UI Design for iOS App Development

Using SwiftUI

—

Bear Cahill

Apress®

UI Design for iOS App Development

Using SwiftUI

Bear Cahill

Apress®

UI Design for iOS App Development: Using SwiftUI

Bear Cahill
Denton, TX, USA

ISBN-13 (pbk): 978-1-4842-6448-5 ISBN-13 (electronic): 978-1-4842-6449-2
https://doi.org/10.1007/978-1-4842-6449-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 NY Plazar, New York, NY 10014. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6448-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Bear Cahill has been a developer since he was 12. After getting his B.S. in Computer Science, he worked at several companies before going freelance as an iOS developer. Bear has written multiple books on software development, teaches for several corporate education companies, and develops online courses for Lynda.com/LinkedIn Learning. Ultimately, however, Bear loves to code.

About the Technical Reviewer

Felipe Laso is a Senior Systems Engineer working at Lextech Global Services. He's also an aspiring game designer/programmer. You can follow him on Twitter @iFeliLM or on his blog.

CHAPTER 1

Introducing SwiftUI

First, thank you for reading at least this much of the first chapter. It's tempting to skip it. However, I'm the type of person that reads the foreword, the preface, and so on. Someone thought it important enough to write and include it; maybe it's worth it.

When learning a new IDE, language, or user interface design tool, it can be hard to know where to start. I'll say this: if you don't know Swift, learning SwiftUI will be very tough. In fact, if you don't know about Xcode, iOS development, and the various frameworks related to it, learning SwiftUI isn't the best place to start (see Figure 1-1).

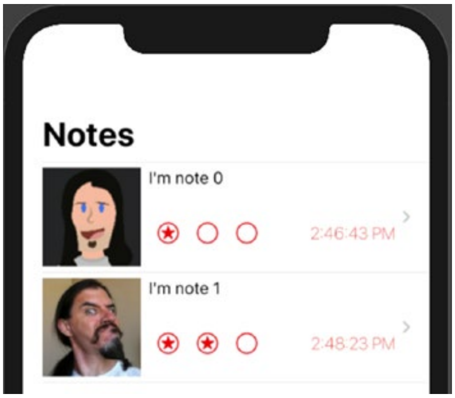


Figure 1-1. *SwiftUI Interface Example*

This isn't a book on Swift, Xcode, iOS frameworks, or UIKit. Being familiar with those is important if not required.

© Bear Cahill 2021
B. Cahill, *UI Design for iOS App Development*,
https://doi.org/10.1007/978-1-4842-6449-2_1

Exercises

I've included one or more exercises per chapter. Some are shorter and some are longer. In each case, there is also one or more End of Chapter (EOC) zip file of the code for you to review.

Many chapters build on the same code throughout the chapter. So there's only one EOC file with the full result.

The point of each exercise is practice. I want you to go through the process of employing what you're learning. I highly recommend experimenting with variations of the exercises as your curiosity prompts you.

I also strongly encourage repetition. If you repeat an exercise a handful of times to the point that you can just knock it out with familiarity, you'll be better off when you're done with this book.

Concepts

Much of SwiftUI will feel like Swift. That's good if you know Swift. You'll feel somewhat comfortable passing closures, chaining calls, handling optionals, and so on.

However, SwiftUI has a very state-driven concept to the UI. The user interface is a display of the state. If a value (the state) changes, the UI should reflect that so it needs to render again.

If the value displayed in a TextField is updated, the interface should display the new value. This is done automatically in SwiftUI with binding. We'll use property wrappers (similar to how Optional is a type with a wrapped value) to pass these values into controls like the TextField.

The TextField will be updated if the value changes. But also changes in the TextField will be stored in the same place as the item passed in. No more getting the text property and storing – SwiftUI cuts out the middle step and just changes the property!

Source of Truth

The concept of these property wrappers is tied to the idea of the "source of truth." If we have the username or email address stored in a property, that can be the source of truth. If the property changes, the UI is updated. If the user types in a new value, it's stored in that same property.

There are different ways of using this concept on value types (e.g., structs) vs. reference types (e.g., classes). We'll explore these in detail in this book.

Old Friends

We'll also look at how to use an existing UI in a SwiftUI-based app. You may have some existing code that works great, and you want to reuse it. No sense in throwing it away if it's still good.

Or you may just not have time to re-create the whole UI in one effort.

New Friends

Of course, we'll look at developing interface designs in SwiftUI. But we'll also look at how to use SwiftUI in Storyboard projects. You may want to migrate to SwiftUI starting in your current UIKit app.

However you decide or need to start using SwiftUI, I hope this book helps get you there.

Combine

If you haven't used the Combine framework yet, you will in this book. This is not a book on Combine, but parts of it are tightly integrated in things we need to do in SwiftUI.

There are Combine aspects sprinkled throughout this book. There's also a chapter specifically intended to go a little deeper into Combine. That framework probably deserves its own book, but we'll dig a bit deeper at times to understand what we're doing and using.

It's All Good

As mentioned, the code is the UI. It doesn't get stored as XML or something, for the UI to get generated from.

But also, the Canvas is the simulator. When you go into Live mode, it's effectively the same as the simulator. It's not perfect, but you can be sure it's close. Also, it's much more than just viewing a rendering of how it's designed without the underlying code (like the Storyboard Preview).

You can even design your preview to display for various color schemes, devices, and so on (see Figure 1-2).

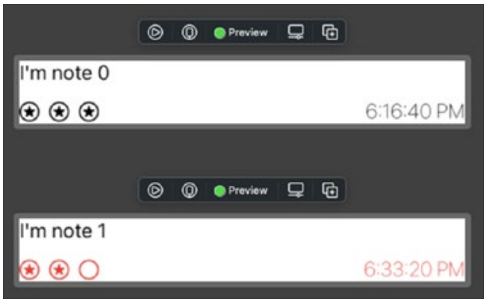


Figure 1-2. *Multiple Previews of One Element*

The key thing for me here is that we need to rethink how we think of the user interface. Instead of creating items with attributes, we call modifiers on those items. They in turn return items, and we repeat as we chain the calls together.

Our UI is tied to our state, and they stay in sync. Changes to the state update the UI. Changes in the UI update the state.

If you're not careful, that may mean everything is tightly coupled. But we're going to break things down and use a lot of functionality built into SwiftUI. In the end, we'll see that many aspects of the interface work the same. So in the past, what took several building blocks can now be done with one or two.

Platforms

We'll be focusing on iOS development with SwiftUI. However, in many cases, the code is the same for the Apple Watch, macOS, iPadOS, Apple TV, and who knows what's coming.

We'll look at a couple examples of the UI from iOS copied into a watch project. The changes will be minimal to get it to work. SwiftUI is a bit more abstracted. Tell it to render a Picker, and it will figure out what that means given the platform.

Let's Get to Codin'

Hazzah!

CHAPTER 2

Take It Easy

In this chapter, we'll ease into SwiftUI by seeing it in action. If it feels slow, good! The beginning is the only time to lay a foundation and that needs to be solid. Rock solid. Like math, a spoken language, or many other skills, if we don't get this down now, we'll be lost later.

Code + UI

If you've done UI development in Xcode in the past, you know that combining the UI design and code is possible. However, they aren't hot swappable. You don't change a background color to red in the code and then open Interface Builder and see that change. At least, now without some special coding.

With SwiftUI, you can think of the code and the UI as one thing. Effectively, it is. In the past, the UI was translated into XML. That wasn't very readable nor easy to edit correctly. Now the UI is generated from the SwiftUI code. As you make changes to the code, the preview is updated to show the changes.

Moreover, if you modify the UI in the preview canvas, it updates the code. Let's look at an example starting with a new project.

B. Cahill, *UI Design for iOS App Development*,
https://doi.org/10.1007/978-1-4842-6449-2_2

YOUR FIRST SWIFTUI APP

We're going to start by creating a project from a template, analyzing what's created, and changing the UI for our purposes.

1. Open Xcode and start a new project (Figure 2-1).

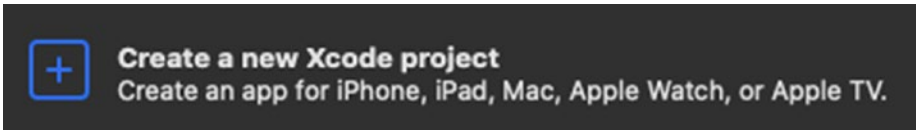

Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.

Figure 2-1. *Create New Project Option in Xcode*

   If Xcode is already running, select File ➤ New… ➤ Project (⇧⌘N).

2. Select the iOS App template and click Next (Figure 2-2).

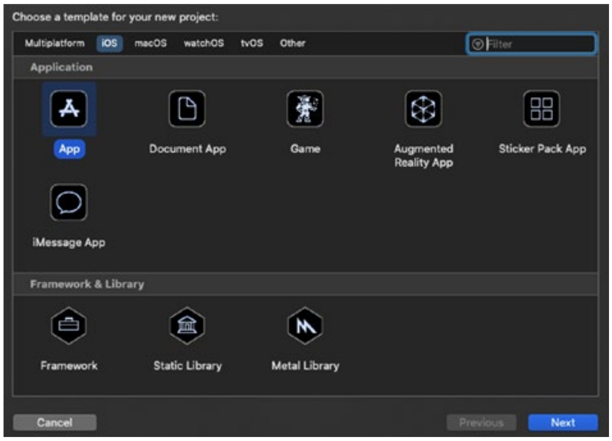
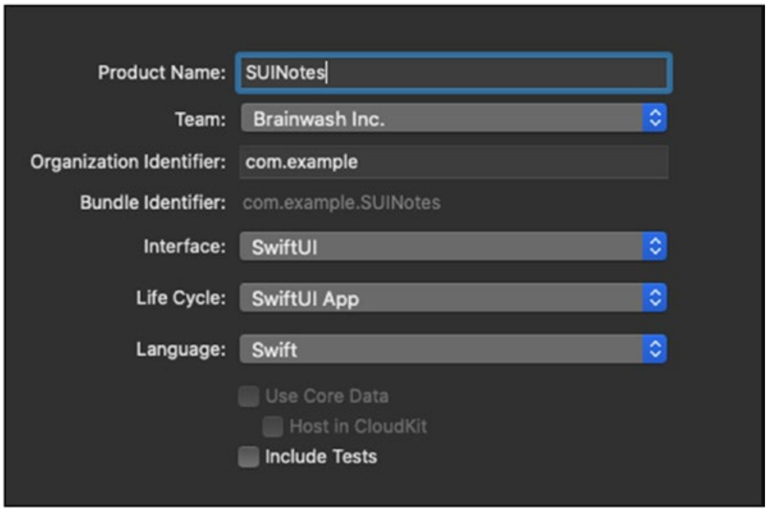

Choose a template for your new project:

Figure 2-2. *iOS App Template*

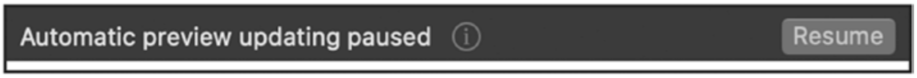
3. Set your product name and other details including the Language (Swift), User Interface (SwiftUI), and Life Cycle (SwiftUI App) (Figure 2-3).



Product Name: SUINotes
Team: Brainwash Inc.
Organization Identifier: com.example
Bundle Identifier: com.example.SUINotes
Interface: SwiftUI
Life Cycle: SwiftUI App
Language: Swift

Figure 2-3. *Project Options*

4. When your project is created, preview updating may be paused. If so, click the Resume button (Figure 2-4).



Automatic preview updating paused

Figure 2-4. *Resume Automatic Preview*

Once the preview updates, ytou'll see your first SwiftUI. Congrats. No one is more proud of you than me.

Hello SwiftUI

Of course, this is the typical "Hello World" example. Let's look just briefly at what we have line by line.

We're importing SwiftUI on line 9. That's new. That's where the various SwiftUI items are defined obviously.

On line 11 is the first line of code for our new app. We have a struct called ContentView which implements a protocol named View. We can see the definition of view with ^⌘ + click "View."

So anything implementing View needs to have a property called "body." That property must have a getter that returns the type specified by the associatedtype of the Body: something that implements View.

Back in our code, we see that ContentView implements View. The return type is "some View," and the body of that computed body property is a Text.

Note We'll get into the "some View" and opaque types later. For now, just know that whatever is returned from our body computed property must implement View.

You've probably already guessed that Text is like a label. We create it with a String, and there it is on the UI.

Modifiers

As with other methods of UI development, SwiftUI elements can be modified. Text has modifiers like font, color, alignment, and so on.

We can add a modifier to our Text element to make the font red like this (Figure 2-5):

```
Text("Hello, World!")
       .foregroundColor(.red)
```

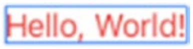


Figure 2-5. *Text with Red Foreground*

And you'll notice the UI updates in the preview.

Similarly, we can bold our text by chaining another modifier. We may want to split these onto multiple lines for the sake of cleanliness (Figure 2-6).

```
Text("Hello, World!")
    .foregroundColor(.red)
    .bold()
```

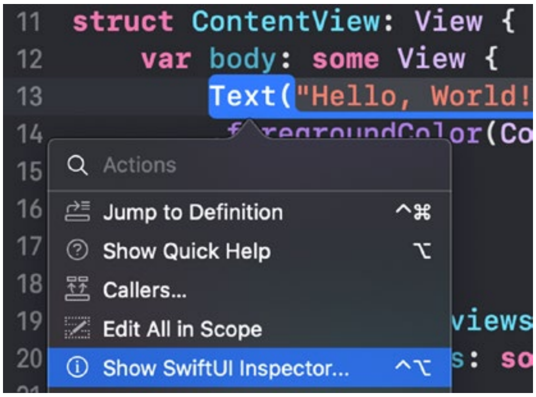
Figure 2-6. *Text with Bold, Red Foreground*

As you can imagine, there are many visual variations you can have on a given UI item. That translates to a lot of modifiers with many parameters. It's a lot to learn and remember.

SwiftUI Inspector

Fortunately, you don't have to remember all of the modifiers. Xcode is here to help!

If you ⌘ + **click** the Text item, you'll see a pop-up menu. From that, select "Show SwiftUI Inspector..." (Figure 2-7).



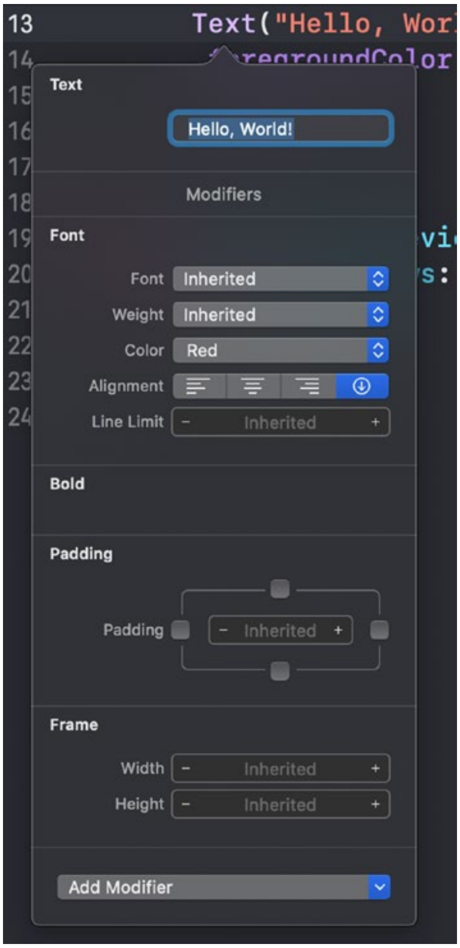
```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!
            .foregroundColor(Co
```

Figure 2-7. *SwiftUI Inspector Menu Item*

Note You can alternatively ^⌥ **+ click** the Text and go right to the SwiftUI Inspector.

Once the SwiftUI Inspector is displayed, we see that there are a variety of attributes we can set with modifiers.

Not only that, but we can add modifiers with the drop-down at the bottom (Figure 2-8).



```
Text("Hello, Wor
  regroundColor
vi
s:
```

Text: Hello, World!

Font: Inherited
Weight: Inherited
Color: Red
Line Limit: Inherited
Padding: Inherited
Width: Inherited
Height: Inherited

Figure 2-8. *SwiftUI Inspector*

As you make selections on these controls for the modifiers, your code is updated to reflect the choices you make.

Note the empty section titled Bold. That's there because we manually put in that modifier, but there are no settings for it.

The better way to bold the text would be through modifiers in the Font section of the inspector.

Exercise time!

MODIFY WITH MODIFIERS

In this exercise, I want you to use modifiers in the SwiftUI Inspector to get your "Hello World" text to match the following example. Give it a shot on your own before reviewing the steps. Also, remove the .bold() line of code to have a better starting place.

```
Text("Hello, World!")
  .foregroundColor(Color.red)
```

Here's what I want your text to look like by only using the SwiftUI Inspector (Figure 2-9).

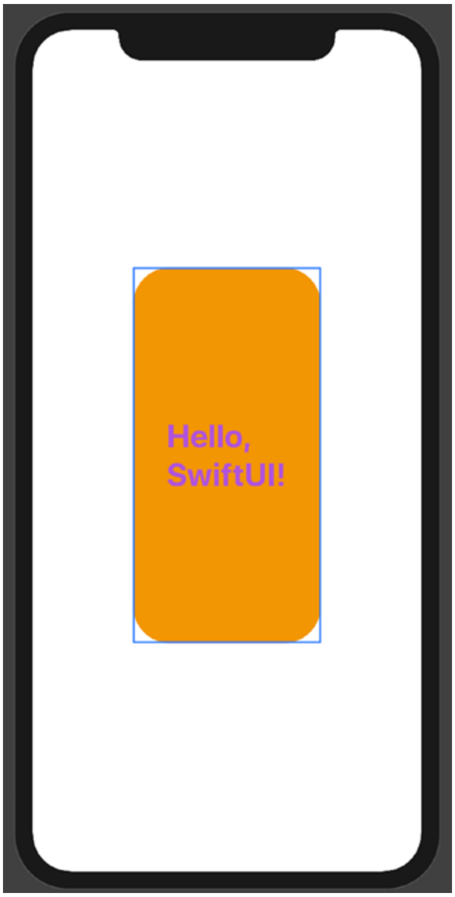


Figure 2-9. *Updated UI Preview*

You can easily see several things have changed:

1. Text (“Hello, SwiftUI!”)
2. Font color
3. Font size
4. Font weight
5. Spacing of the words

6. Size of the Text
7. Background color
8. Corner radius

Feel free to play around with the various modifiers and settings of their values. Practice with these types of things will make you more comfortable and natural in making these changes.

Ideally, you'll get to where you don't have to stop, think, wonder, and search for the right settings.

Here's the resulting settings in the SwiftUI Inspector I used to get the Text item the way I wanted it:

I made three changes in the Font area: the font itself, the weight, and the color.

I changed the padding to 30 (you can edit the number directly by clicking it).

I also changed the frame's width and height manually to 200 and 400, respectively.

I used the Add Modifier drop-down list to add two other modifiers: Background and Corner Radius. Once added, I was able to set their values.

Here's what the code looks like after these changes:

```
Text("Hello, SwiftUI!")
    .font(.largeTitle)
    .fontWeight(.bold)
    .foregroundColor(Color.purple)
    .padding(30.0)
    .frame(width: 200.0, height: 400.0)
    .background(Color.orange)
    .cornerRadius(40.0)
```

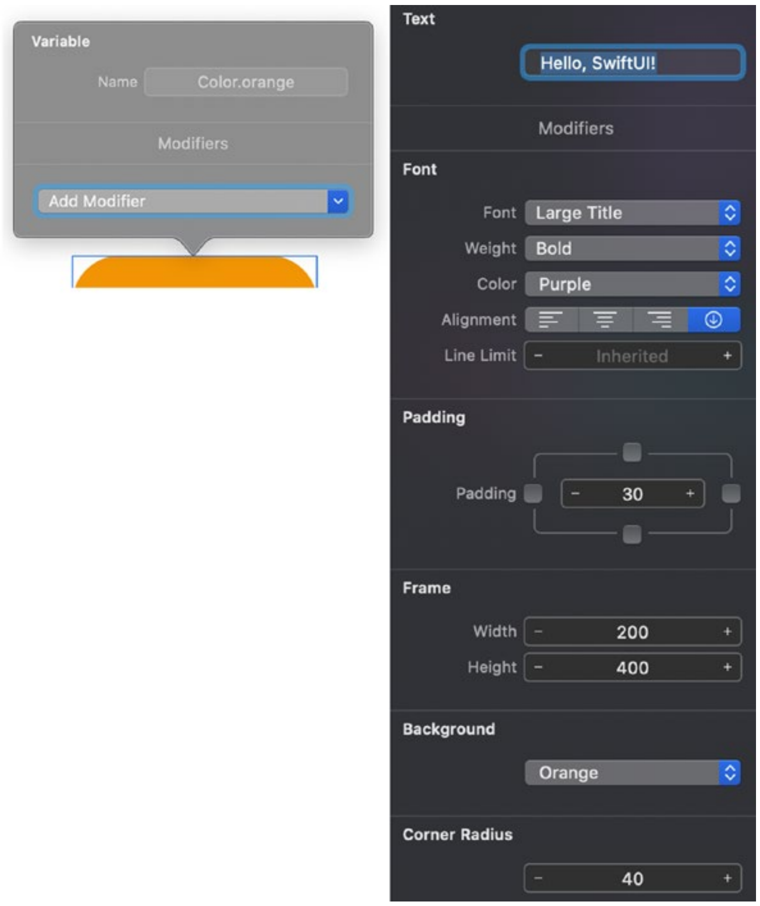
Again, please get comfortable using these controls and associating the changes with the code and UI. Next, we'll look at some other ways to do these tasks.

Attributes Inspector

You can ^⌥ + **click** the UI item like you can on the Text in the code. However, the SwiftUI Inspector may not look the same.

Notice, in this case, there's only a couple of options available in the pop-up (Figure 2-10).

To see all of the same modifiers you saw before, the Attributes Inspector is a reliable option.


Variable
Name: Color.orange
Modifiers
Add Modifier

Text
Hello, SwiftUI!
Modifiers
Font
Font: Large Title
Weight: Bold
Color: Purple
Alignment
Line Limit: Inherited
Padding
Padding: 30
Frame
Width: 200
Height: 400
Background
Orange
Corner Radius
40

Figure 2-10. *Pop-up Menu in the Canvas*

ATTRIBUTES INSPECTOR

In this exercise, we'll see that the same modifiers can be accessed via the visual UI design in the Canvas. However, there's no need to shift your mind from "working in code" to "working in the UI." They are the same. The code is the UI.

1. Open the Inspector Pane (on the right) with this button on the top right of the Xcode window (Figure 2-11).

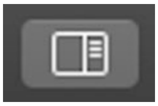


Figure 2-11. *Inspector Pane Button*

2. Select the Attributes Inspector at the top of the Inspector Pane (Figure 2-12).

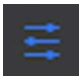
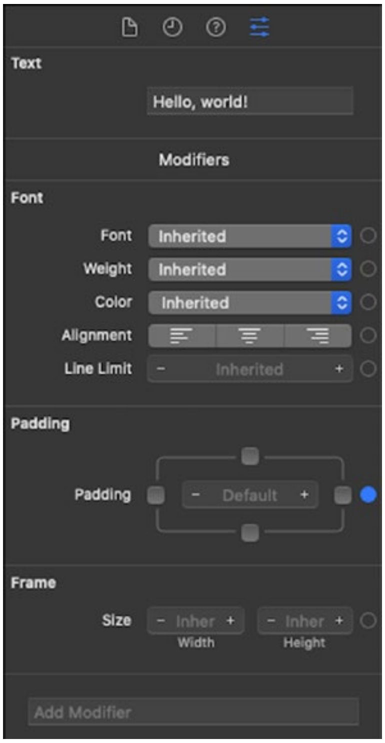


Figure 2-12. *Attributes Inspector Tab*

3. Click the "Hello, SwiftUI!" Text item in the UI or in the code to have the attributes show up in the Attributes Inspector (Figure 2-13).



Text
Hello, world!
Modifiers
Font
Font: Inherited
Weight: Inherited
Color: Inherited
Alignment
Line Limit: Inherited
Padding
Padding: Default
Frame
Size: Inher (Width), Inher (Height)
Add Modifier

Figure 2-13. *Attributes Inspector*

If not all of the modifiers show up, click the item in the code.

Notice that the values are the same as before. Also, the Add Modifier drop-down list is available.

From here, you can make the same choice and changes.

4. Change the background color, text, and other settings and verify the changes in the UI and code.

You probably see that there are various ways to do the same things. You can edit code, make changes in the SwiftUI Inspector and in the Attributes Inspector.

And hopefully you're thinking of the code and the UI as one thing: ideally that the code *is* the UI.

Stacks of Stacks

A screen in an app with only one element isn't much of a user interface. Nor is it common. But there's only one element returned from the computed body property. What are we to do?

Most of the time, the one item we return will contain many other items. So it's a container of other items. And as we'll see, it's often a container of containers of items and so on.

Two common containers we'll see are horizontal stacks (HStack) and vertical stacks (VStack).

Horizontal stacks stack horizontally. You can probably guess that vertical stacks stack vertically. If you're familiar with the stack view from Interface Builder, you're probably already where I'm going.

One easy way to embed an item into a stack is via the pop-up menu from ⌘ **+ click** (see Figure 2-14).