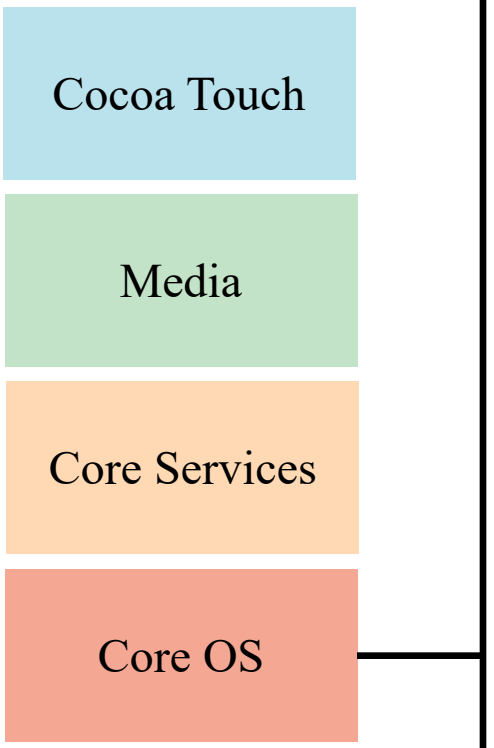
Mobile Design – iOS

Outline

- iOS Overview
- Swift Language
- Xcode Basics
- XCode
- Design Strategy: Model-View-Controller (MVC)
- Multiple Views & View Controllers
- CocoaPods: Use External Dependencies
- Package Manager: new experience, alternative to CocoaPods
- References

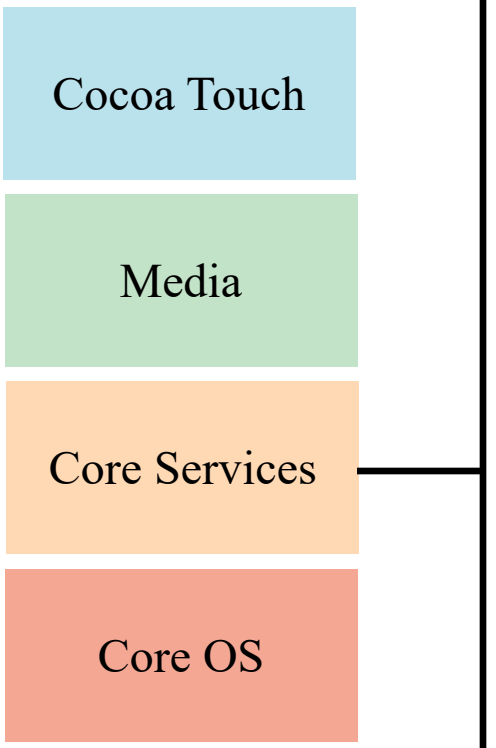
iOS Overview - What's in iOS?



Core OS

OSX Kernel	Power Management
Mach 3.0	Keychain Access
BSD	Certificates
Sockets	File System
Security	Bonjour

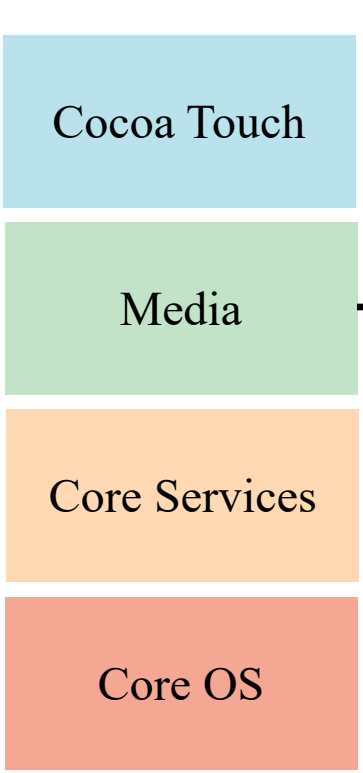
iOS Overview - What's in iOS?



Core Services

Collections	Core Location
Address Book	Net Services
Networking	Threading
File Access	Preferences
SQLite	URL Utilities

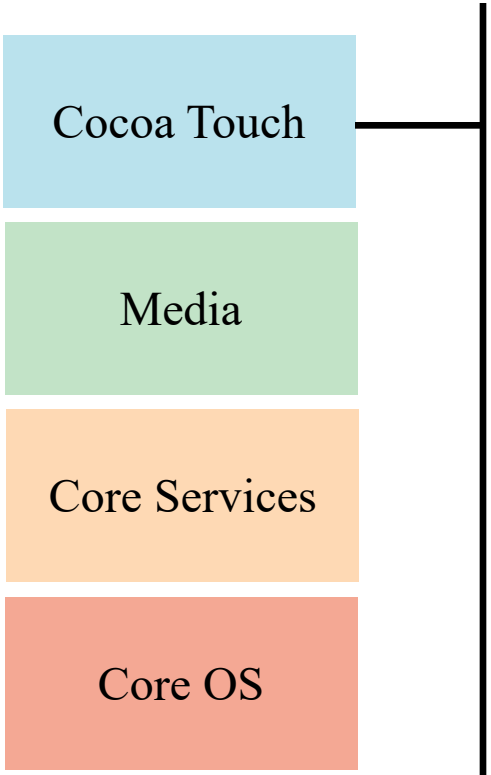
iOS Overview - What's in iOS?



Media

Core Audio	JPEG, PNG, TIFF
OpenAL	PDF
Audio Mixing	Quartz (2D)
Audio Recording	Core Animation
Video Playback	OpenGL ES

iOS Overview - What's in iOS?



Cocoa Touch

Multi-Touch	Alerts
Core Motion	Web View
View Hierarchy	Map Kit
Localization	Image Picker
Controls	Camera

iOS Platform Components

- Tools: Xcode, Instruments
- Language: Swift
- Frameworks: Foundation, Core Data, UIKit, Core Motion, Map Kit, WebKit, SwiftUI, etc.
- Design Strategy: MVC (Storyboard), MVVM (SwiftUI)

Swift

- Swift Introduction
- Define simple values
- Control-flow: if-else
- Define a function
- Define a class
- Inherit a class

Introduction to Swift

Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. for iOS, macOS, watchOS, tvOS, iPadOS, Linux, and Windows.

- Swift is designed to work with Apple's Cocoa and Cocoa Touch frameworks and the large body of existing Objective-C (ObjC) code written for Apple products. It is built with the open source LLVM compiler framework and has been included in Xcode since version 6.
- Swift was introduced at Apple's 2014 Worldwide Developers Conference (WWDC).
- Version 2.2 was made open-source software under the Apache License 2.0 on December 3, 2015, for Apple's platforms and Linux.
- Latest version is Swift 5.6 (March 14, 2022).
- See: https://swift.org/

Introduction to Swift (cont'd)

Swift is friendly to new programmers. Swift removes the occurrence of large classes of common programming errors by adopting modern programming patterns:

- **Variables** are **always initialized** before use.
- **Array indices** are checked for **out-of-bounds errors**.
- **Integers** are checked for overflow.
- *Optionals* ensure that nil values are handled explicitly.
- Memory is managed automatically.
- Error handling allows controlled recovery from unexpected failures.

Swift – simple values

Use **let** to make a constant and **var** to make a variable.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
let label = "The width is "
```

To include values in a string:

```
let apples = 3
let appleSummary = "I have \(apples) apples."
```

Most times the compiler infers the type of constant/variable for you. But sometimes you have to write the variable type explicitly:

```
let implicitInteger = 70
let explicitDouble: Double = 70
```

Swift – simple values (cont'd)

To create **arrays** and **dictionaries**:

```
var shoppingList = ["catfish", "water", "tulips", "blue
paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

To create an **empty array** or **dictionary**, use the initializer syntax.

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

Swift – Control Flow

Example: use **if** to make conditionals:

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
```

An optional value either contains a value or contains **nil** to indicate that a value is missing (append ? to any type).

```
var optionalName: String? = "John Appleseed"
if let name = optionalName {
    print("Hello, \(name)")   //name != nil
}
```

Swift – Define a function

Use **func** to declare a function. Call a function by following its name with a list of arguments in parentheses. Use **->** to separate the parameter names and types from the function's return type.

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

Swift – Define a class

Define a **class**:

```
class Shape {
    var numberOfSides = 0

    //called when an instance is created (Constructor)
    init(numberOfSides: Int) {
        self.numberOfSides = numberOfSides
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Create a **class instance**:

```
let square = Shape(numberOfSides: 4)
square.simpleDescription()
```

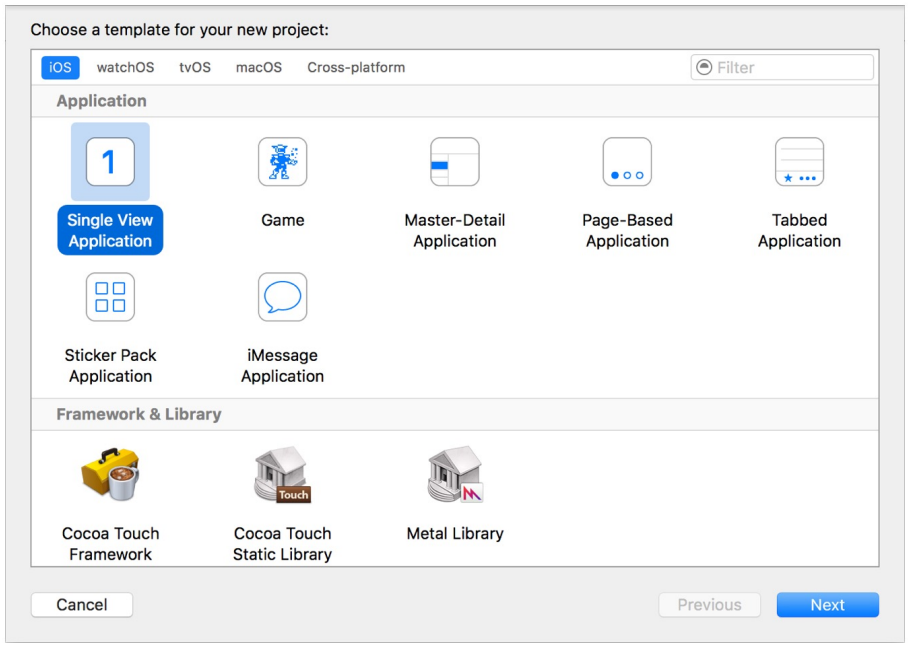
Swift – Inherit a class

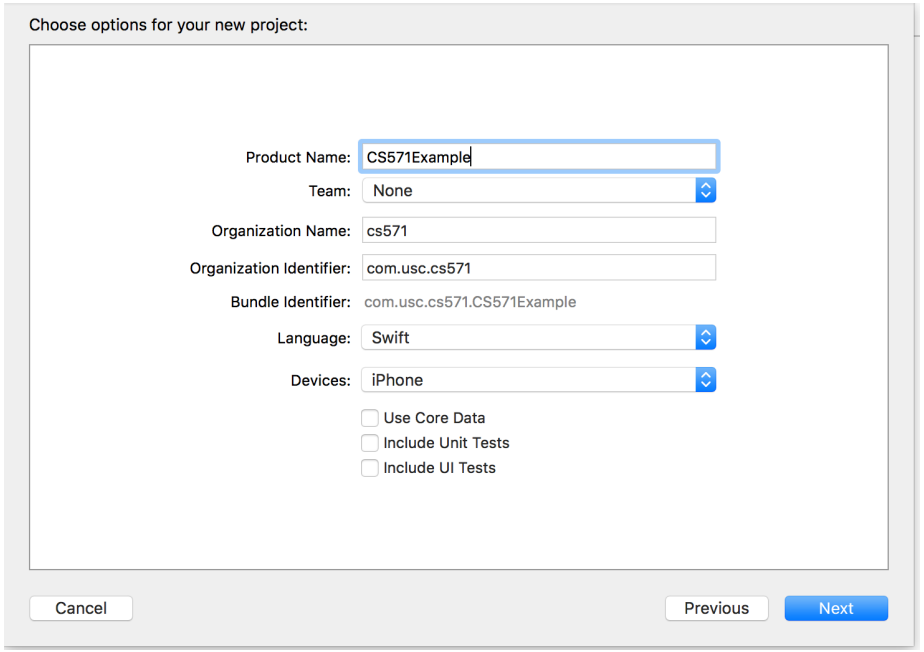
```
class Square: Shape {
    var sideLength: Double

    init(sideLength: Double, numberOfSides: Int) {
        self.sideLength = sideLength
        super.init(numberOfSides: numberOfSides)
    }

    func area() -> Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with \(sideLength)."
    }
}
```

Xcode Basics

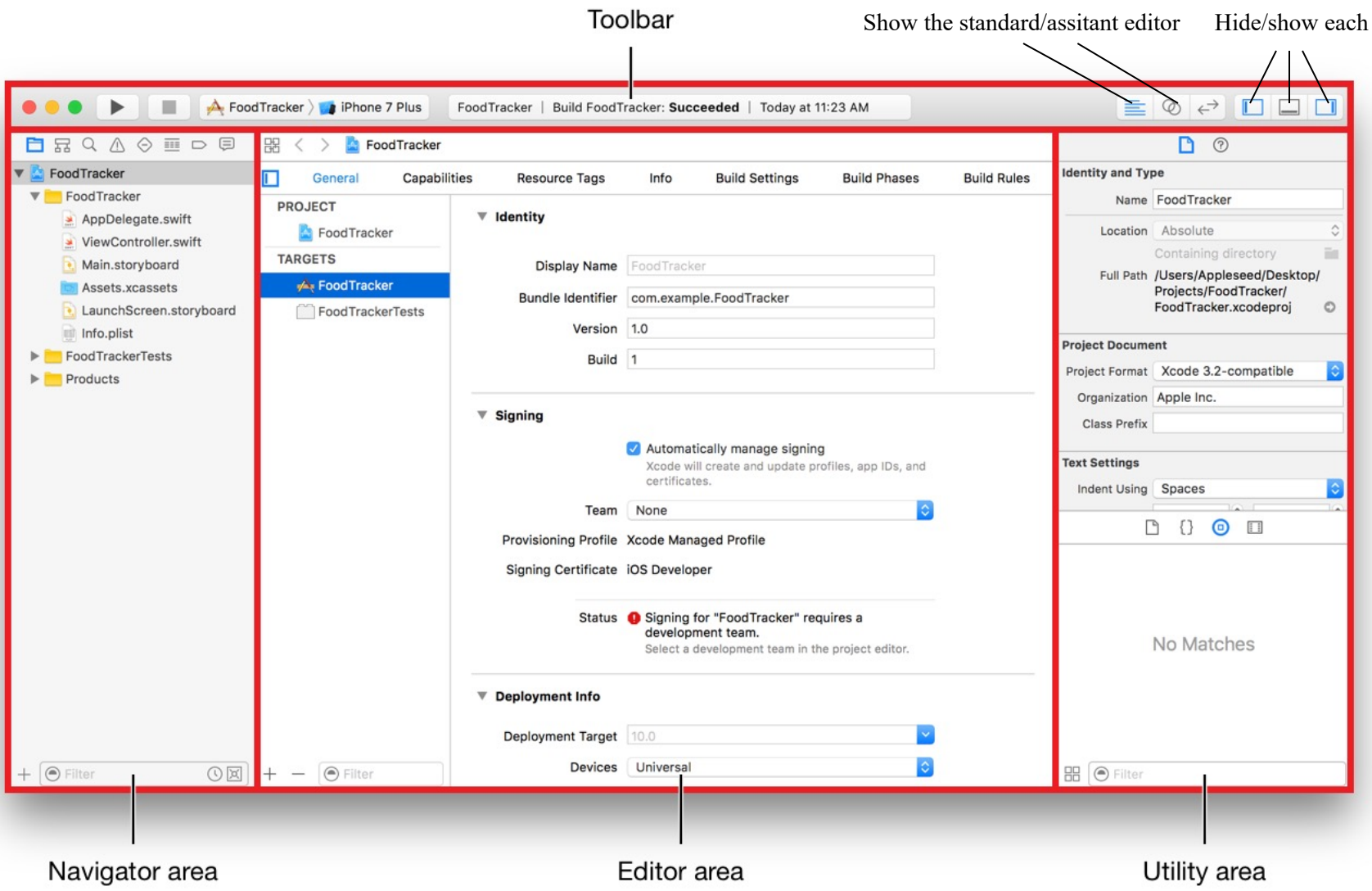
- Create a new project
- Get familiar with Xcode
- Design UI in storyboard
- Set view controller for the UI
- View controller lifecycle
- Connect UI to code
- Run your app in the simulator
- Current stable release: *Xcode 13.3* (March 14, 2022)
- See: https://developer.apple.com/xcode/

Create a new project





Get familiar with Xcode

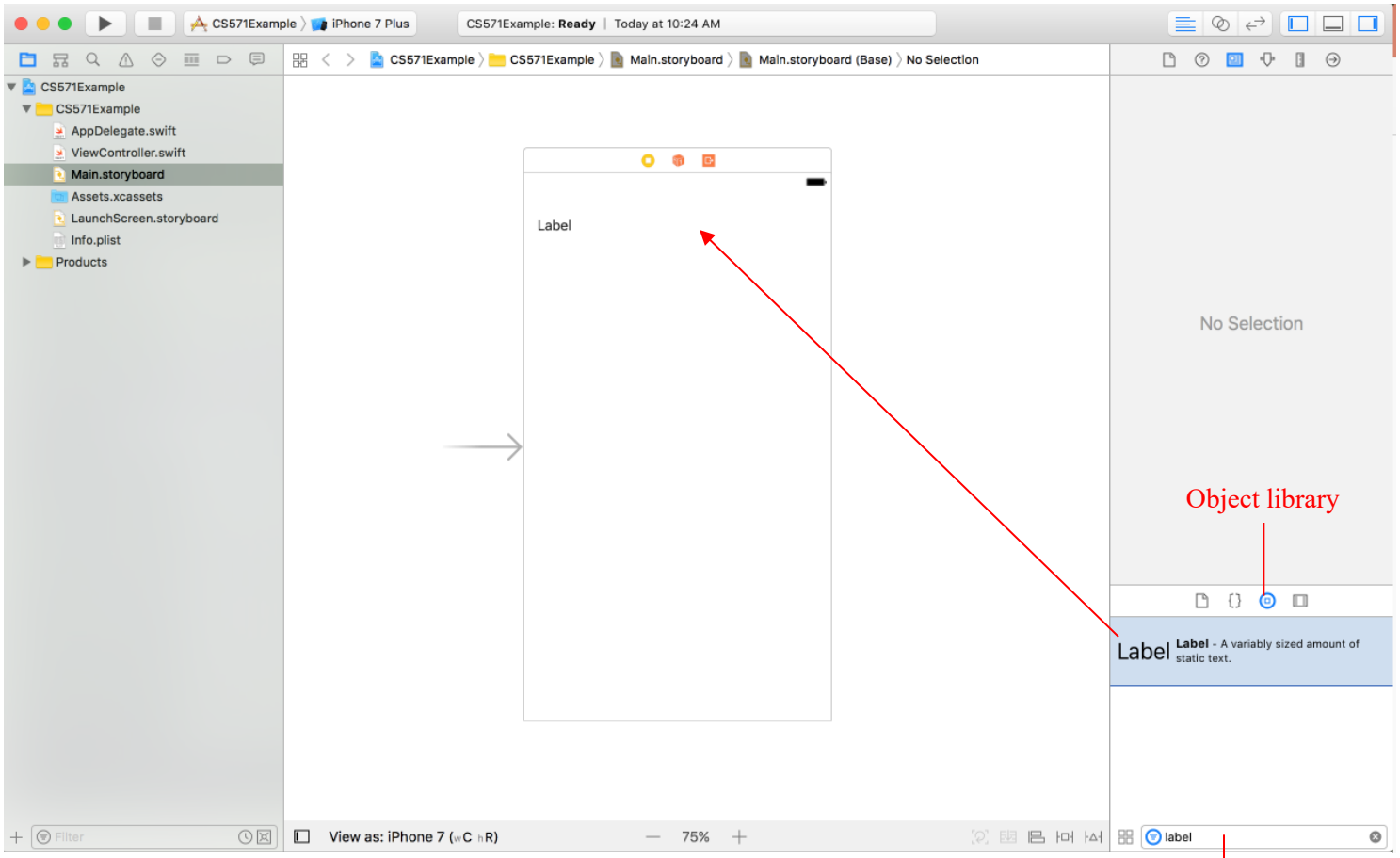


Storyboard

- A **storyboard** is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens;
- A storyboard is composed of a **sequence of scenes**, each of which represents a view controller and its views;
- Scenes are connected by **segue objects**, which represent a transition between two view controllers.

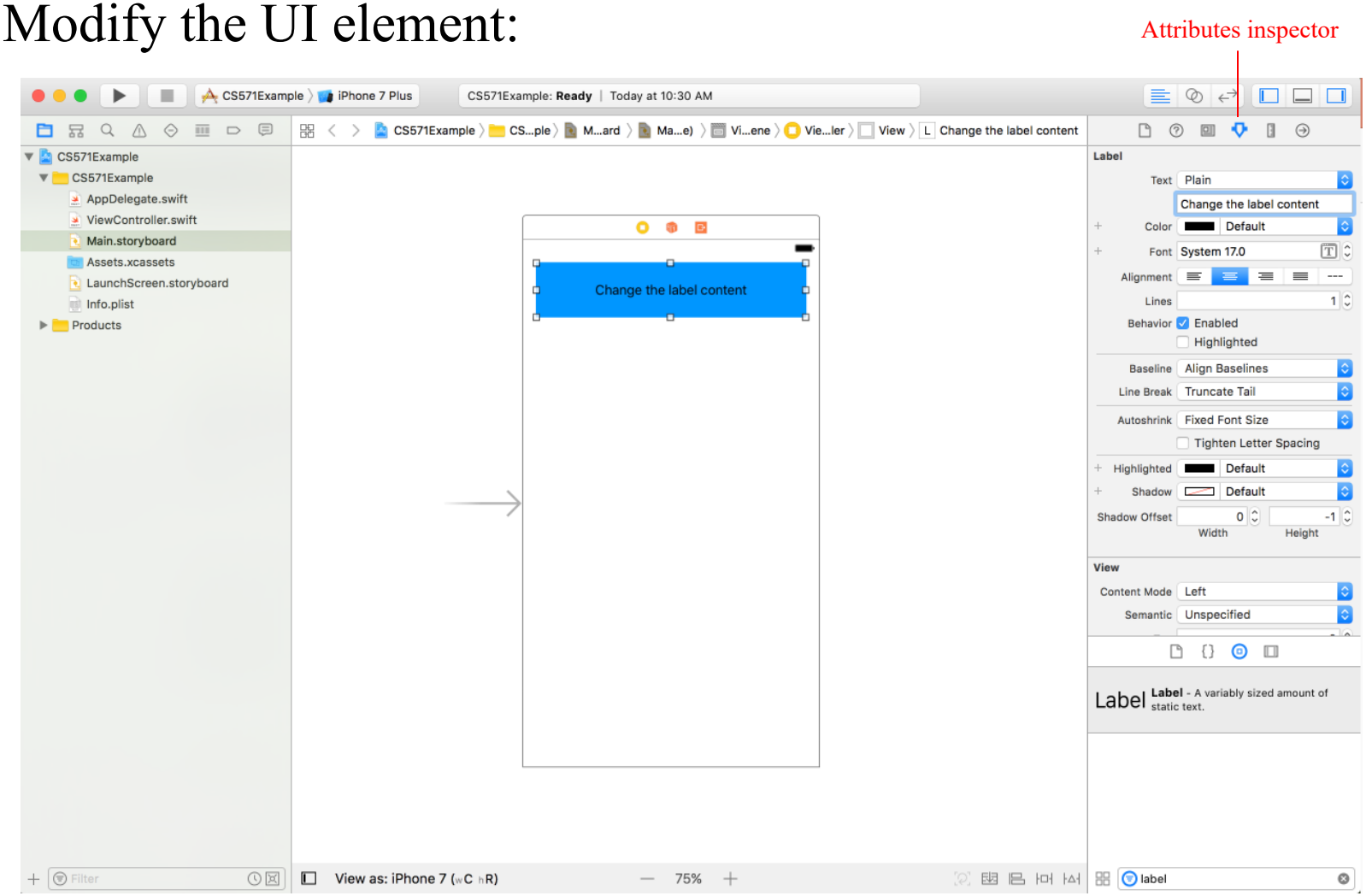
Design UI in storyboard

Add a UI Element to storyboard:



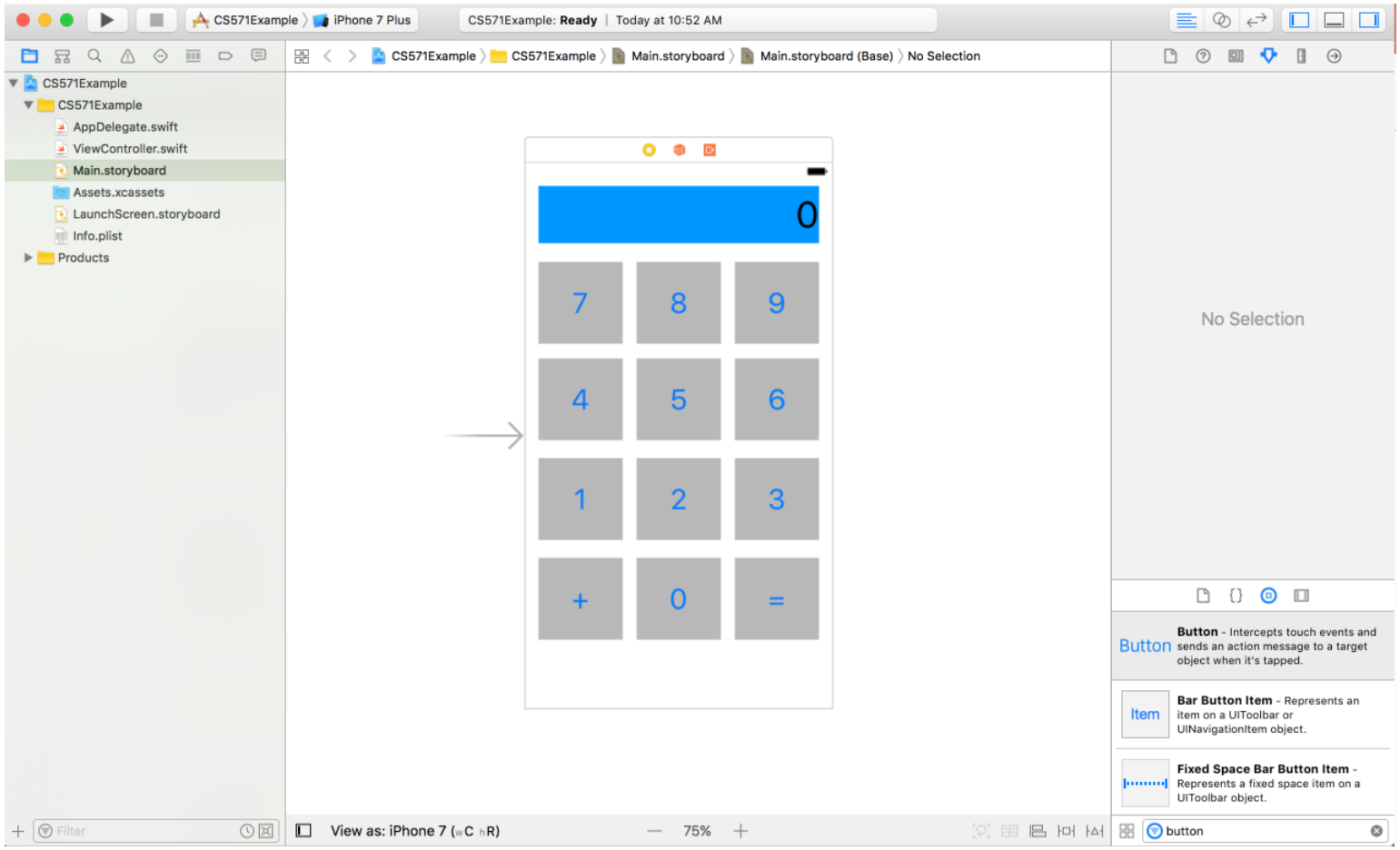
Design UI in storyboard (cont'd)

Modify the UI element:



Design UI in storyboard (cont'd)

Continue to add 12 buttons:



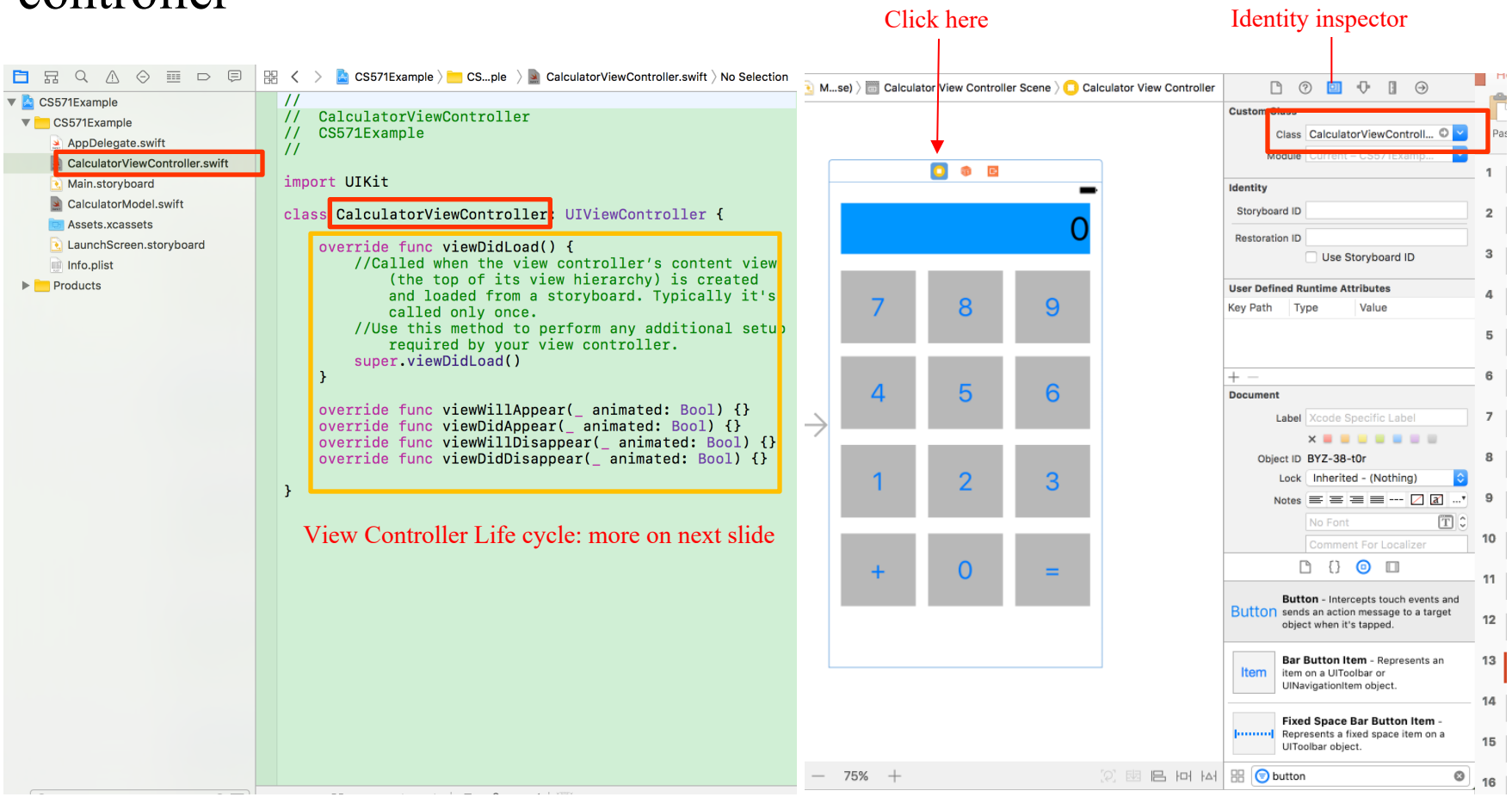
View Controller

Provides the infrastructure for managing the views of your UIKit app.

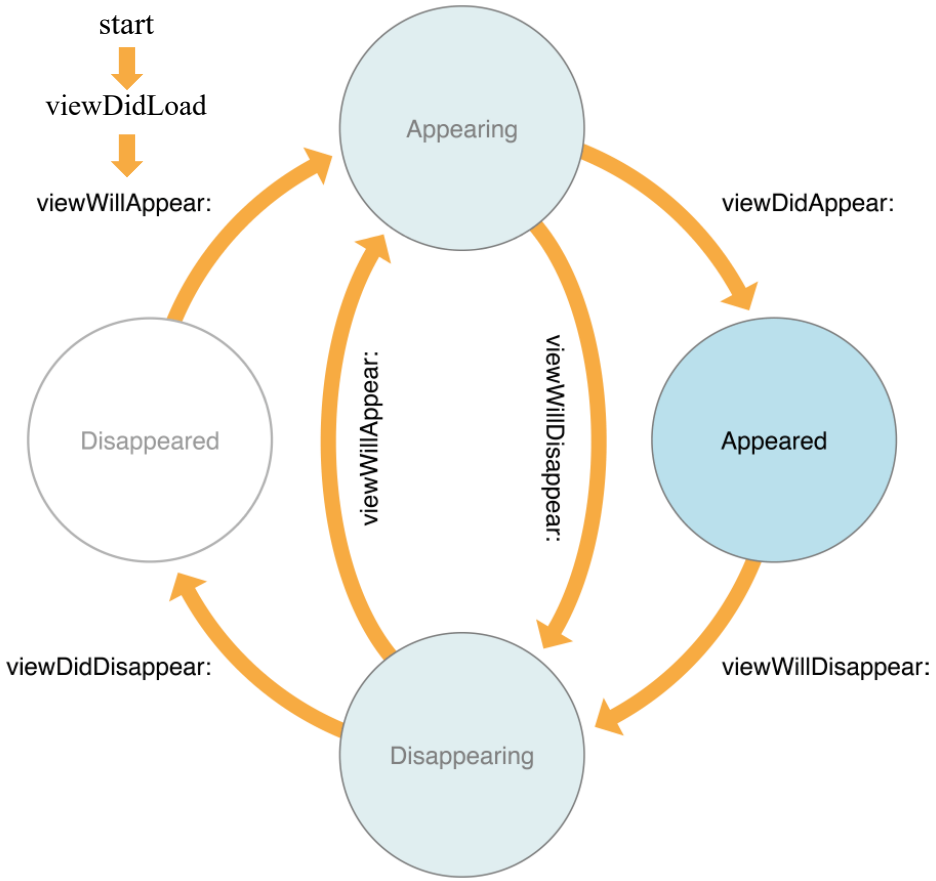
- A **view controller manages a set of views** that make up a portion of your app's user interface.
- It is responsible for **loading** and **disposing** of those **views**, for managing **interactions** with those views, and for coordinating responses with any appropriate data objects.
- View controllers also coordinate their efforts with other controller objects—including other view controllers—and help **manage your app's overall interface**.

Set View Controller for the UI

A common mistake for beginners is forgetting to set the view controller

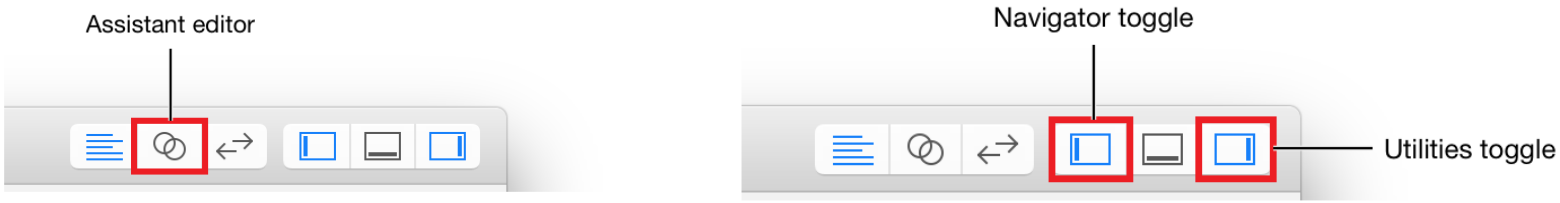


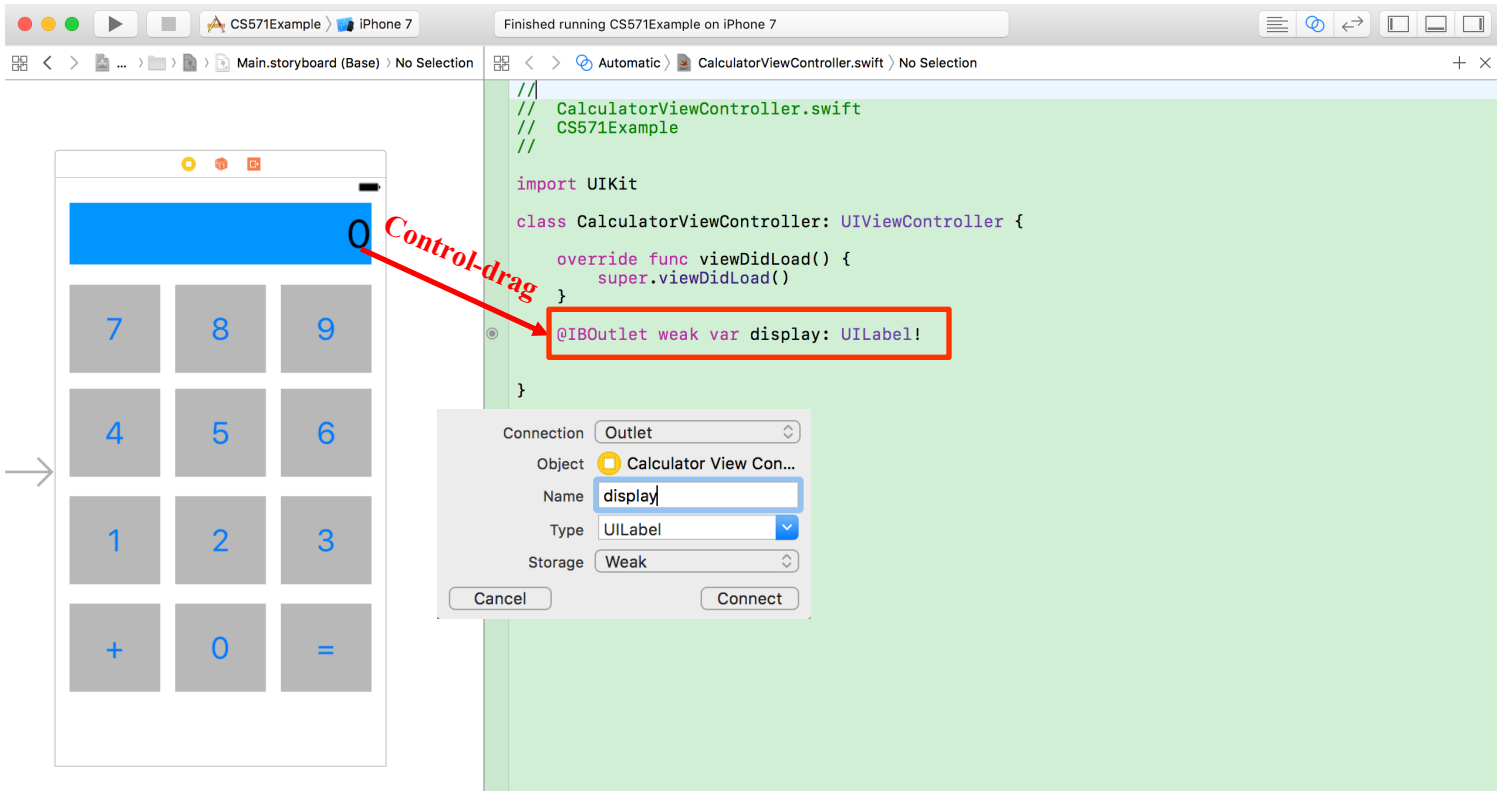
View Controller Lifecycle



An object of the `UIViewController` class (and its subclasses) comes with a set of methods that manage its view hierarchy. iOS automatically calls these methods at appropriate times when a view controller transitions between states.

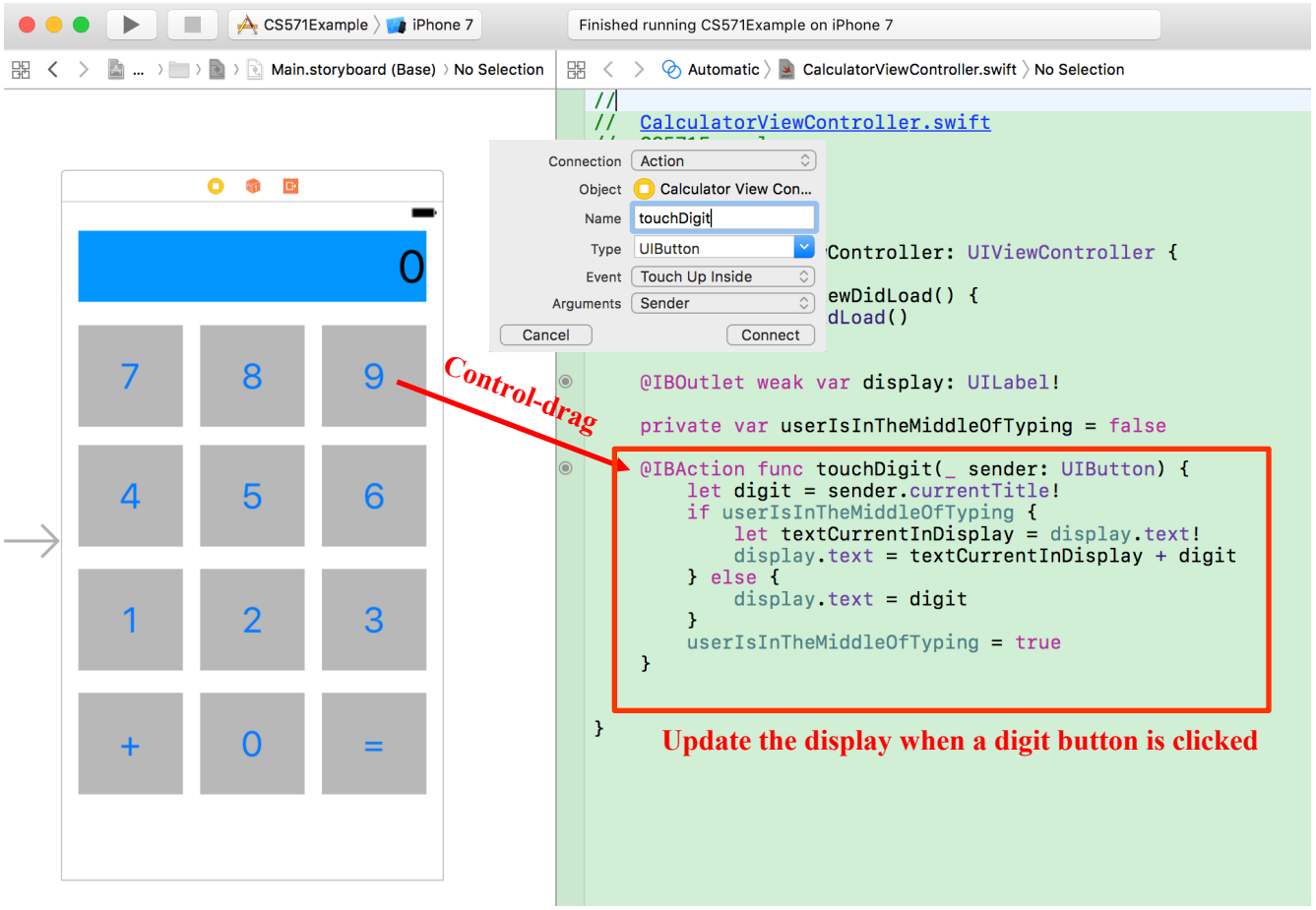
Connect UI to Code





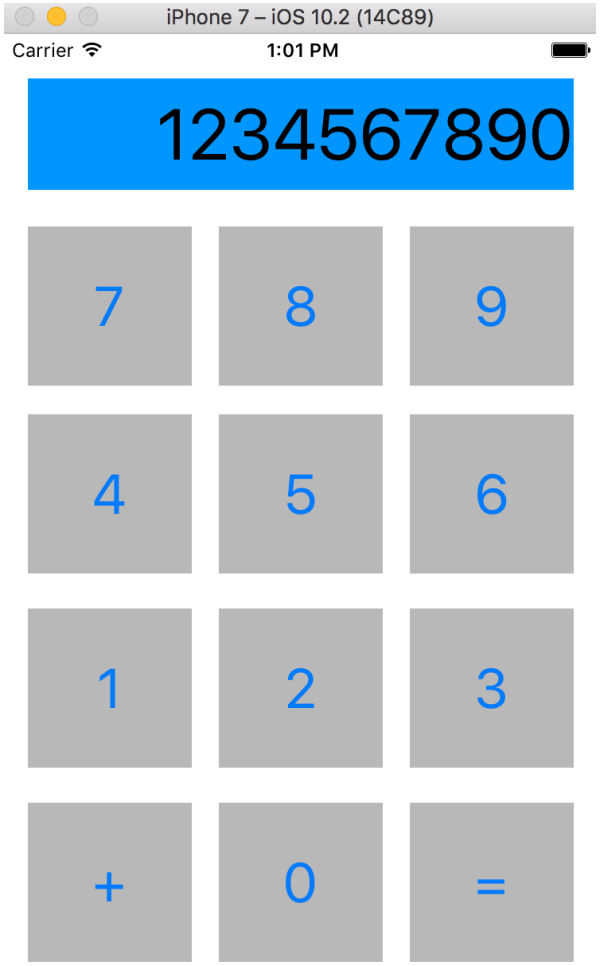
Connect UI to Code (cont'd)

Control-drag a digit button to create an event handler func. Then control-drag all the other digit buttons to the same func.



Run your app in the Simulator

- The Scheme pop-up menu lets you choose which simulator or device you'd like to run your app on.
- Click Run button.
- Click each of the digit buttons to test your app.



Design Strategy: Model-View-Controller (MVC)

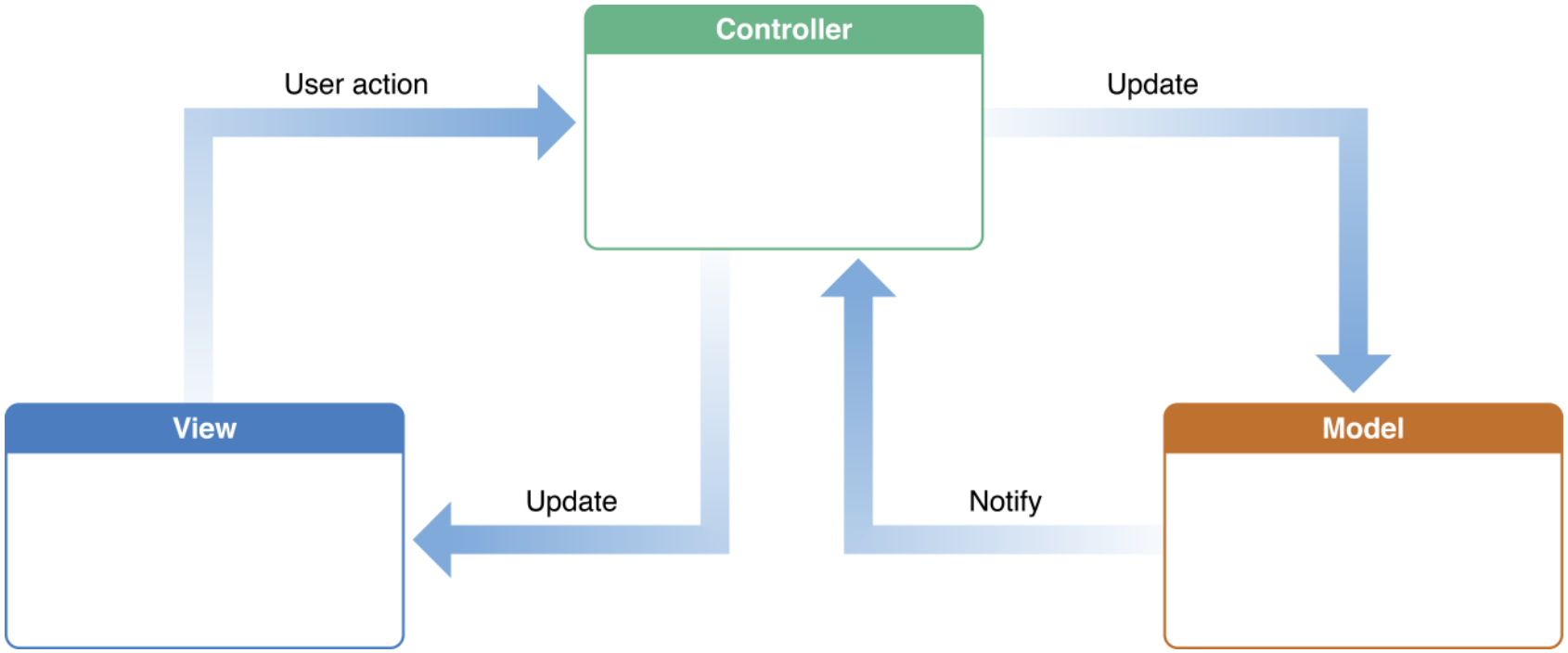
- Why MVC
- How MVC works in iOS development
- Create a calculator model
- Design the UI view
- Controller: connect UI and the model

Why MVC?

MVC is central to a good design for a Cocoa application. The benefits of adopting this pattern are numerous.

- Objects in the applications tend to be more reusable
- The interfaces tend to be better defined
- Applications having an MVC design are also more easily extensible than other applications.
- iOS development technologies and architectures are based on MVC and require that your custom objects play one of the MVC roles.

How MVC works in IOS development



Model = What your application is (but ***not*** how it is displayed)
Controller = How your Model is presented to the user (UI logic)
View = Your Controller's minions

Create a calculator model

What does the **model** do:

•Given the **operands** and **operation symbols**, return the **result**, such as 1+1=2

•Need to deal with 1+2+3+4=? and return any **intermediate results** when a "+" or "=" button is pressed.

•Perform a new **operation**:

- "+": Execute the pending operation to get intermediate result. Save the operation symbol and first operand (the intermediate result) as a pending operation.
- "=": Execute the pending operation to get final result.

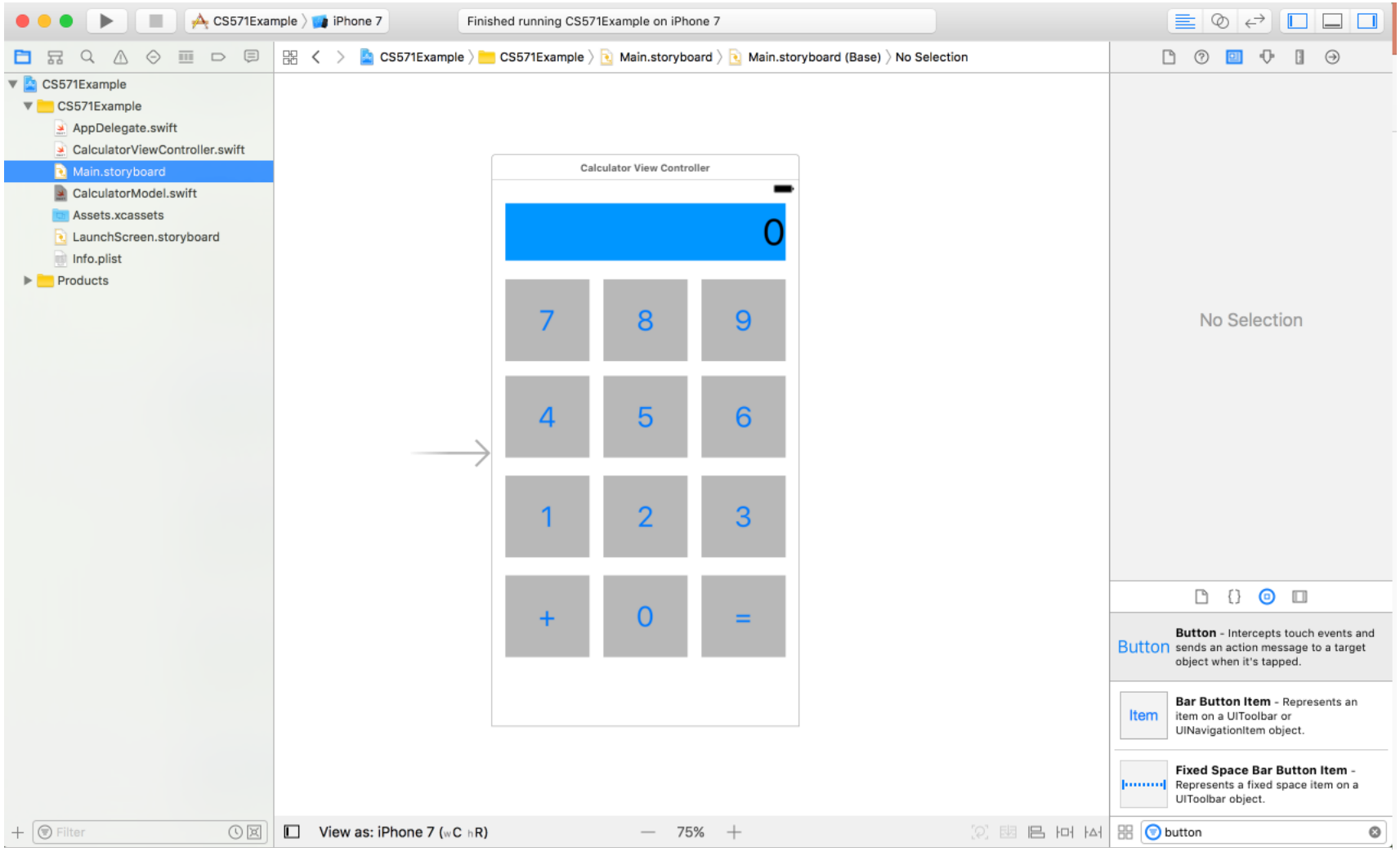
Create a calculator model (cont'd)

```
class CalculatorModel {
    //A dummy calculator model to support simple addition operation
    private var operations: Dictionary<String, Operation> = [
        "+": Operation.AdditionOperation({$0 + $1}),
        "=": Operation.Equal
    ]
    private enum Operation {
        case AdditionOperation((Int, Int) -> Int)
        case Equal
    }
    private struct PendingAdditionOperationInfo {
        var additionFunction: (Int, Int) -> Int
        var firstOperand: Int
    }

    private var accumulator = 0  //intemediate result
    private var pending: PendingAdditionOperationInfo?
    var result: Int { get { return accumulator } }

    func setOperand(operand: Int) {
        accumulator = operand
    }
```

Create a calculator model (cont'd)

```
    func performOperation(symbol: String) {
        if let operation = operations[symbol] {
            switch operation {
                case .AdditionOperation(let function):
                    executePendingAdditionOperation()
                    pending = PendingAdditionOperationInfo(additionFunction:
function, firstOperand: accumulator)
                case .Equal:
                    executePendingAdditionOperation()
            }
        }
    }

    private func executePendingAdditionOperation() {
        if pending != nil {
            accumulator = pending!.additionFunction(pending!.firstOperand,
accumulator)
            pending = nil
        }
    }

}
```

Design the UI view



Controller: connect UI and the model

What does the controller do?

- Get **user actions** from the **UI view**, let the model do the calculation, get results from model and update the UI view.
- **Connection** with the **UI view**:
  - Own the outlet to the display label: can get and update the display
  - Action handlers for all the digit buttons and operation symbol buttons
- **Connection** with the **model**:
  - Send new operands and operation symbols to the model. Let model do the calculation.
  - Get intermediate results and final results from the model

Controller: connect UI and the model (cont'd)

```
import UIKit

class CalculatorViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    private var userIsInTheMiddleOfTyping = false

    private var displayValue: Int {
        get { return Int(display.text!)! }
        set { display.text = String(newValue) }
    }

    private var model = CalculatorModel()

    @IBOutlet weak var display: UILabel!
```

Controller: connect UI and the model (cont'd)

```
    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        if userIsInTheMiddleOfTyping {
            let textCurrentInDisplay = display.text!
            display.text = textCurrentInDisplay + digit
        } else {
            display.text = digit
        }
        userIsInTheMiddleOfTyping = true
    }

    @IBAction func performOperation(_ sender: UIButton) {
        if userIsInTheMiddleOfTyping {
            model.setOperand(operand: displayValue)
            userIsInTheMiddleOfTyping = false
        }
        if let methematicalSymbol = sender.currentTitle {
            model.performOperation(symbol: methematicalSymbol)
        }
        displayValue = model.result
    }
}
```

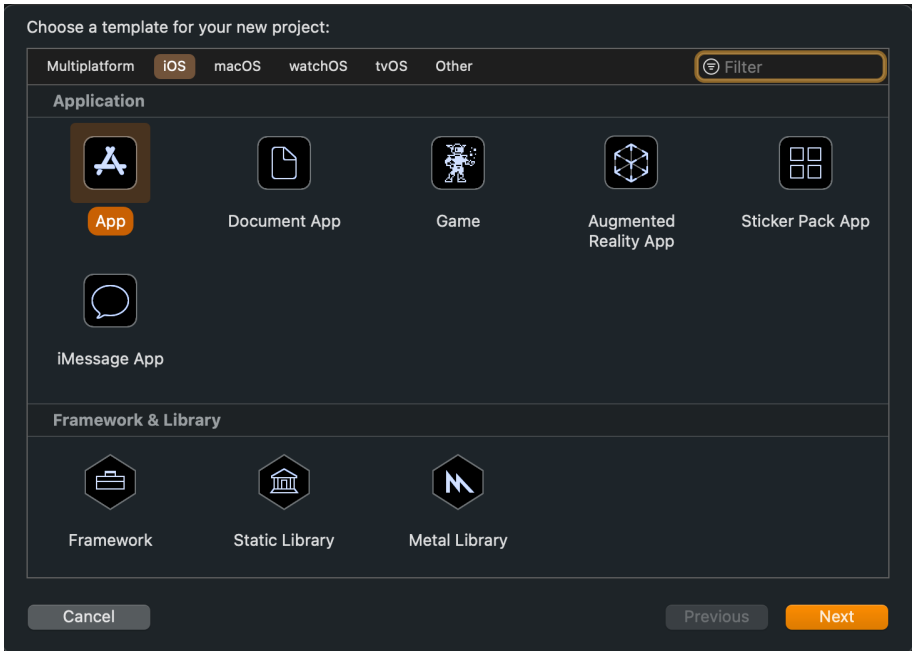
SwiftUI

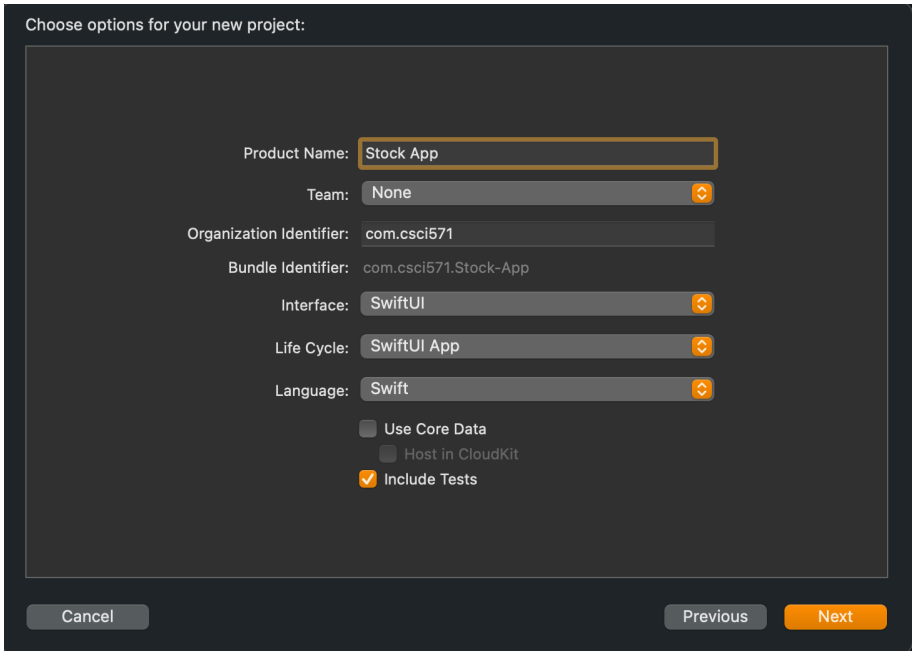
- **New** simple way to build UIs across all Apple platforms
- Use 1 set of tools and APIs
- Uses declarative Swift syntax easy to read
- Works seamlessly with new Xcode design tools
- Keeps code and design in sync
- Automatically supports Dynamic Type, Dark Mode, localization
- See: https://developer.apple.com/xcode/swiftui/

Develop in SwiftUI

- Create a new project
- Design UI with SwiftUI
- Create ViewModel for the View
- View lifecycle
- See the UI changes with Preview
- Build and run the app in simulator

Create a new project

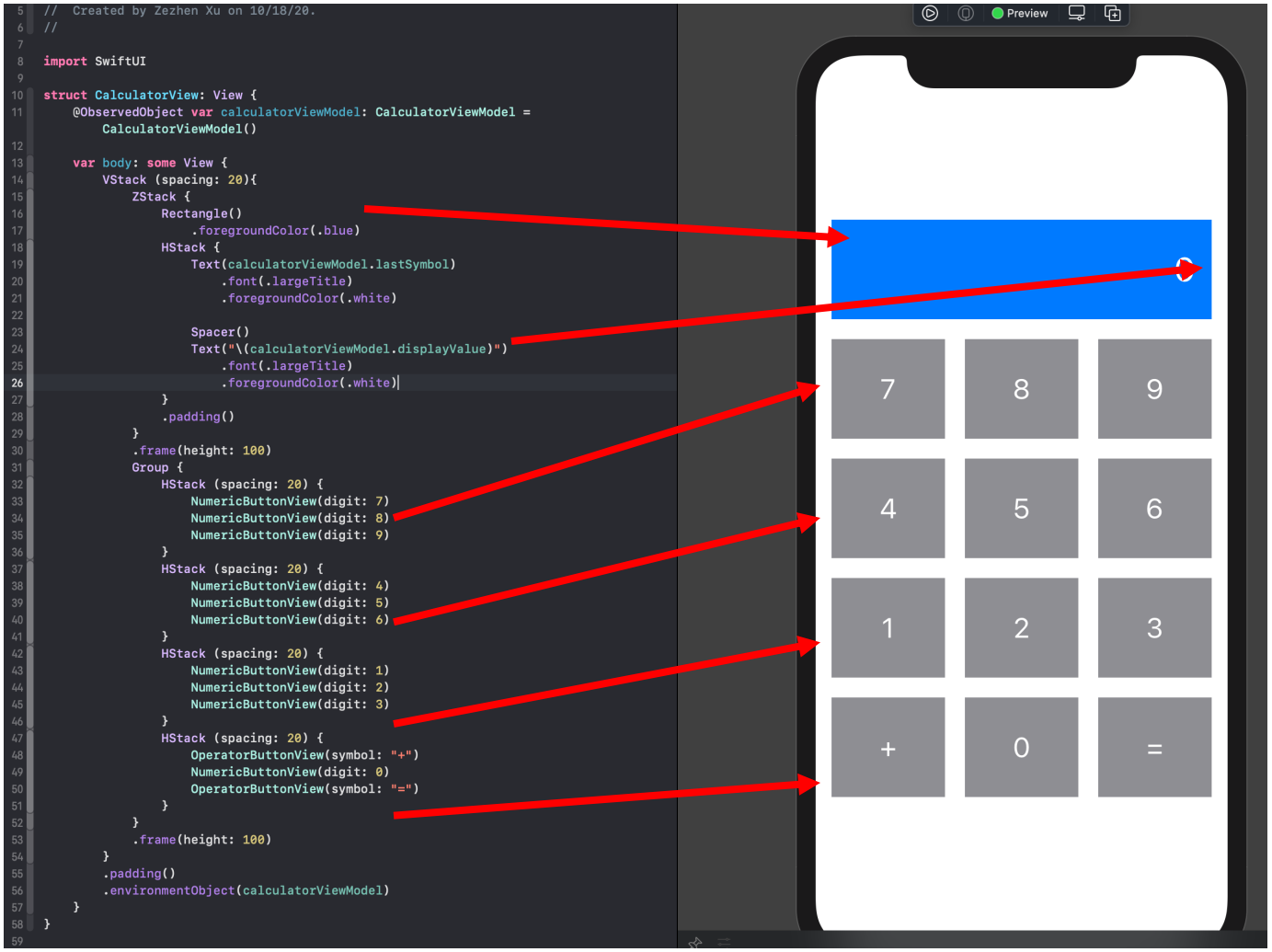




Design UI with SwiftUI (cont'd)

- SwiftUI uses a **declarative syntax** so you can simply state what your user interface should be like.
- SwiftUI's **Views** are very **lightweight**. Extract Views into sub-Views and reuse them.
- You can **change the looks** and functionalities of Views **by appending modifier** after them.
- E.g.: .padding(), .frame(), .foregroundColor(), .onGestureTap(), .onAppear()

Design UI with SwiftUI (cont'd)



Design UI with SwiftUI (cont'd)

```
67  struct NumericButtonView: View {
68      @EnvironmentObject var calculatorViewModel: CalculatorViewModel
69
70      var digit: Int
71      var body: some View {
72          Button(action: {calculatorViewModel.digitTouched(digit)}, label: {
73              ZStack {
74                  Rectangle()
75                      .foregroundColor(.gray)
76                  Text("\(digit)")
77                      .font(.title)
78                      .foregroundColor(.white)
79                      .padding()
80              }
81          })
82
83      }
84  }
```

Design UI with SwiftUI (cont'd)

```
88  struct OperatorButtonView: View {
89      @EnvironmentObject var calculatorViewModel: CalculatorViewModel
90
91      var symbol: String
92      var body: some View {
93          Button(action: {
94              calculatorViewModel.performOperation(symbol)
95          },
96          label: {
97              ZStack {
98                  Rectangle()
99                      .foregroundColor(.gray)
100                 Text("\(symbol)")
101                     .font(.title)
102                     .foregroundColor(.white)
103                     .padding()
104             }
105         })
106     }
107 }
```

Create ViewModel for the View

- SwiftUI is **data driven**. The Views simply shows the data inside of ViewModels in graphic format.
- ViewModels usually **inherits "ObservableObject**", it will send updates to Views.
- Some fields in ViewModel have @Published property wrapper. ViewModels will only send changes to View if those fields are modified.
- You can also **manually send update to Views** inside ViewModels using objectWillChange.send(). Views listen to those changes by using .onReceive() modifier.

Create ViewModel for the View (cont'd)

```swift
import Foundation

class CalculatorViewModel: ObservableObject {
    private var model = CalculatorModel()

    @Published var displayValue: Int = 0
    @Published var lastSymbol: String = ""

    func digitTouched(_ digit: Int) {
        displayValue = displayValue*10 + digit
    }

    func performOperation(_ symbol: String) {
        lastSymbol = symbol
        model.setOperand(operand: displayValue)
        model.performOperation(symbol: symbol)
        if symbol == "=" {
            displayValue = model.result
        }
        else {
            displayValue = 0
        }
    }
}
```

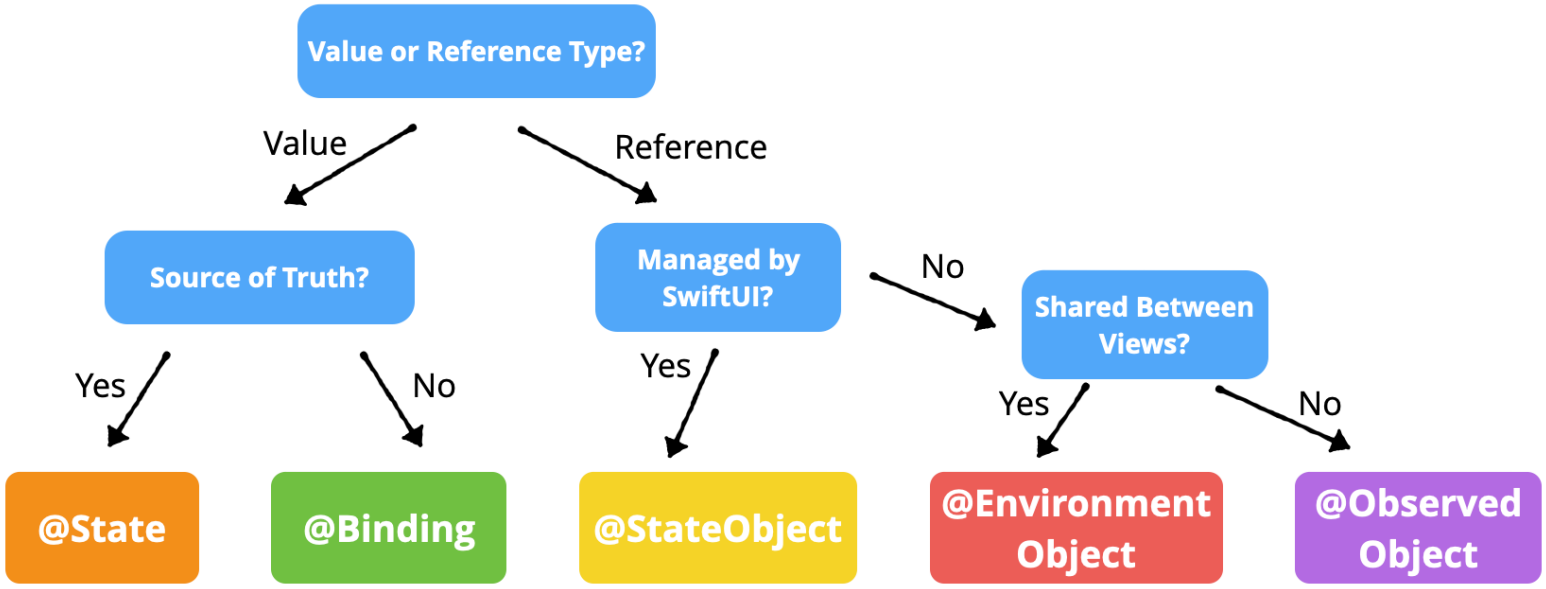
Access to Model

@Published fields

Functions called by View to change Model. Within the functions, @Published fields will change. View catches those changes and update the UI automatically

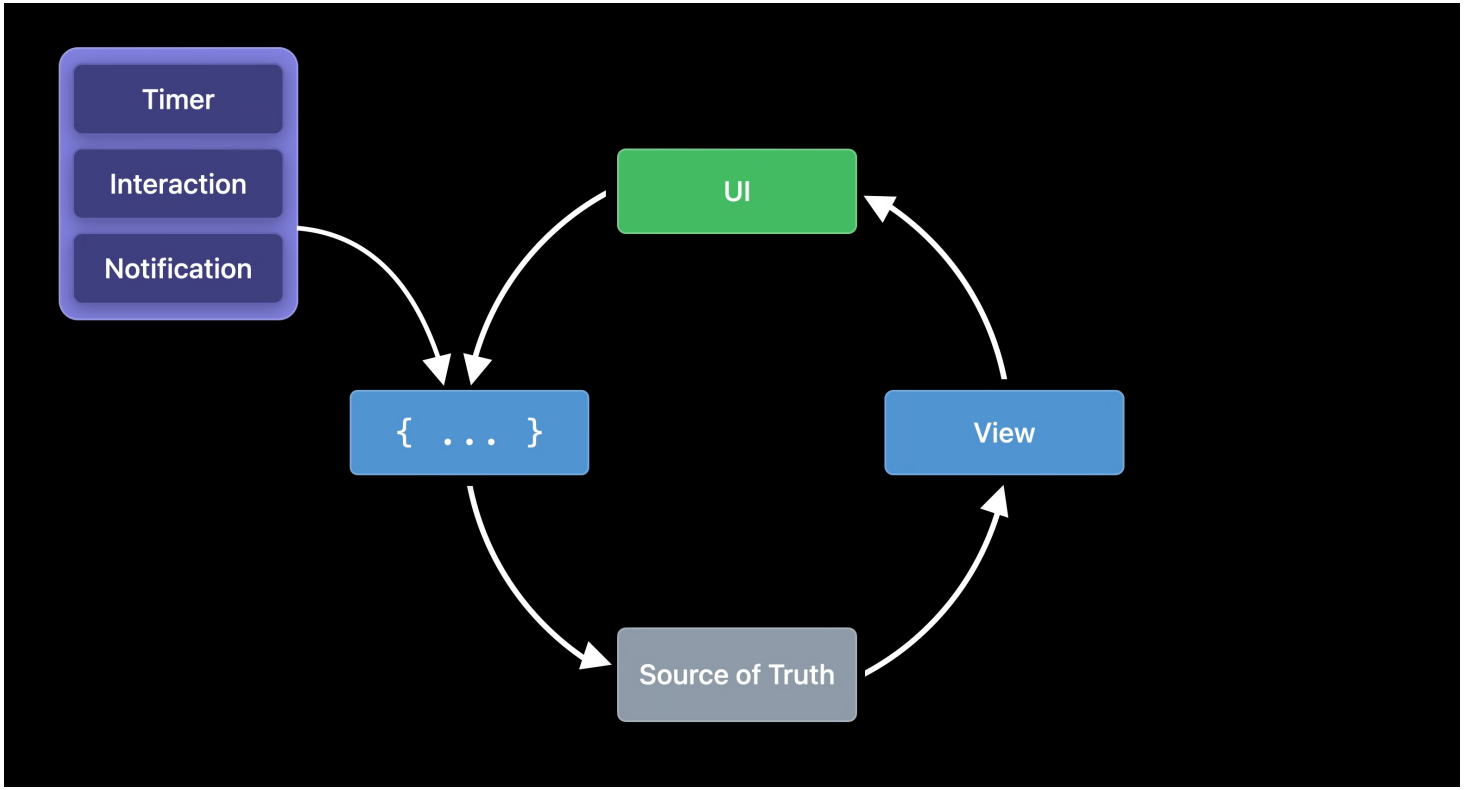
Create ViewModel for the View (cont'd)

Not only @ObservedObject will trigger a View update, changes to @State, @StateObject, @EnvironementObject, will also **trigger View updates**.



Life Cycle?

- SwiftUI does not really have a life cycle. **Views update** when **Source of Truth** (ViewModels, states, etc.) **change**.



See the UI Changes in Preview

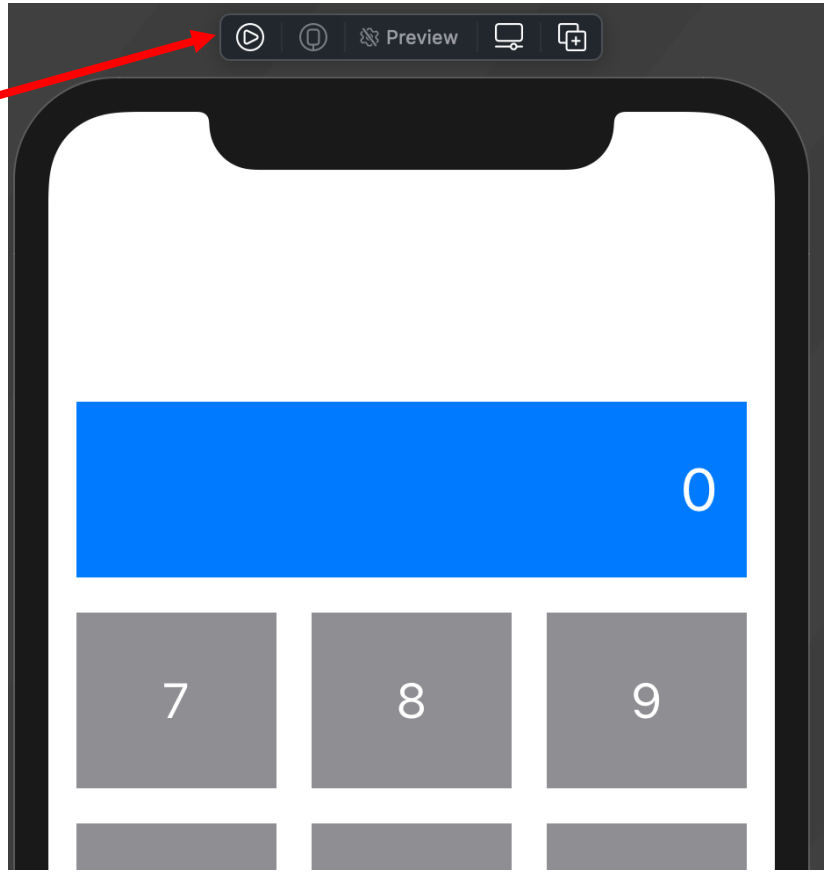
- When creating a SwiftUIView file, the boilerplate code provides a **Preview class**.
- Make necessary modifications this class to correctly display a preview. E.g., adding mock data, change preview device, change orientation, etc.
- Add multiple views in previews to **see how your UI look on different devices**.

```
60  struct CalculatorView_Previews: PreviewProvider {
61      static var previews: some View {
62          CalculatorView()
63              .previewDevice("iPhone 11")
64      }
65  }
```

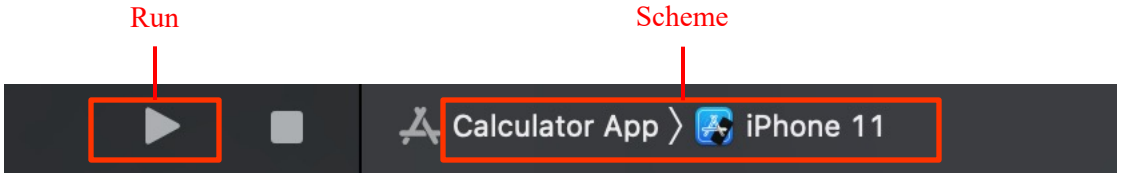
See the UI Changes in Preview

To run the View in real time, press **play button**.

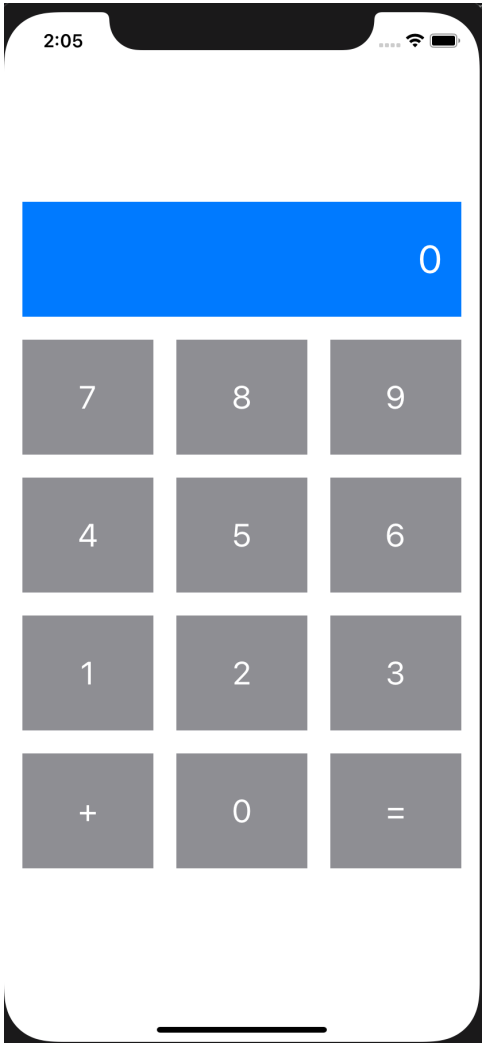
You can now interact with the live preview.



Run your app in the Simulator



- The Scheme pop-up menu lets you choose which simulator or device you'd like to run your app on.
- Click Run button.
- Live preview is like a partial simulator.
- Some functionality only works in simulator now.



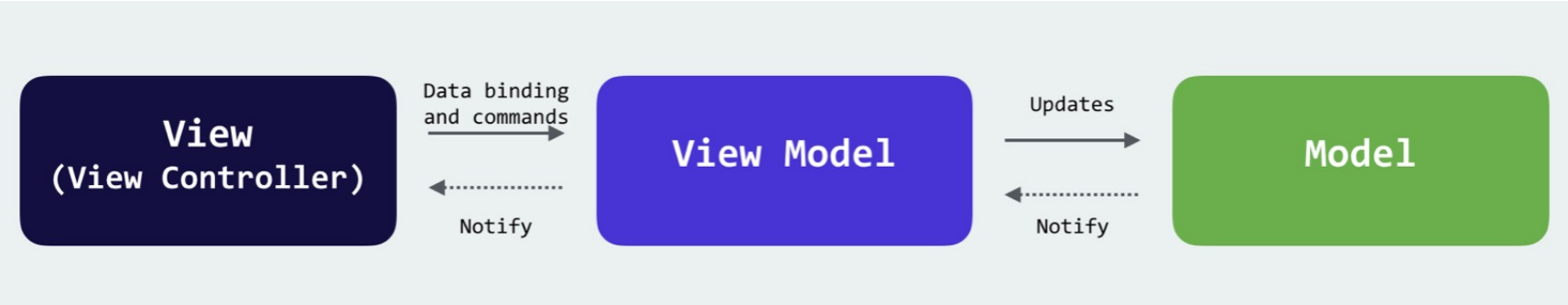
Design Strategy for SwiftUI: Model-View-ViewModel (MVVM)

- Why MVVM
- How MVVM works in iOS development with SwiftUI

Why MVVM?

- **MVVM** is easy for **separate unit testing**. We can test Model and ViewModel easily without touching any View related logic.
- **Objects** in the applications tend to be **more reusable**. A ViewModel can be used by multiple Views, and declarative Views can easily be reused.
- **SwiftUI** is still a **very new framework**, no one knows what's the best design pattern for it. **MVVM generally works the best** for SwiftUI Apps.
- Maybe new design patterns will be invented for SwiftUI.

How MVVM works in iOS development



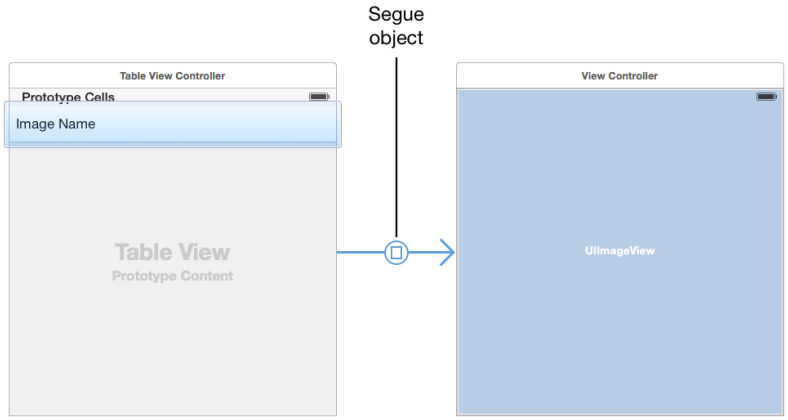
Model = Data Structures, Core Logic
View Model = All data to be presented to the user
View = Show data in View Model to the user as UI

Multiple Views & View Controllers

- Back to MVC (non SwiftUI)
- **Segue**
- Create a **segue between View Controllers**
- Table View Controller & its data source
- Use the prepare method to pass data between view controllers
- Embed a View Controller in a Navigation Controller

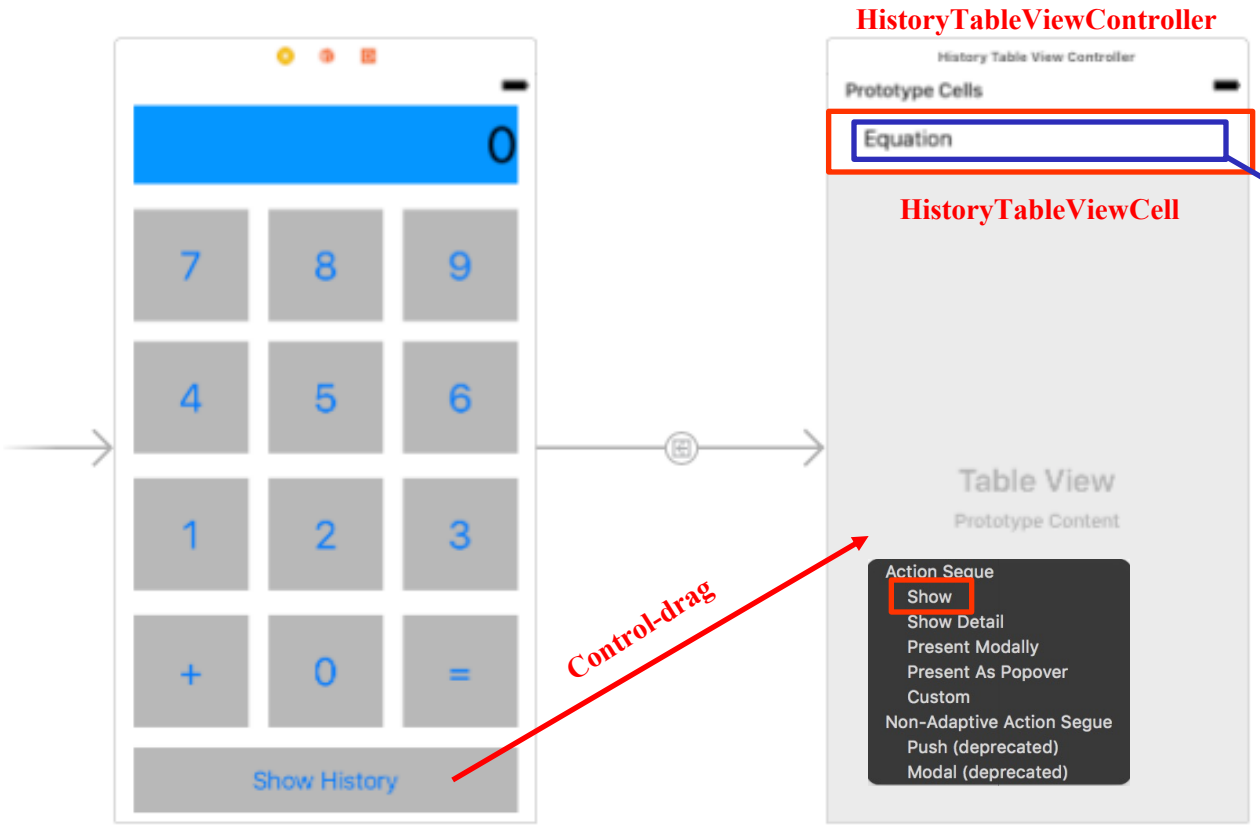
Segue

- A ***segue*** defines a **transition between two view controller**s in your app's storyboard file.
- The **starting point** of a segue is the **button**, **table row**, or **gesture** recognizer that initiates the segue.
- The **end point** of a segue is the view controller you want to display.



Create a segue between View Controllers

Let's say we want to add a "Show History" button at the bottom of the calculator view. And want to display each past equation as a separate row in a Table View.




```
class
HistoryTableViewCell:
UITableViewCell {

    @IBOutlet weak var
equationLabel: UILabel!

}
```

Table View Controller & its data source

```
var equations = [String]()

override func numberOfSections(in tableView: UITableView) -> Int {
        return 1 //return number of sections
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    return equations.count //return number of rows
}

//To configure and set data for your cells
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
"historyTableViewCell", for: indexPath)
    if let historyTableViewCell = cell as? HistoryTableViewCell {
        let equation = equations[indexPath.row]
        historyTableViewCell.equationLabel.text = equation
    }
    return cell
}
```

Question: Where does the *equations* data come from?
See next slide.

Pass data between view controllers

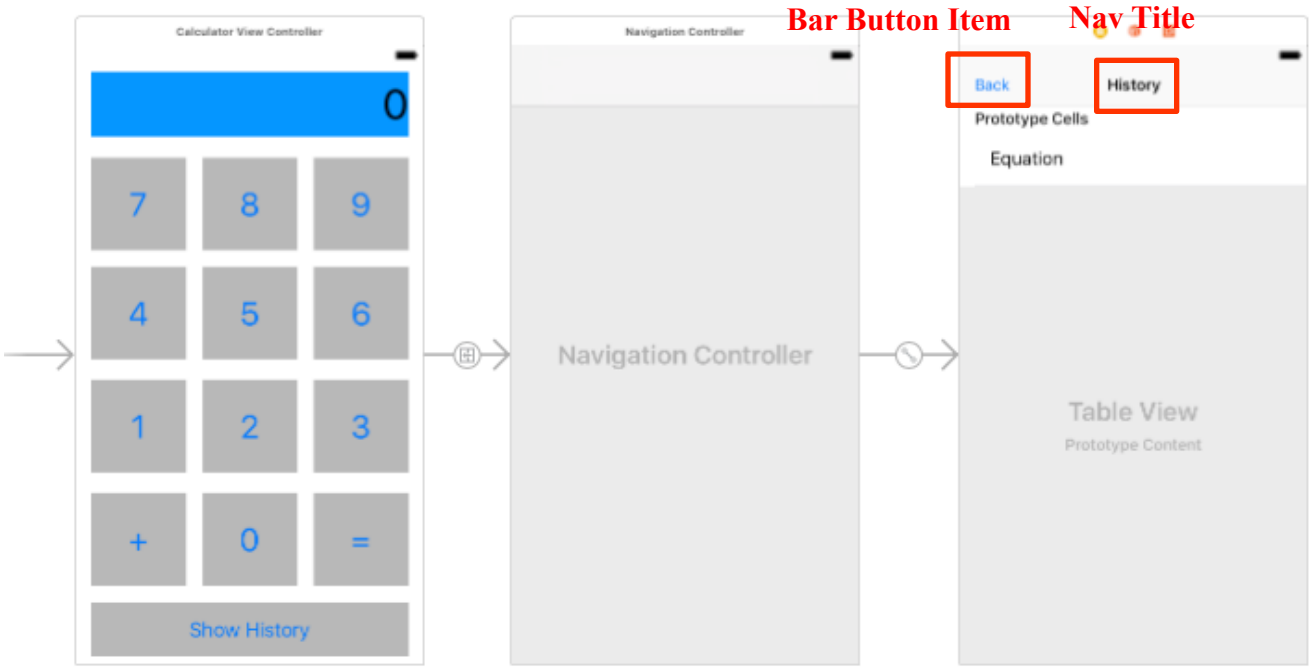
Add a ***prepare*** method in CalculatorViewController to "prepare for" the segue between CalculatorViewController and HistoryTableViewController.

```
// In a storyboard-based application, you will often want to do a
little preparation before navigation
 override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let historyTableViewController = segue.destination as?
HistoryTableViewController {
        historyTableViewController.equations = model.history
    }
 }
```

We also have to modify the model to save equations in history.

Embed a View Controller in a Navigation Controller

Navigation controller allows us to add a title and back button on the top of our history table.

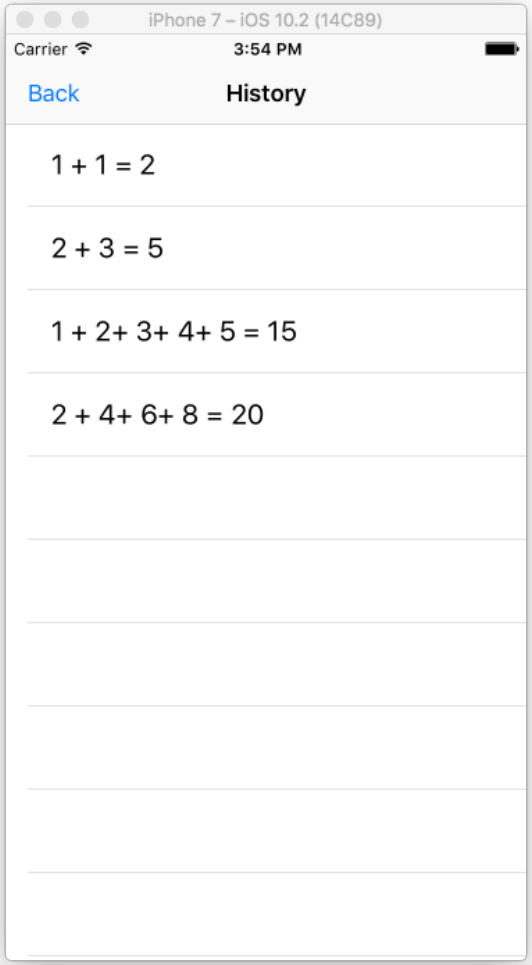


Select the Table View Controller, then choose Editor -> Embeded in -> Navigation Controller

You may find the Table View doesn't show history after this. Why?

Hint: take a look at the "prepare" method. We need to update that method!

Demo



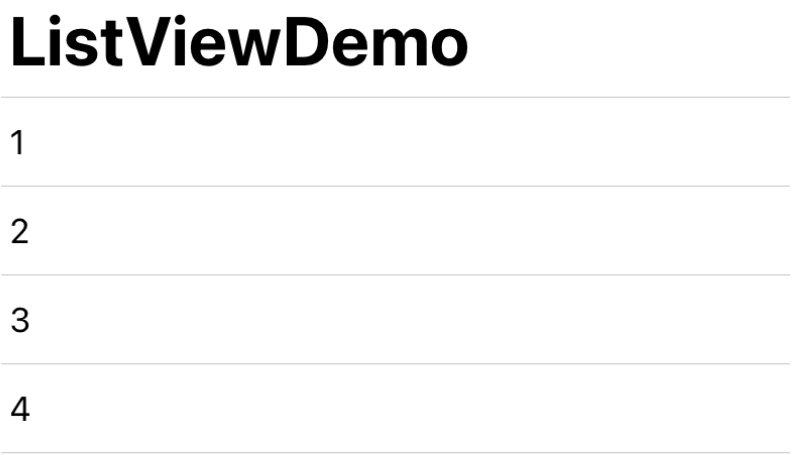
SwiftUI: Multiple Views

- No longer use Segue
- **Navigation Controller** and **List**
- Use **NavigationLink** to go to a new View
- List View and **LazyStacks**
- Use value and @EnviromentValue to pass data.

NavigationController and List

- **NavigationView** will create a **navigation stack**. You can add navigationTitle to it.
- List is SwiftUI's wrapper for UITableView. There is no need to write delegates and data sources anymore! Just provide a list of views to be embedded.

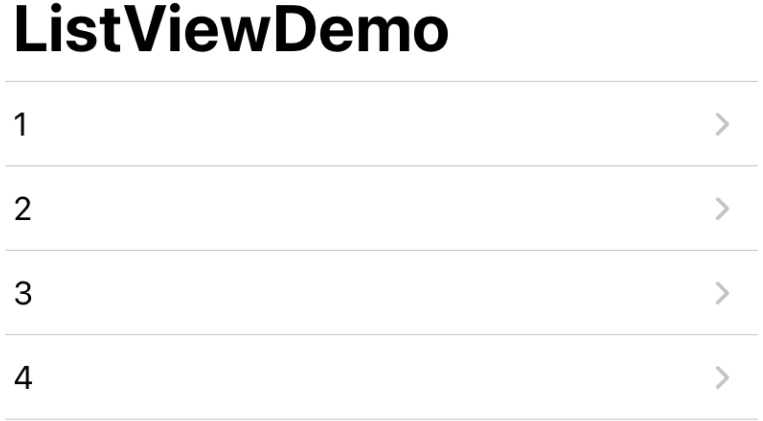
```
10  struct ListDemoView: View {
11      var body: some View {
12          NavigationView {
13              List {
14                  ForEach (1..<5) { num in
15                      Text("\(num)")
16                  }
17              }
18              .navigationTitle("ListViewDemo")
19          }
20      }
21  }
```



Use NavigationLink to Go to a New View

- With in NavigationView, **add NavigationLink** View. When the user clicks the View, NavigationView will push the new view onto the stack.
- Change EmptyView() to any destination View

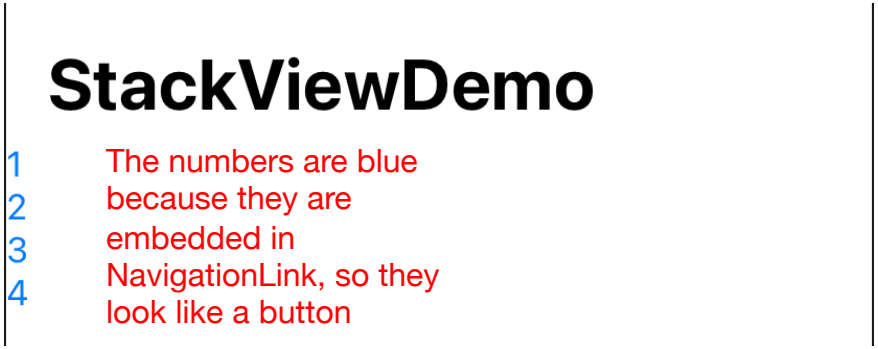
```
10  struct ListDemoView: View {
11      var body: some View {
12          NavigationView {
13              List {
14                  ForEach (1..<5) { num in
15                      NavigationLink(
16                          destination: EmptyView(),
17                          label: {
18                              Text("\(num)")
19                          })
20                  }
21              }
22              .navigationTitle("ListViewDemo")
23          }
24      }
25  }
```



List View and LazyStacks

- **LazyStacks** are native and a new addition to SwiftUI, they act like List View, but are different.
- LazyVStack, LazyHStack, LazyVGrid, LazyHGrid are all new this year (released in June 2020).

```
10  struct ListDemoView: View {
11      var body: some View {
12          NavigationView {
13              VStack{
14                  LazyVStack (alignment: .leading) {
15                      ForEach (1..<5) { num in
16                          NavigationLink(
17                              destination: EmptyView(),
18                              label: {
19                                  Text("\(num)")
20                              })
21                      }
22                  }
23                  Spacer()
24              }
25              .navigationTitle("StackViewDemo")
26          }
27      }
28  }
```



Use Value and @EnvironementValue to Pass Data

- To pass data when switching Views, one way is to **pass data** as a parameter directly to the new View

```
10  struct ListDemoView: View {
11      var body: some View {
12          NavigationView {
13              List {
14                  ForEach (1..<5) { num in
15                      NavigationLink(
16                          destination: MockDestinationView(mockModel:
                                MockModel(mockData: num)),
17                          label: {
18                              Text("\(num)")
19                          })
20                  }
21              }
22              .navigationTitle("StackViewDemo")
23          }
24      }
25  }
26
27  struct MockDestinationView: View {
28      let mockModel: MockModel
29
30      var body: some View {
31          Text("\(mockModel.mockData)")
32      }
33  }
34
35  struct MockModel {
36      var mockData: Int;
37  }
```

Use Value and @EnvironementObject to Pass Data (cont'd)

- You can also **pass data** to all sub-Views using **EnvironmentObject**

```
10  struct ListDemoView: View {
11      @ObservedObject var mockModel: MockModel
12      var body: some View {
13          NavigationView {
14              List {
15                  ForEach (1..<5) { num in
16                      NavigationLink(
17                          destination: MockDestinationView(),
18                          label: {
19                              Text("\(num)")
20                          })
21                  }
22              }
23              .environmentObject(mockModel)
24              .navigationTitle("StackViewDemo")
25          }
26      }
27  }
28
29  struct MockDestinationView: View {
30      @EnvironmentObject var mockModel: MockModel;
31
32      var body: some View {
33          Text("\(mockModel.mockData)")
34      }
35  }
36
37  class MockModel: ObservableObject {
38      var mockData: Int = 10;
39  }
```

HTTP Networking with NSMutableURLRequest

```
let request = NSMutableURLRequest(url: URL(string:
"https://www.google.com")!)
URLSession.shared.dataTask(with: request as URLRequest) {
(data, response, error) in
        guard let httpResponse = response as?
               HTTPURLResponse else {
               //Error
               return
        }

        if httpResponse.statusCode == 200 {
               //Http success
        }
         else {
               //Http error
       }
}.resume()
```

HTTP Networking with AlamoFire

```
14      func getJSON(url: String, callback: @escaping (_ json: JSON) -> Void) {
15          if let url = URL(string: (url)) {
16              print("requesting: \(url)")
17              AF.request(url).validate().responseJSON { (response) in
18                  if let data = response.data {
19                      let json = JSON(data)
20                      callback(json)
21                      return
22                  }
23              }
24          }
```

JSON parsing using Codable

```
do { //Try to parse data to an object of type objectType
let object = try JSONDecoder().decode(objectType.self, from: data)
} //Throws various exceptions if parsing failed
catch DecodingError.dataCorrupted(let context) {
        print(context.debugDescription)
} catch DecodingError.keyNotFound(let key, let context) {
        print("\(key.stringValue) was not
found,\(context.debugDescription)")
} catch DecodingError.typeMismatch(let type, let context) {
        print("\(type) was expected, \(context.debugDescription)")
} catch DecodingError.valueNotFound(let type, let context) {
        print("no value was found for \(type),
\(context.debugDescription)")
} catch let error {
        print(error)
}
```

JSON parsing by extending SwiftyJSON

- Your Model should inherit JSONable protocol
- Use the extended function JSON.to() to parse incoming data.

```swift
protocol JSONable {
    init?(parameter: JSON)
}

extension JSON {
    func to<T>(type: T?) -> Any? {
        if let baseObj = type as? JSONable.Type {
            if self.type == .array {
                var arrObject: [Any] = []
                for obj in self.arrayValue {
                    let object = baseObj.init(parameter: obj)
                    arrObject.append(object!)
                }
                return arrObject
            } else {
                let object = baseObj.init(parameter: self)
                return object!
            }
        }
        return nil
    }
}
```

CocoaPods: Use External Dependencies

- CocoaPods introduction and install
- Add external dependencies

CocoaPods

- CocoaPods **manages dependencies** for your Xcode projects.
- You **specify** the **dependencies** for your project in a simple text file: your **Podfile**. CocoaPods recursively resolves dependencies between libraries, fetches source code for all dependencies, and creates and maintains an Xcode workspace to build your project.
- Install CocoaPods:

```
$ sudo gem install cocoapods
```

- To use it in your Xcode projects, run it in your project directory:

```
$ pod init
```

Add dependecies by CocoaPods

- Add dependencies in a text file named **Podfile** in your Xcode project directory

```
target ‘MyApp’ do
  use_frameworks!

  pod 'McPicker'
  pod 'SwiftSpinner'

end
```

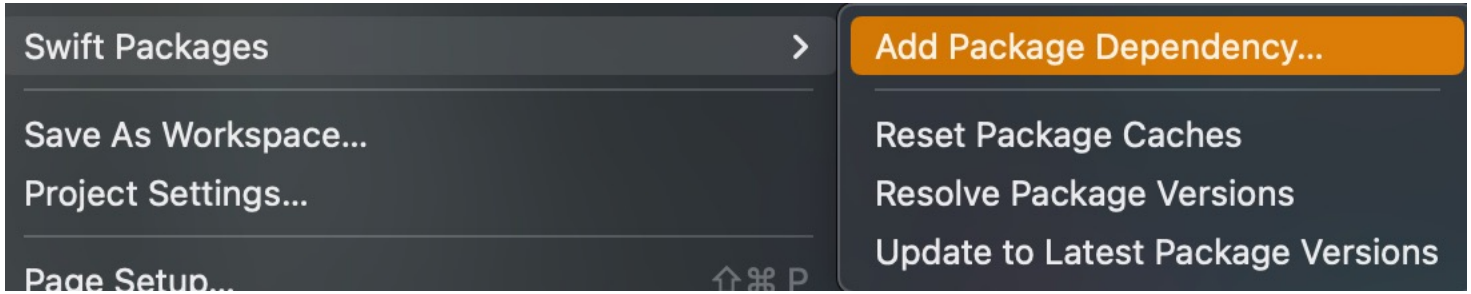
- Install the dependencies in your project:

```
$ pod install
```

- Make sure to always open the Xcode workspace (*.xcworkspace) instead of the project file (*. xcodeproj) when you use CocoaPods with your project

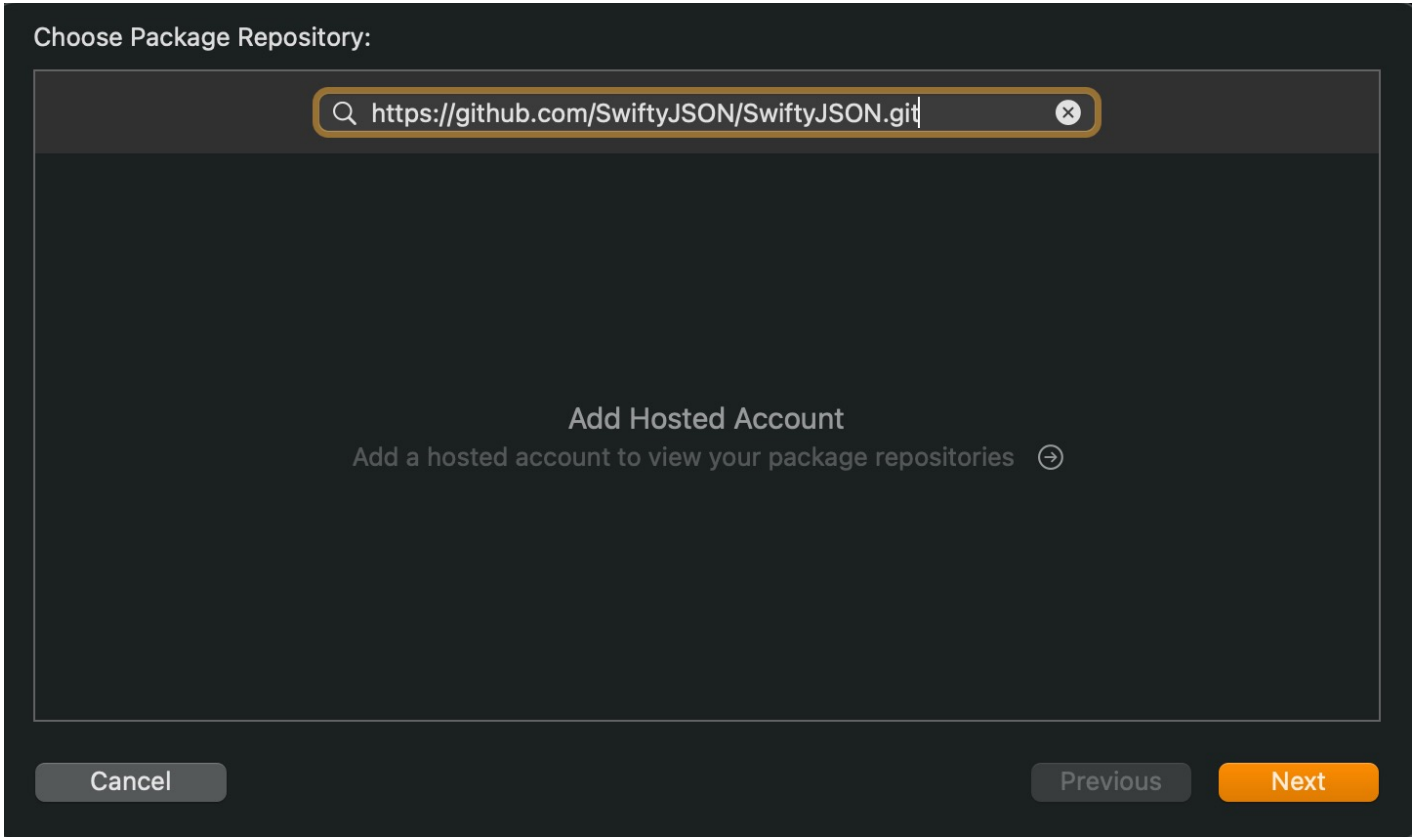
Swift Package Manager

- In Toolbar, click File -> Swift Packages -> Add Package Dependency to add packages.
- New **alternative to using CocoaPods**



Swift Package Manager (cont'd)

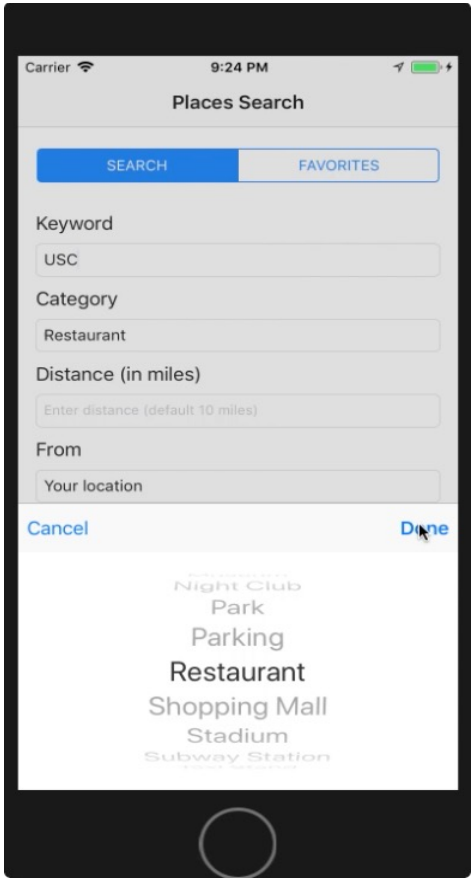
- Then copy the **git** repo link and add the dependency



UIPickerView drop-in solution - McPicker

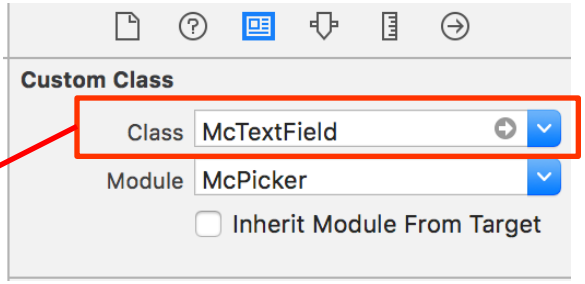
- The UIPickerView is an **alternative of dropdown list** in iOS. However, it usually takes up a lot of spaces on the screen.
- So instead of showing the UIPickerView directly, the McPicker allows us to bind it with a Text Field and display it when the Text Field is tapped.
- Usage: add "McPicker" in the Podfile and run **pod install**

```
target 'MyApp' do
  use_frameworks!

  pod 'McPicker'
end
```



UIPickerView drop-in solution – McPicker (cont'd)

- Set the custom class of a Text Field to "McTextField", and control-drag it into the code




```
import McPicker

@IBOutlet weak var mcTextField: McTextField!

override func viewDidLoad() {
  let data: [[String]] = [["Option1", "Option2", "Option3", "Option4"]]
    let mcInputView = McPicker(data: data)
    mcTextField.inputViewMcPicker = mcInputView

    mcTextField.doneHandler = { [weak mcTextField] (selections) in
        mcTextField?.text = selections[0]!
                //do something if user selects an option and taps done
  }

    mcTextField.cancelHandler = { [weak mcTextField] in
        //do something if user cancels
  }
}
```

Activity Indicator - SwiftSpinner

- There are circumstances in which you don't want the user to see the current screen contents while you are loading or processing data.
- The SwiftSpinner uses dynamic blur and translucency to overlay the current screen contents and display an activity indicator with text (or the so called "spinner").
- It's super easy to use:

```
import SwiftSpinner

SwiftSpinner.show("Connecting to satellite...")
//connecting
SwiftSpinner.show("Failed to connect, waiting...",
animated: false)

SwiftSpinner.hide()
```

Activity Indicator – SwiftSpinner (cont'd)

- This is how the activity looks like

Activity Indicator – SwiftUI

- **SwiftUI** provides a native activity indicator called **ProgressView**()

ProgressView() with value

```
ProgressView("Downloading…", value: 50, total: 100)
    .padding()
```

ProgressView() without value

Loading web images with KingFisher

- KingFisher is a library that works just like an **Image view.**
- It can **load**, and even **cache web images**.
- You can use any Image view modifiers for KFImage view.

```
KFImage(URL(string: news.urlToImage) ?? URL(string:
    "https://www.publicdomainpictures
    .net/pictures/30000/velka/plain-white-background.jpg")!)
```

Animation – SwiftUI

- SwiftUI provides an easy way to do animations.
- Wrap withAnimation{} around your code when changing @State. It's done!

```
10  struct AnimationDemoView: View {
11      @State var hello: Bool = true
12      var body: some View {
13          VStack {
14              if hello {
15                  Text("Hello World")
16              }
17              else
18              {
19                  Text("Goodbye World")
20              }
21              Button("Change") {
22                  withAnimation{
23                      hello = !hello
24                  }
25              }
26          }
27
28      }
29  }
```

References

- A perfect IOS App example with step-by-step instructions
- IOS course by Stanford : Developing iOS 11 Apps with Swift
- iTunes U collections are moving to Podcasts
- The online Swift Language guide by Apple
- iBook: The Swift Programming Language (Swift 5.1)
- iBook: App Development with Swift
- https://developer.apple.com/videos/play/wwdc2020/10040/
- https://developer.apple.com/xcode/swiftui/