SwiftUI Course

Flavio Copes

Updated August 3, 2026

Contents

Learn SwiftUI views, modifiers, layout, state, actions, lists, forms, input controls, and modern data-driven navigation.

What you'll learn: Build a small adaptive SwiftUI interface whose layout, state, collections, forms, and navigation follow one clear data model.

SwiftUI foundations

Build declarative interfaces from views, modifiers, stacks, and images.

Introduction to SwiftUI

Learn how SwiftUI uses declarative views, state, modifiers, scenes, and previews to build interfaces across Apple platforms.

SwiftUI is Apple's framework for building user interfaces with Swift.

You can use it across iOS, iPadOS, macOS, watchOS, tvOS, and visionOS. Many views adapt their appearance and behavior to the platform where they run.

SwiftUI uses a **declarative** approach. You describe what the interface should display for the current data, and SwiftUI updates the affected views when that data changes.

This is different from manually creating every view and telling it how to update after each event.

Does SwiftUI replace UIKit and AppKit?

No.

SwiftUI is the recommended starting point for many new interfaces, but UIKit, AppKit, and WatchKit are still supported frameworks.

You can put a SwiftUI view inside an existing UIKit or AppKit app. You can also wrap a UIKit view or view controller and use it inside SwiftUI.

This lets you adopt SwiftUI one screen at a time. It also helps when SwiftUI does not provide a feature your app needs.

The structure of a SwiftUI app

A SwiftUI app starts with a type that conforms to the `App` protocol:

```
import SwiftUI

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

The `@main` attribute marks the app's entry point.

The `body` property returns a scene. A scene represents part of the interface with a lifecycle managed by the system.

`WindowGroup` is the usual main scene for an app. It contains the root view, which is `ContentView` in this example.

Creating a view

You create a custom view by defining a structure that conforms to the `View` protocol:

```
import SwiftUI

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

The `View` protocol requires a `body` computed property. The value returned by `body` describes that part of the interface.

Views can contain other views. Use a stack to arrange several views:

```
struct ContentView: View {
  var body: some View {
    VStack {
      Text("Hello, world!")
      Text("Welcome to SwiftUI")
    }
  }
}
```

`VStack` arranges its child views vertically. SwiftUI also provides `HStack`, `ZStack`, lists, grids, forms, and many other containers.

What does some View mean?

You will see `some View` in almost every SwiftUI file.

`some` creates an **opaque result type**. It tells callers that `body` returns a value conforming to `View` without exposing the long concrete type produced by the view hierarchy.

The compiler still knows the exact underlying type and checks it at compile time. SwiftUI's result builder also lets you write multiple child views and supported conditionals inside `body`.

You do not normally need to write the concrete type yourself.

Configuring views with modifiers

You customize a view by applying modifiers:

```
Text("Hello, world!")
  .font(.title)
  .bold()
  .padding()
```

Each modifier returns a new view that wraps or configures the previous one.

The order can matter. For example, adding a background before padding gives a different result than adding it after padding.

Connecting a view to state

The interface becomes interesting when it changes in response to data.

Use `@State` for a small value owned by one view:

```
struct CounterView: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Count: \(count)")

      Button("Add one") {
        count += 1
      }
    }
  }
}
```

When the button changes `count`, SwiftUI updates the parts of the interface that read it.

Declare state as private and keep it in the highest view that owns the value. Pass a binding to a child view when that child needs to change the value.

Use an observable model for application data shared by several views. `@State` is best for temporary interface state, not permanent storage.

Views are descriptions

A SwiftUI view structure is a lightweight value describing part of the interface. SwiftUI can create that value and evaluate its `body` many times.

Do not treat a view structure like a long-lived view controller. Keep mutable data in state or a model, and avoid starting network requests or other side effects directly while computing `body`.

Use lifecycle modifiers such as `task`, `onAppear`, and `onChange` when the view needs to perform work.

Previewing a view in Xcode

Xcode can show a live preview next to your Swift code:

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

Previews are useful for checking different data, screen sizes, accessibility settings, and appearance modes without navigating through the whole app.

You still need to run the app on simulators and real devices. A preview does not replace normal testing.

Starting a SwiftUI project

Open Xcode and create a new app project. Select SwiftUI for the interface when Xcode offers the choice.

Xcode creates an `App` type, a `WindowGroup`, and an initial view. Start by changing that view and running the project.

Check the deployment target before using a new SwiftUI API. SwiftUI evolves with each Apple platform release, and an API available in the current SDK might require a newer operating system than your app supports.

The official SwiftUI documentation covers the framework, and Apple's SwiftUI pathway is a good place to continue learning.

SwiftUI: exploring views and modifiers

Learn how views and modifiers work in SwiftUI, how a modifier like font() returns a brand new view, and why the order you apply modifiers matters.

In the introduction to SwiftUI post I mentioned views.

SwiftUI is all about views.

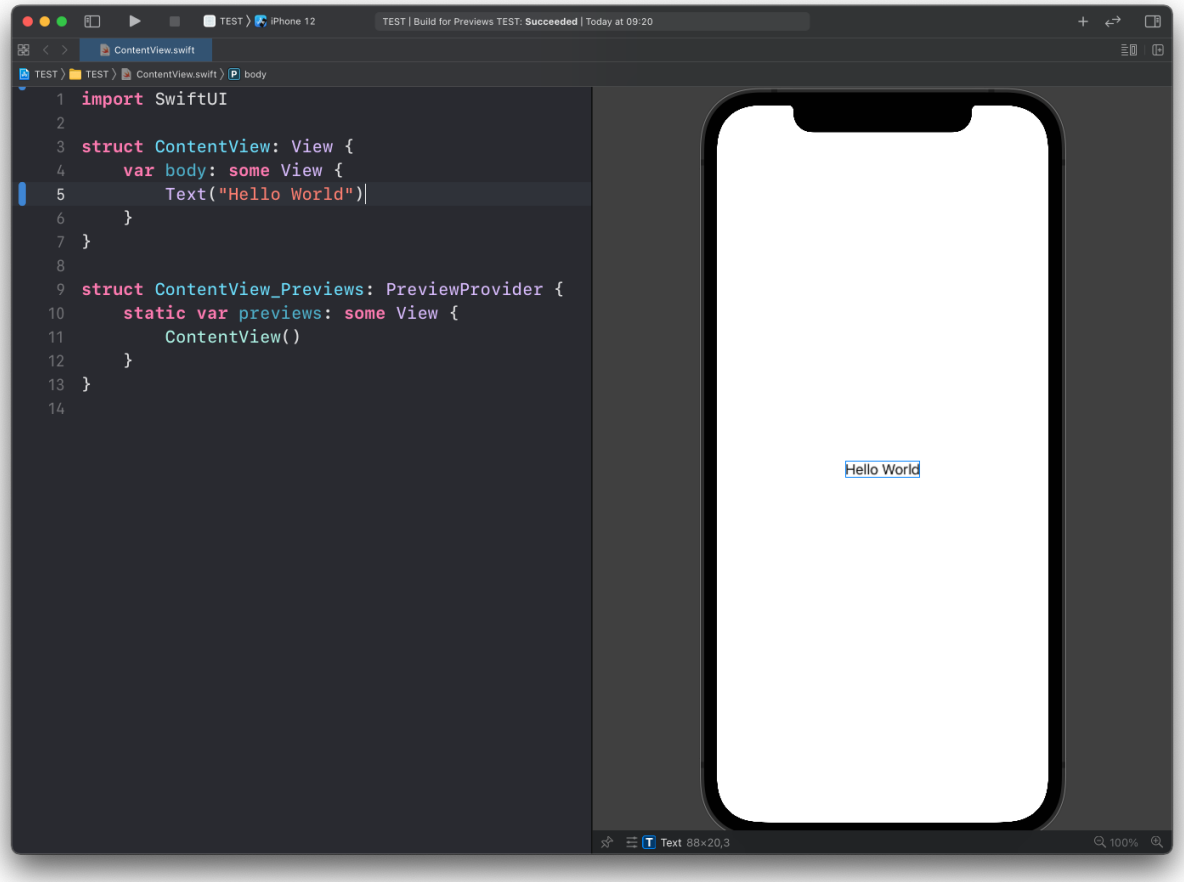
Remember the Hello World app?

```
import SwiftUI

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

`ContentView` is the main view. Its job is to define which views compose our app.

In here, we have a single view, `Text`.

If you run this in Xcode, this is what the app will look like:



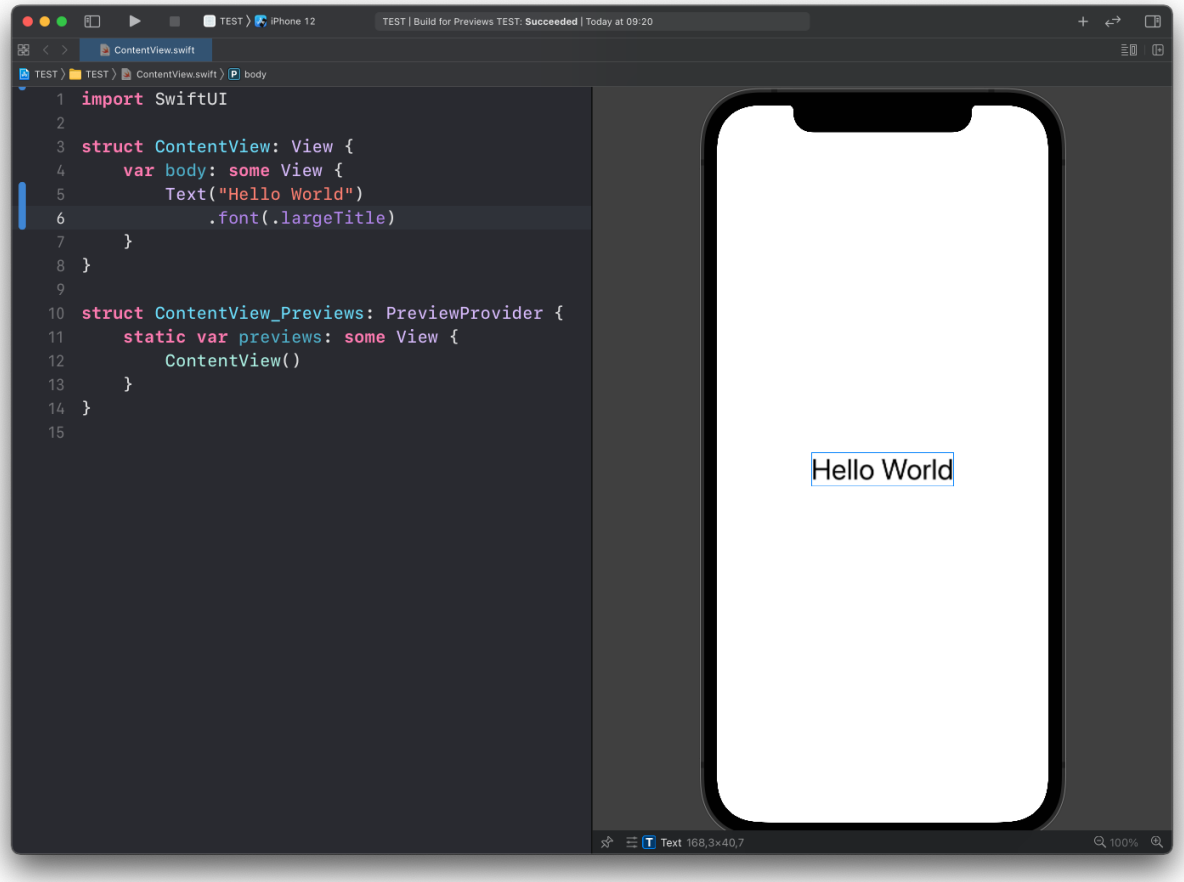
Notice the additional code after the `ContentView` struct: this is how we tell Xcode what to display in the preview panel on the right. It's not part of the app, but it's used in development.

A view can have **modifiers**.

Here's an example of a modifier of the `Text` view, `font()`:

```
struct ContentView: View {
    var body: some View {
        Text("Hello World")
            .font(.largeTitle)
    }
}
```

This modifier takes the `Text` view we created and makes the font larger:



Different views can have different modifiers.

We've just seen the `Text` view so far, and that view has a number of modifiers you can use, including:

- `font()` sets the default font for text in the view
- `background()` sets the view background
- `foregroundColor()` sets the color of the foreground elements displayed by the view
- `padding()` pads the view along all edges

... and many more. In the case of `Text` you can check all the modifiers you can use in this page: https://developer.apple.com/documentation/swiftui/text-view-modifiers.

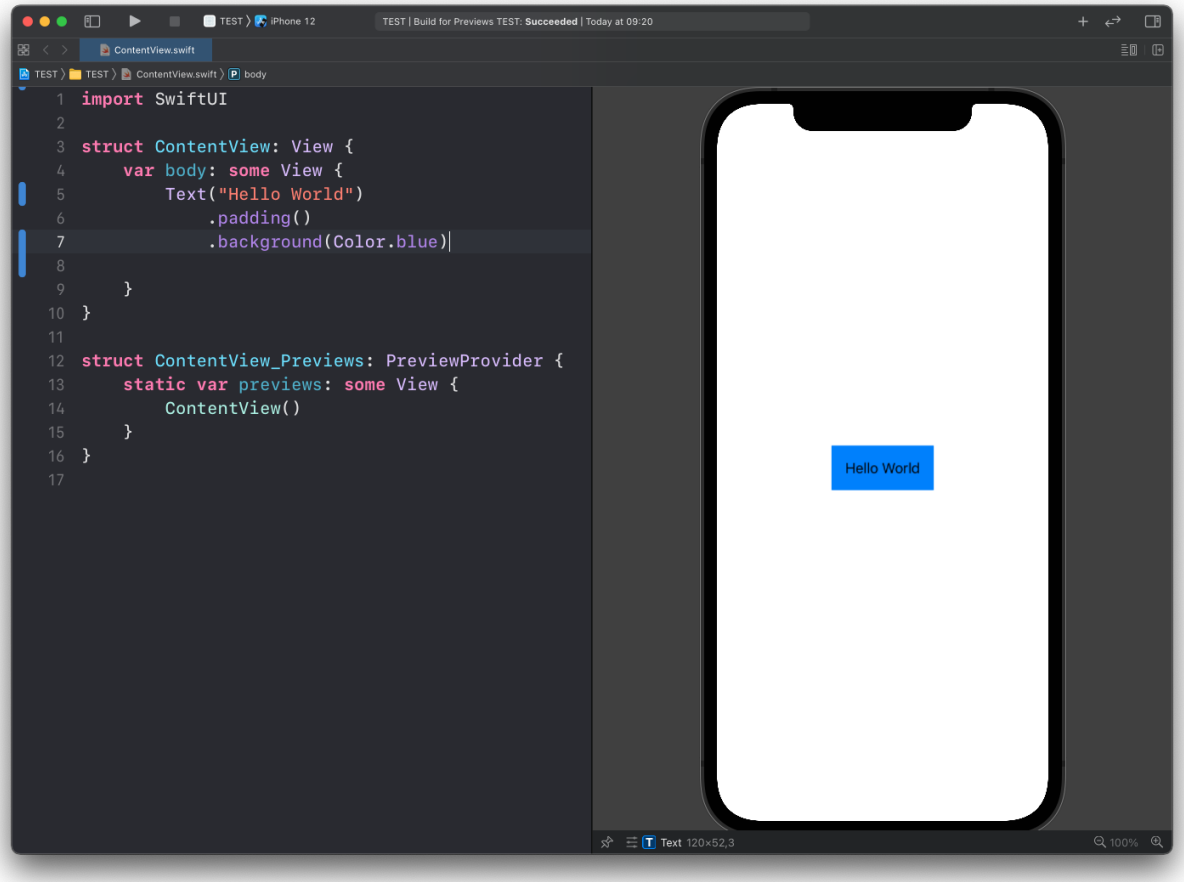
It's important to note that the modifier does not modify the existing view. It actually takes an existing view and creates a new view.

Why is this important? Because this fact causes the order of modifiers to matter.

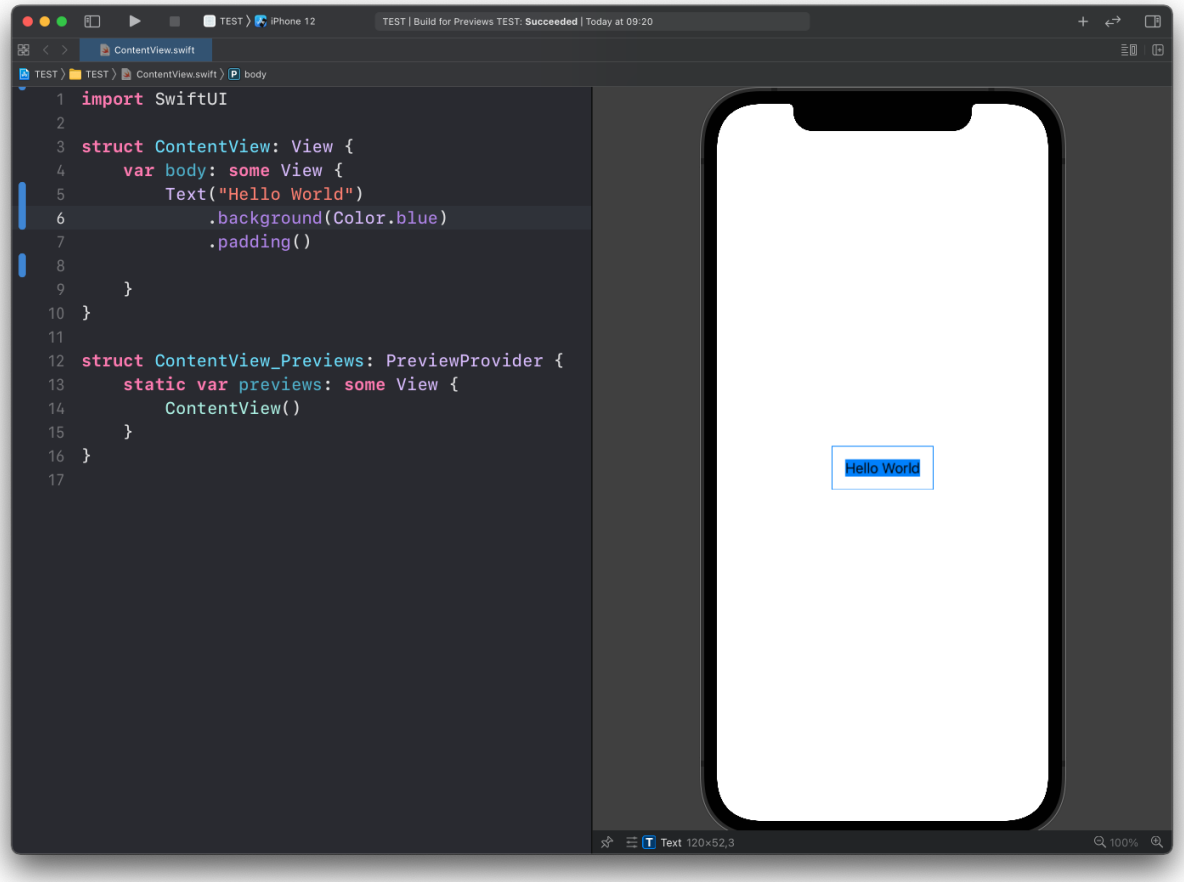
Suppose you want to set the background of the `Text` view, and then add some padding to it.

```
Text("Hello World")
    .padding()
    .background(Color.blue)
```

Here's the result:



but if you invert the 2 modifiers, you get this result:



This is the consequence of modifiers returning a new view once they are applied, and not modifying the existing view.

SwiftUI: stacks and groups

Learn how to lay out multiple views in SwiftUI using HStack, VStack, and ZStack to align them on each axis, plus Group to bundle views together.

No SwiftUI app, beside an Hello World, has just a view.

When you want to add more than one view, you need to add them to a stack.

There are 3 kinds of stacks:

- `HStack` aligns items on the X axis
- `VStack` aligns items on the Y axis
- `ZStack` aligns items on the Z axis

Let's go back to the Hello World app:

```
import SwiftUI

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

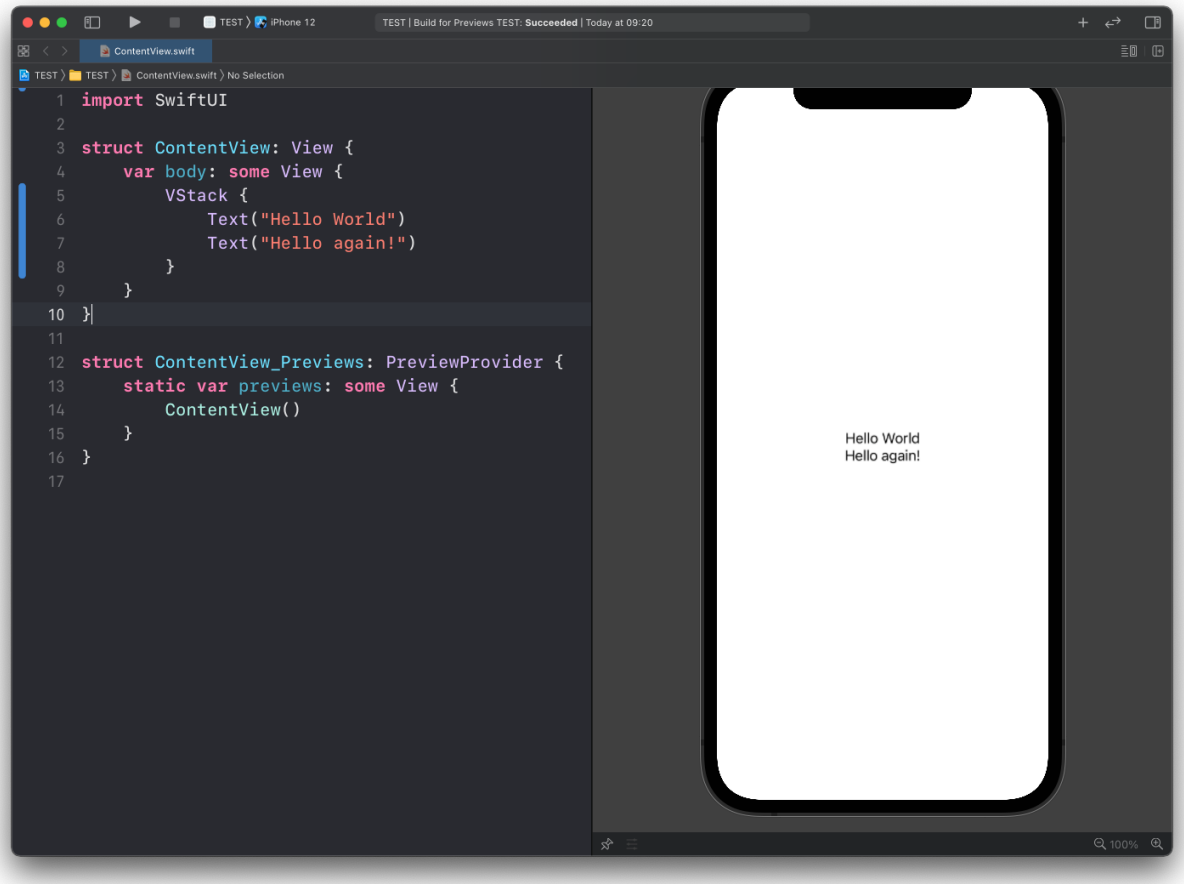
To add a second `Text` view we can't do this:

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

but we have to **embed those views into a stack**.

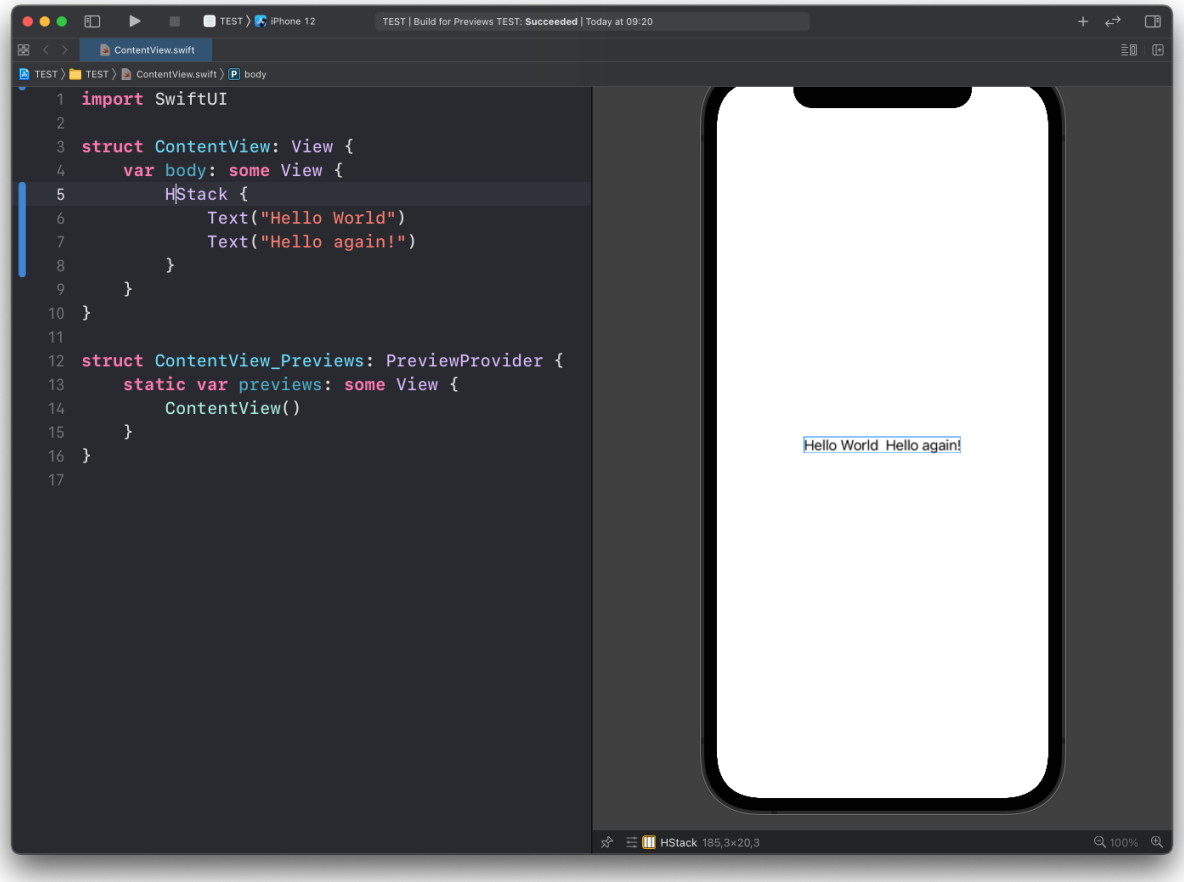
Let's try with `VStack`:

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

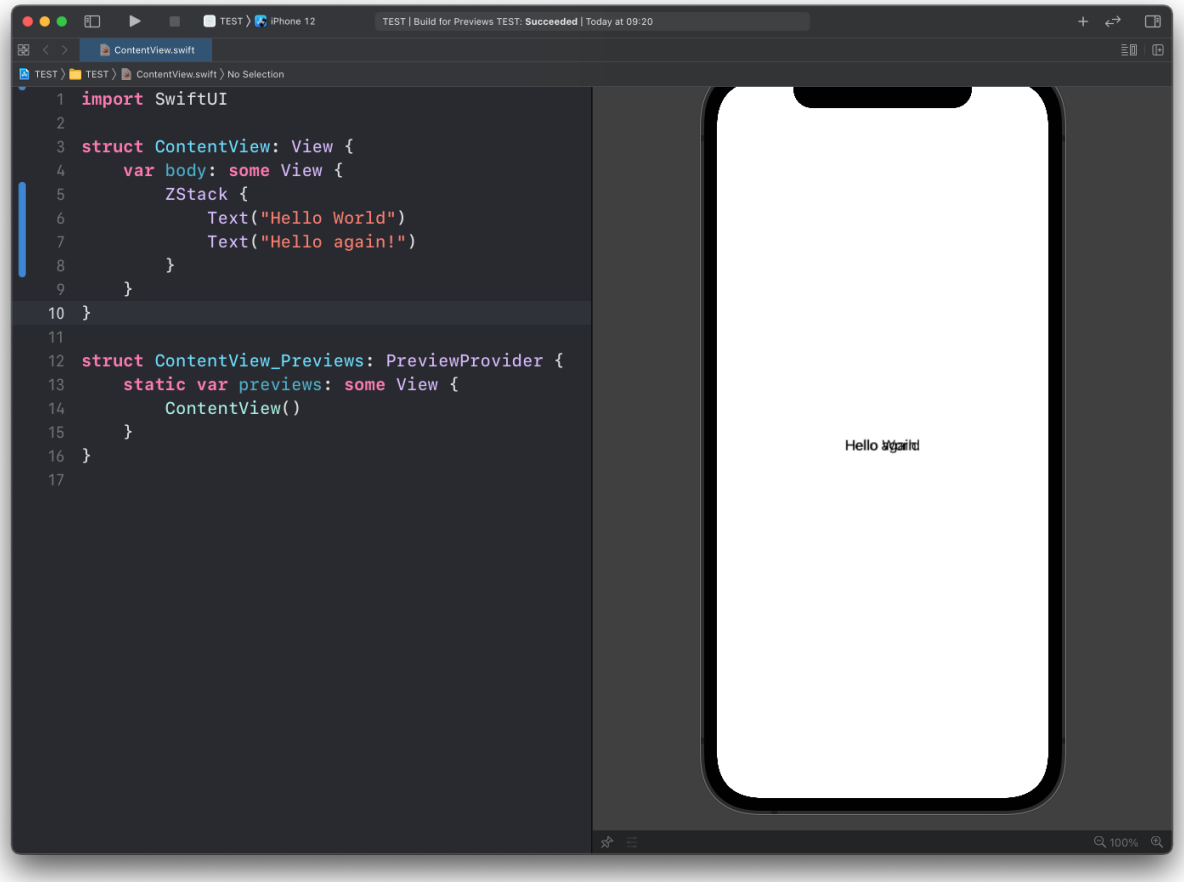


See? The views are aligned vertically, one after the other.

Here's `HStack`:



And here's `ZStack`, which puts items one in front of the other, and in this case generates a mess:



`ZStack` is useful, for example, to put a background image and some text over it. That's the simplest use case you can think of.

In `SwiftUI` we organize all our UI using those 3 stacks.

We also use `Group`, a view that, similarly to stacks, can be used to group together multiple views, but contrary to stack views, it does not affect layout.

```
VStack {
    Group {
        Text("Hello World")
        Text("Hello again!")
    }
}
```

> One use case that might come handy for groups, beside applying modifiers to child views as we'll see next, is that views can only have 10 children. So you can use `Group` to group together up to 10 views into 1

`Group` and the stack views are views too, and so they have modifiers.

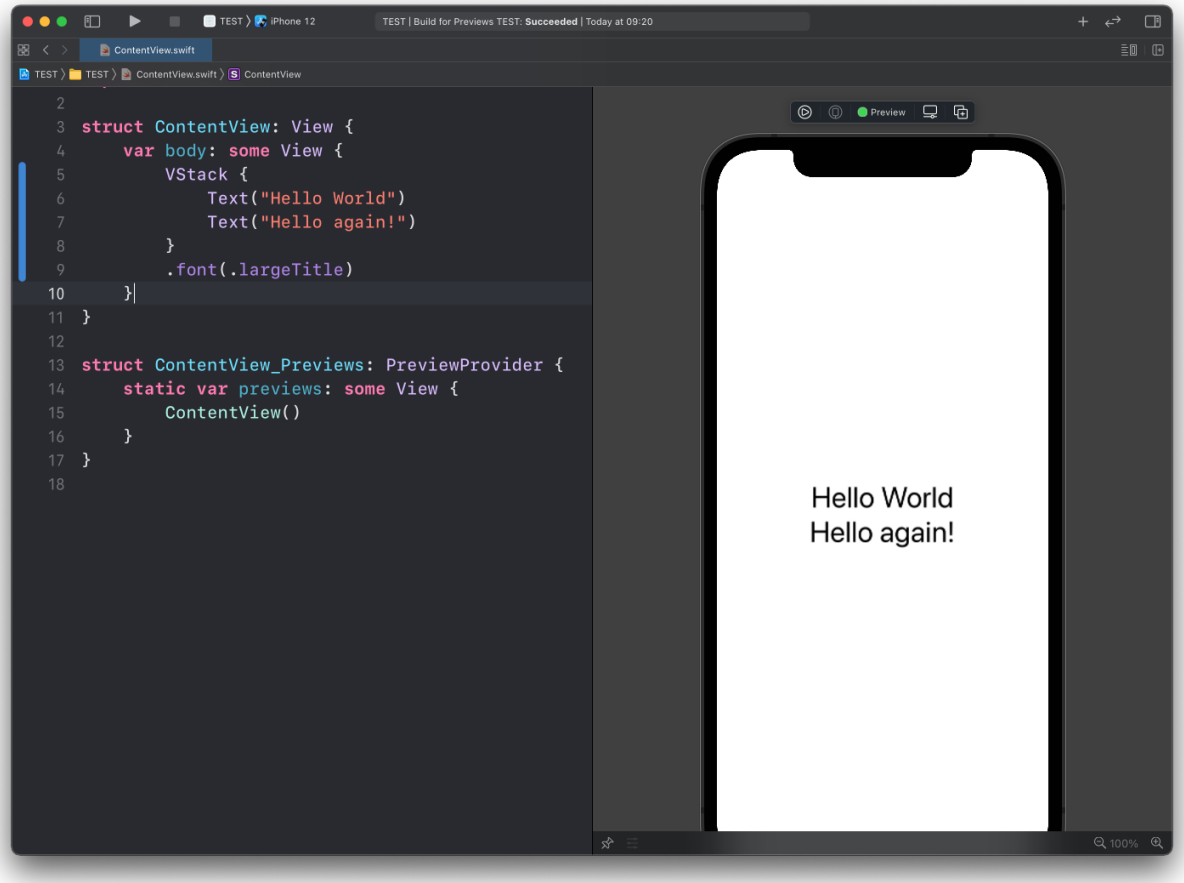
Sometimes modifiers affect the view they are applied to, like in this case:

```
Text("Hello World")
.font(.largeTitle)
```

Sometimes however they are used to apply the same property to multiple views at the same time.

Like this:

```
VStack {
    Text("Hello World")
    Text("Hello again!")
}
.font(.largeTitle)
```



See? By applying the `font()` modifier to the `VStack`, the `.largeTitle` font was applied to both `Text` views.

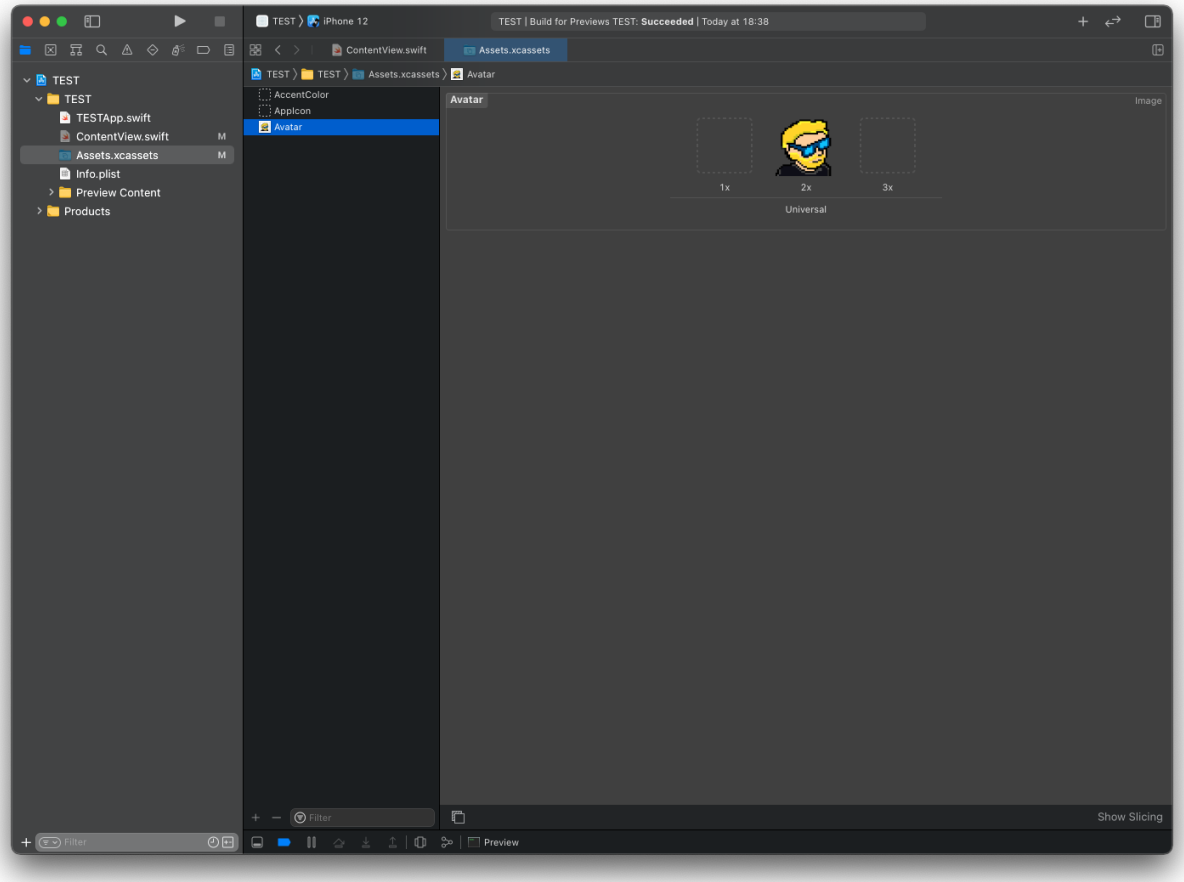
This is valid for modifiers that we call **environment modifiers**. Not every modifier can work this way, but some do, like in the above example.

SwiftUI: images

Learn how to display an image in SwiftUI with the Image view, from loading an asset or a system image to modifiers like resizable, frame, and clipShape.

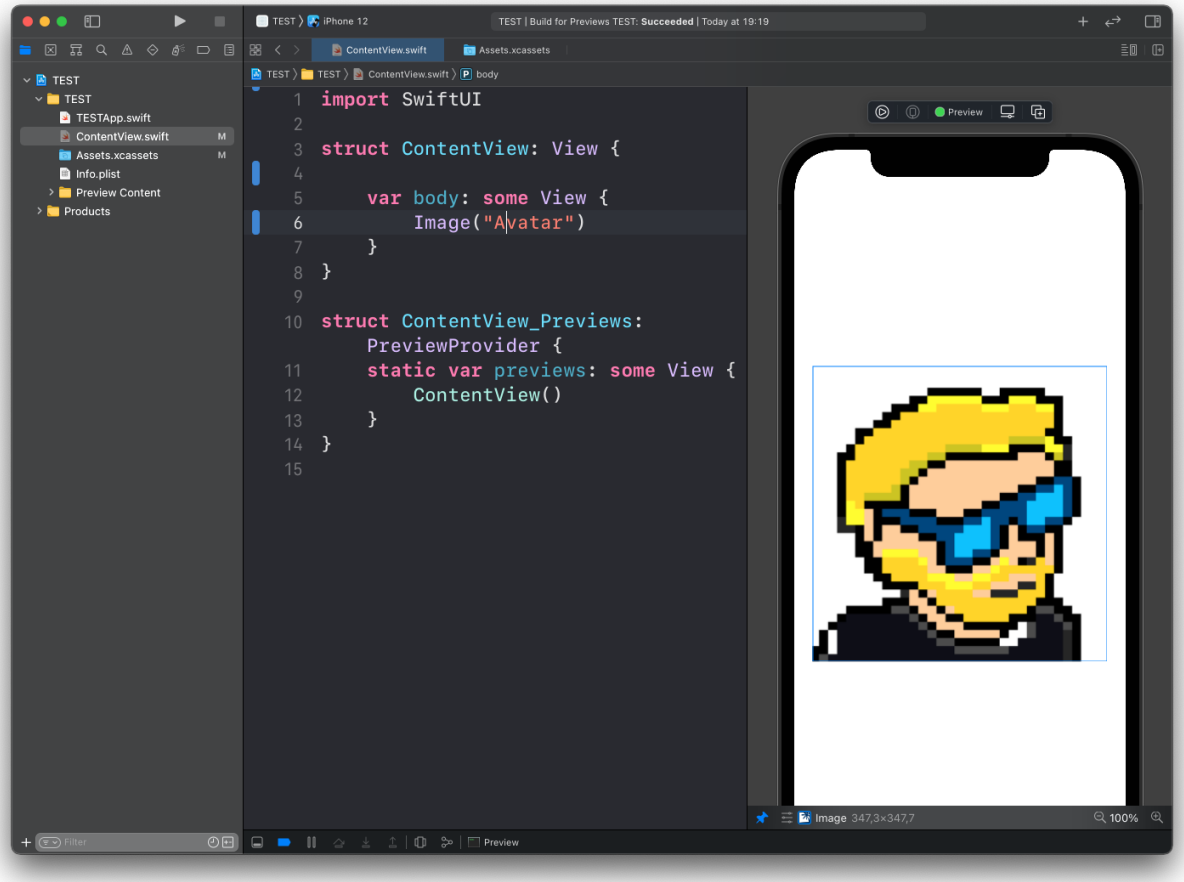
You can display an image in a SwiftUI view by using the `Image` view.

First you need to add the image to a new image set in your `Assets.xcassets` file in the Xcode project navigator.



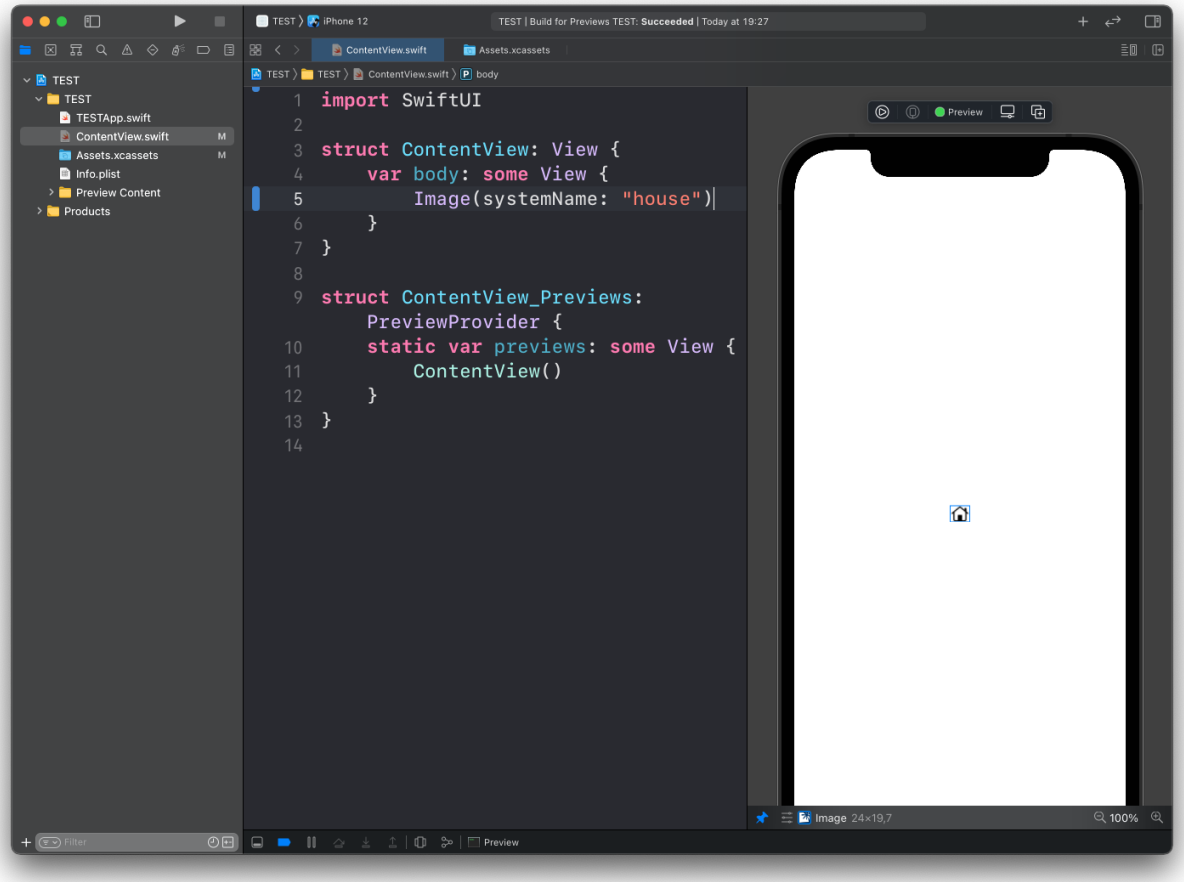
Then you can add the image to your `ContentView` like this:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("Avatar")
    }
}
```



You can also use `Image` to show a system image, using the format `Image(systemName:)`:

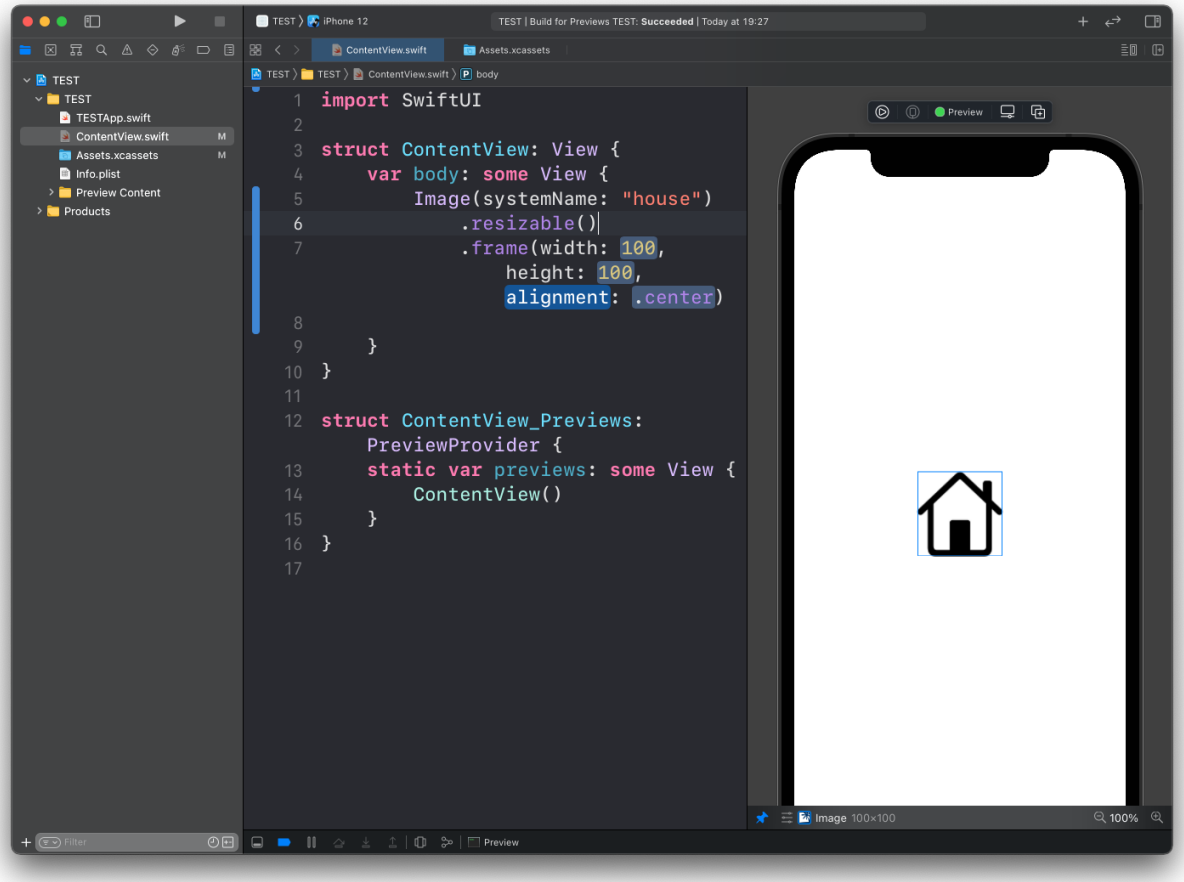
```
struct ContentView: View {
    var body: some View {
        Image(systemName: "house")
    }
}
```



The `Image` view has a set of modifiers you can use, including:

- `.resizable()` to resize the image and adjust to the `.frame()` dimensions
- `.frame()` to set its width/height
- `.clipShape()` to set a clipping shape
- `.border()` to set the color of the border
- `.overlay()` to layer another view in front of it
- `.aspectRatio()` to set the aspect ratio
- `.clipped()` to cut off the image outside of the frame

Example:



Check your understanding: SwiftUI foundations

Check the key ideas from swiftui foundations before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

State and actions

Connect owned state to buttons, conditional content, and alerts.

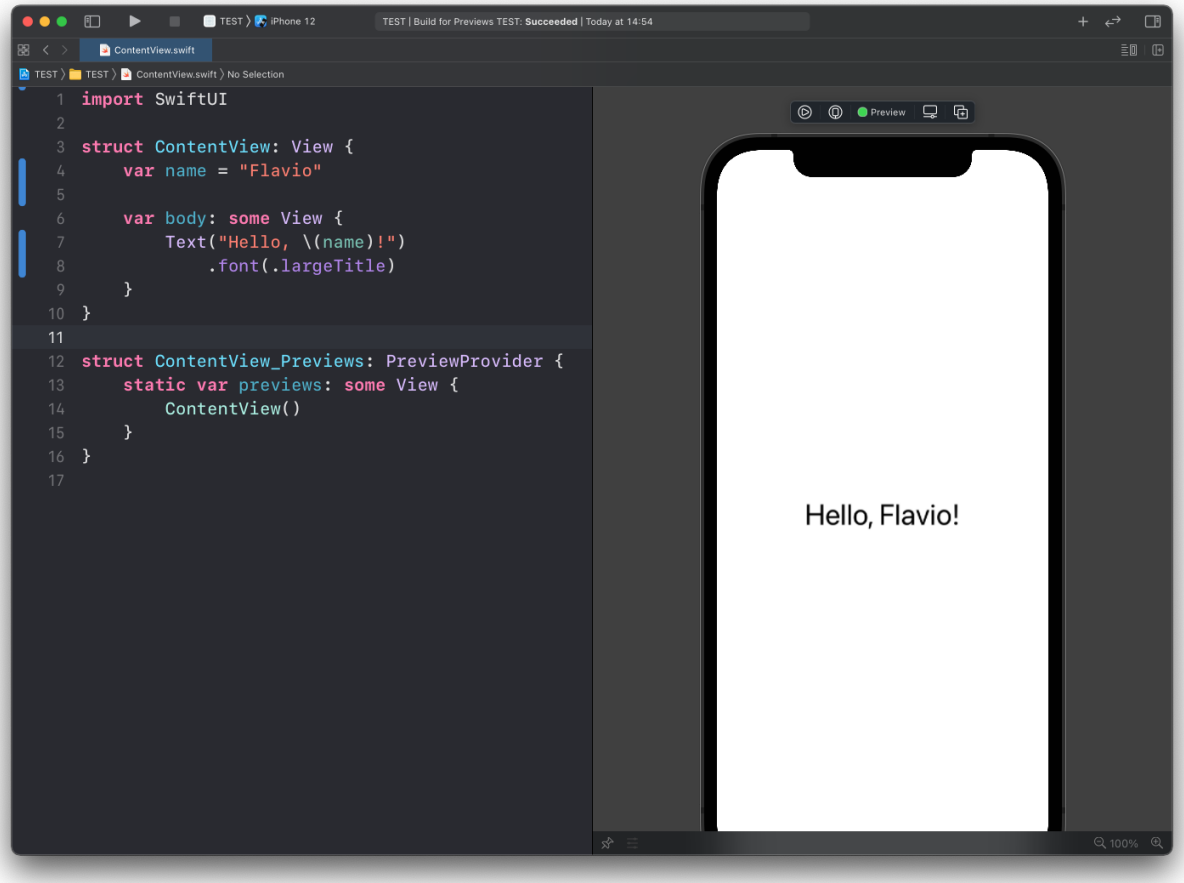
SwiftUI: properties

Learn how properties hold data in SwiftUI views, from simple constants to State and Binding property wrappers that keep the UI in sync with data.

You can add any property to any SwiftUI view, like this:

```
import SwiftUI

struct ContentView: View {
    let name = "Flavio"

    var body: some View {
```

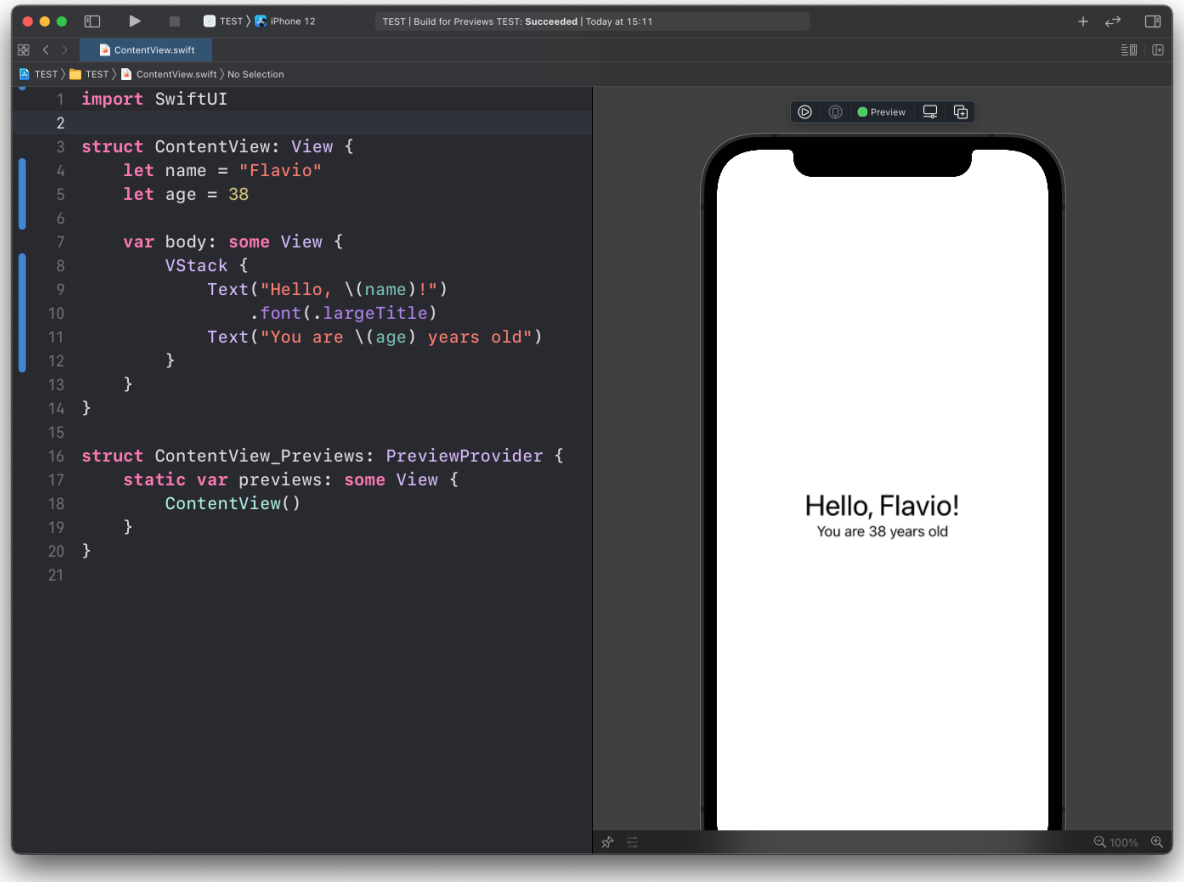
```
        Text("Hello, \(name)!")
            .font(.largeTitle)
    }
}
```



See how I used `let` because the property is a constant.

Here is another example with an integer:

```
import SwiftUI

struct ContentView: View {
    let name = "Flavio"
    let age = 38

    var body: some View {
        VStack {
            Text("Hello, \(name)!")
                .font(.largeTitle)
            Text("You are \(age) years old")
        }
    }
}
```



Constants are fine for data that never changes. Things get interesting when the data has to change.

Why a plain var doesn't work

Suppose you want a counter that increases when the user taps a button. You might try this:

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

    var body: some View {
        Button("Taps: \(count)") {
            count += 1 // does not compile
        }
    }
}
```

This doesn't compile. Views are structs, and a struct can't mutate its own properties from inside `body`.

And even if it could, it wouldn't help. SwiftUI destroys and recreates your view structs constantly, every time something on screen needs updating. Any value stored in a plain property would be reset on the next refresh. Your changes would never stick.

@State

The fix is the `@State` property wrapper:

```
struct ContentView: View {
    @State private var count = 0

    var body: some View {
        Button("Taps: \(count)") {
            count += 1
        }
    }
}
```

`@State` tells SwiftUI: store this value for me, outside the view struct, and keep it alive across refreshes. When the value changes, SwiftUI re-renders the parts of the interface that depend on it.

Two conventions to follow. Mark `@State` properties `private`, because this is state the view owns and nobody else should touch directly. And use it for small value types: a `String`, an `Int`, a `Bool`.

Passing state to children with @Binding

A view often wants a child view to edit its state. Think of a settings screen that owns a `Bool`, and a reusable switch component that flips it.

The parent keeps `@State`. The child declares `@Binding`. To connect them, the parent prefixes the property with `$`:

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

    var body: some View {
        VStack {
            Text(isOn ? "Enabled" : "Disabled")
            PowerSwitch(isOn: $isOn)
        }
    }
}

struct PowerSwitch: View {
    @Binding var isOn: Bool

    var body: some View {
        Toggle("Power", isOn: $isOn)
    }
}
```

The `$` prefix gives you the **projected value** of the state property: a `Binding`. A binding is a read-write connection to the original value. When `PowerSwitch` flips the toggle, it writes straight into the parent's `isOn`, and the `Text` above updates.

The rule of thumb: `@State` for state you own, `@Binding` for state passed to you.

What about bigger objects?

`@State` covers values local to one view. When you have a model class shared across many screens, you mark the class with the `@Observable` macro and pass instances around. The mental model

stays the same: state you own versus state handed to you. We'll get there in a later lesson. For now, `@State` and `@Binding` cover everything a small app needs.

SwiftUI: the Button view and updating the app state

Learn how to use the Button view in SwiftUI to run an action when tapped, and why you need the @State property wrapper to update your app's state.

The `Button` view can be used to display an interactive button element.

We can declare it in this way:

```
Button("Button label") {
    //this happens when it's tapped
}
```

Or in this way:

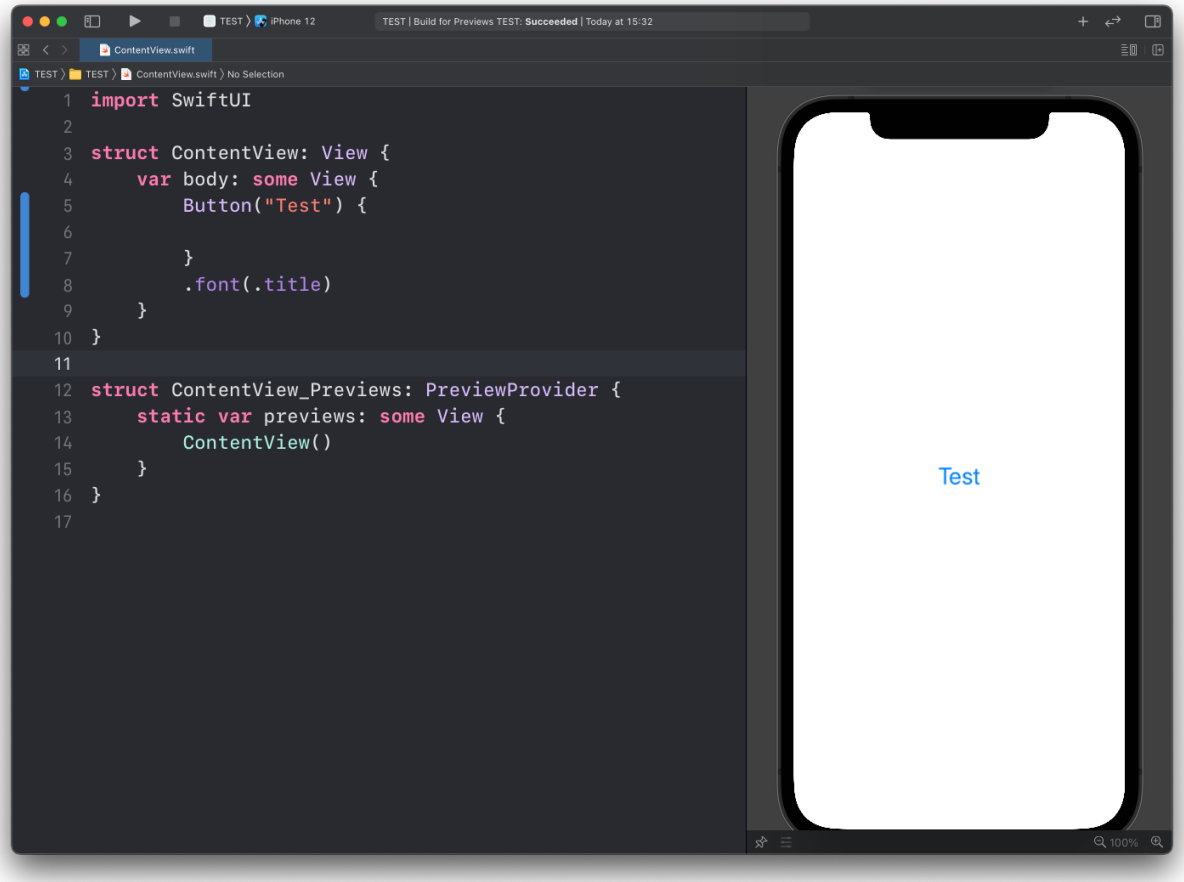
```
Button {
   //this happens when it's tapped
} label: {
   Text("Button label")
}
```

This second way is more common when you have something else as the label of the button, not text. For example an image.

Let's use the first way in a SwiftUI program:

```
struct ContentView: View {
    var body: some View {
        Button("Test") {

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

See? There is a blue text in the app, and you can tap it. It's interactive.



We haven't told it to do anything when tapped, so it does anything.

We can print something to the debugging console:

```
struct ContentView: View {
    var body: some View {
        Button("Test") {
            print("test")
        }
        .font(.title)
    }
}
```

Note that this works only when you run the app, not in the Xcode preview

Now we can add another step to our app. We'll print a property value inside the button label:

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

    var body: some View {
        Button("Count: \(count)") {

        }
        .font(.title)
    }
```

```
}
```

And when it's clicked we increment the count:

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

    var body: some View {
        Button("Count: \(count)") {
            self.count += 1
        }
        .font(.title)
    }
}
```

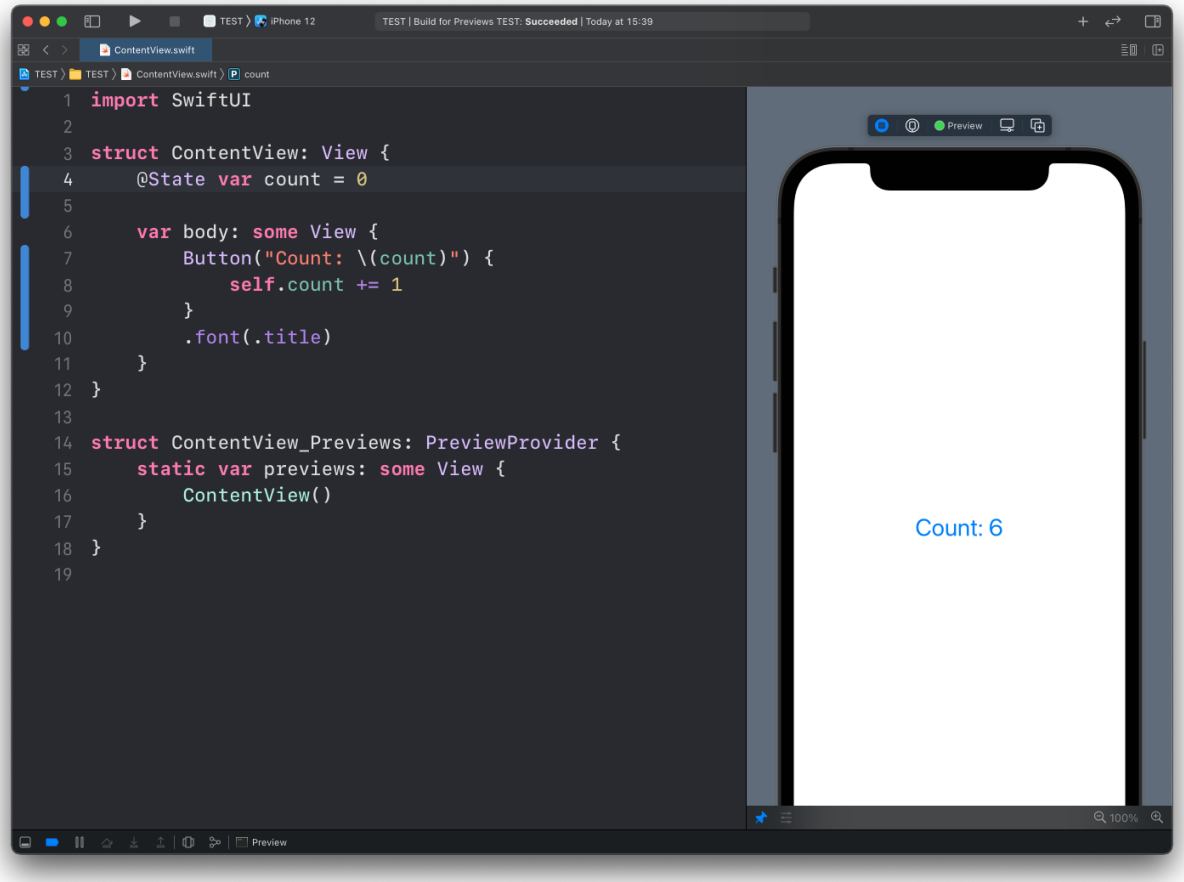
But the app will not compile, with the error

```
Left side of mutating operator isn't mutable: 'self' is immutable
```

We need to use the `@State` property wrapper before we declare the property:

```
struct ContentView: View {
    @State var count = 0

    var body: some View {
        Button("Count: \(count)") {
            self.count += 1
        }
        .font(.title)
    }
}
```

The app will now work and we can click the label to increment the `count` property value:



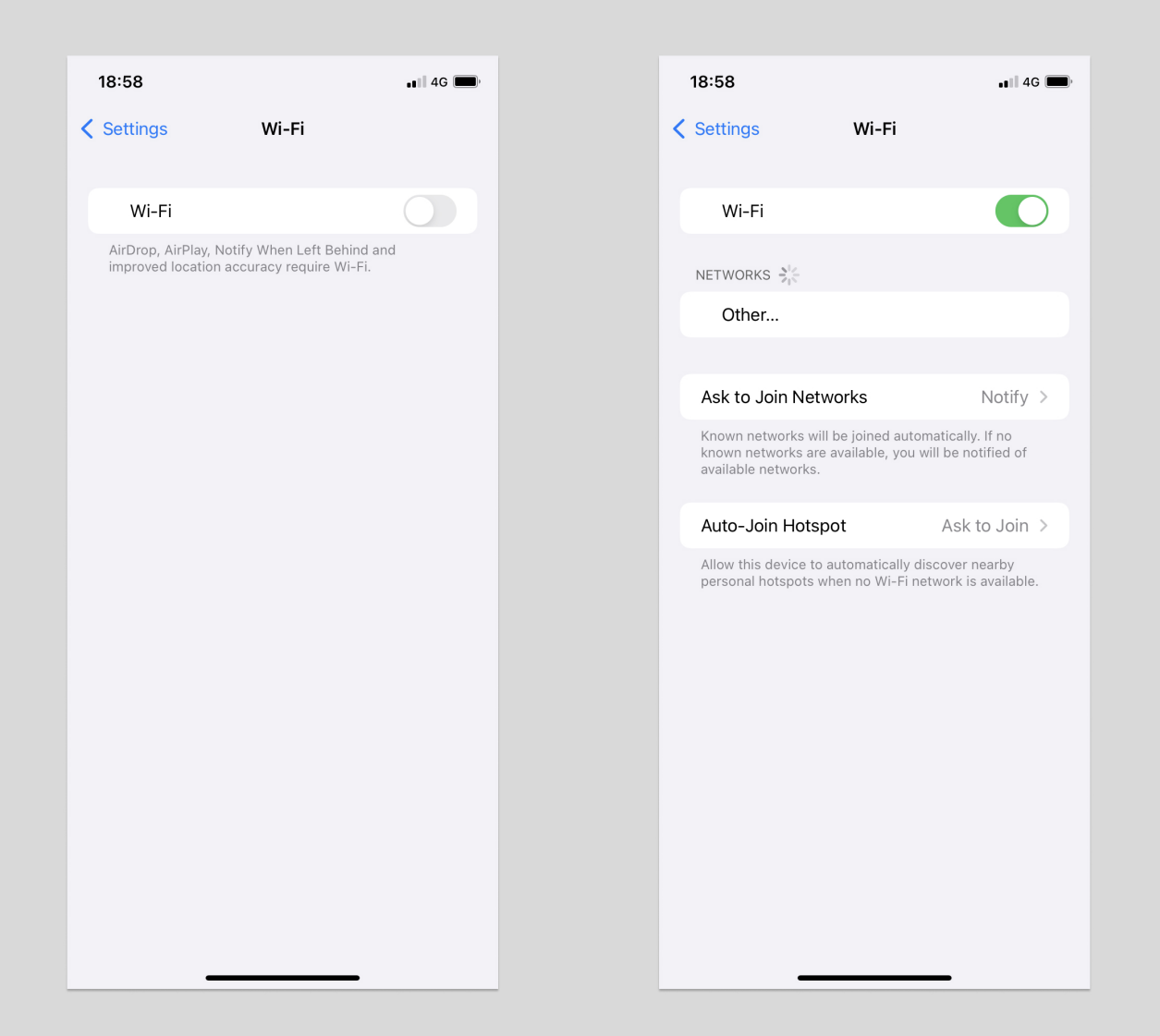
Of course the counter will start from 0 the next time we run it, because we are not persisting the state in any way. We'll get to that later.

SwiftUI: conditionally show items in the view

Learn how to conditionally show or hide views in a SwiftUI form, using an if statement and a Toggle to reveal extra options only when it is enabled.

One common thing to do in a form is to have a toggle and when that toggle is enabled, show a bunch of additional options.

You can see this all the time in the Settings app, for example when you enable WiFi.



How can you do that?

First create a `Form` view with a `Toggle` control:

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

    var body: some View {
        Form {
            Toggle("Enable?", isOn: $enabled)
        }
    }
}
```

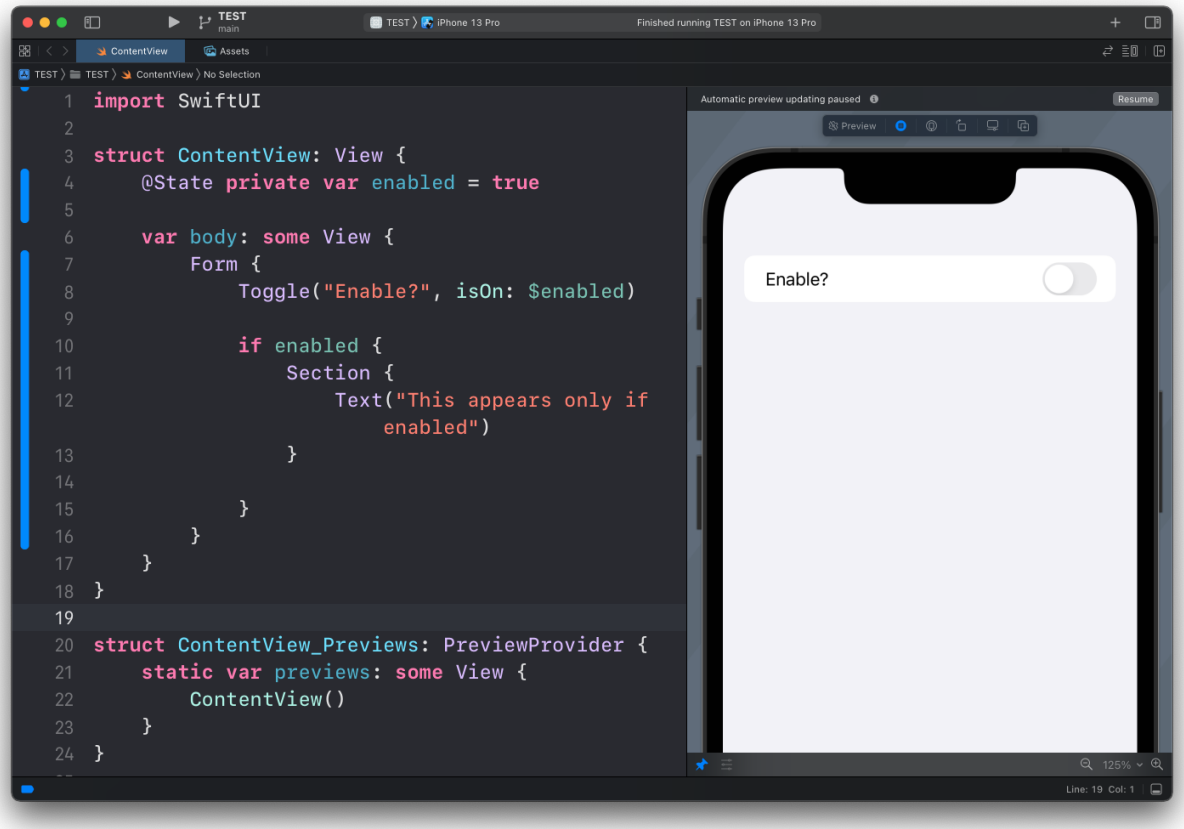
Then add this block after the `Toggle` view:

```
if enabled {
    Section {
        Text("This appears only if enabled")
    }
}
```

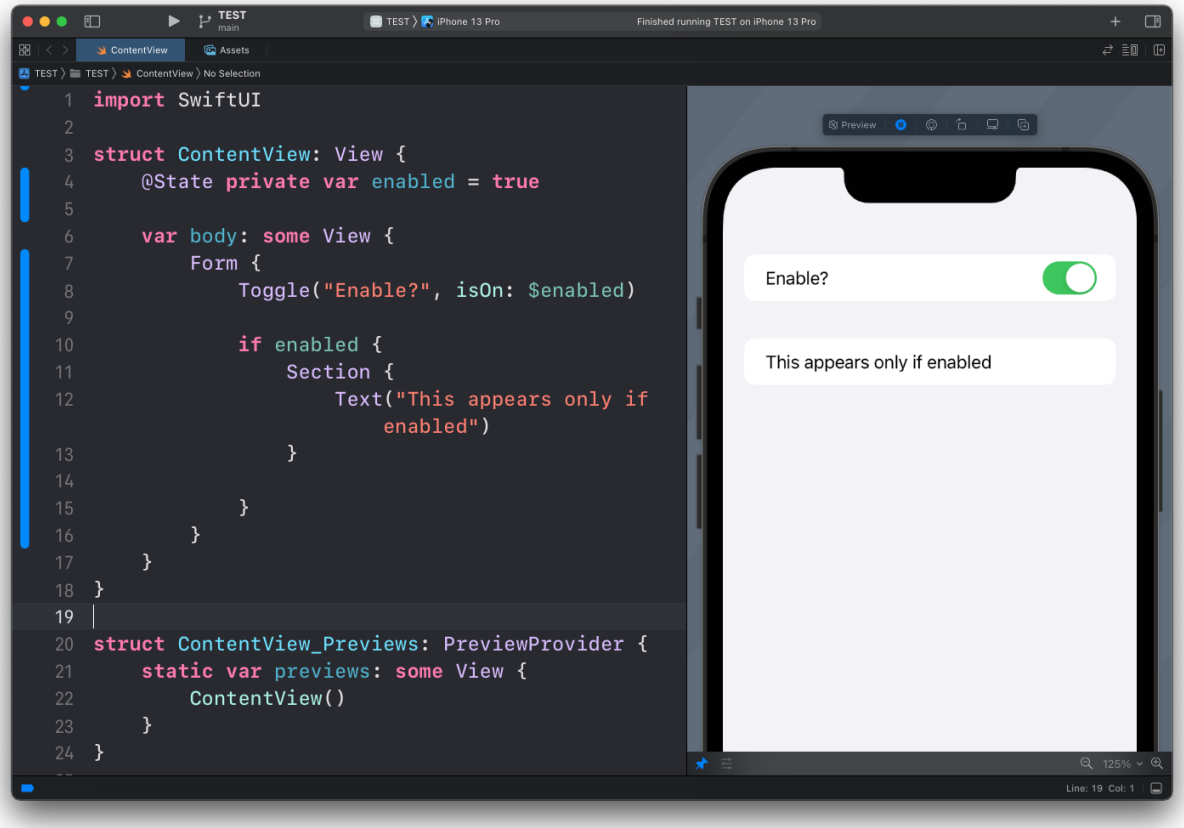
like this:

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

    var body: some View {
        Form {
            Toggle("Enable?", isOn: $enabled)

            if enabled {
                Section {
                    Text("This appears only if enabled")
                }
            }
        }
    }
}
```

Now with the toggle disabled, the Text view will not be visible:



But when you enable the toggle, it appears:



SwiftUI: alert messages

Learn how to show an alert in SwiftUI using the .alert() modifier and an isPresented boolean, so a message pops up when a condition becomes true.

A common way to give feedback to users, but also to help you debug your applications sometimes, is to use alerts.

SwiftUI provides the `.alert()` modifier that we can use to show an alert based on some condition.

Let's start from this example where we have a `Button` with a counter:

```
import SwiftUI

struct ContentView: View {
    @State var count = 0

    var body: some View {
        Button("Count: \(count)") {
            self.count += 1
        }
        .font(.title)
    }
}
```

when `count` reaches 10 I want to show an alert message.

How do we do that?

I can add a `showAlert` boolean property to the `ContentView` and edit the content of the `Button` tap action:

```
struct ContentView: View {
    @State var count = 0
    @State var showAlert = false

    var body: some View {
        Button("Count: \(count)") {
            self.count += 1
            if self.count == 10 {
                showAlert = true
                self.count = 0
            }
        }
        .font(.title)
    }
}
```

When we reach 10, we set `showAlert` to `true` and we set the count to 0.

Then I add the `.alert()` modifier to the `Button` view:

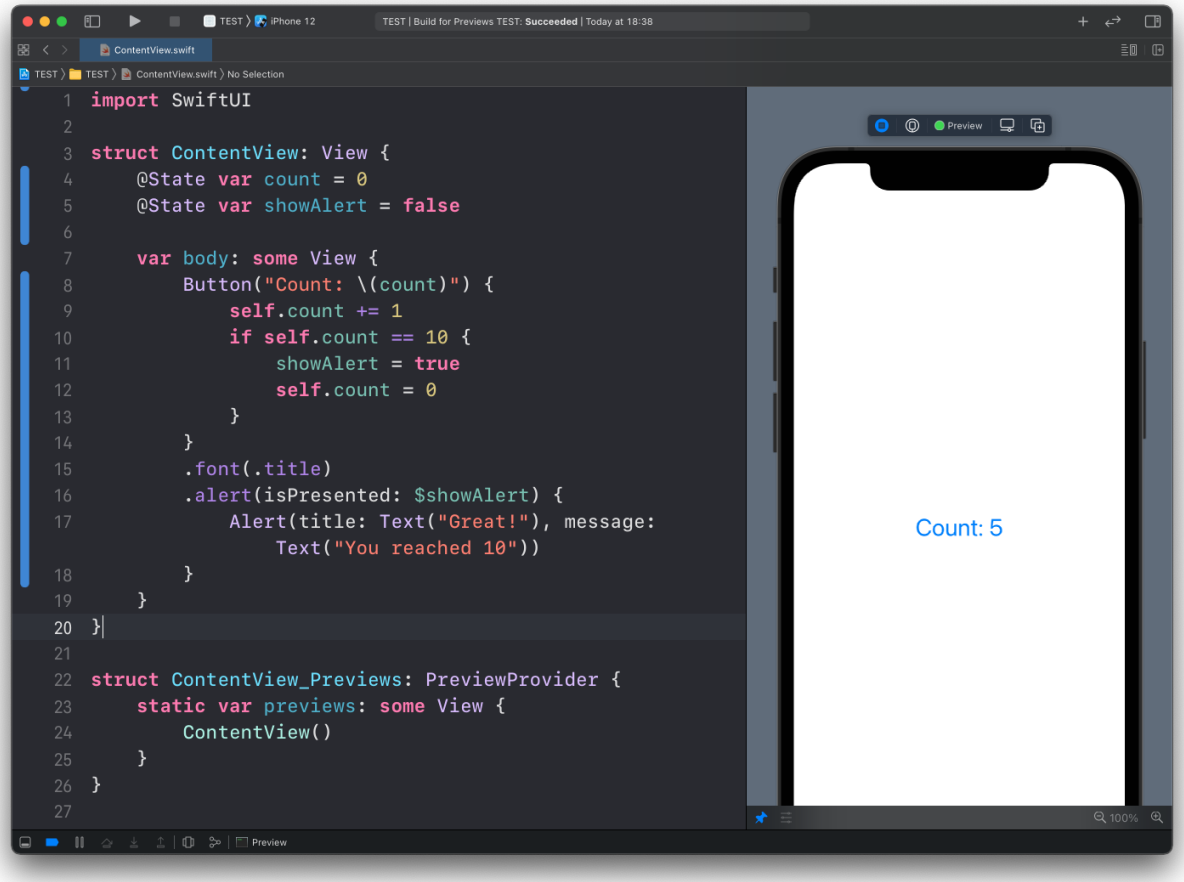
```
.alert(isPresented: $showAlert) {
    Alert(title: Text("Great!"), message: Text("You reached 10"))
}
```

This alert is shown only when the `showAlert` property is true. When we dismiss the alert, the `showAlert` property is automatically set to `false`.

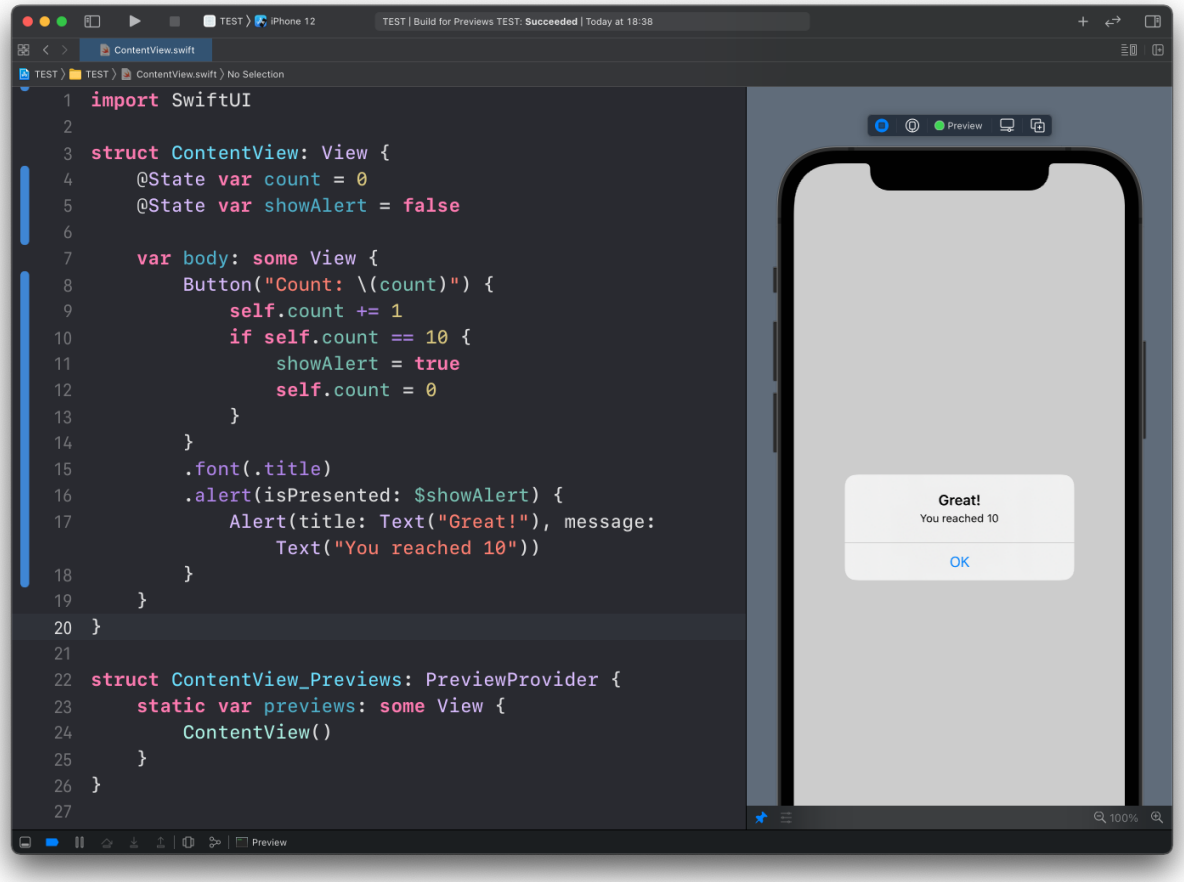
Here's the full code:

```
struct ContentView: View {
    @State var count = 0
    @State var showAlert = false

    var body: some View {
        Button("Count: \(count)") {
            self.count += 1
            if self.count == 10 {
                showAlert = true
                self.count = 0
            }
        }
        .font(.title)
        .alert(isPresented: $showAlert) {
            Alert(title: Text("Great!"), message: Text("You reached 10"))
        }
    }
}
```

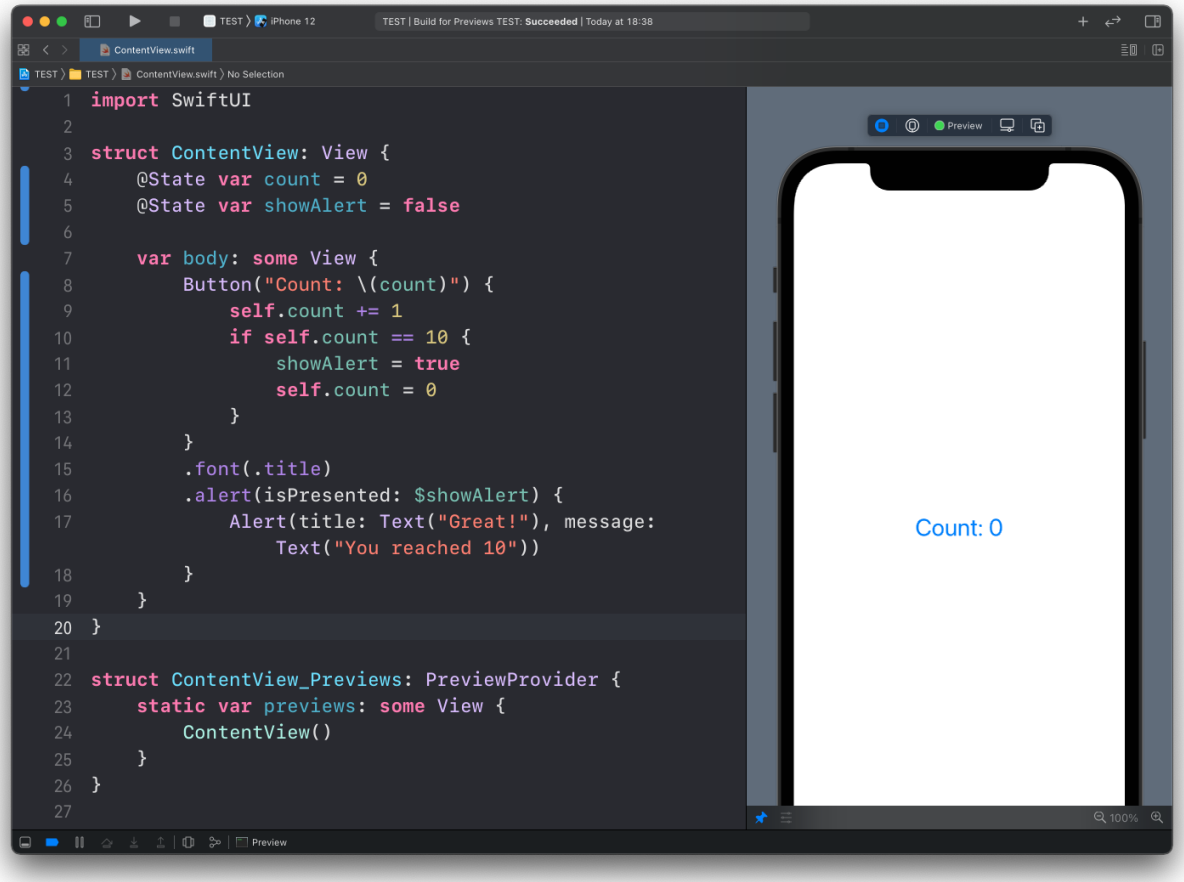
Try running the app. The counter starts at 0. Click the button and the counter will increase:



Until you reach 10:



Then the counter will start again from 0:



Check your understanding: state and actions

Check the key ideas from state and actions before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Collections and layout

Render identifiable data with lists, ForEach, labels, and adaptive spacing.

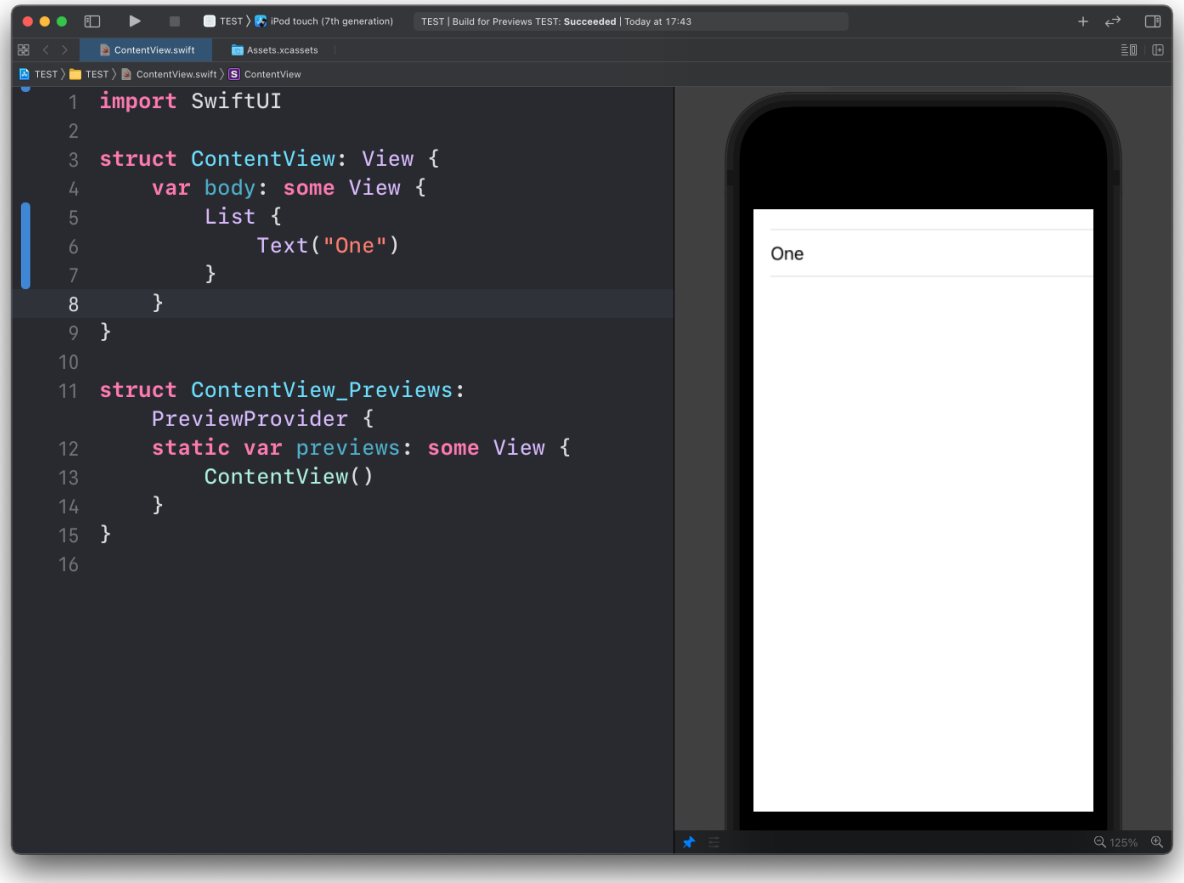
SwiftUI: the List view

Learn how to use the List view in SwiftUI to show data in rows, group items with the Section view, and change its look with the listStyle modifier.

The **List** view is one of the most useful views you'll use in SwiftUI.

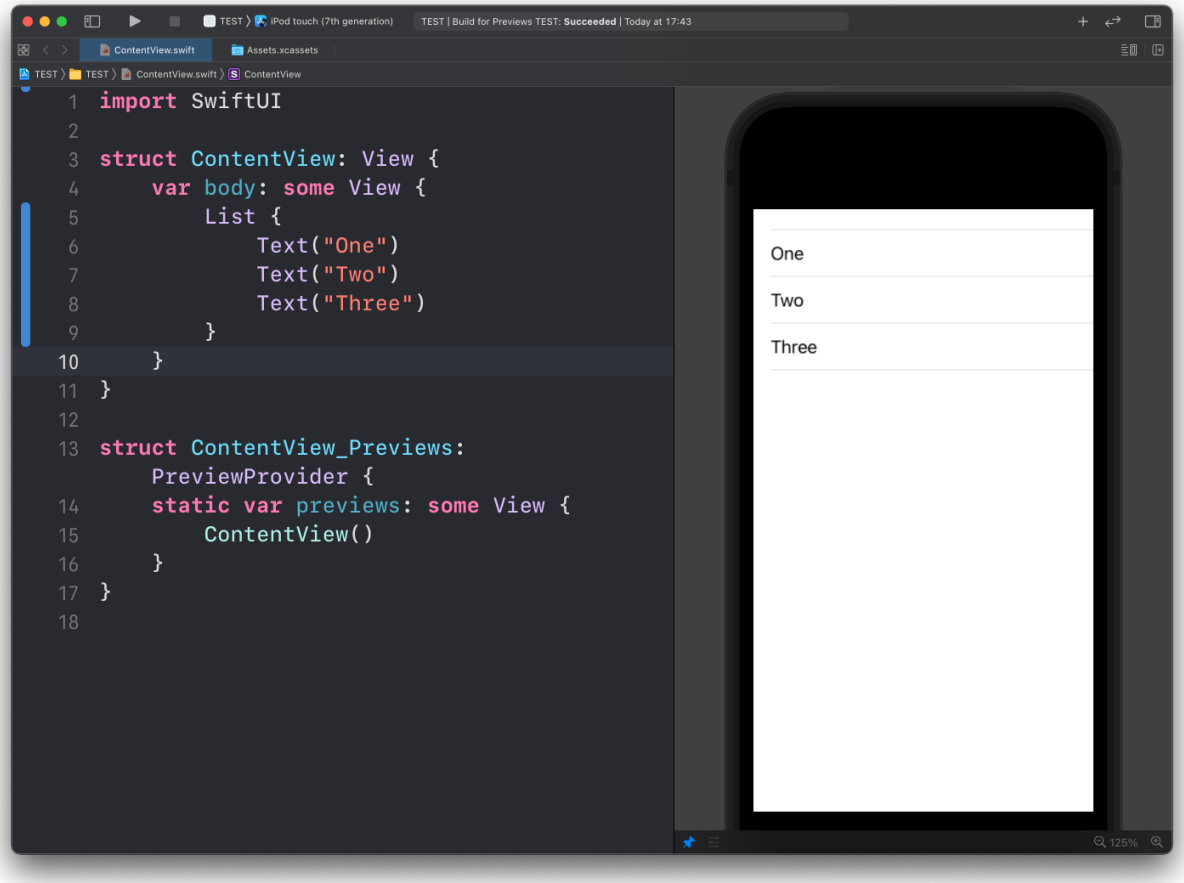
```
List {

}
```

Inside it, you can put a series of views, like `Text` for example:

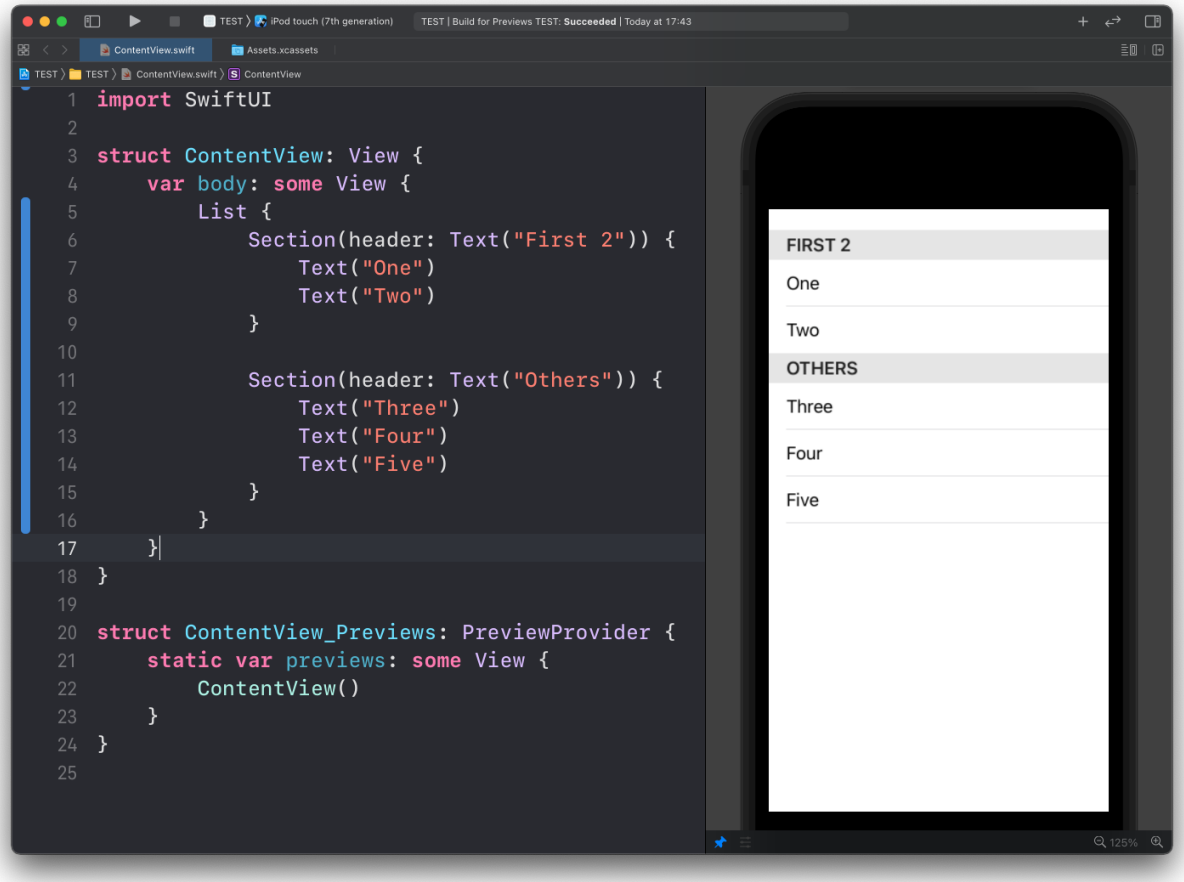


See? `List` recognizes the `Text` child view, and puts it inside a row.

You can put more than one, and each child of List will be put on its own row:



Inside a list, you can group items using the `Section` view, like this:

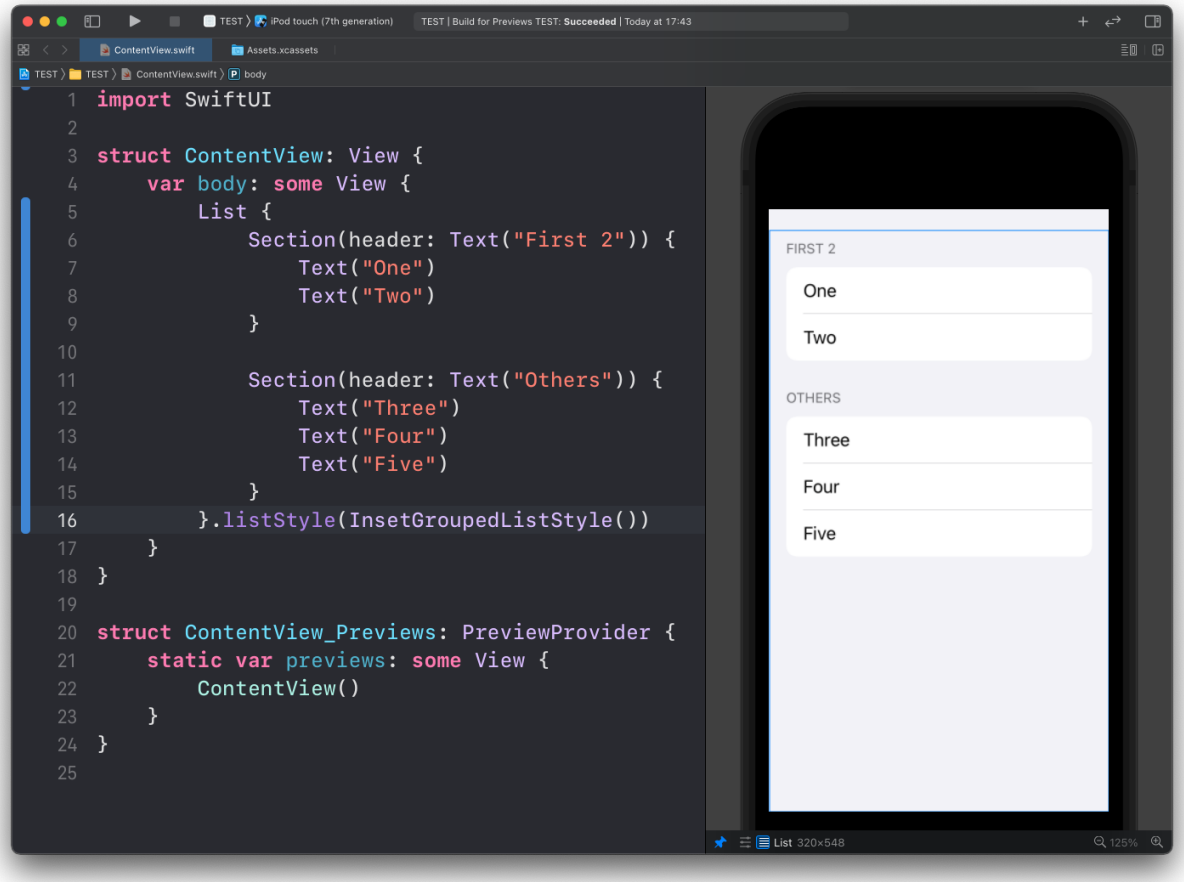


The `listStyle()` modifier of `List` can let you customize the List appearance, using

- `InsetGroupedListStyle`
- `InsetListStyle`
- `SidebarListStyle`
- `GroupedListStyle`
- `PlainListStyle`

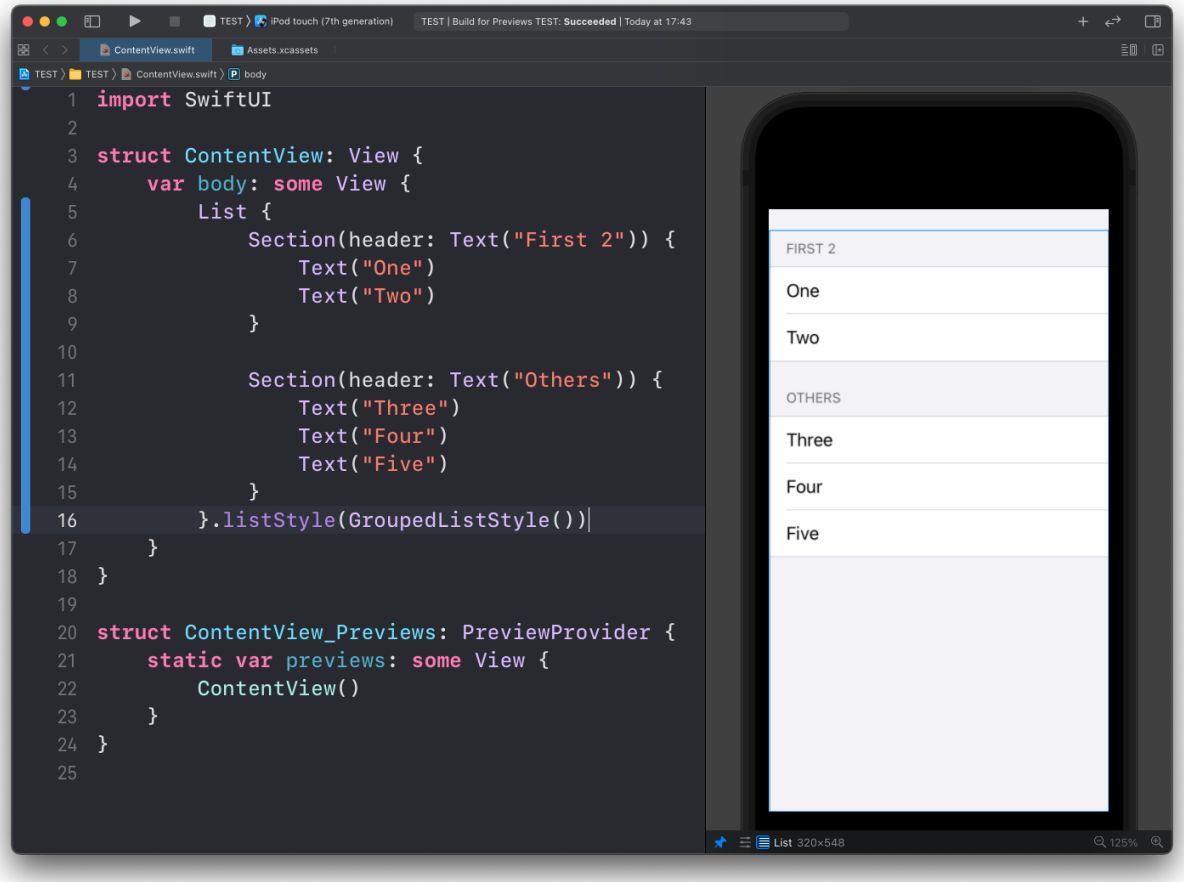
For example here's `InsetGroupedListStyle`:

```
List {
        //...
}.listStyle(InsetGroupedListStyle())
```

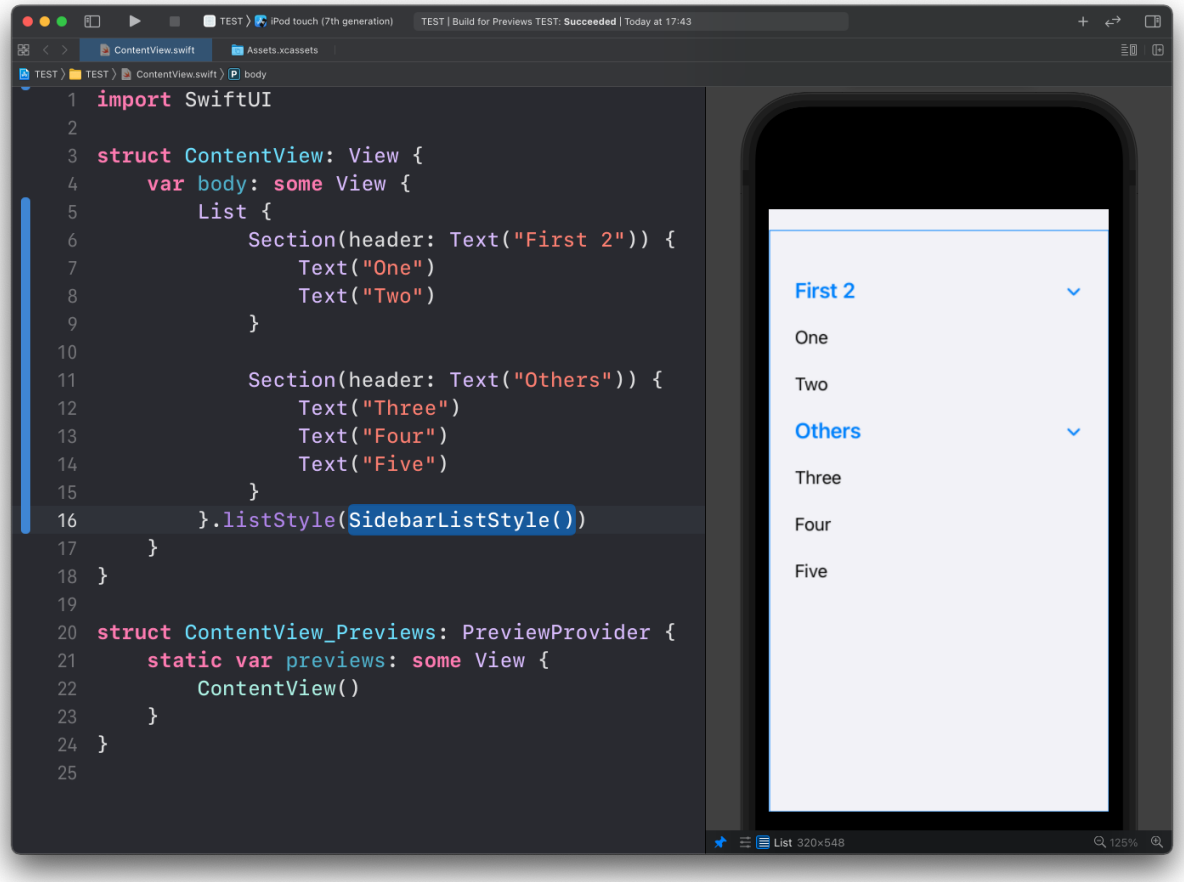


And here's `GroupedListStyle`:

```
List {
        //...
}.listStyle(GroupedListStyle())
```



Here's `SidebarListStyle`:



SwiftUI: the ForEach view

Learn how to use the ForEach view in SwiftUI to loop over a range or an array and generate views, including how the id parameter identifies each item.

The `ForEach` view in SwiftUI is very useful to iterate over an array, or a range, and generate views that we can use.

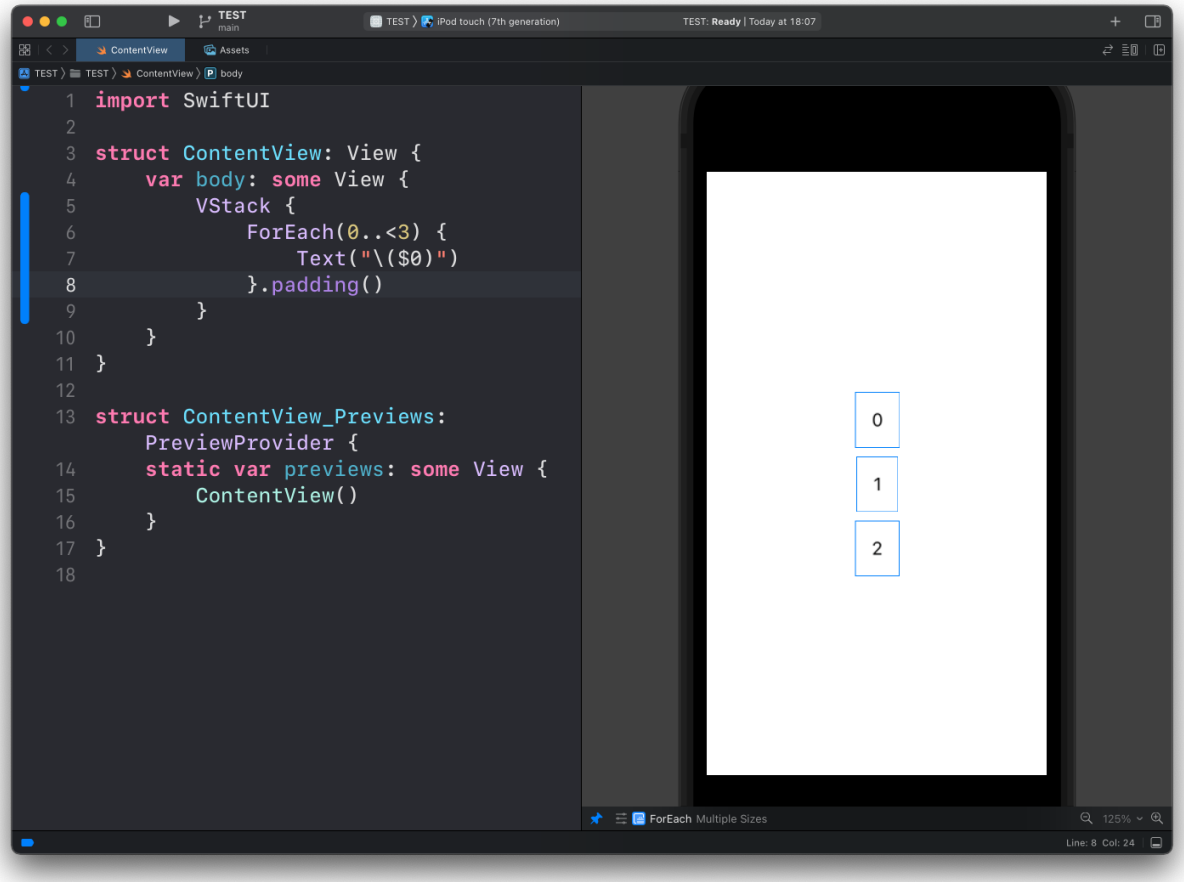
For example, here we create 3 `Text` views that print the numbers from 0 to 2:

```
ForEach(0..<3) {
    Text("\($0)")
}
```

> $0 means the first argument passed to the closure, which in this case it's (in order) the number 0, 1, and 2.

In this example I embed them in a `VStack` otherwise they would overlap:

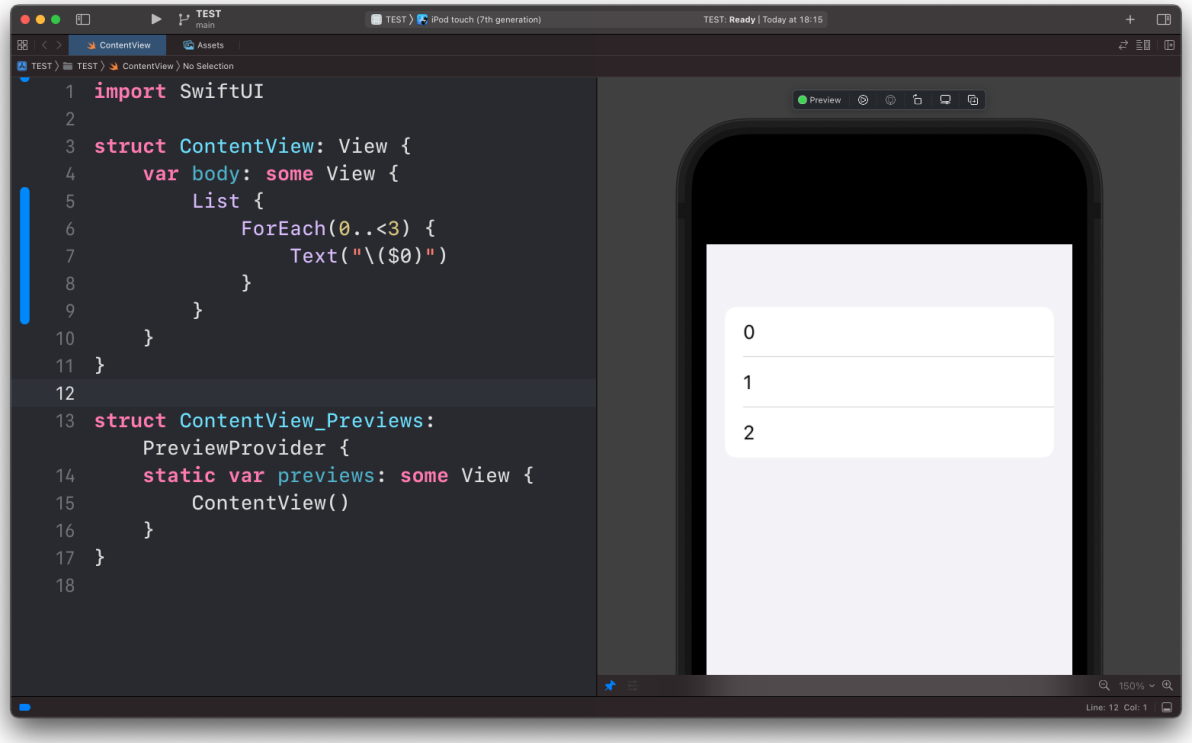
```
VStack {
    ForEach(0..<3) {
        Text("\($0)")
    }.padding()
}
```



Notice how I used the `padding()` modifier to add some spacing

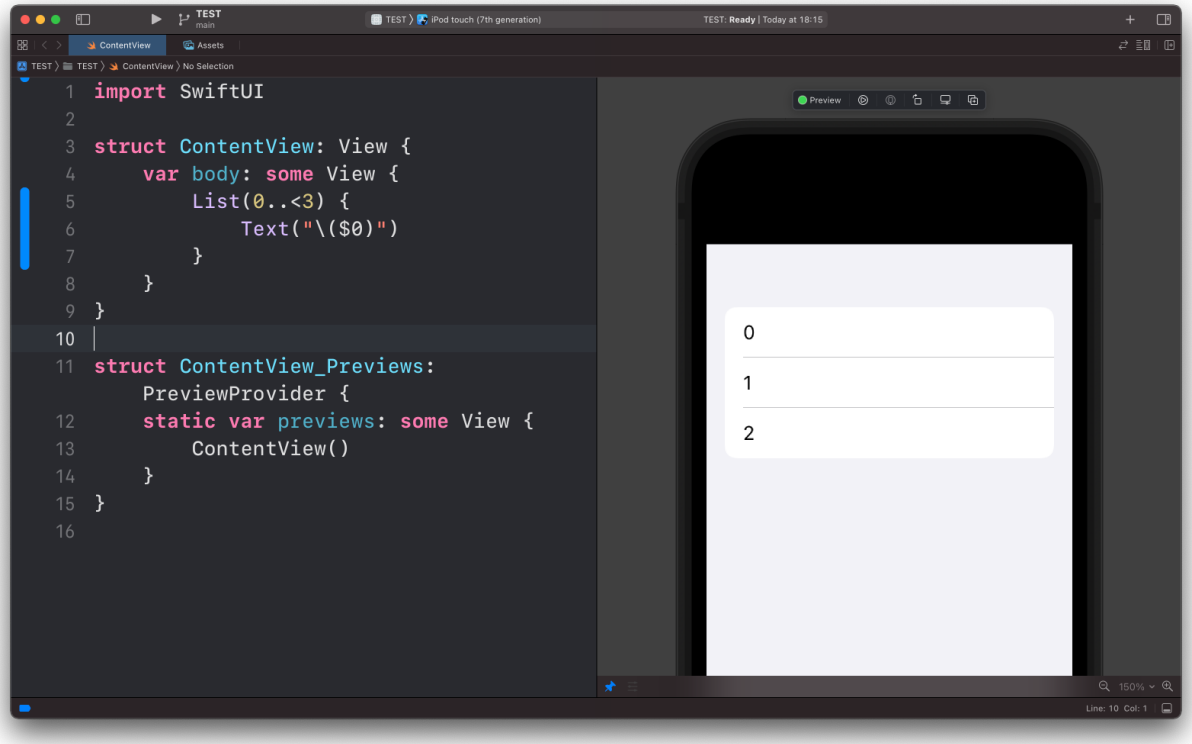
A common way to use `ForEach` is inside a `List` view:

```
List {
    ForEach(0..<3) {
        Text("\($0)")
    }
}
```



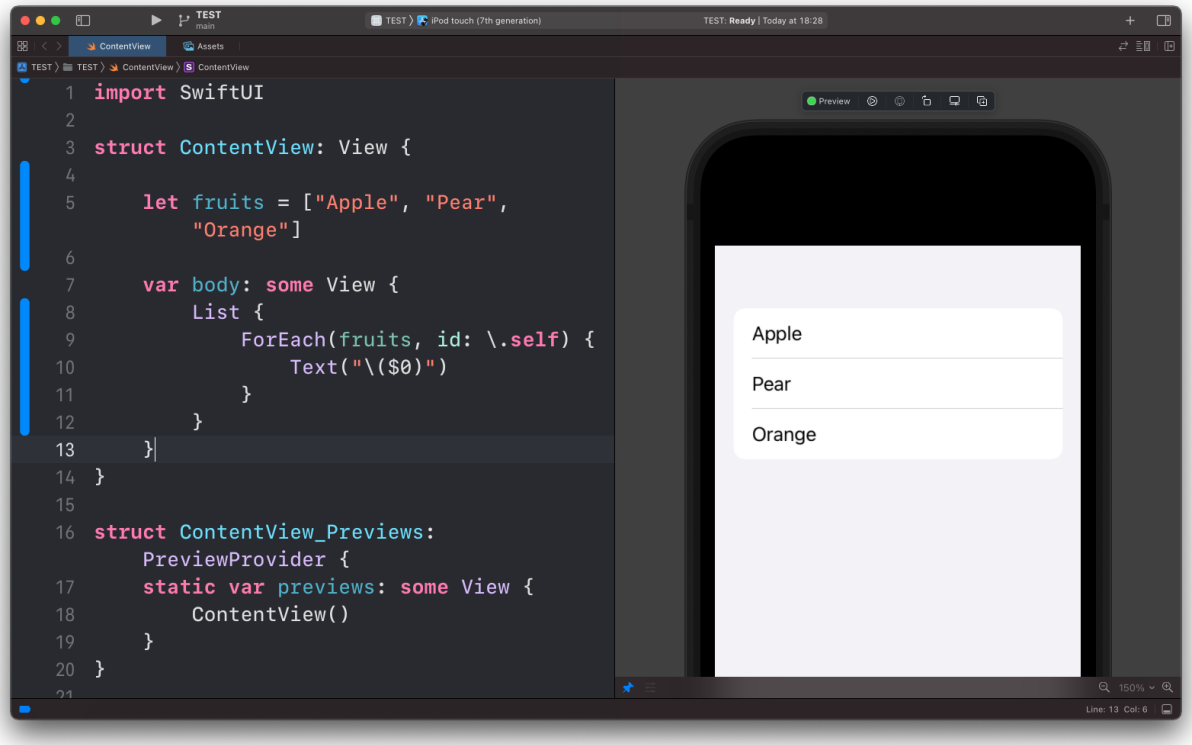
This is such a common thing to do that we can actually omit `ForEach` and iterate directly from `List`:

```
List(0..<3) {
    Text("\($0)")
}
```



Those 2 examples used a range `0..<3`. We can iterate over an array too:

```
let fruits = ["Apple", "Pear", "Orange"]

//...

List {
    ForEach(fruits, id: \.self) {
        Text("\($0)")
    }
}
```



Notice how in this case we have another parameter: `id`.

This is to uniquely identify the item in the array.

Using `\.self` for `id` works for built-in types, in case you are iterating a custom struct you'll need that to conform to the `Identifiable` protocol or provide a unique parameter.

SwiftUI: the Label view

Learn how to use the Label view in SwiftUI to pair an icon with a title, browse SF Symbols, use custom images, and switch styles with labelStyle.

The `Label` view shows an icon and a title together, as a single view.

That's the key idea. Not an icon placed next to some text, but one semantic unit. ”Calendar” with a calendar icon. SwiftUI treats them as one thing, and that unlocks some nice behaviors we'll see below.

You create a Label by passing the title as the first argument, and the icon name to the `systemImage` parameter:

```
Label("Calendar", systemImage: "calendar")
```

This will be the result:

The string you pass to `systemImage` is the name of an **SF Symbols** icon. SF Symbols is Apple's icon library, built into every Apple platform, with thousands of icons designed to align perfectly with the system font.

To find the name of the icon you want, download the free SF Symbols app from the Apple website. You can browse and search every symbol there. Search "calendar" and you'll also find `calendar.badge.plus`, `calendar.circle`, and a dozen more variants.

Using your own images

If the icon you need is an image from your asset catalog instead of an SF Symbol, use the `image` parameter:

```
Label("Profile", image: "avatar")
```

For full control over the icon, use the initializer that takes two view builders, one for the title and one for the icon:

```
Label {
    Text("Profile")
} icon: {
    Image("avatar")
        .resizable()
        .frame(width: 24, height: 24)
        .clipShape(Circle())
}
```

Showing only the icon, or only the title

Sometimes you want the same label to show just the icon, maybe in a compact toolbar. Don't remove the title. Apply the `labelStyle()` modifier instead:

```
Label("Calendar", systemImage: "calendar")
```

```
    .labelStyle(.iconOnly)
```

The built-in styles are `.titleAndIcon`, `.titleOnly` and `.iconOnly`. The default is `.automatic`, which lets the surrounding context decide.

Keeping the title around even when it's hidden matters for accessibility. VoiceOver still reads "Calendar", because the label knows what it represents.

Why not just an HStack?

You could build something that looks identical with an `HStack` containing an `Image` and a `Text`:

```
HStack {
    Image(systemName: "calendar")
    Text("Calendar")
}
```

My advice is to always prefer `Label`. Here's why.

First, accessibility. A Label is one element for VoiceOver. The HStack version is two separate elements, and the icon might get announced on its own, which is just noise.

Second, adaptivity. Views like `List`, `TabView`, menus and toolbars know how to handle labels. A tab bar shows the icon above the title. A menu places the icon on the correct side for the locale. A toolbar might drop the title when space is tight. Label adapts to all these contexts for free. Your custom HStack doesn't.

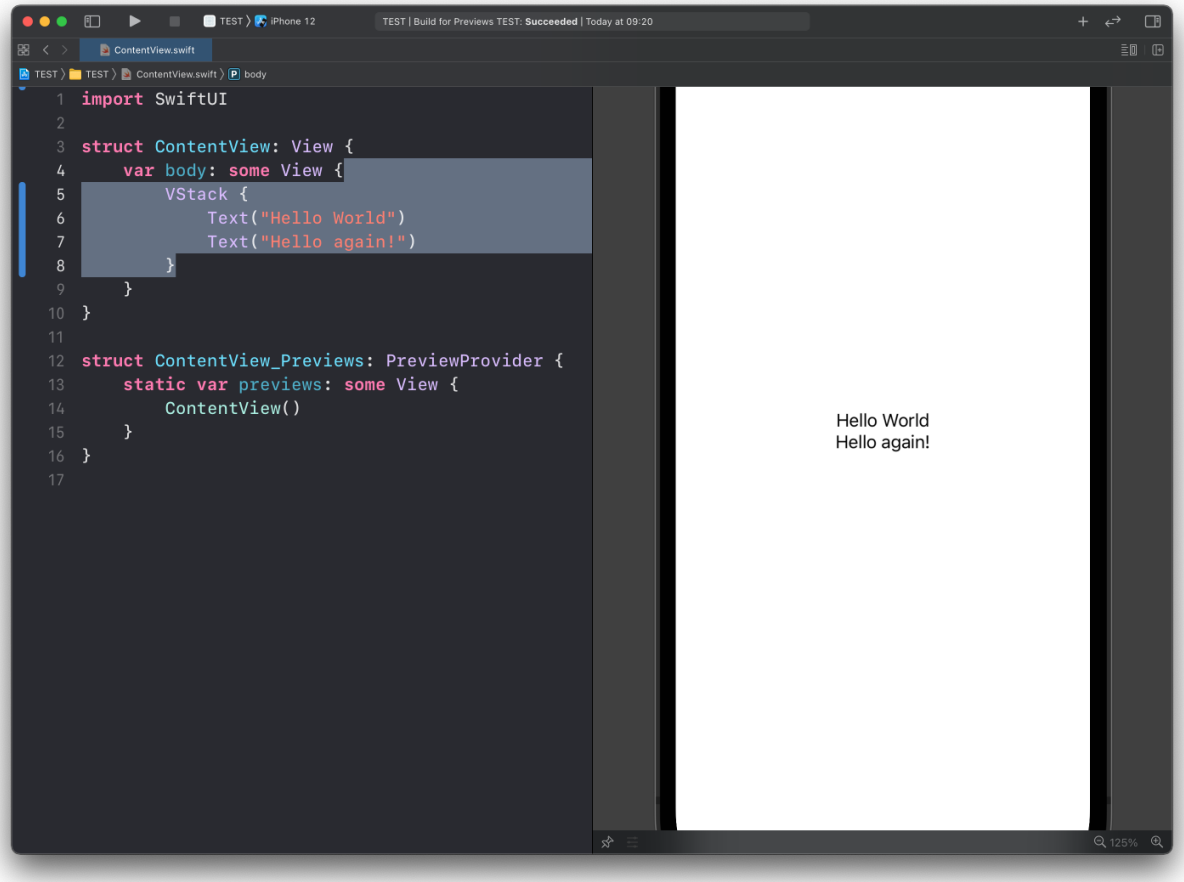
Third, alignment. In a `List`, labels align their icons and titles consistently across rows, even when the icons have different widths.

SwiftUI: spacing

Learn how to control spacing in SwiftUI with the stack spacing parameter, the padding modifier, the Spacer view, and the frame modifier.

In the last SwiftUI tutorial I mentioned how views can be arranged using stacks:

```
VStack {
    Text("Hello World")
    Text("Hello again!")
}
```



Let's talk about spacing. You have three main tools: the `spacing` parameter of stacks, the `padding()` modifier, and the `Spacer` view. Each one solves a different problem.

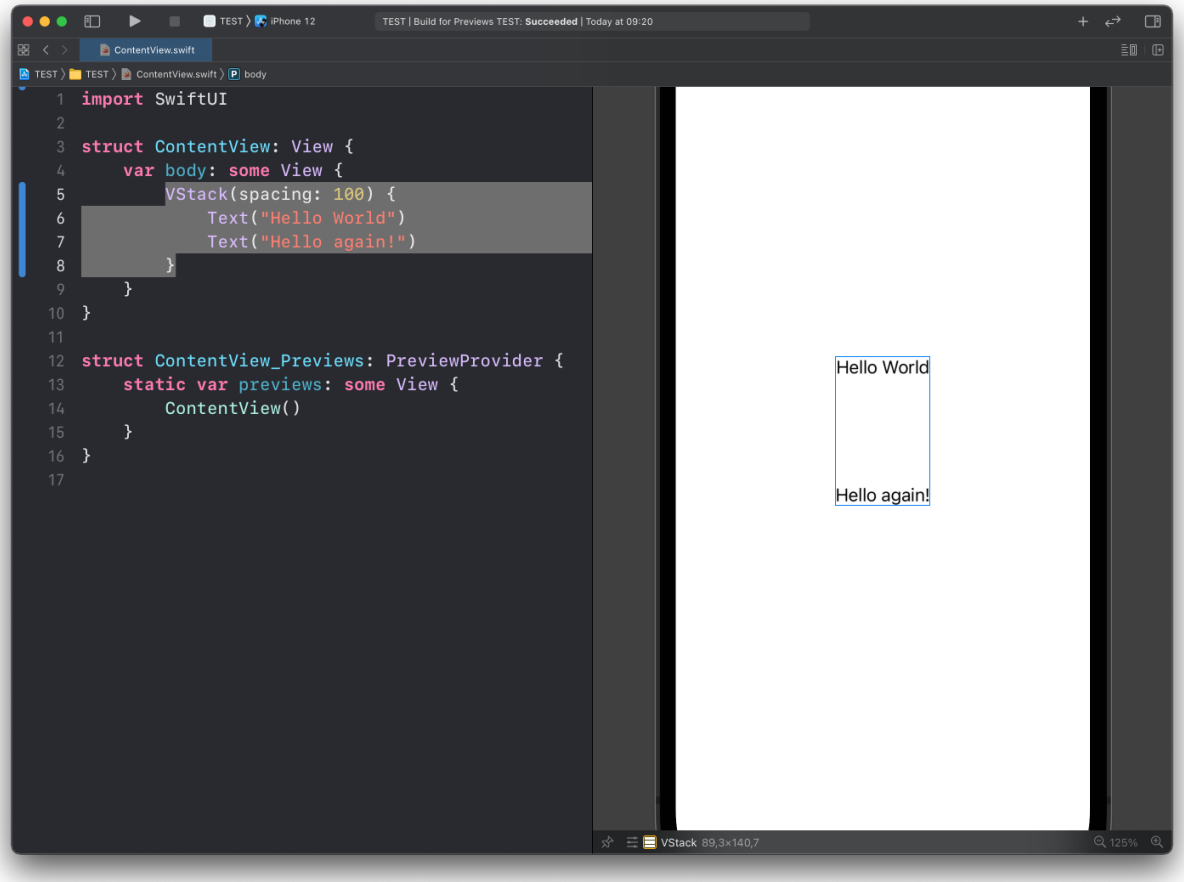
The spacing parameter

See how the two `Text` views sit very close together? Stacks apply a small default spacing between children.

`VStack` (like `HStack`) accepts a `spacing` parameter to change it:

```
VStack(spacing: 100) {
    Text("Hello World")
    Text("Hello again!")
}
```

This puts a 100 points space between the views contained in the `VStack`.



Use `spacing` for the gap between siblings. It applies to every pair of adjacent children in the stack, so it keeps rhythm consistent.

padding()

While `spacing` works between views, `padding()` adds space around one view:

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

With no arguments you get a system-chosen amount on all four edges. You can pick specific edges and an exact amount:

```
Text("Hello World")
    .padding(.horizontal, 16)
    .padding(.top, 8)
```

The edge options are `.top`, `.bottom`, `.leading`, `.trailing`, plus the shortcuts `.horizontal`, `.vertical` and `.all`.

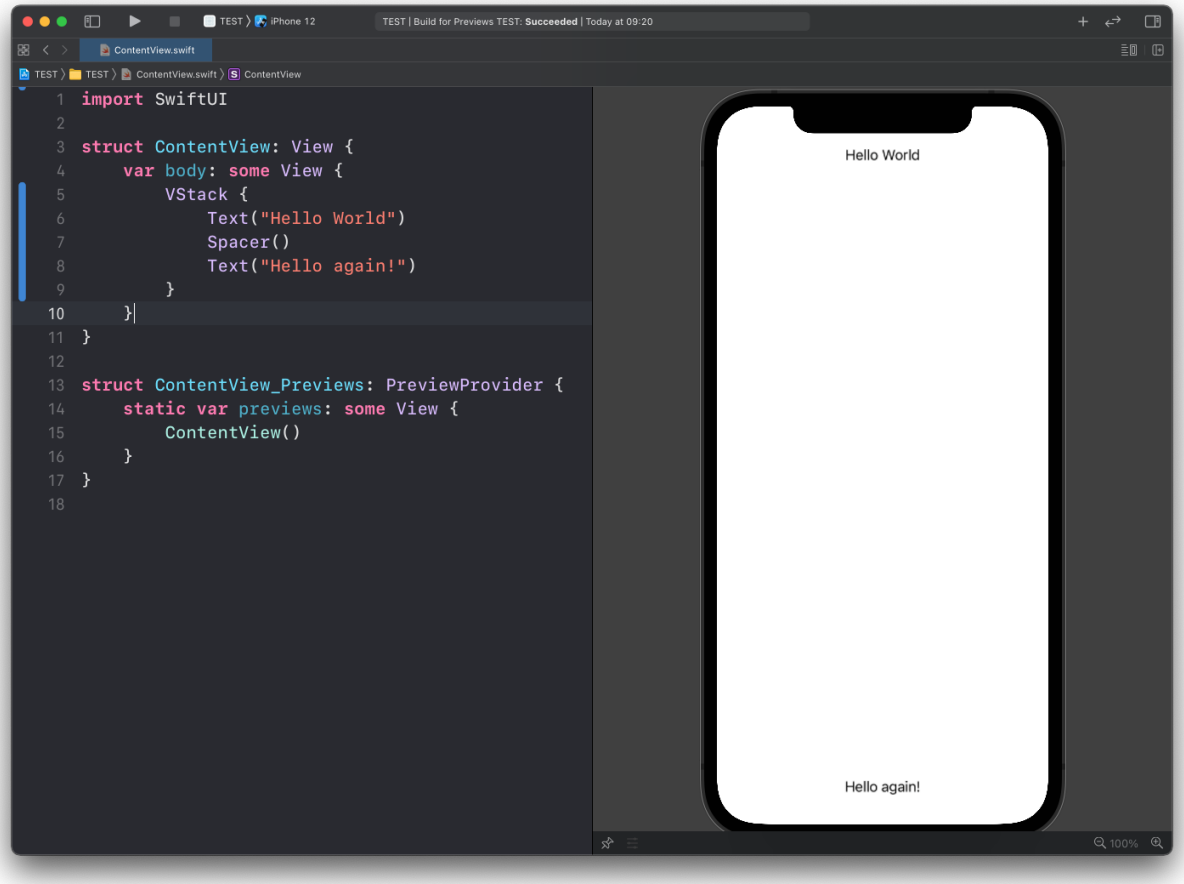
Notice we say `.leading` and `.trailing` instead of left and right. In a right-to-left language the layout flips automatically.

Spacer

`Spacer` is a view that expands to fill all the available space:

```
VStack {
```

```
    Text("Hello World")
    Spacer()
    Text("Hello again!")
}
```



The spacer grabbed everything the texts didn't need, pushing one to the top and one to the bottom. In an `HStack`, a Spacer pushes horizontally instead. That's the classic way to push a button to the trailing edge.

You can limit it to a specific set of points using the `frame()` modifier:

```
VStack {
    Text("Hello World")
    Spacer()
      .frame(height: 20)
    Text("Hello again!")
}
```

Two text views with a Spacer limited to 20 points height using frame modifier

Although if you want a fixed 20 points gap, the `spacing` parameter or `padding()` says that more directly.

frame(maxWidth:) for filling and aligning

One more tool worth knowing early. You can ask a view to take all the available width, and control where its content sits:

```
Text("Hello World")
    .frame(maxWidth: .infinity, alignment: .leading)
```

The text now occupies the full width of its parent, with the content aligned to the leading edge. This is the standard way to left-align text inside a container that would otherwise center it.

My advice

Prefer adaptive spacing over fixed coordinates. Screens vary a lot: an iPhone SE, a Pro Max, an iPad in split view, larger Dynamic Type sizes. `spacing`, `padding()`, `Spacer` and `frame(maxWidth:)` all adapt to whatever space they're given. Magic numbers don't.

If you catch yourself typing something like `.padding(.leading, 37)` to nudge a view into place, step back. There's almost always an alignment, a stack, or a Spacer that expresses what you actually want.

Check your understanding: collections and layout

Check the key ideas from collections and layout before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Forms and input

Build platform-appropriate forms with text fields, toggles, and pickers.

SwiftUI forms

Learn how forms work in SwiftUI by wrapping controls like TextField, Toggle, and Picker in a Form view that styles them right for each platform.

SwiftUI provides several form controls that we can use to get input from the user.

Think about the Settings app on your iPhone. That entire app is a series of forms, built with controls like:

- `TextField`
- `Toggle`
- `Picker`
- and others

All of those are wrapped in a `Form` view:

```
Form {

}
```

What Form gives you over a VStack

You could put the same controls in a `VStack`. They would render, but you would have to style everything yourself.

`Form` tells SwiftUI "this is a form", and SwiftUI takes over the presentation. On iOS you get the grouped, inset look of the Settings app: rows with separators, the right background, correct paddings and fonts. The controls adapt too. A `Picker` inside a Form becomes a row showing the current value, and tapping it navigates to the options.

This is the declarative side of SwiftUI at its best. You declare intent, the platform decides presentation.

Sections

Forms grow. `Section` groups related rows, with an optional header and footer:

```
Form {
    Section("Profile") {
        TextField("Username", text: $username)
    }
    Section {
        Toggle("Allow notifications", isOn: $notificationsEnabled)
    } footer: {
        Text("You can change this later in Settings.")
    }
}
```

The header appears above the group in small capitals. The footer appears below in smaller gray text, perfect for those short explanations you see under rows in the Settings app.

A realistic settings form

Here's a small settings screen combining the controls we'll cover in the next lessons:

```
struct SettingsView: View {
    @State private var username = ""
    @State private var notificationsEnabled = true
    @State private var preview = "Short"

    let previews = ["Off", "Short", "Full"]

    var body: some View {
        Form {
            Section("Account") {
                TextField("Username", text: $username)
            }

            Section("Notifications") {
                Toggle("Allow notifications", isOn: $notificationsEnabled)
                Picker("Preview", selection: $preview) {
                    ForEach(previews, id: \.self) {
                        Text($0)
                    }
                }
            }
        }
```

```
    }
}
```

Forms almost always live inside navigation, so wrap the view in a `NavigationStack` and give it a title:

```
NavigationStack {
    SettingsView()
        .navigationTitle("Settings")
}
```

With about thirty lines you get a screen that looks like Apple built it. No manual styling anywhere.

The same form on other platforms

Run this exact code on a Mac and it renders as a native macOS form: labels aligned in one column, controls in another. You can also opt into the grouped, settings-style look with the `.formStyle(.grouped)` modifier.

On watchOS the rows become the carousel-style list the platform expects. You wrote the form once, and each platform presents it its own way.

We'll see more about forms by covering each individual form control.

SwiftUI forms: TextField

Learn how to use the TextField control in SwiftUI to get text input, bind it to State, configure the keyboard, handle submit, and manage focus.

The first form control we'll see is the simplest: `TextField`.

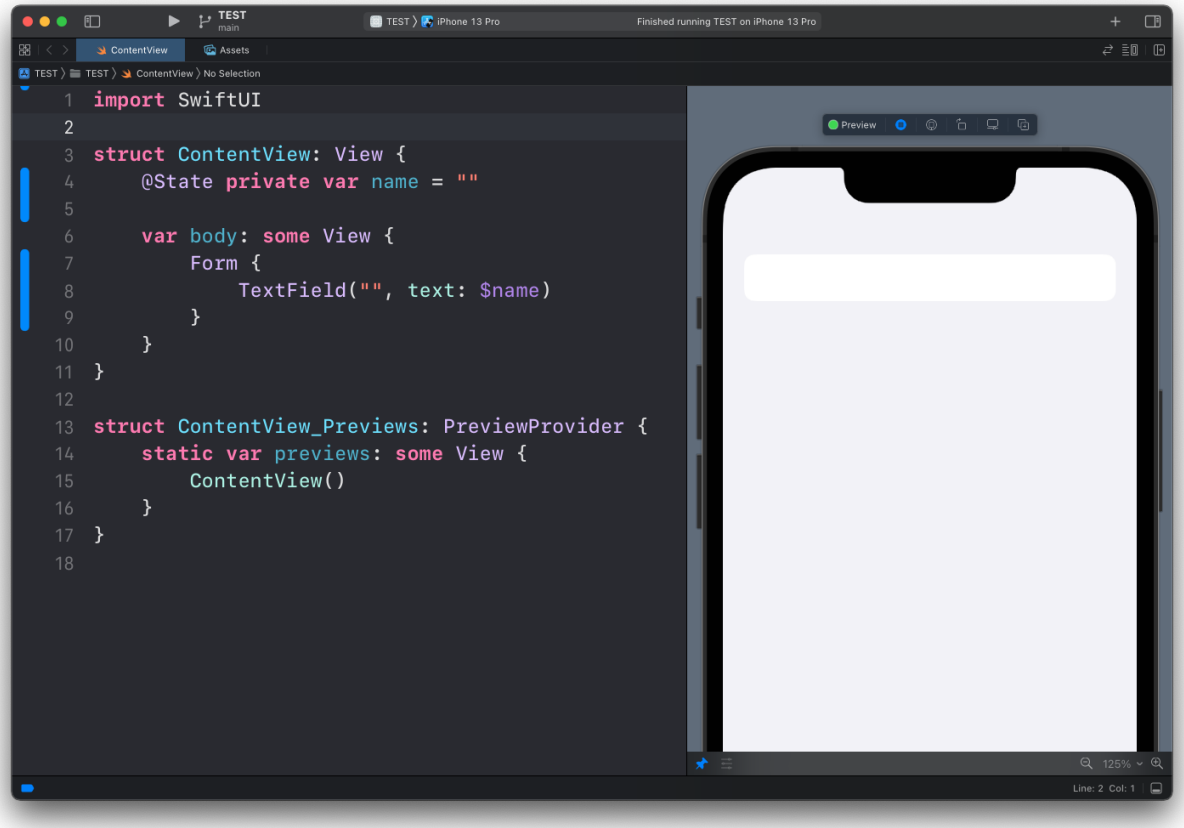
This lets us show some text, like the `Text` view, and it can be edited by the user, so we can get input in the form of text.

Here's the most basic example of `TextField`:

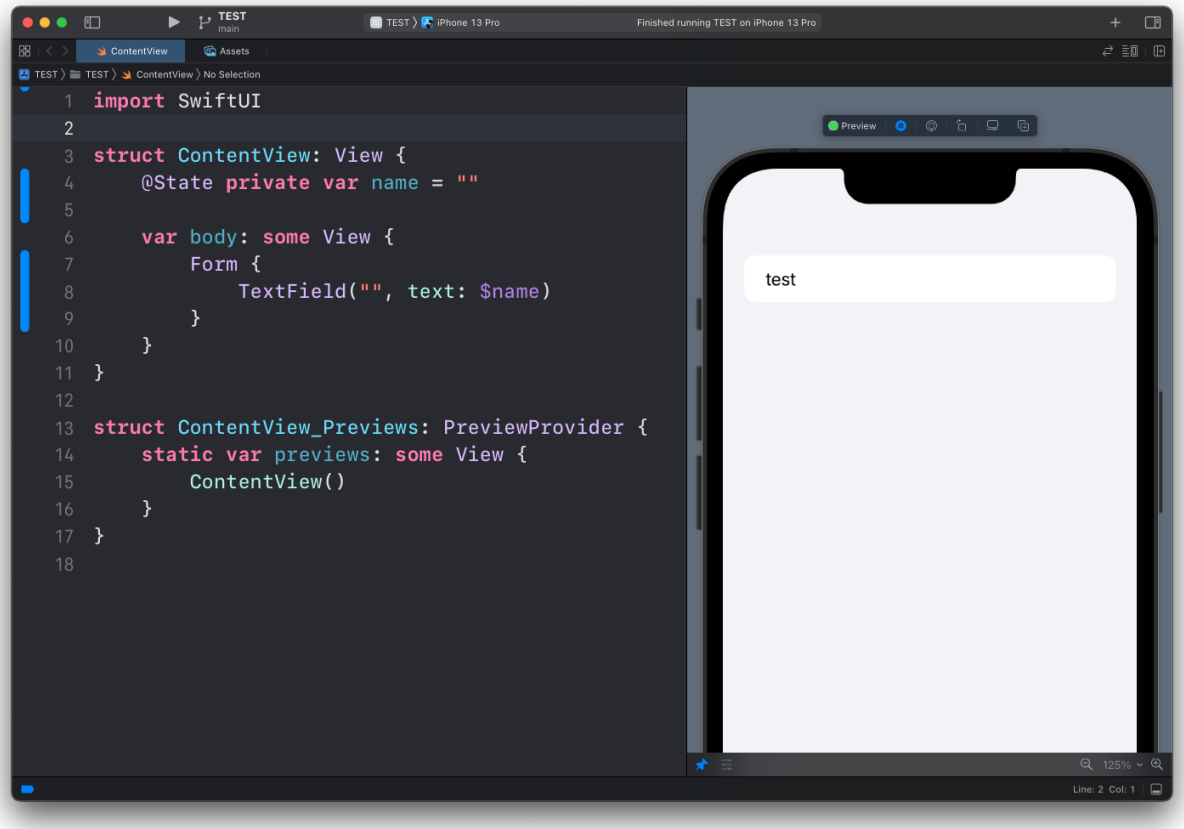
```
struct ContentView: View {
    @State private var name = ""

    var body: some View {
        Form {
            TextField("", text: $name)
        }
    }
}
```

The `name` property is a SwiftUI property wrapped with `@State`, so the view can update it. The `$` prefix passes a binding: the text field writes into `name` every time the user types a character.

Run the code. You can see an empty text field. You can tap on it:

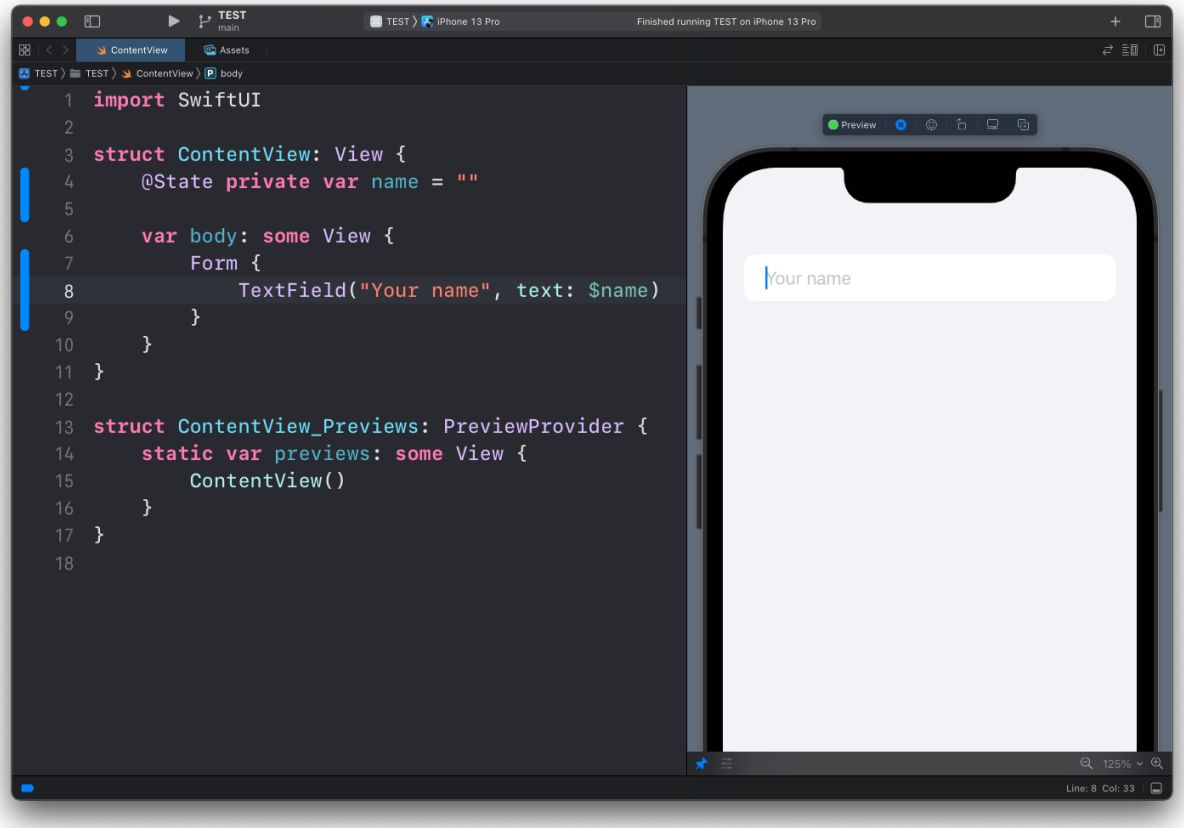


And you can enter any text inside it:



The first argument of `TextField` is the placeholder, a string visualized when the field is empty. Use it to tell the user what goes in the field:

```
TextField("Your name", text: $name)
```



Styling

Inside a `Form`, the row styling comes for free. Outside of one, a text field has no visible border, which looks odd. Add one with `textFieldStyle()`:

```
TextField("Your name", text: $name)
    .textFieldStyle(.roundedBorder)
    .padding()
```

Configuring the keyboard

For specific kinds of input you want the right keyboard, and you want iOS to stop ”helping”. An email field is the classic case:

```
TextField("Email", text: $email)
    .keyboardType(.emailAddress)
    .textInputAutocapitalization(.never)
    .autocorrectionDisabled()
```

`keyboardType(.emailAddress)` puts `@` and `.` on the main keyboard. `textInputAutocapitalization(.never)` stops iOS from capitalizing the first letter. `autocorrectionDisabled()` prevents autocorrect from mangling addresses.

Handling submit

When the user presses return, the `onSubmit` closure runs. You can also change what the return key says with `submitLabel()`:

```
TextField("Search", text: $query)
    .submitLabel(.search)
    .onSubmit {
        performSearch()
    }
```

Now the return key reads ”Search”, and pressing it calls your function.

Passwords

For sensitive input use `SecureField`. Same API, but the characters show as dots and the text is kept out of screenshots and screen recordings:

```
SecureField("Password", text: $password)
```

Controlling focus

Sometimes you want the keyboard to appear as soon as a screen shows up, without waiting for a tap. That's what `@FocusState` is for:

```
struct ContentView: View {
    @State private var name = ""
    @FocusState private var isFocused: Bool

    var body: some View {
        Form {
            TextField("Your name", text: $name)
                .focused($isFocused)
        }
        .onAppear {
            isFocused = true
        }
    }
}
```

Setting `isFocused` to `true` in code focuses the field and raises the keyboard. Setting it to `false` dismisses it. Focus becomes just another piece of state.

SwiftUI forms: Toggle

Learn how to use the Toggle control in SwiftUI to get an on or off choice from the user, binding a Bool value to its isOn parameter like the Settings app.

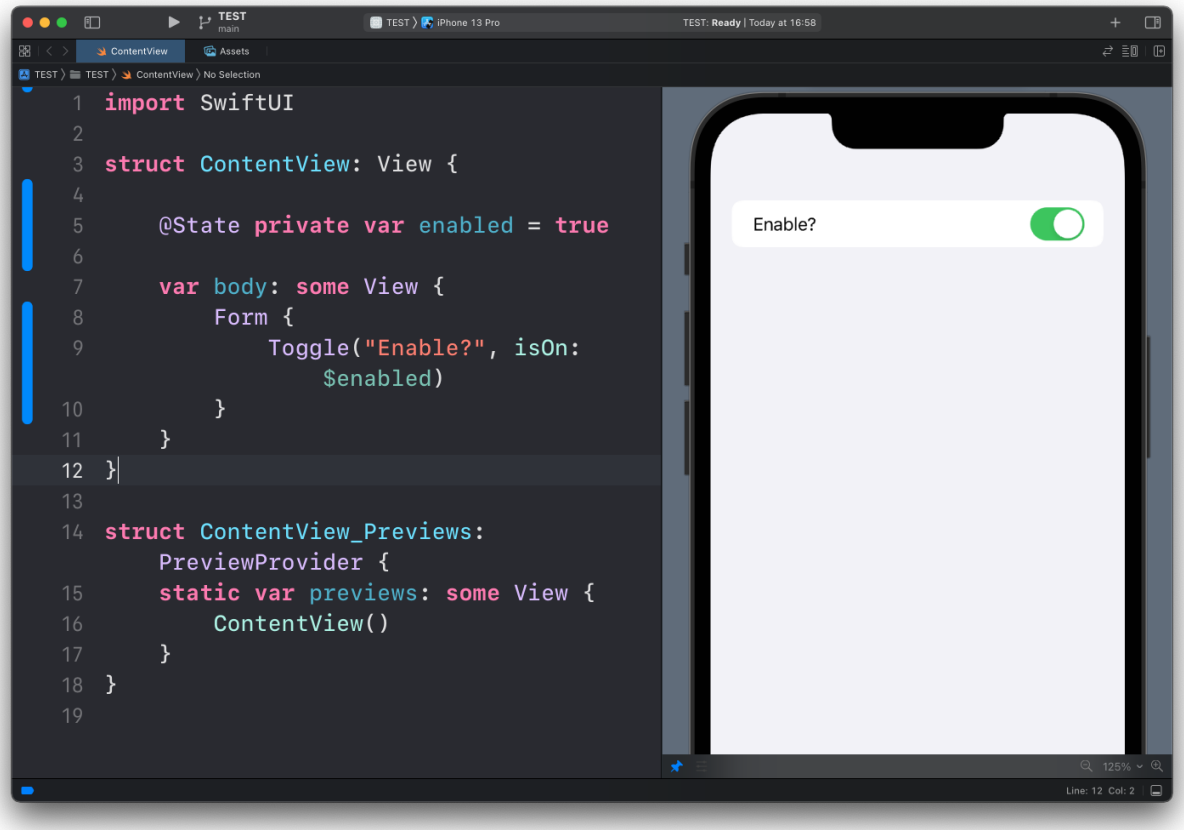
Another common form control is `Toggle`. It gets an on/off choice from the user.

You can see it widely used in the Settings app.

A Toggle needs a `Bool` to work with. We store it in a `@State` property and pass it with the `$` prefix:

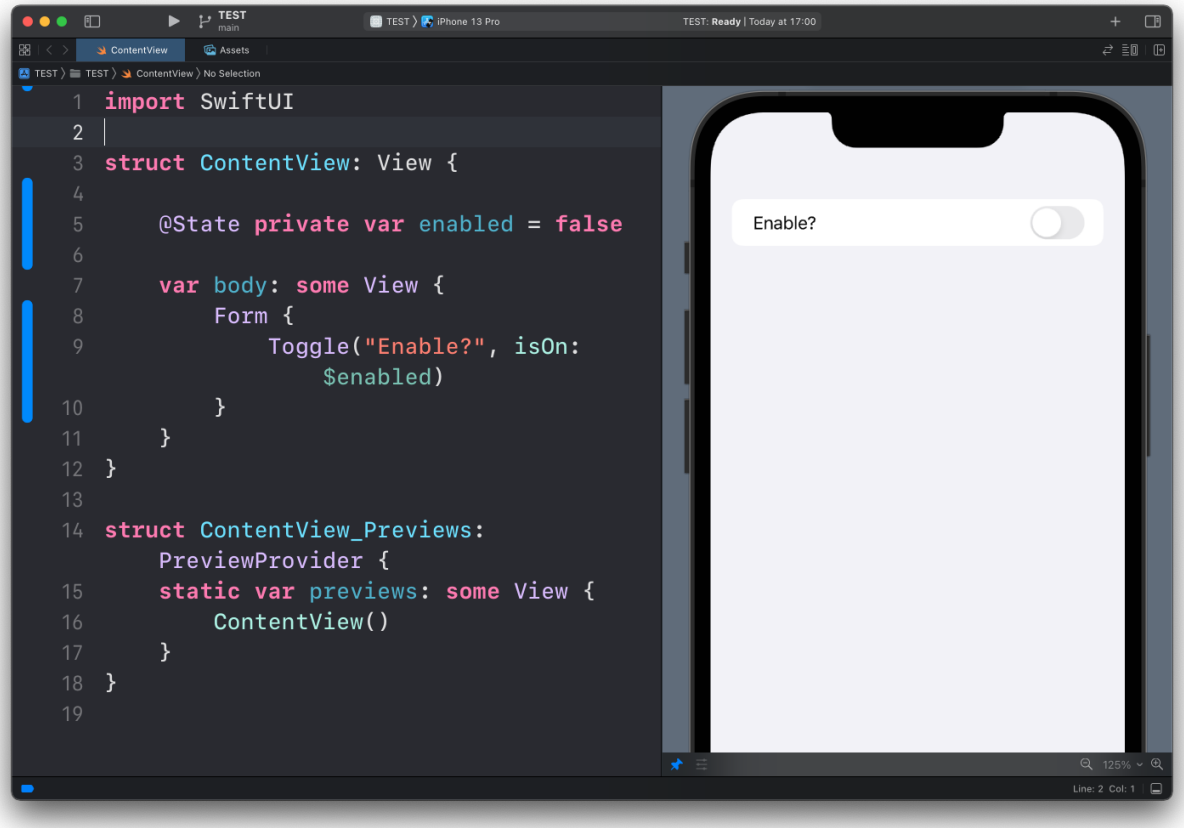
```
struct ContentView: View {
```

```
    @State private var enabled = true

    var body: some View {
        Form {
            Toggle("Enable?", isOn: $enabled)
        }
    }
}
```



It works similarly to a `TextField` view, except instead of a `String` value passed with the `text` parameter, we pass a `Bool` value to `isOn`.

If you set the initial value to `true` the toggle starts enabled, if you set it to `false` it starts disabled:



When the user flips the switch, the bound property updates automatically. When your code changes the property, the switch flips. The binding works in both directions.

Using the value elsewhere

The whole point of binding state is that the rest of the view can react to it. Here's a realistic notifications setting:

```
struct SettingsView: View {
    @State private var notificationsEnabled = false
    @State private var playSounds = true

    var body: some View {
        Form {
            Toggle("Allow notifications", isOn: $notificationsEnabled)

            if notificationsEnabled {
                Toggle("Play sounds", isOn: $playSounds)
            }
        }
    }
}
```

The "Play sounds" row only exists while notifications are on. Flip the first toggle and watch the second row slide in and out. Inside a `Form` or a `List`, SwiftUI animates this insertion for you. Outside those containers, you can request the animation explicitly with `.animation(.default, value: notificationsEnabled)` on the enclosing stack.

Labels with an icon

The label doesn't have to be plain text. Pass a `Label` to get an icon next to it, like the rows in the Settings app:

```
Toggle(isOn: $notificationsEnabled) {
    Label("Notifications", systemImage: "bell.badge")
}
```

Toggle styles

On iOS the default appearance is the switch. You can change it with the `toggleStyle()` modifier.

`.button` turns the toggle into a button that shows a highlighted state when on:

```
Toggle("Bold", isOn: $isBold)
    .toggleStyle(.button)
```

This is great for toolbars and formatting controls, where a switch would look out of place.

On macOS the default inside a form is a checkbox. If you want the iOS-style switch there too, ask for it with `.toggleStyle(.switch)`.

SwiftUI forms: Picker

Learn how to use the Picker control in SwiftUI to choose one option from a list, bind it with selection, and why it needs a NavigationView to work.

After `TextField` and `Toggle`, another common form control is `Picker`. It lets us choose an option among a series of possible ones.

First thing we need to do is to have an array with a list of options:

```
var cities = ["Rome", "Milan", "Venice", "Florence"]
```

Then we need a property to store the selected choice. We wrap it with `@State` as that's something that will change based on the user's input:

```
@State private var selected = "Rome"
```

Finally we use a `Picker` view. We pass 2 parameters. The first is a label, the second is the property used for the selected item, and in the closure we add a Text view for each different option, using a ForEach view:

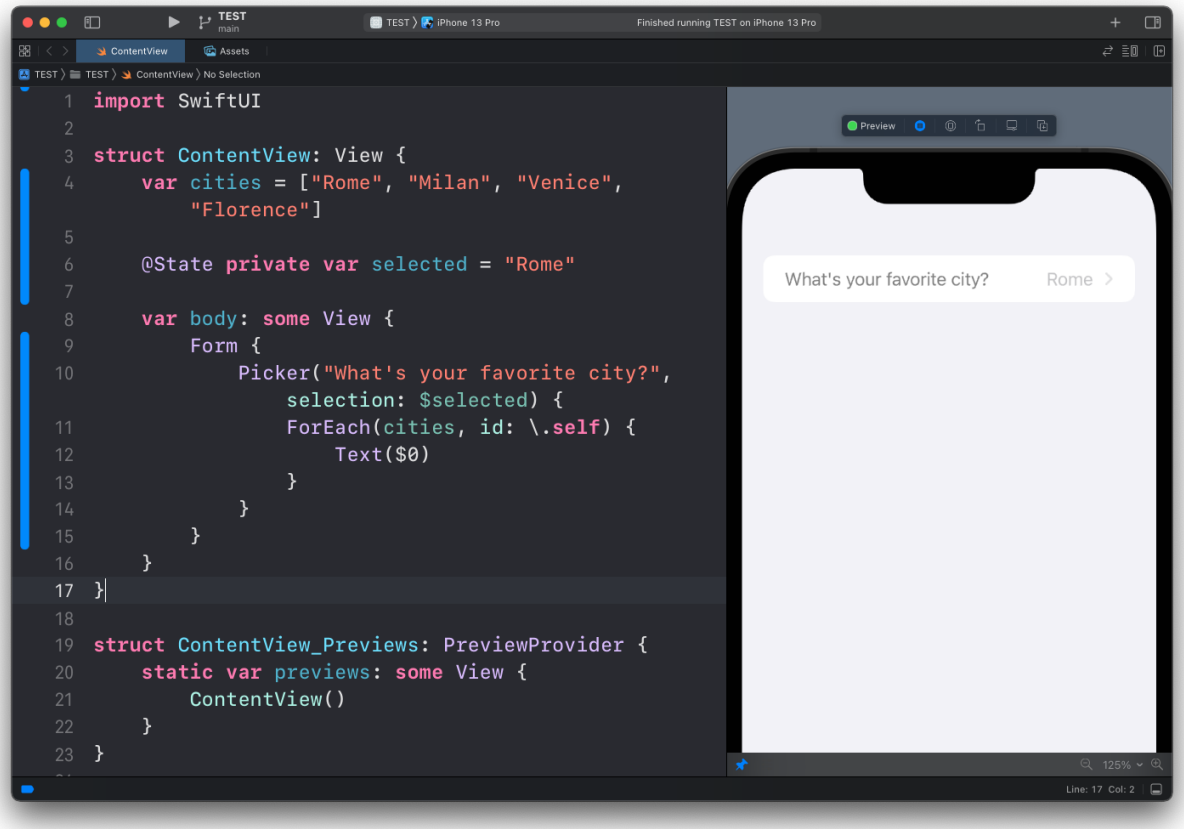
```
Picker("What's your favorite city?", selection: $selected) {
    ForEach(cities, id: \.self) {
        Text($0)
    }
}
```

Here's the full code of our `ContentView`

```
struct ContentView: View {
    var cities = ["Rome", "Milan", "Venice", "Florence"]

    @State private var selected = "Rome"

    var body: some View {
```

```
        Form {
            Picker("What's your favorite city?", selection: $selected) {
                ForEach(cities, id: \.self) {
                    Text($0)
                }
            }
        }
    }
}
```

You can try to run it, and it shows correctly, with the default option visualized:



But even in preview mode, or in the Simulator, you can't tap it.

Why?

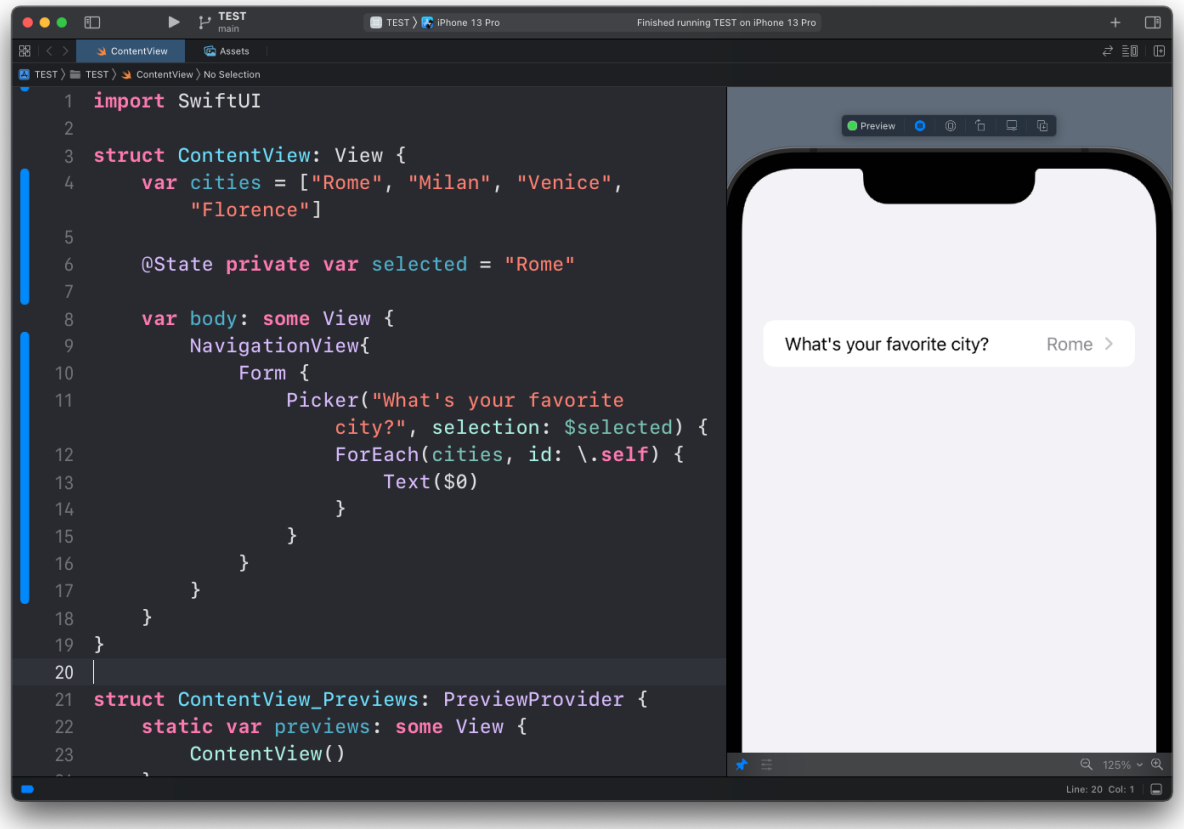
Because you need to wrap it all inside a `NavigationView`:

```
struct ContentView: View {
    var cities = ["Rome", "Milan", "Venice", "Florence"]

    @State private var selected = "Rome"

    var body: some View {
        NavigationView{
            Form {
                Picker("What's your favorite city?", selection: $selected) {
                    ForEach(cities, id: \.self) {
```

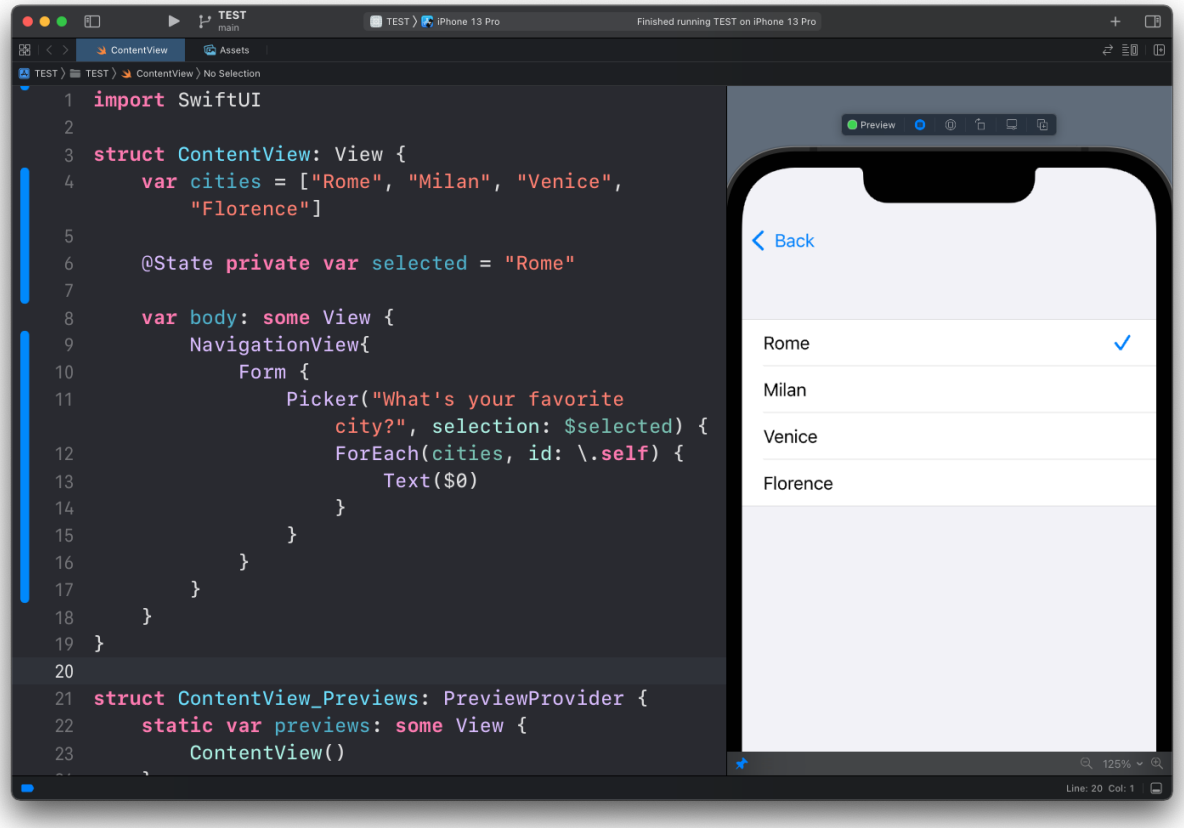
```
                        Text($0)
                    }
                }
            }
        }
    }
}
```

We'll talk about `NavigationView` in another post

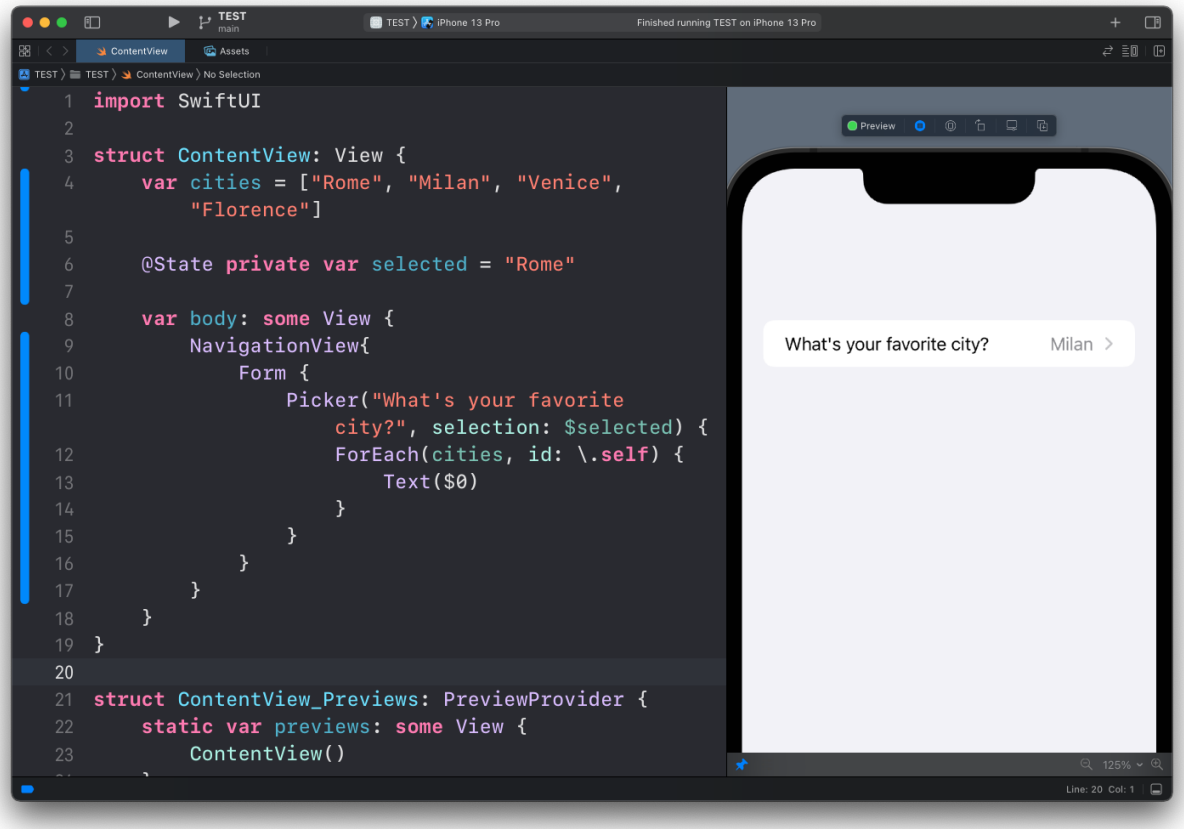
Now running it again, you can see the label turned black instead of gray:



You can tap it, and you'll see the options list, with a navigation link to go back:



Tap one, and you'll see the selected option being visualized instead of the default one:



You can also avoid using an array for the options and use a Text view directly, but you need to use the `tag()` modifier on each view to identify each option:

```
struct ContentView: View {
    @State private var selected = "Rome"

    var body: some View {
        NavigationView {
            Form {
                Picker("What's your favorite city?", selection: $selected) {
                    Text("Rome")
                        .tag("Rome")
                    Text("Milan")
                        .tag("Milan")
                    Text("Venice")
                        .tag("Venice")
                    Text("Florence").tag("Florence")
                }
            }
        }
    }
}
```

Check your understanding: forms and input

Check the key ideas from forms and input before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Richer input and navigation

Use sliders, steppers, date pickers, and modern NavigationStack destinations.

SwiftUI forms: Slider

Learn how to use the Slider control in SwiftUI to let users drag to pick a value, setting its value, in range, and step parameters to control the range.

The `Slider` form control in SwiftUI lets us create a bar that the user can swipe left or right to decrease or increase its value.

We initialize a Slider by setting 3 parameters: `value`, `in`, `step`:

```
@State private var age: Double = 0

//...

Slider(value: $age, in: 0...100, step: 1)
```

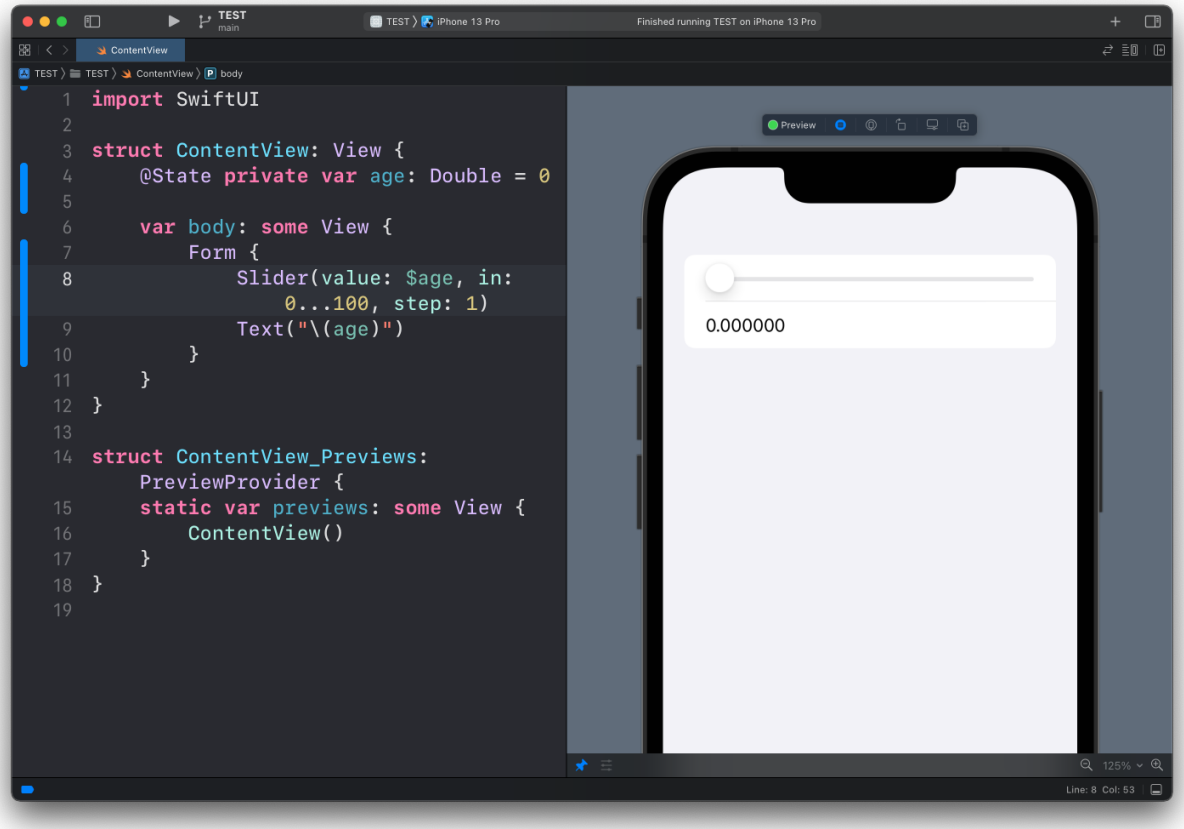
`in` limits the minimum and maximum values we can use.

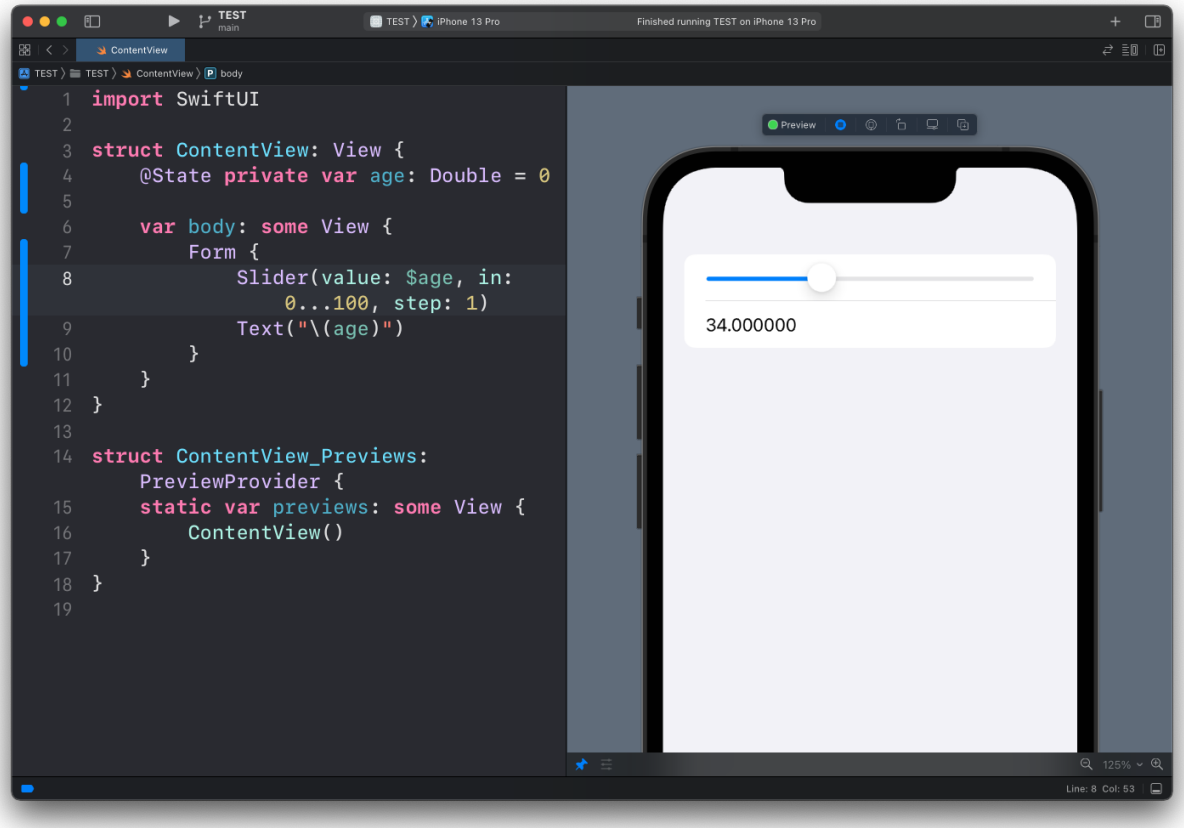
`step` means we can step by a value of 1 at a time, in this case we can go from 0 to 1 to 2 etc. You could use 10, or 0.2, and so on.

Since `Slider` takes a `Double` value, by default we increment the decimals too.

Example:

```
struct ContentView: View {
    @State private var age: Double = 0

    var body: some View {
        Form {
            Slider(value: $age, in: 0...100, step: 1)
            Text("\(age)")
        }
    }
}
```





Notice how I added a `Text` view to show the value of `age`.

Since it's a double, we have lots of decimals.

We could format that, but for this we'll have another post.

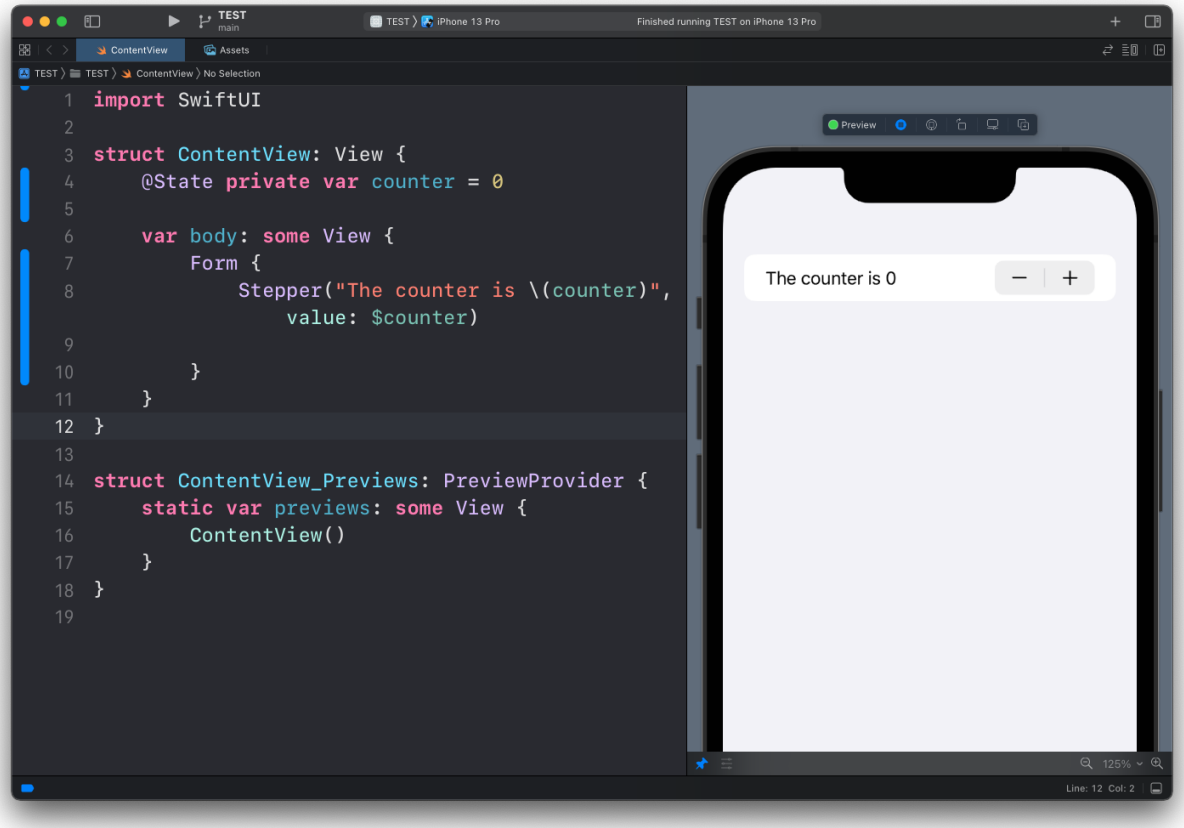
SwiftUI forms: Stepper

Learn how to use the Stepper control in SwiftUI to adjust a number with minus and plus buttons, set a range and step, and format the bound value.

Another useful control we can use in forms is the `Stepper` view, which lets us select a number and gives a `-` and `+` button to decrease or increase it.

We link it to the value of a property with a `@State` property wrapper, in this case `counter`:

```
struct ContentView: View {
    @State private var counter = 0

    var body: some View {
        Form {
            Stepper("The counter is \(counter)", value: $counter)
        }
    }
}
```



Note how the label interpolates the current value. The stepper itself never displays the number, so showing it in the label is the usual pattern.

Limiting the range

You can use the `in` parameter of `Stepper` to limit the range of values it can accept:

```
Stepper("The counter is \(counter)", value: $counter, in: 0...10)
```

When you reach a limit, the button to increase or decrease turns gray and stops responding.

My advice is to always set a range. An unbounded stepper rarely makes sense. Nobody orders minus three coffees.

Stepping by more than 1

By default each tap changes the value by 1. The `step` parameter changes that:

```
Stepper("Font size: \(fontSize)", value: $fontSize, in: 8...72, step: 2)
```

Now each tap moves the value by 2, staying inside the range.

Formatting the value

The bound value can be any numeric type, including `Double`. Interpolating a Double directly prints something like `20.500000`, so format it in the label:

```
@State private var temperature = 20.5

Stepper("Target: \(temperature, specifier: "%.1f")°", value: $temperature, in: 15...30, step:
```

The `specifier` keeps the display at one decimal place while the underlying value stays precise.

Reacting while the user edits

`Stepper` accepts an `onEditingChanged` closure. It receives `true` when an editing session starts and `false` when it ends, which also covers press-and-hold, where the value auto-repeats:

```
Stepper("Quantity: \(quantity)", value: $quantity, in: 1...10) { editing in
    print(editing ? "editing started" : "editing ended")
}
```

You rarely need it, but it's handy to defer expensive work, like saving, until the user is done tapping.

Stepper or Slider?

Use a stepper when the value is a small integer and precision matters: number of guests, item quantity, font size. One tap, one exact increment.

Use a slider for continuous ranges where roughly right is fine, like volume or brightness. A slider covering 1 to 5 feels clumsy, and a stepper covering 0 to 100 means a lot of tapping. Pick the control that matches the size and precision of the range.

SwiftUI forms: DatePicker

Learn how to use the DatePicker control in SwiftUI to let users pick a date and time, and how displayedComponents limits it to just the date or time.

The `DatePicker` form control in SwiftUI lets us create a .. date picker.

How does it work?

First we create a property of type `Date`:

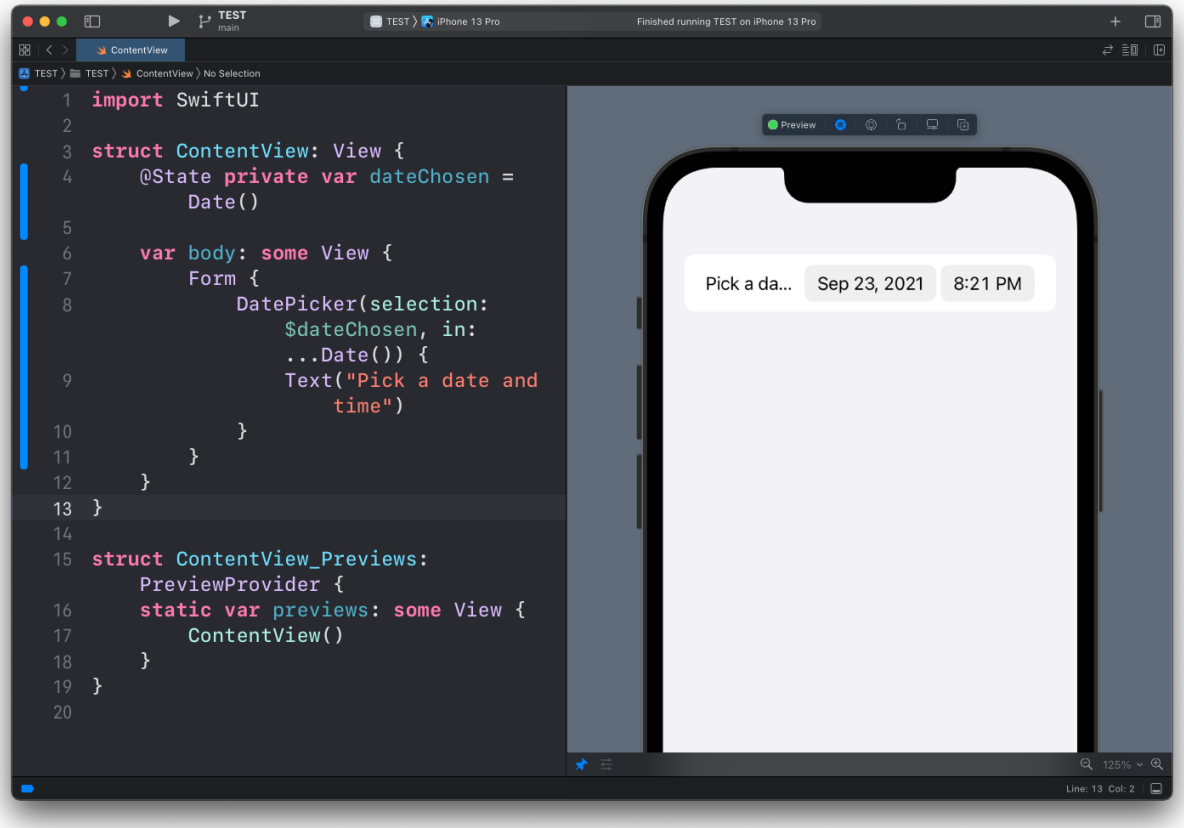
```
@State private var dateChosen = Date()
```

> We use @State so that we can modify this value from our `DatePicker` view

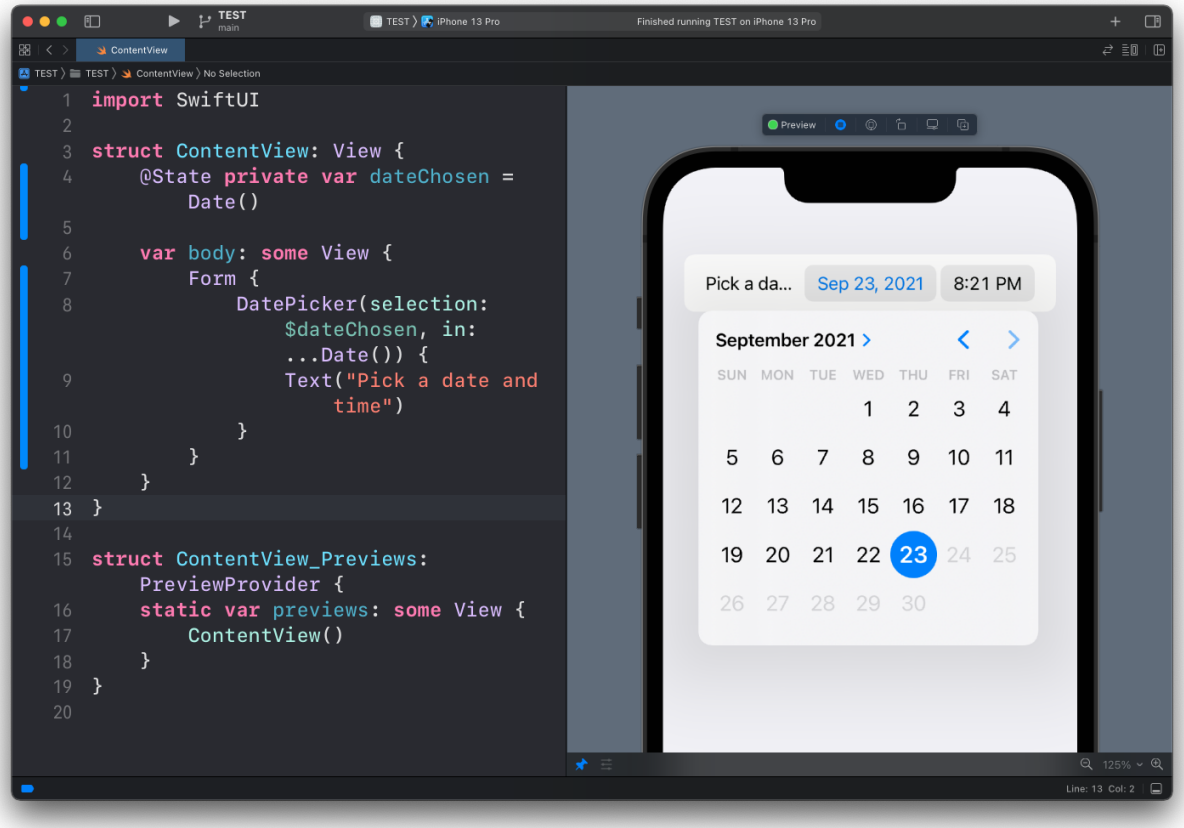
Then we link that property to the `DatePicker` view:

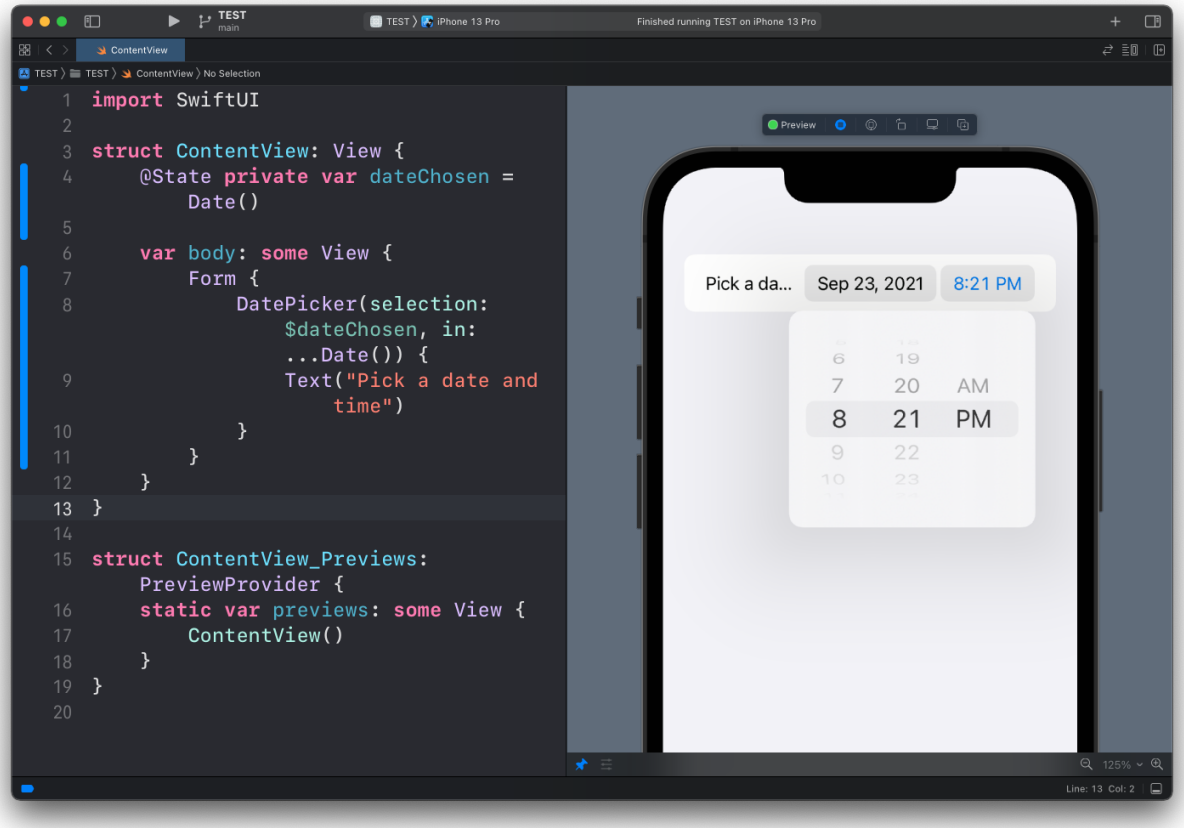
```
DatePicker(selection: $dateChosen, in: ...Date()) {
    Text("Pick a date and time")
}
```

Here's how it looks:



Tapping on each different part (date or time) will show a dedicate picker UI element:



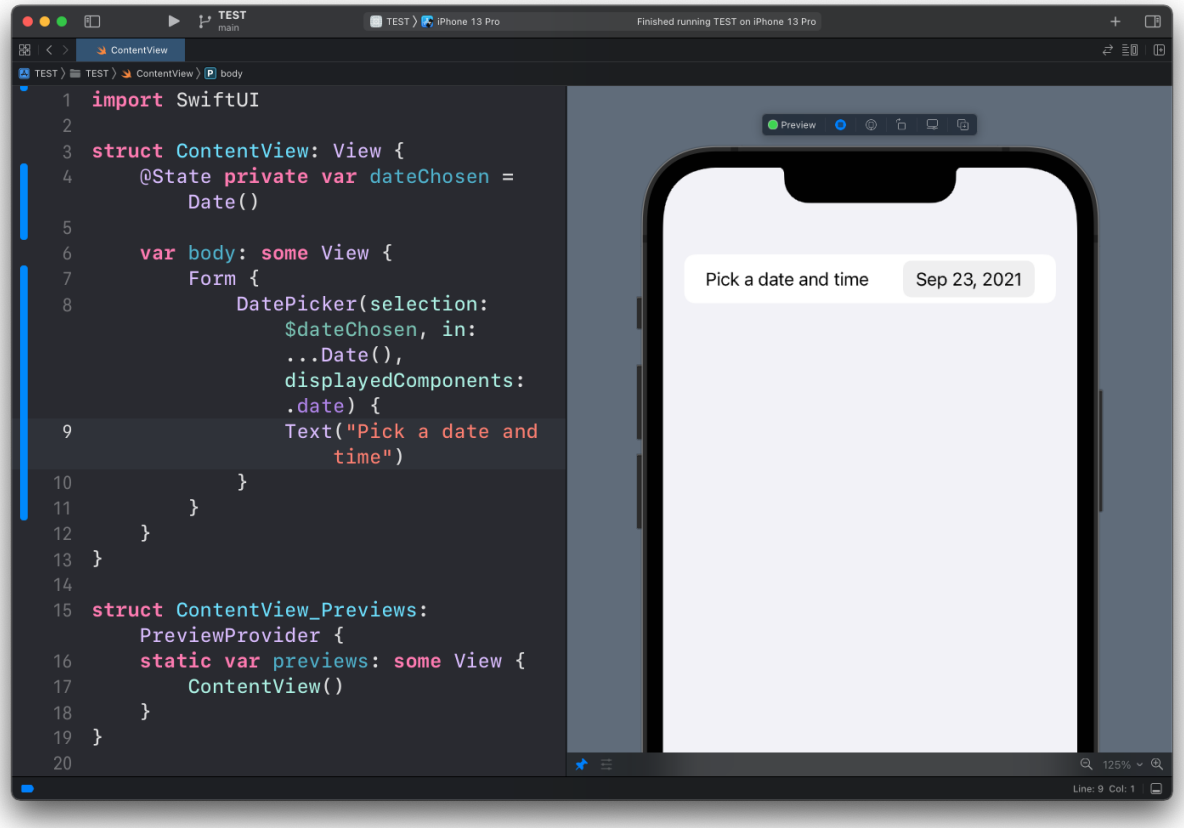


Here's the full code of this example:

```
struct ContentView: View {
    @State private var dateChosen = Date()

    var body: some View {
        Form {
            DatePicker(selection: $dateChosen, in: ...Date()) {
                Text("Pick a date and time")
            }
        }
    }
}
```

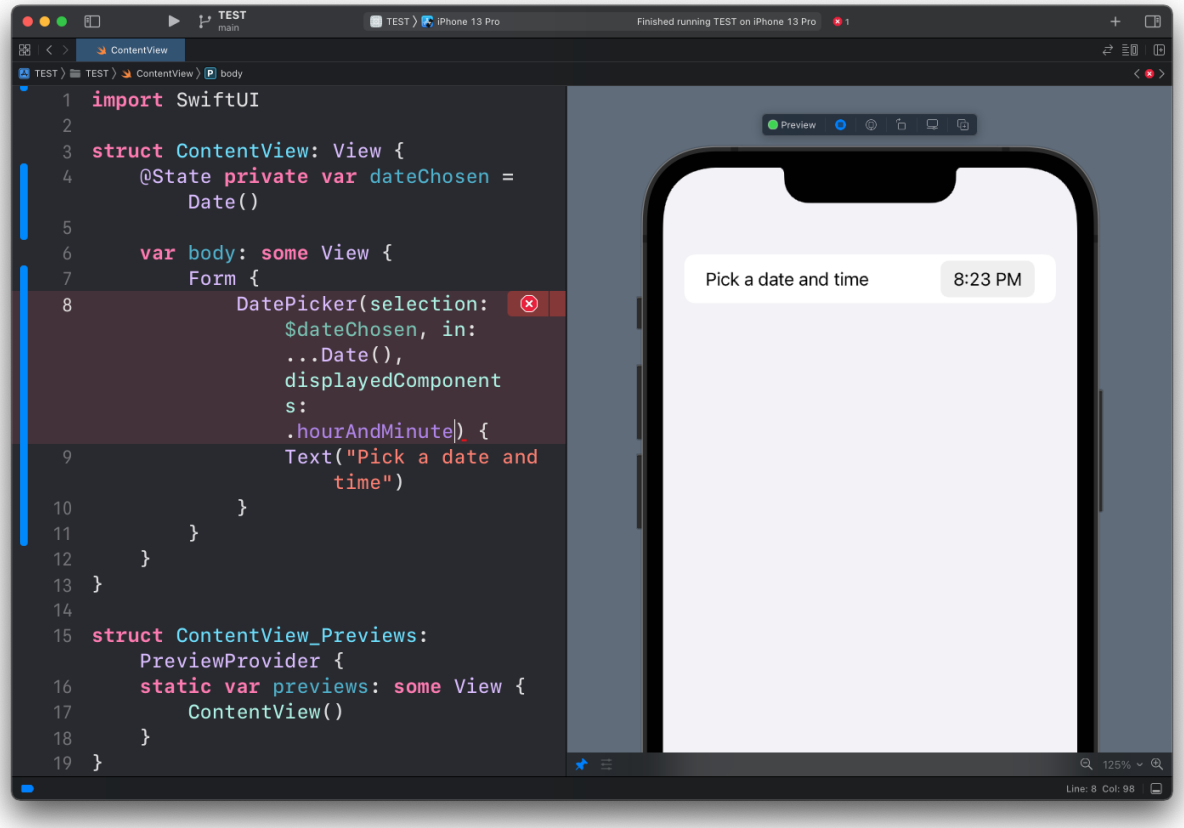
You can choose to only show one particular element of the date with the `displayedComponents` property, like just the date:

```
DatePicker(selection: $dateChosen, in: ...Date(), displayedComponents: .date) {
    Text("Pick a date and time")
}
```



or just the time:

```
DatePicker(selection: $dateChosen, in: ...Date(), displayedComponents: .hourAndMinute) {
    Text("Pick a date and time")
}
```



Navigate with NavigationStack

Present destinations from stable data and keep navigation state explicit when the app needs to inspect or restore it.

`NavigationView` appears in older SwiftUI tutorials, but Apple deprecated it. New code should use `NavigationStack` for one-column navigation. The change is not cosmetic: `NavigationView` hid its navigation state, so you could not inspect it, restore it, or drive it from code without hacks. `NavigationStack` makes that state an explicit value your app can own.

The simplest stack owns its navigation internally. You describe destinations as data, and tell the stack how to turn each data type into a screen:

```
NavigationStack {
  List(notes) { note in
    NavigationLink(note.title, value: note.id)
  }
  .navigationDestination(for: UUID.self) { id in
    NoteDetail(id: id)
  }
}
```

Tapping a link pushes its value onto the navigation path. The `navigationDestination(for:)` modifier matches values by type: every `UUID` pushed on this stack becomes a `NoteDetail`. This split between ”what was selected” and ”how it is presented” is what makes the API composable — links stay tiny, and destinations live in one place.

When the app needs programmatic navigation, bind the path to state:

```
struct NotesRoot: View {
  @State private var path: [UUID] = []

  var body: some View {
    NavigationStack(path: $path) {
      List(notes) { note in
        NavigationLink(note.title, value: note.id)
      }
      .navigationDestination(for: UUID.self) { id in
        NoteDetail(id: id)
      }
    }
  }
}
```

Now navigation is just data. Appending an id pushes a screen: `path.append(note.id)`. Setting `path = []` pops back to the root. A deep link handler can build the whole array in one assignment and the stack renders the full hierarchy, back button included. If you need to push values of different types onto the same stack, use `NavigationPath` instead of a typed array — it accepts any `Hashable` value.

Push stable, lightweight identifiers instead of whole model objects. A common mistake is pushing a full `Note` struct into the path. It works at first, but the pushed copy goes stale the moment the model changes, and restoring state or handling a deep link means reconstructing entire objects instead of writing an id. Let the destination view resolve the current model from the identifier it receives.

One pointer for later: on iPad and Mac, apps with a sidebar and a detail area should reach for `NavigationSplitView` instead. It manages the columns, and its detail column can still contain a `NavigationStack` of its own.

Check your understanding: richer input and navigation

Check the key ideas from richer input and navigation before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/swiftui/.