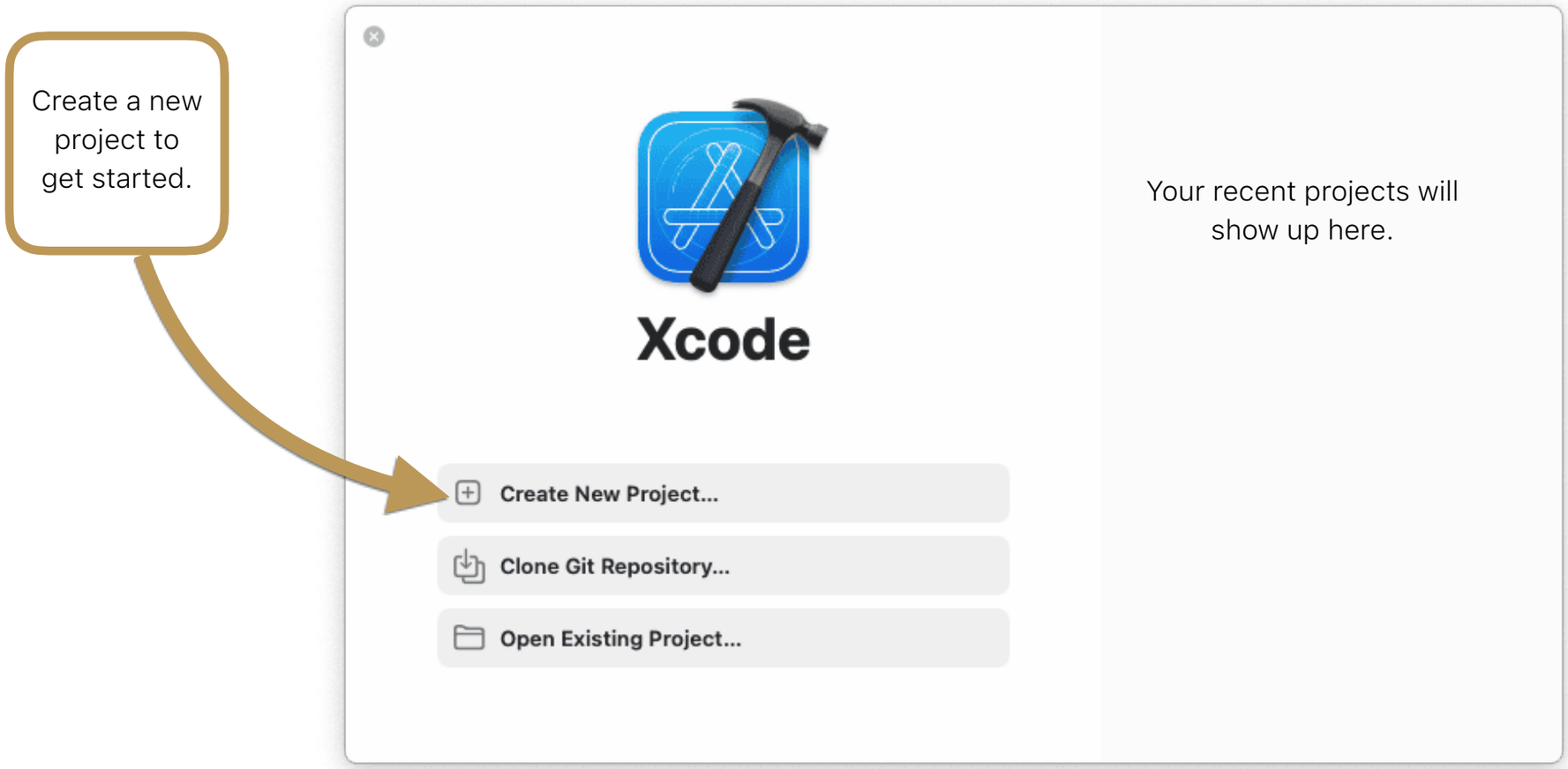
The easiest way to download Xcode is through the App Store.
Start by searching for **Xcode** and then clicking the “Get” button.



?

Do I need to pay for a Developer account?
No, if you are just learning SwiftUI development, you do not need a paid developer account.
You will need to pay when you are ready to publish your app to the App Store though.

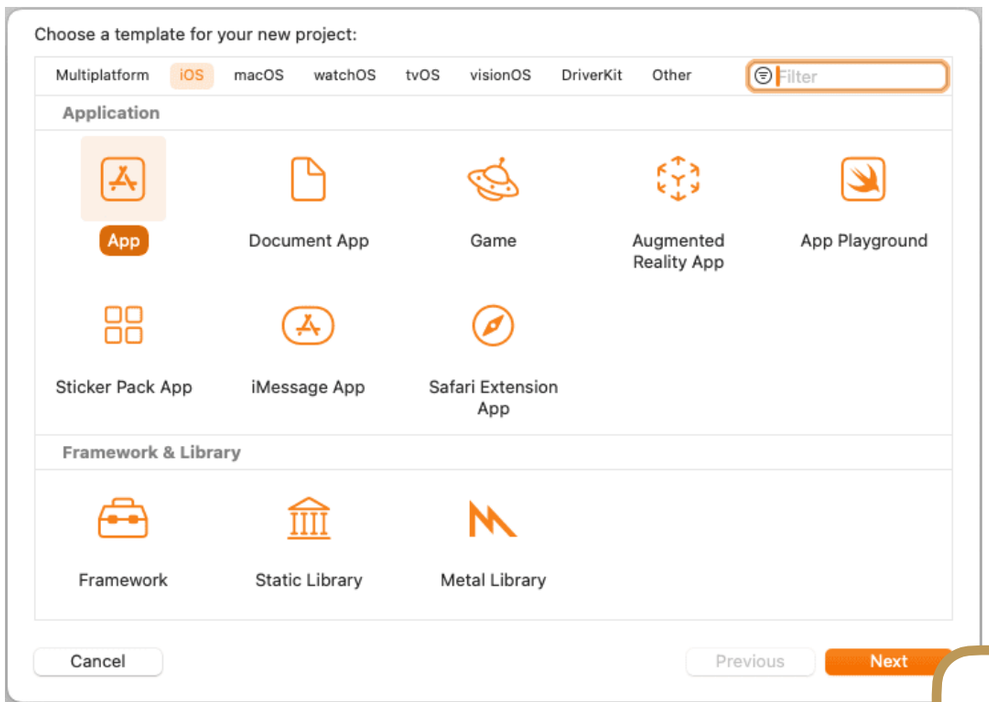
Starting Xcode

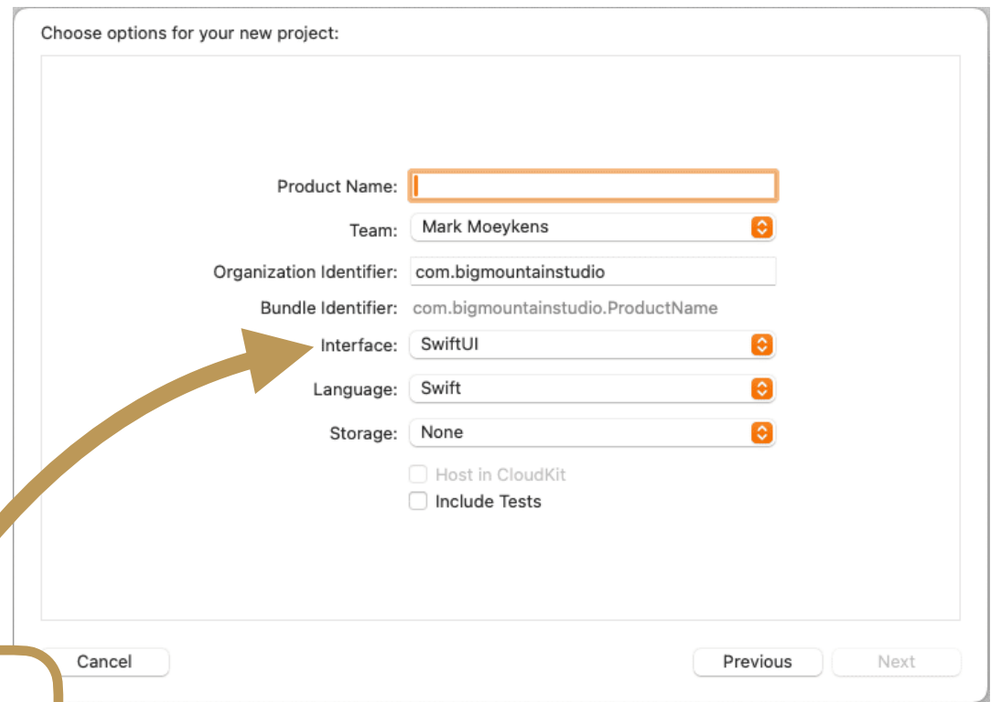


New Project

To follow along with this book and try the code examples, select **iOS** at the top and then **App** and then click **Next**.

Your product name will be your app name. You might not have a team to select and that is OK. Your Organization Identifier is simply a unique id that no one else in the world has (Xcode defaults to reverse URL to keep it unique.)



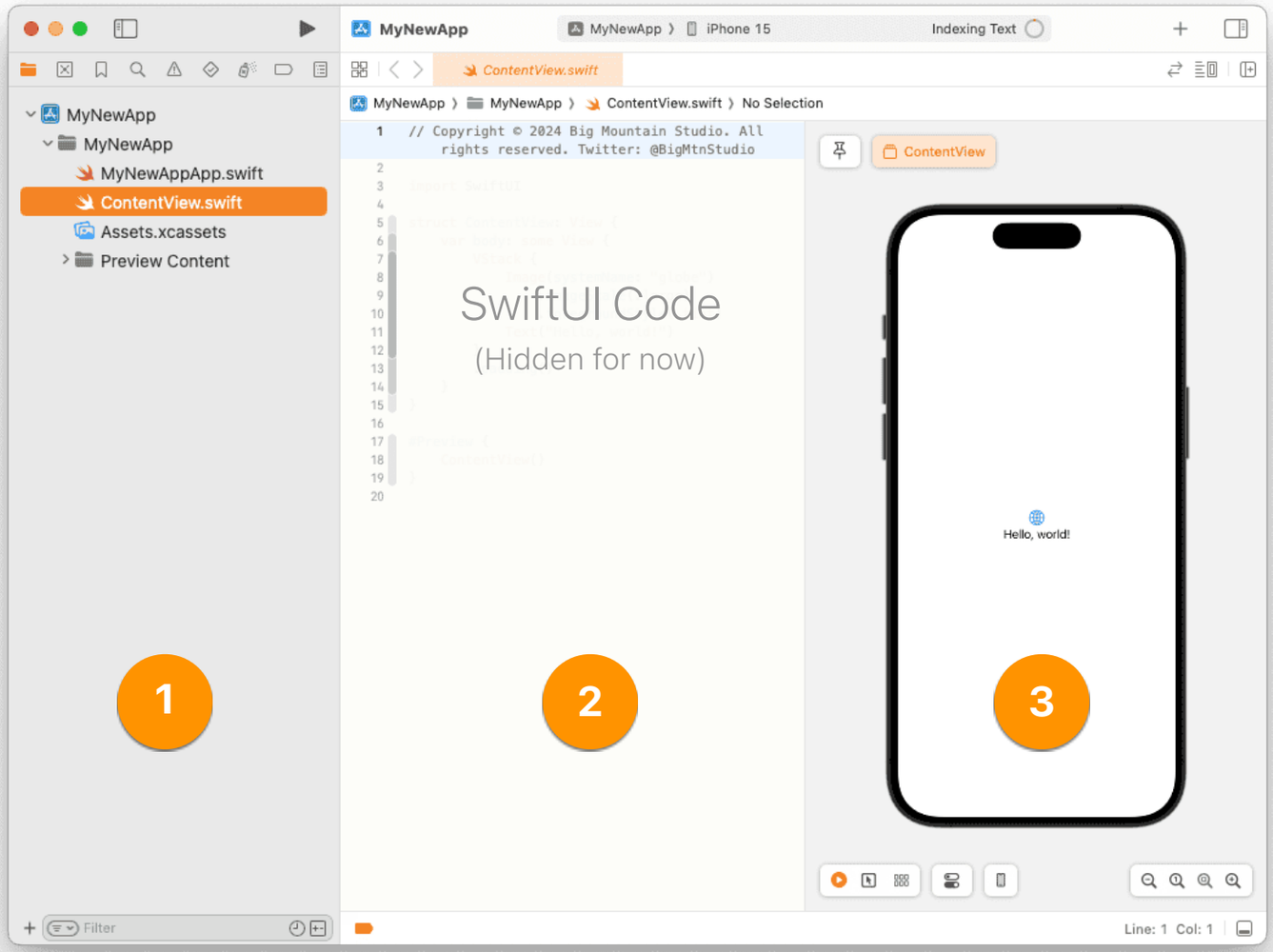


Make sure **SwiftUI** is selected for Interface.

Xcode User Interface

Xcode is divided into 3 areas when it first opens up:

1. **Project Navigator**: This is where your files are organized. You add more files for more screens.
2. **Code Editor**: This is where you will be writing code. In this screenshot, you are looking at code that creates a view using SwiftUI.
3. **Preview or Canvas**: When working on a user interface, this is where the preview of your SwiftUI in the code editor will render. In this example, you are looking at a visual representation of the SwiftUI code.



SWIFTUI CONCEPTS

If you are absolutely **new** to SwiftUI, you should definitely **read through this chapter** to establish some basic concepts that you can think with.

Building Screens with “Views”

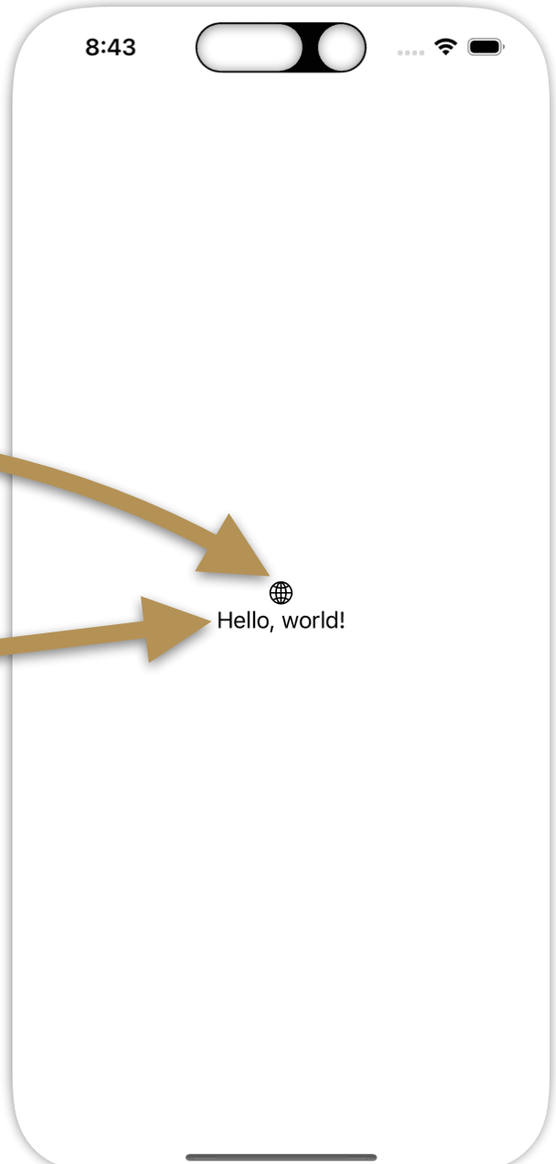
A “**view**” in SwiftUI is a component of the user interface (UI) or screen.

Developers use many views to create the UI and functionality they need for their apps.

Apple’s News App

Views & Modifiers

In SwiftUI, you build a UI with **Views** and then you change those views with **Modifiers**.

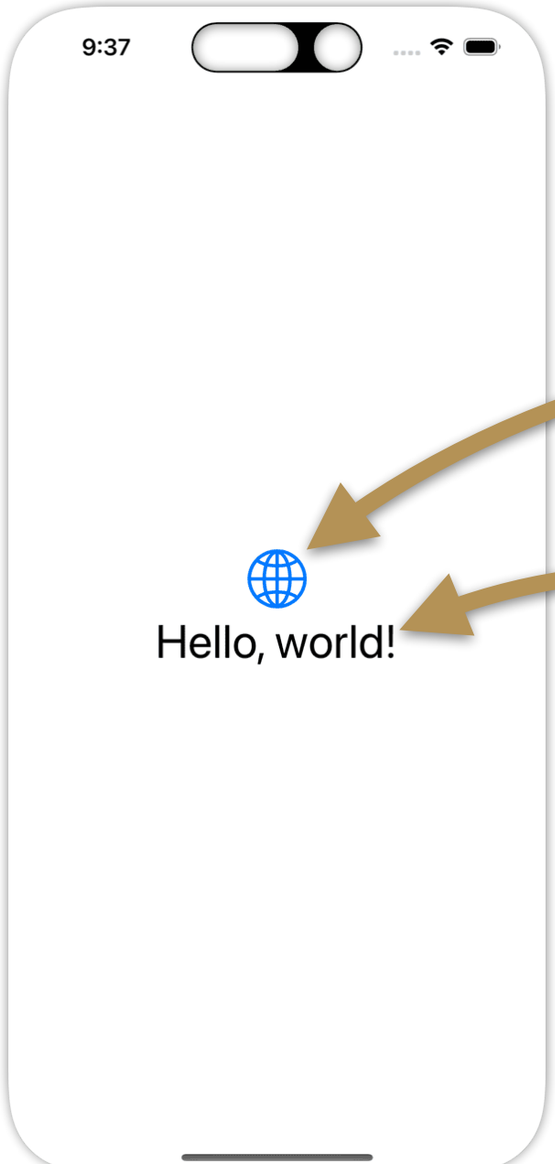


View

No modifiers

View

No modifiers



View

Modifiers:

- Large size
- Blue color

View

Modifiers:

- Larger text size

Note: Most views you add to the screen will appear in the center.

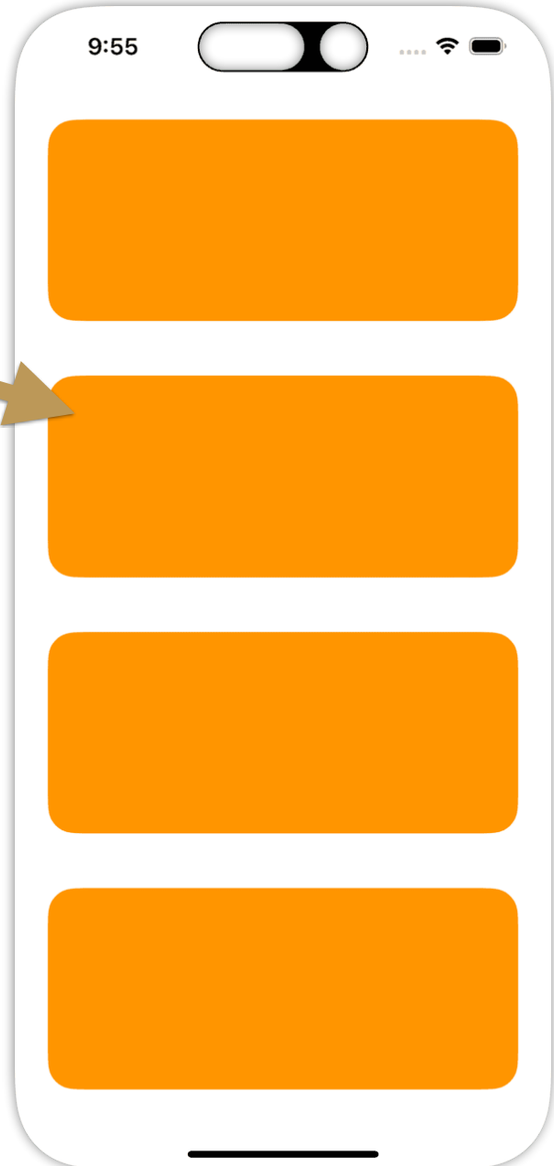
Vertical Layout Container

Views can be organized in containers.

Some containers organize views in one direction.

This is called **Stack**.

Here is an example of a **Vertical Stack** or as SwiftUI calls it, a "**VStack**".



Stacks are views too.

They are views that can have modifiers applied to them.

This VStack contains 4 orange views.

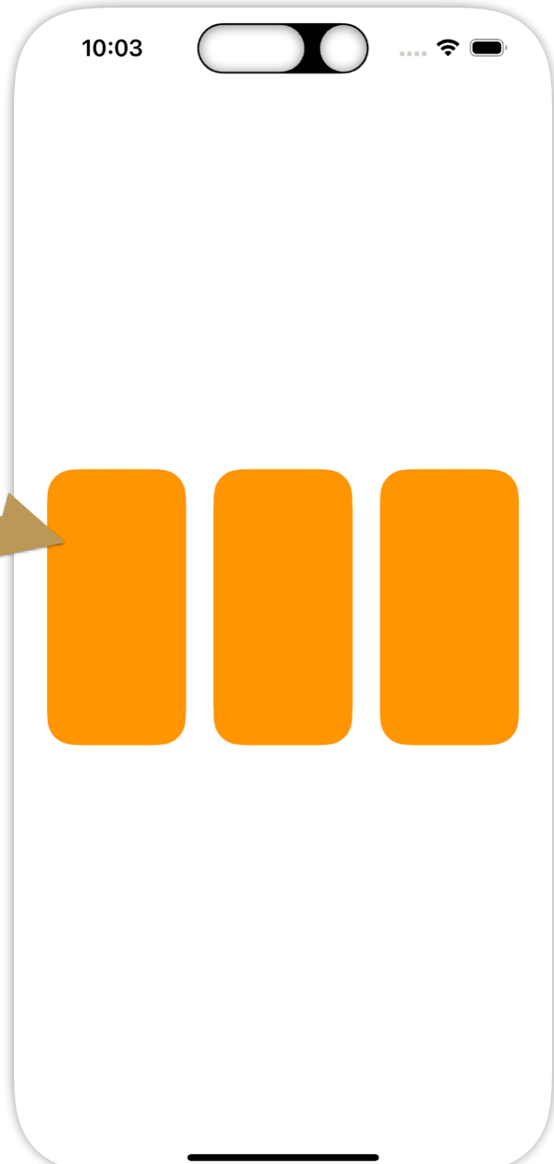
The VStack is modified to have extra spacing or padding around it.

Note: The VStack is THE most used view to create your UI (user interface/screen).

HStack Container

There is another stack that can organize views horizontally.

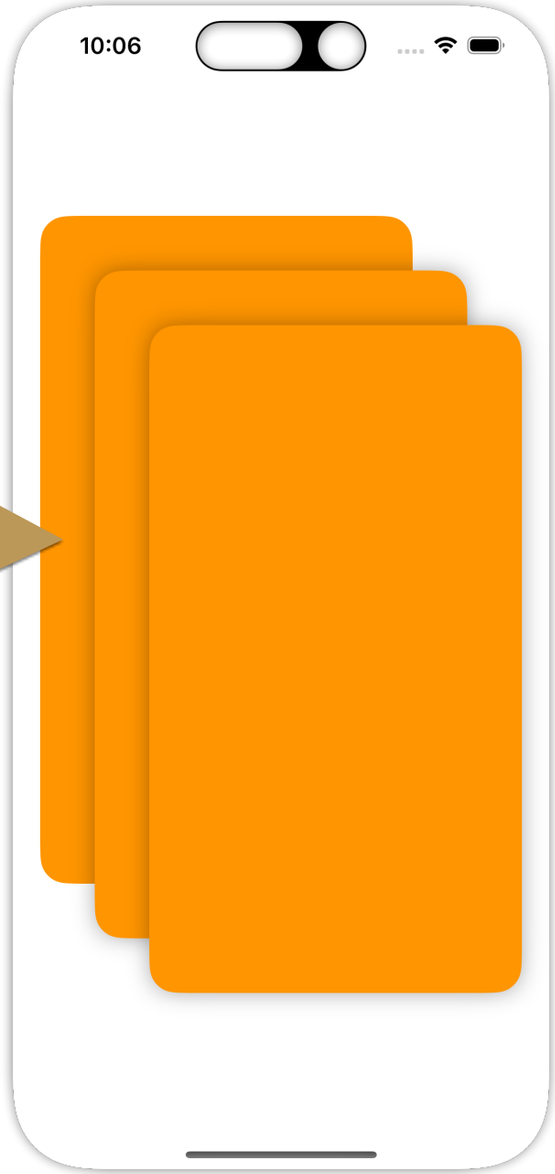
SwiftUI calls this horizontal stack an **HStack**.



Depth Layout Container

This container will organize your views so they are one on top of another.

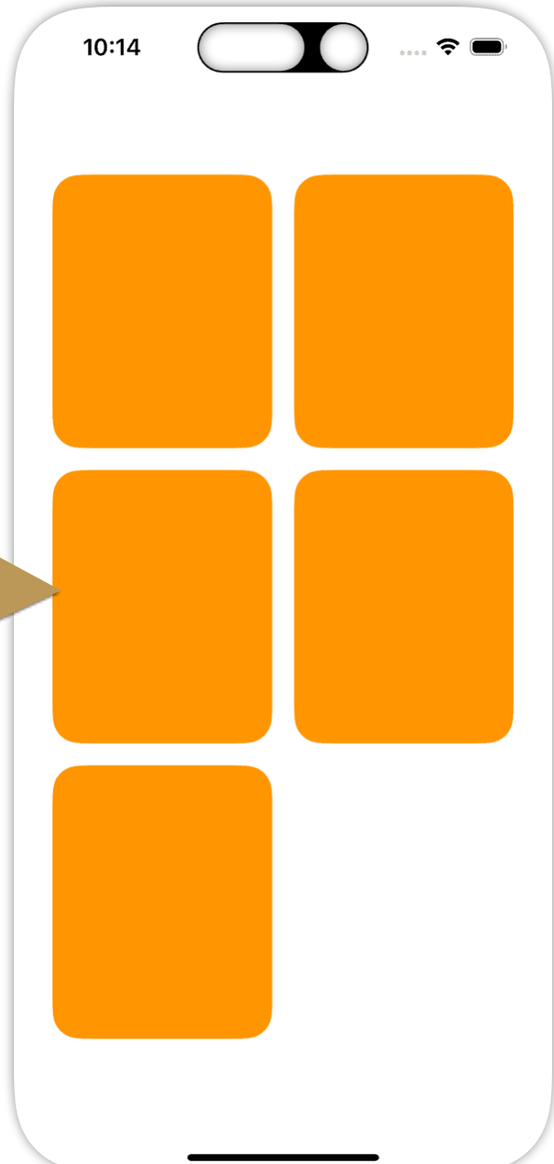
This is called the Depth Stack or **ZStack**.



Grid Layout Container

SwiftUI also offers **grids** to layout views.

There is one for **horizontal** and **vertical** layouts.



Nesting

You can “nest” or put views within views.

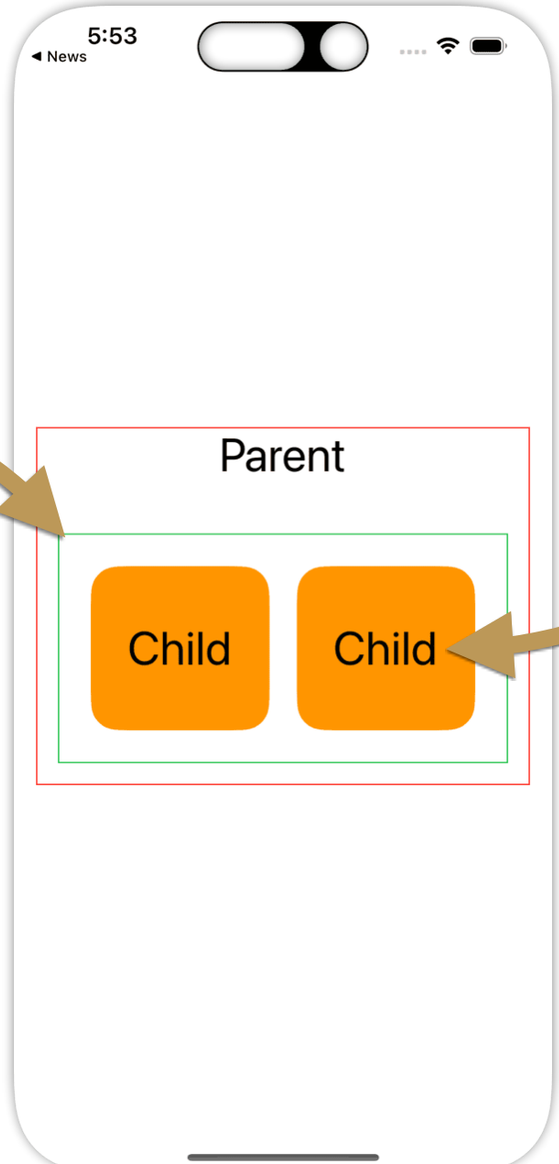
It is common in programming to express a hierarchy of objects as a **parent** and **child** relationship.

In this example, there is a VStack that has an HStack nested within it.

The HStack has two more orange views nested within it.

The view on the outside (VStack) is the parent view (sometimes called the “**root view**”).

All views within the parent view are child views.



Modifiers on Parent Views
Some modifiers can be set on the parent view and it will also apply to all children in the container.

In this example, the font size is set on the parent and all the views use it.

The orange color is also set on the HStack parent view and all the child views within (shapes) use it.

?

How deep can nesting go?
There is no limit. You can continue putting views within views forever.

Layout Example

Now that you know these layout stacks, you can start to guess how views like these might be arranged using SwiftUI.

VStack

Two text views arranged **vertically**.

HStack

A text view and button arranged **horizontally**.

ZStack

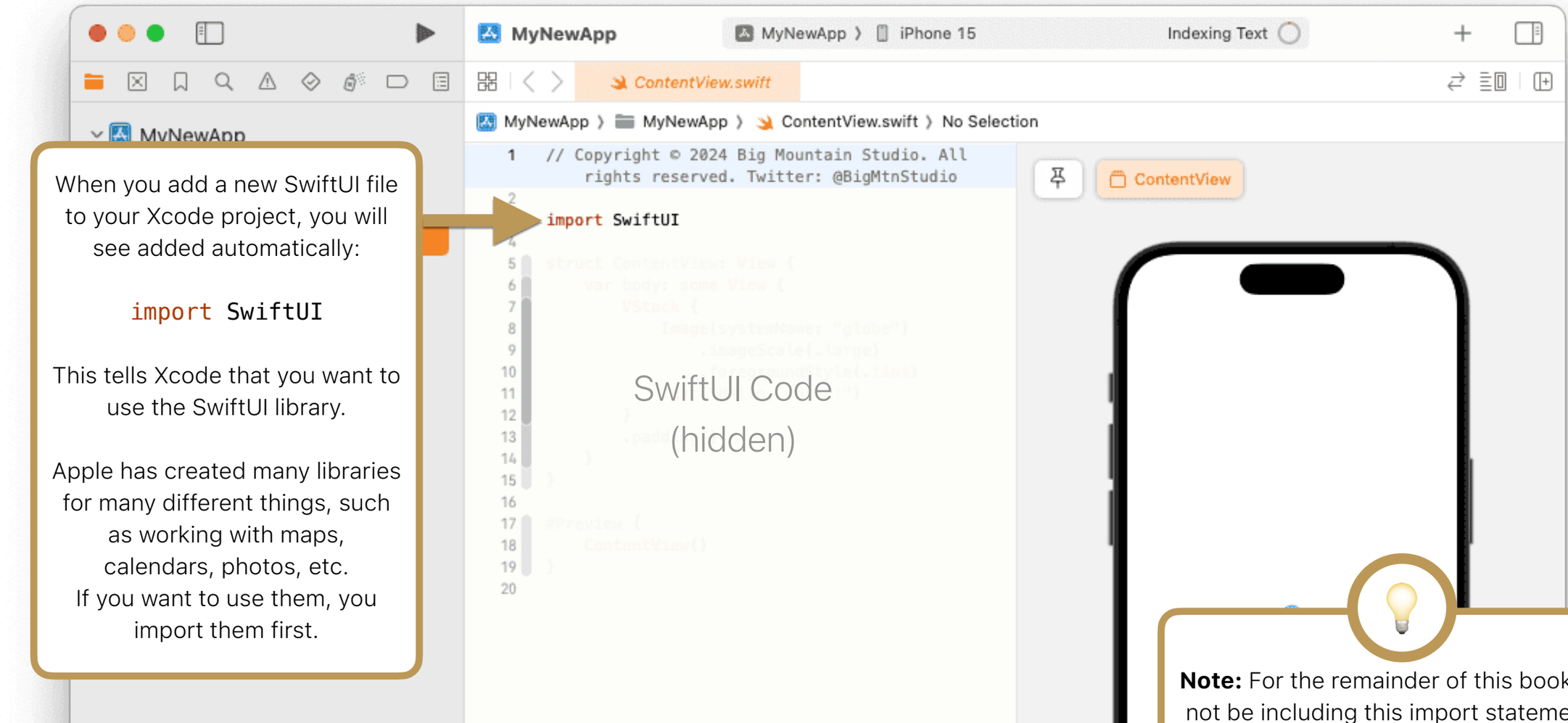
An image on top of two color views.

SWIFTUI CODE

There is one place where all SwiftUI screens start.

You will learn more about this one place as it will serve as your foundation for all SwiftUI work you do in the future.

Import SwiftUI



When you add a new SwiftUI file to your Xcode project, you will see added automatically:

```
import SwiftUI
```

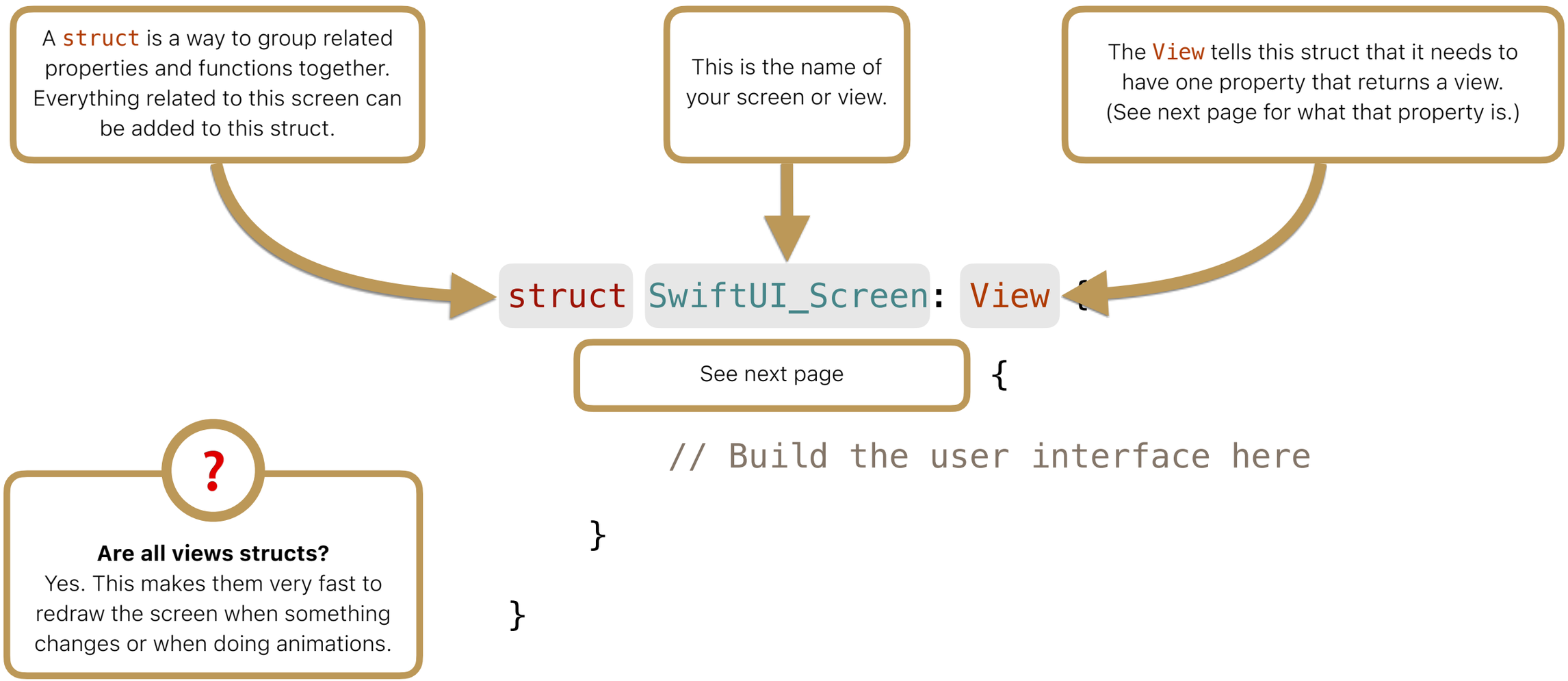
This tells Xcode that you want to use the SwiftUI library.

Apple has created many libraries for many different things, such as working with maps, calendars, photos, etc. If you want to use them, you import them first.

Note: For the remainder of this book I will not be including this import statement in the code samples but just know it is needed wherever you use SwiftUI.

Where it All Starts

Now that you understand the basic concepts of SwiftUI, let's see how you build these views.
Here is how every SwiftUI screen you will build starts:



The Body Property

All views require one property: `body`.

SwiftUI uses what is returned from the `body` property (a `View`) to draw on the screen of your app.

The VStack, HStack, and ZStack are all examples of views that can come from this property.

The `var` keyword in Swift says that this property can give you a value that can change (different from a value that can not change).

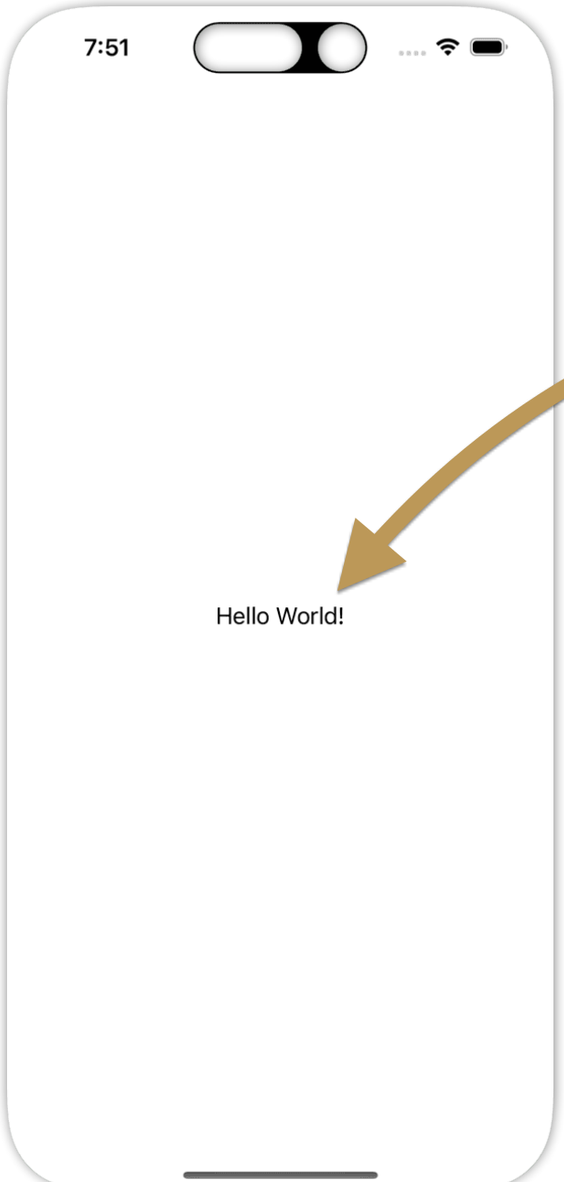
```
struct SwiftUI_Screen: View {

    var body: some View {

        // Build the user interface here

    }

}
```

This is the name of the property. **Note**: You should not change this name. It should always be "`body`".

The `some` keyword helps SwiftUI know that some kind of view will be returned so that it knows for sure it can draw it on the screen.

There are MANY kinds of views, but SwiftUI only has to know that the body property returns `some` kind of view to work with.
You will learn more about this later.

The View




```
struct BasicSyntax: View {

    var body: some View {

        get {

            return Text("Hello World!") // Adds a text view to the screen

        }

    }

}
```

You can get values and set values to properties in Swift.
This property can only have a `get`.

The `return` keyword sends the view back when SwiftUI requests it to draw on the screen.

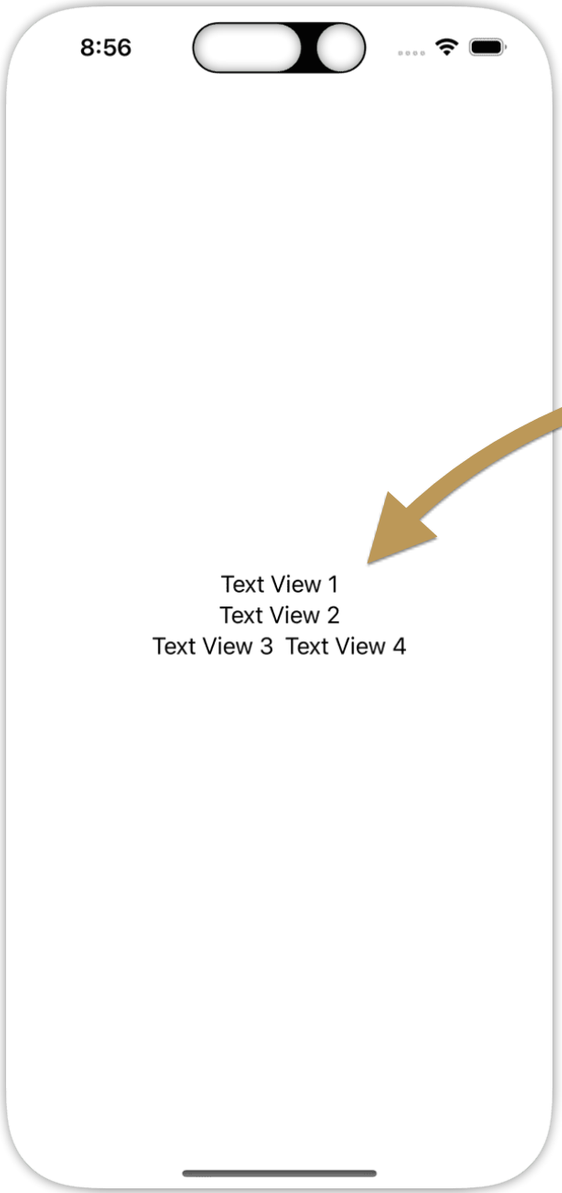
Shorter (more common)

```
struct BasicSyntax: View {

    var body: some View {

        Text("Hello World!") // Adds a text view to the screen

    }

}
```

The Swift language says that if there is only a `get`, you don't have to explicitly write it.

It also says if you are only returning one thing, you do not need to write `return`.

How to Display Multiple Views




```
struct MultipleViews: View {
    var body: some View {
        VStack {
            Text("Text View 1")
            Text("Text View 2")

            HStack {
                Text("Text View 3")
                Text("Text View 4")
            }
        }
    }
}
```

Use container views to hold multiple views you want to display.

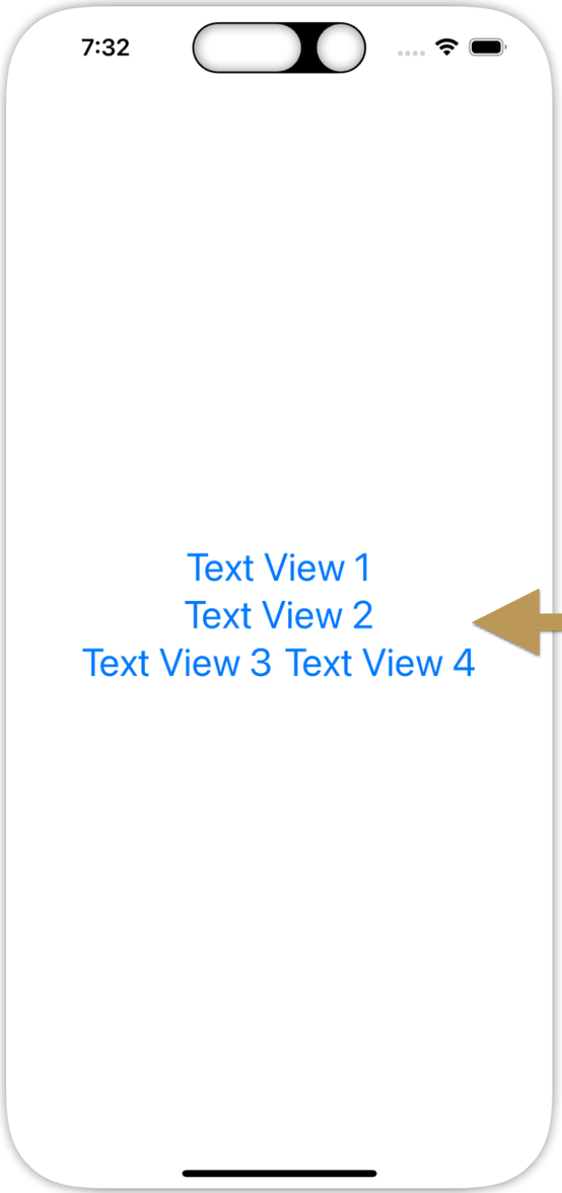
Note: Those braces after the `VStack { }` and `HStack { }` are called "closures".

They "close" around other views.

This is where you add all the views you want to be contained within them.

This is a common pattern for views that can contain other views.

Modifiers




```
struct Modifiers: View {
    var body: some View {
        VStack {
            Text("Text View 1")
            Text("Text View 2")

            HStack {
                Text("Text View 3")
                Text("Text View 4")
            }
        }
        .font(.title)
        .foregroundStyle(.tint)
    }
}
```

These lines of code **modify** or change the views they are attached to.

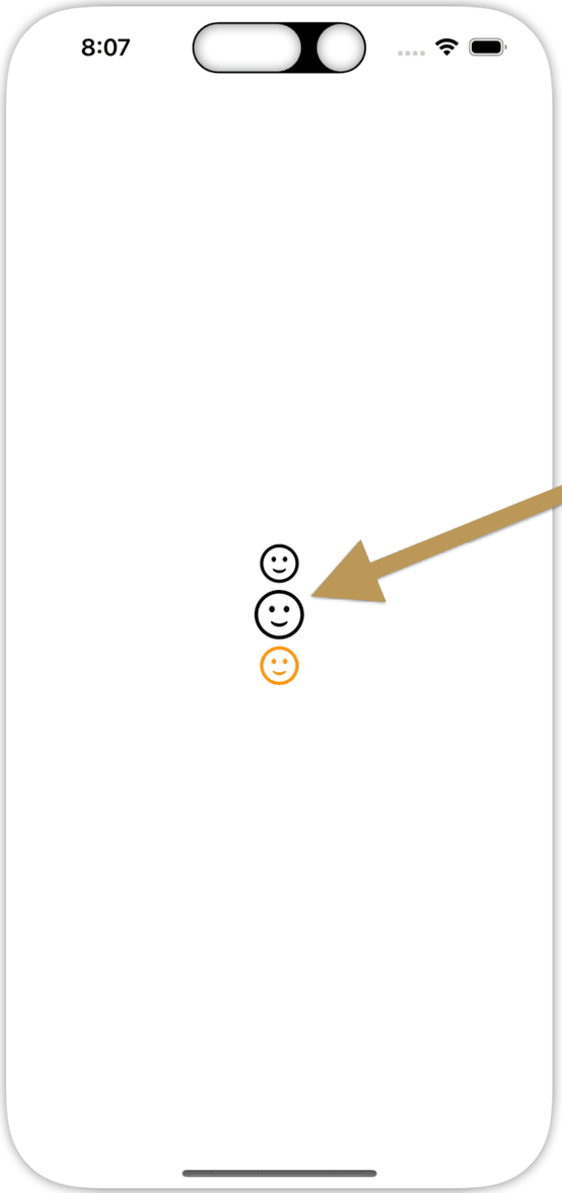
That is why they are called "**modifiers**".

When on a container view, the change can happen to all of the child views.

1. **Font** - The font modifier changes how text appears, such as making it bigger or smaller.
2. **Title** - Notice a specific size is not mentioned here. That is because this size is relative to the default size. Your app users can change the text size on their devices. So this text can get bigger or smaller but it will always be bigger than the default size.
3. **ForegroundStyle** - Use this modifier to change colors and styles of views.
4. **Tint** - This is the color buttons or interactive elements will usually use. You can change this tint color. The default is blue.

Note: Try moving these modifiers around to different views to see what happens.

Symbol Images




```
struct Symbols: View {
    var body: some View {
        VStack {
            Image(systemName: “smiley")

            Image(systemName: "smiley")
                .imageScale(.large)

            Image(systemName: "smiley")
                .foregroundStyle(.orange)
        }
        .font(.title)
    }
}
```

This **imageScale** modifier will make the image relatively bigger than the rest.

?

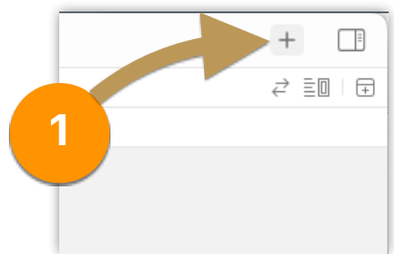
Does the font modifier affect symbols?
Yes! Symbols are considered fonts.

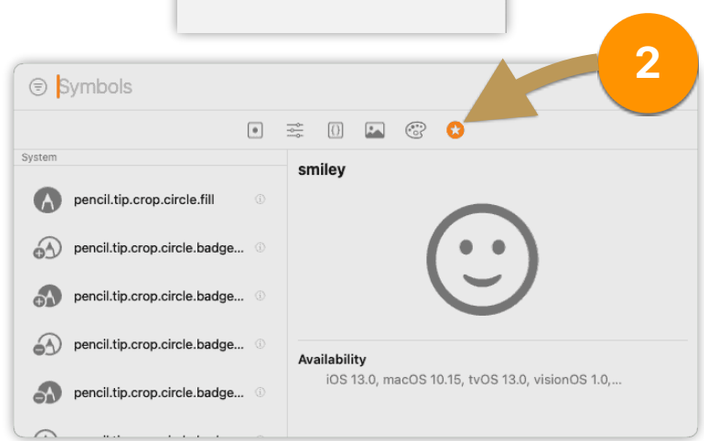
SwiftUI has an **Image** view that can create icons or “**symbols**”.

There are thousands you can choose from.

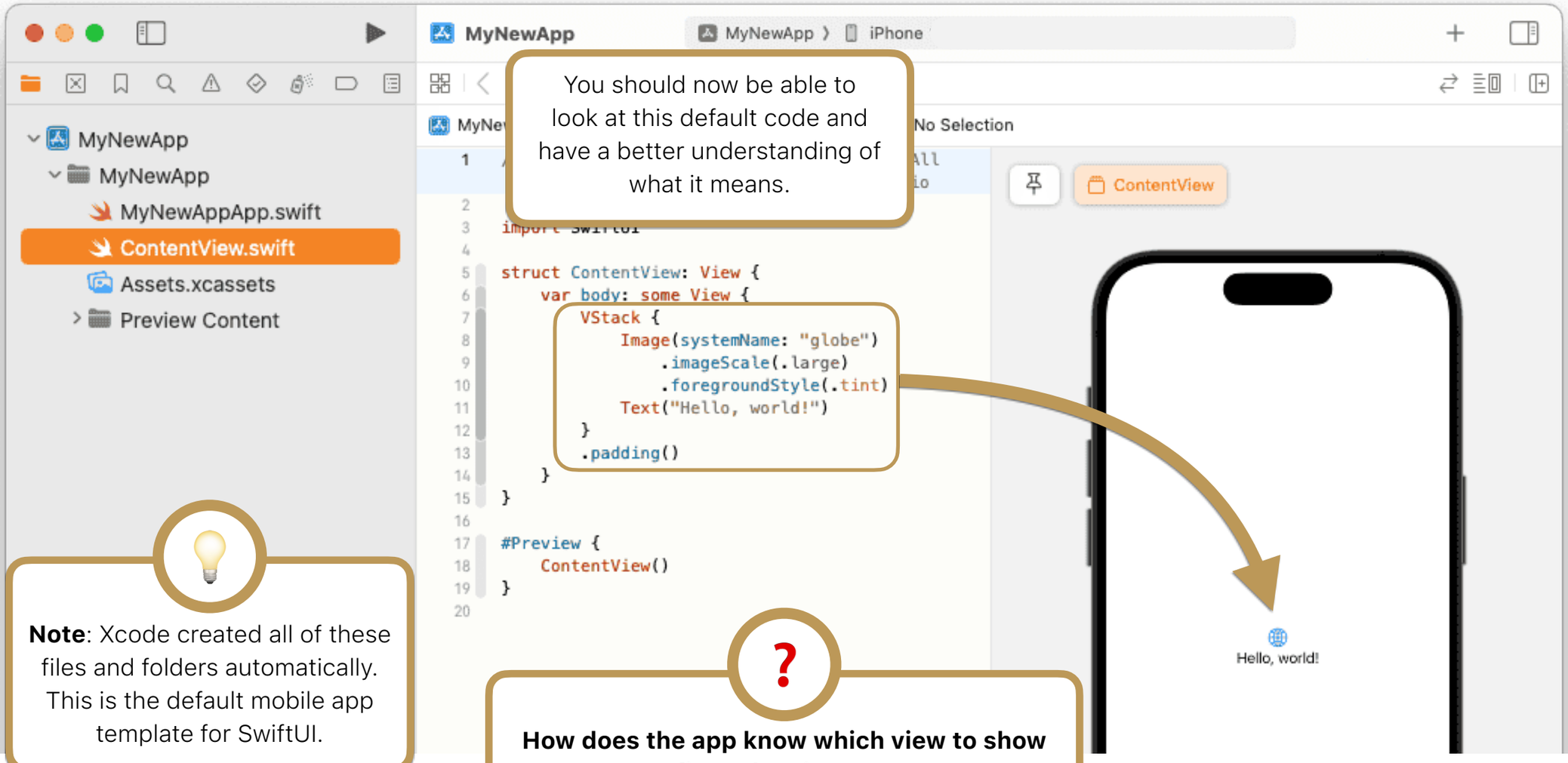
To browse them:

1. Open the Library (+ button in upper right of Xcode or Command + Shift + L).
2. Click the last tab (star icon).
3. Scroll or search for what you want.





How does your project know which SwiftUI screen to show?

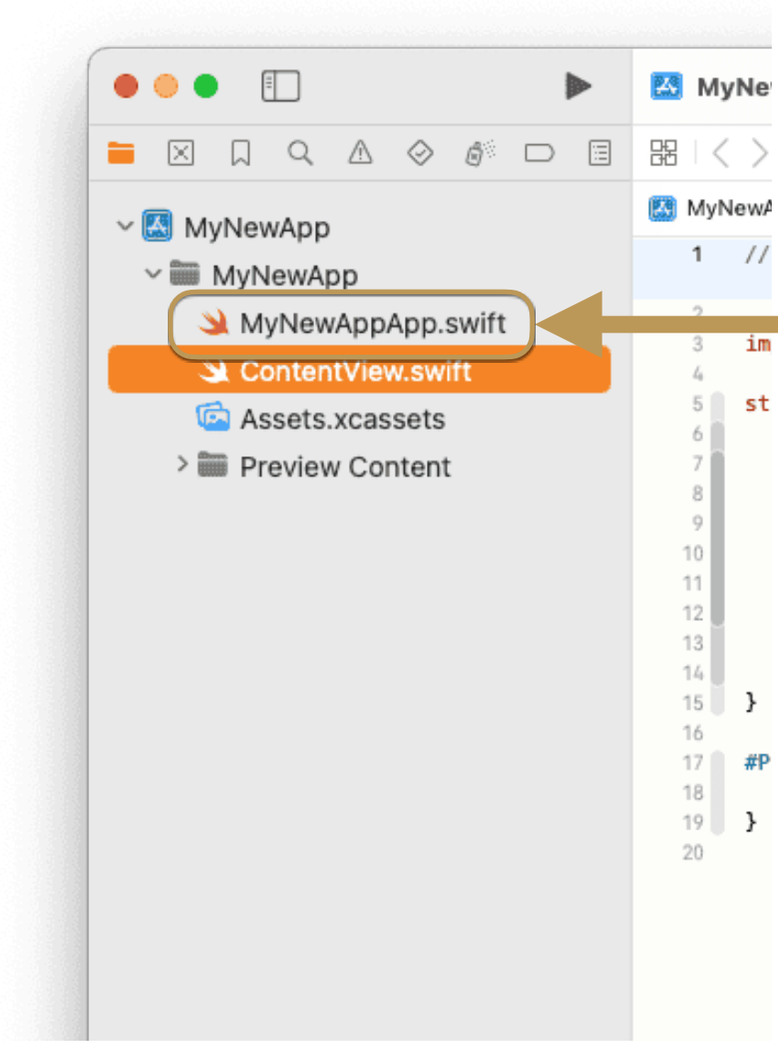


You should now be able to look at this default code and have a better understanding of what it means.

Note: Xcode created all of these files and folders automatically. This is the default mobile app template for SwiftUI.

How does the app know which view to show first when it runs?
Good question. Let's cover that on the next page.

The Project Starting Point



The app's starting point is specified in this file that usually ends with "…**App.swift**".

Xcode added this file for you.

When your app runs, this file is used as the starting point so it knows what SwiftUI view to show **first**.

App File - The App Starting Point

Your application's starting point is like putting on a show, or a play. There are a lot of different parts to organize so the show can go on!

This `@main` attribute is like the front door to the show.

It is used to define the application's entry point.

Many things can happen in a scene. The `WindowGroup` is like a stage manager that coordinates all the different scenes that appears.

The WindowGroup is a type of Scene.

It can handle multiple windows (screens) **but in iOS, it is just one screen**.

```
@main
struct SomeSwiftUIApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Think of `App` as the director of a play that decides what the user sees when the play (app) starts.

The `App` decides what the opening Scene is.

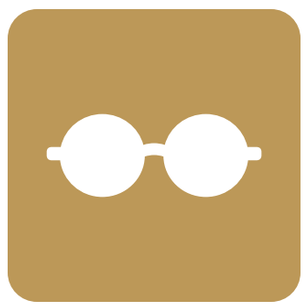
The `Scene` is like the stage where the play happens.

It is the part that the user can see and interact with.

The `ContentView` is the first actor you want to appear on the stage.

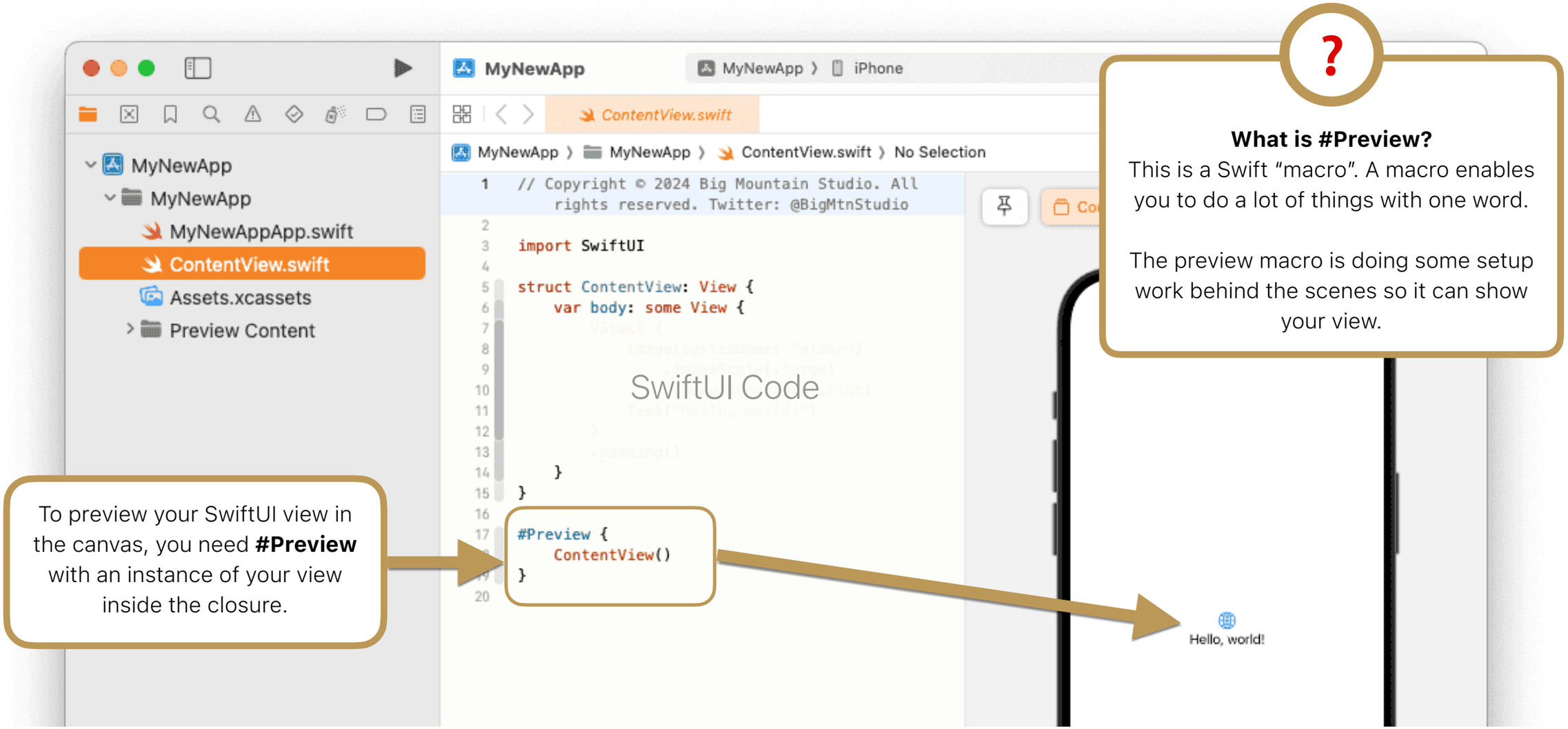
You will decide which view you want to show first here.

PREVIEWS



In this chapter, you will learn how to view your SwiftUI code with different options right inside of Xcode.

Preview


MyNewApp
MyNewApp
MyNewApp
MyNewAppApp.swift
ContentView.swift
Assets.xcassets
Preview Content
MyNewApp 〉 iPhone
ContentView.swift
MyNewApp 〉 MyNewApp 〉 ContentView.swift 〉 No Selection
1 // Copyright © 2024 Big Mountain Studio. All rights reserved. Twitter: @BigMtnStudio
3 import SwiftUI
5 struct ContentView: View {
6 var body: some View {
14 }
15 }
17 #Preview {
18 ContentView()
19 }
SwiftUI Code
Hello, world!
?
What is #Preview?
This is a Swift “macro”. A macro enables you to do a lot of things with one word.
The preview macro is doing some setup work behind the scenes so it can show your view.
To preview your SwiftUI view in the canvas, you need #Preview with an instance of your view inside the closure.

Previews Options

```
#Preview {
    ContentView()
}
```

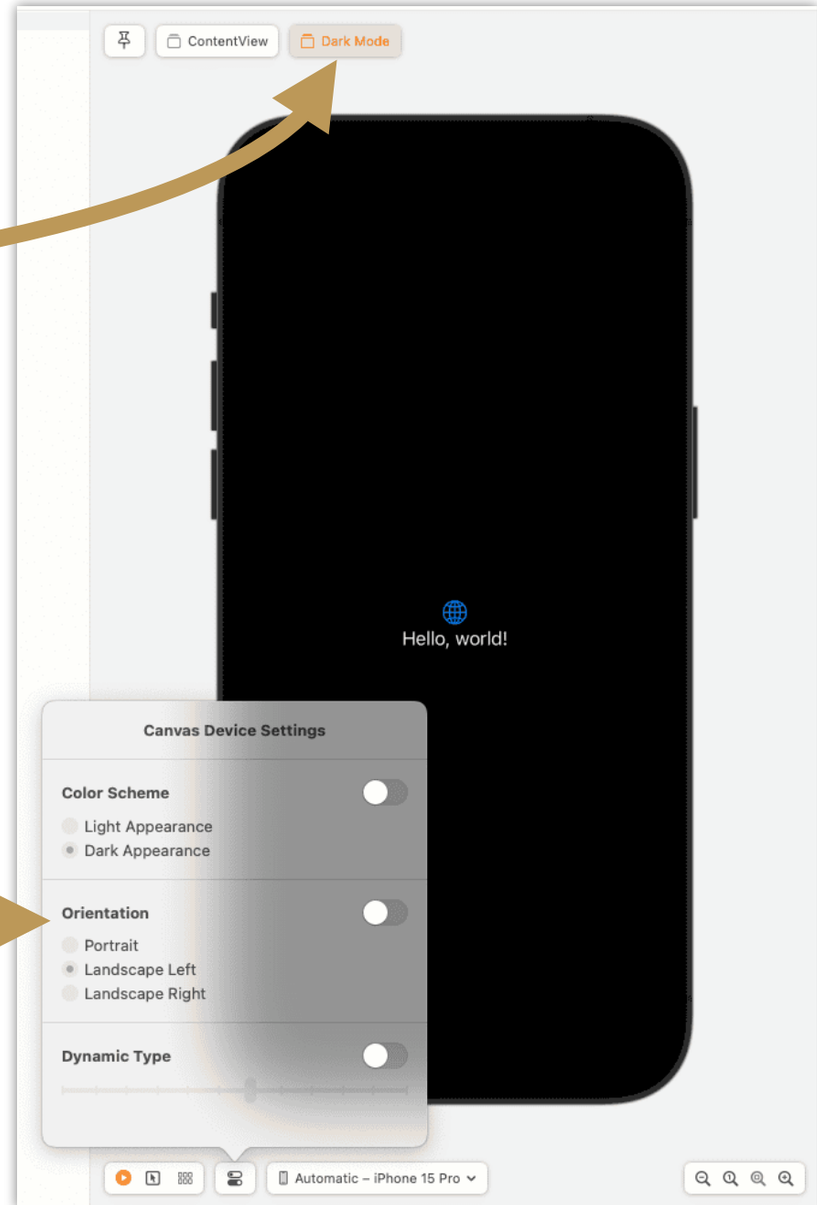
Create as many previews as you need.

You can give your previews a name.

```
#Preview("Dark Mode") {
    ContentView()
        .preferredColorScheme(.dark)
}
```

You can also use a modifier to change the color.

Choose from a variety of options to change the way your device looks so you can see how your view responds.



VIEWS

In this chapter, you will get an overview of some of the more popular SwiftUI **views** and **modifiers**.

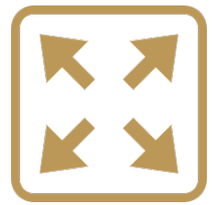
SwiftUI View Layout Behavior

In SwiftUI, you may wonder why some views layout differently than others.
You can observe two layout behaviors when it comes to the size and layout of views: some pull-in, while others push out.

Pull-In Views

Some views will use as little space on the UI as possible.
They will only use as much space as needed to display their own content.

Examples
Text views
Image views



Push-Out Views

Some views will push out and take up as much space as it can within the view.
These views will even push other views away.

Examples
Color views
Shape views

Pull-In & Push-Out Views

Pull-In Views

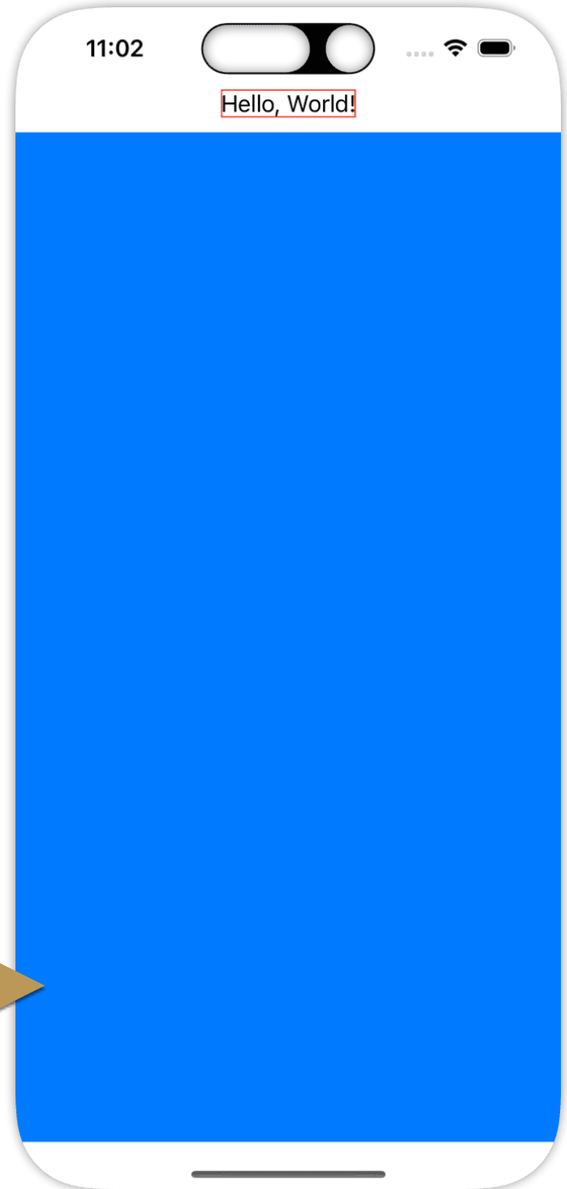
```
struct PullInPushOutViews: View {
    var body: some View {
        Text("Hello, World!")
            .border(Color.red)
    }
}
```

You can tell from the `border` modifier that the Text view is only as big as its contents.

Push-Out Views

If you add a Color, it will **push out** and take up all available space and push the Text view to the top.

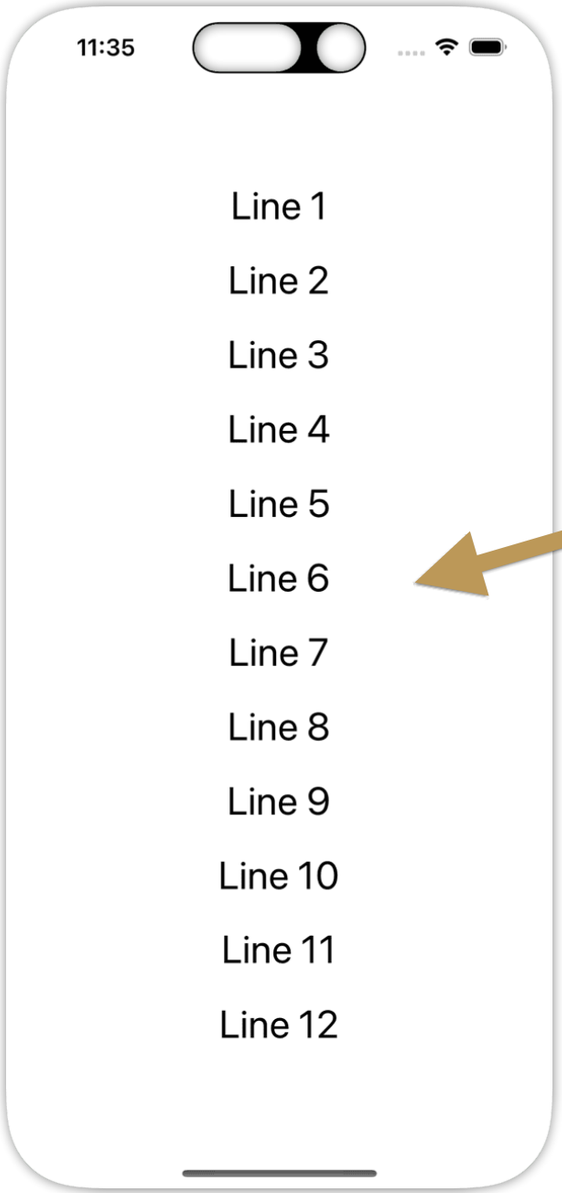
```
struct PullInPushOutViews: View {
    var body: some View {
        Text("Hello, World!")
            .border(Color.red)

        Color.blue
    }
}
```



VStack

Use the VStack to arrange views vertically.

VStack - Spacing



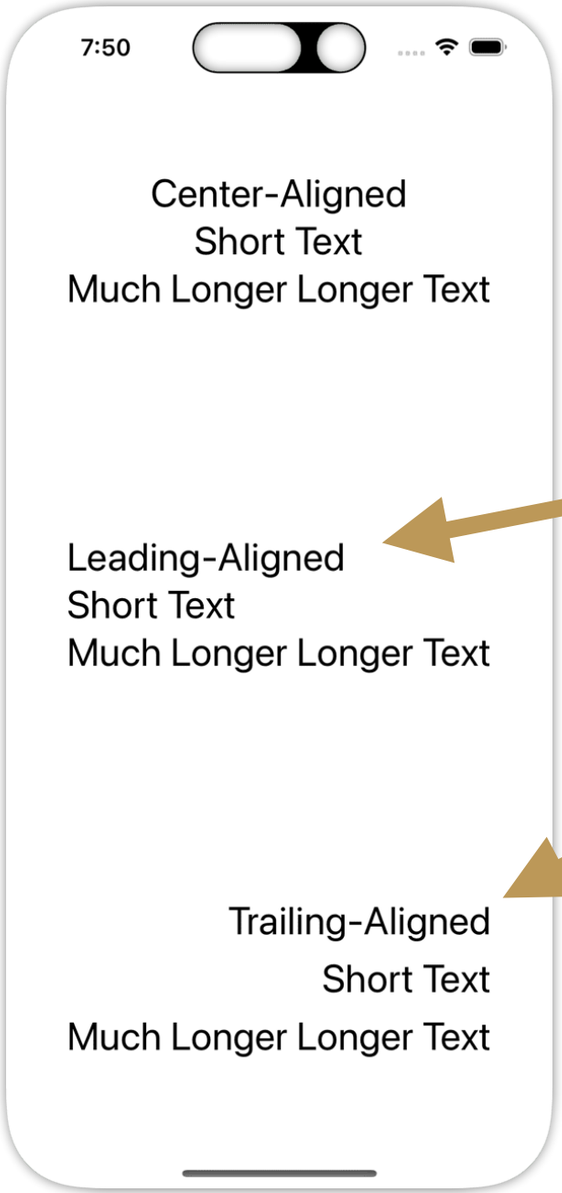

```
struct TheVStackView: View {
    var body: some View {
        VStack(spacing: 20.0) {
            Text("Line 1")
            Text("Line 2")
            Text("Line 3")
            Text("Line 4")
            Text("Line 5")
            Text("Line 6")
            Text("Line 7")
            Text("Line 8")
            Text("Line 9")
            Text("Line 10")
            Text("Line 11")
            Text("Line 12")
        }
        .font(.title)
    }
}
```

The VStack lets you stack views vertically.

You can specify how much space is between each view too.

Note: The VStack is a pull-in view. It will only grow to the size of its contents.

VStack - Alignment




```swift
struct TheVStackWithAlignmentView: View {
    var body: some View {
        VStack(spacing: 160) {
            VStack {
                Text("Center-Aligned")
                Text("Short Text")
                Text("Much Longer Longer Text")
            }

            VStack(alignment: .leading) {
                Text("Leading-Aligned")
                Text("Short Text")
                Text("Much Longer Longer Text")
            }

            VStack(alignment: .trailing, spacing: 8.0) {
                Text("Trailing-Aligned")
                Text("Short Text")
                Text("Much Longer Longer Text")
            }
        }
        .font(.title)
    }
}
```

Use the alignment parameter to line up all the child views within.

The default is centered (first example).

?

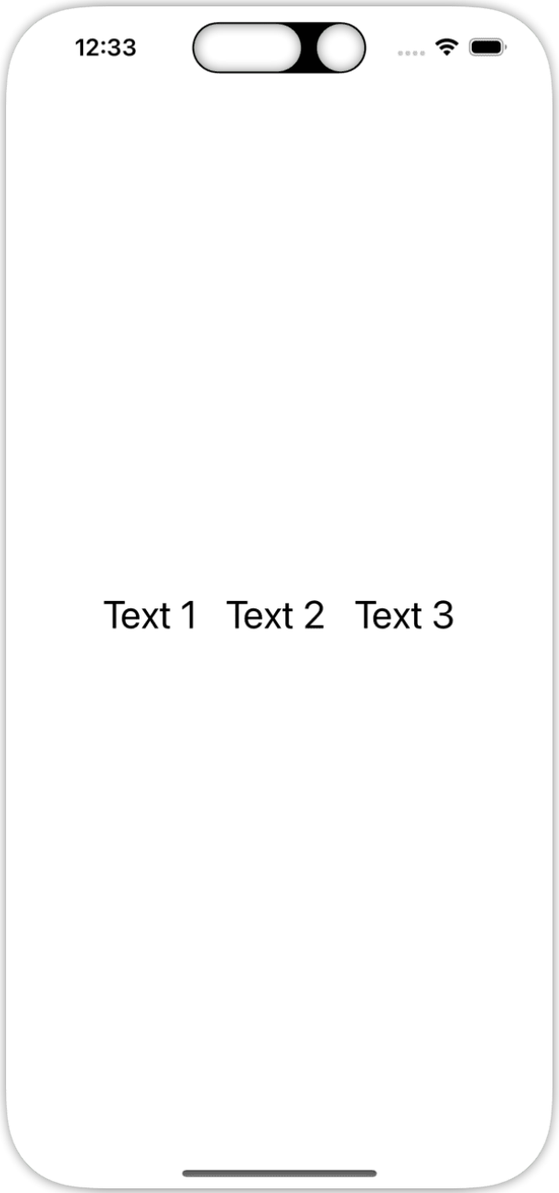
Why are alignment options called “leading” and “trailing”?
Languages can read left-to-right, like English, or right-to-left, like Arabic.

Leading specifies where the beginning of that language starts and **trailing** is the other side.

HStack

Use the HStack to arrange views horizontally.

HStack - Spacing



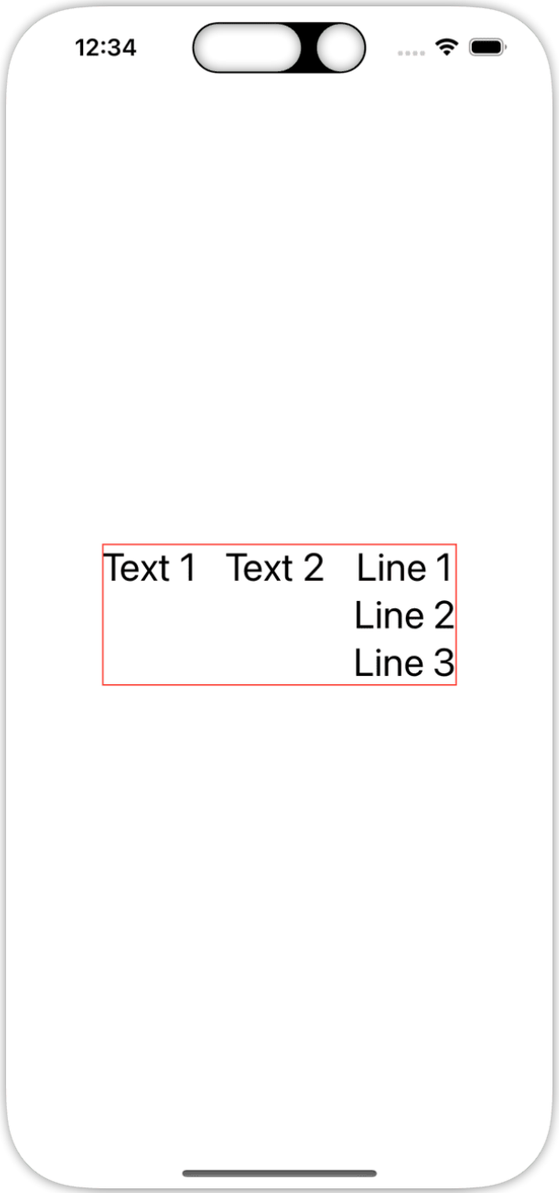

```
struct TheHStackView: View {
    var body: some View {
        HStack(spacing: 20.0) {
            Text("Text 1")

            Text("Text 2")

            Text("Text 3")
        }
        .font(.title)
    }
}
```

The HStack lets you stack views horizontally.

You can specify how much space is between each view.

Note: The HStack is a pull-in view. It will only grow to the size of its contents.

HStack - Alignment



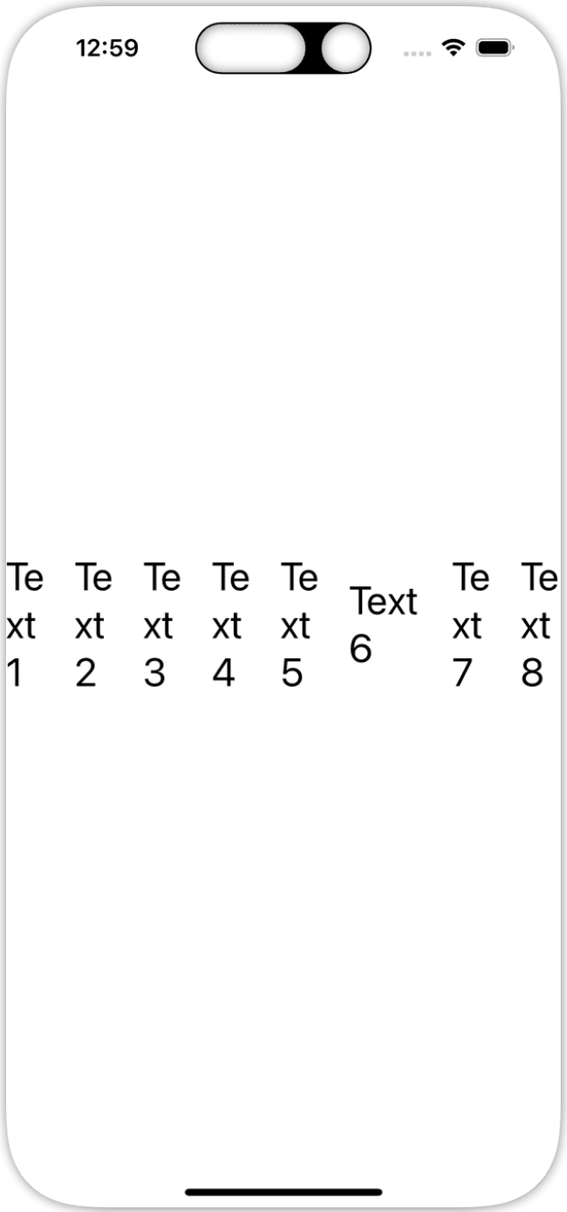

```
struct TheHStackWithAlignmentView: View {
    var body: some View {
        HStack(alignment: .top, spacing: 20.0) {
            Text("Text 1")

            Text("Text 2")

            VStack {
                Text("Line 1")
                Text("Line 2")
                Text("Line 3")
            }
        }
        .font(.title)
        .border(Color.red)
    }
}
```

Use the alignment parameter to line up all the child views within.

Options:

- .top
- .center (default)
- .bottom

Too Many Views




```
struct TheHStackTooManyView: View {
    var body: some View {
        HStack(spacing: 20.0) {
            Text("Text 1")

            Text("Text 2")

            Text("Text 3")

            Text("Text 4")

            Text("Text 5")

            Text("Text 6")

            Text("Text 7")

            Text("Text 8")
        }
        .font(.title)
    }
}
```

If you add too many views to any stack view, they will start to get squished together.

Another potential problem is the user could increase the text size on their device which will enlarge text in your app. So you will have to test for that.

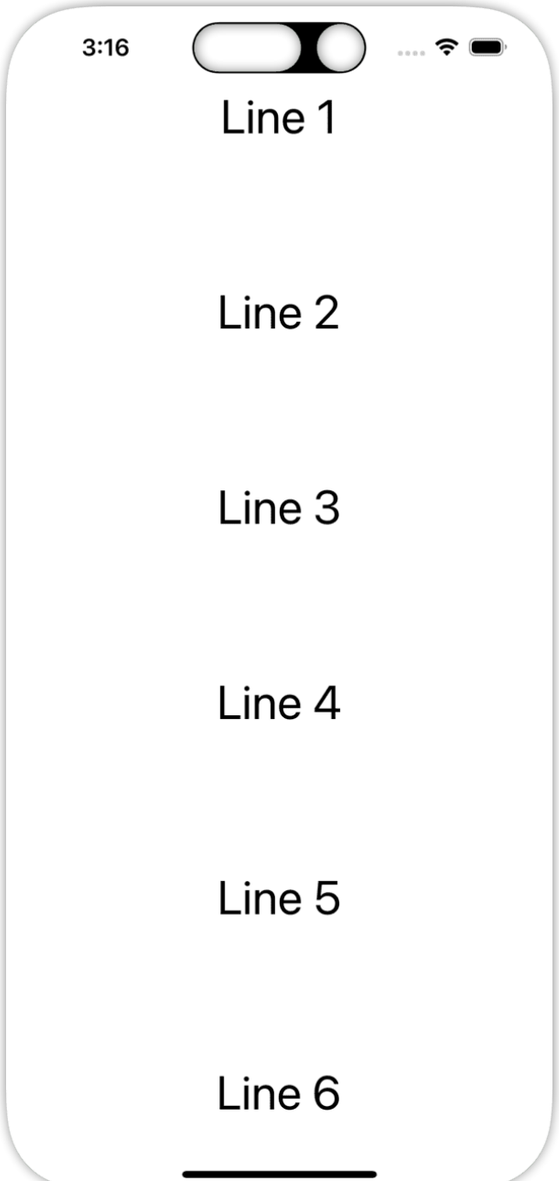
?

What if I don't want the views to be squished but to scroll off the side of the screen?
SwiftUI offers a ScrollView just for that purpose. Let's take a look at it next.

ScrollView

Add views to this container view so that they can scroll vertically or horizontally.

ScrollView: Vertical



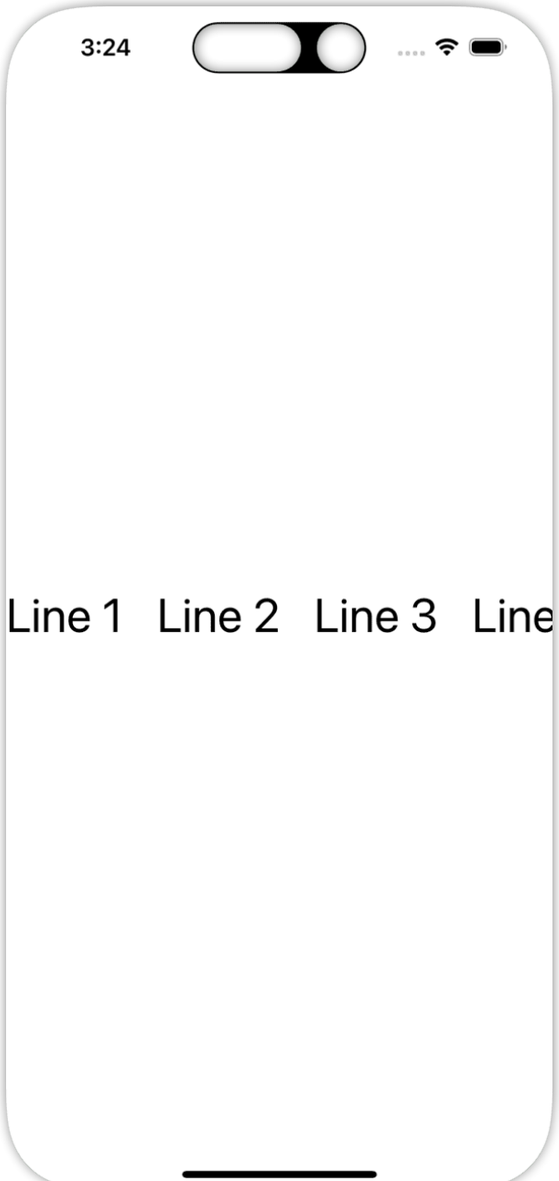

```
struct TheScrollView_Vertical: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 100.0) {
                Text("Line 1")
                Text("Line 2")
                Text("Line 3")
                Text("Line 4")
                Text("Line 5")
                Text("Line 6")
                Text("Line 7")
                Text("Line 8")
            }
            .font(.largeTitle)
        }
    }
}
```

The **ScrollView** is a container that will allow the user to scroll the view content.

Adding more spacing here to push the views off the screen.

The **largeTitle** parameter is the largest relative font size.

ScrollView: Horizontal




```
struct TheScrollView_Horizontal: View {
    var body: some View {
        ScrollView(.horizontal) {
            HStack(spacing: 24.0) {
                Text("Line 1")
                Text("Line 2")
                Text("Line 3")
                Text("Line 4")
                Text("Line 5")
                Text("Line 6")
                Text("Line 7")
                Text("Line 8")
            }
            .font(.largeTitle)
        }
    }
}
```

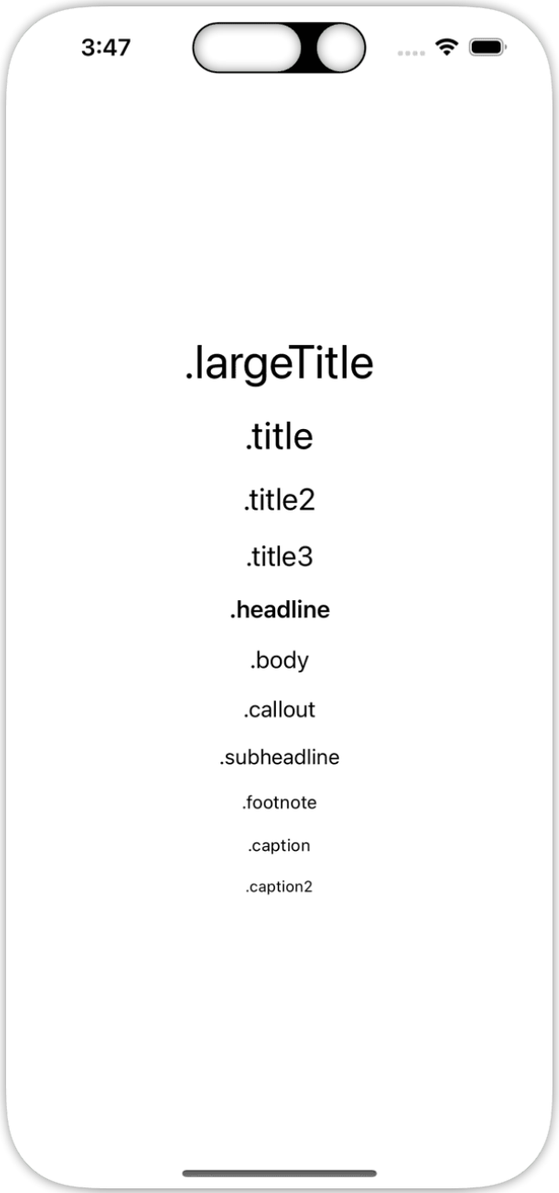
To scroll horizontally you can specify **.horizontal** for the scroll direction.

Then you can use the HStack to arrange views horizontally.

Text

Use the Text view to display any text in your UI.

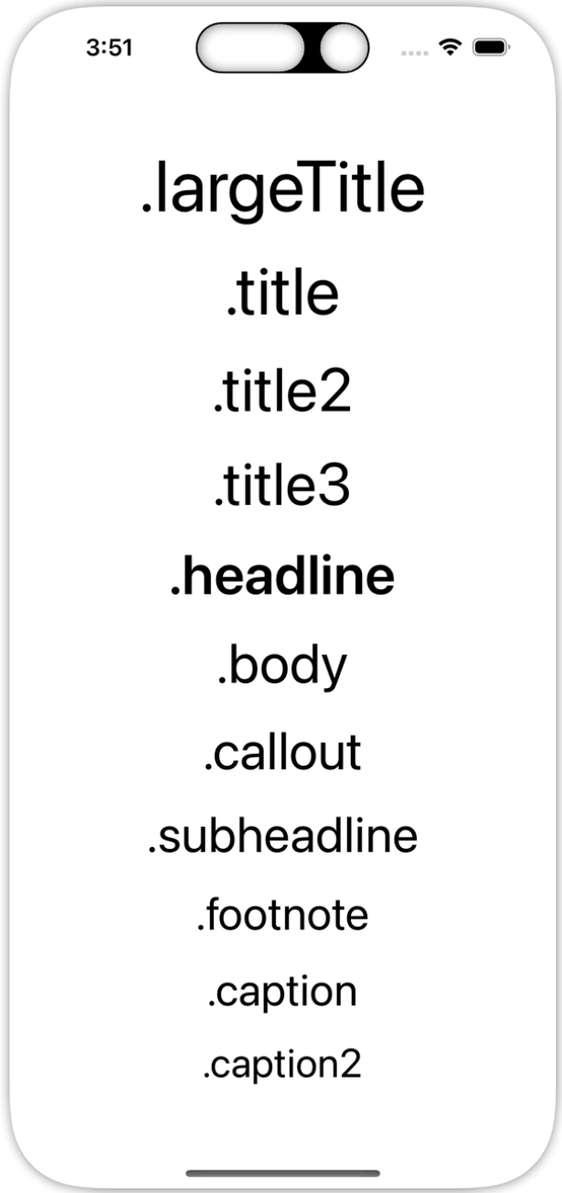
Text Styles




```
struct TheTextView_Styles: View {
    var body: some View {
        VStack(spacing: 16.0) {
            Text(".largeTitle")
                .font(.largeTitle)
            Text(".title")
                .font(.title)
            Text(".title2")
                .font(.title2)
            Text(".title3")
                .font(.title3)
            Text(".headline")
                .font(.headline)
            Text(".body")
                .font(.body)
            Text(".callout")
                .font(.callout)
            Text(".subheadline")
                .font(.subheadline)
            Text(".footnote")
                .font(.footnote)
            Text(".caption")
                .font(.caption)
            Text(".caption2")
                .font(.caption2)
        }
    }
}
```

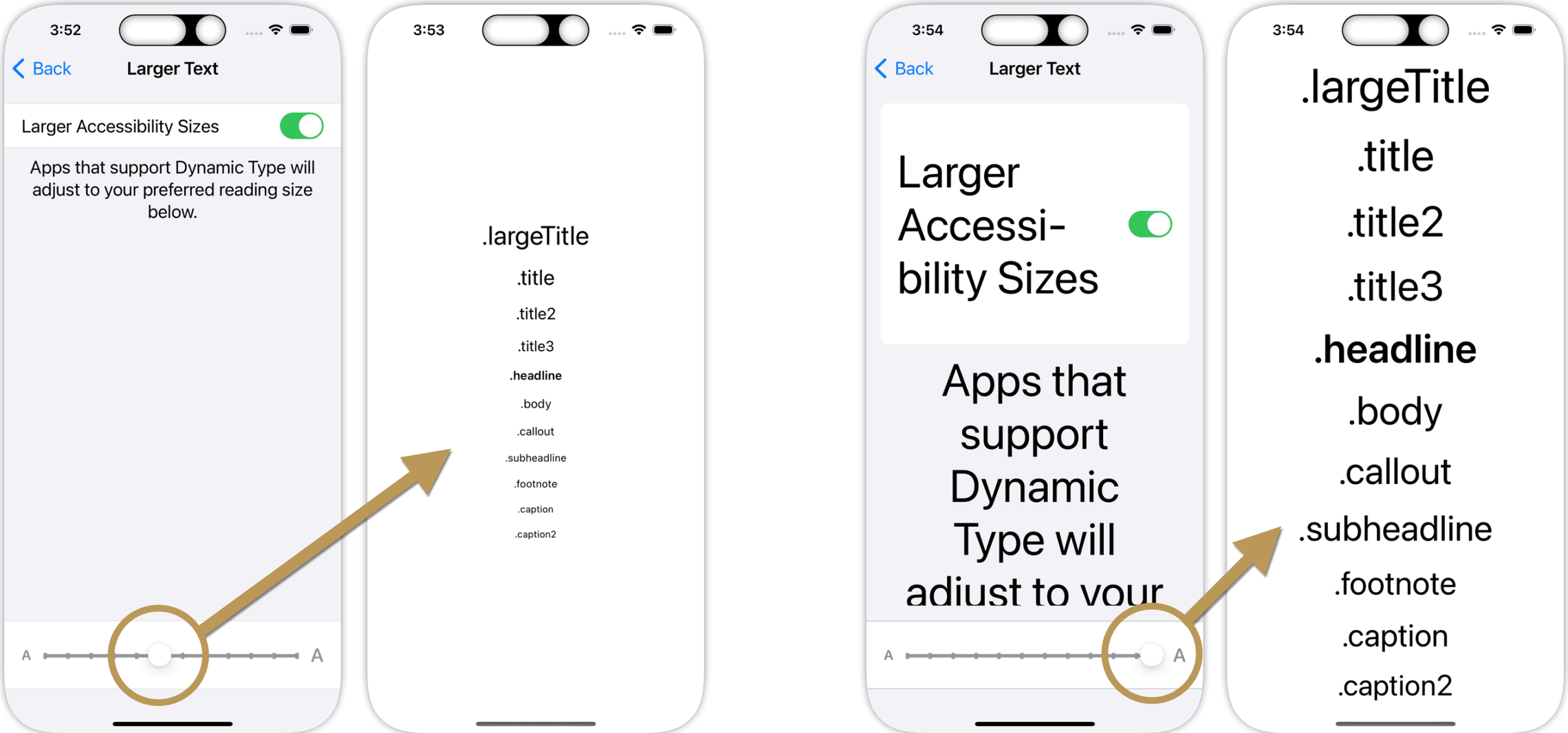
Here are all the relative sizes you can use for your Text views.

Remember, they are relative because they can all get larger or smaller depending on the user's settings on their device.

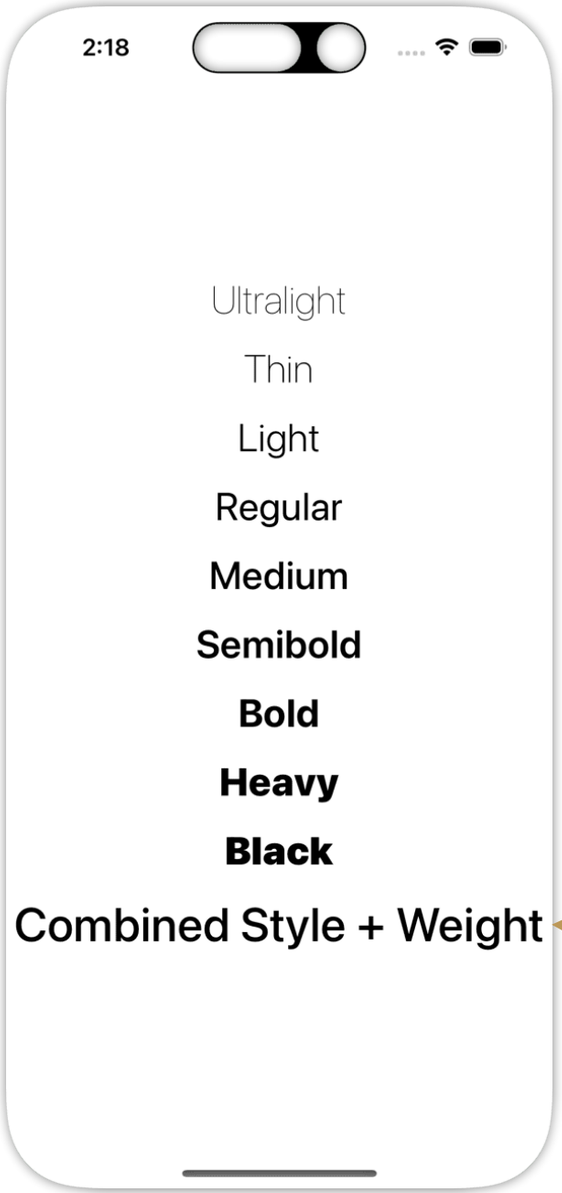


Text: Accessibility Size

Text styles are relative to what the user has set in the device text size settings.



Text: Weight




```
struct TheTextView_Weight: View {
    var body: some View {
        VStack(spacing: 16.0) {
            Text("Ultralight")
                .fontWeight(.ultraLight)
            Text("Thin")
                .fontWeight(.thin)
            Text("Light")
                .fontWeight(.light)
            Text("Regular")
                .fontWeight(.regular)
            Text("Medium")
                .fontWeight(.medium)
            Text("Semibold")
                .fontWeight(.semibold)
            Text("Bold")
                .fontWeight(.bold)
            Text("Heavy")
                .fontWeight(.heavy)
            Text("Black")
                .fontWeight(.black)

            Text("Combined Style + Weight")
                .font(.largeTitle.weight(.medium))
        }
        .font(.title)
    }
}
```

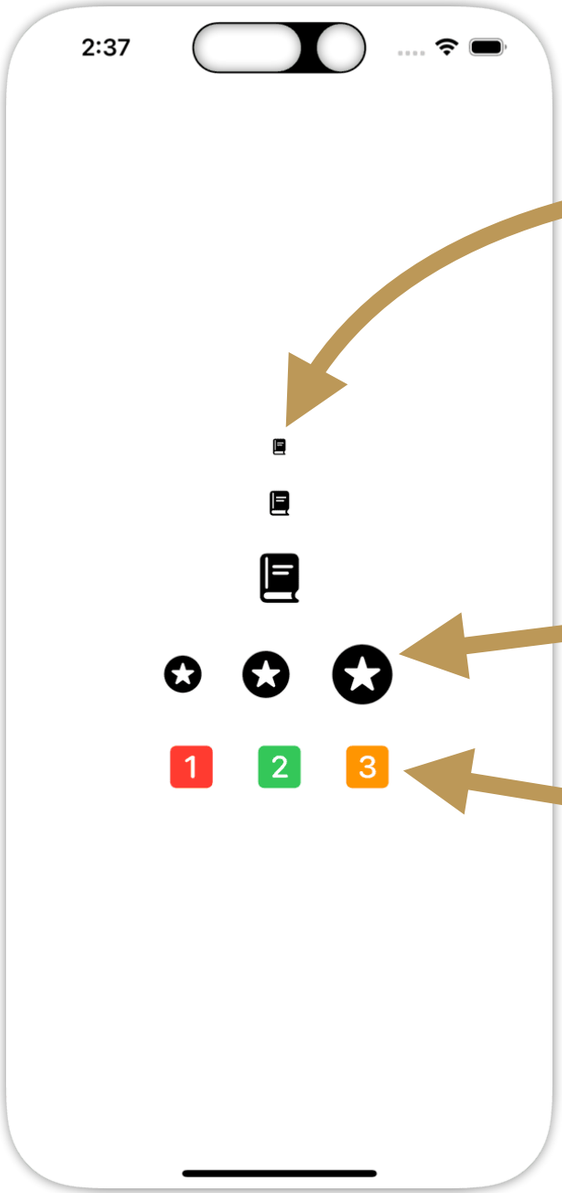
The font weight determines the thickness of the text.

Note: You can combine the text style and the weight within the **font** modifier.

Image

Use the Image view to display symbols (shown earlier) or graphical images.

Image: Symbol




```
struct TheImageView: View {
    var body: some View {
        VStack(spacing: 24.0) {
            Image(systemName: "text.book.closed.fill")
                .font(.caption2)

            Image(systemName: "text.book.closed.fill")

            Image(systemName: "text.book.closed.fill")
                .font(.largeTitle)

            HStack(spacing: 24.0) {
                Image(systemName: "star.circle.fill")
                    .imageScale(.small)
                Image(systemName: "star.circle.fill")
                    .imageScale(.medium)
                Image(systemName: "star.circle.fill")
                    .imageScale(.large)
            }
            .font(.largeTitle)

            HStack(spacing: 24.0) {
                Image(systemName: "1.square.fill")
                    .foregroundStyle(.red)
                Image(systemName: "2.square.fill")
                    .foregroundStyle(.green)
                Image(systemName: "3.square.fill")
                    .foregroundStyle(.orange)
            }
            .font(.largeTitle)
        }
    }
}
```

There are thousands of symbols you can choose from.

Since symbols are fonts, you can use the **font** modifier to change its size.

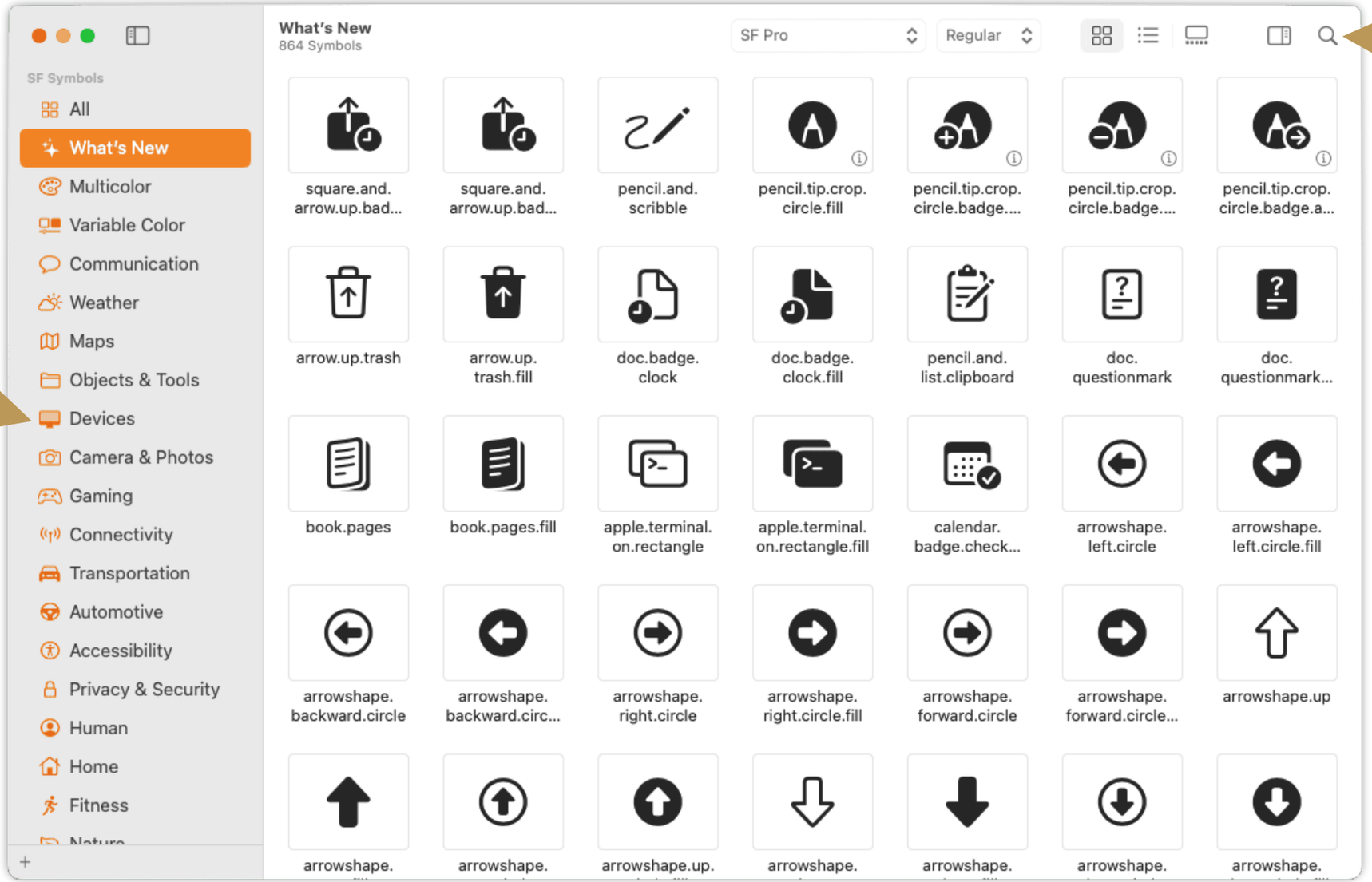
Use the **imageScale** modifier to make the size bigger or smaller.

Use the **foregroundStyle** to change the color.

SF Symbols App

Download and use the SF Symbols app to browse symbols. You can copy the name of the symbol to use within your Xcode project images.

Use the categories on the side to narrow down your search.



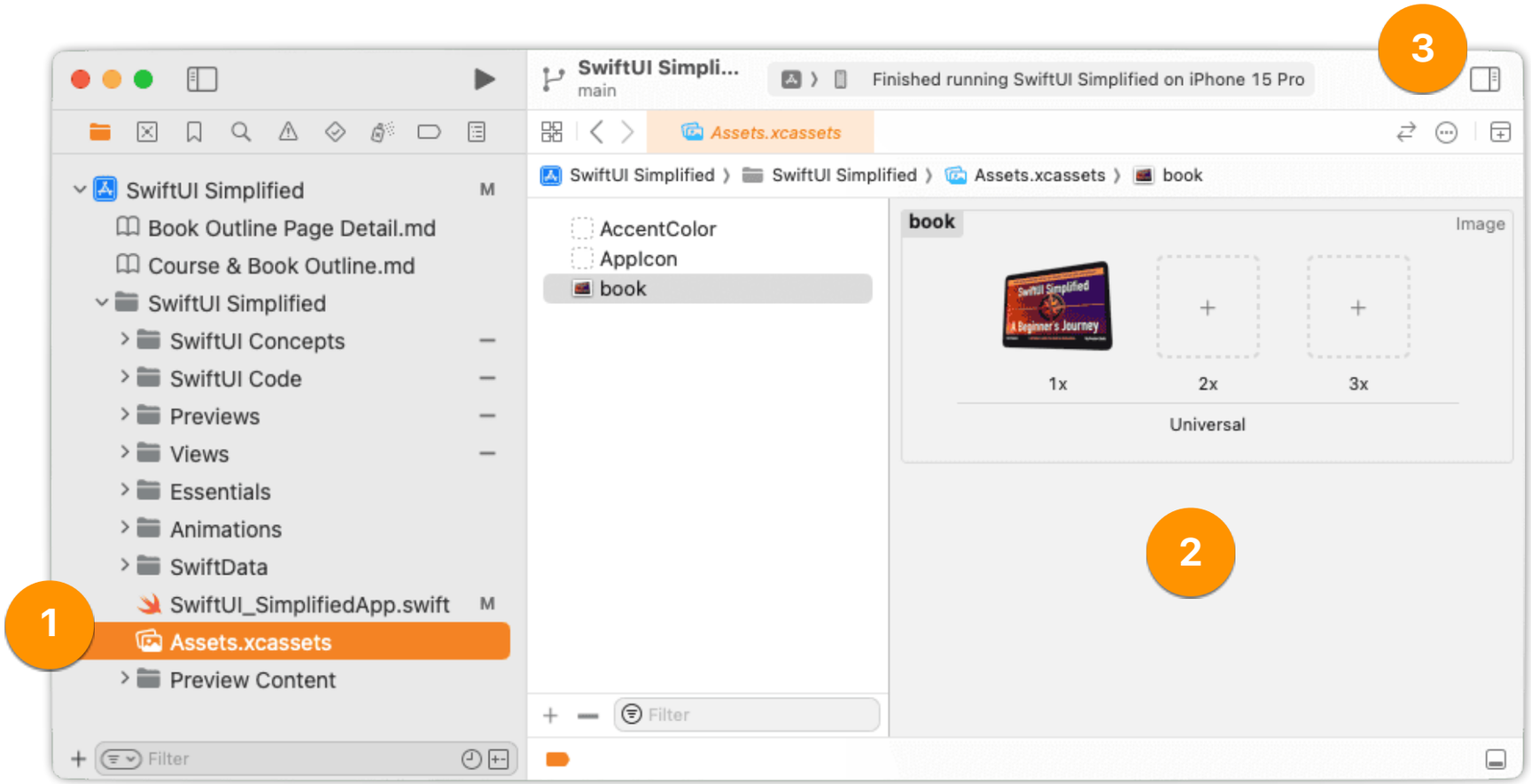
You can also use the search function.

Tip: Copy the *name* of the symbol with **Command + Shift + C**.

Adding Images to Your Project

To add images:

1. Select the Assets.xcassets folder.
2. Drag an image into the middle area.
3. Open the right side panel for more asset settings.



Why are there 3 slots (1x, 2x, 3x)?

This was needed when there were more devices that supported a variety of resolutions. After iPhone 8, however, only high resolutions are needed (3x).

You can change this image to use a single scale. (See next page)

Switching to Single Scale

To switch to single scale:

1. Click the right pane button in the upper-right corner of Xcode.
2. Scroll down to "Scales" and select "**Single Scale**" from the drop down.

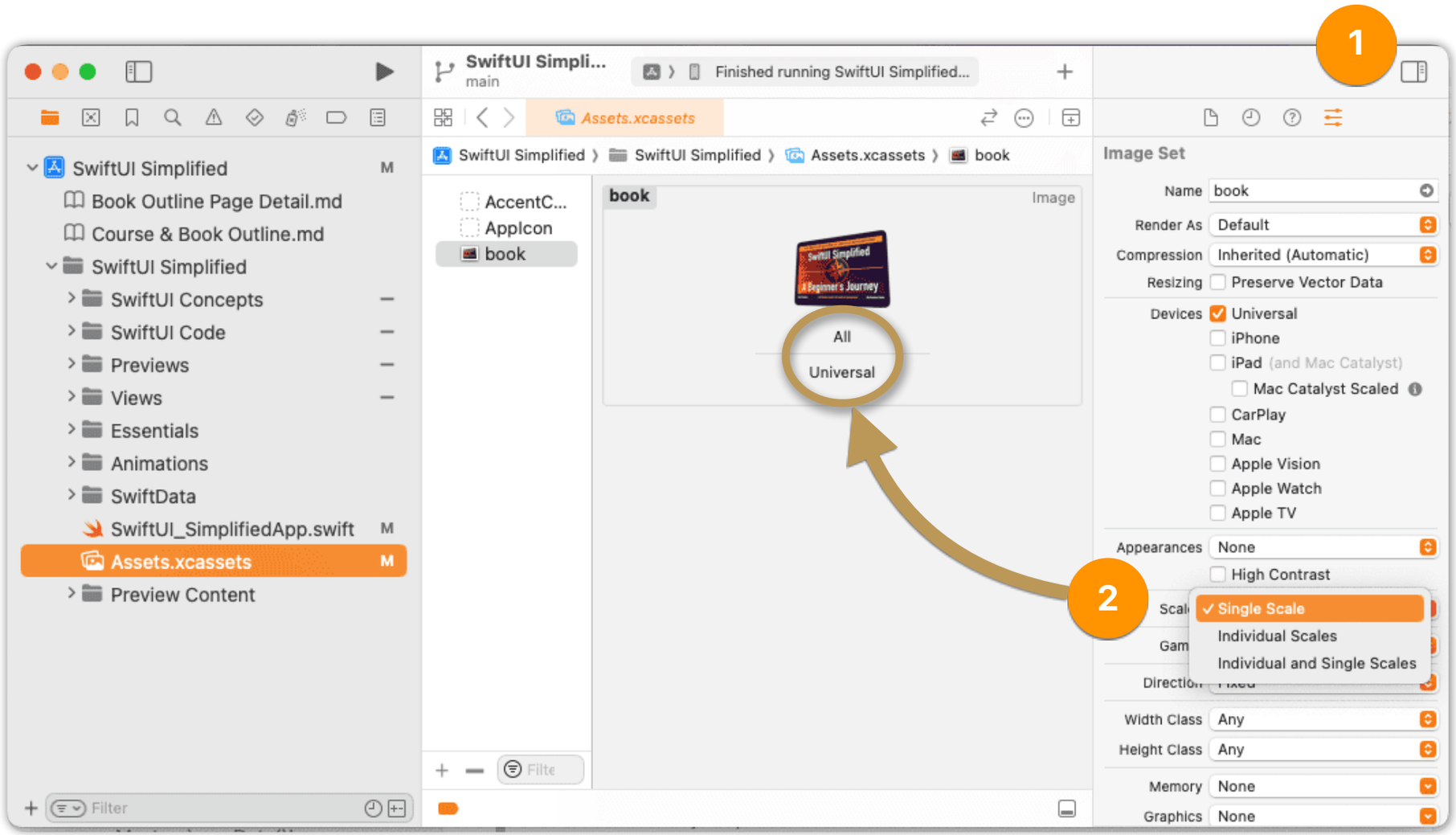
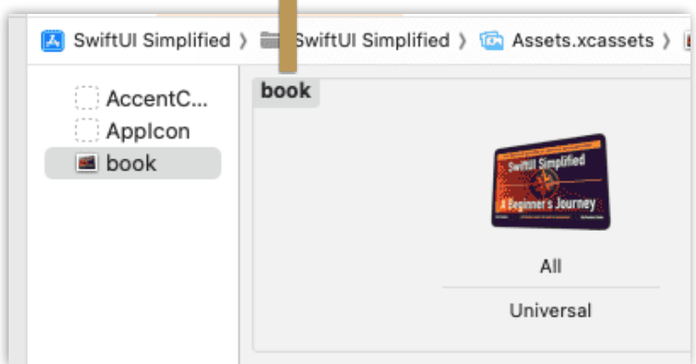


Image: Displaying


```
struct TheImageView_Displaying: View {
    var body: some View {
        Image(.book)
    }
}
```



The Image view you want to use is the one that looks like this:

`Image(_ resource:)`

The resource is the image in the Xcode Assets (xcassets) folder.

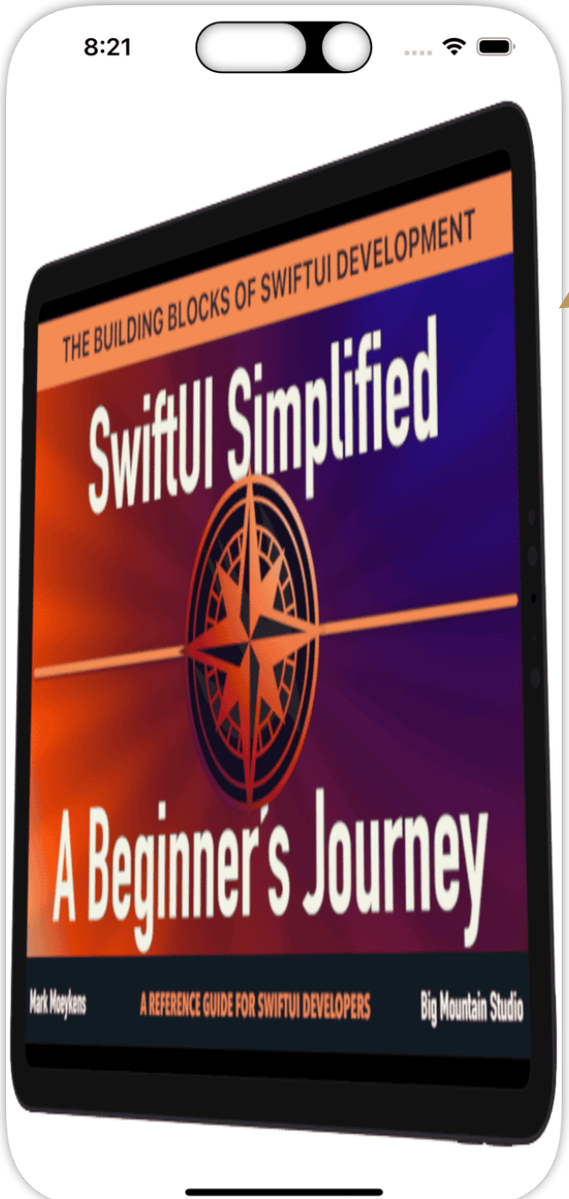
Images added to the assets folder are given a generated name (`.book`) you can code with based on the name in the assets folder.

?

Why doesn't the image fit the screen?
The image is being displayed using its original size. To make it fit, you will have to use some modifiers to resize it based on the available space it has.

The Image is a pull-in view, it is only using as much space as its contents. But if the image is large, you might think it's a push-out view.

Image: Resizing




```
struct TheImageView_Resizing: View {
    var body: some View {
        Image(.book)
            .resizable()
    }
}
```

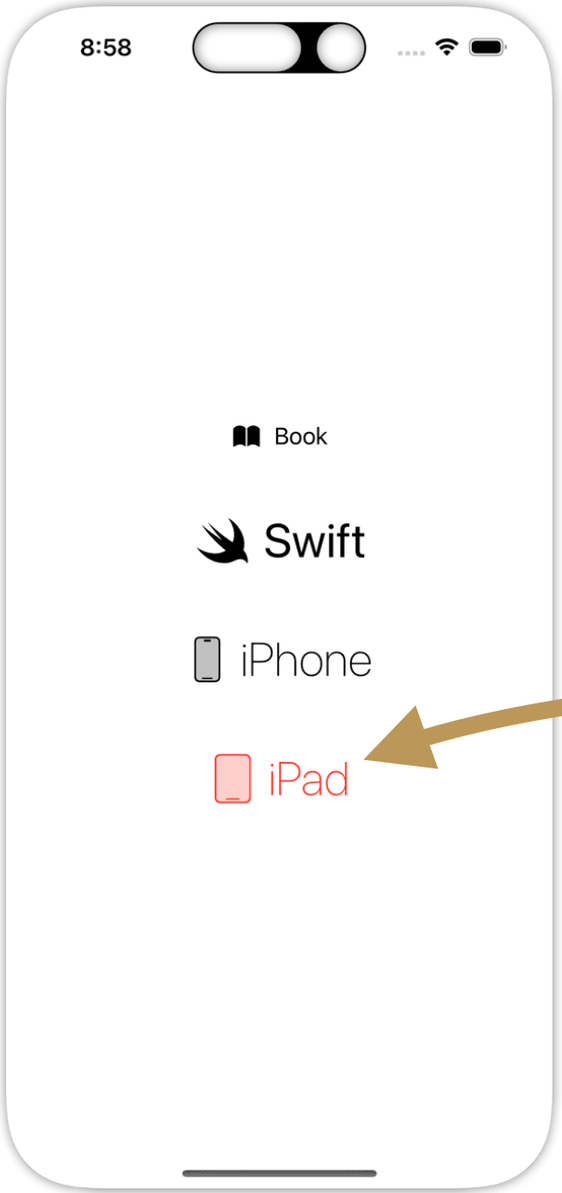
Use the **resizable** modifier to allow the image to fit within the parent.
To have it maintain its aspect ratio, you can use the **scaledToFit** modifier.

```
struct TheImageView_Resizing: View {
    var body: some View {
        Image(.book)
            .resizable()
            .scaledToFit()
    }
}
```

Label

Use the Label view to conveniently combine images and text.

Label: Symbols & Text




```
struct TheLabelView_SymbolAndText: View {
    var body: some View {
        VStack(spacing: 45.0) {
            Label("Book", systemImage: “book.fill")

            Label("Swift", systemImage: "swift")
                .font(.largeTitle)

            Label("iPhone", systemImage: "iphone")
                .font(.largeTitle.weight(.thin))

            Label("iPad", systemImage: "ipad")
                .font(.largeTitle.weight(.thin))
                .foregroundStyle(Color.red)
        }
    }
}
```

The Label view can combine text and a symbol (SF Symbol).

This is a pull-in view, it will only use as much space as the image and text require.

You can use modifiers to change the font style, weight, and color.

Label: Images & Text

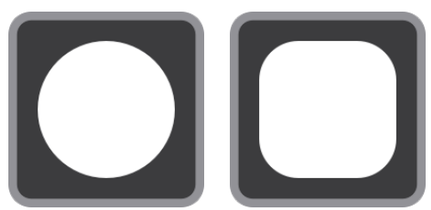

```
struct TheLabelView_ImagesAndText: View {
    var body: some View {
        VStack {
            Label("SwiftUI Simplified", image: .book)
        }
    }
}
```

```
Label {
    Text("SwiftUI Simplified")
} icon: {
    Image(.book)
        .resizable()
        .scaledToFit()
}
```

You can use the Label view with images in your asset folder too.

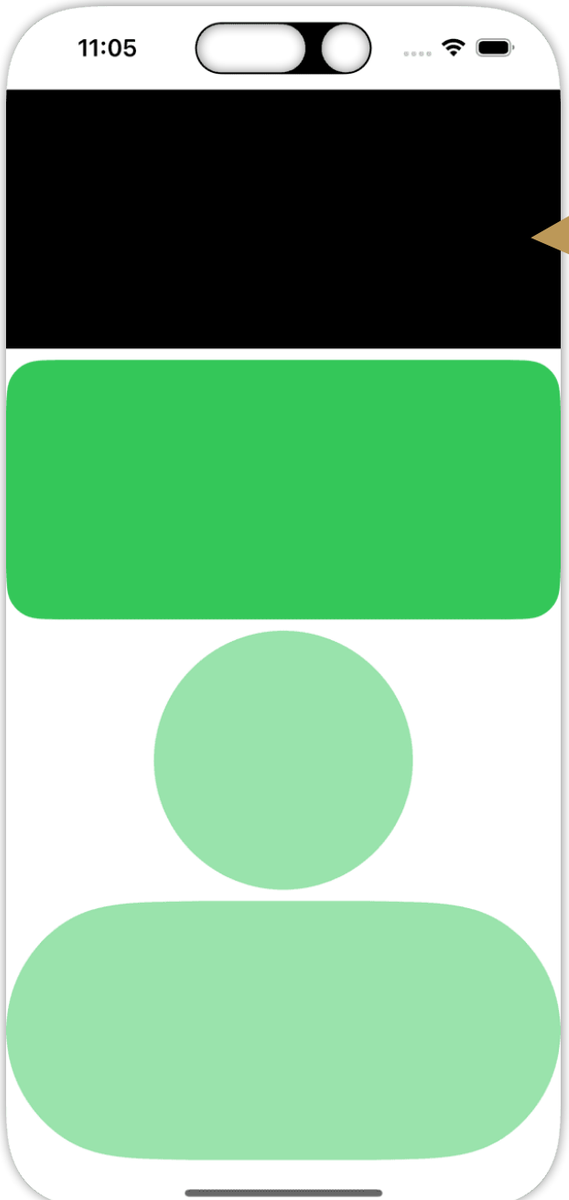
If the image is too large, you might have to use the Label's other initializer that allows you to customize the text and image separately.

Shapes



Use shapes to enhance your UI or individual views within your UI.

Shapes and Fill




```
struct Shapes: View {
    var body: some View {
        VStack {
            Rectangle()

            RoundedRectangle(cornerRadius: 25.0)
                .fill(Color.green)

            Circle()
                .fill(Color.green)
                .opacity(0.5)

            Capsule()
                .fill(Color.green.opacity(0.5))
        }
    }
}
```

You can create a variety of shapes in SwiftUI. If you do not specify a color, then the shape will be the same color as Text.

Shapes are push-out views and take up as much space as given.

You can also see they distribute space evenly with other shapes.

In this screen, they are all the same height.

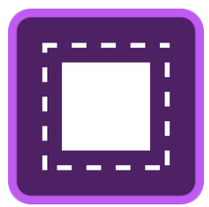
Use the **fill** modifier to change the color of a shape.

Use the **opacity** modifier to make the shape (and its color) less solid.

Opaque means “solid”. A value of 1 means completely solid (not transparent). Less than one makes it less solid.

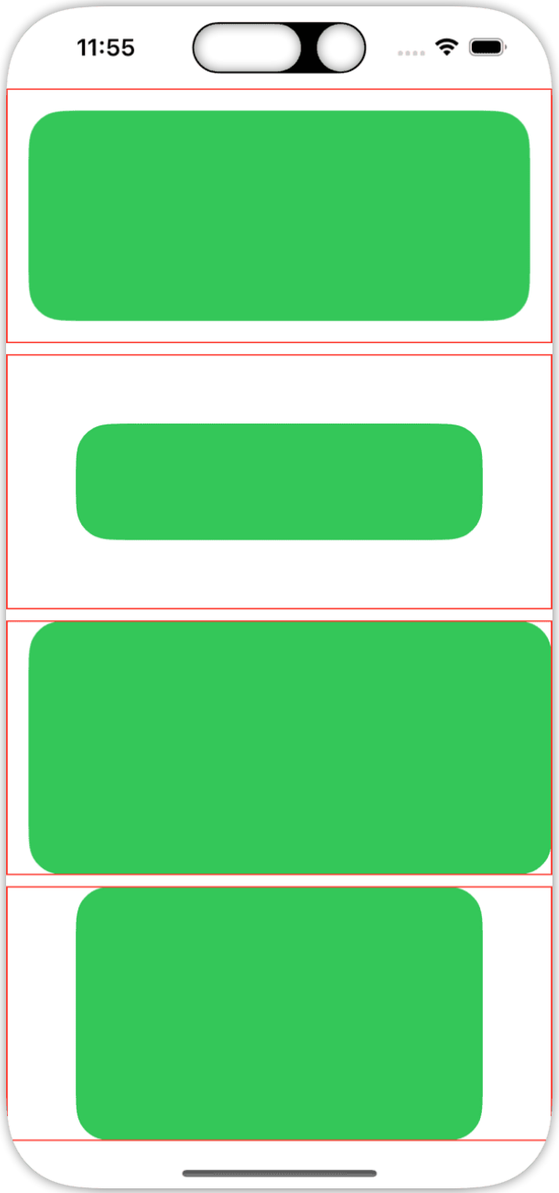
Optionally, you can add opacity directly to the color.

Padding



Use padding to add spacing around views.

Padding Views




```
struct ThePaddingModifier: View {
    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 25.0)
                .padding()
                .border(.red)

            RoundedRectangle(cornerRadius: 25.0)
                .padding(50)
                .border(.red)

            RoundedRectangle(cornerRadius: 25.0)
                .padding(.leading)
                .border(.red)

            RoundedRectangle(cornerRadius: 25.0)
                .padding(.horizontal, 50)
                .border(.red)
        }
        .foregroundStyle(.green)
    }
}
```

Padding adds space around a view and expands its frame.

It is easier to see the views frame by using the **border** modifier.

You can specify padding **size**.

You can also specify a **side** where the padding should be. (Other options: top, bottom, trailing.)

You can specify opposite sides for padding by using **horizontal** or **vertical**.

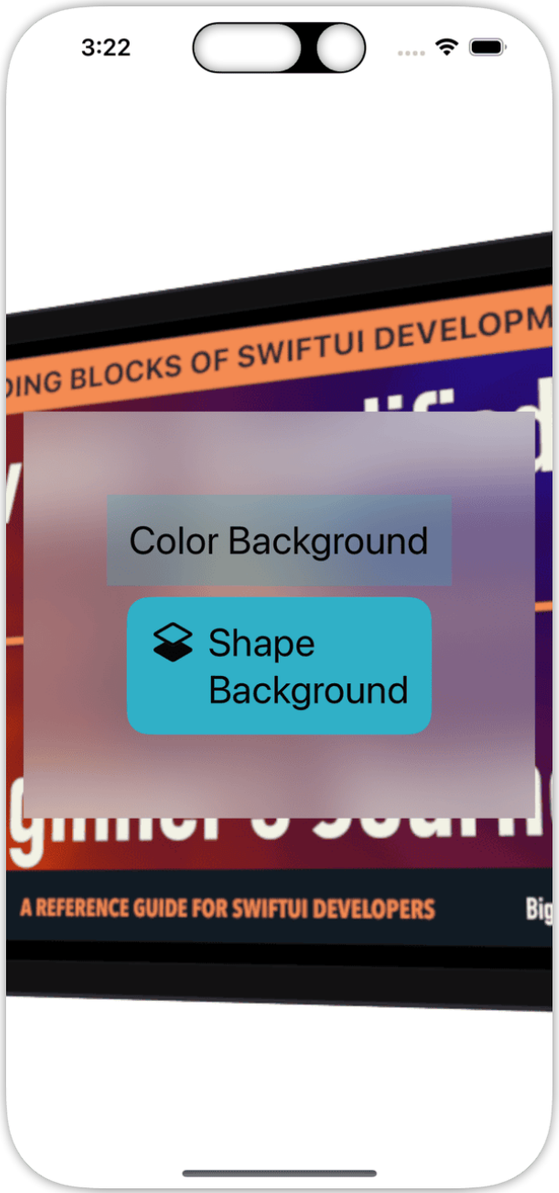
Note: If no fill modifier is used, a shape will use the foregroundStyle color.

Layers

You learned earlier that you can use a ZStack to layer views.

There is also another way to layer one view on top of or behind another view that stays within the bounds of that view.

Background Layer




```
struct Layer_Background: View {

    var body: some View {

        VStack {

            Text("Color Background")

                .padding()

                .background(Color.teal.opacity(0.25))


            Label("Shape Background", systemImage: "square.2.layers.3d.bottom.filled")

                .padding()

                .background(Color.teal, in: .rect(cornerRadius: 16))

        }

        .padding(60)

        .background(.ultraThinMaterial)

        .background(Image(.book))

        .font(.title)

    }

}
```

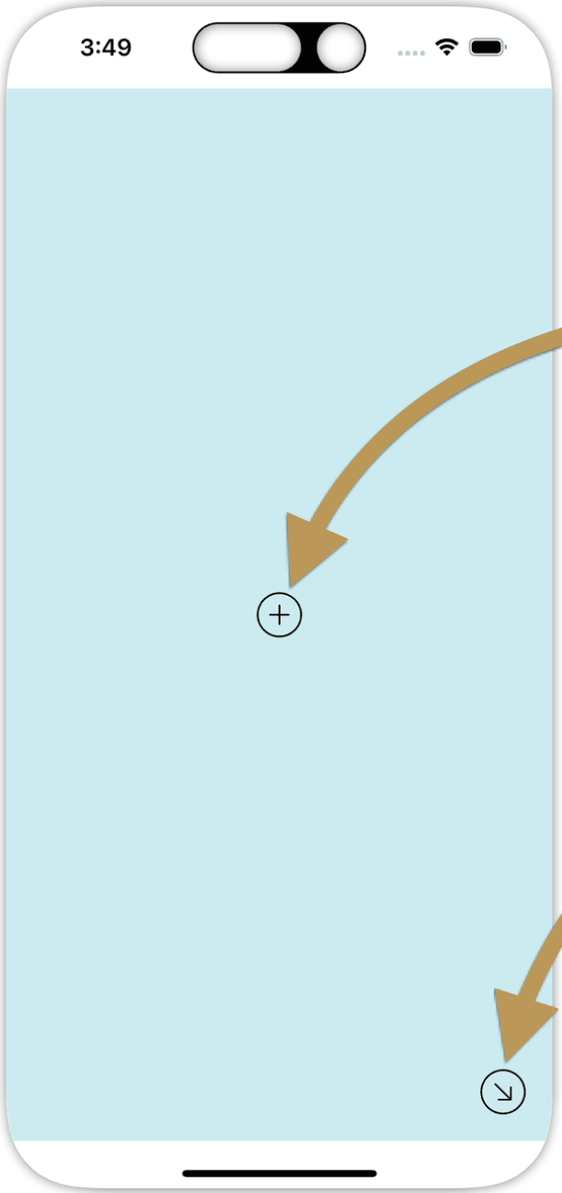
Use the **background** modifier to add views, colors, or shapes behind other views.

The background can include colors and shapes combined.

The .rect is short for “rectangle”. This is another way to create a shape.

You can layer multiple backgrounds. The first background is using what is called a “material” (blur). Materials have different thicknesses that control how transparent they are.

Overlay Layer




```
struct Layer_Overlay: View {

    var body: some View {

        Color.teal.opacity(0.25)

            .overlay(Image(systemName: "plus.circle"))

            .overlay(alignment: .bottomTrailing) {

                Image(systemName: "arrow.down.right.circle")

                    .padding()

            }

            .font(.largeTitle.weight(.thin))

    }

}
```

Overlay allows you to put views on top of another view.

By default, views are placed in the center.

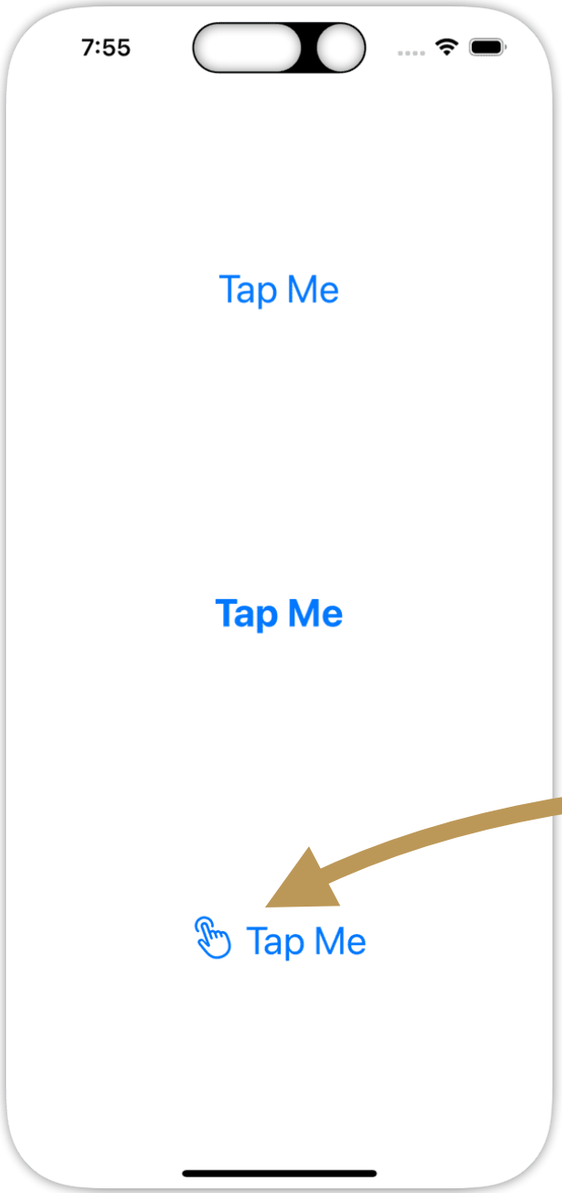
You can specify where you want the view on top to be positioned with the **alignment** parameter.

Note: Here you see the alignment operator gives you a closure to construct a view. The background parameter does too.

Button

Use a Button view to respond to a user's touch to perform some action.

Button: Ways to Create




```
struct TheButtonView_WaysToCreate: View {
    var body: some View {
        VStack(spacing: 200) {
            Button("Tap Me") {
                // Code
            }

            Button {
                // Code
            } label: {
                Text("Tap Me")
                    .fontWeight(.bold)
            }

            Button("Tap Me", systemImage: "hand.tap") {
                // Code
            }
        }
        .font(.title)
    }
}
```

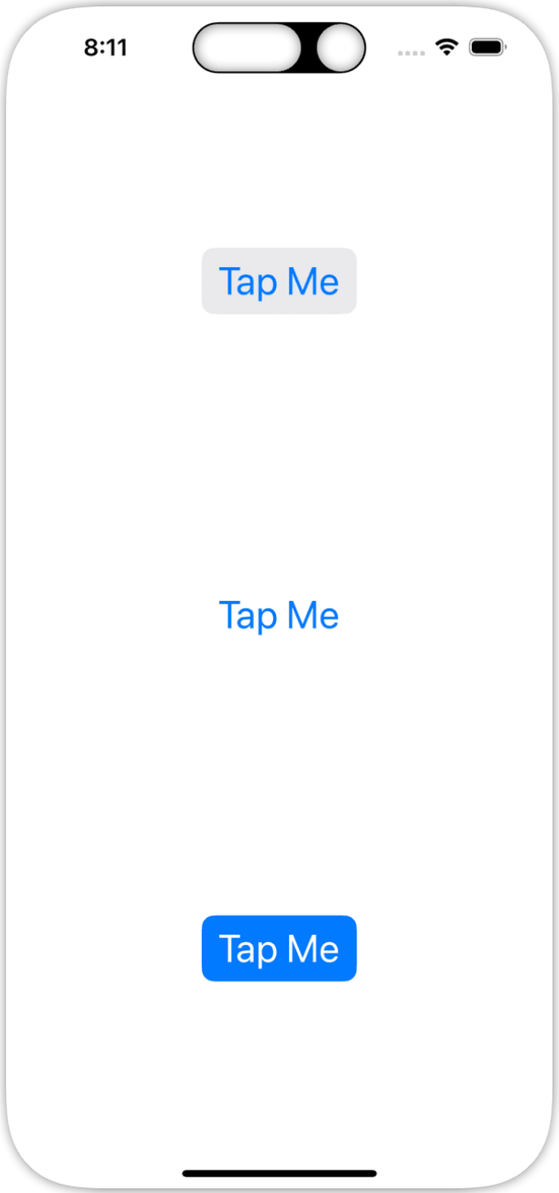
The minimum needed for a Button is just text.

You can add any view you want within a button by using the **label** closure.

There is a way to create buttons with a symbol too.

Note: The font modifier also works on button text.

Button: Ways to Style




```
struct TheButtonView_WaysToStyle: View {

    var body: some View {

        VStack(spacing: 200.0) {

            Button("Tap Me") { }

                .buttonStyle(.bordered)


            Button("Tap Me") { }

                .buttonStyle(.borderless)


            Button("Tap Me") { }

                .buttonStyle(.borderedProminent)

        }

        .font(.title)

    }

}
```

You can make a button more noticeable by applying different **styles** to it.

?

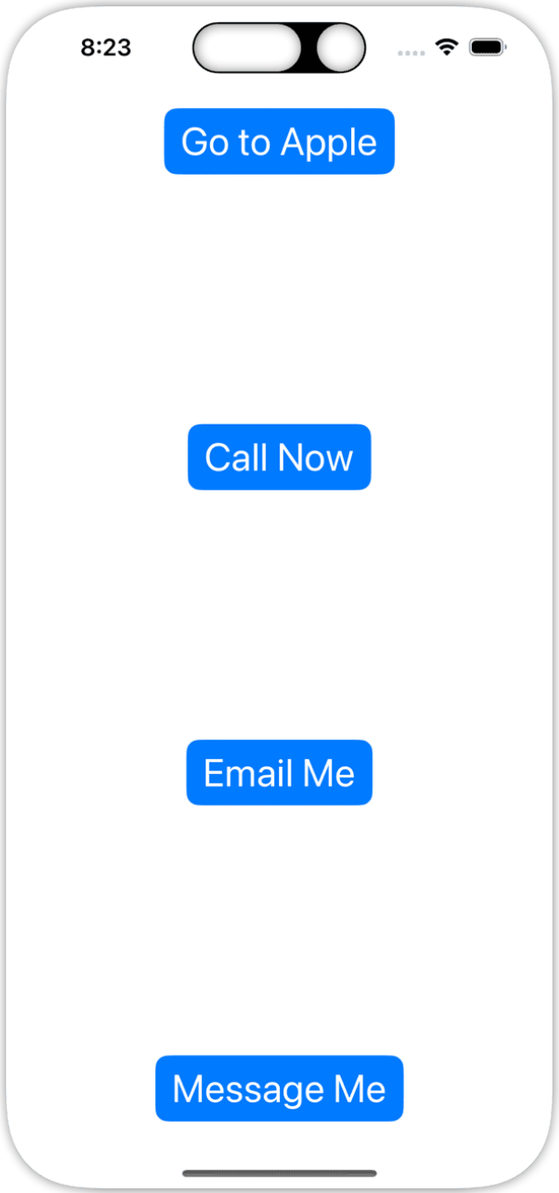
Can I specify the button style on a parent view and have it applied to all buttons within it?
Yes! If you use the buttonStyle modifier on the VStack, all buttons will adopt that button style (unless there is a button style specified on the button itself to override it.

Link

The Link is very similar to Button.

Use link to navigate to web pages, make phone calls, send text messages, or to email.

The Link View




```
struct TheLinkView: View {

    var body: some View {

        VStack(spacing: 180.0) {

            Link("Go to Apple", destination: URL(string: "https://www.apple.com")!)


            Link("Call Now", destination: URL(string: "tel:8005551234")!)


            Link("Email Me", destination: URL(string: "mailto:myemail@swiftui.com")!)


            Link("Message Me", destination: URL(string: "sms:+18885551212&body=Hello")!)

        }

        .buttonStyle(.borderedProminent)

        .font(.title)

    }

}
```

A Link view works like a Button view but it has a **destination** parameter where you can use a URL to perform different actions.

Note: You can apply button styles and font styles to Link views.

List

Use the List view to easily display a sequence of data, such as data held in an array.

List: Displaying Array


```
struct TheListView_Introduction: View {

    let lines = ["Line 1", "Line 2", "Line 3", "Line 4", "Line 5"]


    var body: some View {

        List(lines, id: \.self) { line in

            Text(line)

                    .padding()

        }

         .font(.title)

    }

}
```

You can pass in an array of data (`lines`) and the List view will give you a closure with a reference to each item (`line`).

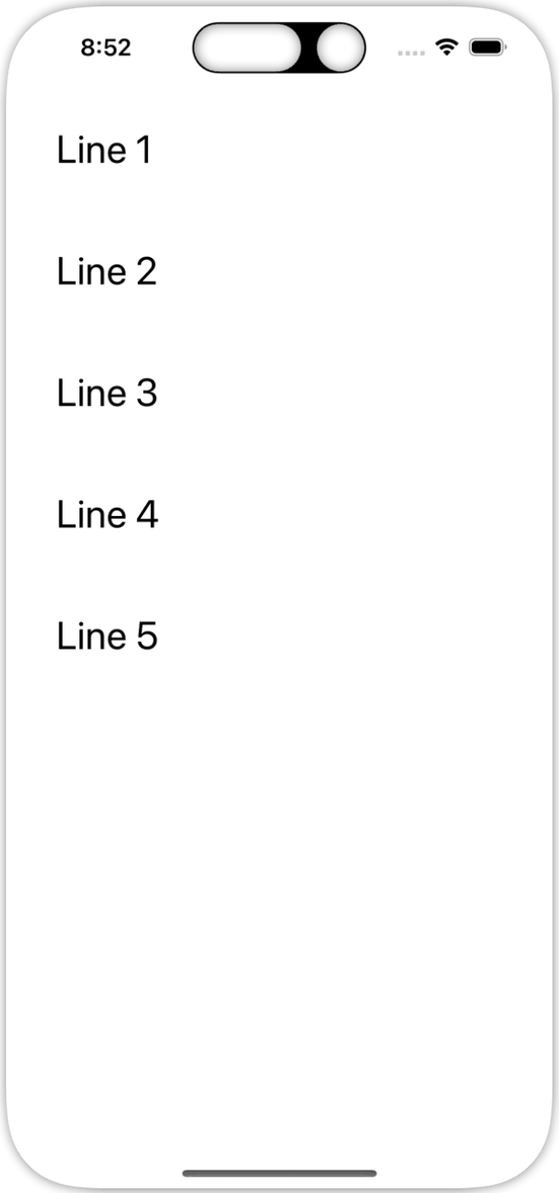
The **id** parameter helps SwiftUI uniquely identify each element in the array so it can properly manage it.

In this example, you are telling the list to use the value itself as the id.

The backslash (`\`) is like a shortcut to access a property. In this example, it is accessing the `self` property of each line.

Note: You can add a font modifier to the List and it will be applied to all Text views generated within it.

List: Style



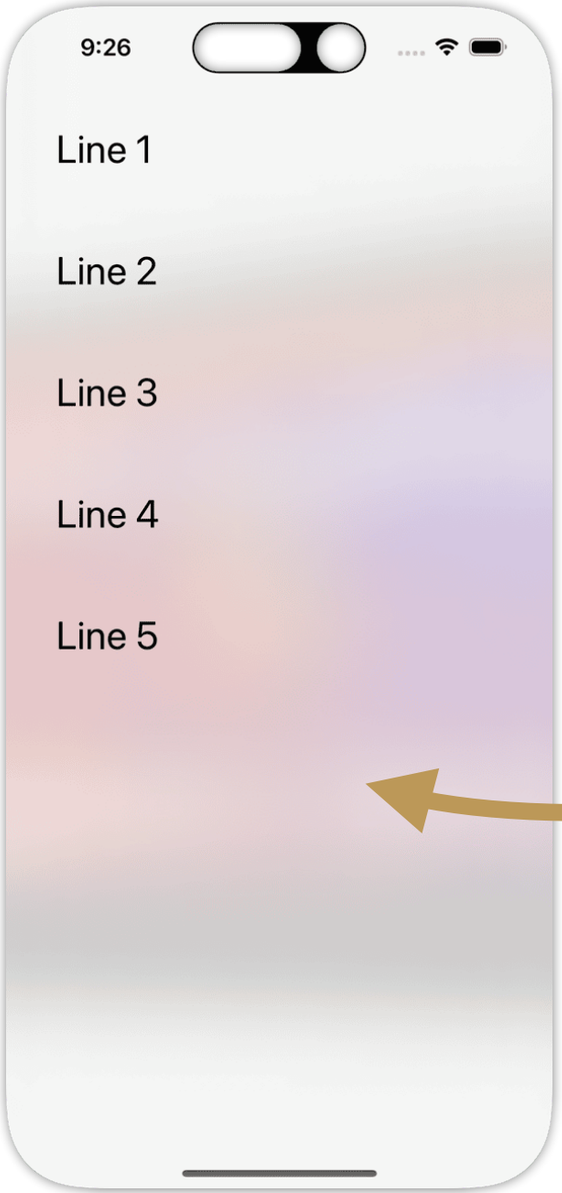

```
struct TheListView_ListStyle: View {

    let lines = ["Line 1", "Line 2", "Line 3", "Line 4", "Line 5"]


    var body: some View {

        List(lines, id: \.self) { line in

            Text(line)

                .padding()

                .listRowSeparator(.hidden)

        }

        .listStyle(.plain)

        .font(.title)

    }

}
```

You might not want the default style so you can remove it with the **listStyle** modifier.

The **plain** list style removed the gray background and the padding around the outside of the rows.

To remove the lines, you have to use the **listRowSeparator** modifier INSIDE the List.

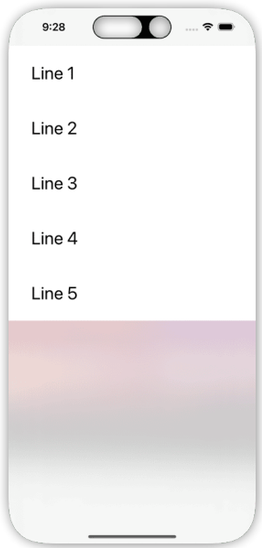
List: Background




```
struct TheListView_Background: View {

    let lines = ["Line 1", "Line 2", "Line 3", "Line 4", "Line 5"]


    var body: some View {

        List(lines, id: \.self) { line in

            Text(line)

                .padding()

                .listRowSeparator(.hidden)

                .listRowBackground(Color.clear)

        }

        .listStyle(.plain)

        .background(.regularMaterial)

        .background(Image(.book))

        .font(.title)

    }

}
```

To provide a background that the user can see, you need to make the row background color clear.

If you do **not**, it will look like this:



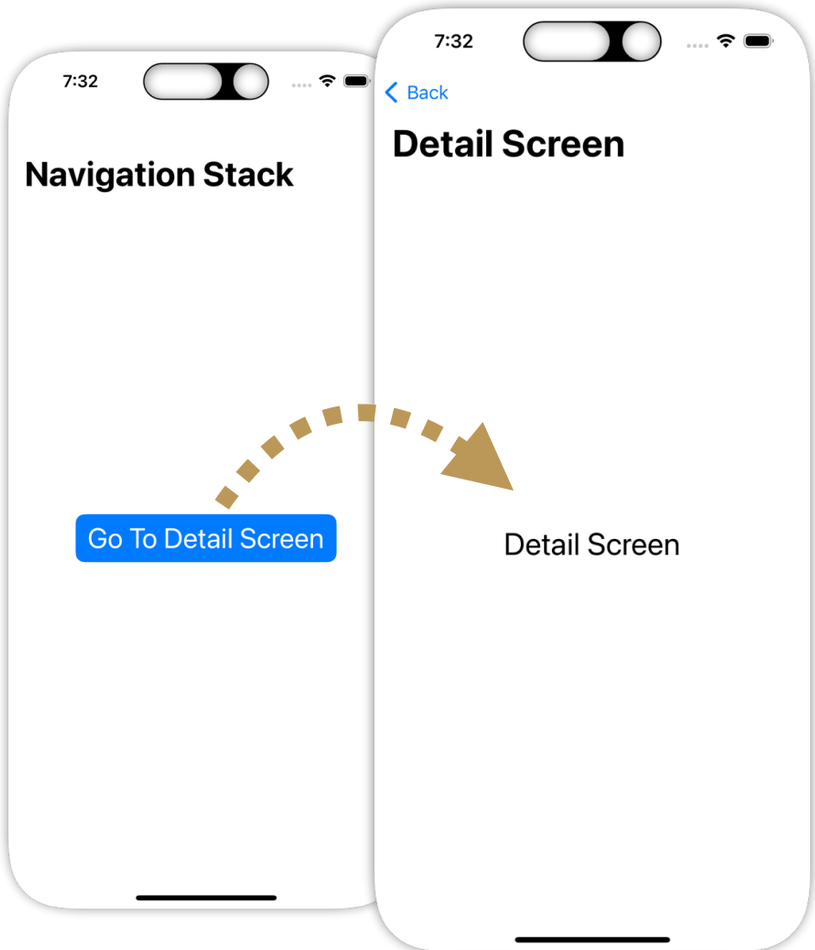
Note: To make the background show, you have to change the list style to plain.

NavigationStack

Use the NavigationStack to go from one screen to another.

NavigationStack: Concept

NavigationStack



A **NavigationStack** is a special container that allows you to navigate to other views.

It is like a stack of views.

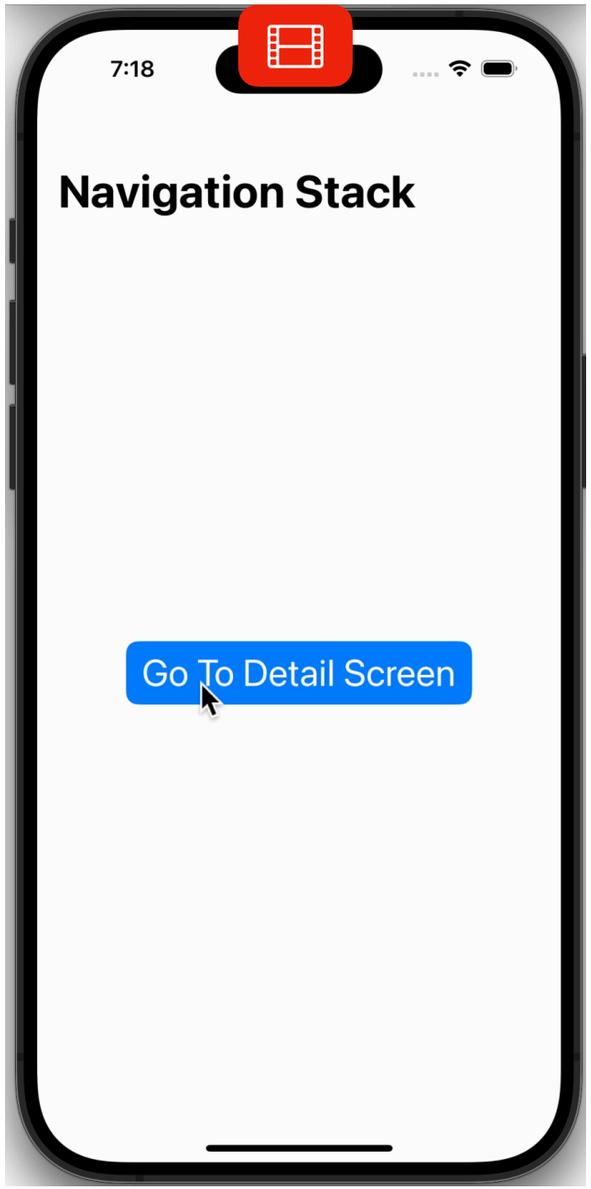
When you navigate to a new view, it adds that view to the **top** of the stack.

When you navigate back, it removes the view from the top.

Terms: Here are some common terms developers use with navigation:

- **Push** - You push a view to the top of the stack.
- **Pop** - You pop a view off the top of the stack.

NavigationStack: With NavigationLink




```
struct TheNavigationStack_WithNavigationLink: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink("Go To Detail Screen") {
                    DetailView()
                }
                .buttonStyle(.borderedProminent)
            }
            .font(.title)
            .navigationTitle("Navigation Stack")
        }
    }
}
```

```
struct DetailView: View {
    var body: some View {
        VStack {
            Text("Detail Screen")
        }
        .font(.title)
        .navigationTitle("Detail Screen")
    }
}
```

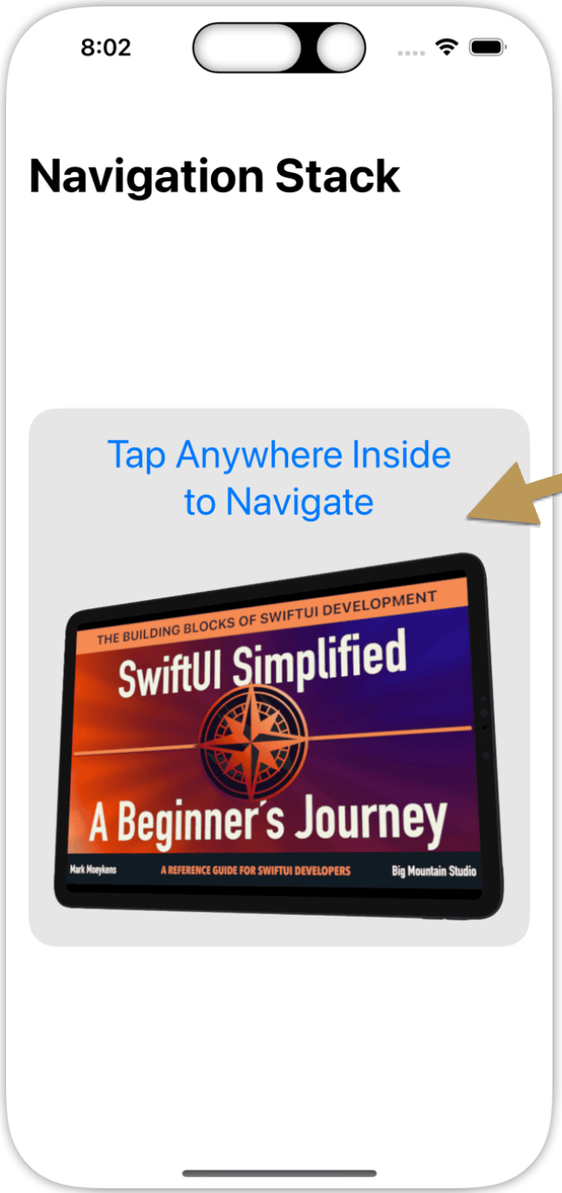
You navigate using the **NavigationLink** view. The NavigationLink only works when within a NavigationStack.

Note: You can add button styles to a NavigationLink.

Use the **navigationTitle** modifier to put text in the navigation bar at the top.

Warning: The **navigationTitle** modifier has to go on the view WITHIN the NavigationStack, not on the NavigationStack itself.

NavigationLink: Options



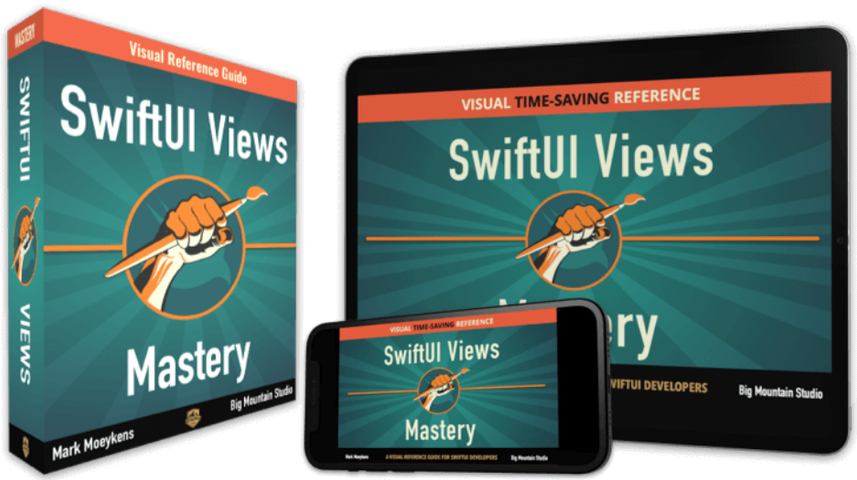

```
struct NavigationStack_NavigationLinkOptions: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink {
                    DetailView()
                } label: {
                    VStack {
                        Text("Tap Anywhere Inside to Navigate")
                        Image(.book)
                            .resizable()
                        .scaledToFit()
                    }
                    .padding()
                    .background(Color.black.opacity(0.1), in: .rect(cornerRadius: 20))
                }
                .padding()
            }
            .font(.title)
            .navigationTitle("Navigation Stack")
        }
    }
}
```

The NavigationLink view has another option where you can provide any view you want instead of just text.

Tap on any view within the label closure to navigate.

SwiftUI Views Mastery

LEARN MORE WITH THE COMPLETE, VISUAL TIME-SAVING REFERENCE



- Over **1,000** pages of SwiftUI
- Over **700** screenshots/videos showing you what you can do so you can quickly come back and reference the code
- Learn all the ways to work with and modify images
- See the many ways you can use color as views
- Discover the different gradients and how you can apply them
- Find out how to implement action sheets, modals, popovers and custom popups
- Master all the layout modifiers including background and overlay layers, scaling, offsets padding and positioning
- How do you hide the status bar in SwiftUI? Find out!
- ***This is just the tip of the mountain!***

LEARN MORE AND SAVE 10% ON THIS BOOK!

ARCHITECTURE

Learn how to organize your files, manage view data, and pass view data to other screens.

Goal of Architecture

The whole goal of architecture in computer programming is:

To make development easier.

That's it.

There is no other goal.

Architecture is not complex, not scientific, and not hard to understand.

It is not something you have to follow exactly from other people's recommendations either.

You can evaluate other people's architecture and if it does not make your life easier, don't use it.

How should architecture help?

It should make your project:

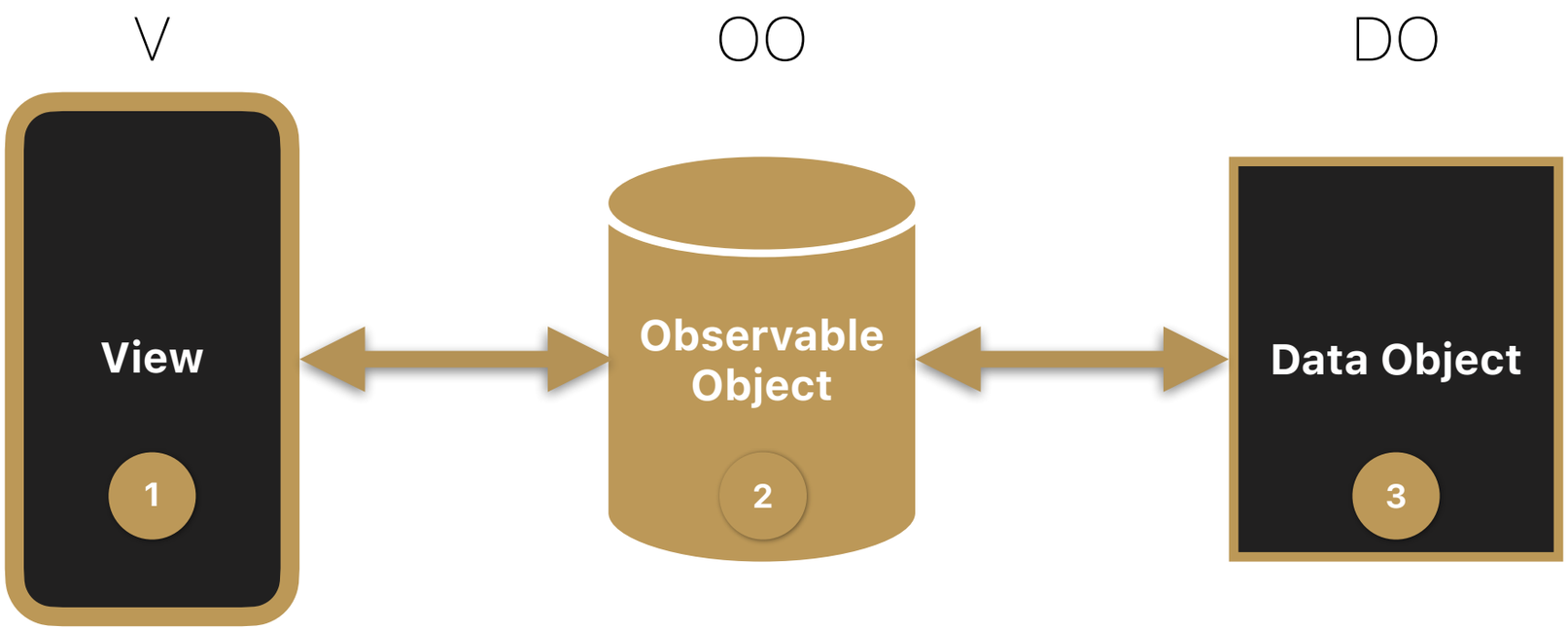
- Easier to code
- Easier to read your code
- Easier to find the code you are looking for
- Easier to predict where you want things to go
- Easier to make changes
- Easier to add new features
- Less complex
- Easier for others to understand your code (if you get help)
- Easier for YOU to understand YOUR code when you come back to it later
- Easier to test your code (if you wanted to write tests)

Now that you understand the goal, I will share with you one architecture that is specific to SwiftUI that is **simple**, **native**, **effective**, and **scalable**.

VOODO Architecture

The VOODO architecture is something I made up to avoid confusion with other architectures and rules and ideas one might already have with existing architectures.

It consists of 3 parts:



This is how Apple intended for SwiftUI apps to be built.

Let's look at each part and see how they all fit together and how you can make use of it!

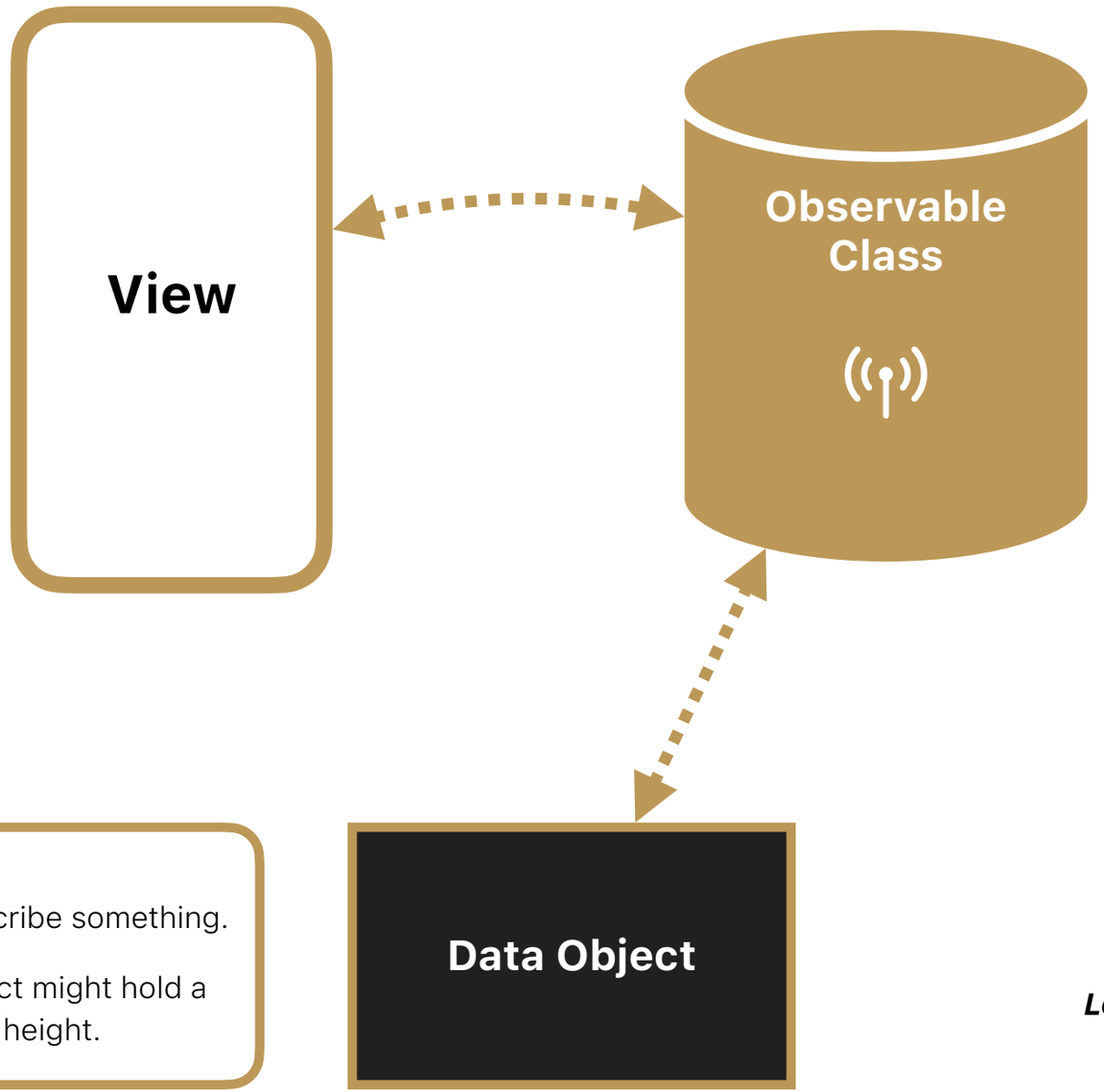
The 3 Main Parts

1

Views

SwiftUI screens are composed of views that are assembled together to present the user with something to look at and interact with.

All SwiftUI apps are going to have at least one view.



2

Observable Object

SwiftUI views can connect to what is called an observable class.

This class is special in that it can be "observed" by a view.

It alerts the view when its values change so the view can automatically update itself and show the new values.

These classes are easy to set up and use because Apple intended them to work directly with SwiftUI views.

3

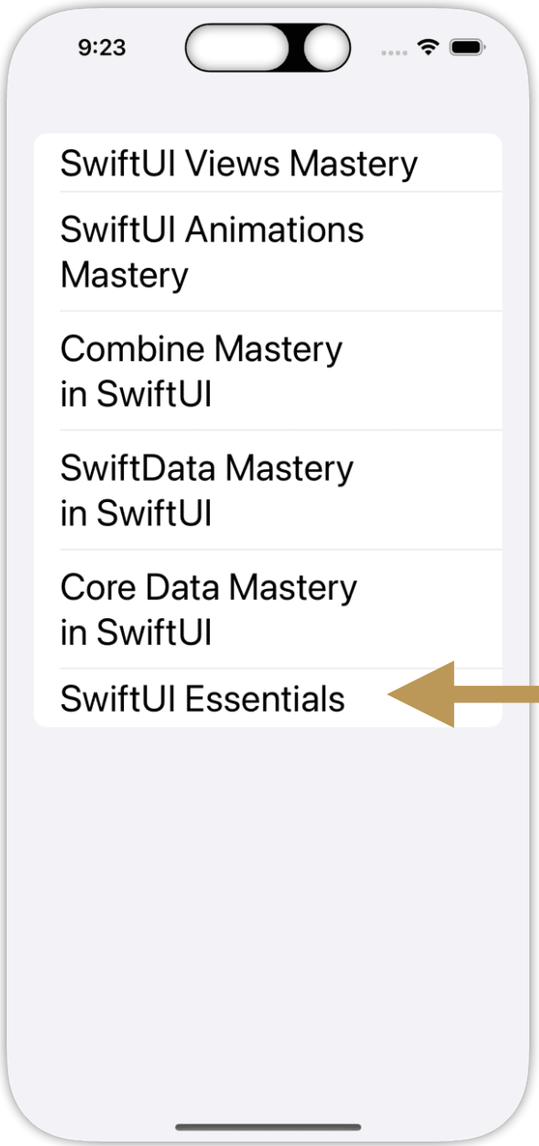
Data Objects

Data objects hold values that describe something.

For example, a person data object might hold a person's name, age, and height.

Let's look at examples of each part!

3. Data Object



I will explain the parts in reverse order so as not to overwhelm you.

If you want to build this screen, you might start with creating the data objects needed to represent the data seen here.

Start simple because you can always add more to your data object later.

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

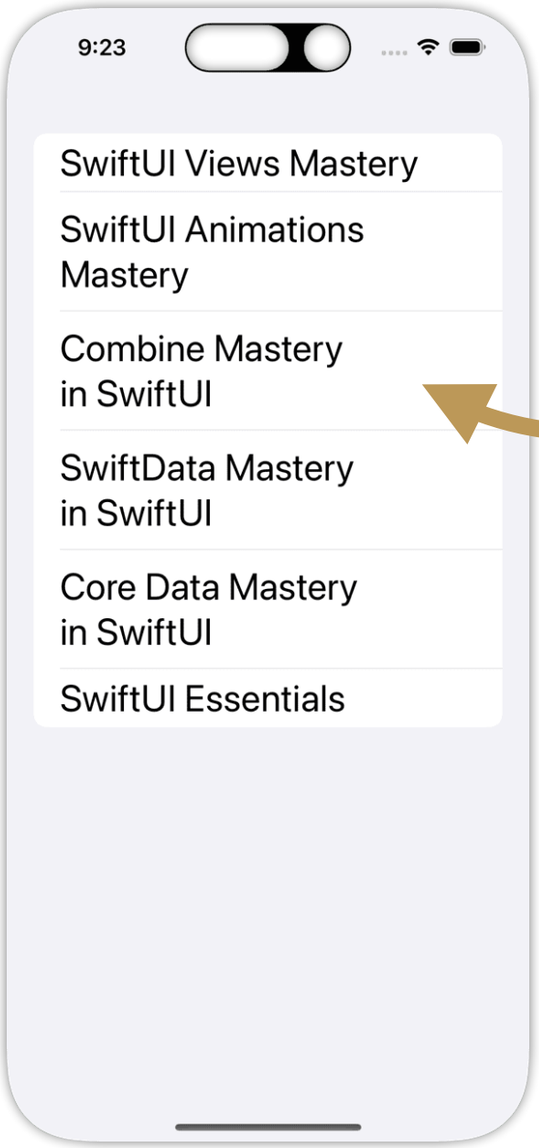
struct
You learned about structs earlier in the book when talking about views.
It is simply a way to group related properties and functions.
This will work great for a data object.

Identifiable
SwiftUI likes it when data objects have a unique identifier (id).
Identifiable is known as a *protocol,* which is a programmatic way to provide rules.
The Identifiable protocol just has one rule:
The struct must have an **id** property.
If you have many BookDO objects in an array, they will all be unique now.

The view will use these data objects to display data on the screen.

Note: Sometimes data objects are also called "data models" or simply "models".

2. Observable Object



The observable object is a great place for collecting your data and getting it ready to be presented in the view.

It will notify the view of data changes so the view knows to update itself.

```
@Observable
class BooksOO {
    var books: [BookDO] = []

    func fetch() {
        books = [BookDO(name: "SwiftUI Views Mastery"),
                 BookDO(name: "SwiftUI Animations Mastery"),
                 BookDO(name: "Combine Mastery in SwiftUI"),
                 BookDO(name: "SwiftData Mastery in SwiftUI"),
                 BookDO(name: "Core Data Mastery in SwiftUI"),
                 BookDO(name: "SwiftUI Essentials")]
    }
}
```

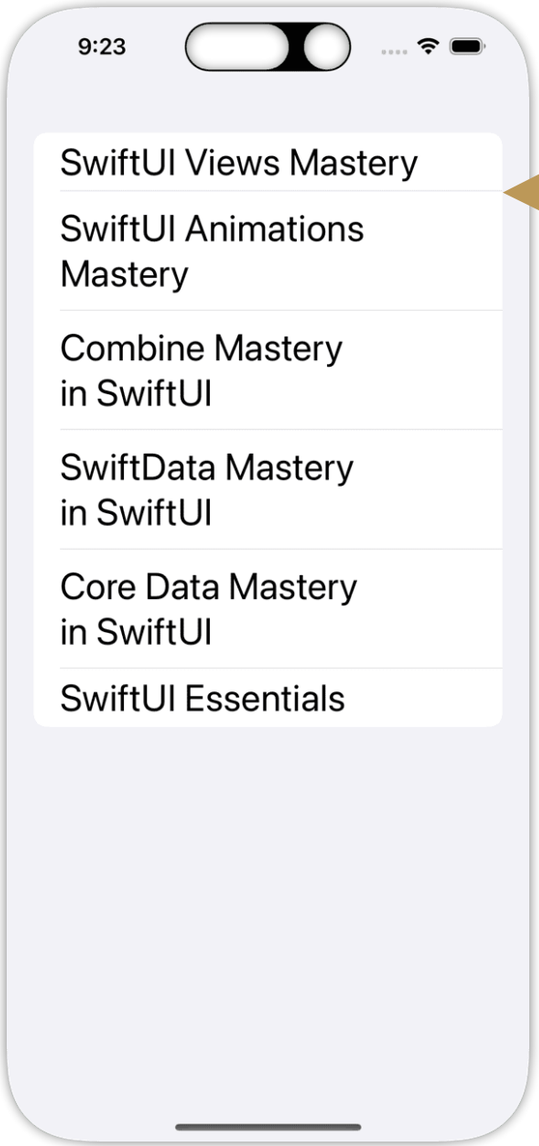
Using `@Observable` is a special macro that can let the view observe the public properties and get notified when they change.

Remember, a macro enables you to do a lot of things with one word.

The @Observable macro is adding code behind the scenes that will alert the view whenever the `books` property changes so the view updates automatically.

The `fetch` function gets all the data together and assigns it to the books property. When this happens, the SwiftUI view that uses it will automatically get redrawn to show the new data.

1. View




```
struct BooksView: View {
    @State private var oo = BooksOO()

    var body: some View {
        List(oo.books) { book in
            Text(book.name)
        }
        .font(.title)
        .onAppear {
            oo.fetch()
        }
    }
}

#Preview {
    BooksView()
}
```

The observable object the view is using (BooksOO) is assigned to a property marked with `@State`.

What is @State?
Normally, properties in a struct can be assigned a value but then cannot be changed (immutable).

The @State allows the property to be changed (mutable) and knows how to hold observable objects so when its properties change, the view knows to update itself.

This view can be recreated many times to update what is displayed. @State will preserve the property's value while the view is recreated so it does not reset.

The `onAppear` is a modifier where you can add any code you want to run when the view appears.
When this BooksView appears, the fetch function is called.

All Together

1 View

```
struct BooksView: View {
    @State private var oo = BooksOO()

    var body: some View {
        List(oo.books) { book in
            Text(book.name)
        }
        .font(.title)
        .onAppear {
            oo.fetch()
        }
    }
}
```

2 Observable Object

```
@Observable
class BooksOO {
    var books: [BookDO] = []

    func fetch() {
        books = [BookDO(name: "SwiftUI Views Mastery"),
                 BookDO(name: "SwiftUI Animations Mastery"),
                 BookDO(name: "Combine Mastery in SwiftUI"),
                 BookDO(name: "SwiftData Mastery in SwiftUI"),
                 BookDO(name: "Core Data Mastery in SwiftUI"),
                 BookDO(name: "SwiftUI Essentials")]
    }
}
```

3 Data Object

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

Some of your VOODO will be this small or even smaller.

How you organize these parts in your project is up to you. You will get some ideas coming up.

Naming is hard!

Naming Conventions
For clarity, I used a naming convention as it relates to the VOODO architecture.

This is for demonstration in the book but for your own project, you can come up with a naming convention that makes it clear to you.

Always remember that architecture is for you to decide. You are in control.
It is to make YOUR life easier.
So do what makes your life easier and your project easier for you to follow and understand.

1 View

`BooksView`
`AccountView`
`DetailView`

2 Observable Object

`BooksViewOO`
`AccountViewOO`
`DetailViewOO`

3 Data Object

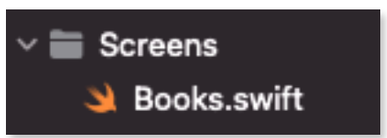
`BookDO`
`AccountDO`
`DetailDO`

VOODO File Organization

Here are some ideas for organizing your VOODO. Remember, there is no one right way. Choose one that will make your life easier.

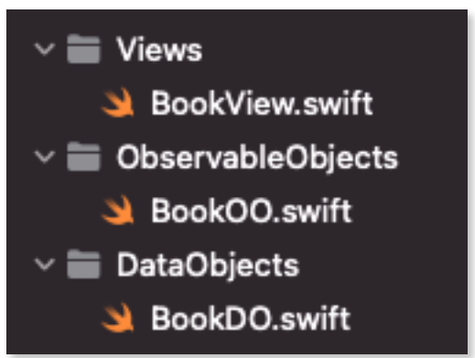
Single File

You could store all of the related parts in one file.



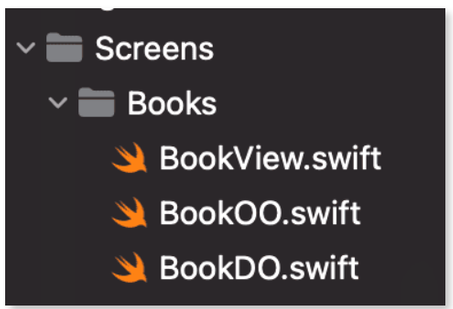
Separate Folders

You could create a folder for each part.



Separate but Same Folder

You could store all of the related parts in one folder.



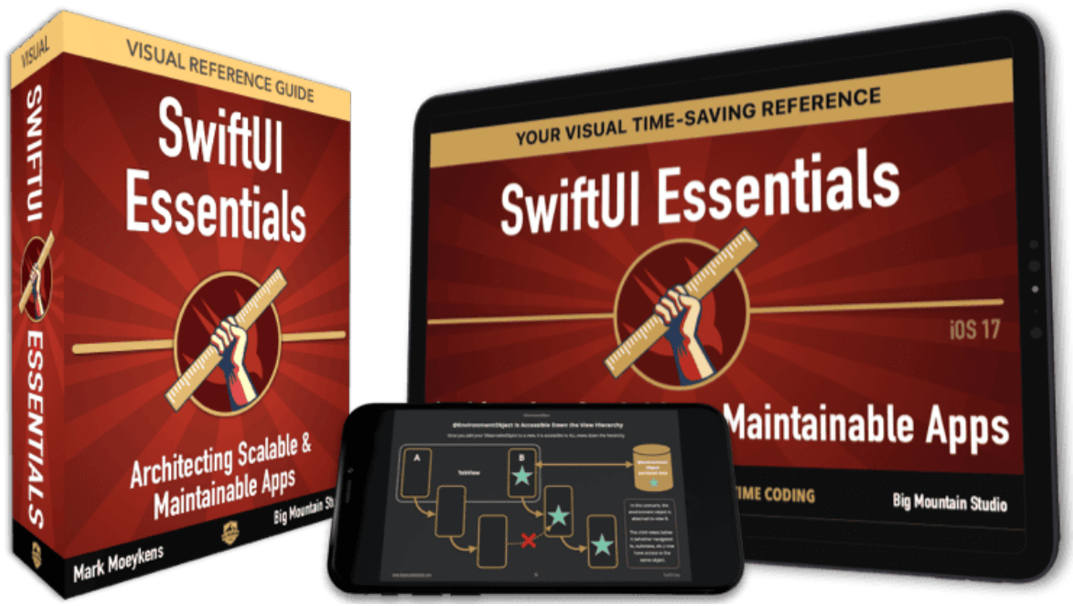
Choose whatever makes your life or your teams' lives easier.

You can choose one of these or none of these ideas. Remember, when it comes to architecture...
YOU ARE IN CONTROL! 😃

This is the way I personally prefer. 👆
I find when I work on one part, I usually have to work on the other parts and finding them is super easy this way.

SwiftUI Essentials

LEARN MORE ABOUT ARCHITECTING SCALABLE & MAINTAINABLE SWIFTUI APPS



Working with data in SwiftUI is super confusing. I know, I was there trying to sort it all out. That's why I made this simple to read book so that anyone can learn it.

- What is the difference between @Binding and @Bindable?
- How and when should you use @Environment?
- What is @StateObject and when should you use it?
- How is @State different from @StateObject?
- How can you have data update automatically from parent to child views?
- How can you work with a data model and still be able to preview your views while creating them?
- How do you persist data even after your app shuts down?
- Working with JSON
- How to work with input events to make your app even better

LEARN MORE AND SAVE 10% ON THIS BOOK!

ANIMATIONS

Give your UI life by animating views on the screen.

You will learn the basics of animation to give your UI subtle hints of being more alive.

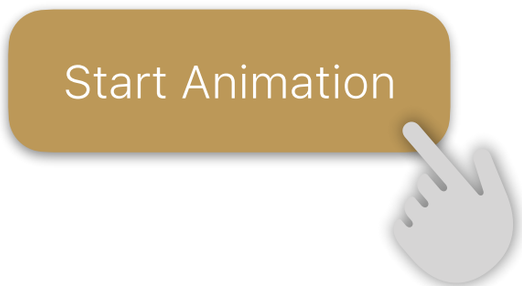
Concepts

Trigger

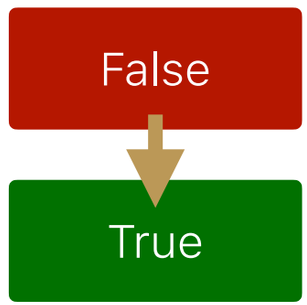
There needs to be a "trigger" that starts the animation.

The usual flow happens like this:

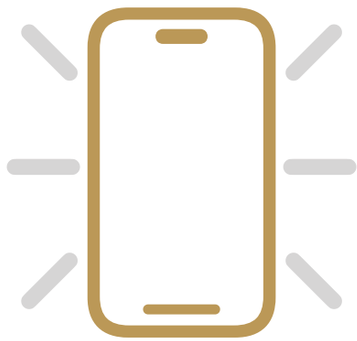
1. Trigger (button tap, onAppear, etc.)
2. Property is changed
3. Animation happens between the start state and end state



1. Trigger (event)



2. Property Change



3. Animation

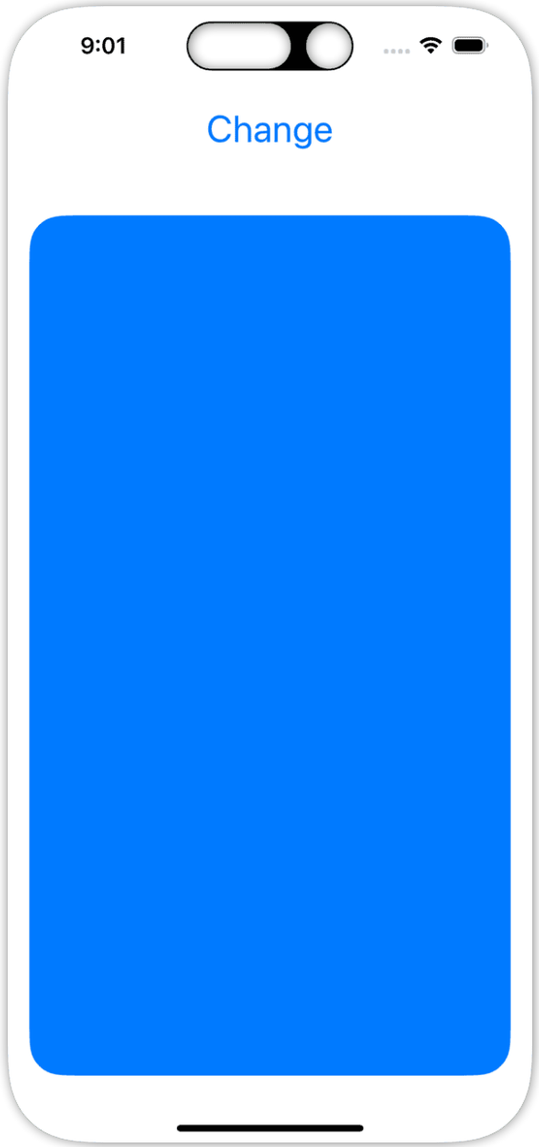
Change

For animation to happen, something about a view has to *change*.

The change could be the view's color, position, size, or some other property of the view.

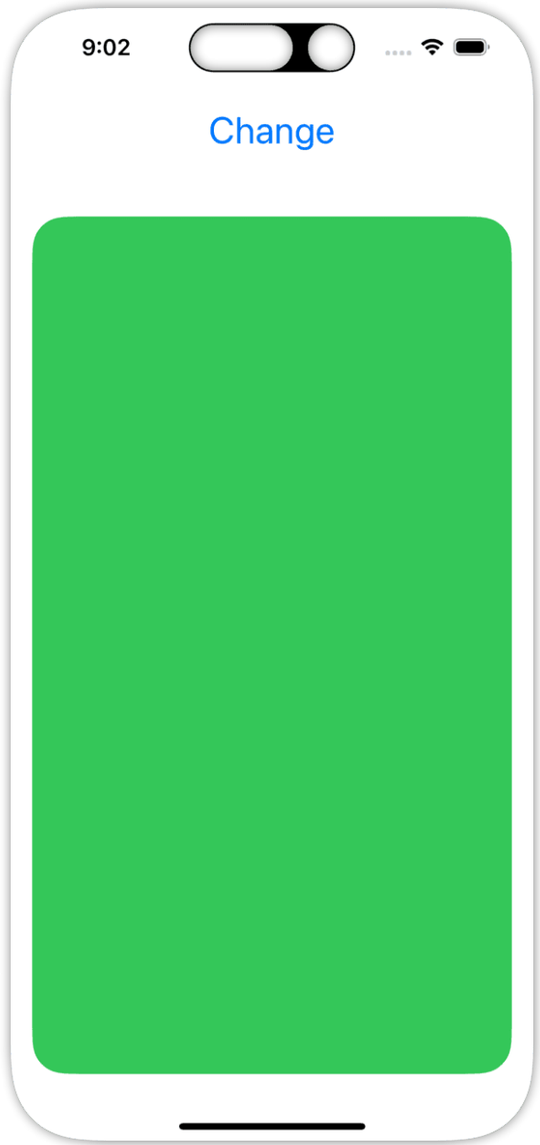
Start State *Animated Change* *End State*

Animation Introduction Example




```
struct Animation_Intro: View {
    @State private var change: Bool = false

    var body: some View {
        VStack(spacing: 20) {
            Button("Change") {
                change.toggle()
            }

            RoundedRectangle(cornerRadius: 25.0)
                .foregroundStyle(change ? .green : .blue)
                .animation(.default, value: change)
        }
        .padding()
        .font(.title)
    }
}
```

1. A button triggers a property change.
2. The property changes the color.
3. The animation modifier animates the change.



Code Deep Dive

There is a lot happening in this code so let's go over it more.

```
struct Animation_Intro: View {
    @State private var change: Bool = false

    var body: some View {
        VStack(spacing: 20) {
            Button("Change") {
                change.toggle()
            }

            RoundedRectangle(cornerRadius: 25.0)
                .foregroundStyle(change ? .green : .blue)
                .animation(.default, value: change)
        }
        .padding()
        .font(.title)
    }
}
```

1

What is "toggle()"?

When a property is a boolean it has a toggle function that allows you to switch between true and false.

This means you can hit the button repeatedly and it will continue to change between true and false.

2

Changing Values with the Ternary Operator

The change between green and blue is using what is called the "ternary operator" ("ternary" means "3 parts").
It follows this pattern:

`Condition ? True part : False part`

If `change` is true, the foregroundStyle uses the green color, else if `change` is false, it uses the blue color.

3

This SwiftUI animation modifier examines the begin and end state and figures out how to change those values incrementally over time for you so it looks animated.

1. The first parameter, **default**, says to use the default animation.
2. The second parameter, **value**, is where you specify what property is changing.

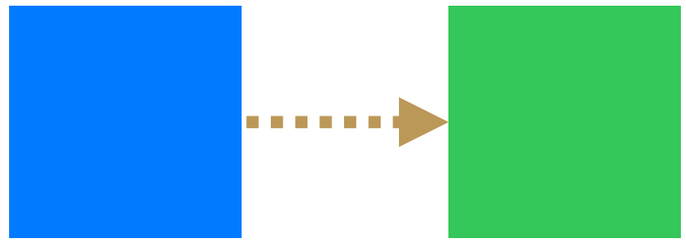
What can be animated in SwiftUI?

How do I know what can be animated?
As a general rule, almost every modifier that has a numeric value can be animated.

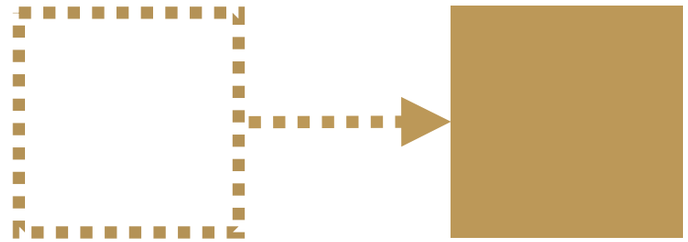
Examples:

- **Colors** (colors are composed of numbers that represent the red, blue and green values)
- **Positions** (could be using X and Y values or alignment)
- **Opacity** ("opacity" specifies the "solidness" of a view, the lower the opacity, the more transparent it becomes.)
- **Sizes** (how big or small a view changes to)
- **Padding** (changes in padding can be animated)
- **Text Size** (either through text styles or specific font sizes)

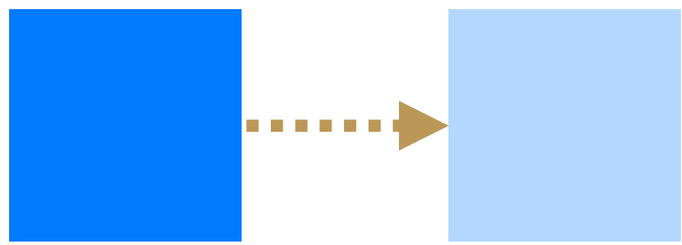
Colors



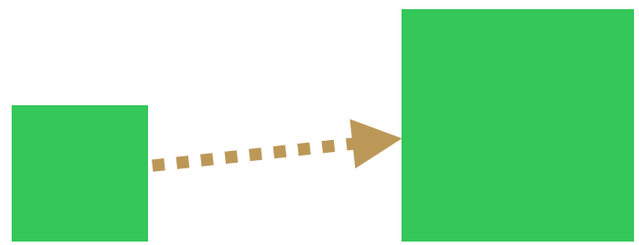
Positions



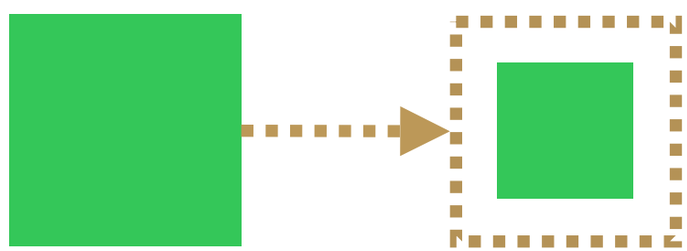
Opacity



Sizes



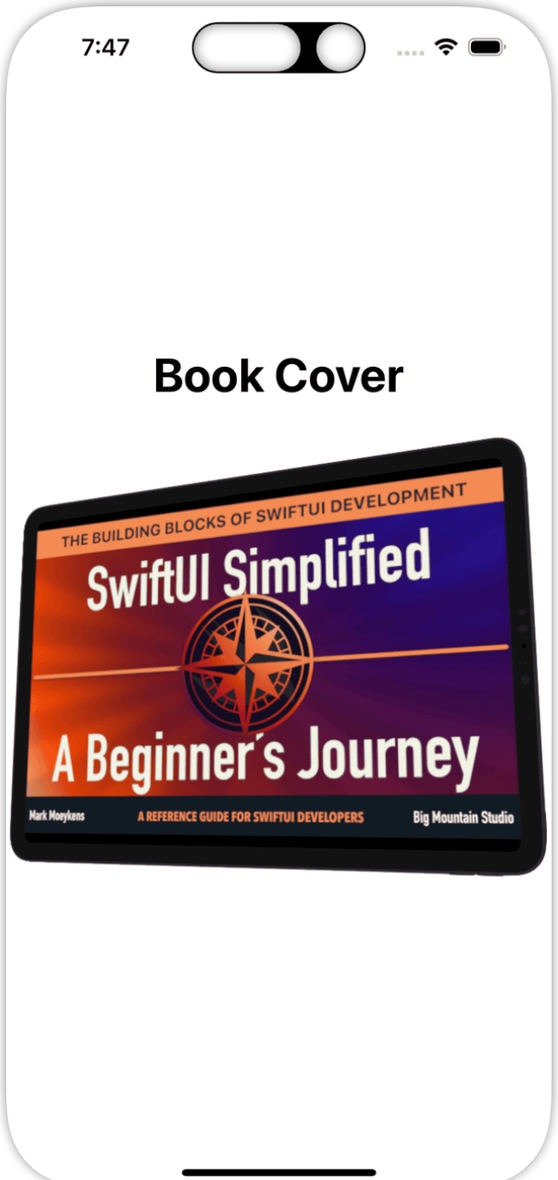
Padding



Text Size

Hello! Hello!

Opacity Example



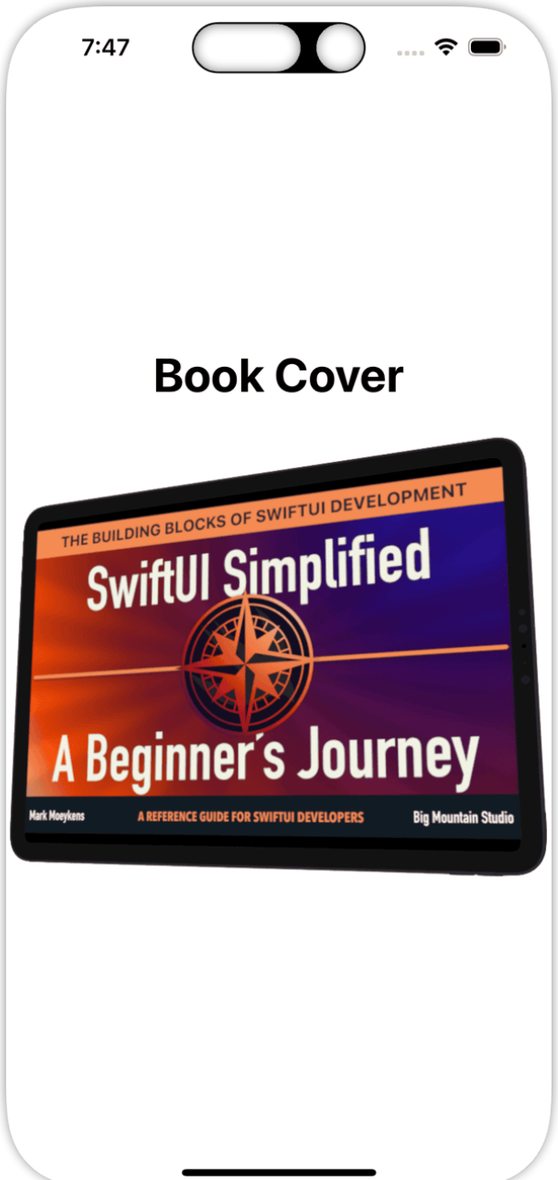

```
struct Animation_Opacity: View {
    @State private var animate = false

    var body: some View {
        VStack {
            Text("Book Cover")
                .font(.largeTitle.weight(.bold))

            Image(.book)
                .resizable()
                .scaledToFit()
                .opacity(animate ? 1 : 0)
                .animation(.default, value: animate)
        }
        .onAppear {
            animate = true
        }
    }
}
```

1
2
3

In this example, the book image fades in when the view appears.

1. The code in the **onAppear** modifier is run when the view appears.
2. The book image initially has the **opacity** set to zero which means it is invisible. But when the code in onAppear is run, the opacity changes to 1 (visible). This is what is being animated. The book will fade into view.
3. The animation modifier is on the Image so only the book image gets animated.

Animation Scope



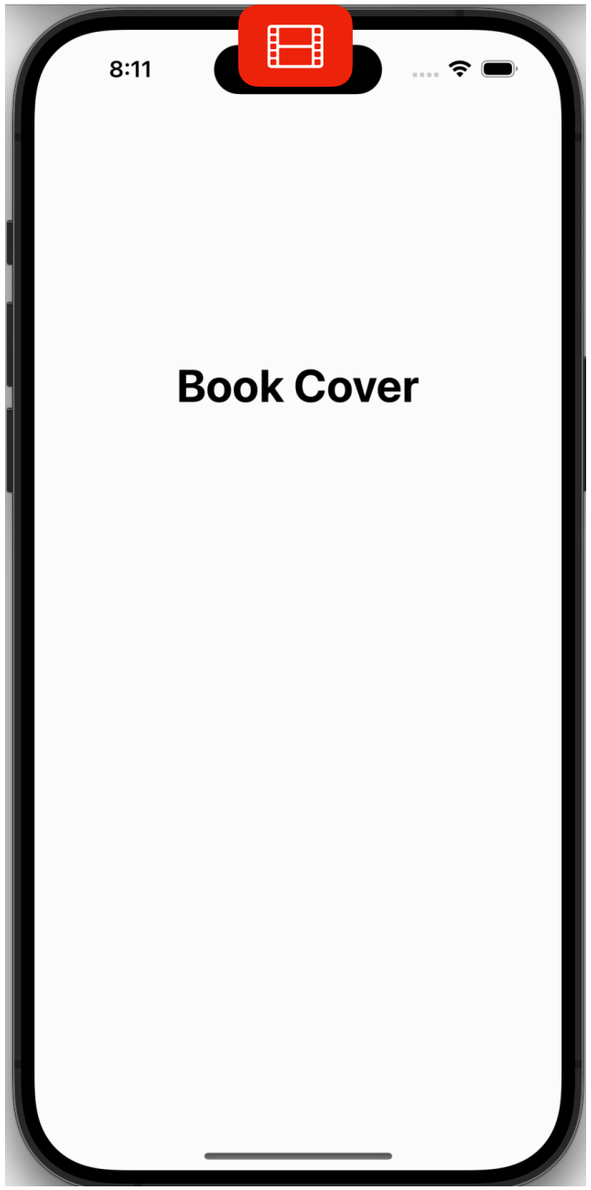

```
struct Animation_Scope: View {
    @State private var animate = false

    var body: some View {
        VStack {
            Text("Book Cover")
                .font(.largeTitle.weight(.bold))

            Image(.book)
                .resizable()
                .scaledToFit()
        }
        .opacity(animate ? 1 : 0)
        .animation(.default, value: animate)
        .onAppear {
            animate = true
        }
    }
}
```

You can change what gets animated by moving the animation modifier.

In the previous example, the animation modifier was on the Image view, so only the Image was animated.

In this example, the opacity and animation modifiers are moved to the VStack so now the VStack and everything within it will fade into view when onAppear runs.

Animation Delay



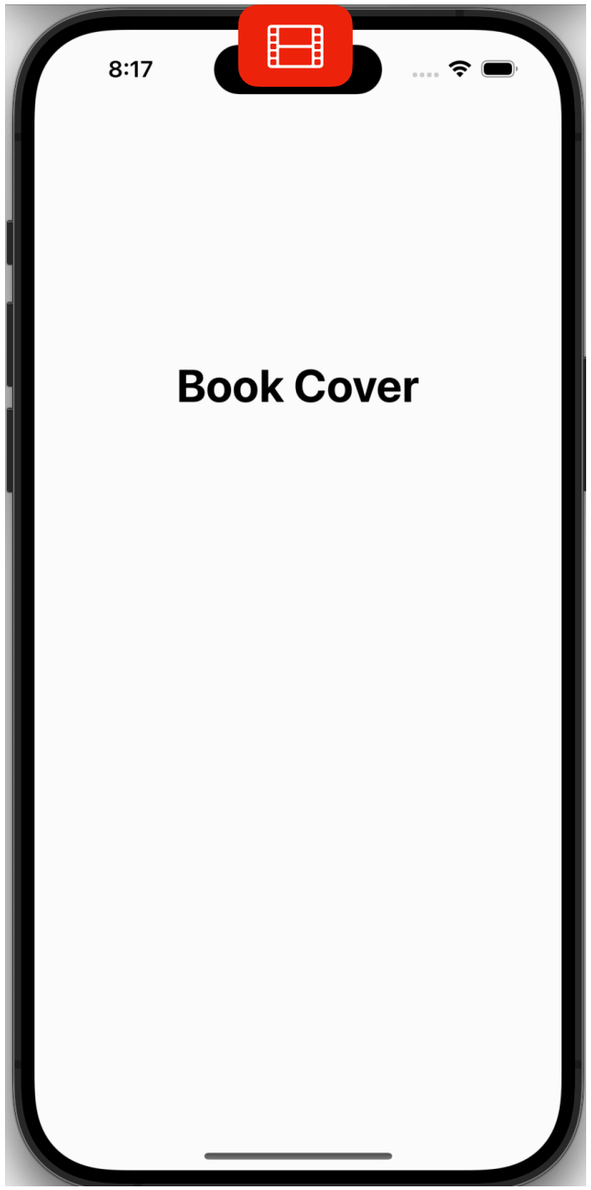

```
struct Animation_Delay: View {
    @State private var animate = false

    var body: some View {
        VStack {
            Text("Book Cover")
                .font(.largeTitle.weight(.bold))

            Image(.book)
                .resizable()
                .scaledToFit()
                .opacity(animate ? 1 : 0)
                .animation(.default.delay(1), value: animate)
        }
        .onAppear {
            animate = true
        }
    }
}
```

If you tested the examples on the previous pages, you may have noticed that the animations executes quickly and you might not have even seen the animation take place.

You can delay the start of the animation by attaching the **delay** method on the type of animation (`default`). You specify the number of seconds.

In this example, the delay is set to one second.

Offset Example



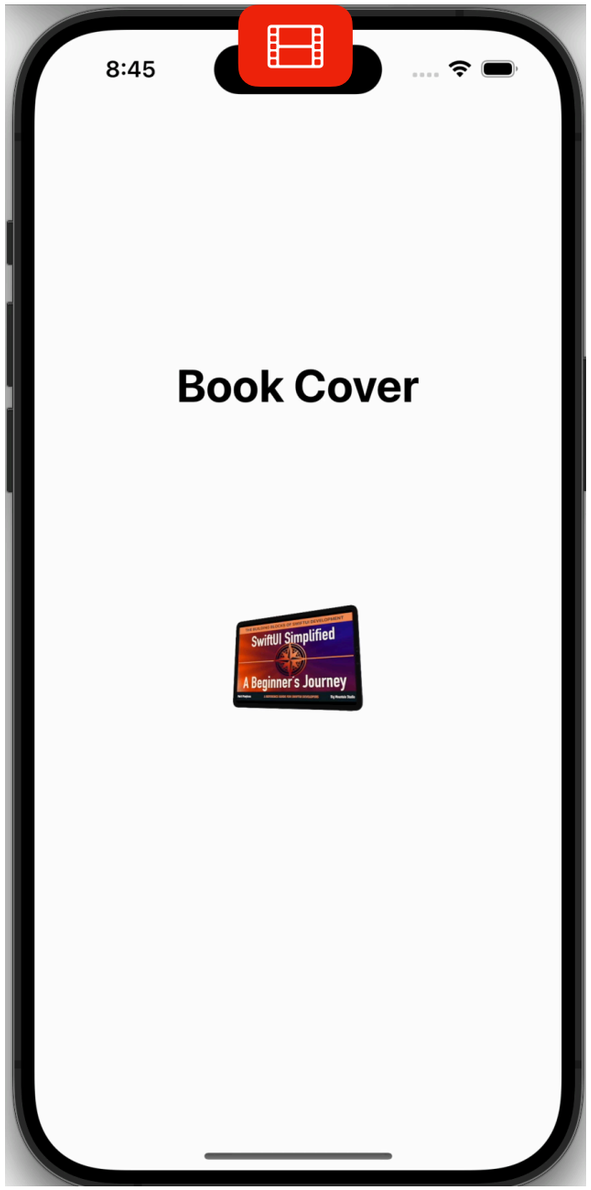

```
struct Animation_Offset: View {
    @State private var animate = false

    var body: some View {
        VStack {
            Text("Book Cover")
                .font(.largeTitle.weight(.bold))

            Image(.book)
                .resizable()
                .scaledToFit()
                .offset(x: animate ? 0 : 400, y: 0)
                .animation(.default.delay(1), value: animate)
        }
        .onAppear {
            animate = true
        }
    }
}
```

In this example, the Image is offset by 400 on the x axis so it is off the screen.

When animate is set to true, it will slide in from the right side into view.

Note: The **offset** modifier allows you to reposition a view on the x (right and left) and y (up and down) axes.

Scaling (Size) Example




```
struct Animation_Scale: View {
    @State private var animate = false

    var body: some View {
        VStack {
            Text("Book Cover")
                .font(.largeTitle.weight(.bold))

            Image(.book)
                .resizable()
                .scaledToFit()
                .scaleEffect(animate ? 1 : 0.25)
                .animation(.default.delay(1), value: animate)
        }
        .onAppear {
            animate = true
        }
    }
}
```

In this example, the **scaleEffect** modifier initially sets the Image view to be 25% of its original size (0.25).

When animate is set to true, the Image will scale or change to 100% of its original size (1).

Note: The **scaleEffect** modifier lets you change the size of a view. The default scale is 1 or 100%. Increasing or decreasing this number will make the view larger or smaller.

SwiftUI Animations Mastery

DO YOU LIKE ANIMATIONS? WOULD YOU LIKE TO SEE HUNDREDS OF VIDEO ANIMATION EXAMPLES WITH THE CODE?

SwiftUI made animations super easy...except when it isn't. Most new SwiftUI developers can get a simple animation working but as soon as they want to customize it, they get lost and waste more time than they should trying to get it working. This book will help you with that struggle.

- ✓ Learn all the animation types and options with embedded video samples and code
- ✓ Master spring animations
- ✓ Master transitions for views that are inserted and removed from the screen
- ✓ Learn how matchedGeometryReader should really work
- ✓ Customize animations with speeds, delays, and durations
- ✓ Understand how to animate changes from one view to another, such as from list to detail

LEARN MORE AND SAVE 10% ON THIS BOOK!

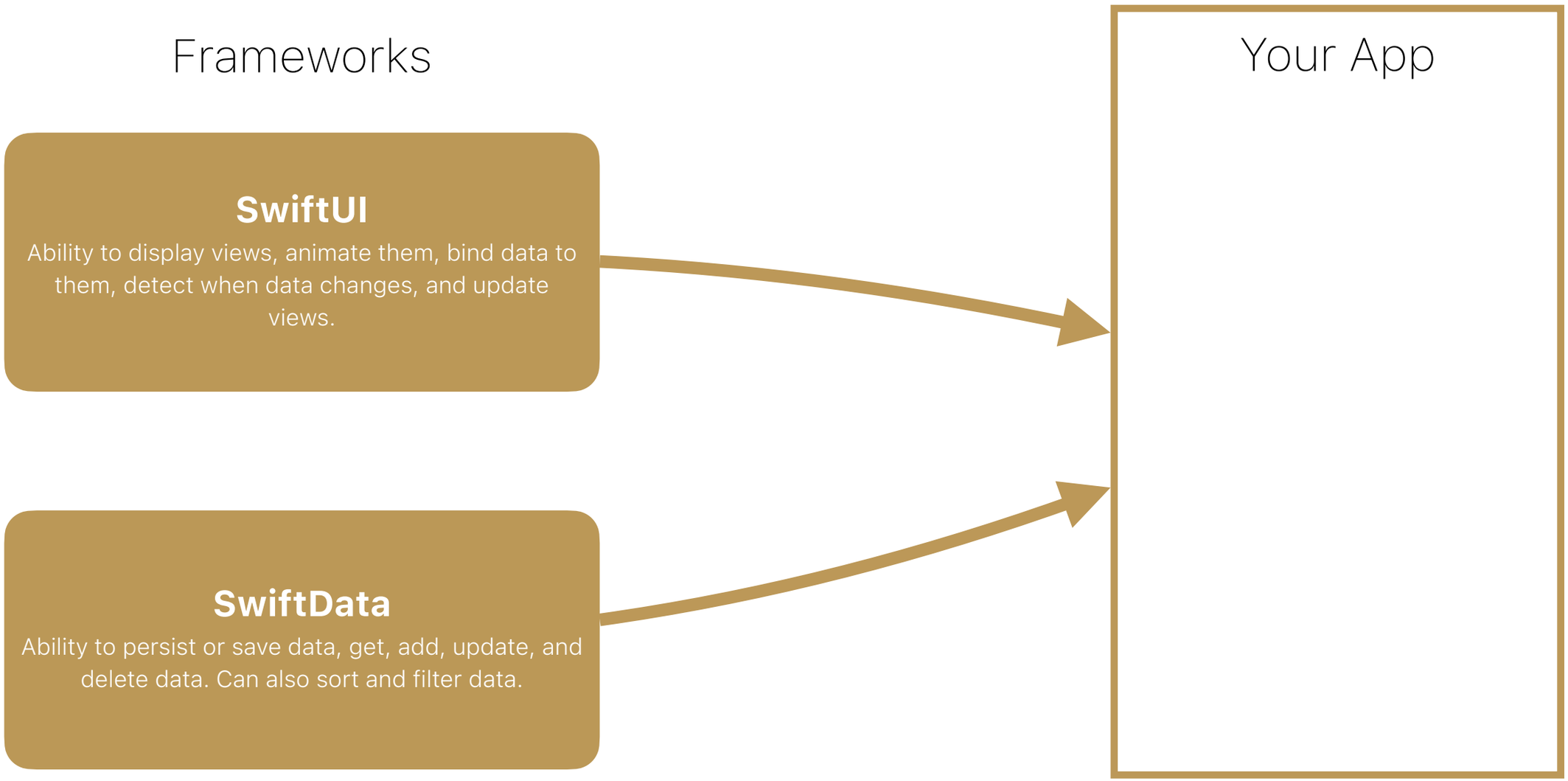
SWIFTDATA

Many apps work with data in some way.

In this chapter, you will get an introduction to how SwiftUI works with and displays data by using what is called SwiftData.

What is SwiftData?

SwiftData is a framework you can add to your app so your app can work with data.
A “framework” is like a book of functionality that you can add to your app to increase its abilities in some way. SwiftUI is a framework too.



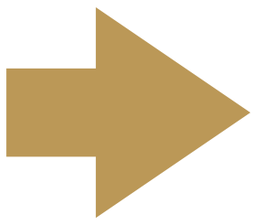
Data Objects (Models)

Earlier in this book you learned about the VOODO architecture.
As part of that architecture, you learned about Data Objects.

In SwiftData, you define a data object in a special way. SwiftData calls them "Models".
This allows SwiftData to persist the data (so the user can shut down the app and the data will not go away).

Data Object

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



SwiftData Model

```
import SwiftData

@Model
class BookModel {
    Var id = UUID()
    var name: String

    init(name: String) {
        self.name = name
    }
}
```

To access the SwiftData framework, you have to **import** it.

(You may have noticed that when you create SwiftUI views, there is an "import SwiftUI" at the top. That is to give you access to the SwiftUI framework.)

Previously, you learned about the #Preview and @Observable macros.

@Model is a SwiftData macro that helps your data object work with SwiftData.

You may have noticed that your data object is a **class** and not a struct now.

SwiftData models have to be classes.

They also will need an initializer (init).
See the next page for more information.

Initializers (init)

In Swift, an initializer (often abbreviated as "**init**") is a special method that prepares an instance of a class for use.

This is where you put code to set up your new object with the initial data you want.

```
import SwiftData

@Model
class BookModel {
    let id = UUID()
    var name: String

    init(name: String) {
    self.name = name
    }
}
```

Using the Initializer

When you want to create a new BookModel data object, you can do so like this:

```
var book = BookModel.init(name: "SwiftUI Simplified")
```

Developers do **not** usually call the init function directly.

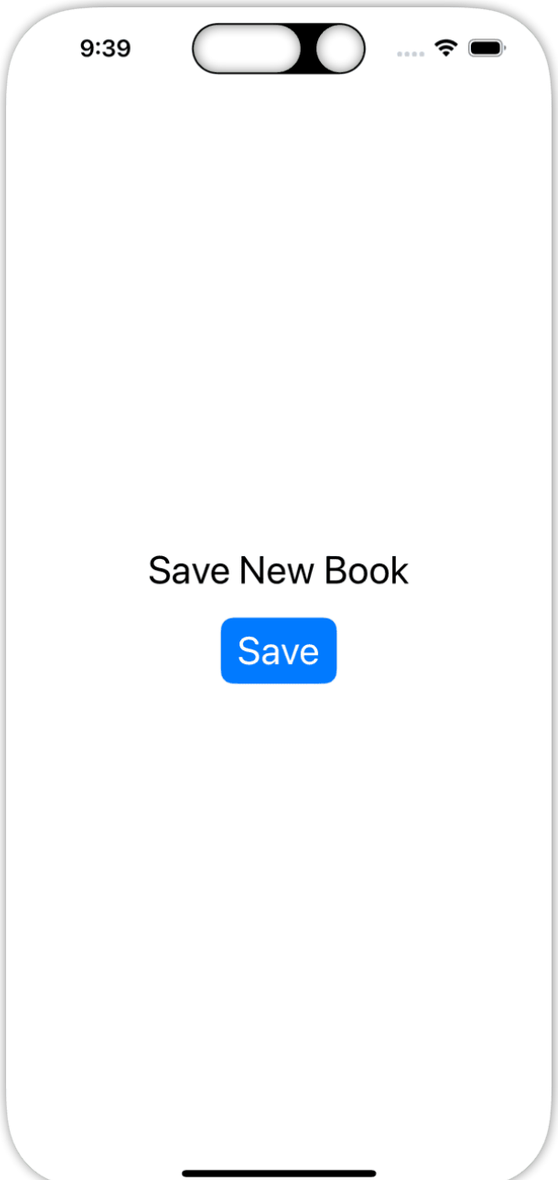
Instead, you can leave out the init and create a new object like this:

```
var book = BookModel(name: "SwiftUI Simplified”)
```

The string value you pass in is what gets set to the name property (self.name).

If your method's parameter label has the same name as the property you are setting (both are `name`), you can use `self` to distinguish it like we do here.

How to save?




```
struct SwiftData_NewBook: View {
    var body: some View {
        VStack {
            Text("Save New Book")


            Button("Save") {
                let newBook = BookModel(name: "My New Book")

                // How to save?
            }
            .buttonStyle(.borderedProminent)
        }
        .font(.title)
    }
}
```

Now that you know how to create a new instance of a SwiftData model, how do you save it?

In this example, if you tap the Save button a new SwiftData model is created but not yet saved.

If you run this view and tap the Save button, you will get an error that says it cannot find a "**container**".

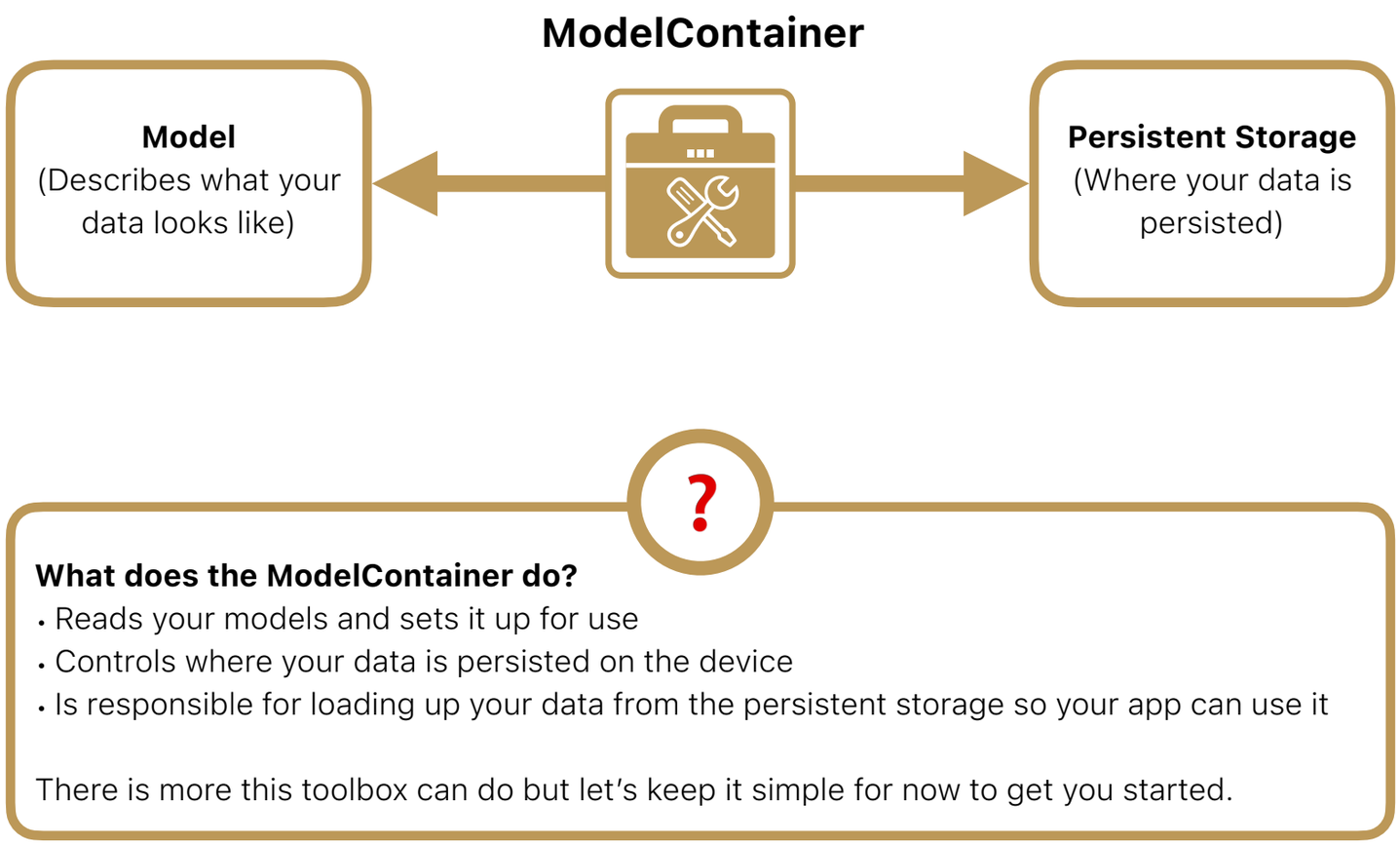
What is a container?
A container is short for "model container". It is a tool that can persist your model data to storage on the device.

The ModelContainer

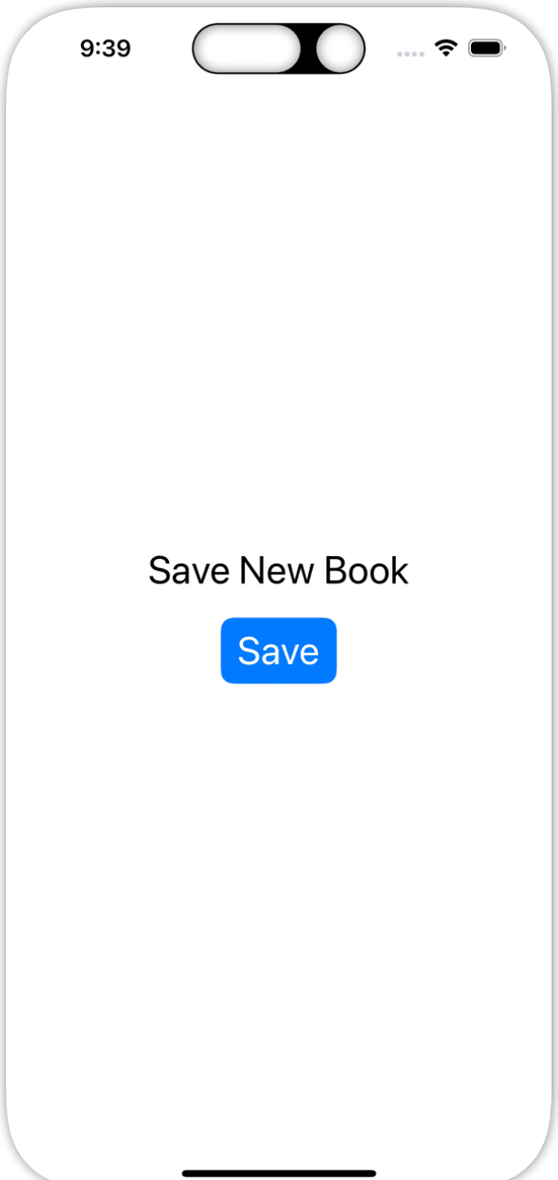
Previously, you got an error when creating a SwiftData model because **SwiftData didn't know WHERE to persist your data**.

The error message said it could not find a "**container**", which represents the place where your data gets persisted.

Models need to be associated with a container.



The modelContainer Modifier




```
struct SwiftData_NewBook: View {
    var body: some View {
        VStack {
            Text("Save New Book")

            Button("Save") {
                let newBook = BookModel(name: "My New Book")

                // How to save?
            }
            .buttonStyle(.borderedProminent)
        }
        .font(.title)
    }
}

#Preview {
    SwiftData_NewBook()
        .modelContainer(for: BookModel.self, inMemory: true)
}
```

You use a modifier to create a **ModelContainer**.

With this modifier, you specify the model you want to associate with the permanent storage.

Now, your models will have a home when they get saved.

What is inMemory for?
When you are developing with SwiftData, you might not want to persist a lot of test data in your app.

By setting inMemory to true, you can temporarily save the data in memory. It will no longer be permanent.

App File

Previews

So far, you have seen how the **modelContainer** was used inside the #Preview for a single view.

App

For your app, it will be used in the App file, when your app starts with its initial view.

You have two options:

1. You can add it to the first view your app shows and that view and all views it navigates to can access and use that data.
2. You can also add it to the WindowGroup and ALL views, no matter how you get to them, will have access to your data.

```
import SwiftUI

@main
struct SwiftUI_SimplifiedApp: App {
    var body: some Scene {
        WindowGroup {
            SwiftData_NewBook()
                .modelContainer(for: BookModel.self, inMemory: true)
        }
    }
}
```

1

```
import SwiftUI

@main
struct SwiftUI_SimplifiedApp: App {
    var body: some Scene {
        WindowGroup {
            SwiftData_NewBook()
        }
        .modelContainer(for: BookModel.self, inMemory: true)
    }
}
```

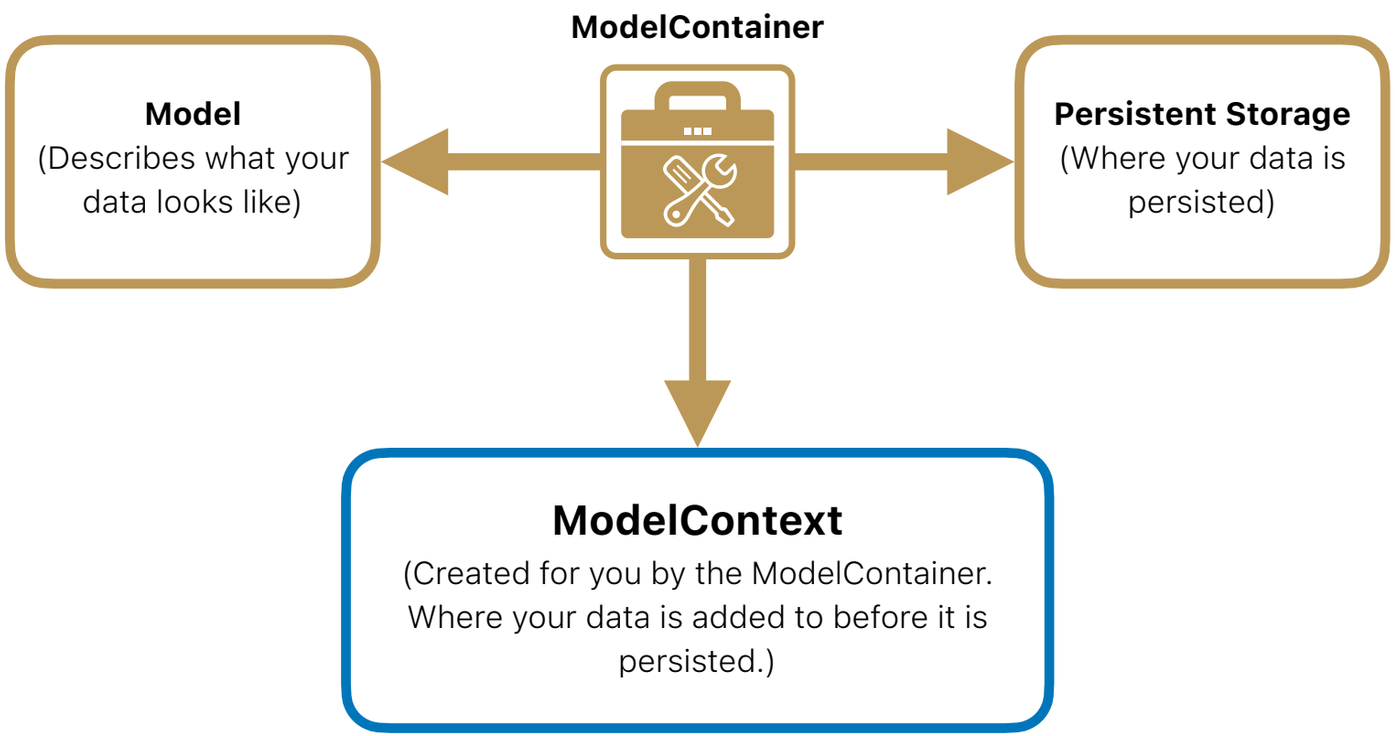
2

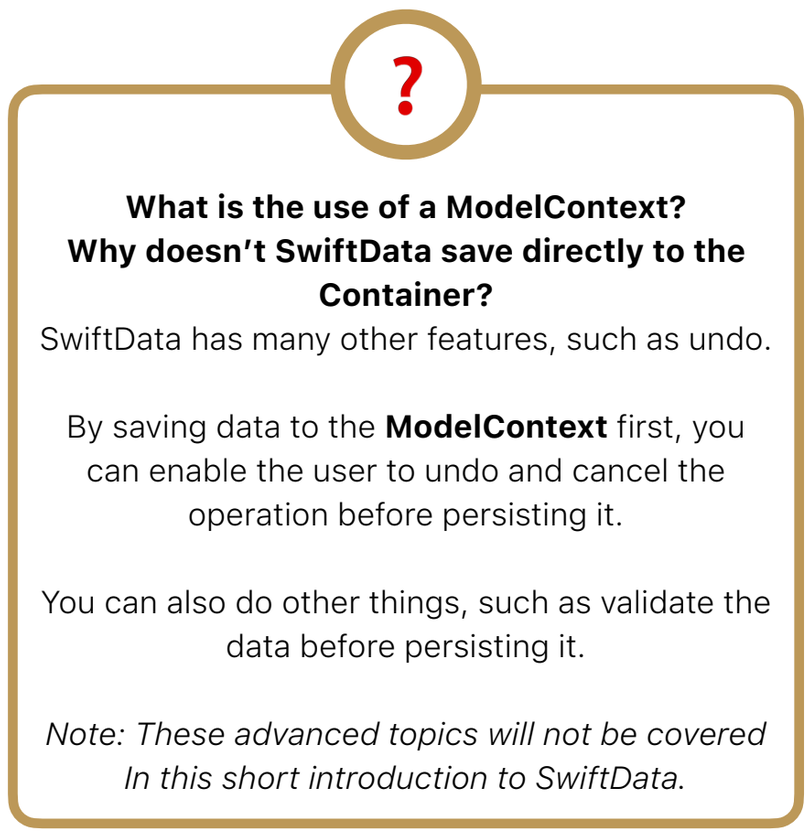
ModelContext

Before you can save new data, there is one more piece of the puzzle we need. That is a **ModelContext**.

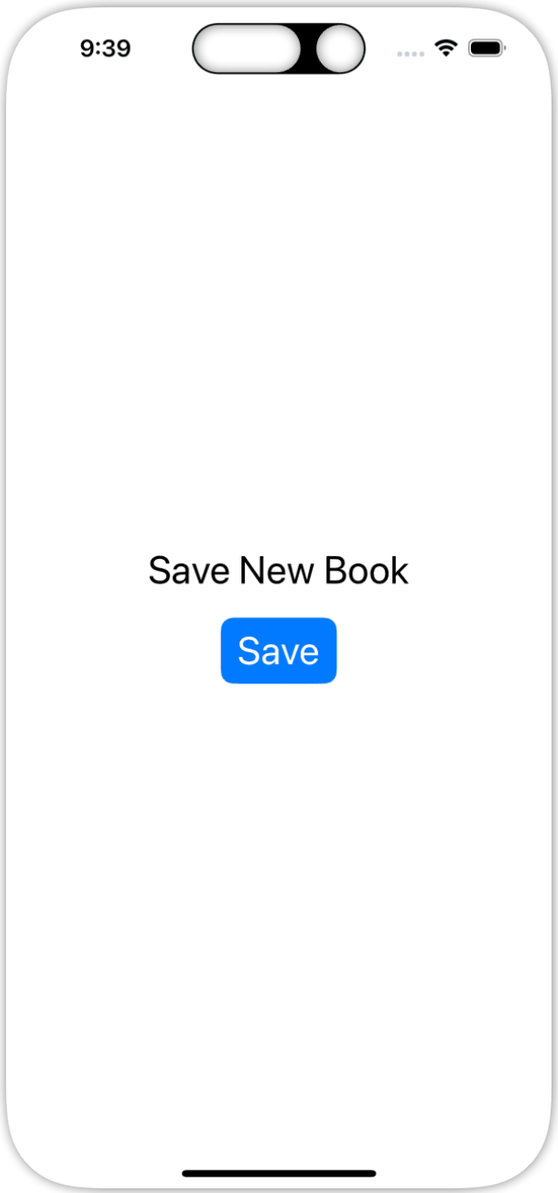
In programming, a "**context**" usually refers to a place in memory where special things can happen without other things getting in the way. It's like a reserved room for a special purpose.

In SwiftData, the ModelContainer will create this special room for you to do SwiftData operations such as inserting new data, deleting data, updating data or just getting data. This special room is called the **ModelContext**.





Accessing and Using the ModelContext




```
struct SwiftData_NewBook: View {
    @Environment(\.modelContext) private var modelContext

    var body: some View {
        VStack {
            Text("Save New Book")

            Button("Save") {
                let newBook = BookModel(name: "My New Book")

                // How to save?
                modelContext.insert(newBook)
            }
            .buttonStyle(.borderedProminent)
        }
        .font(.title)
    }
}

#Preview {
    SwiftData_NewBook()
        .modelContainer(for: BookModel.self, inMemory: true)
}
```

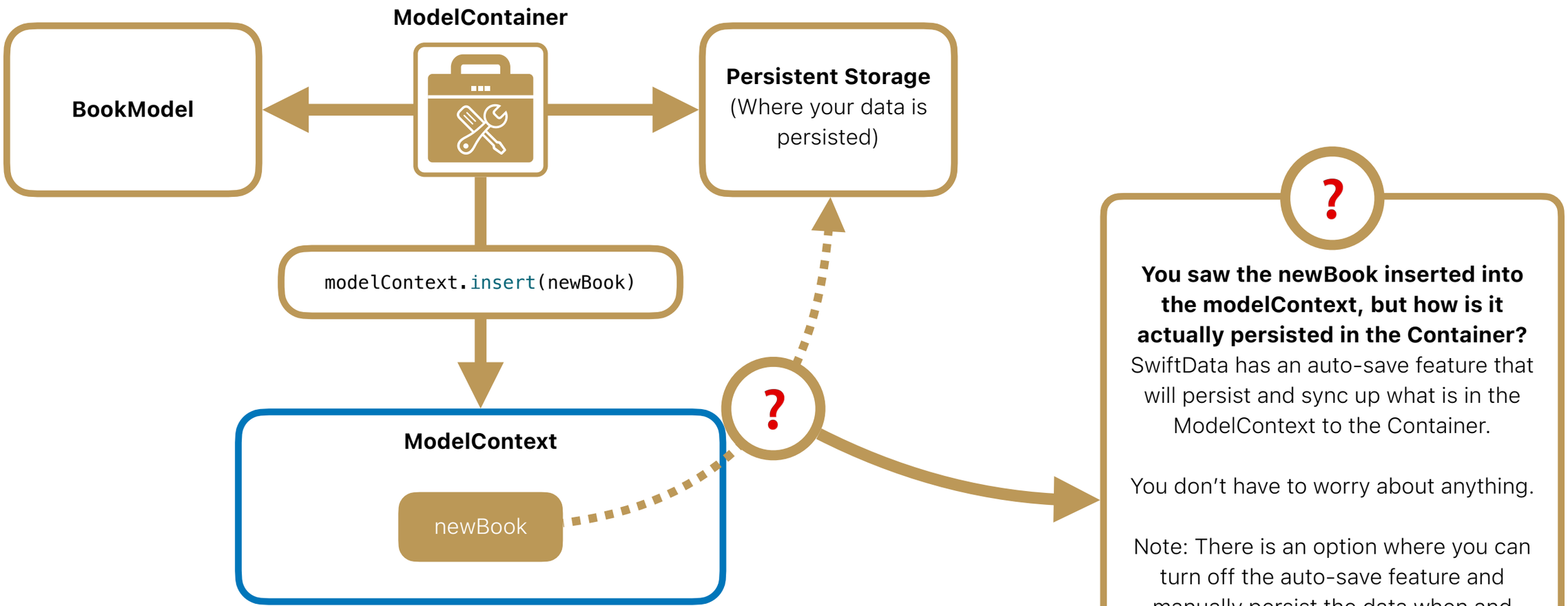
2

3

1

1. The **modelContainer** modifier not only associates your models with the persistent storage, it also creates the **ModelContext** so you can do SwiftData operations (fetch, insert, update, delete).
2. The modelContainer added the ModelContext to the Environment. This makes it easily accessible from this view and any view this one might navigate to. You use **@Environment** to say you want to get something from the environment and put it into a property you can use in your code.
3. You use the modelContext property to **insert** your model into it.

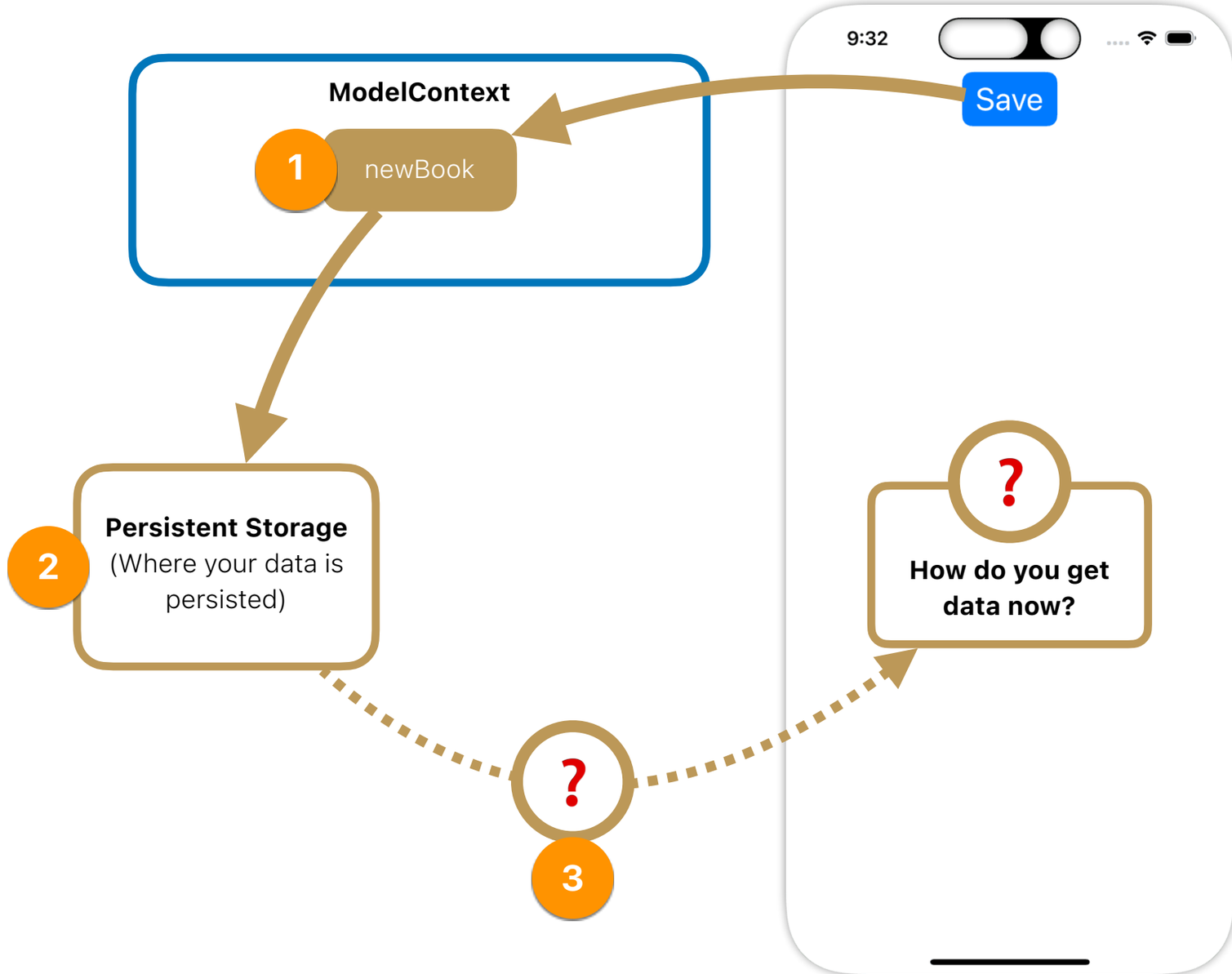
How is the data saved?



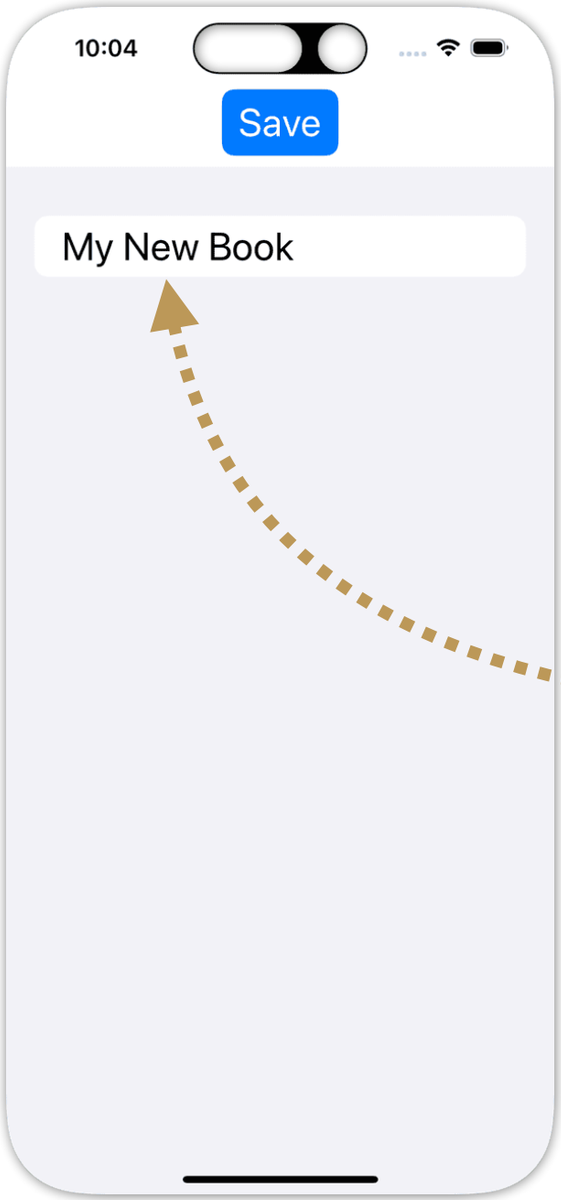
How do you get data?

You have seen how you can:
1. Insert data (newBook) into the model context
2. It automatically gets persisted to the storage.
3. But how can you retrieve data to show on the UI?

SwiftData has a special way that allows you to easily get the data you want.



@Query




```swift
import SwiftData // Needed

struct SwiftData_BookList: View {
    @Environment(\.modelContext) private var modelContext
    @Query private var books: [BookModel]

    var body: some View {
        VStack {
            Button("Save") {
                let newBook = BookModel(name: "My New Book")

                modelContext.insert(newBook)
            }
            .buttonStyle(.borderedProminent)

            List(books) { book in
                Text(book.name)
            }
        }
        .font(.title)
    }
}

#Preview {
    SwiftData_BookList()
        .modelContainer(for: BookModel.self, inMemory: true)
}
```

1

2

@Query to the rescue!

SwiftData has a @Query attribute you can add to properties that look in the persistent store to find all matching types.

In this example:

1. The query looks for all of the BookModel objects and assigns the results to books.
2. When the books property has values, the List will display them all and show just the BookModel.name property in a Text view.

?

Do I need to re-run the query when any data changes?

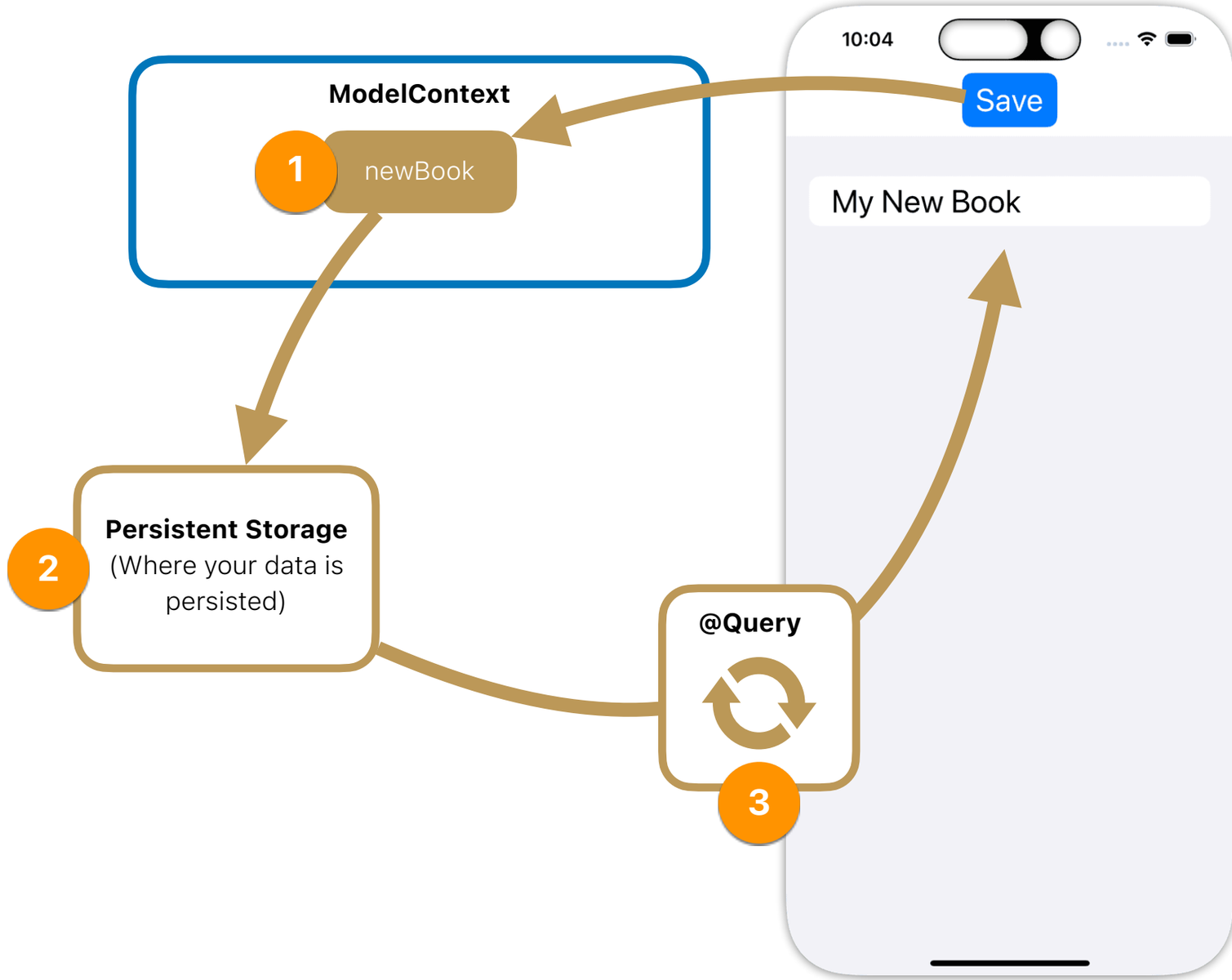
No. The query keeps track of any changes and updates the property automatically.

@Query Keeps the UI Updated

The query you added will automatically and constantly look for changes in the container (persistent store) and update your UI.

1. Insert data (newBook) into the model context
2. It automatically gets persisted to the storage
3. **The @Query monitors for changes and will update your UI with data.**

SwiftData makes it easy for you to save and query data to display on your UI!



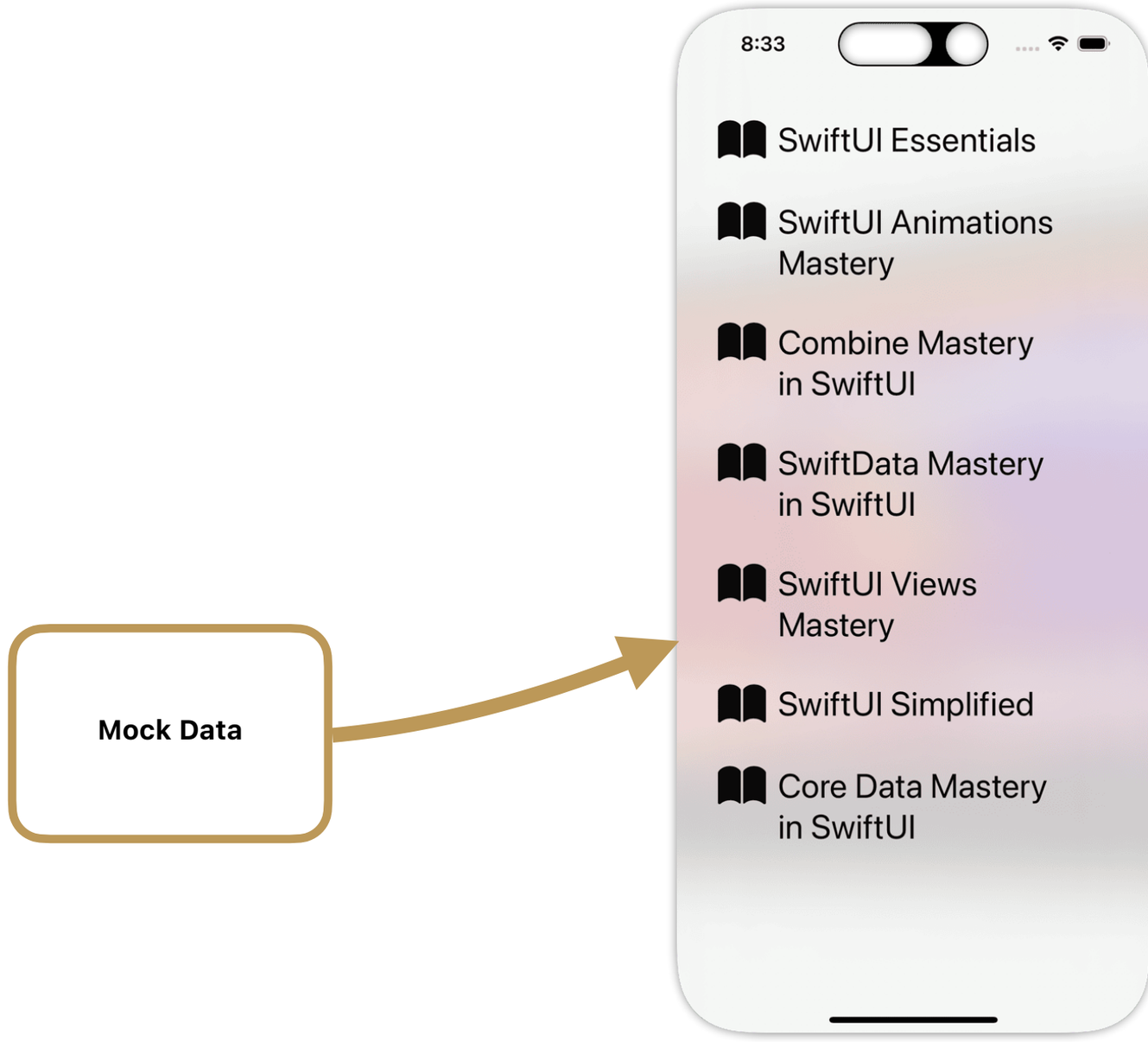
Mock Data

What does “mock” mean?
In programming, “mock” means “**not real**” or “**fake**”.

Many times you want to see your preview of the UI with mock/fake data so you get a better idea of what it will look like.

But how can you add mock data to SwiftData without it persisting (and without having to manually add it all the time)?

There are different strategies to accomplish this but I will share with you what I think is a good solution to accomplish this.



Another modelContainer Modifier

ModelContainer Modifier 1

Earlier, you learned that the `modelContainer` modifier sets up your persistent storage and model context so you can query, insert, update, and delete data.

You attach the **modelContainer** to a view like so:

```
.modelContainer(for: BookModel.self, inMemory: true)
```

1

You use a modifier to create a **ModelContainer**.

With this modifier, you specify the model you want to associate with the permanent storage.

Now, your models will have a home when they get saved.

ModelContainer Modifier 2

There is another modifier you can use where you can create a separate ModelContainer object and pass it into the modelContainer modifier:

```
let container = ModelContainer(for: BookModel.self ...)

.modelContainer(container)
```

2

This creates a great opportunity to create a ModelContainer and use it to add your mock data.

This way, when your new model container is attached to the view (using the modelContainer modifier) there will already be data available to read.

Will the mock data be stored permanently?
No. You will learn how to create a custom ModelContainer so that data is stored only in memory and not permanently.

Creating a Custom ModelContainer Overview

You now have an idea of how the custom ModelContainer with the mock data will be applied to your view using the second modelContainer modifier.

Now, let's look at the 3 steps to creating this custom model container.

1. The Property

First, you will learn where I add this property so it will be easy for you to find in the future.

I will show you the basic setup and you will learn some new Swift keywords that make it all work.

2. The ModelContainer

There are mainly two things you want to specify when creating a custom model container:

1. The model(s) you want to associate with it.
2. Specifying that you want it to only be temporary data and not permanently store it.

2. The Mock Data

Your ModelContainer gives you the context (ModelContext) which you can use to add data too.

When data is added to this context, SwiftData will automatically save it.

Once your mock data is saved, it will show up in your SwiftUI previews!

1. Creating the ModelContainer Property

Here is the basic setup.

There is a lot happening here so let's briefly cover all the points:

1. **Extension** - This allows you to add more functions or properties to another struct or class without it being inside the same file.
2. **@MainActor** - This is an advanced topic. It simply allows this property to work correctly with the UI.
3. **Static** - This means you do not have to create a new BookModel to use it. You can just use it like `BookModel.previewContainer`.
4. **ModelContainer** - This property will return a ModelContainer. Usually, the modelContainer modifier will create the ModelContainer for you. In this case, you will create your own ModelContainer.

```
1 extension BookModel {

2     @MainActor

3     static var previewContainer: ModelContainer {
                                                  4
        // 2. Create the ModelContainer

        let container = ModelContainer. . .



        // 3. Add mock data



        return container

    }

}
```

2. Creating the Custom ModelContainer

```
extension BookModel {
    @MainActor
    static var previewContainer: ModelContainer {
        let container = try! ModelContainer(for: BookModel.self,
                                            configurations: ModelConfiguration(isStoredInMemoryOnly: true))

        // Add mock data


        return container
    }
}
```

Now to create the ModelContainer.

1. **try!** - Creating a model container can fail and cause your app to crash. You want to know right away if it fails. Using try with the exclamation point says, "try to create a model container and if there is a problem, do not let the app continue, crash the app instead."
2. **BookModel.self** - You are telling the ModelContainer that you will be working with book models.
3. **ModelConfiguration** - You want to specify you do not want to permanently persist this data in the storage. Previously, you did this with `inMemory: true`. Unfortunately, this ModelContainer initializer doesn't have that parameter so you have to create a ModelConfiguration that specifies you do not want to persist the data.

3. Adding the Mock Data

You can now add the mock data.

1. **container.mainContext** - You saw earlier that you can add data using modelContext.insert. In this example, the container's mainContext is a ModelContext. The ModelContext is where you add, insert, and delete data from before it is persisted.
2. **BookModel** - Now you can create new models to insert into the ModelContext.

```
extension BookModel {
    @MainActor
    static var previewContainer: ModelContainer {
        let container = try! ModelContainer(for: BookModel.self,
                                            configurations: ModelConfiguration(isStoredInMemoryOnly: true))

        container.mainContext.insert(BookModel(name: "SwiftUI Simplified"))
        container.mainContext.insert(BookModel(name: "SwiftUI Essentials"))
        container.mainContext.insert(BookModel(name: "SwiftUI Views Mastery"))
        container.mainContext.insert(BookModel(name: "SwiftUI Animations Mastery"))
        container.mainContext.insert(BookModel(name: "SwiftData Mastery in SwiftUI"))
        container.mainContext.insert(BookModel(name: "Core Data Mastery in SwiftUI"))
        container.mainContext.insert(BookModel(name: "Combine Mastery in SwiftUI"))

        return container
    }
}
```

Summary

This might be the most complicated part of this whole book so let's just summarize what you did here.

1. First, you created a property on the model you wanted to add mock data to.
2. Then you created a ModelContainer specifying that model you wanted to add mock data to and you told it to only store the data in memory (not persist it).
3. Then you created and added the mock data to the context (mainContext) and SwiftData saved this data so now you can use it in your SwiftUI Preview.

1

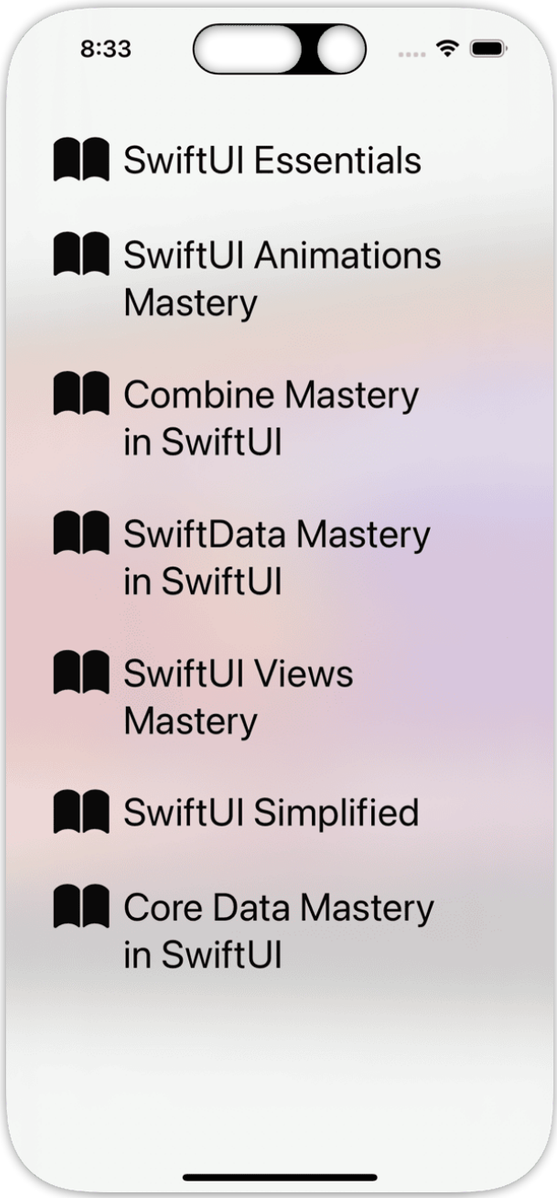
2

3

```
extension BookModel {
    @MainActor
    static var previewContainer: ModelContainer {
        let container = try! ModelContainer(for: BookModel.self,
                                            configurations: ModelConfiguration(isStoredInMemoryOnly: true))

        container.mainContext.insert(BookModel(name: "SwiftUI Simplified"))
        container.mainContext.insert(BookModel(name: "SwiftUI Essentials"))
        container.mainContext.insert(BookModel(name: "SwiftUI Views Mastery"))
        container.mainContext.insert(BookModel(name: "SwiftUI Animations Mastery"))
        container.mainContext.insert(BookModel(name: "SwiftData Mastery in SwiftUI"))
        container.mainContext.insert(BookModel(name: "Core Data Mastery in SwiftUI"))
        container.mainContext.insert(BookModel(name: "Combine Mastery in SwiftUI"))

        return container
    }
}
```

Now that you have a property that returns a ModelContainer with mock data inside its model context, it is time to add it to your preview using the other **modelContainer** modifier.

Using the Mock Data




```swift
import SwiftUI
import SwiftData

struct SwiftData_WithMockData: View {
    @Query private var books: [BookModel]

    var body: some View {
        List(books) { book in
            Label(book.name, systemImage: "book.fill")
                .foregroundStyle(Color.primary)
                .padding(4)
                .listRowSeparator(.hidden)
                .listRowBackground(Color.clear)
        }
        .padding()
        .listStyle(.plain)
        .background(.regularMaterial)
        .background(Image(.book))
        .font(.title)
    }
}

#Preview {
    SwiftData_WithMockData()
        .modelContainer(BookModel.previewContainer)
}
```

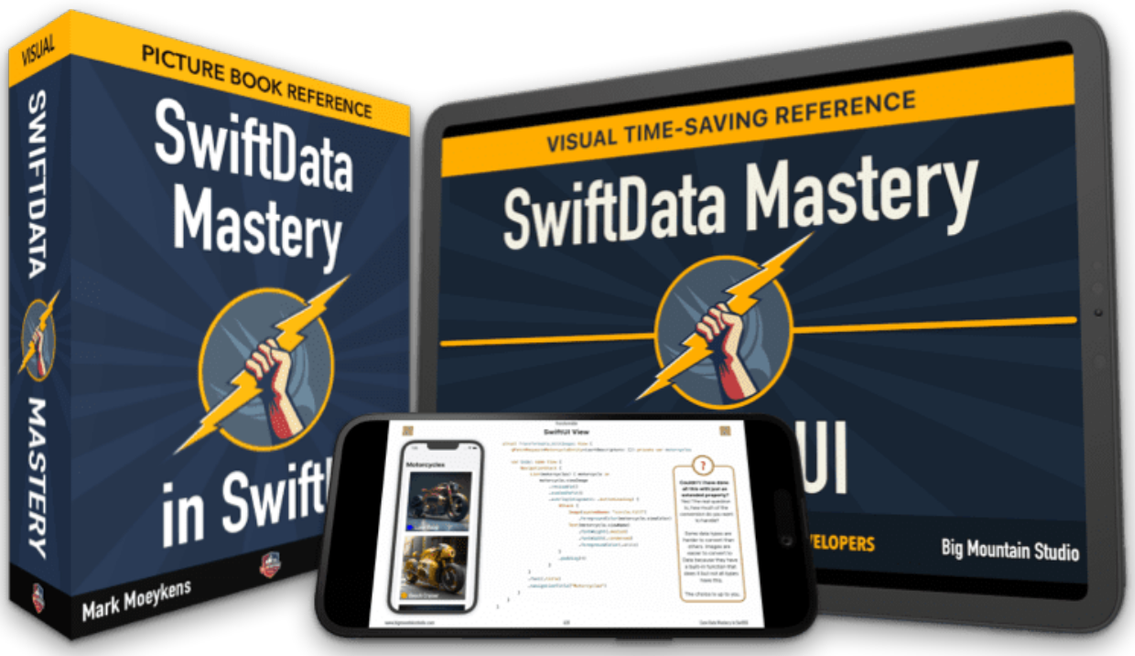
Use the second modelContainer modifier to add your custom ModelContainer object that has mock data stored in memory.

This mock data can be queried and will now show in your preview.

This enables you to better build your UI with realistic mock data.

SwiftData Mastery in SwiftUI

LEARN MORE ABOUT THE SWIFTDATA FRAMEWORK VISUALLY SO YOU CAN SAVE TIME BUILDING APP FEATURES.



- Over **500** pages - The largest SwiftData book for SwiftUI
- What are the 4 main concepts that will help you finally understand how SwiftData works?
- How can you start using SwiftData in just 3 steps and under 10 minutes?
- How can you use AI to create a lot of mock data?
- How can you make your life much easier when using SwiftData and SwiftUI?
- How can you not only get data, but also sort, filter, and animate with a query?
- How do you sync data across multiple devices?
- What is one object you can use to prevent your UI from hanging and data from getting corrupted when doing a large number of SwiftData operations?
- BONUS: Get 4 SwiftData apps with integrations for MapKit, PhotosUI, and Charts.

LEARN MORE AND SAVE 10% ON THIS BOOK!

THE END

I hope you enjoyed this journey into SwiftUI!

This was just the beginning.

Continue your journey...

Hi, I'm Mark Moeykens

I'm a full-time mobile developer with over three decades of programming experience. I have created desktop, web and mobile apps in many fields including insurance, banking, transportation, health, and sales. I have also given talks on programming and enjoy breaking down complex subjects into simple parts to teach in a practical and useful manner.

youtube.com/markmoeykens

Find tutorials on iOS topics where I guide you step-by-step through all different aspects of development.

Website: www.bigmountainstudio.com

Join my climber's camp and see what products I have available, learn something new and see what I am working on.

- *Read articles*
- *Find courses*
- *Download books*

@BigMtnStudio

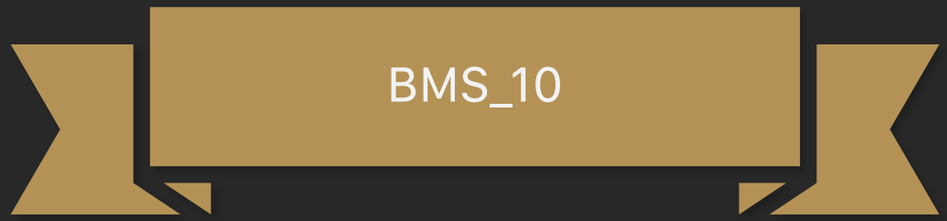
Stay up-to-date on what I'm learning and working on. These are the most real-time updates you will find.

@BigMtnStudio

Do you prefer hanging out in Instagram? Then follow and get bite-sized chunks of dev info.

You Get 10% Off!

Take Advantage of This Introductory Discount!



Because you got this book you get an introductory discount of **10% off everything** in the store. This is to encourage you to continue your SwiftUI journey and keep getting better at your craft. **Enter the code above on checkout or click the button below.** 👇

ACTIVATE DISCOUNT

(**Note**: You should have gotten an email with this coupon code too. Make sure you opt-in for even better discounts (up to 40%) through sales in the future. Go here for instructions on how to do this.)

SwiftUI Essentials

ARCHITECTING SCALABLE & MAINTAINABLE SWIFTUI APPS

Working with data in SwiftUI is super confusing. I know, I was there trying to sort it all out. That's why I made this simple to read book so that anyone can learn it.

- ✓ How to architect your app
- ✓ Learn what binding is
- ✓ What is @StateObject and when should you use it?
- ✓ How is @State different from @StateObject?
- ✓ How can you have data update automatically from parent to child views?
- ✓ How can you work with a data model and still be able to preview your views while creating them?
- ✓ How do you persist data even after your app shuts down?
- ✓ Working with JSON

SAVE 10% ON THIS BOOK!

SwiftUI Views Mastery

THE COMPLETE, VISUAL TIME-SAVING REFERENCE

- ✓ Over **1,000** pages of SwiftUI
- ✓ Over **700** screenshots/videos showing you what you can do so you can quickly come back and reference the code
- ✓ Learn all the ways to work with and modify images
- ✓ See the many ways you can use color as views
- ✓ Discover the different gradients and how you can apply them
- ✓ Find out how to implement action sheets, modals, popovers and custom popups
- ✓ Master all the layout modifiers including background and overlay layers, scaling, offsets padding and positioning
- ✓ How do you hide the status bar in SwiftUI? Find out!
- ✓ ***This is just the tip of the mountain!***

SAVE 10% ON THIS BOOK!

SwiftUI Animations Mastery

DO YOU LIKE ANIMATIONS? WOULD YOU LIKE TO SEE HUNDREDS OF VIDEO ANIMATION EXAMPLES WITH THE CODE?

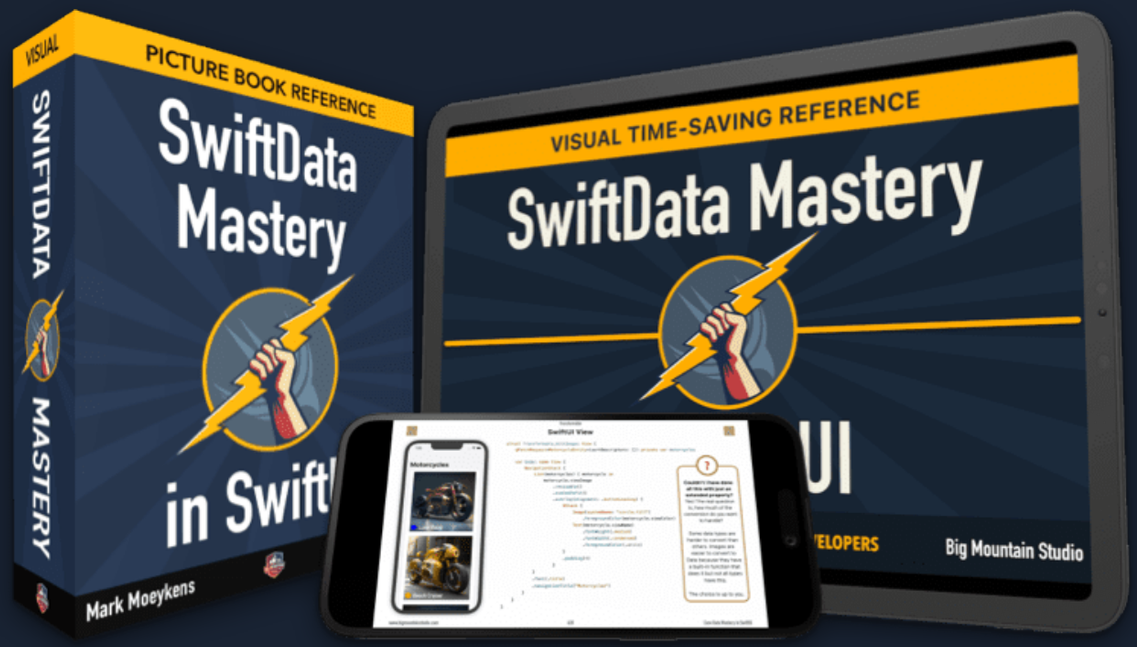
SwiftUI made animations super easy...except when it isn't. Most new SwiftUI developers can get a simple animation working but as soon as they want to customize it, they get lost and waste more time than they should trying to get it working. This book will help you with that struggle.

- ✓ Learn all the animation types and options with embedded video samples and code
- ✓ Master spring animations
- ✓ Master transitions for views that are inserted and removed from the screen
- ✓ Learn how matchedGeometryReader should really work
- ✓ Customize animations with speeds, delays, and durations
- ✓ Understand how to animate changes from one view to another, such as from list to detail

SAVE 10% ON THIS BOOK!

SwiftData Mastery in SwiftUI

QUICKLY LEARN APPLE'S NEW SWIFTDATA FRAMEWORK VISUALLY SO YOU CAN SAVE TIME BUILDING APP FEATURES.

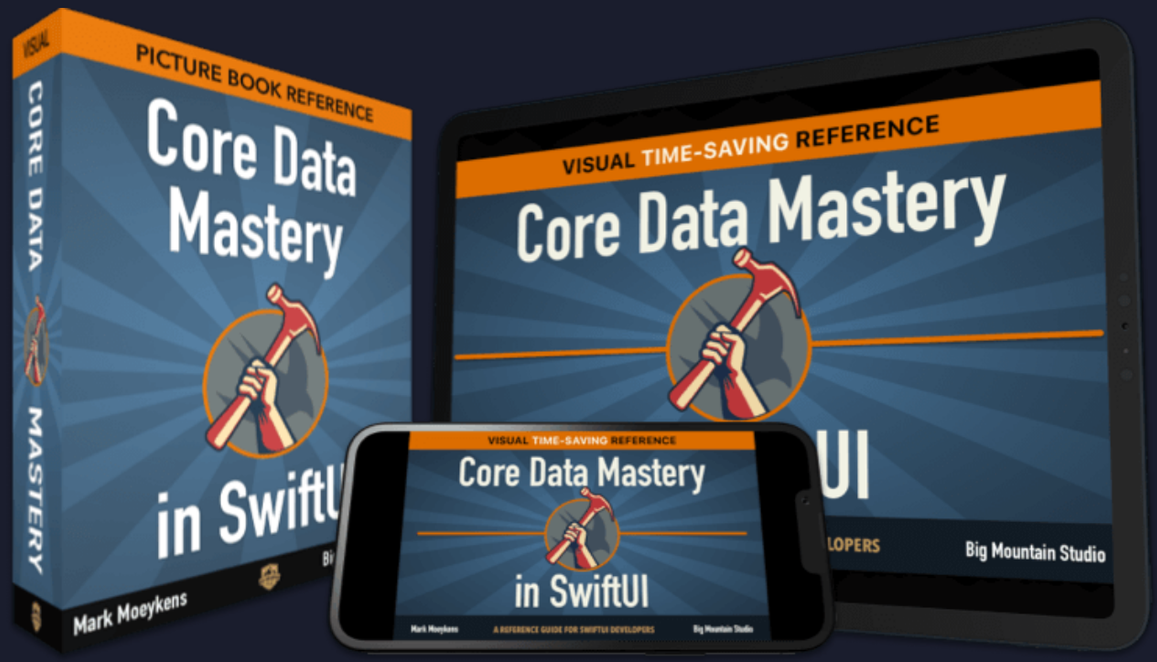


- ✓ Over **500** pages - The largest SwiftData book for SwiftUI
- ✓ What are the 4 main concepts that will help you finally understand how SwiftData works?
- ✓ How can you start using SwiftData in just 3 steps and under 10 minutes?
- ✓ How can you use AI to create a lot of mock data?
- ✓ How can you make your life much easier when using SwiftData and SwiftUI?
- ✓ How can you not only get data, but also sort, filter, and animate with a query?
- ✓ How do you sync data across multiple devices?
- ✓ What is one object you can use to prevent your UI from hanging and data from getting corrupted when doing a large number of SwiftData operations?
- ✓ BONUS: Get 4 SwiftData apps with integrations for MapKit, PhotosUI, and Charts.

SAVE 10% ON THIS BOOK!

Core Data Mastery in SwiftUI

QUICKLY LEARN APPLE'S CORE DATA FRAMEWORK VISUALLY SO YOU CAN SAVE TIME BUILDING APP FEATURES.



- ✓ Over **500** pages - The largest Core Data book for SwiftUI
- ✓ What are the 4 main concepts that will help you finally understand how Core Data works?
- ✓ How can you start using Core Data in just 4 steps and under 10 minutes?
- ✓ How can you use mock data in Core Data and preview it while developing your UI?
- ✓ How can you make your life much easier when using Core Data and SwiftUI?
- ✓ How can you not only get data, but also sort, filter, and animate with a fetch request?
- ✓ What can you do if you want to group your data into sections?
- ✓ How can you use the managed object context to insert, update, and delete data?
- ✓ Much, much more. Click the link to see.

SAVE 10% ON THIS BOOK!

Combine Mastery in SwiftUI

HAVE YOU TRIED TO LEARN COMBINE AND FAILED LIKE I DID...MULTIPLE TIMES?

I finally figured out the secret to understanding Combine after 2 years and now I'm able to share it with you in this visual, game-changing reference guide.

- ✓ How can you architect your apps to work with Combine?
- ✓ Which Swift language topics should you know specifically that will allow you to understand how Combine works?
- ✓ What are the important 3 parts of Combine that allows you to build new data flows?
- ✓ How can Combine kick off multiple API calls all at one time and handle all incoming data to show on your screen? Using about **12 lines of code**...which includes error handling.

SAVE 10% ON THIS BOOK!