HACKING WITH SWIFT

SWIFTUI EDITION

Learn to make iOS apps with real-wo Swift projects

FREE SAMPLE

Paul Hudson

Chapter 1

WeSplit

Learn the basics of SwiftUI with your first project

WeSplit: Introduction

In this project we're going to be building a check-splitting app that you might use after eating at a restaurant – you enter the cost of your food, select how much of a tip you want to leave, and how many people you're with, and it will tell you how much each person needs to pay.

This project isn't trying to build anything complicated, because its real purpose is to teach you the basics of SwiftUI in a useful way while also giving you a real-world project you can expand on further if you want.

You'll learn the basics of UI design, how let users enter values and select from options, and how to track program state. As this is the first project, we'll be going nice and slow and explaining everything along the way – subsequent projects will slowly increase the speed, but for now we're taking it easy.

This project – like all the projects that involve building a complete app – is broken down into three stages:

1. A hands-on introduction to all the techniques you'll be learning.
2. A step-by-step guide to build the project.
3. Challenges for you to complete on your own, to take the project further.

Each of those are important, so don't try to rush past any of them.

In the first step I'll be teaching you each of the individual new components in isolation, so you can understand how they work individually. There will be lots of code, but also some explanation so you can see how everything works just by itself. This step is an *overview*: here are the things we're going to be using, here is how they work, and here is how you use them.

In the second step we'll be taking those concepts and applying them in a real project. This is where you'll see how things work in practice, but you'll also get more context – here's *why* we want to use them, and here's how they fit together with other components.

In the final step we'll summarizing what you learned, and then you'll be given a short test to

make sure you've really understood what was covered. You'll also be given three challenges: three wholly new tasks that you need to complete yourself, to be sure you're able to apply the skills you've learned. We don't provide solutions for these challenges (so please don't write an email asking for them!), because they are there to test *you* rather than following along with a solution.

Anyway, enough chat: it's time to begin the first project. We're going to look at the techniques required to build our check-sharing app, then use those in a real project.

So, launch Xcode now, and choose Create A New Xcode Project. You'll be shown a list of options, and I'd like you to choose iOS and Single View App, then press Next. On the subsequent screen you need to do the following:

- For Product Name please enter "WeSplit".
- For Organization Identifier you can enter whatever you want, but if you have a website you should enter it with the components reversed: "hackingwithswift.com" would be "com.hackingwithswift". If you don't have a domain, make one up – "me.yourlastname.yourfirstname" is perfectly fine.
- For Language please make sure you have Swift selected.
- For User Interface please select SwiftUI.
- Make sure all the checkboxes at the bottom are *not* checked.

In case you were curious about the organization identifier, you should look at the text just below: "Bundle Identifier". Apple needs to make sure all apps can be identified uniquely, and so it combines the organization identifier – your website domain name in reverse – with the name of the project. So, Apple might have the organization identifier of "com.apple", so Apple's Keynote app might have the bundle identifier "com.apple.keynote".

When you're ready, click Next, then choose somewhere to save your project and click Create. Xcode will think for a second or two, then create your project and open some code ready for you to edit.

Later on we're going to be using this project to build our check-splitting app, but for now

we’re going to use it as a sandbox where we can try out some code.

Let’s get to it!

Understanding the basic structure of a SwiftUI app

When you create a new SwiftUI app, you'll get a selection of files and maybe 100 lines of code in total. Most of the code doesn't do anything, and is just there as placeholders to give you something to fill in – you can safely ignore it for now, but as you progress through this course that will change.

Inside Xcode you should see the following files in the space on the left, which is called the project navigator:

- AppDelegate.swift contains code for managing your app. It used to be common to add code here, but these days it's quite rare.
- SceneDelegate.swift contains code for launching one window in your app. This doesn't do much on iPhone, but on iPad – where users can have multiple instances of your app open at the same time – this is important.
- ContentView.swift contains the initial user interface (UI) for your program, and is where we'll be doing all the work in this project.
- Assets.xcassets is an *asset catalog* – a collection of pictures that you want to use in your app. You can also add colors here, along with app icons, iMessage stickers, and more.
- LaunchScreen.storyboard is a visual editor for creating a small piece of UI to show when your app is launching.
- Info.plist is a collection of special values that describe to the system how your app works – which version it is, which device orientations you support, and more. Things that aren't code, but are still important.
- Preview Content is a yellow group, with Preview Assets.xcassets inside – this is another asset catalog, this time specifically for example images you want to use when you're designing your user interfaces, to give you an idea of how they might look when the program is running.

All our work for this project will take place in ContentView.swift, which Xcode will already

have opened for you. It has some comments at the top – those things marked with two slashes at the start – and they are ignored by Swift, so you can use them to add explanations about how your code works.

Below the comments are ten or so lines of code:

```
import SwiftUI

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

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

Before we start writing our own code, it's worth going over what all that does, because a couple of things will be new.

First, **import SwiftUI** tells Swift that we want to use all the functionality given to us by the SwiftUI framework. Apple provides us with many frameworks for things like machine learning, audio playback, image processing, and more, so rather than assume our program wants to use everything ever we instead say which parts we want to use so they can be loaded.

Second, **struct ContentView: View** creates a new struct called **ContentView**, saying that it conforms to the **View** protocol. **View** comes from SwiftUI, and is the basic protocol that must be adopted by anything you want to draw on the screen – all text, buttons, images, and more are all views, including your own layouts that combine other views.

Third, **var body: some View** defines a new computed property called **body**, which has an interesting type: **some View**. This means it will return something that conforms to the **View** protocol, but that extra **some** keyword adds an important restriction: it must always be the *same* kind of view being returned – you can't sometimes return one type of thing and other times return a different type of thing.

We'll look at this feature more shortly, but for now just remember that it means "one specific sort of view must be sent back from this property."

The **View** protocol has only one requirement, which is that you have a computed property called **body** that returns **some View**. You can (and will) add more properties and methods to your view structs, but **body** is the only thing that's required.

Fourth, **Text("Hello World")** creates a text view using the string "Hello World". Text views are simple pieces of static text that get drawn onto the screen, and will automatically wrap across multiple lines as needed.

Below the **ContentView** struct you'll see a **ContentView_Previews** struct, which conforms to the **PreviewProvider** protocol. This piece of code won't actually form part of your final app that goes to the App Store, but is instead specifically for Xcode to use so it can show a preview of your UI design alongside your code.

These previews use an Xcode feature called the canvas, which is usually visible directly to the right of your code. You can customize the preview code if you want, and they will only affect the way the canvas shows your layouts – it won't change the actual app that gets run.

Xcode can only show the canvas on macOS Catalina or later. If you don't see the canvas and are already running Catalina, go to the Editor menu and select Canvas.

If you don't have Catalina, you'll need to run your code in the Simulator in order to see how it looks.

Tip: Very often you'll find that an error in your code stops Xcode's canvas from updating – you'll see something like "Automatic preview updating paused", and can press Resume to fix

it. As you'll be doing this a lot, let me recommend an important shortcut: Option+Cmd+P does the same as clicking Resume.

Creating a form

Many apps require users to enter some sort of input – it might be asking them to set some preferences, it might be asking them to confirm where they want a car to pick them up, it might be to order food from a menu, or anything similar.

SwiftUI gives us a dedicated view type for this purpose, called **Form**. Forms are scrolling lists of static controls like text and images, but can also include user interactive controls like text fields, toggle switches, buttons, and more.

You can create a basic form just by wrapping the default text view inside **Form**, like this:

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

If you're using Xcode's canvas, you'll see it change quite dramatically: before Hello World was centered on a white screen, but now the screen is a light gray, and Hello World appears in the top left in white.

What you're seeing here is the beginnings of a list of data, just like you'd see in the Settings app. We have one row in our data, which is the Hello World text, but we can add more freely and have them appear in our form immediately:

```
Form {
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
}
```

In fact, you can have as many things inside a form as you want, although if you intend to add more than 10 SwiftUI requires that you place things in groups to avoid problems.

For example, this code shows ten rows of text just fine:

```
Form {
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
}
```

But this attempts to show 11, which is not allowed:

```
Form {
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
    Text("Hello World")
}
```

Tip: In case you were curious why 10 rows are allowed but 11 is not, this is a limitation in

SwiftUI: it was coded to understand how to add one thing to a form, how to add two things to a form, how to add three things, four things, five things, and more, all the way up to 10, but not beyond – they needed to draw a line somewhere. This limit of 10 children inside a parent actually applies everywhere in SwiftUI.

If you wanted to have 11 things inside the form you should put some rows inside a **Group**:

```
Form {
    Group {
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
    }

    Group {
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
        Text("Hello World")
    }
}
```

Groups don't actually change the way your user interface looks, they just let us work around SwiftUI's limitation of ten child views inside a parent – that's text views inside a form, in this instance.

If you *want* your form to look different when split its items into chunks, you should use the **Section** view instead. This splits your form into discrete visual groups, just like the Settings app does:

```
Form {
    Section {
        Text("Hello World")
    }

    Section {
        Text("Hello World")
        Text("Hello World")
    }
}
```

There's no hard and fast rule when you should split a form into sections – it's just there to group related items visually.

Adding a navigation bar

By default iOS allows us to place content anywhere on the screen, including under the system clock and the home indicator. This doesn't look great, which is why by default SwiftUI ensures components are placed in an area where they can't be covered up by system UI or device rounded corners – an area known as the *safe area*.

On an iPhone 11, the safe area spans the space from just below the notch down to just above the home indicator. You can see it in action with a user interface like this one:

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

Try running that in the iOS simulator – press the Play button in the top-left corner of Xcode's window, or press Cmd+R.

You'll see that the form starts below the notch, so by default the row in our form is fully visible. However, forms can also scroll, so if you swipe around in the simulator you'll find you can move the row up so it goes under the clock, making them both hard to read.

A common way of fixing this is by placing a navigation bar at the top of the screen. Navigation bars can have titles and buttons, and in SwiftUI they also give us the ability to display new views when the user performs an action.

We'll get to buttons and new views in a later project, but I do at least want to show you how to add a navigation bar and give it a title, because it makes our form look better when it scrolls.

You've seen that we can place a text view inside a section by adding **Section** around the text view, and that we can place the section inside a **Form** in a similar way. Well, we add a navigation bar in just the same way, except here it's called **NavigationView**.

```
var body: some View {
    NavigationView {
        Form {
            Section {
                Text("Hello World")
            }
        }
    }
}
```

When you see that code in Xcode's canvas, you'll notice there's a large gray space at the top of your UI. Well, that's our navigation bar in action, and if you run your code in the simulator you'll see the form slides under the bar as it moves to the top of the screen.

You'll usually want to put some sort of title in the navigation bar, and you can do that by attaching a *modifier* to whatever you've placed inside. Modifiers are regular methods with one small difference: they always return a new instance of whatever you use them on.

Let's try adding a modifier to set the navigation bar title for our form:

```
NavigationView {
    Form {
        Section {
            Text("Hello World")
        }
    }
    .navigationBarTitle(Text("SwiftUI"))
}
```

When we attach the **.navigationBarTitle()** modifier to our form, Swift actually creates a new form that has a navigation bar title plus all the existing contents you provided.

When you add a title to a navigation bar, you'll notice it uses a large font for that title. You can get a small font with a slightly different call to **navigationBarTitle()**:

```
.navigationBarTitle("SwiftUI", displayMode: .inline)
```

You can see how Apple uses these large and small titles in the Settings app: the first screen says "Settings" in large text, and subsequent screens show their titles in small text.

Because it's so common to use large titles, there's a shortcut version you can use that lets you use a plain string rather than a text view:

```
.navigationBarTitle("SwiftUI")
```

Modifying program state

There's a saying among SwiftUI developers that our "views are a function of their state," but while that's only a handful of words it might be quite meaningless to you at first.

If you were playing a fighting game, you might have lost a few lives, scored some points, collected some treasure, and perhaps picked up some powerful weapons. In programming, we call these things *state* – the active collection of settings that describe how the game is right now.

When you quit the game that state will get saved as a save game, and when you come back to the game later you can reload your game to get back to where you were. But *while* you're playing, that's all just called *state*: all the integers, strings, Booleans, and more, all stored in RAM to describe what you are doing right now.

When we say SwiftUI's views are a function of their state, we mean that the way your user interface looks – the things people can see and what they can interact with – are determined by the state of your program. For example, they can't tap Continue until they have entered their name in a text field.

That in itself might sound obvious, but this is actually very different from the alternative that was used previously: your user interface was determined by a sequence of events. So, what the user sees right now is because they've been using your app for a while, have tapped various things, might have logged in or refreshed their data, and so on.

The "sequence of events" approach means it's very hard to store the state of an app, because the only way to get back the perfect state would be to play back the exact sequence of events that the user performed. This is why so many apps just don't even try to save your state, even slightly – your news app won't go back to the last article you were reading, Twitter won't remember if you were part-way through typing a reply to someone, and Photoshop forgets any undo state you had stacked up.

Let's put this into practice with a button, which in SwiftUI can be created with a title string and an action closure that gets run when the button is tapped:

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

    var body: some View {
        Button("Tap Count: \(tapCount)") {
            self.tapCount += 1
        }
    }
}
```

That code looks reasonable enough: create a button that says "Tap Count" plus the number of times the button has been tapped, then add 1 to **tapCount** whenever the button is tapped.

However, it won't build; that's not valid Swift code. You see, **ContentView** is a struct, which might be created as a constant. If you think back to when you learned about structs, that means it's *immutable* – we can't change its values freely.

When creating struct methods that want to change properties, we need to add the **mutating** keyword: **mutating func doSomeWork()**, for example. However, Swift doesn't let us make mutating computed properties, which means we can't write **mutating var body: some View** – it just isn't allowed.

This might seem like we're stuck at an impasse: we want to be able to change values while our program runs, but Swift won't let us because our views are structs.

Fortunately, Swift gives us a special solution called a *property wrapper*: a special attribute we can place before our properties that effectively gives them super-powers. In the case of storing simple program state like the number of times a button was tapped, we can use a property wrapper from SwiftUI called **@State**, like this:

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

```
    var body: some View {
        Button("Tap Count: \(tapCount)") {
            self.tapCount += 1
        }
    }
}
```

That small change is enough to make our program work, so you can now build it and try it out.

@State allows us to work around the limitation of structs: we know we can't change their properties because structs are fixed, but **@State** allows that value to be stored separately by SwiftUI in a place that *can* be modified.

Yes, it feels a bit like a cheat, and you might wonder why we don't use classes instead – they *can* be modified freely. But trust me, it's worthwhile: as you progress you'll learn that SwiftUI destroys and recreates your structs frequently, so keeping them small and simple structs is important for performance.

Tip: There are several ways of storing program state in SwiftUI, and you'll learn all of them. **@State** is specifically designed for simple properties that are stored in one view. As a result, Apple recommends we add **private** access control to those properties, like this: **@State private var tapCount = 0**.

Binding state to user interface controls

SwiftUI's **@State** property wrapper lets us modify our view structs freely, which means as our program changes we can update our view properties to match.

However, things are a little more complex with user interface controls. For example, if you wanted to create an editable text box that users can type into, you might create a SwiftUI view like this one:

```
struct ContentView: View {
    var body: some View {
        Form {
            TextField("Enter your name")
            Text("Hello World")
        }
    }
}
```

That tries to create a form containing a text field and a text view. However, that code won't compile because SwiftUI wants to know where to store the text in the text field.

Remember, views are a function of their state – that text field can only show something if it reflects a value stored in your program. What SwiftUI wants is a string property in our struct that can be shown inside the text field, and will also store whatever the user types in the text field.

So, we could change the code to this:

```
struct ContentView: View {
    var name = ""

    var body: some View {
```

```
        Form {
            TextField("Enter your name", text: name)
            Text("Hello World")
        }
    }
}
```

That adds a **name** property, then uses it to create the text field. However, that code *still* won't work because Swift needs to be able to update the **name** property to match whatever the user types into the text field, so you might use **@State** like this:

```
@State private var name = ""
```

But that still isn't enough, and our code still won't compile.

The problem is that Swift differentiates between "show the value of this property here" and "show the value of this property here, *but write any changes back to the property*."

In the case of our text field, Swift needs to make sure whatever is in the text is also in the **name** property, so that it can fulfill its promise that our views are a function of their state – that everything the user can see is just the visible representation of the structs and properties in our code.

This is what's called a *two-way binding*: we bind the text field so that it shows the value of our property, but we also bind it so that any changes to the text field also update the property.

In Swift, we mark these two-way bindings with a special symbol so they stand out: we write a dollar sign before them. This tells Swift that it should read the value of the property but also write it back as any changes happen.

So, the correct version of our struct is this:

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

```
    var body: some View {
        Form {
            TextField("Enter your name", text: $name)
            Text("Hello World")
        }
    }
}
```

Try running that code now – you should find you can tap on the text field and enter your name, as expected.

Before we move on, let's modify the text view so that it shows the user's name directly below their text field:

```
Text("Your name is \(name)")
```

Notice how that uses **name** rather than **$name**? That's because we don't want a two-way binding here – we want to *read* the value, yes, but we don't want to write it back somehow, because that text view won't change.

So, when you see a dollar sign before a property name, remember that it creates a two-way binding: the value of the property is read, but also written.

Creating views in a loop

It's common to want to create several SwiftUI views inside a loop. For example, we might want to loop over an array of names and have each one be a text view, or loop over an array of menu items and have each one be shown as an image.

SwiftUI gives us a dedicated view type for this purpose, called **ForEach**. This can loop over arrays and ranges, creating as many views as needed. Even better, **ForEach** doesn't get hit by the 10-view limit that would affect us if we had typed the views by hand.

ForEach will run a closure once for every item it loops over, passing in the current loop item. For example, if we looped from 0 to 100 it would pass in 0, then 1, then 2, and so on.

For example, this creates a form with 100 rows:

```
Form {
    ForEach(0 ..< 100) { number in
        Text("Row \(number)")
    }
}
```

Because **ForEach** passes in a closure, we can use shorthand syntax for the parameter name, like this:

```
Form {
    ForEach(0 ..< 100) {
        Text("Row \($0)")
    }
}
```

ForEach is particularly useful when working with SwiftUI's **Picker** view, which lets us show various options for users to select from.

To demonstrate this, we're going to define a view that:

1. Has an array of possible student names.
2. Has an **@State** property storing the currently selected student.
3. Creates a **Picker** view asking users to select their favorite, using a two-way binding to the **@State** property.
4. Uses **ForEach** to loop over all possible student names, turning them into a text view.

Here's the code for that:

```
struct ContentView: View {
    let students = ["Harry", "Hermione", "Ron"]
    @State private var selectedStudent = "Harry"

    var body: some View {
        Picker("Select your student", selection:
$selectedStudent) {
            ForEach(0 ..< students.count) {
                Text(self.students[$0])
            }
        }
    }
}
```

There's not a lot of code in there, but it's worth clarifying a few things:

1. The **students** array doesn't need to be marked with **@State** because it's a constant; it isn't going to change.
2. The **selectedStudent** property starts with the value "Harry" but can change, which is why it's marked with **@State**.
3. The **Picker** has a label, "Select your student", which tells users what it does and also provides something descriptive for screen readers to read aloud.
4. The **Picker** has a two-way binding to **selectedStudent**, which means it will start showing a selection of 0 but update the property as the user moves the picker.