Why SwiftUI?

Declarative UI Programing

Why
Swift UI

Declarative UI Programming

SILICON VALLEY
CODE CAMP

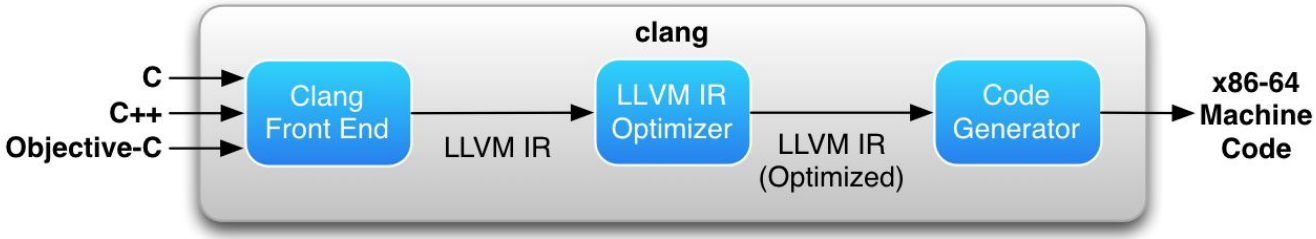
Android Pre iPhone

"As a consumer I was blown away. I wanted one immediately. But as a Google engineer, I thought 'We're going to have to start over,'" Googler Chris DeSalvo is quoted as saying. "What we had suddenly looked just so ... '90s [...] It's just one of those things that are obvious when you see it."

iOS First

What can we learning from iOS

- iPhone Hardware
- App Store
- Human Interface Guidelines (HIG) [Material DeSign]
- Dark Theme
- Xcode with AutoLayout [Constraint Layout]
- **Swift [Kotlin]**
- **SwiftUI [Android Jetpack Compose] (Declarative UI)**



Evolution of a new Language

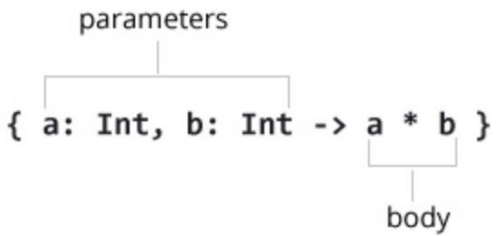
Objective C rap! ⟶ Swift

```
- (void)letsMakeASandwich:(void (^)(void))sandwichIngredientsBlock
{
    NSLog(@"Bread");
    sandwichIngredientsBlock();
    NSLog(@"Bread");
}
```

Closure Expression Syntax

Closure expression syntax has the following general for

```
{ (parameters) -> return type in
    statements
}
```



Removed Objective-C / NS Code / UIkit

Declarative vs Imperative Programming

Declarative programming:

is a programming paradigm … that expresses the logic of a computation without describing its control flow.

Or

Declarative Programming is like asking your friend to draw a landscape. You don't care how they draw it, that's up to them.

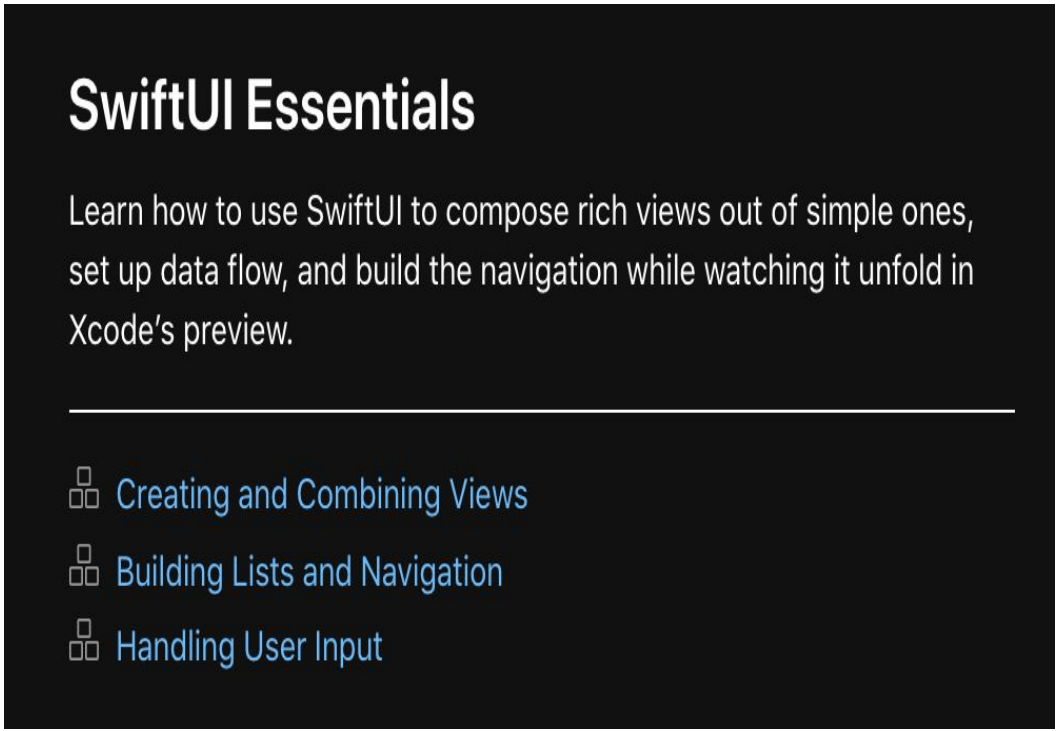
Imperative programming:

is a programming paradigm that uses statements that change a program's state.

Or

Imperative Programming is like your friend listening to Bob Ross tell them how to paint a landscape. While good ole Bob Ross isn't exactly commanding, he is giving them step by step directions to get the desired result.

Declarative UI programming



Building the UI

- Build a Basic UI
- Connect the UI to Code
- Work with View Controllers
- Implement a Custom Control
- Define Your Data Model

Working with Table Views

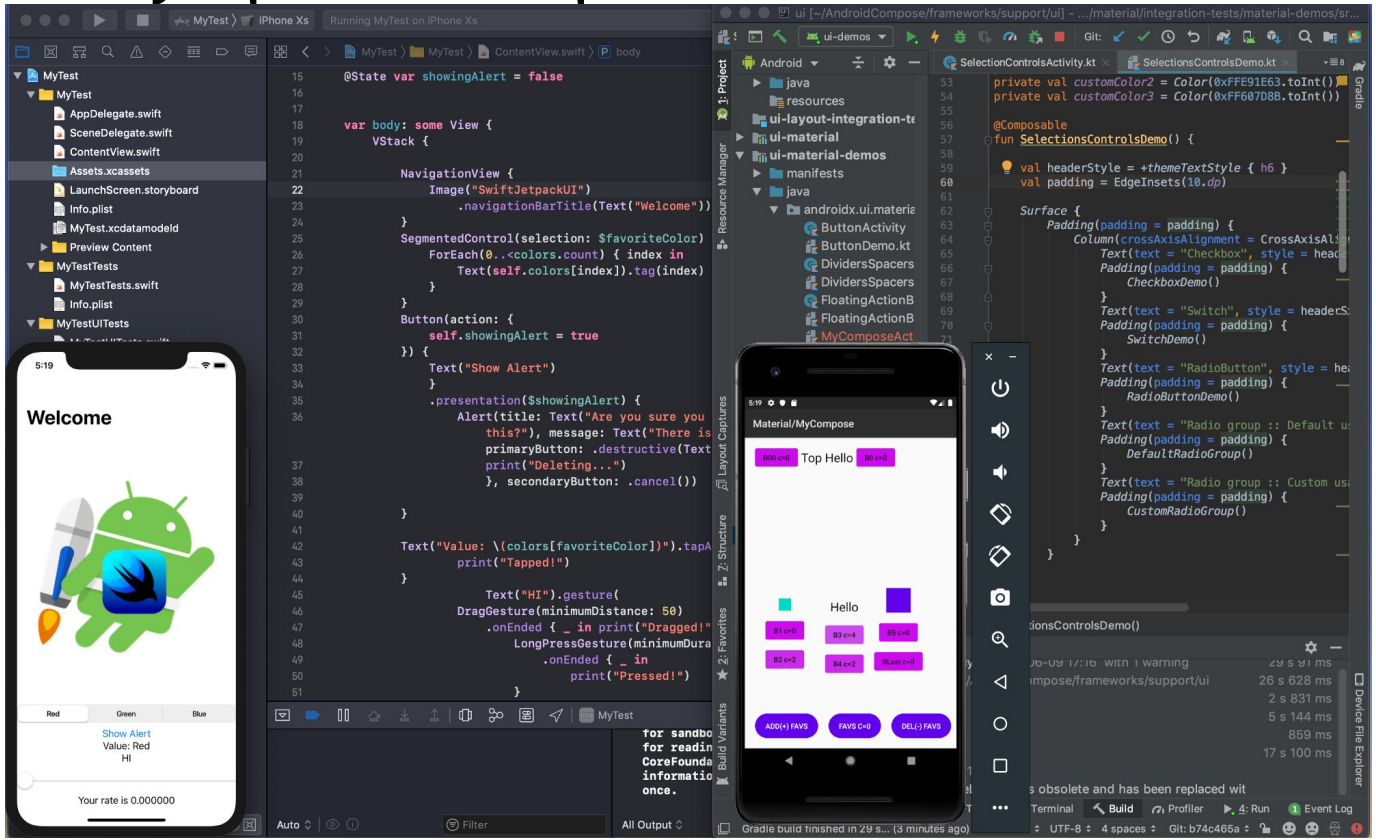
- Create a Table View
- Implement Navigation
- Implement Edit and Delete Behavior
- Persist Data

Kotlin is Swift

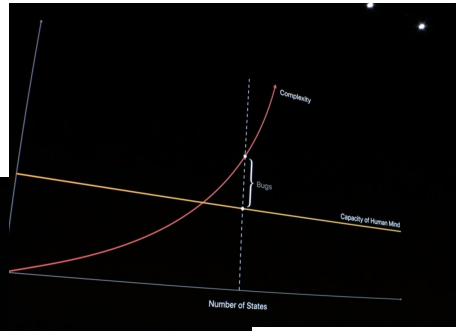
```
fun greet(name: String, day: String): String {
    return "Hello $name, today is $day."
}
greet("Bob", "Tuesday")

class Shape {
    var numberOfSides = 0
    fun simpleDescription() =
        "A shape with $numberOfSides sides."
}

var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

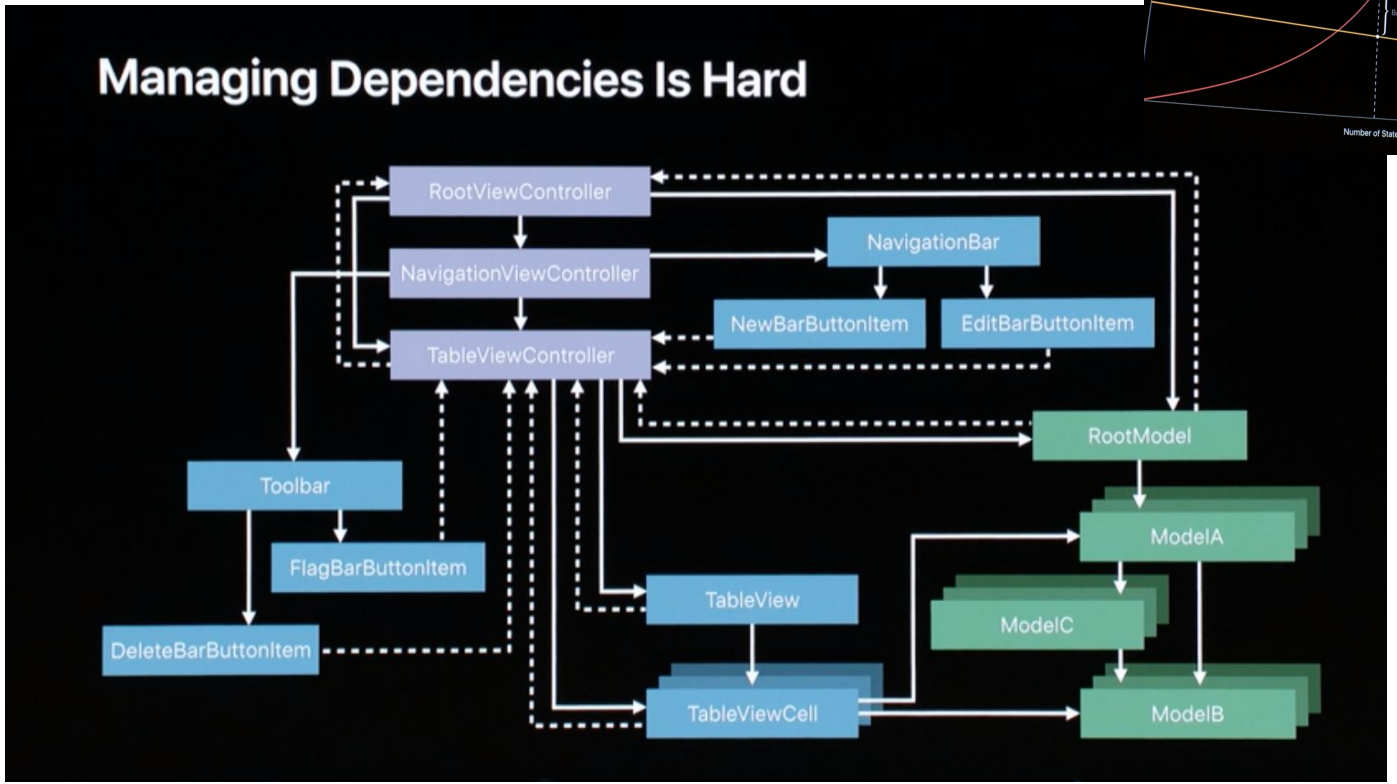
```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", day: "Tuesday")

class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}

var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

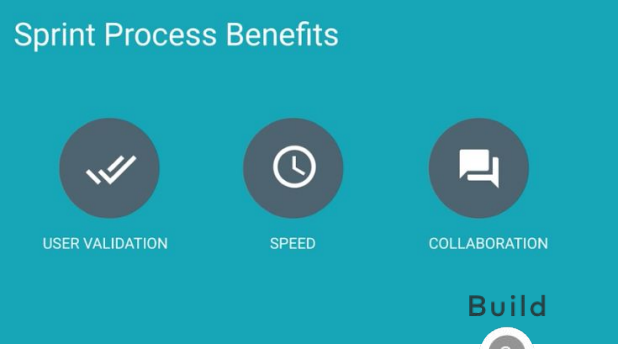
Android Jetpack Compose

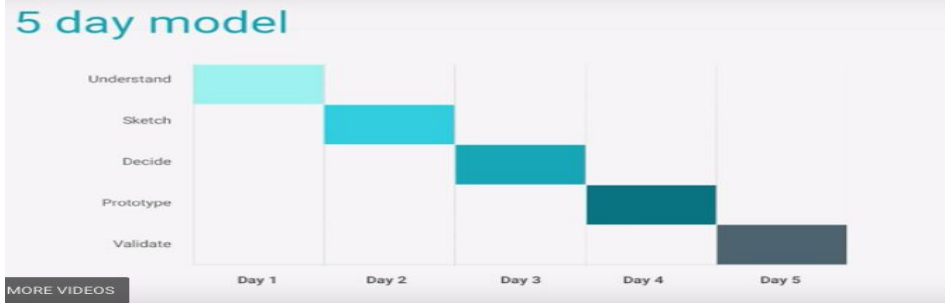


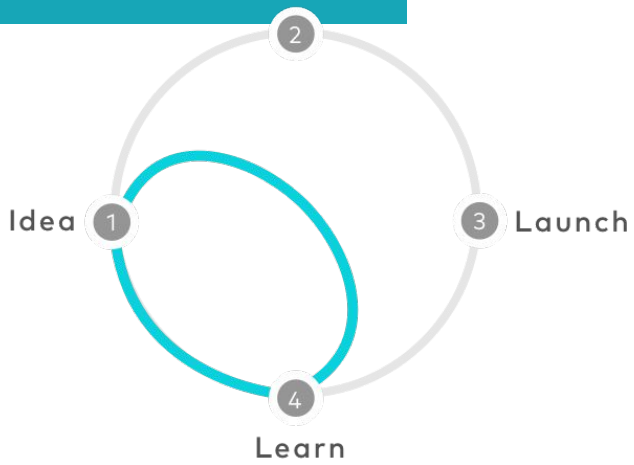
Building @ Speed of Thought

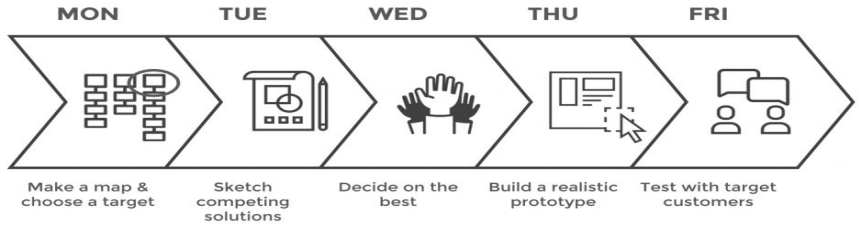


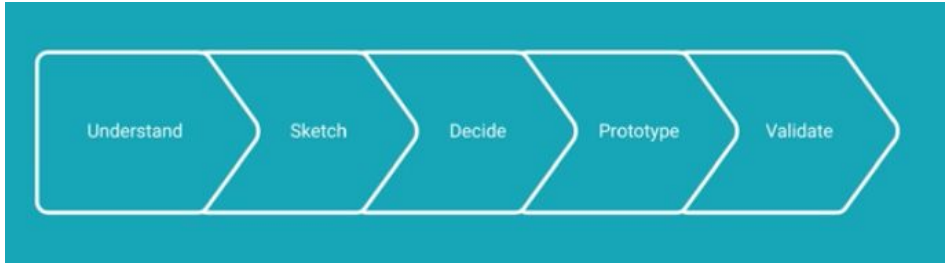






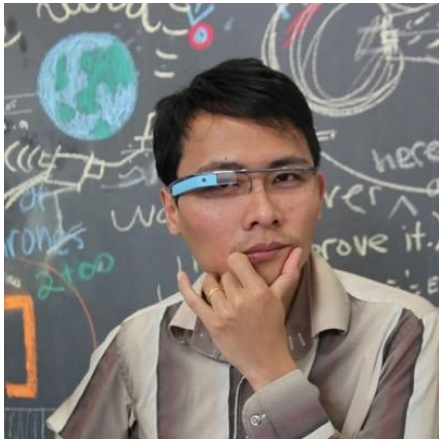






DeSign Sprint Process

https://www.gv.com/sprint/



Tom Chi - Home & Away Teams

Love of SwiftUI

SwiftUI is Swift

1. Only in Swift
2. Using language constructs for UI
3. Simple to understand
4. Simple to use
5. Remove legacy NS
6. Reactive
7. Declarative

Swift 5.2(5.x ?) features in SwiftUI

- **Opaque return types** — generic protocols can now be used as return types. But still strongly typed
- **Omitted return keywords** — return keyword can now be omitted for single-expression functions.
- **Function builders** — enables the builder pattern to be implemented using closures.
- **Property Wrapper** — access with additional behavior. Property values can be automatically wrapped using specific types such as views. Also makes it much easier to define various kinds of bindable properties.

SwiftUI View (using 5.1Language Features)

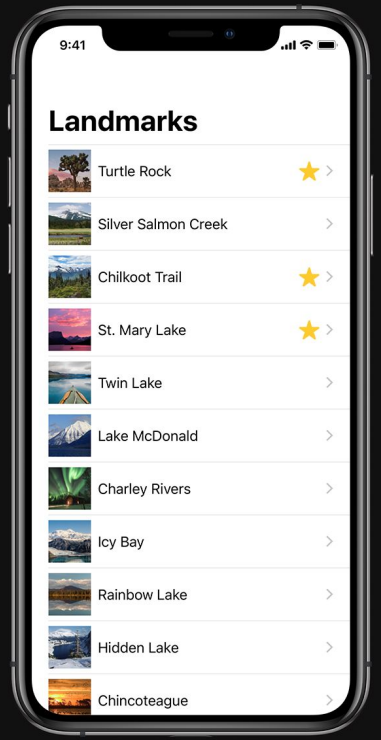
```
struct ContentView: View {
    @State var showSubtitle: Bool   ← property wrapper

    var body: some View {           ← opaque return type
    VStack {                        ← trailing closure Function Builder
      Image(uiImage: image)
      Text(title)
      Toggle(isOn: $showSubtitle) {
          Text(subtitle)
      }
     }                              ← omit return type
    }
}
```

```swift
struct HeaderView: View {
    let image: UIImage
    let title: String
    let subtitle: String

    var body: some View {
        var builder = VStackBuilder()
        builder.add(Image(uiImage: image))
        builder.add(Text(title))
        builder.add(Text(subtitle))
        return builder.build()
    }
}
```

Describe your layout just once.

Declare the content and layout for any state of your view. SwiftUI knows when that state changes, and updates your view's rendering to match.

```
List(landmarks) { landmark in
    HStack {
        Image(landmark.thumbnail)
        Text(landmark.name)
        Spacer()

        if landmark.isFavorite {
            Image(systemName: "star.fill")
                .foregroundColor(.yellow)
        }
    }
}
```

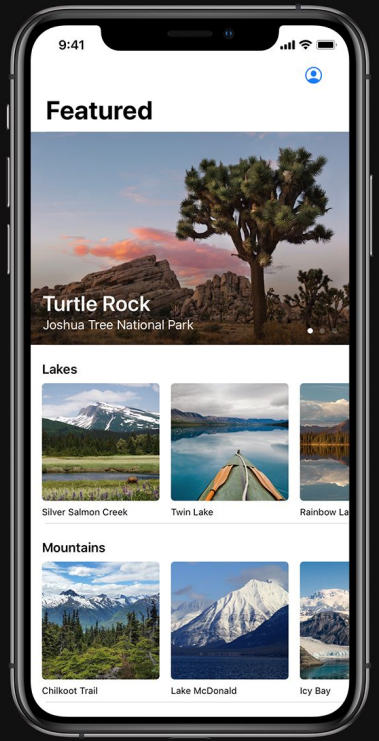


Apple Foldable Phone/Tablet

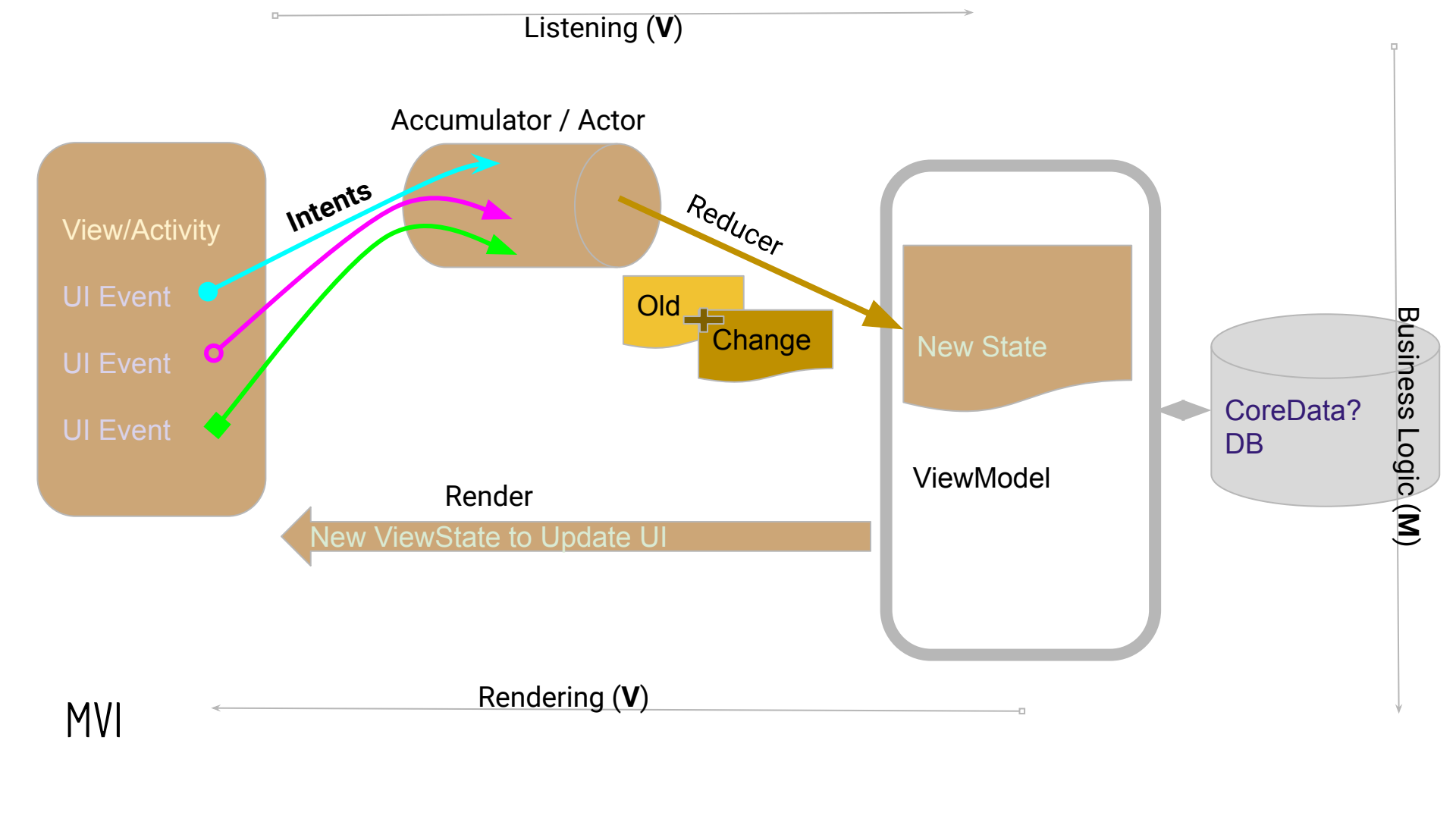
Build reusable components.

Combine small, single-responsibility views into larger, more complex interfaces. Share your custom views between apps designed for any Apple platform.

```
struct FeatureCard: View {
    var landmark: Landmark

    var body: some View {
        landmark.featureImage
            .resizable()
            .aspectRatio(3/2, contentMode: .fit)
            .overlay(TextOverlay(landmark))
    }
}
```



And One UI to Rule Them All !!!


Listening (V)
Accumulator / Actor
View/Activity
UI Event
UI Event
UI Event
Intents
Reducer
Old
Change
New State
ViewModel
CoreData?
DB
Business Logic (M)
Render
New ViewState to Update UI
Rendering (V)
MVI

Transition to SwiftUI

1. Tools
2. Building UIs
3. Handel Data
4. Working with existing (legacy) code

Key Features:

- Declarative
- Automatic
- Compositional

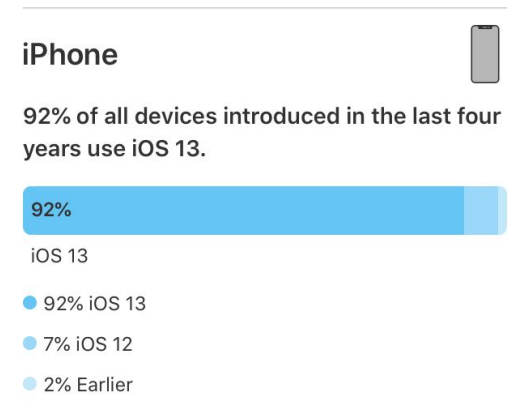
OS & Tools

SwiftUI runs on:

iOS 13+

macOS 10.15+

tvOS 13+ & watchOS 6+ ONLY



Benefits of Swift for UI

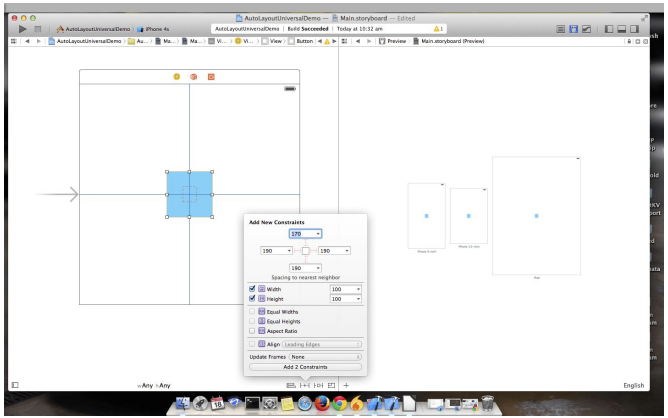
- Programmatic and storyboard-based design at the same time.
- No storyboard XML issues with **source control!**
- User interface gets checked by the Swift compiler.

<u>UIKit class names to SwiftUI names:</u>

UI[*name*] → *name*

UISlider is now Slider

UIButton is now Button



Architecture of the System

1. ~~**AppDelegate.swift.**~~ This is responsible for **monitoring external events**, such as if another app tries to send you a file to open.
2. ~~**SceneDelegate.swift**~~. This is responsible for **managing the way your app is shown**, such as letting multiple instances run at the same time or taking action when one moves to the background.
3. ~~**ContentView.swift.**~~ This is our **initial piece of user interface**. If this were a UIKit project, this would be the `ViewController` class that Xcode gave us.
4. **Assets.xcassets**. This is an asset catalog, which stores all the **images and colors** used in our project.
5. ~~**LaunchScreen.storyboard.**~~ This is the screen that gets shown while your app is loading.
6. **Info.plist is a property list file,** which in this instance is used to store system-wide settings for our app – what name should be shown below its icon on the iOS home screen, for example.
7. A group called **Preview Content**, which contains another asset catalog called Preview Assets.

Canvas Tools

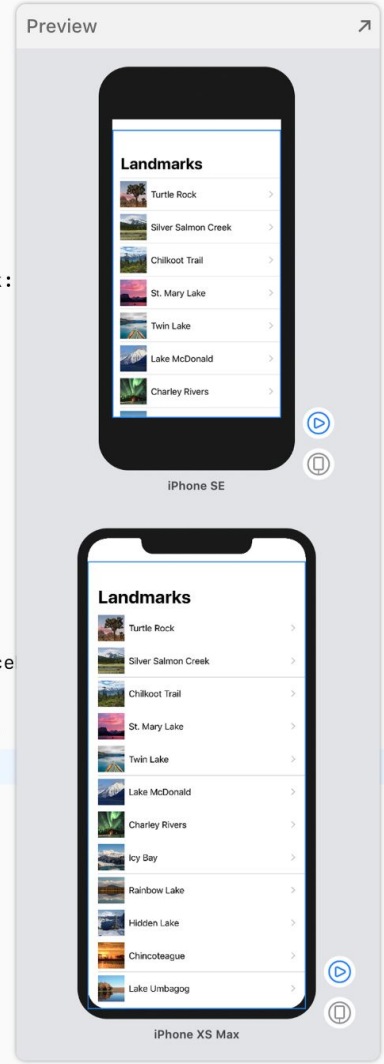
XCode Preview with the Canvas

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading) {
            Text("Turtle Rock")
                .font(.title)
            Text("Joshua Tree National Park")
                .font(.subheadline)
        }
    }
}

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

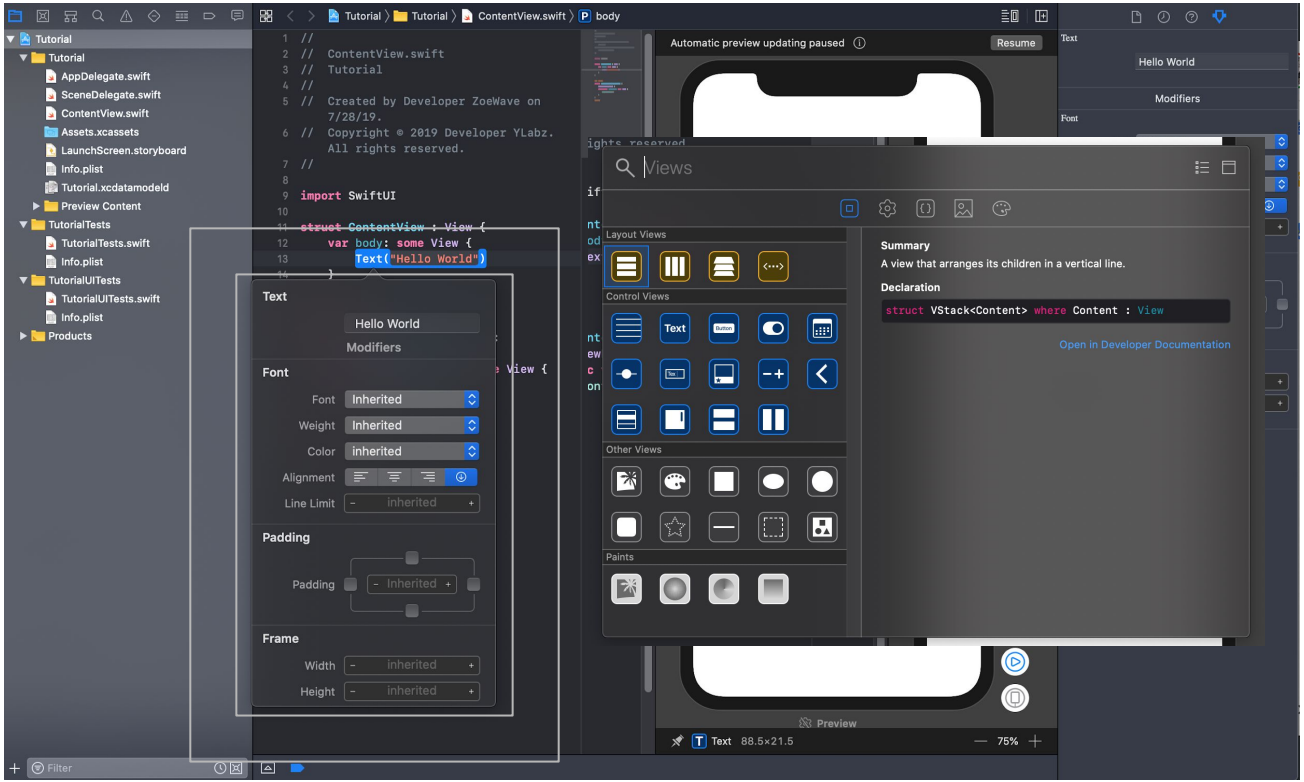
```swift
struct ContentView: View {
    var body: some View {
        Text("Hello SwiftUI!")
    }
}
```

Explore the Canvas

```
16  struct LandmarkList_Previews: PreviewProvider {
17      static var previews: some View {
18          ForEach(["iPhone SE", "iPhone XS Max"], id: \.sel
19              LandmarkList()
20                  .previewDevice(PreviewDevice(rawValue: de
21                  .previewDisplayName(deviceName)
22          }
23      }
24  }
```



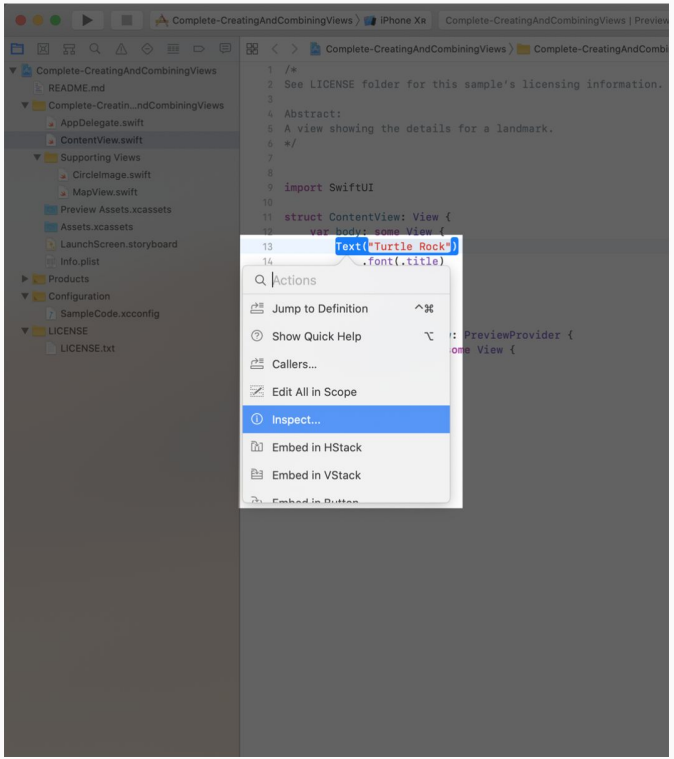
Exploring the Options (Discoverability)



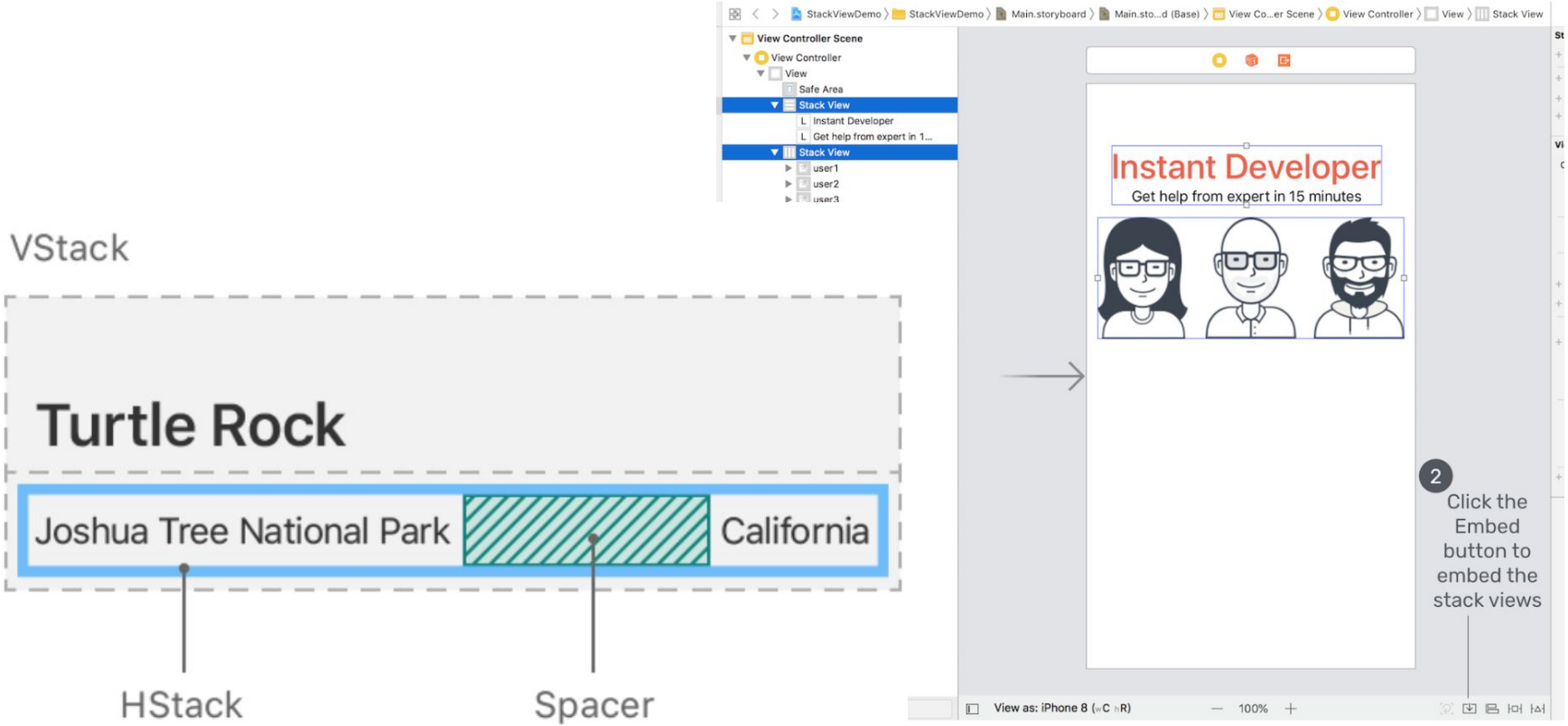
Text View & Stacks

Customize the Text View

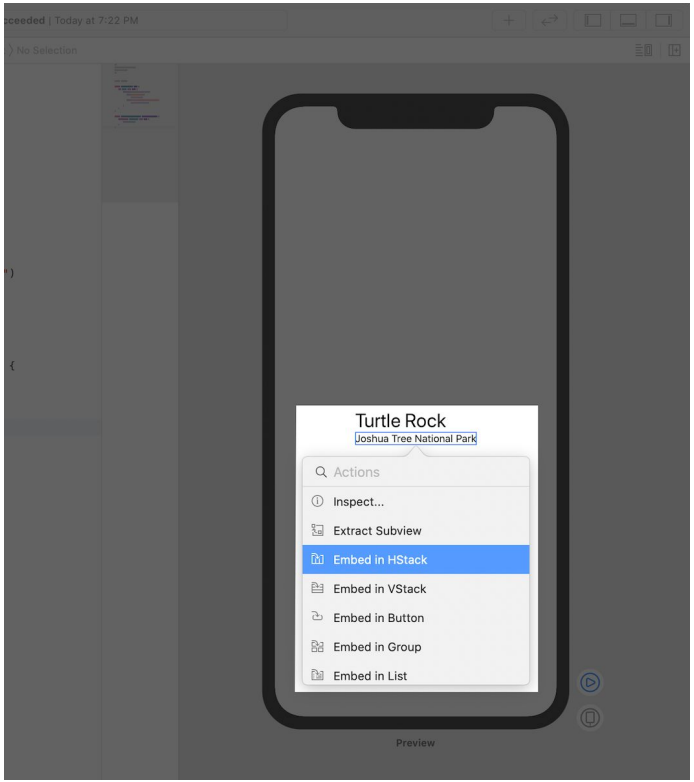
```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Turtle Rock")
            .font(.title)
            .color(.green)
    }
}

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



Combine Views Using Stacks



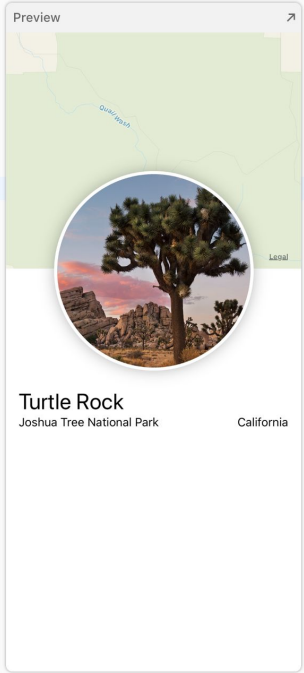
Stacks are the backbone of components



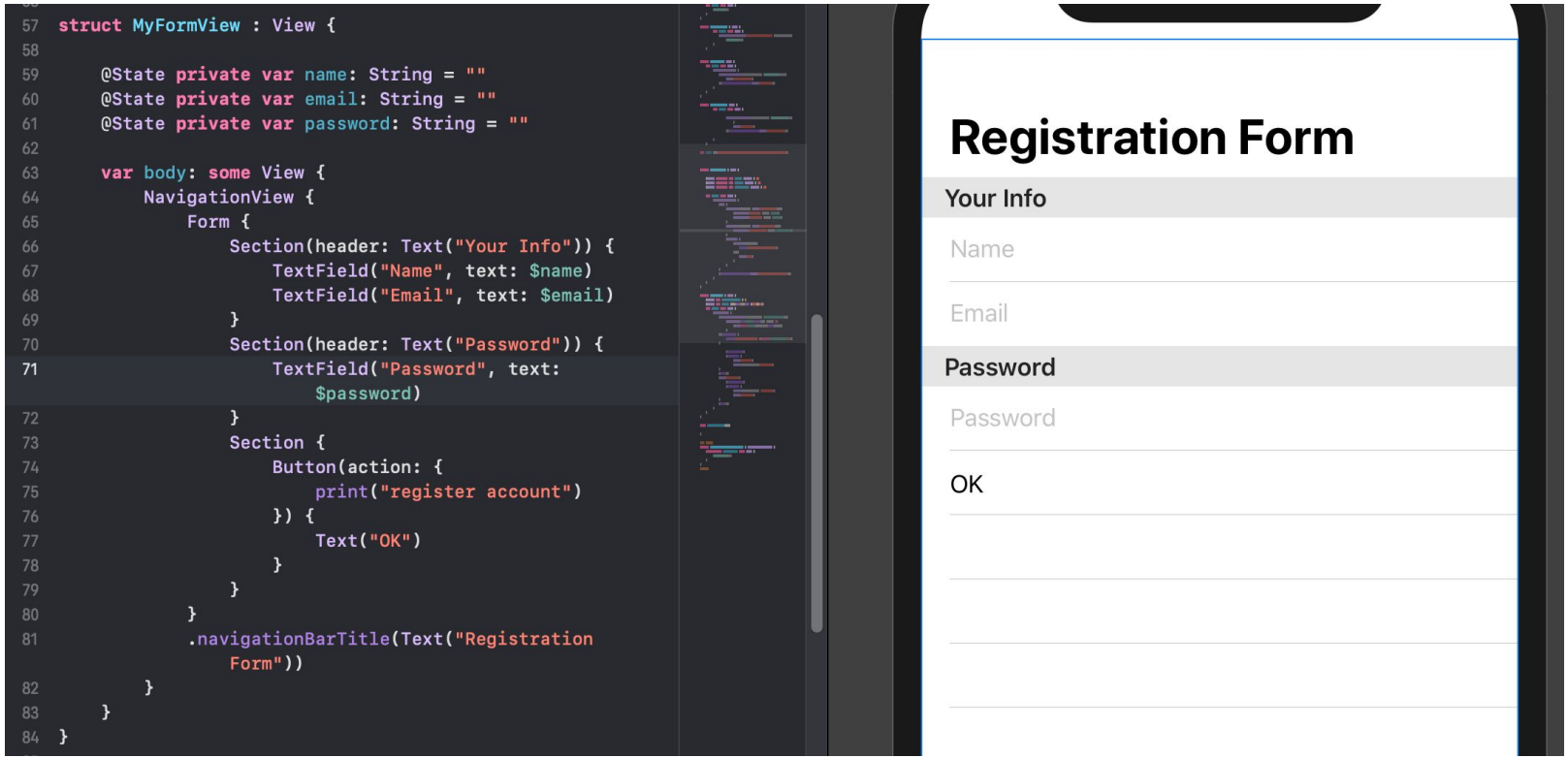
ContentView.swift

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            MapView()
                .edgesIgnoringSafeAreaView(.top)
                .frame(height: 300)

            CircleImage()
                .offset(y: -130)
                .padding(.bottom, -130)

            VStack(alignment: .leading) {
                Text("Turtle Rock")
                    .font(.title)
                HStack(alignment: .top) {
                    Text("Joshua Tree National Park")
                        .font(.subheadline)
                    Spacer()
                    Text("California")
                        .font(.subheadline)
                }
            }
            .padding()

            Spacer()
        }
    }
}
```

Preview



Forms

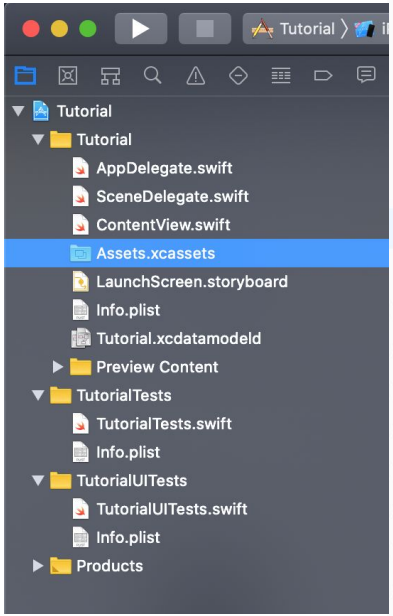

struct MyFormView : View {
@State private var name: String = ""
@State private var email: String = ""
@State private var password: String = ""
var body: some View {
NavigationView {
Form {
Section(header: Text("Your Info")) {
TextField("Name", text: $name)
TextField("Email", text: $email)
}
Section(header: Text("Password")) {
TextField("Password", text:
$password)
}
Section {
Button(action: {
print("register account")
}) {
Text("OK")
}
}
}
.navigationBarTitle(Text("Registration
Form"))
}
}
}
Registration Form
Your Info
Name
Email
Password
Password
OK

Groups

```swift
//  Created by 晋先森 on 2019/6/9.
//  Copyright © 2019 晋先森. All rights reserved.
//

import SwiftUI

struct GroupPage : View {

    var body: some View {
        VStack {
            Group {
                Text("Hello World !")
                Text("Hello World !")
                }
                .padding(5)
                .border(Color.orange.gradient,
                        width: 1,
                        cornerRadius: 5)
            Group {
                Text("Hello World !")
                Text("Hello World !")
                }.padding(5)
                .border(Color.black.gradient,
                        width: 1,
                        cornerRadius: 5)
            Group {
                Text("Hello World !")
                Text("Hello World !")
                }.padding(5)
                .border(Color.purple.gradient,
                        width: 1,
                        cornerRadius: 5)
            }.navigationBarTitle(Text("Group"))
    }
}
```

https://jinxiansen.github.io/Swif

Create a Custom View

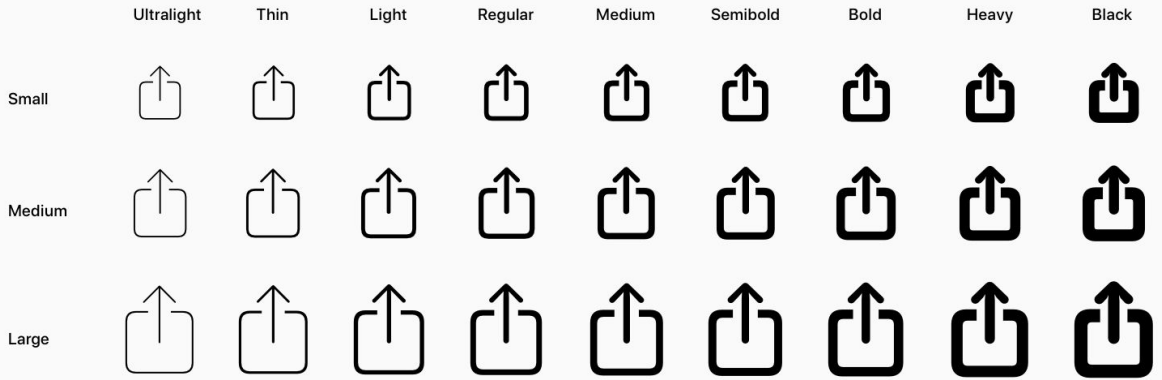



```swift
import SwiftUI

struct CircleImage: View {
    var body: some View {
        Image("turtlerock")
            .clipShape(Circle())
            .overlay(
                Circle().stroke(Color.white, lineWidth: 4))
            .shadow(radius: 10)
    }
}

struct CircleImage_Preview: PreviewProvider {
    static var previews: some View {
        CircleImage()
    }
}
```

SF Symbols

SF Symbols

SF Symbols provides a set of over 1,500 consistent, highly configurable symbols you can use in your app. Apple designed SF Symbols to integrate seamlessly with the San Francisco system font, so the symbols automatically ensure optical vertical alignment with text for all weights and sizes. SF Symbols are available in a wide range of weights and scales to help you create adaptable designs.

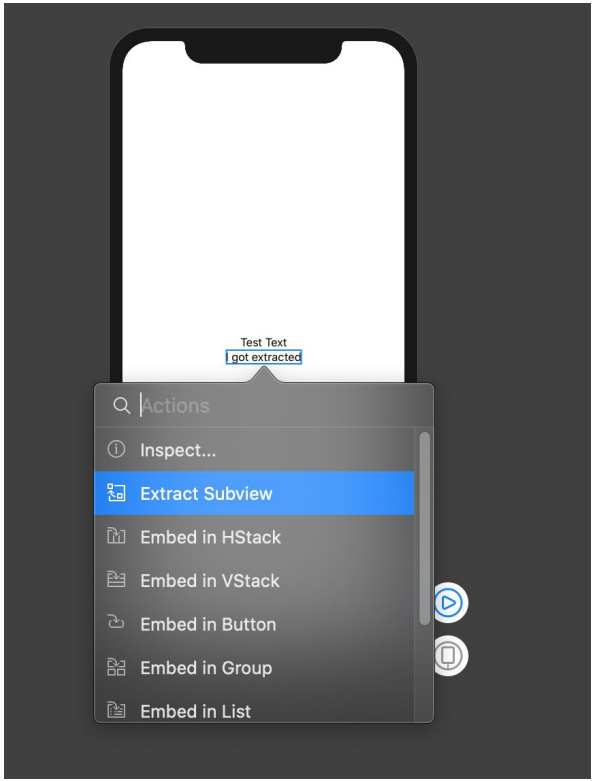


	phone.down.circle	Phone app
	phone.down.circle.fill	Phone app
	teletype	Teletype feature
	realtimetext	Real-time text feature
	video	FaceTime app
	video.fill	FaceTime app
	video.circle	FaceTime app
	video.circle.fill	FaceTime app
	video.slash	FaceTime app
	video.slash.fill	FaceTime app
	video.badge.plus	FaceTime app
	video.badge.plus.fill	FaceTime app
	arrow.up.right.video	FaceTime app
	arrow.up.right.video.fill	FaceTime app
	arrow.down.left.video	FaceTime app
	arrow.down.left.video.fill	FaceTime app
	questionmark.video	FaceTime app
	questionmark.video.fill	FaceTime app
	questionmark.video.rtl	FaceTime app
	questionmark.video.fill.rtl	FaceTime app
	envelope	Mail app
	envelope.fill	Mail app
	envelope.circle	Mail app
	envelope.circle.fill	Mail app

Reusable Components

Extract Subview / Embed in View

```swift
//  iDevExample
//
//  Created by Developer ZoeWave on 8/4/19.
//  Copyright © 2019 Developer YLabz. All rights
    reserved.
//

import SwiftUI

struct ContentView : View {
    let msg : String
    let myText : Text

    var body: some View {
    VStack {
        Text("\(msg)")
        ExtractedView()
        }
    }
}

struct ExtractedView : View {
    var body: some View {
        return Text("I got extracted")
    }
}


#if DEBUG
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        ContentView(msg: "Test Text", myText:
            Text("test"))
    }
}
```



Navigation

Navigation Between Views

A NavigationLink wraps whatever content we desire with a gesture recognizer. When the user taps on that content, the user is taken to the view specified as the destination.

```
struct ContentView : View {
    var body: some View {
        NavigationView {
            NavigationLink(destination: DetailView()) {
                Text("Click")
            }.navigationBarTitle(Text("Navigation"))
        }
    }
}
```

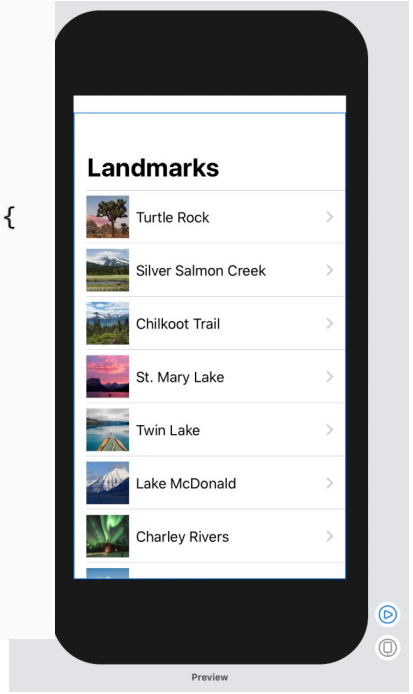
Navigation Link with Data (Prepare for Segway)

```
11 struct HomeView : View {
12     var body: some View {
13         NavigationView {
14             VStack {
15                 Text("Welcome Home")
16                 NavigationLink(destination: ContentView(msg: "Test Text", myText: Text("test"))) {
17                     Button("Press me") {print("I got pressed")}
18                     Text("press Me")
19                 }
20                 .navigationBarTitle("Home")
21             }
22         }
23     }
24 }
25
```

```
11 struct ContentView : View {
12     let msg : String
13     let myText : Text
14     var body: some View {
15
16     VStack { Text("\(msg)")
17             myText}
18     }
19 }
```

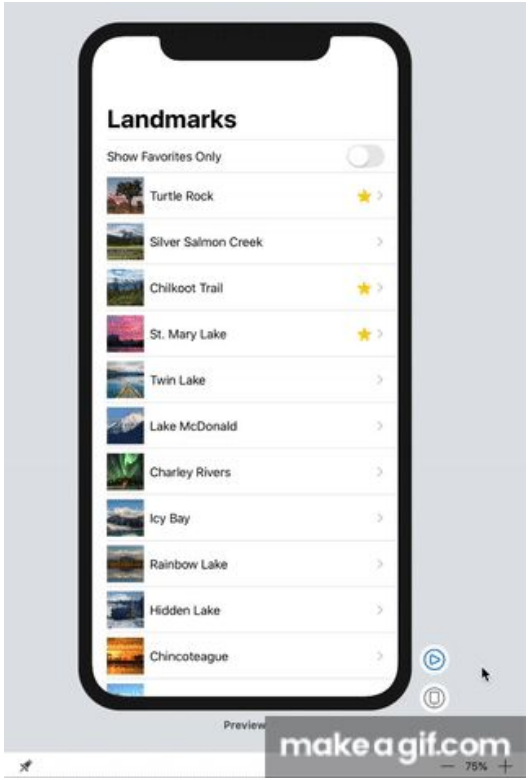
Building Lists and Navigation (UITableViewController)

```
3  struct LandmarkList: View {
4      var body: some View {
5          NavigationView {
6              List(landmarkData) { landmark in
7                  NavigationLink(destination: LandmarkDetail(landmark: landmark)) {
8                      LandmarkRow(landmark: landmark)
9                  }
10             }
11             .navigationBarTitle(Text("Landmarks"))
12         }
13     }
14 }
15
```



Simple Logic to UI

```swift
import SwiftUI

struct LandmarkList: View {
    @State var showFavoritesOnly = false

    var body: some View {
        NavigationView {
            List(landmarkData) { landmark in
                if !self.showFavoritesOnly || landmark.isFavorite {
                    NavigationLink(destination: LandmarkDetail(landmark: landmark)) {
                        LandmarkRow(landmark: landmark)
                    }
                }
            }
            .navigationBarTitle(Text("Landmarks"))
        }
    }
}

struct LandmarkList_Previews: PreviewProvider {
    static var previews: some View {
        LandmarkList()
    }
}
```



NavigationButton is Deprecated (change fast)

NavigationButton

A button that triggers a navigation presentation when pressed.

Declaration

```
struct NavigationButton<Label, Destination> where Label : View, Destination :
    View
```

SDKs

iOS 13.0–13.0
Deprecated

Mac Catalyst 13.0–13.0
Deprecated

tvOS 13.0–13.0
Deprecated

watchOS 6.0–6.0
Deprecated

Xcode 11.0–11.0
Deprecated

Animation

Animations and Views

```
// Animated Changes

struct PlayButton : View {
    @Binding var isPlaying: Bool

    var body: some View {
        Button(action: {
            withAnimation { self.isPlaying.toggle() }
        }) {
            Image(systemName: isPlaying ? "pause.circle" : "play.circle")
        }
    }
}
```

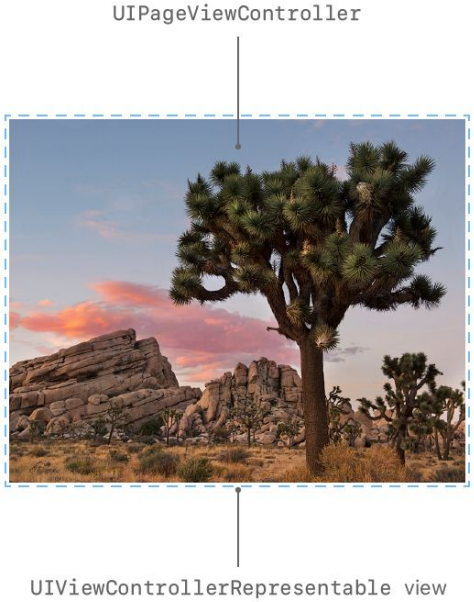
Framework Integration

~~UIKit and SwiftUI Views~~

Section 1

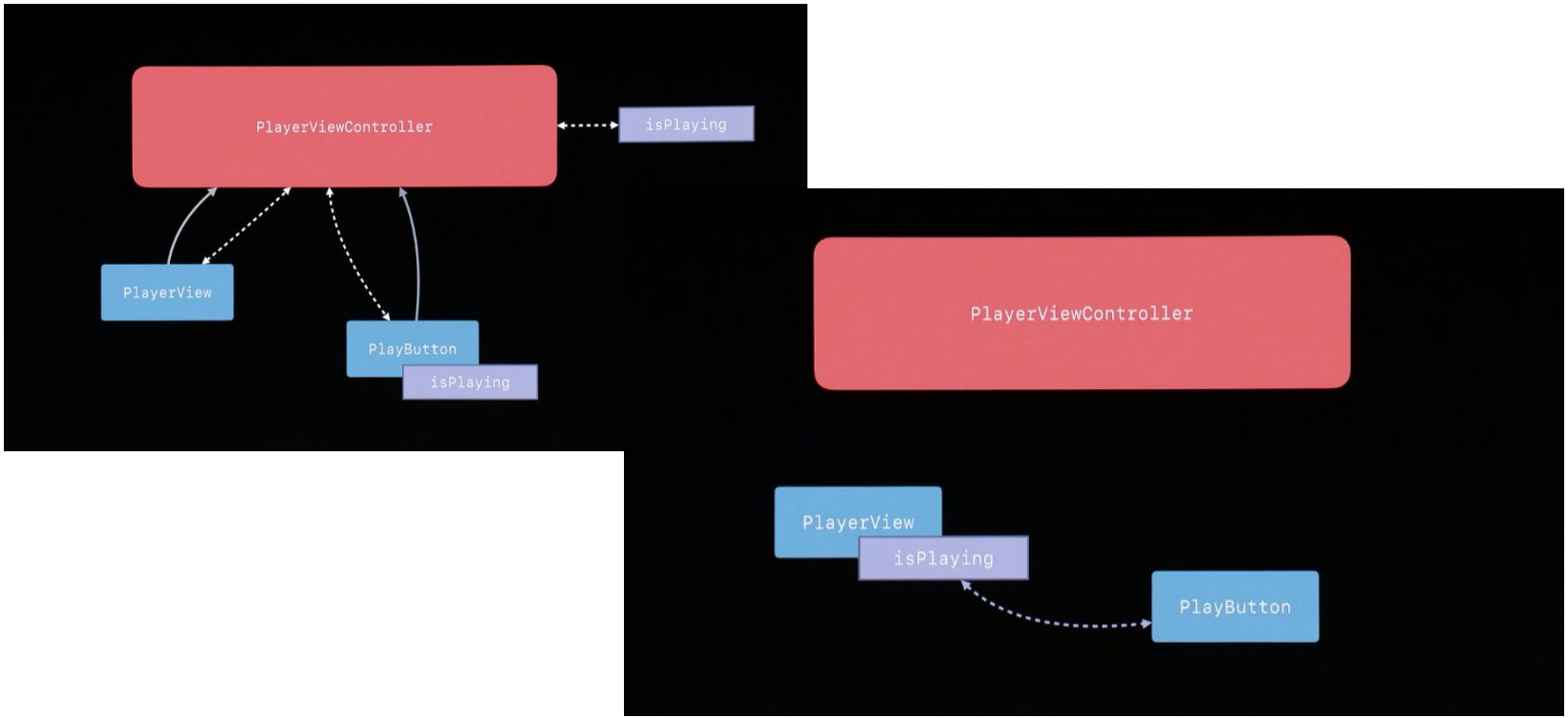
Create View to Represent a UIPageViewController

To represent UIKit views and view controllers in SwiftUI, you create types that conform to the `UIViewRepresentable` and `UIViewControllerRepresentable` protocols. Your custom types create and configure the UIKit types that they represent, while SwiftUI manages their life cycle and updates them when needed.

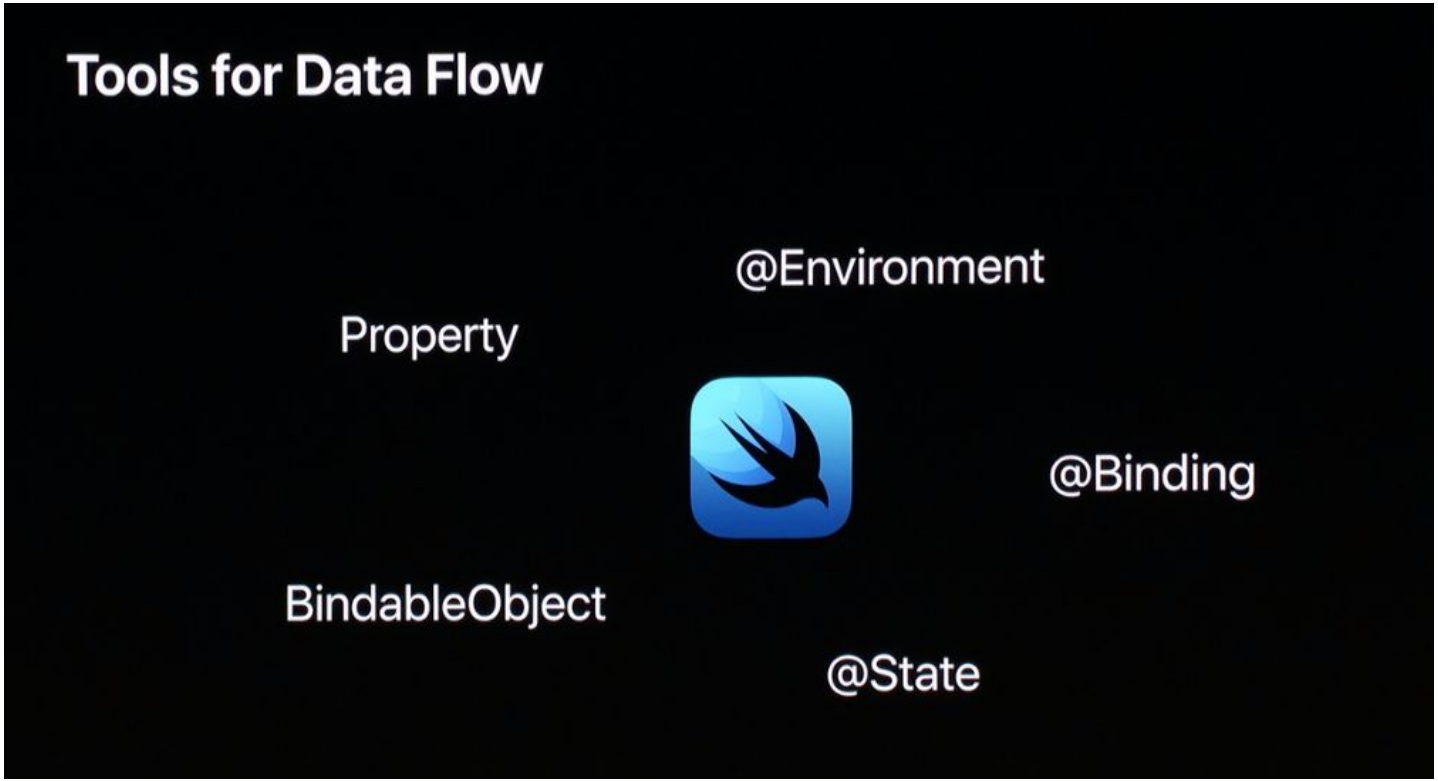


Data Flow & State

MVVM not MVC with Reactive Programming



Managed Data and State



State & variable management require only 2 steps

Define your data (single source of truth) as your model.

- For local managed storage, Use @State
- For data that you manage, just conform to the **BindableObject** Protocol and setup the publisher (from Combine Framework)

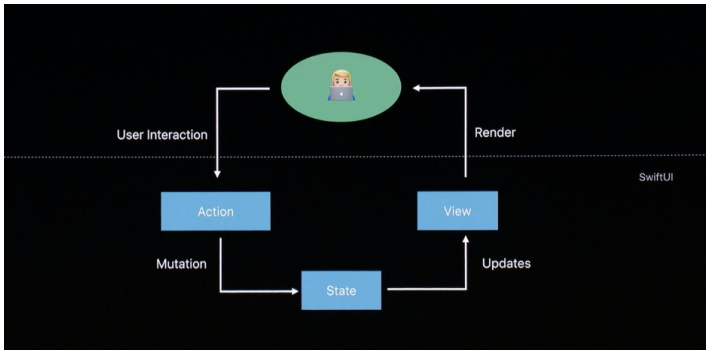
Setup a dependency on the model

- When you build your view, pass in the reference to the model.

**SwiftUI will keep track of the synchronization for you!*

Key points to remember

1. Any time we access data that is a dependency.
2. We should only have one **single source of truth**. Sometimes to accomplish this we need to lift the data up to a common ancestor.
3. Views have lots of different functions: Layout, Visual Effects, Navigation, ***Views are for Data***, Gestures, Drawing and Animation.
4. All data flow is in one direction.



@State Property Wrapper

- Every @State is a source of truth
- Views are a function of state, NOT a result of a sequence of events
- Any change rebuilds the View and all its children
- Mark @State private so you remember it is bound to the current view.

```
struct HikeView: View {
var hike: Hike
@State private var showDetail = false
...
   Button(action: {
      withAnimation {self.showDetail.toggle()}
   }
```

@Binding Property Wrapper

- Derivable form @State passed from the parent
- Use the “$” to pass the binding reference otherwise we will get a copy.
- Framework lets you decide where you keep your data

```
struct myView : View {
    @Binding var passInData: String

@State private var someName: String = "hi" // create your data
// pass the data to the view
MyView(someName : $someName) // create a binding & pass it down

}
```

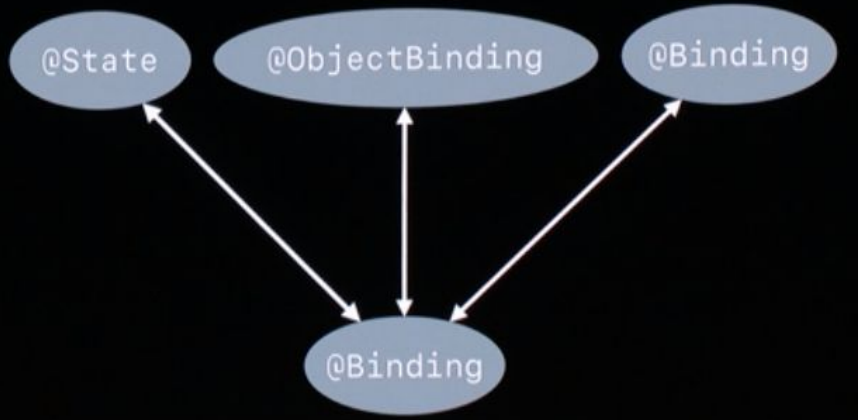
Data Binding

@Binding

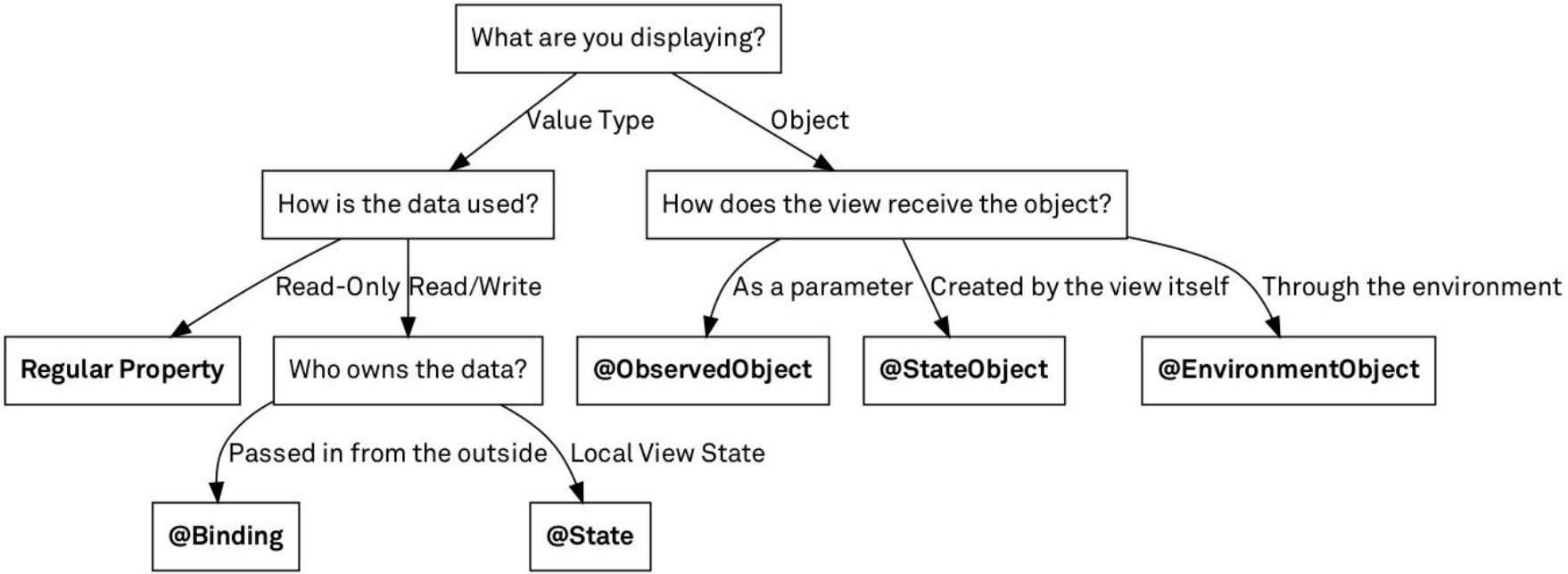
First class reference to data

Great for reusability

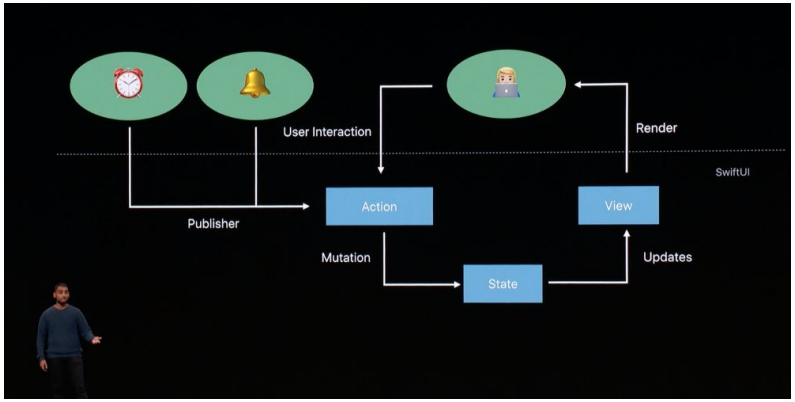
Use $ to derive from source



Let's Explain



This draft by Chris Eidhof from objc.io nicely summarizes the rules and decisions outlined on this page.



Working with External Changes and Data

Combine Framework

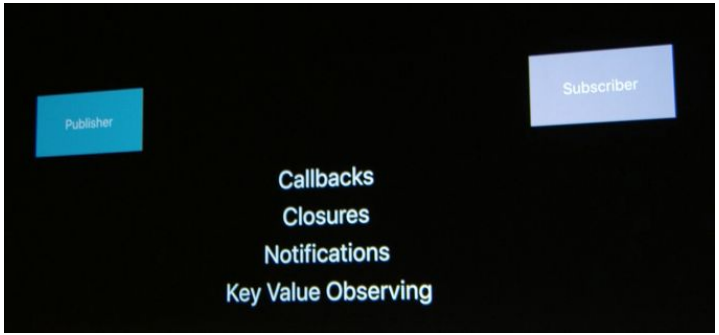
Asynchronous Programming with Swift

Combine is a declarative Swift framework for processing events and values received over time: UI events, network responses and other unpredictable types of data events can easily be published and consumed with the Combine framework.

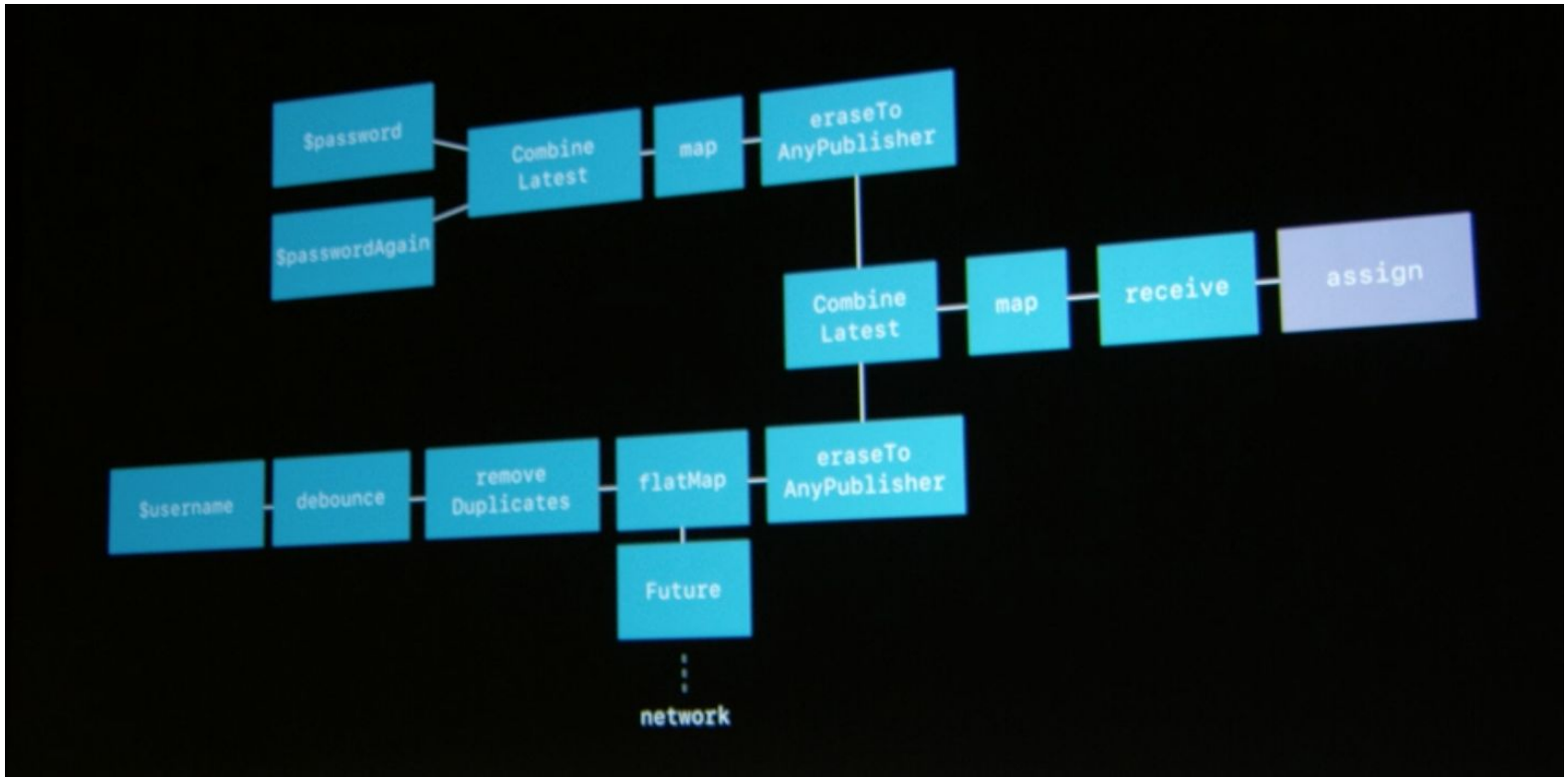
Using Combine

Joseph Heck – Version 0.5, 2019-07-30

https://heckj.github.io/swiftui-notes/



Why Combine is called Combine



Publisher & Subscriber (Rx Programming)

The **Publisher protocol declares a type that can deliver a sequence of values over time**. Publishers have operators to act on the values received from upstream publishers and republish them.

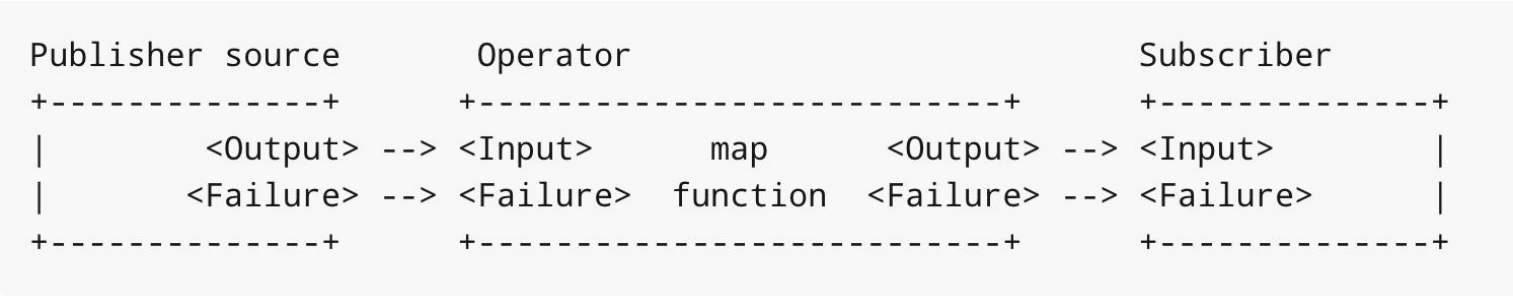
At the end of a chain of publishers, a Subscriber acts on elements as it receives them. Publishers only emit values when explicitly requested to do so by subscribers. This puts your **subscriber code in control of how fast** it receives events from the publishers it’s connected to.

Combine Framework

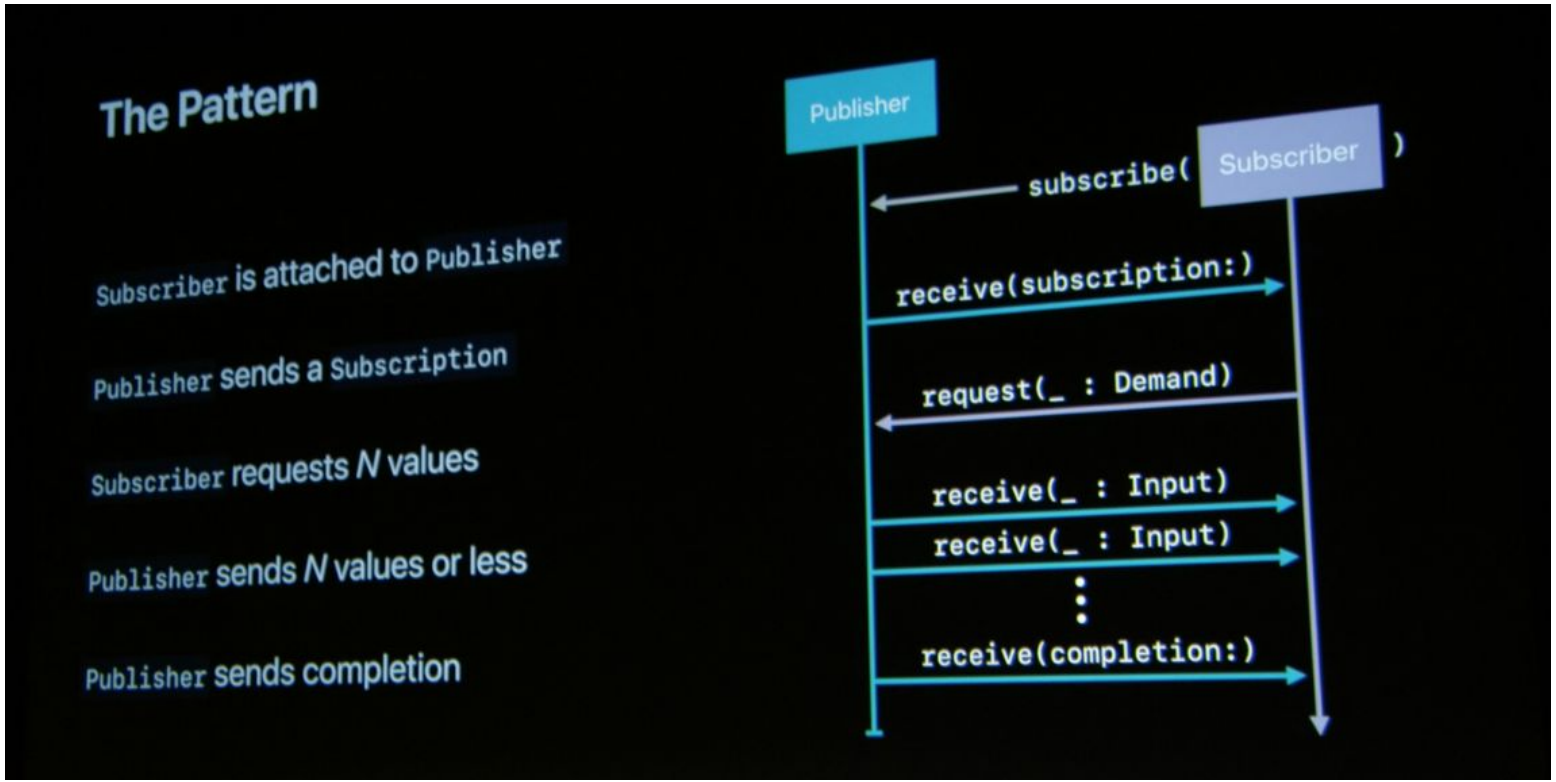
Publishers: is a struct value type that registers a subscriber and declaratively sets how to handle values & errors.

Operators: Is a value type that connects the publisher with the subscriber and describes a behavior for changing values (i.e. from a type to an Int)

Subscriber: is a reference type that receives values .



Combine Flow



WWDC Publisher Example Explained

```
// Using Publishers with Combine

let trickNamePublisher = NotificationCenter.default.publisher(for: .newTrickDownloaded)
    .map { notification in
        return notification.userInfo?["data"] as! Data
    }
    .flatMap { data in
        return Just(data)
            .decode(MagicTrick.self, JSONDecoder())
            .catch {
                return Just(MagicTrick.placeholder)
            }
    }
    .publisher(for: \.name)
    .receive(on: RunLoop.main)
```

String

Never

assertNoFailure

retry

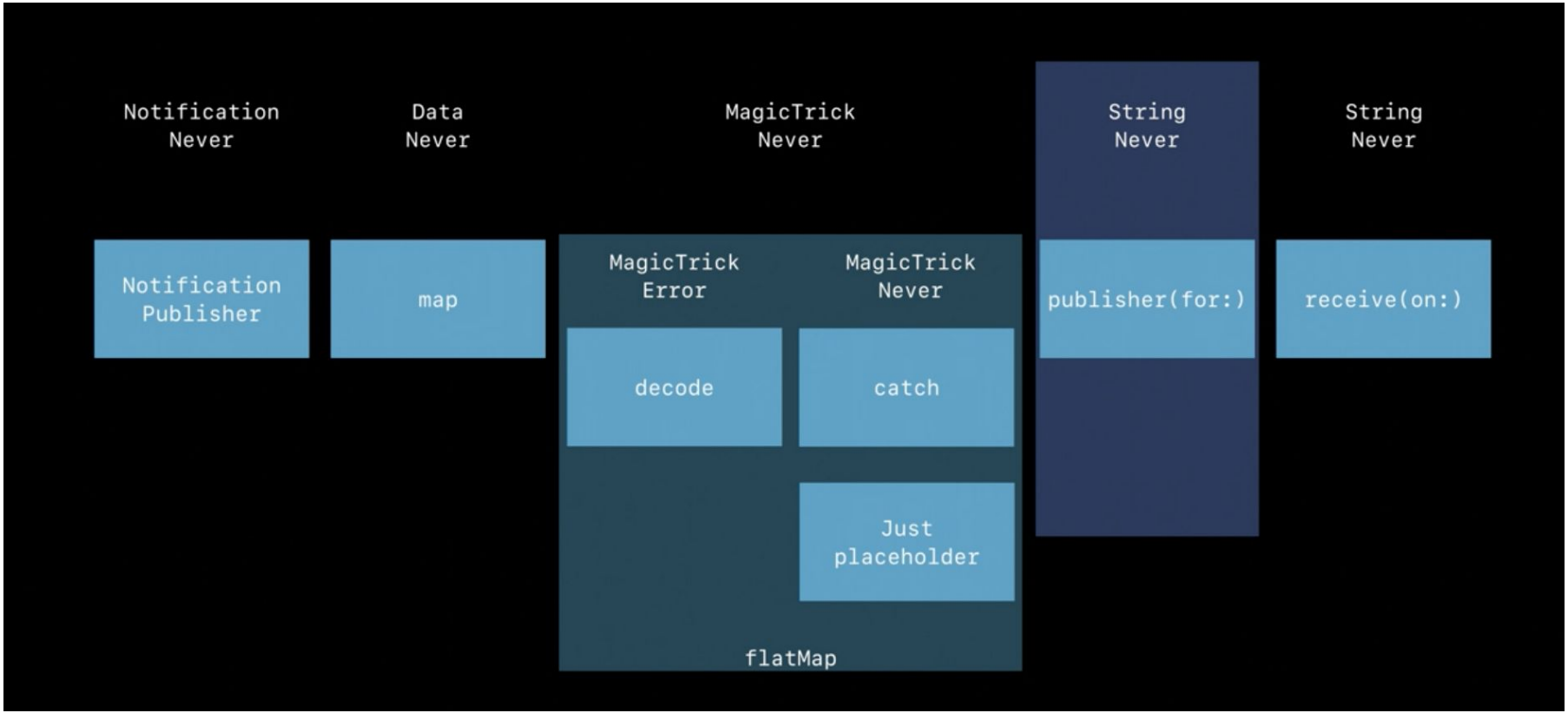
catch

mapError

setFailureType

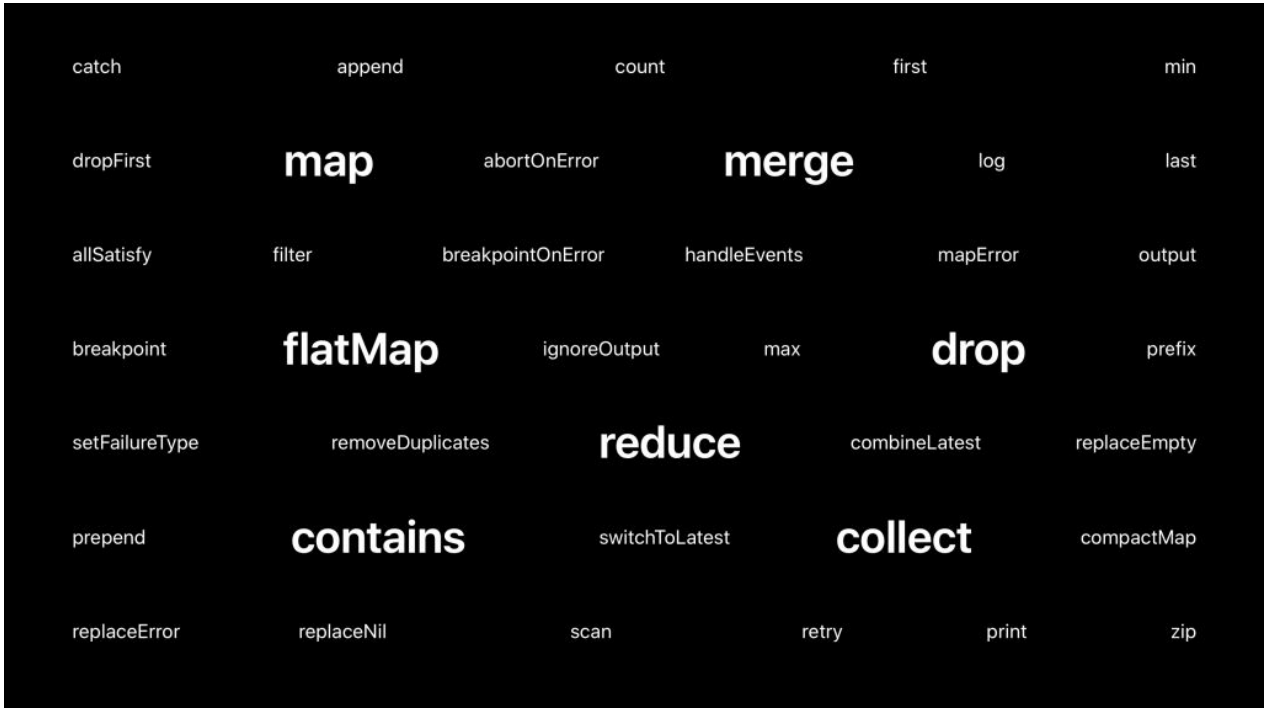
Stacked Return Types

WWDC Example Publisher Flow



Operators

Operators can help with error handling and retry logic, buffering and prefetch, controlling timing, and supporting debugging.



Three rules for Subscribing

1. Subscriber will call receive (subscription) once and only once.
2. The Subscriber will receive zero or more values
3. The Publisher will call one final (optional) completion when done or error

```
protocol Subscriber {
    associatedtype Input
    associatedtype Failure: Error

    func receive(subscription: Subscription)
    func receive(_ value: Subscribers.Demand)
    func receive(completion: Subscribers.Completion<Failure>)
}
```

Built in Subscriber

`assign` applies values passed down from the publisher to an object defined by a keypath. The keypath is set when the pipeline is created. An example of this in swift might look like:

```
.assign(to: \.isEnabled, on: signupButton)
```

`sink` accepts a closure that receives any resulting values from the publisher. This allows the developer to terminate a pipeline with their own code. This subscriber is also extremely helpful when writing unit tests to validate either publishers or pipelines. An example of this in swift might look like:

```
.sink { receivedValue in
    print("The end result was \(String(describing: receivedValue))")
}
```

Subscriber

```
// Using Subscribers with Combine

let trickNamePublisher = ... // Publisher of <String, Never>

let canceller = trickNamePublisher.sink { trickName in
    // Do Something with trickName
}
```

Subjects

A subject is a publisher that you can use to “inject” values into a stream, by calling its `send()` method. This can be useful for adapting existing imperative code to the Combine model.

Subjects / Scheduler

Subjects:

- **PassthroughSubject:** Subscribe to events.
- **CurrentValueSubject:** PassthroughSubject with most recent element.

A **Scheduler** basically defines when and how to execute a closure. It can be really useful when we, for example, want to delay the execution of a specific function. **Publish on main thread.**

Both are also useful for creating publishers from objects conforming to the ***BindableObject*** *protocol within SwiftUI.*

BindableObject Protocol

```
protocol BindableObject {
    associatedtype PublisherType : Publisher where PublisherType.Failure == Never

    var didChange: PublisherType { get }
}
```

Combine SwiftUI Example

```
// Combine with SwiftUI

class WizardModel : BindableObject {
    var trick: WizardTrick { didSet { didChange.send() } }
    var wand: Wand? { didSet { didChange.send() } }

    let didChange = PassthroughSubject<Void, Never>()
}

struct TrickView: View {
    @ObjectBinding var model: WizardModel

    var body: some View {
        Text(model.trick.name)
    }
}
```

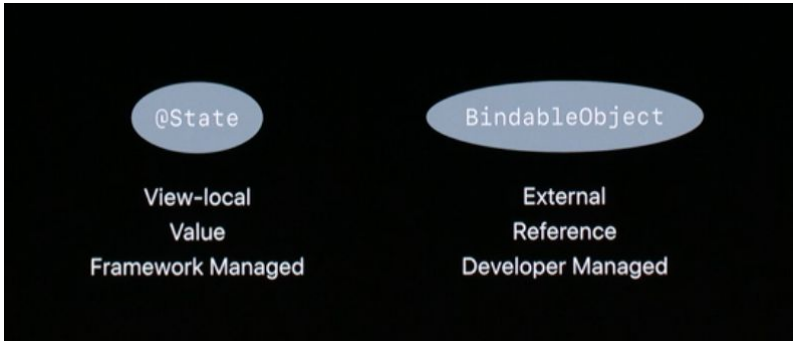
Working with External Data has Two Steps

1. Create a single source of truth to external data with the BindableObject Protocol. This is your data, SwiftUI just needs to know how to react to changes.
2. Add some state / Create a dependency which calls the closure

```
struct PlayerView: View {
      @State private var somethingThatChages : Double = 0.0
      var body: some View{
          Text("\somethingThatChanges")
          .onReceive(MyPublisher.myDataChange { whatChanged in
              self.somethingThatChages = whatChanged
           } // Closure called when things change.
       }
}
```

ObjectBinding Protocol (**@ObjectBinding**)

- External data you own and manage
- Reference type
- Great for the model you already have

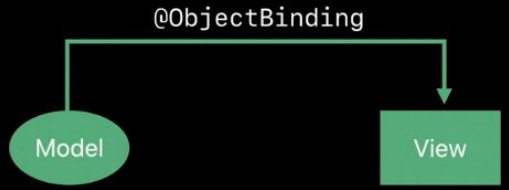


1. Create a source of truth by taking our models and conforming them to bindableObject Protocol
2. When you access the data you create a dependency.

Creating Dependencies on BindableObject

Pass directly with @ObjectBinding

Automatic dependency tracking




```
struct MyView : View {
    @ObjectBinding var model: MyModelObject
    ...
}

MyView(model: modelInstance)
```

```
class MyModelObject : BindableObject {
    var didChange: PassThoughSubject<void,Never>() // <-- Publisher

    func advn() {
        didChange.send() // <-- Send changes
    }

}
```

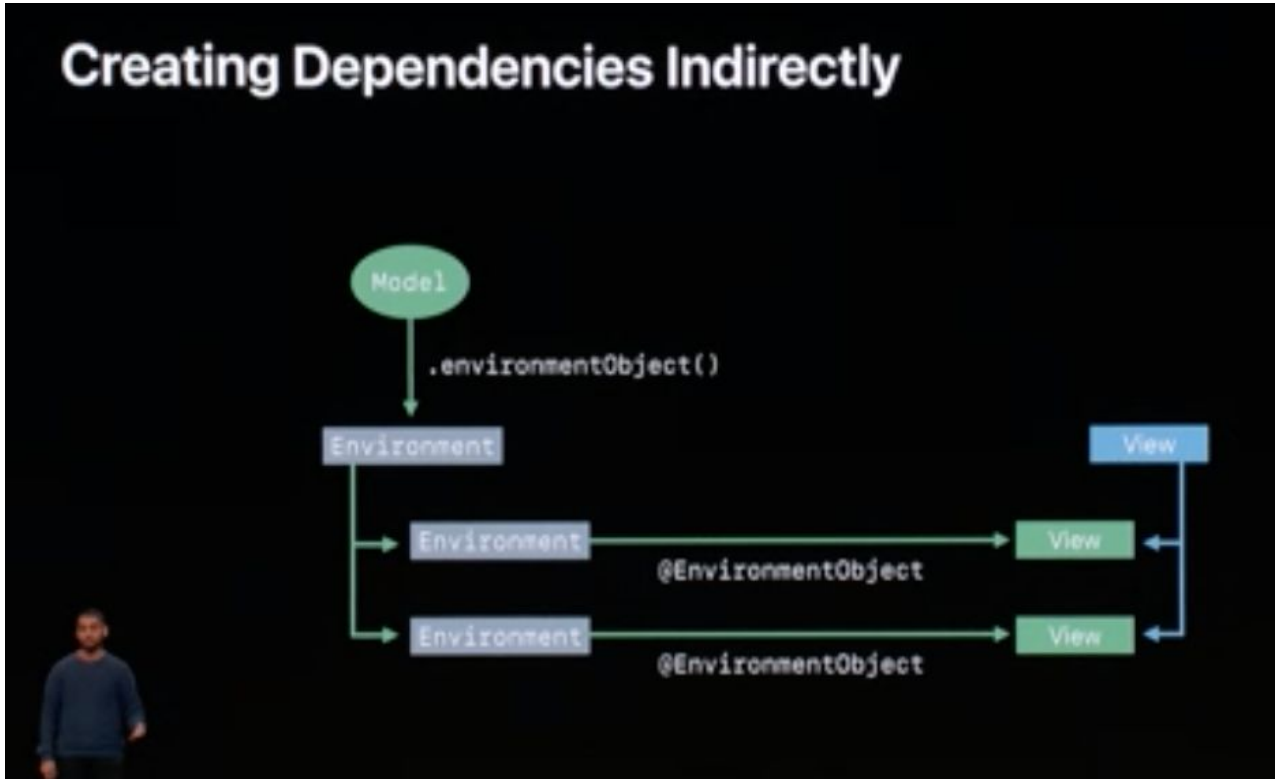
WWDC Example @ObjectBinding

Important Notes!

Because Views in SwiftUI are value types, anytime you are using a reference type we should be using the @ObjectBinding property wrapper.

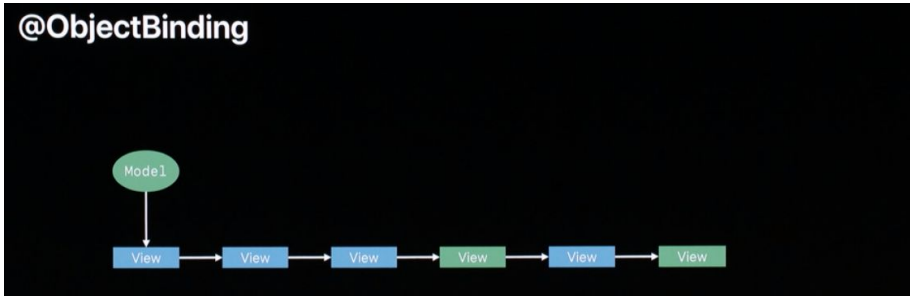
This way the framework will know when the data changes and can keep our view hierarchy up to date.

@EnvironmentObject Creating Dependencies Indirectly

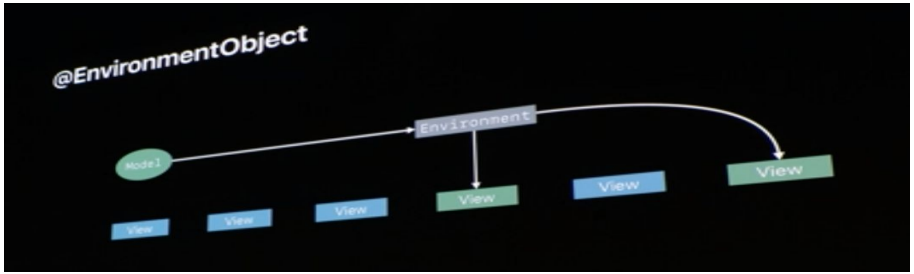


@ObjectBinding vs @EnvironmentObject

@ObjectBinding has to passed down the entire view stack



@EnvirnmentObject is a convenience

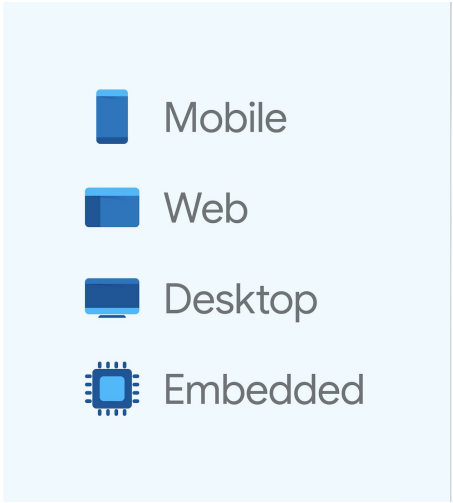


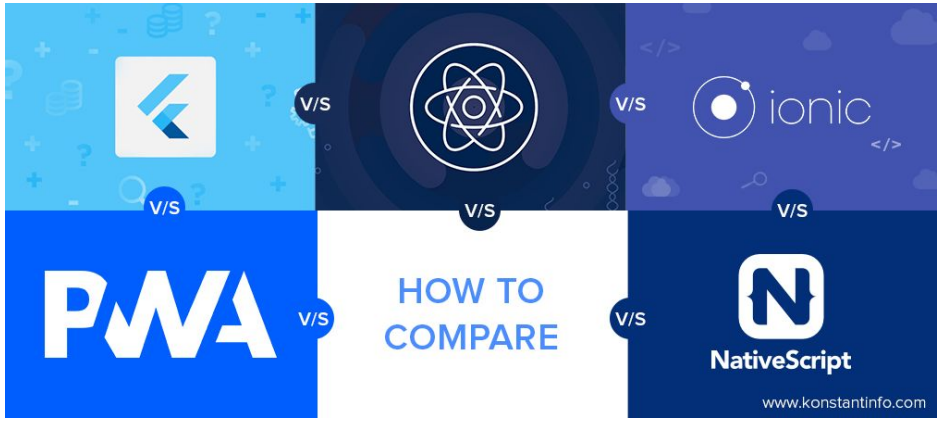
Understanding Data Storage

- ObservableObject Protocol - @Publisher used with you ViewModel
- **@ObservedObject** - bind an ObservableObject to your view
- **@State** - store and persist a state relevant to your view
- **@Binding** - used to pass a value controlled and persisted by a parent view
- **@EnvironmentObject** - data needed by all views.

~~Flutter / React Native~~ Not Needed Anymore

Flutter





SwiftUI is the **Future**!