

SwiftUI Development

iOS 17 • Xcode 15

STUDENT GUIDE



Contact

About Objects
11951 Freedom Drive
Suite 700
Reston, VA 20190

main: 571–346–7544
email: info@aboutobjects.com
web: www.aboutobjects.com

Course Information

Author: Jonathan Lehr
Revision: 2.1.0
Last Update: 07/07/2024

Classroom materials for a course that provides a rapid introduction to iOS development in SwiftUI. Includes coverage of the Combine framework and structured concurrency.

Copyright Notice

© Copyright 2020–2024 About Objects, Inc.

Under the copyright laws, this documentation may not by copied, photographed, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without express written consent.

All other copyrights, trademarks, registered trademarks and service marks throughout this document are the property of their respective owners.

All rights reserved worldwide. Printed in the USA.

SwiftUI Development

STUDENT GUIDE

About the Author

Jonathan Lehr

Co-Founder and VP, Training
About Objects

jonathan@aboutobjects.com



SECTION ONE

SwiftUI Basics

The SwiftUI Framework

- Integrates with UIKit, AppKit (macOS), and WatchKit.
- UIKit integration depends on UIKit components such as UIView, UIViewController.
 - Dependencies are largely hidden; however,
 - Your code may sometimes need to call directly into UIKit.
- Allows declarative specification of UI.
- Aims to reduce the amount of state maintained by UI objects.
- Makes extensive use of *property wrappers* to enable reactive behavior.

Developer Tools

- Xcode's SwiftUI Preview Pane
- iPhone, iPad, and Apple Watch Simulators
- SF Symbols
- Instruments

View Protocol

- Views are typically represented by Swift structs that conform to the `View` protocol.
- The `View` protocol defines a computed property named `body` that is accessed on state changes to render the view's content.

```
public protocol View {  
  
    /// The type of view representing the body of this view.  
    /// When you create a custom view, Swift infers this type from your  
    /// implementation of the required `body` property.  
    associatedtype Body : View  
  
    /// The content and behavior of the view.  
    @ViewBuilder var body: Self.Body { get }  
}
```

Defining a View

- Define a struct that conforms to the View protocol.
- Implement the body computed property defined in the protocol.

```
import SwiftUI

struct MyView: View {

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

ViewBuilders

- Note that in the View protocol, body is annotated with @ViewBuilder.
 - Leverages Swift's @resultBuilder feature.
 - Allows the trailing closure to define up to ten views.
 - Automatically nests views in a TupleView struct returned by the the body property's getter.

```
import SwiftUI

struct MyView: View {

    var body: some View {
        Text("One")
        Text("Two")
    }
}
```

@resultBuilder

- Streamlines code defining nested data structures — useful for implementing DSLs.
- Provides a number of `buildBlock()` result building methods.
- For example, you could define a custom `StringBuilder`:

```
@resultBuilder struct StringBuilder {
    static func buildBlock(_ components: String...) -> String {
        components.joined(separator: "")
    }
}
```

- You could then use `@StringBuilder` to annotate a closure passed to a struct's initializer, as follows:

```
struct StringGroup {
    var text: String
    init(@StringBuilder builder: () -> String) {
        text = builder()
    }
}
```

- `@StringBuilder` could then be used to annotate properties:

```
@StringBuilder var message: String {
    "Hello"
    " "
    "World"
    "!"
}
// The following would print "Hello World!"
```

```
print(message)
```

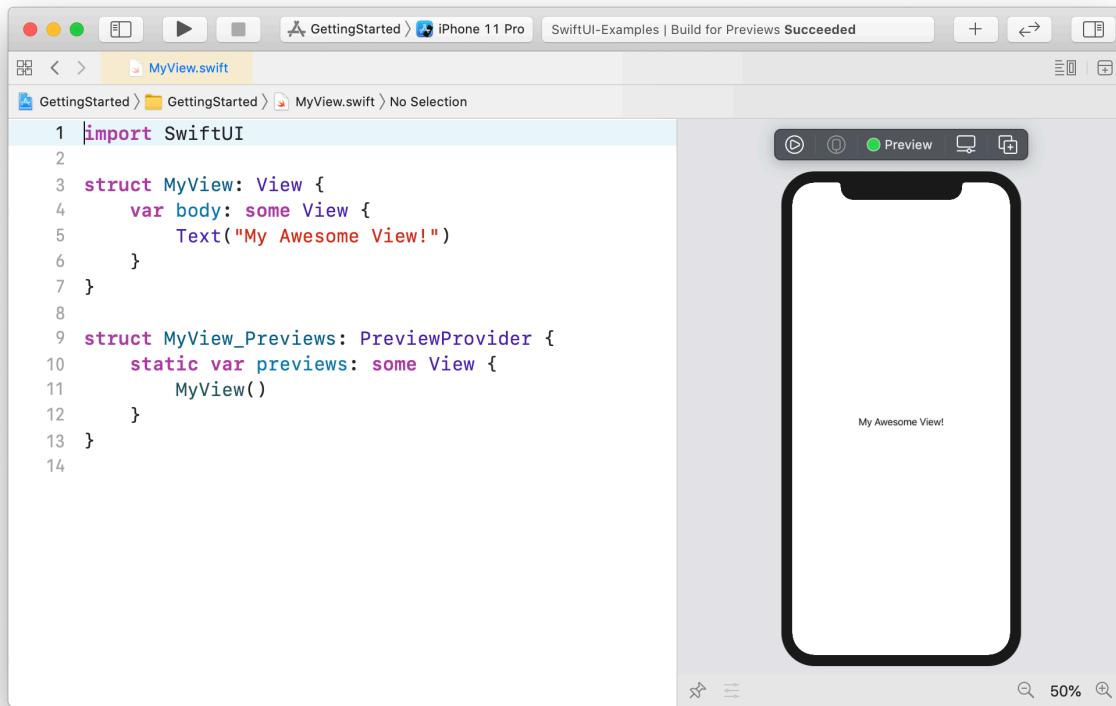
Xcode Previews

Define a struct that conforms to the `PreviewProvider` protocol to implement a live preview in Xcode.

```
struct MyView_Previews: PreviewProvider {  
    static var previews: some View {  
        MyView()  
    }  
}
```

Xcode automatically executes a Simulator instance and presents it in a canvas view on the right.

The canvas view can be toggled on and off via **Cmd-Option-Return**.



Images and Symbols

- `Image` is a struct that defines a `View` wrapper for image data.
- You can initialize by providing the name of an image in one of your app's asset catalogs.

```
Image(name: "Foo")
```

- `SFSymbols` — vector art provided and maintained by Apple's design team.
- Use the `SFSymbols` app to browse the collection.
- Use the `systemName` parameter to specify an `SFSymbol`:

```
Image(systemName: "sunrise.fill")
```

Container Views

- An important layout mechanism is provided by *layout container* views such as HStack, VStack, and ZStack.
- Layout containers initiate a simple sequence of steps:
 1. Container views offer a size to each of their children.
 2. Child views then determine how much of the offered size they want to use.
 3. The container views then position their children.
- Note that there are two other types of container views: *collection containers*, and *presentation containers* that we'll cover later.

Modifiers

- Use modifiers to configure a view's appearance and behavior.
- Modifiers are methods that return the current view wrapped in a view that provides some additional feature(s).
- For example, the code below applies a `foregroundColor(_:_)` modifier to a `Text` view:

```
Text("Hello World!")  
    .foregroundColor(.green)
```

Hello World!

- Modifiers can be chained:

```
Text("Hello World!")  
    .foregroundColor(.green)  
    .padding()  
    .border(.purple)
```

Hello World!

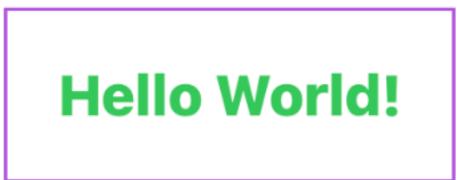
View-Specific Modifiers

- View types can provide their own custom modifiers.
- For example, `fontWeight(_:_)` is declared in `Text`. Trying to apply it to something that is not an instance of `Text` is a compile error.

```
Text("Hello World!")  
    .foregroundColor(.green)  
    .padding()  
    .border(.purple)  
    .fontWeight(.heavy) // Compile error.
```

To fix the above error, change the order of the modifiers:

```
Text("Hello World!")  
    .fontWeight(.heavy)  
    .foregroundColor(.green)  
    .padding()  
    .border(.purple)  
  
// .fontWeight(_:_:) works here because .foregroundColor(_:_:  
// returns Text, whereas the next modifier, .padding(),  
// returns View.
```



Hello World!

Managing Layout

Views for Managing Layout

- Stack views:
 - HStack, VStack, ZStack
 - LazyHStack, LazyVStack
- Grid views: LazyHGrid and LazyVGrid
- Lists: List and ForEach (note that ForEach defers layout to its container view.)
- Group views and Spacer views
- Navigation views and sheets

Modifiers for Managing Layout

- padding
- frame
- layoutPriority
- overlay and background

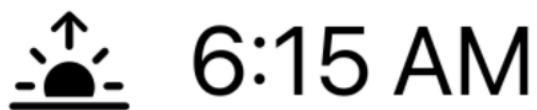
Stack Views

- Stacks offer space to their children in order, from the ‘least flexible’ child to the ‘most flexible.’
- Examples of layout flexibility:
 - An Image calculates its size based on its content, and always prefers to be that size.
 - A Text also calculates its size from its content, but is a little more flexible than an Image.
 - A RoundedRectangle is even more flexible — it simply takes whatever size is offered.
- After calculating the sizes of its children, a stack sizes itself to fit.
- Use the `.layoutPriority(Double)` modifier to override the default order in which space is offered.
 - Default priority is 0.

Horizontal Stacks

- To define a horizontal layout, you can nest subviews in an instance of `HStack`, initialized with a `ViewBuilder`.
- You can optionally define vertical alignment and/or spacing between nested elements as parameter values.
- Modifiers can be applied directly to a stack, as shown below.

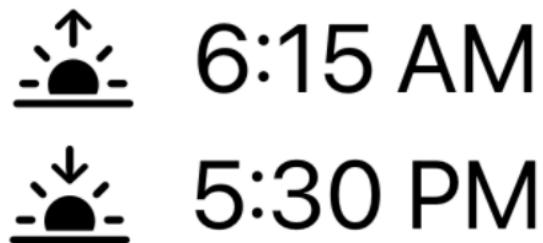
```
struct WeatherView: View {  
    var body: some View {  
        HStack(spacing: 18) {  
            Image(systemName: "sunrise.fill")  
            Text("6:15 AM")  
        }  
        .font(.system(size: 36))  
    }  
}
```



Vertical Stacks

- As with instances of HStack, a VStack can be initialized with a ViewBuilder, as well as spacing and horizontal alignment (both optional).

```
struct WeatherListView: View {  
    var body: some View {  
        VStack(spacing: 8) {  
            HStack(spacing: 18) {  
                Image(systemName: "sunrise.fill")  
                Text("6:15 AM")  
            }  
            HStack(spacing: 18) {  
                Image(systemName: "sunset.fill")  
                Text("5:30 PM")  
            }  
        }  
        .font(.system(size: 36))  
    }  
}
```



ZStacks

- Similar to the other stack types, you initialize a ZStack with a ViewBuilder, as well as an optional alignment parameter.
- You can specify alignment to any edge or corner, for example .top, or .topLeading.

```
ZStack {  
    RoundedRectangle(cornerRadius: 15)  
        .fill(.orange)  
    RoundedRectangle(cornerRadius: 15)  
        .fill(Color.white.opacity(0.9))  
    RoundedRectangle(cornerRadius: 15)  
        .stroke(.orange, lineWidth: 5)  
    Image(systemName: "star.circle")  
        .font(.system(size: 60, weight: .light))  
        .foregroundColor(.indigo.opacity(0.5))  
}  
.frame(height: 80)
```



Custom Views

- To create a custom view type, define a struct that conforms to the `View` protocol.
- That requires your struct to implement the `body` property.

```
struct WeatherCell: View {  
    let imageName: String  
    let time: String  
  
    var body: some View {  
        HStack(spacing: 18) {  
            Image(systemName: imageName)  
            Text(time)  
        }  
    }  
}
```

- Creating instances:

```
struct WeatherListView: View {  
    var body: some View {  
        VStack(spacing: 8) {  
            WeatherCell(imageName: "sunrise.fill", time: "6:15 AM")  
            WeatherCell(imageName: "sunset.fill", time: "5:30 PM")  
        }  
        .font(.system(size: 36))  
    }  
}
```

Shapes – 1

- Use shapes for simple design composition. (For more sophisticated custom drawing, use a Canvas view.)
- Use ShapeStyle objects to configure fill and stroke styles, for example using a Material or Gradient style.
- The example below strokes a Circle with an AngularGradient:

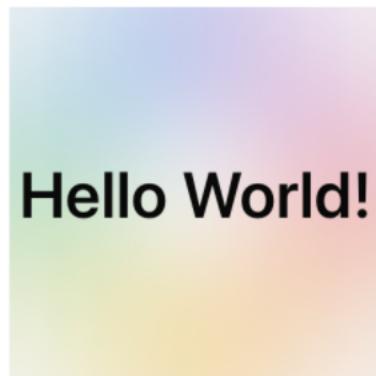
```
let gradient = AngularGradient(  
    colors: [.red, .orange, .yellow, .green, .blue, .purple, .pink],  
    center: .center  
)  
  
var body: some View {  
    ZStack {  
        Circle()  
            .stroke(gradient, lineWidth: 20)  
            .frame(width: 100, height: 100)  
    }  
}
```



Shapes – 2

- The following example adds a background material to a Text view and places it on top of the Circle from the previous example.

```
let gradient = AngularGradient(  
    colors: [.red, .orange, .yellow, .green, .blue, .purple, .pink],  
    center: .center  
)  
  
var body: some View {  
    ZStack {  
        Circle()  
            .stroke(gradient, lineWidth: 20)  
            .frame(width: 100, height: 100)  
        Text("Hello World!")  
            .font(.system(size: 24).bold())  
            .frame(width: 140, height: 140)  
            .background(.ultraThinMaterial)  
    }  
}
```



Conditional Views

- SwiftUI supports using `if...else` constructs and `switch` statements as a way to conditionally include views.

```
struct ConditionalView: View {  
    @State var isSunny = false  
  
    var body: some View {  
        VStack {  
            Spacer()  
            if isSunny {  
                Text("The weather is sunny.")  
                    .padding()  
                    .frame(width: 140, height: 140)  
                    .border(.yellow, width: 6)  
            } else {  
                Image(systemName: "cloud.sun.rain.fill")  
                    .font(.system(size: 60))  
                    .padding(20)  
                    .background(.gray)  
            }  
            Button(action: toggleWeather,  
                  label: { Text("Toggle") })  
            Spacer()  
        }  
        .symbolRenderingMode(.multicolor)  
        .font(.system(size: 24))  
    }  
  
    func toggleWeather() {  
        isSunny.toggle()  
    }  
}
```

GeometryReader

- GeometryReader is view that acts as a wrapper for views that need access to their size and position.
- You configure a geometry reader with a closure that takes a GeometryProxy as its only argument. The proxy is a wrapper for information about the container view's frame.

```
struct Geometry: View {  
    var body: some View {  
        HStack {  
            GeometryReader { geometry in  
                let width = geometry.size.width / 2  
                let height = geometry.size.height / 2  
                Color.blue  
                    .frame(width: width, height: height)  
                    .position(x: width, y: height)  
            }  
        }  
        .frame(width: 180, height: 80, alignment: .center)  
        .background(.yellow)  
    }  
}
```



SECTION TWO

Property Wrappers

Property Wrappers – 1

Property wrappers make it convenient to observe (and react to) changes to properties in pre-defined ways.

For example, suppose you want certain properties to always contain capitalized strings.

You could add property observers on a case-by-case basis:

```
struct Person {  
    var firstName: String {  
        didSet { firstName = firstName.capitalized }  
    }  
    // Etc...  
}
```

Property Wrappers – 2

However, that approach could lead to dual maintenance. Instead, you could define a wrapper struct to encapsulate the required behavior:

```
@propertyWrapper struct Capitalized {
    var wrappedValue: String
    didSet { wrappedValue = wrappedValue.capitalized }
}

init(wrappedValue: String) {
    self.wrappedValue = wrappedValue.capitalized
}
}
```

- `@propertyWrapper` streamlines definition of wrapper structs.
- Only requirement is a property named `wrappedValue`.
- Can optionally provide a computed property named `projectedValue`.
 - We'll see later how `projectedValue` gets used.

Property Wrappers – 3

You could then wrap individual properties by simply annotating them:

```
struct Person {  
    @Capitalized var firstName: String  
    @Capitalized var lastName: String  
}  
  
func testCapitalized() {  
    var fred = Person(firstName: "fred", lastName: "smith")  
    print("\(fred.firstName) \(fred.lastName)")  
    // Prints "Fred Smith"  
}
```

Property Wrappers – 4

Conceptually, a property wrapper is just a struct that contains a wrapped value.

```
// A wrapper struct that accesses its stored value via a computed
// property.
struct CapitalizedString {
    private var _wrappedValue: String

    var wrappedValue: String {
        get { _wrappedValue }
        set { _wrappedValue = newValue.capitalized }
    }

    init(wrappedValue: String) {
        self._wrappedValue = wrappedValue.capitalized
    }
}
```

However, this could be awkward to use without the `@propertyWrapper` feature:

```
struct Person {
    ...

    var middleName: CapitalizedString
}

func testCapitalized() {
    fred.middleName = CapitalizedString(wrappedValue: "xavier")
    print(fred.middleName.wrappedValue)
    // Prints "Xavier"
}
```

SECTION THREE

Bindings and State

@State

- @State is a property wrapper for a value that represents temporary view state.
- Underlying type is:

```
@frozen @propertyWrapper struct State<Value>
```

- @State causes the wrapped value to be copied to heap, and maintained there on your behalf. That allows a View to ‘remember’ the state it was in it (the View) is recreated.
- When the wrapped value changes, SwiftUI automatically calls body on any views that reference the property to get their updated content.
- The *projected value* of an @State property is of type Binding<T>.

@State Example

```
struct HelloView: View {  
    @State private var isSayingHello = true  
  
    var body: some View {  
        HStack {  
            Text(isSayingHello ? "Hello!" : "Goodbye.")  
            Button(action: toggle,  
                   label: { Text("Toggle") })  
        }  
    }  
  
    func toggle() {  
        isSayingHello.toggle()  
    }  
}
```

Before toggling:

Hello! [Toggle](#)

After toggling:

Goodbye. [Toggle](#)

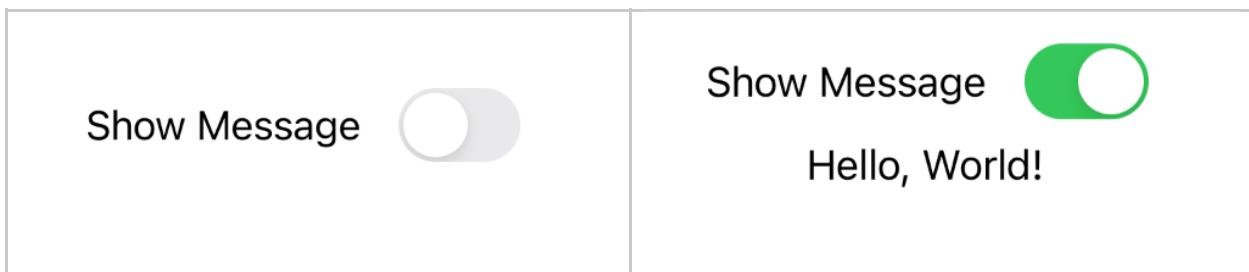
@Binding

- @Binding is a property wrapper for a read-write value.
- Underlying type is:

```
@propertyWrapper struct Binding<Value>
```

- Use a \$ prefix to pass a binding (rather than its underlying value) as an argument.

```
struct ContentView: View {  
    @State private var isVisible = false  
  
    var body: some View {  
        VStack {  
            Toggle("Show Message", isOn: $isVisible)  
  
            Text(isVisible ? "Hello, World!" : "")  
        }  
        .frame(width: 180)  
    }  
}
```



Animation

- SwiftUI can animate changes to any of the following:
 - The view hierarchy (a View being added or removed)
 - A subset of `ViewModifier` argument values
 - Shapes (e.g., Rectangle, Circle, Path, etc.)

Implicit Animations

- Define an implicit animation with the `.animation(_:, value:)` view modifier.

```
struct AnimateAllTheThings: View {
    @State var isRotating = false

    var body: some View {
        VStack {
            Spacer()
            Text("Hello, World!")
                .bold()
                .padding()
                .foregroundColor(.white)
            // Animatable modifiers
            .background(isRotating ? .pink : .purple)
            .opacity(isRotating ? 0.5 : 1.0)
            // Animatable geometry effects
            .transformEffect(CGAffineTransform(translationX: 0,
                                              y: isRotating ? 100 : 0))
            .scaleEffect(isRotating ? CGSize(width: 1.5, height: 1.5)
                                  : CGSize(width: 1, height: 1))
            .rotationEffect(Angle(degrees: isRotating ? 400 : 0))
            // Implicit animation configuration
            .animation(.easeInOut(duration: 1), value: isRotating)
        Spacer()
        Button(action: rotate, label: { Text("Rotate") })
            .buttonStyle(.bordered)
        Spacer()
    }
}

private func rotate() {
    isRotating.toggle()
    DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
        isRotating.toggle()
    }
}
```

Explicit Animations

- Define an explicit animation as the body of a call to the `withAnimation()` function.

```
struct AnimatedCrossDissolve: View {  
    @State private var isDefault = true  
  
    var body: some View {  
        ZStack {  
            if (isDefault) {  
                Circle()  
                    .fill(Color.blue)  
                    .frame(width: 100, height: 100)  
            } else {  
                Rectangle()  
                    .fill(Color.green)  
                    .frame(width: 100, height: 100)  
            }  
        }  
        .onTapGesture { crossDissolve() }  
    }  
  
    func crossDissolve() {  
        withAnimation(.easeInOut(duration: 1)) {  
            isDefault.toggle()  
        }  
    }  
}
```

Gestures – 1

- SwiftUI provides modifiers for configuring actions to be performed in response to gestures.
- Gesture modifiers provide three optional callbacks: `updating(_:body:)`, `onChanged(_:)`, and `onEnded(_:)`.
- For convenience, `View` provides an `onTapGesture(_:)` modifier.

```
struct TapGestureView: View {  
    @State private var isAlternateColor = false  
  
    var body: some View {  
        Circle()  
            .fill(isAlternateColor ? .green : .blue)  
            .frame(width: 80, height: 80)  
            .onTapGesture {  
                isAlternateColor.toggle()  
            }  
    }  
}
```

Gestures – 2

- For more complex gestures, use the general-purpose `gesture(_:) modifier.`

```
struct RotationGestureView: View {  
    @State private var angle = Angle(degrees: 0)  
  
    var rotationGesture: some Gesture {  
        RotationGesture()  
            .onChanged { angle = $0 }  
    }  
  
    var body: some View {  
        Color.blue  
            .frame(width: 160, height: 80)  
            .rotationEffect(angle)  
            .gesture(rotationGesture)  
    }  
}
```

@Published

- The `@Published` property wrapper publishes changes to its wrapped value.
- Under the hood the property wrapper publishes changes by calling `objectWillChange.send()` on `self` (the `ObservableObject` that contains it).
- SwiftUI views automatically update any time an `@Published` property they're bound to publishes a value change.
- Note that unlike `@State` (and `@StateObject`, `@Binding`, and `@ObservedObject`, for that matter), the *projected value* of `@Published` is a *publisher* (not a binding)!
- `@Published` properties are generally only used in view models. Note that they can't be used in structs — only in classes.

ObservableObject

- The ObservableObject protocol defines a single property, `objectWillChange` of type `ObjectWillChangePublisher`.
- The protocol conforms to `AnyObject`, so it can only be implemented by a class (i.e., it can't be implemented by a struct or enum).
- That means instances are *passed by reference*, unlike structs and enums, which are passed by value.
- An ObservableObject calls its `objectWillChange` publisher's `send()` method to broadcast pending value changes to any of its `@Published` properties.

@StateObject

- `@StateObject` is a property wrapper for an `ObservableObject`.
- Underlying type is:

```
@propertyWrapper struct StateObject<ObjectType> where ObjectType : ObservableObject
```

- SwiftUI creates a new instance of the `ObservableObject` once per instance of the View that declares the object.
- When published properties of the observable object change, SwiftUI automatically updates any parts of the view that depend on their values.
- The *projected value* of a `@StateObject` property is a binding to the properties of the wrapped value (which is typically a view model).

@Published Example – 1

@Published Property: ViewModel Implementation

```
struct Person {
    var name: String
    var age: Int
}

final class PersonViewModel: ObservableObject {
    @Published var person: Person?

    init() {
        loadPerson()
    }

    // Simulated fetch
    private func loadPerson() {
        person = Person(name: "Fred Smith", age: 30)
    }

    func change(name: String, age: Int) {
        person?.name = name
        person?.age = age
    }
}
```

@Published Example – 2

@Published Property: View References

```
let values:[(String, Int)] = [
    ("Rob Jones", 27), ("Jill Brown", 25), ("Jan Smith", 33),
    ("Joe James", 27), ("Pat Marks", 25), ("Will Trent", 33),
]

struct PersonView: View {
    @StateObject var viewModel = PersonViewModel()

    var body: some View {
        Form {
            HStack {
                Text("Name:")
                Text("\(viewModel.person?.name ?? "unknown")")
            }
            HStack {
                Text("Age:")
                Text("\(viewModel.person?.age ?? 0)")
            }
            HStack {
                Button(action: change, label: { Text("Change") })
            }
        }
    }

    func change() {
        let (name, age) = values[Int.random(in: 0..
```

@ObservedObject

- `@ObservedObject` is another property wrapper for an `ObservableObject`.
- Underlying type is:

```
@propertyWrapper struct ObservedObject<ObjectType> where ObjectType : ObservableObject
```

- An observed object is just a reference to an `ObservableObject` owned by another view, and passed in as an argument to an initializer.
- The *projected value* of an `@ObservedObject` property is a binding to the properties of the wrapped value (which is typically a view model).

@ObservedObject Example – 1

- We could take PersonView from the previous example and simply change the annotation on the viewModel property.

Original

```
struct PersonView: View {
    @StateObject var viewModel = PersonViewModel()
```

New

```
struct PersonView_1: View {
    @ObservedObject var viewModel: PersonViewModel
```

- Other views could then potentially share a reference to that same viewModel:

```
struct PersonView_2: View {
    @ObservedObject var viewModel: PersonViewModel

    var body: some View {
        VStack {
            HStack {
                Text("Name:")
                Text("\(viewModel.person?.name ?? "unknown")")
            }
            HStack {
                Text("Age:")
                Text("\(viewModel.person?.age ?? 0)")
            }
        }
    }
}
```

@ObservedObject Example – 2

- A parent view could then instantiate the view model as a `@StateObject` and pass it as parameter to both views:

```
struct ObservedObjectView: View {  
    @StateObject private var personViewModel = PersonViewModel()  
  
    var body: some View {  
        TabView {  
            PersonView_1(viewModel: personViewModel)  
                .tabItem {  
                    Image(systemName: "person")  
                    Text("Person Form")  
                }  
            PersonView_2(viewModel: personViewModel)  
                .tabItem {  
                    Image(systemName: "person.2")  
                    Text("Person Stack")  
                }  
        }  
    }  
}
```

@Environment

- Use the `@Environment` property wrapper to access values stored in the environment.
- It takes an initializer value of type `KeyPath` that identifies the environment value you're interested in.

```
@Environment(\.colorScheme) var colorScheme: ColorScheme
```

- System-provided values are defined in the `EnvironmentValues` struct.
- You can use the `environment(_:_:)` view modifier to set or override values, but in most cases, there are existing view modifiers for that purpose.

@Environment Example

```
struct EnvironmentContentView: View {
    // Reads the current color scheme inherited from
    // parent view
    @Environment(\.colorScheme) var colorScheme: ColorScheme

    var body: some View {
        VStack(spacing: 4) {
            Text("Hello, World!")
            Text("Have a Nice " +
                (colorScheme == .dark ? "Night!" : "Day!"))
            Image(systemName: colorScheme == .dark
                ? "moon.circle.fill"
                : "sun.max.fill")
        }
        // Explicitly sets values in VStack's environment.
        // Note that there are existing modifiers for each of these.
        .environment(\.font, Font.system(size: 30))
        .environment(\.symbolRenderingMode, .multicolor)
    }
}

struct Environment_Previews: PreviewProvider {
    static var previews: some View {
        EnvironmentContentView()
        EnvironmentContentView()
        // Sets the color scheme in the environment
        .preferredColorScheme(.dark)
    }
}
```

@EnvironmentObject

- `@EnvironmentObject` is property wrapper for an `ObservableObject` passed in the environment.
- Underlying type is:

```
@propertyWrapper struct EnvironmentObject<ObjectType> where ObjectType : ObservableObject
```

- You use an environment object as an alternative to an `@ObservedObject` when you want to share the reference more broadly, without having to explicitly pass it everywhere it may be needed.
- The *projected value* of an `@EnvironmentObject` property is a binding to the properties of the wrapped value (which is typically a view model).

@EnvironmentObject Example – 1

- We could again take PersonView from the previous example and simply change the annotation on the viewModel property.

Original

```
struct PersonView_1: View {
    @ObservedObject var viewModel = PersonViewModel()
```

New

```
struct PersonView_3: View {
    @EnvironmentObject var viewModel: PersonViewModel
```

- Other views could also inherit a reference to that view model.

```
struct PersonView_4: View {
    @EnvironmentObject var viewModel: PersonViewModel

    var body: some View {
        VStack {
            HStack {
                Text("Name:")
                Text("\(viewModel.person?.name ?? "unknown")")
            }
            HStack {
                Text("Age:")
                Text("\(viewModel.person?.age ?? 0)")
            }
        }
    }
}
```

@EnvironmentObject Example – 2

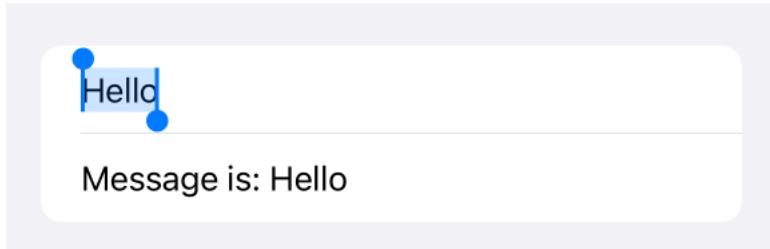
- A parent view could then instantiate the view model as a `@StateObject` and add it to the environment of a view by using the `environmentObject(:_)` view modifier:

```
struct EnvironmentObjectView: View {  
    @StateObject private var personViewModel = PersonViewModel()  
  
    var body: some View {  
        TabView {  
            PersonView_3()  
                .tabItem {  
                    Image(systemName: "person")  
                    Text("Person Form")  
                }  
            PersonView_4()  
                .tabItem {  
                    Image(systemName: "person.2")  
                    Text("Person Stack")  
                }  
        }  
        .environmentObject(personViewModel)  
    }  
}
```

Editing Text – 1

- Use `TextField` to edit a single line of text, and `TextEditor` for multiline editing.
- You configure a text field with a placeholder string and a *binding* to the value to be edited.

```
struct TextFieldBasicsView: View {  
    @State private var message = ""  
  
    var body: some View {  
        Form {  
            TextField("Message", text: $message)  
            Text("Message is: \(message)")  
        }  
    }  
}
```



Editing Text – 2

- You can use an `@Published` property of your view model to track the text field's value, and update model values if it changes.

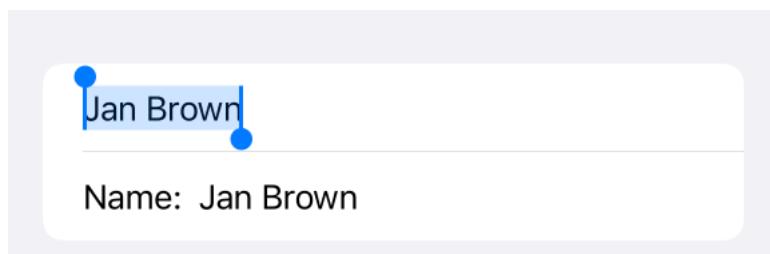
```
final class PersonViewModel_2: ObservableObject {
    @Published var person: Person?

    @Published var name: String = "" {
        didSet { person?.name = name }
    }
}
```

- Then configure the field with a binding to the published property.

```
struct EditablePersonView: View {
    @EnvironmentObject private var viewModel: PersonViewModel_2

    var body: some View {
        Form {
            TextField("Name", text: $viewModel.name)
            HStack {
                Text("Name:")
                Text("\(viewModel.person?.name ?? "unknown")")
            }
        }
    }
}
```



SECTION FOUR

Working with

Collections

ForEach – 1

- Use ForEach to configure a list of repeating views based on a collection of data.
- If the elements of the collection don't conform to Identifiable, you need to provide an additional id parameter in the initializer.

```
struct Collections_1: View {  
    let strings = ["First Item", "Second Item", "Third Item",  
                  "Fourth Item", "Fifth Item", "Sixth Item"]  
  
    var body: some View {  
        VStack {  
            ForEach(strings, id: \.self) { string in  
                Text(string)  
            }  
        }  
    }  
}
```

First Item
Second Item
Third Item
Fourth Item
Fifth Item
Sixth Item

ForEach Example – 1

```
struct Collections_1: View {
    var body: some View {
        let colors: [Color] = [.red, .orange, .yellow, .green, .blue,
                              .indigo, .purple, .cyan, .teal, .brown]

        VStack {
            ForEach(colors, id: \.self) { color in
                ColorDescription(color: color)
            }
        }
    }
}

struct ColorDescription: View {
    let color: Color

    var body: some View {
        HStack {
            HStack {
                Text("\(color.description.capitalized)")
                    .font(.system(size: 26, weight: .light))
                Spacer()
            }
            .frame(width: 100)

            color
        }
        .padding()
    }
}
```

ForEach – 2

- Conformance with Identifiable simply requires a property name `id` whose values are guaranteed to be unique.

```
struct Contact: Identifiable {
    let id = UUID()
    var name: String
}

struct Collections_2: View {
    let contacts: [Contact] = [
        Contact(name: "Jill Smith"),
        Contact(name: "Jan Brown"),
        Contact(name: "Bob Jones"),
    ]

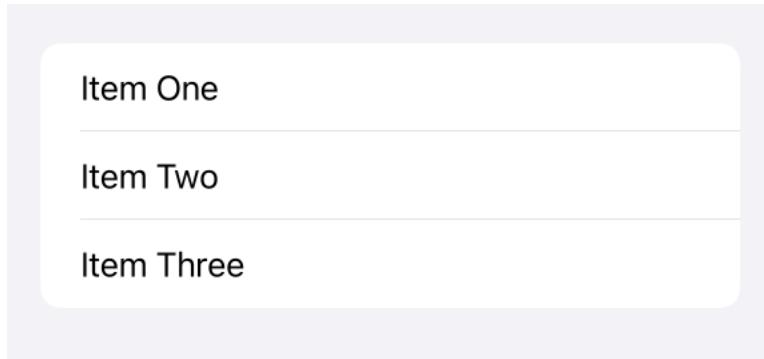
    var body: some View {
        VStack {
            ForEach(contacts) { contact in
                Text(contact.name)
            }
        }
    }
}
```

Jill Smith
Jan Brown
Bob Jones

List Views – 1

- A List presents its content in tabular form in a scrollable area.
- It draws separators between elements by default.

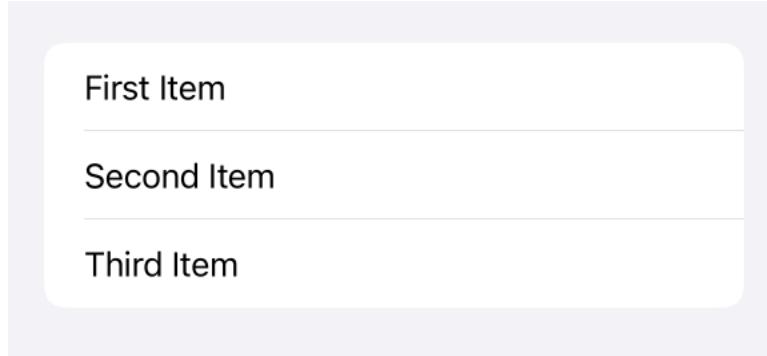
```
struct Collections_3_1: View {  
    var body: some View {  
        List {  
            Text("Item One")  
            Text("Item Two")  
            Text("Item Three")  
        }  
    }  
}
```



List Views – 2

- You can nest a `ForEach` to populate a scrollable list based on a collection of values:

```
struct Collections_3: View {  
    let strings = ["First Item", "Second Item", "Third Item"]  
  
    var body: some View {  
        List {  
            ForEach(strings, id: \.self) { string in  
                Text(string)  
            }  
        }  
    }  
}
```

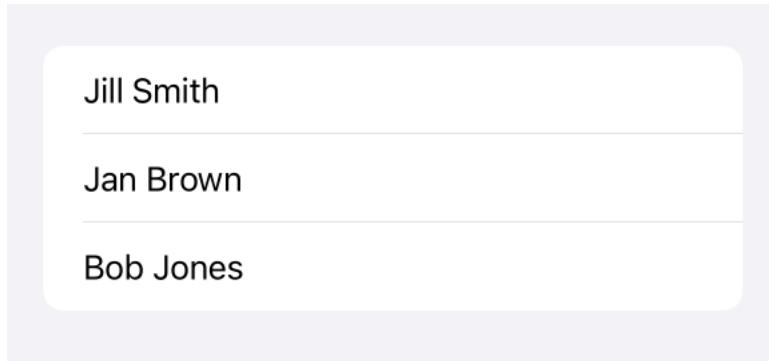


List Views – 3

- You can also use a List without a nested ForEach.

```
let contacts: [Contact] = [
    Contact(name: "Jill Smith"),
    Contact(name: "Jan Brown"),
    Contact(name: "Bob Jones"),
]

struct Collections_4: View {
    var body: some View {
        List(contacts) { contact in
            Text(contact.name)
        }
    }
}
```



List Views – 4

- Lists come with built-in support for single and multiple selection.
- For multiple selection, bind the selection initializer argument to a Set generically typed to the type of the model object's id property.

```
struct Collections_4_1: View {  
    @State private var selections: Set<UUID> = []  
  
    var body: some View {  
        NavigationView {  
            List(contacts, selection: $selections) { contact in  
                Text(contact.name)  
            }  
            .navigationTitle("Contacts")  
            .toolbar { EditButton() }  
        }  
    }  
}
```



SECTION FIVE

Observation

The Observation Framework

Observation is a Swift framework that defines a mechanism for tracking changes that doesn't depend on Combine.

Observation is designed to work with reference types, so your model object types will be classes rather than structs.

To enable automatic observation of property value changes, simply prefix class declarations with an `@Observable` macro, as shown below:

```
@Observable class BookViewModel {  
    var book = Book()  
}
```

Observable Properties

By default, any properties you define in an `@Observable` type have change tracking enabled. The macro expansion provides for this by prefixing your properties with `@ObservationTracked`.

Any value change to an observable property with change tracking enabled will trigger notifications to all of its observers.

To disable change tracking for a given property, simply prefix its declaration with `@ObservationIgnored`.

```
@ObservationIgnored var greeting = "Hello World!"
```

Primary Property Wrappers

When working with objects defined as `@Observable`, many of the property wrappers described in the **Bindings and State** section are no longer relevant. However, `@State`, `@Environment`, and the new `@Bindable` property wrapper may be useful in some cases.

`@State`

Use where you would have previously used `@StateObject`.

`@Bindable`

Use where a view needs bindings to an observable object.

`@Environment`

Use where you want to make an observable object globally visible throughout a hierarchy of views.

Observation Example – 1

```
import Observation
import SwiftUI

// MARK: - Model Objects
@Observable class Author {
    var name: String

    init(name: String = "") {
        self.name = name
    }
}

@Observable class Book {
    var title: String
    var author: Author

    init(title: String = "", author: Author = Author()) {
        self.title = title
        self.author = author
    }
}

// MARK: - View Model
@Observable class BookViewModel {
    var book = Book(title: "Book One",
                    author: Author(name: "Fred Smith"))
}
```

Observation Example – 2

```
// MARK: - Views
struct ObservableBookContentView: View {
    @State private var viewModel = BookViewModel()

    var body: some View {
        Form {
            Section {
                TextInputView(book: viewModel.book)
            }
            Section {
                SummaryViewOne(book: viewModel.book)
            }
            Section {
                SummaryViewTwo()
                    .environment(viewModel)
            }
        }
    }
}

struct TextInputView: View {
    @Bindable var book: Book

    var body: some View {
        VStack {
            TextField("Title", text: $book.title)
            TextField("Author", text: $book.author.name)
        }
    }
}
```

Observation Example – 3

```
struct SummaryViewOne: View {
    let book: Book

    var body: some View {
        VStack(alignment: .leading, spacing: 8) {
            Text("Summary One").bold()
            Text("Title: \(book.title)")
            Text("Author: \(book.author.name)")
        }
        .font(.callout.monospaced())
    }
}

struct SummaryViewTwo: View {
    @Environment(BookViewModel.self) var viewModel: BookViewModel

    var body: some View {
        VStack(alignment: .leading, spacing: 8) {
            Text("Summary Two").bold()
            Text("Title: \(viewModel.book.title)")
            Text("Author: \(viewModel.book.author.name)")
        }
        .font(.callout.smallCaps())
    }
}
```

Tracking Collections and Computed Properties

Observation works seamlessly with properties that contain collections, as well as with computed properties. The code in the following pages includes examples of both.

When working with a collection, direct modifications to the collection itself, as well as modifications to objects nested in the collection trigger updates automatically.

If a computed property depends on a stored property, changes to that stored property will cause any associated views to re-invoke the computed property accessor.

Change Tracking Example – 1

```
@Observable class BookListViewModel {  
    let testData = [  
        Book(title: "Book One", author: Author(name: "Fred Smith")),  
        Book(title: "Book Two", author: Author(name: "Jan Jones")),  
        Book(title: "Book Three", author: Author(name: "Bill Yu")),  
    ]  
  
    // Tracked collection property  
    var books: [Book] = []  
  
    // Computed properties  
    var count: Int { books.count }  
    var isEmpty: Bool { books.isEmpty }  
    var lastBook: Book { books.last ?? Book() }  
  
    init() { books = testData }  
  
    func append(_ book: Book) {  
        books.append(book)  
    }  
  
    func removeLastBook() {  
        books.removeLast()  
    }  
}
```

Change Tracking Example – 2

```
struct ObservableBookListContentView: View {
    let viewModel: BookListViewModel

    var body: some View {
        Text("Count: \(viewModel.count)")
            .font(.subheadline)
        Form {
            Section {
                TextInputView(book: viewModel.lastBook)
            }
            List {
                ForEach(viewModel.books, id: \.title) { book in
                    BookCell(book: book)
                }
            }
        }
        BookListControls(viewModel: viewModel)
    }
}

struct BookCell: View {
    let book: Book

    var body: some View {
        VStack(alignment: .leading, spacing: 8) {
            Text(book.title)
                .font(.headline)
            Text(book.author.name)
                .font(.subheadline)
        }
    }
}
```

Change Tracking Example – 3

```
struct BookListControls: View {
    let viewModel: BookListViewModel

    var body: some View {
        HStack {
            Button("Delete Last Cell") {
                if !viewModel.isEmpty {
                    viewModel.removeLastBook()
                }
            }
            .disabled(viewModel.isEmpty)
            Spacer()
            Button("Restore Last Cell") {
                if viewModel.count < 3 {
                    viewModel.append(
                        viewModel.testData[viewModel.count]
                    )
                }
            }
            .disabled(viewModel.count == 3)
        }
        .padding()
    }
}
```

SECTION SIX

Navigation and Modal Presentation

Presenting a Sheet

- There are two pairs of view modifiers for presenting a sheet temporarily on screen. One of the pairs is shown below.
- What differs is the first parameter, which is used to toggle presentation.
- In other words, you can either use an optional or a boolean to determine whether the sheet is currently shown.

```
func sheet<Content>(  
    isPresented: Binding<Bool>,  
    onDismiss: (() -> Void)? = nil,  
    @ViewBuilder content: @escaping () -> Content) -> some View where Content : View  
  
func sheet<Item, Content>(  
    item: Binding<Item?>,  
    onDismiss: (() -> Void)? = nil,  
    @ViewBuilder content: @escaping (Item) -> Content) -> some View where Item :  
Identifiable, Content : View
```

- The other pair of view modifiers take the same arguments, but substitute the function name `fullScreenCover` for `sheet`.
- As the name suggests, the former presents a sheet covering the entire screen, whereas the latter is presented in a card-style interface.

Sheet Example – 1

```

struct Sheets_1: View {
    @State private var isShowingSheet = false

    var body: some View {
        VStack {
            TitledButton(title: "Main View",
                        action: showSheet,
                        label: "Show Sheet",
                        backgroundColor: .brown)
        }
        .sheet(isPresented: $isShowingSheet) {
            TitledButton(title: "Sheet",
                        action: dismissSheet,
                        label: "Dismiss Sheet",
                        backgroundColor: .orange)
        }
    }

    private func showSheet() {
        isShowingSheet = true
    }
    private func dismissSheet() {
        isShowingSheet = false
    }
}

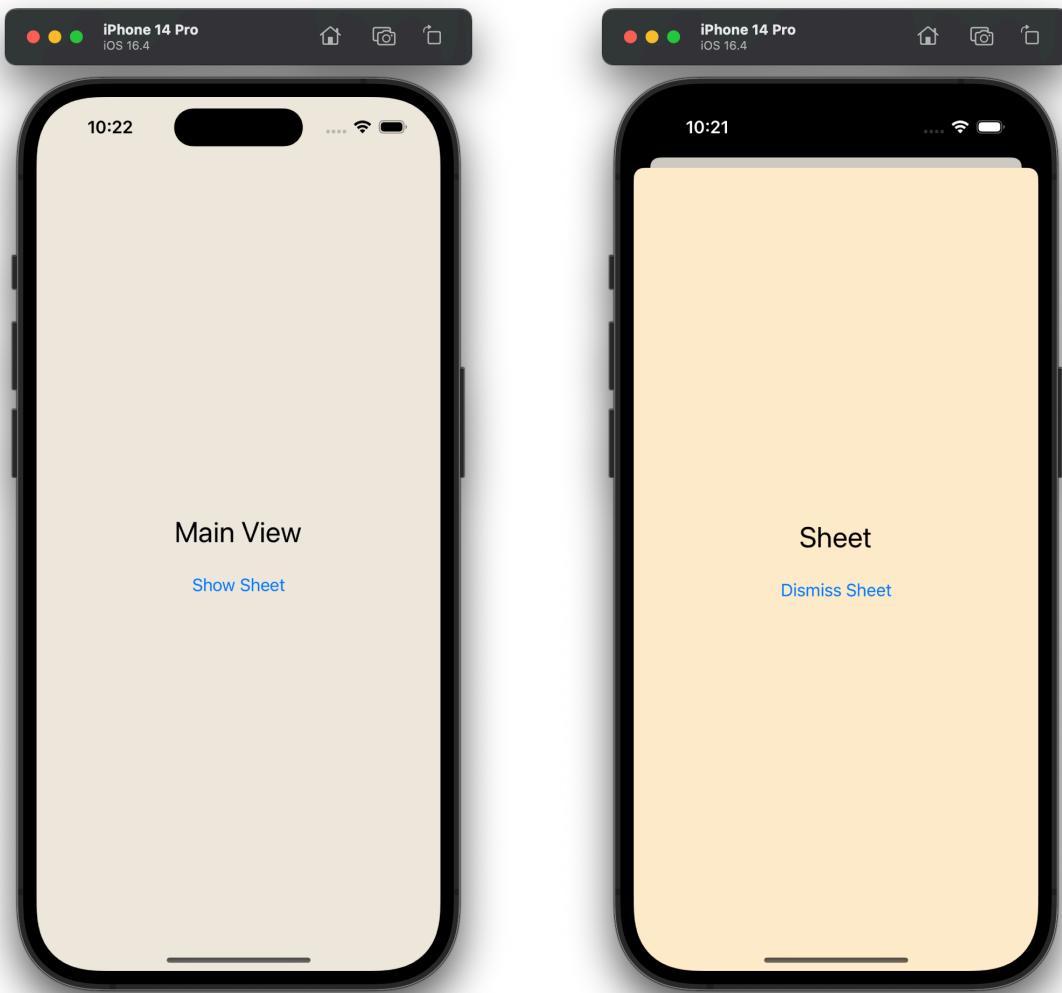
struct TitledButton: View {
    let title: String
    let action: () -> Void
    let label: String
    let backgroundColor: Color

    var body: some View {
        VStack() {
            Text(title)
                .padding()
                .font(.title)
            Button(action: action, label: { Text(label) })
        }
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .background(backgroundColor.opacity(0.2))
    }
}

```

Sheet Example – 2

Presenting a Sheet



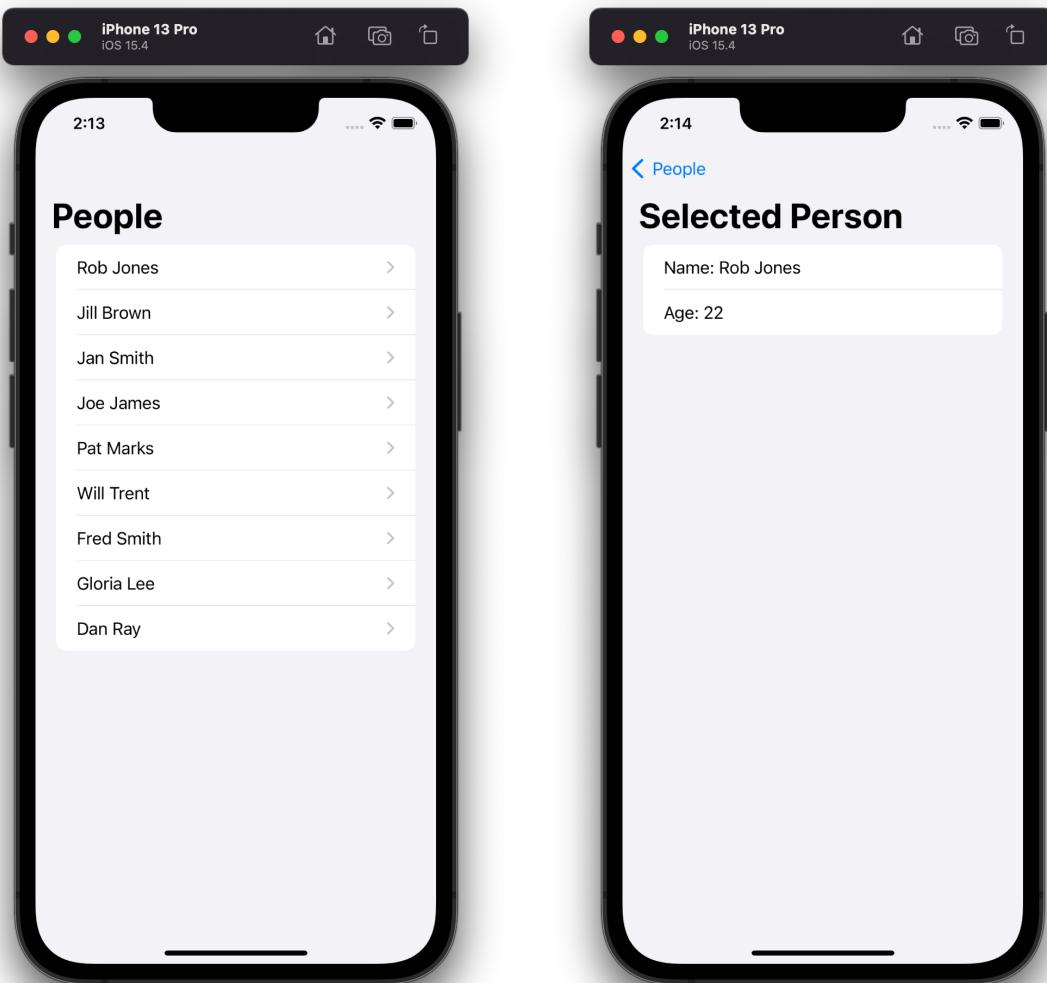
Navigation Stacks

- A NavigationStack wraps its content with a navigation bar and provides navigation behavior on behalf of its nested view.
- To implement navigation, include one or more instances of NavigationLink inside the navigation view's content.
- Nested container views can add a navigationTitle(:_) modifier to provide a title to be presented in the navigation bar
- (Note that a navigation stack's content doesn't have to be a List, though that's what's shown in the following example.)

```
struct Navigation_1: View {  
    @StateObject private var viewModel = PeopleViewModel()  
  
    var body: some View {  
        NavigationStack {  
            List(viewModel.people, id: \.name) { person in  
                NavigationLink(person.name) {  
                    Form {  
                        Text("Name: \(person.name)")  
                        Text("Age: \(person.age)")  
                    }  
                    .navigationTitle("Selected Person")  
                }  
            }  
            .navigationTitle("People")  
        }  
    }  
}
```

Navigation Example

A Navigation Stack with Navigation Links



Toolbars

- Use the `toolbar(_:)` view modifier to add content to the navigation bar.
- You can directly add items such as buttons, but for more control, nest the items in instances of `ToolbarItem`, which will allow you to specify each item's placement.
- (Note that you will need a `ForEach` to support editing a list, as shown in the example.)

Toolbars Example

```

struct Navigation_2: View {
    @StateObject private var viewModel = PeopleViewModel()
    var body: some View {
        NavigationView {
            List {
                ForEach(viewModel.people, id: \.name) { person in
                    NavigationLink(person.name) {
                        PersonCell(person: person)
                    }
                }
                .onDelete { indexSet in
                    delete(at: indexSet)
                }
            }
            .navigationTitle("People")
            .toolbar {
                ToolbarItem {
                    EditButton()
                }
                ToolbarItem(placement: .navigationBarLeading) {
                    Button(action: {}, label: { Image(systemName: "plus.circle") })
                }
            }
        }
    }
}

private func add() { // ... }
private func delete(at indexSet: IndexSet) {
    viewModel.people.remove(at: indexSet.first ?? 0)
}
}

struct PersonCell: View {
    let person: Person
    var body: some View {
        Form {
            Text("Name: \(person.name)")
            Text("Age: \(person.age)")
        }
        .navigationTitle("Selected Person")
    }
}

```

SECTION SEVEN

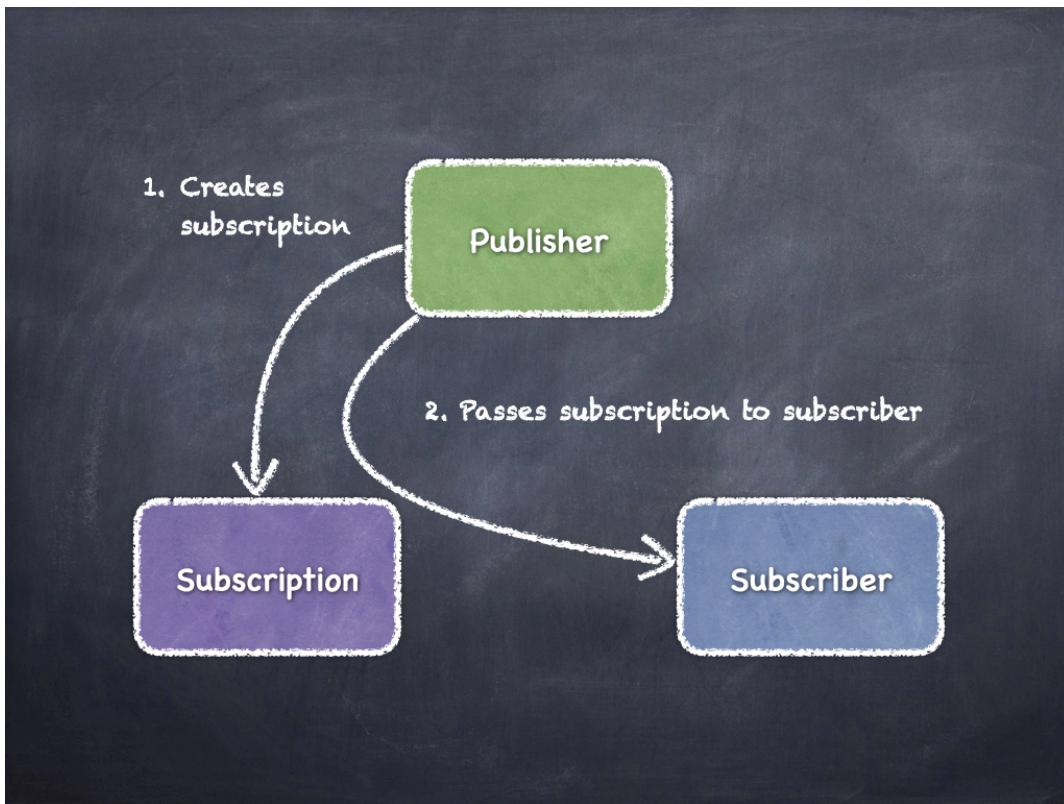
Concurrency

Overview

- The Publisher protocol declares the API for objects that deliver values over time. That allows you to work with the values in a fashion similar to working with a collection.
- Publishers have methods, termed *operators*, that allow them to be chained together, to control the flow of data and perform transformations, as needed.
- Subscribers (objects conforming to the Subscriber protocol) sit at the end of a chain of publishers and act upon the resulting values.
- Note that a publisher only emits values when requested by a subscriber, giving your code control over the pace of the data flow.
- Several framework types expose publishers through their API, including Timer, NotificationCenter, and URLSession.

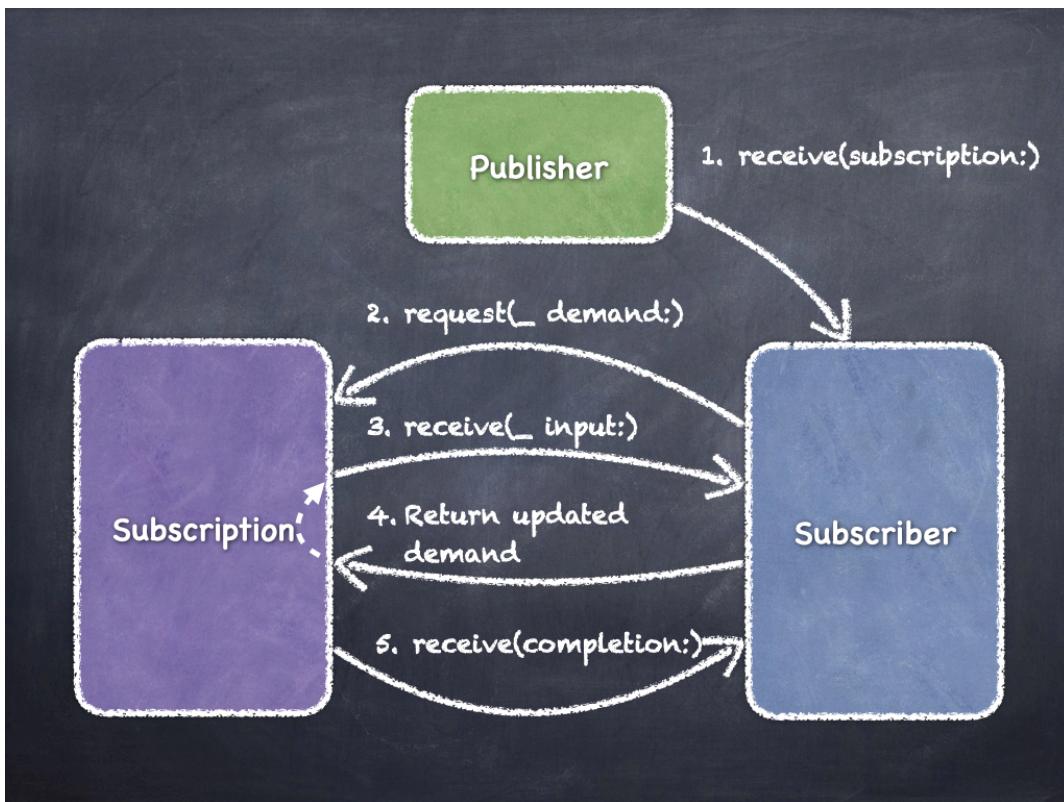
Publishers and Subscribers

- A publisher creates a subscription that ‘pushes’ one or more values to a subscriber.
- The publisher passes the subscriber to the subscription to establish a connection.
- The subscriber also receives a reference to the subscription.
This is important because the subscription must be stored in a property in order to keep it alive.



Subscriptions

- A subscriber requests values from its subscription, passing an initial demand (i.e. how many values it wishes to receive)
- The subscription then calls the subscriber's `receive(_:)` method as many times as necessary (based on demand).
- When finished, the subscription calls the subscriber's `receive(completion:)` method.



Using the sink Operator

- You can use the `sink(receiveCompletion:receiveValue:)` operator to subscribe to a publisher — for example an `@Published` property.
- Note that the return value is a subscription, which must be stored to allow for later cancellation. *If the subscription isn't stored, it's cancelled immediately.*

```
import SwiftUI
import Combine

final class PersonViewModel_4: ObservableObject {
    @Published var person: Person = Person(name: "Fred Smith",
                                             age: 32)

    private var subscriptions: Set<AnyCancellable> = []

    init() {
        $person
            .sink { person in
                print(person.name)
            }
            .store(in: &subscriptions)
    }
}
```

Chaining a map Operator

- The code below adds a name published property to the previous example to allow the Combine operators to work directly with the stream of text.
- It also inserts a `map(_:) operator` in the chain to modify the text on the fly, and then store it back in the `uppercasedName` property.
- The `debounce(for:scheduler:)` operator sets a minimum time period before a publisher can emit an event.

```
final class PersonViewModel_4_1: ObservableObject {
    @Published var person: Person = Person(name: "Jan Brown", age: 33)
    @Published var name: String = "" {
        didSet { person.name = name }
    }
    @Published var uppercasedName: String = ""

    private var subscriptions: Set<AnyCancellable> = []

    init() {
        name = person.name

        $name
            .debounce(for: 1, scheduler: RunLoop.main)
            .map { text in
                text.uppercased()
            }
            .sink { text in
                self.uppercasedName = text
            }
            .store(in: &subscriptions)
    }
}
```

Structured Concurrency

- Structured concurrency is a relatively new set of technologies in Swift that provides support for `async/await` and actors.
- Its goal is to simplify and streamline code that deals with concurrency, while helping developers avoid data races by using tasks as the fundamental unit of concurrency.
- Tasks created via `async let` or task groups are child tasks of an originating task, and can't outlive their parent's lifetime. This provides scoping around concurrency, and thus structure.
- The keyword `await` defines a suspension point. A call to an `async` function must be annotated with `await`. A function that can suspend must be annotated with `async`.
- Compare the code below with the previous Combine unit test example shown in **Loading a URL**.

```
func testFetchQuotesWithAsyncAwait() async throws {
    let req = URLRequest(url: url)
    let (data, response) = try await URLSession.shared.data(for: req)

    if let wrapper = try? JSONDecoder().decode(QuotesWrapper.self,
                                                from: data) {
        print(response, "\n", wrapper.quotes)
        XCTAssertFalse(wrapper.quotes.isEmpty)
    } else {
        XCTFail("Unable to fetch quotes from url \(url)")
    }
}
```

Using async/await – 1

- Typical code for fetching a thumbnail image using closures:

```
func fetchThumbnail(for id: String,
                    completion: @escaping (UIImage?, Error?) -> Void){
    let request = thumbnailURLRequest(for: id)
    let size = CGSize(width: 40, height: 40)
    let task = URLSession.shared.dataTask(with: request) {
        data, response, error in

        if let error = error {
            completion(nil, error)
        } else if (response as? HTTPURLResponse)?.statusCode != 200 {
            completion(nil, FetchError.badID)
        } else {
            guard let image = UIImage(data: data!) else {
                // Easy to forget this...
                completion(nil, FetchError.badImage)
                return
            }
            image.prepareThumbnail(of: size) { thumbnail in
                guard let thumbnail = thumbnail else {
                    // Easy to forget this...
                    completion(nil, FetchError.badImage)
                    return
                }
                completion(thumbnail, nil)
            }
        }
    }
    task.resume()
}
```

Using async/await – 2

- The method below can simply return the thumbnail image or throw an error.
- Easier to write the implementation correctly, and easier to write the calling code.

```
func fetchThumbnail(for id: String) async throws -> UIImage {  
    let request = thumbnailURLRequest(for: id)  
    let (data, response) = try await URLSession.shared.data(  
        for: request)  
  
    guard (response as? HTTPURLResponse)?.statusCode == 200 else {  
        throw FetchError.badID  
    }  
  
    let image = UIImage(data: data)  
    guard let thumbnail = await image?.thumbnail else {  
        throw FetchError.badImage  
    }  
  
    return thumbnail  
}
```

Calling an async Method

- The code below shows how to use a Task to call an async method from a synchronous context — in this case, some SwiftUI code.

```
struct ThumbnailView: View {  
    var viewModel: ViewModel  
    @State private var image: UIImage?  
  
    var body: some View {  
        if let image = self.image {  
            Image(uiImage: image)  
                .onAppear {  
                    Task {  
                        self.image = try? await self.viewModel  
                            .fetchThumbnail(for: "X")  
                    }  
                }  
        }  
    }  
}
```