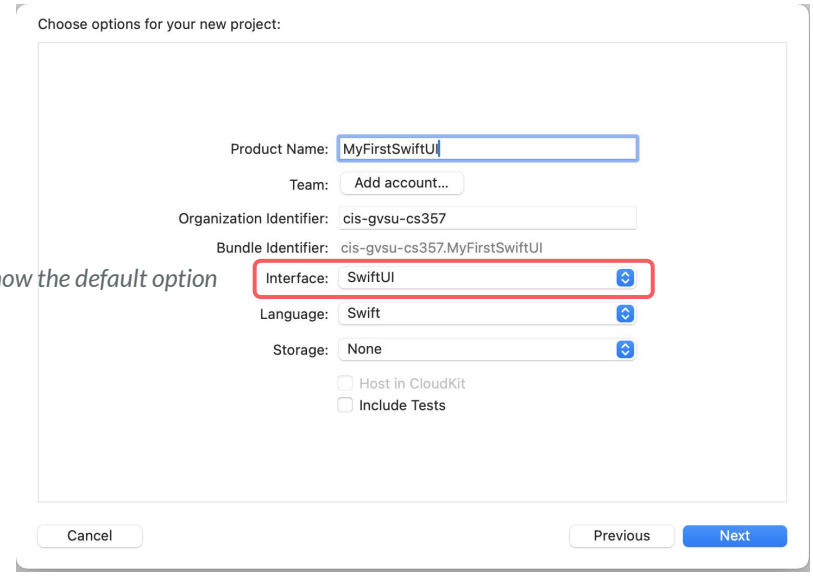
Introduction to SwiftUI

History of SwiftUI

- June 2019: Introduced at WWDC 2019, available on iOS 13
- SwiftUI gets updated along with newer iOS releases

iOS 13	SwiftUI 1.0	
iOS 14	SwiftUI 2.0	• More pure SwiftUI code • AppDelegate.swift is replaced by the App protocol • SceneDelegate.swift is replaced by the Scene protocol • New property wrappers: @xxxx
iOS 15	SwiftUI 3.0	AsyncImage, Navigation, charts, improved widgets
iOS 16	SwiftUI 4.0	More charts, Grid
iOS 17	SwiftUI 5.0	Animations, ScrollView improvement, Window Group, Augmented Reality

Creating a New Project

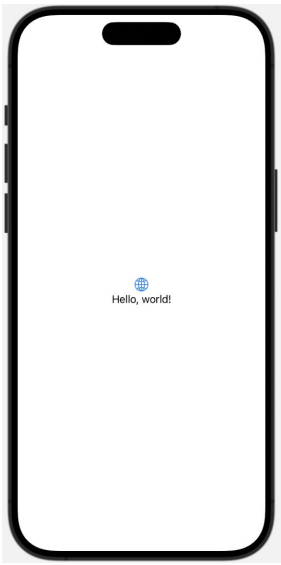


SwiftUI is now the default option

Hello World in SwiftUI (XCode 15)

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

```
import SwiftUI
struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundStyle(.tint)
            Text("Hello, world!")
        }
        .padding()
    }
}
```




```
// XCode 14
struct MyPreview: PreviewProvider {
    struct var previews: some View {
        ContentView()
    }
}
```

```
// XCode 15
#Preview {
    ContentView()
}
```

Jetpack Compose vs. SwiftUI

	Android Jetpack Compose	iOS SwiftUI
Building Blocks	Kotlin functions (annotated with `@Composable`)	Objects (created from `View` struct)
UI construction	Invoke the @Composable	Invoke the object constructor (The `init()` method)
Parent/Child Widget Hierarchy	Invoking a (child) @Composable from within a parent @Composable	Creating a (child) View struct in the body of property of
UI updates	Re-invoke the @Composable	Reevaluate the body property
When UI refreshed?	State variable changes	State variable changes
Use of lambdas	As a parameter of the function	As a parameter of the `init()` method

SwiftUI View

- View is a function of a state
  - Modifying the state triggers UI update
- States are variables that affect the rendition of the UI
- State variables can be either
  - Declared **locally** with the `@State` property wrapper
  - Passed into View (as arguments to its `init()` method)
  - Supplied by a ViewModel
- iOS maintains separate lifetime of the `Views` and the `struct` that defines them
  - The lifetime of the struct is shorter than View lifetime. The struct is gone when view rendering completes

SwiftUI View LifeCycle

Similar concept with Jetpack Compose (Re)Composition

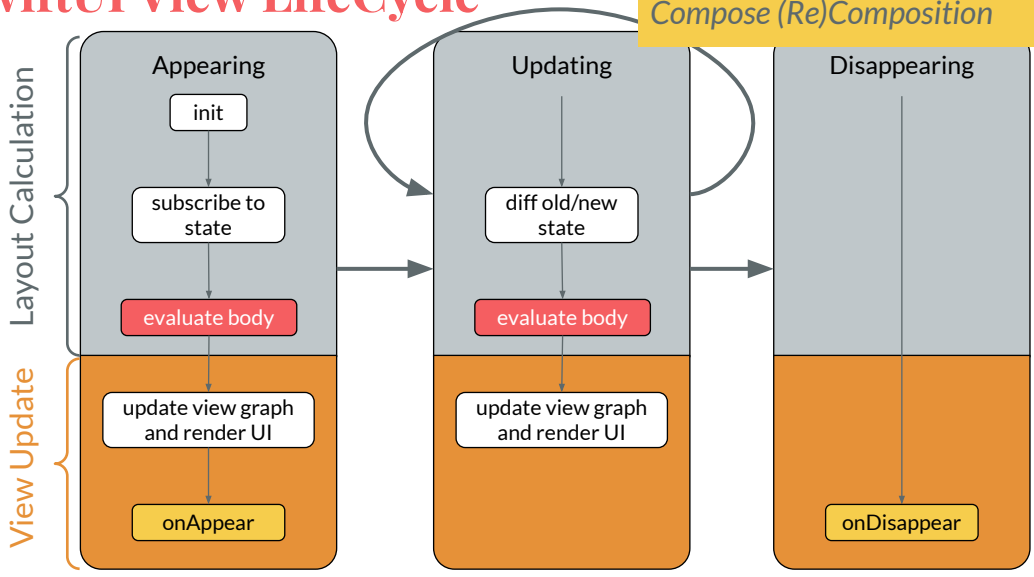


Image credit: https://www.vadimbulavin.com/swiftui-view-lifecycle/

@State Property Wrapper

```
struct ContentView: View {
    @State var counter = 1
    var body: some View {
        VStack {
            Text("Value at \(counter)")
            Button("Add 1", action: {
                counter += 1
            })
        }
    }
}
```

```
struct ContentView: View {
    var body: some View {
        // Incorrect placement of @State
        @State var counter = 1
        VStack {
            Text("Value at \(counter)")
            Button("Add 1", action: {
                counter += 1
            })
        }
    }
}
```

@State must be used **outside** of the body property

Important Property Wrappers

- @State: defines a single source of truth (for updating UI) for value type
- @StateObject: same as @State but for reference type
- @Binding: defines a reference to a previously declared @State
  - The parent component defines a @State variable Z
  - Children that depends on Z can access the state using @Binding
  - Two-way binding of data between parent and child views
- @Published: provides external changes/events
- @Environment: defines a reference to a previously declared environment object
  - Env object does not have to be passed to descendants
  - Children, grandchildren, grand-grandchildren, ... can access the environment

Reference: https://developer.apple.com/videos/play/wwdc2019/226/

M-V-VM in iOS

Wiring up With ViewModel

```
import SwiftUI

struct YourViewHere: View {
  @StateObject var vm =  MyViewModel()

  var body: some View {
    VStack {
       Text(“Counter is \(vm.counter)”)
       Button(“Add”) {
         vm.addOne()
       }
    }
  }
}
```

```
import SwiftUI

class MyViewModel: ObservableObject {
  @Published var counter: Int = 0

  init(/* args */) {
    // More code if needed for constructor
  }

  func addOne() {
    counter += 1
  }
}
```


Using Multiple ViewModels

```
import SwiftUI

struct YourViewHere: View {
  @StateObject var vm1 =  ViewModelOne()
  @StateObject var vm1 =  ViewModelTwo()

  var body: some View {
    VStack {
       Text(“Counter is \(vm1.counter)”)
       Button(“Add”) {
         vm1.add()
       }
    }
  }
}
```

Sharing ViewModel Instances

```
import SwiftUI

struct ContentView: View {
  @StateObject var vm1 =  MyViewModel()

  var body: some View {
    VStack {
       // other widgets
       AnotherView(viewmodel: vm1)
       YetAnother(yvm: vm1)
       // other widgets
    }
  }
}
```

```
import SwiftUI

struct AnotherView: View {
  @ObservedObject var viewmodel: MyViewModel

  var body: some View {
    VStack {
       Text("Counter is \(self.vm.counter)")
       Button ("Hello") {
         self.viewmodel.performSomeWork()
       }
    }
  }
}
struct YetAnother: View {
   @ObserveObject var yvm: MyViewModel
}
```


Customizing Widget Appearance

Android Jetpack Compose

- Invoke the Widget function
- Use the ***function arguments*** for ***properties specific*** to the Widget
  - font is specific to Text
  - onClick is specific to Button
  - readOnly is specific to TextField
- Use the Modifier argument for customization of ***properties common to all the widgets***
  - padding
  - background color
  - width/height

SwiftUI

- Invoke the Widget constructor() and you obtain an instance of an object of that Widget
- Use the ***constructor arguments*** for ***properties specific*** to the Widget
- Use the View extension function(s) for customization of ***properties common to all the widgets***

Customization Examples

```
// Android Jetpack Compose: invoke the Button @Composable
val myBtnStyle = Modifier.padding(16.dp)
                         .background(Color.Blue)
Button(onClick = { print("I'm clicked") },
       modifier = myBtnStyle) {
  Text("Click Me")
}
```

```
// SwiftUI: invoke the Button constructor
Button(action: { print ("I'm clicked") }) {
  Text("Click Me")
}.background(Color.blue).padding(16)
```

```
// Alternative syntax
// Remember that you are creating an OBJECT
let b = Button(action: { print ("I'm clicked") }) {
  Text("Click Me")
}

b.background(Color.blue).padding(16)
```


Customization Examples (Less Lambda)

```
// Android Jetpack Compose
fun buttonWork() {
  print ("I'm clicked")
}
val myBtnStyle = Modifier.padding(16.dp)
                         .background(Color.Blue)

Button(onClick = ::buttonWork, modifier = myBtnStyle) {
  Text("Click Me")
}
```

```
// SwiftUI
func buttonWork() {
  print ("I'm clicked")
}



let b = Button(action: buttonWork) {
  Text("Click Me")
}
b.background(Color.blue).padding(16)
```