SUAL TIME-SAVING REFERENCE

iOS 18

SwiftUI Views Quick Start

Mark Moeykens

FREE SAMPLE

Big Mountain Studio

Version: 23-JUNE-2024

Are you new to Swift and SwiftUI?

Are you a complete beginner to the Swift language?

My good friend Chris Ching has you covered!

Before using my reference books you should have an understanding of the Swift language and how to get around Xcode.

Chris does a great job of getting absolute beginners introduced to the world of Swift and SwiftUI programming with his amazing and **free** program.

START THE FREE CHALLENGE TODAY

Book Quick Links

HOW TO USE

This is a visual **REFERENCE GUIDE**. Find a screenshot of something you want to learn more about or produce in your app and then read it and look at the code.

Read what is on the screenshots to learn more about the views and what they can do.
You can also read the book from beginning to end. The choice is yours.

CONVENTIONS

Embedded Videos

The ePUB version of the book supports embedded videos.

The PDF version does not.

This play button indicates that this is a playable video in the ePUB format.

But in **PDF** and **Kindle** it renders as simply a screenshot.

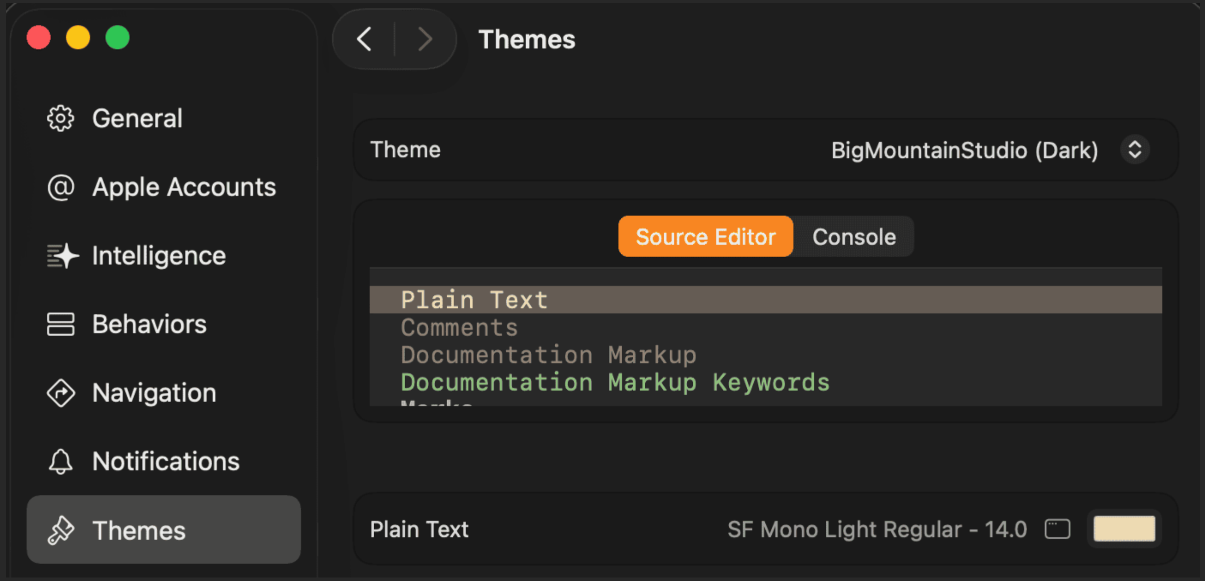
Note: In some ePUB readers you might have to **tap TWICE** (2) to play the video.

Custom Xcode Themes

The Xcode theme I use for my code is custom made. I have one for light and for dark modes.

Feel free to download them from GitHub.

Dark



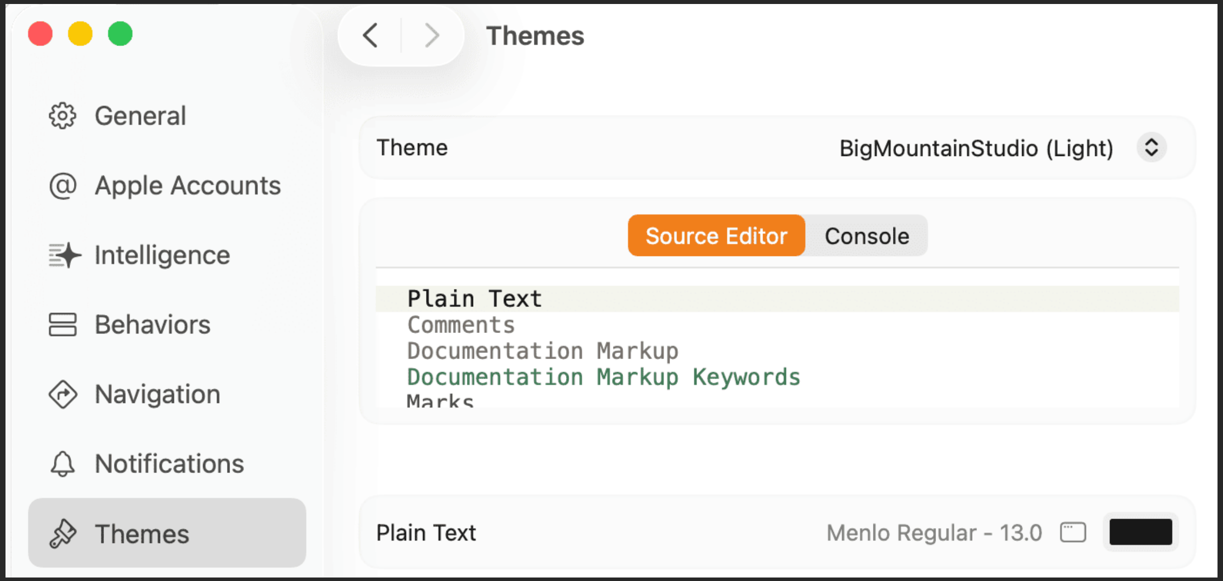
Themes

Theme: BigMountainStudio (Dark)

```
Plain Text
Comments
Documentation Markup
Documentation Markup Keywords
```

Plain Text: SF Mono Light Regular - 14.0

Light



Themes

Theme: BigMountainStudio (Light)

```
Plain Text
Comments
Documentation Markup
Documentation Markup Keywords
Marks
```

Plain Text: Menlo Regular - 13.0

Dark Mode Theme

Light Mode Theme

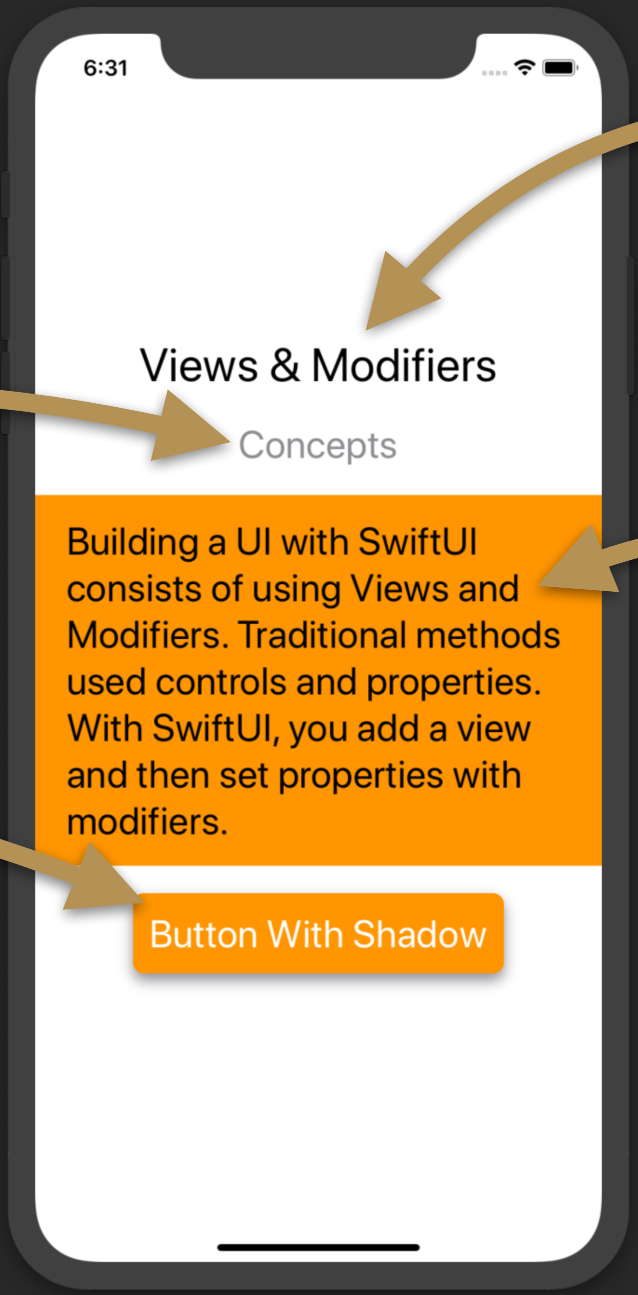
SWIFTUI

Basic 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.

Views And Modifiers

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



Building a UI with SwiftUI consists of using Views and Modifiers. Traditional methods used controls and properties. With SwiftUI, you add a view and then set properties with modifiers.

View

Modifiers:
- Large title text size

View

Modifiers:
- Title text size
- Gray text color

View

Modifiers:
- Title text size
- Orange background color
- Stretched to fit device width

View

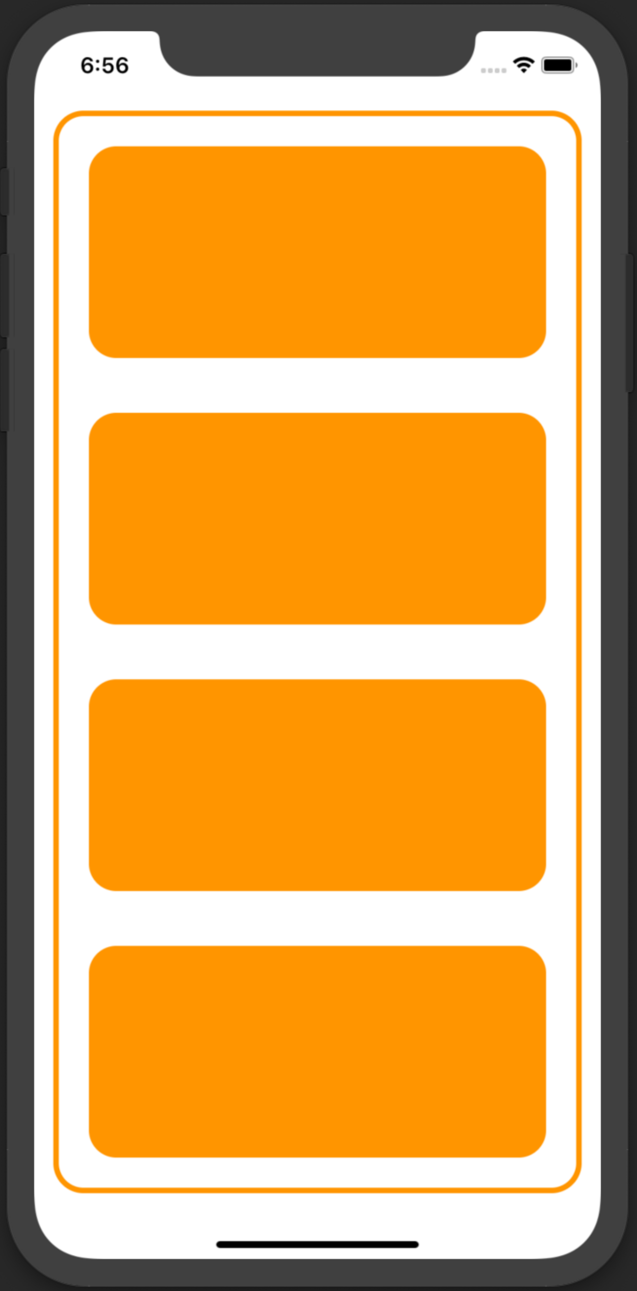
Modifiers:
- Title text size
- White text color
- Orange background color
- Rounded corners
- Shadow

Containers - 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**".

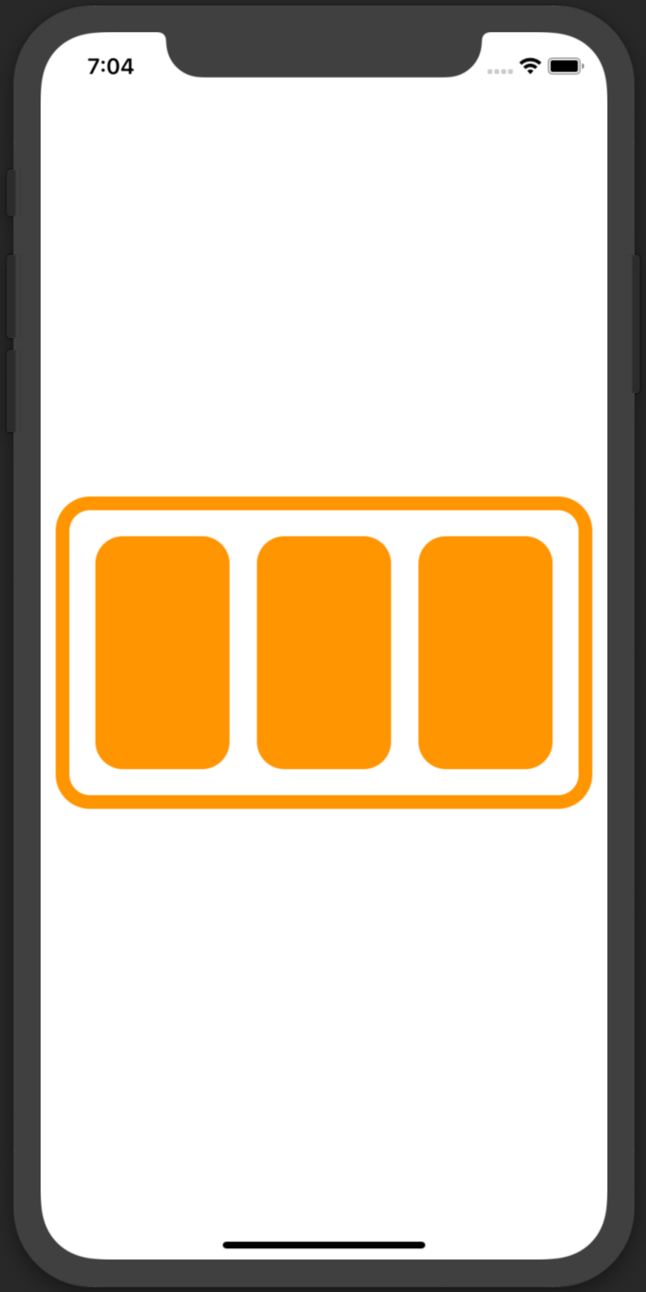
Stacks are views too. They are views that can have modifiers applied to them.



Horizontal Layout Container

There is another stack that can organize views horizontally.

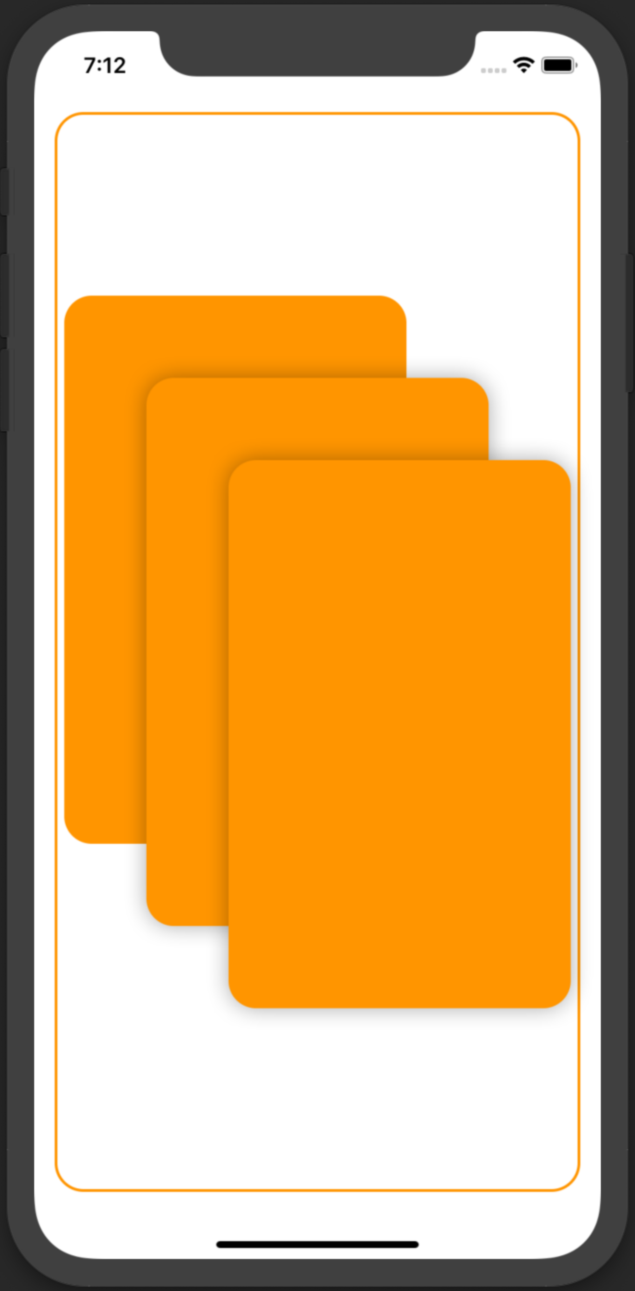
SwiftUI calls this horizontal stack an **HStack**.



Depth Layout Container

Another stack view will organize your views so they are one on top of another.

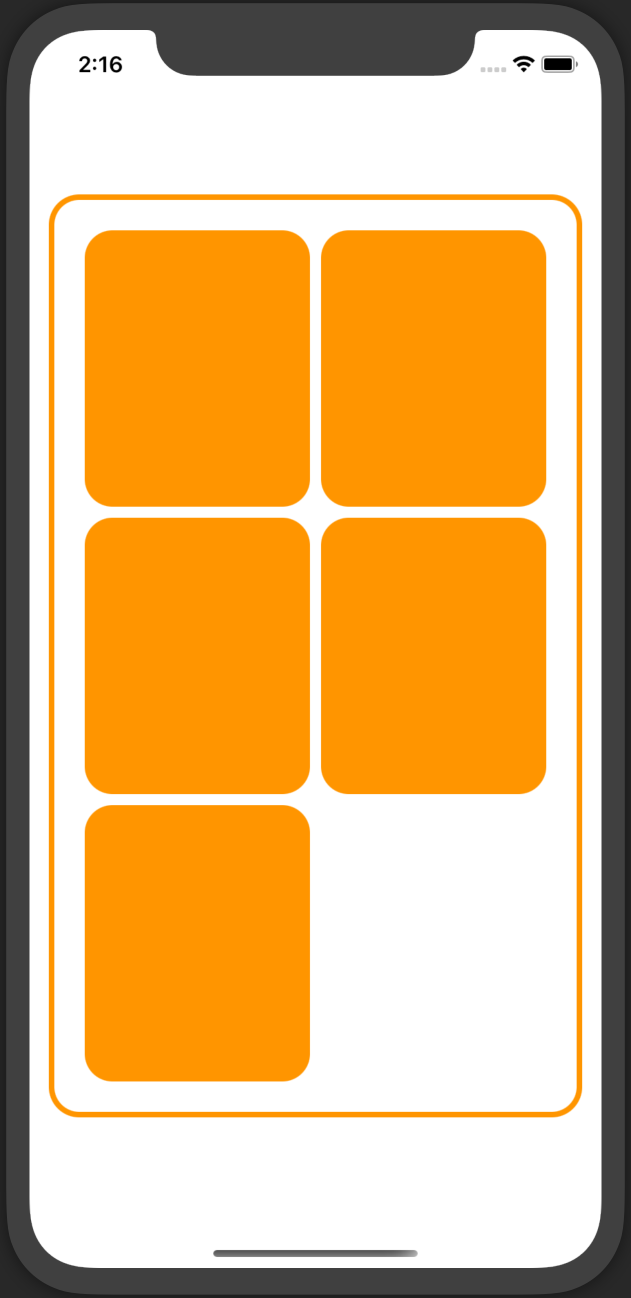
This is called the Depth Stack or **ZStack**.



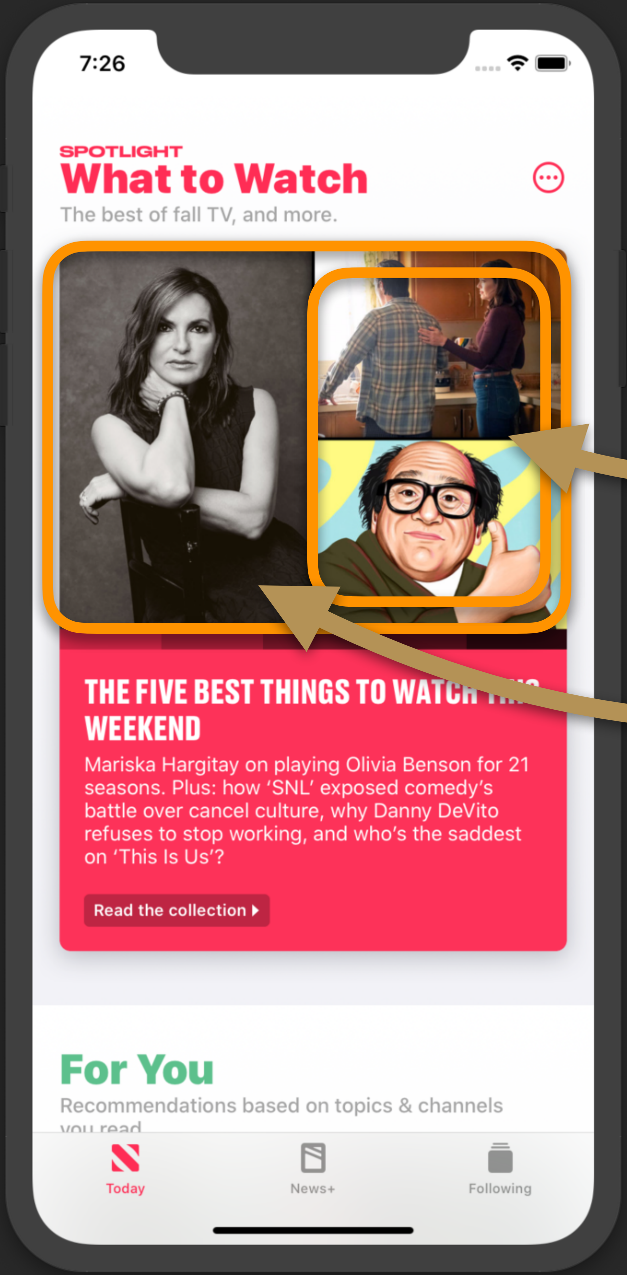
Grid Layout Container

iOS 14

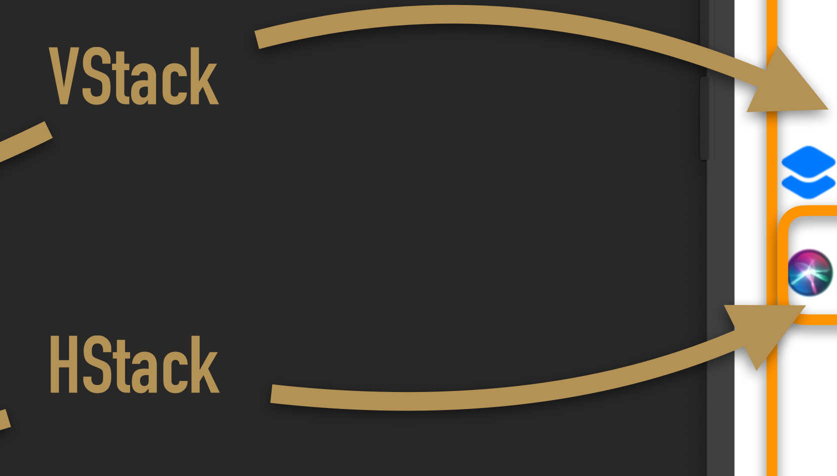
In the second version of SwiftUI, the grid container view was introduced. There is one for horizontal and vertical layouts.

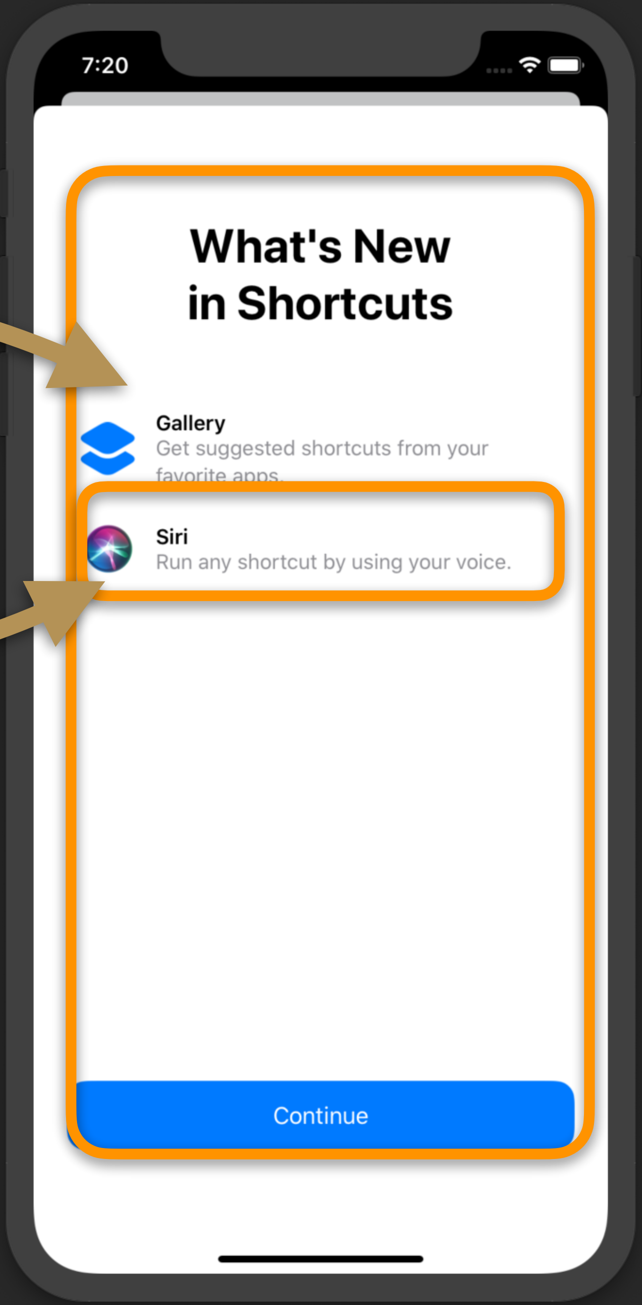


Layout Examples


7:26
SPOTLIGHT
What to Watch
The best of fall TV, and more.
THE FIVE BEST THINGS TO WATCH THIS WEEKEND
Mariska Hargitay on playing Olivia Benson for 21 seasons. Plus: how 'SNL' exposed comedy's battle over cancel culture, why Danny DeVito refuses to stop working, and who's the saddest on 'This Is Us'?
Read the collection ▸
For You
Recommendations based on topics & channels you read
Today
News+
Following

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




7:20
What's New in Shortcuts
Gallery
Get suggested shortcuts from your favorite apps.
Siri
Run any shortcut by using your voice.
Continue

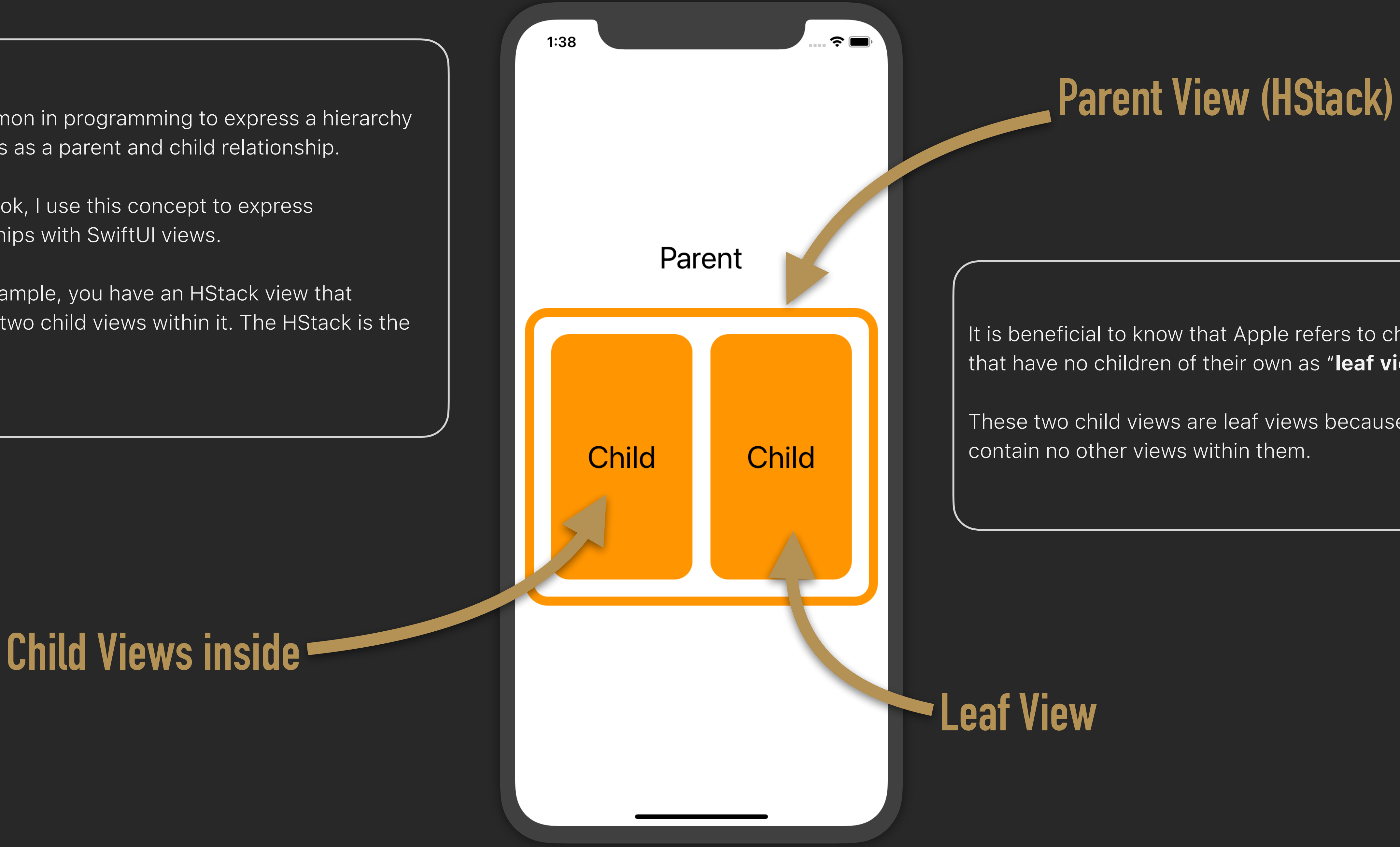
In this book, you will be seeing hundreds of layout examples. Pretty soon, it will become a natural ability of yours to recognize layout.

Relationships - Parent & Child

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

In this book, I use this concept to express relationships with SwiftUI views.

In this example, you have an HStack view that contains two child views within it. The HStack is the parent.



It is beneficial to know that Apple refers to child views that have no children of their own as "**leaf views**".

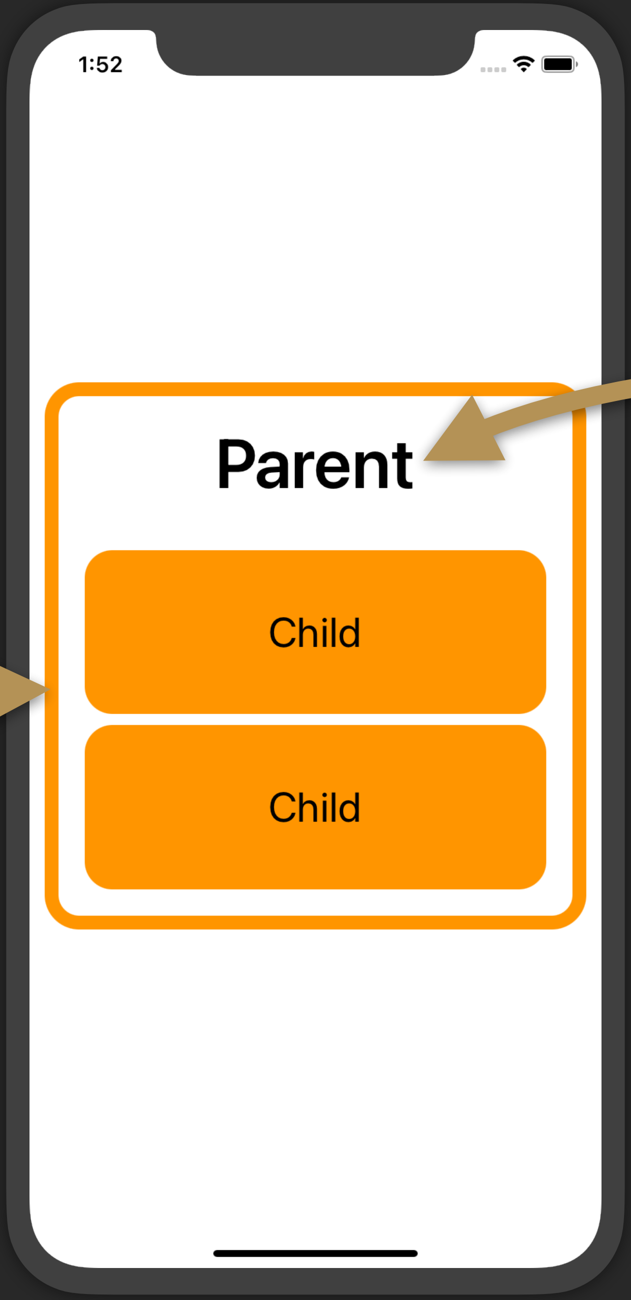
These two child views are leaf views because they contain no other views within them.

Relationships - And Modifiers

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 the child views use it.

The "Parent" text does not use the font size because it overrides it with a larger font size.



Text View (Child)

Overriding Modifiers:

- Font size of 50 points

VStack View (Parent)

Modifiers applied to all child views:

- Font size of 30 points

Understanding the Syntax

If you have used Swift in the past, then the SwiftUI syntax may look a little different.

It may not be readily apparent just how this code can even compile. This chapter is to help you understand how the code is able to work.

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:

A `struct` is a way to group related properties and functions together. Everything related to this screen can be added to this struct.

This is the name of your screen or view.

The `View` tells SwiftUI that this struct can be used to create something visual on the screen.

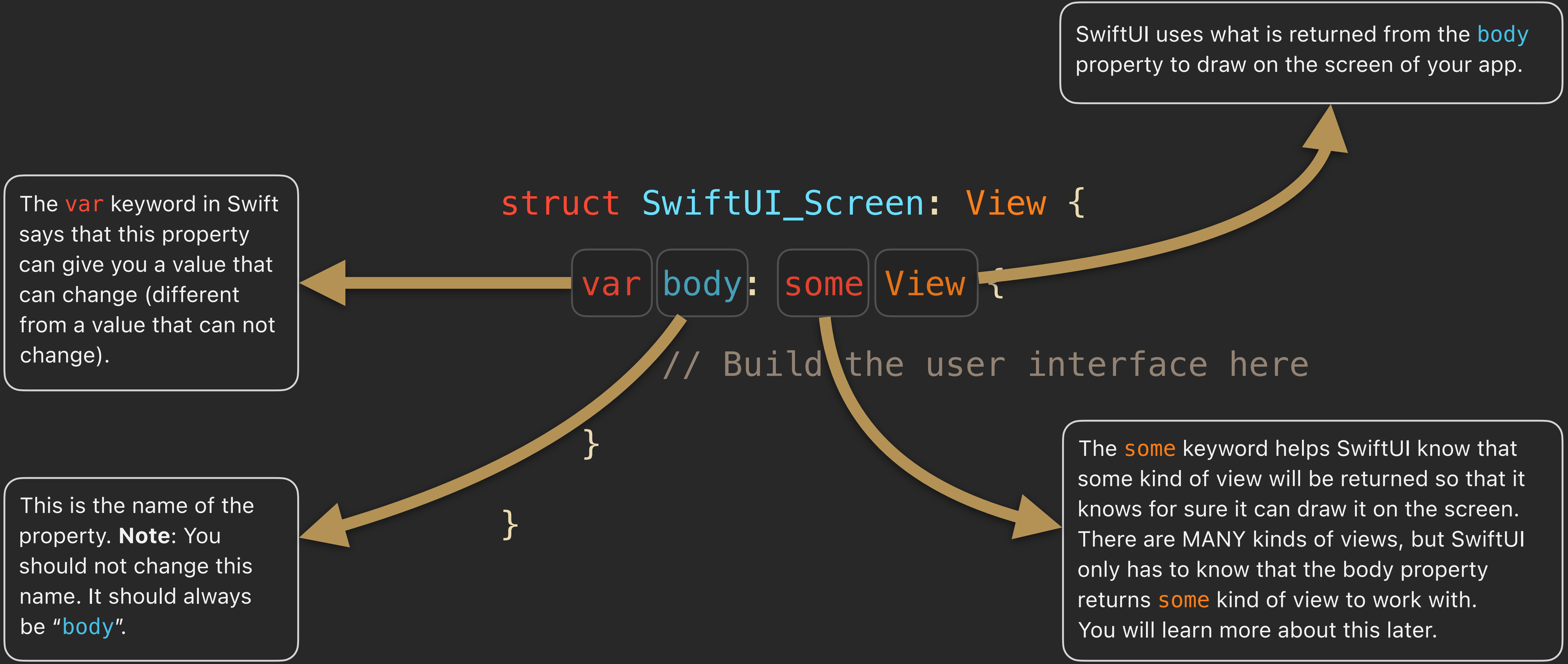
```
struct SwiftUI_Screen: View {
    See next page {

        // Build the user interface here

    }
}
```

?

Are all views structs?
Yes. This makes them very fast to redraw the screen when something changes or when doing animations.

The Body Property

All views require one property: body.



```swift
struct SwiftUI_Screen: View {
    var body: some View {
        // Build the user interface here
    }
}
```

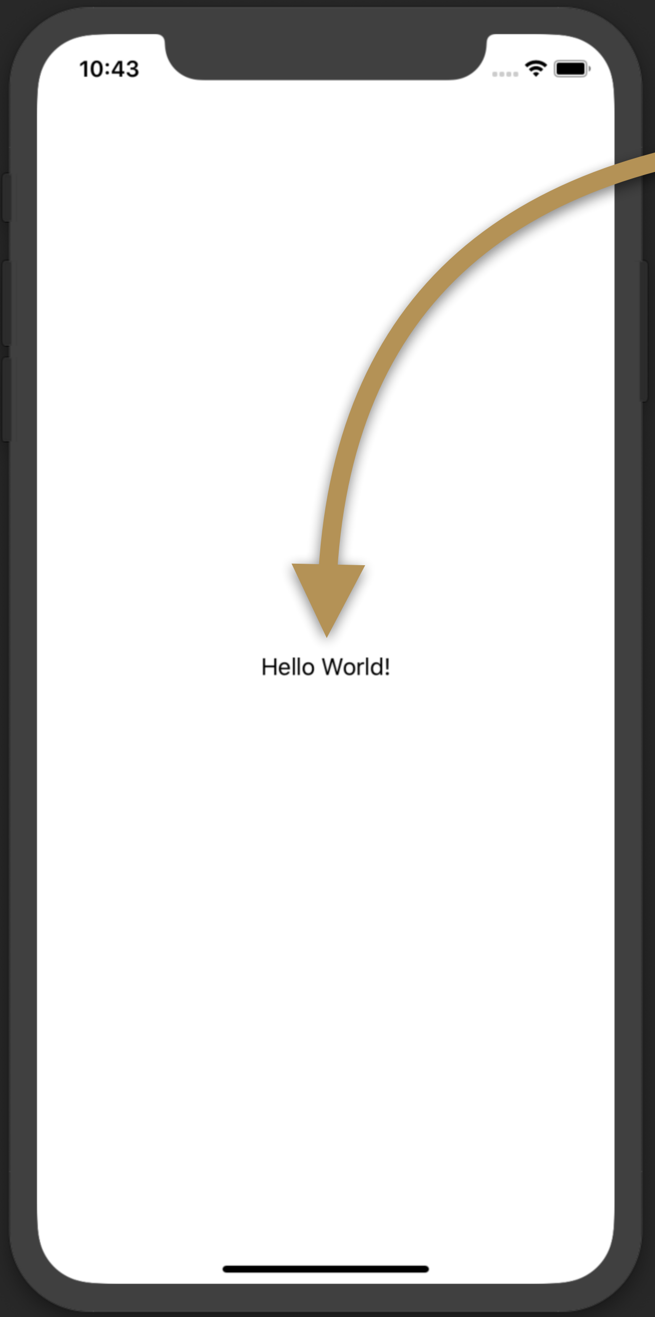
The var keyword in Swift says that this property can give you a value that can change (different from a value that can not change).

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

SwiftUI uses what is returned from the body property to draw on the screen of your app.

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

Views in SwiftUI are structs that conform to the View protocol.

There is just one property to implement, the **body** property.

This is where you design your screens.

?

If "body" is a property, then where is the "get" and "return" syntax?
In Swift, these are optional under certain circumstances.

If not needed, they are usually not used to keep the code clean or simpler.

Property Getters

Properties can have a getter and setter. But when a property has no setter, it's called a "**read-only**" property.
And when the property does not store a value, it is called a "**computed**" property.
This is because the value is computed or generated every time the property is read.
In this example, personType is a **computed read-only** property.

```
struct Person {
    // Computed read-only property (no setter, value is not stored)
    var personType: String {
        get {
            return "human"
        }
    }
}
```

You can further simplify this property in two ways:

1. When the code inside the get is a single expression (one thing), the getter will just return it automatically. You can remove return.
   See "Change 1" in the code example.

2. When a property is read-only (no setter), we can remove the get.
   Just know that these changes are **optional**.

```
// Change 1 - Remove the return
struct Person {
    var personType: String {
        get {
            "human"
        }
    }
}
```

```
// Change 2 - Remove the get
    var personType: String {
        "human"
    }
}
```

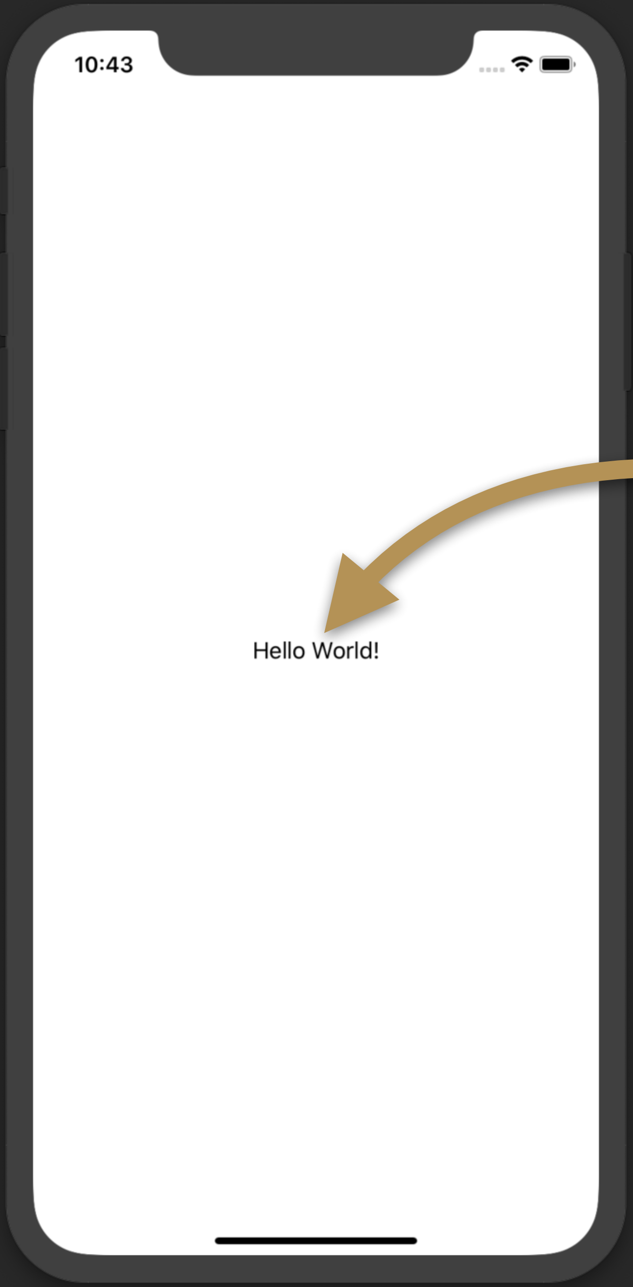
Now, when looking at this property again, you can better understand and see that it is written without the extra get and return keywords.

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

SwiftUI With Property Getters

Since these property changes are optional, you can, for example, write the previous SwiftUI syntax with a get and return inside the **body** property. This might look more familiar to you now:

```swift
// SwiftUI with the get and return keywords
struct BasicSyntax: View {
    var body: some View {
        get {
            return Text("Hello World!")
        }
    }
}
```

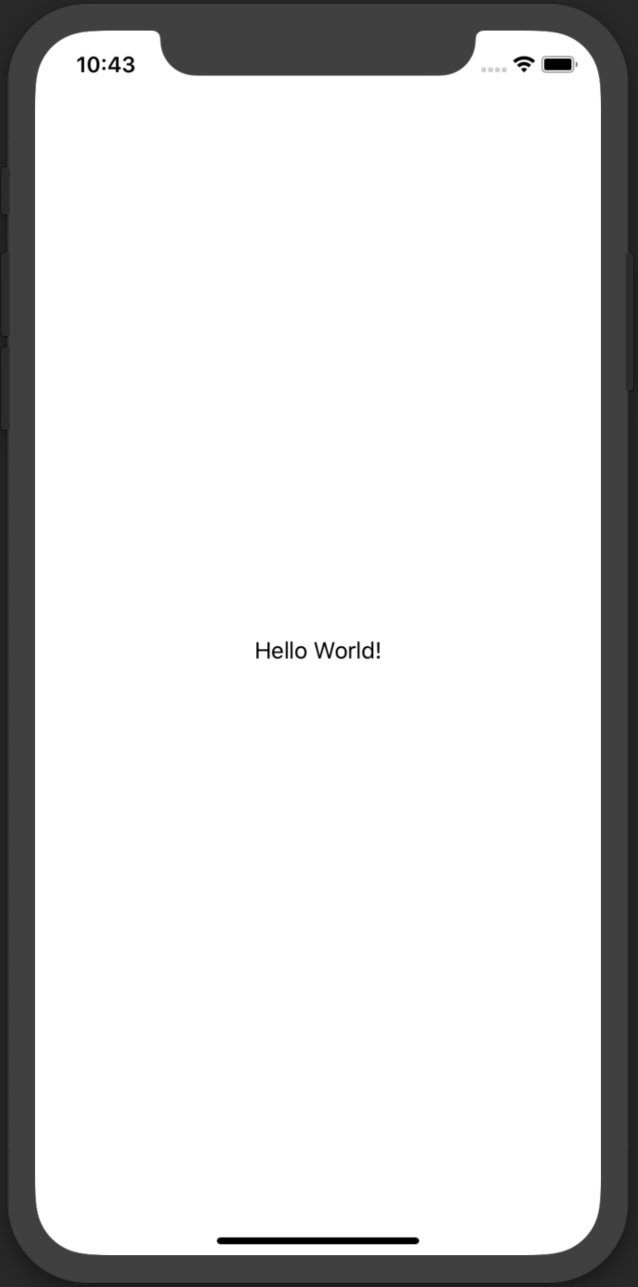


Looking at this code again, you notice the some keyword here.

Normally, when defining a type for a property, you wouldn't see this word.

So, what does the some keyword do?

Opaque Types



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

The View returned from the body property is called an "Opaque Type".

The keyword some is specifying that it is an opaque type.

?

What exactly is an opaque type?

Well, the English definition for the word "opaque", when referring to languages, means "hard or impossible to understand."

In the code example, it is "hard or impossible to understand" which type of view is being returned but at least you know it is a view.

When this View (BasicSyntax) is used by iOS to draw the screen, it doesn't have to know that the type Text is being returned. It is OK with just knowing that some View is being returned and can use that view to draw the screen.

And so you can return anything in that body property as long as it conforms to the View protocol.

For more information on Opaque Types, I recommend referring to the Swift Programming Language documentation.

Understanding Opaque Types

Here is a way to help you understand opaque types.

Let's say you have a box of books.

You know there are books inside, but you don't know exactly which books they are. You can't see the titles of the books.

That is like an opaque type in Swift.

It hides the details of what is inside, but you know they are books and you can do things with them, like put them on the book shelf.

Box of Views

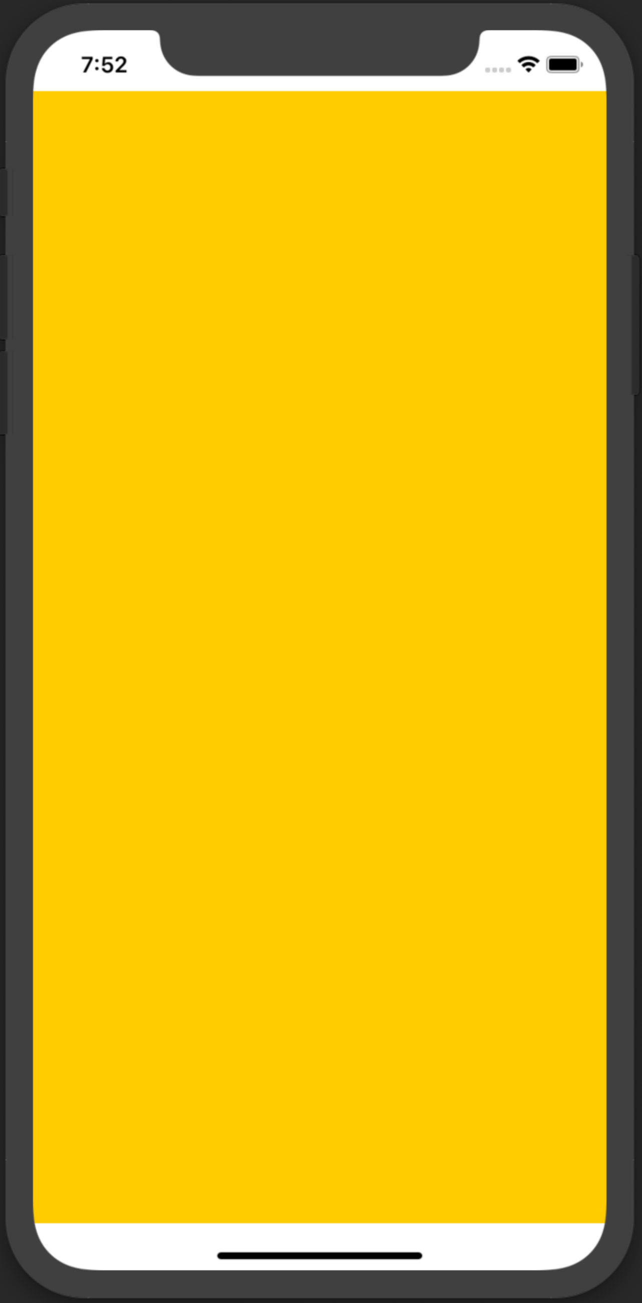
In this case, you have a box of different views.

They could be text, images, views for layout, etc.

You don't care which specific type it is as long as it is a View.

There is another important thing to know about opaque types too...

Opaque Types (some Keyword)



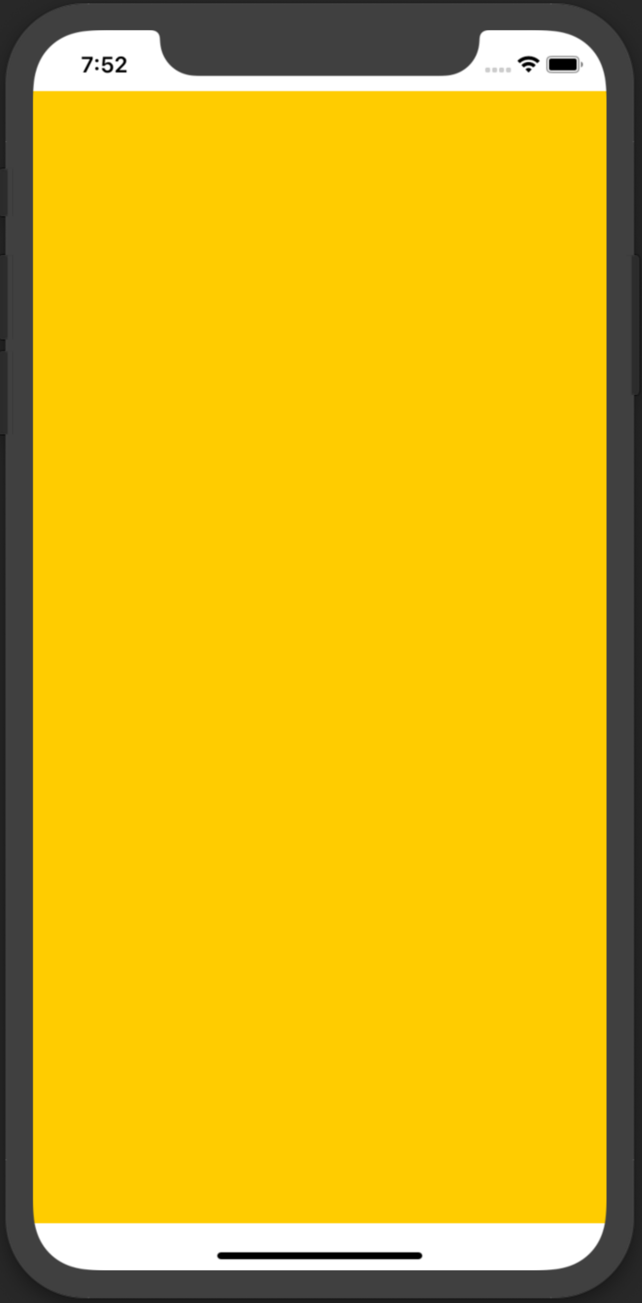
You already know from the previous page that what is returned from the body property is something that conforms to the View protocol.
But what you also need to know is when returning an opaque type (using the some keyword), is that **all possible return types must all be of the same type**.
In most cases you are only returning one type. But you might have a scenario like this:

```swift
struct UnderstandingTheSomeKeyword: View {
    var isYellow = true

    // The keyword "some" tells us that whatever we return, it has to:
    // 1. Conform to the View protocol
    // 2. Has to ALWAYS be the same type of View that is returned.
    var body: some View {

        // ERROR: Function declares an opaque return type, but the return statements
           in its body do not have matching underlying types
        if isYellow {
            return Color.yellow // Color type does not match the Text type
        }

        return Text("No color yellow") // Text type does not match the color type
    }
}
```

The body property returns a Color and a Text type. This violates the some keyword.

Opaque Types Solution

The solution would be to change the views returned so they are **all the same TYPE**. The body now returns the same type of view (Color).

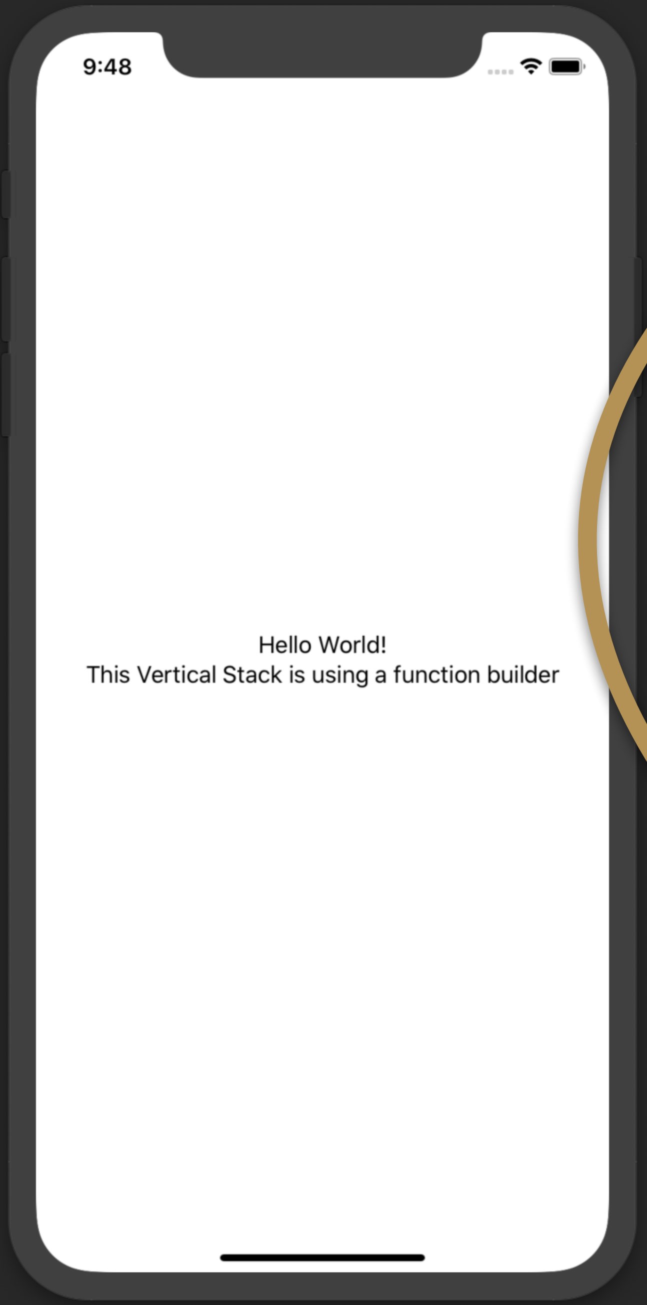


```
struct UnderstandingTheSomeKeywordSolution: View {
    var isYellow = true

    // The keyword "some" tells us that whatever we return, it has to:
    // 1. Conform to the View protocol
    // 2. Has to ALWAYS be the same type of View that is returned.

    var body: some View {

        if isYellow {
            return Color.yellow
        }

        return Color.clear
    }
}
```

✅

Now, the body property always returns a Color type. This satisfies the some keyword.

View Containers



Hello World!
This Vertical Stack is using a function builder

```
struct Example: View {
    var body: some View {
        VStack {
            Text("Hello World!")
            Text("This Vertical Stack is using a function builder")
        }
    }
}
```

Note: If you don't specify a VStack here, SwiftUI will use one by default.

So far, you have learned that body is a computed read-only property and can only return ONE object that is some View. **What if you want to show multiple views though?**

You learned earlier about the concept of "container" views. These are views that can contain other views. Remember, the body property can only return **one** view. You will get an error if you try to return more than one view in the body property.

In the example above, the VStack (Vertical Stack) is that one view being returned. And that vertical stack is a container with two more views inside of it.

The VStack is using a "trailing closure," which just means that it is a code block that is passed into the initializer to be run by the VStack. You have probably seen this before in Swift, this is not new.

What is new in Swift is the ability to create multiple, new views within the constructor like this. Before we get into this though, let's better understand how this constructor works.

View Container Initialization

In Swift, you usually see parentheses during initialization but **with a trailing closure, the parentheses are optional.**
You can add them and the code will still work just fine.

See "Change 1" in the code example.

This change may start looking more familiar to you.

```swift
struct Example: View {
    var body: some View {
        VStack {
            Text("Hello World!")
            Text("This Vertical Stack is using a function builder")
        }
    }
}

// Change 1 - Add parentheses and parameter name
struct Example: View {
    var body: some View {
        VStack(content: {
            Text("Hello World!")
            Text("This Vertical Stack is using a function builder")
        })
    }
}
```

?

How does the VStack know how to accept the multiple views like this?
This is new in Swift. To better understand this, take a look at the VStack's initializer.

The alignment and spacing parameters are optional, that is why you don't see them in the examples above. But notice before the content parameter there is @ViewBuilder syntax.

This is what allows you to declare multiple views within the content parameter's closure.

```swift
// VStack initializer
init(alignment: HorizontalAlignment = .center,
     spacing: CGFloat? = nil,
     @ViewBuilder content: () -> Content)
```

@ViewBuilder Parameter Attribute

The @ViewBuilder parameter attribute allows Swift to build multiple child views from within a closure.

?

How many child views can I build within a closure?

Before iOS 17 there was a limit of 10 views. With iOS 17, there is no longer a limit.

?

What if I need to support iOS 16 or prior?

Then you will be limited to 10 views.

You can workaround this by nesting more views though. Take a look at this example.

```
struct ViewBuilderExample: View {
    var body: some View {
        VStack {
            Text("View 1")
            Text("View 2")
            Text("View 3")
            Text("View 4")
            Text("View 5")
            Text("View 6")
            Text("View 7")
            Text("View 8")
            Text("View 9")
            Text("View 10")
            Text("View 11") // Will no longer cause an error
        }
    }
}
```

iOS 17

You can now have more than 10 views.

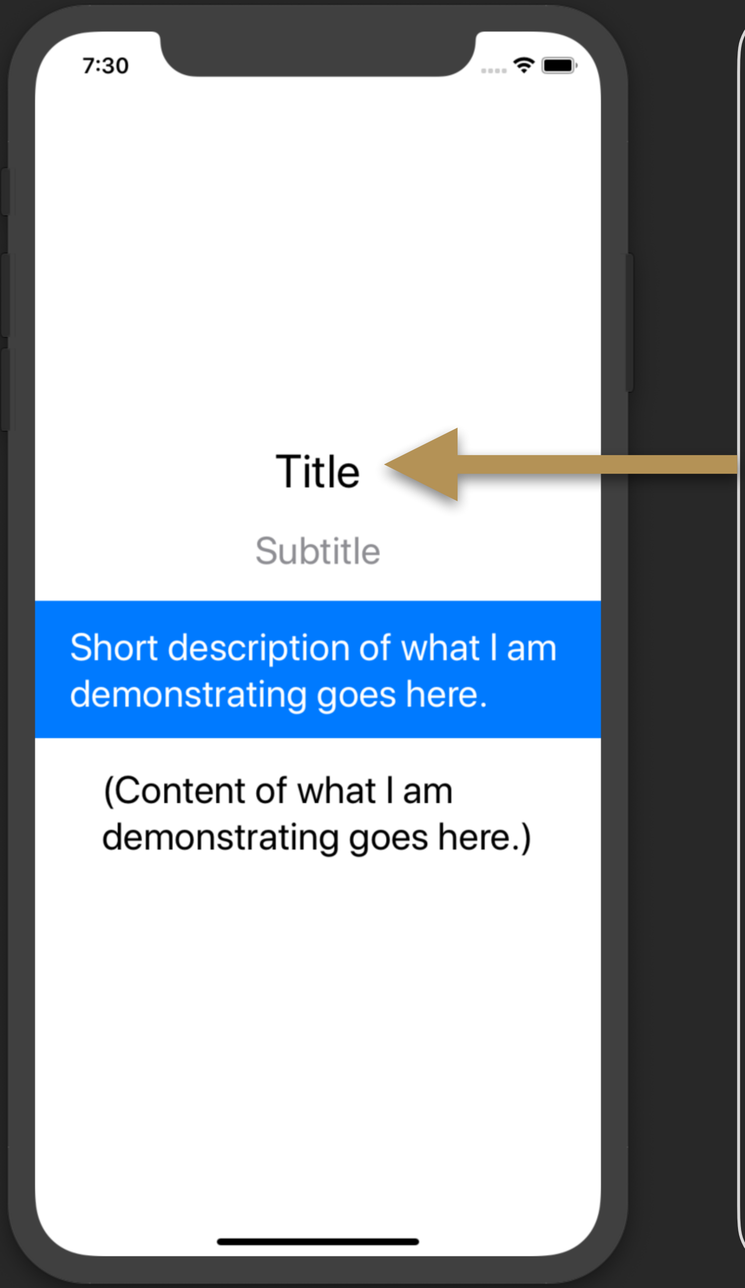
```
struct ViewBuilderExample: View {
    var body: some View {
        VStack {
            ... // Text views 1 - 5
            Text("View 6")
            Text("View 7")
            Text("View 8")
            Text("View 9")
            VStack {
                Text("View 10")
                Text("View 11")
            }
        }
    }
}
```

My Template

If you are completely new to SwiftUI you may wonder what a lot of this code means right at the beginning of the book. I have “templates” that contains a title, subtitle and a short description on most SwiftUI screens.

I will take you through step-by-step on how I build this template that I use throughout the book. I will describe each one only briefly because each modifier I apply to the views here are described in more detail throughout the book within their own sections.

My Basic Template


7:30

Title

Subtitle

Short description of what I am demonstrating goes here.

(Content of what I am demonstrating goes here.)

Here is my basic template I use throughout the book to explain views and modifiers.

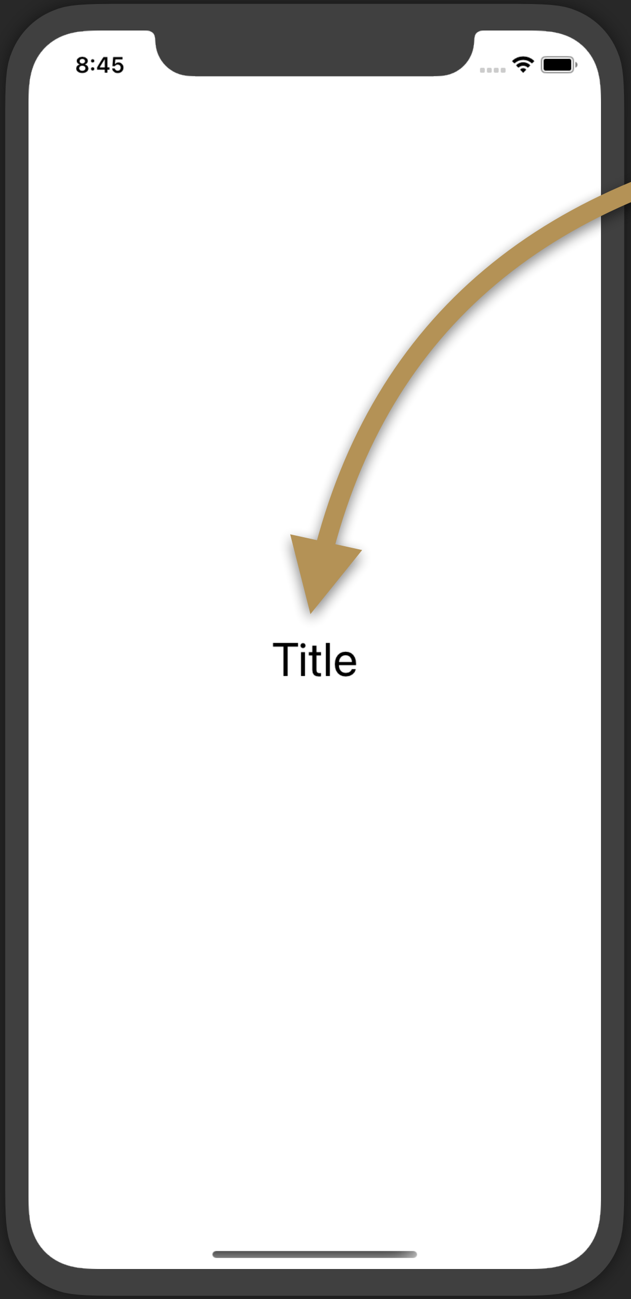
In the next pages I'm going to explain how this is built in SwiftUI. I want to make sure you understand these parts because you will see them everywhere in this book.

I want to remove any confusion right at the beginning so it doesn't get in your way to learning the topics in the book.

Let's start with the title.

Starting with the Title

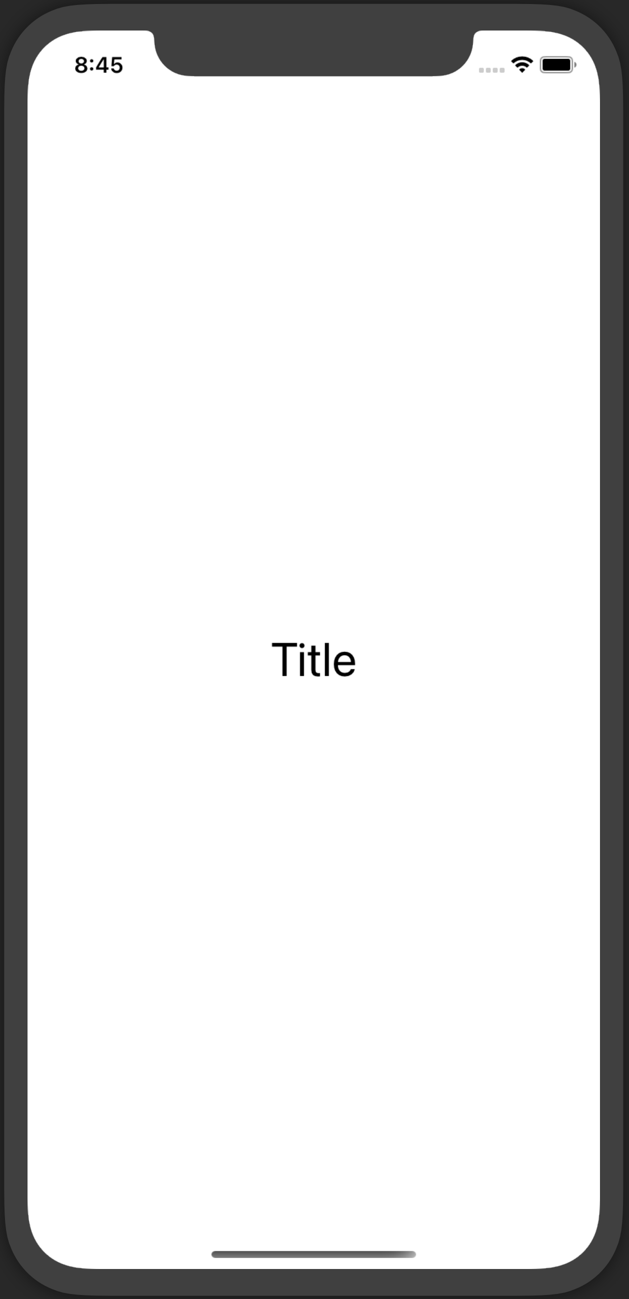
```
struct Title: View {
    var body: some View {
        Text("Title") // Create text on the screen
            .font(.largeTitle) // Use a font modifier to make text larger
    }
}
```



Here, you have a `Text` view. You want to make it larger so you use the `font` modifier so you can set the size to a SwiftUI preset size called `largeTitle` (this is the largest preset size).

There are more ways you can change the size of text that are covered in this book in the **Control Views** chapter, in the section called **Text**.

Add a VStack



```
struct AddVStack: View {
    var body: some View {
        // Only one view can be returned from the body property.
        // Add 20 points between views within this container.
        VStack(spacing: 20) { // VStack is a container view that can hold many views
            Text("Title")
                .font(.largeTitle)
        }
    }
}
```

VStack

The body property can **only return one view**. You will get an error if you have two views.

So, you need to use a container view that will contain multiple views. The vertical stack (VStack) is the perfect choice here.

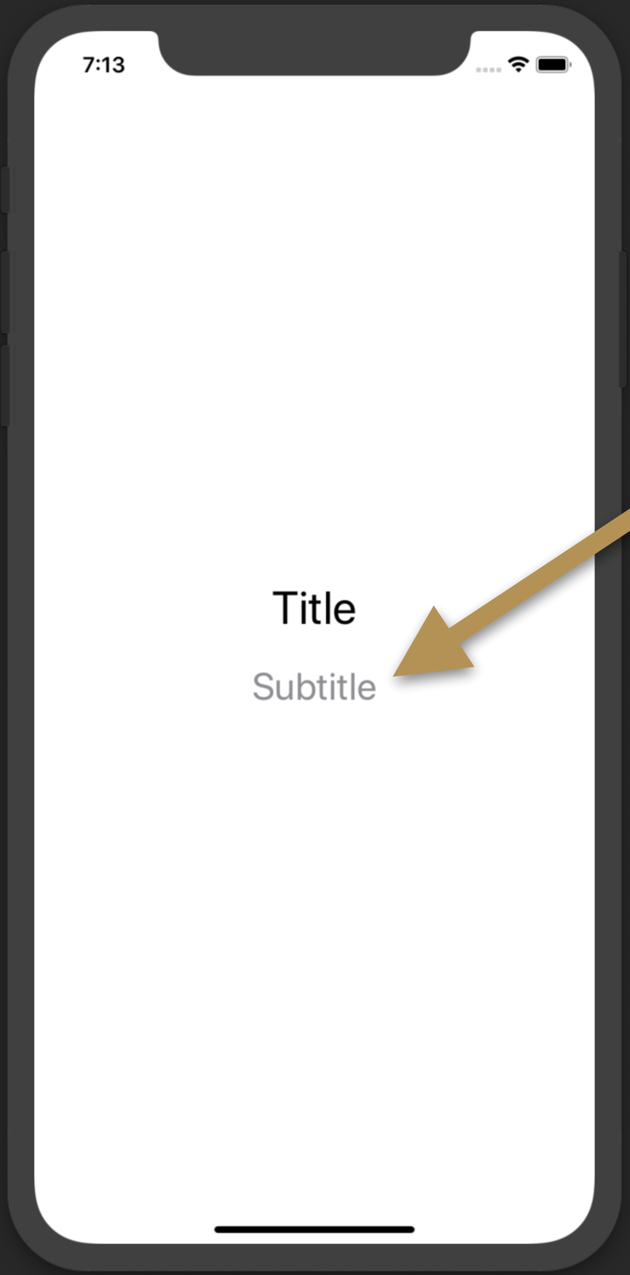
Now you can add more views to the VStack and it will arrange them vertically.

Spacing

The VStack has an optional parameter you can use in its initializer to specify how many points of spacing you want in between views. (***Note****: spacing does not add spacing to the top or bottom of the VStack.*)

Now, let's add the subtitle text.

Adding the Subtitle



```
struct Subtitle: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Title")
                .font(.largeTitle)

            Text("Subtitle")
                .font(.title) // Set to be the second largest font.
                .foregroundColor(Color.gray) // Change text color to gray.
        }
    }
}
```

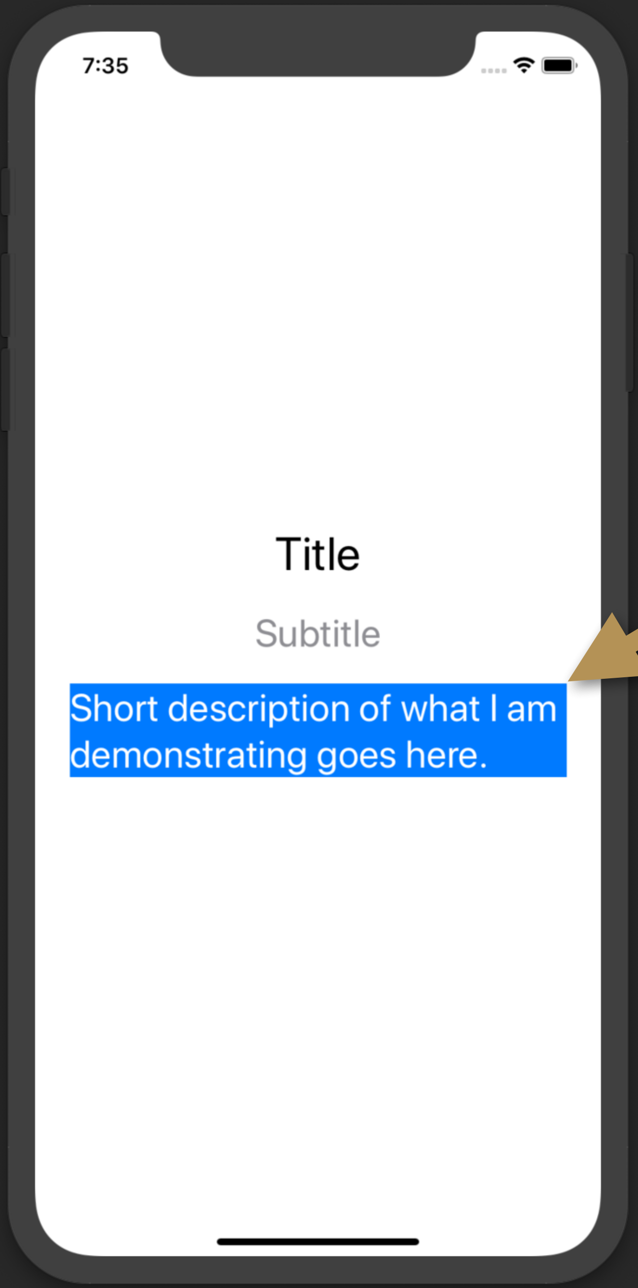
Subtitle

The subtitle is another text view. This time, you set the size to be the second largest preset size with the title parameter.

Finally, you modify the view to change the text color to gray. (***Note***: *instead of using Color.gray you can also use just .gray.*)

Now, let's add the description text.

Add the Description with a Background Color



```
struct Description1: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Title")
                .font(.largeTitle)

            Text("Subtitle")
                .font(.title)
                .foregroundColor(.gray)

            Text("Short description of what I am demonstrating goes here.")
                .font(.title)
                .foregroundColor(Color.white)
                .background(Color.blue) // Add the color blue behind the text.
        }
    }
}
```

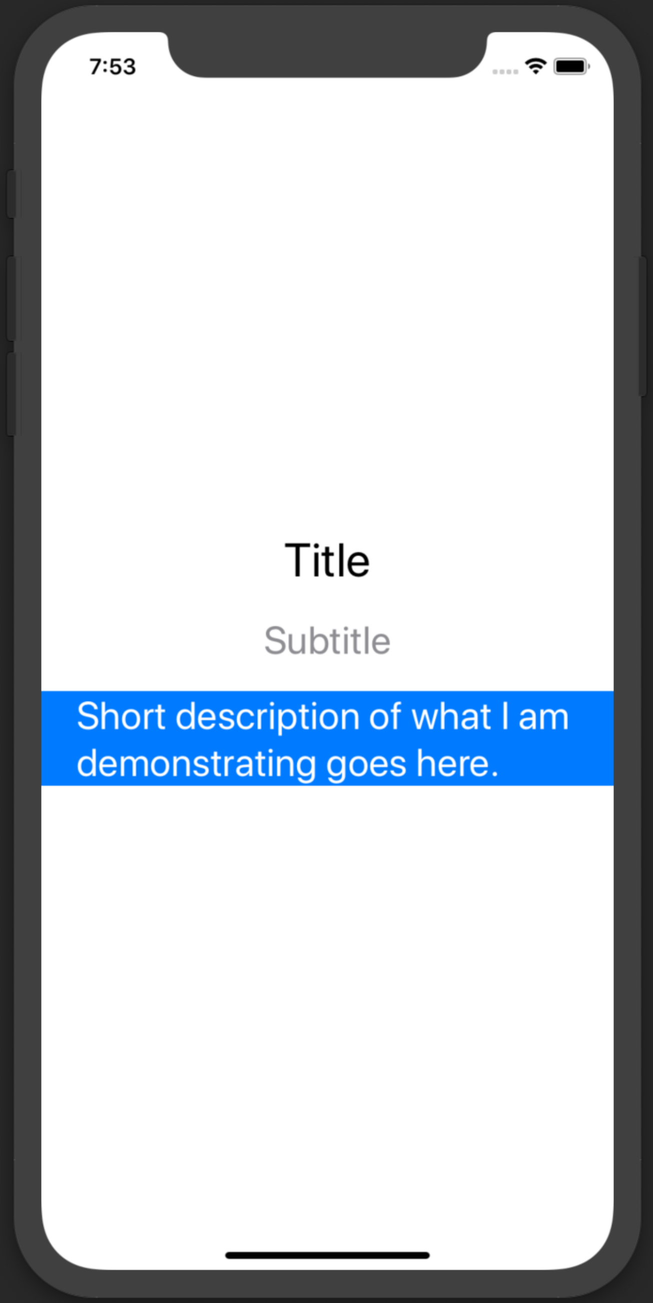
With the description text view, you are now familiar with the `font` and `foregroundColor` modifiers. But now you want to add a color behind the text. So you use the `background` modifier to set a color.

The important thing to notice here is it is not a background**Color** modifier. That does not exist. It is a `background` modifier because it adds a layer **behind** the view.

`Color.blue` is actually a view. So the `background` modifier is adding a blue view on a layer behind the text.

We want this view to extend to the edges of the screen. So let's add that next.

Adding a Frame Modifier

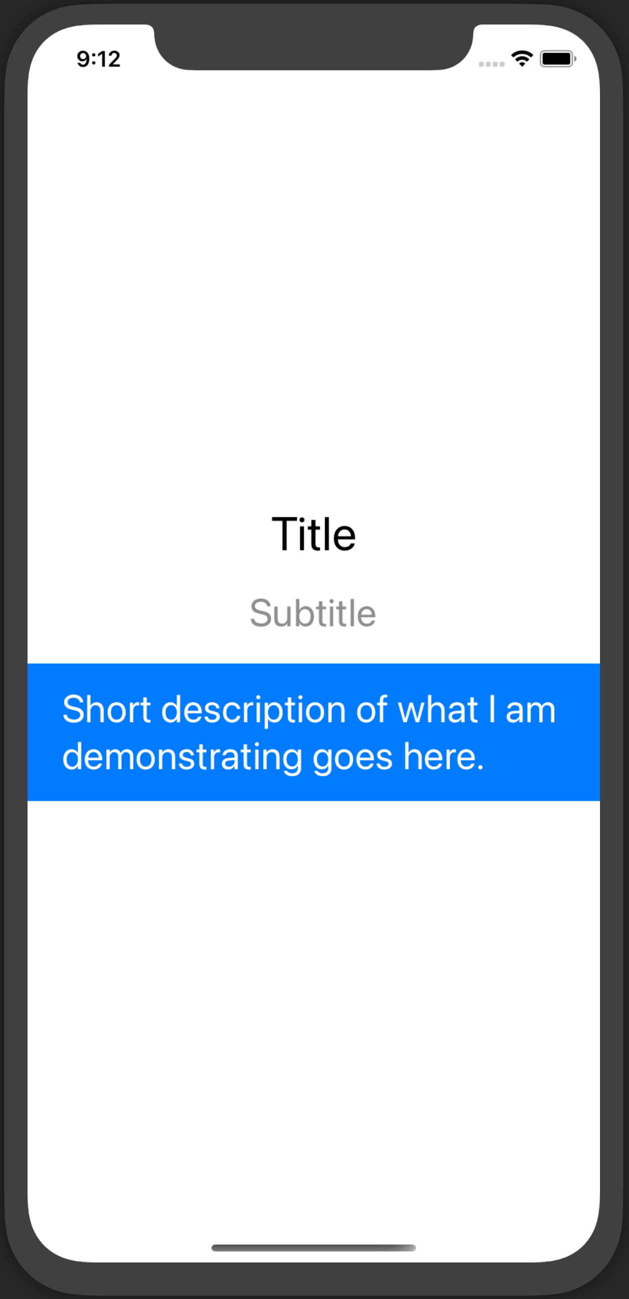


```
struct Description2: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Title")
                .font(.largeTitle)

            Text("Subtitle")
                .font(.title)
                .foregroundColor(.gray)

            Text("Short description of what I am demonstrating goes here.")
                .frame(maxWidth: .infinity) // Extend until you can't go anymore.
                .font(.title)
                .foregroundColor(Color.white)
                .background(Color.blue)
        }
    }
}
```

To extend the text to the edges of the device, we use the `frame` modifier. You don't need to set a fixed value. Instead, you can just modify the text view and say its frame's maximum width can extend to `infinity` until it hits its parent's frame and then will stop. Its parent's frame is the `VStack`.

This is looking good. It would look better though if there was more space around the text that pushed out the blue background.

Add Padding Around the Text View

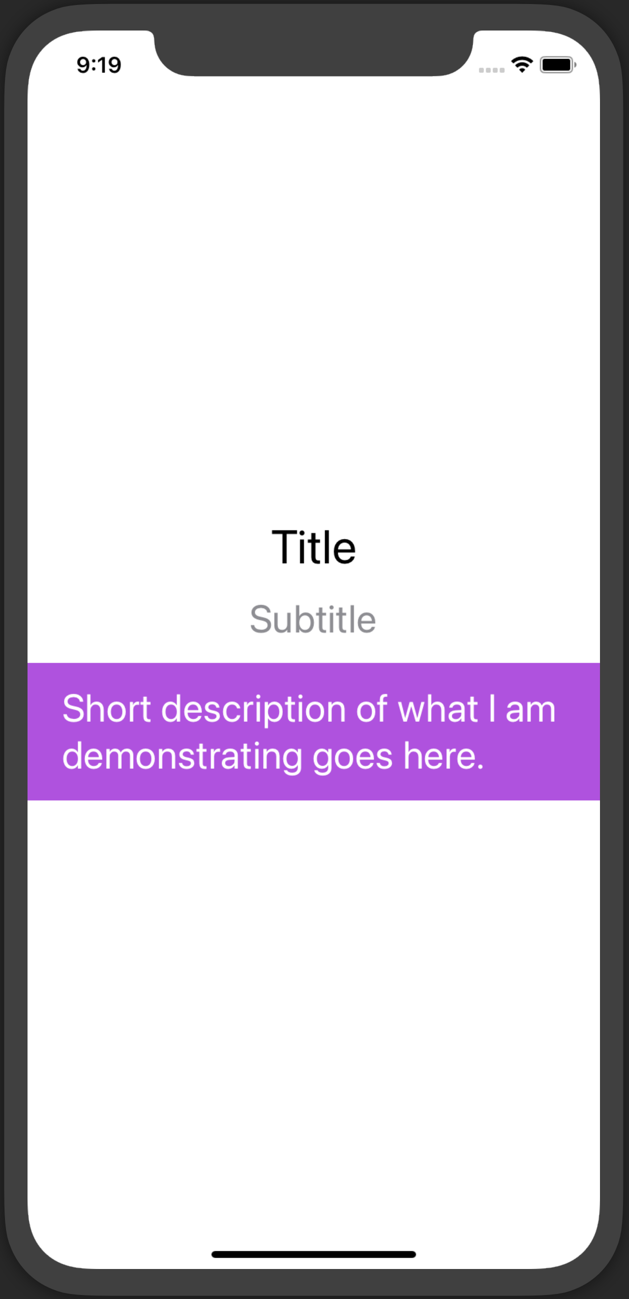


```
struct Description3: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Title")
                .font(.largeTitle)

            Text("Subtitle")
                .font(.title)
                .foregroundColor(.gray)

            Text("Short description of what I am demonstrating goes here.")
                .frame(maxWidth: .infinity)
                .font(.title)
                .foregroundColor(Color.white)
                .padding() // Add space all around the text
                .background(Color.blue)
        }
    }
}
```

Padding

Use the padding modifier to add space around a view. Remember, the order of modifiers matter. You can add the padding modifier anywhere as long as it is BEFORE the background modifier. If it was after the background, it would add space around the blue background. We want the space between the text and the background.

Version 2 of the Template



Title

Subtitle

Short description of what I am demonstrating goes here.

```
struct Version2: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Title",
                       subtitle: "Subtitle",
                       desc: "Short description of what I am demonstrating goes here.",
                       back: .purple, textColor: .white)
        }
        .font(.title)
    }
}
```

Version 2

When I updated the book with SwiftUI 2, I wanted a more efficient way of adding a title, subtitle and description.

So I made my own view, called HeaderView, where I can pass in the information and it will format it.

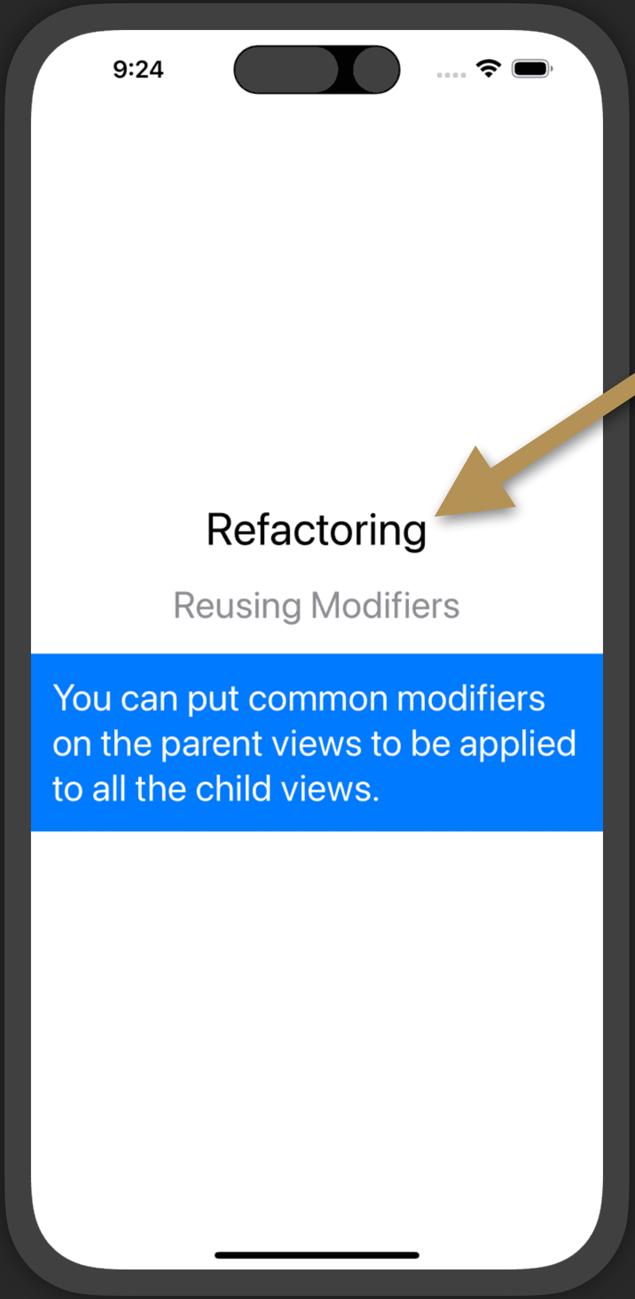
As you can see, this saves repetitive code and space.

If you're interested in how this is done, look in the Xcode project that comes with the paid book bundle for the file “**HeaderView.swift**”.

SwiftUI Basics

Now that you understand this basic template I use for demonstrating topics, I will start using it. Be sure to read what is on each screenshot (or find the text in the code to read).

Scope and Overriding



Refactoring

Reusing Modifiers

You can put common modifiers on the parent views to be applied to all the child views.

```
struct ScopeAndOverriding: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Refactoring")
                .font(.largeTitle)

            Text("Reusing Modifiers")
                ~~.font(.title)~~
                .foregroundStyle(Color.gray)

            Text("You can put common modifiers on the parent views to be
                 applied to all the child views.")
                ~~.font(.title)~~
                .frame(maxWidth: .infinity)
                .foregroundStyle(Color.white)
                .padding()
                .background(Color.blue)
        }
        .font(.title)
    }
}
```

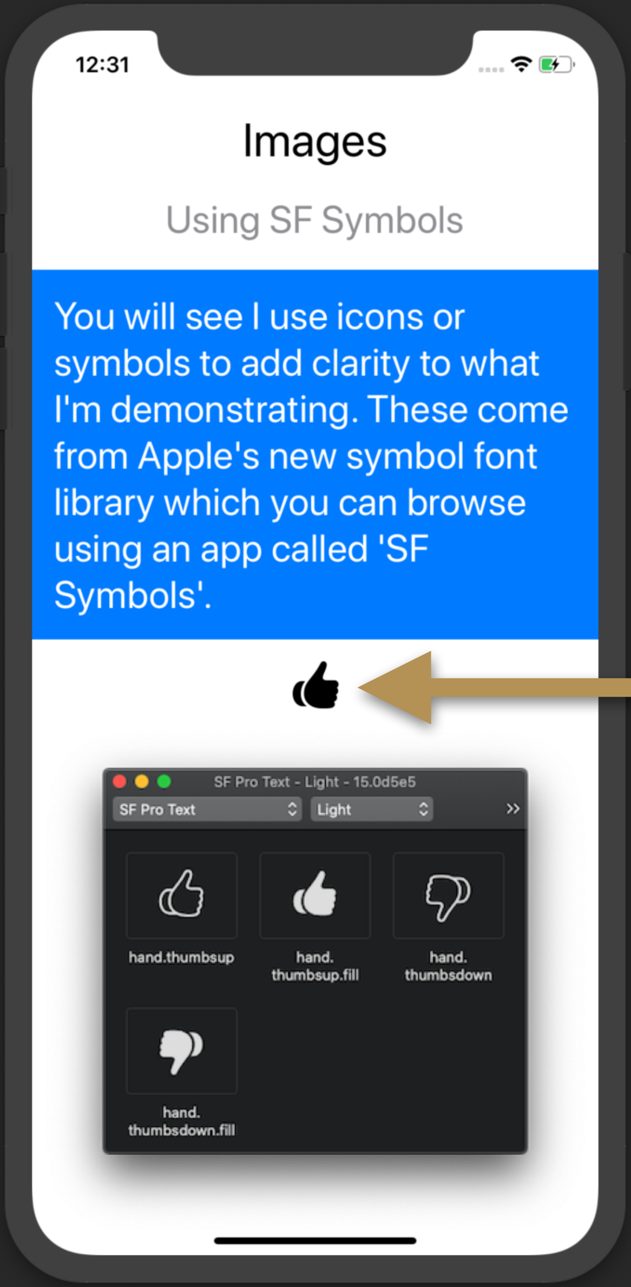
?

Why isn't the first Text view affected?
This Text view has its own font modifier. This means it overrides the parent's font modifier.

Look for duplicate modifiers and move them to the parent to apply to all views within.

Moving this font modifier to the VStack will apply the `title` style to all Text views within.

Short Introduction to Symbols



Images

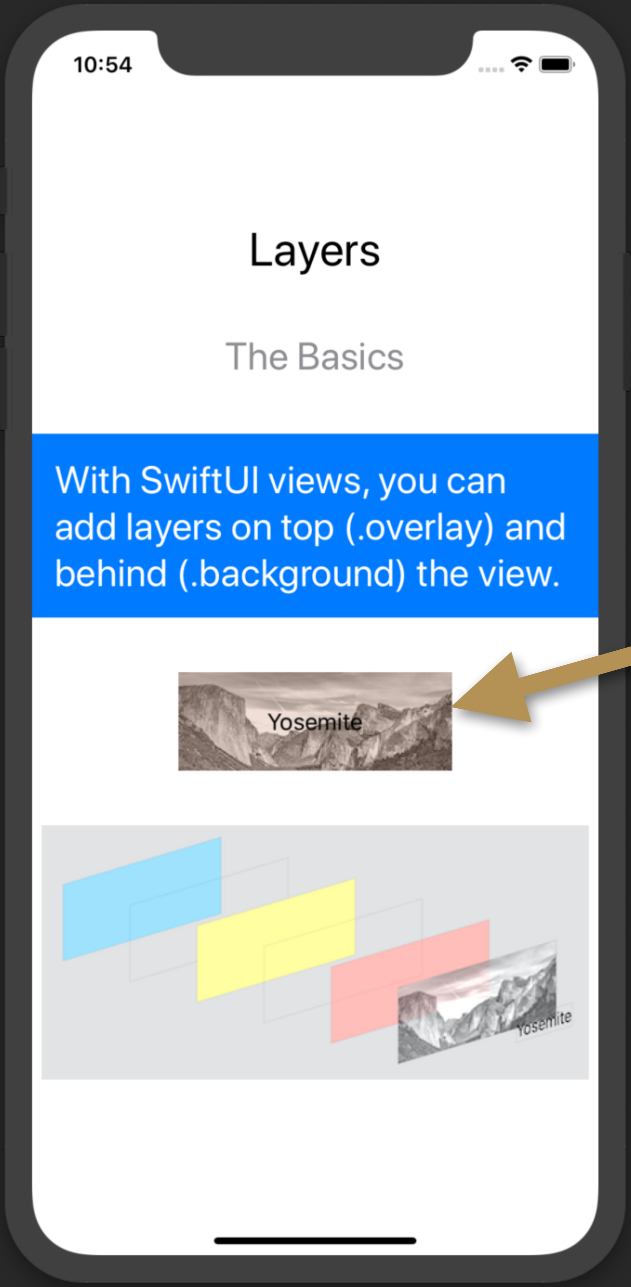
Using SF Symbols

You will see I use icons or symbols to add clarity to what I'm demonstrating. These come from Apple's new symbol font library which you can browse using an app called 'SF Symbols'.

```
struct SymbolsIntro: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Images")
                .font(.largeTitle)
            Text("Using SF Symbols")
                .foregroundColor(.gray)

            Text("You will see I use icons or symbols to add clarity to what I'm demonstrating.
                These come from Apple's new symbol font library which you can browse using an
                app called 'SF Symbols'.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.blue)
                .foregroundColor(Color.white)

            // Use "systemName" when you want to use "SF Symbols"
            Image(systemName: "hand.thumbsup.fill")
                .font(.largeTitle) // Make the symbol larger

            Image("SF Symbols") // Regular image from Assets.xcassets
        }
        .font(.title)
        .ignoresSafeArea(edges: .bottom) // Ignore the bottom screen border
    }
}
```

Even though an `Image` view is used to initialize a symbol, you use the `font` modifier to change its size. These symbols actually come from fonts. So use font modifiers to change them. There is a whole section that covers this. Go here to download and install the SF Symbols app.

Layers


10:54

Layers

The Basics

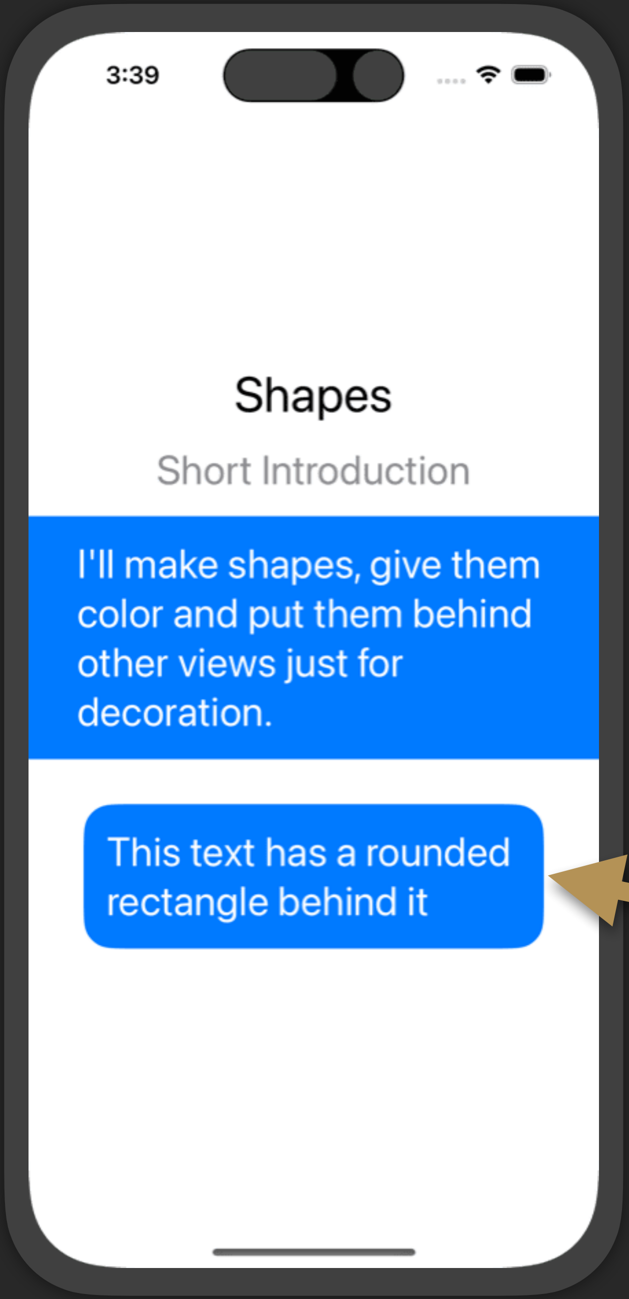
With SwiftUI views, you can add layers on top (.overlay) and behind (.background) the view.

```
VStack(spacing: 40) {
    Text("Layers")
        .font(.largeTitle)

    Text("The Basics")
        .foregroundColor(.gray)

    Text("With SwiftUI views, you can add layers on top (.overlay) and behind (.background) the 
view.")
        .frame(maxWidth: .infinity)
        .padding()
        .background(Color.blue)
        .foregroundColor(Color.white)

    Image("yosemite") // Show an image from Assets.xcassets
        .opacity(0.7) // Make image only 70% solid
        .background(Color.red.opacity(0.3)) // Layer behind image
        .background(Color.yellow.opacity(0.3)) // Layer behind red
        .background(Color.blue.opacity(0.3)) // Layer behind yellow
        .overlay(Text("Yosemite")) // Layer on top of image

    Image("Layers")
}
.font(.title)
```

I use layers (`background` and `overlay`) early in the book so I want to make sure you understand this concept.

Both of these modifiers are explained in detail in their own sections.

Short Introduction to Shapes



Shapes

Short Introduction

I'll make shapes, give them color and put them behind other views just for decoration.

This text has a rounded rectangle behind it

```
struct Shapes: View {
    var body: some View {
        VStack(spacing: 15) {
            Text("Shapes")
                .font(.largeTitle)

            Text("Short Introduction")
                .foregroundStyle(Color.gray)

            Text("I'll make shapes, give them color and put them behind other views just for
                 decoration.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.blue)
                .foregroundStyle(Color.white)

            Text("This text has a rounded rectangle behind it")
                .foregroundStyle(Color.white)
                .padding()
                .background {
                    RoundedRectangle(cornerRadius: 20) // Create the shape
                        .fill(Color.blue) // Make the shape blue
                }
                .padding()
        }
        .font(.title)
    }
}
```

RoundedRectangle is a common shape.

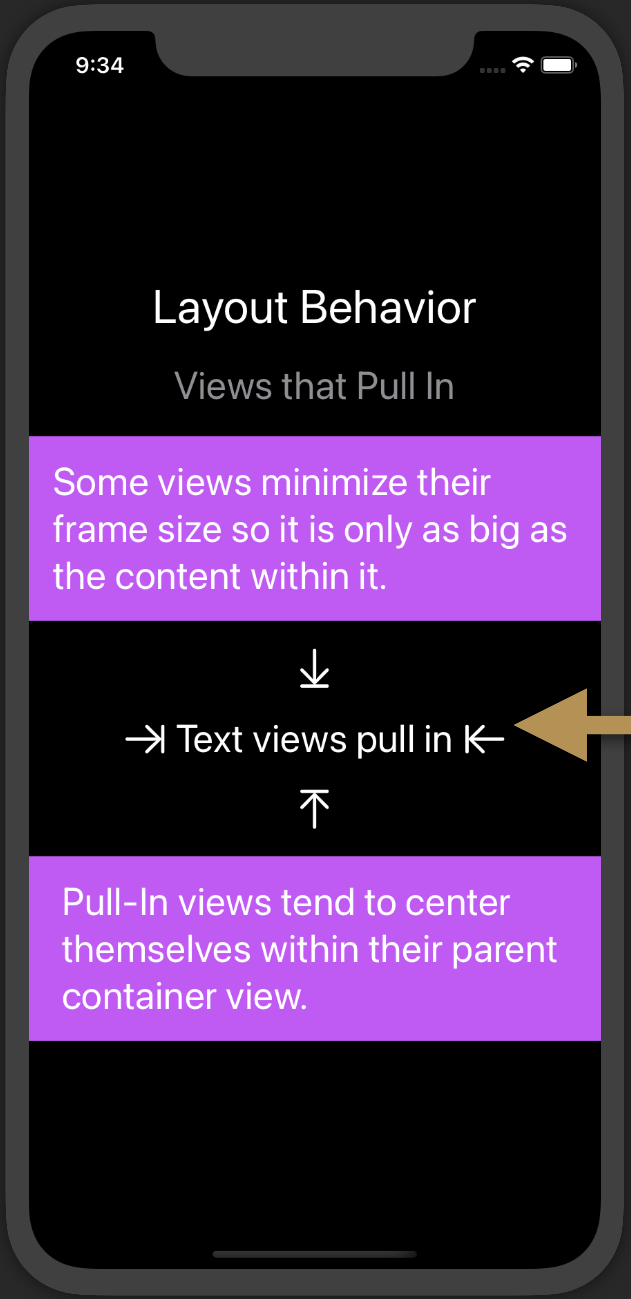
Layout Behavior

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

1. Some views pull in to be as small as possible to fit their content. (I will refer to these as “pull-in” views.)
2. Some views push out to fill all available space. (I will refer to these as “push-out” views.)

Knowing these two behaviors can help you predict what will happen when using the different views.

Some Views Pull In



Layout Behavior

Views that Pull In

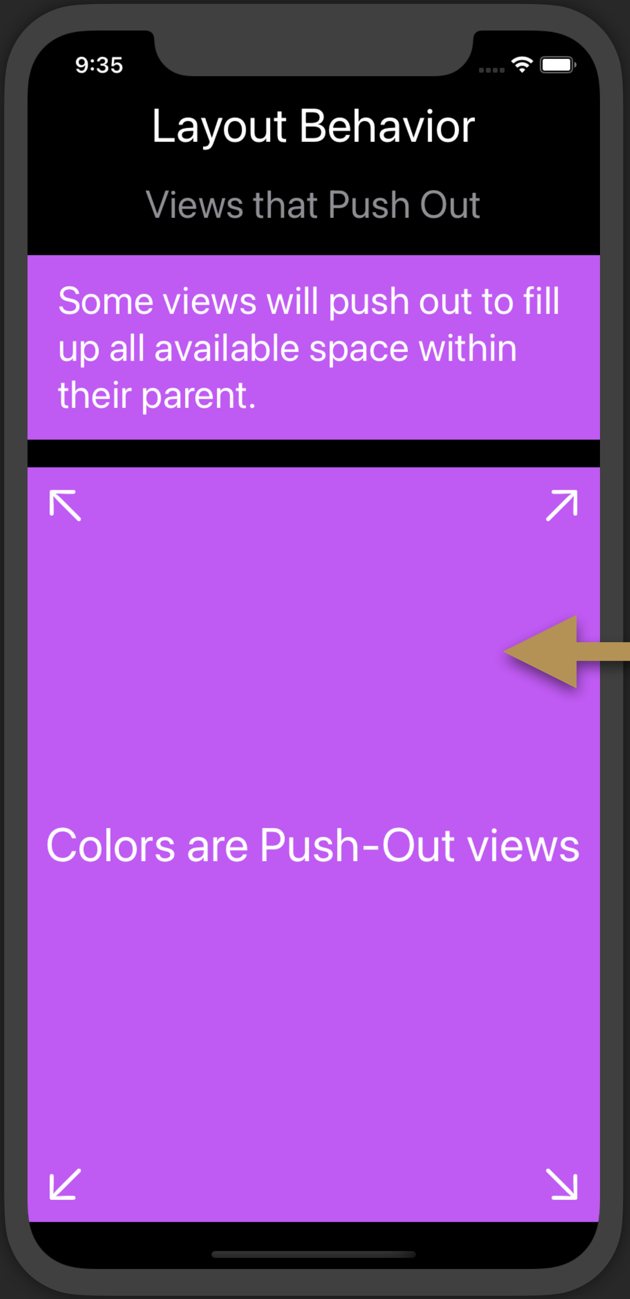
Some views minimize their frame size so it is only as big as the content within it.

Text views pull in

Pull-In views tend to center themselves within their parent container view.

```
struct ViewSizes_Pull_In: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Layout Behavior").font(.largeTitle)
            Text("Views that Pull In").foregroundColor(.gray)
            Text("Some views minimize their frame size so it is only as big as the
                 content within it.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.purple)
                .foregroundColor(Color.white)

            Image(systemName: "arrow.down.to.line.alt")

            HStack { // Order views horizontally
                Image(systemName: "arrow.right.to.line.alt")
                Text("Text views pull in")
                Image(systemName: "arrow.left.to.line.alt")
            }

            Image(systemName: "arrow.up.to.line.alt")

            Text("Pull-In views tend to center themselves within their parent container
                 view.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.purple)
                .foregroundColor(Color.white)
        }.font(.title)
    }
}
```

Some Views Push Out



Layout Behavior

Views that Push Out

Some views will push out to fill up all available space within their parent.

Colors are Push-Out views

```
struct ViewSizes_Push_Out: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Layout Behavior")
            Text("Views that Push Out")
                .font(.title).foregroundColor(.gray)
            Text("Some views will push out to fill up all available space within their parent.")
                .frame(maxWidth: .infinity).padding().font(.title)
                .background(Color.purple)

            Color.purple
                // Add 5 layers on top of the color view
                .overlay(
                    Image(systemName: "arrow.up.left")
                        .padding() // Add spacing around the symbol
                    , alignment: .topLeading) // Align within the layer
                .overlay(
                    Image(systemName: "arrow.up.right")
                        .padding(), alignment: .topTrailing)
                .overlay(
                    Image(systemName: "arrow.down.left")
                        .padding(), alignment: .bottomLeading)
                .overlay(
                    Image(systemName: "arrow.down.right")
                        .padding(), alignment: .bottomTrailing)
                .overlay(Text("Colors are Push-Out views"))
        }.font(.largeTitle) // Make all text and symbols larger
    }
}
```

Colors are push-out views.

For the most part, I will be telling you if a view is a push-out view or a pull-in view at the beginning of the sections.

SEE YOUR WORK

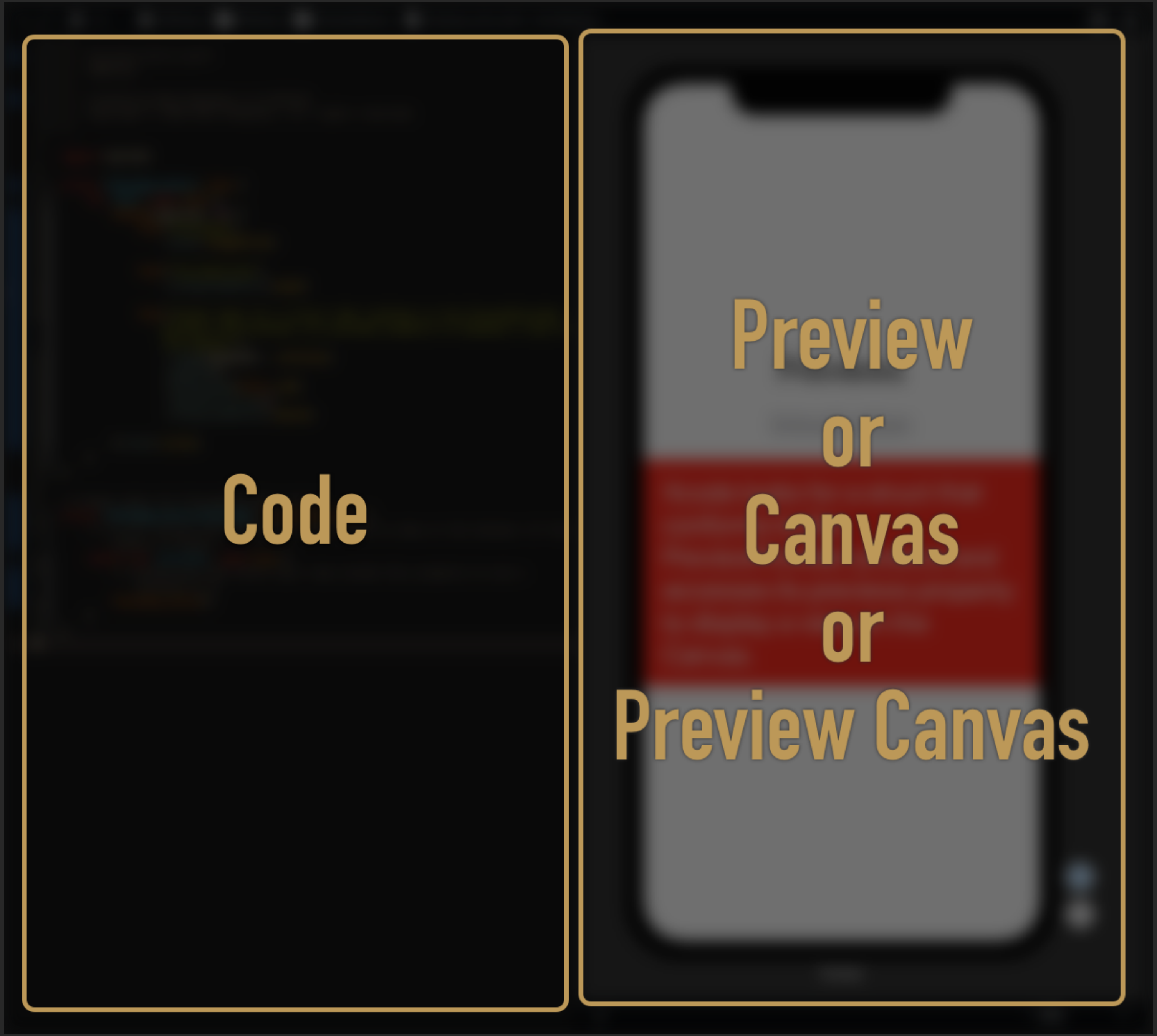
Preview Options

As you practice these examples, you might want to see your SwiftUI working on different devices in different modes, including light or dark mode or with different accessibility settings.

You can do all of this **without** even having to launch the Simulator.

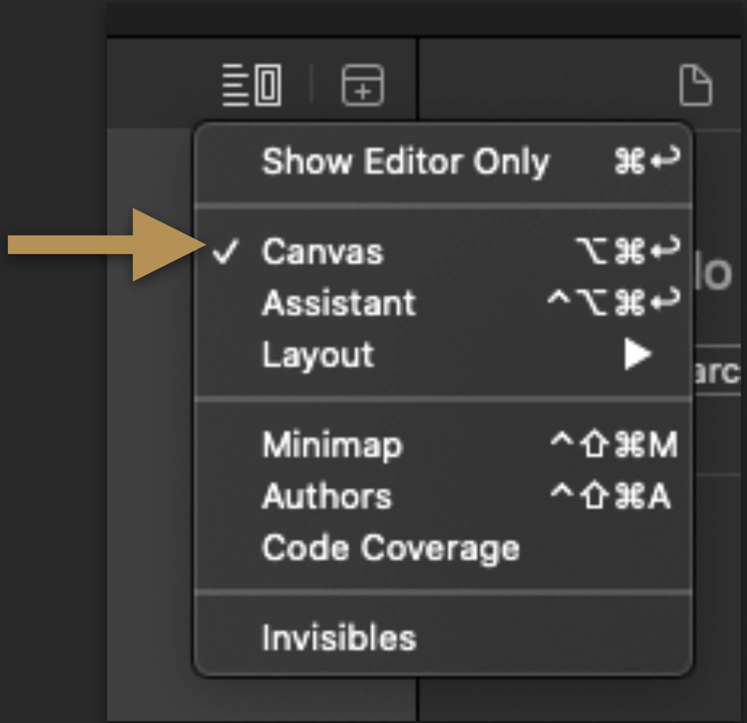
When using SwiftUI, you get a preview canvas that will show you how your views will render.

The Canvas - What is it?

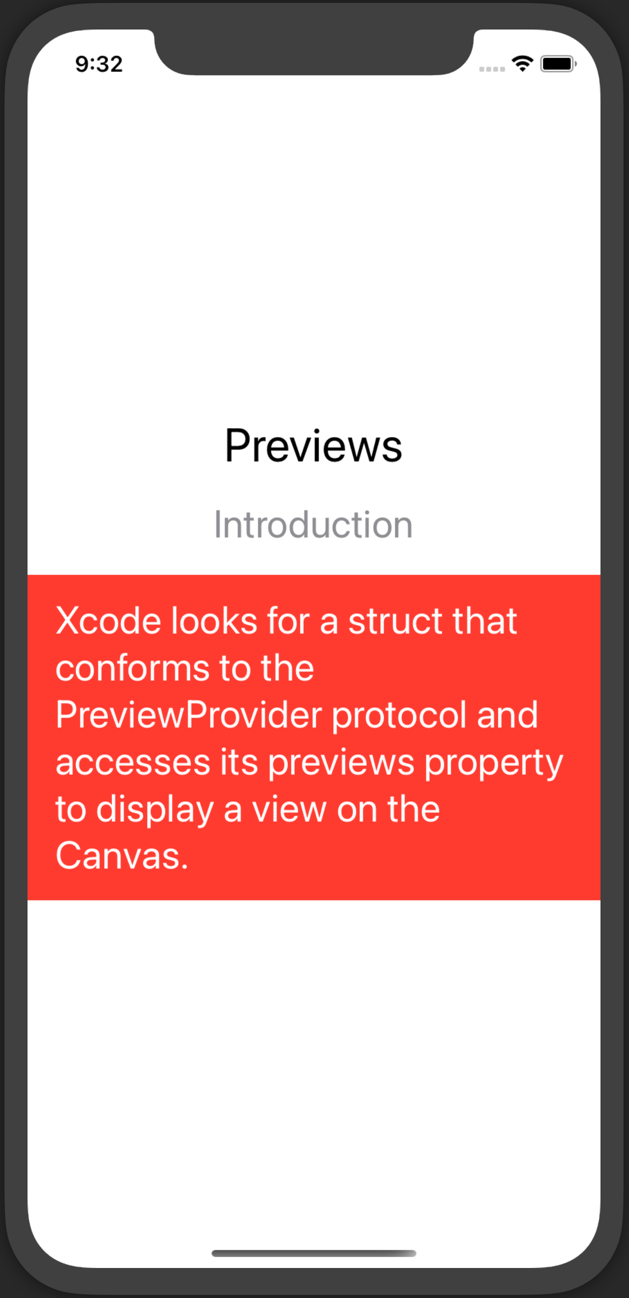


The canvas is the area next to the code that shows you a preview of how your SwiftUI will look. You might also hear this called the "**Preview**" or "**Preview Canvas**".

If you do not see this pane, click on the Editor Options button on the top right of your code window and click Canvas:



Introduction


9:32

Previews

Introduction

Xcode looks for a struct that conforms to the PreviewProvider protocol and accesses its previews property to display a view on the Canvas.

```
struct Previews_Intro: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Previews")
                .font(.largeTitle)

            Text("Introduction")
                .foregroundColor(.gray)

            Text("Xcode looks for a struct that conforms to the PreviewProvider protocol and
                accesses its previews property to display a view on the Canvas.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.red)
                .foregroundColor(.white)

        }.font(.title)
    }
}
```

```
// Xcode looks for PreviewProvider struct
struct Previews_Intro_Previews: PreviewProvider {
    // It will access this property to get a view to show in the Canvas (if the Canvas is shown)
    static var previews: some View {
        // Instantiate and return your view inside this property to see a preview of it
        Previews_Intro()
    }
}
```

#Preview

iOS 17

The preview syntax has evolved into less code.

Before iOS 17

```
struct Previews_Intro_Previews: PreviewProvider {
    static var previews: some View {
        Previews_Intro()
    }
}
```

iOS 17 and after

```
#Preview {
    Previews_Intro()
}
```

After iOS 17, the previews use the #Preview macro that hides a lot of other code that makes this all work.
You will see both examples of these in the companion project.

?

What are Swift macros?
Swift macros allow developers to save time when coding.

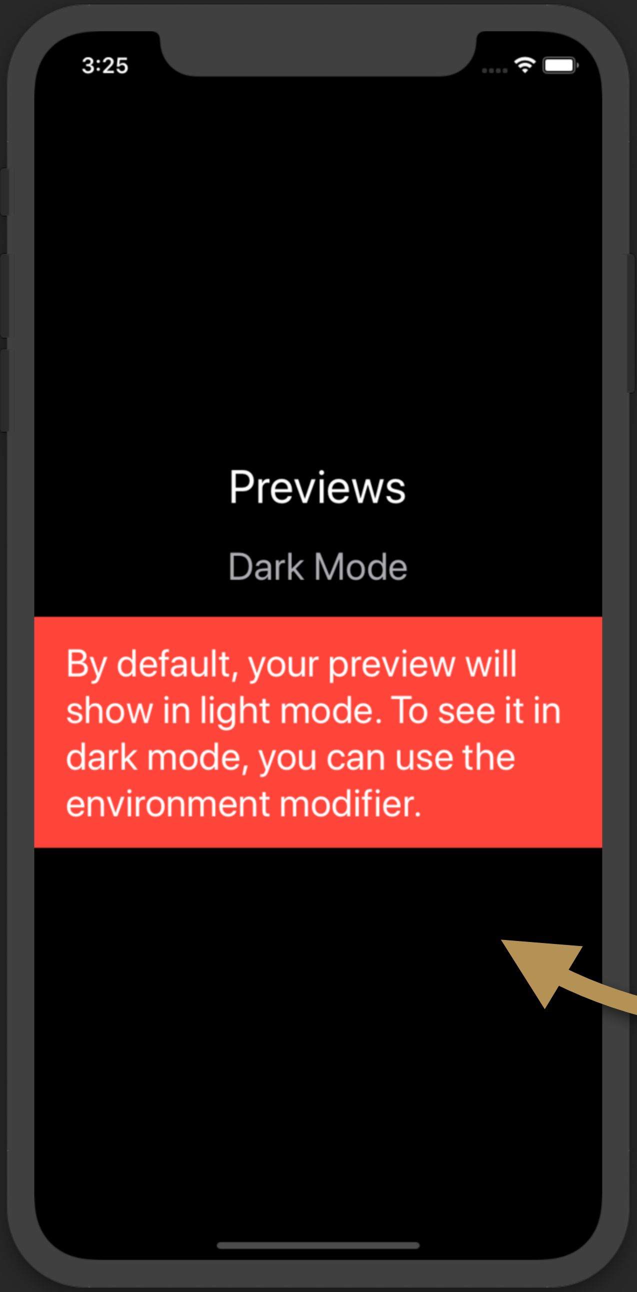
In the field of Computing, the word "macro" means, *"a single instruction that expands automatically into a set of instructions to perform a particular task"*.

In Swift, you can type in simple keywords like "#Preview" and these macros will then transform the code Into more code before it gets compiled.

A macro is kind of like a command that you can write and Swift will know all the things it has to do (or code it has to generate) before your code gets compiled.

💡 To know if a keyword is a macro or not, right-click it in Xcode and if you see the option "Expand Macro", then you know it's a macro.

Dark Mode



Previews

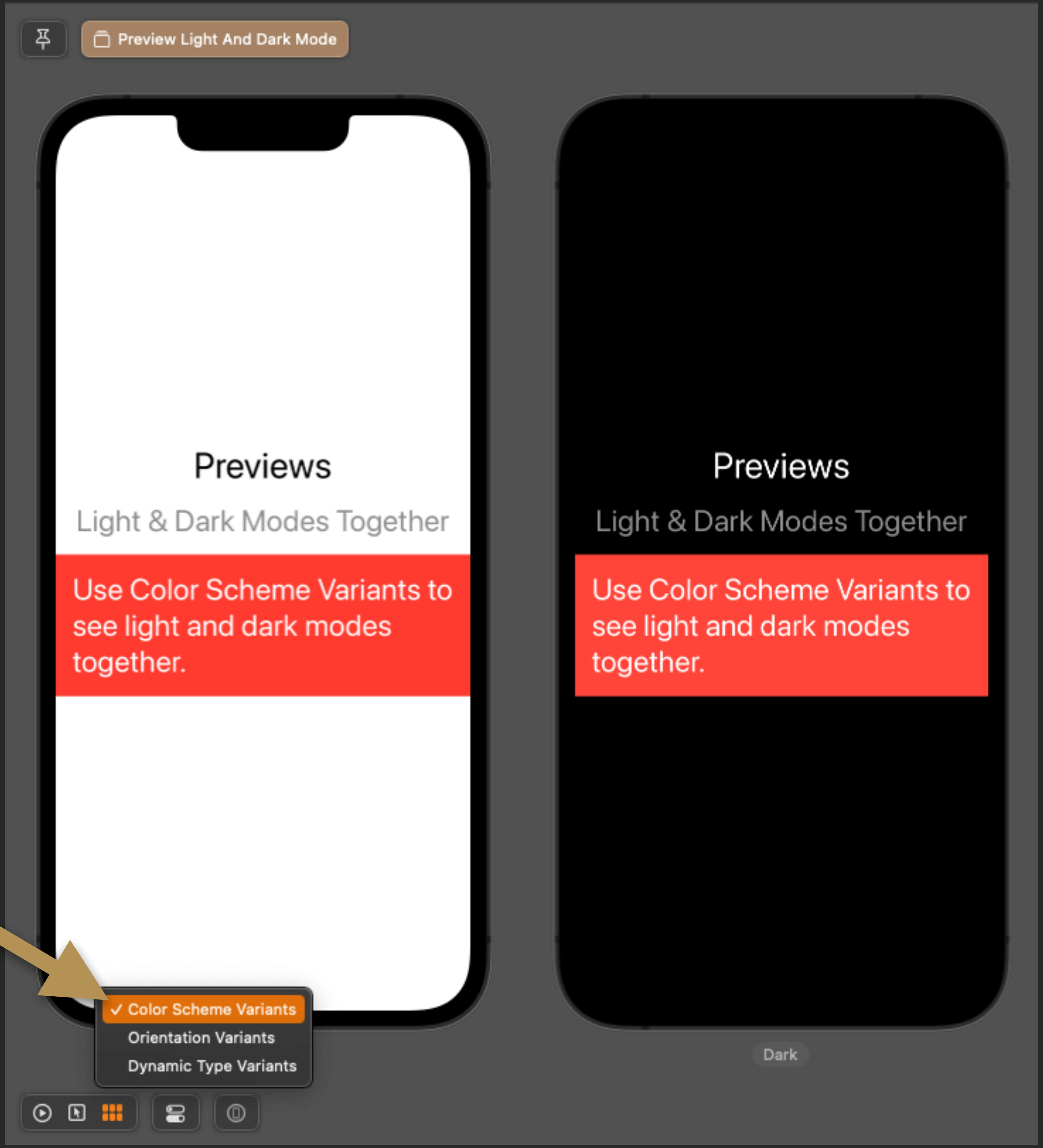
Dark Mode

By default, your preview will show in light mode. To see it in dark mode, you can use the environment modifier.

```
struct Preview_DarkMode: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Previews").font(.largeTitle)
            Text("Dark Mode").foregroundStyle(.gray)
            Text("By default, your preview will show in light mode. To see it in
                 dark mode, you can use the environment modifier.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.red)
                .foregroundStyle(.white)

        }.font(.title)
    }
}
```

```
#Preview {
    Preview_DarkMode()
        .preferredColorScheme(.dark)
}
```

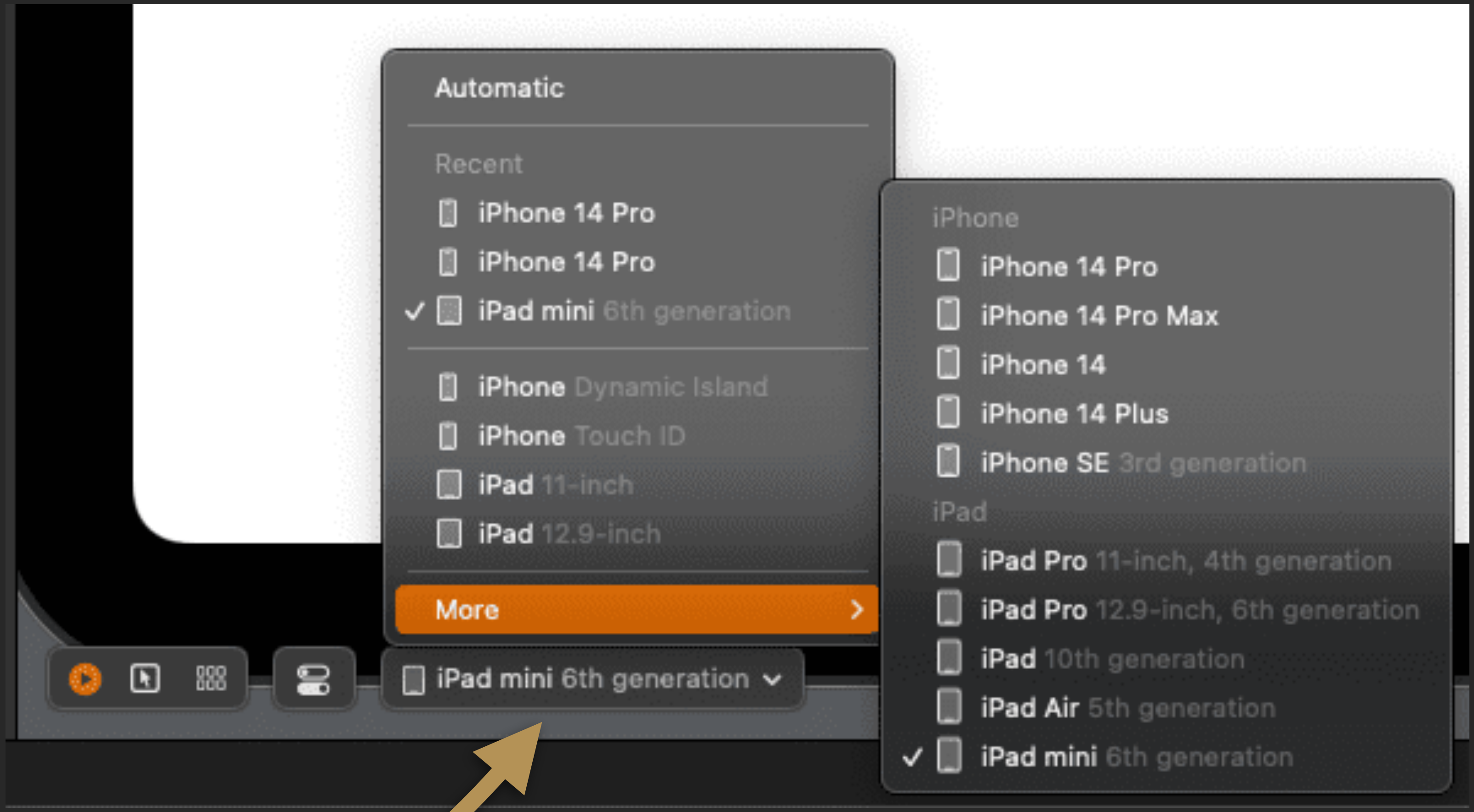
Light & Dark Modes Together


Preview Light And Dark Mode
Previews
Light & Dark Modes Together
Use Color Scheme Variants to see light and dark modes together.
Previews
Light & Dark Modes Together
Use Color Scheme Variants to see light and dark modes together.
✓ Color Scheme Variants
Orientation Variants
Dynamic Type Variants
Dark

Look in the corner for preview options and select "Color Scheme Variants".

Changing Devices

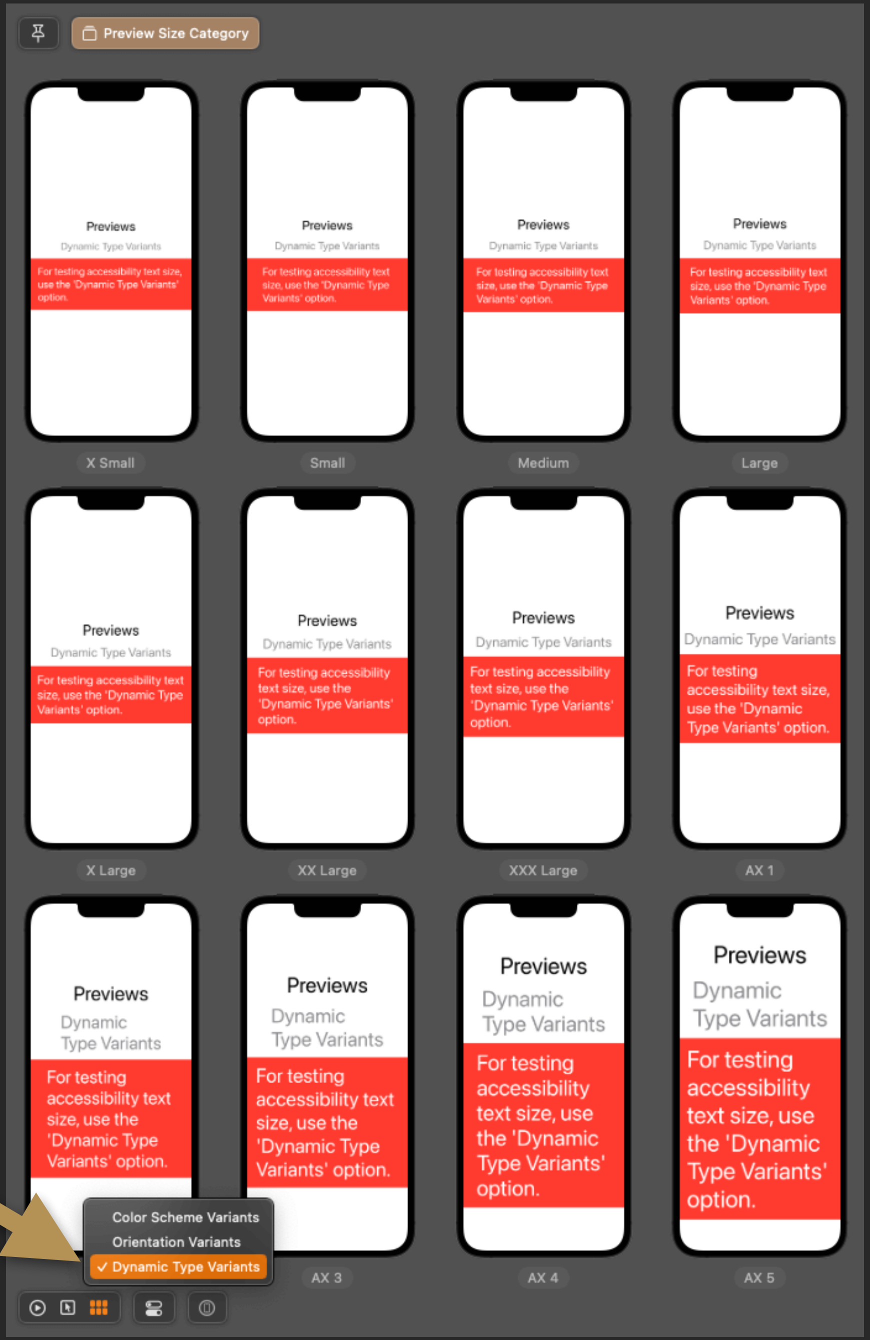
By default, your canvas will use the simulator you currently have selected (upper left in Xcode). You can preview a different device by selecting a different option from the canvas.



Dynamic Type Variants

While designing the UI of your app you will want to make sure it looks good for all sizes of type (which a user can change in the Settings app).

Look in the corner for preview options and select "Dynamic Type Variants".

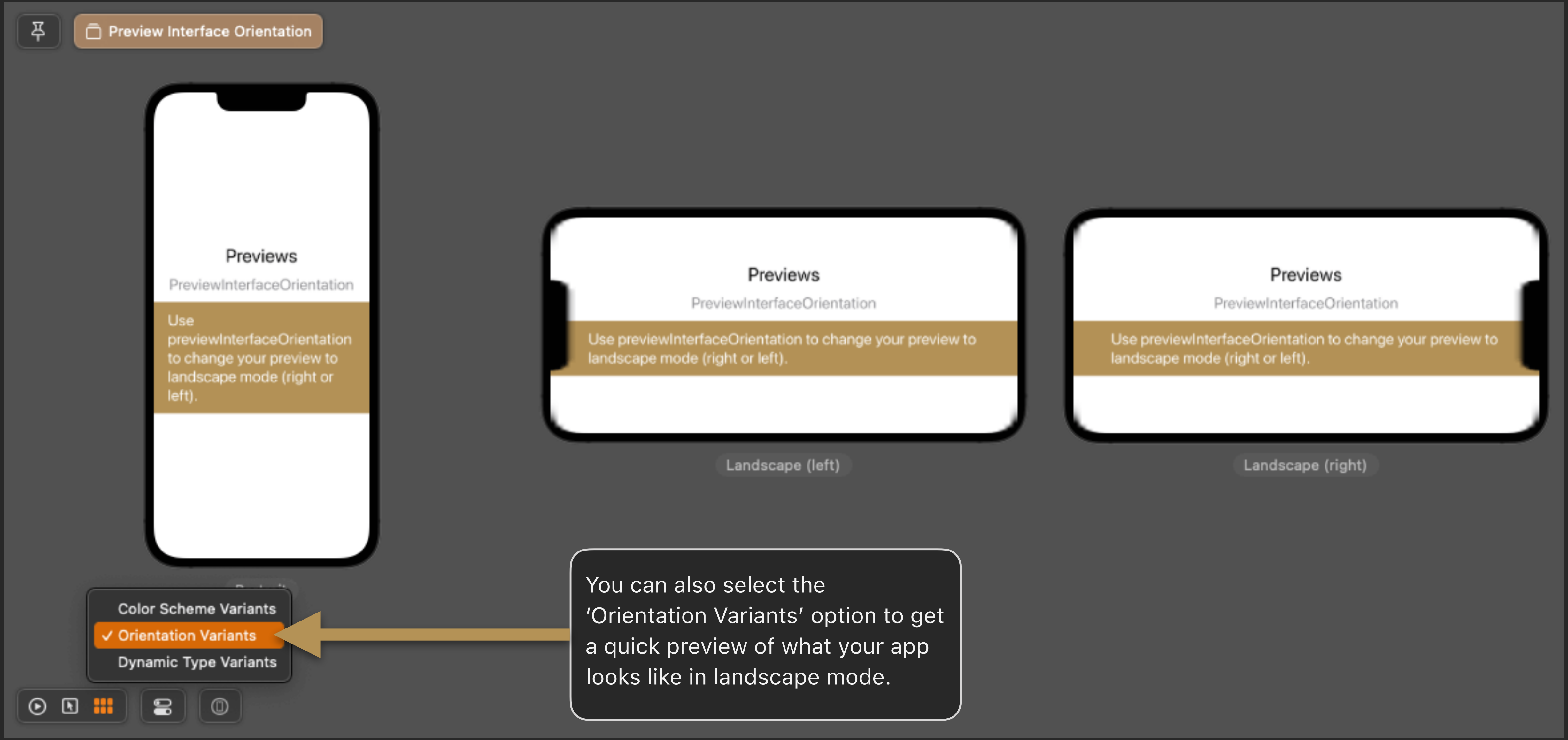


Previews

Dynamic Type Variants

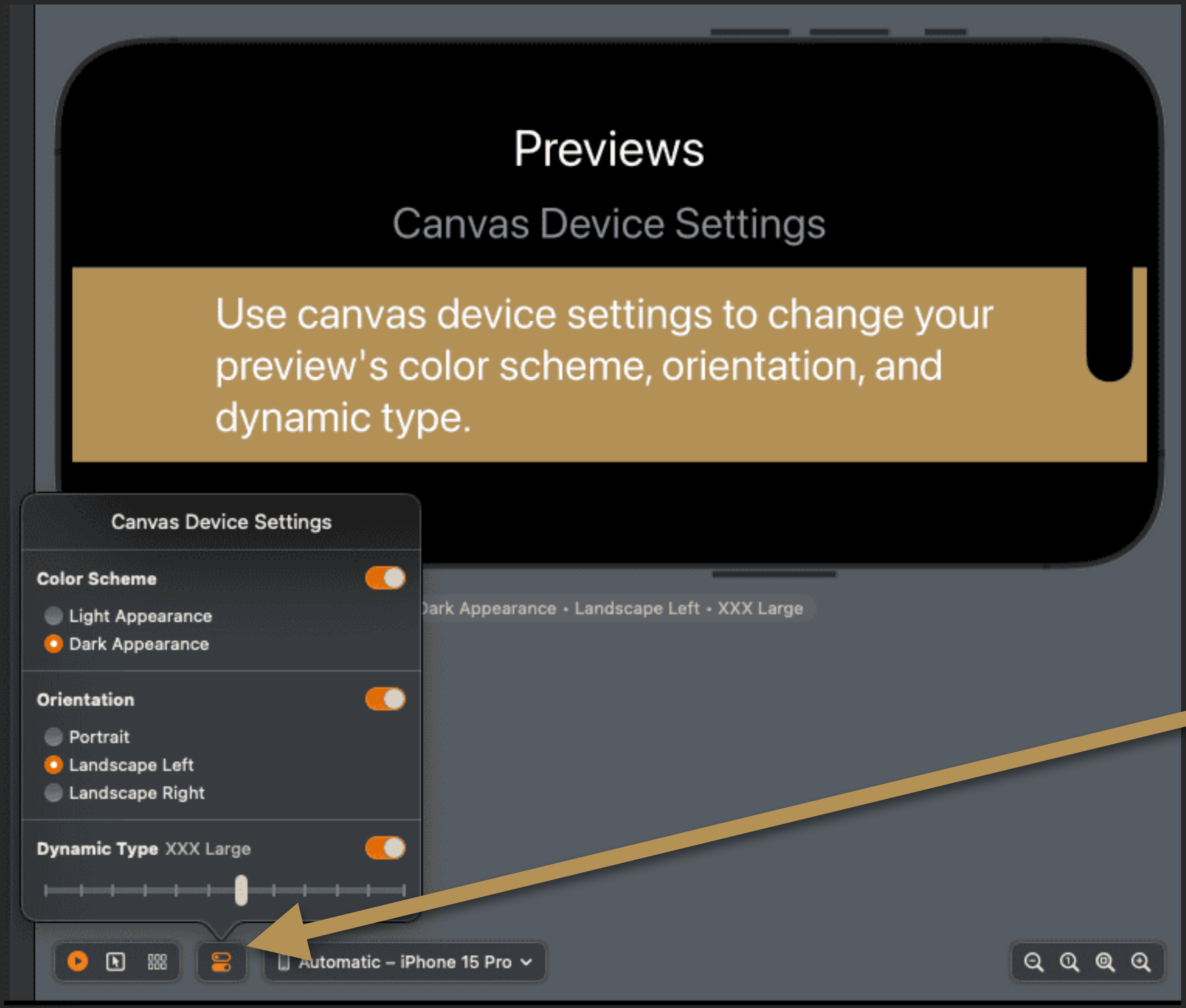
For testing accessibility text size, use the 'Dynamic Type Variants' option.

Orientation Variants


Preview Interface Orientation
Previews
PreviewInterfaceOrientation
Use previewInterfaceOrientation to change your preview to landscape mode (right or left).
Landscape (left)
Landscape (right)
Color Scheme Variants
✓ Orientation Variants
Dynamic Type Variants

You can also select the 'Orientation Variants' option to get a quick preview of what your app looks like in landscape mode.

Canvas Device Settings



Previews

Canvas Device Settings

Use canvas device settings to change your preview's color scheme, orientation, and dynamic type.

Canvas Device Settings

Color Scheme
- Light Appearance
- [✓ Dark Appearance]

Orientation
- Portrait
- [✓ Landscape Left]
- Landscape Right

Dynamic Type XXX Large

Dark Appearance • Landscape Left • XXX Large
Automatic – iPhone 15 Pro

Note: The changes you make in the Canvas Device Settings will apply to ALL canvases.

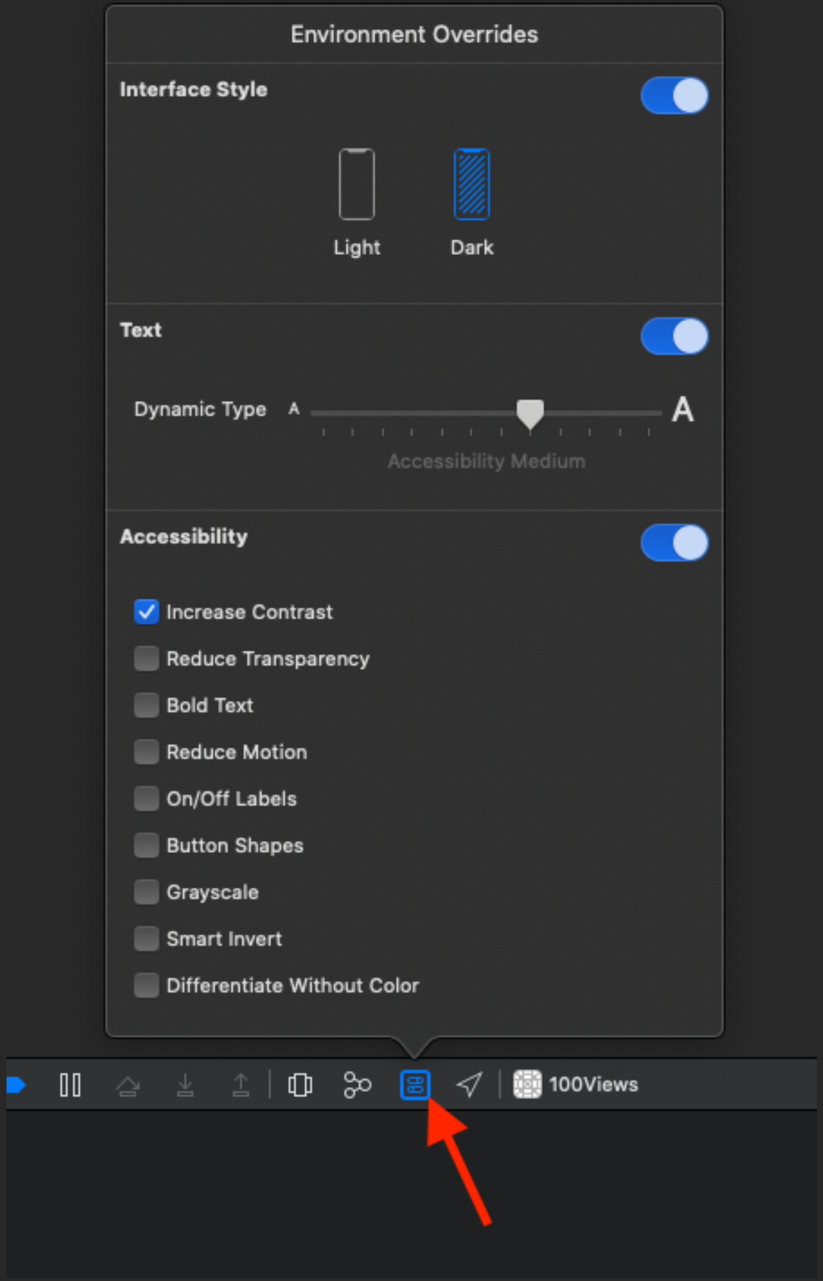
Preview Traits

iOS 18

```swift
struct Preview_CanvasDeviceSettings: View {
    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("Previews",
                       subtitle: "Canvas Device Settings",
                       desc: "Use canvas device settings to change your preview's color scheme,
                             orientation, and dynamic type.",
                       textColor: .white)
        }
        .font(.title)
    }
}

#Preview("Left", traits: .landscapeLeft) {
    Preview_CanvasDeviceSettings()
}

#Preview("Right", traits: .landscapeRight) {
    Preview_CanvasDeviceSettings()
}
```

Note: You can use the traits parameter to set the landscape orientation.

Environment Overrides



Environment Overrides

Interface Style [✓ On]

Light [✓ Dark]

Text [✓ On]

Dynamic Type: Accessibility Medium

Accessibility [✓ On]

- [x] Increase Contrast
- [ ] Reduce Transparency
- [ ] Bold Text
- [ ] Reduce Motion
- [ ] On/Off Labels
- [ ] Button Shapes
- [ ] Grayscale
- [ ] Smart Invert
- [ ] Differentiate Without Color

If you prefer to see your work in the **Simulator** then you can access many of the options mentioned through the Environment Overrides options.

This button will show up when you **run your app** in the debugging toolbar at the bottom of Xcode.

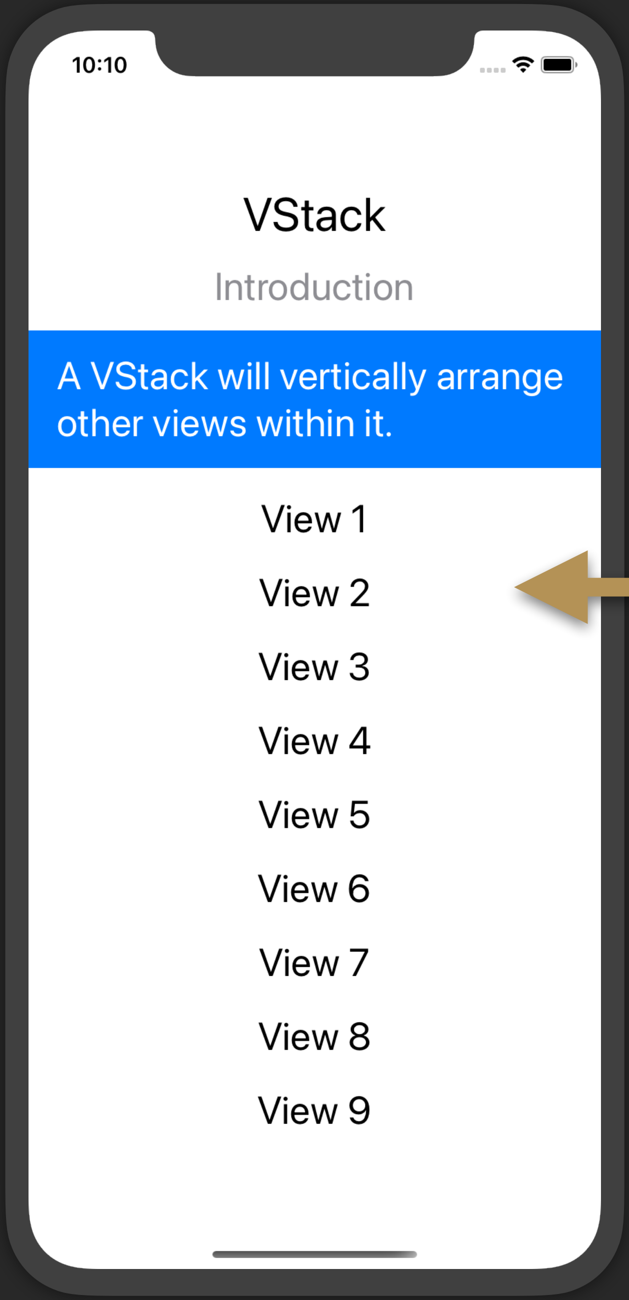
LAYOUT VIEWS

Chapter Quick Links

VStack

VStack stands for “Vertical Stack”. It is a pull-in container view in which you pass in up to ten views and it will compose them one below the next, going down the screen.

Introduction


10:10

VStack

Introduction

A VStack will vertically arrange other views within it.

View 1

View 2

View 3

View 4

View 5

View 6

View 7

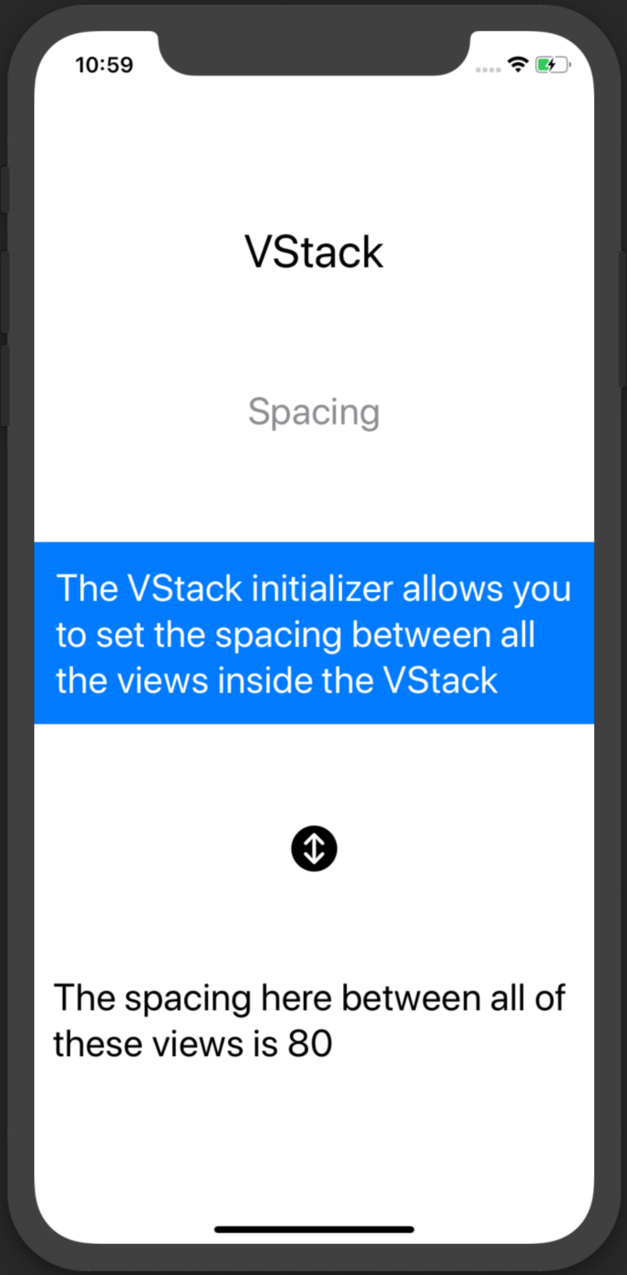
View 8

View 9

```
struct VStack_Intro : View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("VStack",
                       subtitle: "Introduction",
                       desc: "A VStack will vertically arrange other views within it.",
                       back: .blue, textColor: .white)

            Text("View 1")
            Text("View 2")
            Text("View 3")
            Text("View 4")
            Text("View 5")
            Text("View 6")
            Text("View 7")
            Text("View 8")
            Text("View 9")
        }
        .font(.title)
    }
}
```

In SwiftUI, container views, like the VStack, can contain as many views as you like.

Spacing



VStack

Spacing

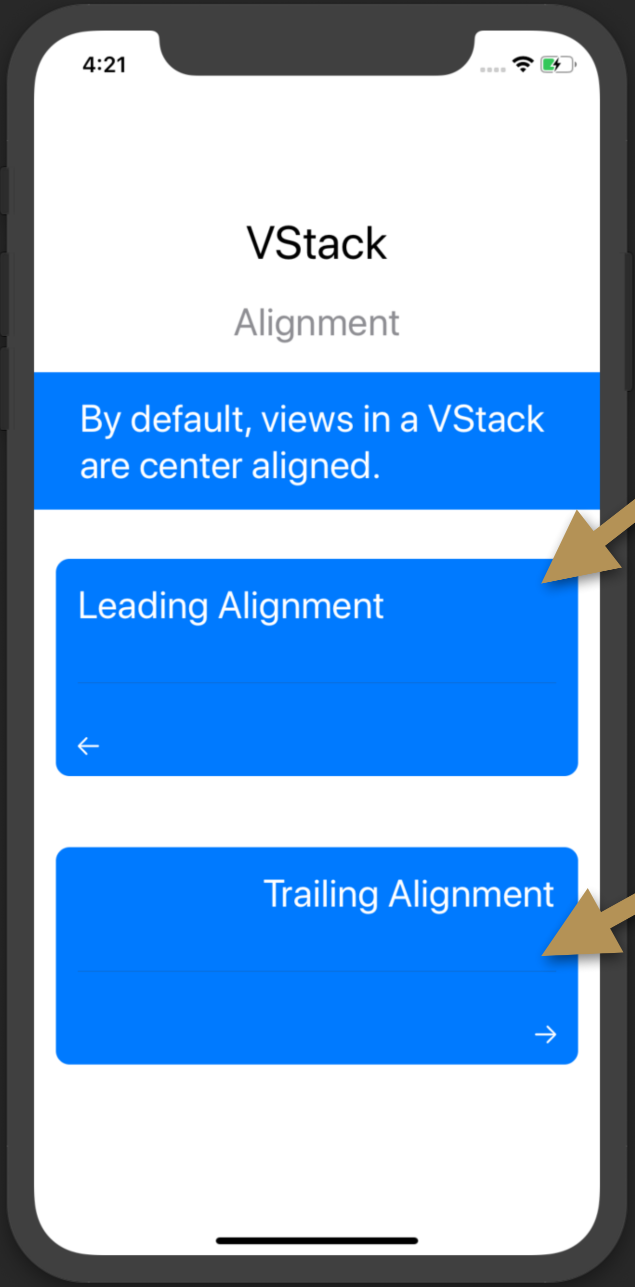
The VStack initializer allows you to set the spacing between all the views inside the VStack

The spacing here between all of these views is 80

```
VStack(spacing: 80) {
    Text("VStack")
        .font(.largeTitle)

    Text("Spacing")
        .font(.title)
        .foregroundColor(.gray)

    Text("The VStack initializer allows you to set the spacing between all the views inside the
         VStack")
        .frame(maxWidth: .infinity)
        .padding()
        .background(Color.blue).font(.title)
        .foregroundColor(.white)

    Image(systemName: "arrow.up.and.down.circle.fill")
        .font(.largeTitle)

    Text("The spacing here between all of these views is 80")
        .font(.title)
}
```

Set spacing in the initializer.

Alignment



```
VStack(spacing: 20) {
    Text("VStack")
        .font(.largeTitle)
    Text("Alignment")
        .font(.title)
        .foregroundColor(.gray)
    Text("By default, views in a VStack are center aligned.")
        ...

    VStack(alignment: .leading, spacing: 40) {
        Text("Leading Alignment")
            .font(.title)
        Divider() // Creates a thin line (Push-out view)
        Image(systemName: "arrow.left")
    }
    .padding()
    .foregroundColor(Color.white)
    .background(RoundedRectangle(cornerRadius: 10)
    .foregroundColor(.blue))
    .padding()

    VStack(alignment: .trailing, spacing: 40) {
        Text("Trailing Alignment")
            .font(.title)
        Divider()
        Image(systemName: "arrow.right")
    }
    .padding()
    .foregroundColor(Color.white)
    .background(RoundedRectangle(cornerRadius: 10)
    .foregroundColor(.blue))
    .padding()
}
```

Set alignment in the initializer.

iOS 14

LazyVStack

This SwiftUI chapter is locked in this preview.

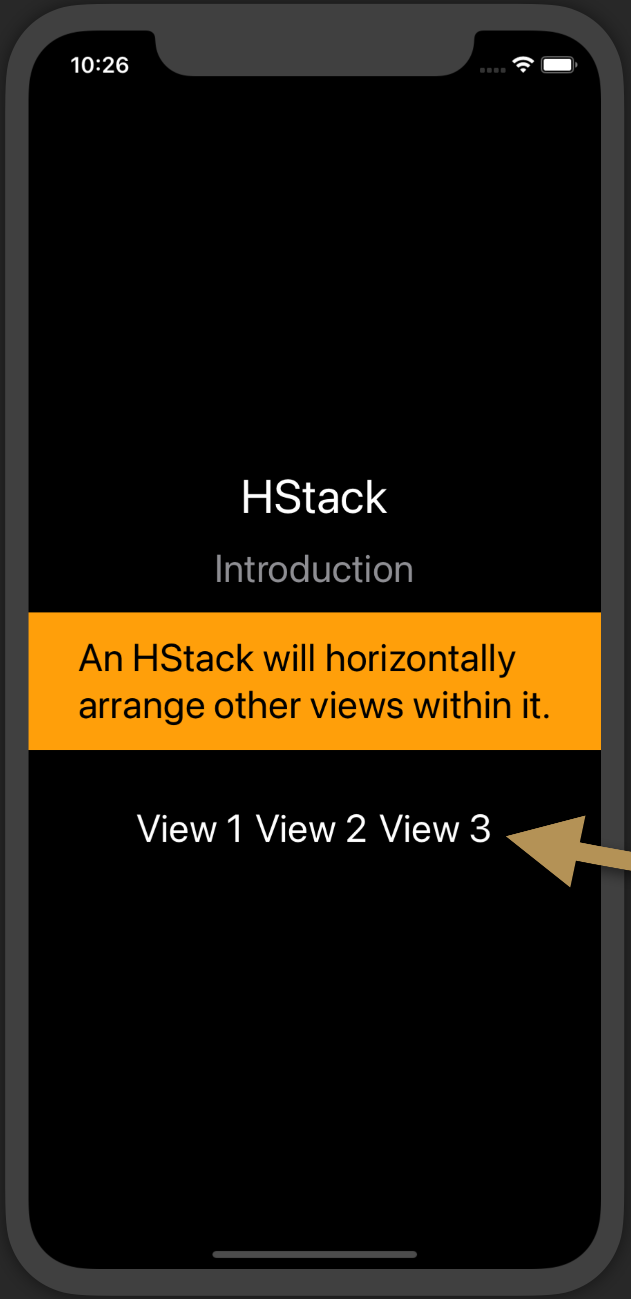
The Lazy Vertical Stack is similar to the VStack. It's "lazy" because if you have views scrolling off the screen, SwiftUI will not load them unless it needs to show them on the screen. The VStack does not do this. The VStack loads all child views when displayed.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

HStack

HStack stands for “Horizontal Stack”. It is a pull-in container view in which you pass in up to ten views and it will compose them side-by-side.

Introduction



HStack

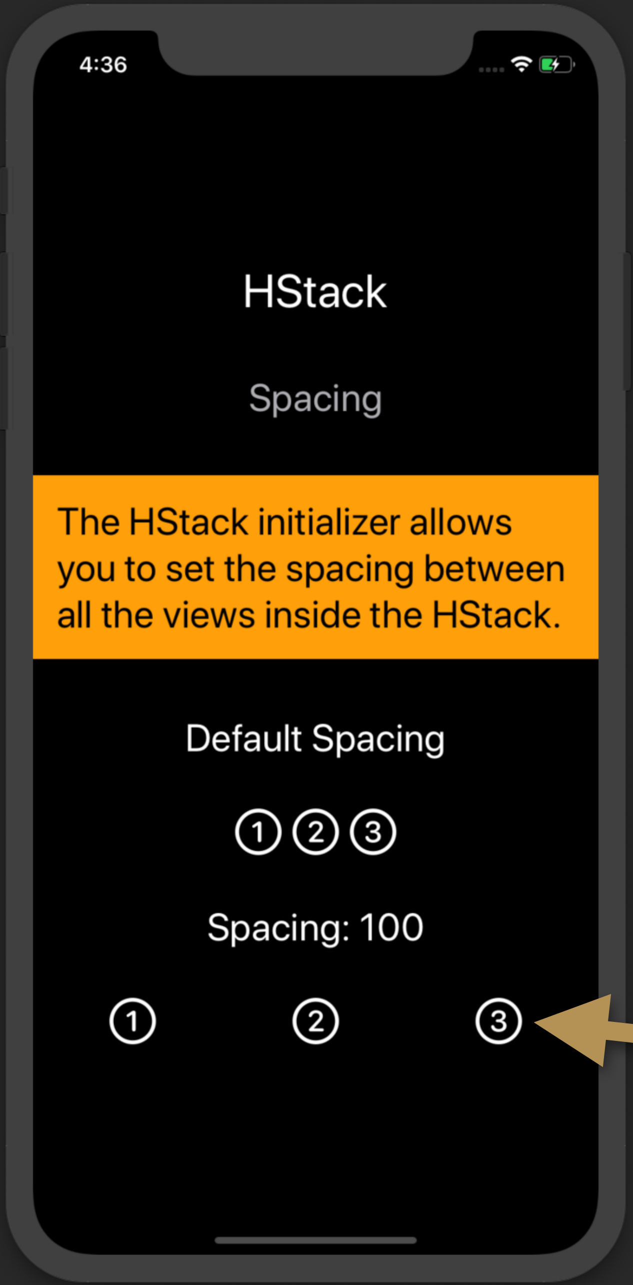
Introduction

An HStack will horizontally arrange other views within it.

View 1 View 2 View 3

```
struct HStack_Intro: View {
    var body: some View {
        VStack(spacing: 40) {
            HeaderView("HStack",
                       subtitle: "Introduction",
                       desc: "An HStack will horizontally arrange other views within it.",
                       back: .orange)

            HStack {
                Text("View 1")
                Text("View 2")
                Text("View 3")
            }
        }
        .font(.title)
    }
}
```

Spacing



HStack

Spacing

The HStack initializer allows you to set the spacing between all the views inside the HStack.

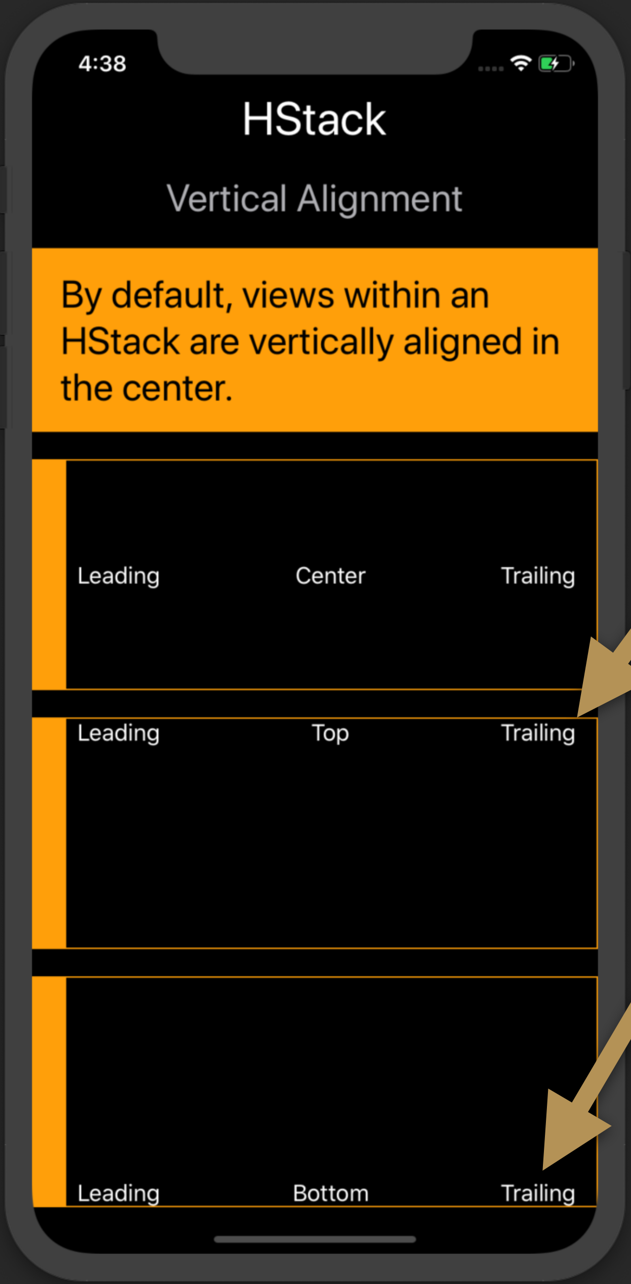
Default Spacing

Spacing: 100

```
VStack(spacing: 40) {
    Text("HStack")
        .font(.largeTitle)

    Text("Spacing")
        .font(.title)
        .foregroundColor(.gray)

    Text("The HStack initializer allows you to set the spacing between all the views inside the
          HStack")
        .frame(maxWidth: .infinity)
        .padding()
        .background(Color.orange).font(.title)
        .foregroundColor(.black)

    Text("Default Spacing")
        .font(.title)
    HStack {
        Image(systemName: "1.circle")
        Image(systemName: "2.circle")
        Image(systemName: "3.circle")
    }.font(.largeTitle)

    Divider()

    Text("Spacing: 100")
        .font(.title)
    HStack(spacing: 100) {
        Image(systemName: "1.circle")
        Image(systemName: "2.circle")
        Image(systemName: "3.circle")
    }.font(.largeTitle)
}
```

Set spacing in the initializer.

Alignment

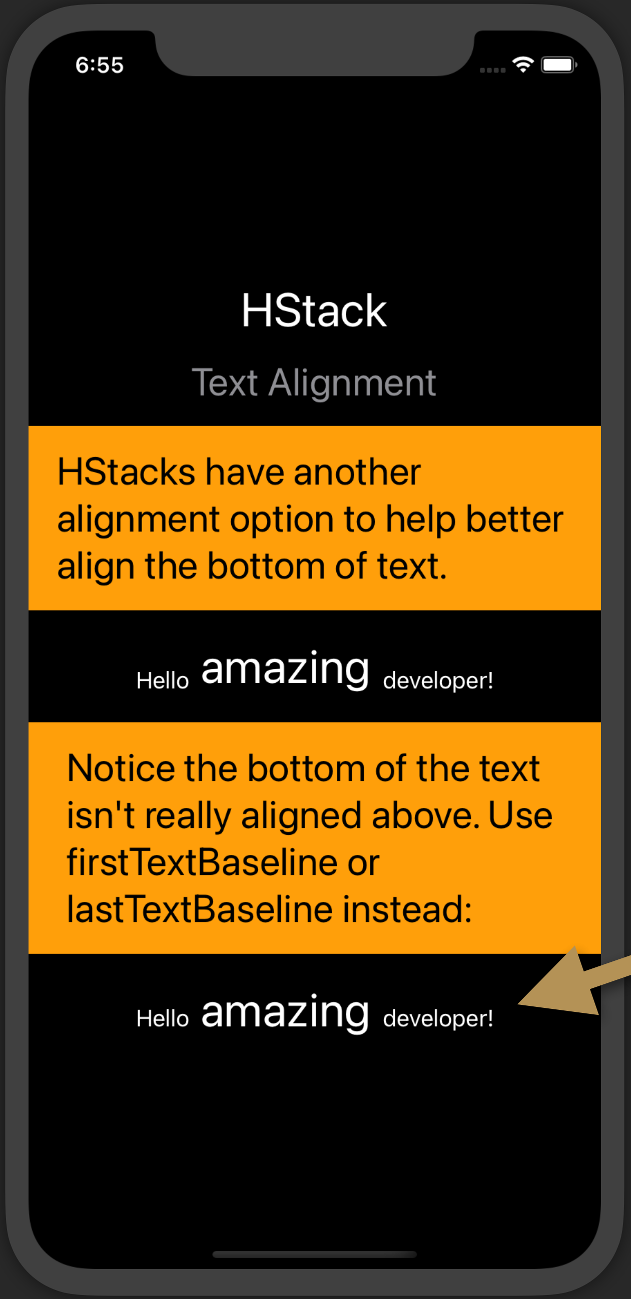


By default, views within an HStack are vertically aligned in the center.

```
Text("By default, views within an HStack are vertically aligned in the center.")
...
HStack {
    Rectangle().foregroundColor(.orange).frame(width: 25)
    Text("Leading")
    Spacer()
    Text("Center")
    Spacer()
    Text("Trailing")
        .padding(.trailing)
}
.border(Color.orange)
HStack(alignment: .top) {
    Rectangle().foregroundColor(.orange).frame(width: 25)
    Text("Leading")
    Spacer()
    Text("Top")
    Spacer()
    Text("Trailing")
        .padding(.trailing)
}
.border(Color.orange)
HStack(alignment: .bottom) {
    Rectangle().foregroundColor(.orange).frame(width: 25)
    Text("Leading")
    Spacer()
    Text("Bottom")
    Spacer()
    Text("Trailing")
        .padding(.trailing)
}
.border(Color.orange)
```

Set alignment in the initializer.

Text Alignment


6:55

HStack

Text Alignment

HStacks have another alignment option to help better align the bottom of text.

Hello amazing developer!

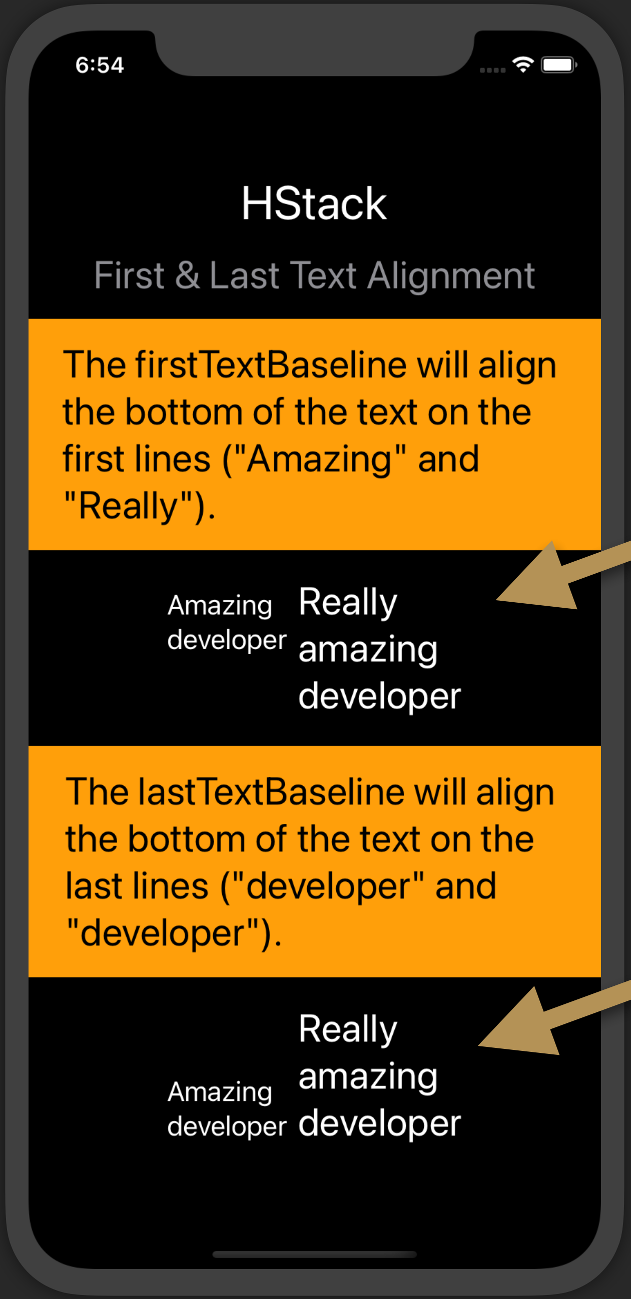
Notice the bottom of the text isn't really aligned above. Use firstTextBaseline or lastTextBaseline instead:

Hello amazing developer!

```
struct HStack_TextAlignment: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("HStack",
                       subtitle: "Text Alignment",
                       desc: "HStacks have another alignment option to help better align the
                             bottom of text.",
                       back: .orange)

            HStack(alignment: .bottom) {
                Text("Hello")
                Text("amazing")
                    .font(.largeTitle)
                Text(“developer!")
            }
            .font(.body)

            DescView(desc: "Notice the bottom of the text isn't really aligned above. Use
                           firstTextBaseline or lastTextBaseline instead:", back: .orange)

            HStack(alignment: .firstTextBaseline) {
                Text("Hello")
                Text("amazing")
                    .font(.largeTitle)
                Text("developer!")
            }
            .font(.body)
        }
        .font(.title)
    }
}
```

This will align the text normally.
But what’s the difference between first and last text baseline? See on the next page.

First & Last Text Alignment


6:54

HStack

First & Last Text Alignment

The firstTextBaseline will align the bottom of the text on the first lines ("Amazing" and "Really").

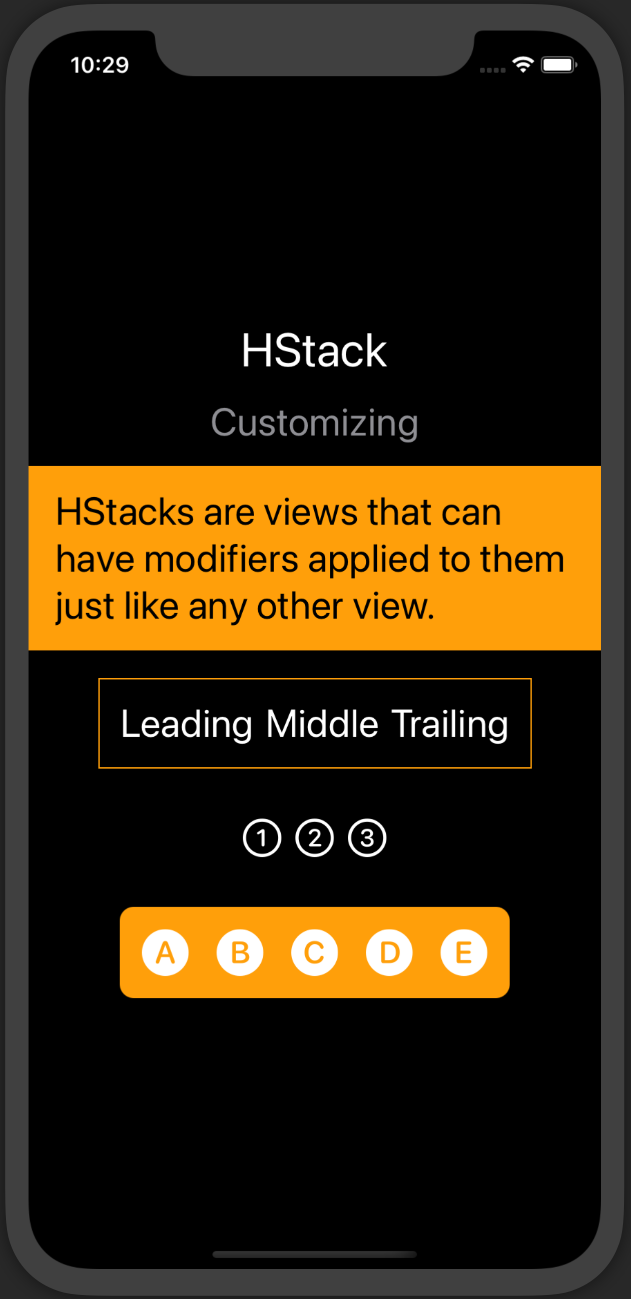
Amazing developer Really amazing developer

The lastTextBaseline will align the bottom of the text on the last lines ("developer" and "developer").

Amazing developer Really amazing developer

```swift
struct HStack_TextAlignment_FirstLast: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("HStack",
                       subtitle: "First & Last Text Alignment",
                       desc: "The firstTextBaseline will align the bottom of the text on the
                            first lines (\"Amazing\" and \"Really\").",
                       back: .orange)

            HStack(alignment: .firstTextBaseline) {
                Text("Amazing developer")
                    .font(.title3)
                Text("Really amazing developer")
            }
            .frame(width: 250)

            DescView(desc: "The lastTextBaseline will align the bottom of the text on the last
                            lines (\"developer\" and \"developer\").", back: .orange)

            HStack(alignment: .lastTextBaseline) {
                Text("Amazing developer")
                    .font(.title3)
                Text("Really amazing developer")
            }
            .frame(width: 250)
        }
        .font(.title)
    }
}
```

Customization


10:29

HStack

Customizing

HStacks are views that can have modifiers applied to them just like any other view.

Leading Middle Trailing

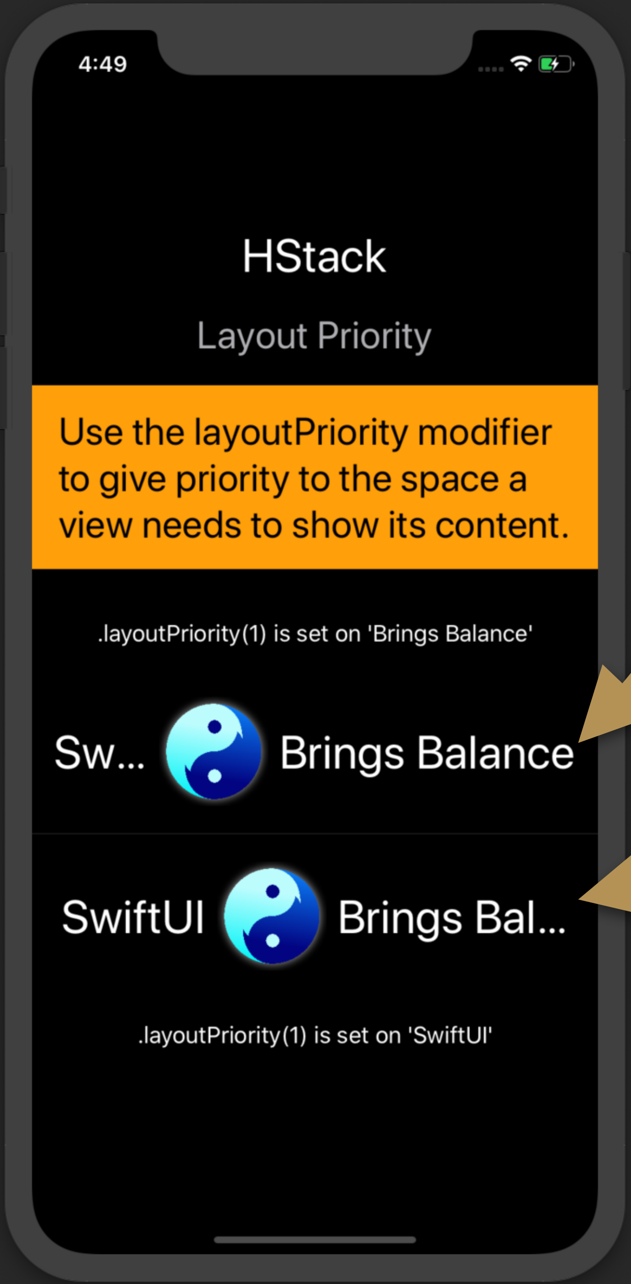
① ② ③

A B C D E

```
struct HStack_Customizing : View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("HStack",
                       subtitle: "Customizing",
                       desc: "HStacks are views that can have modifiers applied to them just
                             like any other view.",
                       back: .orange)

            HStack {
                Text("Leading")
                Text("Middle")
                Text("Trailing")
            }
            .padding()
                .border(Color.orange) // Create a 2 point border using the color specified

            HStack(spacing: 10) {
                Image(systemName: "1.circle")
                Image(systemName: "2.circle")
                Image(systemName: "3.circle")
            }.padding()

            HStack(spacing: 20) {
                Image(systemName: "a.circle.fill")
                Image(systemName: "b.circle.fill")
                Image(systemName: "c.circle.fill")
                Image(systemName: "d.circle.fill")
                Image(systemName: "e.circle.fill")
            }
            .font(.largeTitle).padding()
            .background(RoundedRectangle(cornerRadius: 10)
            .foregroundColor(.orange))
        }
        .font(.title)
    }
}
```

Layout Priority

When using a horizontal stack with text views within it, there's a chance that text might truncate if you are not allowing them to wrap. In this case, you can prioritize which one will truncate last with layout priority. The default value is 0. The higher the number, the higher the priority to have enough space to not be truncated.



4:49

HStack

Layout Priority

Use the layoutPriority modifier to give priority to the space a view needs to show its content.

.layoutPriority(1) is set on 'Brings Balance'

Sw... Brings Balance

SwiftUI Brings Bal...

.layoutPriority(1) is set on 'SwiftUI'

```
HStack {
    Text("SwiftUI")
        .font(.largeTitle).lineLimit(1) // Don't let text wrap
    Image("SwiftUI")
        .resizable()
        .frame(width: 80, height: 80)
    Text("Brings Balance")
        .font(.largeTitle)
        .layoutPriority(1) // Truncate last
}
.padding([.horizontal])
Divider()

HStack {
    Text("SwiftUI")
        .font(.largeTitle)
        .layoutPriority(1) // Truncate last
    Image("SwiftUI")
        .resizable()
        .frame(width: 80, height: 80)
    Text("Brings Balance")
        .font(.largeTitle).lineLimit(1) // Don't let text wrap
}
.padding(.horizontal)
```

Note: You can learn more about layout priority in the chapter "Layout Modifiers", section "LayoutPriority".

iOS 14

LazyHStack

The Lazy Horizontal Stack is similar to the HStack. It's "lazy" because if you have views scrolling off the screen, SwiftUI will not load them unless it needs to show them on the screen. The HStack does not do this. The HStack loads all child views when displayed.

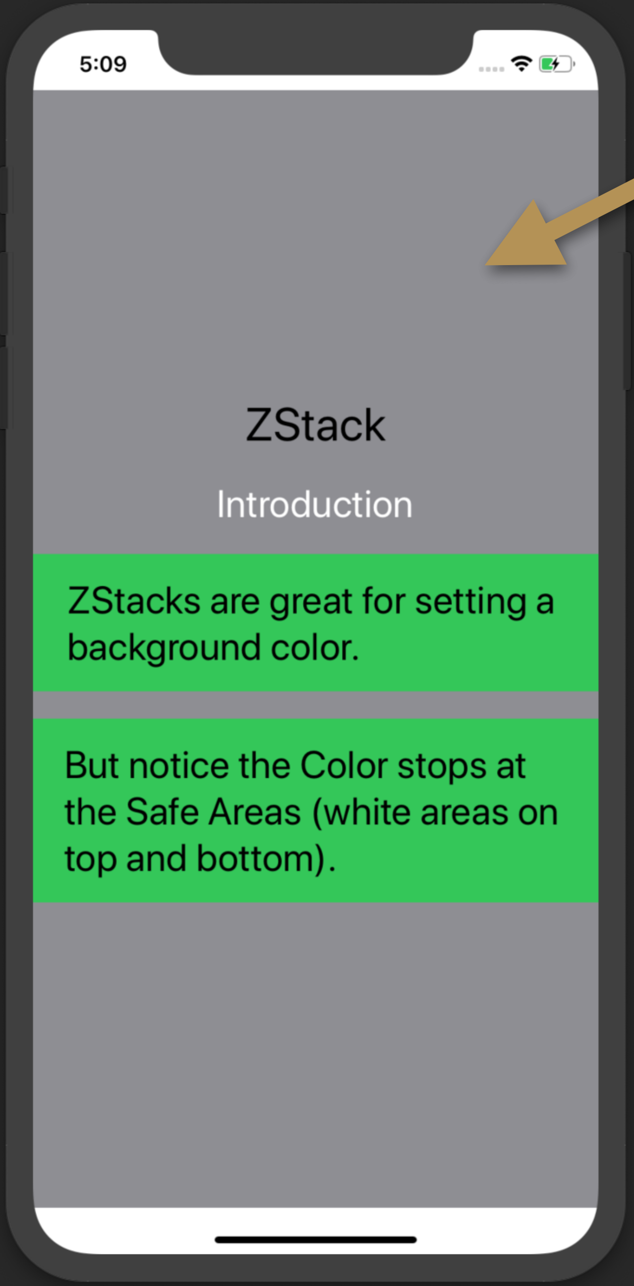
Depth (Z) Stack

A Depth Stack (ZStack) is a pull-in container view. It is a view that overlays its child views on top of each other. (“Z” represents the Z-axis which is depth-based in a 3D space.)

You learned earlier about creating layers with the background and overlay modifiers. ZStack is another way to create layers with views that control their own sizing and spacing.

So, the ZStack is a pull-in container view but you may think it is a push-out view because of the first example but it’s actually the color that is pushing out.

Introduction



ZStack

Introduction

ZStacks are great for setting a background color.

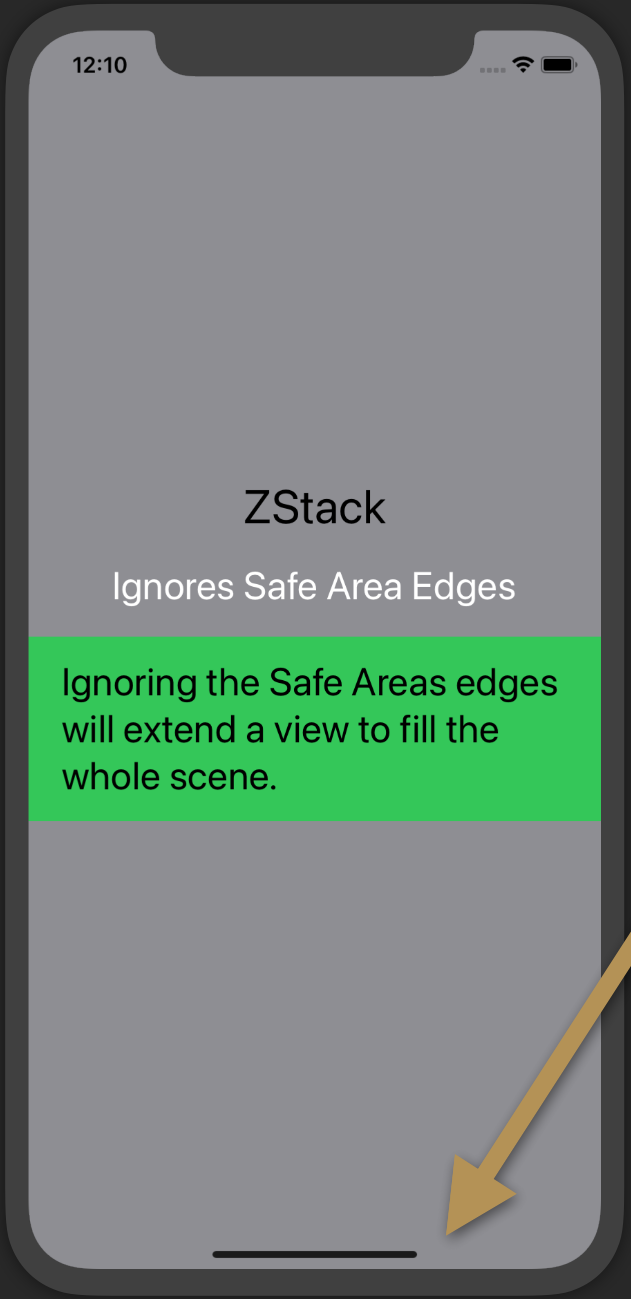
But notice the Color stops at the Safe Areas (white areas on top and bottom).

```
ZStack {
    // LAYER 1: Furthest back
    Color.gray // Yes, Color is a view!

    // LAYER 2: This VStack is on top.
    VStack(spacing: 20) {
        Text("ZStack")
            .font(.largeTitle)

        Text("Introduction")
            .foregroundColor(.white)

        Text("ZStacks are great for setting a background color.")
            .frame(maxWidth: .infinity)
            .padding()
            .background(Color.green)

        Text("But notice the Color stops at the Safe Areas (white areas on top and bottom).")
            .frame(maxWidth: .infinity)
            .padding()
            .background(Color.green)
    }
    .font(.title)
}
```

You set depth by the order of the views inside the ZStack.

Note: The Color view is a push-out view. It is pushing out the ZStack container view.

Ignores Safe Area Edges



ZStack

Ignores Safe Area Edges

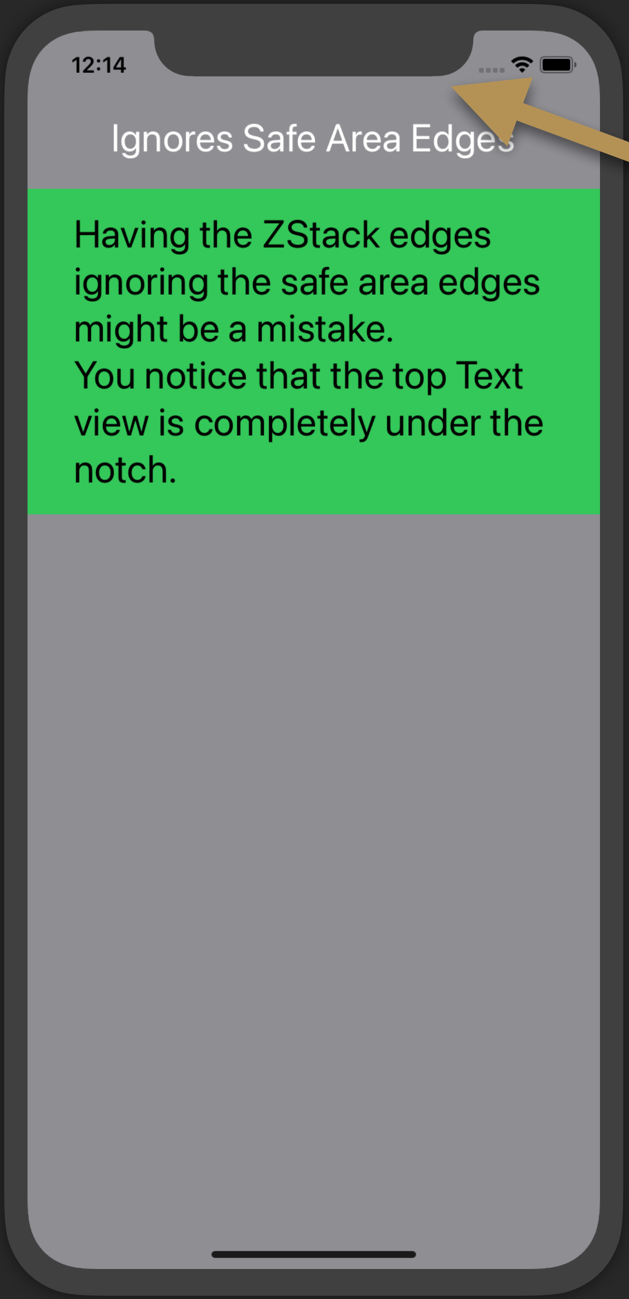
Ignoring the Safe Areas edges will extend a view to fill the whole scene.

```
ZStack {
    Color.gray

    VStack(spacing: 20) {
        Text("ZStack")
            .font(.largeTitle)

        Text("Edges Ignoring Safe Area")
            .foregroundColor(.white)

        Text("Ignoring the Safe Areas will extend a view to fill the whole scene.")
            .frame(maxWidth: .infinity)
            .padding()
            .foregroundColor(.white)
            .background(Color.green)

    }
    .font(.title)
}
.ignoresSafeArea(.all) // Ignore the safe areas
```

Allows views to extend past the safe areas.

Learn more about what Safe Areas are and ways to ignore edges in the chapter "Layout Modifiers" in the section "Ignores Safe Area".

Background Problem


12:14

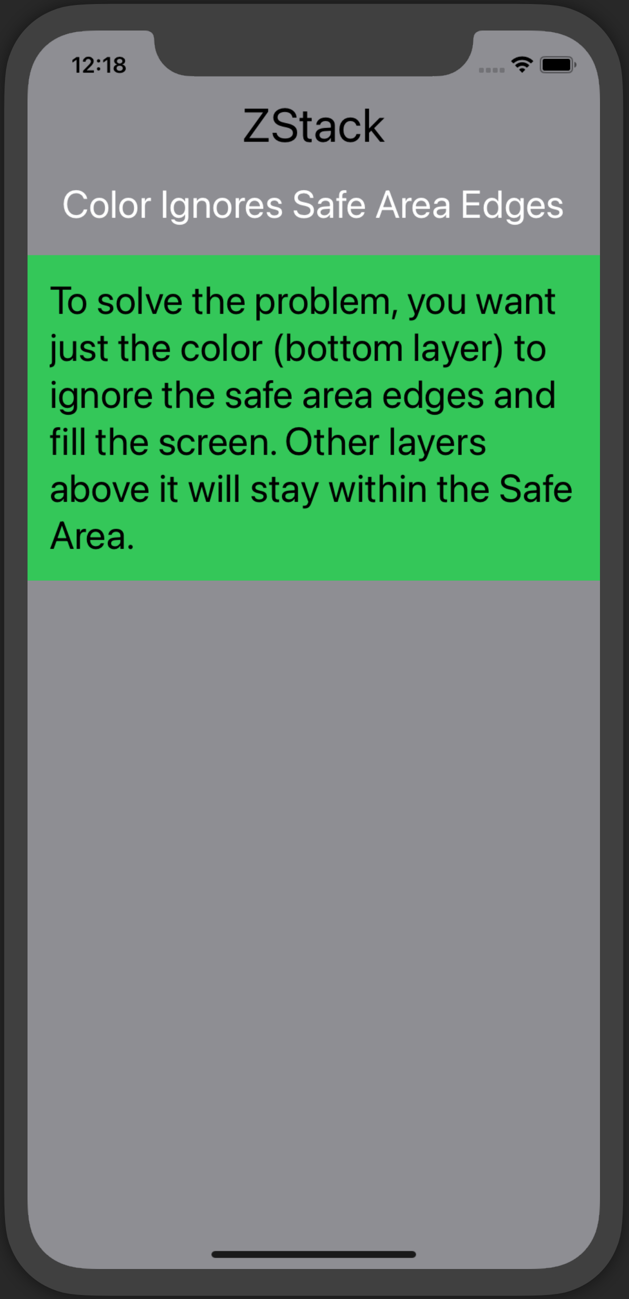
Ignores Safe Area Edges

Having the ZStack edges ignoring the safe area edges might be a mistake.
You notice that the top Text view is completely under the notch.

```
struct ZStack_BackgroundColor_Problem: View {
    var body: some View {
        ZStack {
            Color.gray

            VStack(spacing: 20) {
                Text("ZStack") // This view is under the notch
                    .font(.largeTitle)

                Text("Ignores Safe Area Edges")
                    .foregroundColor(.white)

                Text("Having the ZStack edges ignoring the safe area edges might be a mistake.
                     \nYou notice that the top Text view is completely under the notch.")
                    .frame(maxWidth: .infinity)
                    .padding()
                    .background(Color.green)

                Spacer() // Added a spacer to push the views up.
            }
            .font(.title)
        }
        .ignoresSafeArea()
    }
}
```

Ignores all Safe Area edges.

Background Solution


12:18

ZStack

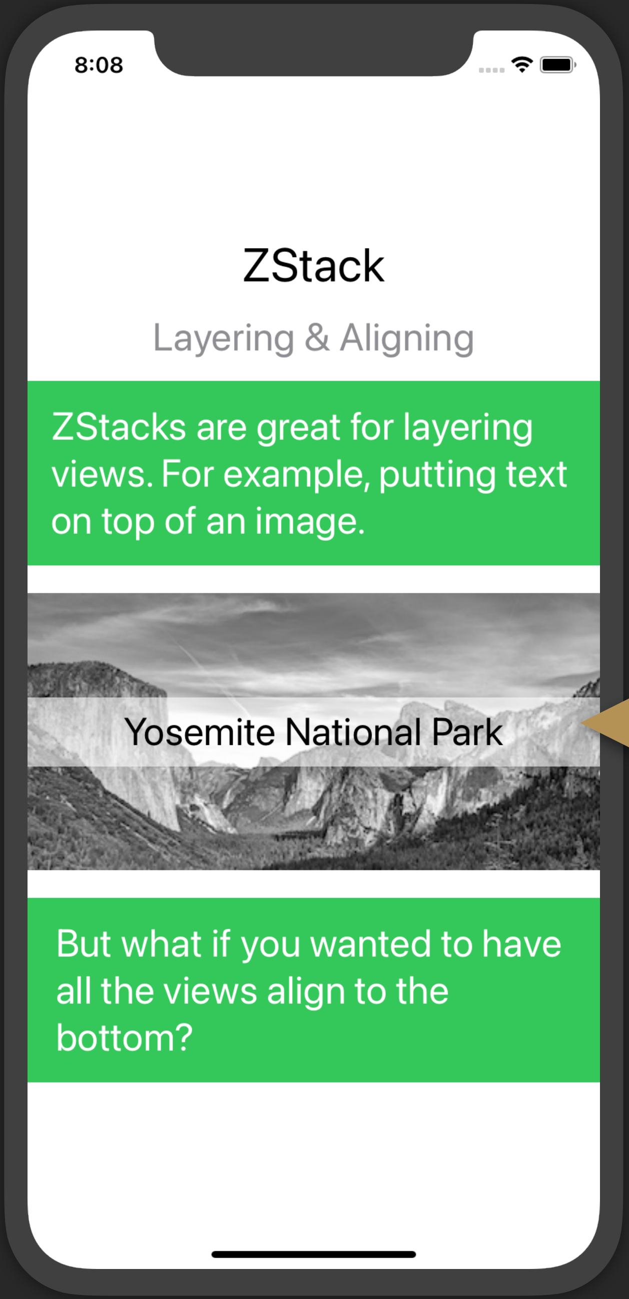
Color Ignores Safe Area Edges

To solve the problem, you want just the color (bottom layer) to ignore the safe area edges and fill the screen. Other layers above it will stay within the Safe Area.

```
struct ZStack_BackgroundColor_Solution: View {
    var body: some View {
        ZStack {
            Color.gray
                .ignoresSafeArea() // Have JUST the color ignore the safe areas edges, not 
the VStack.

            VStack(spacing: 20) {
                Text("ZStack")
                    .font(.largeTitle)

                Text("Color Ignores Safe Area Edges")
                    .foregroundColor(.white)

                Text("To solve the problem, you want just the color (bottom layer) to ignore
                    the safe area edges and fill the screen. Other layers above it will stay
                    within the Safe Area.")
                    .frame(maxWidth: .infinity)
                    .padding()
                    .background(Color.green)
                Spacer()
            }
            .font(.title)
        }
    }
}
```

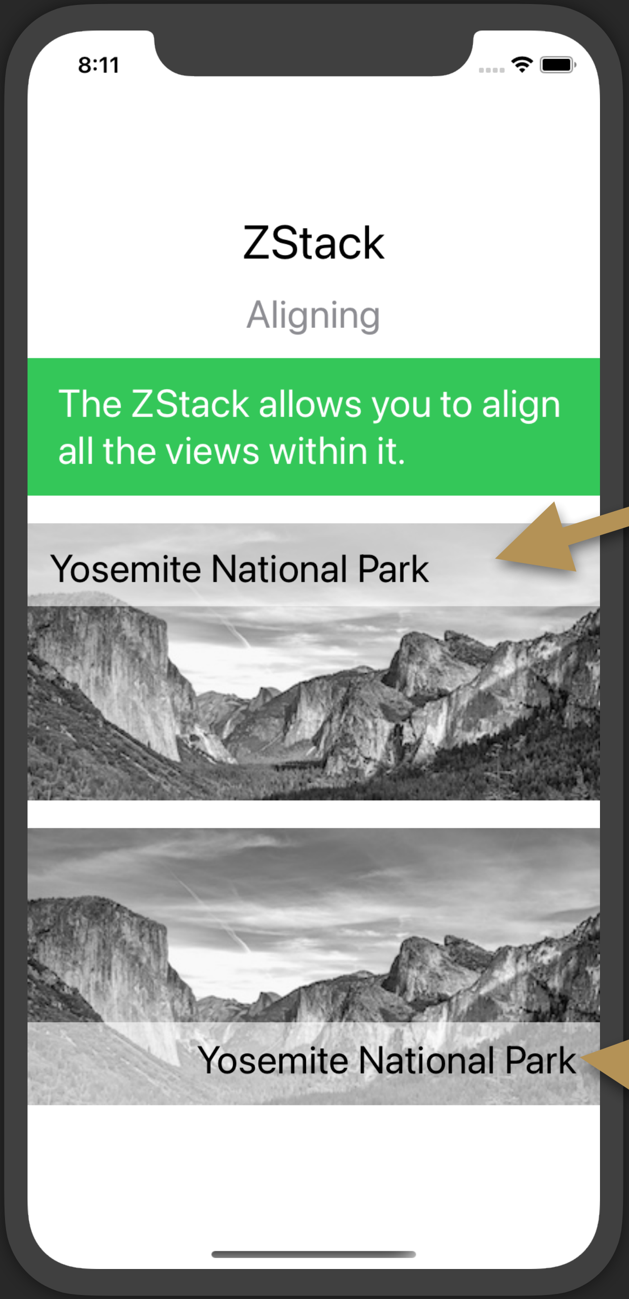
Remove ignoresSafeArea modifier from the ZStack and put it just on the color.

Layering


8:08
ZStack
Layering & Aligning
ZStacks are great for layering views. For example, putting text on top of an image.
Yosemite National Park
But what if you wanted to have all the views align to the bottom?

```
struct ZStack_Layering: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("ZStack",
                       subtitle: "Layering & Aligning",
                       desc: "ZStacks are great for layering views. For example, putting text on
                              top of an image.", back: .green, textColor: .white)

            ZStack {
                Image("yosemite_large")
                    .resizable() // Allows image to change size
                    .scaledToFit() // Keeps image the same aspect ratio when resizing

                Rectangle()
                    .fill(Color.white.opacity(0.6))
                    .frame(maxWidth: .infinity, maxHeight: 50)

                Text("Yosemite National Park")
                    .font(.title)
                    .padding()
            }

            DescView(desc: "But what if you wanted to have all the views align to the bottom?",
                     back: .green, textColor: .white)
        }
        .font(.title)
    }
}
```

Aligning



```
struct ZStack_Aligning: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("ZStack",
                       subtitle: "Aligning",
                       desc: "The ZStack allows you to align all the views within it.",
                       back: .green, textColor: .white)

            ZStack(alignment: .topLeading) {
                Image("yosemite_large")
                    .resizable()
                    .aspectRatio(contentMode: .fit)

                Rectangle()
                    .fill(Color.white.opacity(0.6))
                    .frame(maxWidth: .infinity, maxHeight: 60)

                Text("Yosemite National Park")
                    .font(.title)
                    .padding()
            }

            ZStack(alignment: .bottomTrailing) {
                Image("yosemite_large")
                    .resizable()
                    .aspectRatio(contentMode: .fit)
```

Use the alignment parameter in the ZStack's initializer to set where you want all views within to be aligned.

```
                Rectangle()
                    .fill(Color.white.opacity(0.6))
                    .frame(maxWidth: .infinity, maxHeight: 60)

                Text("Yosemite National Park")
                    .font(.title)
                    .padding()
            }
        }
        .font(.title)
    }
}
```

Alignment Choices

- center
- leading
- trailing
- top
- bottom
- topLeading
- topTrailing
- bottomLeading
- bottomTrailing

iOS 16

Grid

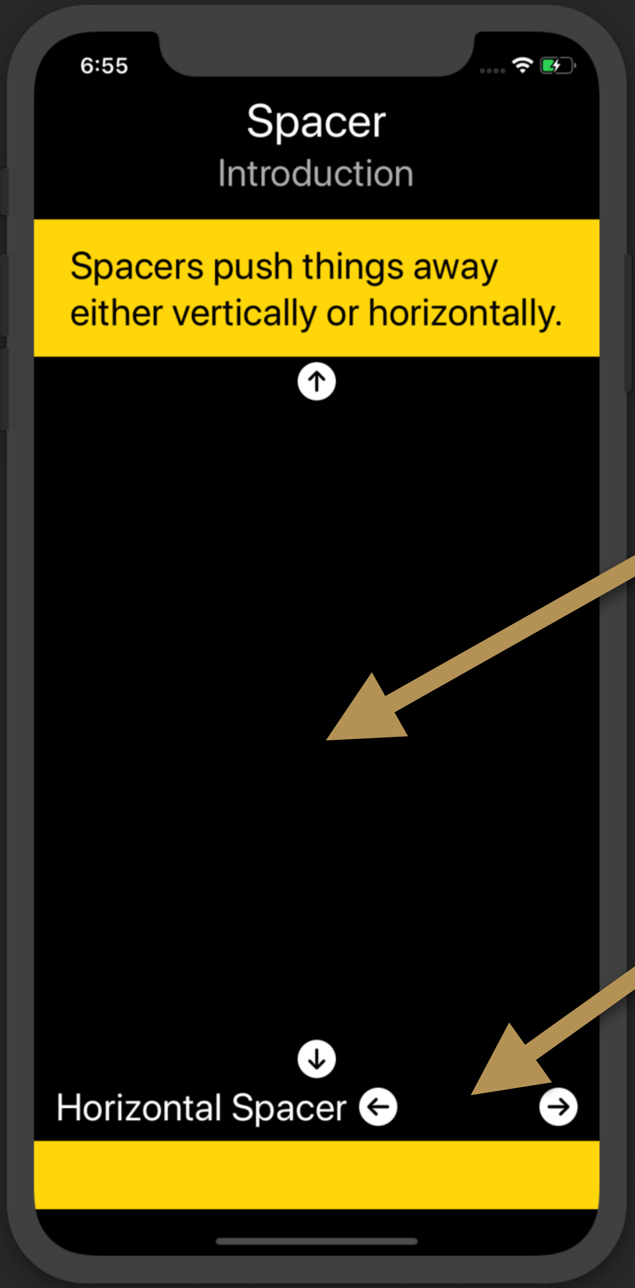
The grid will layout views in row olumns. What is different bout the grid and other layout views is that it will remember and maintain the widths of all the views in each row and apply the same width to all views in that same column.

Grids are pull-in views, they

Spacer

You may notice that when you add new pull-in views, such as Text views, they appear in the center of the screen. You can use the Spacer to push these views apart, away from the center of the screen.

Introduction


6:55

Spacer

Introduction

Spacers push things away either vertically or horizontally.

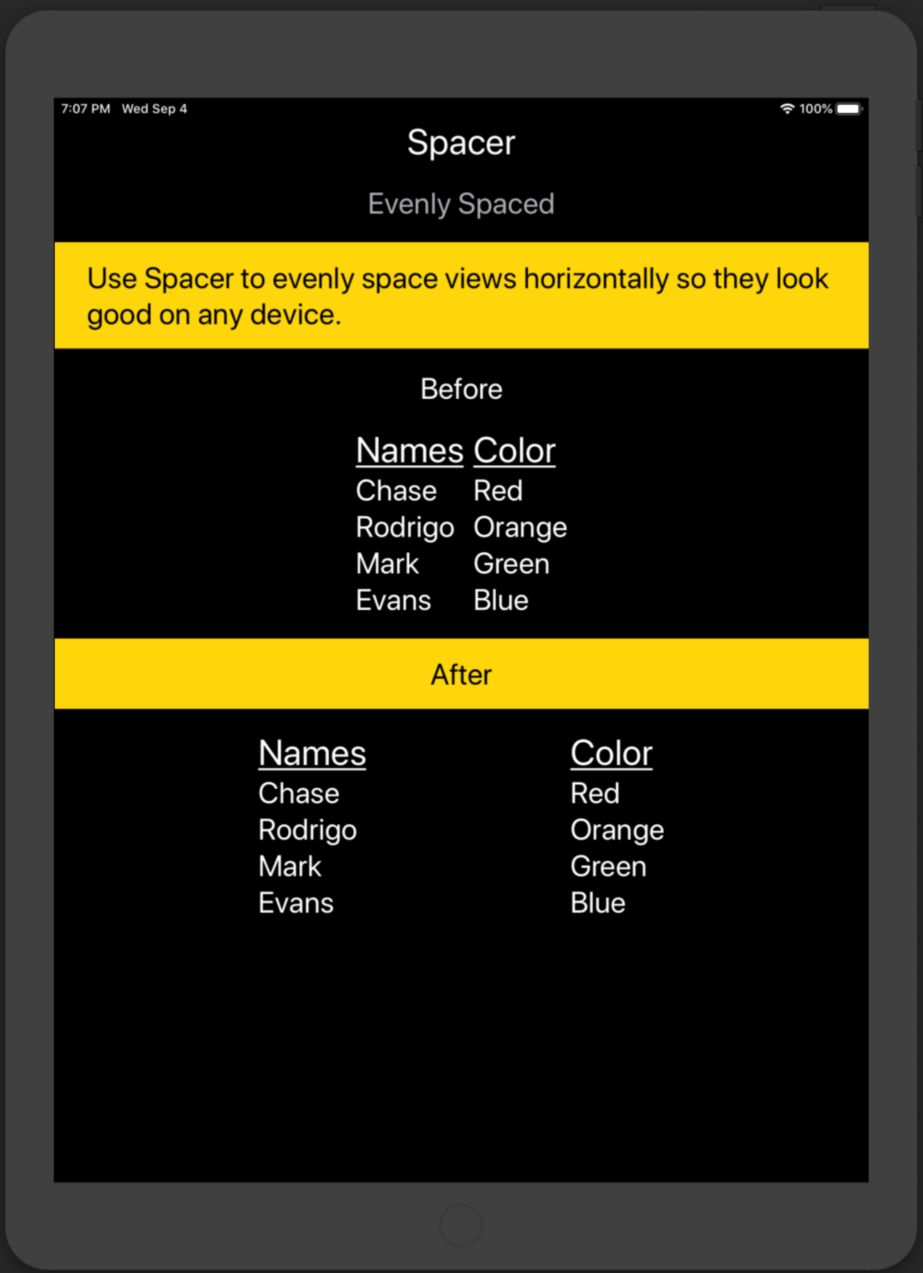
Horizontal Spacer

```
VStack {
    Text("Spacer")
        .font(.largeTitle)

    Text("Introduction")
        .foregroundColor(.gray)

    Text("Spacers push things away either vertically or horizontally")
        ...

    Image(systemName: "arrow.up.circle.fill")

    Spacer()

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

    HStack {
        Text("Horizontal Spacer")

        Image(systemName: "arrow.left.circle.fill")

        Spacer()

        Image(systemName: "arrow.right.circle.fill")
    }
    .padding(.horizontal)

    Color.yellow
        .frame(maxHeight: 50) // Height can decrease but not go higher than 50
}
.font(.title) // Apply this font to every view within the VStack
```

Pushes away vertically when in a VStack.

Pushes away horizontally when in an HStack.

Evenly Spaced



7:07 PM Wed Sep 4 100%

Spacer

Evenly Spaced

Use Spacer to evenly space views horizontally so they look good on any device.

Before

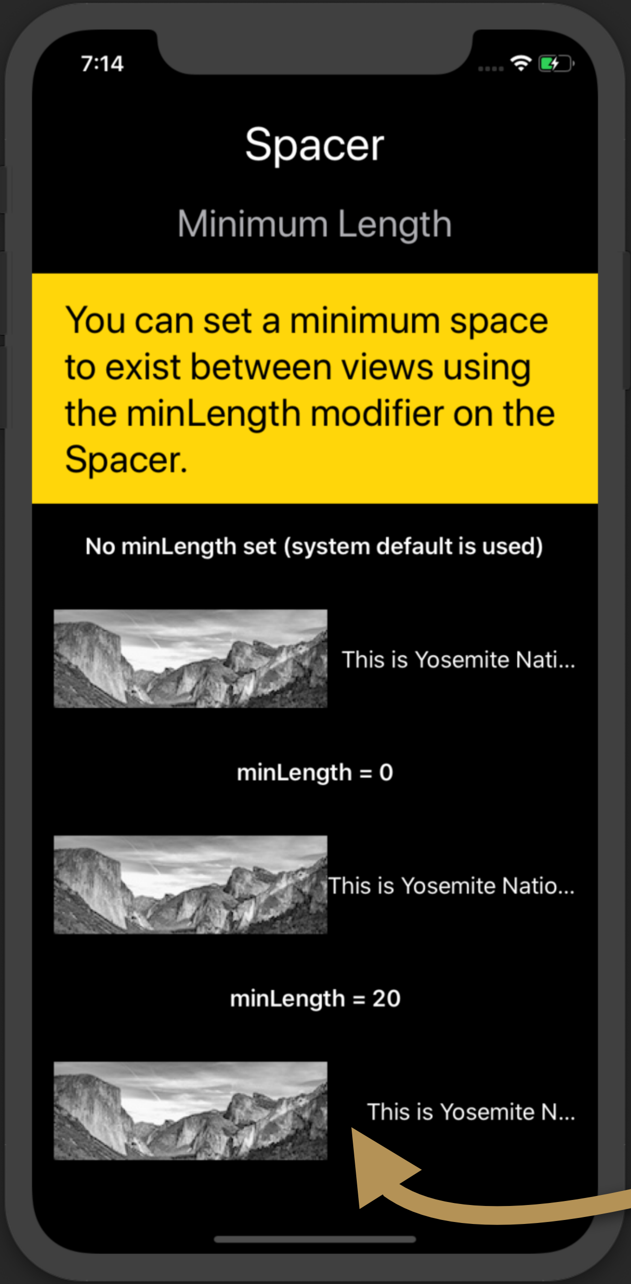
Names	Color
Chase	Red
Rodrigo	Orange
Mark	Green
Evans	Blue

After

Names	Color
Chase	Red
Rodrigo	Orange
Mark	Green
Evans	Blue

```
Text("Use Spacer to evenly space views horizontally so they look good on any
device.")
...
Text("After")
...
HStack {
    Spacer()

    VStack(alignment: .leading) {
        Text("Names")
            .font(.largeTitle)
            .underline()
        Text("Chase")
        Text("Rodrigo")
        Text("Mark")
        Text("Evans")
    }

    Spacer()

    VStack(alignment: .leading) {
        Text("Color")
            .font(.largeTitle)
            .underline()
        Text("Red")
        Text("Orange")
        Text("Green")
        Text("Blue")
    }

    Spacer()
}
```

Minimum Length


7:14

Spacer

Minimum Length

You can set a minimum space to exist between views using the minLength modifier on the Spacer.

No minLength set (system default is used)

This is Yosemite Nati...

minLength = 0

This is Yosemite Natio...

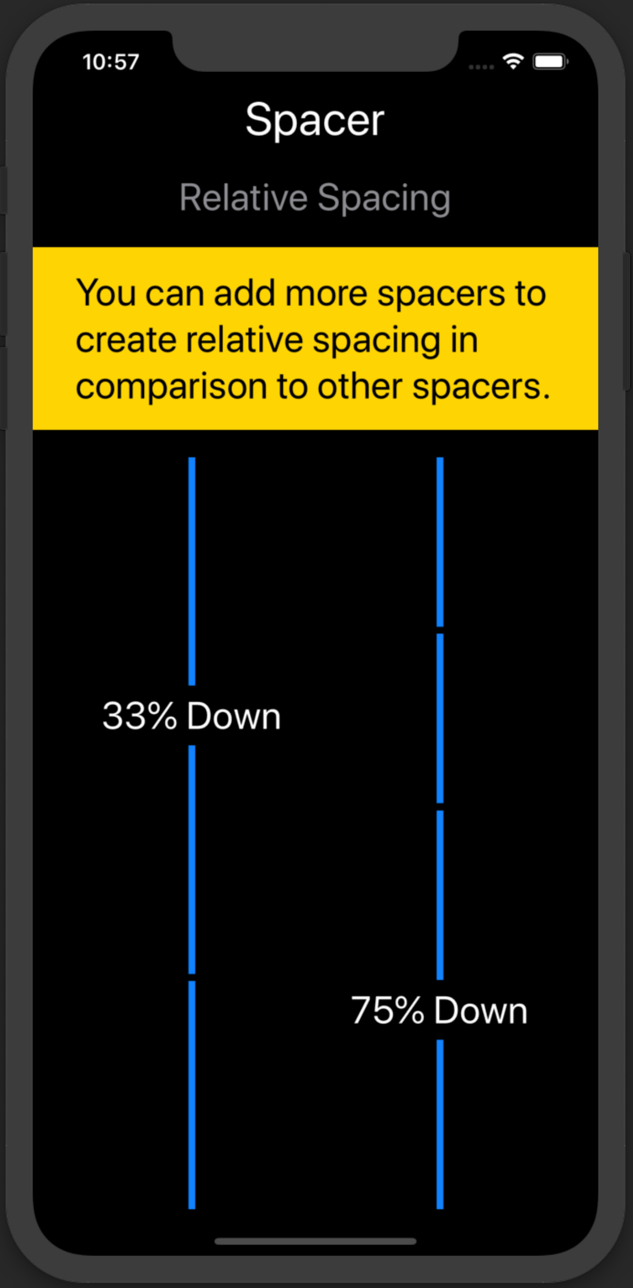
minLength = 20

This is Yosemite N...

```
VStack(spacing: 10) {
    Text("Spacer")
        .font(.largeTitle)
    Text("Minimum Length")
        .font(.title)
        .foregroundColor(.gray)
    Text("You can set a minimum space to exist between views using the minLength modifier on the
            Spacer.")
        ...
    Text("No minLength set (system default is used)")
        .bold()
    HStack {
        Image("yosemite")
        Spacer()
        Text("This is Yosemite National Park").lineLimit(1)
    }.padding()

    Text("minLength = 0")
        .bold()
    HStack {
        Image("yosemite")
        Spacer(minLength: 0)
        Text("This is Yosemite National Park").lineLimit(1)
    }.padding()

    Text("minLength = 20")
        .bold()
    HStack {
        Image("yosemite")
        Spacer(minLength: 20)
        Text("This is Yosemite National Park").lineLimit(1)
    }.padding()
}
```

Set the minimum length in the Spacer's initializer.

Relative Spacing with Spacers



You can add more spacers to create relative spacing in comparison to other spacers.

```
struct Spacer_RelativeSpacing: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Spacer").font(.largeTitle)
            Text("Relative Spacing").foregroundColor(.gray)
            Text("You can add more spacers to create relative spacing in comparison to other
                spacers.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.yellow).foregroundColor(.black)
            HStack(spacing: 50) {

                VStack(spacing: 5) {
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                    Text("33% Down")
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                }

                VStack(spacing: 5) {
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                    Spacer()
```

Spacers are views and can be modified like views.

```
                        .frame(width: 5)
                        .background(Color.blue)
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                    Text("75% Down")
                    Spacer()
                        .frame(width: 5)
                        .background(Color.blue)
                }
            }
        }.font(.title)
    }
}
```

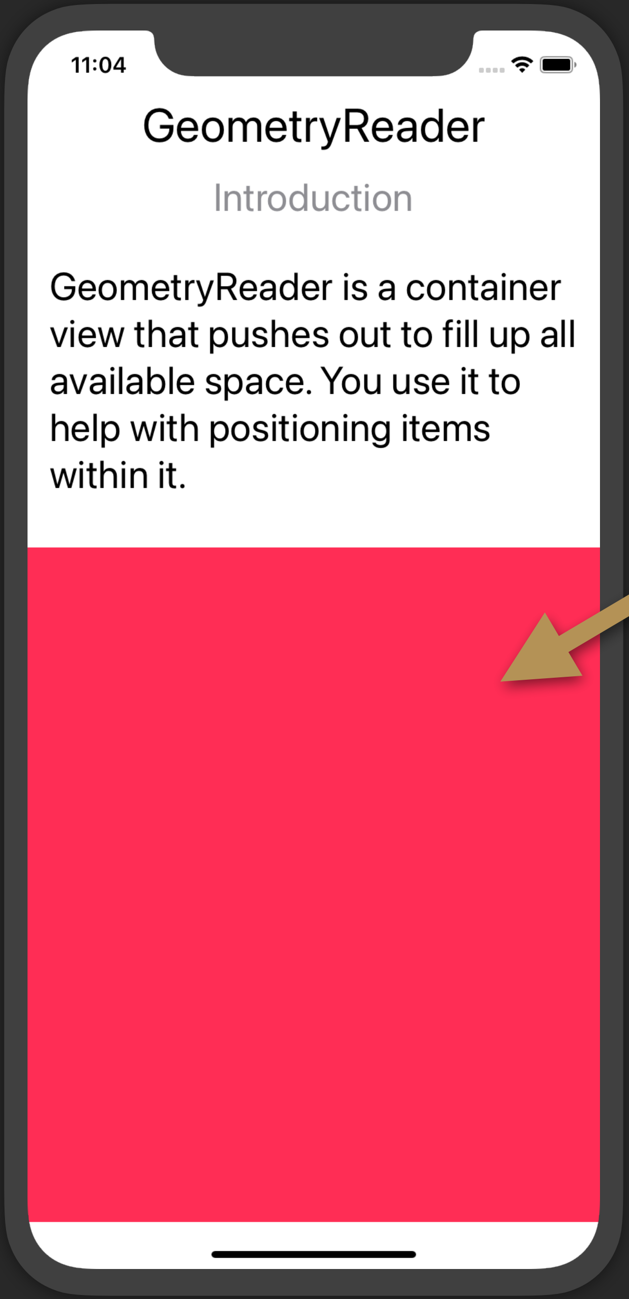
Note: You can also use Spacers horizontally to place views a percentage from the leading or trailing sides of the screen.

GeometryReader

It is difficult, if not impossible, to get the size of a view. This is where the GeometryReader can help.

The GeometryReader is similar to a push-out container view in that you can add child views to it. It will allow you to inspect and use properties that can help with positioning other views within it. You can access properties like height, width and safe area insets which can help you dynamically set the sizes of views within it so they look good on any size device.

Introduction


11:04

GeometryReader

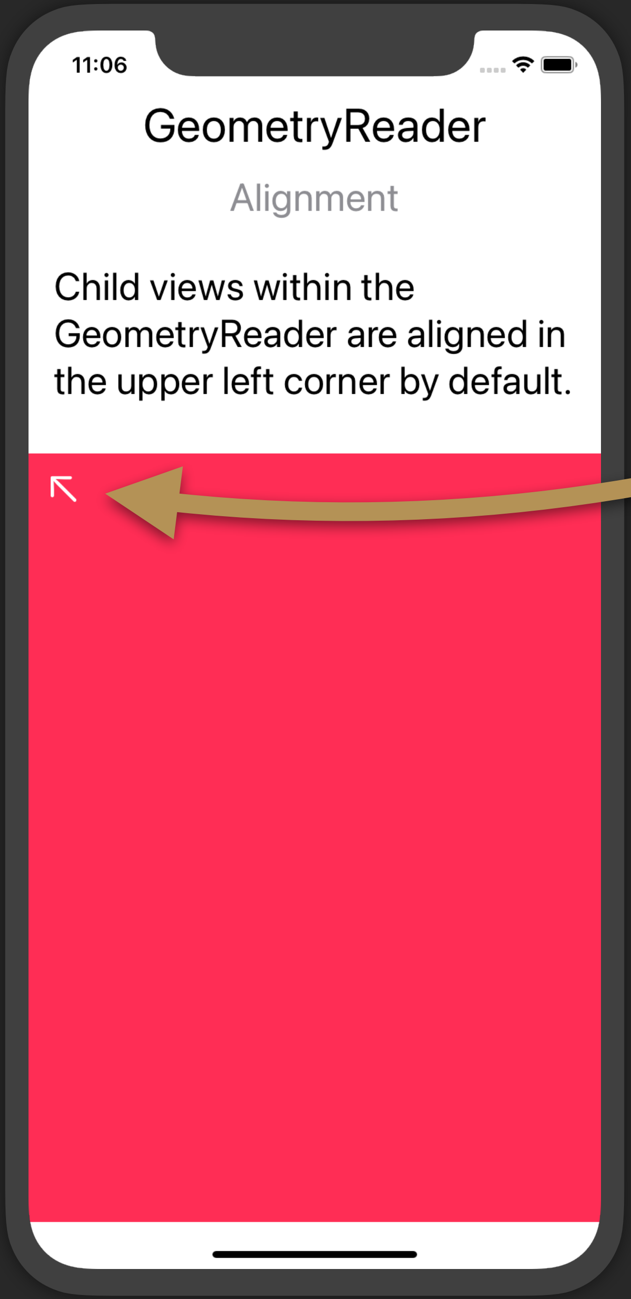
Introduction

GeometryReader is a container view that pushes out to fill up all available space. You use it to help with positioning items within it.

```
struct GeometryReader_Intro : View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("GeometryReader", subtitle: "Introduction", desc: "GeometryReader is a
container view that pushes out to fill up all available space. You use it to help with
positioning items within it.",
                       back: .clear)

            GeometryReader { _ in
                // No child views inside
            }
           .background(Color.pink)
        }
        .font(.title)
    }
}
```

In SwiftUI, when you see the word "**geometry**", think size and position.

Alignment



GeometryReader

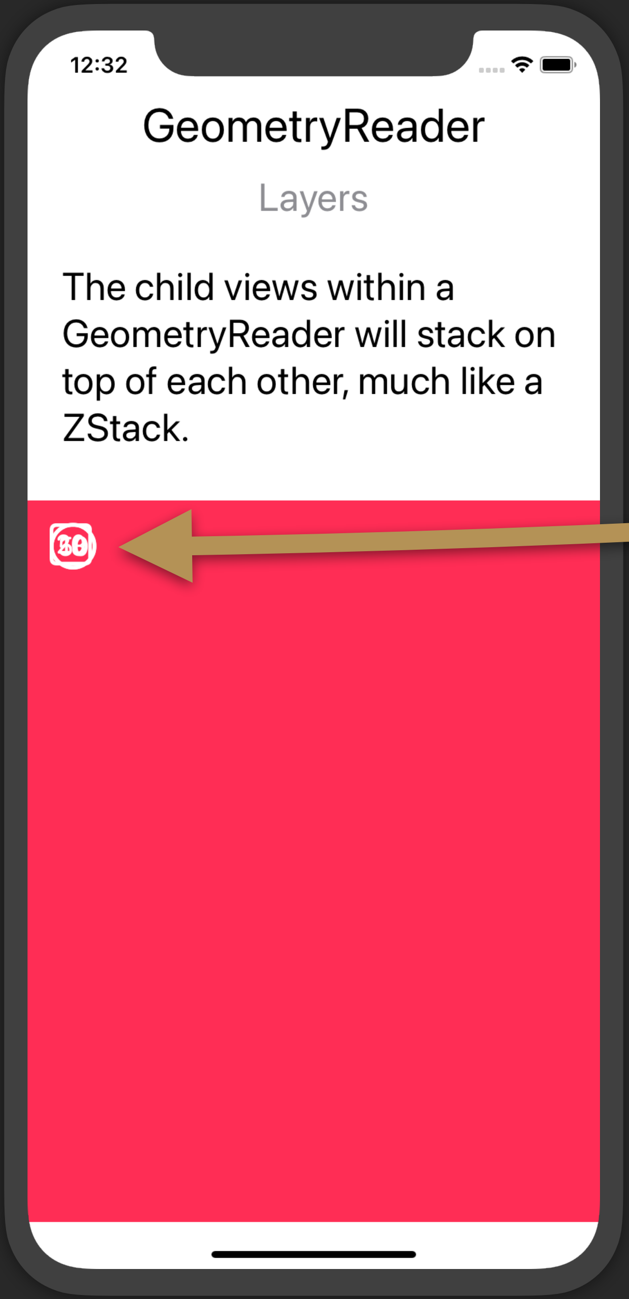
Alignment

Child views within the GeometryReader are aligned in the upper left corner by default.

```
struct GeometryReader_Alignment: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("GeometryReader", subtitle: "Alignment", desc: "Child views within the
GeometryReader are aligned in the upper left corner by default.", back: .clear)

            GeometryReader {_ in
                Image(systemName: "arrow.up.left")
                    .padding()
            }
            .background(Color.pink)
        }
        .font(.title)
    }
}
```

Notice that there is no alignment or positioning specified on the image.

Layers


12:32

GeometryReader

Layers

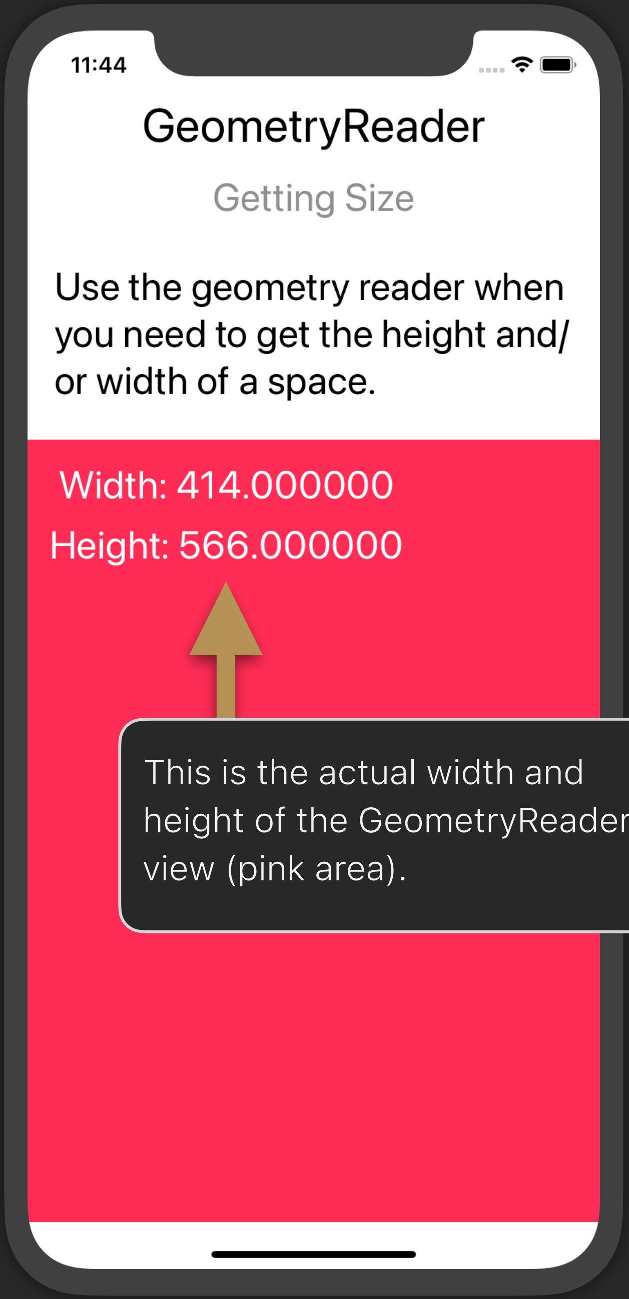
The child views within a GeometryReader will stack on top of each other, much like a ZStack.

```
struct GeometryReader_Layers: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("GeometryReader", subtitle: "Layers", desc: "The child views within a
GeometryReader will stack on top of each other, much like a ZStack.",
                       back: .clear)

            GeometryReader {_ in
                Image(systemName: "18.circle")
                    .padding()
                Image(systemName: "20.square")
                    .padding()
                Image(systemName: "50.circle")
                    .padding()
            }
            .font(.largeTitle)
            .foregroundColor(.white)
            .background(Color.pink)
        }
        .font(.title)
    }
}
```

Note, I wouldn't recommend using a GeometryReader in place of a ZStack.

ZStack provides convenient alignment options for layout that GeometryReader does not.

Getting Size



11:44

GeometryReader

Getting Size

Use the geometry reader when you need to get the height and/or width of a space.

Width: 414.000000

Height: 566.000000

This is the actual width and height of the GeometryReader view (pink area).

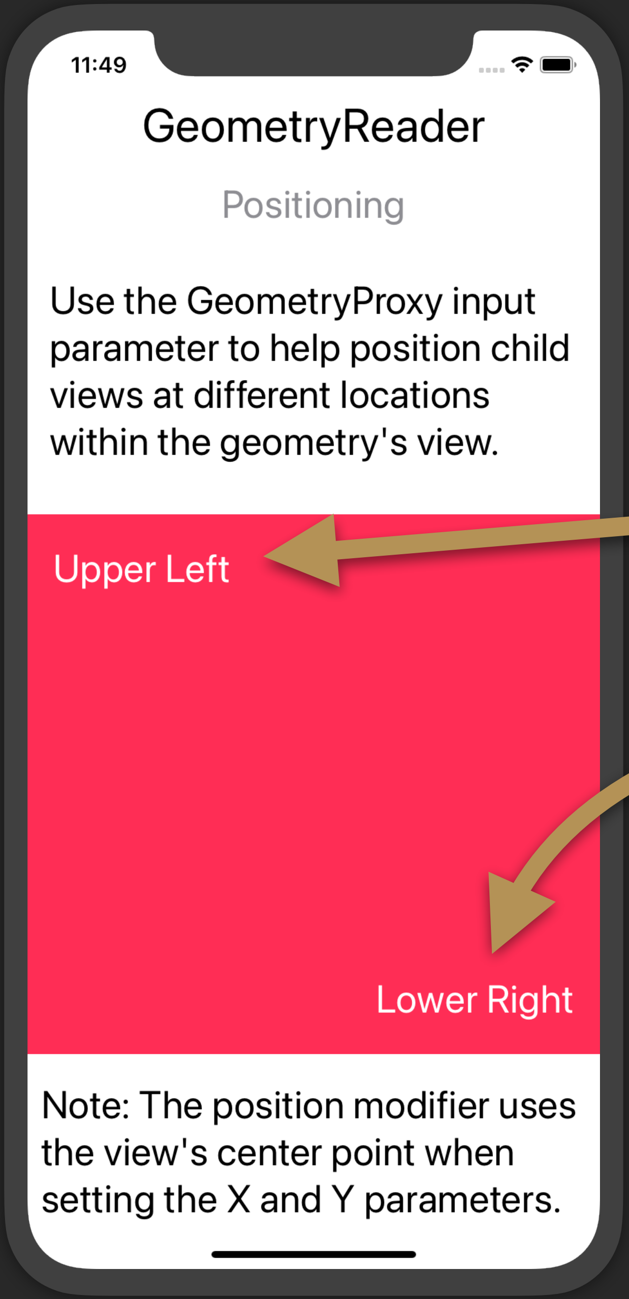
```
struct GeometryReader_GettingSize : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("GeometryReader", subtitle: "Getting Size", desc: "Use the geometry
reader when you need to get the height and/or width of a space.",
                       back: .clear)


            GeometryReader { geometryProxy in
                VStack(spacing: 10) {
                    Text("Width: \(geometryProxy.size.width)")
                    Text("Height: \(geometryProxy.size.height)")
                }
                .padding()
                .foregroundColor(.white)
            }
            .background(Color.pink)
        }
        .font(.title)
    }
}
```

Define a parameter to reference the geometry's coordinate space from a "proxy".

The GeometryProxy is a representation of the GeometryReader's size and coordinate space.

The geometryProxy.size will give you access to the height and width of the space the GeometryReader is taking up on the screen.

Positioning


11:49

GeometryReader

Positioning

Use the GeometryProxy input parameter to help position child views at different locations within the geometry's view.

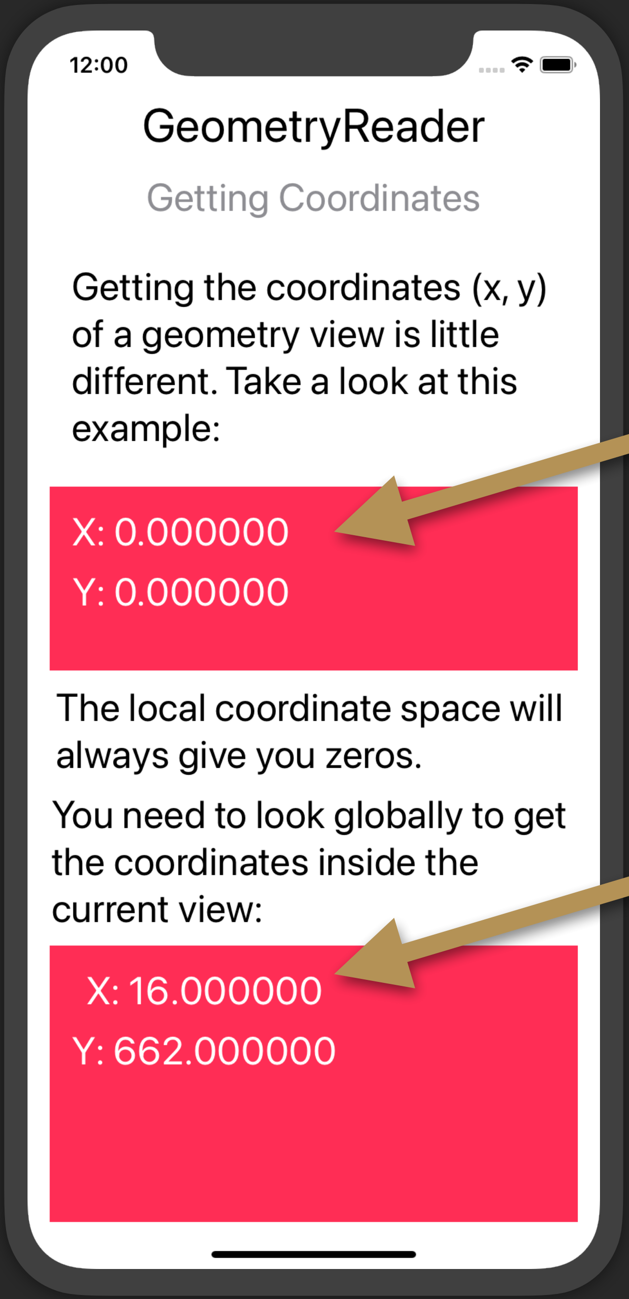
Upper Left

Lower Right

Note: The position modifier uses the view's center point when setting the X and Y parameters.

```swift
struct GeometryReader_Positioning: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("GeometryReader").font(.largeTitle)
            Text("Positioning").font(.title).foregroundColor(.gray)
            Text("Use the GeometryProxy input parameter to help position child views at
different locations within the geometry's view.")
                .font(.title)
                .padding()

            GeometryReader { geometryProxy in
                Text("Upper Left")
                    .font(.title)
                    .position(x: geometryProxy.size.width/5,
                              y: geometryProxy.size.height/10)

                Text("Lower Right")
                    .font(.title)
                    .position(x: geometryProxy.size.width - 90,
                              y: geometryProxy.size.height - 40)
            }
            .background(Color.pink)
            .foregroundColor(.white)

            Text("Note: The position modifier uses the view's center point when setting the X
and Y parameters.")
                .font(.title)
        }
    }
}
```

Getting Coordinates


12:00

GeometryReader

Getting Coordinates

Getting the coordinates (x, y) of a geometry view is little different. Take a look at this example:

X: 0.000000

Y: 0.000000

The local coordinate space will always give you zeros.

You need to look globally to get the coordinates inside the current view:

X: 16.000000

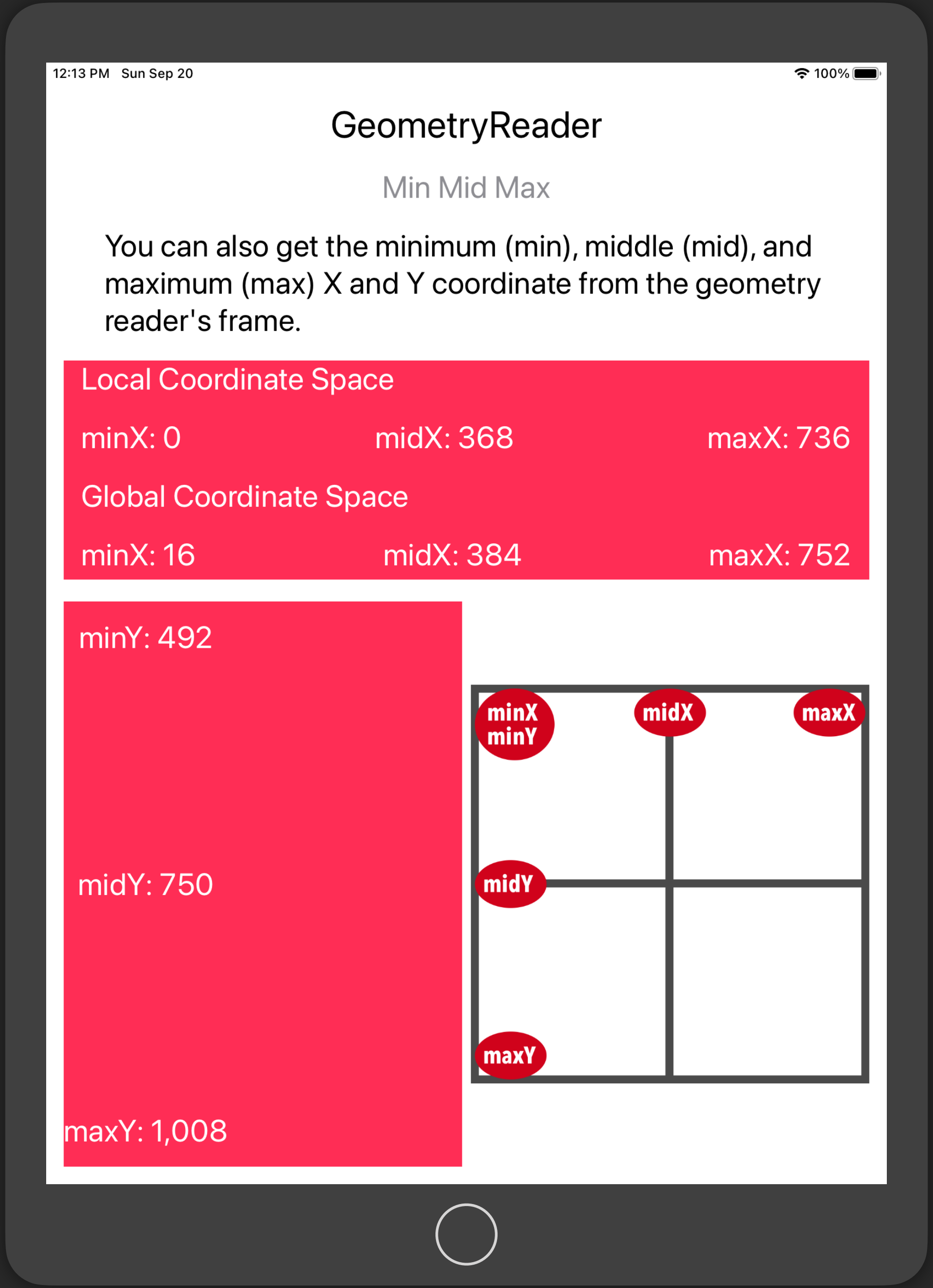
Y: 662.000000

```swift
struct GeometryReader_GettingCoordinates : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("GeometryReader", subtitle: "Getting Coordinates", desc: "Getting the
coordinates (x, y) of a geometry view is little different. Take a look at this example:",
back: .clear)

            GeometryReader { geometryProxy in
                VStack(spacing: 10) {
                    Text("X: \(geometryProxy.frame(in: CoordinateSpace.local).origin.x)")
                    Text("Y: \(geometryProxy.frame(in: CoordinateSpace.local).origin.y)")
                }
                .foregroundColor(.white)
            }
            .background(Color.pink)

            Text("The local coordinate space will always give you zeros.")
            Text("You need to look globally to get the coordinates inside the current view:")
            GeometryReader { geometryProxy in
                VStack(spacing: 10) {
                    Text("X: \(geometryProxy.frame(in: .global).origin.x)")
                    Text("Y: \(geometryProxy.frame(in: .global).origin.y)")
                }
                .foregroundColor(.white)
            }
            .background(Color.pink)
            .frame(height: 200)
        }
        .font(.title)
        .padding(.horizontal)
    }
}
```

I left out "CoordinateSpace" in this example (it's optional).

The global coordinate space is the entire screen. We are looking at the origin of the geometry proxy's frame within the entire screen.

Min Mid Max Coordinates



GeometryReader

Min Mid Max

You can also get the minimum (min), middle (mid), and maximum (max) X and Y coordinate from the geometry reader's frame.

Local Coordinate Space

minX: 0 midX: 368 maxX: 736

Global Coordinate Space

minX: 16 midX: 384 maxX: 752

minY: 492

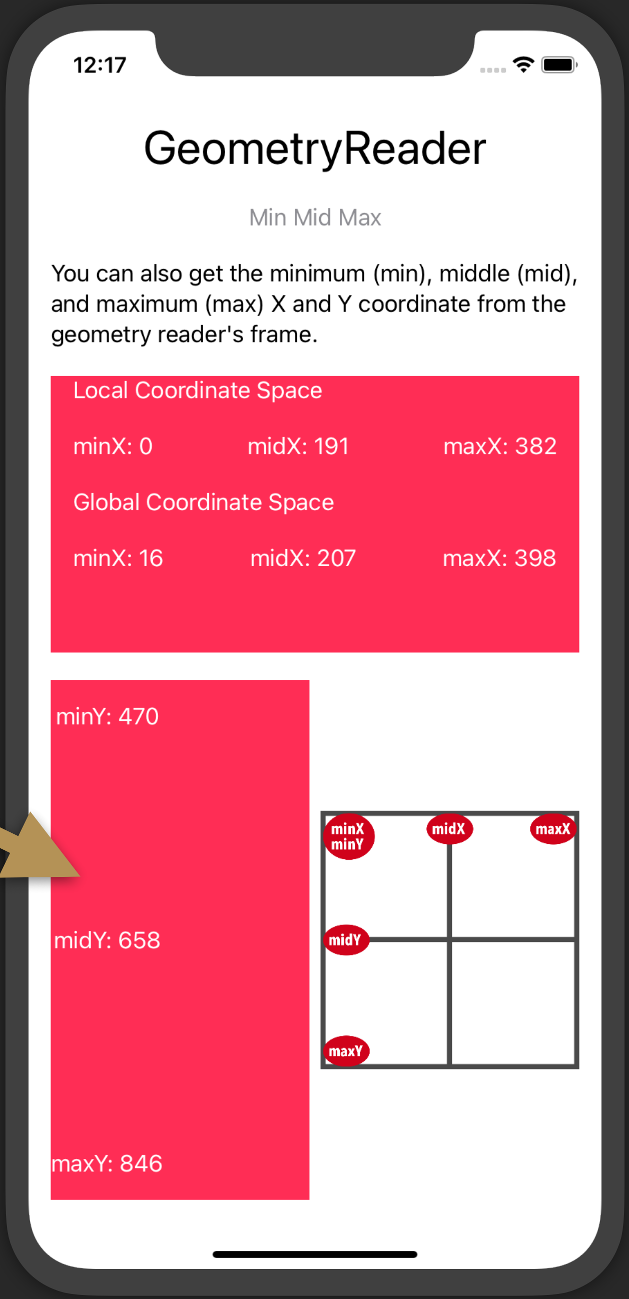
midY: 750

maxY: 1,008

```swift
struct GeometryReader_MinMidMax: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("GeometryReader", subtitle: "Min Mid Max", desc: "You can 
also get the minimum (min), middle (mid), and maximum (max) X and Y coordinate from 
the geometry reader's frame.", back: .clear)

            GeometryReader { geometry in
                VStack(alignment: .leading, spacing: 20) {
                    Text("Local Coordinate Space")
                    HStack(spacing: 10) {
                        // I'm converting to Int just so we don't have so many zeros
                        Text("minX: \(Int(geometry.frame(in: .local).minX))")
                        Spacer()
                        Text("midX: \(Int(geometry.frame(in: .local).midX))")
                        Spacer()
                        Text("maxX: \(Int(geometry.frame(in: .local).maxX))")
                    }
                    Text("Global Coordinate Space")
                    HStack(spacing: 10) {
                        // I'm converting to Int just so we don't have so many zeros
                        Text("minX: \(Int(geometry.frame(in: .global).minX))")
                        Spacer()
                        Text("midX: \(Int(geometry.frame(in: .global).midX))")
                        Spacer()
                        Text("maxX: \(Int(geometry.frame(in: .global).maxX))")
                    }
                }.padding(.horizontal)
            }
```

Min Mid Max Coordinates Continued

```
            .frame(height: 200)
            .foregroundColor(.white)
            .background(Color.pink)

            HStack {
                GeometryReader { geometry in
                    VStack(spacing: 10) {
                        Text("minY: \(Int(geometry.frame(in: .global).minY))")
                        Spacer()
                        Text("midY: \(Int(geometry.frame(in: .global).midY))")
                        Spacer()
                        Text("maxY: \(Int(geometry.frame(in: .global).maxY))")
                    }.padding(.vertical)
                }
                .foregroundColor(.white)
                .background(Color.pink)

                Image("MinMidMax")
                    .resizable()
                    .aspectRatio(contentMode: .fit)
            }
        }
        .font(.title)
        .padding()
    }
}
```


12:17

GeometryReader

Min Mid Max

You can also get the minimum (min), middle (mid), and maximum (max) X and Y coordinate from the geometry reader's frame.

Local Coordinate Space

minX: 0 midX: 191 maxX: 382

Global Coordinate Space

minX: 16 midX: 207 maxX: 398

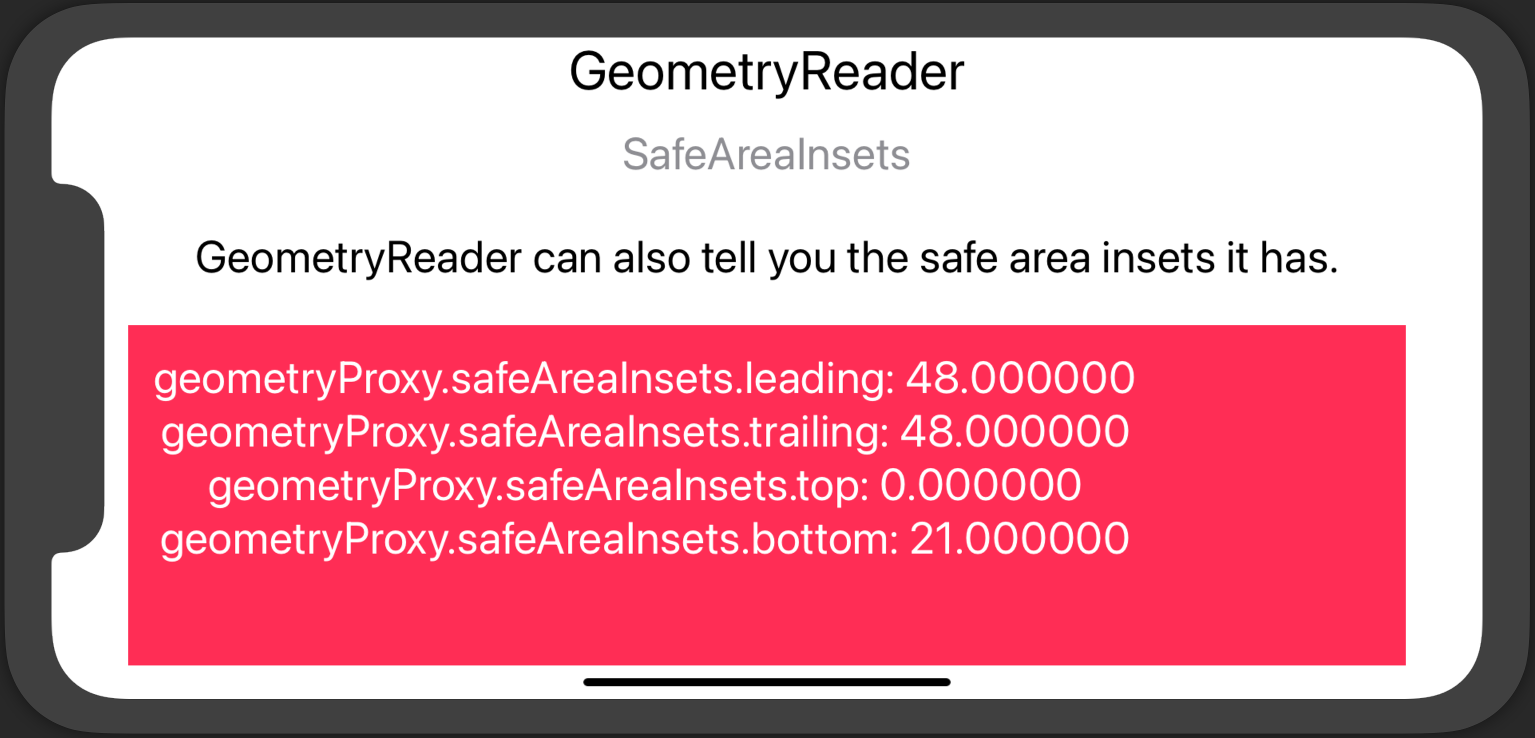
minY: 470

midY: 658

maxY: 846

Notice how the min, mid and max values change as the geometry reader adapts to different device sizes.

Safe Area Insets


GeometryReader

SafeAreaInsets

GeometryReader can also tell you the safe area insets it has.

```
geometryProxy.safeAreaInsets.leading: 48.000000
geometryProxy.safeAreaInsets.trailing: 48.000000
geometryProxy.safeAreaInsets.top: 0.000000
geometryProxy.safeAreaInsets.bottom: 21.000000
```

```
HeaderView("GeometryReader", subtitle: "SafeAreaInsets", desc: "GeometryReader can also tell you the safe area insets it has.",
back: .clear)

GeometryReader { geometryProxy in
    VStack {
        Text("geometryProxy.safeAreaInsets.leading: \(geometryProxy.safeAreaInsets.leading)")
        Text("geometryProxy.safeAreaInsets.trailing: \(geometryProxy.safeAreaInsets.trailing)")
        Text("geometryProxy.safeAreaInsets.top: \(geometryProxy.safeAreaInsets.top)")
        Text("geometryProxy.safeAreaInsets.bottom: \(geometryProxy.safeAreaInsets.bottom)")
    }
    .padding()
}
.background(Color.pink)
.foregroundColor(.white)
```

iOS 14

LazyHGrid

This SwiftUI chapter is locked in this preview.

Similar to an HStack, the Lazy Horizontal Grid will layout views horizontally but can be configured to use multiple rows and scroll off the screen. The word "lazy" here means that the child views are only created when SwiftUI needs them. This is called "lazy loading".

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

iOS 14

LazyVGrid

Similar to an HStack, the Lazy Vertical Grid will layout views vertically but can be configured to use multiple columns and scroll off the screen. The word "lazy" here means that the child views are only created when SwiftUI needs them. This is called "lazy loading".

iOS 14

ScrollViewReader

This SwiftUI chapter is locked in this preview.

The Scroll View Reader g… scroll view visible by aut… …make a view within a

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

iOS 15

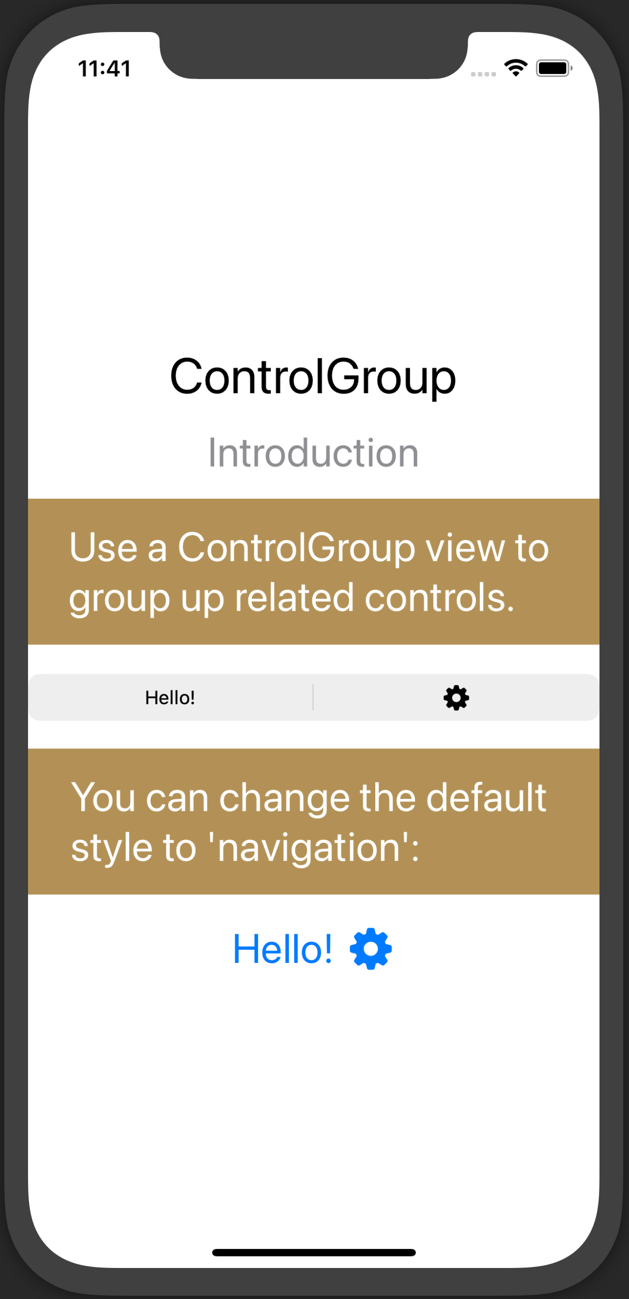
ControlGroup

Use the ControlGroup to put similar types of controls together, such as buttons. In my opinion, the use of this seems limited.

This is a pull-in view.

iOS 15

Introduction


11:41

ControlGroup

Introduction

Use a ControlGroup view to group up related controls.

You can change the default style to 'navigation':

```swift
struct ControlGroup_Intro: View {
    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("ControlGroup",
                       subtitle: "Introduction",
                       desc: "Use a ControlGroup view to group up related controls.")

            ControlGroup {
                Button("Hello!") { }
                Button(action: {}) {
                    Image(systemName: "gearshape.fill")
                }
            }

            DescView(desc: "You can change the default style to 'navigation':")
            ControlGroup {
                Button("Hello!") { }
                Button(action: {}) {
                    Image(systemName: "gearshape.fill")
                }
            }
            .controlGroupStyle(.navigation)
        }
        .font(.title)
    }
}
```

Note: You may ask yourself when you would use this.

I think it makes more sense inside of toolbars as well as on macOS.

iOS 16

Table

You can bind a collection of objects to a table and then each property to a table column.

In iOS though you will only s

iOS 16

ViewThatFits

ViewThatFits is a very unique c that can contain multiple views but will only draw **one** of the views. That's right, it will only draw the view that fits the space it is g

This SwiftUI chapter is locked in this preview.

You specify the larger view firs en progressively smaller views after that. The first view's size will be evaluated and if it can't fit in the given space, then the next view's size will be evaluated and so on.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

If none of the views fit, the l

iOS 17

ContainerRelativeFrame

This SwiftUI chapter is locked in this preview.

The containerRelativeFrame m[illegible]sed to help you change the size of your views, relative to the size of the container it is within.

⚠️By container, I do **not** mean a[illegible]ontainer view like an HStack or VStack. SwiftUI is very specific as to what a "container" is.

💡A container can be the d[illegible]w (ScrollView, Form, List).

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

iOS 17

ContentUnavailableView

This SwiftUI chapter is locked in this preview.

The ContentUnavailableView is [illegible]d to be displayed when co[illegible] your app is unavailable. But as a layout view, you can use it to quickly put together text, image and even buttons in a pre-formatted way.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

CONTROL VIEWS

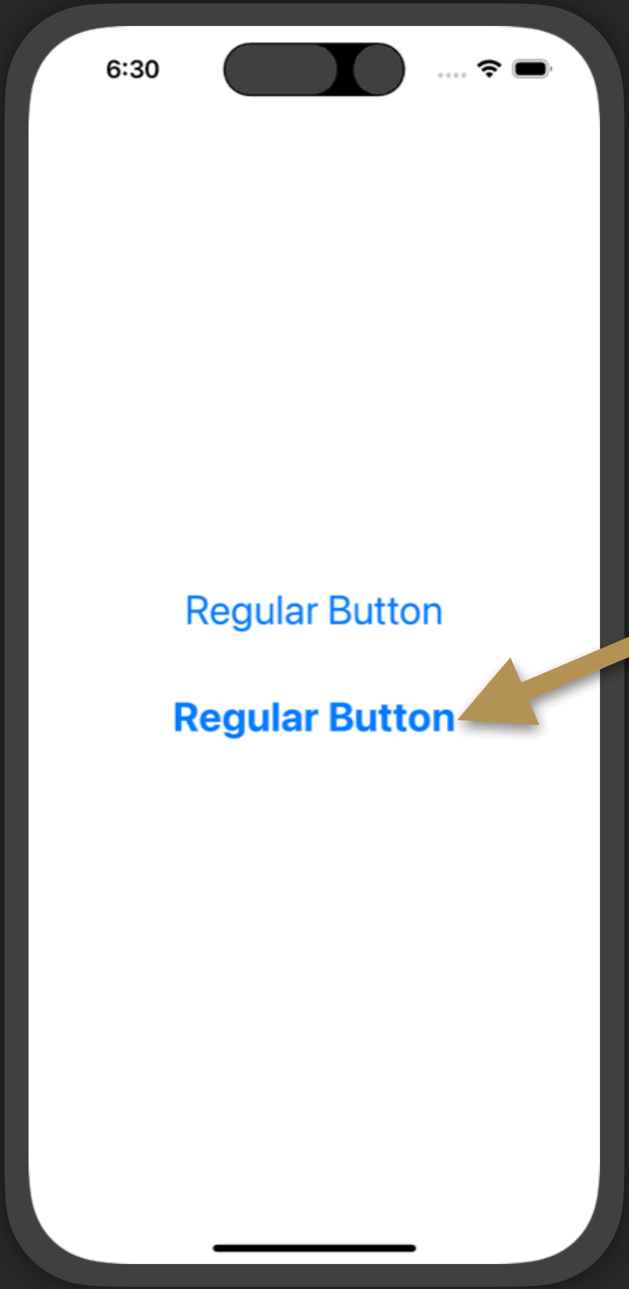
Chapter Quick Links

Return to Book Quick Links

Button

The Button is a pull-in view with a wide range of composition and customization options to be presented to the user. The button can be just text, just an image or both combined.

Introduction



```swift
struct RegularButton : View {
    var body: some View {
        VStack(spacing: 40) {
            Button("Regular Button") {
                // Code here
            }
            
            Button {
                // Code here
            } label: {
                Text("Regular Button")
                    .bold()
            }
        }
        .font(.title) // Make all fonts use the title style
    }
}
```

If you just want to show the default text style in a button then you can pass in a string as the first parameter.

This is the quickest and easiest way to make a button.

Use this initializer to customize the content within the button.

For more text customization options, see the chapter on Text.

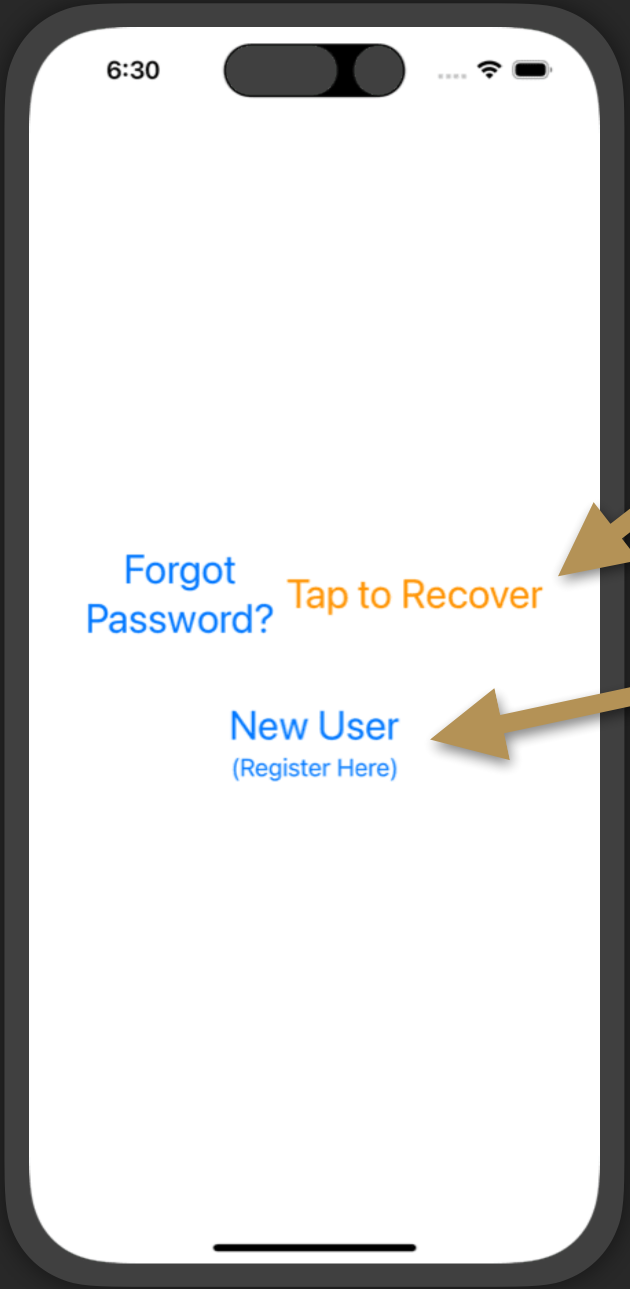
Text Composition

```
struct Button_TextModifiers : View {
    var body: some View {
        VStack(spacing: 40) {
            Button {} label: {
                Text("Forgot Password?")
                Text("Tap to Recover")
                    .foregroundStyle(.orange)
            }

            Button {} label: {
                VStack {
                    Text("New User")
                    Text("(Register Here)").font(.body)
                }
            }
        }
        .font(.title)
    }
}
```

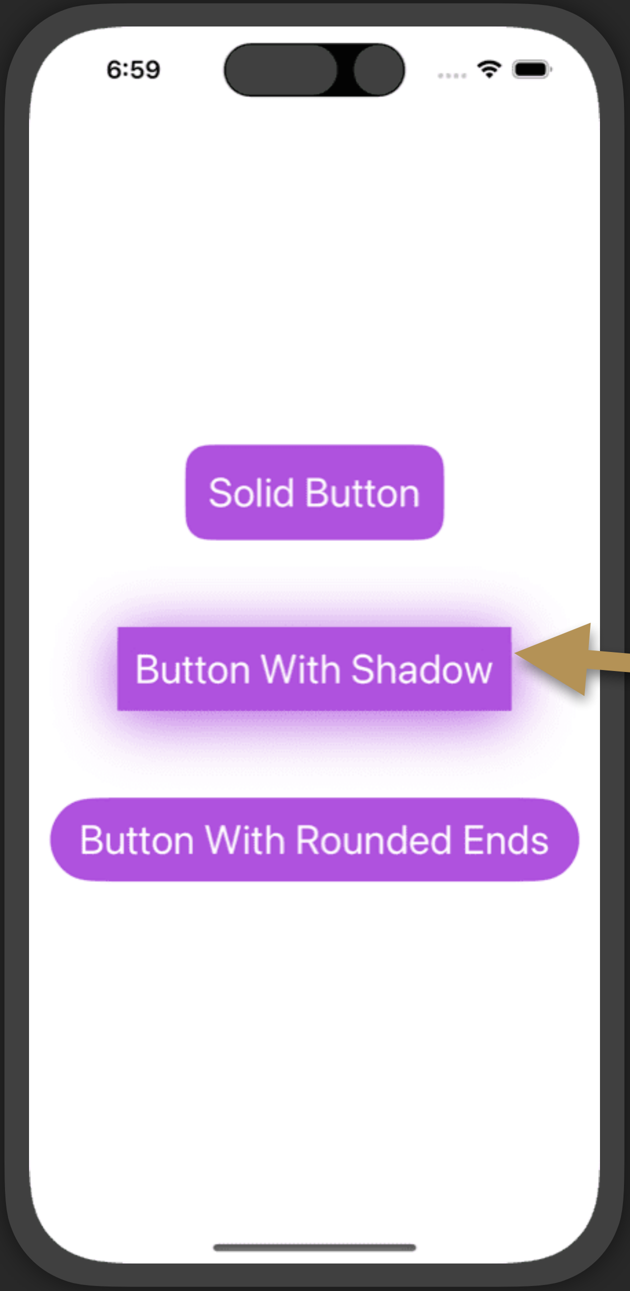
You can add more than one text view to a button and format them separately.

Views arranged horizontally by default.

You can use any layout or container view you want within the label content.



With Backgrounds



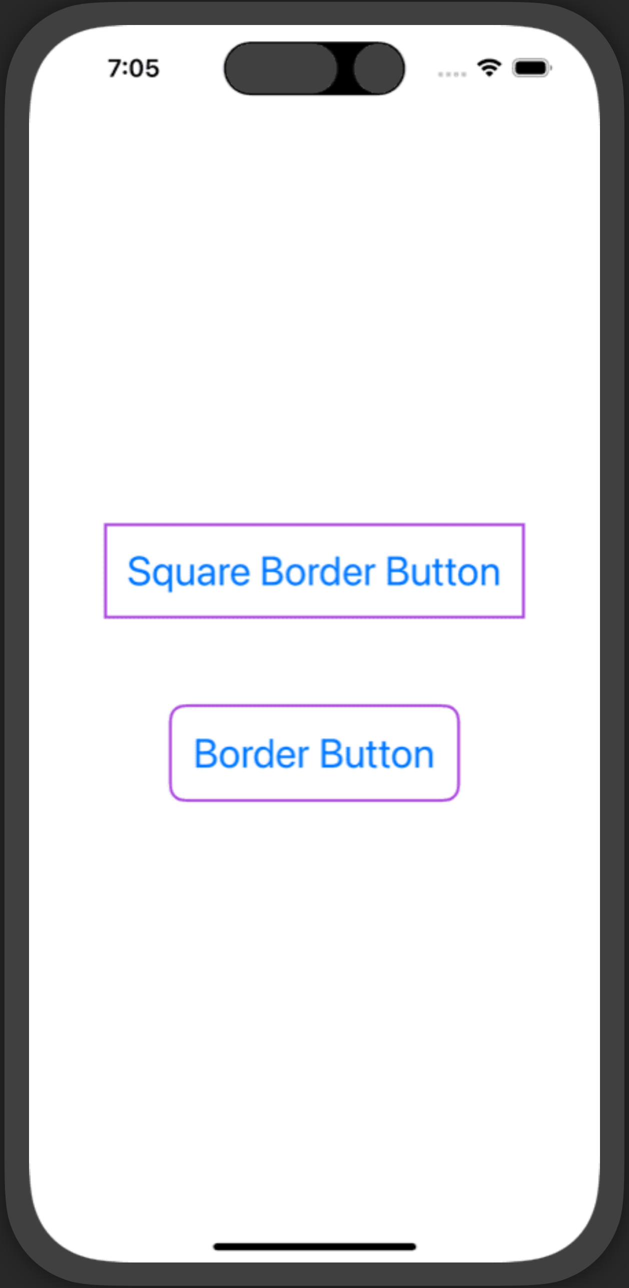
```swift
struct Button_WithBackgrounds : View {
    var body: some View {
        VStack(spacing: 60) {
            Button(action: {}) {
                Text("Solid Button")
                    .padding()
                    .foregroundStyle(.white)
                    .background(Color.purple)
                    .clipShape(RoundedRectangle(cornerRadius: 16))
            }
            Button(action: {}) {
                Text("Button With Shadow")
                    .padding(12)
                    .foregroundStyle(.white)
                    .background(Color.purple)
                    .shadow(color: Color.purple, radius: 20, y: 5)
            }

            Button(action: {}) {
                Text("Button With Rounded Ends")
                    .padding(EdgeInsets(top: 12, leading: 20, bottom: 12, trailing: 20))
                    .foregroundStyle(.white)
                    .background(Color.purple, in: Capsule())
            }
        }
        .font(.title)
    }
}
```

You can add various backgrounds and shadows to buttons to further customize them.

Note: It is a good idea to put your modifiers within the button instead of on the button itself. So when the button is tapped, the whole button flashes.

You can apply shapes as backgrounds; such as Capsule, Rectangle, Circle, and RoundedRectangle.

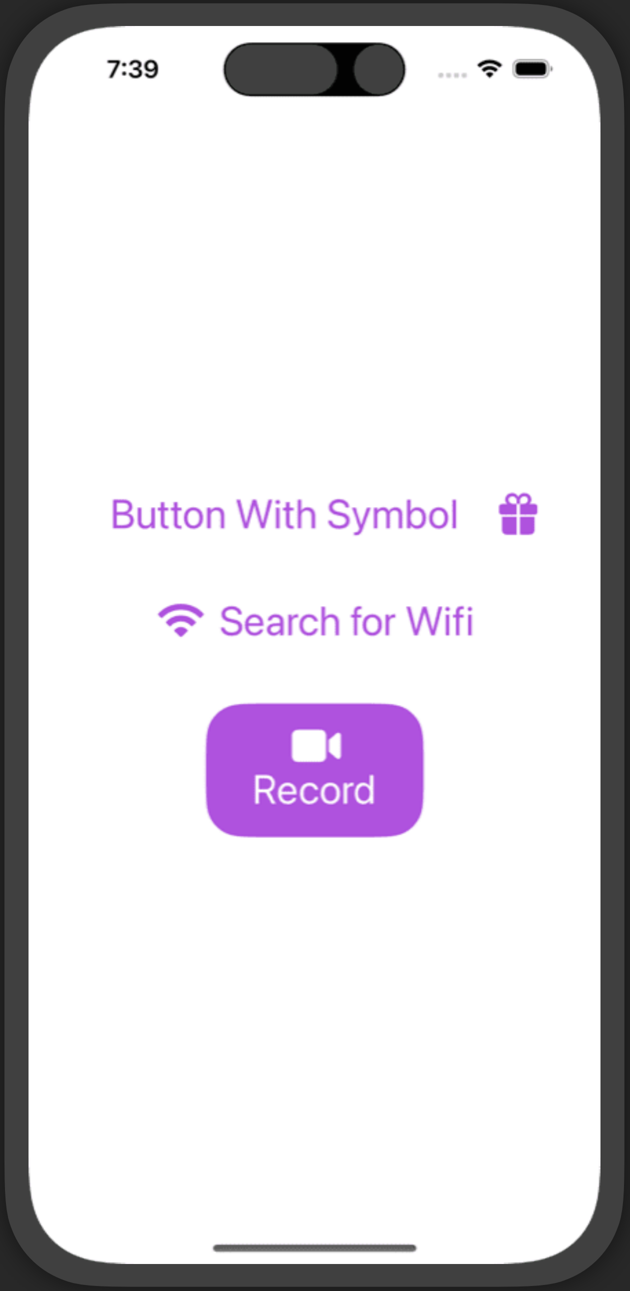
With Borders



```
struct Button_WithBorders: View {
    var body: some View {
        VStack(spacing: 60) {
            Button(action: {}) {
                Text("Square Border Button")
                    .padding()
                    .border(Color.purple, width: 2)
            }

            Button(action: {}) {
                Text("Border Button")
                    .padding()
                    .background {
                        RoundedRectangle(cornerRadius: 10)
                            .stroke(Color.purple, lineWidth: 2)
                    }
            }
        }
        .font(.title)
    }
}
```

Here are a couple of options if you are looking for a way to add a border around your button.

With SF Symbols



```
struct Button_WithSymbols : View {
    var body: some View {
        VStack(spacing: 40) {
            Button(action: {}) {
                Text("Button With Symbol")
                    .padding(.horizontal)
                Image(systemName: "gift.fill")
            }

            Button(action: {}) {
                Label("Search for Wifi", systemImage: "wifi")
            }

            Button(action: {}) {
                VStack {
                    Image(systemName: "video.fill")
                    Text("Record")
                        .padding(.horizontal)
                }
                .padding()
                .foregroundStyle(Color.white)
                .background(RoundedRectangle(cornerRadius: 25))
            }
        }
        .tint(.purple)
        .font(.title)
    }
}
```

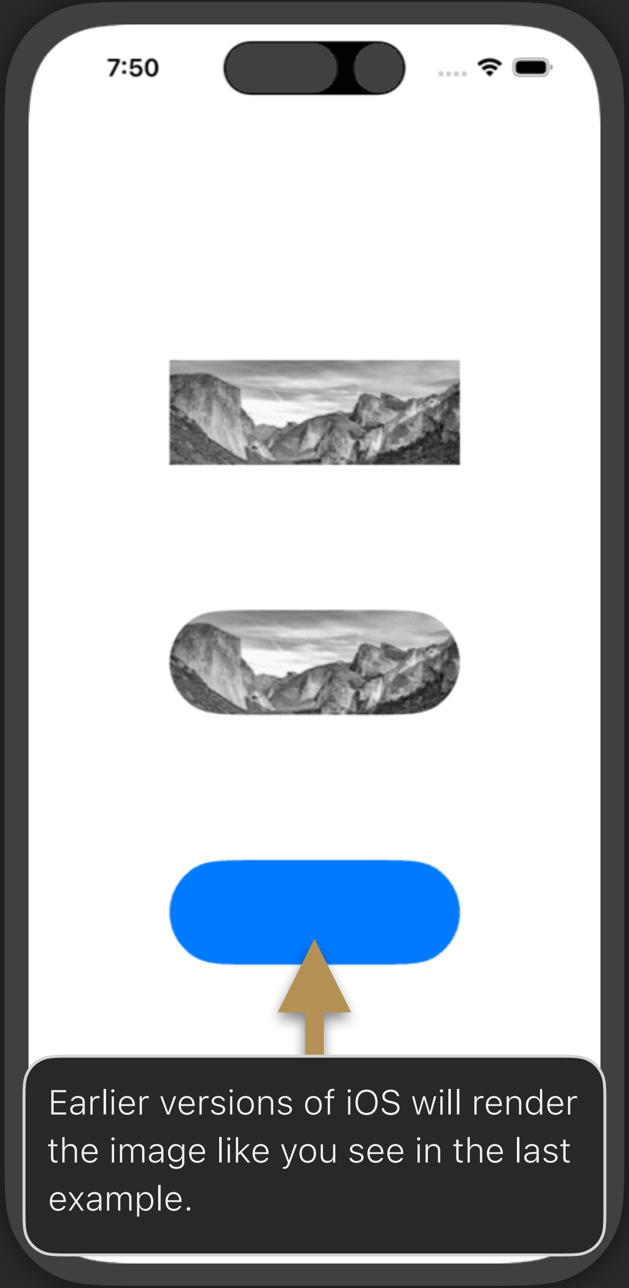
You can add SF Symbols to your buttons with the Image(systemName:) view.

Or you can use what is called a Label that groups texts and an SF Symbol together.

Or use a layout view to arrange your text and images.

For even more ways to customize buttons, see the chapter on Paints where you can learn how to apply the 3 different gradients to them.

With Images



Earlier versions of iOS will render the image like you see in the last example.

```
struct Button_WithPhotos: View {
    var body: some View {
        VStack(spacing: 100) {
            Button(action: {}) {
                Image("yosemite")
            }

            Button(action: {}) {
                Image("yosemite")
                    .renderingMode(.original)
                    .clipShape(Capsule())
            }

            Button(action: {}) {
                Image("yosemite")
                    .renderingMode(.template)
                    .clipShape(Capsule())
            }
        }
        .font(.title)
    }
}
```

If you are using a photo for the button content, it should show up fine from iOS 15+.

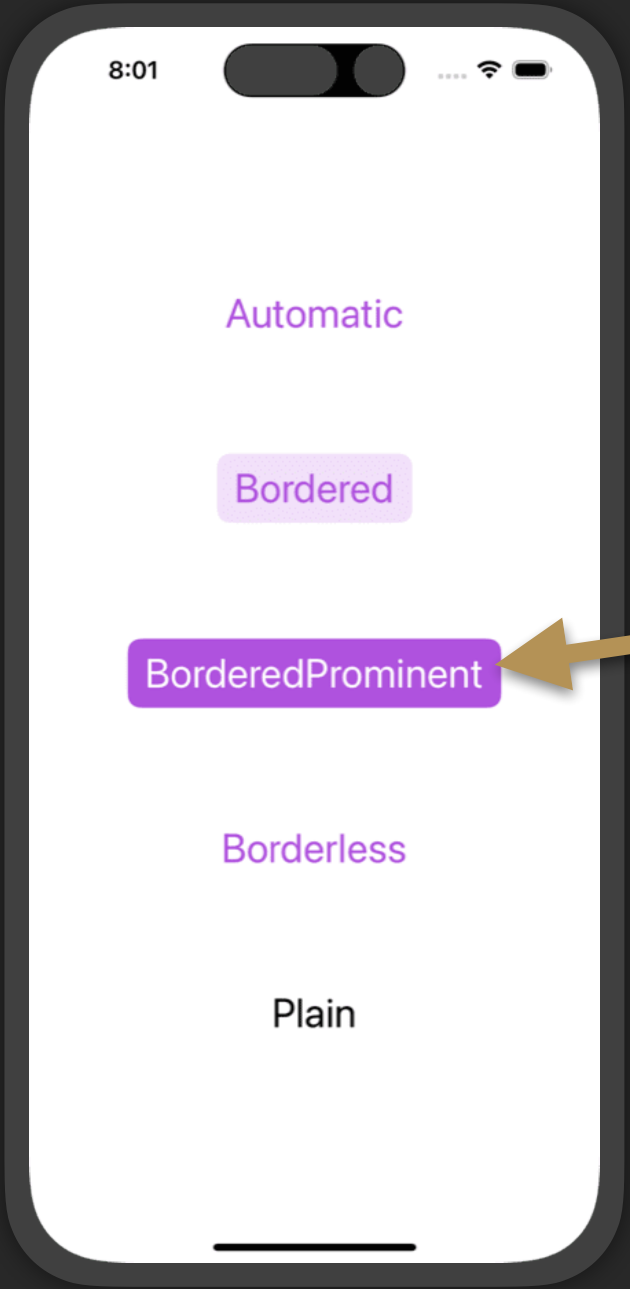
Before iOS 15 you might have to add the renderingMode modifier to make the image show.

Notice I'm also using a clipShape modifier to modify the shape of the image.

If your image has transparency, you can change the renderingMode to template.
This allows you to assign the template different colors.
By default, all buttons use a blue tint that you can override.

ButtonStyle

iOS 15



```
struct Button_ButtonStyle: View {
    var body: some View {
        VStack(spacing: 80.0) {
            Button("Automatic") { }
                .buttonStyle(.automatic)

            Button("Bordered") { }
                .buttonStyle(.bordered)

            Button("BorderedProminent") { }
                .buttonStyle(.borderedProminent)

            Button("Borderless") { }
                .buttonStyle(.borderless)

            Button("Plain") { }
                .buttonStyle(.plain)
        }
        .font(.title)
        .tint(.purple)
    }
}
```

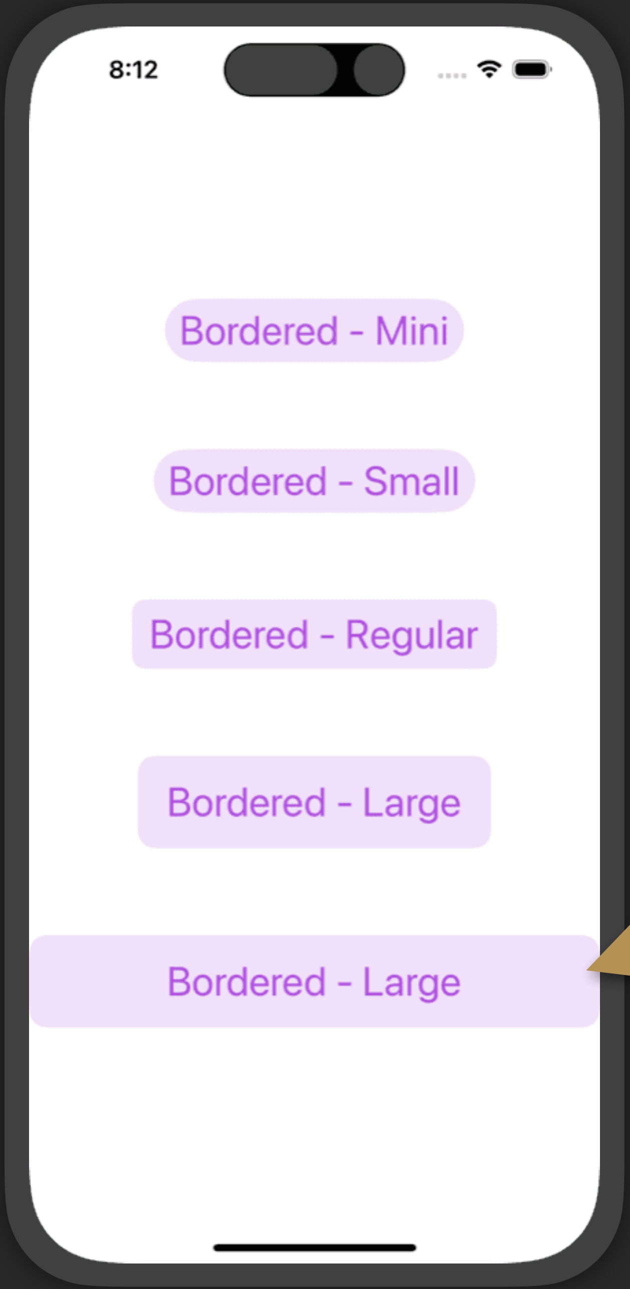
You can apply preset button styles to your buttons with the buttonStyle modifier.

Note: The text here becomes the primary color (black for light mode, white for dark mode). You can always override this with the foregroundStyle modifier.

Color does not get applied to plain buttons.

ControlSize

iOS 15



```
struct Button_ControlSize: View {
    var body: some View {
        VStack(spacing: 60.0) {
            Button("Bordered - Mini") { }
                .controlSize(.mini)

            Button("Bordered - Small") { }
                .controlSize(.small)

            Button("Bordered - Regular") { }
                .controlSize(.regular)

            Button("Bordered - Large") { }
                .controlSize(.large)

            Button(action: {}) {
                Text("Bordered - Large")
                    .frame(maxWidth: .infinity)
            }
            .controlSize(.large)
        }
        .buttonStyle(.bordered)
        .tint(.purple)
        .font(.title)
    }
}
```

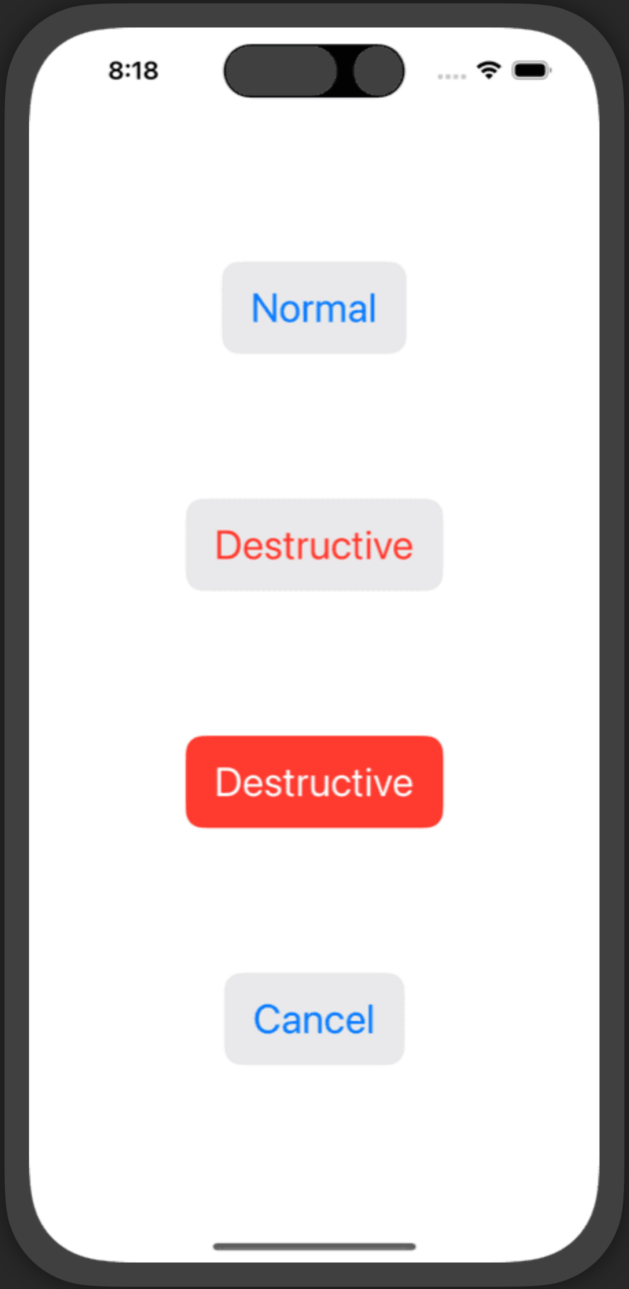
Use controlSize to change the amount of **padding** around the content of the button.

You can still change the size manually and the shape will be the same.

Notice how the buttonStyle is on the parent view and gets applied to all the child views that are buttons.

Role

iOS 15

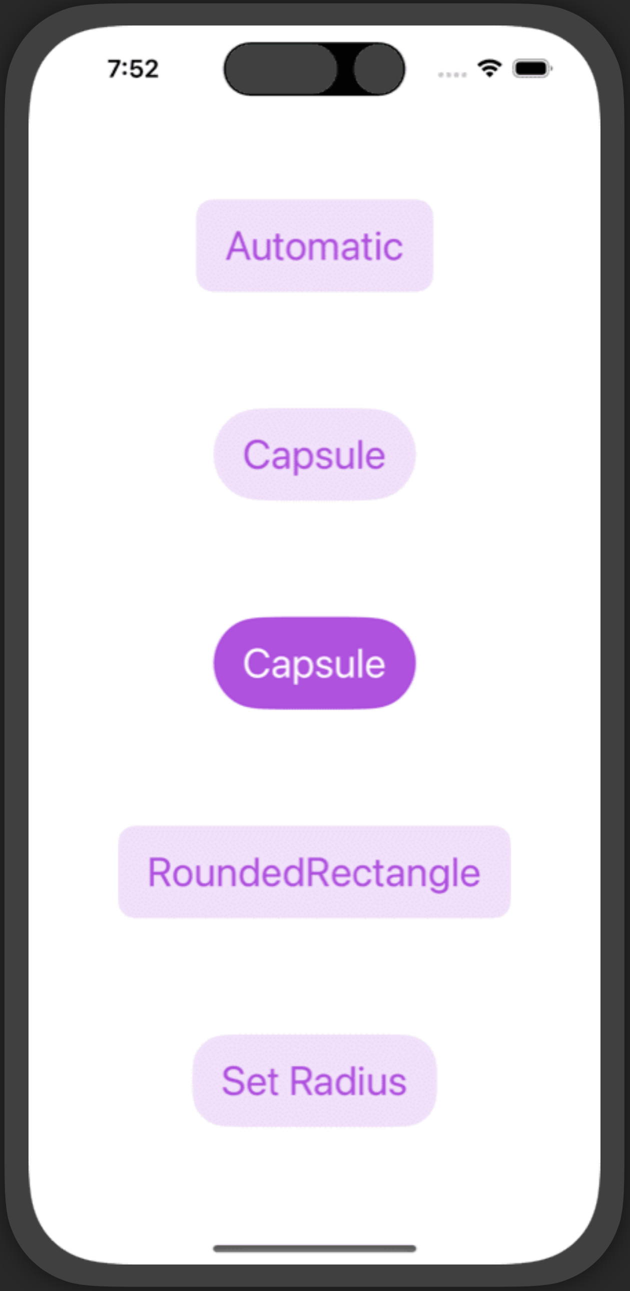


```
struct Button_Role: View {
    var body: some View {
        VStack(spacing: 100.0) {
            Button("Normal") { }

            Button("Destructive", role: .destructive) { }

            Button("Destructive", role: .destructive) { }
                .buttonStyle(.borderedProminent)

            Button("Cancel", role: .cancel) { }
        }
        .buttonStyle(.bordered)
        .controlSize(.large)
        .font(.title)
    }
}
```

Use the role parameter to specify the kind of button you have.

iOS 15

ButtonBorderShape



```swift
struct Button_ButtonBorderShape: View {
    var body: some View {
        VStack(spacing: 80.0) {
            Button("Automatic") { }
                .buttonBorderShape(.automatic)

            Button("Capsule") { }
                .buttonBorderShape(.capsule)

            Button("Capsule") { }
                .buttonBorderShape(.capsule)
                .buttonStyle(.borderedProminent)

            Button("RoundedRectangle") { }
                .buttonBorderShape(.roundedRectangle)

            Button("Set Radius") { }
                .buttonBorderShape(.roundedRectangle(radius: 24))
        }
        .buttonStyle(.bordered)
        .controlSize(.large)
        .font(.title)
        .tint(.purple)
    }
}
```

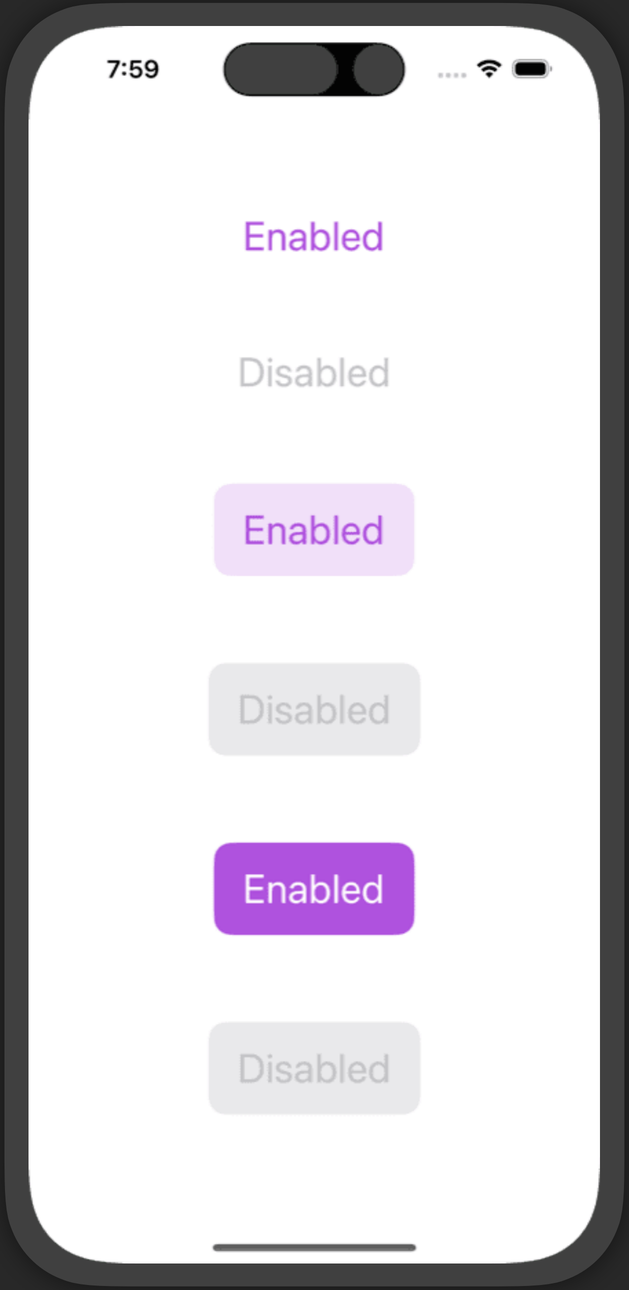
Set a button's shape to capsule or rounded rectangle on **bordered** and **bordered prominent** buttons.

Bordered prominent buttons already create a rounded rectangle but you can override this and show a capsule shape.

You can use the radius option to set your own a custom radius.

Note: This modifier **ONLY** works on buttons that are bordered or borderedProminent.

Disabled

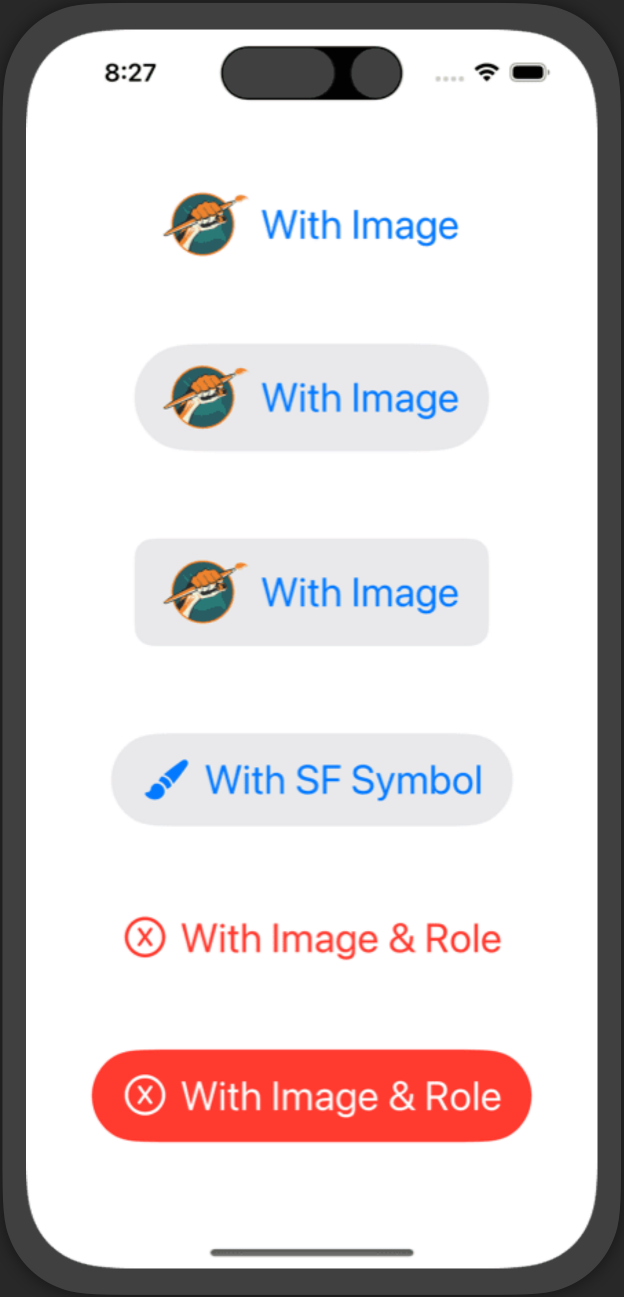


```
struct Button_Disabled: View {
    var body: some View {
        VStack(spacing: 60) {
            Button("Enabled") { }

            Button("Disabled") { }
                .disabled(true)

            Button("Enabled") { }
                .buttonStyle(.bordered)

            Button("Disabled") { }
                .buttonStyle(.bordered)
                .disabled(true)

            Button("Enabled") { }
                .buttonStyle(.borderedProminent)

            Button("Disabled") { }
                .buttonStyle(.borderedProminent)
                .disabled(true)
        }
        .controlSize(.large)
        .font(.title)
        .tint(.purple)
    }
}
```

Use the disabled modifier to prevent the user from interacting with buttons.

Initialize with Image

iOS 17



```
struct Button_InitWithImage: View {
    var body: some View {
        VStack(spacing: 60.0) {
            Button("With Image", image: .bookLogo) { }

            Button("With Image", image: .bookLogo) { }
                .buttonStyle(.bordered)

            Button("With Image", image: .bookLogo) { }
                .buttonBorderShape(.roundedRectangle)
                .buttonStyle(.bordered)

            Button("With SF Symbol", systemImage: "paintbrush.pointed.fill") { }
                .buttonStyle(.bordered)

            Button("With Image & Role", systemImage: "x.circle", role: .destructive) { }

            Button("With Image & Role", systemImage: "x.circle", role: .destructive) { }
                .buttonStyle(.borderedProminent)
        }
        .controlSize(.extraLarge)
        .font(.title)
    }
}
```

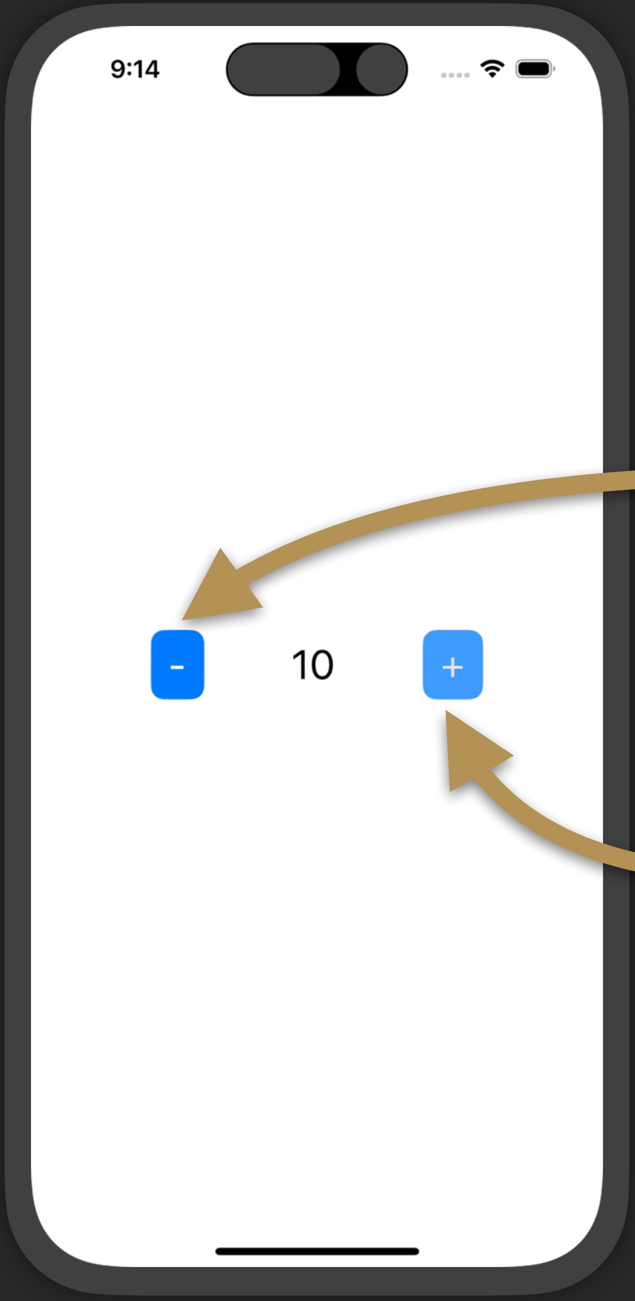
Buttons can be created by specifying an image resource or SF Symbol.

Adding a buttonStyle uses a capsule shape instead of the rounded rectangle by default. You can override this with the buttonBorderShape modifier.

You can also specify the role with this initializer.

Repeat Behavior

iOS 17



```
struct Button_RepeatBehavior: View {
    @State private var buttonPressedCount = 0

    var body: some View {
        HStack(spacing: 60.0) {
            Button("-") {
                buttonPressedCount -= 1
            }
            .buttonRepeatBehavior(.enabled)

            Text(buttonPressedCount, format: .number)

            Button("+") {
                buttonPressedCount += 1
            }
            .buttonRepeatBehavior(.enabled)
        }
        .buttonStyle(.borderedProminent)
        .font(.title)
    }
}
```

When the buttonRepeatBehavior is enabled, users can long-press a button to have its code repeated continuously.

You will notice the repeating happens slowly at first and then speeds up.

iOS 16

Chart

This SwiftUI content is locked in this preview.

A Chart is a container view. It c "marks" (representations of values) which can be drawn in a variety of different ways. There are many different ways you can configure the chart and marks. This chapter will cover many of those options.

This is a push-out view.

UNLOCK THE BOOK TODAY FOR ONLY $55!

iOS 14

ColorPicker

The ColorPicker control al[illegible]o give users the abilit[illegible]t a color. This could be useful if you want to allow users to set the color of visual ele[illegible]s on the user interface.

This is a push-out horizo

iOS 14

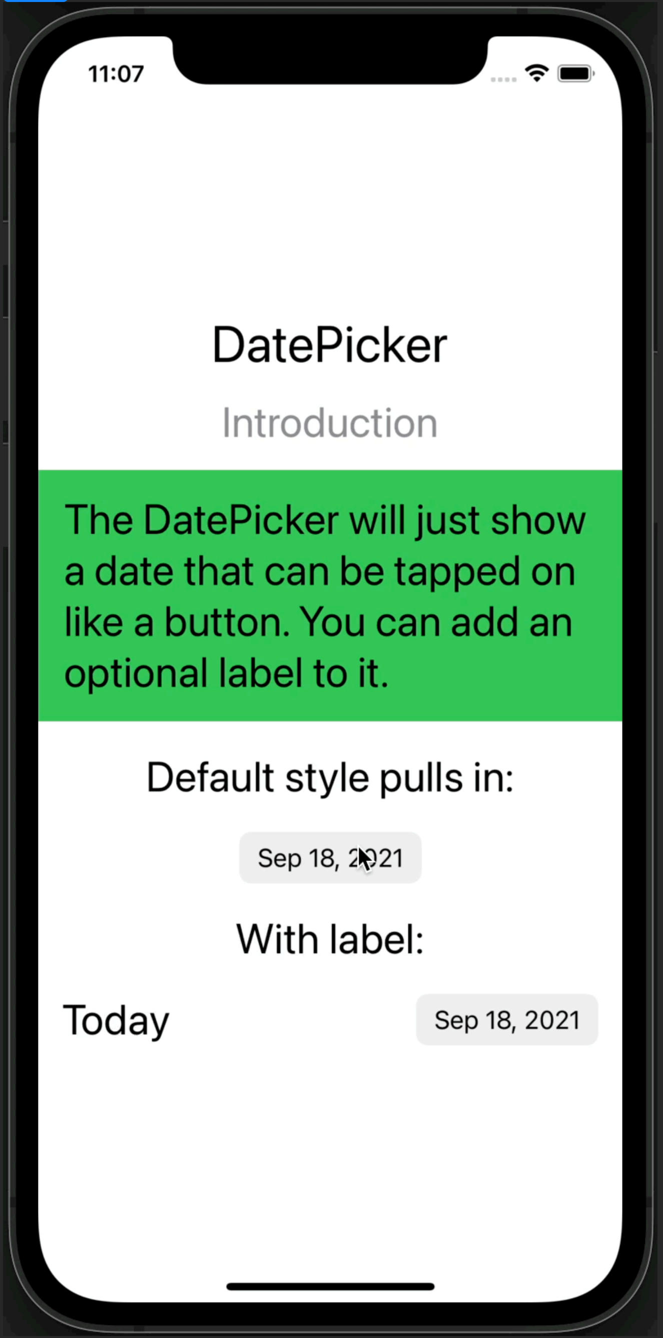
DatePicker

The date picker provides a way for the user to select a date and time. You bind the selected date to a property. You can read this property to find out what was selected or set this property for the DatePicker to show the date you want.
(Note: If you have to support the DatePicker for iOS 13, then it will look different from what you see in this chapter.)

This is a push-out view.

Introduction

iOS 14


11:07

DatePicker

Introduction

The DatePicker will just show a date that can be tapped on like a button. You can add an optional label to it.

Default style pulls in:

Sep 18, 2021

With label:

Today Sep 18, 2021

```
struct DatePicker_Intro: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("DatePicker",
                       subtitle: "Introduction",
                       desc: "The DatePicker will just show a date that can be tapped on like a
                             button. You can add an optional label to it.", back: .green)

            Text("Default style pulls in:")

            DatePicker("Today", selection: $date, displayedComponents: .date)
                .labelsHidden()
                .padding(.horizontal)

            Text("With label:")

            DatePicker("Today", selection: $date, displayedComponents: .date)
                .padding(.horizontal)

        }.font(.title)
    }
}
```

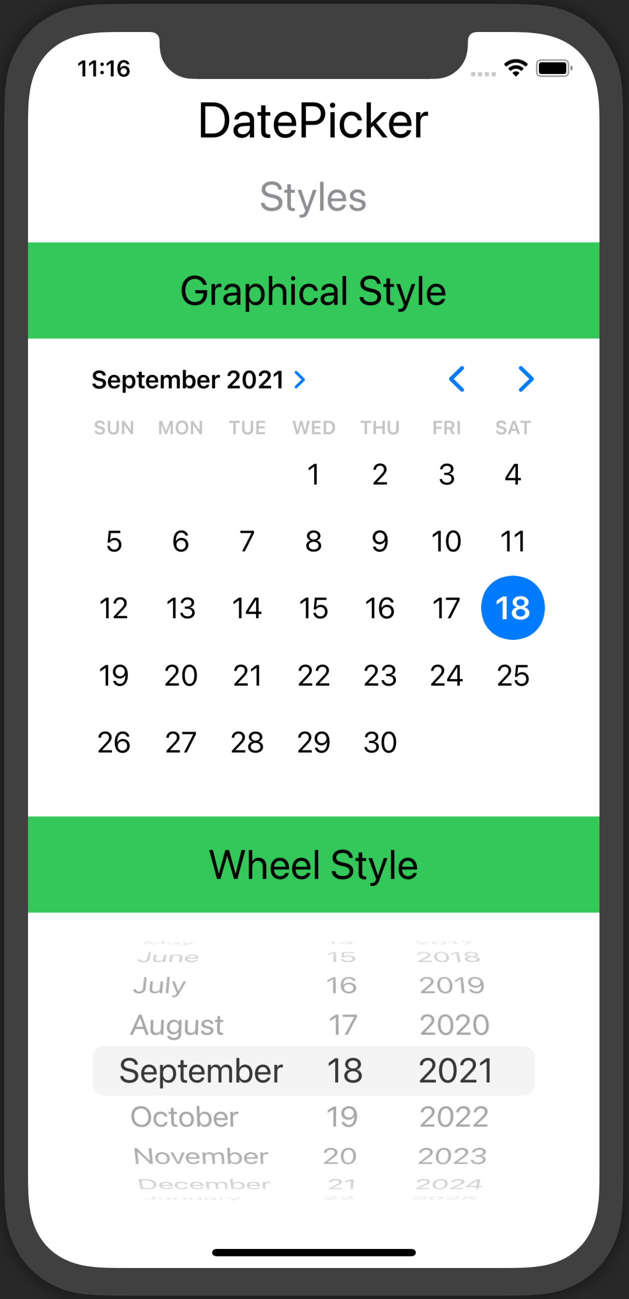
Hiding the label makes this view pull in.

What you see here is representative of the **compact** date picker style (text representation of the date).

There are other styles available...

Styles

iOS 14



```swift
struct DatePicker_Styles: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 0) {
            HeaderView("DatePicker",
                       subtitle: "Styles",
                       desc: "Graphical Style", back: .green)

            DatePicker("Birthday", selection: $date, displayedComponents: .date)
                .datePickerStyle(.graphical)
                .frame(width: 320)

            DescView(desc: "Wheel Style", back: .green)
            DatePicker("Birthday", selection: $date, displayedComponents: .date)
                .datePickerStyle(.wheel)
                .labelsHidden()

        }
        .font(.title)
        .ignoresSafeArea(edges: .bottom)
    }
}
```

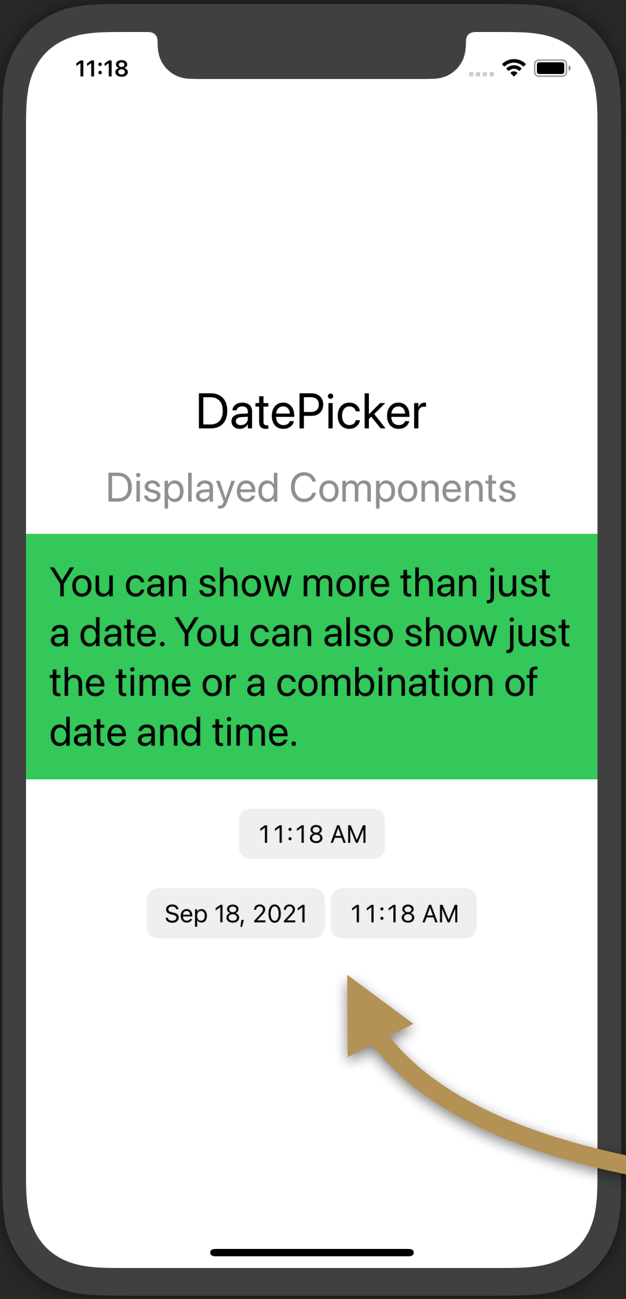
Notice we didn't have to hide the labels on the graphical style. It's not shown. (But you should keep it set for accessibility purposes.)

For datePickerStyle, use:

< iOS 15	`GraphicalDatePickerStyle()` `WheelDatePickerStyle()`
iOS 15+	`.graphical` `.wheel`

Displayed Components

iOS 14


11:18

DatePicker

Displayed Components

You can show more than just a date. You can also show just the time or a combination of date and time.

11:18 AM

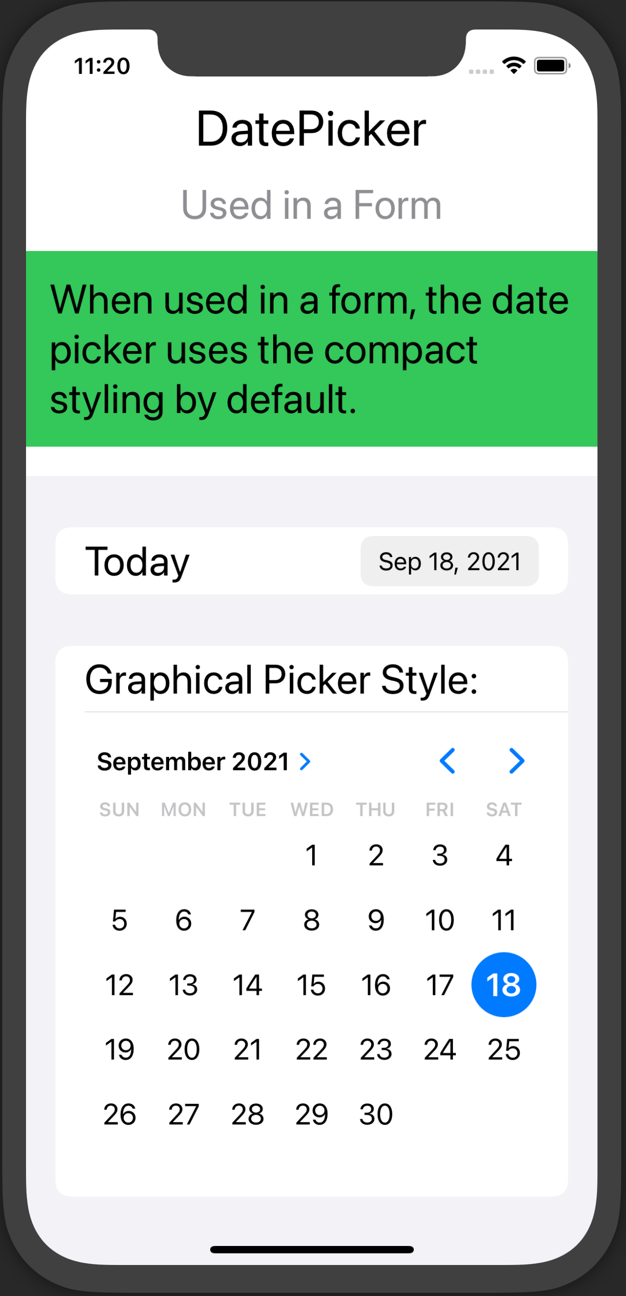
Sep 18, 2021 11:18 AM

```
struct DatePicker14_DisplayedComponents: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("DatePicker - iOS 14+",
                       subtitle: "Displayed Components",
                       desc: "You can show more than just a date. You can also show just the
                             time or a combination of date and time.", back: .green)

            DatePicker("Today", selection: $date, displayedComponents: .hourAndMinute)
                .labelsHidden()
                .padding(.horizontal)

            DatePicker("Today", selection: $date, displayedComponents: [.hourAndMinute, .date])
                .labelsHidden()
                .padding(.horizontal)
                .buttonStyle(.bordered)
        }
        .font(.title)
    }
}
```

Note: The order of the displayed components does not affect the displayed order. The hour and minute still come second.

Displayed in Form

iOS 14


11:20

DatePicker

Used in a Form

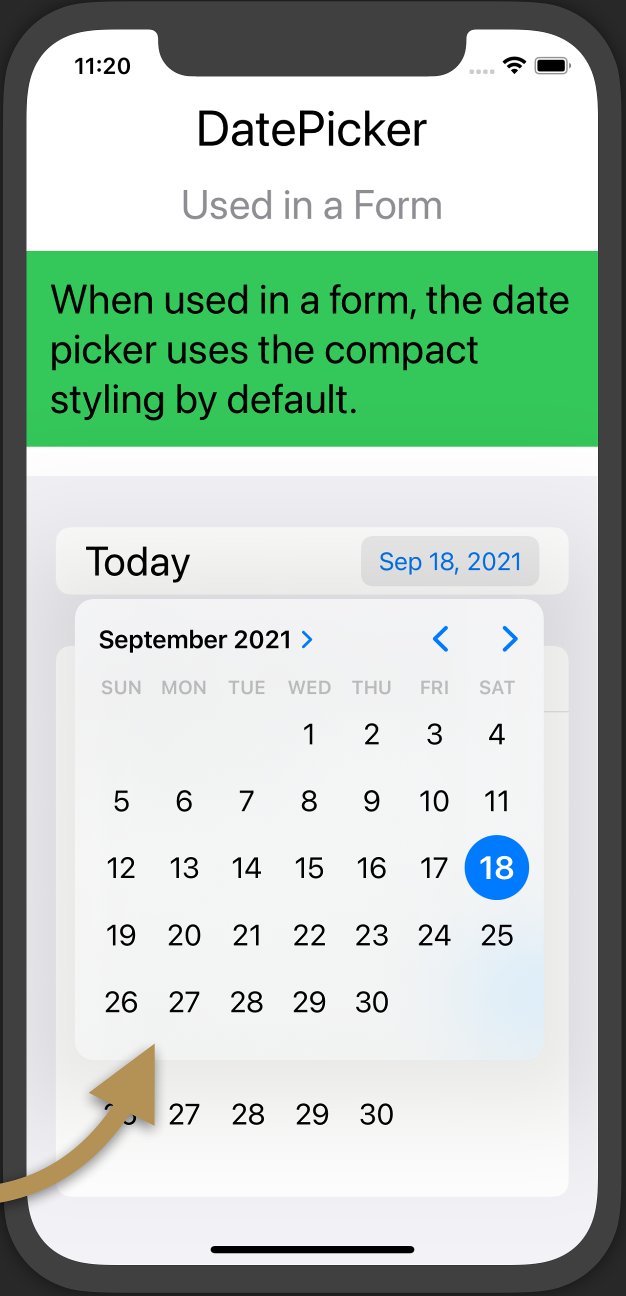
When used in a form, the date picker uses the compact styling by default.

Today Sep 18, 2021

Graphical Picker Style:

```
struct DatePicker_InForm: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("DatePicker",
                       subtitle: "Used in a Form",
                       desc: "When used in a form, the date
picker uses the compact styling by default.",
                       back: .green)

            Form {
                DatePicker("Today", selection: $date,
                           displayedComponents: .date)

                Section {
                    Text("Graphical Picker Style:")
                    DatePicker("Birthday", selection: $date,
                               displayedComponents: .date)
                        .datePickerStyle(.graphical)
                }
            }
        }
        .font(.title)
    }
}
```

When the compact style is tapped, a pop up shows the graphical date picker.


11:20

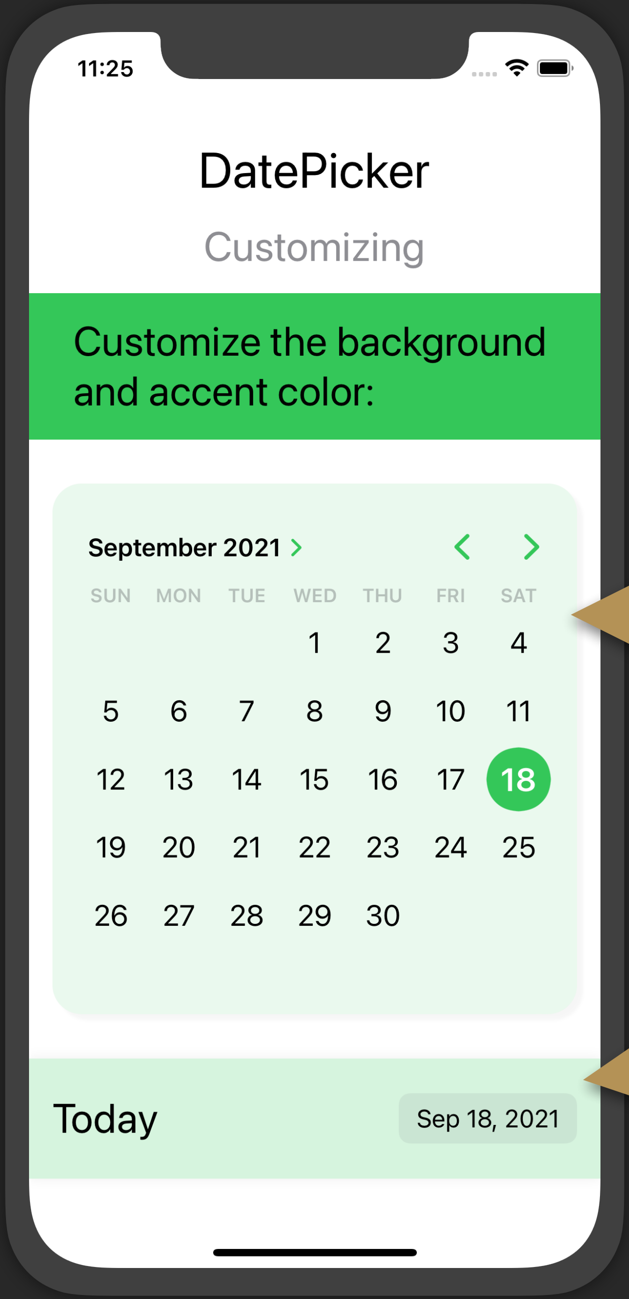
DatePicker

Used in a Form

When used in a form, the date picker uses the compact styling by default.

Today Sep 18, 2021

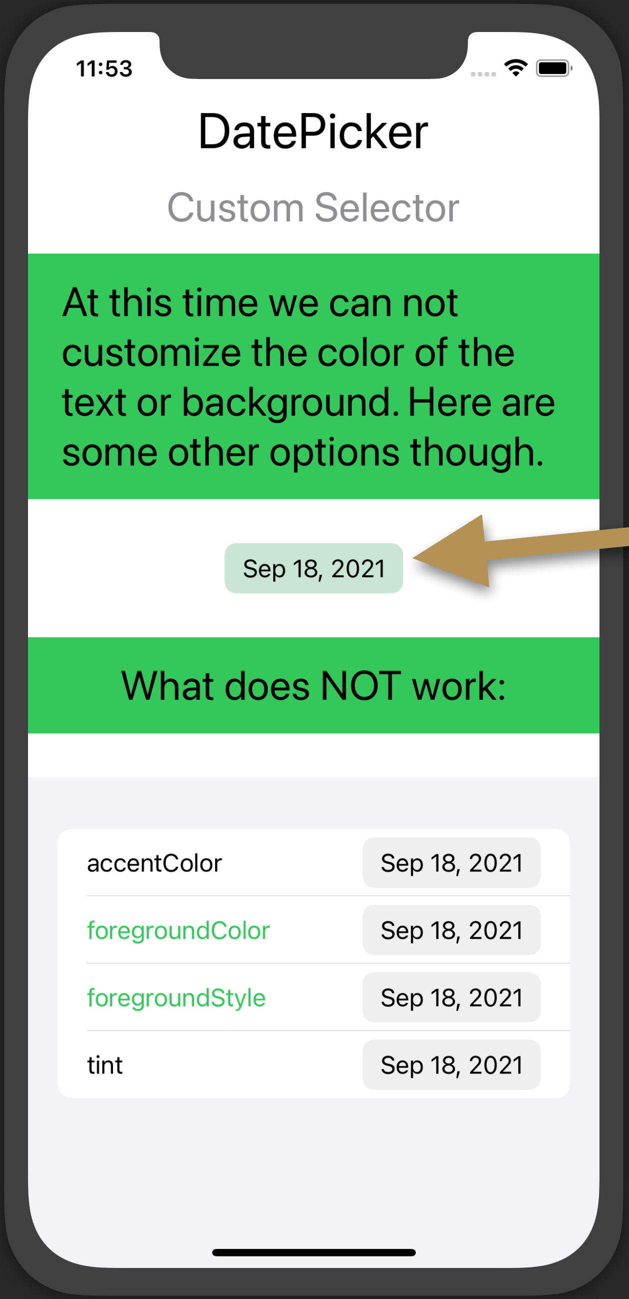
Customizing



```
struct DatePicker_Customizing: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 30) {
            HeaderView("DatePicker",
                       subtitle: "Customizing",
                       desc: "Customize the background and accent color:", back: .green)

            DatePicker("Birthday", selection: $date, displayedComponents: .date)
                .datePickerStyle(.graphical)
                .accentColor(.green)
                .padding()
                .background(RoundedRectangle(cornerRadius: 20)
                                .fill(Color.green)
                                .opacity(0.1)
                                .shadow(radius: 1, x: 4, y: 4))
                .padding(.horizontal)

            DatePicker("Today", selection: $date, displayedComponents: .date)
                .frame(height: 50)
                .padding()
                .background(Rectangle()
                                .fill(Color.green)
                                .shadow(radius: 4)
                                .opacity(0.2))
        }.font(.title)
    }
}
```

Custom Selector

iOS 14



DatePicker

Custom Selector

At this time we can not customize the color of the text or background. Here are some other options though.

Sep 18, 2021

What does NOT work:

accentColor	Sep 18, 2021
foregroundColor	Sep 18, 2021
foregroundStyle	Sep 18, 2021
tint	Sep 18, 2021

```
struct DatePicker_CustomSelector: View {
    @State private var date = Date()

    var body: some View {
        VStack(spacing: 30) {
            HeaderView("DatePicker",
                       subtitle: "Custom Selector",
                       desc: "At this time we can not customize the color of the text or
                              background. Here are some other options though.", back: .green)

            DatePicker("Today", selection: $date, displayedComponents: .date)
                .labelsHidden()
                .background(RoundedRectangle(cornerRadius: 8, style: .continuous)
                                .fill(Color.green).opacity(0.2))

            DescView(desc: "What does NOT work:", back: .green)
            Form {
                DatePicker("accentColor", selection: $date, displayedComponents: .date)
                    .accentColor(.green)

                DatePicker("foregroundColor", selection: $date, displayedComponents: .date)
                    .foregroundColor(.green)

                DatePicker("foregroundStyle", selection: $date, displayedComponents: .date)
                    .foregroundStyle(.green, .green, .green)

                DatePicker("tint", selection: $date, displayedComponents: .date)
                    .tint(.green)
            }
            .font(.body)
        }
        .font(.title)
    }
}
```

Using a cornerRadius of 8 and a continuous corner style is the best I could get to match the existing gray background.

iOS 14

DisclosureGroup

The disclosure group gives yo[illegible]o expand and collapse vie[illegible]it. You supply it with child views that you want collapsed/expanded.

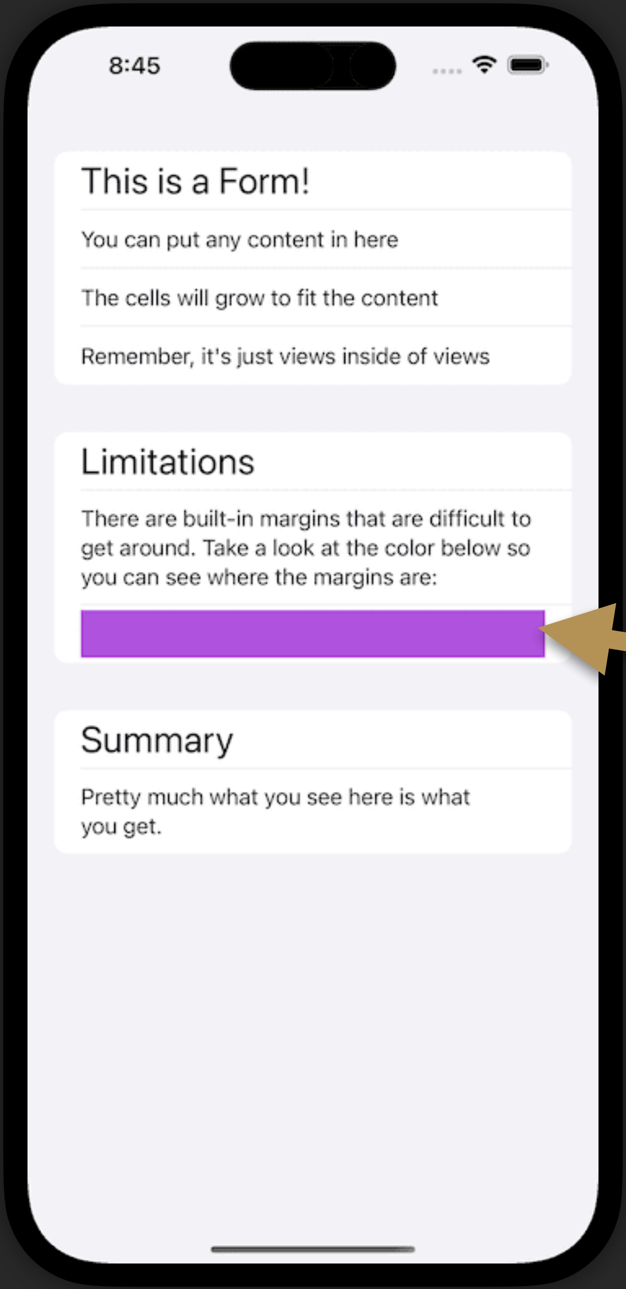
This is a pull-in view.

Form

The Form view is a great choice when you want to show settings, options, or get some user input. It is easy to set up and customize as you will see on the following pages.

This is a push-out view.

Introduction


8:45

This is a Form!

You can put any content in here

The cells will grow to fit the content

Remember, it's just views inside of views

Limitations

There are built-in margins that are difficult to get around. Take a look at the color below so you can see where the margins are:

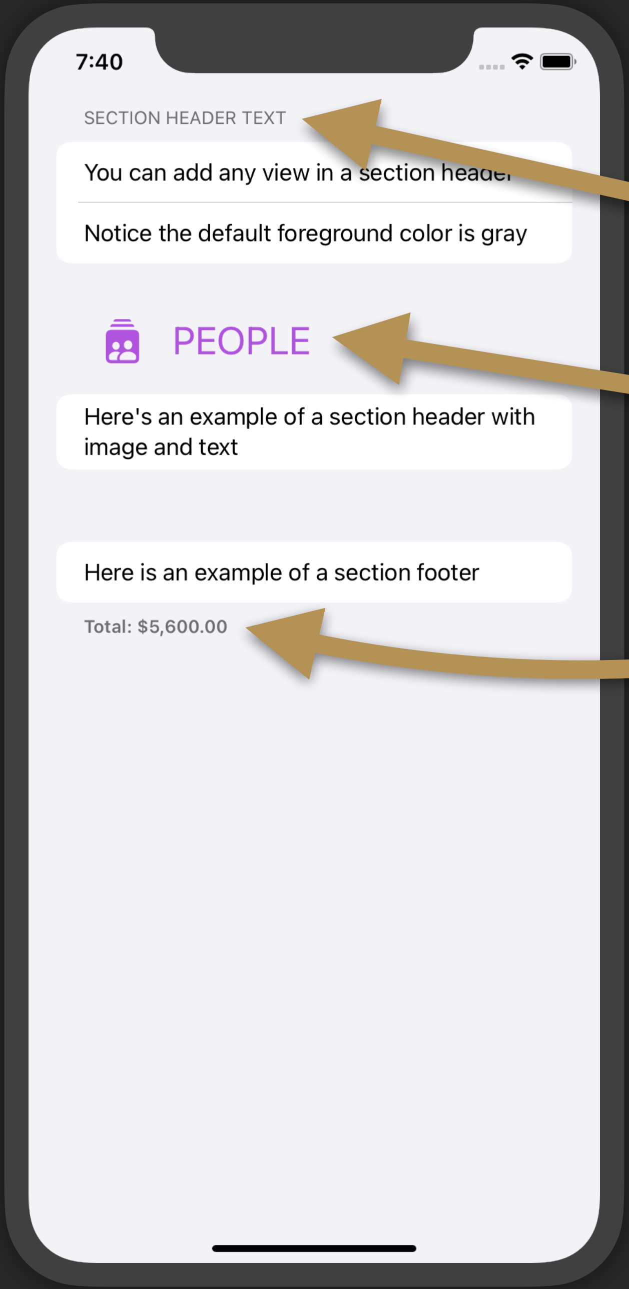
Summary

Pretty much what you see here is what you get.

```
struct Form_Intro : View {
    var body: some View {
        Form {
            Section {
                Text("This is a Form!")
                    .font(.title)
                Text("You can put any content in here")
                Text("The cells will grow to fit the content")
                Text("Remember, it's just views inside of views")
            }

            Section {
                Text("Limitations")
                    .font(.title)
                Text("There are built-in margins that are difficult to get around. Take a look
                    at the color below so you can see where the margins are:")
                Color.purple
            }

            Section {
                Text("Summary")
                    .font(.title)
                Text("Pretty much what you see here is what you get.")
            }
        }
    }
}
```

Forms come with a built-in scroll view if the contents exceed the height of the screen.

Section Headers and Footers


7:40

SECTION HEADER TEXT

You can add any view in a section header

Notice the default foreground color is gray

PEOPLE

Here's an example of a section header with image and text

Here is an example of a section footer

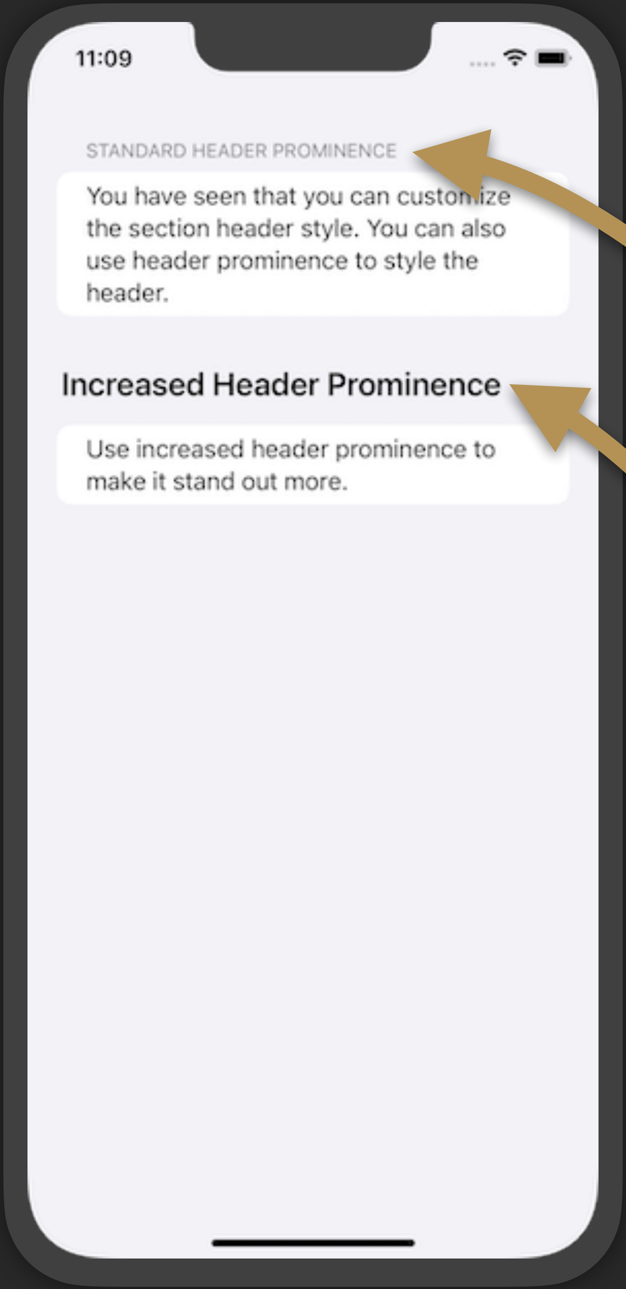
Total: $5,600.00

```swift
struct Form_HeadersAndFooters : View {
    var body: some View {
        Form {
            Section {
                Text("You can add any view in a section header")
                Text("Notice the default foreground color is gray")
            } header: {
                Text("Section Header Text")
            }
           Section {
                Text("Here's an example of a section header with image and text")
            } header: {
                SectionTextAndImage(name: "People", image: "person.2.square.stack.fill")
            }
           Section {
                Text("Here is an example of a section footer")
            } footer: {
                Text("Total: $5,600.00").bold()
            }
        }
    }
}
```

```swift
struct SectionTextAndImage: View {
    var name: String
    var image: String
    var body: some View {
        HStack {
            Image(systemName: image).padding(.trailing)
            Text(name)
        }
        .padding()
        .font(.title)
        .foregroundStyle(Color.purple)
    }
}
```

Header Prominence

iOS 15


11:09

STANDARD HEADER PROMINENCE

You have seen that you can customize the section header style. You can also use header prominence to style the header.

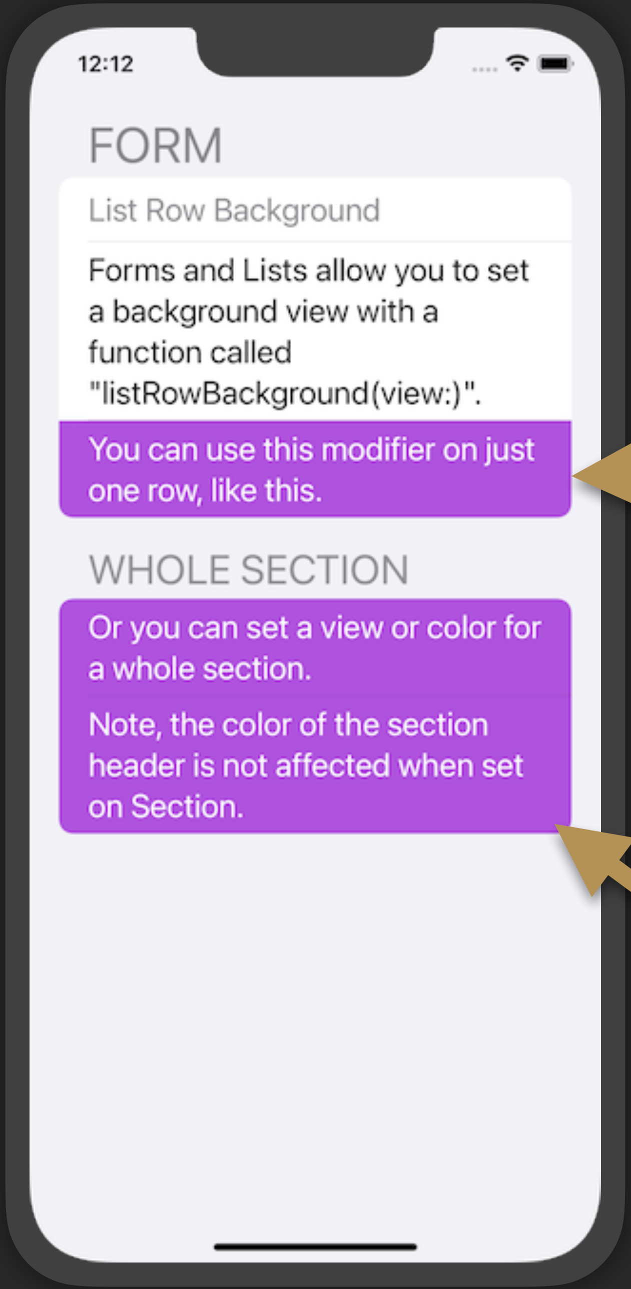
Increased Header Prominence

Use increased header prominence to make it stand out more.

```
struct Form_HeaderProminence: View {
    var body: some View {
        Form {
            Section {
                Text("You have seen that you can customize the section header style. You can
                     also use header prominence to style the header.")
            } header: {
                Text("Standard Header Prominence")
            }
            .headerProminence(.standard)

            Section {
                Text("Use increased header prominence to make it stand out more.")
            } header: {
                Text("Increased Header Prominence")
            }
            .headerProminence(.increased)
        }
    }
}
```

Note: I have found that I can put this modifier on the Section or the Text inside the header closure for it to work.

List Row Background


12:12

FORM

List Row Background

Forms and Lists allow you to set a background view with a function called "listRowBackground(view:)".

You can use this modifier on just one row, like this.

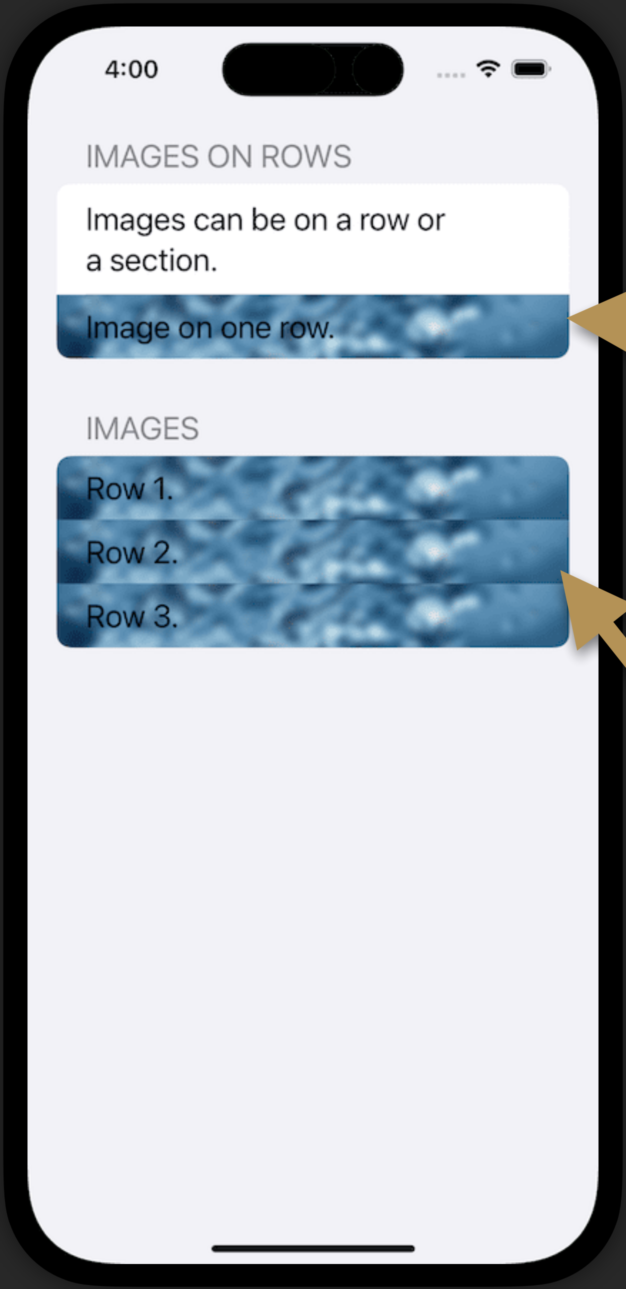
WHOLE SECTION

Or you can set a view or color for a whole section.

Note, the color of the section header is not affected when set on Section.

```
struct Form_ListRowBackground : View {
    var body: some View {
        Form {
            Section {
                Text("List Row Background")
                    .foregroundStyle(.gray)

                Text("Forms and Lists allow you to set a background view with a function called
\"listRowBackground(view:)\".")

                Text("You can use this modifier on just one row, like this.")
                    .listRowBackground(Color.purple)
                    .foregroundStyle(.white)
            } header: {
                Text("Form").font(.largeTitle)
            }

            Section {
                Text("Or you can set a view or color for a whole section.")

                Text("Note, the color of the section header is not affected when set on
Section.")
                    .fixedSize(horizontal: false, vertical: true)
            } header: {
                Text("Whole Section")
                    .font(.title).foregroundStyle(.gray)
            }
            .foregroundStyle(.white)
            .listRowBackground(Color.purple)
        }
        .font(.title2)
    }
}
```

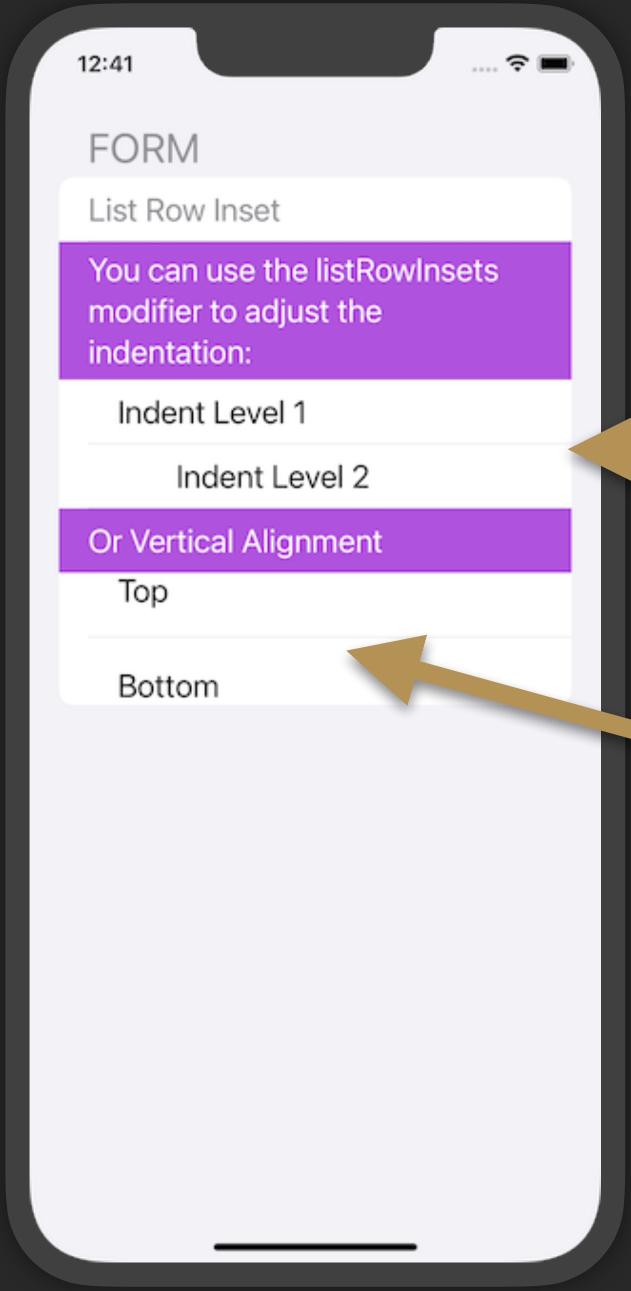
Background Images



```
struct Form_RowBackgroundImage : View {
    var body: some View {
        Form {
            Section {
                Text("Images can be on a row or a section.")
                Text("Image on one row.")
                    .listRowBackground(Image("water")
                        .blur(radius: 3))
            } header: {
                Text("Images on Rows")
            }

            Section {
                Text("Row 1.")
                Text("Row 2.")
                Text("Row 3.")
            } header: {
                Text("Images")
            }
            .listRowBackground(Image("water")
                .blur(radius: 3))
        }
        .font(.title2)
    }
}
```

Notice when the image is on the section, it is repeated for all rows.

List Row Inset


12:41

FORM

List Row Inset

You can use the listRowInsets modifier to adjust the indentation:

Indent Level 1

Indent Level 2

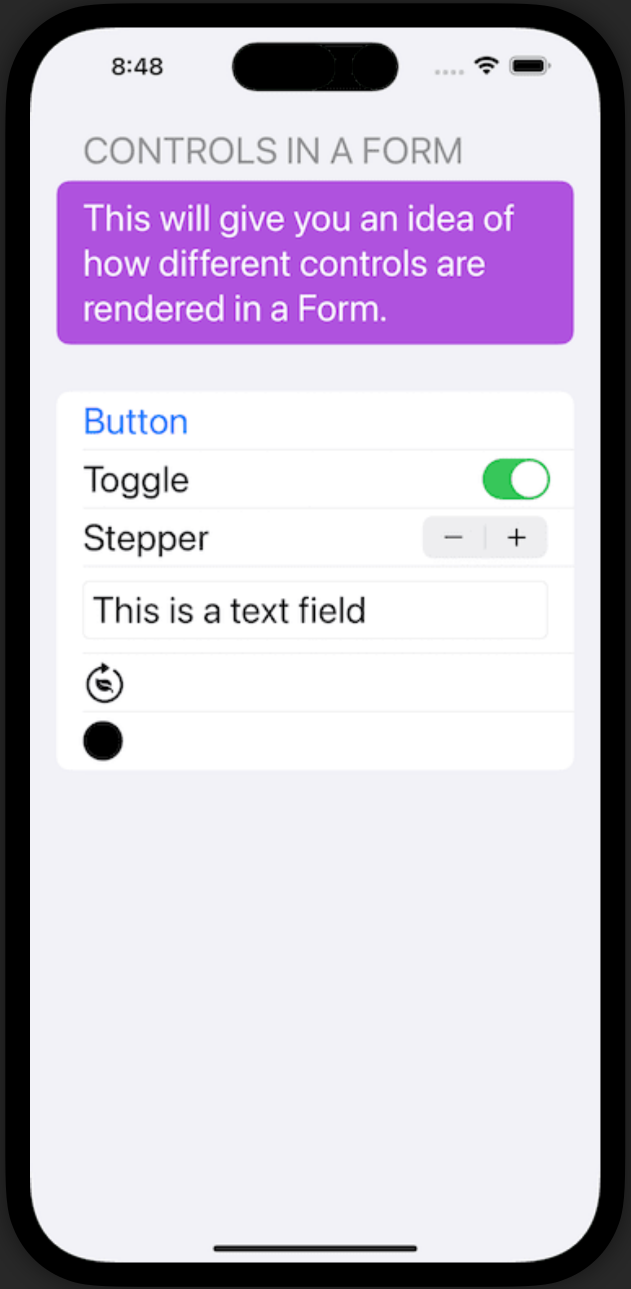
Or Vertical Alignment

Top

Bottom

```
struct Form_ListRowInset : View {
    var body: some View {
        Form {
            Section {
                Text("List Row Inset")
                    .foregroundStyle(.gray)
                Text("You can use the listRowInsets modifier to adjust the indentation:")
                    .foregroundStyle(.white)
                    .listRowBackground(Color.purple)

                Text("Indent Level 1")
                    .listRowInsets(EdgeInsets(top: 0, leading: 40, bottom: 0, trailing: 0))

                Text("Indent Level 2")
                    .listRowInsets(EdgeInsets(top: 0, leading: 80, bottom: 0, trailing: 0))
                Text("Or Vertical Alignment")
                    .foregroundStyle(.white)
                    .listRowBackground(Color.purple)

                Text("Top")
                    .listRowInsets(EdgeInsets(top: -20, leading: 40, bottom: 0, trailing: 0))

                Text("Bottom")
                    .listRowInsets(EdgeInsets(top: 20, leading: 40, bottom: 0, trailing: 0))
            } header: {
                Text("Form")
                    .font(.title)
                    .foregroundStyle(.gray)
            }
        }
        .font(.title2)
    }
}
```

With Controls


8:48

CONTROLS IN A FORM

This will give you an idea of how different controls are rendered in a Form.

Button

Toggle

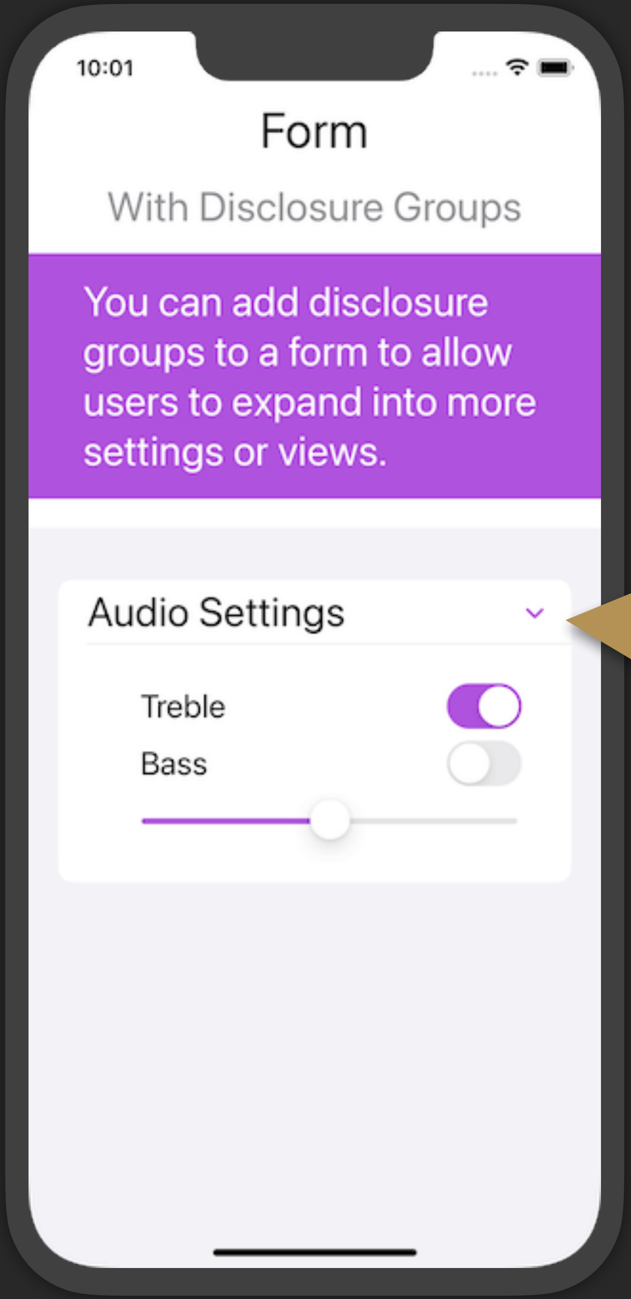
Stepper

This is a text field

```
struct Form_WithControls : View {
    @State private var isOn = true
    @State private var textFieldData = "This is a text field"

    var body: some View {
        Form {
            Section {
                Text("This will give you an idea of how different controls are rendered in a
                     Form.")
                    .foregroundStyle(.white)
                    .listRowBackground(Color.purple)
            } header: {
                Text("Controls in a form")
                    .font(.title)
                    .foregroundStyle(Color.gray)
            }
            Section {
                Button(action: {}) { Text("Button") }
                Toggle(isOn: $isOn) { Text("Toggle") }
                Stepper(onIncrement: {}, onDecrement: {}) { Text("Stepper") }
                TextField("", text: $textFieldData)
                    .textFieldStyle(.roundedBorder)
                Image(systemName: "leaf.arrow.circlepath").font(.title)
                Circle()
                    .frame(height: 30)
            }
        }
        .font(.title)
    }
}
```

With the Disclosure Group

iOS 14



Form

With Disclosure Groups

You can add disclosure groups to a form to allow users to expand into more settings or views.

```
struct Form_WithDisclosureGroup: View {
    @State private var settingsExpanded = true
    @State private var trebleOn = true
    @State private var bassOn = false
    @State private var levels = 0.5

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Form",
                       subtitle: "With Disclosure Groups",
                       desc: "You can add disclosure groups to a form to allow users to expand
into more settings or views.",
                       back: .purple, textColor: .white)

            Form {
                DisclosureGroup("Audio Settings", isExpanded: $settingsExpanded) {
                    VStack {
                        Toggle("Treble", isOn: $trebleOn)
                        Toggle("Bass", isOn: $bassOn)
                        Slider(value: $levels)
                    }
                    .font(.title2)
                    .padding()
                }
            }
        }
        .font(.title)
        .tint(.purple)
    }
}
```

Normally the tint would also change the DisclosureGroup's label. But in a Form, it does not.

See **Control Views** > DisclosureGroup for more info.

iOS 16

Gauge

Use the gauge to show a value … range. The value and range can be shown in a linear or circular way.

When the gauge is shown in
When the gauge is shown in

iOS 14

GroupBox

This SwiftUI chapter is locked in this preview.

The group box is container for grouping similar items together.

The GroupBox is a pull-i

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

iOS 14

Label

The label view is a pretty s[illegible]w that will handle the [illegible] spacing and formatting of an image and text that you pass into it.

This is a pull-in view.

iOS 16

LabeledContent

This SwiftUI chapter is locked in this preview.

You may have noticed some co[illegible]ave this pattern of a label [illegible] and the control on the right. Think of this view as a template where you can supply a label on the left and your own content on the right.

This view pushes out horizo[illegible]

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

iOS 14

Link

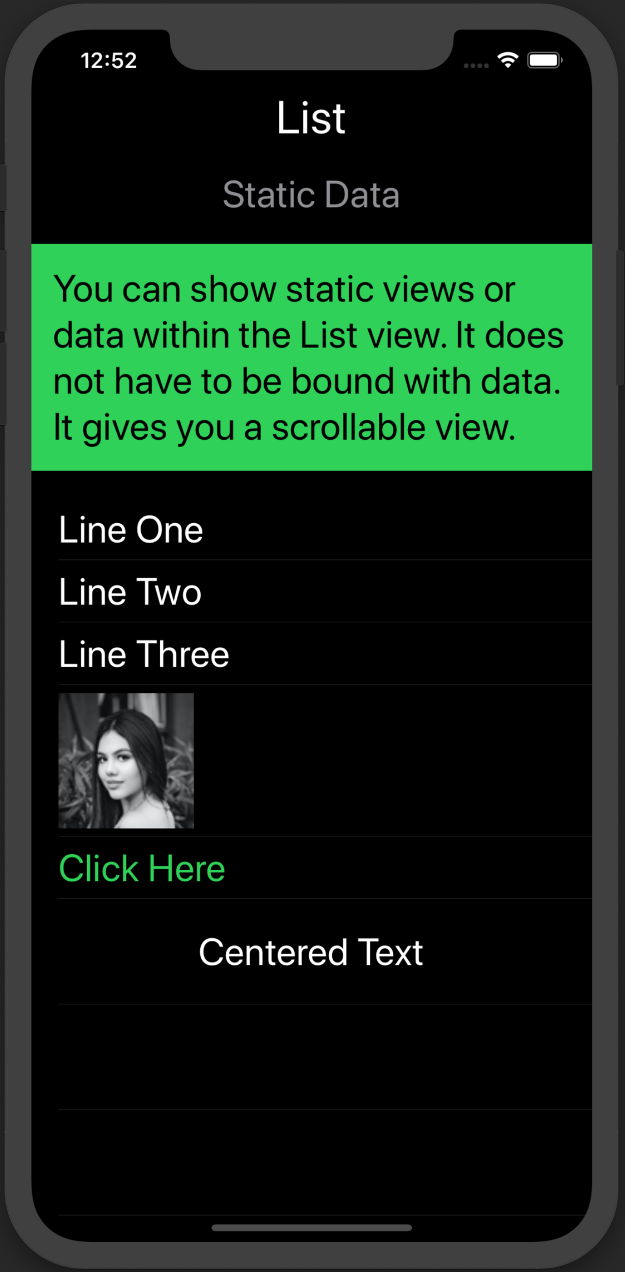
The Link is similar to the Button or the NavigationLink except it can navigate you to a place outside your app.

This is a pull-in view.

List

Using a List view is the most efficient way of displaying vertically scrolling data. You can display data in a ScrollView, as you will see later on, but it will not be as efficient in terms of memory or performance as the List view.

With Static Views


12:52

List

Static Data

You can show static views or data within the List view. It does not have to be bound with data. It gives you a scrollable view.

Line One

Line Two

Line Three

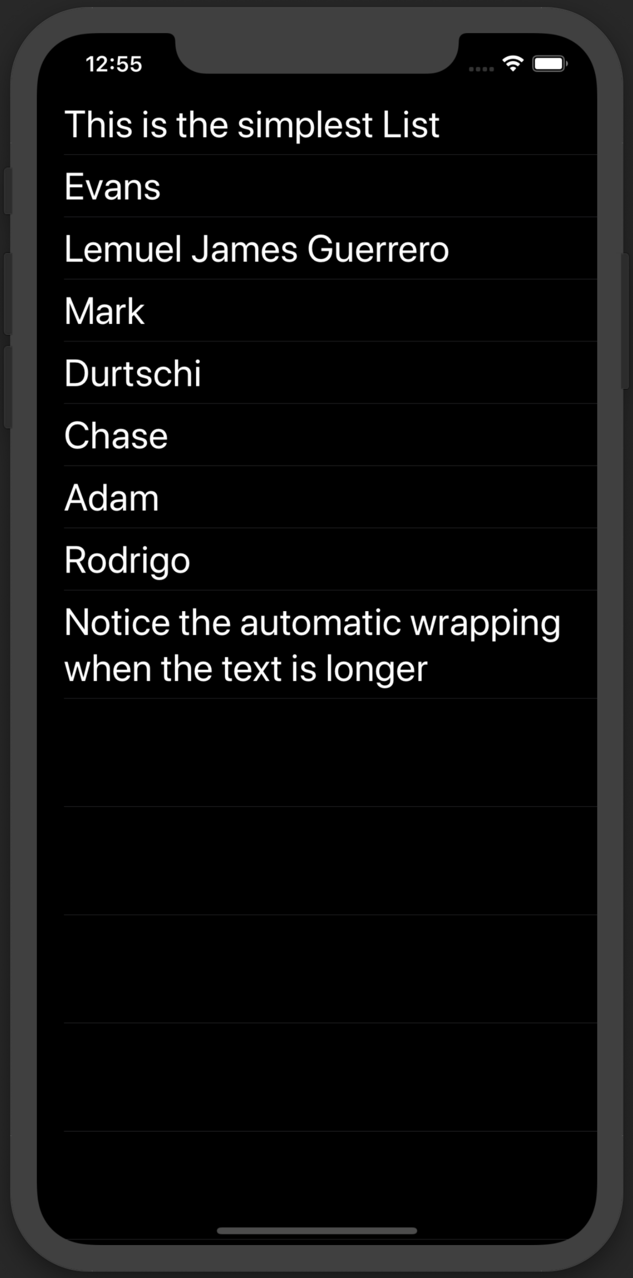
Click Here

Centered Text

```
struct List_WithStaticData: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("List").font(.largeTitle)
            Text("Static Data").font(.title).foregroundColor(.gray)
            Text("You can show static views or data within the List view. It does not have to be
                 bound with data. It gives you a scrollable view.")
                .frame(maxWidth: .infinity)
                .font(.title).padding()
                .background(Color.green)
               .foregroundColor(.black)
            List {
                Text("Line One")
                Text("Line Two")
                Text("Line Three")
                Image("profile")
                Button("Click Here", action: {})
                    .foregroundColor(.green)
                HStack {
                    Spacer()
                    Text("Centered Text")
                    Spacer()
                }.padding()
            }
            .font(.title)
        }
    }
}
```

Note: Like other container views, you can now have as many views as you want inside it.

With Data



This is the simplest List
Evans
Lemuel James Guerrero
Mark
Durtschi
Chase
Adam
Rodrigo
Notice the automatic wrapping when the text is longer

```
struct List_WithData : View {
    var stringArray = ["This is the simplest List", "Evans", "Lemuel James Guerrero", "Mark",
"Durtschi", "Chase", "Adam", "Rodrigo", "Notice the automatic wrapping when the text is longer"]

    var body: some View {
        List(stringArray, id: \.self) { string in
            Text(string)
        }
        .font(.largeTitle) // Apply this font style to all items in the list
    }
}
```

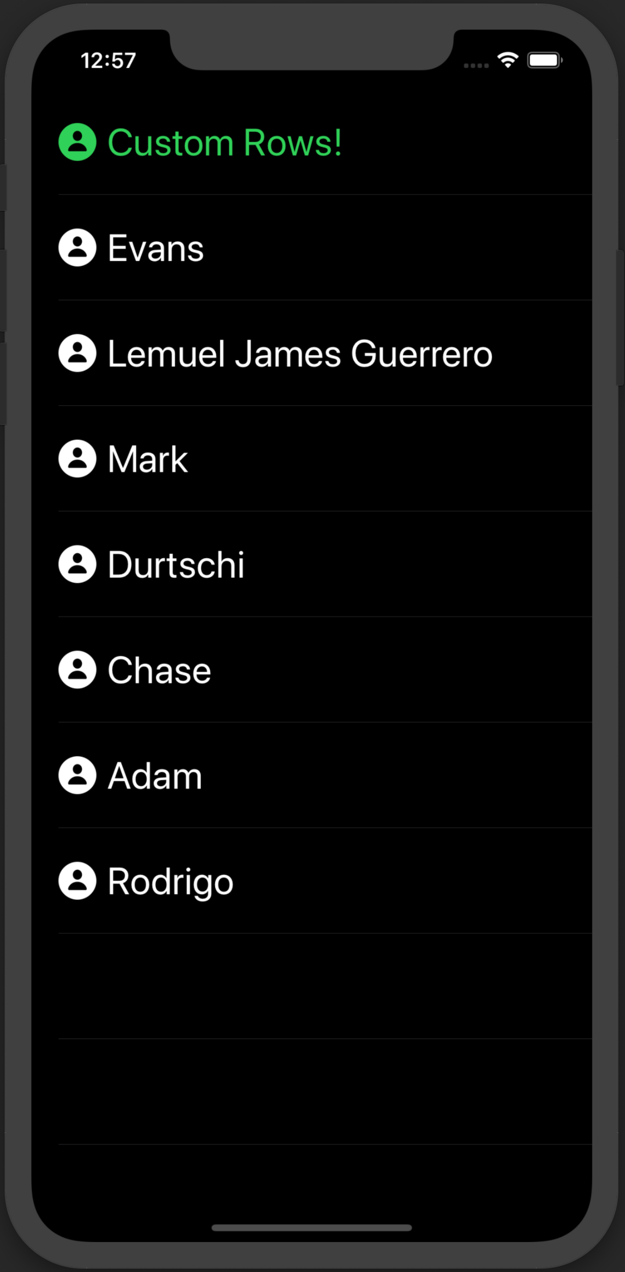
The List view can iterate through an array of data and pass in one item at a time to its closure.

What is that .id parameter?

You use this parameter to tell the List how it can uniquely identify each row by which value. The List needs to know this so it can compare rows by this value to perform different operations like reordering and deleting rows for us.

In this scenario, we are using “self” to say, “Just use the value of the string itself to uniquely identify each row.”

Custom Rows


12:57

Custom Rows!

Evans

Lemuel James Guerrero

Mark

Durtschi

Chase

Adam

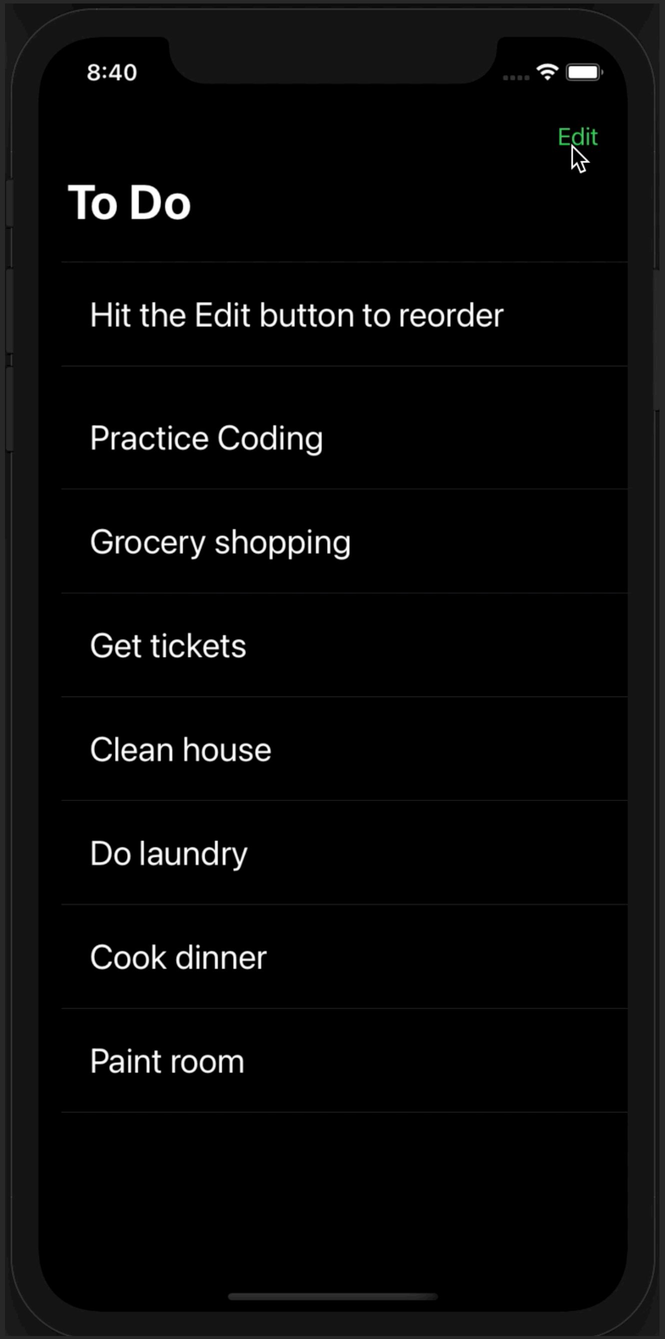
Rodrigo

```
struct List_CustomRows : View {
    var data = ["Custom Rows!", "Evans", "Lemuel James Guerrero", "Mark", "Durtschi", "Chase",
"Adam", "Rodrigo"]

    var body: some View {
        List(data, id: \.self) { datum in
            CustomRow(content: datum)
        }
    }
}
```

```
struct CustomRow: View {
    var content: String

    var body: some View {
        HStack {
            Image(systemName: "person.circle.fill")
            Text(content)
            Spacer()
        }
        .foregroundColor(content == "Custom Rows!" ? Color.green : Color.primary)
        .font(.title)
        .padding([.top, .bottom])
    }
}
```

Extracting rows into separate views is a common practice in SwiftUI. You can then have a separate preview just for the row.

Move Rows


8:40

Edit

To Do

Hit the Edit button to reorder

Practice Coding

Grocery shopping

Get tickets

Clean house

Do laundry

Cook dinner

Paint room

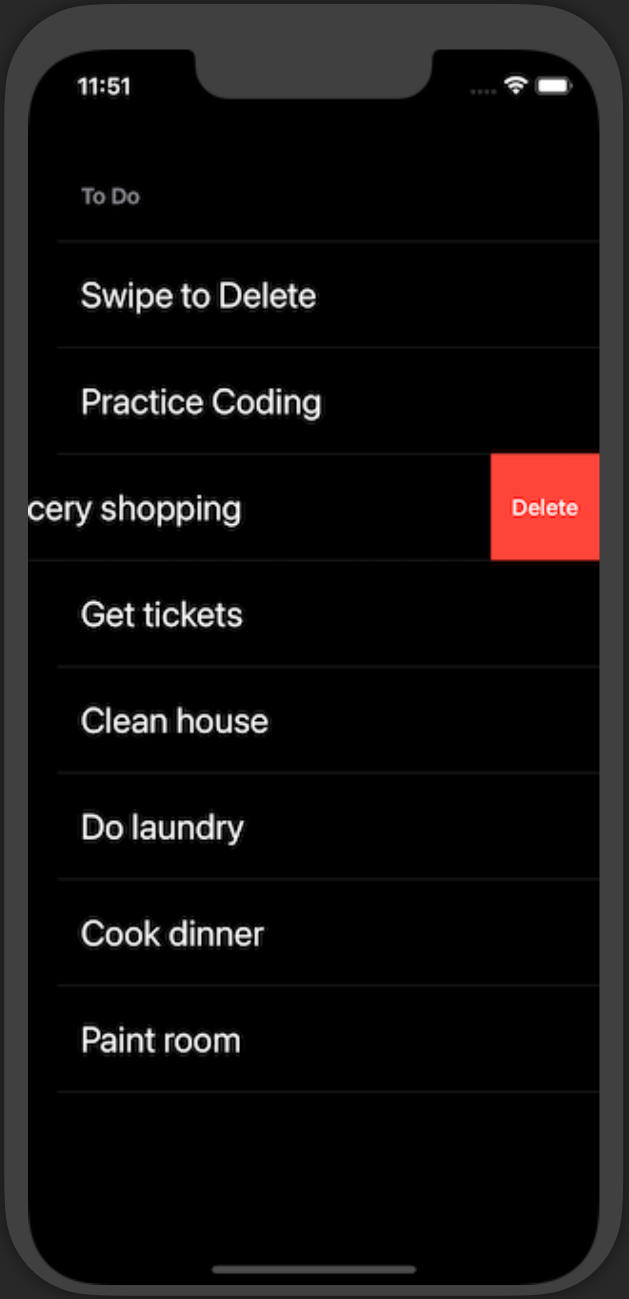
```
struct List_MoveRow : View {
    @State var data = ["Hit the Edit button to reorder", "Practice Coding", "Grocery shopping",
"Get tickets", "Clean house", "Do laundry", "Cook dinner", "Paint room"]

    var body: some View {
        NavigationView {
            List {
                ForEach(data, id: \.self) { datum in
                    Text(datum).font(Font.system(size: 24)).padding()
                }
                .onMove { source, destination in
                    data.move(fromOffsets: source, toOffset: destination)
                }
            }
            .navigationTitle("To Do")
            .toolbar {
                ToolbarItem { EditButton() }
            }
        }
        .tint(.green) // Changes color of buttons
    }
}
```

The onMove modifier goes on the ForEach, not the List.

What is EditButton()?

This is a built-in function that returns a view (Button) that will automatically toggle edit mode on the List. Its text says “Edit” and then when tapped you will see the move handles appear on the rows and the button text says “Done”.

Delete Rows

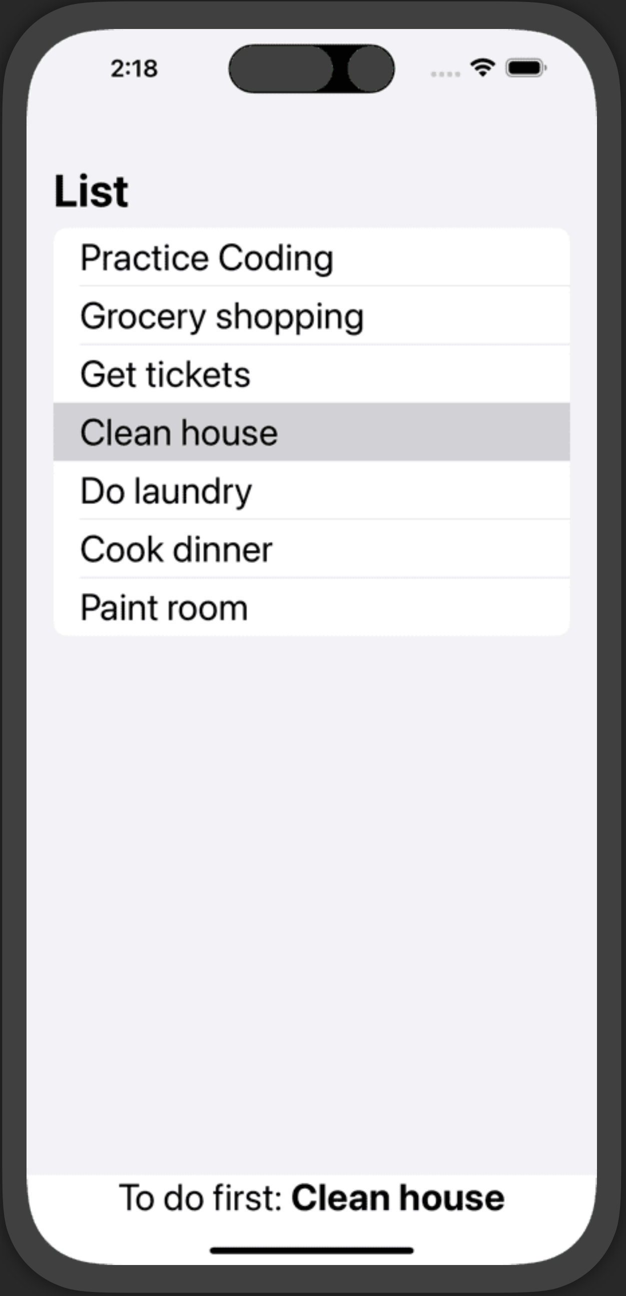


```
struct List_Delete : View {
    @State var data = ["Swipe to Delete", "Practice Coding", "Grocery shopping", "Get tickets",
"Clean house", "Do laundry", "Cook dinner", "Paint room"]

    var body: some View {
        List {
            Section {
                ForEach(data, id: \.self) { datum in
                    Text(datum).font(Font.system(size: 24)).padding()
                }
                .onDelete { offsets in
                    data.remove(atOffsets: offsets)
                }
            } header: {
                Text("To Do")
                    .padding()
            }
        }
        .listStyle(.plain)
    }
}
```

onDelete, onMove, onInsert

These three functions only work on views that implement the DynamicViewContent protocol. Currently, the only view that conforms to the DynamicViewContent protocol is the ForEach view. So these functions are only available on a ForEach view, not a List view.

Selecting a Row


2:18

List

Practice Coding
Grocery shopping
Get tickets
Clean house
Do laundry
Cook dinner
Paint room

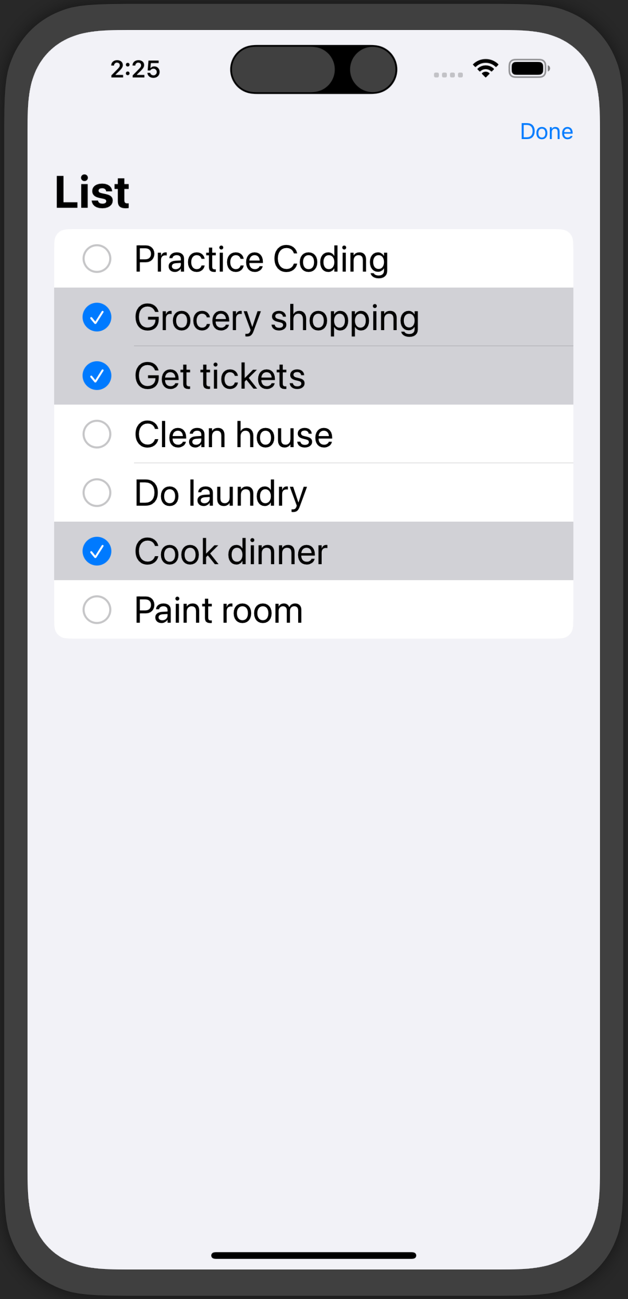
To do first: **Clean house**

```
struct List_Selection_Single: View {
    @State private var data = ["Practice Coding", "Grocery shopping", "Get tickets",
                                "Clean house", "Do laundry", "Cook dinner", "Paint room"]
    @State private var selection: String?

    var body: some View {
        NavigationStack {
            VStack(spacing: 0) {
                List(data, id: \.self, selection: $selection) { item in
                    Text(item)
                }
                Text("To do first: ") +
                Text(selection ?? "")
                    .bold()
            }
            .font(.title)
            .navigationTitle("List")
        }
    }
}
```

Allow rows to be selected.
Start by using an optional type to store which row is selected.
The type should match what you set for the id on the List.

Then bind the selection parameter to your @State property above using the dollar sign ($).

Selecting Multiple Rows


2:25

Done

List

- [ ] Practice Coding
- [x] Grocery shopping
- [x] Get tickets
- [ ] Clean house
- [ ] Do laundry
- [x] Cook dinner
- [ ] Paint room

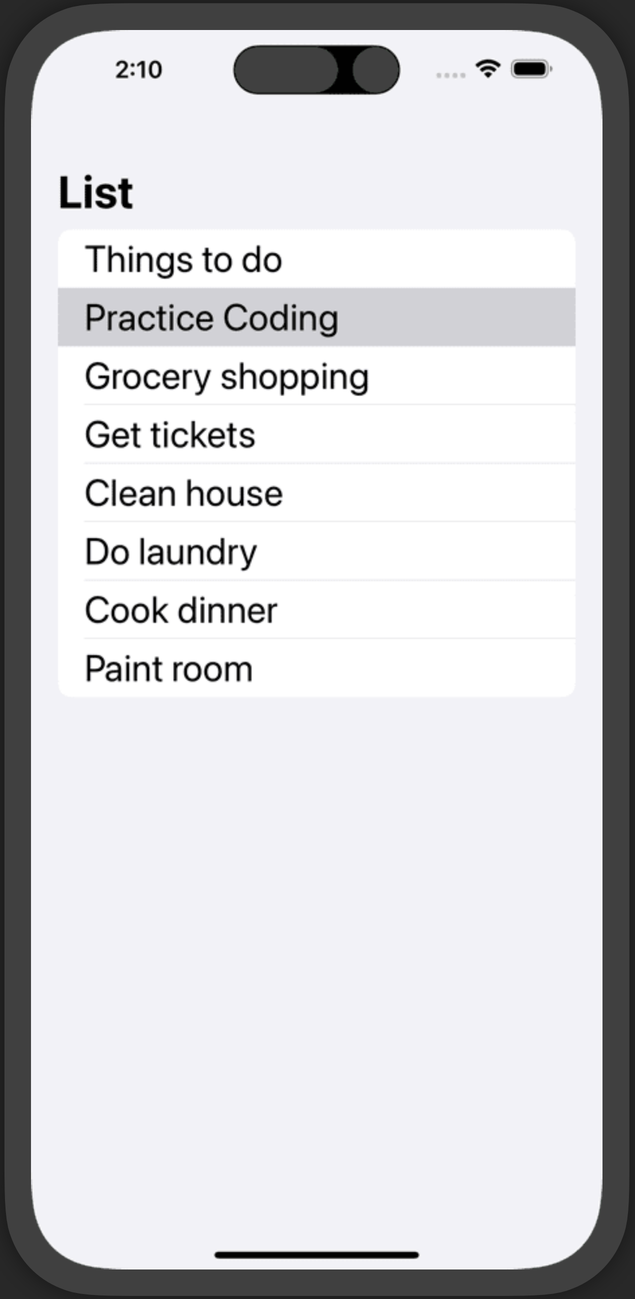
```
struct List_Selection_Multiple: View {
    @State private var data = ["Practice Coding", "Grocery shopping", "Get tickets",
                               "Clean house", "Do laundry", "Cook dinner", "Paint room"]
    @State private var selections = Set<String>()

    var body: some View {
        NavigationStack {
            List(data, id: \.self, selection: $selections) { item in
                Text(item)
            }
            .font(.title)
            .navigationTitle("List")
            .toolbar { EditButton() }
        }
    }
}
```

By changing the type to a Set, the List will automatically know it can hold multiple selection values.

You need the edit button to enable edit mode for multiple row selection.

SelectionDisabled

iOS 17



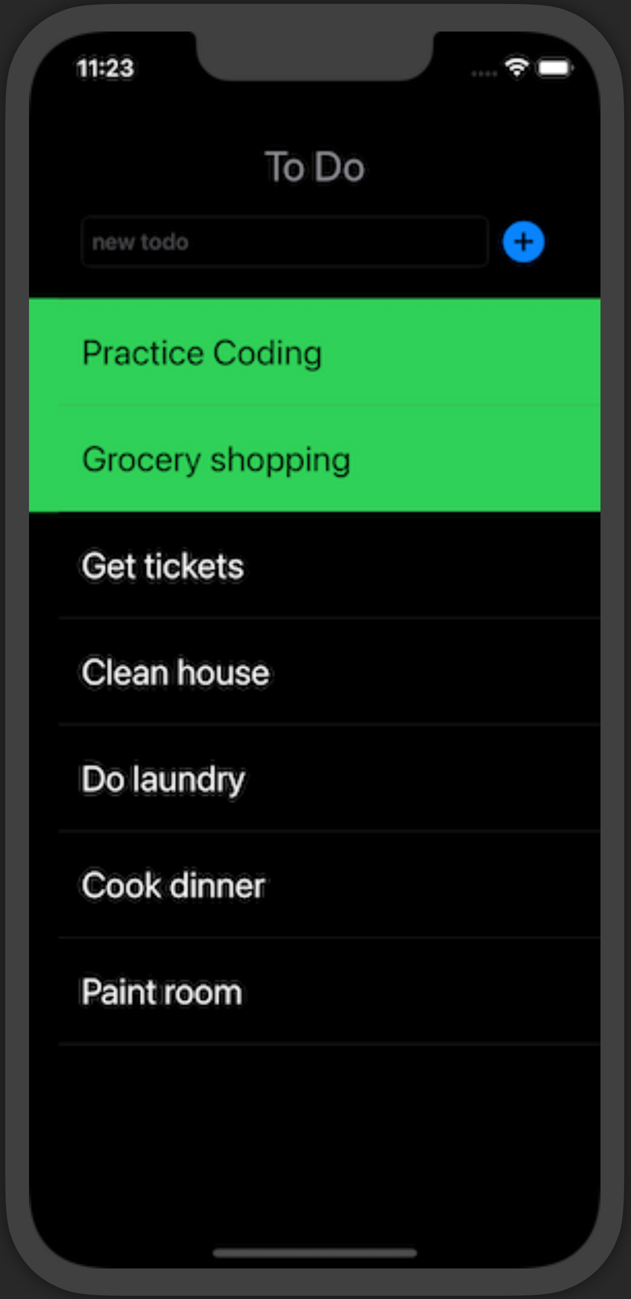
List

Things to do
Practice Coding
Grocery shopping
Get tickets
Clean house
Do laundry
Cook dinner
Paint room

```
struct List_SelectionDisabled: View {
    @State private var data = ["Things to do", "Practice Coding", "Grocery shopping",
                               "Get tickets", "Clean house", "Do laundry", "Cook dinner", "Paint room"]
    @State private var selection: String?

    var body: some View {
        NavigationStack {
            VStack(spacing: 0) {
                List(data, id: \.self, selection: $selection) { item in
                    Text(item)
                        .selectionDisabled(item == "Things to do")
                }
            }
            .font(.title)
            .navigationTitle("List")
        }
    }
}
```

If you do not want a row to be selected, you can use the selectionDisabled modifier.

List Row Background



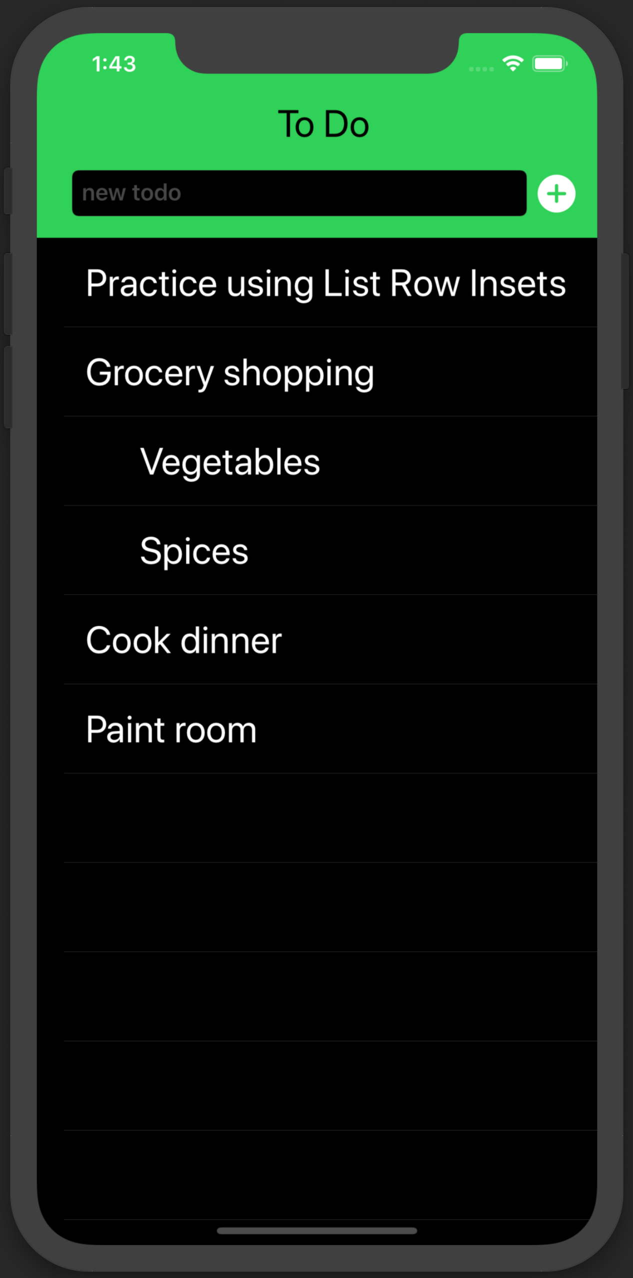
```
struct Todo: Identifiable {
    let id = UUID()
    var action = ""
    var due = ""
    var isIndented = false
}

struct List_ListRowBackground : View {
    @State private var newToDo = ""

    @State var data = [
        Todo(action: "Practice Coding", due: "Today"),
        Todo(action: "Grocery shopping", due: "Today"),
        Todo(action: "Get tickets", due: "Tomorrow"),
        Todo(action: "Clean house", due: "Next Week"),
        Todo(action: "Do laundry", due: "Next Week"),
        Todo(action: "Cook dinner", due: "Next Week"),
        Todo(action: "Paint room", due: "Next Week")
    ]

    var body: some View {
        List {
            Section {
                ForEach(data) { datum in
                    Text(datum.action)
                        .font(Font.system(size: 24))
                        .foregroundColor(self.getTextColor(due: datum.due))
                        // Turn row green if due today
                        .listRowBackground(datum.due == "Today" ? Color.green : Color.clear)
                        .padding()
                }
```

Notice the .listRowBackground function is on the view inside the ForEach. You want to call this function on whatever view will be inside the row, not on the List itself.

List Row Background Continued

```
            } header: {
                VStack {
                    Text("To Do").font(.title)
                    HStack {
                        TextField("new todo", text: $newToDo)
                            .textFieldStyle(.roundedBorder)
                        Button(action: {
                            data.append(Todo(action: newToDo))
                            newToDo = ""
                        }) {
                            Image(systemName: "plus.circle.fill").font(.title)
                        }
                    }
                }
                .padding(.bottom)
            }
        }
        .listStyle(.plain)
    }

    // This logic was inline but the compiler said it was "too complex" 🤷‍♂️
    private func getTextColor(due: String) -> Color {
        due == "Today" ? Color.black : Color.primary
    }
}
```

List Row Inset



```
struct List_ListRowInsets : View {
    @State private var newToDo = ""

    @State var data = [
        Todo(action: "Practice using List Row Insets", due: "Today"),
        Todo(action: "Grocery shopping", due: "Today"),
        Todo(action: "Vegetables", due: "Today", isIndented: true),
        Todo(action: "Spices", due: "Today", isIndented: true),
        Todo(action: "Cook dinner", due: "Next Week"),
        Todo(action: "Paint room", due: "Next Week")
    ]

    var body: some View {
        VStack {
            VStack {
                Text("To Do")
                    .font(.title)
                    .foregroundColor(.black)
                HStack {
                    TextField("new todo", text: $newToDo)
                        .textFieldStyle(.roundedBorder)
                    Button(action: {
                        data.append(Todo(action: newToDo))
                        newToDo = ""
                    }) {
                        Image(systemName: "plus.circle.fill")
                            .font(.title)
```

See next page for the code that insets the rows.

List Row Inset Continued

```
                    .foregroundColor(.white)
                }
            }
        }
        .padding()
        .background(Color.green)

        List {
            ForEach(self.data) { datum in
                Text(datum.action)
                    .font(.title)
                    .padding()
                // Inset row based on data
                    .listRowInsets(EdgeInsets(top: 0,
                                              leading: datum.isIndented ? 60 : 20,
                                              bottom: 0, trailing: 0))
            }
        }
        .listStyle(.plain)
    }
  }
}
```

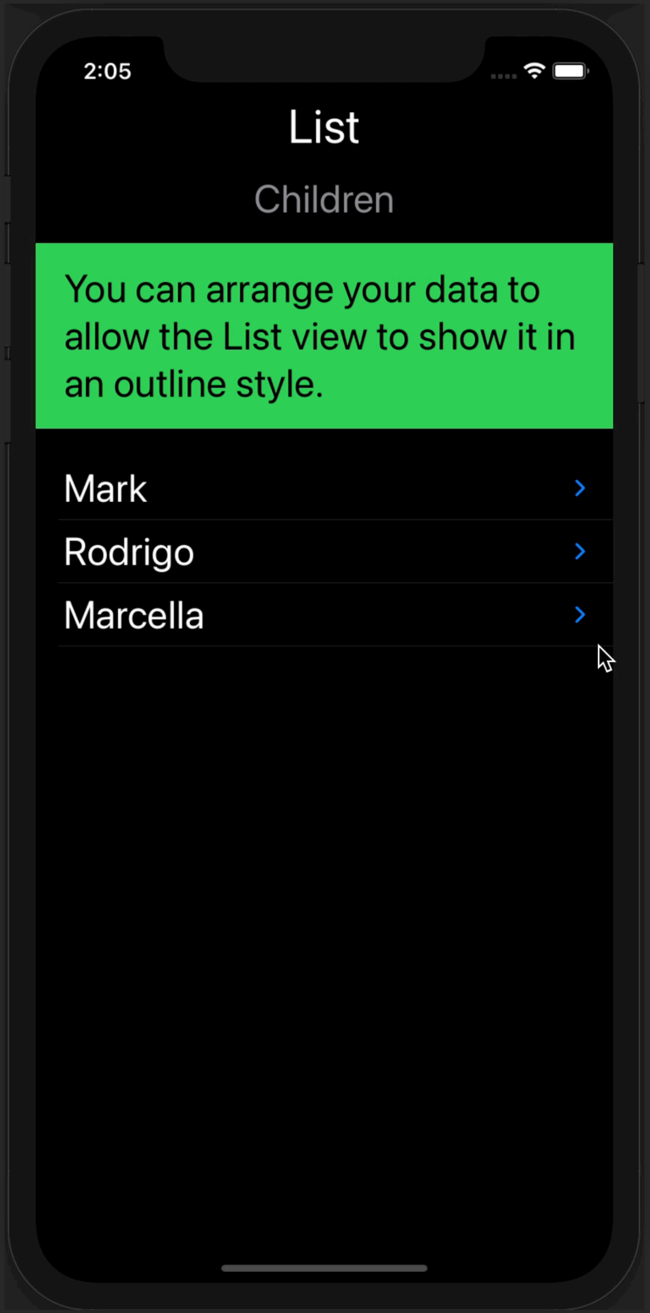
I'm using a condition here to determine just how much to inset the row.

Note: If you want to remove all default edge insets (setting all to zero), you can use:
`.listRowInsets(EdgeInsets())`

Note: If you want a row content to go to not be indented at all (so it touches the edge of the list, then set the leading edge to zero.

List With Children

iOS 14


2:05

List

Children

You can arrange your data to allow the List view to show it in an outline style.

Mark

Rodrigo

Marcella

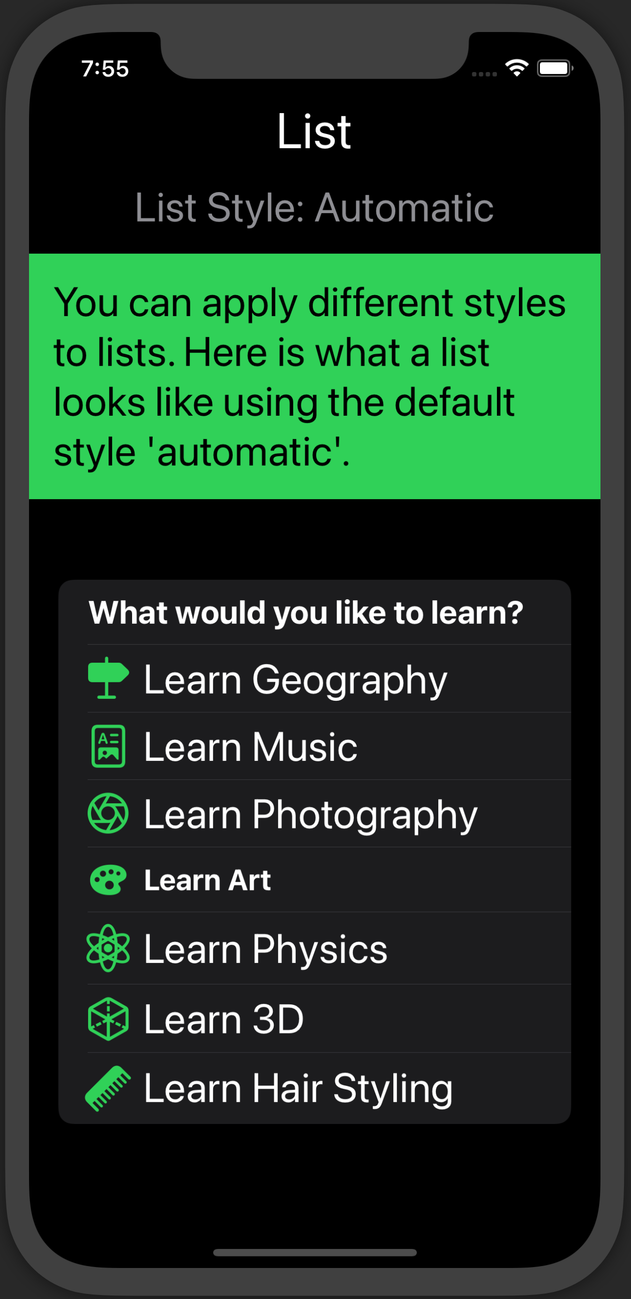
```
// Need to conform to Identifiable
struct Parent: Identifiable {
    var id = UUID()
    var name = ""
    var children: [Parent]? // Had to make this optional
}
```

If you have nested data, this could be a good way to represent it in a List.

```
struct List_WithChildren: View {
    var parents = [Parent(name: "Mark",
                          children: [Parent(name: "Paola")]),
                   Parent(name: "Rodrigo",
                          children: [Parent(name: "Kai"), Parent(name: "Brennan"),
                                     Parent(name: "Easton")]),
                   Parent(name: "Marcella",
                          children: [Parent(name: "Sam"), Parent(name: "Melissa"),
                                     Parent(name: "Melanie")])]

    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("List",
                       subtitle: "Children",
                       desc: "You can arrange your data to allow the List view to show it in an
                              outline style.", back: .green)

            List(parents, children: \.children) { parent in
                Text("\(parent.name)")
            }
        }
        .font(.title)
    }
}
```

Use the List init with the children parameter and use a key path to point to your nested property.

ListStyle: Automatic


7:55

List

List Style: Automatic

You can apply different styles to lists. Here is what a list looks like using the default style 'automatic'.

What would you like to learn?

- Learn Geography
- Learn Music
- Learn Photography
- **Learn Art**
- Learn Physics
- Learn 3D
- Learn Hair Styling

```swift
struct List_ListStyle_Automatic: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("List",
                       subtitle: "List Style: Automatic",
                       desc: "You can apply different styles to lists. Here is what a list looks
                              like using the default style 'automatic'.",
                       back: .green)
            List {
                Text("What would you like to learn?")
                    .font(.title2)
                    .fontWeight(.bold)
                Label("Learn Geography", systemImage: "signpost.right.fill")
                Label("Learn Music", systemImage: "doc.richtext")
                Label("Learn Photography", systemImage: "camera.aperture")
                Label("Learn Art", systemImage: "paintpalette.fill")
                    .font(Font.system(.title3).weight(.bold))
                Label("Learn Physics", systemImage: "atom")
                Label("Learn 3D", systemImage: "cube.transparent")
                Label("Learn Hair Styling", systemImage: "comb.fill")
            }
            .accentColor(.green)
            .listStyle(.automatic)
        }
        .font(.title)
    }
}
```

Note: You do not have to apply this modifier if the value is automatic. This is the default.

iOS 14 iOS 15 iOS 16

Additional List Content

Discover even more you can do with a list including: customizing separators (or removing them), swipe actions, section separators, safe area insets, apply tints and header/footer list styles.

This SwiftUI chapter is locked in this preview.

iOS 14

Menu

The Menu view allows you t[illegible] atta[illegible]h actio[illegible] to a view. Yo[illegible] some buttons (with or without images) and define a label, or a visible view to th[illegible] When the user taps the label, the actions will show.

This is similar to the **conte**[illegible] modifier (in the Contro[illegible] Modifiers chapter) where you can attach a menu to any view that becomes visible when yo[illegible] long-press the view.

This is a pull-in view.

iOS 16

MultiDatePicker

The multi-date picker allows ... ct many ... erstand is it does not allow the user to select a range of dates...just individua

The selected dates are not ordered.

This is a push-out view.

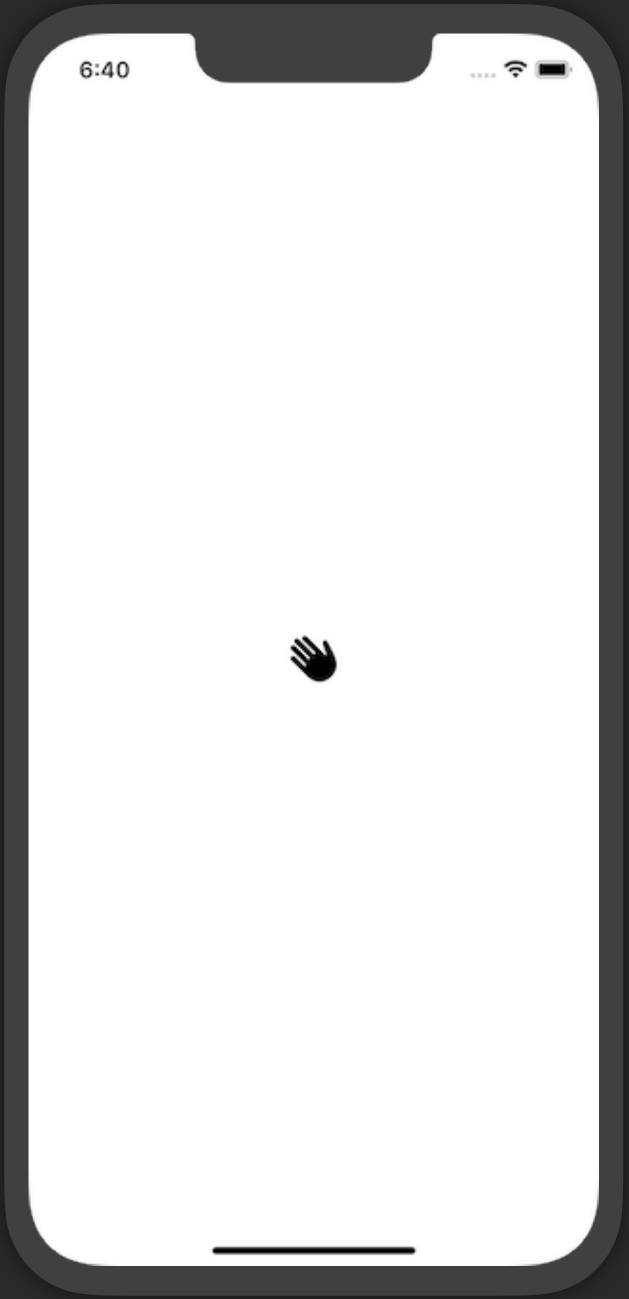
iOS 16

NavigationStack

The NavigationStack is slightly different in that it will fill the whole screen when used. You will never have to specify its size. It is used to help you navigate from one screen to another with built-in ways to navigate forward and back. This chapter will also teach you how to use and customize the navigation stack.

This is a push-out view.

Introduction

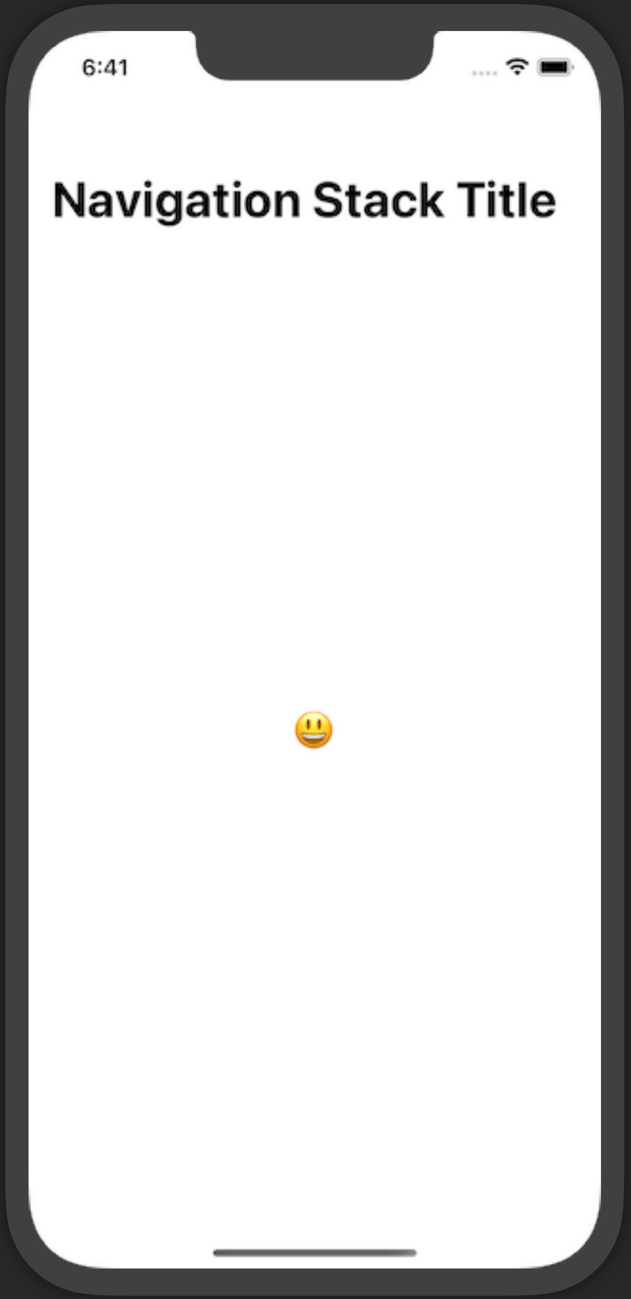


```
struct Navigation_Intro : View {
    var body: some View {
        NavigationStack {
            Image(systemName: "hand.wave.fill")
                .font(.largeTitle)
        }
    }
}
```

By default, the navigation stack will not show anything if there is no title modifier added for it to use.

The navigation stack bar is there at the top, you just can't see it.

Navigation Title



```
struct Navigation_NavigationTitle: View {
    var body: some View {
        NavigationStack {
            Text("😃")
                .navigationTitle("Navigation Stack Title")
        }
        .font(.title)
    }
}
```

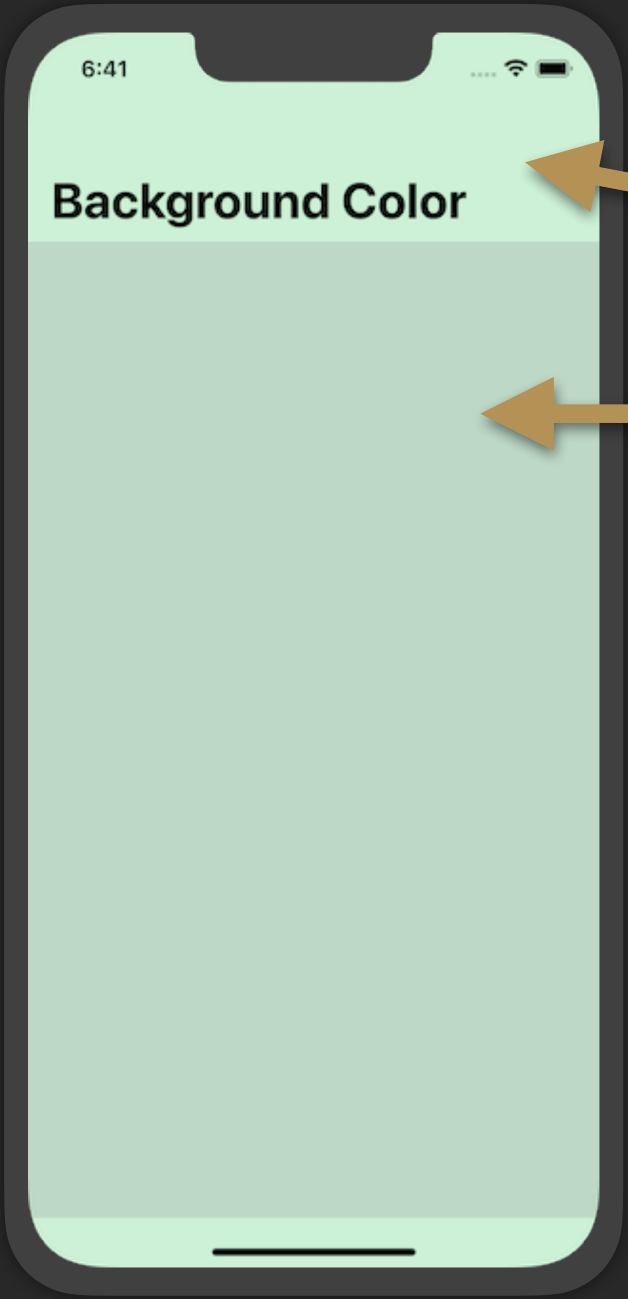
The navigation stack will look for a navigation title modifier INSIDE itself.

By default, a larger navigation bar is used for the navigation stack.

⚠️

Warning: If you put the navigation title on the navigation stack itself, it will not work.

Background Color



```
struct Navigation_BackgroundColor: View {
    var body: some View {
        NavigationStack {
            ZStack {
                Color.green.opacity(0.25)
                    .ignoresSafeArea()

                Color.gray.opacity(0.25)
            }
            .navigationTitle("Background Color")
        }
    }
}
```

The contents of the navigation stack is within the safe area.

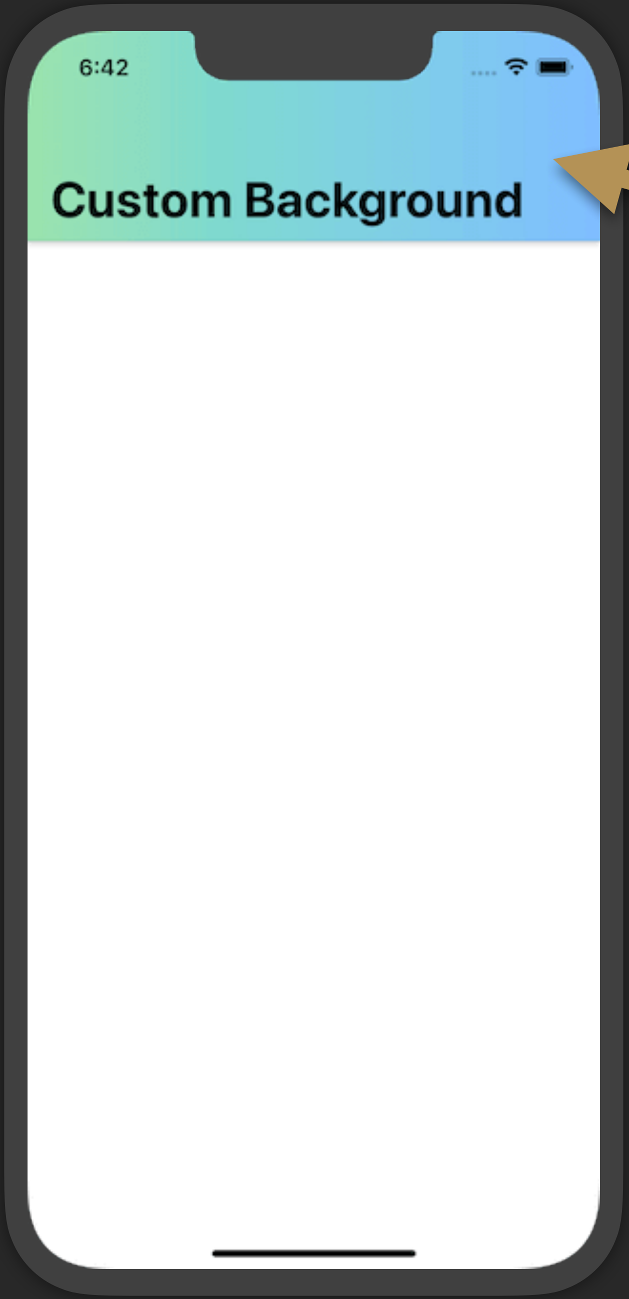
Where the nav bar is on top is an "unsafe area".

If you want a color to go behind the navigation stack's title bar, then you will have to ignore that safe area edge (or all edges).

This gray color gives you a better idea where the safe area is and where the edge is against the navigation stack's title bar.

Custom Background

iOS 15



```
struct Navigation_CustomBackground: View {
    var body: some View {
        NavigationStack {
            VStack {
                Divider()
                    .background(
                        LinearGradient(colors: [.green, .blue],
                                       startPoint: .leading,
                                       endPoint: .trailing)
                        .opacity(0.5)
                        .shadow(.drop(radius: 2, y: 2)),
                        ignoresSafeAreaEdges: .top)

                Spacer()
            }
            .navigationTitle("Custom Background")
        }
    }
}
```

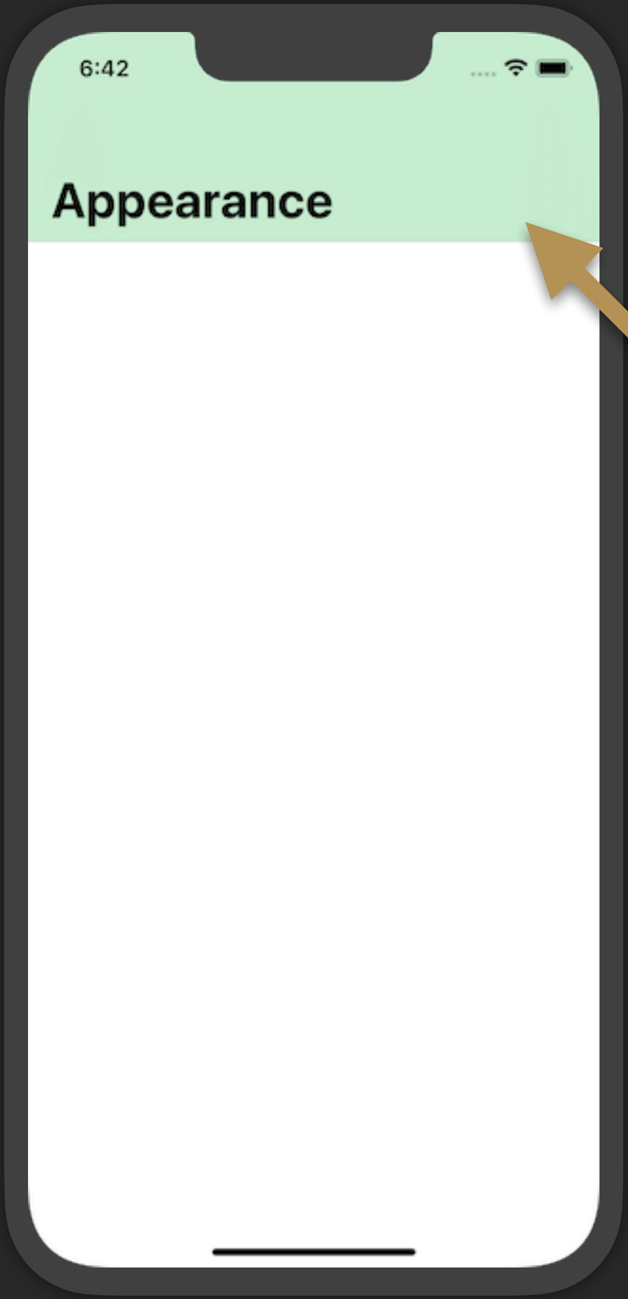
This background modifier will automatically ignore all safe area edges. That's why it expands into the nav bar area to create your nav background.

Use this optional parameter to specify which edges the background should ignore. By default it ignores all edges.

Note: This will change just this screen's navigation stack, not all nav stack though. To change all navigation stacks, see next page.

UINavigationBarAppearance

iOS 15

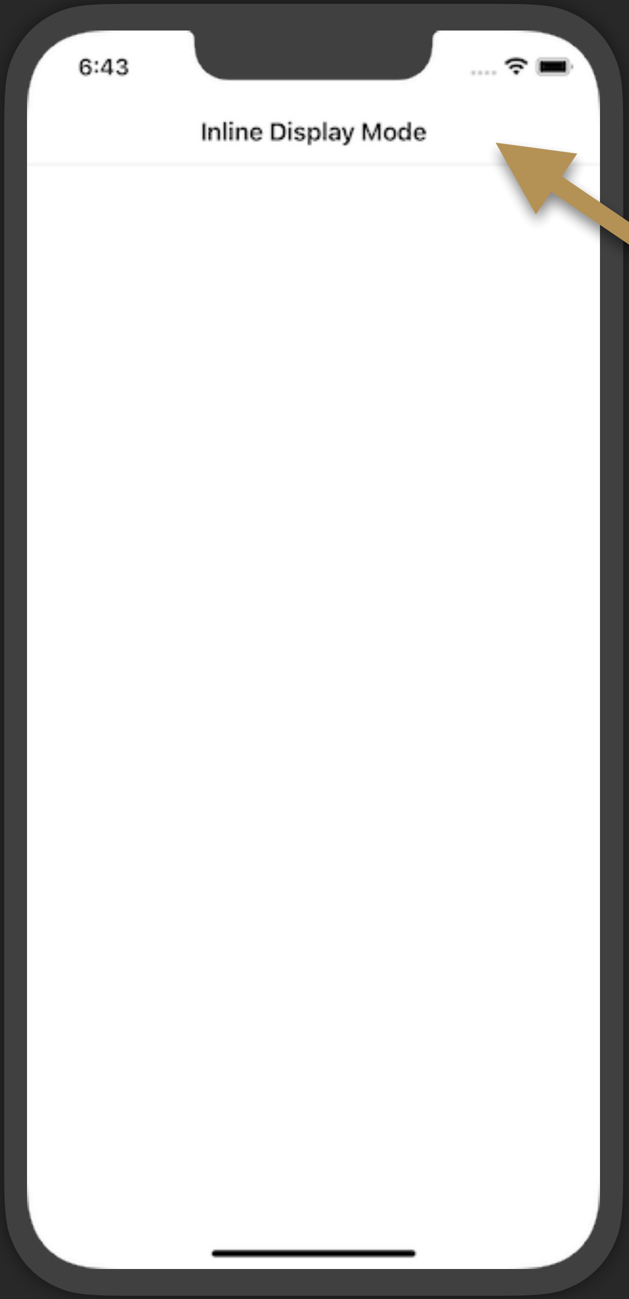


```
struct Navigation_UINavigationBarAppearance: View {
    var body: some View {
        NavigationStack {
            VStack {

            }
            .navigationTitle("Appearance")
            .font(.title)
        }
        .onAppear {
            let appearance = UINavigationBarAppearance()

            appearance.backgroundColor = UIColor(Color.green.opacity(0.25))

            UINavigationBar.appearance().standardAppearance = appearance
            UINavigationBar.appearance().scrollEdgeAppearance = appearance
        }
    }
}
```

Use UINavigationBarAppearance to apply style/color to ALL navigation bars.

Note: You could also set this when the app starts on the very first view of your app.

Display Mode

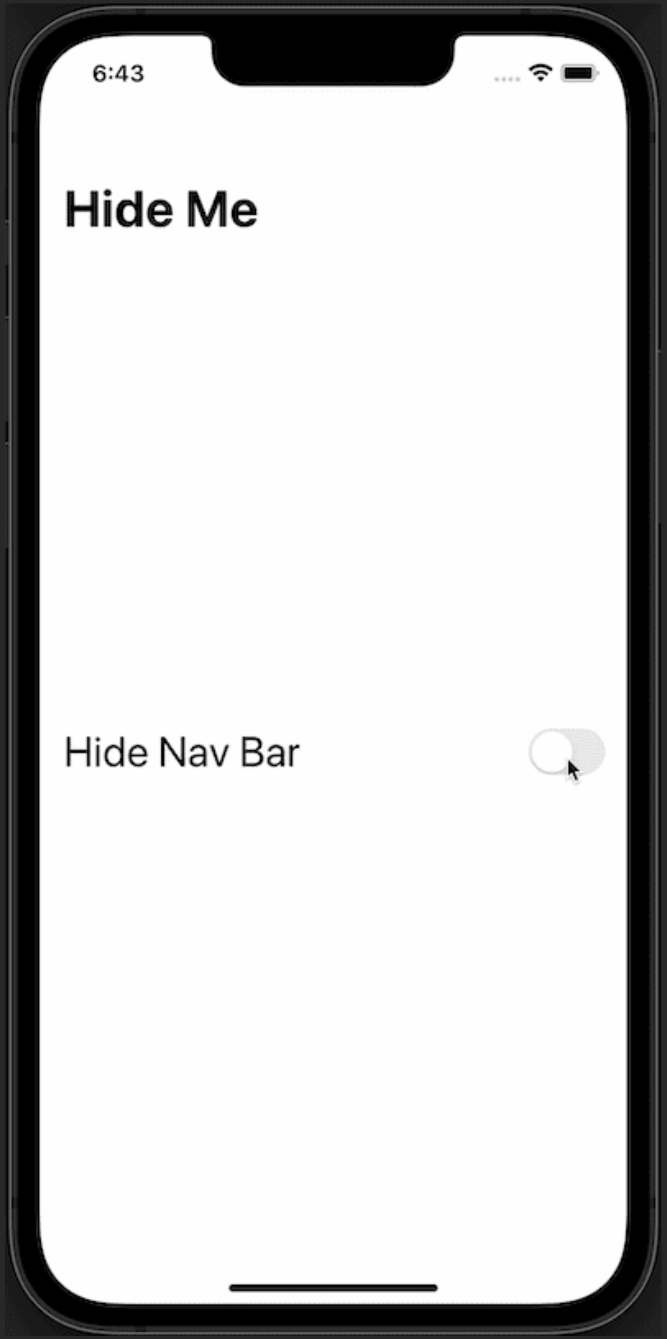


```
struct Navigation_DisplayMode: View {
    var body: some View {
        NavigationStack {
            VStack {
                Divider()
                Spacer()
            }
            .navigationTitle("Inline Display Mode")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}
```

Using .inline will render the smaller nav bar.

Note: If you navigate from this screen, the next one will also have an inline bar title.

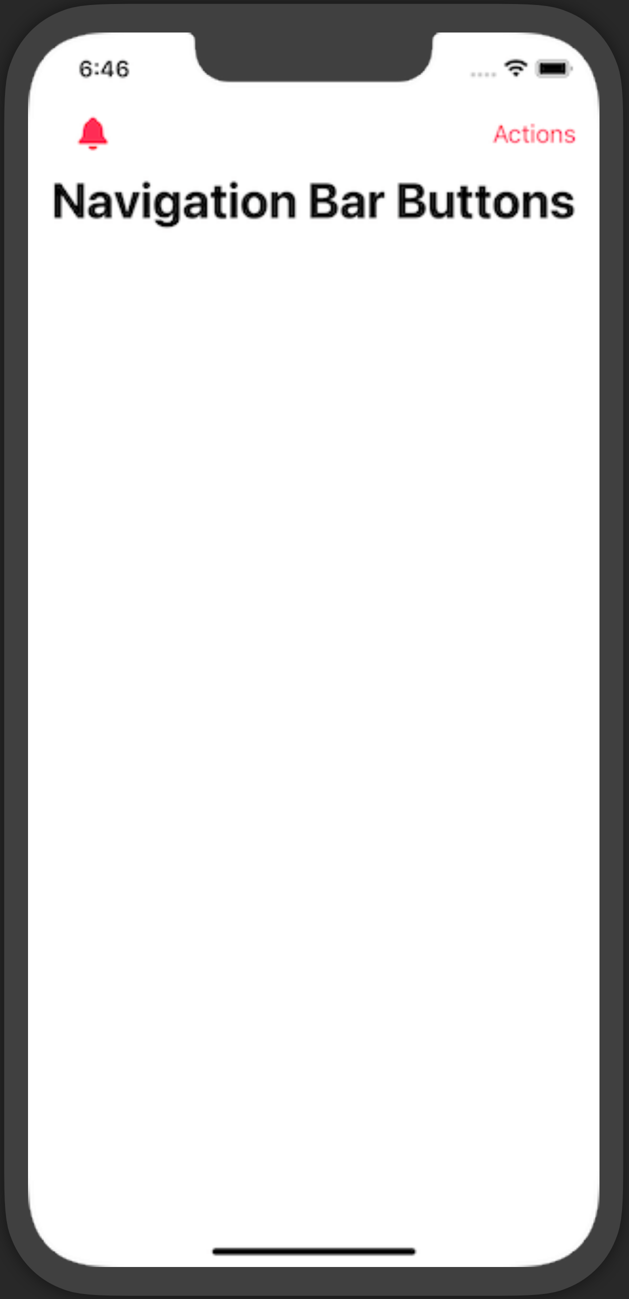
NavigationBarHidden



```
struct Navigation_BarHidden: View {
    @State private var isHidden = false

    var body: some View {
        NavigationStack {
            Toggle("Hide Nav Bar", isOn: $isHidden)
                .font(.title)
                .padding()
                .navigationBarHidden(isHidden)
                .navigationTitle("Hide Me")
        }
    }
}
```

Use this modifier if you want to hide the navigation bar. This is helpful when you want to control navigation yourself.

Navigation Bar Items



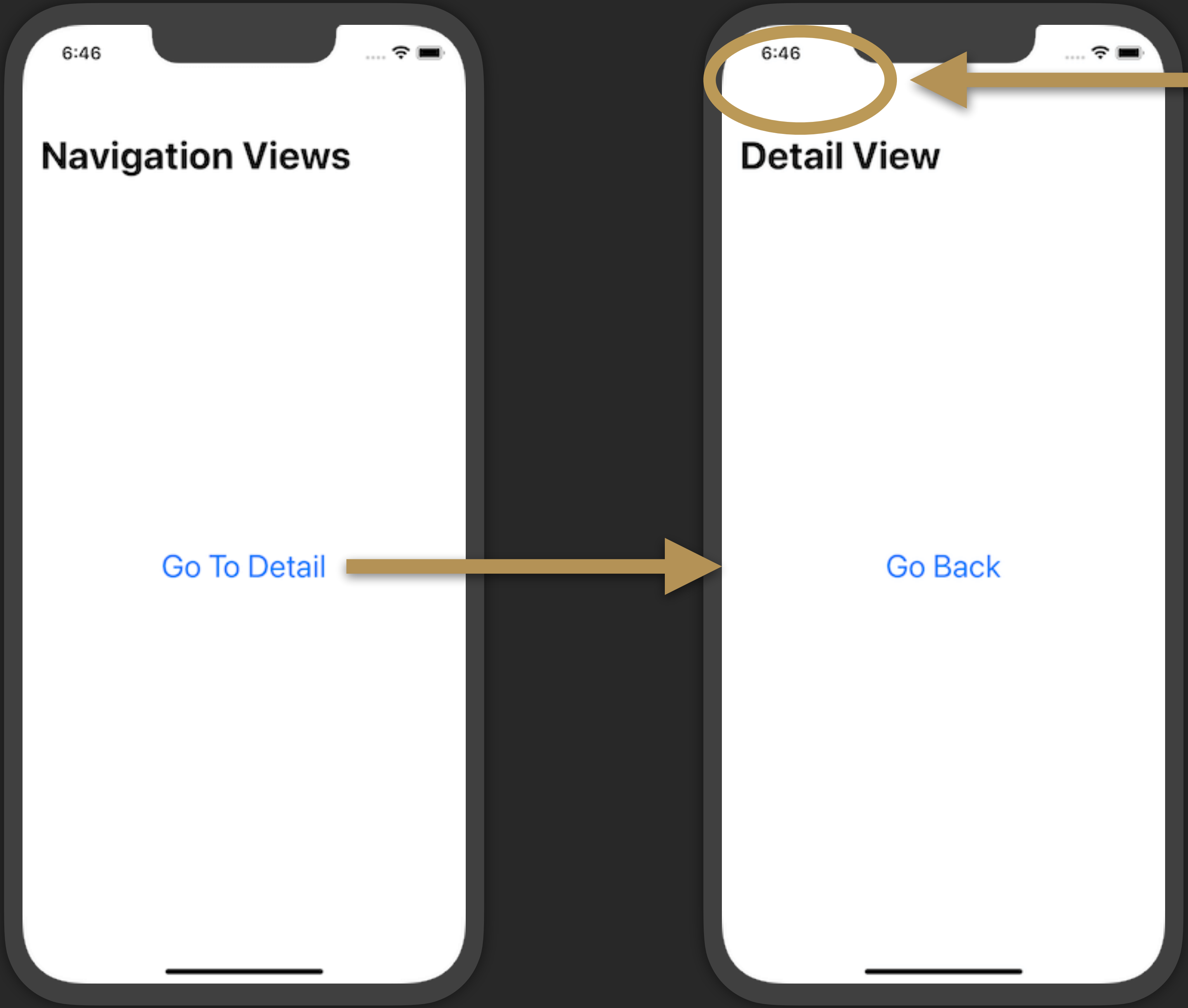
```
struct Navigation_NavBarItems : View {
    var body: some View {
        NavigationStack {
            VStack {
            }
            .navigationTitle("Navigation Bar Buttons")
            .toolbar {
                ToolbarItem(placement: .navigationBarLeading) {
                    Button(action: {}) {
                        Image(systemName: "bell.fill")
                            .padding(.horizontal)
                    }
                }
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Actions", action: { })
                }
            }
            .tint(.pink)
        }
    }
}
```

You can add navigation bar buttons to the leading or trailing (or both) sides of a navigation bar.

For more ways on how to use the toolbar modifier, go to Controls Modifiers > Toolbar section.

Use tint to change the color of the buttons.

NavigationBarBackButtonHidden



You can hide the back button in the navigation bar for views by using a modifier. (Code on next page.)

This is good in scenarios where you supply another button to navigate the user back or you want to supply your own custom back button (see next example for custom back button).

(Code) NavigationBarBackButtonHidden

```
// First Screen
struct Navigation_BackButtonHidden: View {
    var body: some View {
        NavigationStack {
            NavigationLink("Go To Detail", destination: BackButtonHiddenDetail())
                .font(.title)
                .navigationTitle("Navigation Views")
        }
    }
}

// Second Screen
struct BackButtonHiddenDetail: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        Button("Go Back") {
            dismiss()
        }
        .font(.title)
        .navigationTitle("Detail View")
        // Hide the back button
        .navigationBarBackButtonHidden(true)
    }
}
```

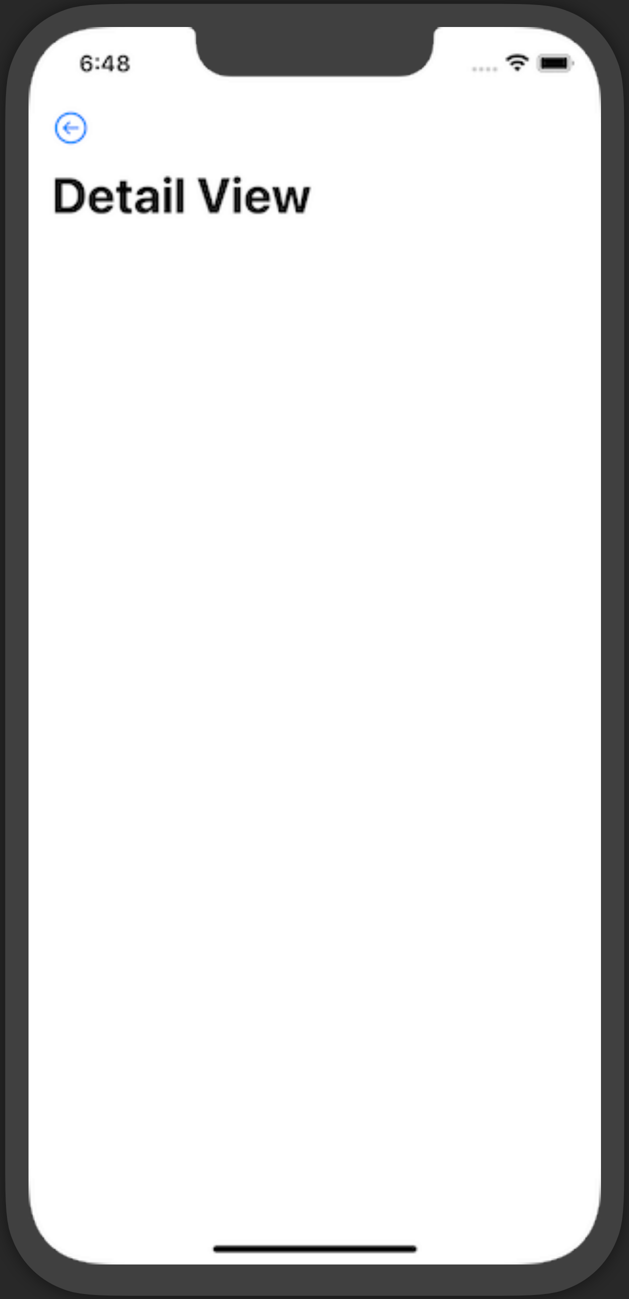
Use NavigationLink to navigate to a new screen.
More about NavigationLink in the next section.

This will allow you to navigate backward.

Dismissing what is being presented will navigate you back to the previous screen.

The back button is hidden. This allows you to create a custom back button where you might want to add logic to it.

Custom Back Button



```
struct Navigation_CustomBackButton: View {
    var body: some View {
        NavigationStack {
            NavigationLink("Go To Detail",
                           destination: Navigation_CustomBackButton_Detail())
            .font(.title)
            .navigationTitle("Navigation Views")
        }
    }
}
// Second Screen
struct Navigation_CustomBackButton_Detail: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack {
        }
        .navigationTitle("Detail View")
        .navigationBarBackButtonHidden(true)
        .toolbar {
            ToolbarItem(placement: .topBarLeading) {
                Button(action: {
                    dismiss()
                }) {
                    Label("Back", systemImage: "arrow.left.circle")
                }
            }
        }
    }
}
```

Hide the system back button and then use toolbar modifier to add a leading button.

Toolbar buttons can't show text and images at the same time.

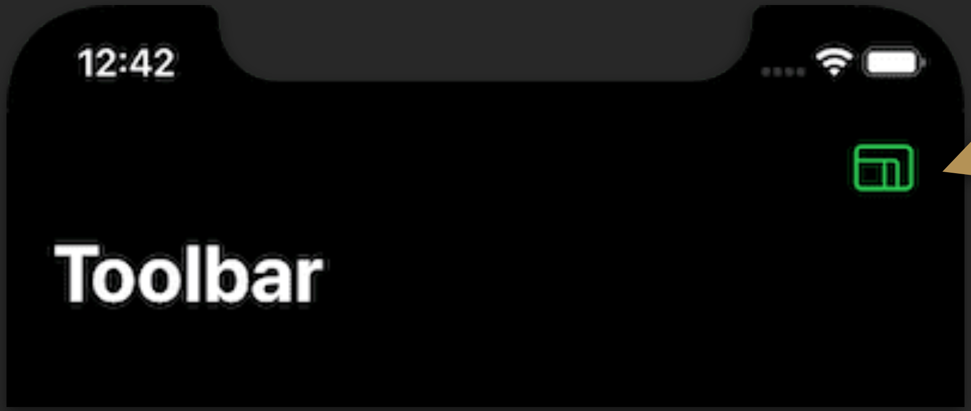
If you want text and image, then hide the whole nav bar (previous example).

Warning: By hiding the back button, you will lose the ability to swipe back to the previous screen.

Customizing the Nav Bar

Learn about more options for adding buttons and customizing the NavigationView with the toolbar.

To explore this functionality, go to **Controls Modifier > Toolbar**.



iOS 16

NavigationSplitView

This SwiftUI chapter is locked in this preview.

You have probably seen on an [illegible] macOS an app that has 2 or 3 vertical panes. In the far left pane you can tap on items that shows more details to the right. This is a common navigational pattern that can be achieved with the navigation split view.

This is a push-out view.

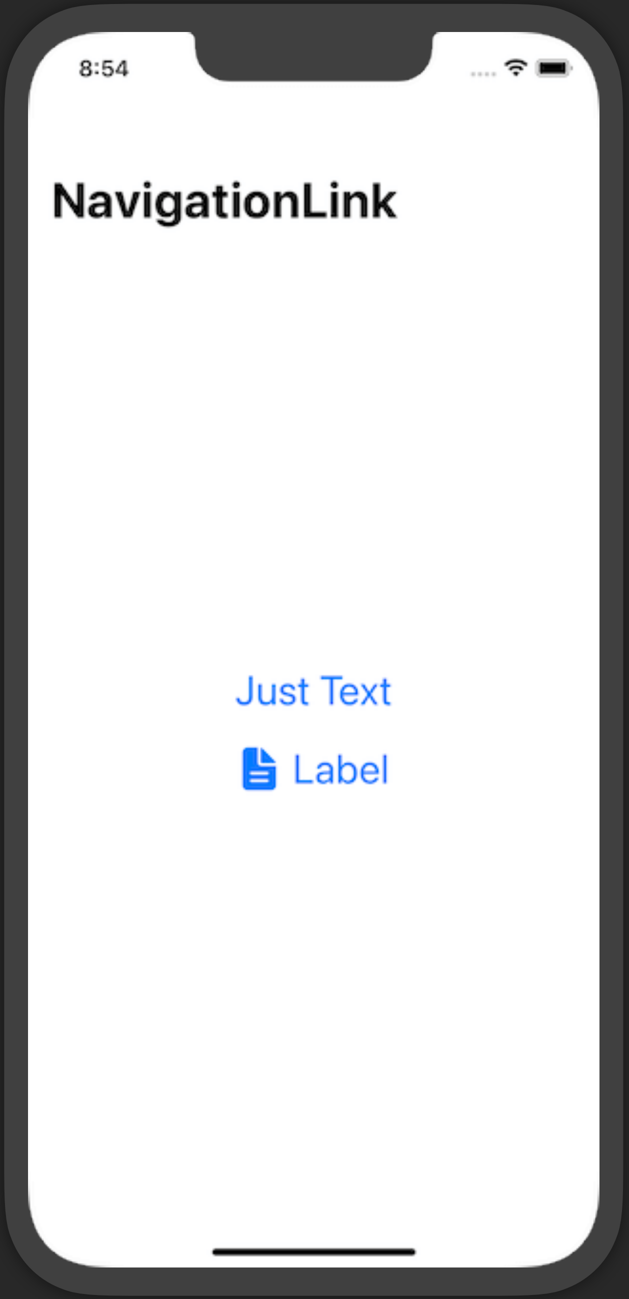
SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

NavigationLink

The NavigationLink is your way to navigate to another view. It ONLY works inside of a NavigationView. The appearance is just like a Button. You can customize it just like you can customize a Button too.

This is a pull-In view.

Introduction



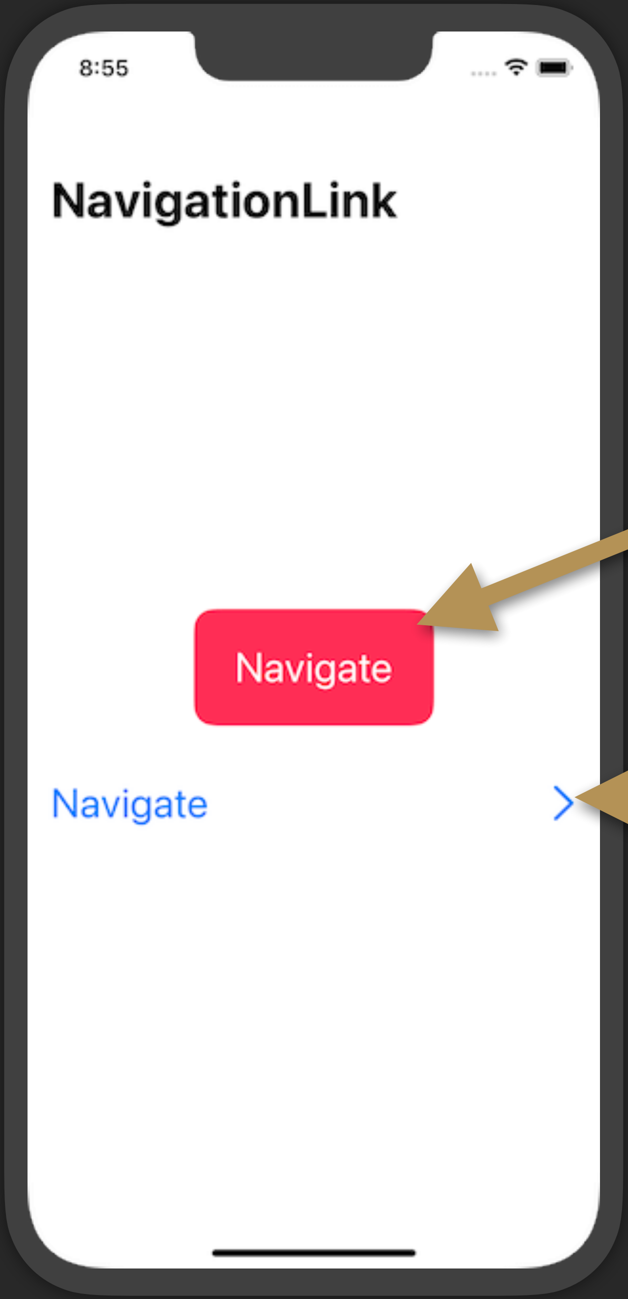
```
struct NavLink_Intro: View {
    var body: some View {
        NavigationStack {
            VStack(spacing: 20) {
                NavigationLink("Just Text", destination: SecondView())

                NavigationLink {
                    SecondView()
                } label: {
                    Label("Label", systemImage: "doc.text.fill")
                }
            }
            .navigationTitle("NavigationLink")
        }
        .font(.title)
    }
}

struct SecondView: View {
    var body: some View {
        VStack {
            Text("View 2")
                .font(.largeTitle)
        }
        .navigationTitle("Second View")
    }
}
```

Use NavigationLink to navigate to a new view inside a NavigationStack.

You can also use the label parameter to customize the tappable view to trigger navigation.

These are the basic implementations using just text or a label and a destination. The destination can be any view.

Customization



```swift
struct NavLink_Customization: View {
    var body: some View {
        NavigationStack {
            VStack(spacing: 20) {
                NavigationLink(destination: SecondView()) {
                    Text("Navigate")
                        .foregroundStyle(.white)
                        .padding()
                }
                .buttonStyle(.borderedProminent)
                .tint(.pink)

                NavigationLink(destination: SecondView()) {
                    LabeledContent("Navigate") {
                        Image(systemName: "chevron.right")
                            .foregroundStyle(.accentColor)
                    }
                    .padding()
                }
            }
            .navigationTitle("NavigationLink")
        }
        .font(.title)
    }
}
```

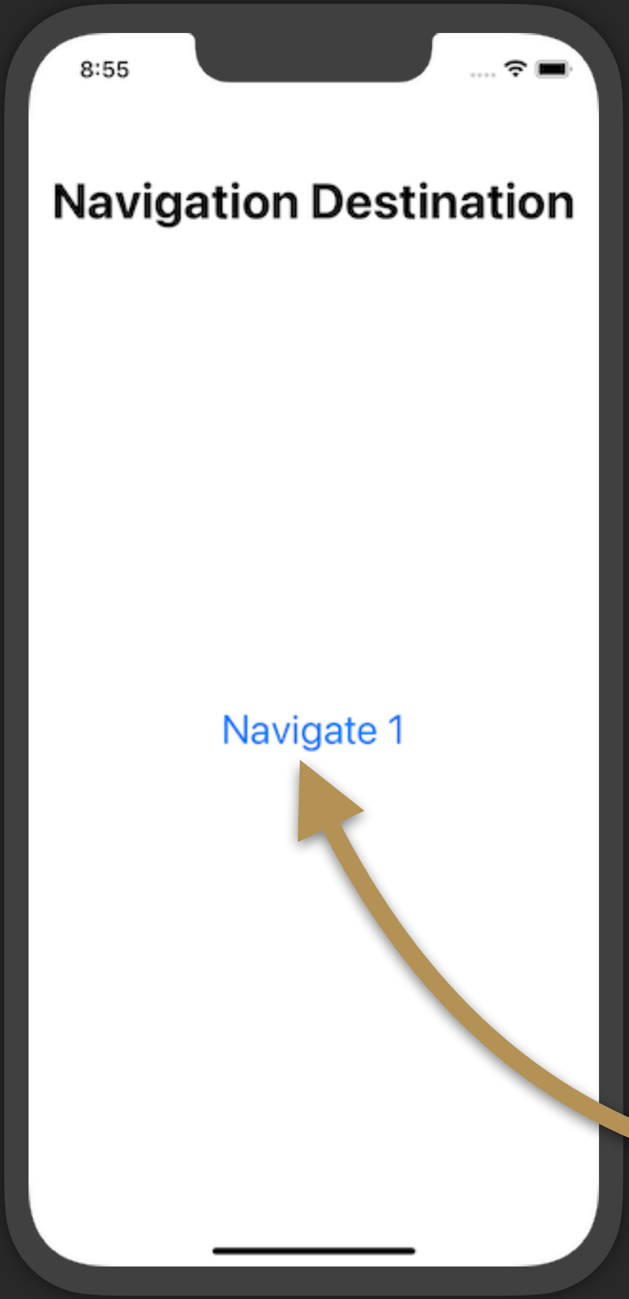
You can customize NavigationLink just like you would with a Button.

The trailing closure is the label parameter. This allows you to compose any view that will navigate you.

Tip: Try to keep your views and modifiers within the closure. Like the Button, anything inside will fade when tapped.

With Navigation Destination

iOS 16



```swift
struct NavLink_WithNavigationDestination: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink(value: "NavigationLink 1") {
                    Text("Navigate 1")
                }
            }
            .navigationDestination(for: String.self) { stringValue in
                Text("Value is: ") + Text("\(stringValue)").bold()
            }
            .navigationTitle("Navigation Destination")
        }
        .font(.title)
    }
}
```

A NavigationLink that uses a value will not do anything by itself. (Notice there is no a destination nor a view here.)

There must be a navigationDestination modifier that is looking for the same **type** (String in this example).

Here are some things to note:

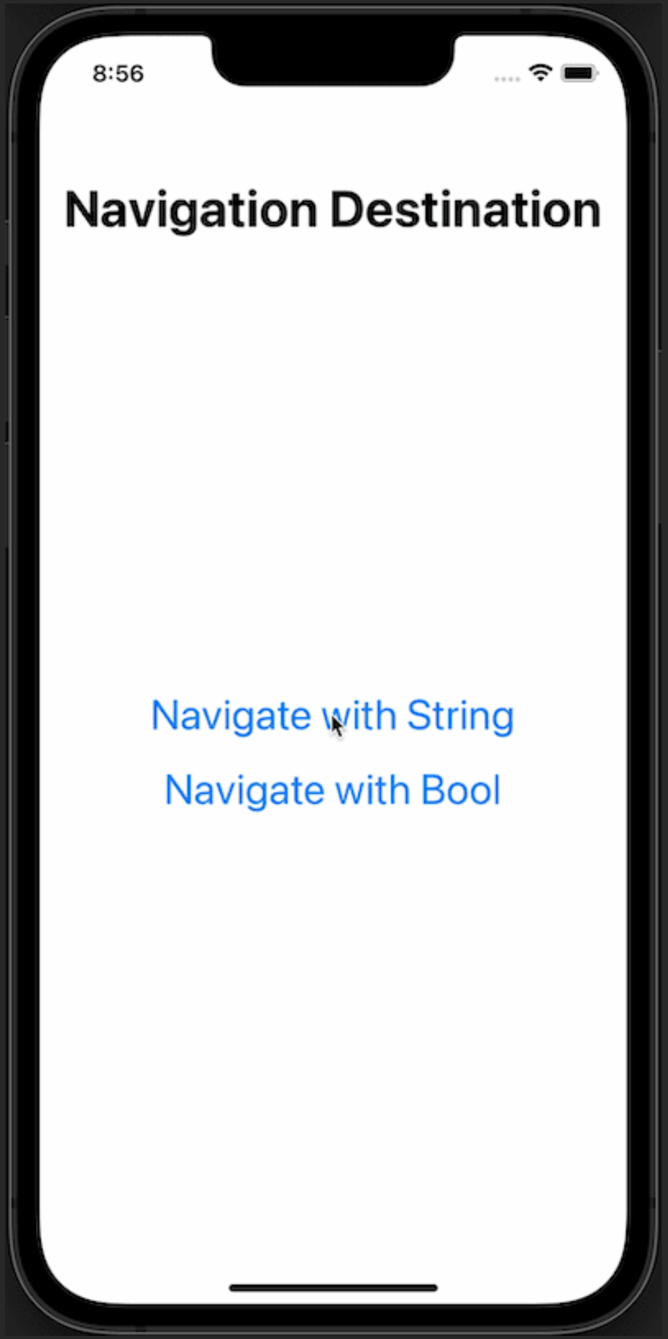
1. The navigationDestination is the modifier that does the navigating.
2. The view within the closure is the destination.
3. The value specified by the NavigationLink is passed into that closure and can be used in the destination view.

This might be a confusing concept to understand at first.

Just know that when the navigation link button is tapped, it will look for a navigationDestination modifier that handles the same type as the value that is set.

With Different Types

iOS 16

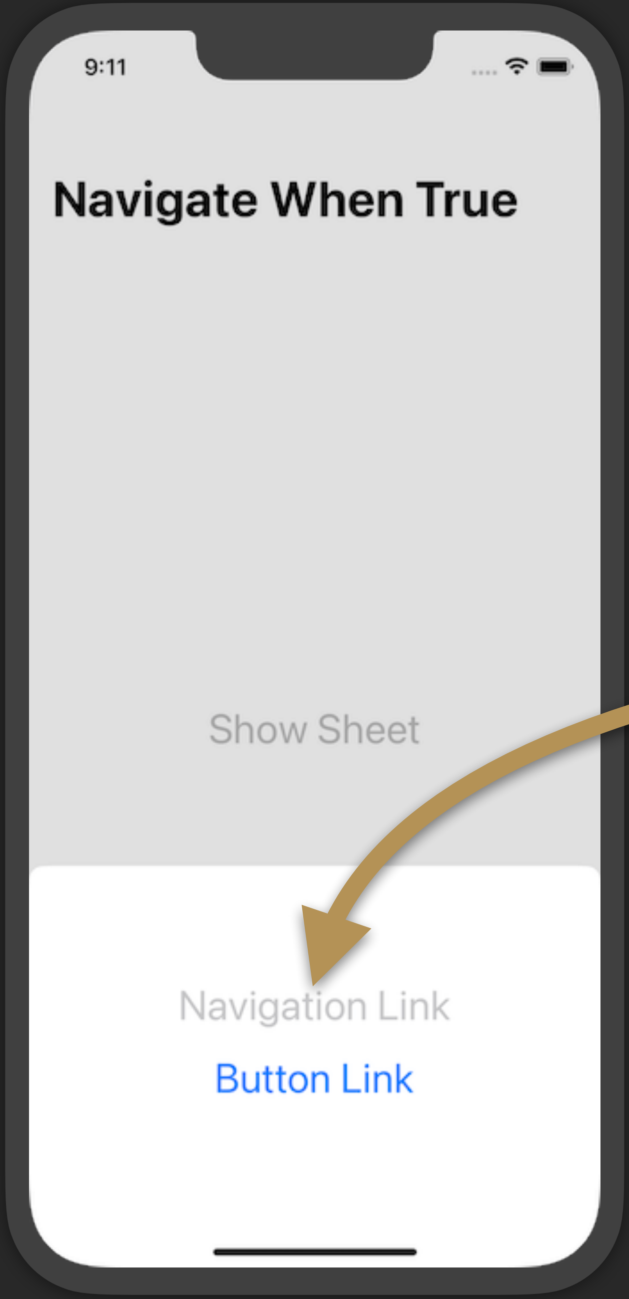


```
struct NavLink_WithDifferentTypes: View {
    var body: some View {
        NavigationStack {
            VStack(spacing: 16.0) {
                NavigationLink(value: "NavigationLink 1") {
                    Text("Navigate with String")
                }
                NavigationLink(value: true) {
                    Text("Navigate with Bool")
                }
            }
            .navigationDestination(for: String.self) { stringValue in
                Text("Value is: ") + Text("\(stringValue)").bold()
            }
            .navigationDestination(for: Bool.self) { boolValue in
                Text("Value is: ") + Text("\(boolValue.description)").bold()
            }
            .navigationTitle("Navigation Destination")
        }
        .font(.title)
    }
}
```

The navigation links will automatically link up to the navigation destination modifiers of the same type.

isPresented

iOS 16



```
struct NavLink_isPresented: View {
    @State private var showSheet = false
    @State private var navigate = false

    var body: some View {
        NavigationStack {
            VStack {
                Button("Show Sheet") { showSheet.toggle() }
            }
            .navigationTitle("Navigate When True")
            .sheet(isPresented: $showSheet) {
                VStack(spacing: 16.0) {
                    NavigationLink(destination: Text("Destination from Link")) {
                        Text("Navigation Link")
                    }
                    Button("Button Link") {
                        showSheet = false
                        navigate = true
                    }
                }
                .presentationDetents([.height(240)])
            }
            .navigationDestination(isPresented: $navigate) {
                Text("Destination from Button")
            }
        }
        .font(.title)
    }
}
```

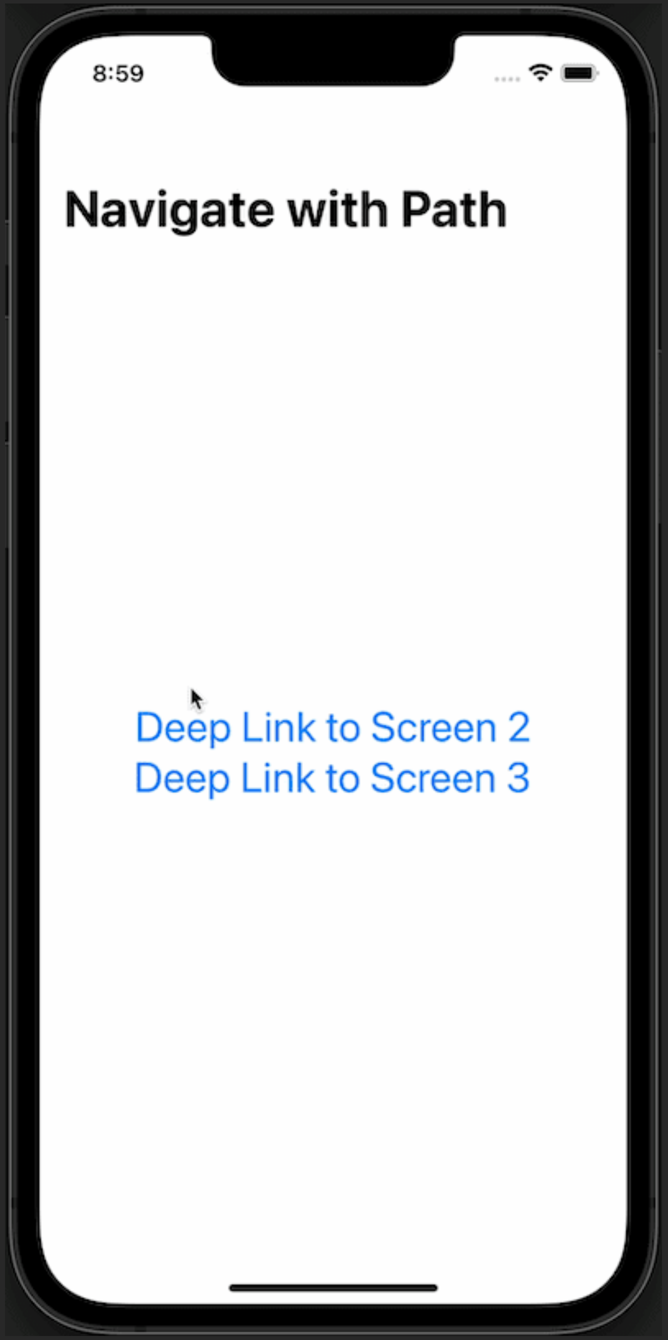
At this time, when a NavigationLink is used inside a sheet it will be disabled.

One solution is to combine a Button and the navigationDestination that uses the isPresented parameter.

Warning: The navigationDestination has to be outside the sheet for this to work.

Navigation Path with Enum

iOS 16



```
enum Screens {
    case screen1
    case screen2
    case screen3
}
```

You can also use an enum to keep track of all of your screens.

```
struct Nav_WithPathAndEnum: View {
    @State private var navPath: [Screens] = []

    var body: some View {
        NavigationStack(path: $navPath) {
            VStack {
                Button("Deep Link to Screen 2") {
                    navPath.append(Screens.screen1)
                    navPath.append(Screens.screen2)
                }
                Button("Deep Link to Screen 3") {
                    navPath.append(Screens.screen1)
                    navPath.append(Screens.screen2)
                    navPath.append(Screens.screen3)
                }
            }
            .navigationDestination(for: Screens.self) { screenEnum in
                NavigationController.navigate(to: screenEnum)
            }
            .navigationTitle("Navigate with Path")
        }
        .font(.title)
    }
}
```

The navPath is now set to the enum.

The NavigationStack's path is bound to an array of enums now.

This navigation destination modifier is now navigating for the Screens enum type.

This NavigationController is handling where to navigate to based on the enum value. (See next page.)

NavigationController

```
class NavigationController {
    @ViewBuilder
    static func navigate(to screen: Screens) -> some View {
        switch screen {
        case .screen1:
            Image(systemName: "1.square.fill").font(.largeTitle).foregroundStyle(.green)
        case .screen2:
            Image(systemName: "2.square.fill").font(.largeTitle).foregroundStyle(.red)
        case .screen3:
            Image(systemName: "3.square.fill").font(.largeTitle).foregroundStyle(.purple)
        }
    }
}
```

This navigate function returns the view to navigate to.

This is a good method when you:

1. Have no data to pass into your views.
2. Pass in the same object/data to all of your screens.
3. Keep data in a global location where all your views can access it.

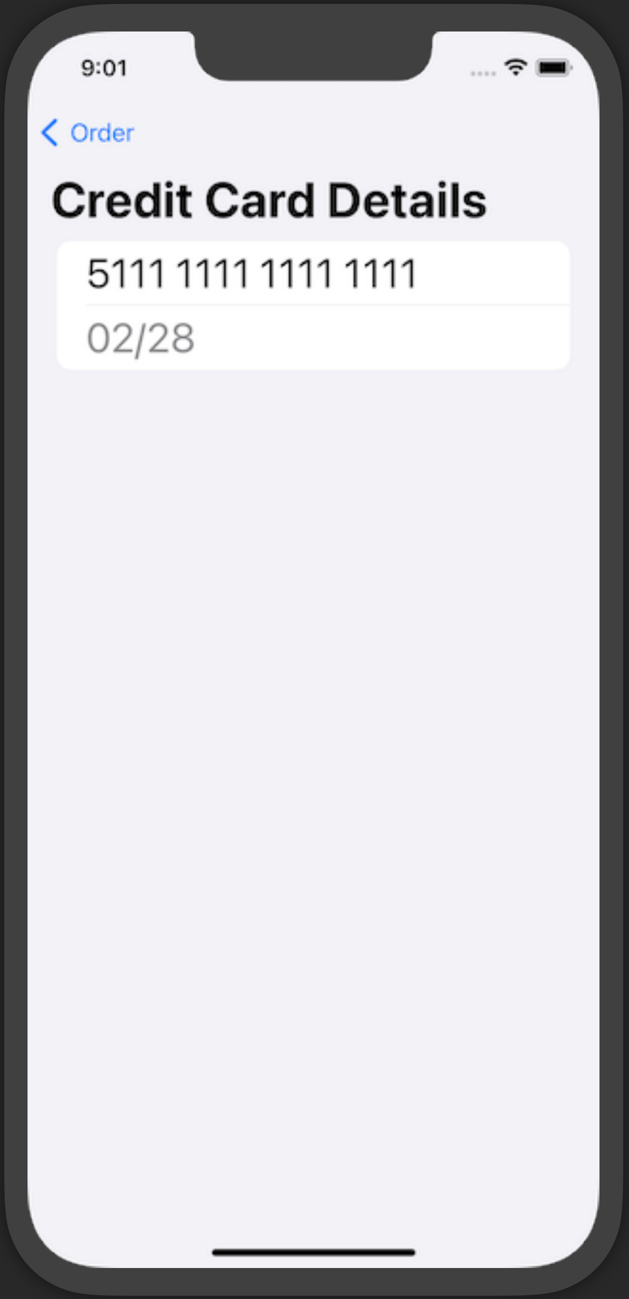
NavigationPath

iOS 16

```
struct ProductForNav: Hashable {
    var name = ""
    var price = 0.0
}
```

```
struct CreditCardForNav: Hashable {
    var number = ""
    var expiration = ""
}
```

Two different types used in navigation.



Credit Card Details

5111 1111 1111 1111

02/28

```
struct Nav_WithNavigationPath: View {
    @State private var product = ProductForNav(name: "Mouse", price: 24.99)
    @State private var cc = CreditCardForNav(number: "5111 1111 1111 1111", expiration: "02/28")
    @State private var navPath = NavigationPath()

    var body: some View {
        NavigationStack(path: $navPath) {
            Form {
                NavigationLink(value: product) { Text(product.name) }
                NavigationLink(value: cc) { Text(cc.number) }
            }
            .navigationTitle("Order")
            .navigationDestination(for: ProductForNav.self) { product in
                Form {
                    Text(product.name)
                    Text(product.price, format: .currency(code: "USD"))
                        .foregroundStyle(.secondary)
                }.navigationTitle("Product Details")
            }
            .navigationDestination(for: CreditCardForNav.self) { cc in
                Form {
                    Text(cc.number)
                    Text(cc.expiration).foregroundStyle(.secondary)
                }.navigationTitle("Credit Card Details")
            }
        }
        .font(.title)
    }
}
```

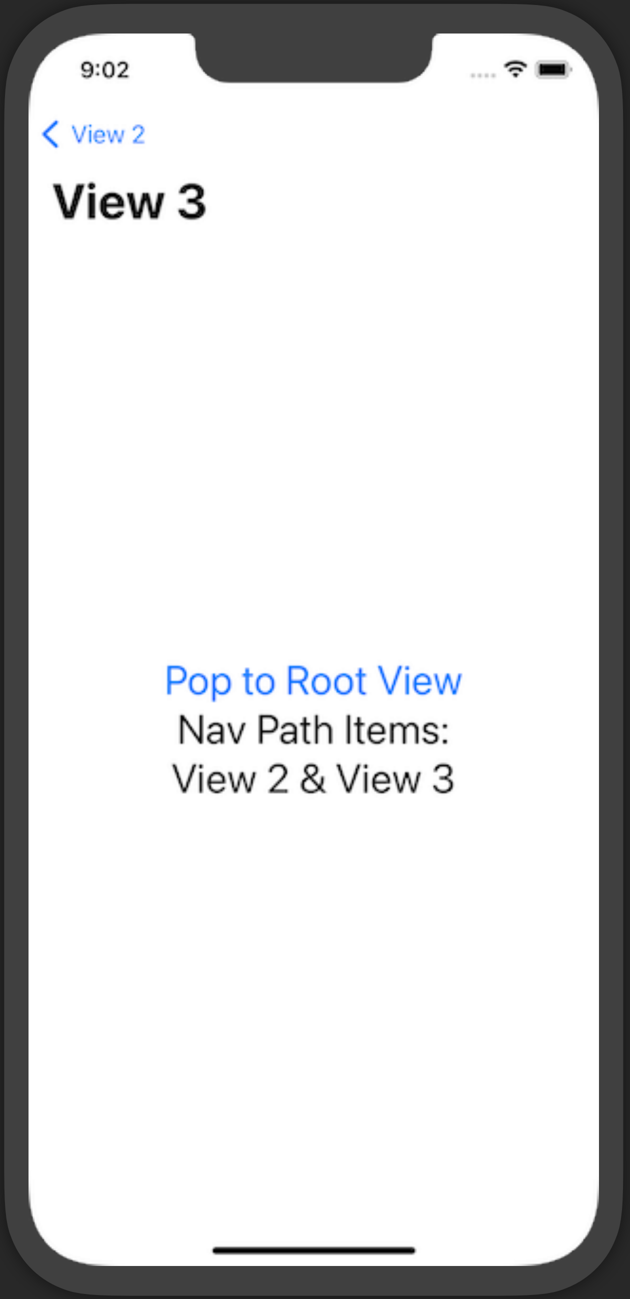
The NavigationPath can hold a diverse collection of data types.

Use this when you are not passing in the same type to all views you navigate to.

You will need a separate navigation destination modifier for each type.

Pop to Root

iOS 16



Nav Path Items:
View 2 & View 3

```
struct NavLink_PopToRoot: View {
    @State private var navPath: [String] = []

    var body: some View {
        NavigationStack(path: $navPath) {
            VStack {
                NavigationLink(value: "View 2") {
                    Text("Go to View 2")
                }
            }
            .navigationTitle("Pop to Root")
            .navigationDestination(for: String.self) { pathValue in
                if pathValue == "View 2" {
                    NavLinkView2(navPath: $navPath)
                } else {
                    NavLinkView3(navPath: $navPath)
                }
            }
        }
        .font(.title)
    }
}
```

Note: The term "**root**" refers to the first view of the navigation stack. "**Pop**" means to remove all other views off of the stack so it shows just the root view.

The key is to make sure your navigation path array (navPath in this example) is accessible to all destination views.

In this example, it is being passed into the destination views.

See next page for how the destination views pop back to the root (first) view.

Pop to Root (The other views)

iOS 16

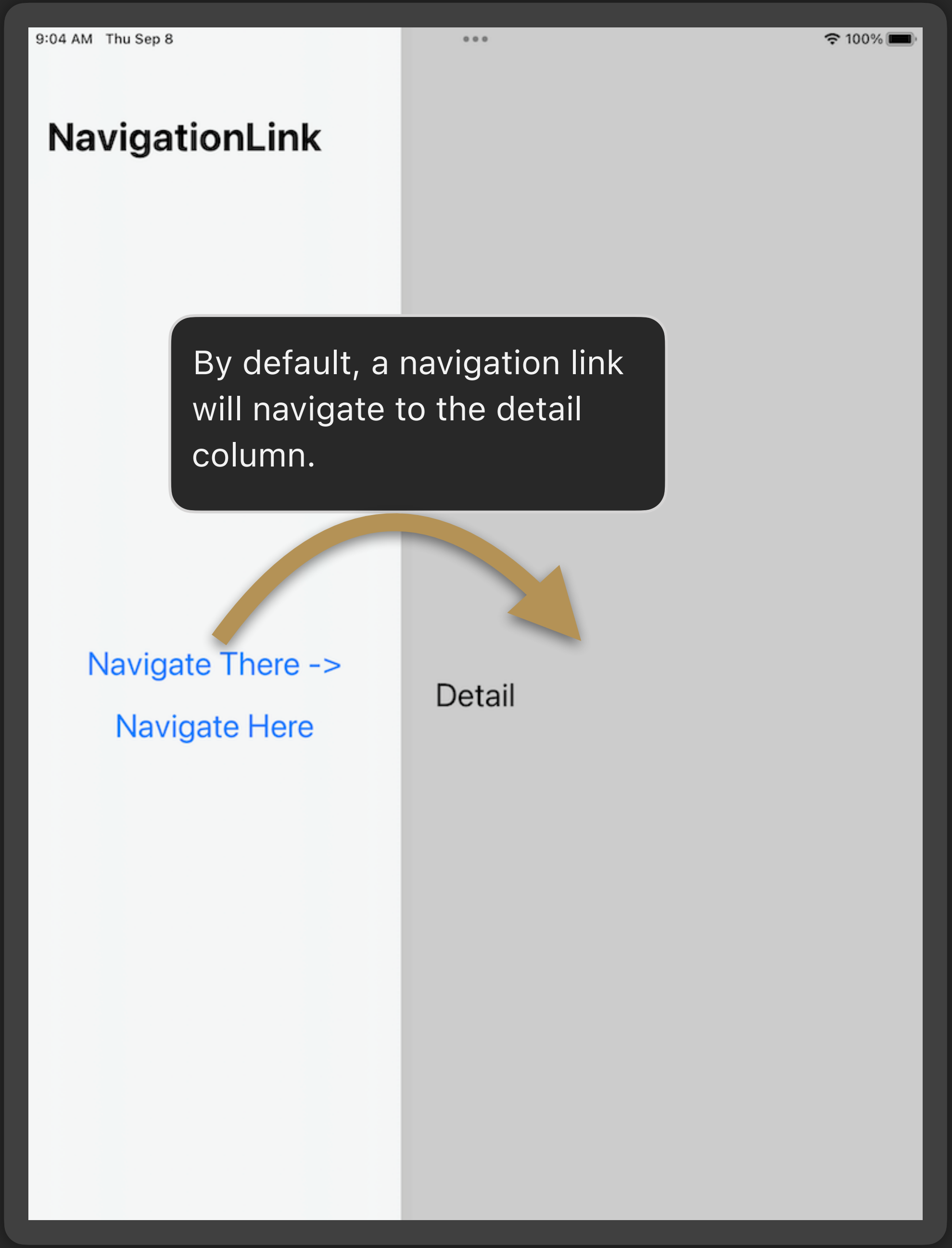
```
struct NavLinkView2: View {
    @Binding var navPath: [String]

    var body: some View {
        VStack(spacing: 20) {
            NavigationLink(value: "View 3") {
                Text("Go to View 3")
            }

            Text("Nav Path Items:")
            Text(navPath, format: .list(type: .and, width: .short))
        }
        .navigationTitle("View 2")
    }
}

struct NavLinkView3: View {
    @Binding var navPath: [String]

    var body: some View {
        VStack {
            Button("Pop to Root View") {
                navPath.removeAll()
            }
            Text("Nav Path Items:")
            Text(navPath, format: .list(type: .and, width: .short))
        }
        .navigationTitle("View 3")
    }
}
```

Make sure your destination views have access to your navigation path array.

This is where the magic happens. To pop to the root (first view), just remove all items from the navigation path array.

iOS 16

Detail Link



By default, a navigation link will navigate to the detail column.

```
struct NavLink_IsDetailLink: View {
    var body: some View {
        NavigationSplitView {
            VStack(spacing: 20) {
                NavigationLink("Navigate There ->") {
                    NavigationDestinationView()
                }

                NavigationLink("Navigate Here") {
                    NavigationDestinationView()
                }
                .isDetailLink(false) // Do not navigate to detail column
            }
            .navigationTitle("NavigationLink")
        } detail: {
            Text("Detail")
        }
        .font(.title)
    }
}

struct NavigationDestinationView: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Navigation Destination")
        }
        .navigationTitle("Destination")
        .font(.title)
    }
}
```

Use isDetailLink modifier to control where navigation happens.

If it is true (default) the destination will appear in the detail column.

If false, the destination will appear in the SAME column.

iOS 14

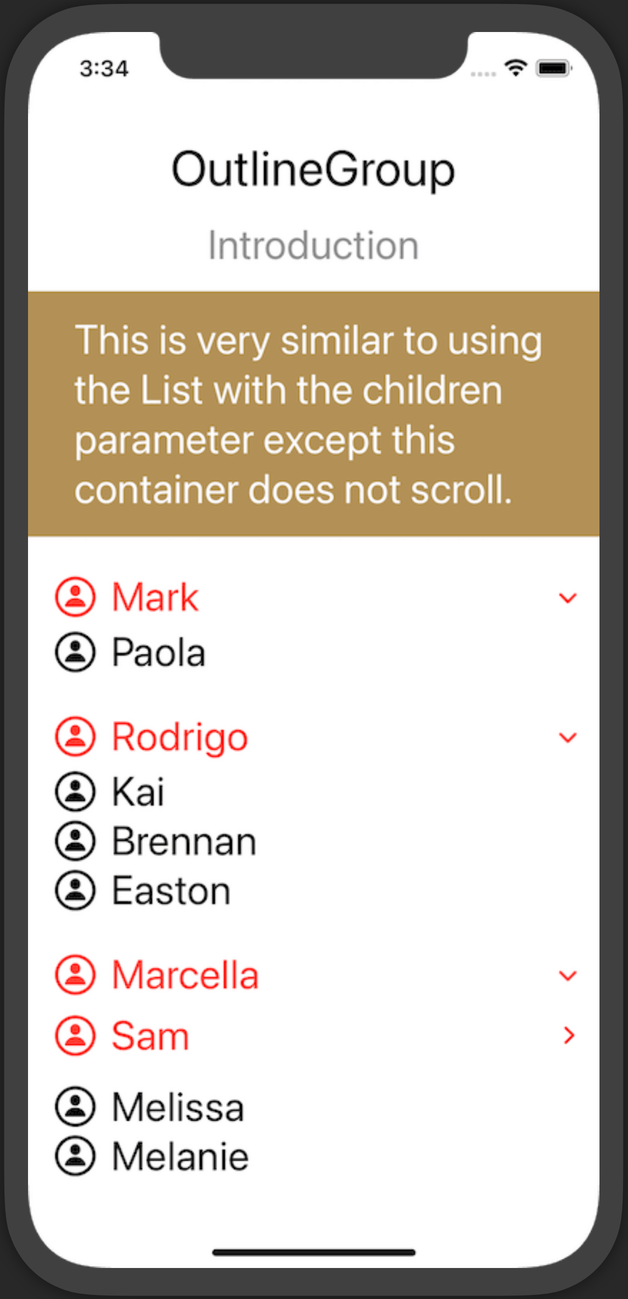
OutlineGroup

OutlineGroups gives you another way to present hierarchical data. It is very similar to using a List with the children parameter. Except this container view does not scroll. It's probably best for limited data.

This is a pull-in view.

Introduction

iOS 14



OutlineGroup

Introduction

This is very similar to using the List with the children parameter except this container does not scroll.

Mark
Paola
Rodrigo
Kai
Brennan
Easton
Marcella
Sam
Melissa
Melanie

```
struct OutlineGroup_Intro: View {
    var parents = [Parent(name: "Mark",
                          children: [Parent(name: "Paola")]),
                   Parent(name: "Rodrigo",
                          children: [Parent(name: "Kai"),
                                     Parent(name: "Brennan"),
                                     Parent(name: "Easton")]),
                   Parent(name: "Marcella",
                          children: [Parent(name: "Sam",
                                            children: [Parent(name: "Joe")]),
                                     Parent(name: "Melissa"),
                                     Parent(name: "Melanie")])]

    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("OutlineGroup",
                       subtitle: "Introduction",
                       desc: "This is very similar to using the List with the children parameter
                             except this container does not scroll.",
                        back: Color.red, textColor: .primary)

            OutlineGroup(parents, children: \.children) { parent in
                HStack {
                    Image(systemName: "person.circle")
                    Text("\(parent.name)")
                    Spacer()
                }
            }
            .padding(.horizontal)
            .tint(.red)
        }
        .font(.title)
    }
}
```

Parent struct on next page.

The parent rows that have children all take on the accent color.

```
// Need to conform to Identifiable
struct Parent: Identifiable {
    var id = UUID()
    var name = ""
    var children: [Parent]? = nil // Had to make this optional
}
```

You can just create an array that holds more of itself for the children data.

Note: Unlike the List with children, the child rows in an OutlineGroup do not indent.

If you want to indent child rows, you will probably have to add a property to your model that indicates if it is child data or level of indent.

iOS 14

OutlineGroup

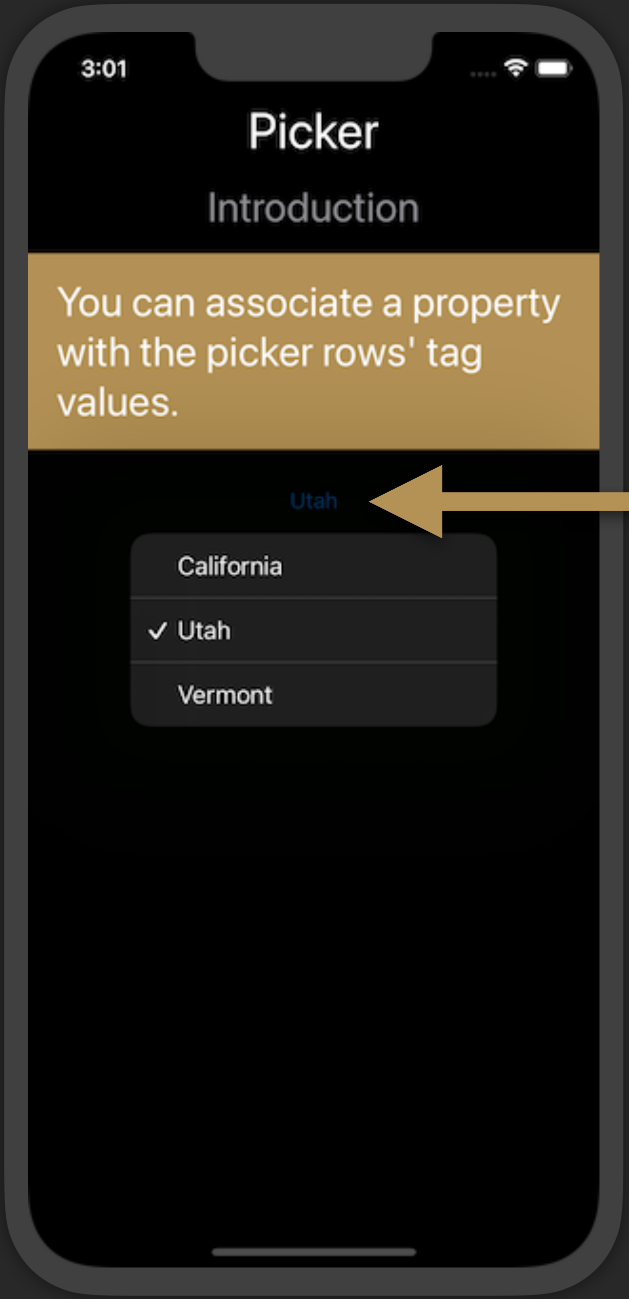
OutlineGroups gives you a [illegible] ay to present hierarchi[illegible] It is very similar to using a List with the children parameter. Except this conta[illegible] view does not scroll. It's probably best for limited data.

This is a pull-in view.

Picker

To get or set a value for the Picker, you need to bind it to a variable. This variable is then passed into the Picker's initializer. Then, all you need to do is change this bound variable's value to select the row you want to show in the Picker. Or read the bound variable's value to see which row is currently selected. One thing to note is that this variable is actually bound to the Picker row's **tag property** which you will see in the following pages.

Introduction


3:01
Picker
Introduction
You can associate a property with the picker rows' tag values.
Utah
California
✓ Utah
Vermont

```
struct Picker_Intro : View {
    @State private var favoriteState = 1

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Introduction",
                       desc: "You can associate a property with the picker rows' tag values.")

            Picker("States", selection: $favoriteState) {
                Text("California").tag(0)
                Text("Utah").tag(1)
                Text("Vermont").tag(2)
            }

            Spacer()
        }
        .font(.title)
    }
}
```

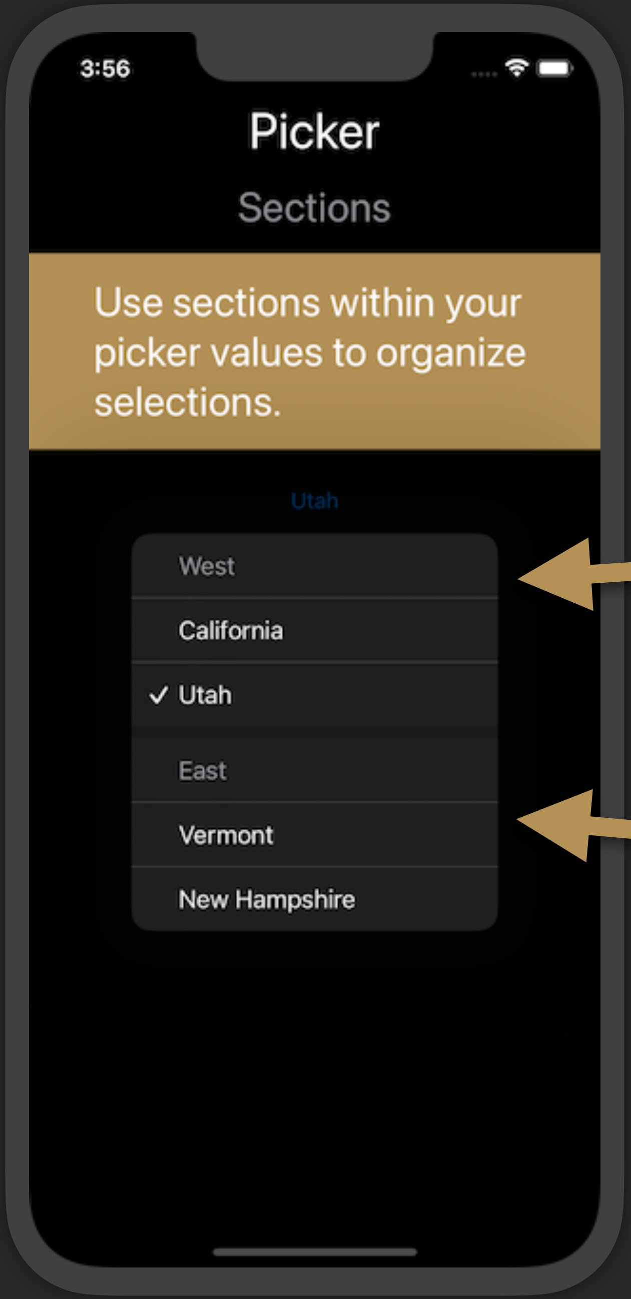
Starting in iOS 15, the default picker style will look like the menu style. (Before, the default was the wheel style.)

iOS 15

This picker is actually binding the tag values to the `favoriteState` property.

Note: Changing the font size does not seem to affect the picker's font size.

Sections


3:56

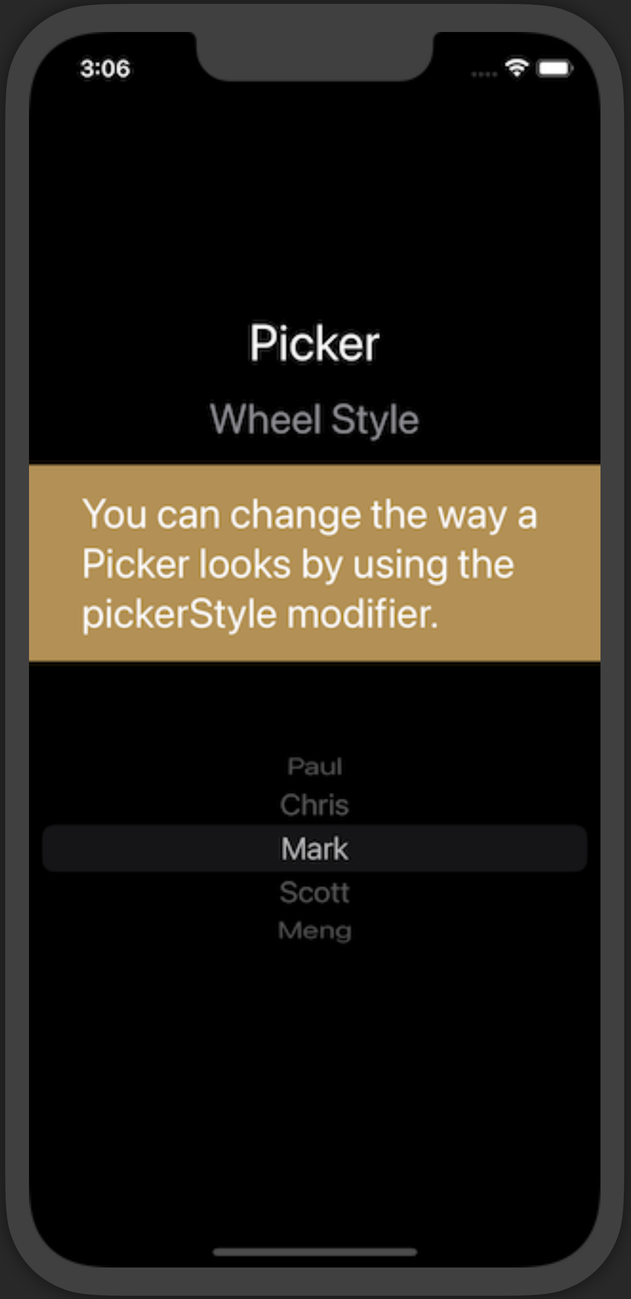
Picker

Sections

Use sections within your picker values to organize selections.

```
struct Picker_Sections: View {
    @State private var favoriteState = 1

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Sections",
                       desc: "Use sections within your picker values to organize selections.")

            Picker("States", selection: $favoriteState) {
                Section {
                    Text("California").tag(0)
                    Text("Utah").tag(1)
                } header: {
                    Text("West")
                }

                Section {
                    Text("Vermont").tag(2)
                    Text("New Hampshire").tag(3)
                } header: {
                    Text("East")
                }
            }

            Spacer()
        }
        .font(.title)
    }
}
```

Wheel Style



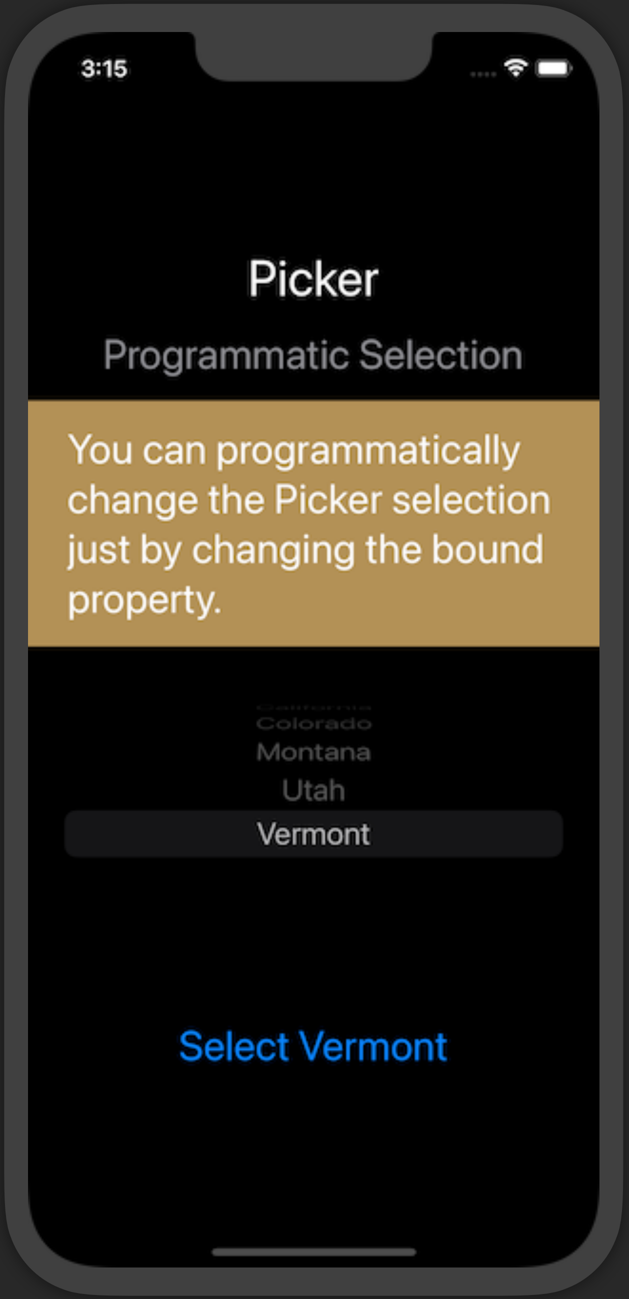
You can change the way a Picker looks by using the pickerStyle modifier.

```
struct Picker_Wheel: View {
    @State private var yourName = "Mark"

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Wheel Style",
                       desc: "You can change the way a Picker looks by using the pickerStyle
                            modifier.")

            Picker("Name", selection: $yourName) {
                Text("Paul").tag("Paul")
                Text("Chris").tag("Chris")
                Text("Mark").tag("Mark")
                Text("Scott").tag("Scott")
                Text("Meng").tag("Meng")
            }
            .pickerStyle(.wheel)
        }
        .font(.title)
    }
}
```

This will be the default value selected in the Picker.

Set the style right on the Picker.

Programmatic Selection

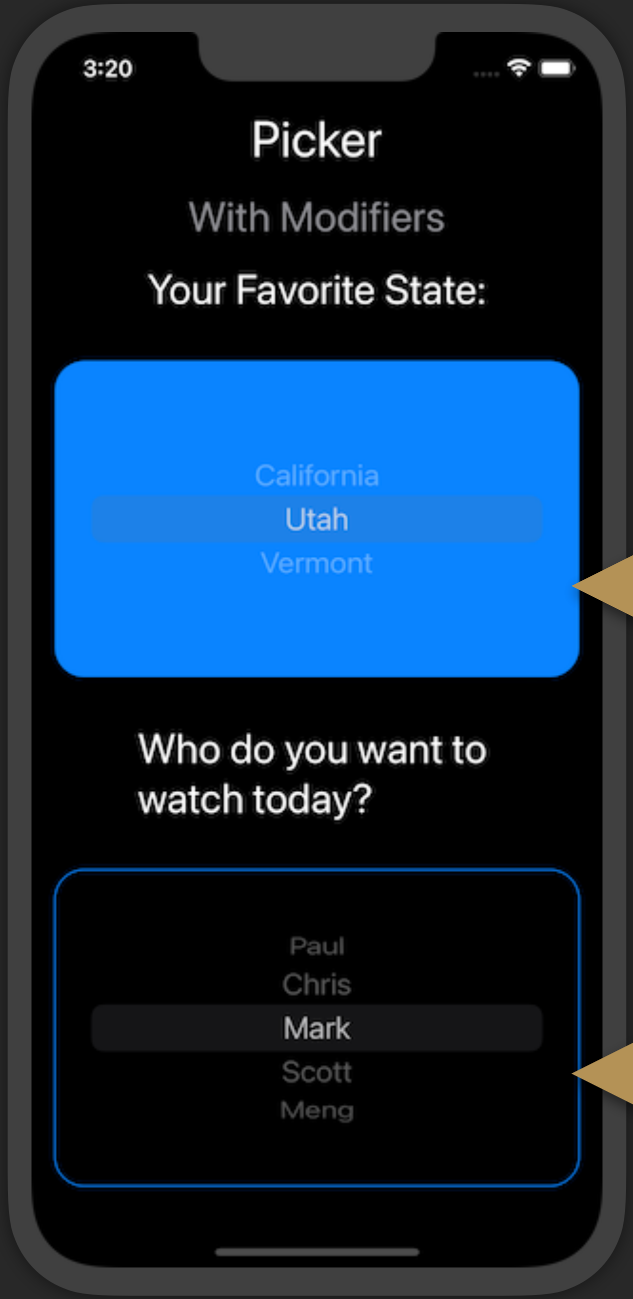

3:15
Picker
Programmatic Selection
You can programmatically change the Picker selection just by changing the bound property.
California
Colorado
Montana
Utah
Vermont
Select Vermont

```swift
struct Picker_ProgrammaticSelection: View {
    @State private var favoriteState = 1

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Programmatic Selection",
                       desc: "You can programmatically change the Picker selection just by
                             changing the bound property.")

            Picker("States", selection: $favoriteState) {
                Text("California").tag(0)
                Text("Colorado").tag(1)
                Text("Montana").tag(2)
                Text("Utah").tag(3)
                Text("Vermont").tag(4)
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)

            Button("Select Vermont") {
                withAnimation {
                    favoriteState = 4
                }
            }
        }
        .font(.title)
    }
}
```

When you change the Picker's bound property value, the Picker then updates and selects the matching row.

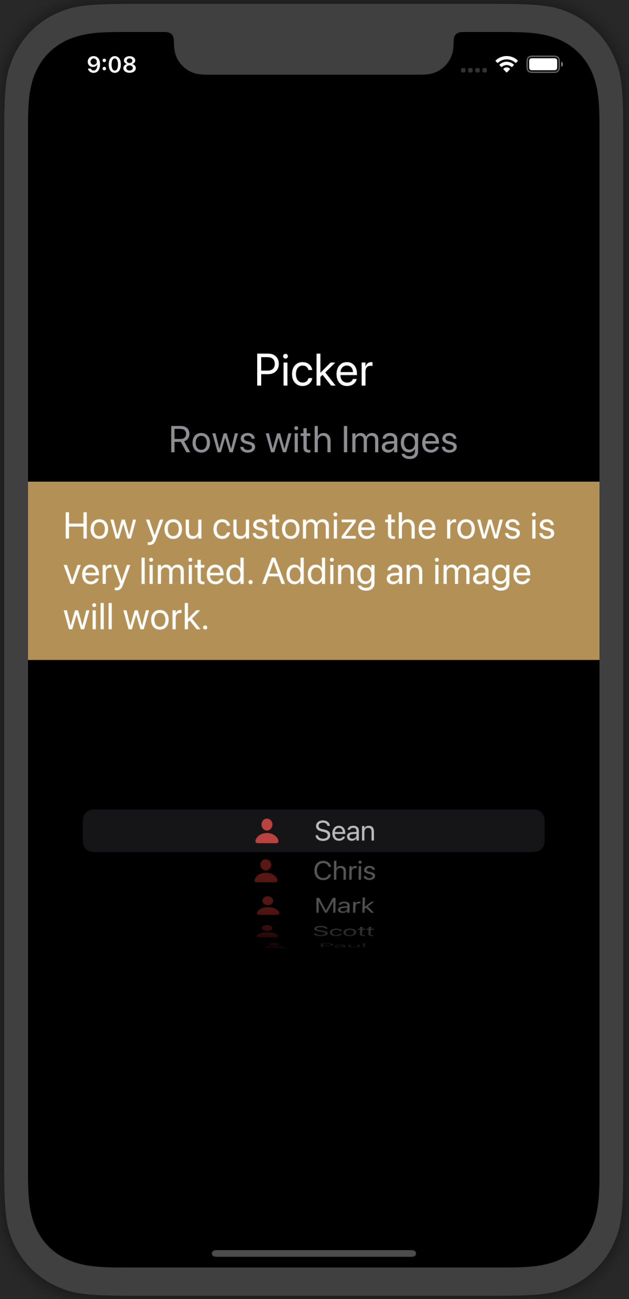
Note: I added withAnimation so you see the wheel actually spin.

Customized



```
struct Picker_Customized : View {
    @State private var favoriteState = 1
    @State private var youTuberName = "Mark"

    var body: some View {
        VStack(spacing: 16) {
            Text("Picker").font(.largeTitle)
            Text("With Modifiers").foregroundColor(.gray)
            Text("Your Favorite State:")
            Picker("Select State", selection: $favoriteState) {
                Text("California").tag(0)
                Text("Utah").tag(1)
                Text("Vermont").tag(2)
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)
            .background(RoundedRectangle(cornerRadius: 20)
                            .fill(Color.blue))
            .padding()

            Text("Who do you want to watch today?")
            Picker("Select person", selection: $youTuberName) {
                Text("Paul").tag("Paul")
                Text("Chris").tag("Chris")
                Text("Mark").tag("Mark")
                Text("Scott").tag("Scott")
                Text("Meng").tag("Meng")
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)
            .background(RoundedRectangle(cornerRadius: 20)
                            .stroke(Color.blue, lineWidth: 1))
            .padding()
        }
        .font(.title)
    }
}
```

Rows with Images

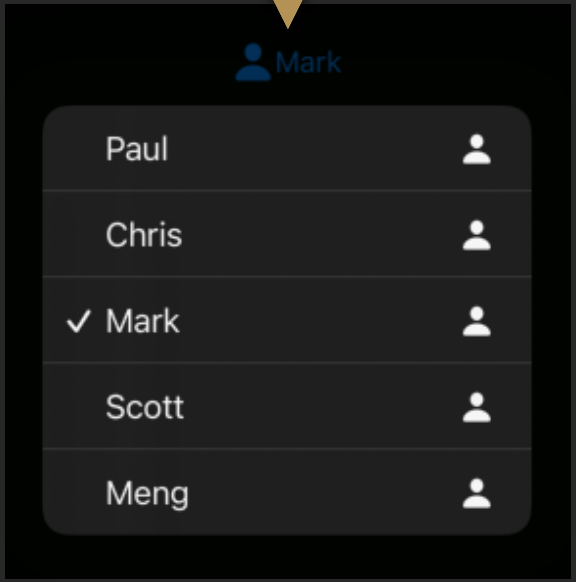


How you customize the rows is very limited. Adding an image will work.

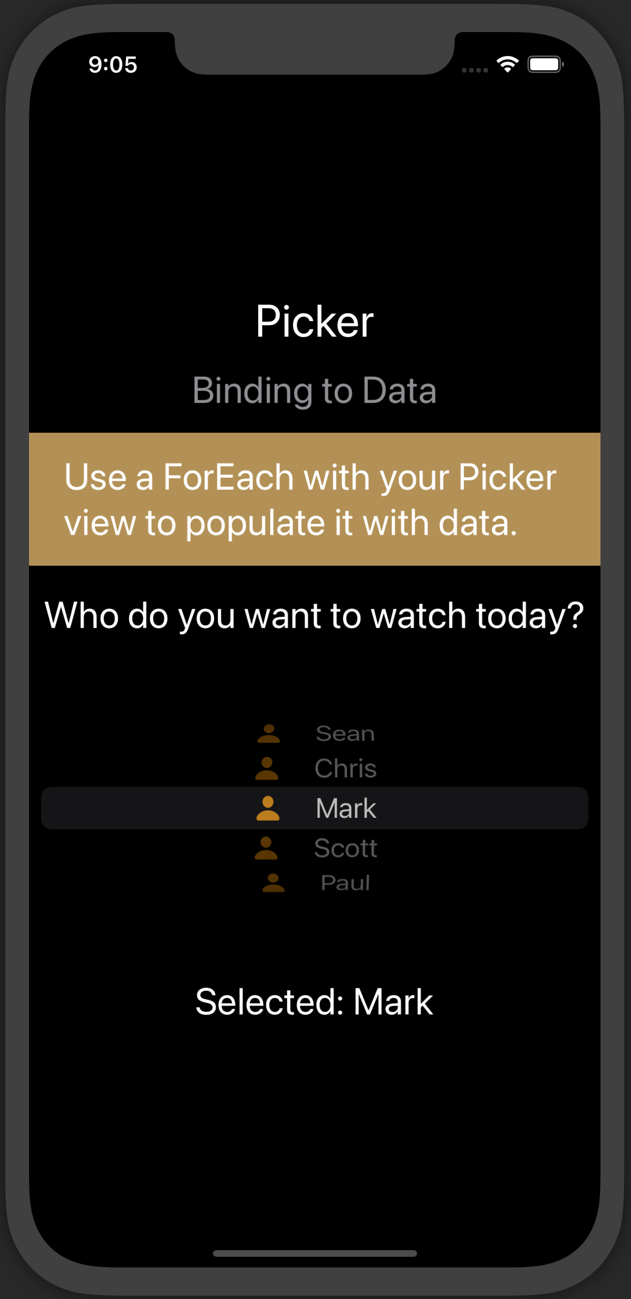
```
struct Picker_RowsWithImages : View {
    @State private var youTuberName = "Mark"

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Rows with Images",
                       desc: "Row customization is limited. Adding an image will work.")
            Picker(selection: $youTuberName, label: Text("")) {
                Row(name: "Paul").tag("Paul")
                Row(name: "Chris").tag("Chris")
                Row(name: "Mark").tag("Mark")
                Row(name: "Scott").tag("Scott")
                Row(name: "Meng").tag("Meng")
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

```
fileprivate struct Row : View {
    var name: String

    var body: some View {
        return HStack {
            Image(systemName: "person.fill")
                .padding(.trailing)
                .foregroundColor(Color.red)
            Text(name)
        }
    }
}
```

When not using the wheel picker style, the picker looks like this with the image.



Binding Rows to Data


9:05

Picker

Binding to Data

Use a ForEach with your Picker view to populate it with data.

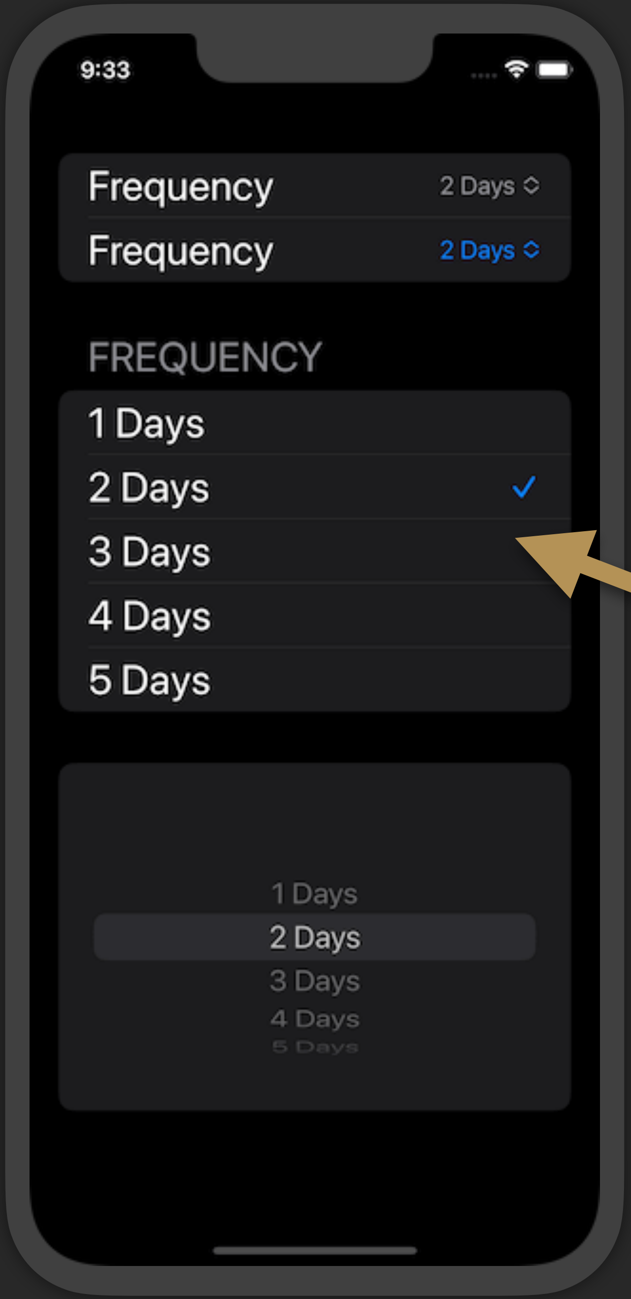
Who do you want to watch today?

Sean
Chris
Mark
Scott
Paul

Selected: Mark

```
struct Picker_BindingToData : View {
    @State private var youTuberName = "Mark"
    var youTubers = ["Sean", "Chris", "Mark", "Scott", "Paul"]

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Binding to Data",
                       desc: "Use a ForEach with your Picker view to populate it with data.")

            Text("Who do you want to watch today?")

            Picker(selection: $youTuberName, label: Text("")) {
                ForEach(youTubers, id: \.self) { name in
                    Row(name: name)
                }
            }
            .pickerStyle(.wheel)

            Text("Selected: \(youTuberName)")
        }
        .font(.title)
    }
}

fileprivate struct Row : View {
    var name: String
    var body: some View {
        HStack {
            Image(systemName: "person.fill")
                .padding(.trailing)
                .foregroundColor(Color.orange)
            Text(name)
        }
    }
}
```

Pickers in Forms



```swift
struct Picker_InForm: View {
    @State private var selectedDaysOption = "2"
    var numberOfDaysOptions = ["1", "2", "3", "4", "5"]

    var body: some View {
        VStack {
            Form {
                Picker("Frequency", selection: $selectedDaysOption) {
                    ForEach(numberOfDaysOptions, id: \.self) {
                        Text("\($0) Days").tag($0)
                    }
                }
                Picker("Frequency", selection: $selectedDaysOption) {
                    ForEach(numberOfDaysOptions, id: \.self) {
                        Text("\($0) Days").tag($0)
                    }
                }
                .pickerStyle(.menu) // Add this modifier to make it use the accent color

                Picker("Frequency", selection: $selectedDaysOption) {
                    ForEach(numberOfDaysOptions, id: \.self) {
                        Text("\($0) Days").tag($0)
                    }
                }
                .pickerStyle(.inline)

                Picker("Frequency", selection: $selectedDaysOption) {
                    ForEach(numberOfDaysOptions, id: \.self) {
                        Text("\($0) Days").tag($0)
                    }
                }
                .pickerStyle(.wheel)
            }
            .navigationTitle("Picker")
        }
        .font(.title)
    }
}
```

Picker styles can alter the appearance of a picker when inside a form. Here are some of the different options.

Notice the inline style created its own section within the form.

iOS 16

PhotosPicker

The photos picker first appears a button on your UI. When tapped it will give the user access to the photos library. You can programmatically filter what you want them to see, such as just videos or time-lapse videos.

The photos picker view (the

iOS 14

ProgressView

The progress view gives y[illegible] different ways to show [illegible] that something is currently happening and optionally give you a way to show the progression of some activity.

This is a pull-in view in s[illegible]

iOS 16

RenameButton

The rename button doesn't d by itself. modifier to implement some code when the button is tapped.

The button can be formatted just like the SwiftUI Button can.

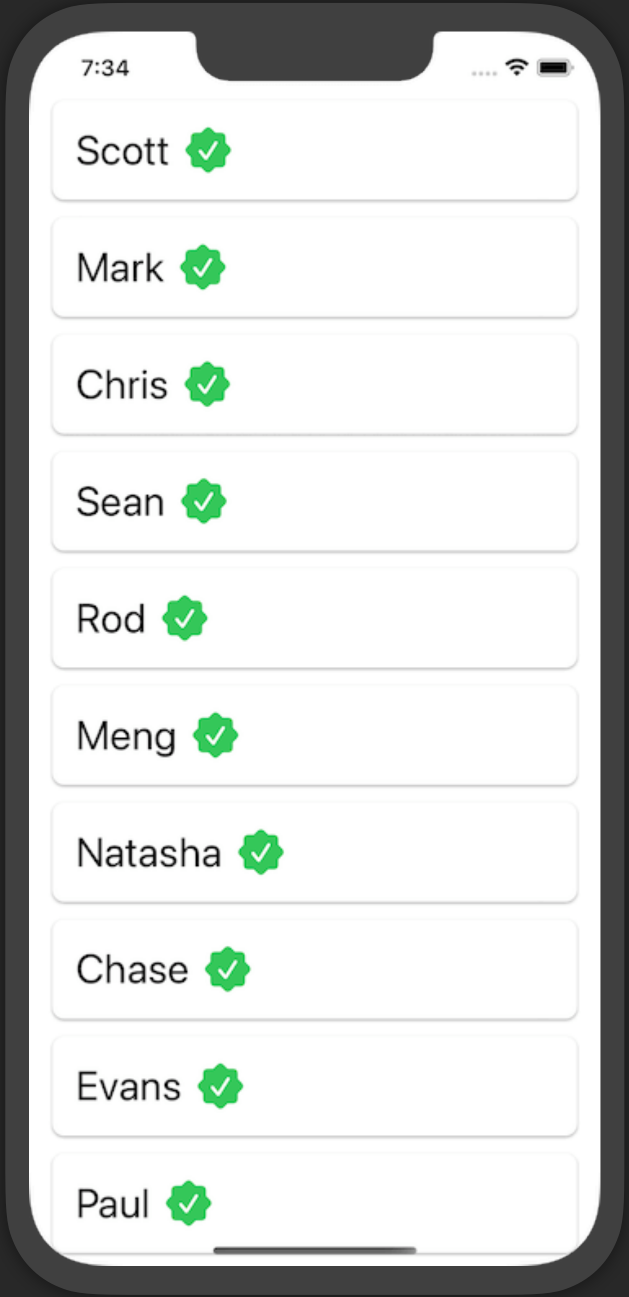
This is a pull-in view.

ScrollView

A ScrollView is like a container for child views. When the child views within the ScrollView go outside the frame, the user can scroll to bring the child views that are outside the frame into view.

A ScrollView is a push-out view in the scroll direction you specify. You can set the direction of a ScrollView to be vertical or horizontal.

Introduction



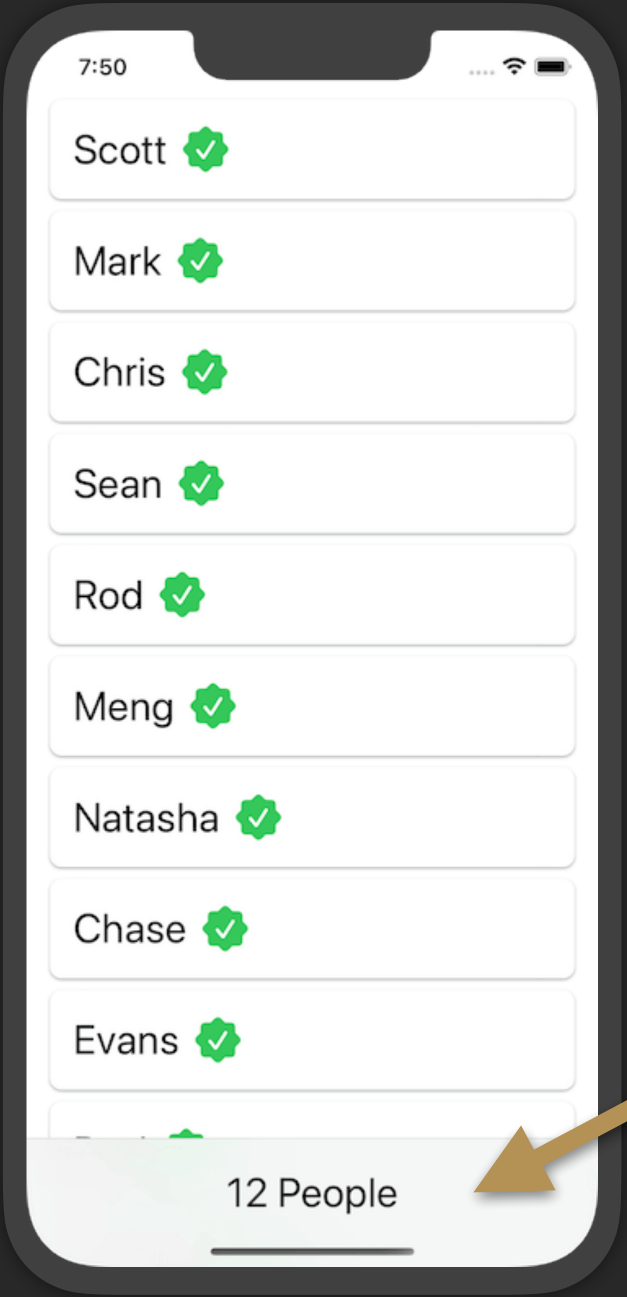
```
struct Scrollview_Intro : View {
    @State private var names = ["Scott", "Mark", "Chris", "Sean", "Rod", "Meng", "Natasha",
"Chase", "Evans", "Paul", "Durtschi", "Max"]
    var body: some View {
        ScrollView {
            ForEach(names, id: \.self) { name in
                HStack {
                    Text(name).foregroundStyle(.primary)
                    Image(systemName: "checkmark.seal.fill")
                        .foregroundStyle(.green)
                    Spacer()
                }
                .padding()
                .background(Color.white.shadow(.drop(radius: 1, y: 1)),
                            in: RoundedRectangle(cornerRadius: 8))
                .padding(.bottom, 4)
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

Wrap the ForEach in a ScrollView.

A Scrollview with a ForEach view is similar to a List. But be warned, the rows are not reusable. It is best to limit the number of rows for memory and performance considerations.

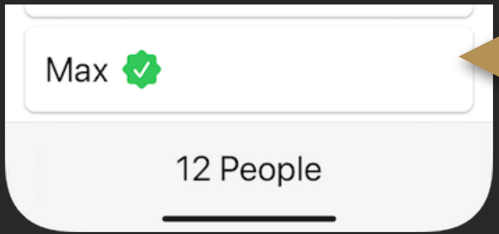
SafeAreaInset

iOS 15



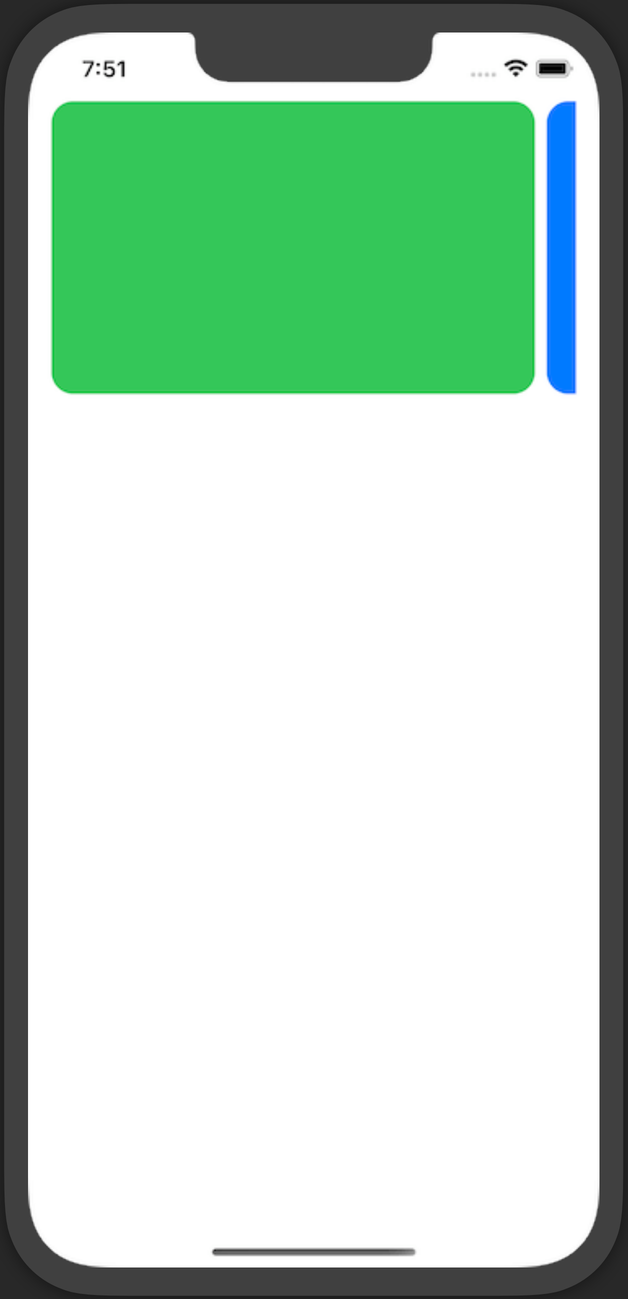
```swift
struct ScrollView_SafeAreaInset: View {
    @State private var names = ["Scott", "Mark", "Chris", "Sean", "Rod", "Meng", "Natasha",
"Chase", "Evans", "Paul", "Durtschi", "Max"]
    var body: some View {
        ScrollView {
            ForEach(names, id: \.self) { name in
                HStack {
                    Text(name)
                        .foregroundStyle(.primary)
                    Image(systemName: "checkmark.seal.fill")
                        .foregroundStyle(.green)
                    Spacer()
                }
                .padding()
                .background(Color.white.shadow(.drop(radius: 1, y: 1)),
                            in: RoundedRectangle(cornerRadius: 8))
            }
            .padding(.horizontal)
        }
        .safeAreaInset(edge: .bottom) {
            VStack(spacing: 20) {
                Divider()
                Text("12 People")
            }
            .background(.regularMaterial)
        }
        .font(.title)
    }
}
```

The rows will scroll behind the safe area inset but when you get to the bottom, the last row will show completely outside of it.



Last row

Scroll Horizontally



```
struct Scrollview_Horizontal : View {
    var items = [Color.green, Color.blue, Color.purple, Color.pink,
                 Color.yellow, Color.orange]

    var body: some View {
        GeometryReader { gr in
            ScrollView(Axis.Set.horizontal, showsIndicators: true) {
                HStack {
                    ForEach(items, id: \.self) { item in
                        RoundedRectangle(cornerRadius: 15)
                            .fill(item)
                            .frame(width: gr.size.width - 60,
                                   height: 200)
                    }
                }
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

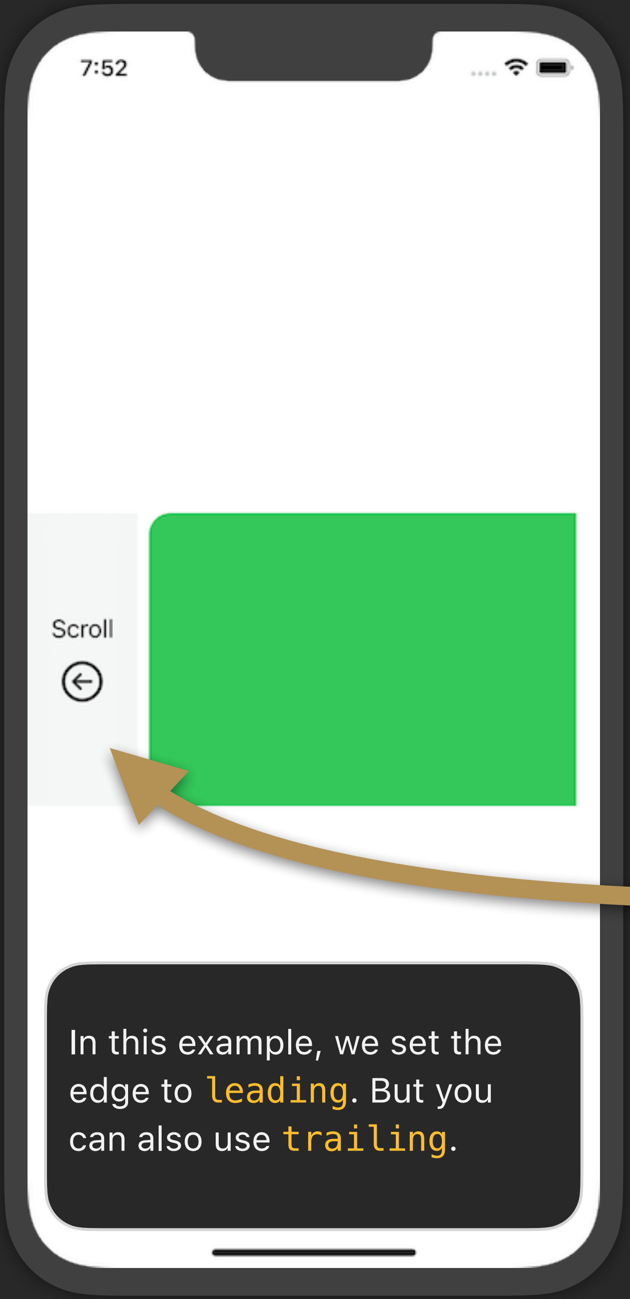
For horizontal scrolling, set the scroll direction to horizontal with an HStack. If the contents extend horizontally beyond the screen's frame, then scrolling will be enabled.

You can also set if you want to show the scroll indicators or not.

The geometry reader is used to always make the width of the view a little smaller than the screen's width so you can see the second view going off the screen.

Leading SafeAreaInset

iOS 15



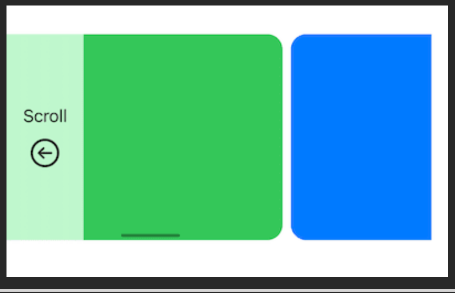
In this example, we set the edge to `leading`. But you can also use `trailing`.

```
struct ScrollView_Horizontal_SafeAreaInset: View {
    var items = [Color.green, Color.blue, Color.purple, Color.pink,
                 Color.yellow, Color.orange]

    var body: some View {
        GeometryReader { gr in
            VStack {
                Spacer()
                ScrollView(Axis.Set.horizontal, showsIndicators: true) {
                    HStack {
                        ForEach(items, id: \.self) { item in
                            RoundedRectangle(cornerRadius: 15)
                                .fill(item)
                                .frame(width: gr.size.width - 60)
                        }
                    }
                }
                .padding(.trailing)
                .safeAreaInset(edge: .leading) {
                    VStack(spacing: 10) {
                        Text("Scroll")
                            .font(.body)
                        Image(systemName: "arrow.left.circle")
                    }
                    .frame(maxHeight: .infinity)
                    .padding(.horizontal)
                    .background(.regularMaterial)
                }
                .frame(height: 200)
                Spacer()
            }
        }
        .font(.title)
    }
}
```

Add the safeAreaInset to a ScrollView to show additional views that will inset the ScrollView's content.

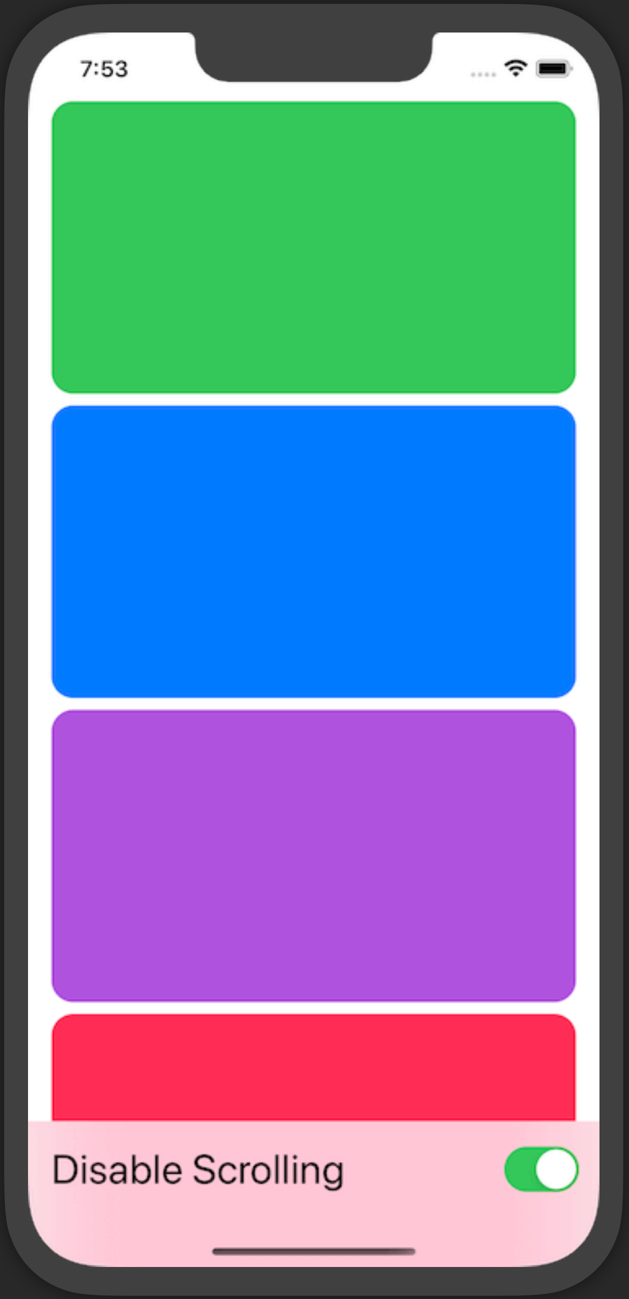
The first item in the ScrollView will be moved just enough to start next to the safeAreaInset view.

When you start scrolling, the contents of the ScrollView will go below your safeAreaInset view:



Disable Scrolling

iOS 16



```
struct Scrollview_Disabled: View {
    @State private var disableScroll = false

    var items = [Color.green, Color.blue, Color.purple, Color.pink,
                 Color.yellow, Color.orange]

    var body: some View {
        ScrollView(showsIndicators: true) {
            ForEach(items, id: \.self) { item in
                RoundedRectangle(cornerRadius: 15)
                    .fill(item)
                    .frame(height: 200)
                    .padding(.horizontal)
            }
        }
        .scrollDisabled(disableScroll)
        .safeAreaInset(edge: .bottom) {
            Toggle("Disable Scrolling", isOn: $disableScroll)
                .padding()
                .background(.regularMaterial)
        }
        .font(.title)
    }
}
```

You can disable scrolling with a modifier that passes in a boolean.

iOS 15

Searchable

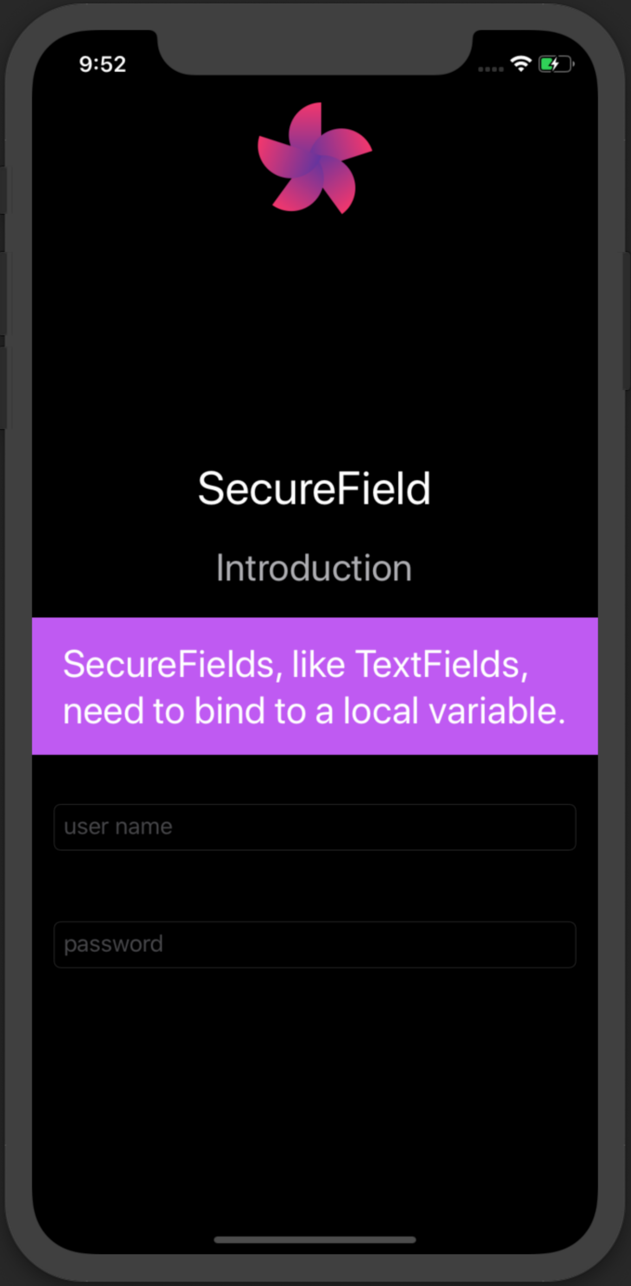
In iOS, the searchable modifier adds a text field to the NavigationView. Using it without a navigation view will show nothing. Use this modifier for situations where you want to give users the ability to either **search** for some data or **filter** an existing set of data already show

SecureField

In order to get or set the text in a SecureField, you need to bind it to a variable. This variable is passed into the SecureField's initializer. Then, all you need to do is change this bound variable's text to change what is in the SecureField. Or read the bound variable's value to see what text is currently in the SecureField.

This is a pull-in control.

Introduction


9:52

SecureField

Introduction

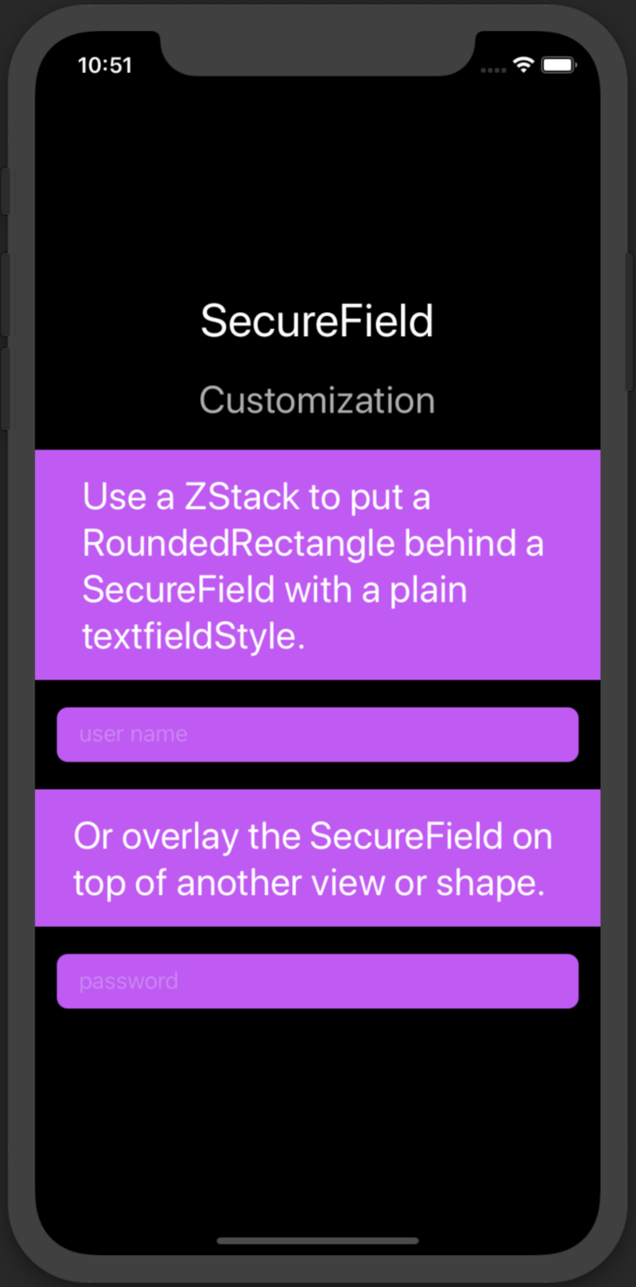
SecureFields, like TextFields, need to bind to a local variable.

```
@State private var userName = ""
@State private var password = ""
...
VStack(spacing: 20) {
    Image("Logo")

    Spacer()

    Text("SecureField")
        .font(.largeTitle)

    Text("Introduction")
        .font(.title)
        .foregroundColor(.gray)

    Text("SecureFields, like TextFields, need to bind to a local variable.")
        ...

    TextField("user name", text: $userName)
        .textFieldStyle(.roundedBorder)
        .padding()

    SecureField("password", text: $password)
        .textFieldStyle(.roundedBorder)
        .padding()

    Spacer()
}
```

For textFieldStyle, use:

< iOS 15	`RoundedBorderTextFieldStyle()`
iOS 15+	`.roundedBorder`

Customizations

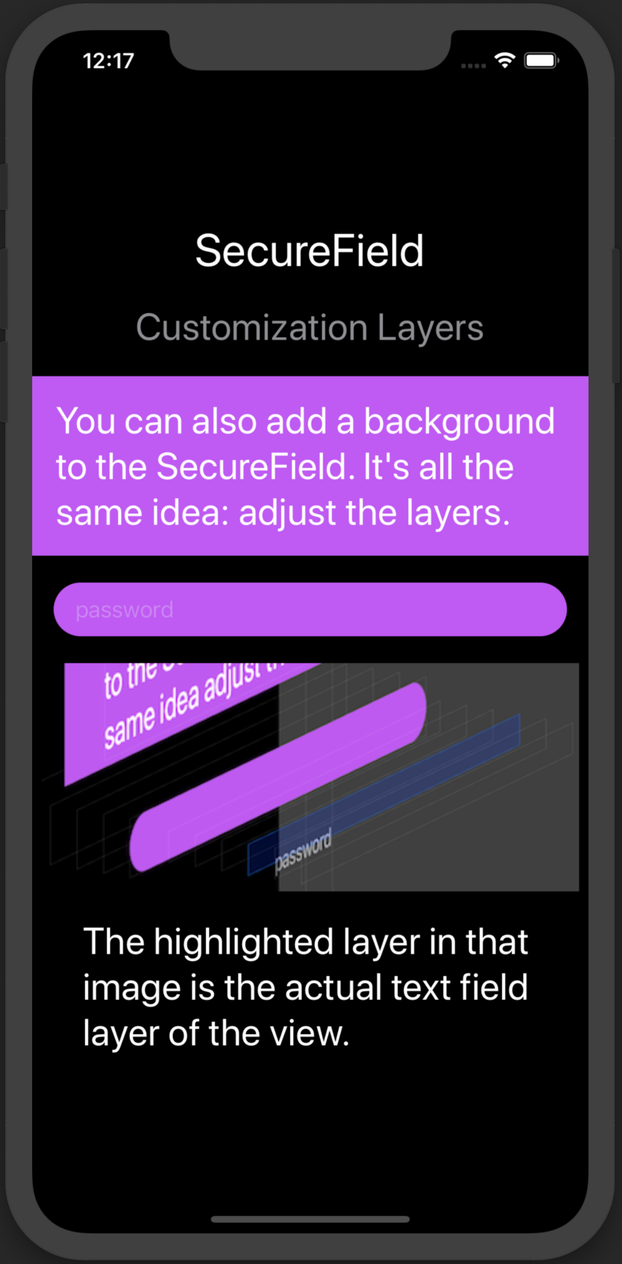


Use a ZStack to put a RoundedRectangle behind a SecureField with a plain textfieldStyle.

Or overlay the SecureField on top of another view or shape.

```
@State private var userName = ""
@State private var password = ""

...

Text("Use a ZStack to put a RoundedRectangle behind a SecureField with a plain textfieldStyle.")
    ...

ZStack{
    RoundedRectangle(cornerRadius: 8)
        .foregroundColor(.purple)
    TextField("user name", text: $userName)
        .foregroundColor(Color.white)
        .padding(.horizontal)
}
.frame(height: 40)
.padding(.horizontal)

Text("Or overlay the SecureField on top of another view or shape.")
    ...

RoundedRectangle(cornerRadius: 8)
    .foregroundColor(.purple)
    .overlay(
        SecureField("password", text: $password)
            .foregroundColor(Color.white)
            .padding(.horizontal)
    )
    .frame(height: 40)
    .padding(.horizontal)
```

Customization Layers


12:17

SecureField

Customization Layers

You can also add a background to the SecureField. It's all the same idea: adjust the layers.

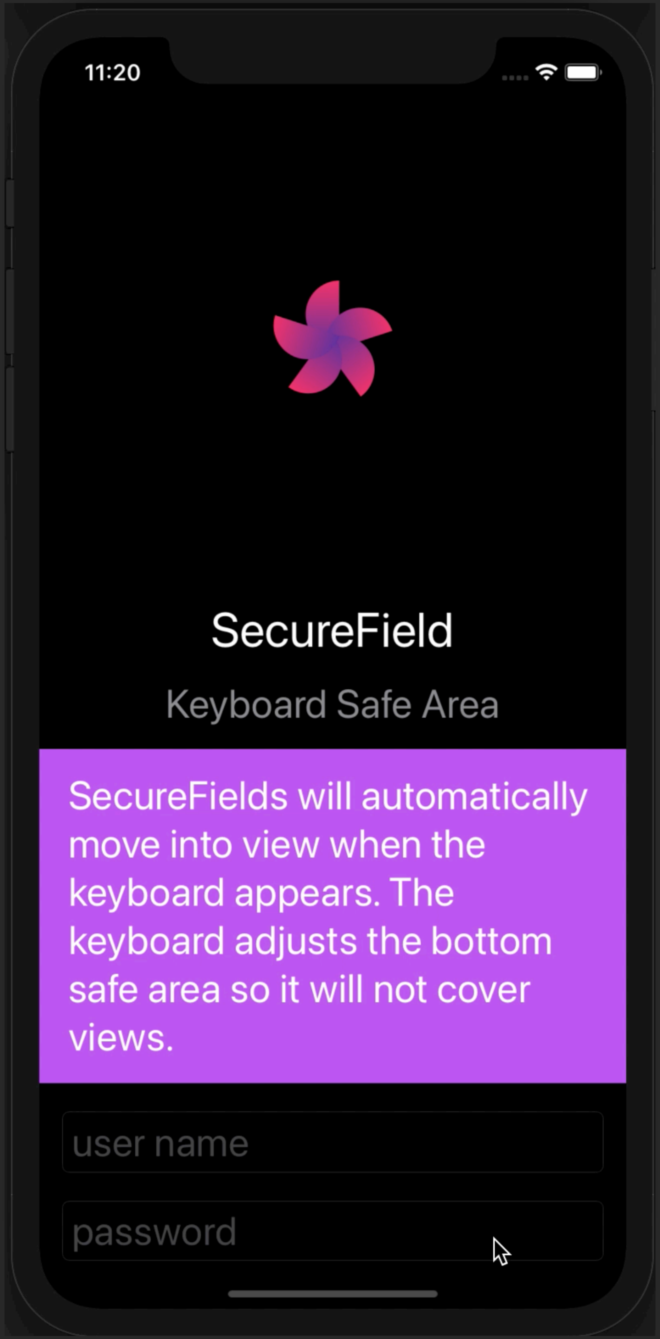
password

The highlighted layer in that image is the actual text field layer of the view.

```
@State private var userName = ""
@State private var password = ""
...
VStack(spacing: 20) {
    Text("SecureField")
        .font(.largeTitle)
    Text("Customization Layers")
        .font(.title)
        .foregroundColor(.gray)
    Text("You can also add a background to the SecureField. It's all the same idea: adjust the 
layers.")
        ...

    SecureField("password", text: $password)
        .foregroundColor(Color.white)
        .frame(height: 40)
        .padding(.horizontal)
        .background(
            Capsule()
                .foregroundColor(.purple)
        )
        .padding(.horizontal)

    Image("SecureFieldLayers")

    Text("The highlighted layer in that image is the actual text field layer of the view.")
        .font(.title)
        .padding(.horizontal)
}
```

Keyboard Safe Area

iOS 14



SecureField

Keyboard Safe Area

SecureFields will automatically move into view when the keyboard appears. The keyboard adjusts the bottom safe area so it will not cover views.

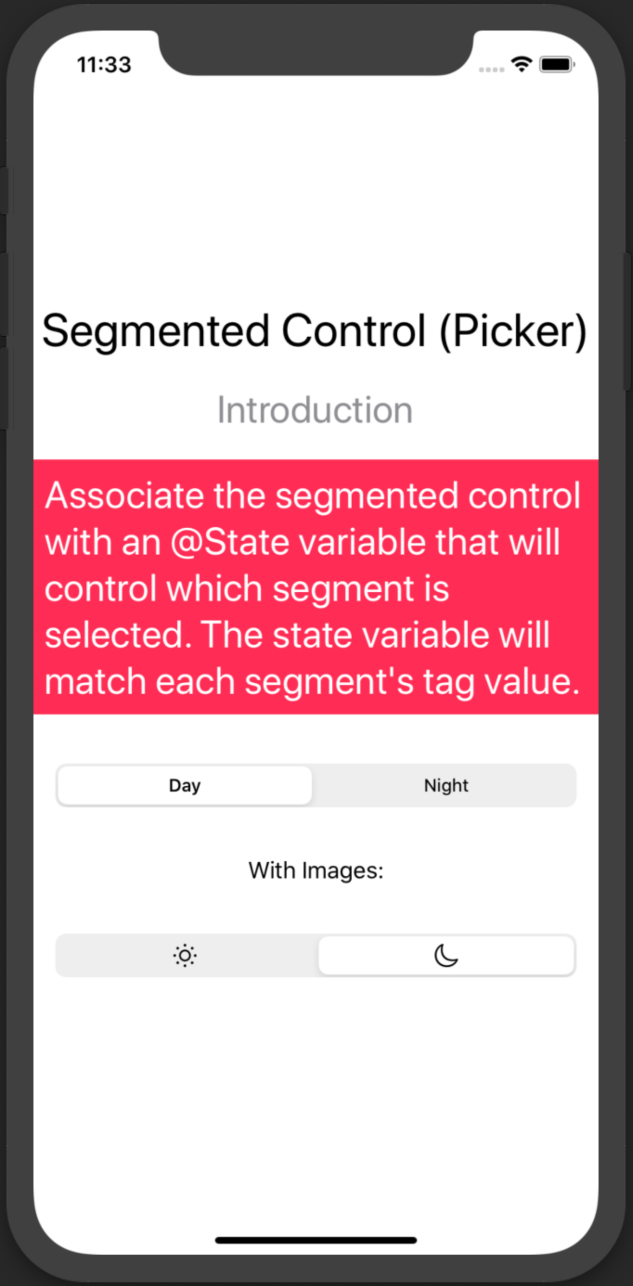
```
struct SecureField_KeyboardSafeArea: View {
    @State private var userName = ""
    @State private var password = ""

    var body: some View {
        VStack(spacing: 20) {
            Spacer()
            Image("Logo")
            Spacer()

            HeaderView("SecureField",
                       subtitle: "Keyboard Safe Area",
                       desc: "SecureFields will automatically move into view when the keyboard
appears. The keyboard adjusts the bottom safe area so it will not cover views.",
                       back: .purple, textColor: .white)

            TextField("user name", text: $userName)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            SecureField("password", text: $password)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)
        }
        .font(.title)
    }
}
```

Segmented Control (Picker)

Segmented controls are now Picker controls with a different picker style set. In order to get or set the selected segment, you need to bind it to a variable. This variable is passed into the segmented control's (Picker's) initializer. Then, all you need to do is change this bound variable's value to change the selected segment. Or read the bound variable's value to see which segment is currently selected.

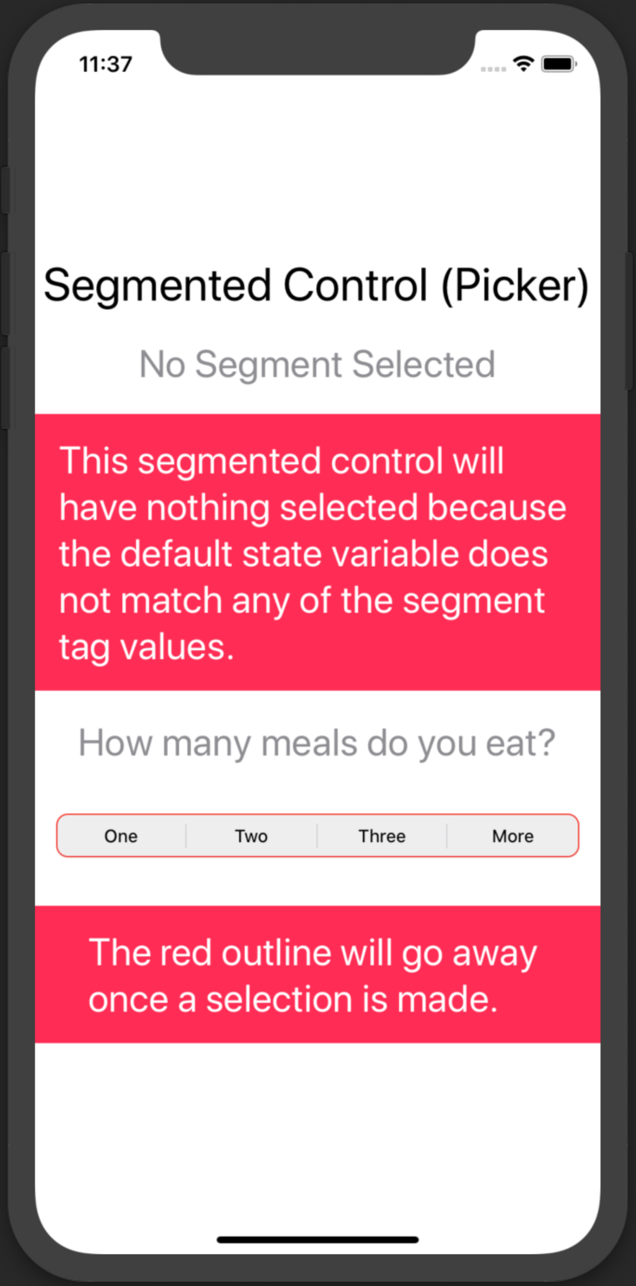
This is a pull-in view.

Introduction


11:33
Segmented Control (Picker)
Introduction
Associate the segmented control with an @State variable that will control which segment is selected. The state variable will match each segment's tag value.
Day
Night
With Images:

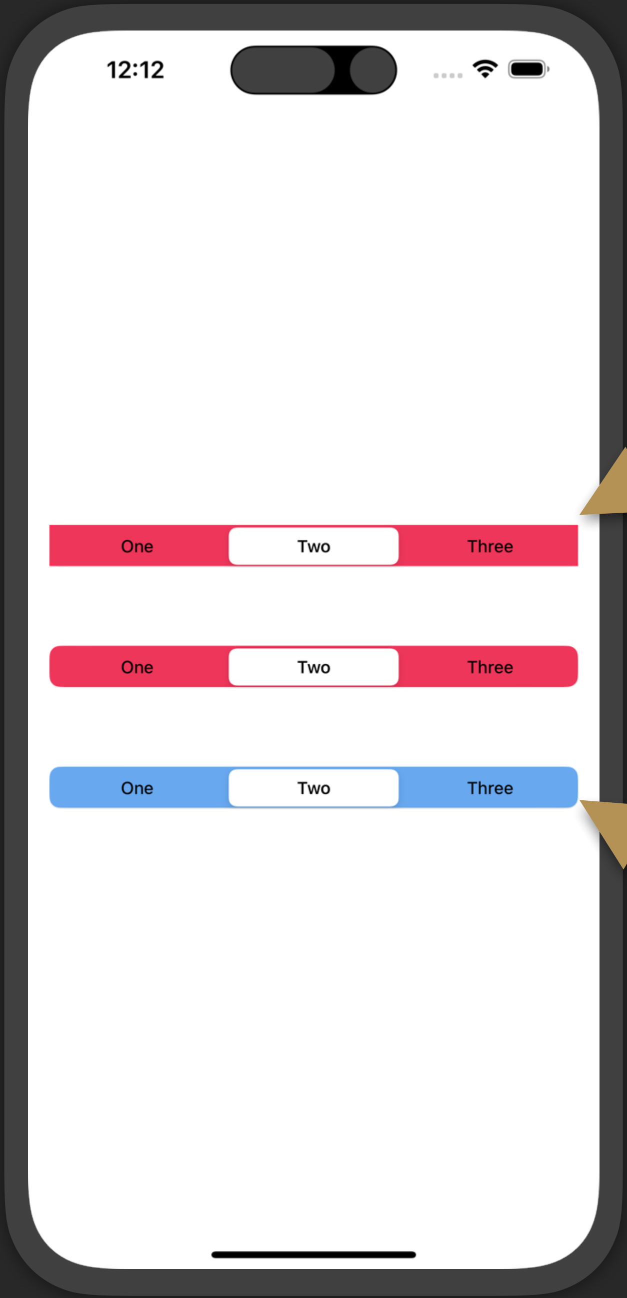
```swift
@State private var dayNight = "day"
@State private var tab = 1

...

VStack(spacing: 20) {
    Text("Segmented Control (Picker)").font(.largeTitle)
    Text("Introduction")
        .font(.title).foregroundColor(.gray)
    Text("Associate the segmented control with an @State variable that will control which
segment is selected. The state variable will match each segment's tag value.")
        ...

    Picker("", selection: $dayNight) {
        Text("Day").tag("day")
        Text("Night").tag("night")
    }
    .pickerStyle(.segmented)
    .padding()

    Text("With Images:")

    Picker("", selection: $tab) {
        Image(systemName: "sun.min").tag(0)
        Image(systemName: "moon").tag(1)
    }
    .pickerStyle(.segmented)
    .padding()
}
```

No Segment Selected


11:37
Segmented Control (Picker)
No Segment Selected
This segmented control will have nothing selected because the default state variable does not match any of the segment tag values.
How many meals do you eat?
One
Two
Three
More
The red outline will go away once a selection is made.

```
@State private var selection = 0

...

VStack(spacing: 20) {
    Text("Segmented Control (Picker)").font(.largeTitle)
    Text("No Segment Selected")
        .font(.title).foregroundColor(.gray)
    Text("This segmented control will have nothing selected because the default state variable 
does not match any of the segment tag values.")
        ...

    Text("How many meals do you eat?")
        .foregroundColor(.gray)
        .font(.title)

    Picker("", selection: $selection) {
        Text("One").tag(1)
        Text("Two").tag(2)
        Text("Three").tag(3)
        Text("More").tag(4)
    }
    .pickerStyle(.segmented)
    .background(RoundedRectangle(cornerRadius: 8)
        .stroke(Color.red, lineWidth: selection == 0 ? 1 : 0))
    .padding()

    Text("The red outline will go away once a selection is made.")
        ...
}
```

Colors



```
struct SegmentedControl_Colors: View {
    @State private var selection = 2

    var body: some View {
        VStack(spacing: 60) {
            Picker("", selection: $selection) {
                Text("One").tag(1)
                Text("Two").tag(2)
                Text("Three").tag(3)
            }
            .pickerStyle(.segmented)
            .background(Color.pink)

            Picker("", selection: $selection) {
                Text("One").tag(1)
                Text("Two").tag(2)
                Text("Three").tag(3)
            }
            .pickerStyle(.segmented)
            .background(Color.pink, in: RoundedRectangle(cornerRadius: 8))

            Picker("", selection: $selection) {
                Text("One").tag(1)
                Text("Two").tag(2)
                Text("Three").tag(3)
            }
            .pickerStyle(.segmented)
            .background(Color.accentColor.opacity(0.6), in: RoundedRectangle(cornerRadius: 8))
        }
        .padding(.horizontal)
    }
}
```

You can change the color of segmented controls by using the background modifier.

Use a rounded rectangle in your background to better fit the existing picker.

Or use your accent color defined in your asset catalog and lighten it with the opacity modifier.

iOS 16

ShareLink

This SwiftUI chapter is locked in this preview.

The share link button gives y… ient way to for users to share some text, a URL, collections of data, photos, etc.

It can be styled and customized just like a button.

This is a pull-in view.

iOS 14

SignInWithAppleButton

Apple provides you with a [illegible] at you use in your app [illegible] your users when it comes time to implementing signing in with Apple.

This is a push-out view.

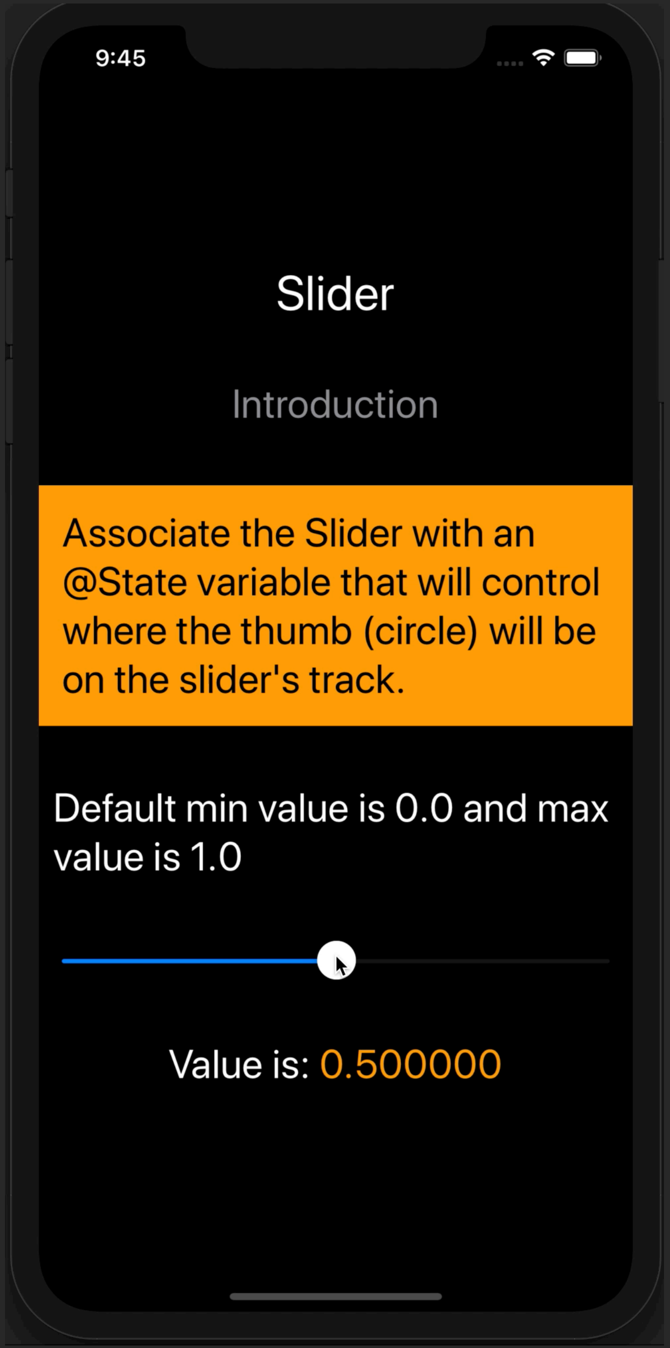
Slider

When using a Slider view, the default range of values is 0.0 to 1.0. You bind the Slider to a state variable, usually a number type, like an Int. But it doesn't have to be a number type. It can be any type that conforms to the Stridable protocol. ("Stride" means to "take steps in a direction; usually long steps".) A type that conforms to Stridable (such as an Int) means it has values that are continuous and can be stepped through and measured. ("Step through", "Stride", I think you see the connection now.)

You use the bound variable to set or get the value the Slider's thumb (circle) is currently at.

This is a pull-in view.

Introduction



Slider

Introduction

Associate the Slider with an @State variable that will control where the thumb (circle) will be on the slider's track.

Default min value is 0.0 and max value is 1.0

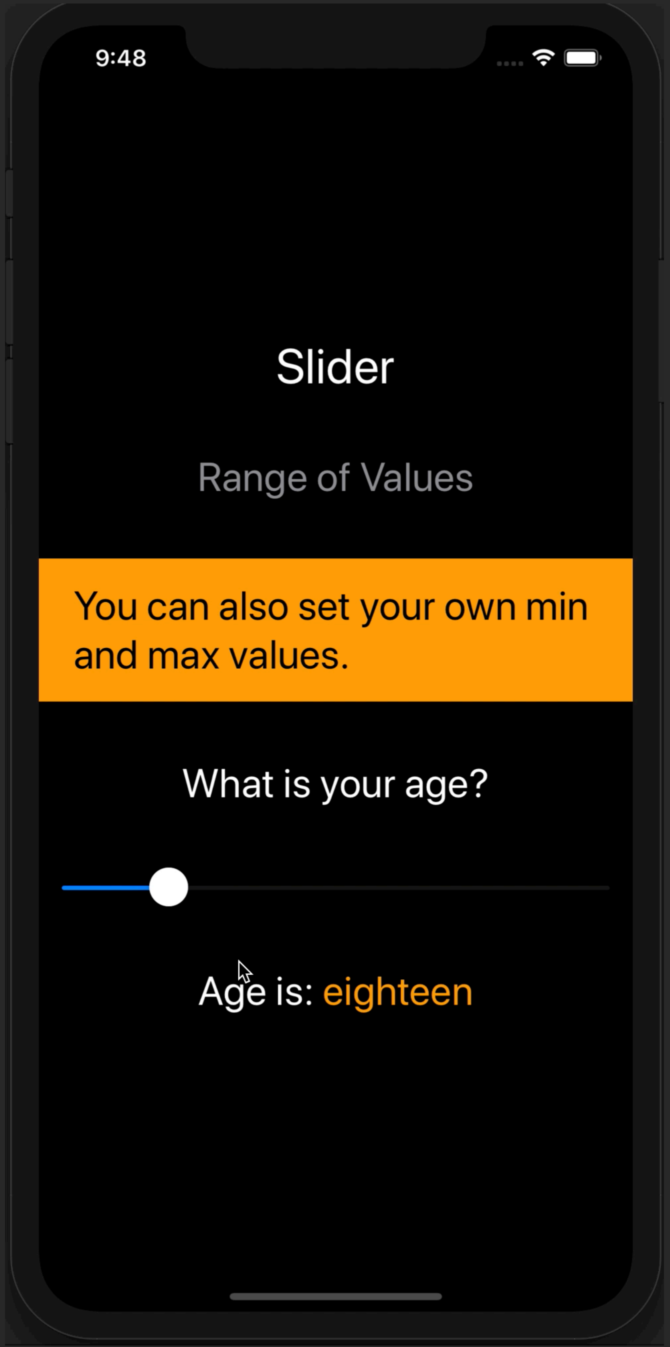
Value is: 0.500000

```
struct Slider_Intro : View {
    @State private var sliderValue = 0.5

    var body: some View {
        VStack(spacing: 40) {
            Text("Slider").font(.largeTitle)
            Text("Introduction").foregroundColor(.gray)
            Text("Associate the Slider with an @State variable that will control where the thumb 
(circle) will be on the slider's track.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange).foregroundColor(Color.black)

            Text("Default min value is 0.0 and max value is 1.0")

            Slider(value: $sliderValue)
                .padding(.horizontal)

            Text("Value is: ") +
                Text("\(sliderValue)").foregroundColor(.orange)

        }.font(.title)
    }
}
```

Value used for the slider.

Set the state variable in the slider's initializer.

Range of Values (Minimum & Maximum)


9:48

Slider

Range of Values

You can also set your own min and max values.

What is your age?

Age is: eighteen

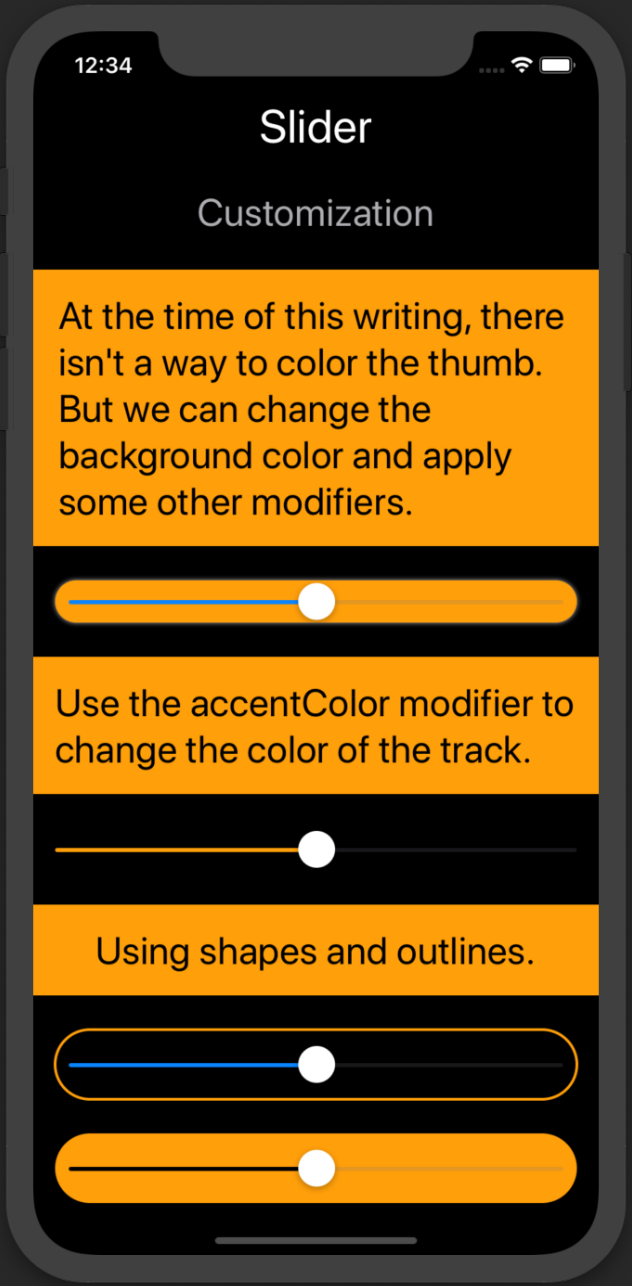
```
struct Slider_RangeOfValues: View {
    @State private var age = 18.0

    let ageFormatter: NumberFormatter = {
        let numFormatter = NumberFormatter()
        numFormatter.numberStyle = .spellOut
        return numFormatter
    }()

    var body: some View {
        VStack(spacing: 40) {
            Text("Slider").font(.largeTitle)
            Text("Range of Values").foregroundColor(.gray)
            Text("You can also set your own min and max values.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange).foregroundColor(Color.black)

            Text("What is your age?")

            Slider(value: $age, in: 1...100, step: 1)
                .padding(.horizontal)

            Text("Age is: ") +
                Text("\(ageFormatter.string(from: NSNumber(value: age))!)")
                    .foregroundColor(.orange)
        }.font(.title)
    }
}
```

Format the slider value into a spelled-out number.

Provide a range here.

The step parameter defines the increment from one value to the next.

Customization


Slider

Customization

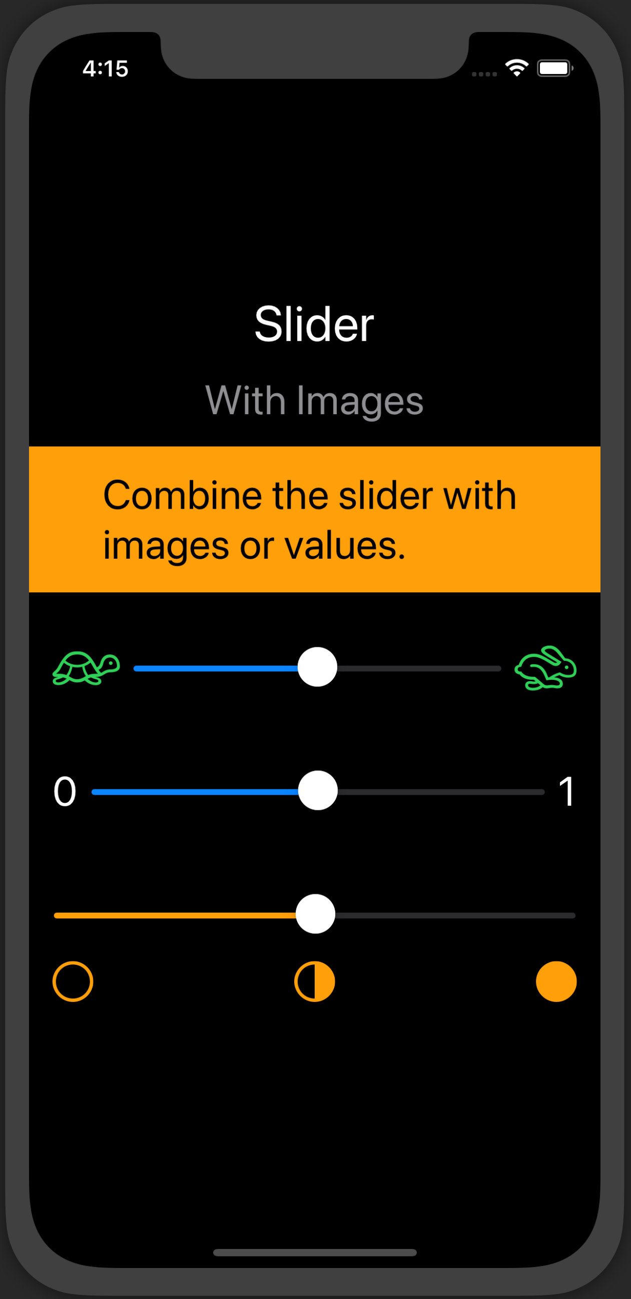
At the time of this writing, there isn't a way to color the thumb. But we can change the background color and apply some other modifiers.

Use the accentColor modifier to change the color of the track.

Using shapes and outlines.

```
@State private var sliderValue = 0.5
...
Text("At the time of this writing, there isn't a way to color the thumb. But we can change the
background color and apply some other modifiers.")
    …
Slider(value: $sliderValue)
    .padding(.horizontal, 10)
    .background(Color.orange)
    .shadow(color: .gray, radius: 2)
    .padding(.horizontal)

Text("Use the accentColor modifier to change the color of the track.")
    ...

Slider(value: $sliderValue)
    .padding(.horizontal)
    .accentColor(.orange)

Text("Using shapes and outlines.")
    ...

Slider(value: $sliderValue)
    .padding(10)
    .background(Capsule().stroke(Color.orange, lineWidth: 2))
    .padding(.horizontal)

Slider(value: $sliderValue)
    .padding(10)
    .background(Capsule().fill(Color.orange))
    .accentColor(.black)
    .padding(.horizontal)
```

With Images

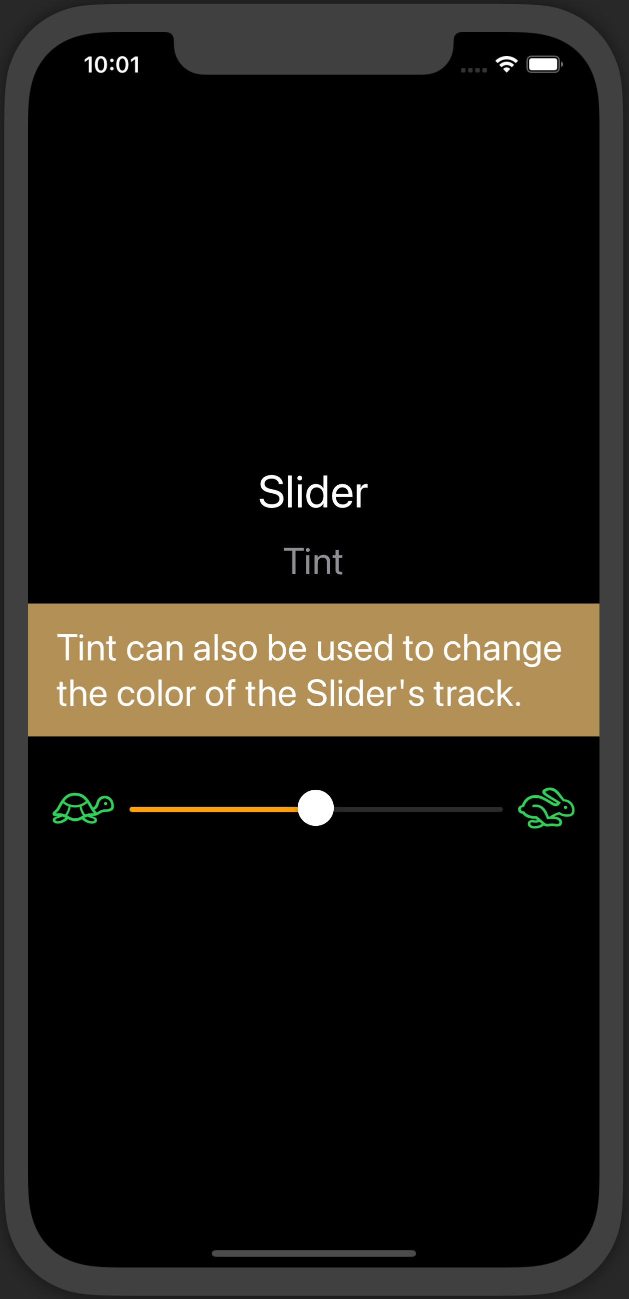


```
struct Slider_WithImages : View {
    @State private var sliderValue = 0.5

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Slider",
                       subtitle: "With Images",
                       desc: "Combine the slider with images or values.",
                       back: .orange, textColor: .black)

            Slider(value: $sliderValue,
                   minimumValueLabel: Image(systemName: "tortoise"),
                   maximumValueLabel: Image(systemName: "hare"), label: {})
                .foregroundColor(.green)
                .padding()

            Slider(value: $sliderValue,
                   minimumValueLabel: Text("0"),
                   maximumValueLabel: Text("1"), label: {})
                .padding()

            VStack {
                Slider(value: $sliderValue)
                    .accentColor(.orange)
                HStack {
                    Image(systemName: "circle")
                    Spacer()
                    Image(systemName: "circle.righthalf.fill")
                    Spacer()
                    Image(systemName: "circle.fill")
                }
                .foregroundColor(.orange)
                .padding(.top, 8)
            }.padding()
        }.font(.title)
    }
}
```

Use minimum and maximum value labels to add text or images to the ends of the slider.

Tint

iOS 15



Tint can also be used to change the color of the Slider's track.

```
struct Slider_Tint: View {
    @State private var sliderValue = 0.5

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Slider",
                       subtitle: "Tint",
                       desc: "Tint can also be used to change the color of the Slider's track.")

            Slider(value: $sliderValue,
                   minimumValueLabel: Image(systemName: "tortoise"),
                   maximumValueLabel: Image(systemName: "hare"), label: {})
                .foregroundColor(.green)
                .tint(.orange)
                .padding()
        }
        .font(.title)
    }
}
```

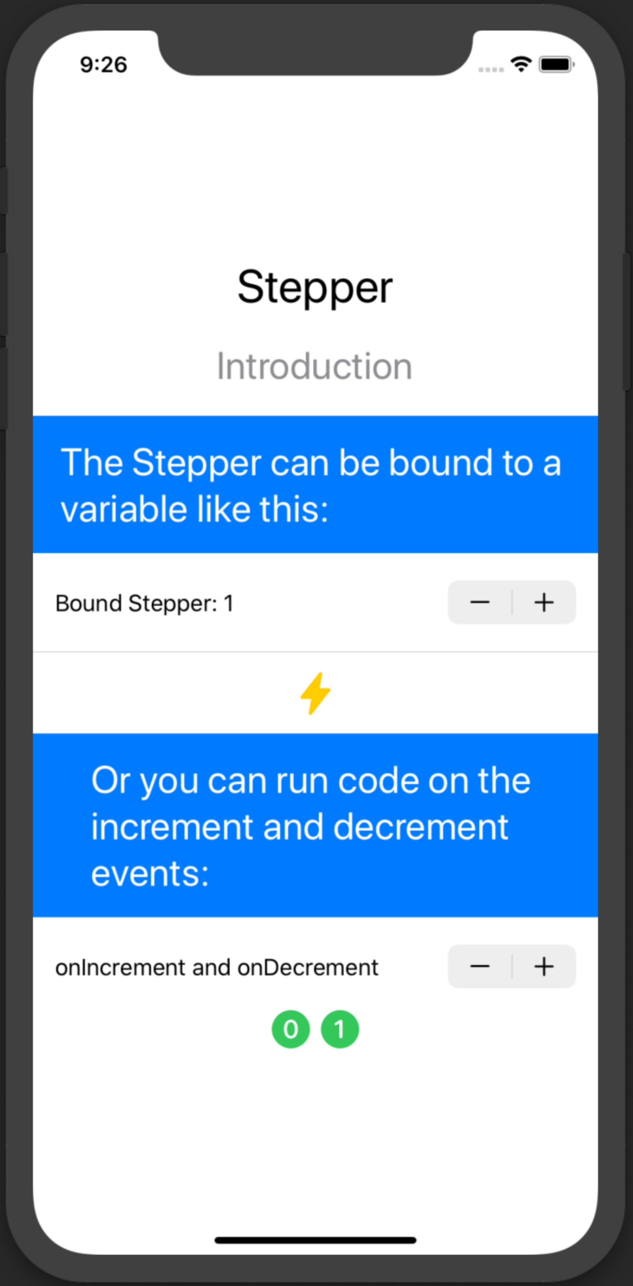
Stepper

When using a Stepper view, you bind it to a state variable, usually a number. But it doesn’t have to be a number type. It can be any type that conforms to the Stridable protocol. (“Stride” means to “take steps in a direction; usually long steps”.) A type that conforms to Stridable means it has values that are continuous and can be stepped through and measured. (“Step through”, “Stride”, I think you see the connection now.)

You use the bound variable to set or get the value it is currently at.

This is a horizontal push-out view. Vertically it is pull-in.

Introduction


9:26

Stepper

Introduction

The Stepper can be bound to a variable like this:

Bound Stepper: 1

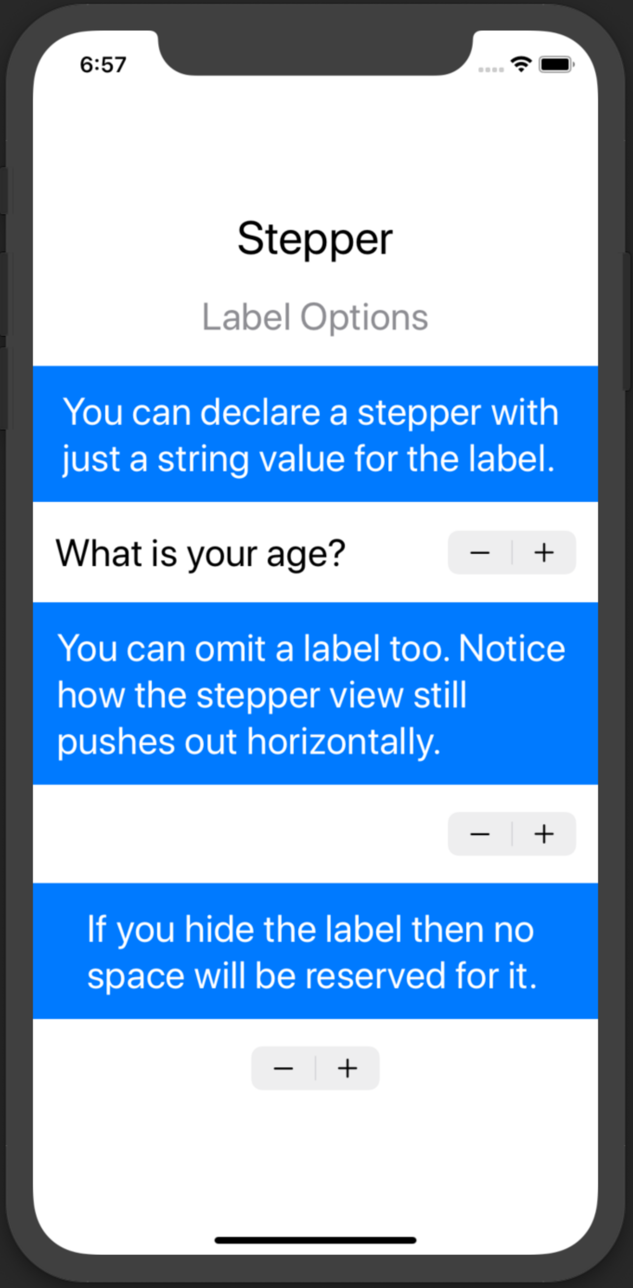
Or you can run code on the increment and decrement events:

onIncrement and onDecrement

0 1

```
@State private var stepperValue = 1
@State private var values = [0, 1]
...
VStack(spacing: 20) {
    Text("Stepper")
        .font(.largeTitle)
    Text("Introduction")
        .font(.title).foregroundColor(.gray)
    Text("The Stepper can be bound to a variable like this:")
        ...

    Stepper(value: $stepperValue) {
        Text("Bound Stepper: \(stepperValue)")
    }.padding(.horizontal)
    Divider()
    Image(systemName: "bolt.fill")
        .font(.title).foregroundColor(.yellow)
    Text("Or you can run code on the increment and decrement events:")
        .frame(maxWidth: .infinity).padding()
        .background(Color.blue).foregroundColor(Color.white)
        .font(.title)
    Stepper(onIncrement: {self.values.append(self.values.count)},
            onDecrement: {self.values.removeLast()}) {
                Text("onIncrement and onDecrement")
    }.padding(.horizontal)
    HStack {
        ForEach(values, id: \.self) { value in
            Image(systemName: "\(value).circle.fill")
        }
    }.font(.title).foregroundColor(.green)
}
```

Label Options



Stepper

Label Options

You can declare a stepper with just a string value for the label.

What is your age?

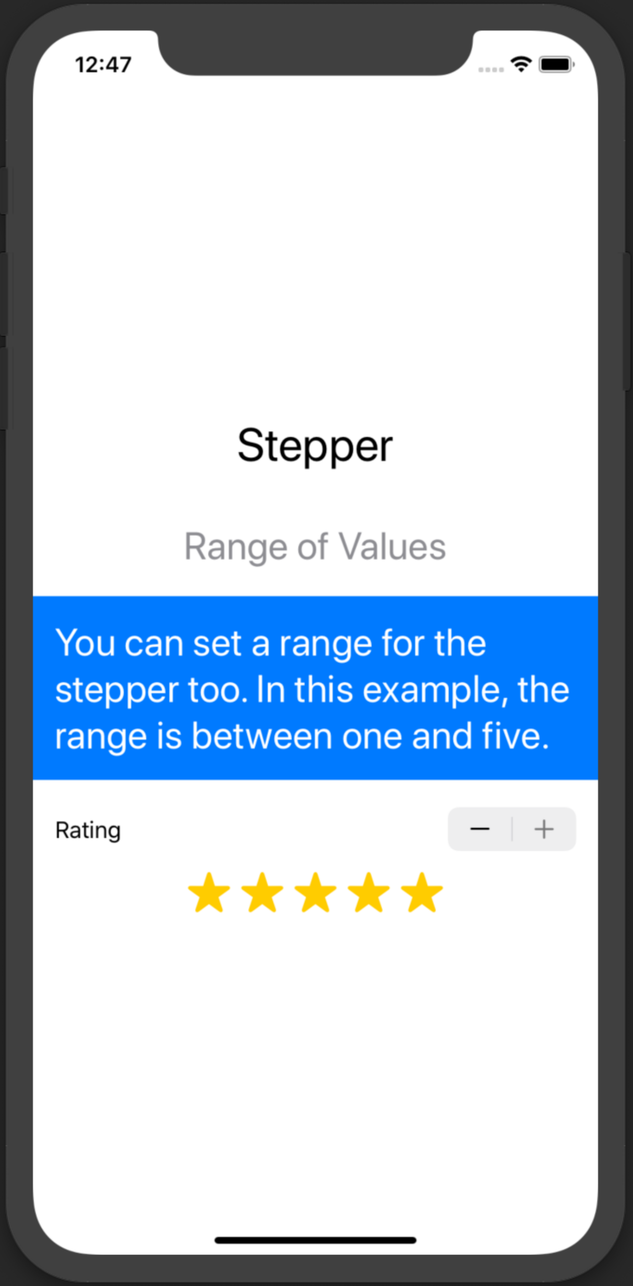
You can omit a label too. Notice how the stepper view still pushes out horizontally.

If you hide the label then no space will be reserved for it.

```
struct Stepper_LabelsHidden: View {
    @State private var stepperValue = 1

    var body: some View {
        VStack(spacing: 20) {
            Text("Stepper").font(.largeTitle)
            Text("Label Options").foregroundColor(.gray)
            Text("You can declare a stepper with just a string value for the label.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.blue).foregroundColor(Color.white)
            Stepper("What is your age?", value: $stepperValue)
                .padding(.horizontal)
            Text("You can omit a label too. Notice how the stepper view still pushes out 
horizontally.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.blue).foregroundColor(Color.white)
            Stepper("", value: $stepperValue)
                .padding(.horizontal)
            Text("If you hide the label then no space will be reserved for it.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.blue).foregroundColor(Color.white)
            Stepper("What is your age?", value: $stepperValue)
                .padding(.horizontal)
                .labelsHidden()
        }.font(.title)
    }
}
```

Note: Even though the label/title is not shown, I would still recommend having one because it will still be used for accessibility purposes.

Range

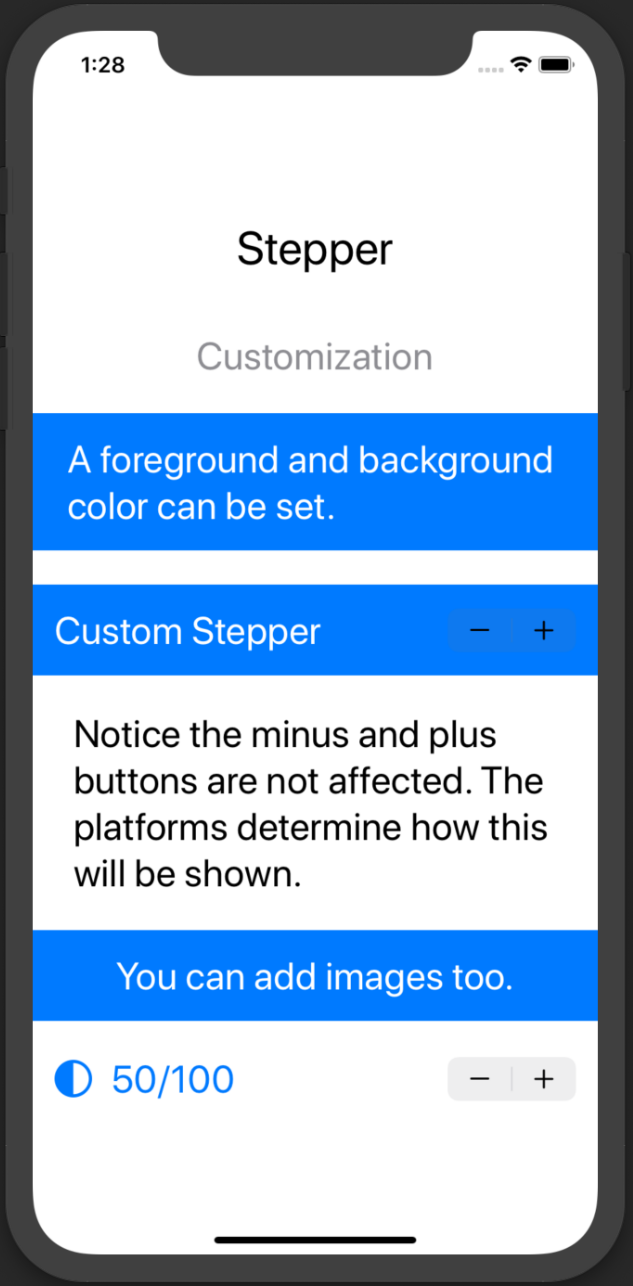


You can set a range for the stepper too. In this example, the range is between one and five.

```
@State private var stars = 0

VStack(spacing: 20) {
    Text("Stepper")
        .font(.largeTitle)
        .padding()
    Text("Range of Values")
        .font(.title)
        .foregroundColor(.gray)
    Text("You can set a range for the stepper too. In this example, the range is between one and 
five.")
        ...

    Stepper(value: $stars, in: 1...5) {
        Text("Rating")
    }.padding(.horizontal)

    HStack {
        ForEach(1...stars, id: \.self) { star in
            Image(systemName: "star.fill")
        }
    }
    .font(.title)
    .foregroundColor(.yellow)
}
```

When the Stepper reaches the range limits, the corresponding plus or minus button will appear as disabled. In this screenshot, notice the plus button is disabled.

Customization



Stepper

Customization

A foreground and background color can be set.

Custom Stepper − +

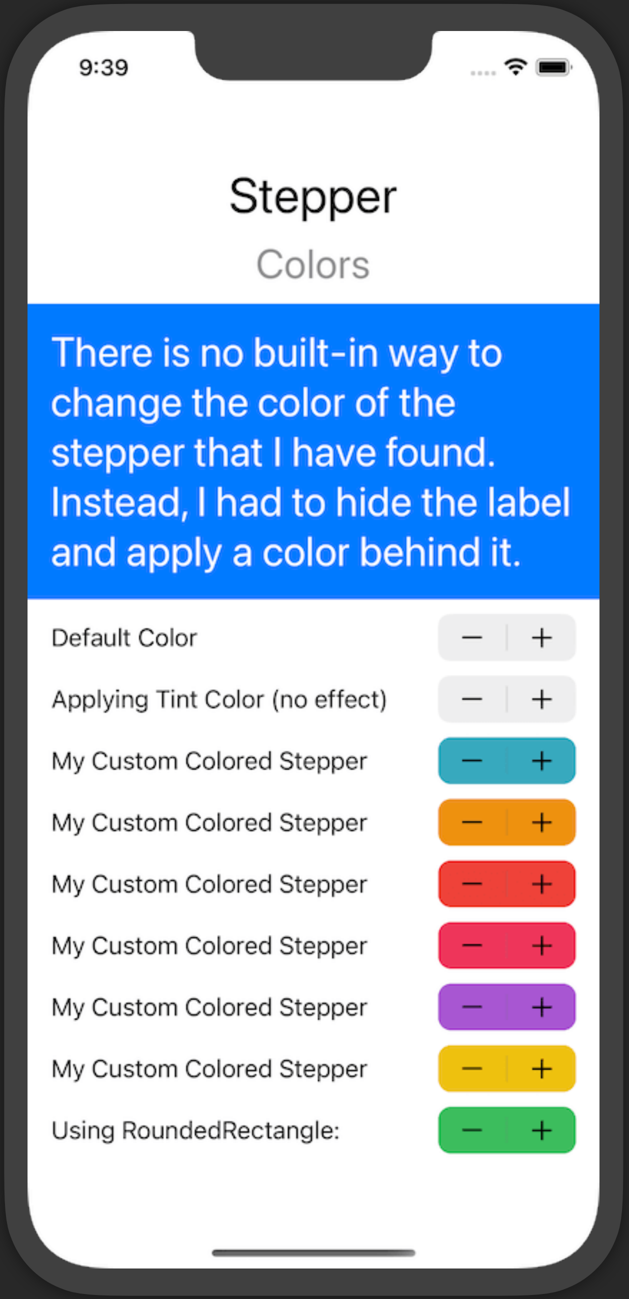
Notice the minus and plus buttons are not affected. The platforms determine how this will be shown.

You can add images too.

50/100 − +

```
@State private var contrast = 50
...
Text("A foreground and background color can be set.")
    ...

Stepper(onIncrement: {}, onDecrement: {}) {
    Text("Custom Stepper")
        .font(.title)
        .padding(.vertical)
}
.padding(.horizontal)
.background(Color.blue)
.foregroundColor(.white)

Text("Notice the minus and plus buttons are not affected. The platforms determine how this will
be shown.")
    ...

Text("You can add images too.")
    .frame(maxWidth: .infinity).padding()
    .background(Color.blue).foregroundColor(Color.white)
    .font(.title)

Stepper(value: $contrast, in: 0...100) {
    // SwiftUI implicitly uses an HStack here
    Image(systemName: "circle.lefthalf.fill")
    Text(" \(contrast)/100")
}
.font(.title)
.padding(.horizontal)
.foregroundColor(.blue)
```

Colors


9:39

Stepper

Colors

There is no built-in way to change the color of the stepper that I have found. Instead, I had to hide the label and apply a color behind it.

Default Color − +

Applying Tint Color (no effect) − +

My Custom Colored Stepper − +

My Custom Colored Stepper − +

My Custom Colored Stepper − +

My Custom Colored Stepper − +

My Custom Colored Stepper − +

My Custom Colored Stepper − +

Using RoundedRectangle: − +

```
Text("There is no built-in way to change the color of the stepper that I have found. Instead, I
had to hide the label and apply a color behind it.")
...

Stepper(value: $contrast, in: 0...100) {
    Text("Applying Tint Color (no effect)")
}
.tint(.blue)

HStack {
    Text("My Custom Colored Stepper")
    Spacer()
    Stepper("", value: $contrast)
        .background(Color.teal, in: RoundedRectangle(cornerRadius: 9))
        .labelsHidden() // Hide the label
}

HStack {
    Text("My Custom Colored Stepper")
    Spacer()
    Stepper("", value: $contrast)
        .background(Color.orange, in: RoundedRectangle(cornerRadius: 9))
        .cornerRadius(9)
        .labelsHidden() // Hide the label
}
```

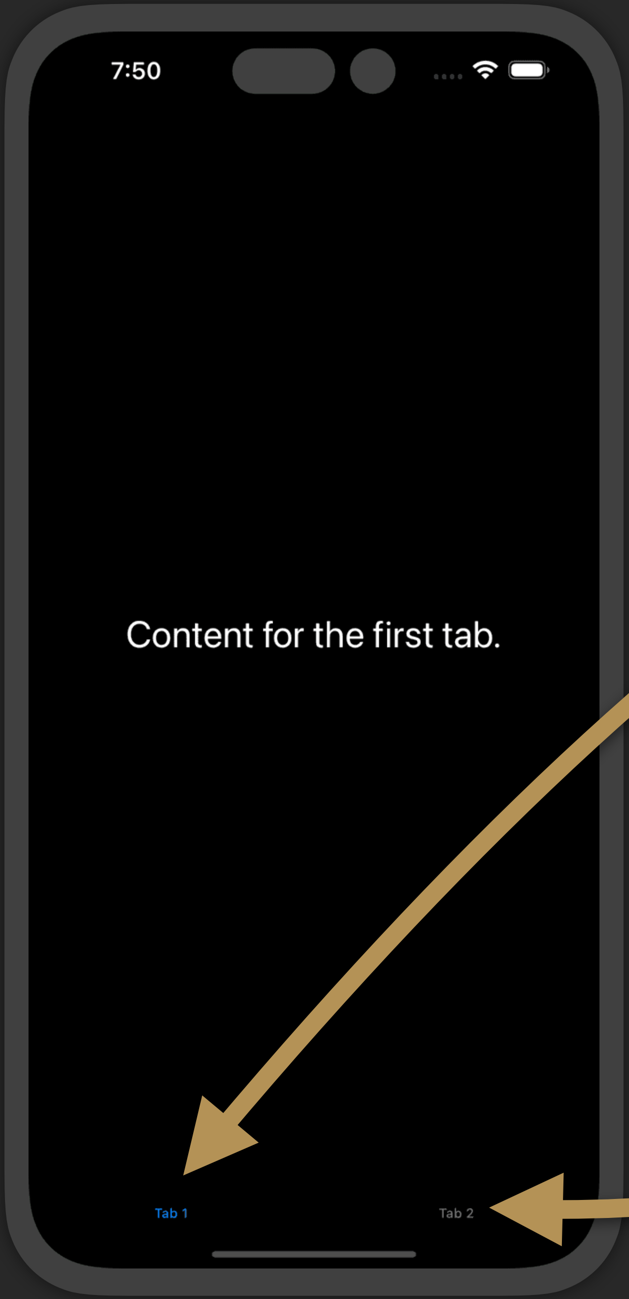
TabView

The TabView acts as a container for child views (Tabs) within it.
These tabs views are individual screens.
It provides tab buttons that allow the user to switch between these child views.

This is a push-out view.

Introduction

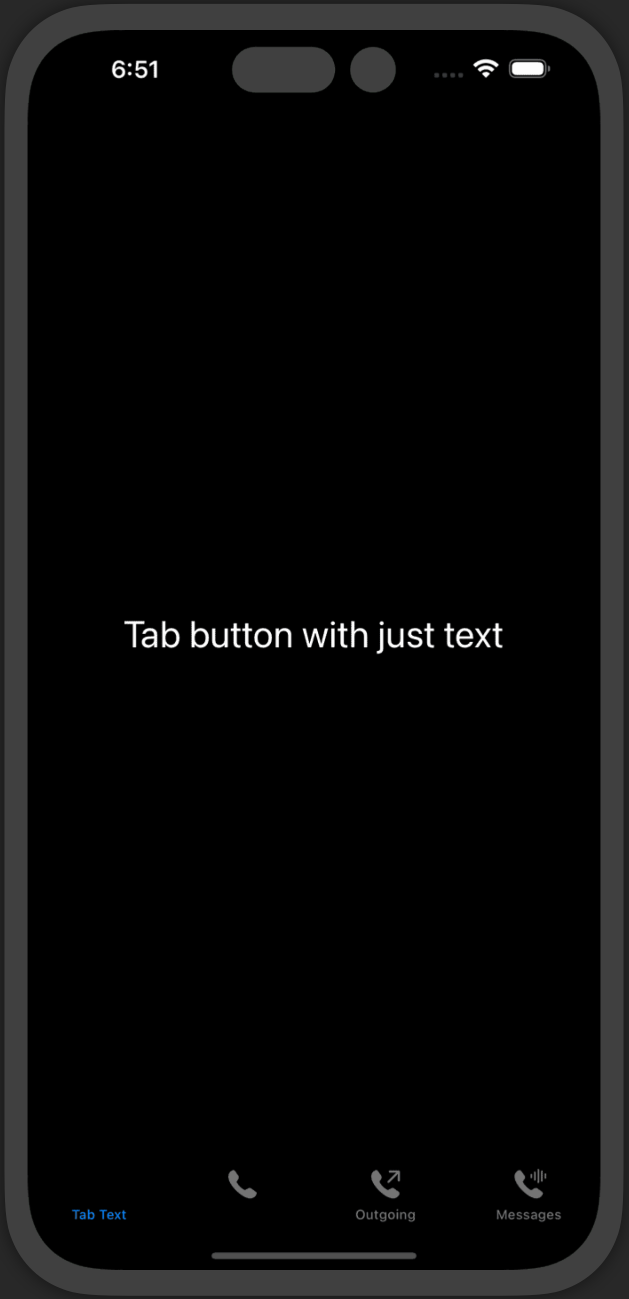
```
struct TabView_Intro : View {
    var body: some View {
        TabView {
            Tab {
                Text("Content for the first tab.")
            } label: {
                Text("Tab 1")
            }
            Tab {
                Text("Content for the second tab.")
            } label: {
                Text("Tab 2")
            }
        }
        .font(.title)
    }
}
```



The TabView can hold multiple Tab views.

The label parameter is where you put the optional text and images for the tab.

Tab Text & Images



```
struct TabView_TabItems : View {
    var body: some View {
        TabView {
            Tab {
                Text("Tab button with just text")
            } label: {
                Text("Tab Text")
            }

            Tab("", systemImage: "phone") {
                Text("Tab with just an image")
            }

            Tab("Outgoing", systemImage: "phone.arrow.up.right") {
                Text("Tab with text and image")
            }

            Tab {
                Text("Tab button using a Label for text and image")
            } label: {
                Label("Messages", systemImage: "phone.badge.waveform")
            }
        }
        .font(.title)
    }
}
```

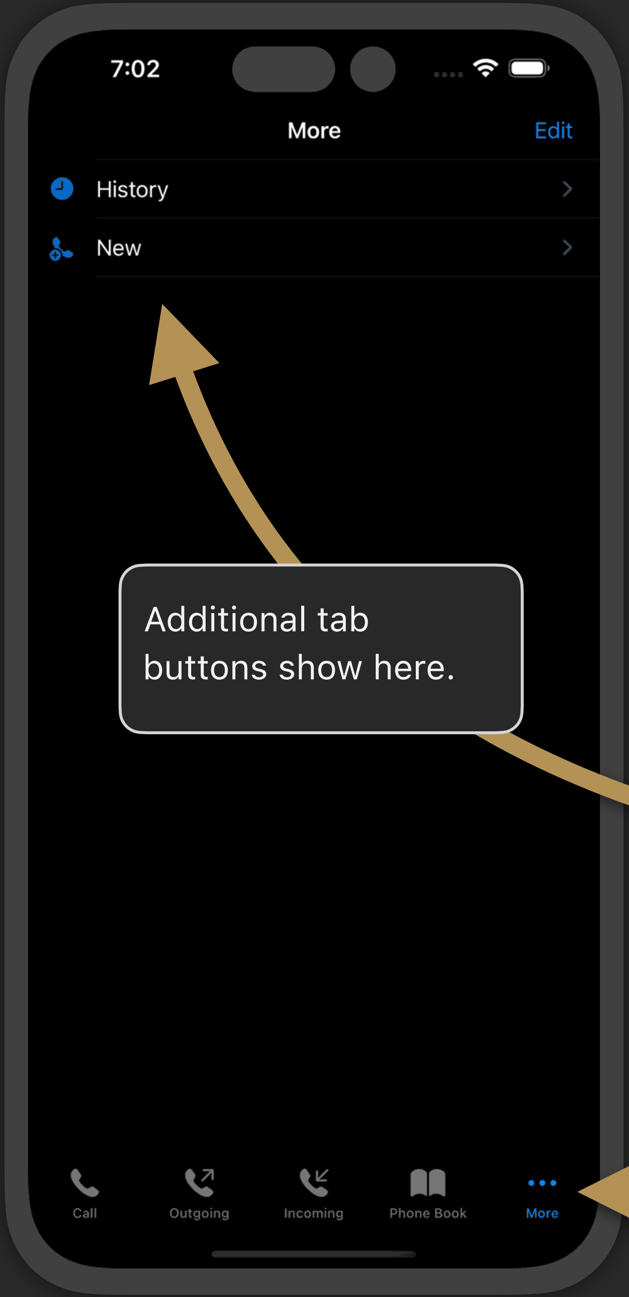
A tab can just show text.

Or just an image.

You can specify both in the Tab initializer.

You can even use a Label for the text and image in the label closure.

Too Many Tabs

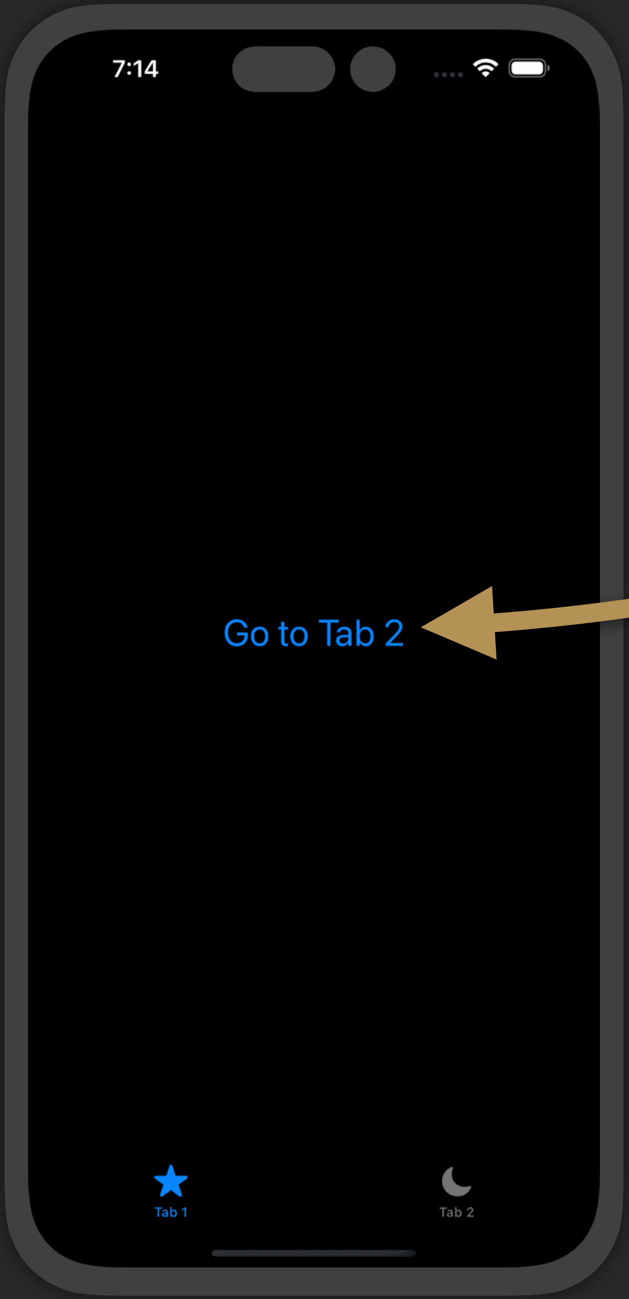


Additional tab buttons show here.

```swift
struct TabView_TooManyTabs : View {
    var body: some View {
        TabView {
            Tab("Call", systemImage: "phone") {
                Text("Call Screen")
            }

            Tab("Outgoing", systemImage: "phone.arrow.up.right") {
                Text("Outgoing Phone Calls Screen")
            }

            Tab("Incoming", systemImage: "phone.arrow.down.left") {
                Text("Incoming Phone Calls Screen")
            }

            Tab("Phone Book", systemImage: "book") {
                Text("Phone Book Screen")
            }

            Tab("History", systemImage: "clock") {
                Text("History Screen")
            }

            Tab("New", systemImage: "phone.badge.plus") {
                Text("New Phone Number")
            }
        }
    }
}
```

When there are too many tabs to fit for the device, the **More** button is created where you can find the rest of the tabs listed out.

Programmatically Switching Tabs



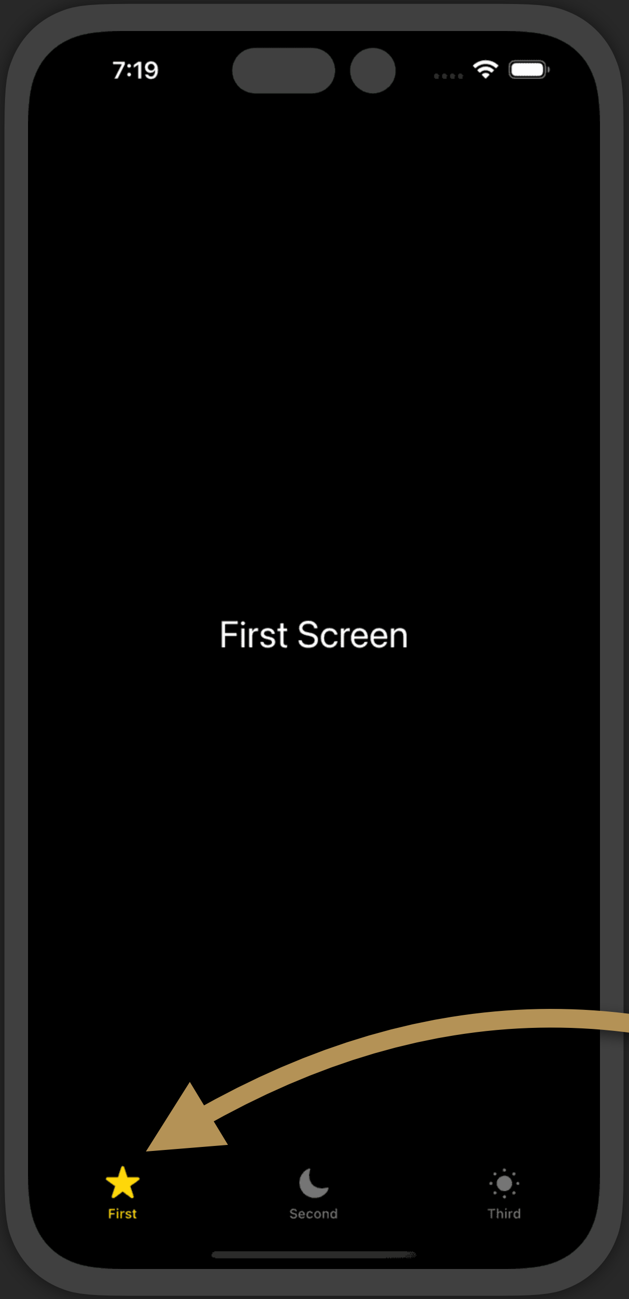
```
struct TabView_Navigating : View {
    @State private var selectedTab = 1

    var body: some View {
        TabView(selection: $selectedTab) {
            Tab("Tab 1", systemImage: "star", value: 1) {
                VStack(spacing: 20) {
                    Button("Go to Tab 2") {
                        selectedTab = 2
                    }
                }
            }

            Tab("Tab 2", systemImage: "moon", value: 2) {
                VStack {
                    Text("Second Screen")
                }
            }
        }
        .font(.title)
    }
}
```

Add a unique value to each Tab you want to programmatically navigate to. You can then bind a variable to the TabView's selection property and change that variable to navigate.

Change the state property bound to the TabView’s selection parameter to switch to a different tab.

Add a value to enable programmatically switching tabs.

Colors

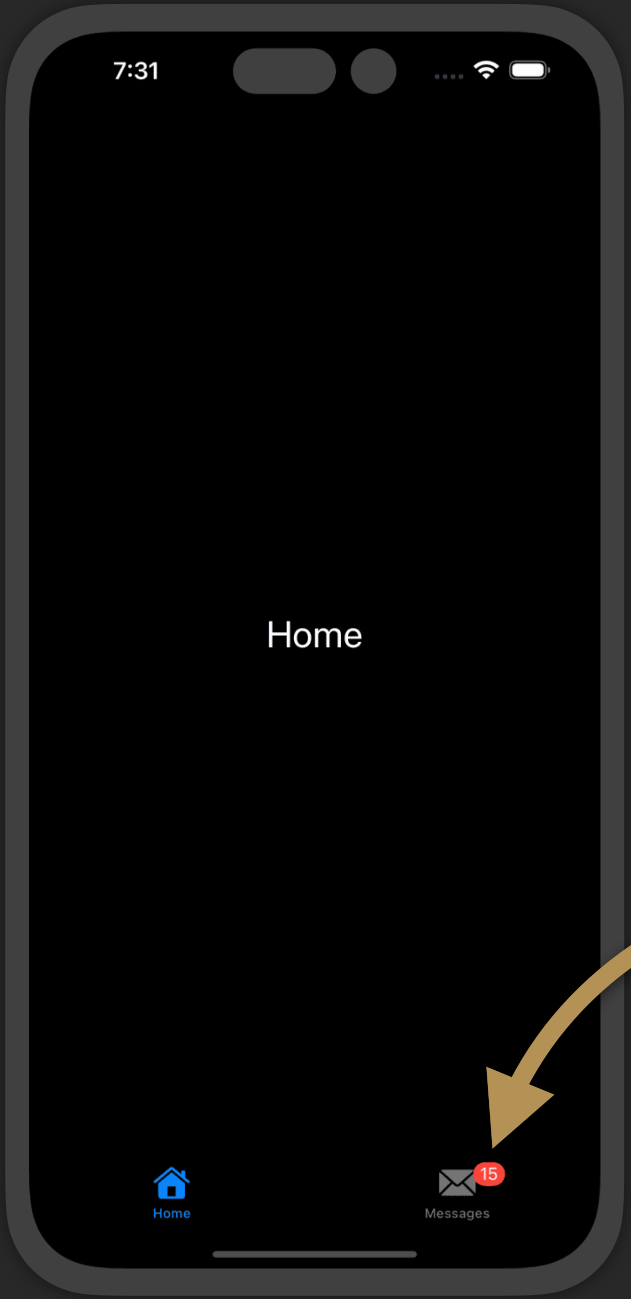


```
struct TabView_Colors: View {
    var body: some View {
        TabView {
            Tab("First", systemImage: "star") {
                Text("First Screen")
            }

            Tab("Second", systemImage: "moon") {
                Text("Second Screen")
            }

            Tab("Third", systemImage: "sun.min") {
                Text("Third Screen")
            }
        }
        .font(.title)
        .tint(.yellow)
    }
}
```

Set the color of the active Tab by setting the tint color for the TabView.

Use tint to change the color of the active tab.

Badge

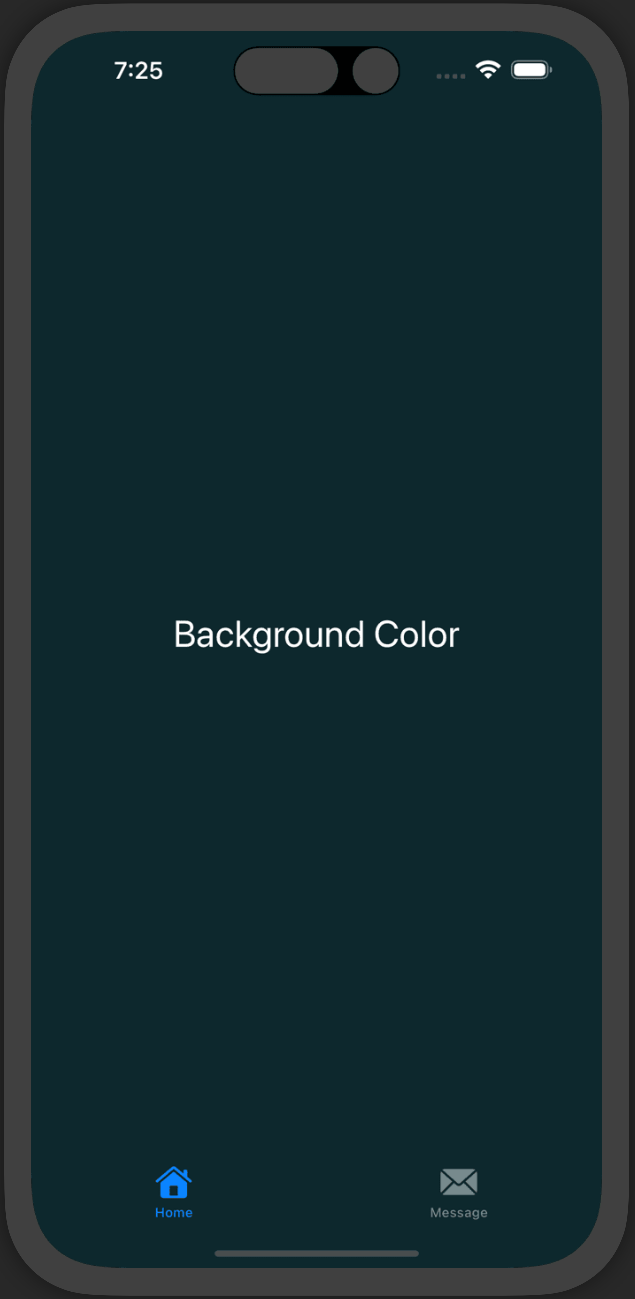


```swift
struct TabView_Badge: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                VStack {
                    Text("Home")
                }
            }

            Tab("Messages", systemImage: "envelope") {
                VStack {
                    Text("Messages")
                }
            }
            .badge(15)
        }
        .font(.title)
    }
}
```

Use the badge modifier to add a number to the Tab.

Note: The operating system will set the color of the badge. There isn't a way to change the color.

Background Color

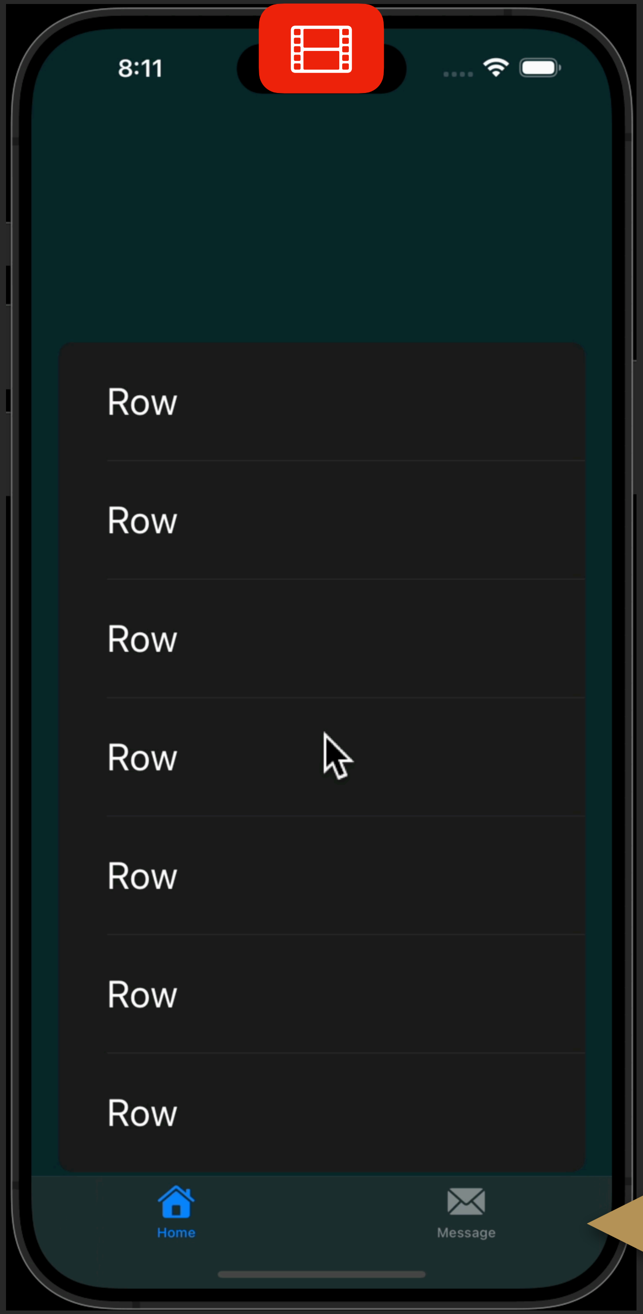


```
struct Toolbar_BackgroundColor: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                ZStack {
                    Color.teal.opacity(0.2)
                        .ignoresSafeArea()

                    Text("Background Color")
                }
            }

            Tab("Message", systemImage: "envelope") {
                Text("No Background Color")
            }
        }
        .font(.title)
    }
}
```

If you add a background color, you will notice that color extends behind the TabView.

This background color is just for this one tab, not the whole TabView.

Background Color When Scrolling



```
struct TabView_Background_Scrolling: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                List(0 ..< 7) { item in
                    Text("Row")
                        .padding()
                }
                .scrollContentBackground(.hidden)
                .background(Color.teal.opacity(0.2))
            }

            Tab("Message", systemImage: "envelope") {
                Text("No Background Color")
            }
        }
        .font(.title)
    }
}
```

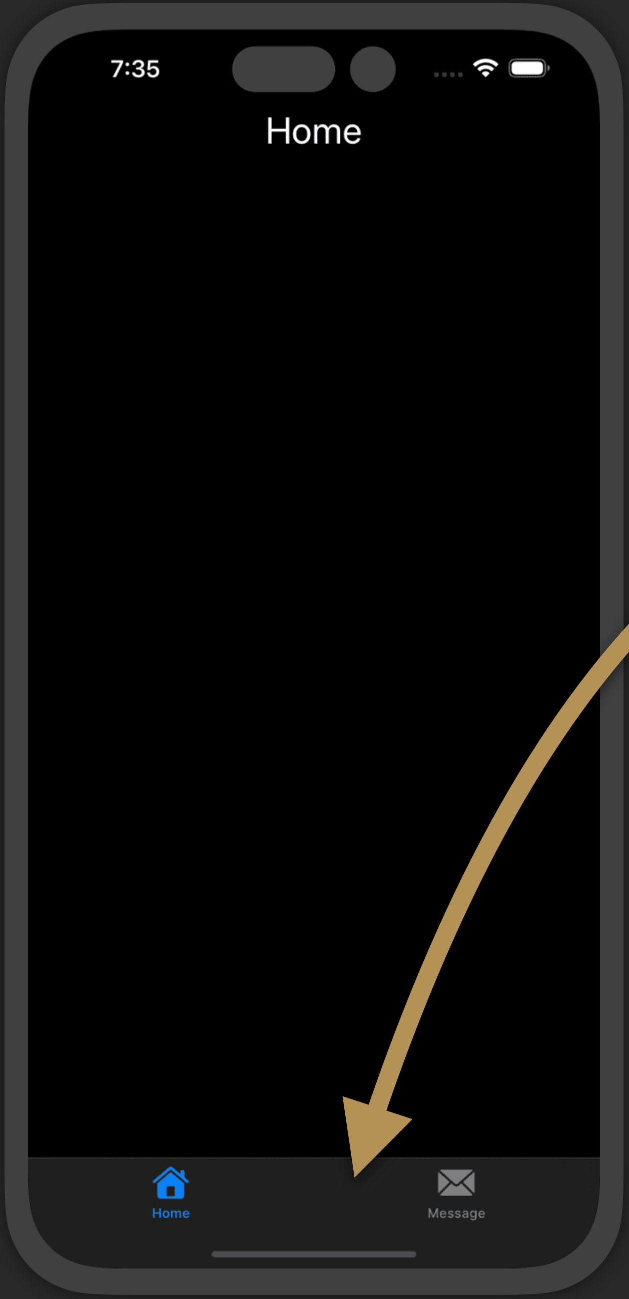
The background that appears when content is scrolled behind it can be customized.

Note: If content/views scroll behind the Tab bar, a background will appear on it.

By default, the system uses a material (blur) but you can also customize this color or style when content scrolls behind it or force this background style to show all the time.

The following pages will show you different ways to do this for different iOS versions your app may need to support.

Background Warning



```
struct TabView_Background: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                VStack {
                    Text("Home")
                    Spacer()
                    Divider()
                        .background(.ultraThinMaterial)
                }
            }

            Tab("Message", systemImage: "envelope") {
                VStack {
                    Text("Messages")
                    Spacer()
                    Divider()
                        .background(Color.brown.opacity(0.5), ignoresSafeAreaEdges: .top)
                }
            }
        }
        .font(.title)
    }
}
```

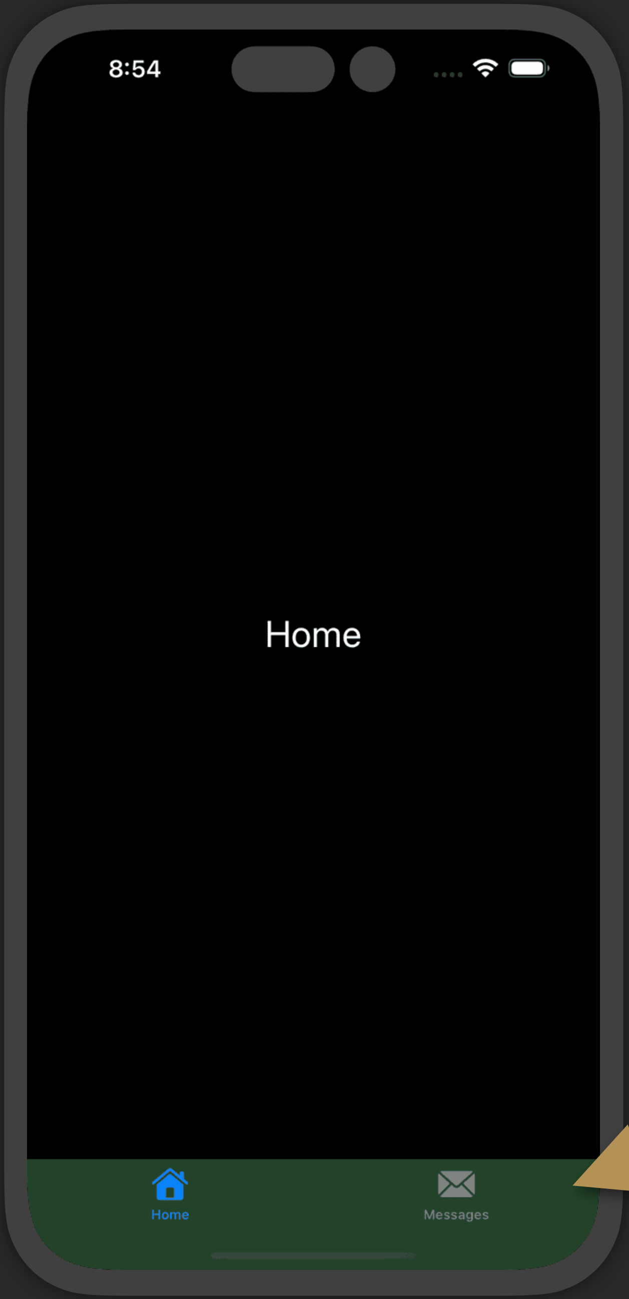
The background modifier allows you to have backgrounds that ignore safe area edges, such as the TabView safe area edge.

In this example, the thin material blur is touching the TabView's safe area edge and, by default, it will ignore **all** safe area edges and go **into** it.

To avoid this problem, use the ignoresSafeAreaEdges parameter and set it to something besides .all or .bottom.

Keep reading for other ways to change the background color.

UITabBarAppearance



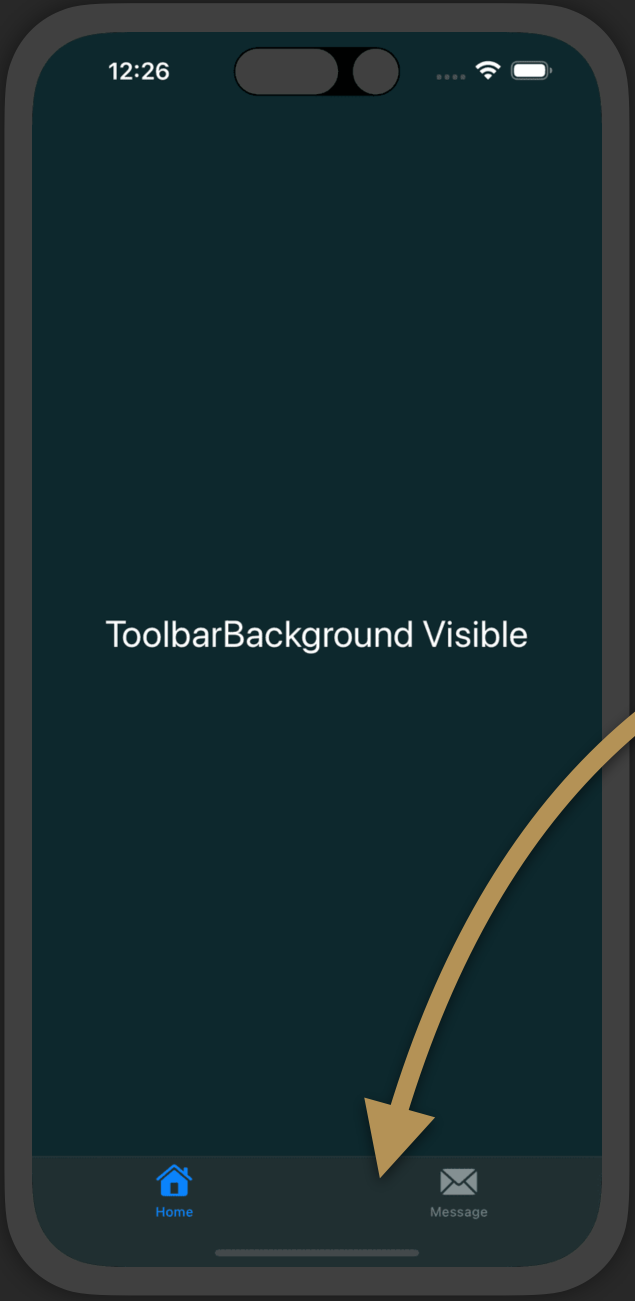
```
struct TabView_UITabBarAppearance: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                VStack {
                    Text("Home")
                }
            }

            Tab("Messages", systemImage: "envelope") {
                VStack {
                    Text("Messages")
                }
            }
        }
        .font(.title)
        .onAppear {
            let appearance = UITabBarAppearance()
            appearance.backgroundEffect = UIBlurEffect(style: .systemUltraThinMaterial)
            appearance.backgroundColor = UIColor(Color.green.opacity(0.2))
            UITabBar.appearance().standardAppearance = appearance
            UITabBar.appearance().scrollEdgeAppearance = appearance
        }
    }
}
```

If you have to support **iOS 15**, this is another way to change the Tab bar appearance for all tabs.

You can set just the effect or just the color if you want or combine them like you see here.

The standardAppearance is used when you have a ScrollView and content is scrolling behind the Tabview.

Toolbar Background Visibility & Color



```swift
struct TabView_ToolbarBackground: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                ZStack {
                    Color.teal.opacity(0.2)
                        .ignoresSafeArea()

                    Text("ToolbarBackground Visible")
                }
                .toolbarBackground(.visible, for: .tabBar)
            }

            Tab("Message", systemImage: "envelope") {
                Text("ToolbarBackground Hidden")
            }
        }
        .font(.title)
    }
}
```

iOS 16

Use the toolbarBackground to give the TabView a system default background.

This is the same background that shows when content scrolls behind the TabView bar.

⚠️

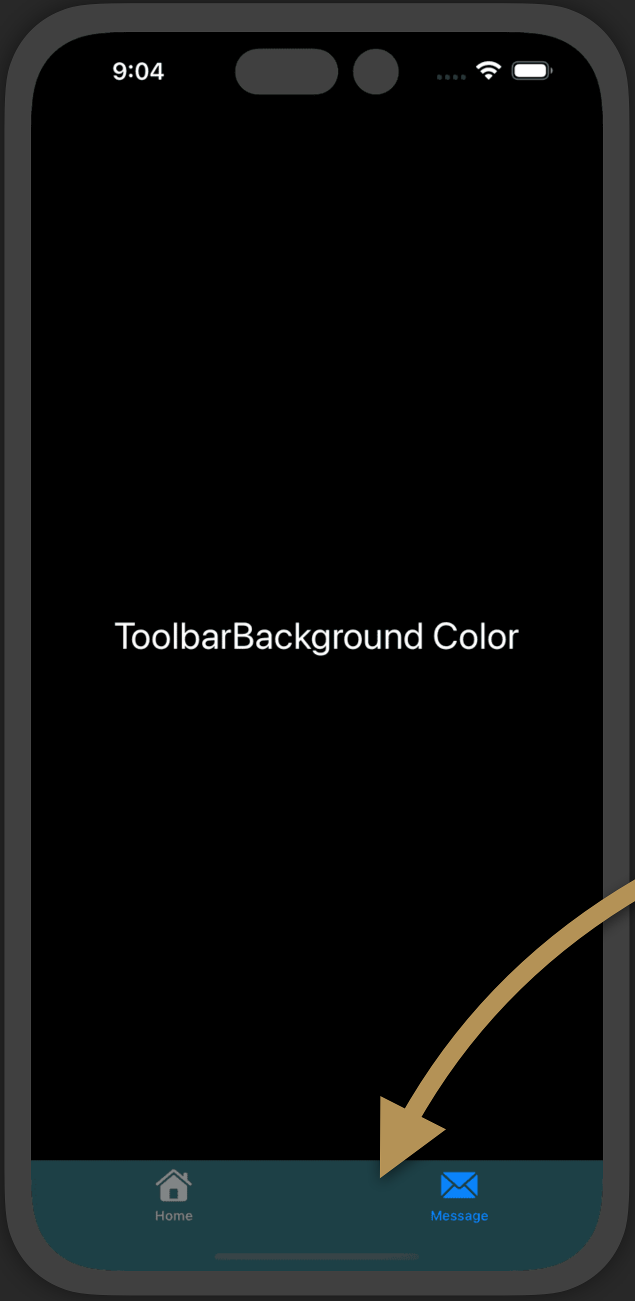
This modifier has no effect when used on the TabView.
You have to use it on a View **within** a Tab.

⚠️ iOS 18

For **iOS 18+**, you will want to use .toolbarBackgroundVisibility modifier.

Toolbar Background Visibility & Color

iOS 18



```
struct TabView_ToolbarBackgroundVisibility: View {
    var body: some View {
        TabView {
            Tab("Home", systemImage: "house") {
                Text("ToolbarBackground Visibility")
                    .toolbarBackgroundVisibility(.visible, for: .tabBar)
            }

            Tab("Message", systemImage: "envelope") {
                VStack {
                    Text("ToolbarBackground Color")
                }
                .toolbarBackground(.teal.opacity(0.25), for: .tabBar)
                .toolbarBackgroundVisibility(.visible, for: .tabBar)
            }
        }
        .font(.title)
    }
}
```

In iOS 18, toolbarBackgroundVisibility is used to show the system default background all the time.

Add a toolbar background color with the visibility to make the color persist all the time.

If you only want to show the color when content is scrolled behind it, then remove the visibility.

iOS 14

TabView - Paging Style

This SwiftUI chapter is locked in this preview.

With the existing TabView, [illegible] ow offers a new style [illegible] that allows the views within a TabView to be able to be swiped horizontally and “snap” into place when the view enters the screen.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY ~~$55~~ **$49.50**!

Text

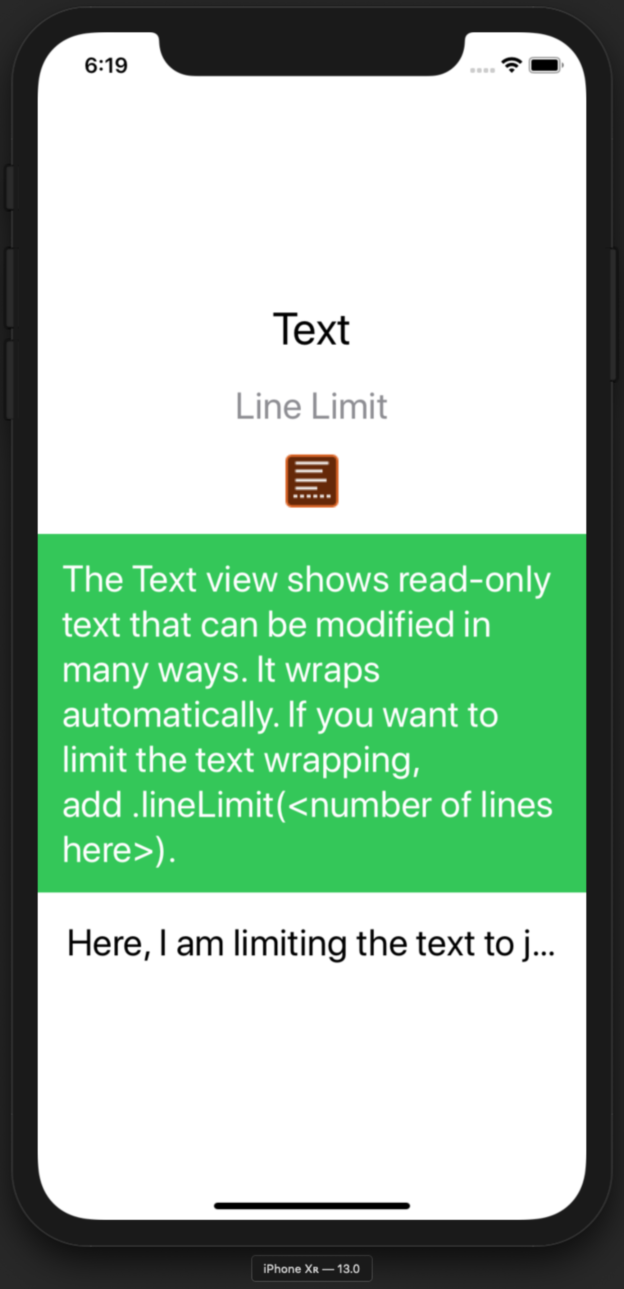
The text view will probably be one of your most-used views. It has many, if not the most, modifiers available to it.

This is a pull-in view.

Text

Line Limit

Text


6:19

Text

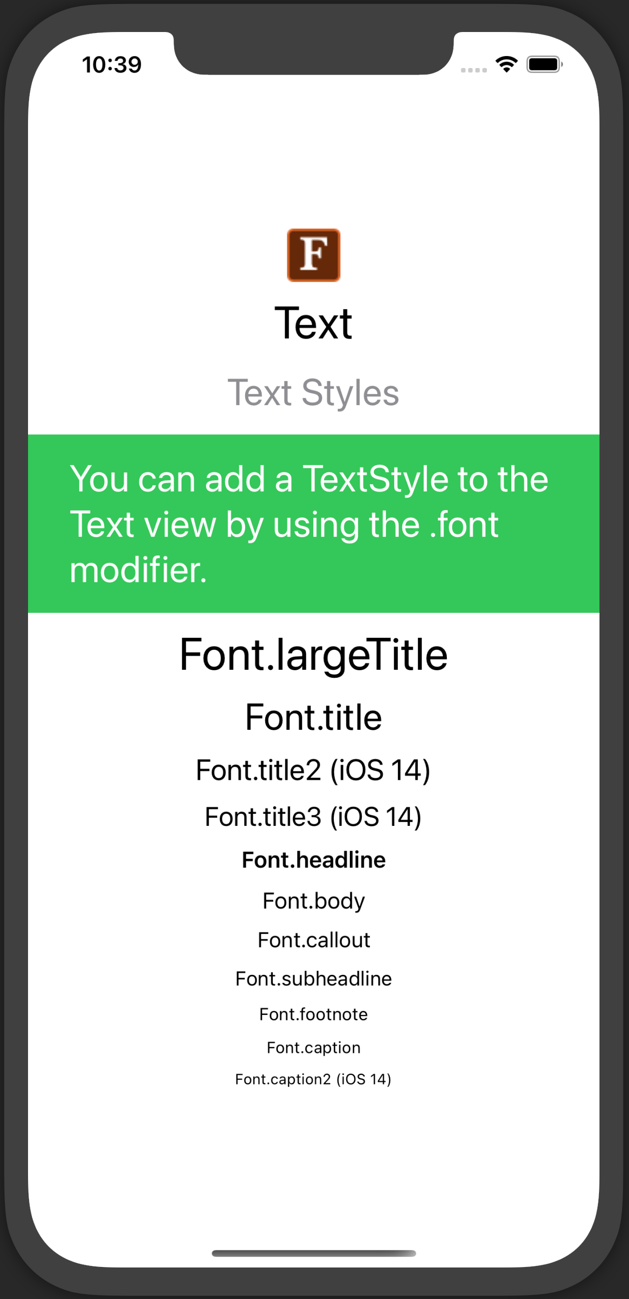
Line Limit

The Text view shows read-only text that can be modified in many ways. It wraps automatically. If you want to limit the text wrapping, add .lineLimit(<number of lines here>).

Here, I am limiting the text to j...

```
VStack(spacing: 20) {
    Text("Text")
        .font(.largeTitle)

    Text("Line Limit")
        .font(.title)
        .foregroundColor(.gray)

    Image("LineLimit")

    Text("The Text view shows read-only text that can be modified in many ways. It wraps 
automatically. If you want to limit the text wrapping, add .lineLimit(<number of lines here>).")
        ...

    Text("Here, I am limiting the text to just one line.")
        .lineLimit(1)
        .font(.title)
        .padding(.horizontal)
}
```

Text Styles


10:39

Text

Text Styles

You can add a TextStyle to the Text view by using the .font modifier.

Font.largeTitle

Font.title

Font.title2 (iOS 14)

Font.title3 (iOS 14)

Font.headline

Font.body

Font.callout

Font.subheadline

Font.footnote

Font.caption

Font.caption2 (iOS 14)

```
struct Text_TextStyles: View {
    var body: some View {
        VStack(spacing: 10) {
            Image("Font")

            HeaderView("Text",
                       subtitle: "Text Styles",
                       desc: "You can add a TextStyle to the Text view by using the .font
                              modifier.",
                       back: .green, textColor: .white)
                .font(.title)


            Group {
                Text("Font.largeTitle").font(.largeTitle)
                Text("Font.title").font(.title)
                Text("Font.title2 (iOS 14)").font(.title2)
                Text("Font.title3 (iOS 14)").font(.title3)
            }
            Group {
                Text("Font.headline").font(.headline)
                Text("Font.body").font(.body)
                Text("Font.callout").font(.callout)
                Text("Font.subheadline").font(.subheadline)
                Text("Font.footnote").font(.footnote)
                Text("Font.caption").font(.caption)
                Text("Font.caption2 (iOS 14)").font(.caption2)
            }
        }
    }
}
```

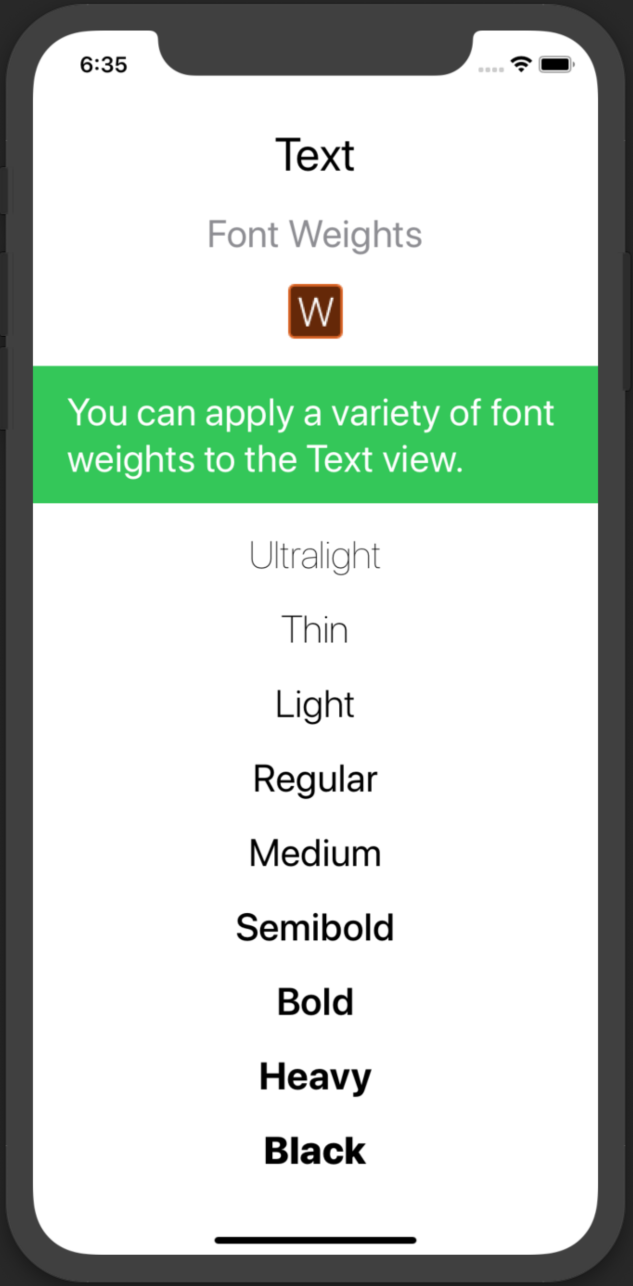
iOS 14

iOS 14

Text

Weights

Text



Text

Font Weights

You can apply a variety of font weights to the Text view.

Ultralight

Thin

Light

Regular

Medium

Semibold

Bold

Heavy

Black

```
Text("Text")
    .font(.largeTitle)
Text("Font Weights")
    .font(.title)
    .foregroundColor(.gray)
Image("FontWeight")
Text("You can apply a variety of font weights to the Text view.")
    .padding()
    .frame(maxWidth: .infinity)
    .background(Color.green)
    .foregroundColor(.white)
    .font(.title)

Group { // Too many views (> 10) in one container
    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)
}.font(.title)
```

Note: The fontWeight modifier can ONLY be applied to Text views.

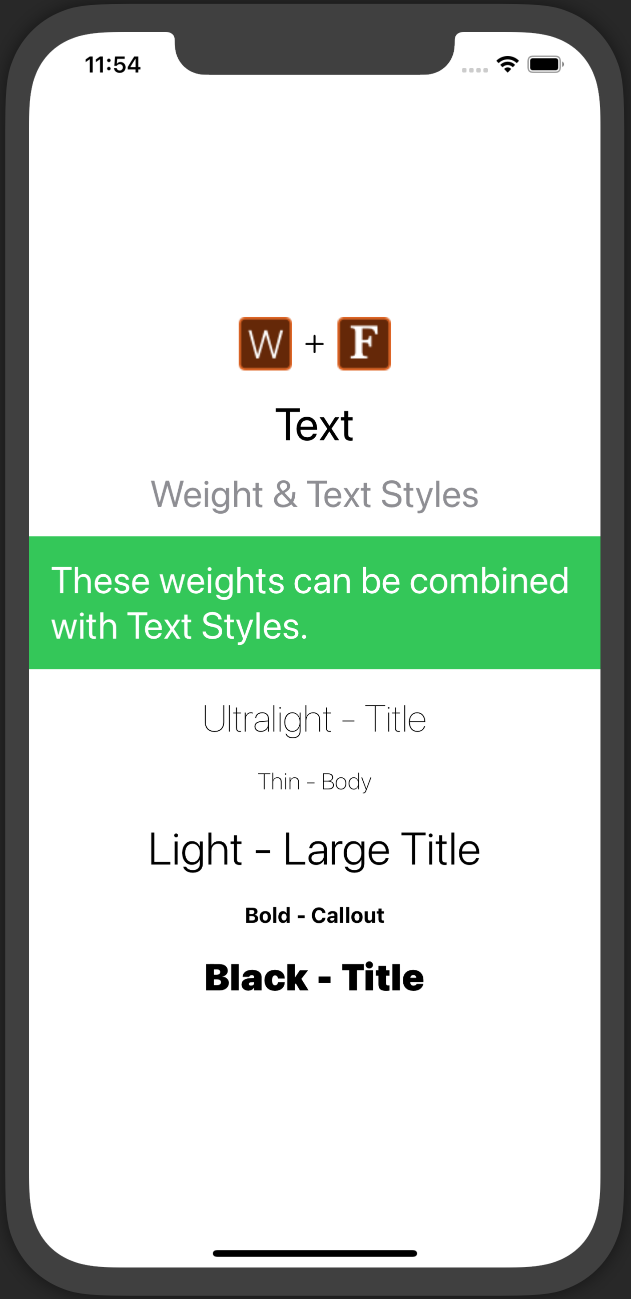
Unlike the font modifier which can be applied to any view.

To apply weight to any view using the font modifier, see next page.

Text

Weight & Text Style Combined

Text


11:54

W + F

Text

Weight & Text Styles

These weights can be combined with Text Styles.

Ultralight - Title

Thin - Body

Light - Large Title

Bold - Callout

Black - Title

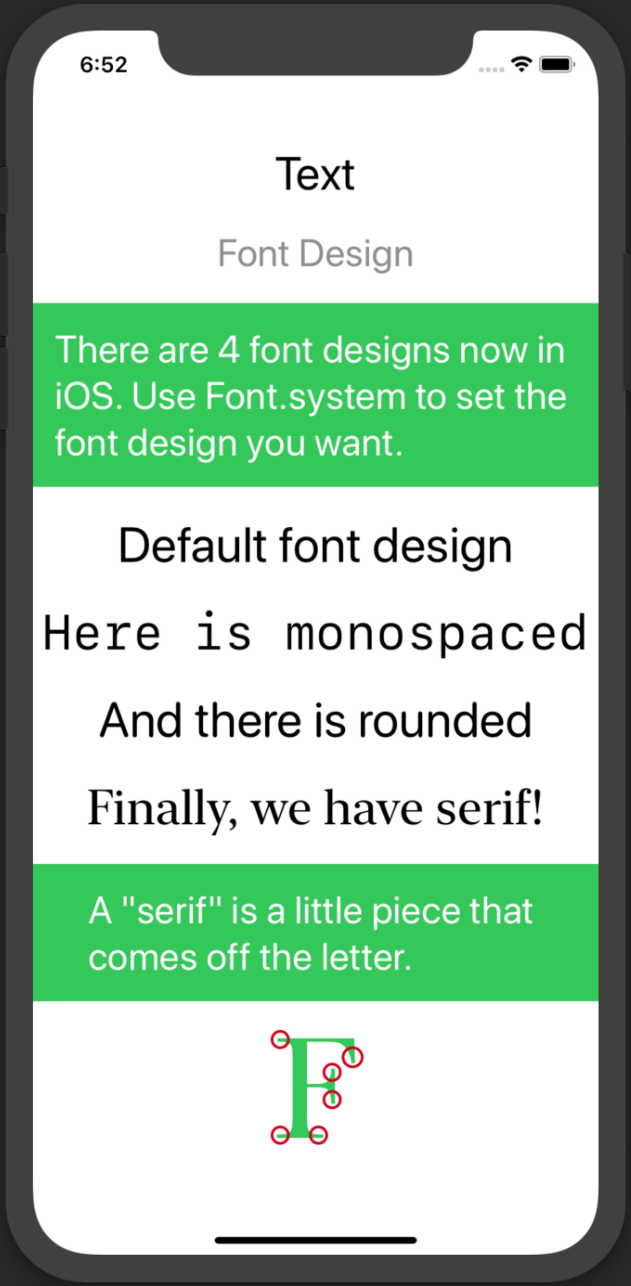
```
struct Text_Weights_TextStyles: View {
    var body: some View {
        return VStack(spacing: 20) {
            HStack {
                Image("FontWeight")
                Image(systemName: "plus")
                Image("Font")
            }

            HeaderView("Text", subtitle: "Weight & Text Styles",
                       desc: "These weights can be combined with Text Styles.",
                       back: .green, textColor: .white)
                .font(.title)

            Text("Ultralight - Title")
                .fontWeight(.ultraLight)
                .font(.title)
            Text("Thin - Body")
                .fontWeight(.thin)
                .font(.body)
            Text("Light - Large Title")
                .fontWeight(.light)
                .font(.largeTitle)
            Text("Bold - Callout")
                .fontWeight(.bold)
                .font(.callout)
            Text("Black - Title")
                .font(Font.title.weight(.black))
        }
    }
}
```

Note: Instead of two modifiers, you can combine text style and weight in just ONE modifier like this.

Text

Font Design

Text


6:52

Text

Font Design

There are 4 font designs now in iOS. Use Font.system to set the font design you want.

Default font design

Here is monospaced

And there is rounded

Finally, we have serif!

A "serif" is a little piece that comes off the letter.

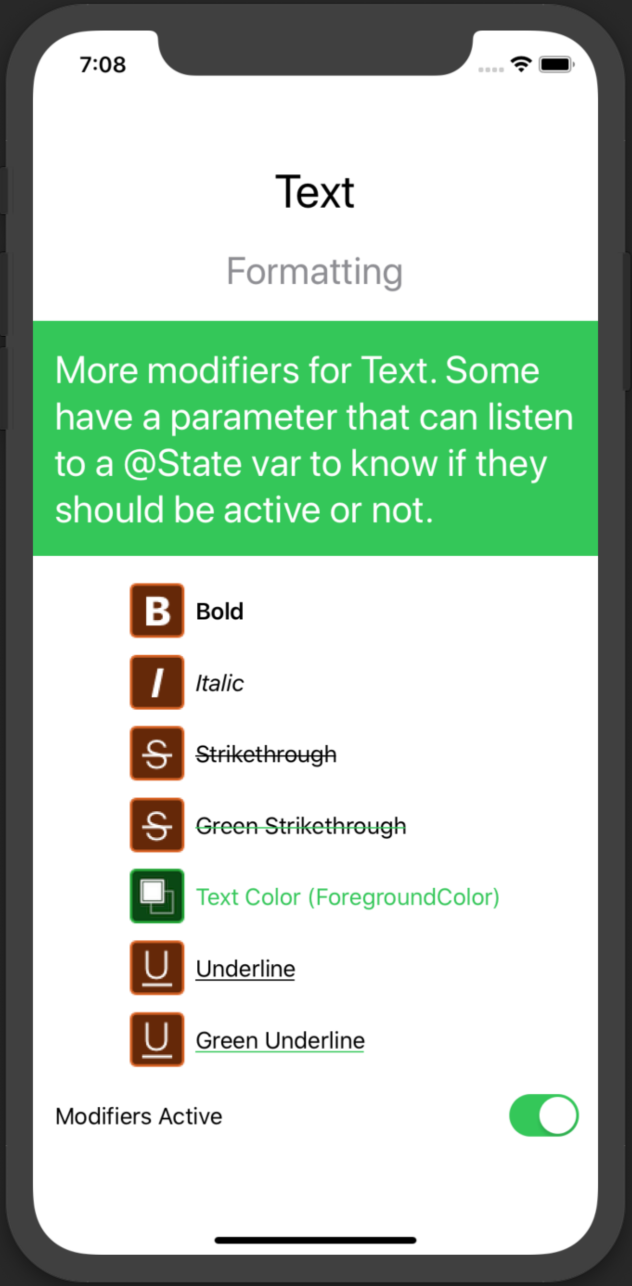
```
struct Text_FontDesign : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("Text", subtitle: "Font Design", desc: "There are 4 font designs now in
                       iOS. Use Font.system to set the font design you want.",
                        back: .green, textColor: .white)

            Text("Default font design")
                .font(Font.system(size: 30, design: Font.Design.default))

            // You can remove the "Font.Design" of the enum
            Text("Here is monospaced")
                .font(.system(size: 30, design: .monospaced))

            Text("And there is rounded")
                .font(.system(.title, design: .rounded))

            Text("Finally, we have serif!")
                .font(.system(.title, design: .serif))

            DescView(desc: "A \"serif\" is a little piece that comes off the letter.",
                back: .green, textColor: .white)

            Image("Serif")
        }
        .font(.title)
    }
}
```

Set the design with a hard-coded size or use a text style.

Text

Formatting

Text


7:08

Text

Formatting

More modifiers for Text. Some have a parameter that can listen to a @State var to know if they should be active or not.

Bold

Italic

~~Strikethrough~~

~~Green Strikethrough~~

Text Color (ForegroundColor)

Underline

Green Underline

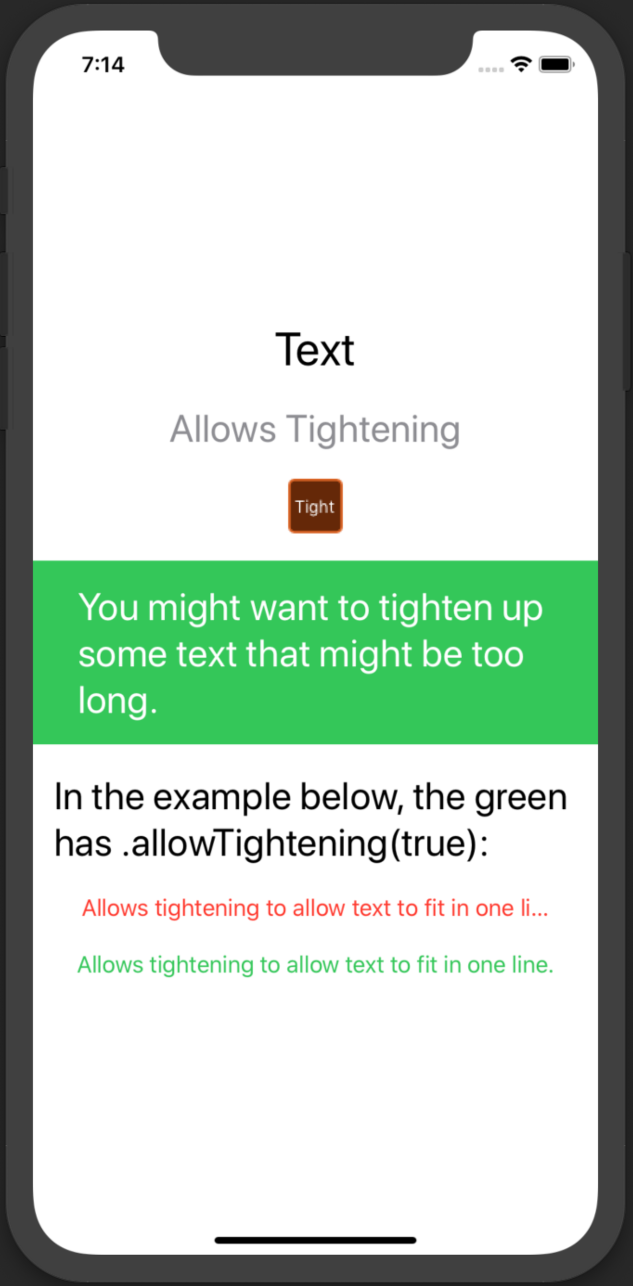
Modifiers Active

```
@State private var modifierActive = true
...
HStack {
    Image("Bold")
    Text("Bold").bold()
}
HStack {
    Image("Italic")
    Text("Italic").italic()
}
HStack {
    Image("Strikethrough")
    Text("Strikethrough").strikethrough()
}
HStack {
    Image("Strikethrough")
    Text("Green Strikethrough")
        .strikethrough(modifierActive, color: .green)
}
HStack {
    Image("ForegroundColor")
    Text("Text Color (ForegroundColor)").foregroundColor(.green)
}
HStack {
    Image("Underline")
    Text("Underline").underline()
}
HStack {
    Image("Underline")
    Text("Green Underline").underline(modifierActive, color: .green)
}
...
Toggle("Modifiers Active", isOn: $modifierActive)
```

Text

Allows Tightening

Text


Text

Allows Tightening

You might want to tighten up some text that might be too long.

In the example below, the green has .allowTightening(true):

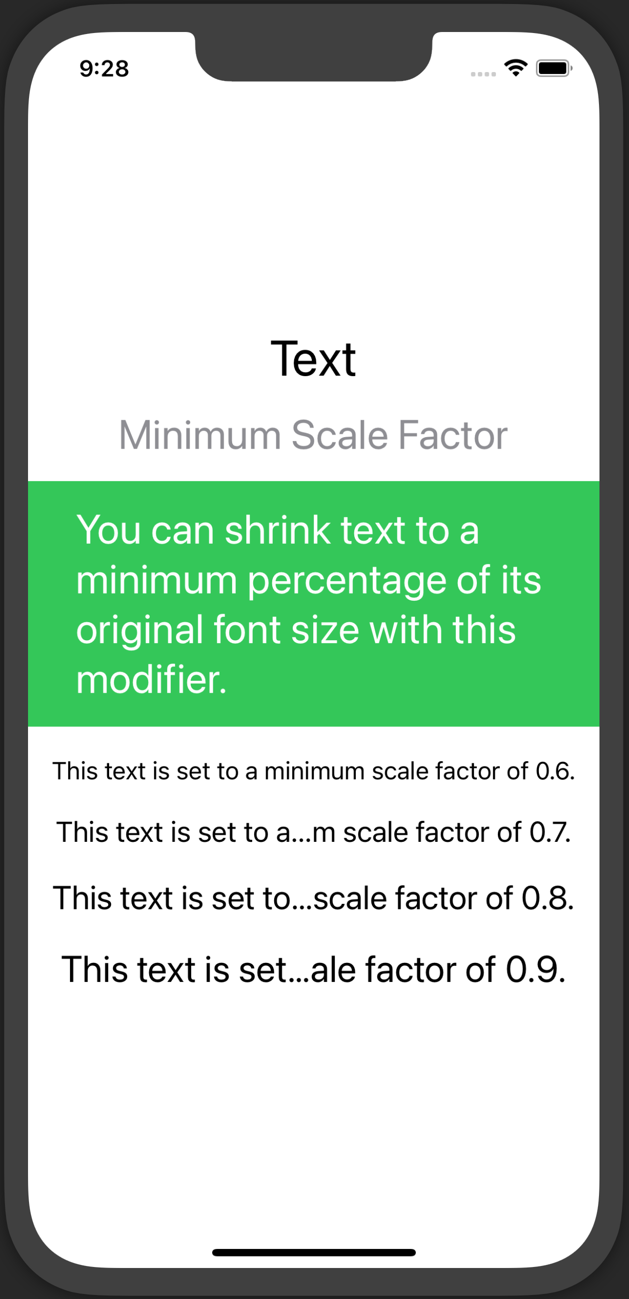
Allows tightening to allow text to fit in one li...

Allows tightening to allow text to fit in one line.

```
VStack(spacing: 20) {
...

    Image("AllowsTightening")

    Text("You might want to tighten up some text that might be too long.")
        ...

    Text("In the example below, the green has .allowTightening(true)")
        ...

    Group {
        Text("Allows tightening to allow text to fit in one line.")
            .foregroundColor(.red)
            .allowsTightening(false)
            .padding(.horizontal)
            .lineLimit(1)
        Text("Allows tightening to allow text to fit in one line.")
            .foregroundColor(.green)
            .allowsTightening(true)
            .padding(.horizontal)
            .lineLimit(1)
    }.padding(.horizontal)
}
```

Allows Tightening can be helpful when you see the last word getting truncated. Applying it may not even fully work depending on just how much space can be tightened. With the default font, I notice I can get a couple of characters worth of space to tighten up.

Minimum Scale Factor


9:28

Text

Minimum Scale Factor

You can shrink text to a minimum percentage of its original font size with this modifier.

This text is set to a minimum scale factor of 0.6.

This text is set to a...m scale factor of 0.7.

This text is set to...scale factor of 0.8.

This text is set...ale factor of 0.9.

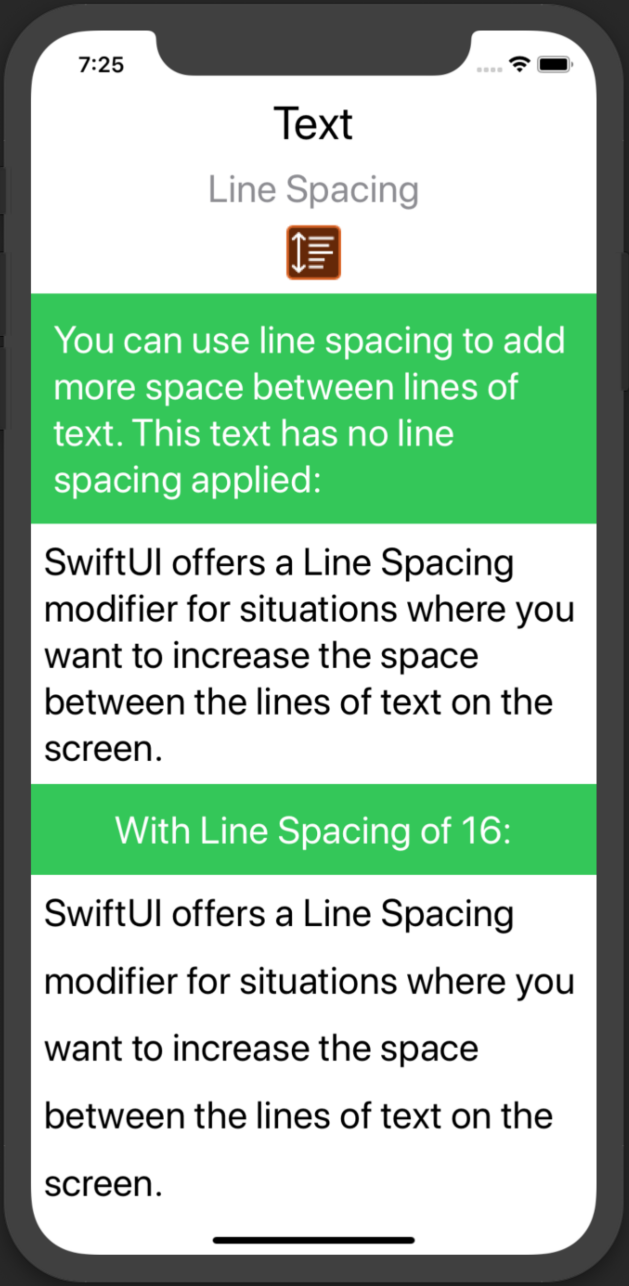
```
struct Text_MinimumScaleFactor : View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Text",
                       subtitle: "Minimum Scale Factor",
                       desc: "You can shrink text to a minimum percentage of its original font
                              size with this modifier.",
                       back: .green, textColor: .white)

            Group {
                Text("This text is set to a minimum scale factor of 0.6.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.6)
                Text("This text is set to a minimum scale factor of 0.7.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.7)
                Text("This text is set to a minimum scale factor of 0.8.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.8)
                Text("This text is set to a minimum scale factor of 0.9.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.9)
            }
            .truncationMode(.middle)
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

.minimumScaleFactor takes a fraction from 0 to 1. For example, 0.3 is 30% of the original size of the font that it can shrink. If the font size is 100, then it can shrink to 30.

Text

Line Spacing

Text



Text

Line Spacing

You can use line spacing to add more space between lines of text. This text has no line spacing applied:

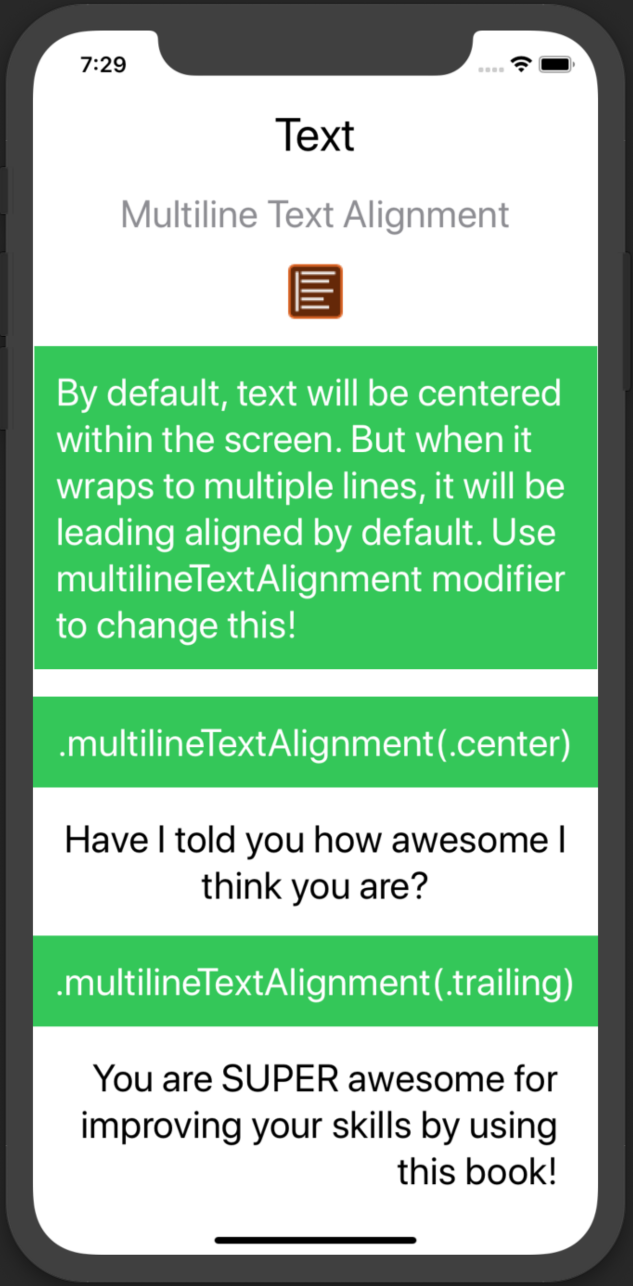
SwiftUI offers a Line Spacing modifier for situations where you want to increase the space between the lines of text on the screen.

With Line Spacing of 16:

SwiftUI offers a Line Spacing modifier for situations where you want to increase the space between the lines of text on the screen.

```
VStack(spacing: 10) {
    Text("Text").font(.largeTitle)
    Text("Line Spacing").font(.title).foregroundColor(.gray)
    Image("LineSpacing")

    Text("You can use line spacing to add more space between lines of text. This text has no 
line spacing applied:")
        .font(.title)
        .frame(maxWidth: .infinity)
        .padding()
        .background(Color.green)
        .foregroundColor(Color.white)

    Text("SwiftUI offers a Line Spacing modifier for situations where you want to increase the 
space between the lines of text on the screen.")
        .font(.title)

    Text("With Line Spacing of 16:")
        .font(.title)
        .frame(maxWidth: .infinity)
        .padding()
        .background(Color.green)
        .foregroundColor(Color.white)

    Text("SwiftUI offers a Line Spacing modifier for situations where you want to increase the 
space between the lines of text on the screen.")
        .lineSpacing(16.0) // Add spacing between lines
        .font(.title)
}
```

Alignment



7:29

Text

Multiline Text Alignment

By default, text will be centered within the screen. But when it wraps to multiple lines, it will be leading aligned by default. Use multilineTextAlignment modifier to change this!

.multilineTextAlignment(.center)

Have I told you how awesome I think you are?

.multilineTextAlignment(.trailing)

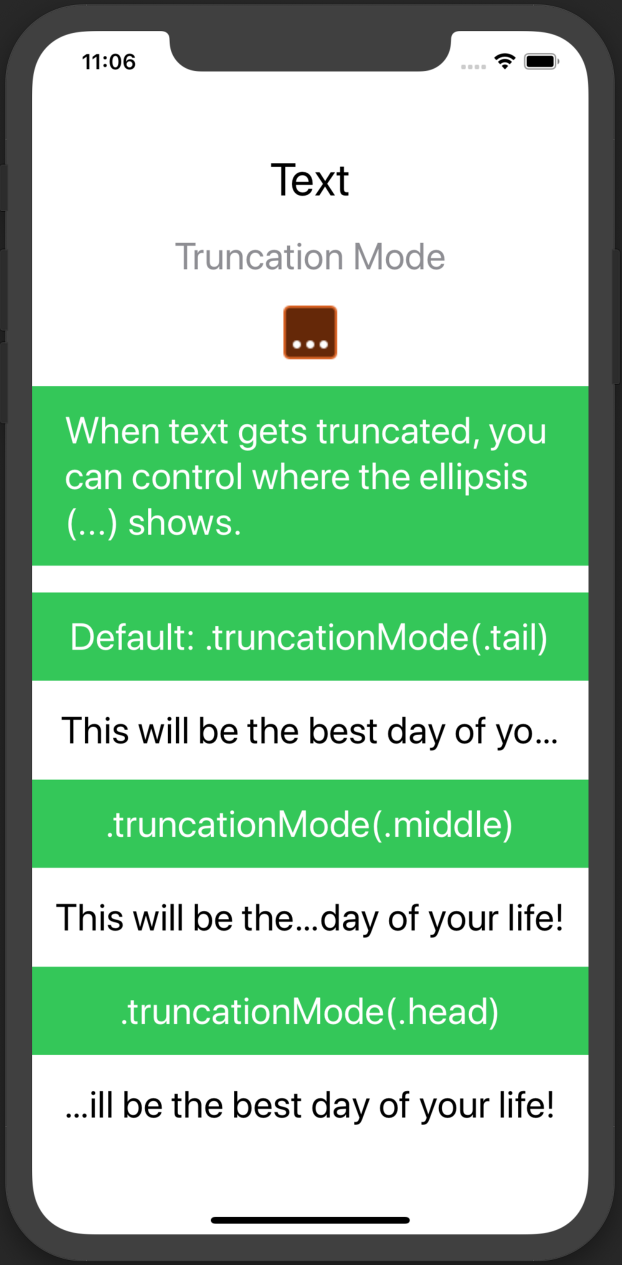
You are SUPER awesome for improving your skills by using this book!

```swift
VStack(spacing: 20) {
    Text("Text").font(.largeTitle)
    Text("Multiline Text Alignment").foregroundColor(.gray)
    Image("MultilineTextAlignment")
    Text("By default, text will be centered within the screen. But when it wraps to multiple
lines, it will be leading aligned by default. Use multilineTextAlignment modifier to change
this!")
        ...

    Text(".multilineTextAlignment(.center)")
        .frame(maxWidth: .infinity)
        .padding()
        .foregroundColor(.white)
        .background(Color.green)

    Text("Have I told you how awesome I think you are?")
        .multilineTextAlignment(.center) // Center align
        .padding(.horizontal)

    Text(".multilineTextAlignment(.trailing)")
        .frame(maxWidth: .infinity)
        .padding()
        .foregroundColor(.white)
        .background(Color.green)
        .allowsTightening(true) // Prevent truncation

    Text("You are SUPER awesome for improving your skills by using this book!")
        .multilineTextAlignment(.trailing) // Trailing align
        .padding(.horizontal)
}
.font(.title) // Apply this text style to all text views
```

Text

Truncation Mode

Text


11:06

Text

Truncation Mode

When text gets truncated, you can control where the ellipsis (...) shows.

Default: .truncationMode(.tail)

This will be the best day of yo...

.truncationMode(.middle)

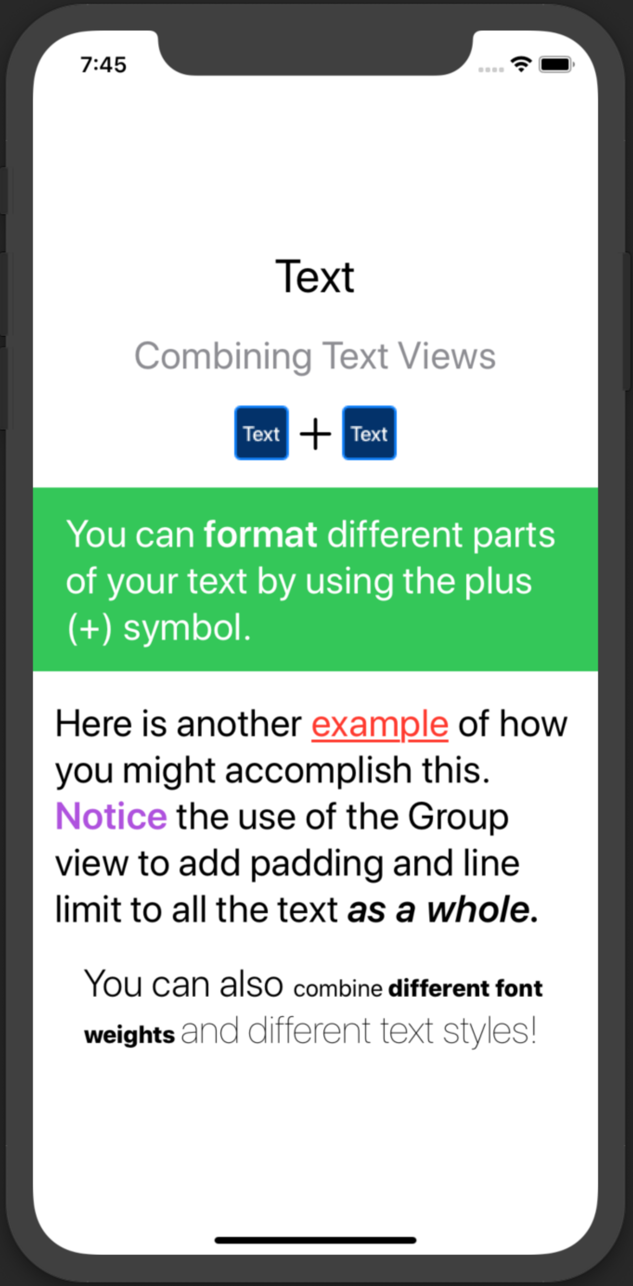
This will be the...day of your life!

.truncationMode(.head)

...ill be the best day of your life!

```
VStack(spacing: 20) {
    Text("Text").font(.largeTitle)
    Text("Truncation Mode").font(.title).foregroundColor(.gray)
    Image("TruncationMode")
    Text("When text gets truncated, you can control where the ellipsis (...) shows.")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
        .font(.title)
    Text("Default: .truncationMode(.tail)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
        .font(.title)
    // Text automatically defaults at end
    Text("This will be the best day of your life!")
        .padding(.horizontal)
        .lineLimit(1)
    Text(".truncationMode(.middle)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
    Text("This will be the best day of your life!")
        .truncationMode(.middle) // Truncate in middle
        .padding(.horizontal)
        .lineLimit(1)
    Text(".truncationMode(.head)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
    Text("This will be the best day of your life!")
        .truncationMode(.head) // Truncate at beginning
        .padding(.horizontal)
        .lineLimit(1)
}
.font(.title)
```

Combining Modified Text


7:45

Text

Combining Text Views

You can **format** different parts of your text by using the plus (+) symbol.

Here is another example of how you might accomplish this. **Notice** the use of the Group view to add padding and line limit to all the text ***as a whole.***

You can also combine **different font weights** and different text styles!

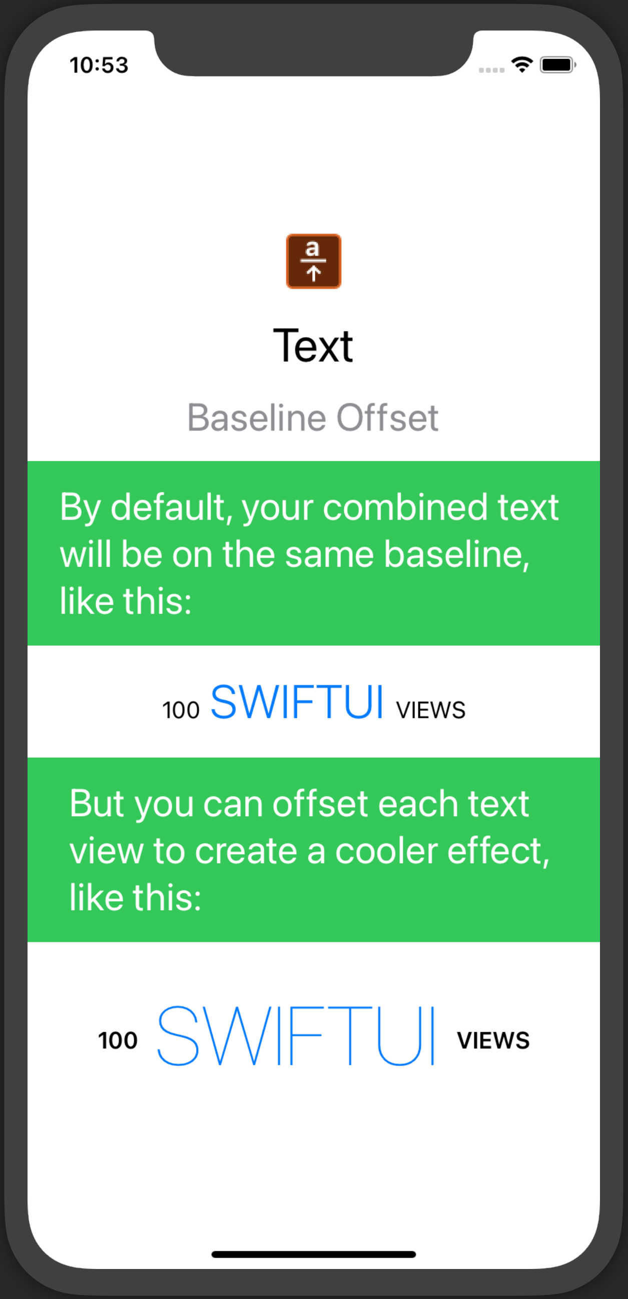
```
Group {
    Text("You can ")
        + Text("format").bold()
        + Text (" different parts of your text by using the plus (+) symbol.")
}
...
Group {
    Text("Here is another ")
        + Text("example").foregroundColor(.red).underline()
        + Text (" of how you might accomplish this. ")
        + Text("Notice").foregroundColor(.purple).bold()
        + Text (" the use of the Group view to add padding and line limit to all the text ")
        + Text("as a whole.").bold().italic()
}
.font(.title)
.padding(.horizontal)

Group {
    Text("You can also ").font(.title).fontWeight(.light)
        + Text("combine")
        + Text(" different font weights ").fontWeight(.black)
        + Text("and different text styles!").font(.title).fontWeight(.ultraLight)
}
.padding(.horizontal)
```

Although you see I'm wrapping my Text views in a Group, it is not required. I only do this so I can apply common modifiers to everything within the Group. See section on the Group view for more information.

Text

Baseline Offset

Text


10:53

Text

Baseline Offset

By default, your combined text will be on the same baseline, like this:

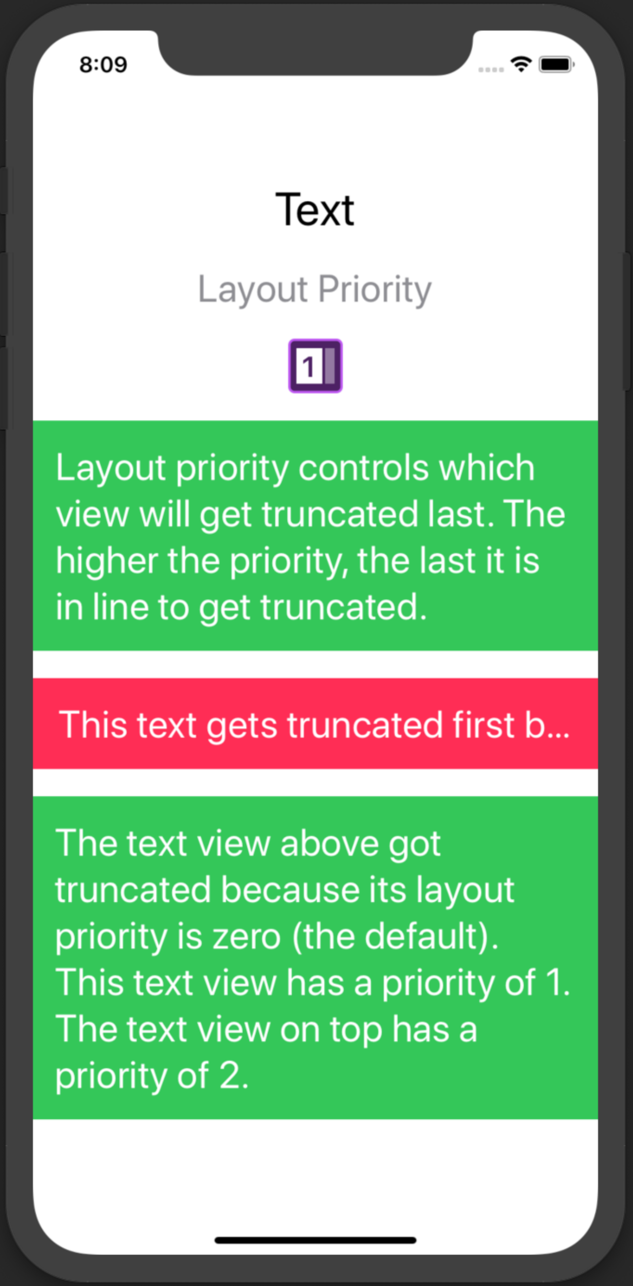
100 SWIFTUI VIEWS

But you can offset each text view to create a cooler effect, like this:

100 SWIFTUI VIEWS

```
struct Text_BaselineOffset : View {
    var body: some View {
        VStack(spacing: 20) {
            Image("BaselineOffset")
            HeaderView("Text",
                       subtitle: "Baseline Offset",
                       desc: "By default, your combined text will be on the same baseline, like
this:", back: .green, textColor: .white)
                .font(.title)
                .layoutPriority(1)

            Text("100")
                + Text(" SWIFTUI ").font(.largeTitle).fontWeight(.light)
                .foregroundColor(.blue)
                + Text ("VIEWS")

            DescView(desc: "But you can offset each text view to create a cooler effect, like
this:", back: .green, textColor: .white)
                .font(.title)

            Group {
                Text("100").bold()
                    + Text(" SWIFTUI ")
                    .font(Font.system(size: 60))
                    .fontWeight(.ultraLight)
                    .foregroundColor(.blue)
                    .baselineOffset(-12) // Negative numbers make it go down
                    + Text ("VIEWS").bold()
            }
        }
    }
}
```

Layout Priority



Text

Layout Priority

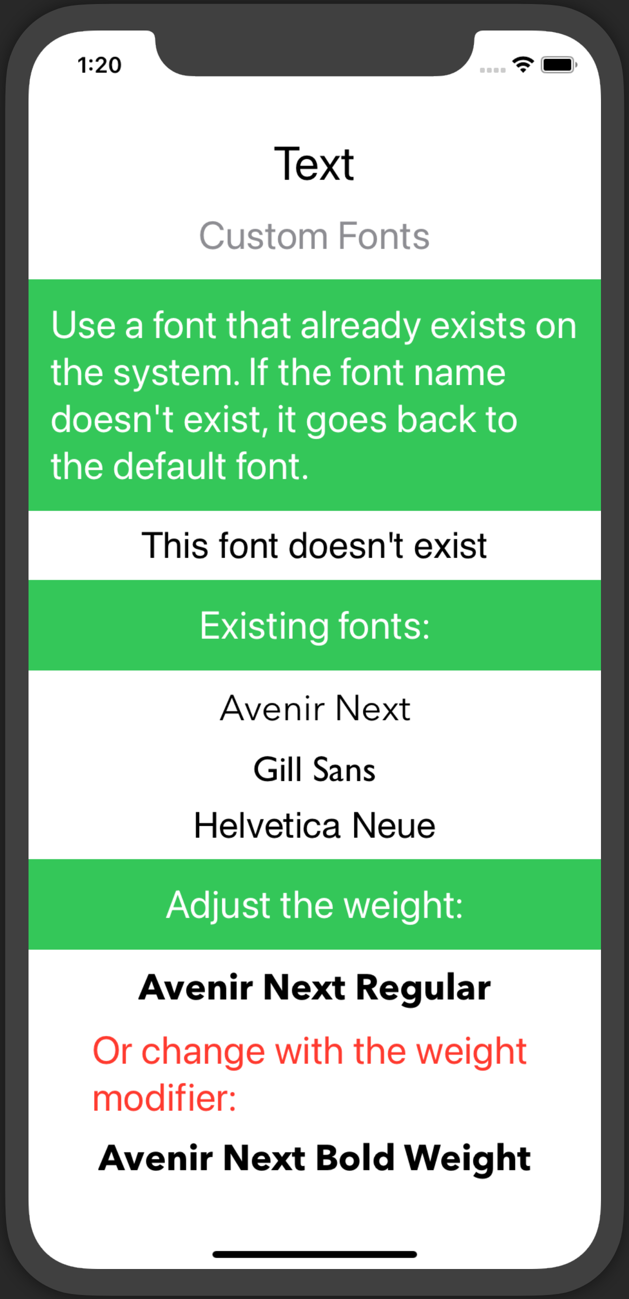
Layout priority controls which view will get truncated last. The higher the priority, the last it is in line to get truncated.

This text gets truncated first b...

The text view above got truncated because its layout priority is zero (the default). This text view has a priority of 1. The text view on top has a priority of 2.

```
Text("Text")
    .font(.largeTitle)
Text("Layout Priority")
    .font(.title)
    .foregroundColor(.gray)

Image("LayoutPriority")

Text("Layout priority controls which view will get truncated last. The higher the priority, the
last it is in line to get truncated.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.green)
    .layoutPriority(2) // Highest priority to get the space it needs

Text("This text gets truncated first because it has no priority.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.pink)

Text("The text view above got truncated because its layout priority is zero (the default). This
text view has a priority of 1. The text view on top has a priority of 2.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.green)
    .layoutPriority(1) // Next highest priority
```

Custom Fonts


1:20

Text

Custom Fonts

Use a font that already exists on the system. If the font name doesn't exist, it goes back to the default font.

This font doesn't exist

Existing fonts:

Avenir Next

Gill Sans

Helvetica Neue

Adjust the weight:

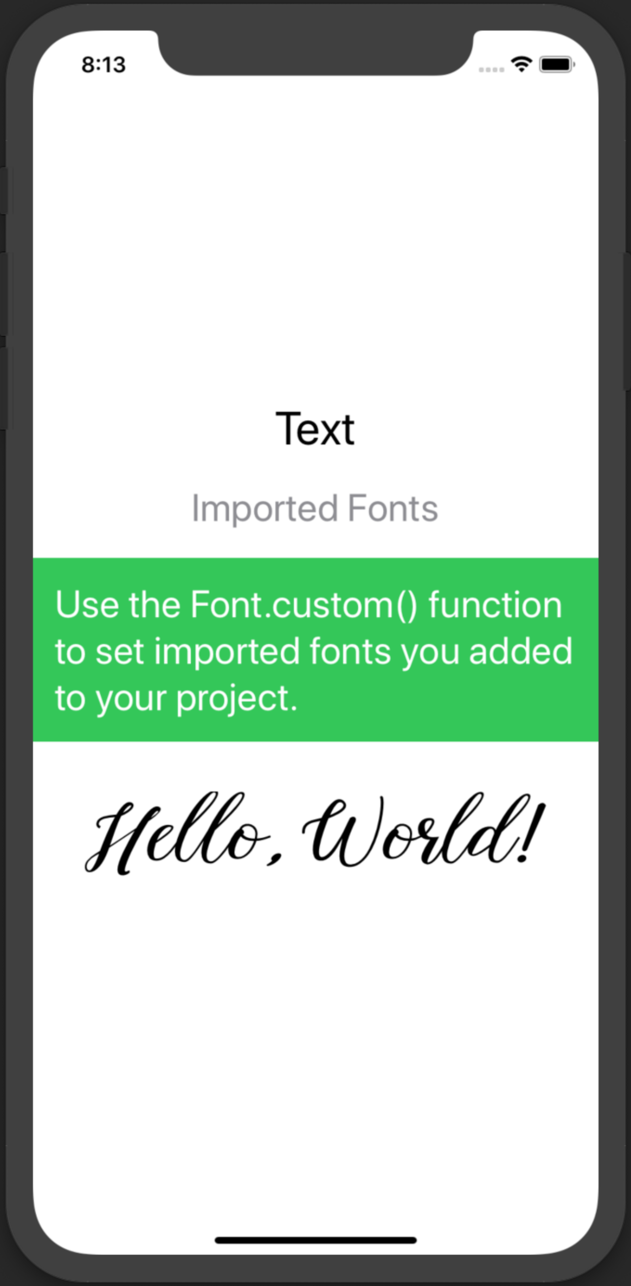
Avenir Next Regular

Or change with the weight modifier:

Avenir Next Bold Weight

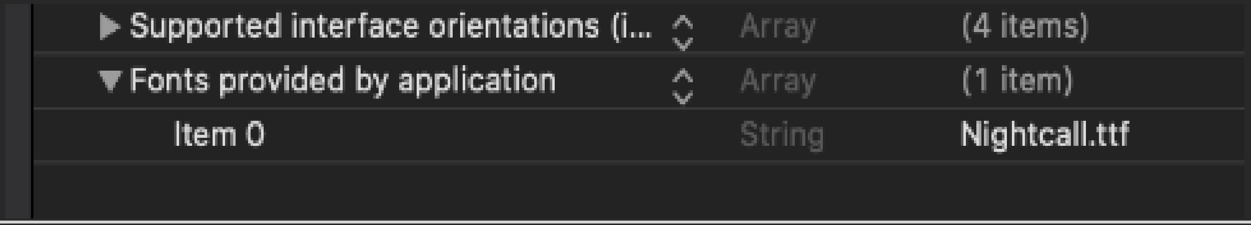
```
struct Text_CustomFont: View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("Text",
                       subtitle: "Custom Fonts",
                       desc: "Use a font that already exists on the system. If the font name
doesn't exist, it goes back to the default font.", back: .green, textColor: .white)

            Text("This font doesn't exist")
                .font(Font.custom("No Such Font", size: 26))

            DescView(desc: "Existing fonts:", back: .green, textColor: .white)

            Text("Avenir Next")
                .font(Font.custom("Avenir Next", size: 26))

            Text("Gill Sans")
                .font(Font.custom("Gill Sans", size: 26))

            Text("Helvetica Neue")
                .font(Font.custom("Helvetica Neue", size: 26))

            DescView(desc: "Adjust the weight:", back: .green, textColor: .white)

            Text("Avenir Next Regular")
                .font(Font.custom("Avenir Next Bold", size: 26))

            Text("Or change with the weight modifier:")
                .foregroundColor(.red)

            Text("Avenir Next Bold Weight")
                .font(Font.custom("Avenir Next", size: 26).weight(.bold))
        }
        .font(.title)
        .ignoresSafeArea(edges: .bottom)
    }
}
```

Imported Fonts


8:13
Text
Imported Fonts
Use the Font.custom() function to set imported fonts you added to your project.
Hello, World!

```
struct Text_CustomFont: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Text")
                .font(.largeTitle)

            Text("Imported Fonts")
                .font(.title)
                .foregroundColor(.gray)

            Text("Use the Font.custom() function to set imported fonts you added to your 
project.")
                ...

            Text("Hello, World!")
                .font(Font.custom("Nightcall", size: 60))
                .padding(.top)
        }
    }
}
```

In order for this to work, you have to add the font file to your project and be sure to have the font file target your project. Then you need to add the font file name to the Info.plist under the "Fonts provided by application" key:



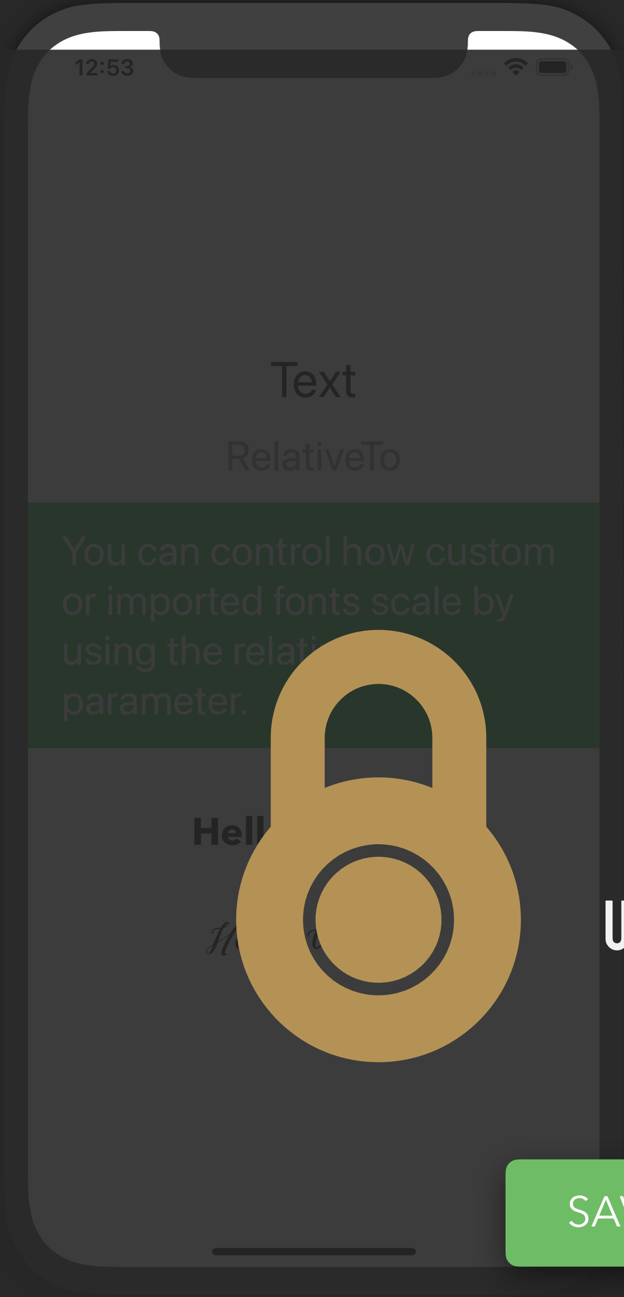
▶ Supported interface orientations (i...	Array	(4 items)
▼ Fonts provided by application	Array	(1 item)
Item 0	String	Nightcall.ttf

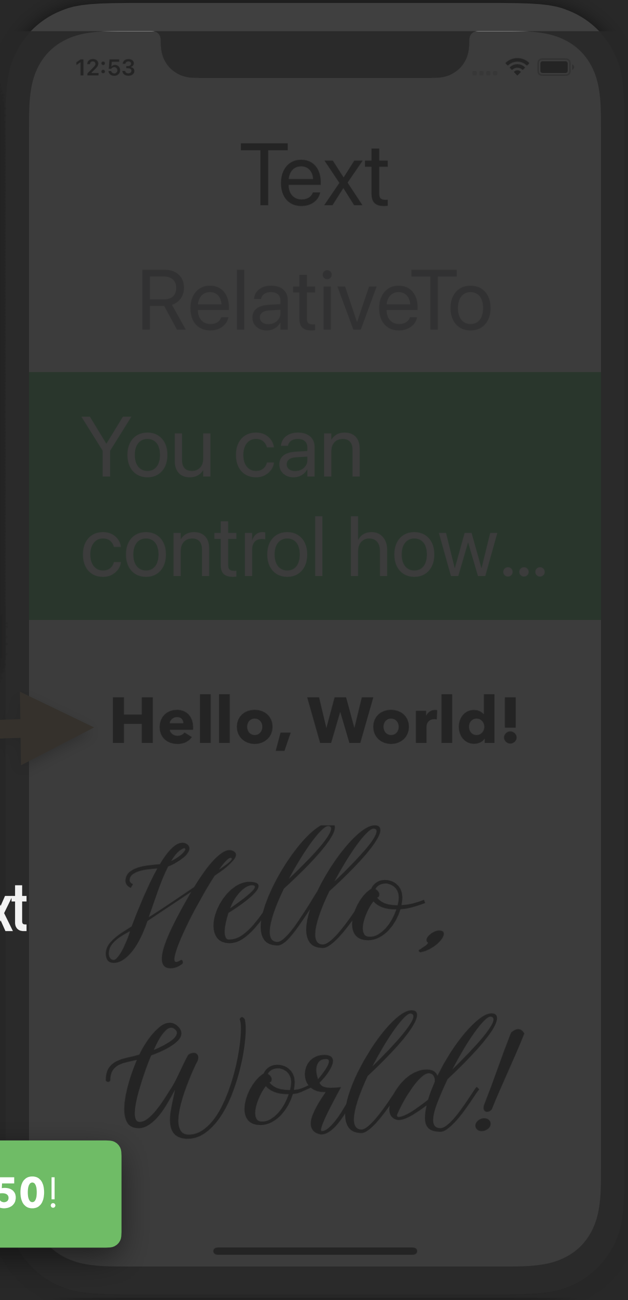
Text

Custom Font Size & RelativeTo

iOS 14 iOS 15 iOS 16 iOS 18 Text

```
struct Text_CustomFontSize_RelativeTo: View {
```


12:53
Text
RelativeTo
You can control how custom or imported fonts scale by using the relati
parameter.
Hell



iOS 14

TextEditor

You can use the TextEditor to provide text input from users that goes beyond just one line.

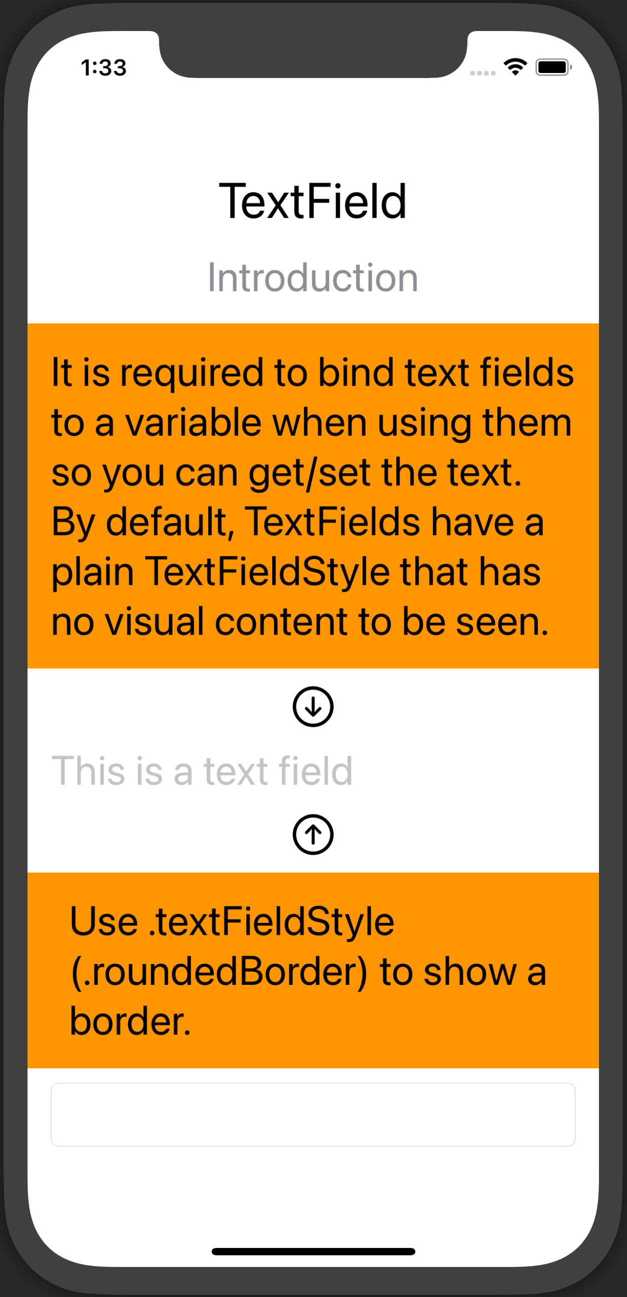
This is a push-out view.

TextField

In order to get or set the text in a TextField, you need to bind it to a variable. This variable is passed into the TextField's initializer. Then, all you need to do is change this bound variable's text to change what is in the TextField. Or read the bound variable's value to see what text is currently in the TextField.

This is a push-out horizontally view.

Introduction



TextField

Introduction

It is required to bind text fields to a variable when using them so you can get/set the text.
By default, TextFields have a plain TextFieldStyle that has no visual content to be seen.

This is a text field

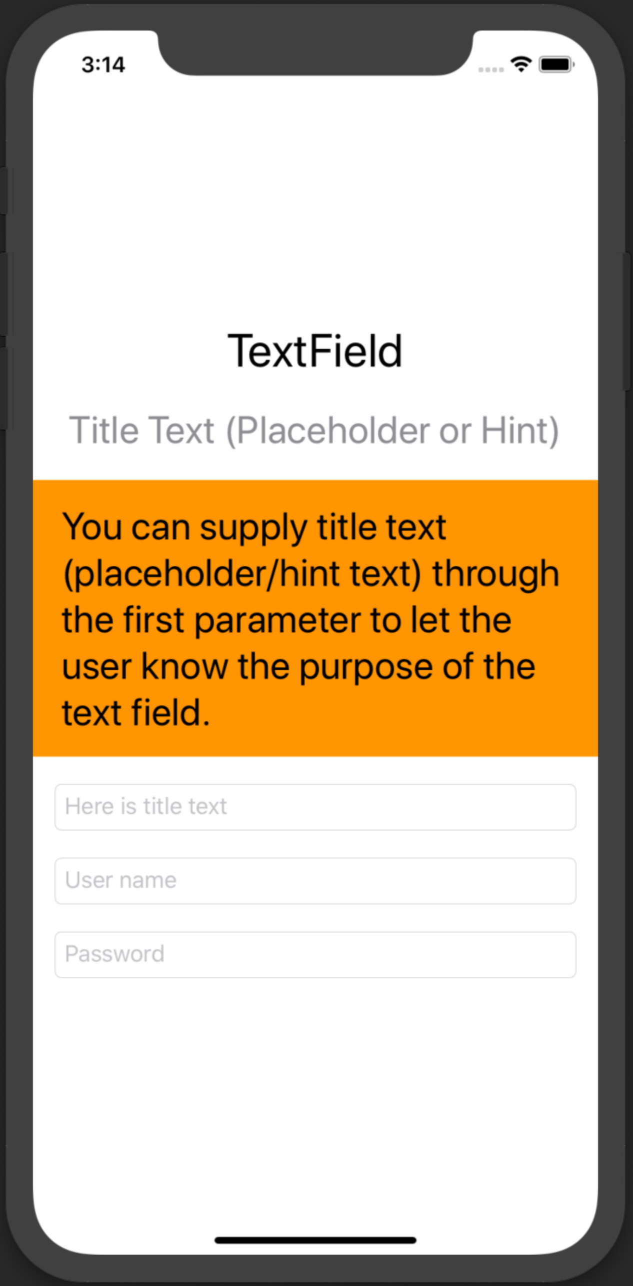
Use .textFieldStyle (.roundedBorder) to show a border.

```
struct TextField_Intro : View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 10) {
            HeaderView("TextField", subtitle: "Introduction",
                       desc: "It is required to bind text fields to a variable when using them
                              so you can get/set the text. \nBy default, TextFields have a plain
                              TextFieldStyle that has no visual content to be seen.",
                       back: .orange)
            Image(systemName: "arrow.down.circle")
            TextField("This is a text field", text: $textFieldData)
                .padding(.horizontal)
            Image(systemName: "arrow.up.circle")

            Text("Use .textFieldStyle (.roundedBorder) to show a border.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange)
            TextField("", text: $textFieldData)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)
        }
        .font(.title)
    }
}
```

For textFieldStyle, use:

< iOS 15	`RoundedBorderTextFieldStyle()`
iOS 15+	`.roundedBorder`

Title (Placeholder or Hint Text)


3:14

TextField

Title Text (Placeholder or Hint)

You can supply title text (placeholder/hint text) through the first parameter to let the user know the purpose of the text field.

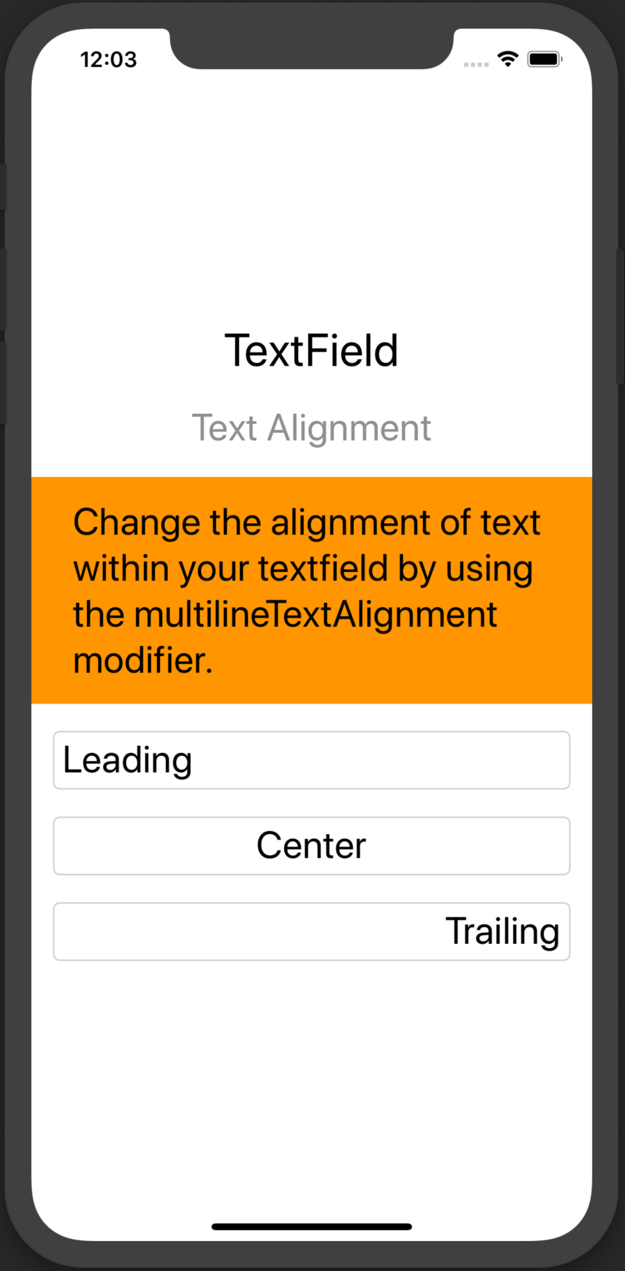
Here is title text

User name

Password

```
struct TextField_Placeholder : View {
    @State private var textFieldData = ""
    @State private var username = ""
    @State private var password = ""

    var body: some View {
        VStack(spacing: 20) {
            Text("TextField")
                .font(.largeTitle)

            Text("Title Text (Placeholder or Hint)")
                .foregroundColor(.gray)

            Text("You can supply title text (placeholder/hint text) through the first parameter
to let the user know the purpose of the text field.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange)

            Group {
                TextField("Here is title text", text: $textFieldData)
                    .textFieldStyle(.roundedBorder)

                TextField("User name", text: $username)
                    .textFieldStyle(.roundedBorder)

                TextField("Password", text: $password)
                    .textFieldStyle(.roundedBorder)
            }
            .padding(.horizontal)
        }.font(.title)
    }
}
```

Text Alignment


12:03

TextField

Text Alignment

Change the alignment of text within your textfield by using the multilineTextAlignment modifier.

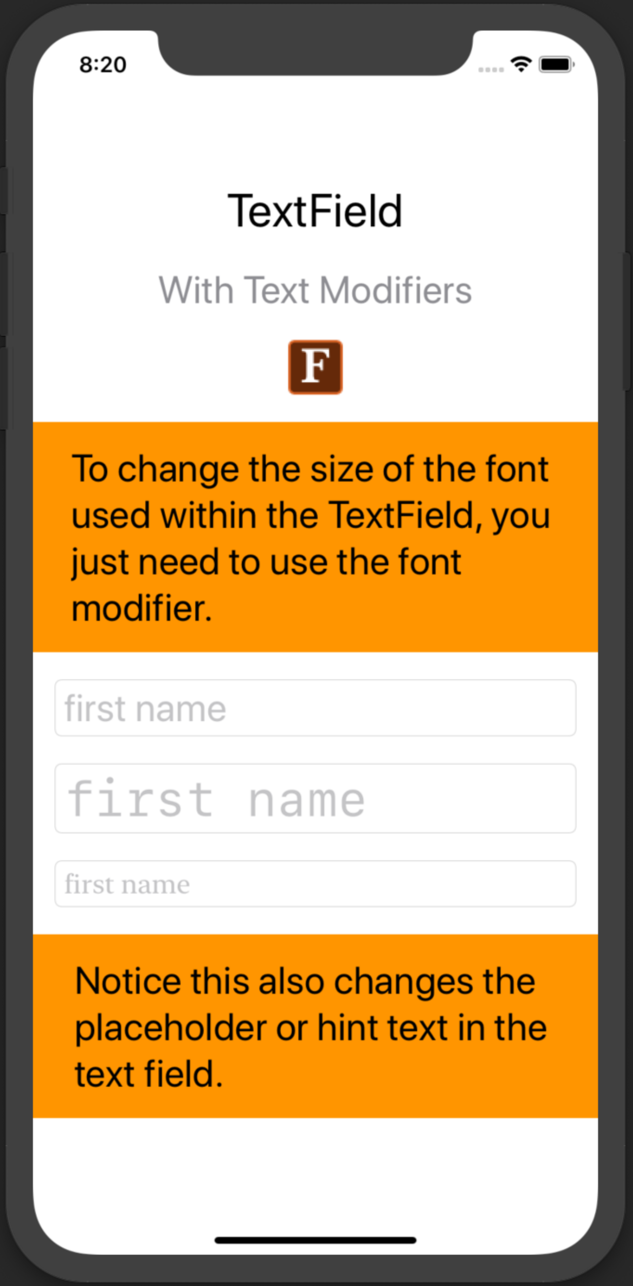
Leading

Center

Trailing

```
struct TextField_Alignment: View {
    @State private var textFieldData1 = "Leading"
    @State private var textFieldData2 = "Center"
    @State private var textFieldData3 = "Trailing"

    var body: some View {
        VStack(spacing: 20) {
            Text("TextField").font(.largeTitle)
            Text("Text Alignment").foregroundColor(.gray)
            Text("Change the alignment of text within your textfield by using the
multilineTextAlignment modifier.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange)

            Group {
                TextField("Leading", text: $textFieldData1)
                    .textFieldStyle(.roundedBorder)
                    .multilineTextAlignment(.leading) // Default

                TextField("Center", text: $textFieldData2)
                    .textFieldStyle(.roundedBorder)
                    .multilineTextAlignment(.center)

                TextField("Trailing", text: $textFieldData3)
                    .textFieldStyle(.roundedBorder)
                    .multilineTextAlignment(.trailing)
            }
            .padding(.horizontal)
        }.font(.title)
    }
}
```

Text Size and Fonts


8:20

TextField

With Text Modifiers

To change the size of the font used within the TextField, you just need to use the font modifier.

first name

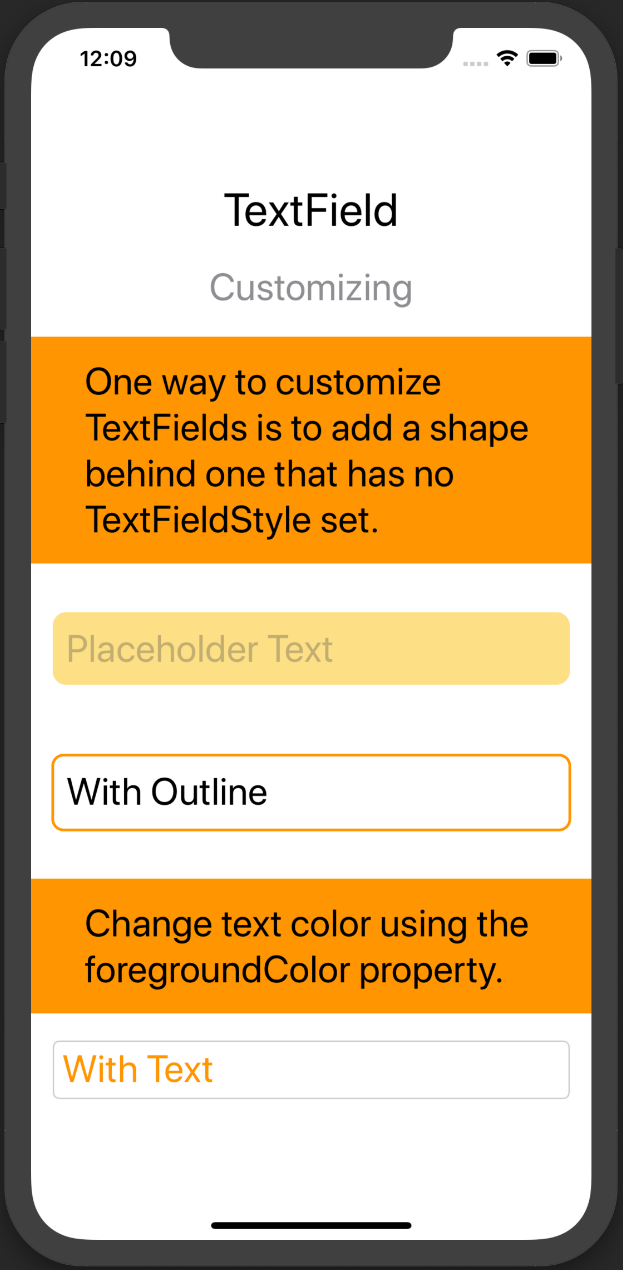
first name

first name

Notice this also changes the placeholder or hint text in the text field.

```
struct TextField_FontSize : View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 20) {
            Text("TextField").font(.largeTitle)
            Text("With Text Modifiers").foregroundColor(.gray)
            Image("Font")
            Text("To change the size of the font used within the TextField, you just need to use 
the font modifier.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.orange)
            Group {
                TextField("first name", text: $textFieldData)
                    .textFieldStyle(.roundedBorder)

                TextField("first name", text: $textFieldData)
                    .font(Font.system(size: 36, design: .monospaced))
                    .textFieldStyle(.roundedBorder)

                TextField("first name", text: $textFieldData)
                    .font(Font.system(size: 20, design: Font.Design.serif))
                    .textFieldStyle(.roundedBorder)
            }
            .padding(.horizontal)

            Text("Notice this also changes the placeholder or hint text in the text field.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.orange)
        }.font(.title)
    }
}
```

Customizing Colors


12:09

TextField

Customizing

One way to customize TextFields is to add a shape behind one that has no TextFieldStyle set.

Placeholder Text

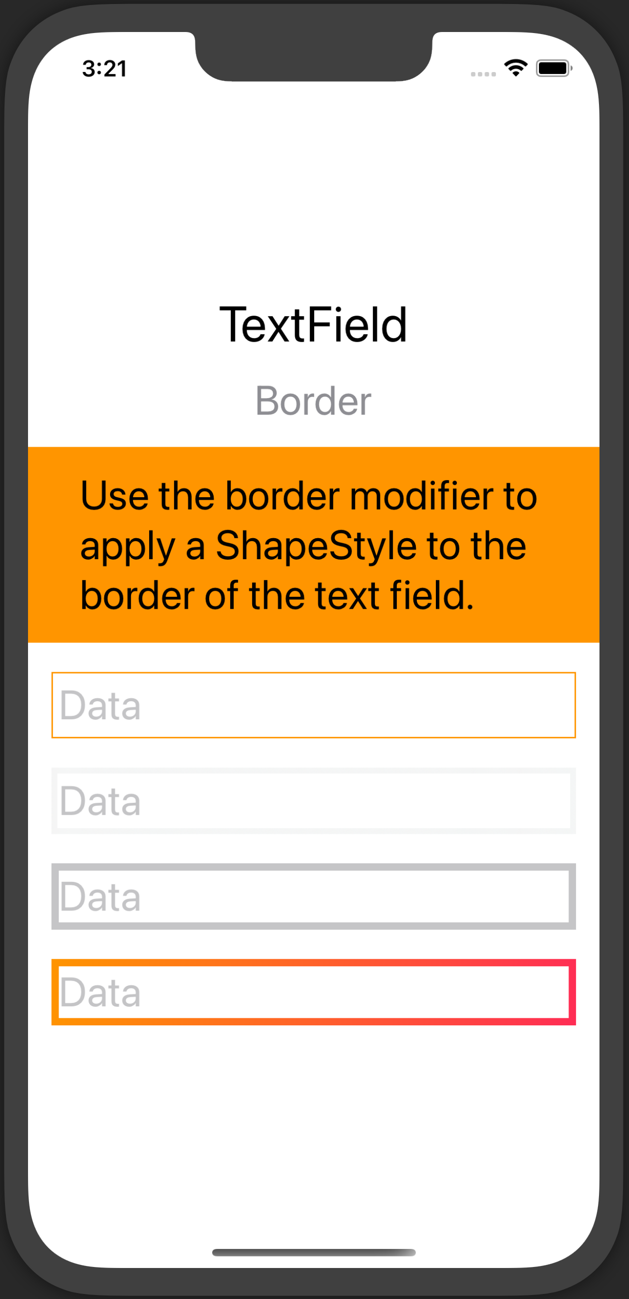
With Outline

Change text color using the foregroundColor property.

With Text

```
struct TextField_Customizing : View {
    @State private var textFieldWithText = "With Text"
    @State private var textFieldNoText = ""
    @State private var withOutline = "With Outline"

    var body: some View {
        VStack(spacing: 20) {
            Text("TextField").font(.largeTitle)
            Text("Customizing").foregroundColor(.gray)
            Text("One way to customize TextFields is to add a shape behind one that has no 
TextFieldStyle set.")
                .frame(maxWidth: .infinity).padding().background(Color.orange)

            TextField("Placeholder Text", text: $textFieldNoText)
                .padding(10)
                .background(RoundedRectangle(cornerRadius: 10)
                    .foregroundColor(Color(hue: 0.126, saturation: 0.47, brightness: 0.993)))
                .padding()
            TextField("Placeholder Text", text: $withOutline)
                .padding(10)
                .overlay(
                    // Add the outline
                    RoundedRectangle(cornerRadius: 8)
                        .stroke(Color.orange, lineWidth: 2)
            )
                .padding()

            Text("Change text color using the foregroundColor property.")
                .frame(maxWidth: .infinity).padding().background(Color.orange)

            TextField("first name", text: $textFieldWithText)
                .textFieldStyle(.roundedBorder)
                .foregroundColor(.orange)
                .padding(.horizontal)
        }.font(.title)
    }
}
```

Border



TextField

Border

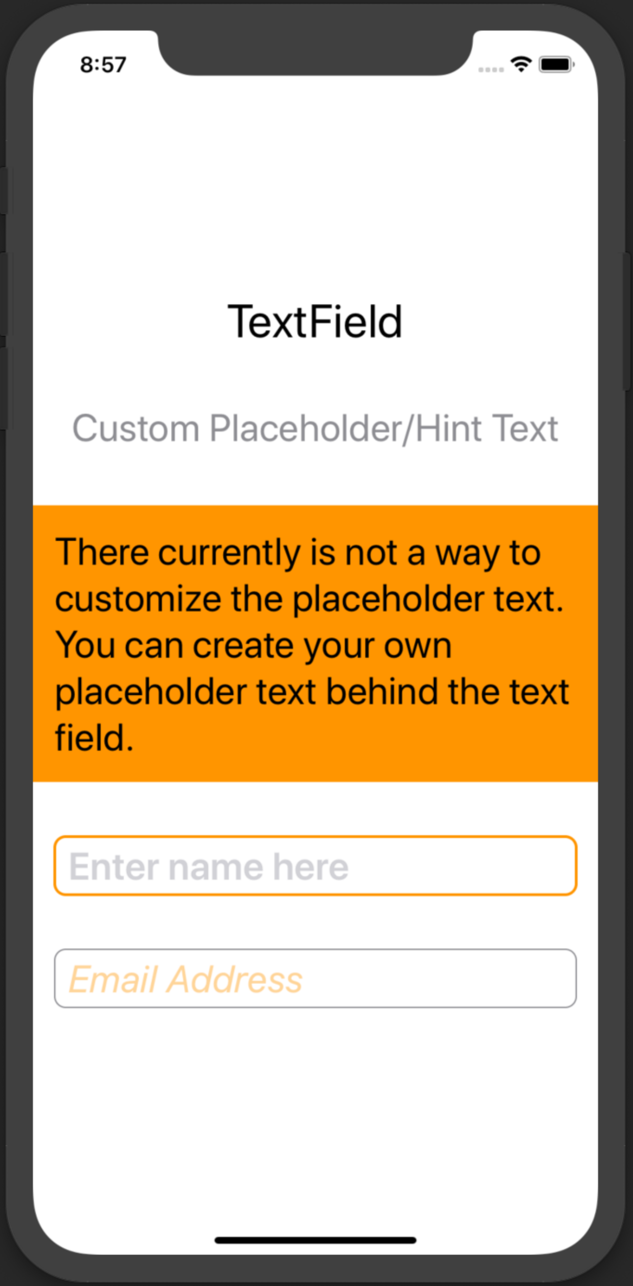
Use the border modifier to apply a ShapeStyle to the border of the text field.

```swift
struct TextField_Border: View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("TextField", subtitle: "Border",
                       desc: "Use the border modifier to apply a ShapeStyle to the border of the
                              text field.",
                       back: .orange)

            Group {
                TextField("Data", text: $textFieldData)
                    .padding(5)
                    .border(Color.orange)

                TextField("Data", text: $textFieldData)
                    .padding(5)
                    .border(.ultraThickMaterial, width: 4)

                TextField("Data", text: $textFieldData)
                    .padding(5)
                    .border(.tertiary, width: 5)

                TextField("Data", text: $textFieldData)
                    .padding(5)
                    .border(LinearGradient(colors: [.orange, .pink],
                                           startPoint: .topLeading,
                                           endPoint: .bottomTrailing), width: 5)
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

iOS 15

Material ShapeStyles are available in iOS 15.

Custom Placeholder/Hint Text



8:57

TextField

Custom Placeholder/Hint Text

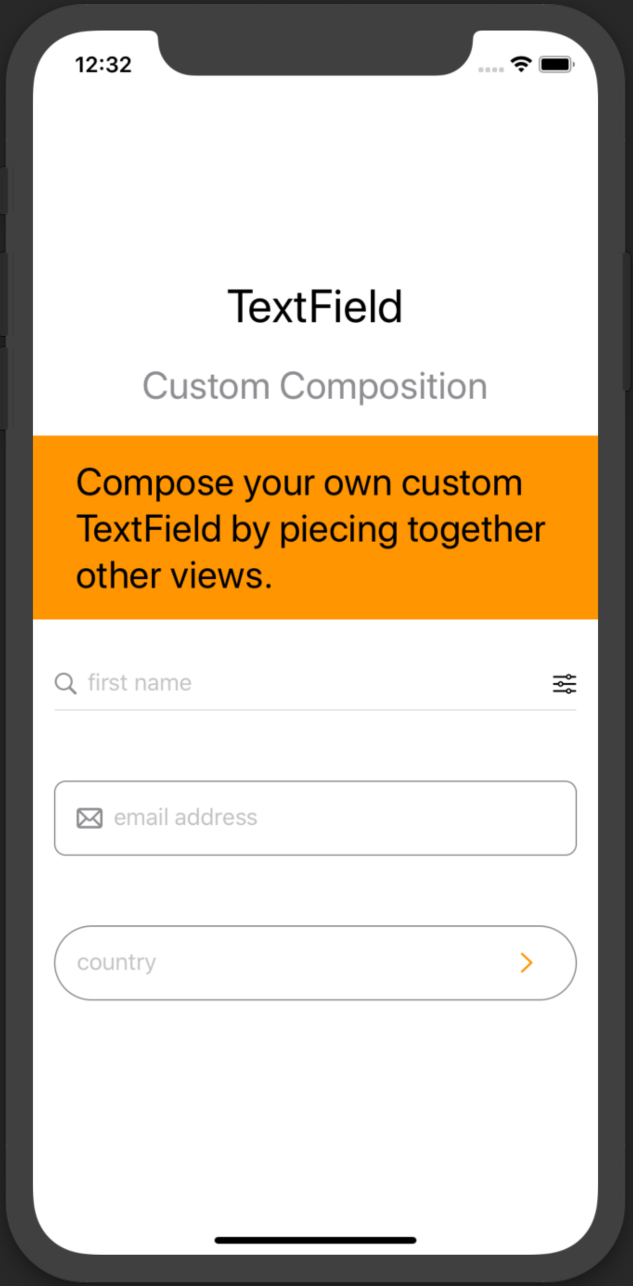
There currently is not a way to customize the placeholder text. You can create your own placeholder text behind the text field.

```swift
struct TextField_CustomPlaceholder: View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 40) {
            Text("TextField").font(.largeTitle)
            Text("Custom Placeholder/Hint Text").foregroundColor(.gray)
            Text("There currently is not a way to customize the placeholder text. You can create 
your own placeholder text behind the text field.")
                .frame(maxWidth: .infinity).padding().background(Color.orange)

            Group {
                // First TextField
                ZStack(alignment: .leading) {
                    // Only show custom hint text if there is no text entered
                    if textFieldData.isEmpty {
                        Text("Enter name here").bold()
                            .foregroundColor(Color(.systemGray4))
                    }
                    TextField("", text: $textFieldData)
                }
                .padding(EdgeInsets(top: 4, leading: 10, bottom: 4, trailing: 10))
                .overlay(
                    // Add the outline
                    RoundedRectangle(cornerRadius: 8)
                        .stroke(Color.orange, lineWidth: 2))

                // Second TextField
                ZStack(alignment: .leading) {
                    if textFieldData.isEmpty {
                        Text("Email Address").italic()
```

Custom Placeholder/Hint Text Continued

```
                                    .foregroundColor(.orange)
                                    .opacity(0.4)
                            }
                            TextField("", text: $textFieldData)
                        }
                        .padding(EdgeInsets(top: 4, leading: 10, bottom: 4, trailing: 10))
                        .overlay(
                            RoundedRectangle(cornerRadius: 8)
                                .stroke(Color.gray, lineWidth: 1))
                    }.padding(.horizontal)
                }.font(.title)
            }
        }
```

Custom Composition


12:32

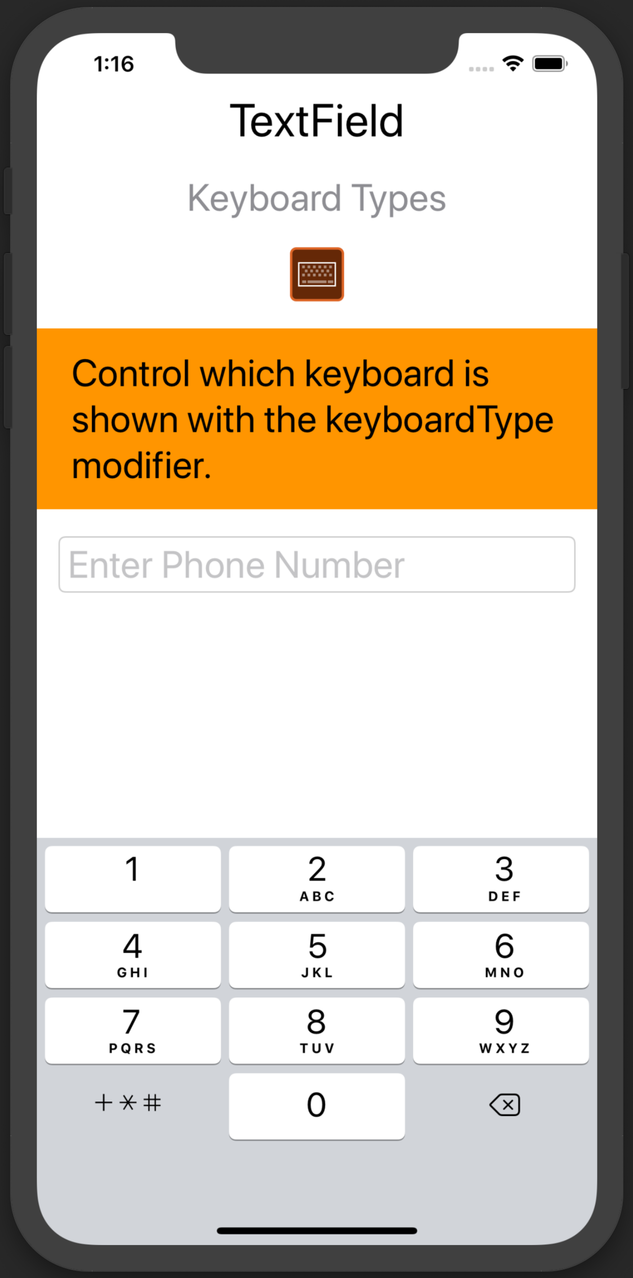
TextField

Custom Composition

Compose your own custom TextField by piecing together other views.

```
@State private var textFieldData = ""
...
VStack {
    HStack {
        Image(systemName: "magnifyingglass").foregroundColor(.gray)
        TextField("first name", text: $textFieldData)
        Image(systemName: "slider.horizontal.3")
    }
    Divider()
}
.padding()

HStack {
    Image(systemName: "envelope")
        .foregroundColor(.gray).font(.headline)
    TextField("email address", text: $textFieldData)
}
.padding()
.overlay(RoundedRectangle(cornerRadius: 8).stroke(Color.gray, lineWidth: 1))
.padding()

HStack {
    TextField("country", text: $textFieldData)
    Button(action: {}) {
        Image(systemName: "chevron.right").padding(.horizontal)
    }
    .accentColor(.orange)
}
.padding()
.overlay(Capsule().stroke(Color.gray, lineWidth: 1))
.padding()
```

Keyboard Type



TextField

Keyboard Types

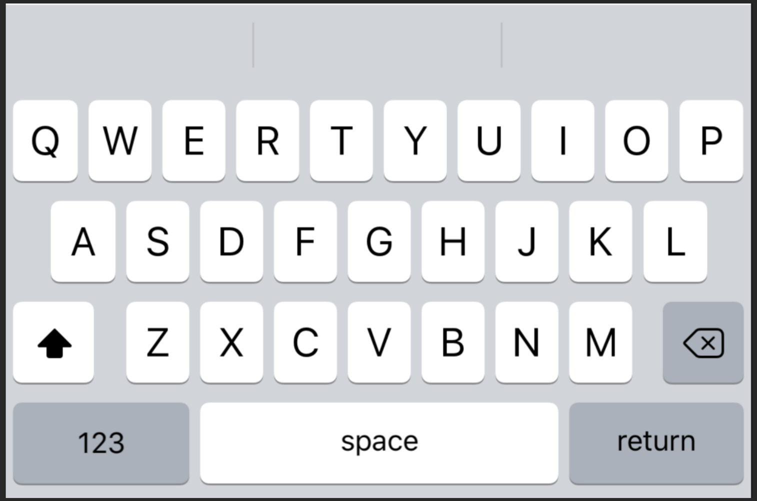
Control which keyboard is shown with the keyboardType modifier.

Enter Phone Number

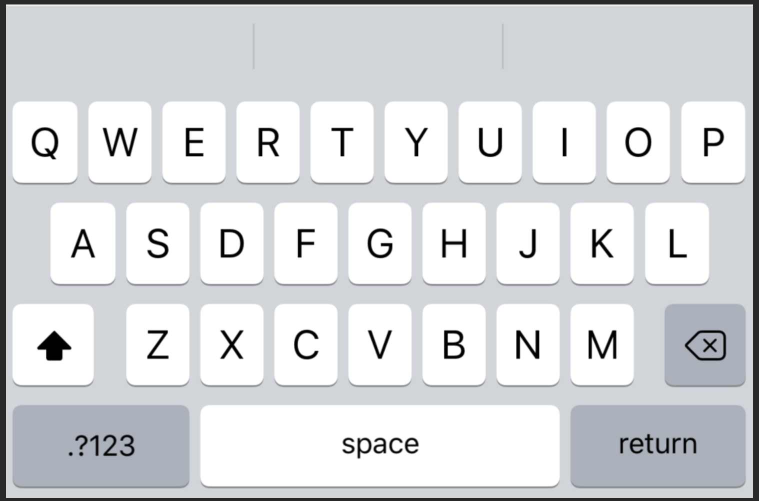
```swift
struct TextField_KeyboardType: View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 20) {
            Text("TextField")
                .font(.largeTitle)
            Text("Keyboard Types")
                .foregroundColor(.gray)

            Image(“KeyboardType")

            Text("Control which keyboard is shown with the keyboardType modifier.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.orange)

            TextField("Enter Phone Number", text: $textFieldData)
                .keyboardType(UIKeyboardType.phonePad) // Show keyboard for phone numbers
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            Spacer()
        }.font(.title)
    }
}
```

Keyboard Types Available

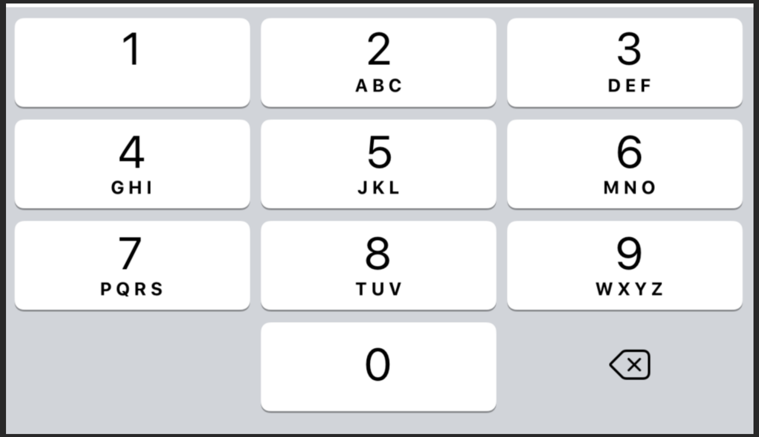
.default



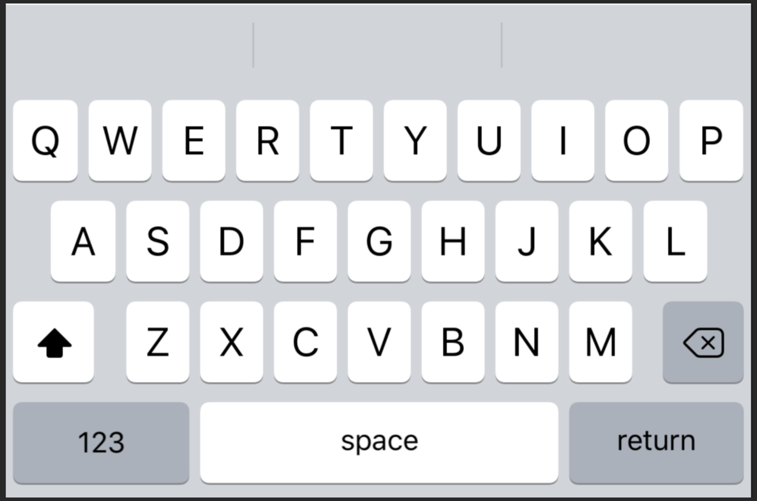
.asciiCapable



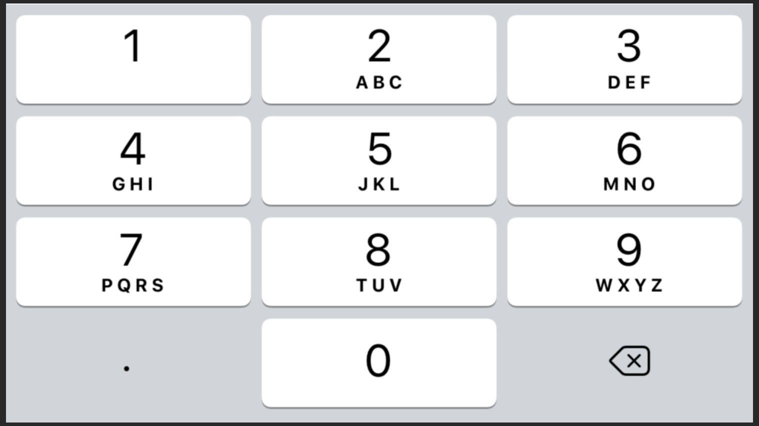
.asciiCapableNumberPad



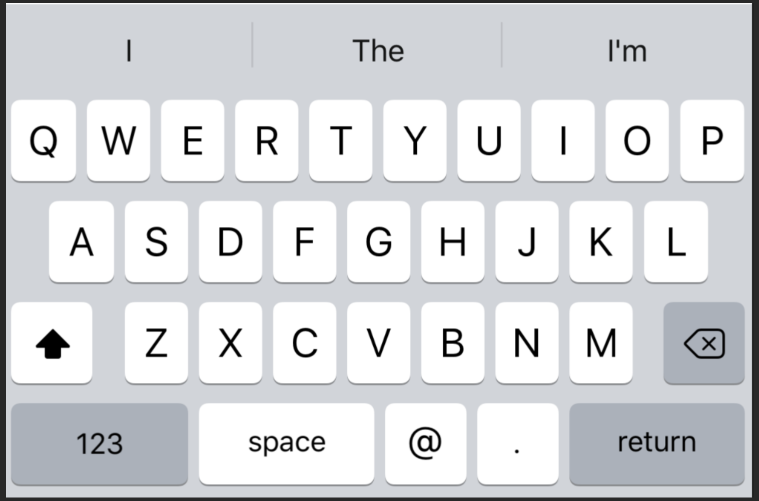
.alphabet



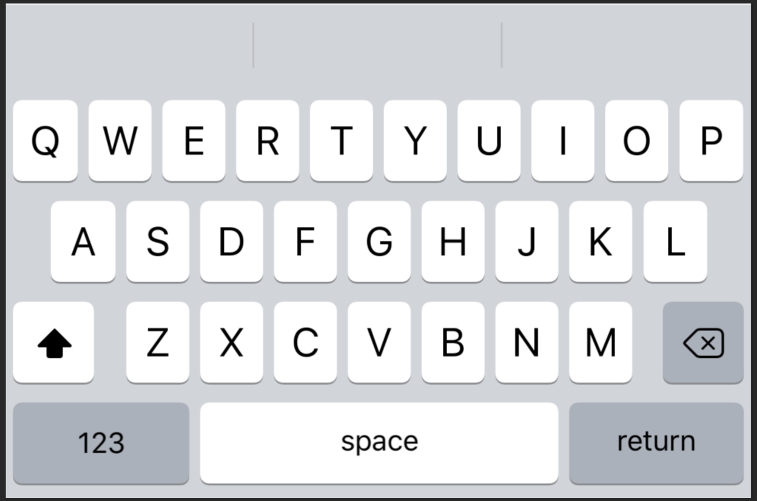
.decimalPad



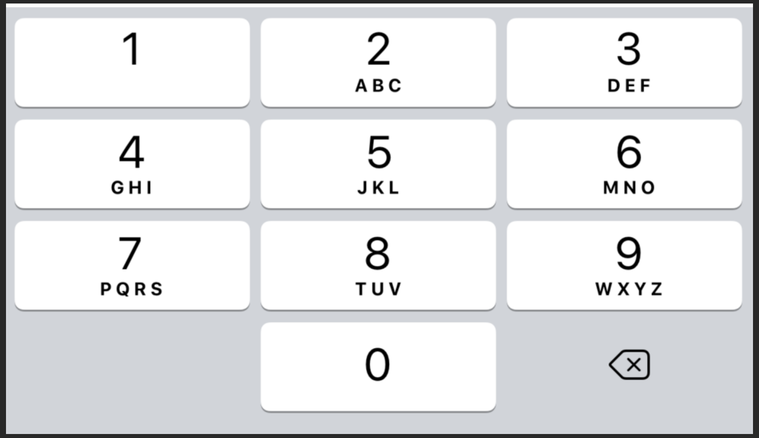
.emailAddress



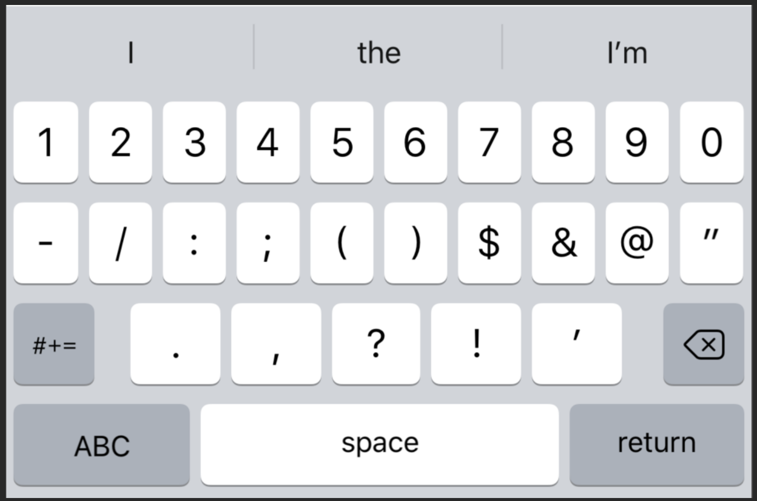
.namePhonePad



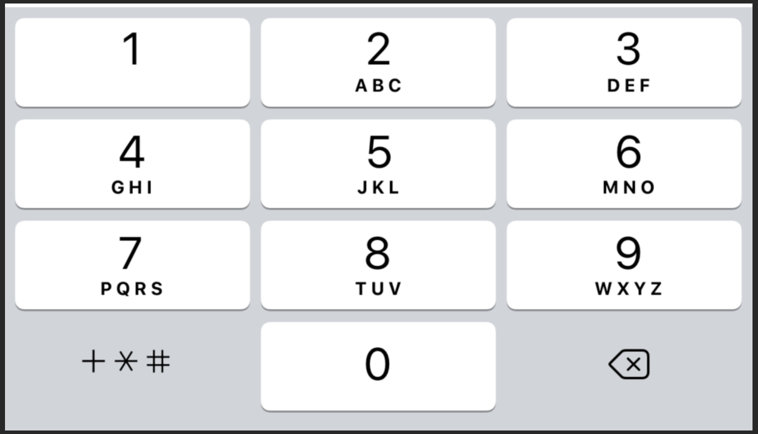
.numberPad



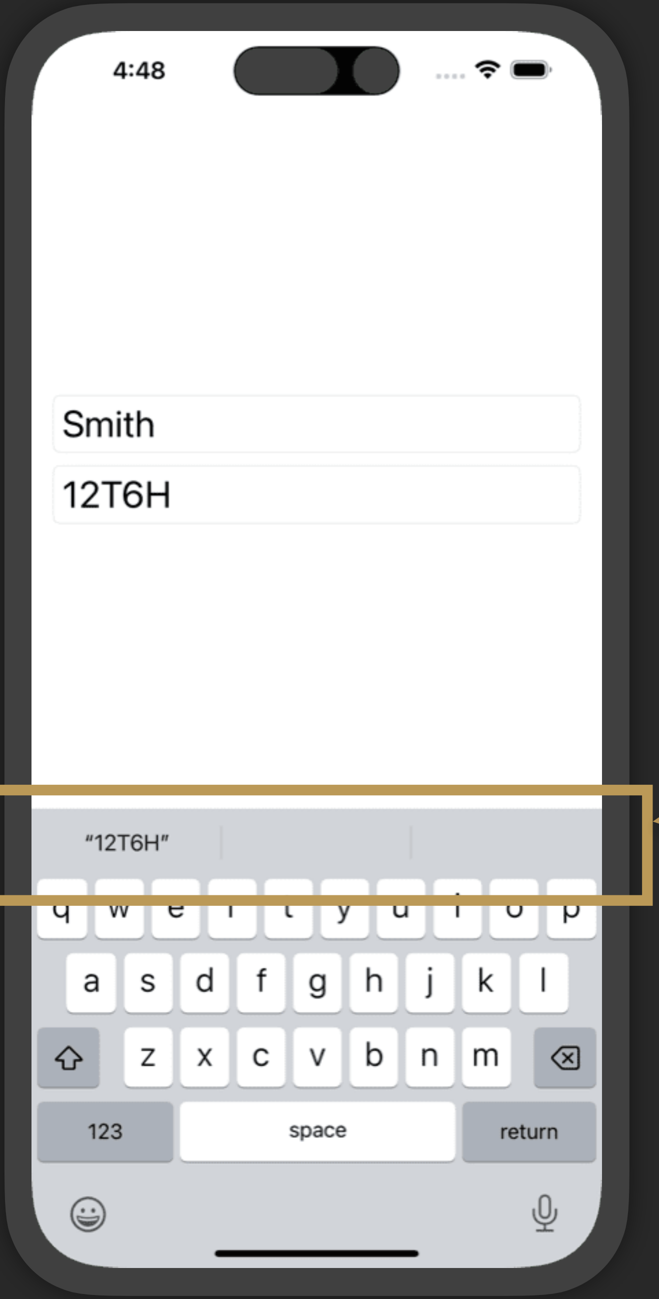
.numbersAndPunctuation



.phonePad



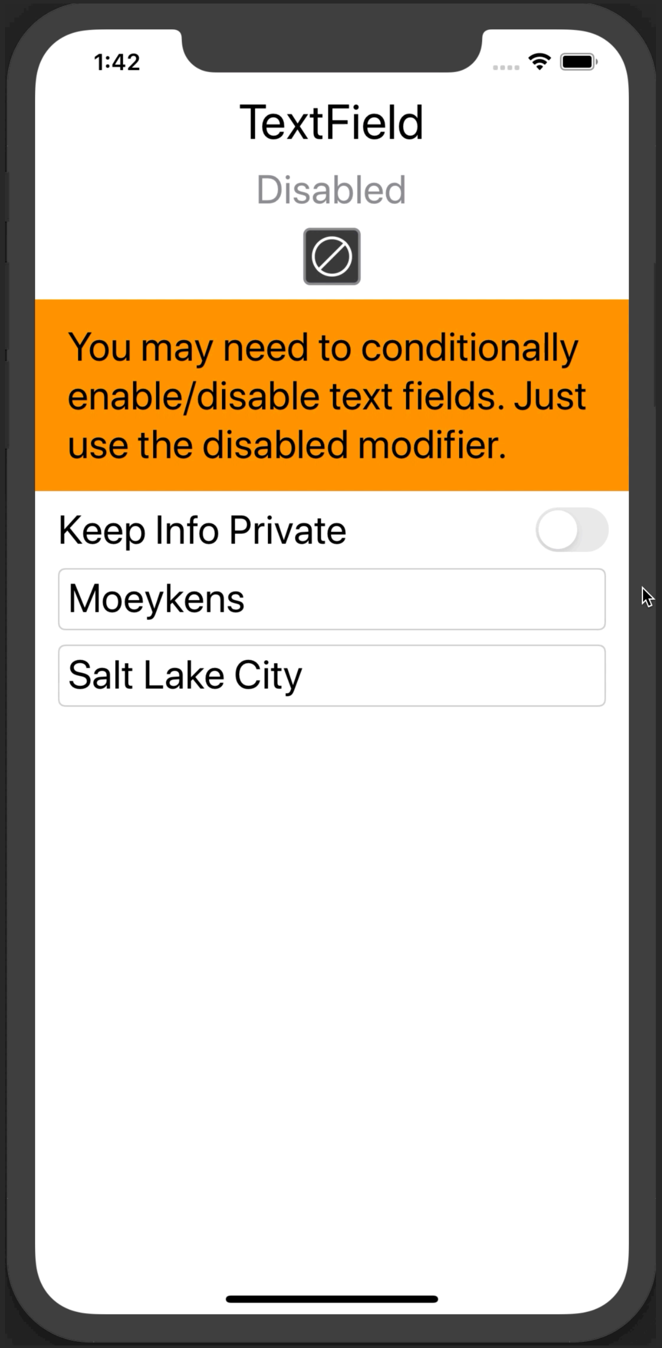
Autocorrection Disabled



```
struct TextField_Autocorrection: View {
    @State private var lastName = ""
    @State private var code = ""

    var body: some View {
        VStack(spacing: 10) {
            TextField("last name", text: $lastName)
                .autocorrectionDisabled(false) // Default: Offer suggestions (redundant)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            TextField("one time unique code", text: $code)
                .autocorrectionDisabled() // Don't offer suggestions
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)
        }
        .font(.title)
    }
}
```

You may have noticed that space above some of the keyboard types that offer autocorrection. You can turn this off with the autocorrectDisabled modifier.

Disable TextFields



TextField

Disabled

You may need to conditionally enable/disable text fields. Just use the disabled modifier.

Keep Info Private

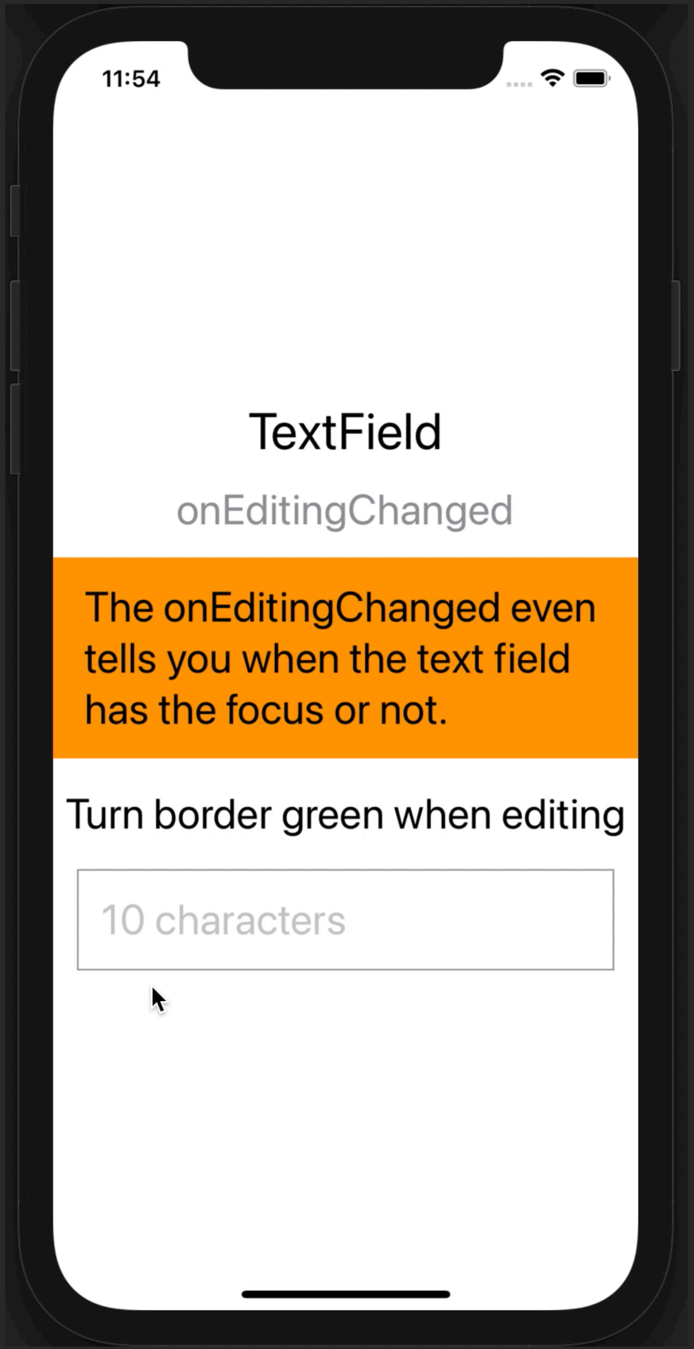
Moeykens

Salt Lake City

```swift
struct TextField_Disabled: View {
    @State private var lastName = "Moeykens"
    @State private var city = "Salt Lake City"
    @State private var disabled = false

    var body: some View {
        VStack(spacing: 10) {
            Text("TextField").font(.largeTitle)
            Text("Disabled").foregroundColor(.gray)
            Image("Disabled")
            Text("You may need to conditionally enable/disable text fields. Just use the
                disabled modifier.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.orange)

            Toggle("Keep Info Private", isOn: $disabled)
                .padding(.horizontal)

            Group {
                TextField("Enter Last Name", text: $lastName)
                TextField("Enter City", text: $city)
            }
            .disableAutocorrection(true)
            .textFieldStyle(.roundedBorder)
            .padding(.horizontal)
            .disabled(disabled) // Don't allow to edit when disabled
            .opacity(disabled ? 0.5 : 1) // Fade out when disabled

            Spacer()
        }.font(.title)
    }
}
```

Note: The disabled modifier applies to ANY VIEW. Not just the TextField view.

onEditingChanged


TextField

onEditingChanged

The onEditingChanged even tells you when the text field has the focus or not.

Turn border green when editing

```
struct TextField_OnEditingChanged: View {
    @State private var text = ""
    @State private var isEditing = false

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("TextField",
                       subtitle: "onEditingChanged",
                       desc: "The onEditingChanged even tells you when the text field has the
                             focus or not.",
                       back: .orange)

            Text("Turn border green when editing")
            TextField("10 characters", text: $text) { isEditing in
                self.isEditing = isEditing
            }
            .padding()
            .border(isEditing ? Color.green : Color.gray)
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

When the cursor is in the text field and the keyboard is showing, isEditing will be true.

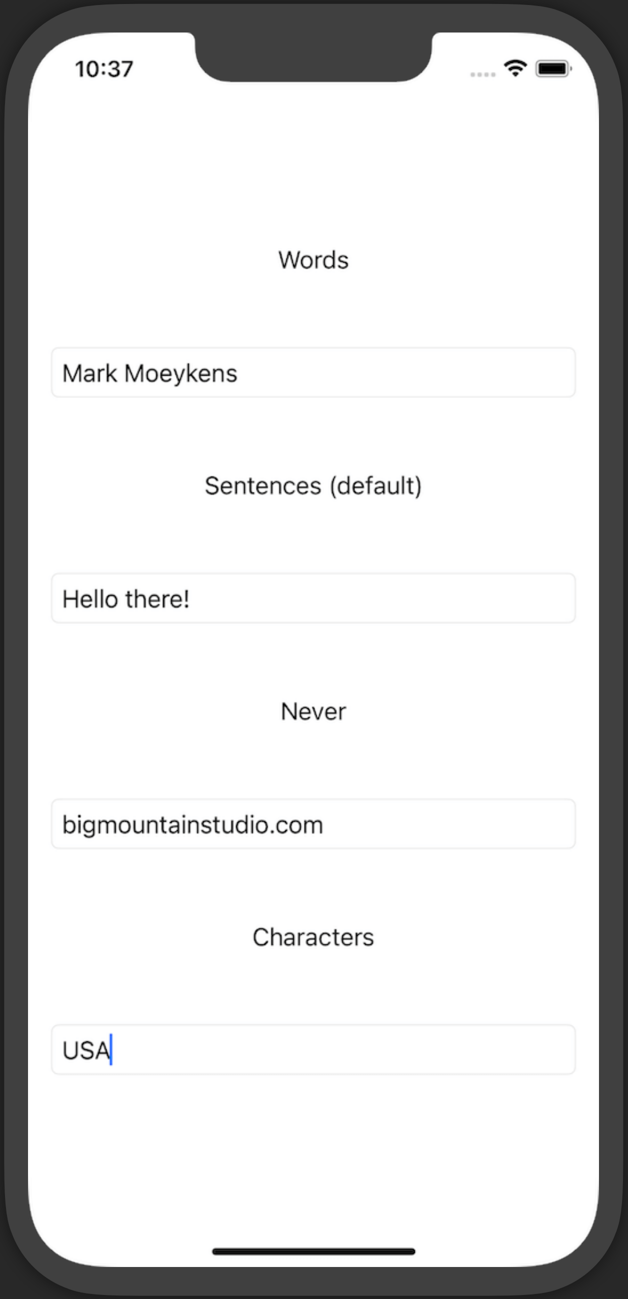
When the keyboard is dismissed and the text field no longer has the focus, isEditing will change to false.

Autocapitalization

iOS 15

```
struct TextField_Autocapitalization: View {
    @State private var textFieldData1 = ""
    @State private var textFieldData2 = ""
    @State private var textFieldData3 = ""
    @State private var textFieldData4 = ""

    var body: some View {
        VStack(spacing: 50) {
            Text("Words")
            TextField("First & Last Name", text: $textFieldData1)
                .textInputAutocapitalization(.words)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            Text("Sentences (default)")
            TextField("Bio", text: $textFieldData2)
                .textInputAutocapitalization(.sentences)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            Text("Never")
            TextField("Web Address", text: $textFieldData3)
                .textInputAutocapitalization(.never)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)

            Text("Characters")
            TextField("Country Code", text: $textFieldData4)
                .textInputAutocapitalization(.characters)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)
        }
    }
}
```

When a user starts typing into a TextField, the first word in each sentence is always capitalized. You can change this behavior with the textInputAutocapitalization modifier.



Words

Mark Moeykens

Sentences (default)

Hello there!

Never

bigmountainstudio.com

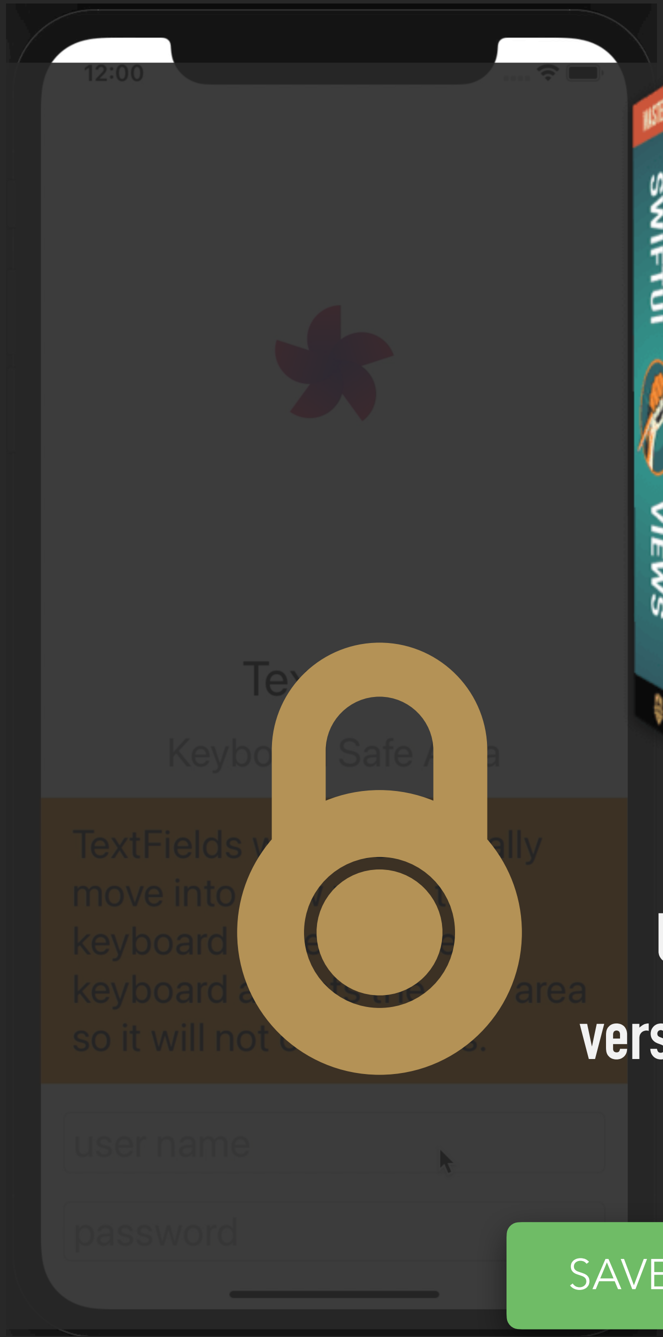
Characters

USA

Keyboard Safe Area

iOS 14

```
struct TextField_KeyboardSafeArea: View {
```

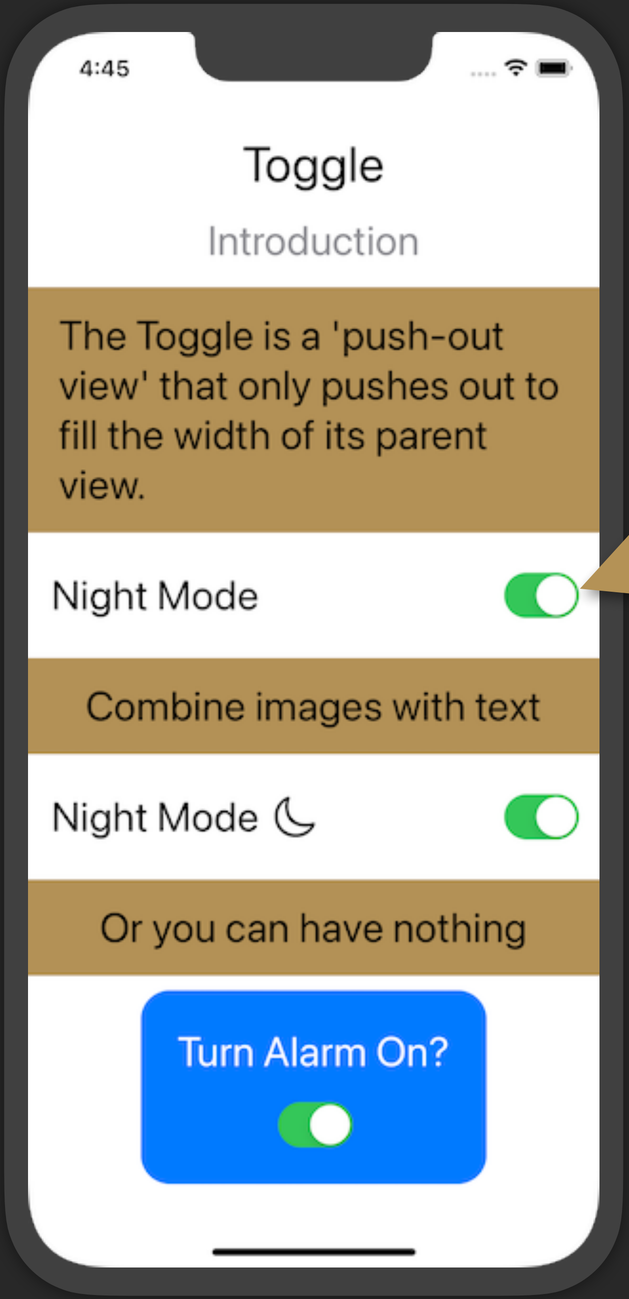


Toggle

The Toggle is a switch that can either be on or off. Much like other controls, you need to bind it to a variable. This variable is passed into the Toggle's initializer. Then, all you need to do is change this bound variable's value to change the Toggle's state on or off. Or read the bound variable's value to see what state the Toggle is currently in.

This is a push-out horizontally view.

Introduction


4:45

Toggle

Introduction

The Toggle is a 'push-out view' that only pushes out to fill the width of its parent view.

Combine images with text

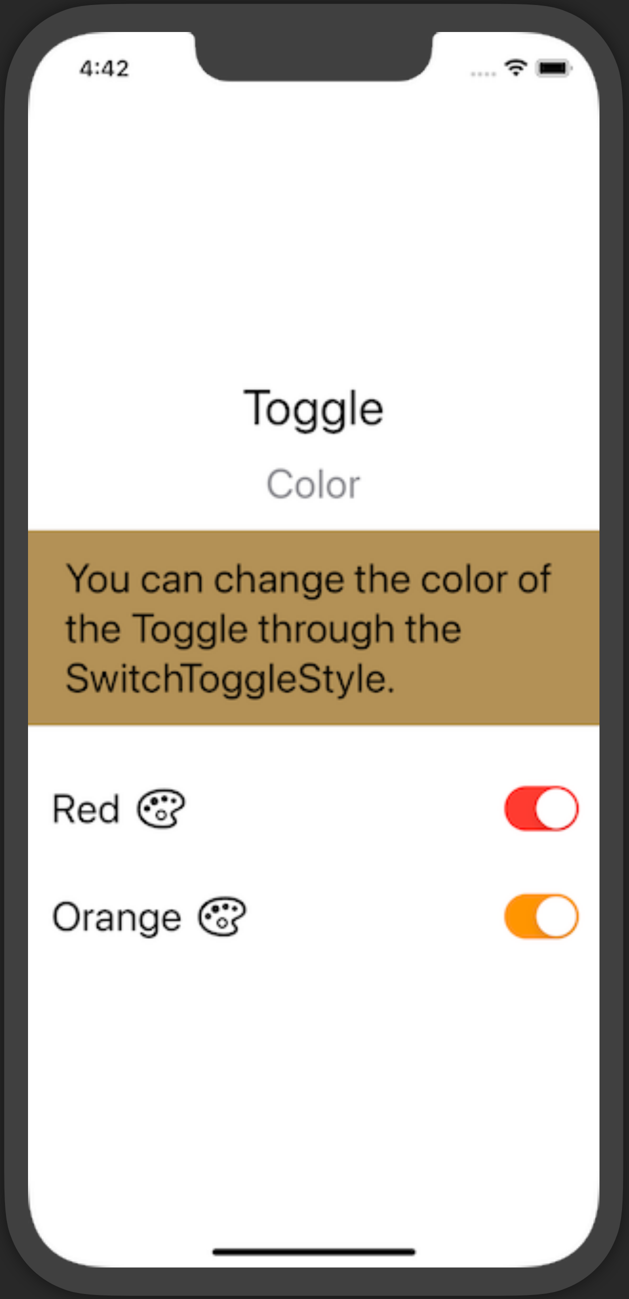
Or you can have nothing

```
struct Toggle_Intro : View {
    @State private var isToggleOn = true

    var body: some View {
        VStack(spacing: 10) {
            HeaderView("Toggle",
                       subtitle: "Introduction",
                       desc: "The Toggle is a 'push-out view' that only pushes out to fill the
                             width of its parent view.")

            Toggle("Night Mode", isOn: $isToggleOn)
                .padding()
                .accentColor(Color.red)

            DescView(desc: "Combine images with text")

            Toggle(isOn: $isToggleOn) {
                Text("Night Mode")
                Image(systemName: "moon")
            }
            .padding()

            DescView(desc: "Or you can have nothing")

            VStack {
                Text("Turn Alarm On?")
                    .foregroundStyle(.white)
                Toggle("Turn this alarm on", isOn: $isToggleOn)
                    .labelsHidden() // Hides the label/title
            }
            .padding(25)
            .background(Color.blue)
            .cornerRadius(20)
        }
        .font(.title)
    }
}
```

Note: Using accentColor directly on a Toggle does not affect its color.

Accent Color

iOS 14



Toggle

Color

You can change the color of the Toggle through the SwitchToggleStyle.

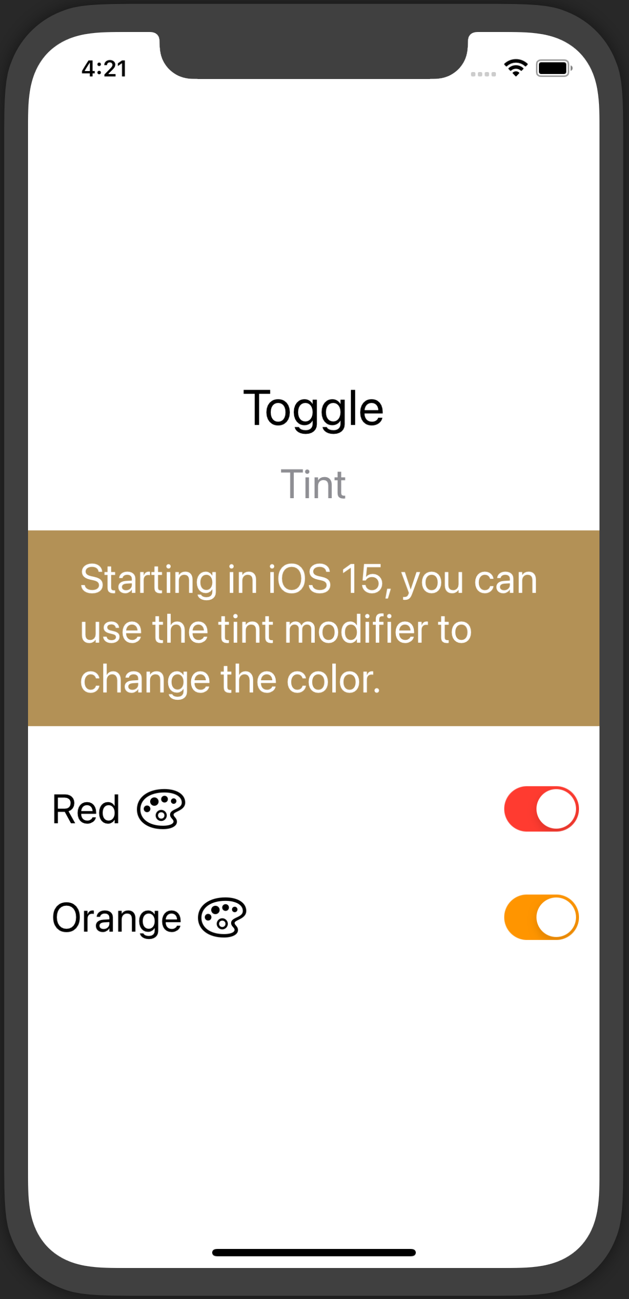
Red

Orange

```swift
struct Toggle_Color: View {
    @State private var isToggleOn = true

    var body: some View {
        VStack(spacing: 40) {
            HeaderView("Toggle",
                       subtitle: "Color",
                       desc: "You can change the color of the Toggle through the
                             SwitchToggleStyle.", back: .blue, textColor: .white)

            Group {
                Toggle(isOn: $isToggleOn) {
                    Text("Red")
                    Image(systemName: "paintpalette")
                }
                .toggleStyle(SwitchToggleStyle(tint: Color.red))

                Toggle(isOn: $isToggleOn) {
                    Text("Orange")
                    Image(systemName: "paintpalette")
                }
                .toggleStyle(SwitchToggleStyle(tint: Color.orange))
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

Tint

iOS 15


4:21

Toggle

Tint

Starting in iOS 15, you can use the tint modifier to change the color.

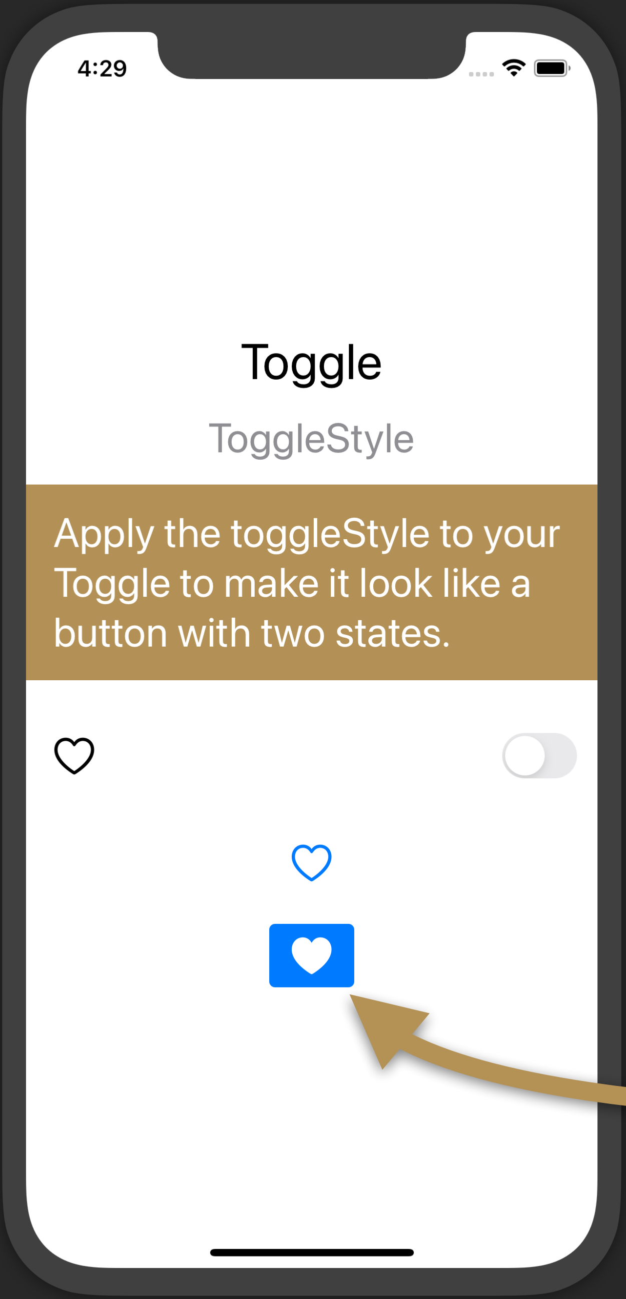
Red

Orange

```
struct Toggle_Tint: View {
    @State private var isToggleOn = true

    var body: some View {
        VStack(spacing: 40) {
            HeaderView("Toggle",
                       subtitle: "Tint",
                       desc: "Starting in iOS 15, you can use the tint modifier to change
                              the color.")

            Group {
                Toggle(isOn: $isToggleOn) {
                    Text("Red")
                    Image(systemName: "paintpalette")
                }
                .tint(.red)

                Toggle(isOn: $isToggleOn) {
                    Text("Orange")
                    Image(systemName: "paintpalette")
                }
                .tint(.orange)
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

ToggleStyle

iOS 15


4:29
Toggle
ToggleStyle
Apply the toggleStyle to your Toggle to make it look like a button with two states.

```
struct Toggle_ToggleStyle: View {
    @State private var isOn = false
    @State private var toggleOn = true

    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("Toggle",
                       subtitle: "ToggleStyle",
                       desc: "Apply the toggleStyle to your Toggle to make it look like a button
                             with two states.")

            Toggle(isOn: $isOn) {
                Image(systemName: "heart")
                    .symbolVariant(isOn ? .fill : .none)
            }.padding()

            Toggle(isOn: $isOn) {
                Image(systemName: "heart")
                    .symbolVariant(isOn ? .fill : .none)
            }
            .toggleStyle(.button)

            Toggle(isOn: $toggleOn) {
                Image(systemName: "heart")
                    .symbolVariant(toggleOn ? .fill : .none)
            }
            .toggleStyle(.button)
        }
        .font(.title)
    }
}
```

These examples are using the symbol variant to switch between filled and not filled SF symbols.

Notice when the toggleStyle is button and it is in the on state, the whole button becomes filled.

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!

BMS_10

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 through sales in the future. Go here for instructions on how to do this.)

PARTNERING INFO

IT'S FREE TO PARTNER UP AND MAKE MONEY. JUST FOLLOW THESE 2-STEPS:

1. Go to bigmountainstudio.com/makemoney and sign up.

2. Start sharing content (social posts, video mentions, blogs, etc.) with your partner link and earn **50%** of all sales.

THE END

I hope you enjoyed this free SwiftUI Views Quick Start!

This was just the beginning of a larger book.