THE ULTIMATE SWIFTUI LAYOUT COOKBOOK

LIMITED FREE EDITION

KARIN PRATER

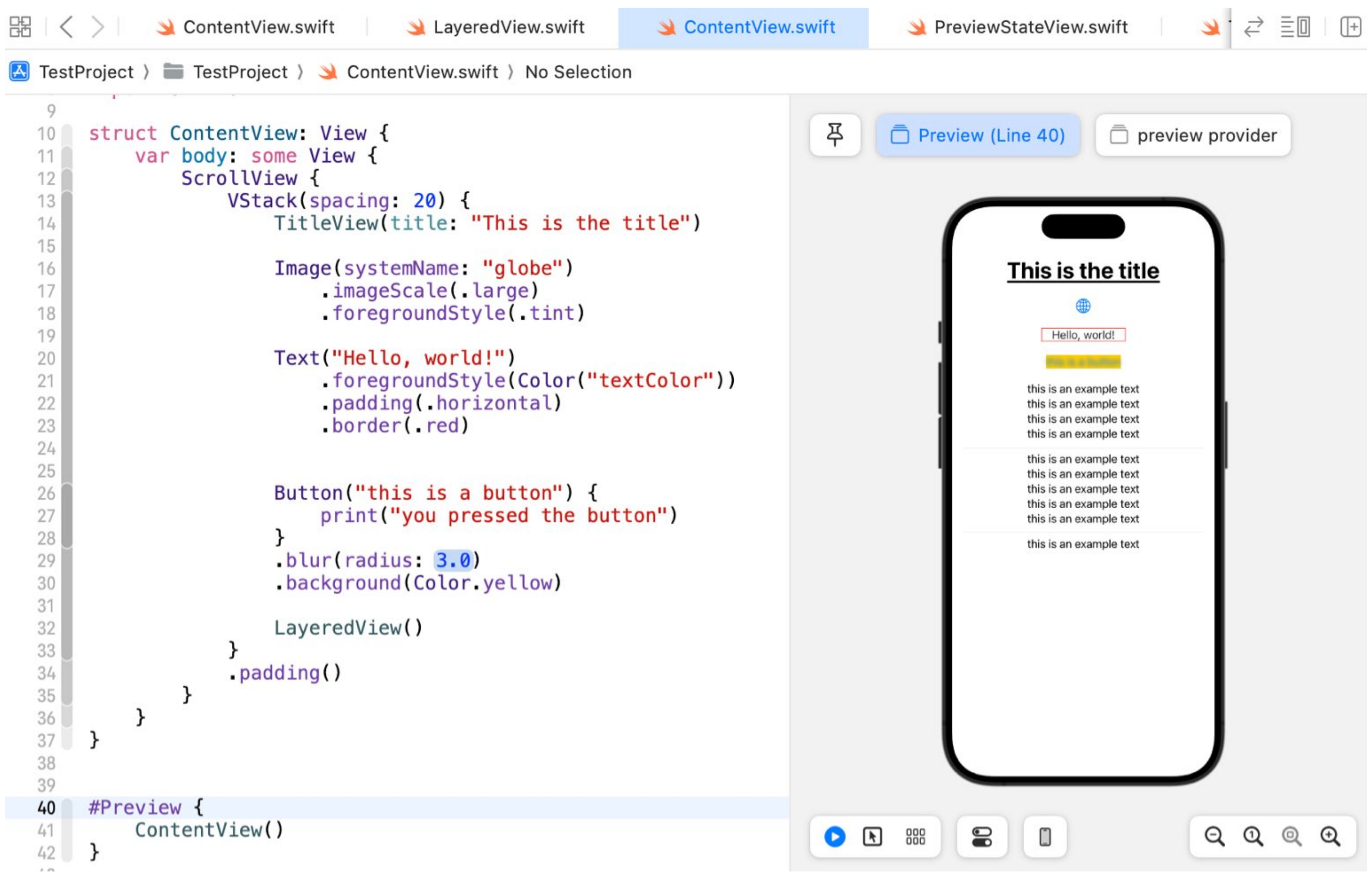
Thank you for reading this free book!

You can get the full book at swiftyplace.com which includes all project files you see in this book.

1. WORKING WITH SWIFTUI IN XCODE

1.1 SHOWING PREVIEWS IN XCODE

In this section, I will guide you on effectively working with Xcode for SwiftUI. Previews have changed with Xcode 15 and use now the Preview macro, whereas before you had to use the PreviewProvider. I will give you examples of both of these features.



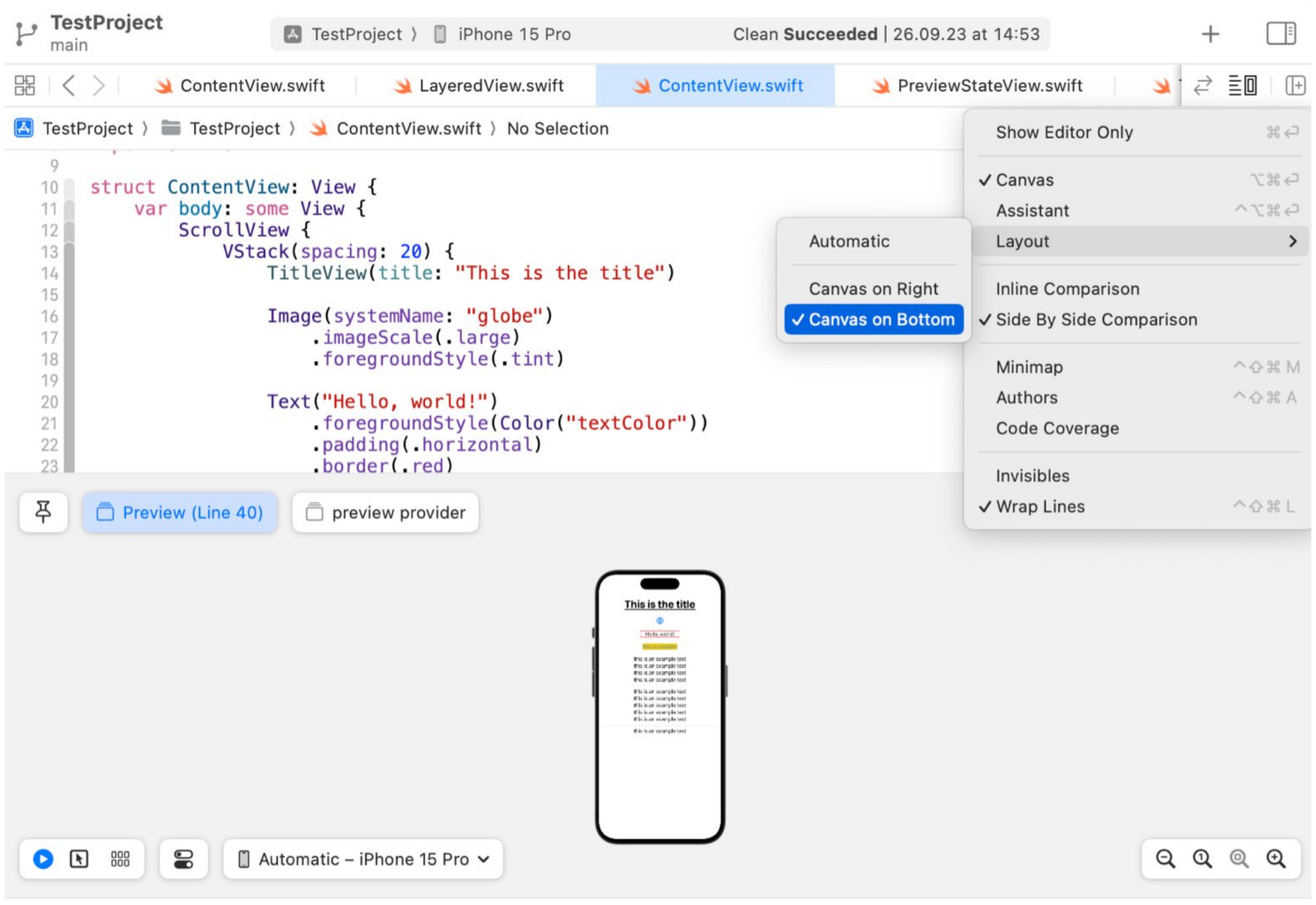
```swift
struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {
                TitleView(title: "This is the title")

                Image(systemName: "globe")
                    .imageScale(.large)
                    .foregroundStyle(.tint)

                Text("Hello, world!")
                    .foregroundStyle(Color("textColor"))
                    .padding(.horizontal)
                    .border(.red)


                Button("this is a button") {
                    print("you pressed the button")
                }
                .blur(radius: 3.0)
                .background(Color.yellow)

                LayeredView()
            }
            .padding()
        }
    }
}


#Preview {
    ContentView()
}
```

Showing and Hiding the Canvas

By default, the canvas is not shown on the right side. To show or hide the canvas, go to the top-right corner and click on the inspector. You can also use the keyboard shortcut **Option + Command + Return**.

Sometimes, when you make changes to your code, the preview may not refresh properly or may not be visible. In such cases, you can use the keyboard shortcut **Option + Command + P** to recreate the preview.

By default, the layout is set to automatic. You can choose between having the canvas on the right or below the editor by selecting the appropriate option. This allows you to maximize the space based on your preferences and the size of the views.



```
struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {
                TitleView(title: "This is the title")

                Image(systemName: "globe")
                    .imageScale(.large)
                    .foregroundStyle(.tint)

                Text("Hello, world!")
                    .foregroundStyle(Color("textColor"))
                    .padding(.horizontal)
                    .border(.red)
```

Preview Macro

The new preview macro has made it simpler and shorter in Xcode previews and is available for Xcode 15:

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

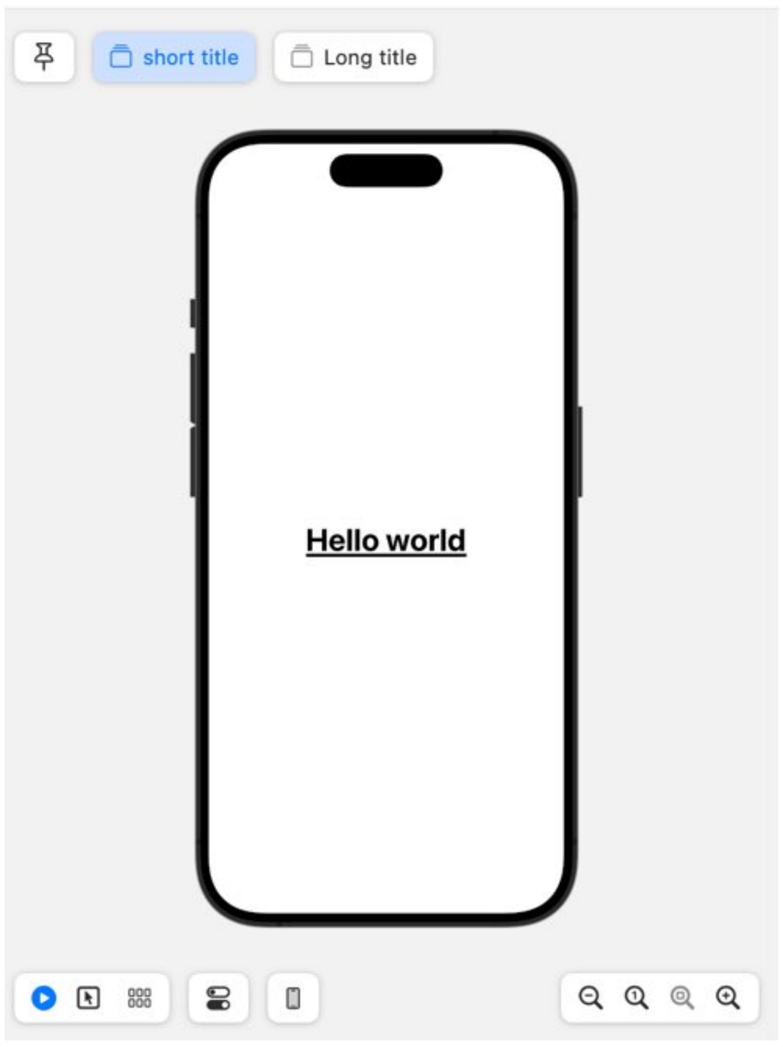
When you generate a new file, such as using the SwiftUI view template, Xcode automatically generates a preview section for that file.

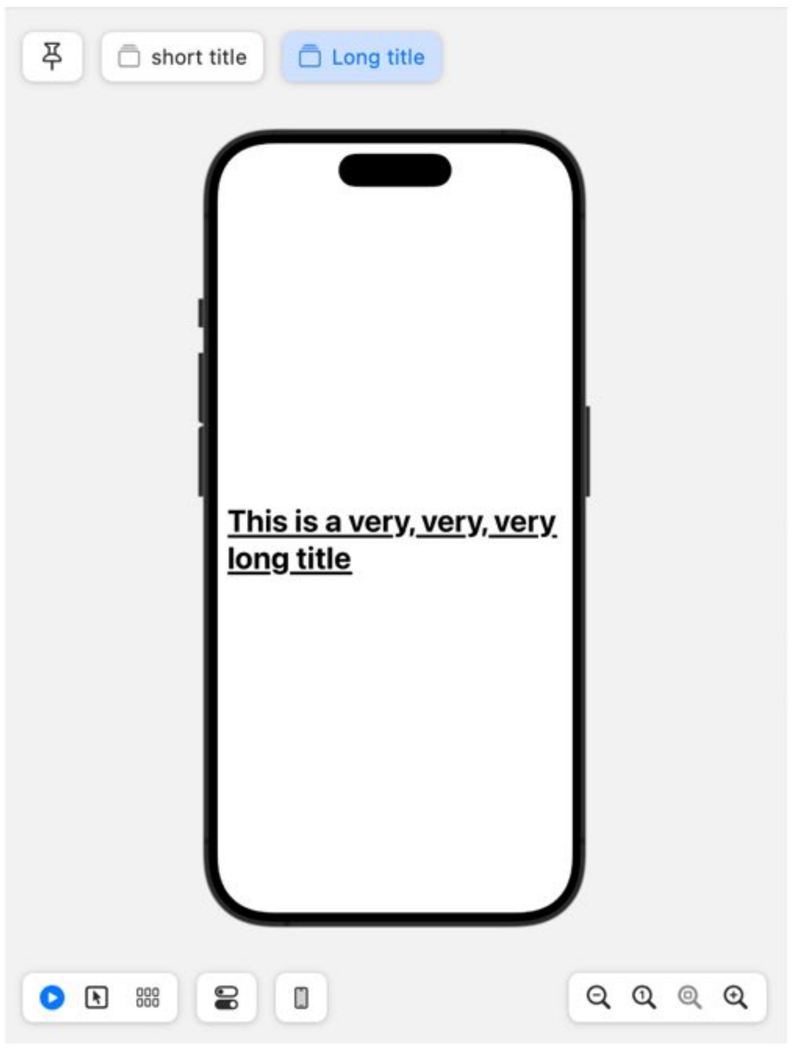
If your project's minimum deployment target is lower than iOS 17 or macOS 14, you need to add a version check before the preview:

```
@available(iOS 17.0, macOS 14.0, tvOS 17.0, watchOS 10.0, *)
#Preview {
    ContentView()
}
```

When working with multiple previews, it is important to ensure that you pass the correct arguments to the views. You can generate multiple previews by using the “preview” macro multiple times. Each preview can only specify one view. If you need to test different input values, you can create multiple previews with varying arguments.

```swift
import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, very long title")
}
```

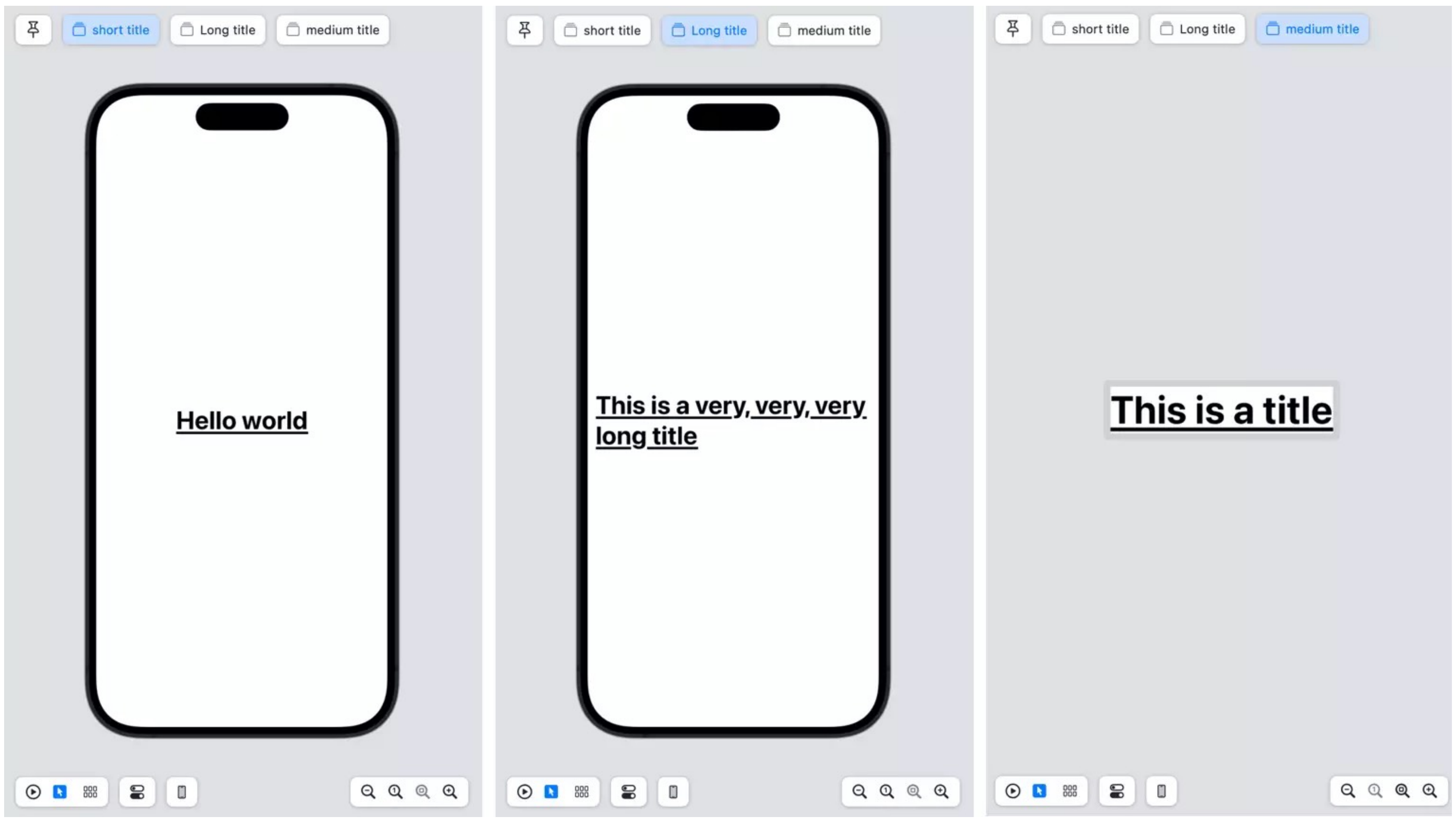




To organize your previews, you can give them names. This helps in distinguishing between different previews and provides a clear description of their purpose.

Additionally, you can set **traits**. In the below example, I used a “sizeThatFitsLayout”. This is useful when you have very small views and you don’t want to show them on a device. Note that this only works when you are in the selectable preview.

```
    #Preview("short title") {
        TitleView(title: "Hello world")
    }
    #Preview("Long title") {
        TitleView(title: "This is a very, very, very long title")
    }

    #Preview("medium title", traits: .sizeThatFitsLayout) {
        TitleView(title: "This is a title")
    }
```



Here is a list of all the available traits:

- fixedLayout(width: CGFloat, height: CGFloat) and sizeTahtFitsLayout
- portrait and portraitUpsideDown
- landscapeLeft and landscapeRight

PreviewProvider

You may come across the preview provider if you are working with an older project that started before Xcode 15.

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

You can show multiple previews in the canvas:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .previewLayout(.sizeThatFits)
            ContentView()
        }
    }
}
```

The preview can be customized by adding preview modifiers:

```
struct ContentView_Preview: PreviewProvider {
    static var previews: some View {
        ContentView()
            .previewLayout(.sizeThatFits)
            .previewDisplayName("preview provider")
    }
}
```

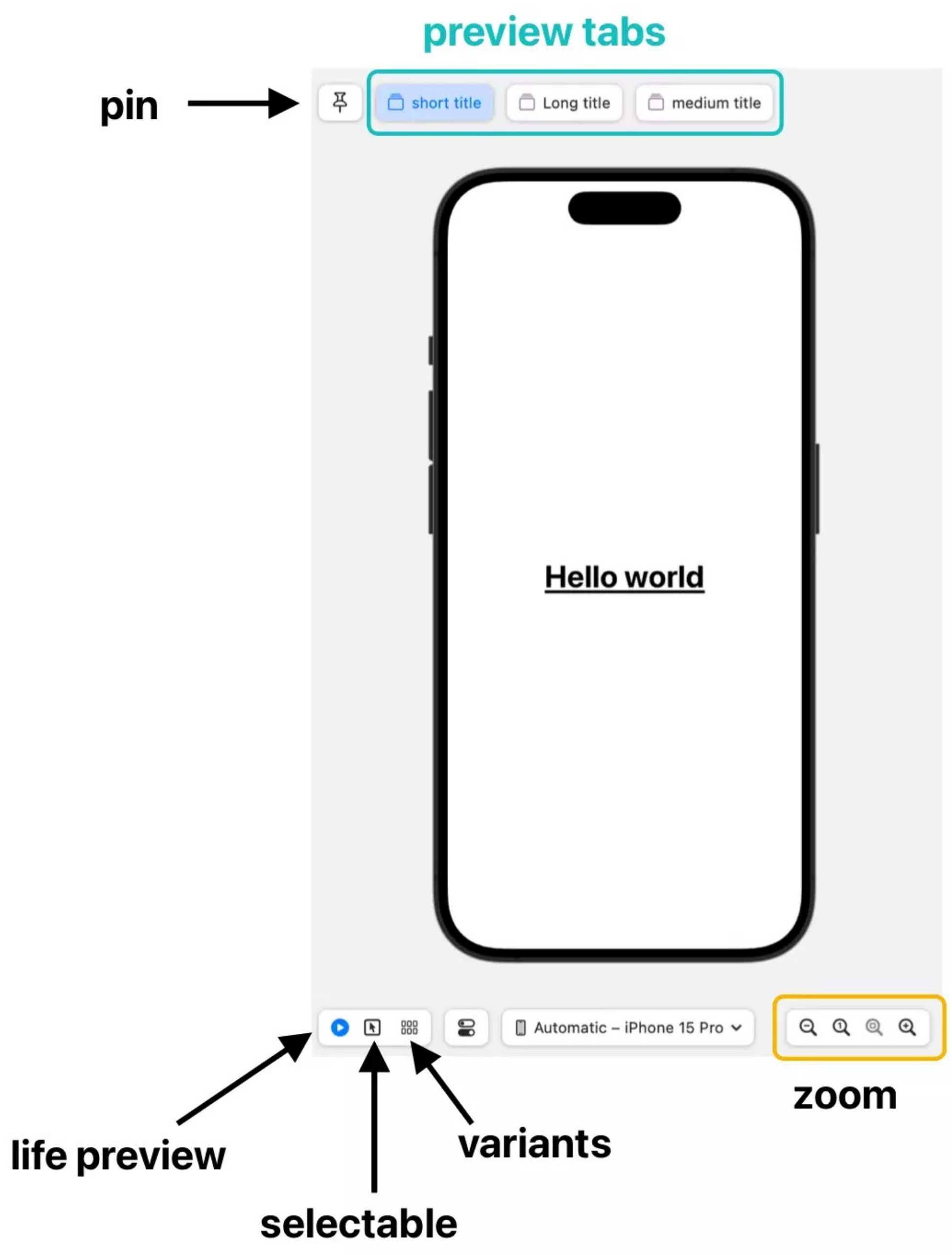
Modifier for minimum size	`.previewLayout(.sizeThatFits)`
Modifier for fixed size	`.previewLayout(.fixed(width: 600, height: 200))`
Set specific device type	`.previewDevice(PreviewDevice(rawValue: "iPhone 8"))`
Set display name tab	`.previewDisplayName("show Iphone 8")`

Change Environment variables:

Change to dark mode	`.environment(\.colorScheme, .dark)`
Change dynamic text	`.environment(\.sizeCategory, .accessibilityLarge)`

1.2 WORKING WITH THE CANVAS IN XCODE

In this section, we will explore the various features and options available in the Canvas in Xcode. The Canvas provides a real-time preview of your SwiftUI layout, allowing you to quickly iterate and test your designs.



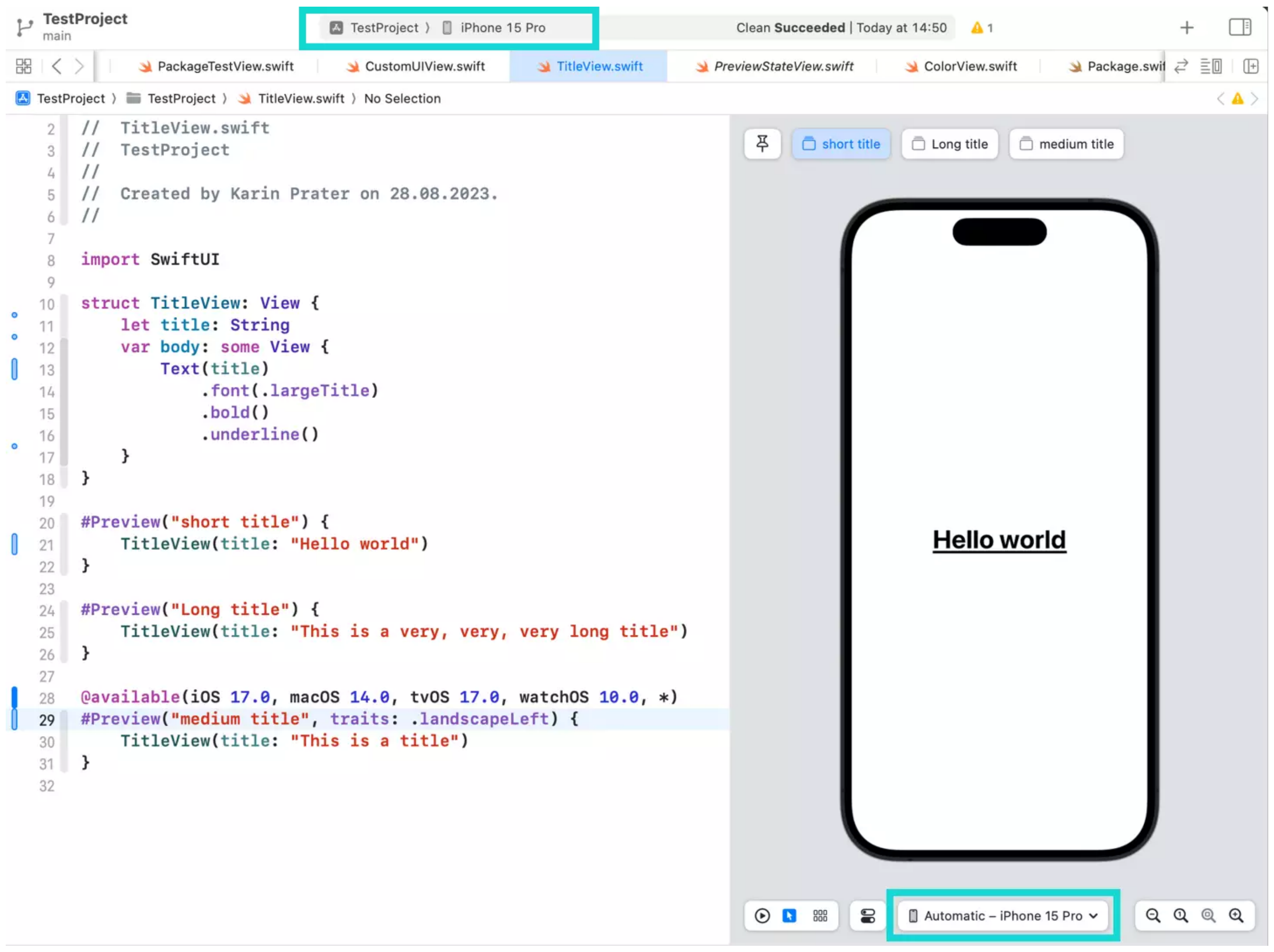
At the bottom of the Canvas, you will find a range of options to enhance your previewing experience. Let’s take a closer look at each of these features:

Zooming and Fit on Screen

On the right side of the Canvas, you can find options to zoom in or fit the layout on the screen. These options help you view your design more closely or fit it to the available space.

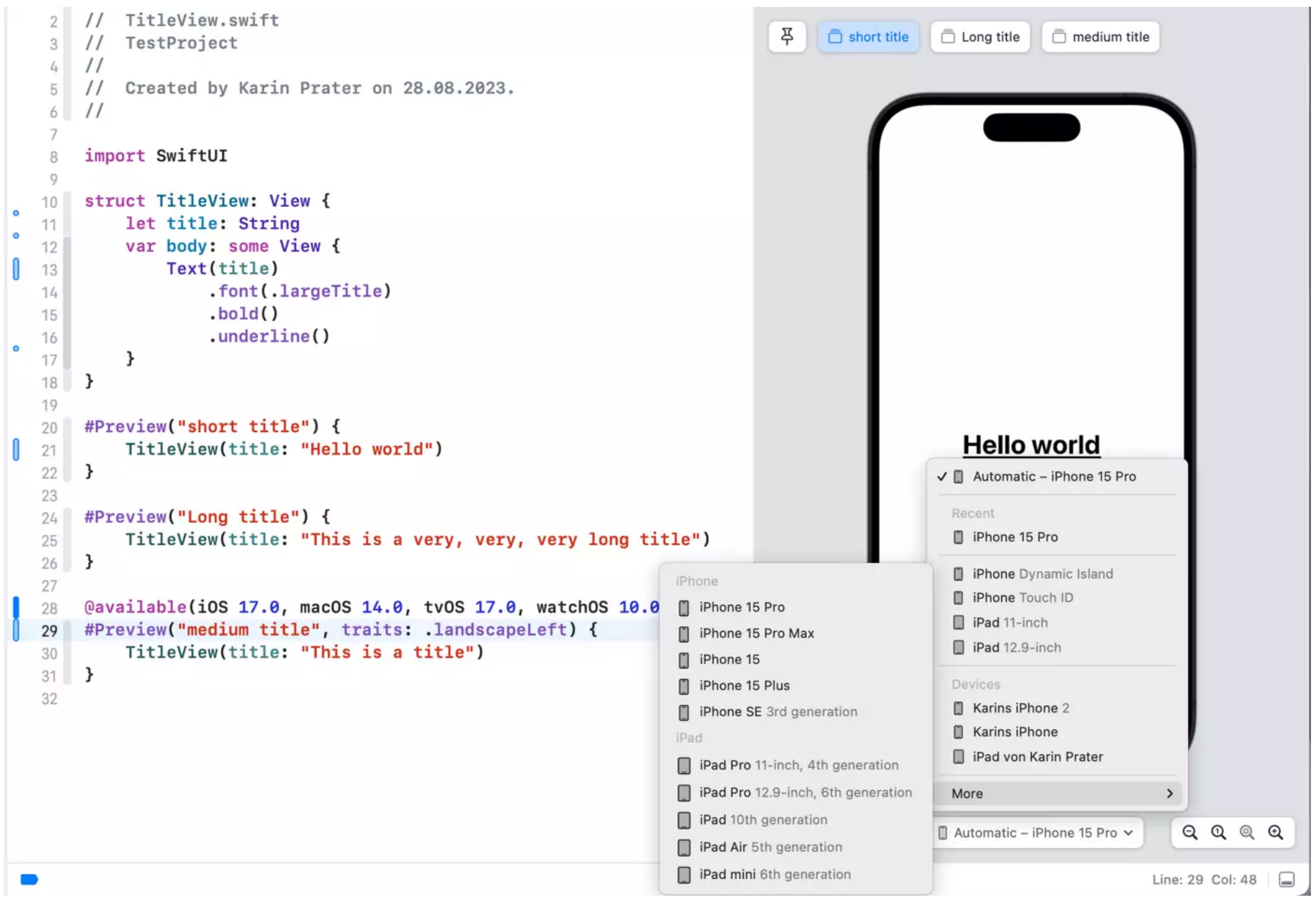
Device Preview

The toggle next to the zoom options allows you to select the device for the preview. By default, the preview matches the device selected for your run target.



```swift
//  TitleView.swift
//  TestProject
//
//  Created by Karin Prater on 28.08.2023.
//

import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, very long title")
}

@available(iOS 17.0, macOS 14.0, tvOS 17.0, watchOS 10.0, *)
#Preview("medium title", traits: .landscapeLeft) {
    TitleView(title: "This is a title")
}
```

However, you can choose to manually select a specific device or let Xcode automatically switch the preview based on your run target.



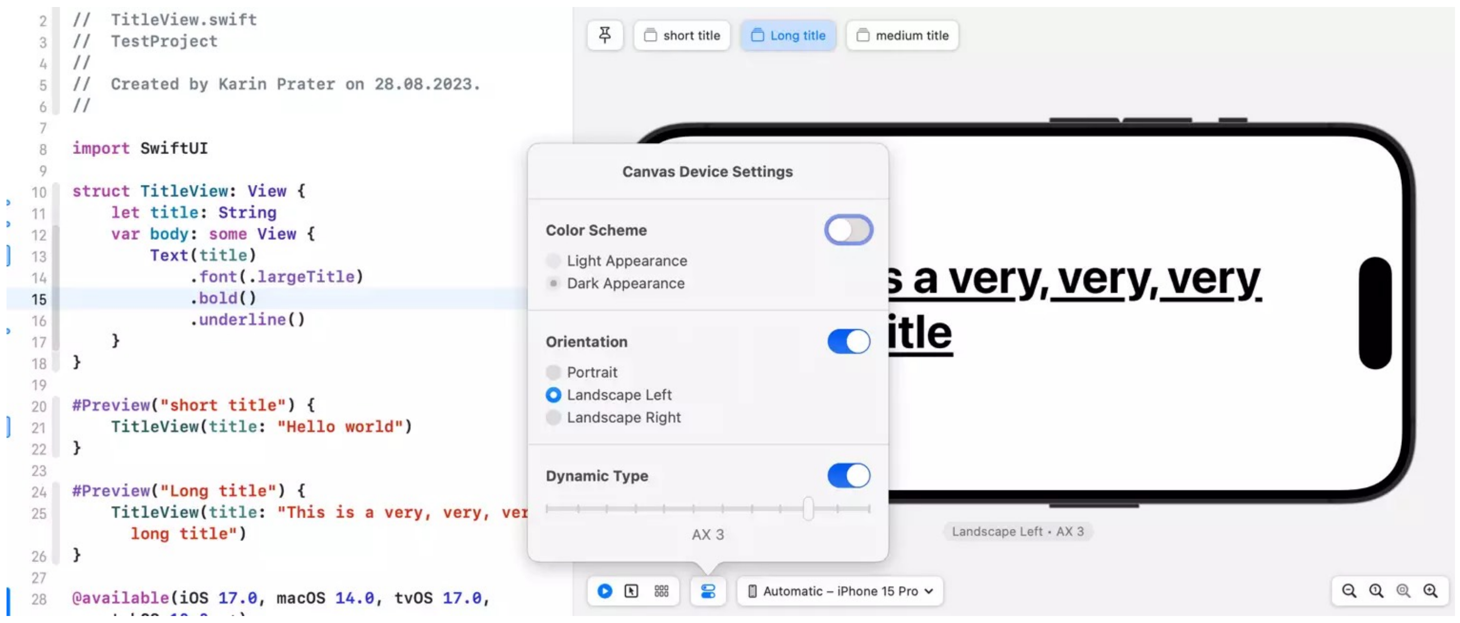
```swift
//  TitleView.swift
//  TestProject
//
//  Created by Karin Prater on 28.08.2023.
//

import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, very long title")
}

@available(iOS 17.0, macOS 14.0, tvOS 17.0, watchOS 10.0
#Preview("medium title", traits: .landscapeLeft) {
    TitleView(title: "This is a title")
}
```

Real Device Preview

In addition to the built-in device previews, you can also use your actual iOS device to preview your SwiftUI layout. To enable this feature, you need to install the Xcode Previews app on your device and allow it in developer mode. Please note that there may be some connectivity issues, so ensure your device is plugged in for a reliable connection.

Device Settings

The “Device Settings” option allows you to customize the preview environment further. You can set specific color schemes, test landscape or portrait orientations, and even experiment with dynamic type sizes. These settings help you ensure your layout adapts well to different scenarios.



```swift
//  TitleView.swift
//  TestProject
//
//  Created by Karin Prater on 28.08.2023.
//

import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, ver
      long title")
}

@available(iOS 17.0, macOS 14.0, tvOS 17.0,
```

Canvas Device Settings

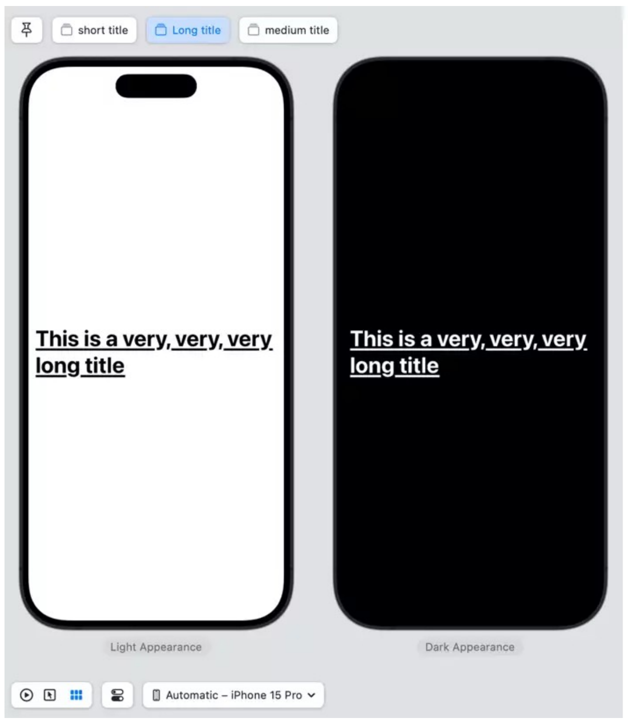
Color Scheme
Light Appearance
Dark Appearance

Orientation
Portrait
[✓ Landscape Left]
Landscape Right

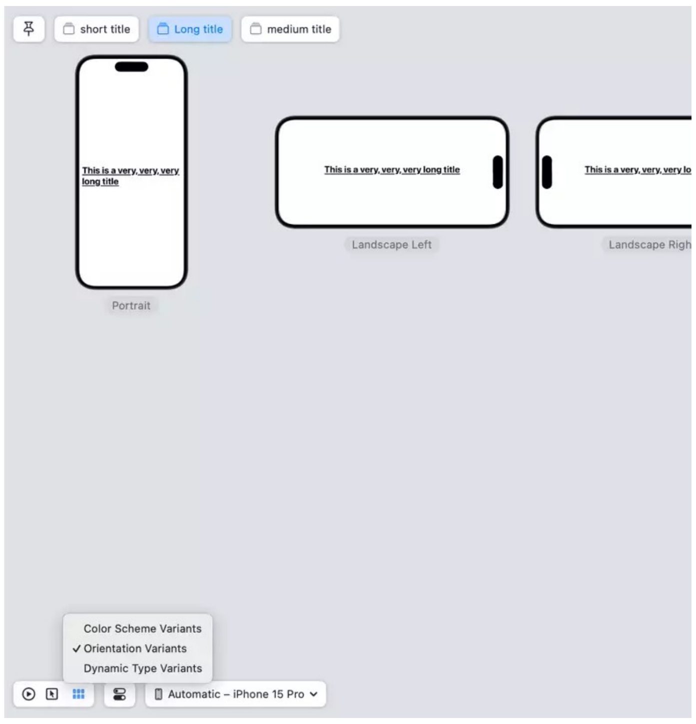
Dynamic Type
AX 3

Variants

To make testing and debugging more efficient, Xcode offers variants for **color schemes, orientations, and dynamic type sizes**. By enabling variants, you can compare different options side by side and quickly identify any issues or inconsistencies in your layout.

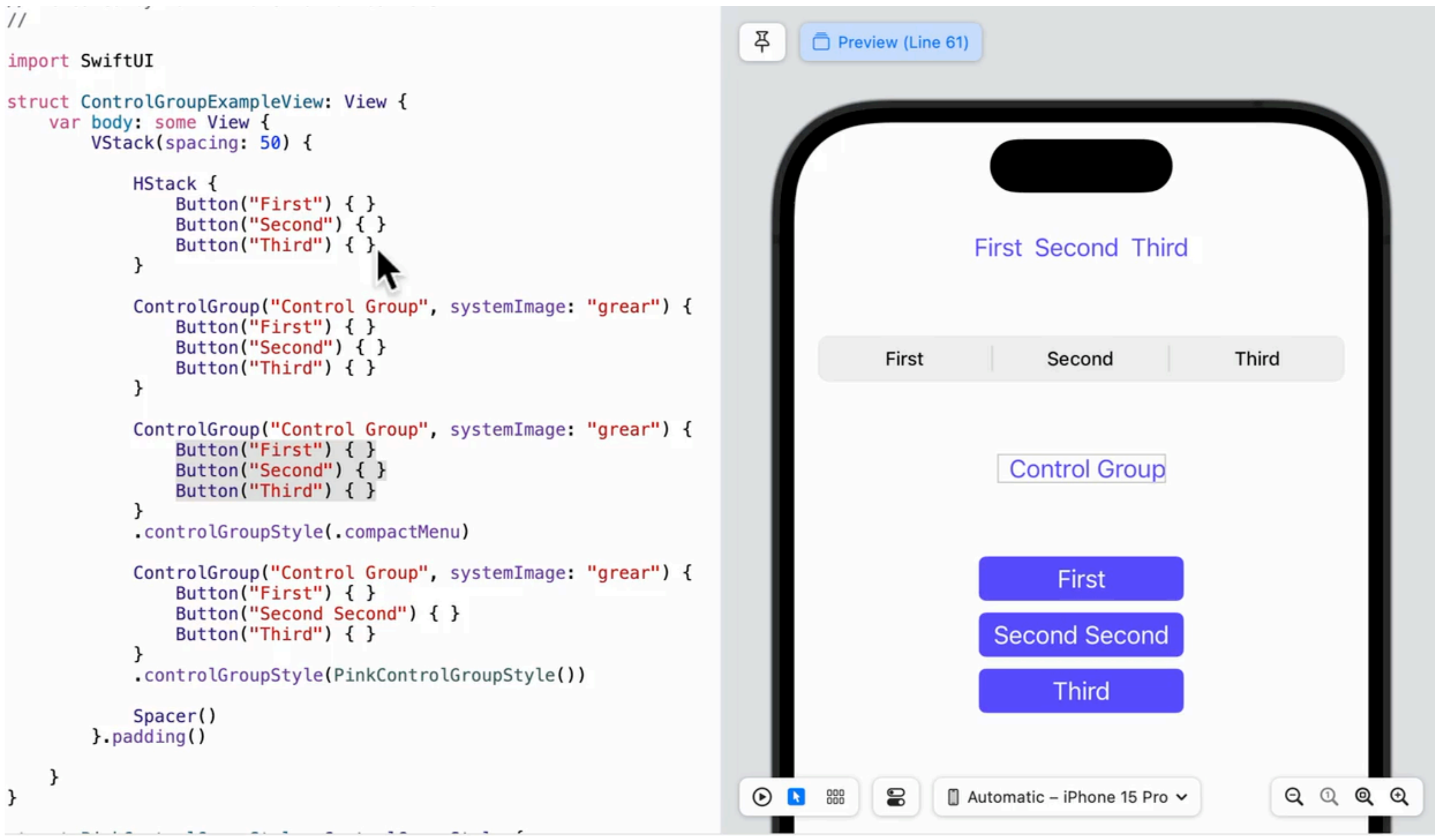


short title
Long title
medium title
This is a very, very, very long title
This is a very, very, very long title
Light Appearance
Dark Appearance
Automatic – iPhone 15 Pro



Selectable Preview

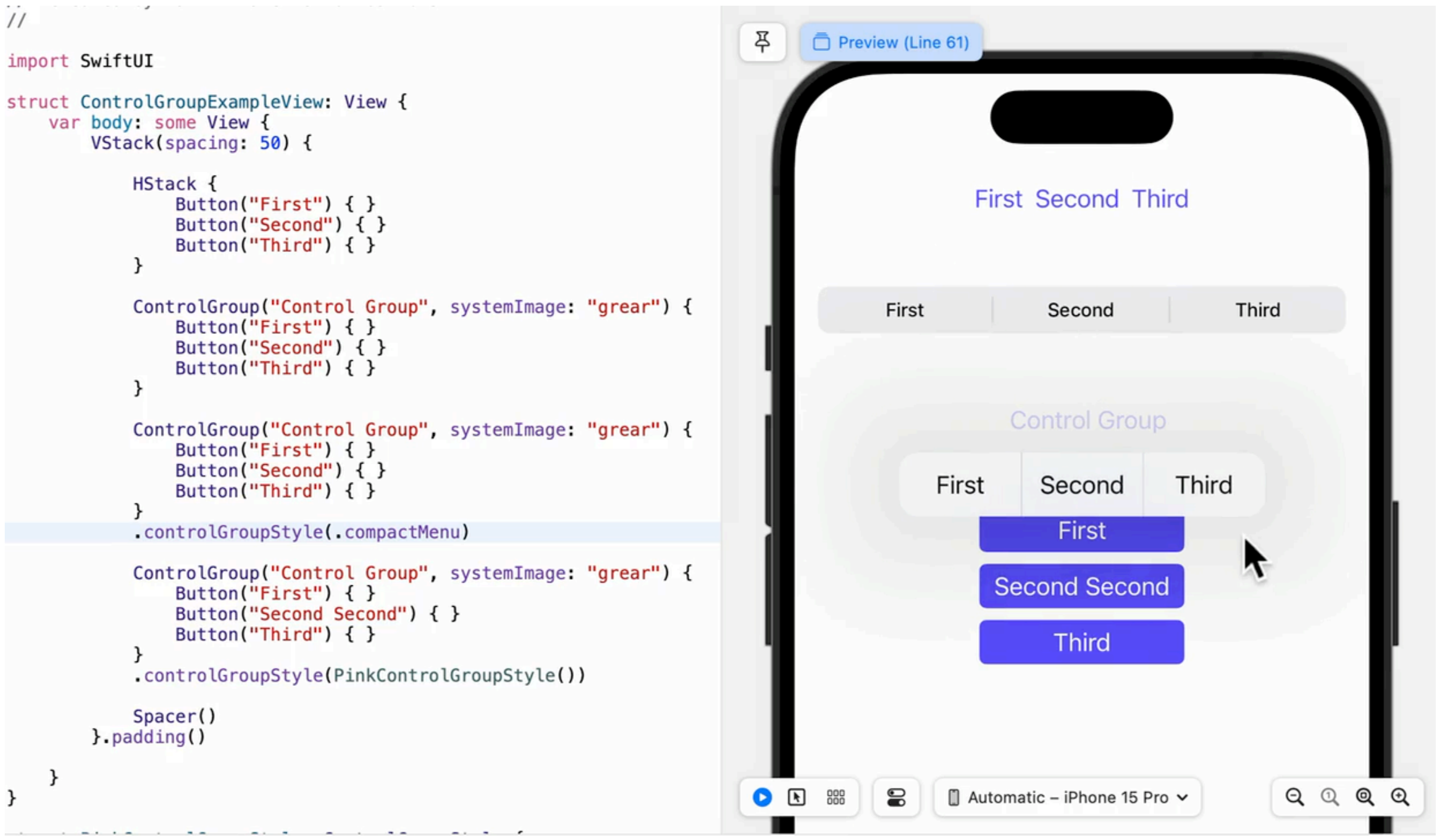
In the canvas area choose the second button in the bottom left corner to use the selectable preview feature. You can double-click on a specific view to highlight the corresponding code in the editor. This feature is particularly useful when working with complex views or collections, as it helps you identify which code snippets correspond to which views.



```swift
//

import SwiftUI

struct ControlGroupExampleView: View {
    var body: some View {
        VStack(spacing: 50) {

            HStack {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }
            .controlGroupStyle(.compactMenu)

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second Second") { }
                Button("Third") { }
            }
            .controlGroupStyle(PinkControlGroupStyle())

            Spacer()
        }.padding()

    }
}
```

Live Preview

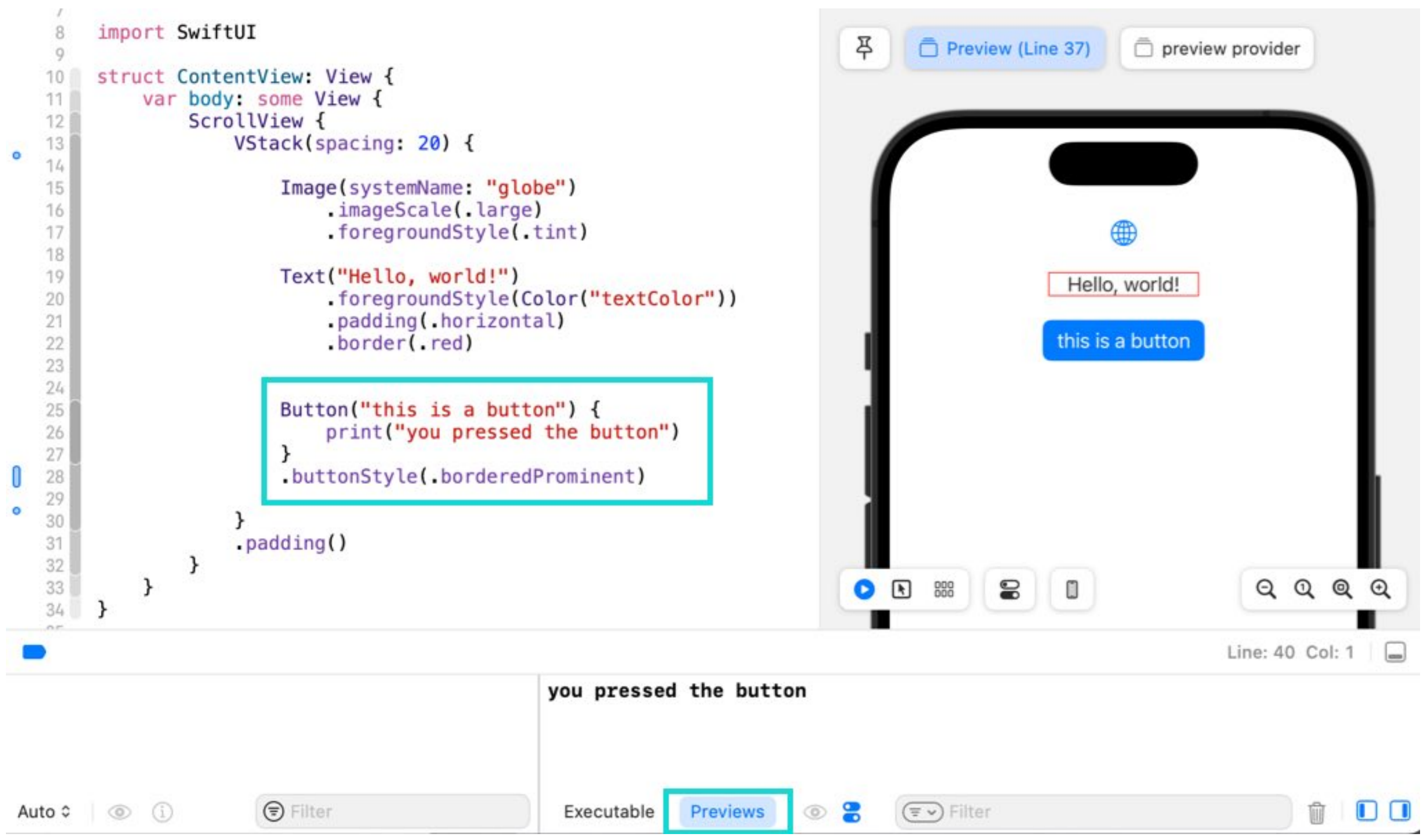
The live preview feature is handy for testing animations and interactions in real time. It allows you to see how your views respond to user interactions, such as tapping a button or scrolling a scroll view.



```swift
//

import SwiftUI

struct ControlGroupExampleView: View {
    var body: some View {
        VStack(spacing: 50) {

            HStack {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second") { }
                Button("Third") { }
            }
            .controlGroupStyle(.compactMenu)

            ControlGroup("Control Group", systemImage: "grear") {
                Button("First") { }
                Button("Second Second") { }
                Button("Third") { }
            }
            .controlGroupStyle(PinkControlGroupStyle())

            Spacer()
        }.padding()

    }
}
```

Debugging with Print Statements

To aid in debugging, the live preview also supports the use of print statements. You can add print statements to your code and observe the output in the debug area of the preview. This helps you verify if certain actions are being executed or if specific code paths are being triggered.

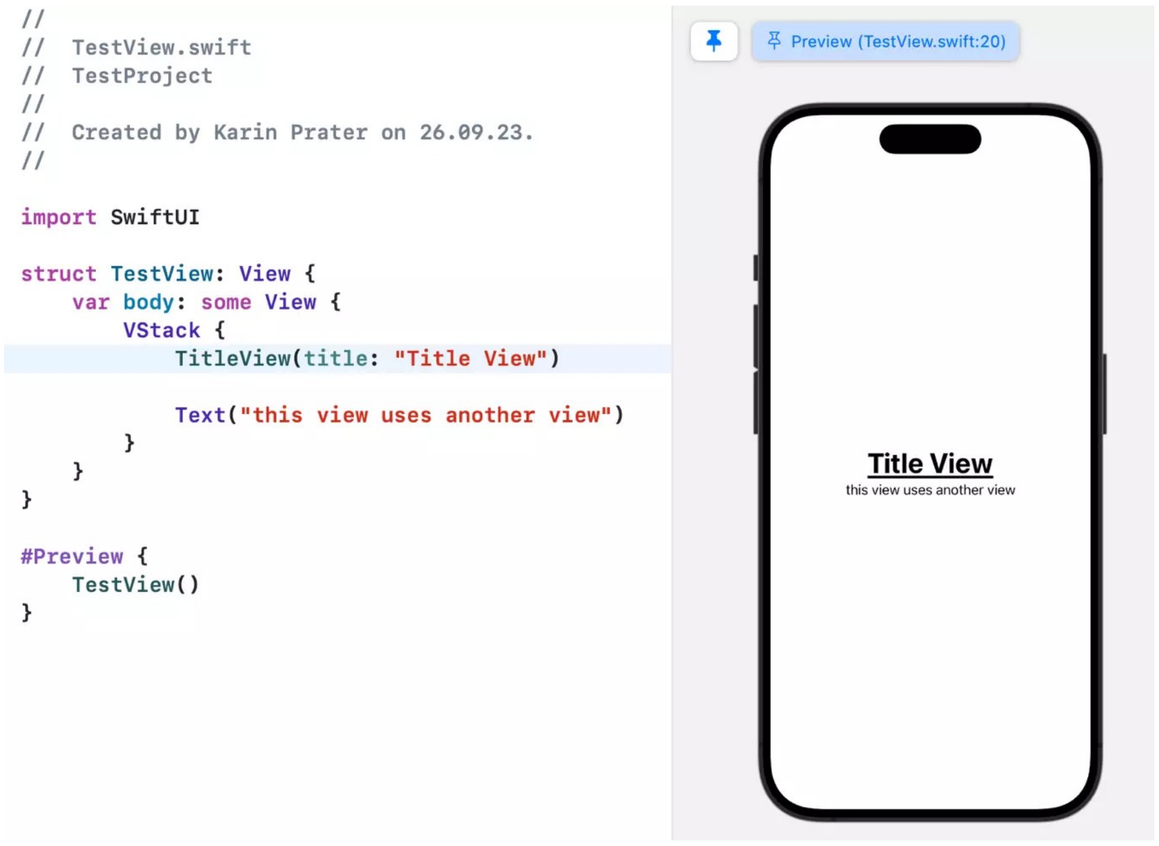


```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {

                Image(systemName: "globe")
                    .imageScale(.large)
                    .foregroundStyle(.tint)

                Text("Hello, world!")
                    .foregroundStyle(Color("textColor"))
                    .padding(.horizontal)
                    .border(.red)

                Button("this is a button") {
                    print("you pressed the button")
                }
                .buttonStyle(.borderedProminent)

            }
            .padding()
        }
    }
}
```

```
you pressed the button
```

Pin Previews

In Xcode, you have the option to pin previews. This allows you to work in the context of a specific view, even when navigating between different files. Pinned previews are displayed at the top and can be easily accessed for quick reference. If you want to remove pinned previews, simply tap on the pin button again.



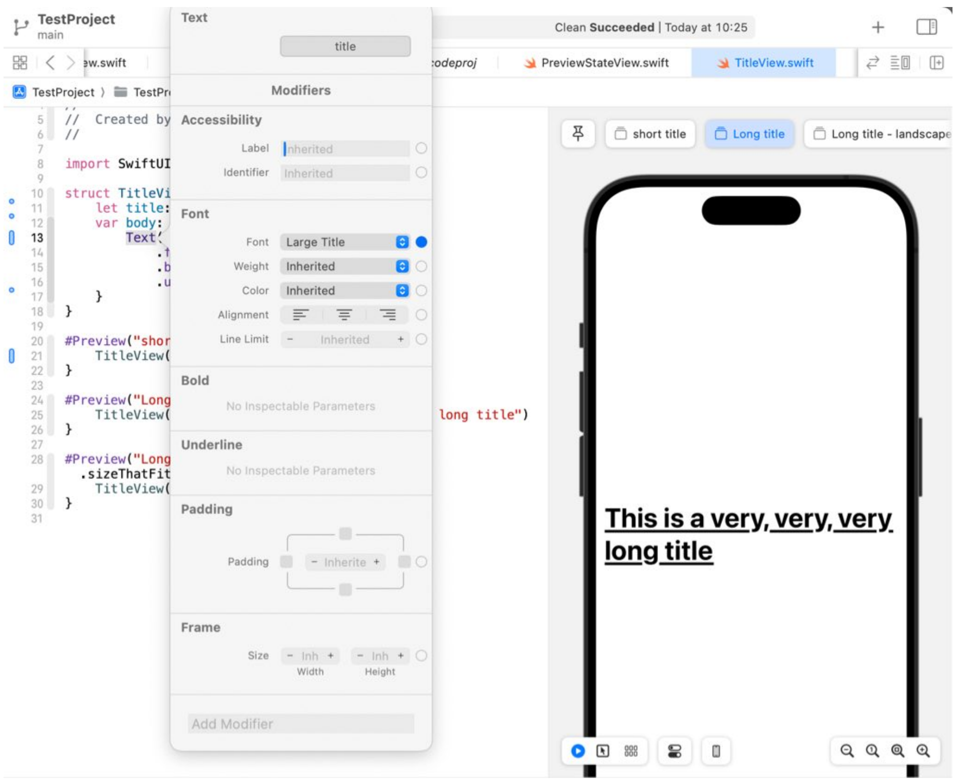
```swift
//
//  TestView.swift
//  TestProject
//
//  Created by Karin Prater on 26.09.23.
//

import SwiftUI

struct TestView: View {
    var body: some View {
        VStack {
            TitleView(title: "Title View")

            Text("this view uses another view")
        }
    }
}

#Preview {
    TestView()
}
```

1.3 QUICK AND EFFICIENTLY EDIT SWIFTUI VIEWS

One of the challenges we often face with SwiftUI is locating the tools we need to make changes to our views. It can be frustrating trying to figure out what can be modified and how to modify it.

SwiftUI Inspector

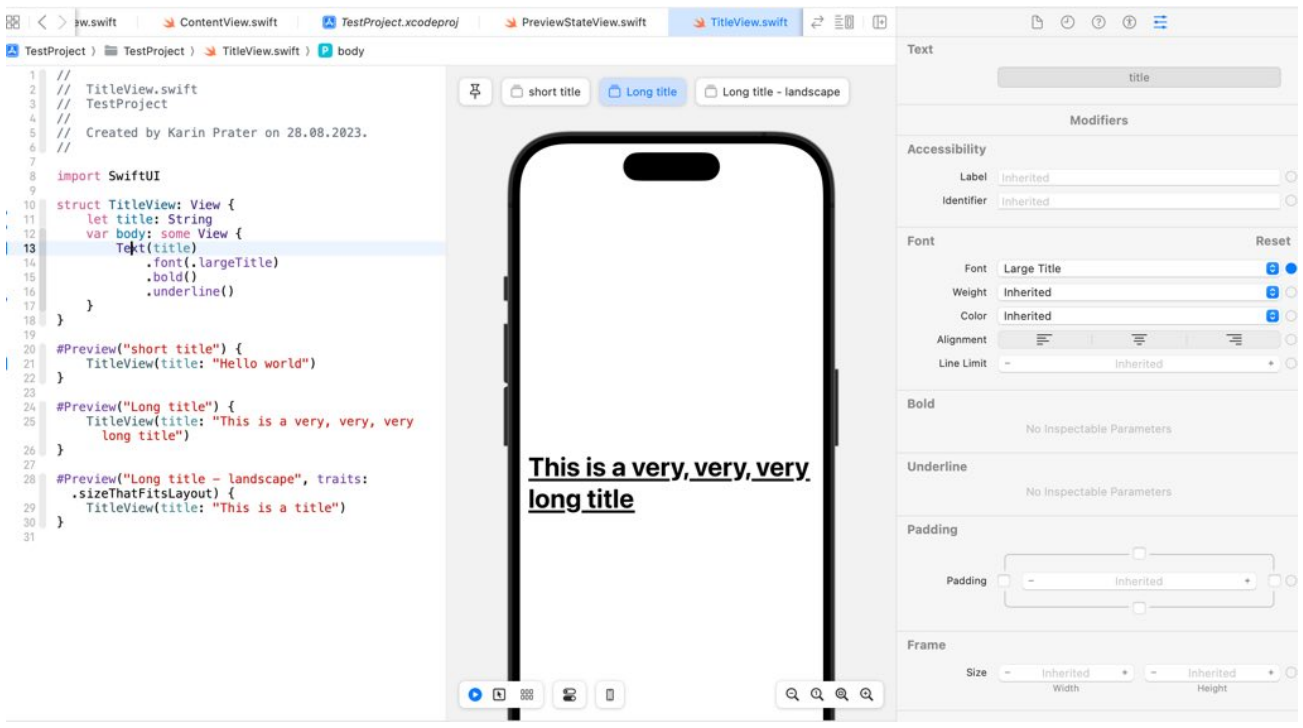
Simply control-click on an element and select "Show SwiftUI Inspector." This brings up a panel where you can quickly modify properties such as accessibility labels, paddings, frames, and even add additional modifiers like blur effects. You can also find these modifications in the editor, where they are represented as lines of code.



Text
title
Modifiers
Accessibility
Label: Inherited
Identifier: Inherited
Font
Font: Large Title
Weight: Inherited
Color: Inherited
Alignment
Line Limit: Inherited
Bold
No Inspectable Parameters
Underline
No Inspectable Parameters
Padding
Padding: Inherite
Frame
Size: Inh (Width), Inh (Height)
Add Modifier

Attribute Inspector area

Another useful tool is the Attribute Inspector area, where you can find a list of arguments and modifiers for a particular view. Here, you can scroll through and easily make changes to properties such as accessibility labels, padding, and more.

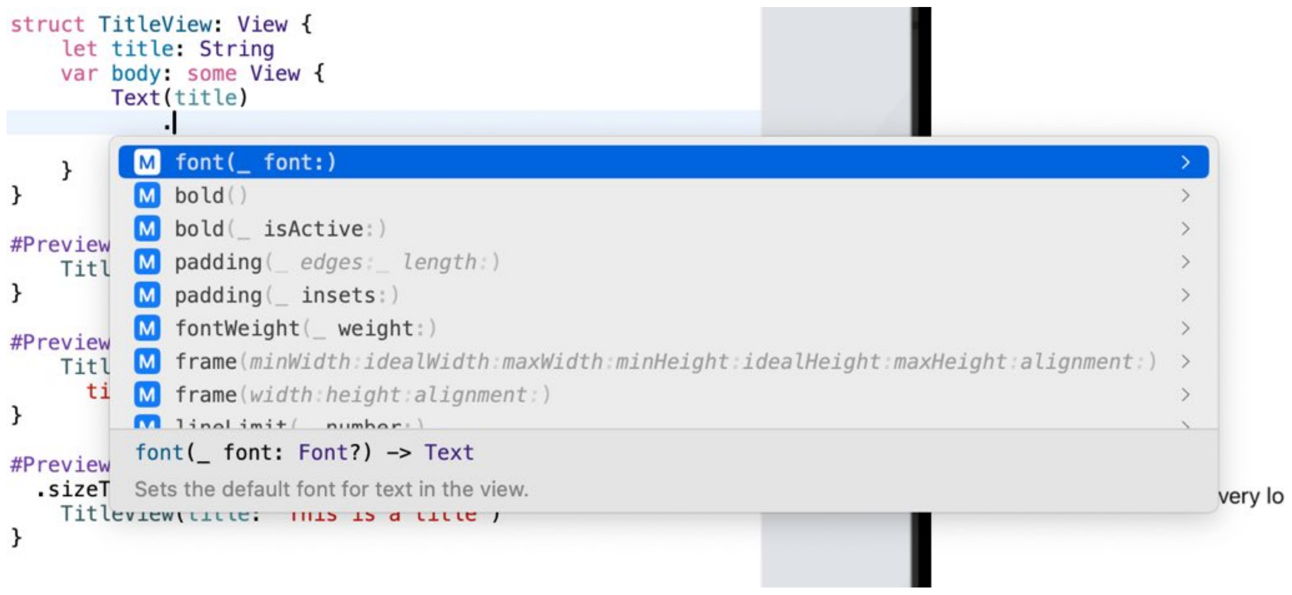


```
//
//  TitleView.swift
//  TestProject
//
//  Created by Karin Prater on 28.08.2023.
//

import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, very
      long title")
}

#Preview("Long title - landscape", traits:
  .sizeThatFitsLayout) {
    TitleView(title: "This is a title")
}
```

Text
title
Modifiers
Accessibility
Label: Inherited
Identifier: Inherited
Font — Reset
Font: Large Title
Weight: Inherited
Color: Inherited
Alignment
Line Limit: Inherited
Bold
No Inspectable Parameters
Underline
No Inspectable Parameters
Padding
Padding: Inherited
Frame
Size: Inherited (Width), Inherited (Height)

Xcode Auto Suggestions

Xcode has also become smarter in suggesting modifiers based on the context. For example, when working with a button, Xcode may suggest using a frame, navigation title, or padding. Similarly, when working with text, it may suggest using a multiline text alignment. These suggestions can save you time and effort in finding the right modifiers for your views.



```
struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .
```

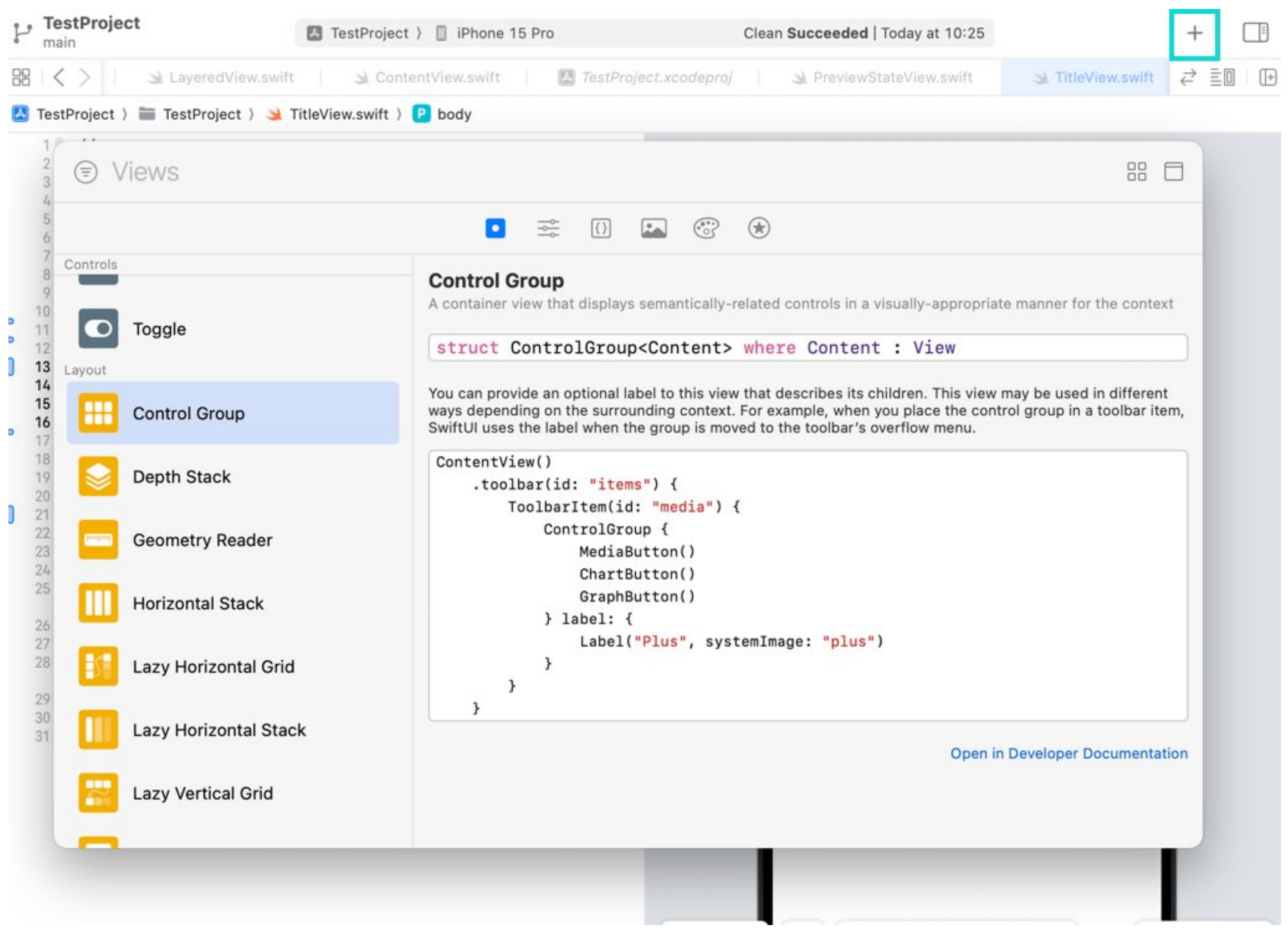
```
font(_ font:)
bold()
bold(_ isActive:)
padding(_ edges:_ length:)
padding(_ insets:)
fontWeight(_ weight:)
frame(minWidth:idealWidth:maxWidth:minHeight:idealHeight:maxHeight:alignment:)
frame(width:height:alignment:)
lineLimit(_ number:)
```

```
font(_ font: Font?) -> Text
```

Sets the default font for text in the view.

Xcode Library

If you prefer a visual approach, you can utilize the Xcode library by clicking on the plus button. Here, you'll find a collection of icons, symbols, assets, colors, images, and code snippets. The library is organized into categories such as modifiers, effects, layout, text, images, list, navigation, and styling. This allows you to quickly browse through different options and easily add them to your code.



Control Group

A container view that displays semantically-related controls in a visually-appropriate manner for the context

```
struct ControlGroup<Content> where Content : View
```

You can provide an optional label to this view that describes its children. This view may be used in different ways depending on the surrounding context. For example, when you place the control group in a toolbar item, SwiftUI uses the label when the group is moved to the toolbar's overflow menu.

```
ContentView()
    .toolbar(id: "items") {
        ToolbarItem(id: "media") {
            ControlGroup {
                MediaButton()
                ChartButton()
                GraphButton()
            } label: {
                Label("Plus", systemImage: "plus")
            }
        }
    }
```

Open in Developer Documentation

By familiarizing yourself with these built-in tools and resources, you can save valuable time that would otherwise be spent searching and googling for solutions. The documentation, in particular, can provide inspiration and helpful code snippets to enhance your SwiftUI skills.

1.4 DEBUGGING LAYOUT ISSUES

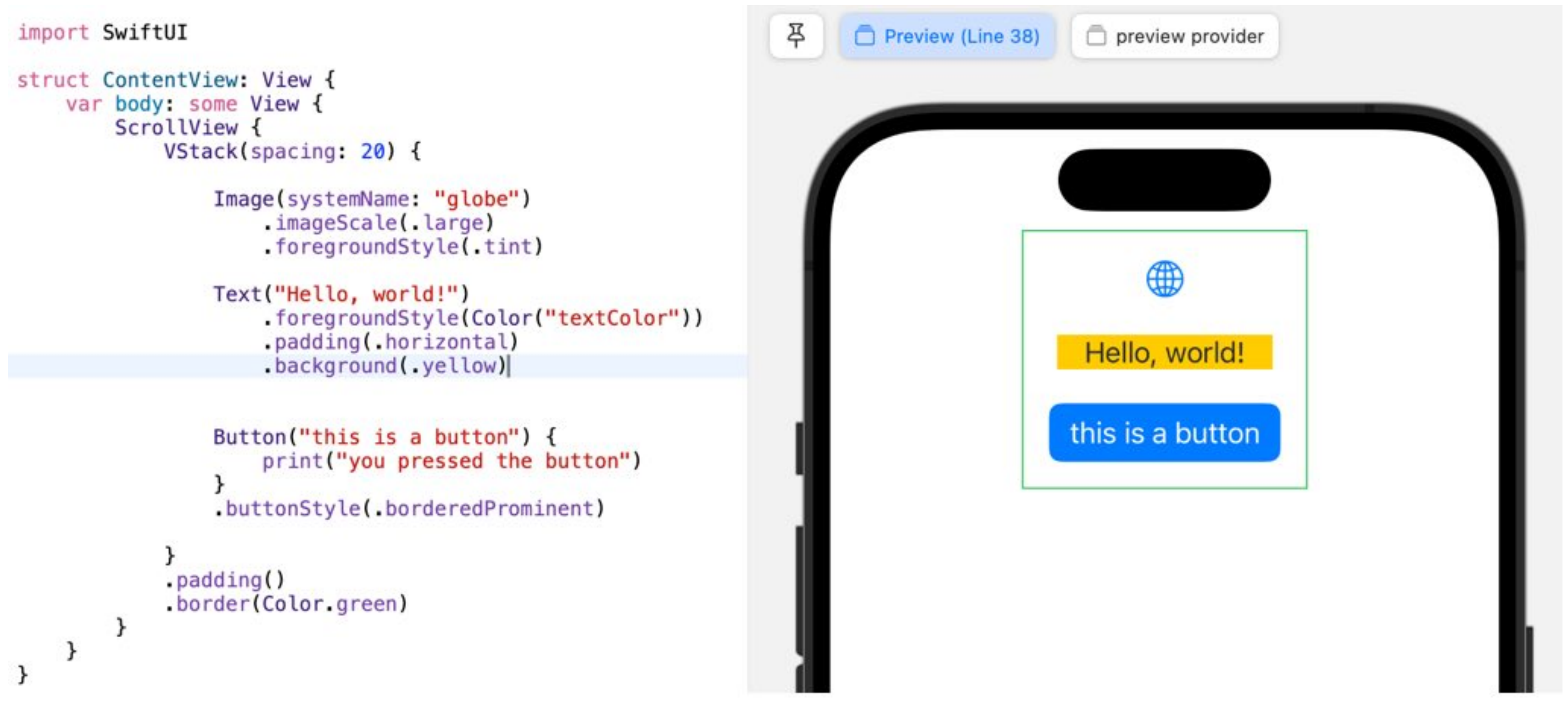
Debugging layout issues with SwiftUI can be a challenging task. I'm here to guide you through some helpful strategies that will make the process much easier.

Using the Selectable Preview

One useful tool for debugging layout issues is the inspector. By selecting a view, you can access information about its size and other properties. For example, you can identify if there is excessive padding causing unexpected spacing between views. By removing or adjusting the padding, you can resolve the issue and achieve the desired layout.

Adding Borders And Background Colors

Additionally, if you have multiple views that could be causing the problem, such as text or buttons, you can add borders or background colors to visually differentiate them. This allows you to pinpoint the specific view that needs adjustment. By zooming in and examining the highlighted view, you can identify any padding modifiers, frames, or offsets that may be causing layout inconsistencies.



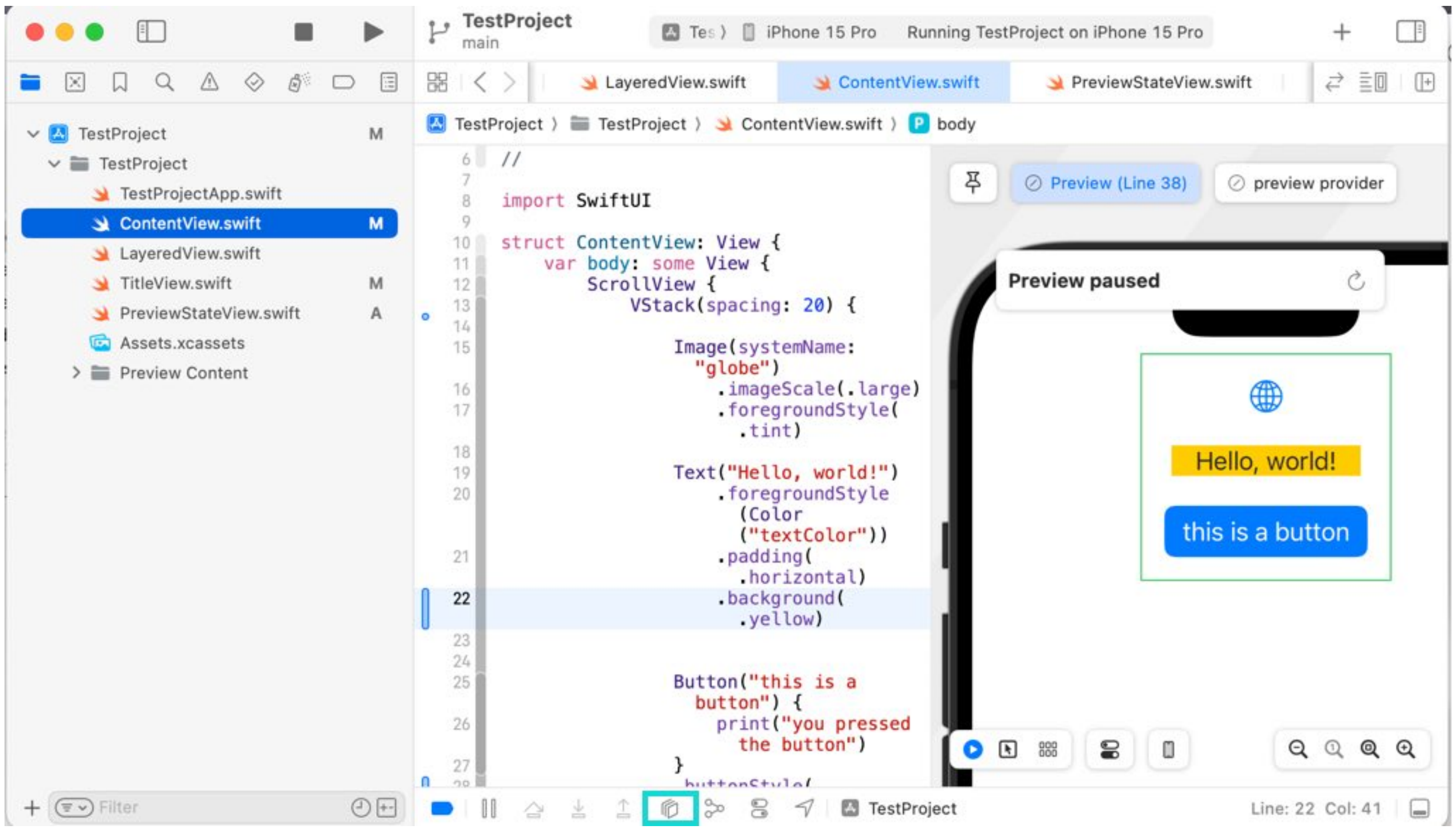
```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {

                Image(systemName: "globe")
                    .imageScale(.large)
                    .foregroundStyle(.tint)

                Text("Hello, world!")
                    .foregroundStyle(Color("textColor"))
                    .padding(.horizontal)
                    .background(.yellow)


                Button("this is a button") {
                    print("you pressed the button")
                }
                .buttonStyle(.borderedProminent)

            }
            .padding()
            .border(Color.green)
        }
    }
}
```

Debug View Hierarchy

Understanding the view hierarchy is crucial for debugging layout issues. SwiftUI provides a debug view hierarchy feature that allows you to visualize the stack of views used in your project. By using this feature, you can navigate through the hierarchy and gain insights into how views are structured.

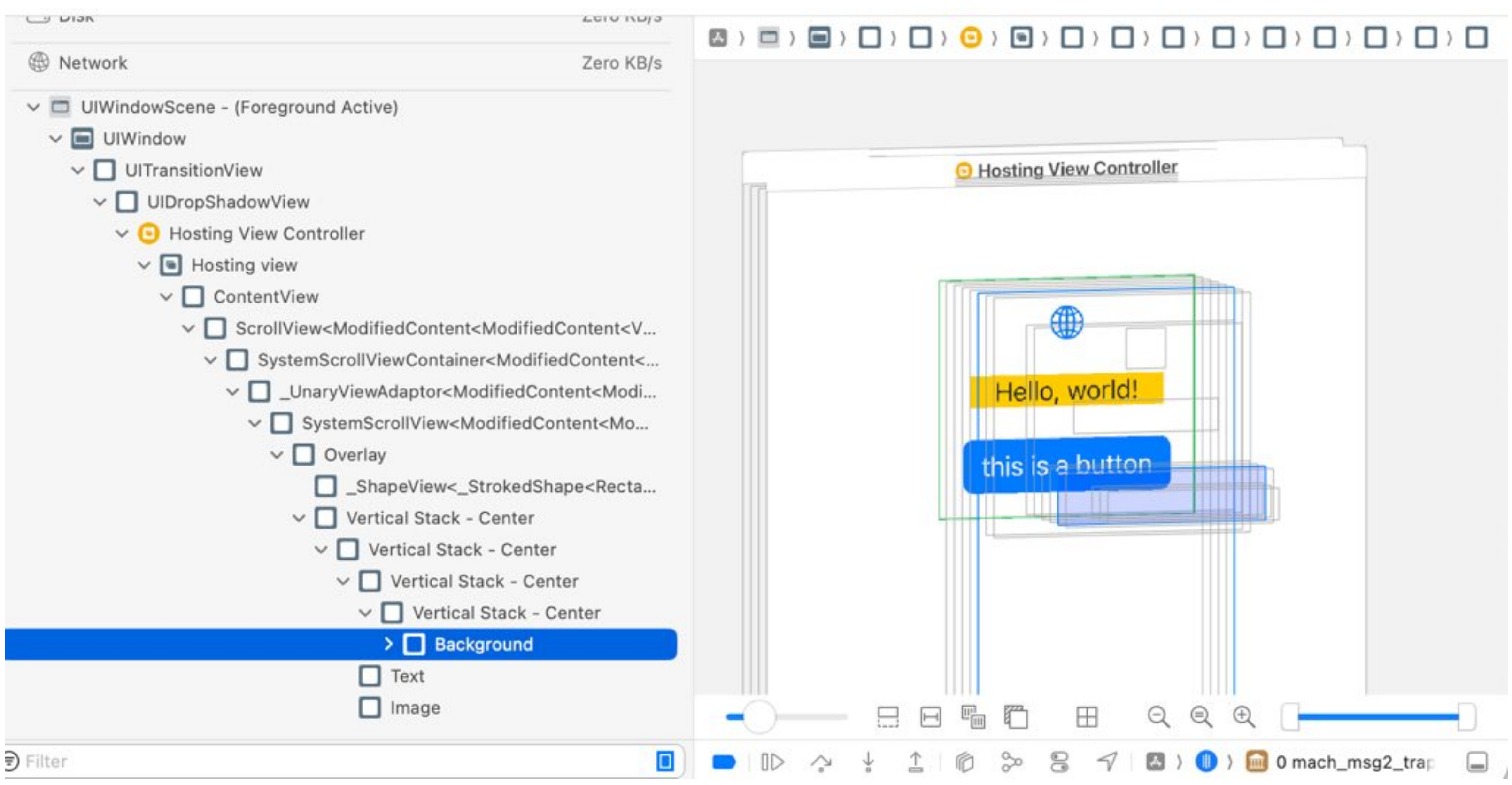
For example, if you want to locate where a specific title is defined, you can use the debug view hierarchy. By selecting the title, you can trace back to its parent views and identify the content view responsible for its creation. This feature is especially helpful when dealing with layered views or complex layouts.

Run your project and select the “Debug View Hierarchy” button at the bottom of the Xcode window:



```
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {

                Image(systemName:
                  "globe")
                    .imageScale(.large)
                    .foregroundStyle(
                      .tint)

                Text("Hello, world!")
                    .foregroundStyle
                      (Color
                      ("textColor"))
                    .padding(
                      .horizontal)
                    .background(
                      .yellow)


                Button("this is a
                  button") {
                    print("you pressed
                      the button")
                }
```

Xcode will pause the simulator and open the Debug View Hierarchy. In the left navigator pan you can select the views and layers:



Monitoring View Updates

Sometimes, you may encounter performance issues or frequent redrawing without understanding which views are causing the problem. In such cases, it's useful to monitor view updates and identify which views are being recreated. You can achieve this by using the print changes statement:

```
struct TitleView: View {
    let title: String

    var body: some View {
       Self._printChanges()
       return Text(title)
            .font(.largeTitle)
            .bold()
            .underline()

    }
}
```

Thus, you can track how many times the view is recreated or updated. This can provide valuable insights into the impact of changes on specific views.

By observing the print statements in the debugger, you can determine which views are being updated and how often. This helps you understand the relationship between view updates and potential layout issues.

1.5 SWIFTUI TREE OF DOOM

When working with SwiftUI, you may encounter situations where your **views become large and complex**, leading to slow previews or unhelpful error messages from Xcode. This is commonly referred to as the "tree of doom" in SwiftUI, where nesting levels can become overwhelming. Here are a few strategies to work against this.

Extracting Subviews

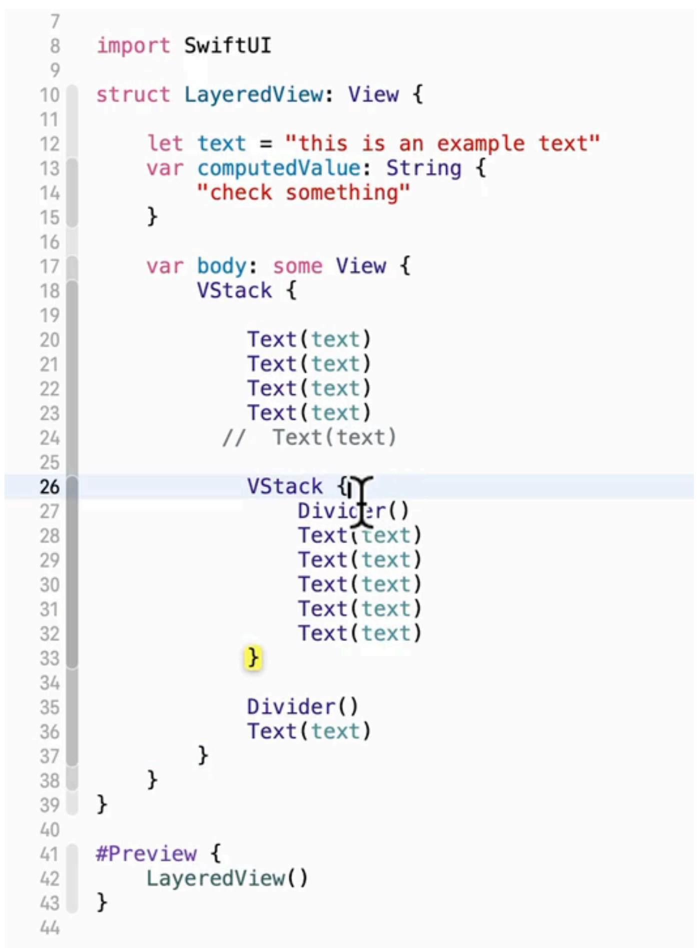
To simplify the view hierarchy, you can use the "Extract Subview" feature by Ctrl-clicking on the problematic view. This will extract the view into a separate subview. You can then rename it to something more meaningful. Personally, I prefer moving these subviews to separate files rather than leaving them within the same file. This way, I can easily navigate through each view file and see its preview directly. It also prevents the subviews from getting lost amidst the main view code.

Small Views

Remember, it's always beneficial to break down your views into smaller, more manageable pieces. Personally, I find it helpful to keep the body of a view smaller than 100 lines. However, you should experiment and find what works best for you, as everyone's preferences and project requirements may vary.

Code Highlighting

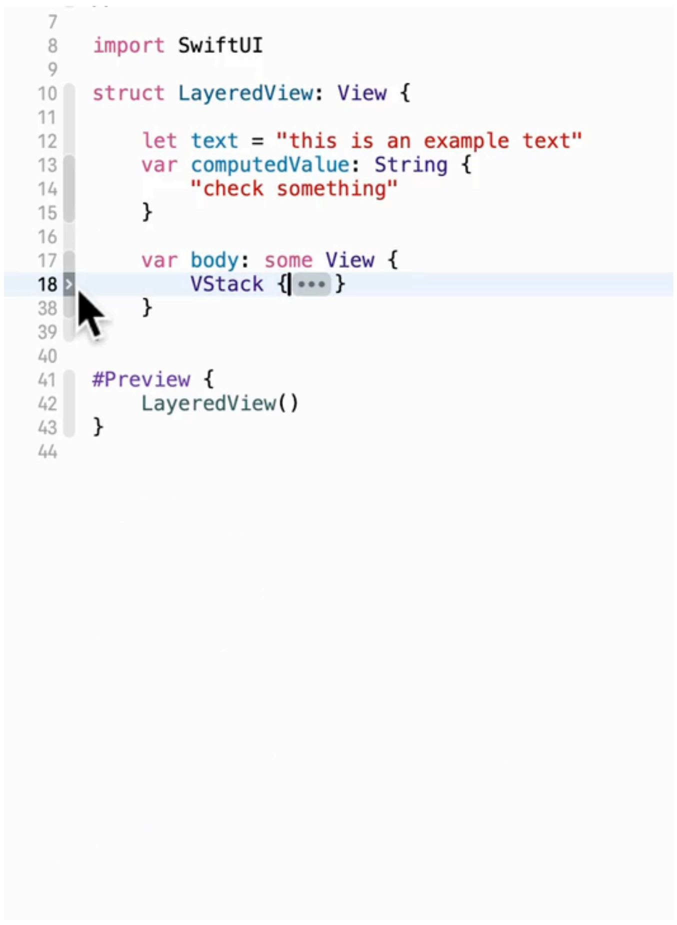
If you encounter long containers, such as a VStack with numerous subviews, it can be challenging to identify where the container ends. To solve this, you can utilize Xcode's highlighting feature. By clicking on the curly braces at the beginning of the container, Xcode will highlight the corresponding closing curly brace, indicating the end of the container. This helps in making changes or adding modifiers to the container.



```swift

import SwiftUI

struct LayeredView: View {

    let text = "this is an example text"
    var computedValue: String {
        "check something"
    }

    var body: some View {
        VStack {

            Text(text)
            Text(text)
            Text(text)
            Text(text)
          //  Text(text)

            VStack {
                Divider()
                Text(text)
                Text(text)
                Text(text)
                Text(text)
                Text(text)
            }

            Divider()
            Text(text)
        }
    }
}

#Preview {
    LayeredView()
}

```

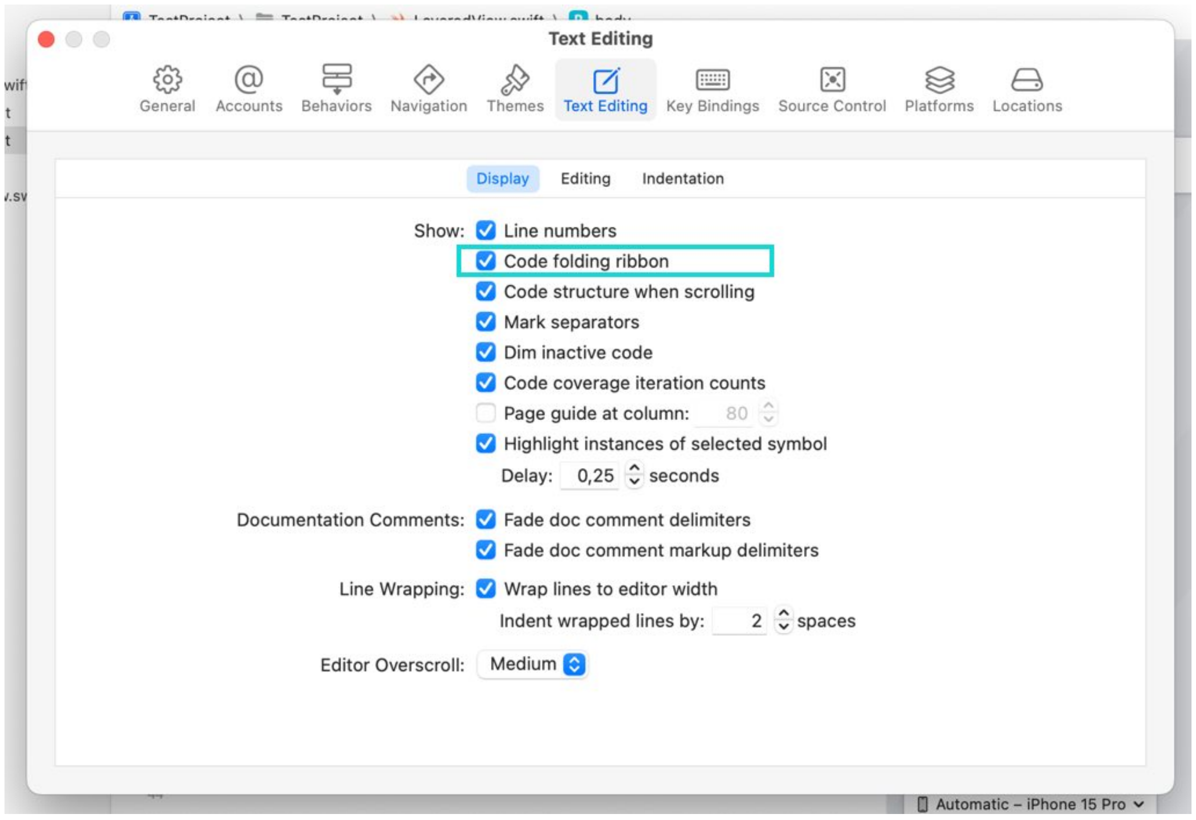
Folding Ribbons

In cases where your code extends beyond the visible area, you can make use of the code folding ribbons. These ribbons allow you to hide or show specific sections of your code, making it easier to focus on the relevant parts.



```swift
import SwiftUI

struct LayeredView: View {

    let text = "this is an example text"
    var computedValue: String {
        "check something"
    }

    var body: some View {
        VStack {•••}
    }
}

#Preview {
    LayeredView()
}
```

To enable or disable the ribbons, go to Xcode settings, specifically the “Text Editor” area, and toggle the “Show Code Folding Ribbons” option.



```
Show: [✓] Line numbers
      [✓] Code folding ribbon
      [✓] Code structure when scrolling
      [✓] Mark separators
      [✓] Dim inactive code
      [✓] Code coverage iteration counts
      [ ] Page guide at column: 80
      [✓] Highlight instances of selected symbol
          Delay: 0,25 seconds
Documentation Comments: [✓] Fade doc comment delimiters
                        [✓] Fade doc comment markup delimiters
Line Wrapping: [✓] Wrap lines to editor width
               Indent wrapped lines by: 2 spaces
Editor Overscroll: Medium
```

By simplifying your complex SwiftUI views, you can enhance the performance of previews, reduce errors, and make your code more maintainable. So, don’t just rely on Xcode’s default behavior, take control of your code and make it more understandable and efficient.

1.6 TYPICAL PROBLEMS WITH XCODE AND SWIFTUI AND HOW TO FIX THEM

Sometimes, while working with Xcode and SwiftUI, you may encounter certain issues that can be a bit frustrating. Unfortunately, Xcode doesn't always provide the most informative error messages. In this section, I will walk you through some common scenarios for errors, explain why they occur, and show you how to resolve them.

Invalid Redefinition of View Names

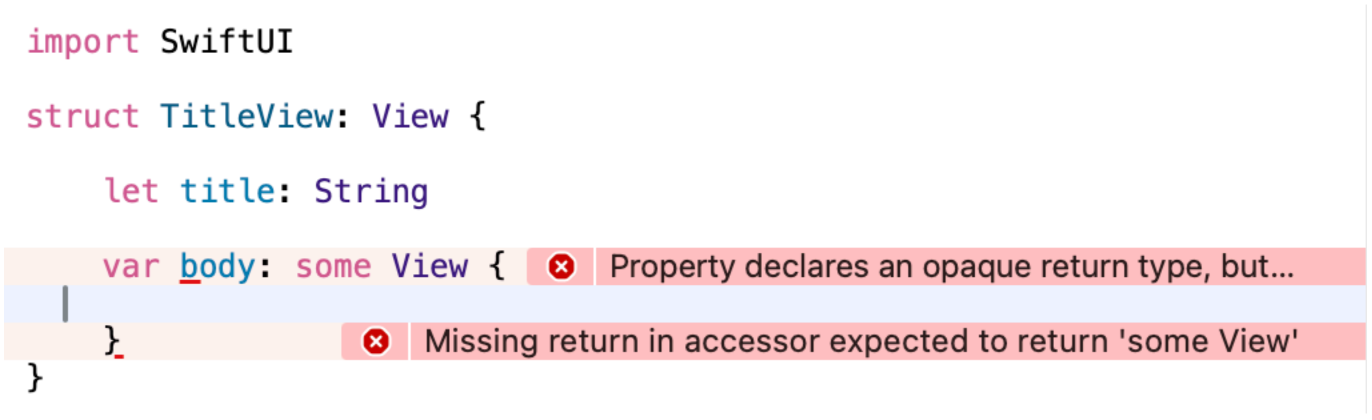
One error that you might come across is when you have a view with the same name declared multiple times. For instance, if you copy a struct called "LayeredView" to your ContentView, you will get an error message saying *"invalid redeclaration of LayeredView."* This error occurs because you have already declared a view with the same name. To fix this, you can use the search function to locate the duplicate view and rename it accordingly.

Uninitialized Properties

Another issue you might encounter is when you have a property declared within a struct but it is not initialized. Xcode will display an error message stating *"property declared as object return type but has no initializer."* To resolve this error, you need to provide an initial value for the property. Even if it's just a placeholder like a Text view, it will help Xcode infer the underlying type correctly.

Missing Return Value for Text Views

Similarly, you might face problems when using Text views. If you don't return anything within the body property, an error will be thrown. For example:



```
import SwiftUI

struct TitleView: View {

    let title: String

    var body: some View {    ⊗ Property declares an opaque return type, but...

    }                ⊗ Missing return in accessor expected to return 'some View'
}
```

To fix this, make sure to return a view within the body property. You can use a Text view with some content, even if it's temporary. However, it's best practice to declare constants or computed values outside of the body property for better code organization.

Missing Environment Objects in SwiftUI Previews

If you use environment objects in your views or subviews, make sure to also add them in the preview. Similarly, you need to inject a context when working with CoreData or SwiftData:

```
#Preview {
    ContentView()
        .environment(MyViewModel())
        .environment(\.managedObjectContext, NewContext())
}
```

Troubleshooting Tips

If you encounter persistent issues and can't figure out the problem, here are a few troubleshooting tips:

1. Uncomment the views you just created and implement them one by one to identify any potential errors.

2. **Clean the build folder** by going to Product > Clean Build Folder and then rebuild your project.

3. Sometimes, **running the project** instead of relying solely on the preview can help resolve issues.

4. In extreme cases, **restarting Xcode or even your Mac** might be necessary to resolve stubborn issues.

Overall, Xcode previews with the Canvas feature are incredibly useful and can save you a lot of time. In the upcoming lessons, you will see how these previews can streamline your development process by eliminating the need to build and run your project repeatedly.

2. PRIMITIVE LAYOUT COMPONENTS

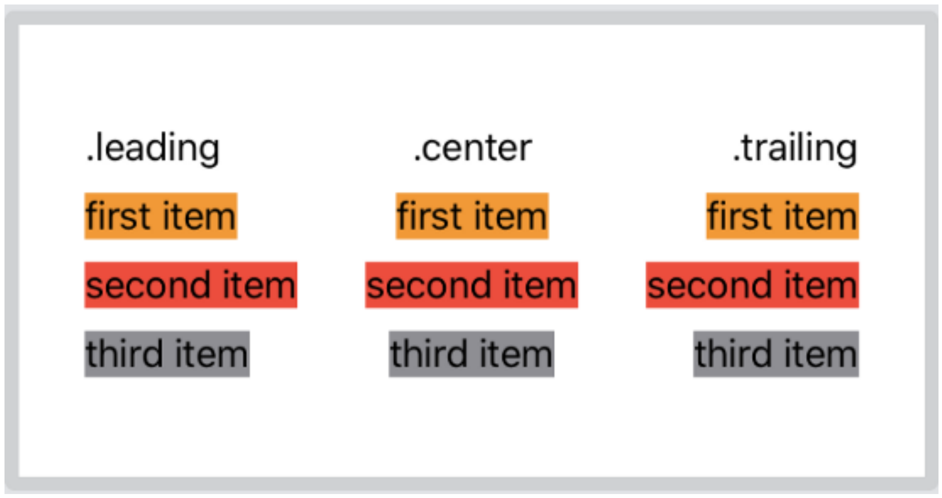
2.1 VSTACK, HSTACK AND ZSTACK

VStack - Vertical Stacking

VStack is a container view that arranges its child views vertically. To create a VStack, you can use the “Embed in VStack” option after control-clicking on the view. By default, a VStack adjusts its size to fit its child views. The size of the VStack is determined by the space occupied by its children.

You can customize the alignment and spacing of the child views within the VStack. For example, you can align the views to the leading edge and set a spacing of 20 between them. Additionally, you can nest multiple VStacks to create more complex layouts.

```
VStack(alignment: .center,
       spacing: 10) {

    Text("first item")
        .background(Color.yellow)

    Text("second item")
        .background(Color.red)

    Text("third item")
        .background(Color.gray)
}
```

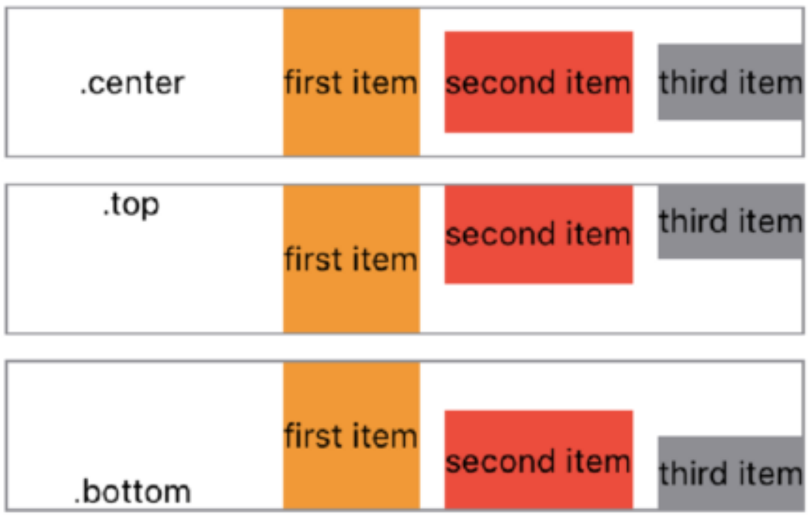


HStack - Horizontal Stacking

HStack is a container view that arranges its child views horizontally. Similar to VStack, you can use the "Embed in HStack” option to create an HStack. The alignment property of an HStack determines how the child views are aligned vertically.

You have various alignment options available such as center, top, bottom, first text baseline, and last text baseline. These options allow you to align the child views based on their text baselines or other criteria. By changing the font size or adding different-sized views, you can observe the effects of alignment within an HStack.

```
HStack(alignment: .firstTextBaseline,
             spacing: 10) {

          Text("first item”)
              .background(Color.yellow)

          Text("second item")
              .background(Color.red)

          Text("third item")
              .background(Color.gray)
      }
```



ZStack - overlay stacking

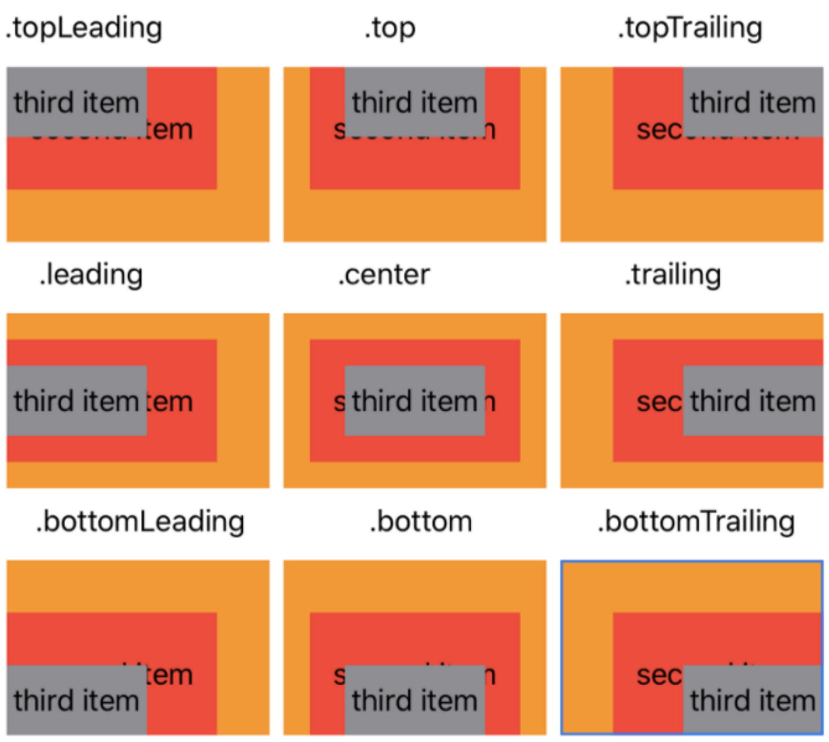
ZStack is a container view that stacks its child views on top of each other, creating a layered effect. The order in which the child views are added to the ZStack determines their stacking order, with the first view being the furthest behind.

You can adjust the stacking order using the **zIndex view modifier** or by changing the order of the child views. Additionally, you can customize the alignment of the child views within the ZStack, such as top, leading, center, or combinations of them.

```
      ZStack(alignment: .center) {
          Text("first item")
              .padding(.vertical, 20)
              .background(Color.yellow)
              .zindex(2)

          Text("second item")
              .padding(.vertical, 10)
              .background(Color.red)

          Text("third item")
              .background(Color.gray)
      }
```

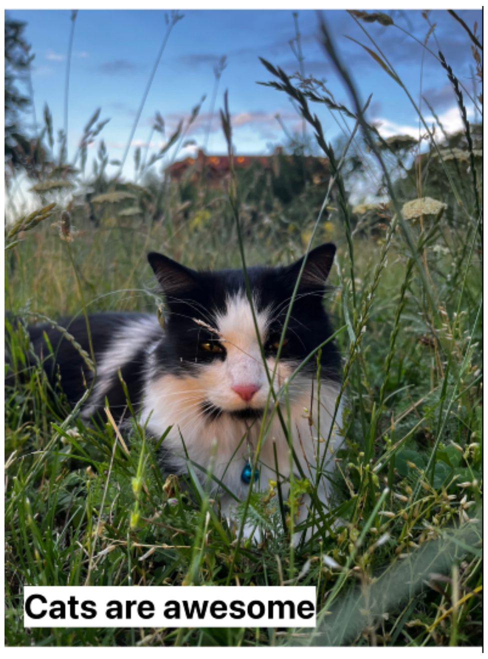
You can also use the alignment property to align the views within the ZStack.

For Text views, you might also want to use **.leadingLastTextBaseline**, and **.trailingFirstTextBaseline** etc.



ZStacks are particularly useful when you want to overlay views on top of each other. You can use them to create visually appealing effects, such as combining images with text or applying gradients and backgrounds.

```
struct CatExampleView: View {
    var body: some View {
        ZStack(alignment: .bottomLeading) {
            ResizableImageView(imageName: "cat_1")

            Text("Cats are awesome")
                .font(.title).bold()
                .background(Color.white)
                .padding()
        }
    }
}
```

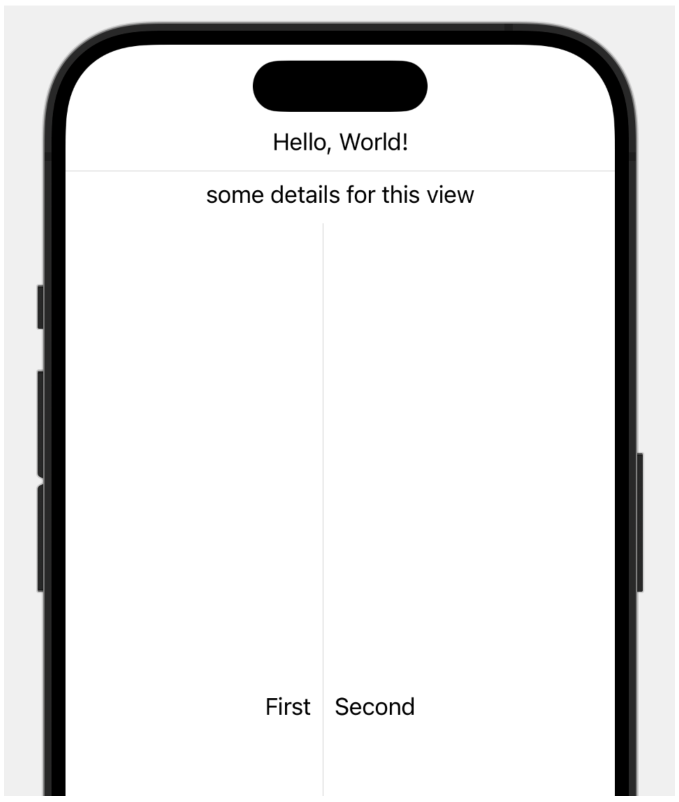


2.2 DIVIDER AND SPACER

In this section, we will explore two fundamental layout views in SwiftUI: dividers and spacers. These views play a crucial role in organizing and structuring your user interface.

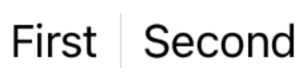
Imagine you have a VStack with various pieces of information. You want to visually separate two specific views within this stack. To achieve this, you can simply add a Divider. This will create a thin line between the two views, providing a clear visual distinction. Depending on whether you place the divider in a VStack or an HStack, it will appear as a horizontal or vertical line respectively. SwiftUI is smart enough to adapt the appearance of the divider based on its container stack.

```
struct DividerExampleView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")

            Divider()
            Text("some details for this view")

            HStack {
                Text("First")
                Divider()
                Text("Second")
            }
          //  .fixedSize()
        }
    }
}
```


Hello, World!
some details for this view
First Second

Dividers, being a “greedy” view, strive to occupy as much space as possible. This means they expand to fill the available space in the layout. For instance, if you want to minimize the height of the divider and have it only as tall as the two text views, you can use the **fixedSize**() modifier.

```
    HStack {
        Text("First")
        Divider()
        Text("Second")
    }
    .fixedSize(horizontal: false, vertical: true)
```

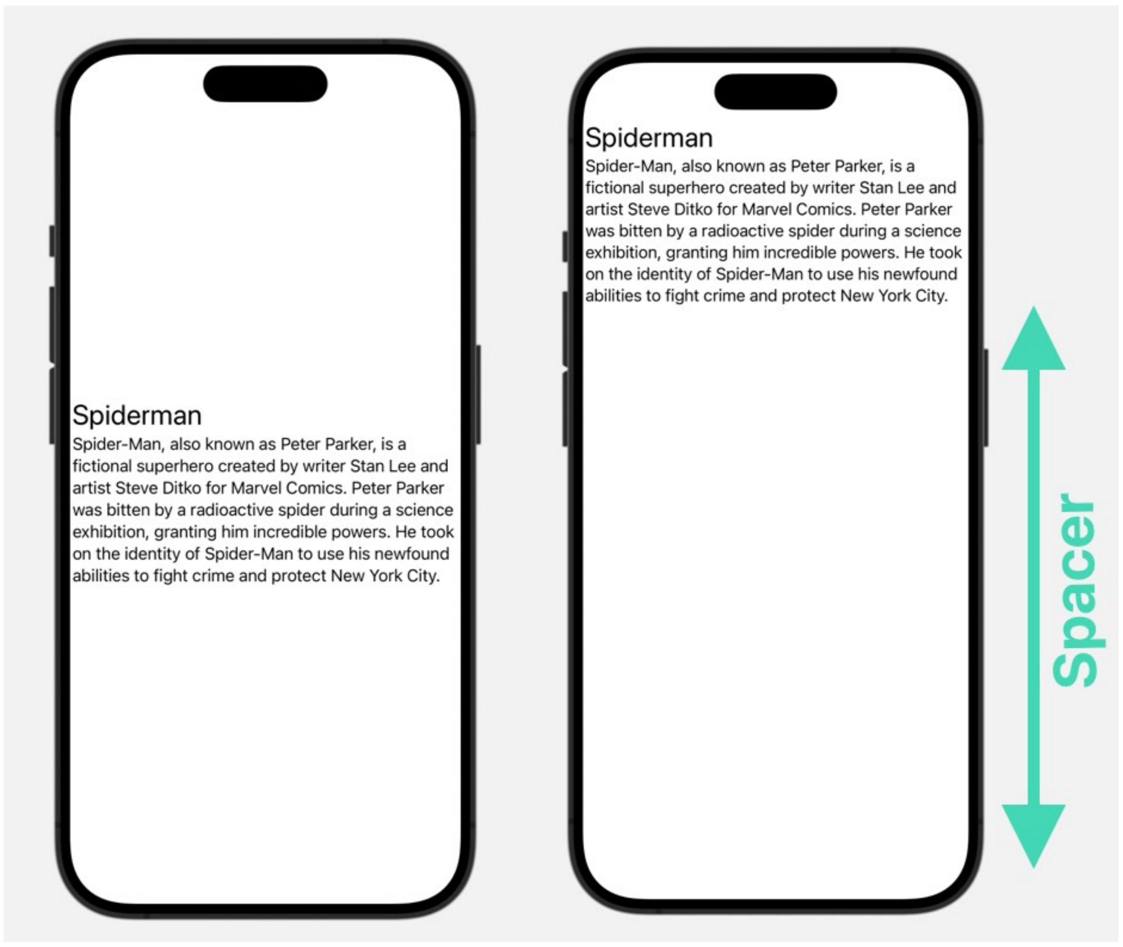


First | Second

By applying the **fixedSize**() modifier, the divider’s height is constrained to match the height of the views it separates, resulting in a more compact appearance.

Moving on to **Spacer**, they are incredibly useful for controlling the distribution of space within a layout. A spacer view takes up all the available space in a given axis and pushes the surrounding views accordingly.

Imagine you have a vertical stack and you want to position it at the top of the screen instead of the default center alignment. To achieve this, you can use a spacer to expand the stack’s height and push it to the top.



Spiderman
Spider-Man, also known as Peter Parker, is a fictional superhero created by writer Stan Lee and artist Steve Ditko for Marvel Comics. Peter Parker was bitten by a radioactive spider during a science exhibition, granting him incredible powers. He took on the identity of Spider-Man to use his newfound abilities to fight crime and protect New York City.

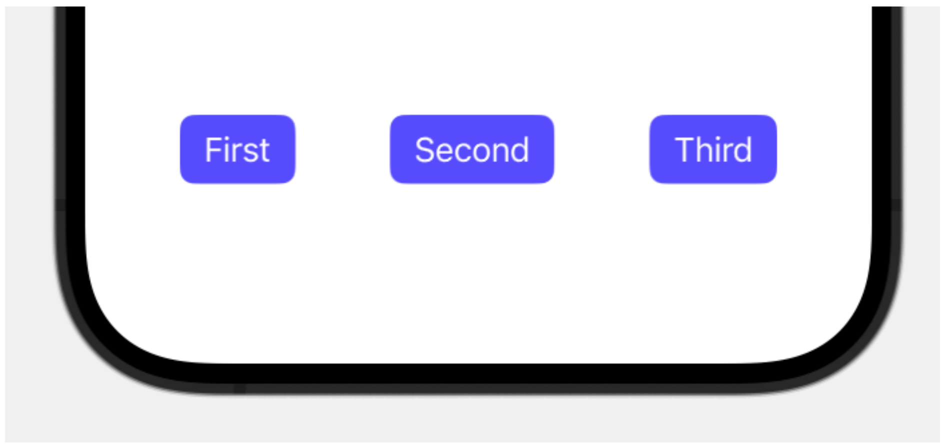
Spiderman
Spider-Man, also known as Peter Parker, is a fictional superhero created by writer Stan Lee and artist Steve Ditko for Marvel Comics. Peter Parker was bitten by a radioactive spider during a science exhibition, granting him incredible powers. He took on the identity of Spider-Man to use his newfound abilities to fight crime and protect New York City.

```
struct SpacerExampleView: View {
    let superhero = SuperHero.example
    var body: some View {
        VStack(alignment: .leading) {
            Text(superhero.name)
                .font(.title)
            Text(superhero.biography)

            Spacer()
        }
    }
}
```

By adding the Spacer view, it occupies the remaining space at the bottom of the screen and pushes the stack upwards, aligning it with the top edge.

Spacers can also be used to adjust the spacing between views. For example, if you have three buttons arranged horizontally, you can add spacers between them to control their positioning.

```
HStack(spacing: 0) {
    Spacer(minLength: 0)
    Button("First") { }
    Spacer(minLength: 10)
    Button("Second") { }
    Spacer(minLength: 10)
    Button("Third") { }
    Spacer(minLength: 0)
}
```



In this case, the spacers distribute the available space evenly between the buttons, resulting in a visually appealing layout. Alternatively, you can use the **Color view as a spacer** by setting its background color to match the desired spacing.

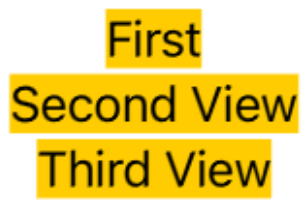
Using dividers and spacers in your SwiftUI layouts allows for greater flexibility and adaptability across different screen sizes.

2.3 GROUP

In SwiftUI, groups are container views that do not handle the layout of their children. Instead, the layout is determined by the container view they are placed in, such as an HStack or VStack. However, groups offer a convenient way to apply the same view modifier to all their children individually.

Where groups shine is their ability to apply view modifiers to multiple child views simultaneously. Let's say you want to add a yellow background to all the text views. With a group, you can simply apply the background modifier to the group:

```
Group {
    Text("First")
    Text("Second View")
    Text("Third View")
}
.background(Color.yellow)
```



Unlike when using a VStack or HStack, where the background modifier would be applied to the entire stack, the group allows you to apply the same modifier to each individual child view. This can be incredibly convenient, especially when you want to avoid duplicating code.

Another use case for groups is when you have conditional code that requires specific view modifiers. For example, let's say you want to display a different view based on whether a user is logged in or not:

```
struct GroupExampleView: View {
    let isLoggedIn: Bool
    var body: some View {
       VStack {
            Group {
                if isLoggedIn {
                    Text("Thank you for signing up")
                } else {
                    Text("You need to log in to get access")
                }
            }
            .foregroundStyle(Color.blue)
        }
    }
}
```

In this case, applying the foregroundColor modifier directly to the conditional code would result in a crash. However, **by wrapping the condition in a group, you can apply the modifier** without any issues.

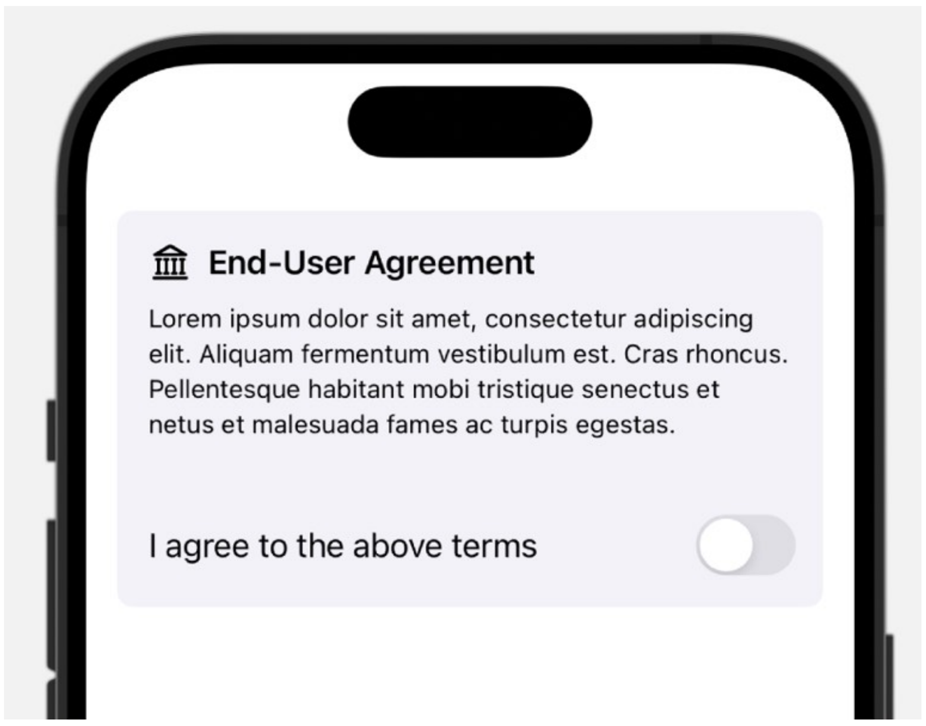
To summarize, groups in SwiftUI do not handle the layout of their children. Instead, they rely on the container view they are placed in to handle the layout. However, groups provide a convenient way to apply the same view modifier to all their children individually, making your code more efficient and avoiding unnecessary repetition.

2.4 GROUPBOX

In SwiftUI, there are various container views available for organizing and styling your interface. One such container is the GroupBox. Unlike the basic Group view, which doesn't apply much styling, the GroupBox allows you to create **card-like layouts** with ease.

To use a GroupBox, you simply define a title or label and the content you want to display. You can also apply some padding to enhance the styling. The label is typically displayed in a headline font style, giving it a prominent appearance.

```
struct GroupBoxExampleView: View {
    @State private var userAgreed: Bool = false
    let agreementText: String = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Aliquam fermentum vestibulum est. Cras rhoncus. Pellentesque habitant mobi
tristique senectus et netus et malesuada fames ac turpis egestas."

    var body: some View {
        GroupBox(label: Label("End-User Agreement",
                              systemImage: "building.columns"),
                 content: {
            Text(agreementText)
                .font(.footnote)

            Toggle(isOn: $userAgreed) {
                Text("I agree to the above terms")
            }
        })
        .groupBoxStyle(.automatic)
        .padding()
    }
}
```


End-User Agreement

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam fermentum vestibulum est. Cras rhoncus. Pellentesque habitant mobi tristique senectus et netus et malesuada fames ac turpis egestas.

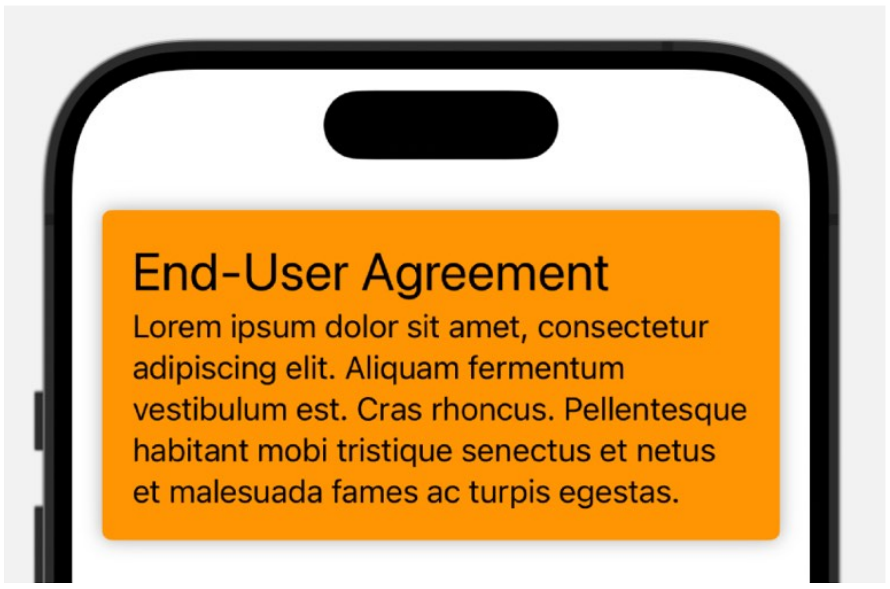
I agree to the above terms

In SwiftUI, many container views have specific styling modifiers tailored to their functionality. You can also create your own custom styling for the GroupBox. By conforming to the **GroupBoxStyle** protocol, you can define a unique appearance and interaction for all GroupBox instances within your view hierarchy.

```
struct OrangeGroupBoxStyle: GroupBoxStyle {
    func makeBody(configuration: Configuration) -> some View {
        VStack(alignment: .leading) {
            configuration.label
                .font(.title)
            configuration.content
        }
        .padding()
        .background(
            RoundedRectangle(cornerRadius: 5.0)
                .fill(Color.orange)
                .shadow(radius: 5)
        )
    }
}
```

In this custom styling example, we can use a VStack to create multiple GroupBox instances with different titles and contents. By applying our orange group box style to these instances, we can see the visual transformation.

```
GroupBox(titleText) {
    Text(agreementText)
}
.groupBoxStyle(OrangeGroupBoxStyle())
```


End-User Agreement

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam fermentum vestibulum est. Cras rhoncus. Pellentesque habitant mobi tristique senectus et netus et malesuada fames ac turpis egestas.

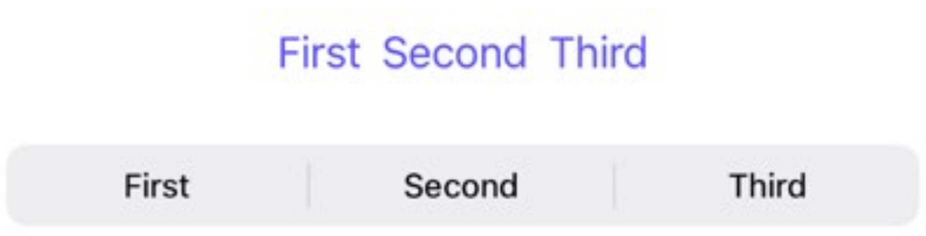
While GroupBox may not be one of the most commonly used views, it can be invaluable in creating visually appealing card-like layouts. Especially in complex apps with numerous subviews, using GroupBox can help break down information and provide a more intuitive user experience.

2.5 CONTROLGROUP

In SwiftUI, control groups are a powerful tool for adding specific styling to control views, such as buttons, that you want to group together. Control groups allow you to create a cohesive and visually appealing layout for your user interface.

To create a control group, you can use the ControlGroup view. Let's compare putting buttons in a HStack vs. a ControlGroup:

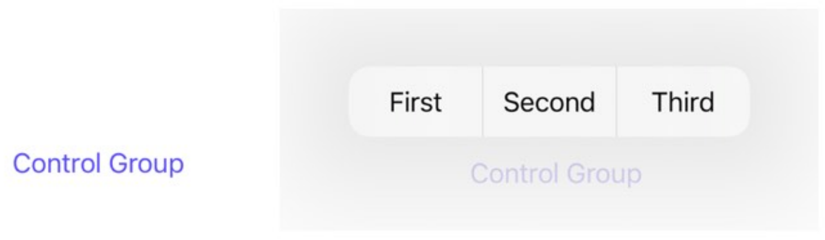
```
HStack {
    Button("First") { }
    Button("Second") { }
    Button("Third") { }
}

ControlGroup("Control Group", systemImage: "grear") {
    Button("First") { }
    Button("Second") { }
    Button("Third") { }
}
```



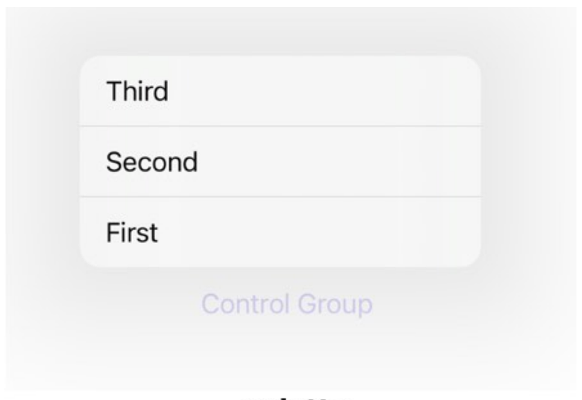
In this example, we have three buttons grouped together within a control group. By wrapping the buttons in a control group, they are visually styled as a cohesive unit. You can still interact with each button individually, but they appear as a single entity.

Control groups also offer various styling variations. For example, you can use system-provided styles like compactMenu to display the control group as a menu:

```
ControlGroup("Control Group", systemImage: "grear") {
    ...
}
.controlGroupStyle(.compactMenu)
```



.compactMenu



.palette

Control groups are particularly useful when you have a set of controls that you want to reuse in multiple places within your app. They automatically adjust their layout based on their placement, making it easy to maintain a consistent design throughout your user interface.

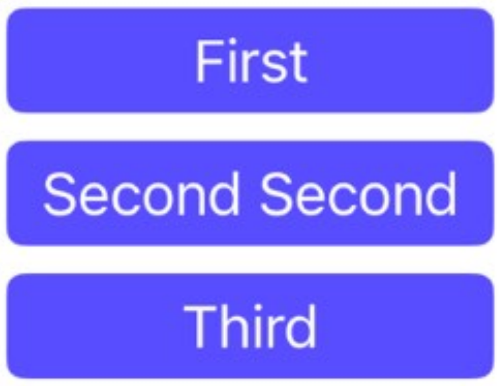
You can also customize the styling of a control group by creating a custom control group style. Here's an example:

```
struct EqualSizControlGroupStyle: ControlGroupStyle {
    func makeBody(configuration: Configuration) -> some View {
        VStack {
            configuration.content
                .foregroundColor(.white)
                .padding(.vertical, 5)
                .padding(.horizontal, 10)
                .frame(maxWidth: .infinity)
                .background(
                    RoundedRectangle(cornerRadius: 5)
                        .fill(Color.accentColor)
                )
        }
        .fixedSize(horizontal: true, vertical: false)
    }
}
```

Here is how you can use your custom Group Styling:

```
ControlGroup("Control Group", systemImage: "grear") {
    Button("First") { }
    Button("Second Second") { }
    Button("Third") { }
}
.controlGroupStyle(EqualSizControlGroupStyle())
```

This group will place all its children in a VStack and add background rectangles that have all the same width:



3. LAYERING VIEWS

In this section, we will explore the concept of layering views in SwiftUI. Layering views allows us to create visually appealing and dynamic user interfaces by combining multiple elements together. I will cover various techniques and modifiers that enable us to control the order and appearance of views within our layouts.

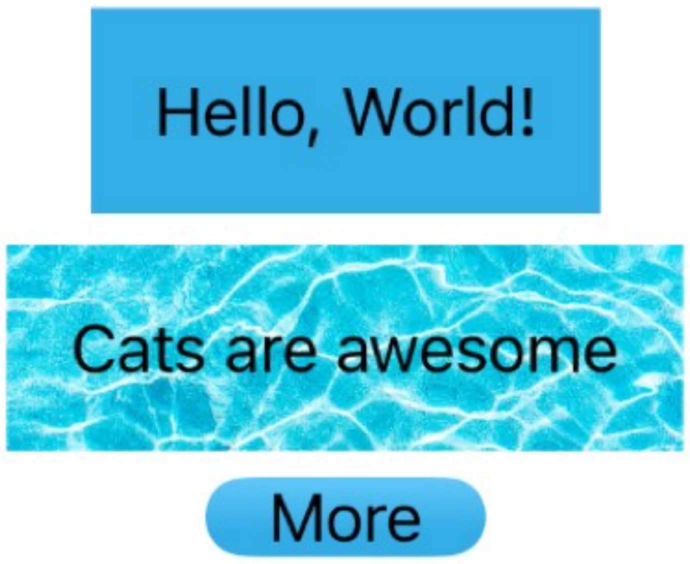
Throughout this section, I will dive into the following key topics:

- **Background Modifier**: Learn how to set a background color or image for your views, providing a solid foundation for your UI elements.
- **Overlay Modifier**: Discover how to overlay additional views on top of existing ones, allowing for creative design choices and visual enhancements.
- **ZStack** container can be used to layer views. I covered this in a previous section
- **ZStack vs Background/Overlay**: Understand the differences between using a ZStack and applying background/overlay modifiers, and when to choose one over the other.
- **Color View and Gradients**: Explore different ways to apply colors to your views, from solid colors to gradients. These views are often used for backgrounds and overlays.
- **Materials**: Discover how to apply materials effects to your view background.

3.1 BACKGROUND MODIFIER

The background modifier in SwiftUI allows to add a background to a view. We can easily add a background to this text by applying the **background** modifier and specifying a view, such as a color, image, or shape for the background:

```
Text("Hello, World!")
    .padding()
    .background {
        Color.cyan
    }

Text("Cats are awesome")
    .padding()
    .background {
        Image("abstract-pool-water")
            .resizable()
            .scaledToFill()
    }
    .clipped()

Text("More")
    .padding(.horizontal)
    .background {
        Capsule().fill(Color.cyan.gradient)
    }
```



The great thing about the background modifier is that it fills out the background of the view it is attached to without increasing the size of the view itself, unlike the ZStack. This makes it perfect for color backgrounds. If you want to **make the view larger, you can use other modifiers like padding or frame**.

In addition to colors, you can also use **images** as backgrounds. By using the resizable modifier, you can resize the image to fit the available space in the background. You can also use the scaleToFill modifier to maintain the aspect ratio of the image while filling the background. **To prevent the image from overflowing, you can use the clipped modifier.**

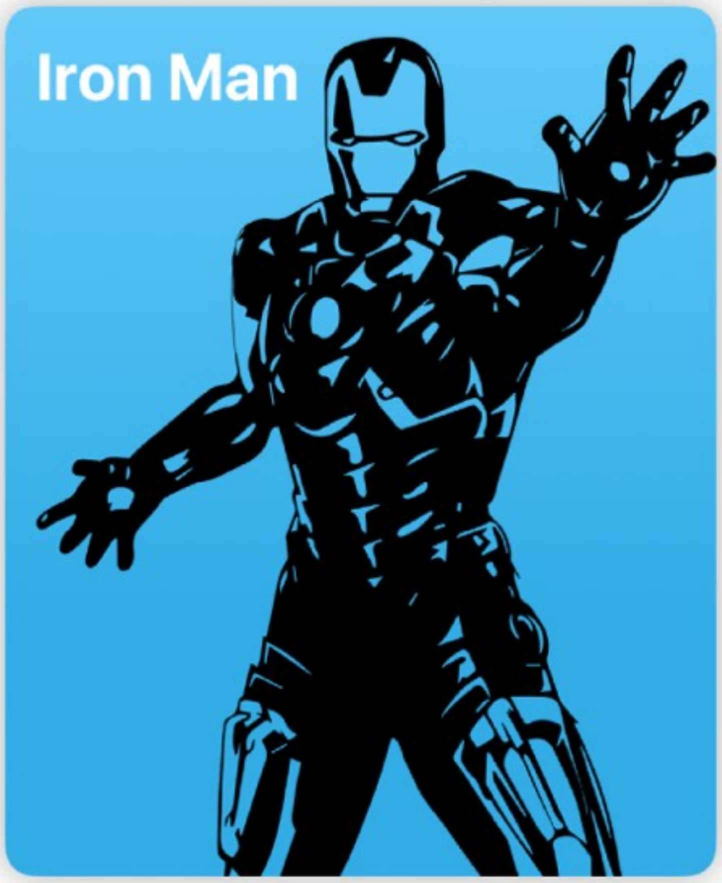
Shapes can also be added as backgrounds. For example, you can add a capsule with a gradient fill using the background modifier.

SuperHero Example Card

Now, let's move on to a more interesting example. Imagine we want to create a superhero card view. We can define a SuperheroView struct conforming to View and add an image of the superhero along with their name. By applying the background modifier, we can add a color or gradient background to the view. To achieve a card-like appearance, we can use the cornerRadius modifier or a rounded rectangle shape.

```
struct SuperHeroView: View {
    let superhero = SuperHero.example2
    var body: some View {
        ResizableImageView(imageName: superhero.imageName)
            .padding([.leading, .top])
            .background(alignment: .topLeading) {
                Text(superhero.name)
                    .font(.largeTitle)
                    .bold()
                    .foregroundStyle(.white)
                    .padding()
            }
            .background(
                RoundedRectangle(cornerRadius: 15)
                    .fill(Color.cyan.gradient)
            )
            .compositingGroup()
```

```
                  .shadow(radius: 10)
                  .padding()
          }
      }
```



The size of the view depends on the image size by default. You can also add multiple background modifiers to further enhance the visual appeal of the card.

It's worth mentioning that when applying the background modifier, **the order of modifiers becomes crucial**. For example, if you want to add text on top of the image, you need to ensure that the text is placed before the background modifier.

Additionally, you can use the **alignment parameter** to control the alignment of the background within the view. I used this parameter to align the superhero tex to the top leading edge.

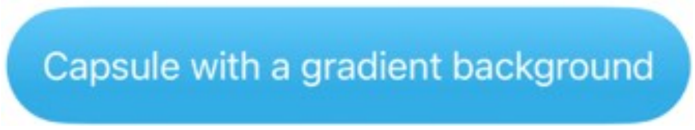
To handle safe areas, the background modifier provides the **ignoresSafeAreaEdges** parameter. This allows you to extend the background into the safe areas.

```
     .background(Color.cyan, ignoresSafeAreaEdges: .top)
```

Background Styles and Shapes

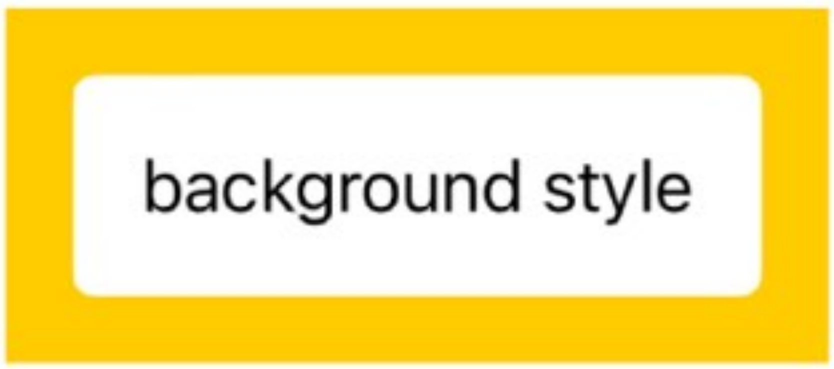
In iOS 15 and macOS 12, Apple introduced new background styles and shapes. You can use these to create more visually appealing backgrounds. There are more convenient ways to place shapes behind a view with the background modifier where you can give a ShapeStyle (e.g. a color or gradient) and a shape:

```
    Text("Capsule with a gradient background")
        .foregroundStyle(.white)
        .padding()
        .background(Color.cyan.gradient, in: Capsule())
```



This can be separated out into two modifiers:

```
    VStack {
        Text("background style")
            .padding()
            .background(in: RoundedRectangle(cornerRadius: 5))
    }
    .padding()
    .background(Color.yellow)
```

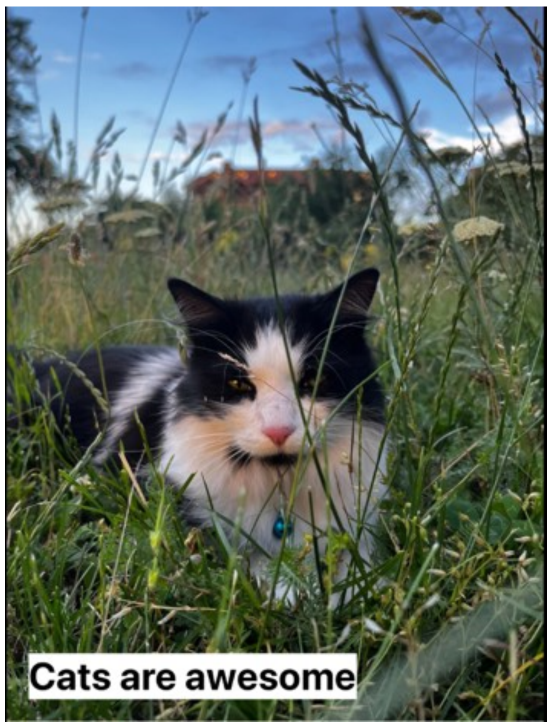


You can set the background style independently for example to change the color of a GroupBox:

```
    GroupBox {
        Text("GroupBox")
    }
    .backgroundStyle(Color.cyan.gradient)
```

3.2 OVERLAY MODIFIER

In SwiftUI, we have the overlay modifier, which allows us to place views on top of another view. Similar to the background modifier that places views behind a certain view, overlay lets us layer views on top.



```
ResizableImageView(imageName: "cat_1")
    .overlay(alignment: .bottomLeading) {
        Text("Cats are awesome")
            .font(.title).bold()
            .background(Color.white)
            .padding()
    }
```

Let's consider another example using a superhero image. Suppose we have a separate SpiderManProfileImageView that displays a profile image of Spider-Man. We can use the overlay modifier to add a white border around this view:

```
struct  SpidermanProfileImage: View {
    var body: some View {
        Image("spiderman_profil")
            .resizable()
            .scaledToFill()
            .frame(width: 200, height: 200)
            .clipShape(Circle())
            .shadow(radius: 5)
            .overlay {
                Circle().stroke(Color.white, lineWidth: 5)
            }
    }
}
```

In this example, we specify the frame size once for both the clipped circle shape and the overlay. They perfectly align with each other.

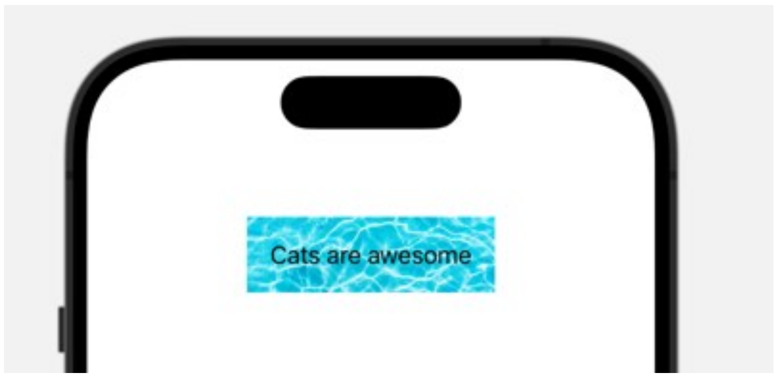
Just like with the background modifier, we have options for styling and alignment. Most of the time, you'll use the overlay modifier with content or alignment. Additionally, you can overlay a whole shape, such as a circle:

```
.overlay(alignment: .bottomLeading) {
    Text("Spider-Man")
}

.overlay(Color.white.opacity(0.5), in: Circle())

.overlay(Color.yellow, ignoresSafeAreaEdges: .top)
```

3.3 ZSTACK VS BACKGROUND/OVERLAY

In SwiftUI, there are different techniques available to achieve layering of views: background, overlay, and stack. Each of these techniques has its own purpose and considerations. Let's explore the differences between them and when to use each approach. To illustrate these concepts, let's use an example from a previous lesson.

```swift
Text("Cats are awesome")
    .padding()
    .background {
        Image("abstract-pool-water")
            .resizable()
            .scaledToFill()
    }
    .clipped()
```

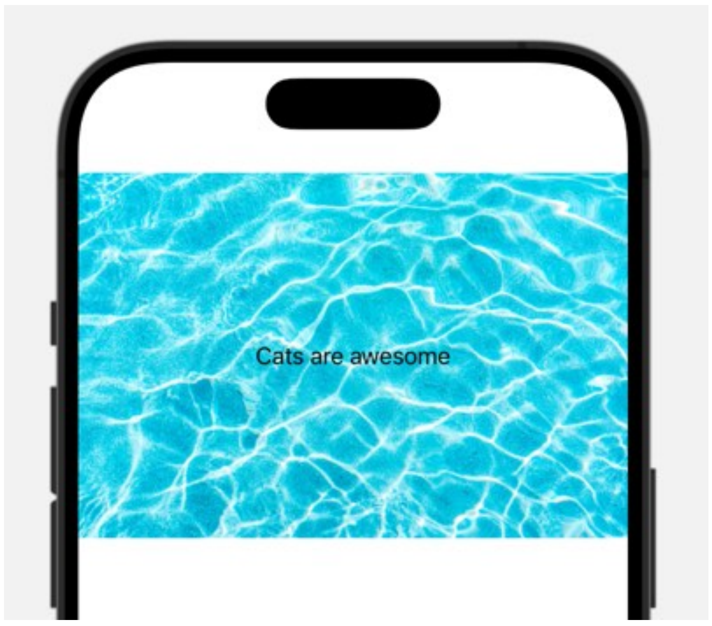


The main view is the text view saying "Cats are awesome". In its background, we have an image view that show a water pool. The size of the view is defined by the text itself and the padding.

The **background, on the other hand, doesn't influence the size of the view** but uses the size of the view it is attached to and passes it down to its child views. As a result, the image only receives the same size as the text.

Now, let's explore how the layout changes when we overlay the text and image **using a ZStack**. We'll need to adjust the image to fit and position the text on top of it.

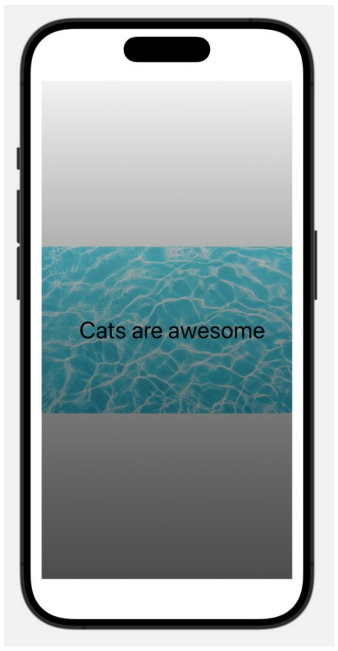
```swift
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()
    Text("Cats are awesome")
        .padding()
}
```



The ZStack considers the size of all its child views. In this case, the image is scaled to fit, making it as large as possible within the horizontal direction. **The ZStack adapts its size to accommodate the largest child view, which, in this case, is the image view.**

When choosing between background, overlay, and ZStack, the decision depends on how you want to control the size of your views. Lets add a color gradient to the above ZStack:

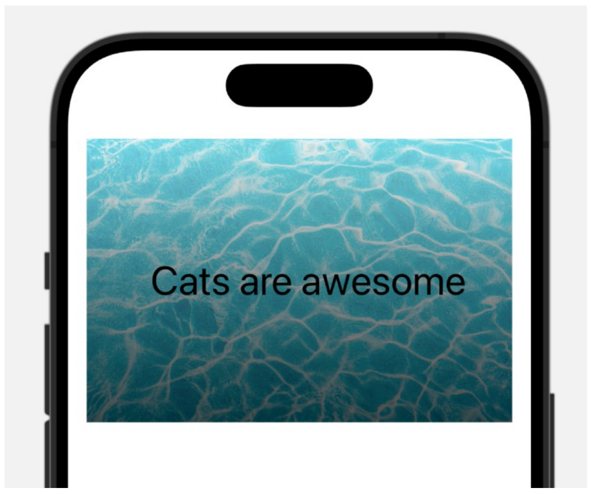
```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()

    LinearGradient(colors: [Color(white: 0.9, opacity: 0.5),
                            Color(white: 0, opacity: 0.7)],
                   startPoint: .top,
                   endPoint: .bottom)

    Text("Cats are awesome")
        .font(.largeTitle)
        .padding(.leading)
}
.padding()
```



However, you may notice that the linear gradient takes up a lot of space and tries to be as big as possible. Consequently, the ZStack adjusts its size to accommodate the largest child view, which, in this case, is the linear gradient.

To restrict the size of the gradient to match the image, using an **overlay modifier is a better solution**. By moving the gradient inside the overlay, its size no longer influences the layout, and the image and gradient can have the same size.

```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()
        .overlay {
            LinearGradient(colors: [Color(white: 0.9, opacity: 0.5),
                                    Color(white: 0, opacity: 0.7)],
                           startPoint: .top,
                           endPoint: .bottom)
        }

    Text("Cats are awesome")
        .font(.largeTitle)
        .padding(.leading)
}
.padding()
```



Another solution is to use the fixedSize modifier to the ZStack and restrict the size in the vertical direction. This way the gradient would not expand more than the image size:

```
ZStack {
    Image(…)
    LinearGradient(…)
    Text(…)
}
.fixedSize(horizontal: false, vertical: true)
```

In some cases, using an overlay modifier instead of a ZStack makes more sense, especially when you want to ensure that the layout is determined by a specific view's size. The background and overlay modifiers allow you to align or resize views relative to other views without affecting the overall layout.

To summarize, when layering views in SwiftUI, consider whether you want views to be properly sized and take up the space they need or if they are secondary and should align or resize with other views. Views that need to be properly sized can be placed in the background or overlay, while views that should align or resize with other views can be placed in a ZStack.

3.4 COLOR VIEW

Colors play a crucial role in creating visually appealing and dynamic user interfaces, and SwiftUI makes it incredibly easy to work with them. Let's start by understanding a fundamental concept: **colors are views themselves**. This means that we can treat colors just like any other view and use them within our view hierarchies. For example, we can add a blue color to a VStack as a view, and **it will expand to occupy as much space as possible**. We can also use other colors like cyan and indigo in a similar manner.

To restrict the expansion of colors, we can **apply a frame modifier to set a specific height or width**. For instance, we can limit the height of our color views to 100 by adding a frame modifier with a height of 100.

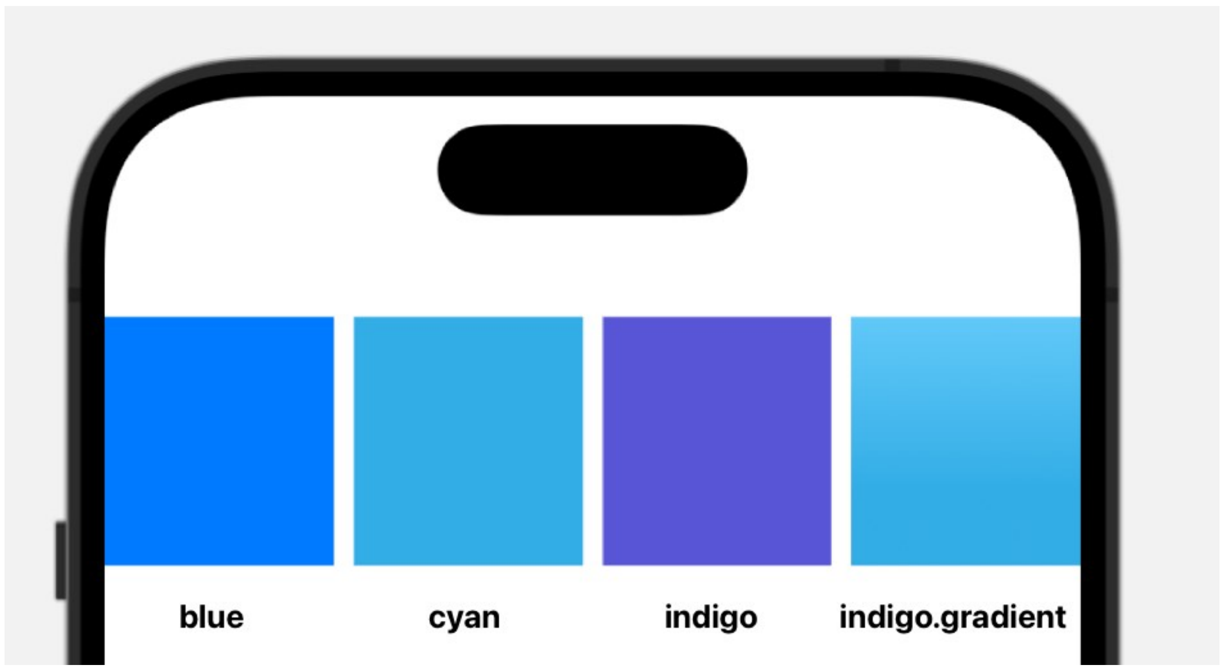
```
Color.cyan
    .frame(height: 100)
```

It's worth noting that colors conform to the ShapeStyle protocol, allowing us to use them as backgrounds with the background modifier. However, if we use a color gradient like

```
Color.cyan.gradient
```

it becomes an AnyGradient and is not considered a view. If you want to use a gradient as a view, use it to fill a shape like:

```
Rectangle().fill(Color.cyan.gradient)
```



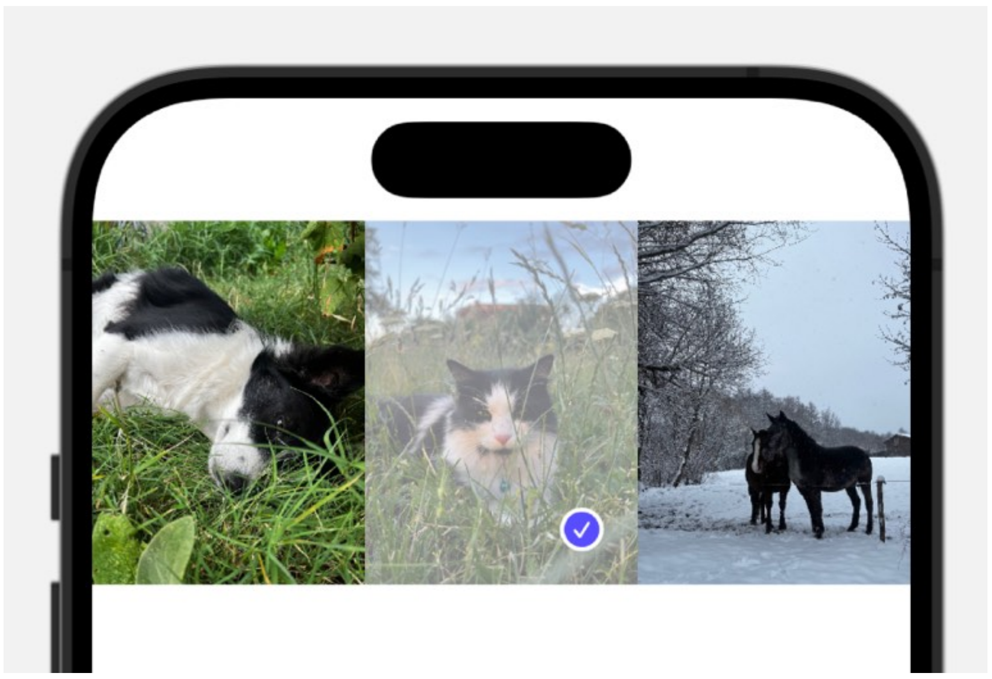
Another use case for colors is filling shapes with a filled shape style. For example, we can use a gradient as a shape style within a background to create visually appealing effects.

In addition to using predefined colors, SwiftUI also provides various options for creating custom colors. We can generate colors using hue, saturation, brightness, red, green, and blue values, or even create semi-transparent colors with a specific opacity. These custom colors offer great flexibility in designing our interfaces.

```
Color(hue: 0.7, saturation: 1, brightness: 1) // bright blue
Color(hue: 1, saturation: 1, brightness: 0.9, opacity: 0.5) // opace pink
Color(red: 1, green: 0, blue: 0) // red
Color(white: 0, opacity: 0.5) // opace gray
```

Example: Image Selection Screen

To demonstrate the usage of colors, let's consider an example. Imagine we have a board displaying multiple images, and we want to highlight the selected images by adding a semi-transparent overlay.



We can achieve this by creating a reusable subview, which takes an image name and a boolean value indicating whether it is selected or not. By leveraging the overlay modifier, we can apply the semi-transparent color to the selected images.

```
struct ImageSelectionView: View {
    let imageName: String
    let isSelected: Bool
    var body: some View {
        ResizableImageView(imageName: imageName)
            .overlay(alignment: .bottomTrailing) {
                if isSelected {
                    ZStack(alignment: .bottomTrailing) {
                        Color(white: 1, opacity: 0.5)

                        Image(systemName: "checkmark.circle.fill")
                            .foregroundColor(.accentColor)
                            .padding(1)
                            .background(Color.white, in: Circle())
                            .padding()
                    }
                }
            }
    }
}
```

To further enhance the selected images, we can add a checkmark icon to indicate their selection status. By using the overlay modifier with a **bottomTrailing** alignment, we can position the checkmark icon precisely where we want it. Additionally, we can add a circle shape behind the image to create a visual distinction.

```
HStack(spacing: 0) {
    ImageSelectionView(imageName: "dog_1",
                       isSelected: false)
    ImageSelectionView(imageName: "cat_1",
                       isSelected: true)
    ImageSelectionView(imageName: "horse_1",
                       isSelected: false)
}
```

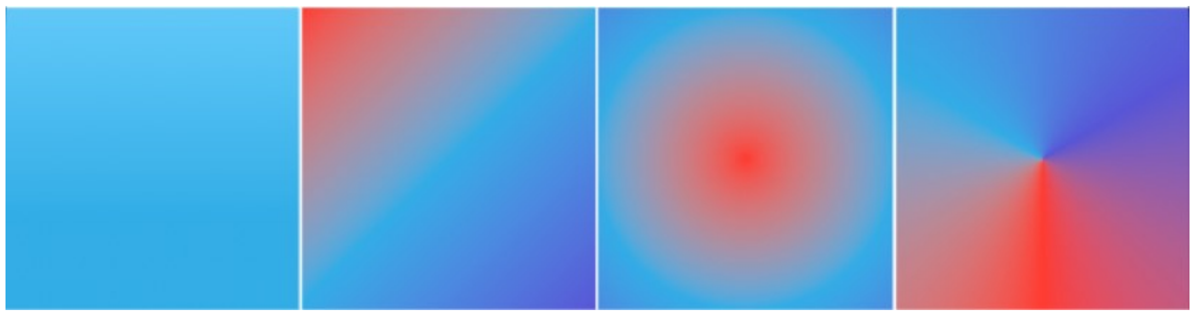
By combining overlays, backgrounds, and alignments, we can create visually appealing and interactive user interfaces. Experimenting with different variations of these techniques will allow you to fine-tune the appearance of your views.

Remember, colors can significantly impact the size and layout of your views. Using overlays and backgrounds judiciously will help maintain the desired visual balance without compromising the overall design.

3.5 GRADIENTS

Gradients like colors are ´greedy´ views and will expand to the available space. SwiftUI provides different types of gradients such as linear, radial, angular, and elliptical gradients. Here are some examples:

```
    HStack(spacing: 1) {
        Rectangle().fill(Color.cyan.gradient)

        LinearGradient(colors: [Color.red, Color.cyan, Color.indigo],
                       startPoint: .topLeading,
                       endPoint: .bottomTrailing)

        RadialGradient(colors: [Color.red, Color.cyan, Color.indigo],
                       center: .center,
                       startRadius: 0,
                       endRadius: 100)

        AngularGradient(colors: [Color.red, Color.cyan, Color.indigo, Color.red],
                        center: .center, angle: .degrees(90))
    }
    .frame(height: 100)
```



Example: Making Text more Readable

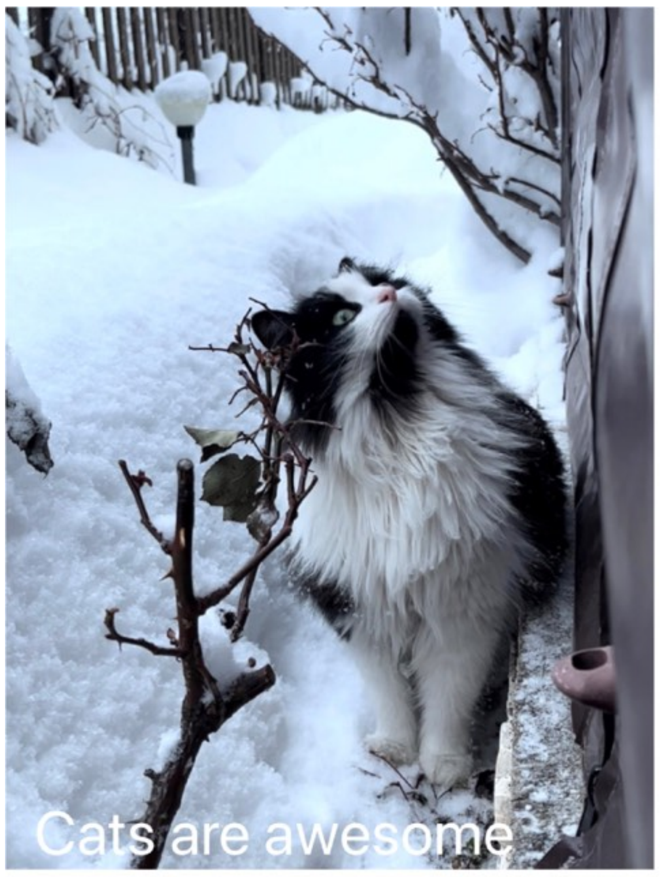
A common use case is to add text on top of an image. Oftentimes this makes the text very difficult to read.

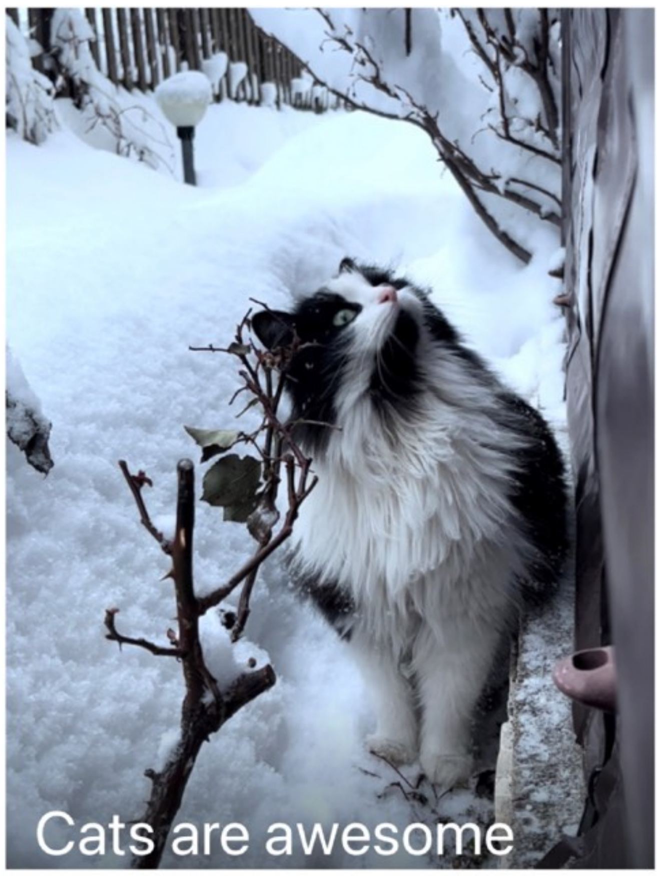
```
    ZStack(alignment: .bottomLeading) {
        Image("cat_4")
            .resizable()
            .scaledToFit()

        Text("Cats are awesome")
            .font(.largeTitle)
            .foregroundStyle(.white)
            .padding(.leading)
    }
```

You can use a gradient that is placed behind the text to increase the contrast. In the below example, I restricted the size of the gradient to a height of 100 points:

```
    ZStack(alignment: .bottomLeading) {
        Image(…)

        LinearGradient(colors: [Color(white: 0, opacity: 0),
                                Color(white: 0, opacity: 0.5)],
                      startPoint: .top,
                      endPoint: .bottom)
            .frame(maxHeight: 100)

        Text(…)
    }
```

This is especially useful if you have high-contrast images like the below winter image:





with opace black gradient

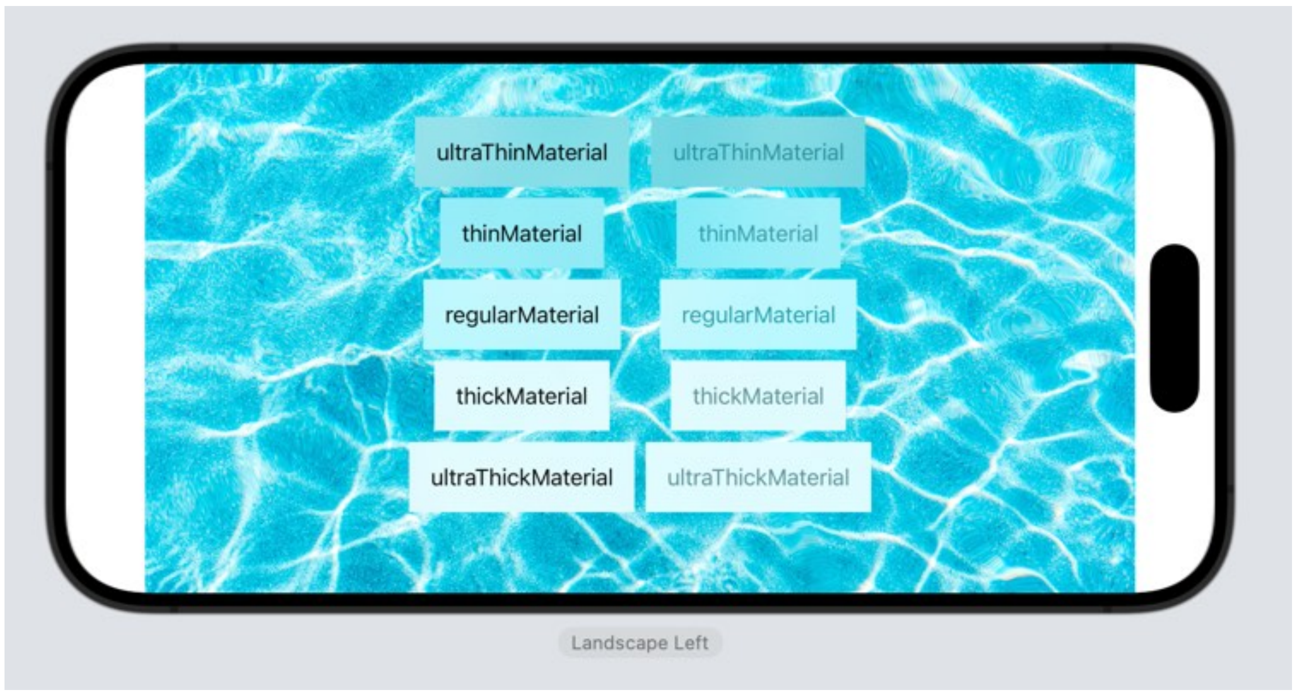
3.6 MATERIALS

Materials in SwiftU can be used to enhance the legibility, readability, and contrast of your views. Materials provide an alternative approach to using colored backgrounds, allowing you to achieve a similar effect with ease.

To better visualize the effect of materials, let's use a ZStack as our example and add an image with high contrast or distinct features, such as a water pool. I am placing text above with a ultra-thin material:

```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFill()

    Text("ultraThinMaterial")
        .padding()
        .background(.ultraThinMaterial)
}
```

By using materials, you can achieve an iced glass effect where the details of the background shimmer through. We can then apply different materials to the background to see their impact. There are five materials available: thin, regular, ultra thin, thick and ultra thick.

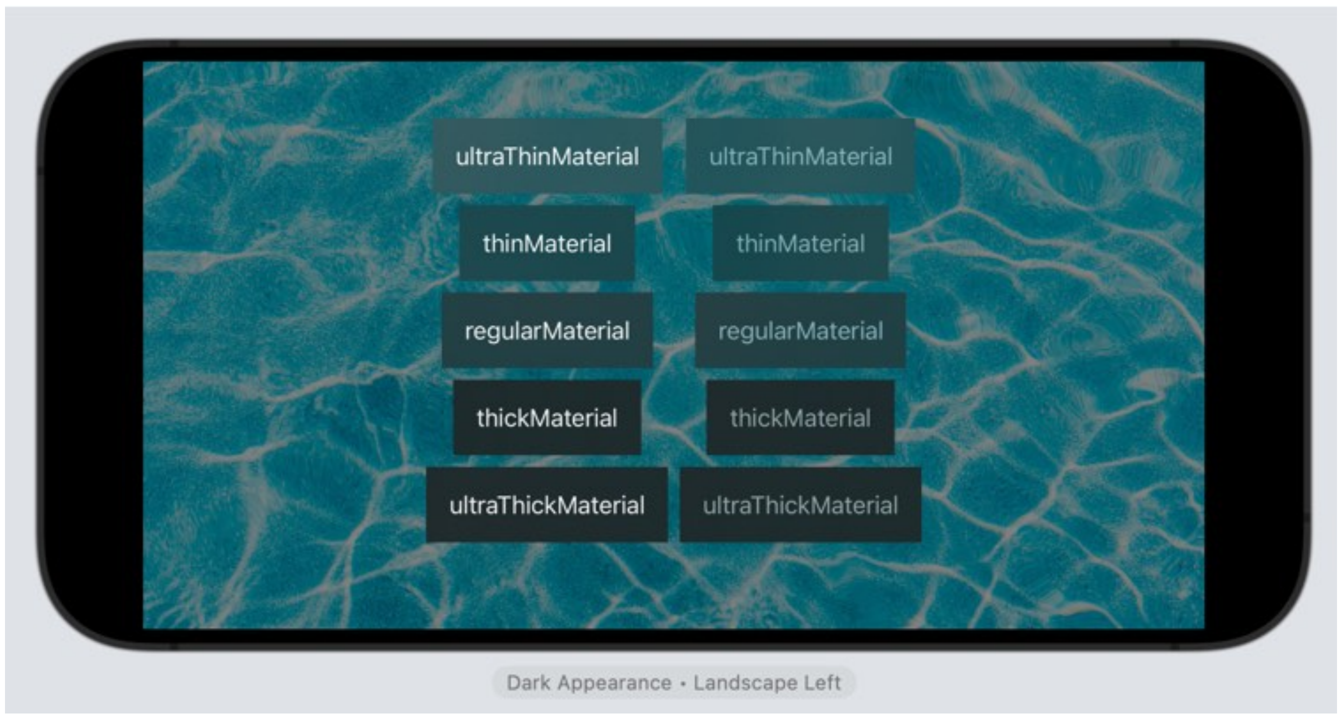


From top to bottom, the materials range from the most transparent to the thickest. You can think of them as different glass sheets with varying degrees of opacity. The choice of material depends on the desired contrast and the foreground colors of your text. For example, a thicker material may provide better contrast for certain text colors.

It's worth noting that materials also work seamlessly in dark mode. When switching to dark mode, the materials adapt automatically, providing a darker appearance. This can be particularly useful for maintaining legibility across different color schemes. You can even use different images or add darker sheets or gradients to achieve the desired effect in dark mode.

```swift
struct MaterialExampleView: View {
    @Environment(\.colorScheme) var colorScheme
    var body: some View {
        ZStack {
            Image("abstract-pool-water")
                .resizable()
                .scaledToFill()

            if colorScheme == .dark {
                Color.black.opacity(0.5)
            }

            HStack {
                Text(…)

                Text(…)
                    .foregroundStyle(.secondary)
            }
        }
    }
}
```

Remember, when presenting information, always ensure that the contrast between the text and the background is sufficient for users to read comfortably. This consideration is especially important for individuals with visual impairments. Materials, along with other techniques like gradients and semi-transparent colors, can greatly enhance the readability of your views.


ultraThinMaterial
ultraThinMaterial
thinMaterial
thinMaterial
regularMaterial
regularMaterial
thickMaterial
thickMaterial
ultraThickMaterial
ultraThickMaterial
Dark Appearance • Landscape Left

4. POSITIONING VIEWS

4.1 HOW TO POSITION VIEWS

4.2 ALIGNMENT GUIDES

4.3 CUSTOM ALIGNMENT GUIDES

4.4 GRID VIEW

4.5 POSITION AND OFFSET MODIFIERS

GET THE FULL BOOK

5. SIZING VIEWS

5.1 HOW THE LAYOUT SYSTEM SIZES AND POSITIONS VIEWS

5.2 FIXED AND FLEXIBLE FRAMES

5.3 FIXEDSIZE

5.4 LAYOUT PRIORITY

5.5 SIZING TEXT VIEWS

5.6 SIZING IMAGES

5.7 UPSCALING IMAGES AND BITMAP VS VECTOR GRAPHICS

5.8 SIZING SYSTEM ICONS

5.9 ASYNCIMAGE

5.10 ASPECT RATIO

5.11 SCALE EFFECT

5.12 CONTENT EDGES: SAFE AREA, PADDING AND MARGINS

5.13 CONTAINER RELATIVE FRAME

5.14 CORNERRADIUS, CLIP AND MASK

6. REUSABLE LAYOUT COMPONENTS

6.1 REUSABLE VIEW MODIFIERS

6.2 CUSTOM CONTAINER VIEWS

GET THE FULL BOOK

7. ADAPTIVE LAYOUT

7.1 ENVIRONMENT VALUES

7.2 VIEWTHATFITS

7.3 CONDITIONAL LAYOUT

8. DYNAMIC DATA

8.1 FOREACH

8.2 LAZY LOADING IN 1-DIMENSION: LAZYVSTACK AND LAZYHSTACK

8.3 LAZY LOADING IN 2-DIMENSION: LAZYVGRID AND LAZYHGRID

8.4 IMAGE GALLERY

9. SCROLLVIEW

9.1 SCROLL DIRECTION

9.2 SCROLL CONTENT SIZE

9.3 SCROLL BEHAVIOUR

9.4 SCROLL OFFSET AND PROGRAMMATIC SCROLLING

9.5 SCROLLVIEW ANIMATIONS

10. SPECIAL SYSTEM CONTAINERS

10.2 LIST

10.3 TABLEVIEW

10.4 FORM

10.5 SECTION AND SUBVIEWS

Thank you for reading this free book!

You can get the full book and video course at swiftyplace.com which includes all project files you see in this book.

Get the book with 50% OFF