Johns Hopkins Engineering

605.487 – iOS Development

Module 1 - UITableView

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

UITableView

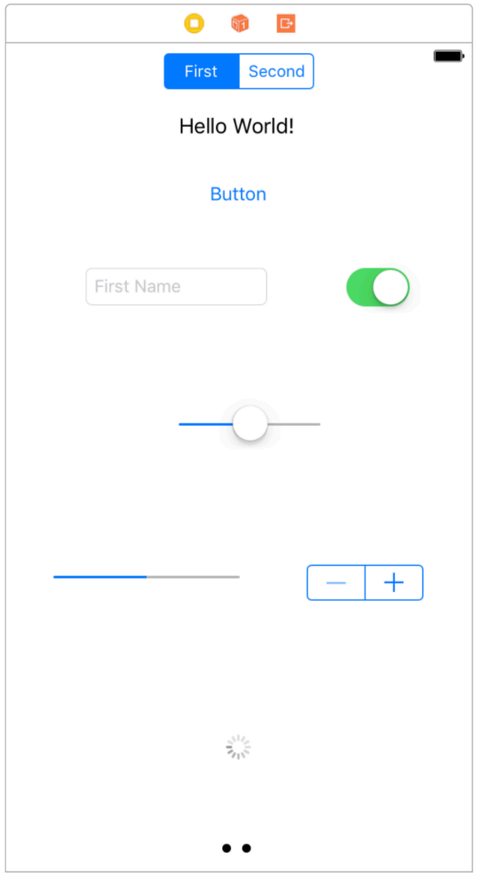
One of the most common ways to layout data in iOS, since the beginning, has been in the form of a UITableView. Before we talk about UITableViews and how to make them, we need to touch on a few more topics:

- What is a view?
- What is Model-View-Controller, and why is it important?
- How are these implemented in Xcode?

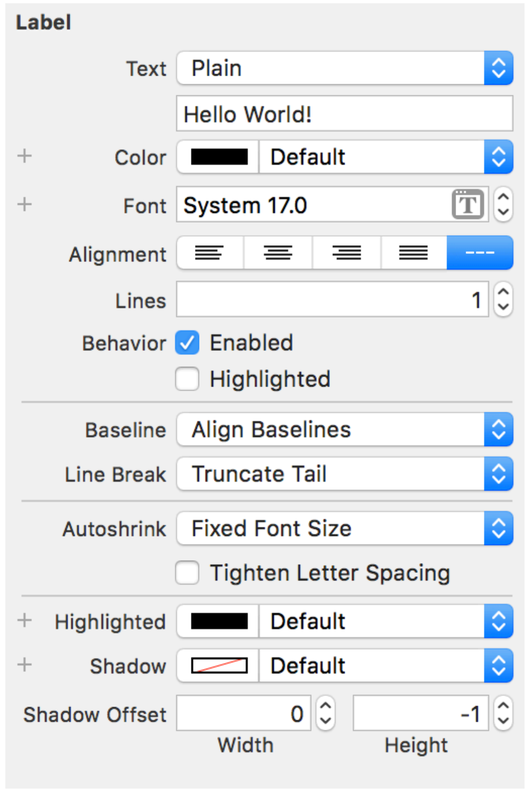
And given that background, we'll discuss:

- What is UITableView?
- How can it be super powerful and not very powerful at the same time?
- How can Protocols and Delegates help?

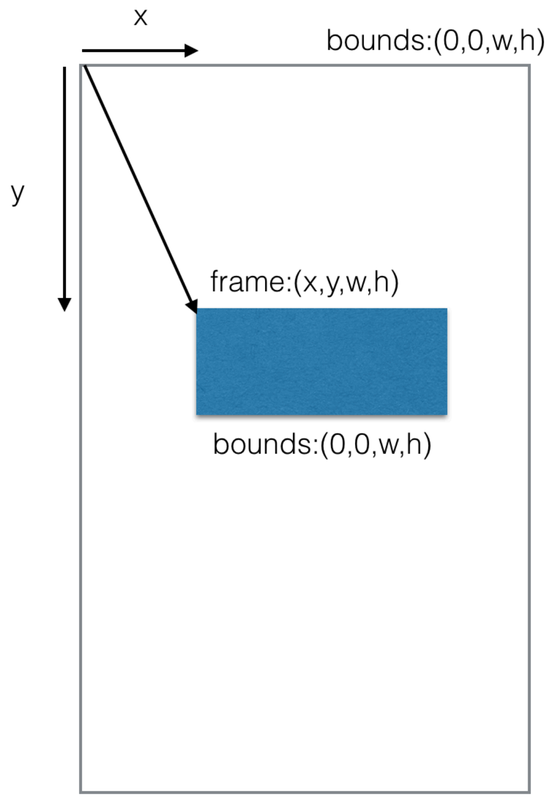
What is a UIView?



UI Widgets in Interface Builder

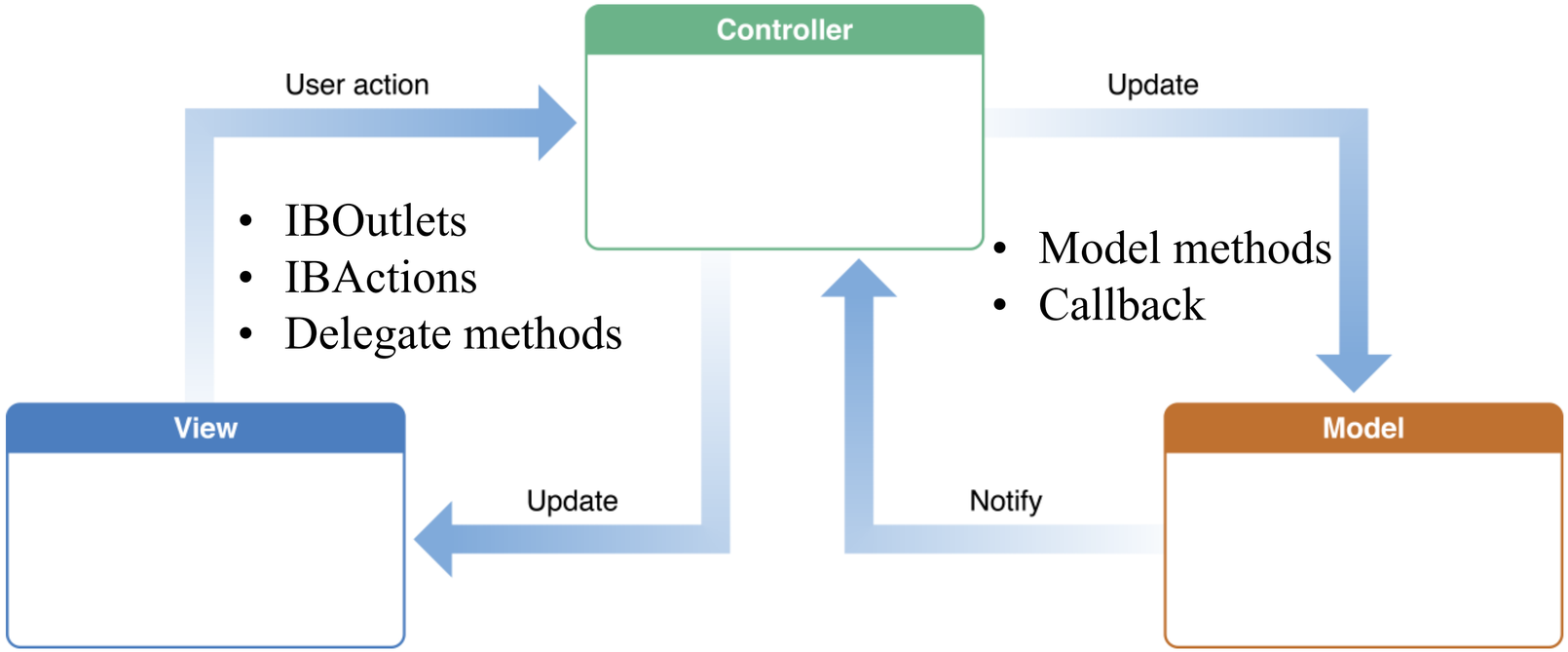


Easy to set attributes



Frame vs Bounds

What is Model View Controller? Why is it important?



- Highly reusable
- Know nothing about model

- App Specific
- Know nothing about views

Graphic courtesy: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

How are these implemented in Xcode?

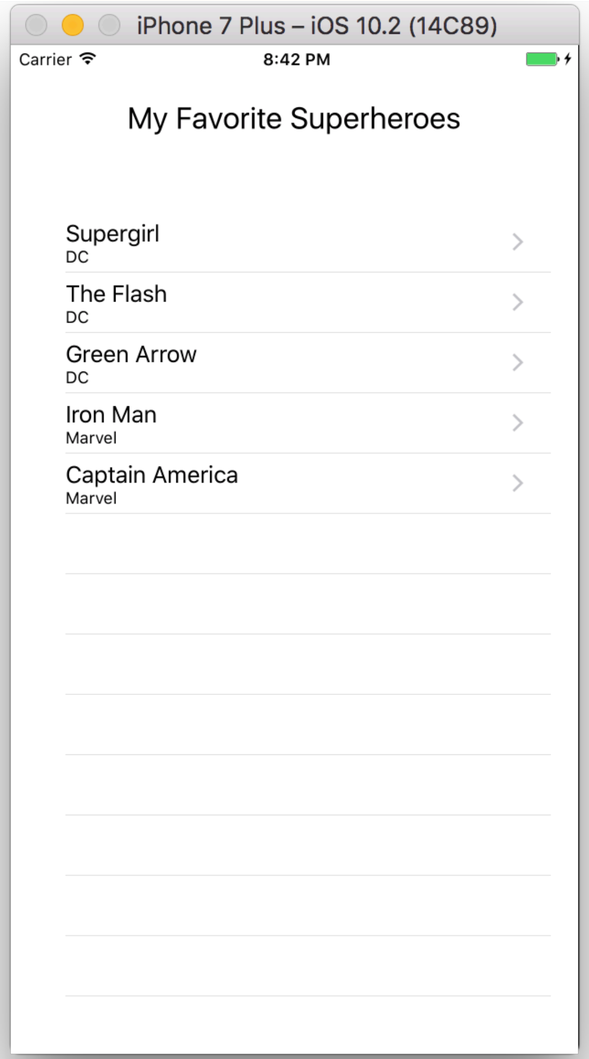
In Xcode:

- **Views** are your UI widgets. These can be laid out in Interface Builder, constrained to the exact location you need them on screen.

- Your **model** logic lives in your custom app classes, interfacing with your custom app data, in structures like databases

- Generic versions of **controllers** are actually provided in the iOS SDK, with the `UIViewController` acting as a high level base class. In your applications, you will subclass the provided UIViewControllers to customize your app's behavior, look, and feel.

What is a UITableView?



Super Powerful? Super Weak?

- The UITableView widget is extremely powerful!
  - It scrolls very smoothly, even for large datasets
  - It can display data in groups called sections
  - It can display titles for the sections, in headers and footers
  - Users can rearrange the rows with the drag of a finger.
  - And you don’t have to write any user interface code!

- But the UITableView widget by itself is not very powerful!
  - It doesn’t know what data to load
  - It doesn’t know the amount of rows to display
  - It doesn’t know how the data should be displayed in the rows.

- These are all things that are very specific to YOUR app. The UIView components however, are specifically designed to be reusable and generic - so how do we bridge the gap?

Protocols and Delegates to the Rescue!

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Protocols and Delegates

Protocols

- A protocol in Swift defines a template of properties, methods and other requirements that a class, structure, or enumeration can implement (or adopt). If one of those types adopts a protocol, it *conforms* to that protocol.

- Protocols can also be extended, allowing you to implement some of the requirements, or to add additional requirements.

```
protocol Person
{
  func getAge() -> Int
  func getFirstName() -> String
  func setGender(_ gender:String)
}
```

Basic Syntax

```
protocol MyProtocol
{
   //definition here…
}

class MyClass: MyParentClass, MyProtocol,
MyOtherProtocol
{
   //class definition here
}
```

Properties

```
protocol Person

{
  var age:Int { get }
  var firstName:String { get }
  var gender:String { get set }


}
```

Methods

```
protocol Person
{
  func getNumberOfYearsTeaching() -> Int

  //no default parameters
  //variadic parameters ok

}
```

Init methods

```
protocol Person
{
   init(firstName:String)
}
```

//when implementing, required keyword must be used

```
class Teacher: Person
{
  required init(firstName:String)
}
```

Conforming to a Protocol

```
class Teacher: Educator, Person


{
    var age:Int
    var firstName:String
    var gender:String

    required init(firstName:String)
    {
        self.firstName = firstName
        self.age = 25
        self.gender = “”
    }



    func getAge() -> Int { return age }
    func getFirstName() -> String { return firstName }
    func setGender(_ gender:String) { self.gender = gender }


}
```

Protocols as Types

- Even though they don’t implement anything, Protocols are still consdiered types
- This means that they can be used as a parameter type in a function, or a constant, variable or property type
- Remember: the type associated restricts the methods that can be called on that object to the ones declared by that protocol

Protocol Inheritance

- Protocols can inherit from other protocols to add other requirements

```
protocol MyProtocol: FirstProtocol, SecondProtocol
{
    //definition here
}
```

Sidebar: Extensions

- Extensions can be used to provide new functionality to a given type – including protocols. This is even true for iOS level types that you don't have the source code for!

```
extension String
{
    func sayLoudly()-> String
    {
        return ”\(self.capitalized)!!!!”
    }
}
```

Protocols and Extensions

- You can use an extension to show that a type conforms to a given protocol. Extensions can exist in the same file as the original class, so separating protocol conformity by extension is a great way to syntactically group your code into logical chunks

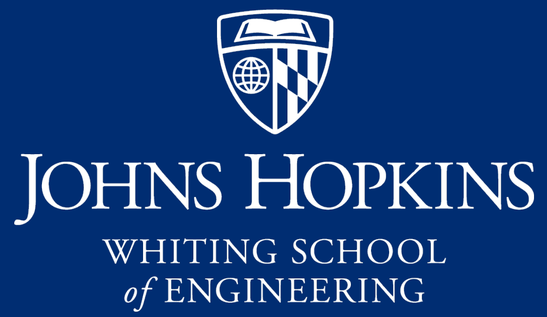
```
extension Person: HumanBeing
{
   var timeOnEarth:String = { “\(age) Years” }
}
```

Delegation

- Classes or structures can choose to *delegate* some of their work to another object. For this to work, the delegate object must conform to a given protocol; this guarantees the delegating object can call the proper methods on the delegate object.

```
protocol EPPInfoDelegate
{
  func getNumberOfYearsTeaching() -> Int
}

class EPPManager: EPPInfoDelegate
{
  func getNumberOfYearsTeaching() -> Int
  {
      ……
      return numYears
  }
}
```

```
class Teacher: Educator, Person
{
  var delegate:EPPInfoDelegate?

  init(delegate:EPPInfoDelegate)
  {
    self.delegate = delegate
  }

  func getNumberOfYearsTeaching() -> Int
  {
    return delegate?.getNumberOfYearsTeaching()
  }
}
```



Customizing UITableViews

Customizing UITableView

- Through the use of delegate methods, you can customize the appearance and behavior of a UITableView
  - UITableViewDelegate - primarily used to customize the appearance of the UITableView and rows, and the behaviors for selection and editing
  - UITableViewDataSource – primarily used to define the data to load into the UITableView, information about table header and footers, and cell capabilities (moving, editing)

Defaults and Optional Functions

- Remember, a protocol can have both *required* and *optional* functions. In fact, some protocols, like UITableViewDelegate, are comprised entirely of optional methods

- Some optional methods may have default implementations defined in case you don't implement them in your code. (`numberOfSections(in tableView: UITableView)`, for example)

Adopting Protocols with an Extension

- As discussed in the Protocols and Delegates demo, you can use an extension to specify that a type conforms to a protocol.

```
extension MyClass: UITableViewDelegate, UITableViewDataSource
{
    //UITableViewDataSource
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
    {
        //……
    }
    //UITableViewDelegate
    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat
    {
        //……
    }
}
```

Setting the Delegate

- One of the most common problems people have with UITableView (and classes that use the delegate pattern in general) is *forgetting to set the delegate objects*.

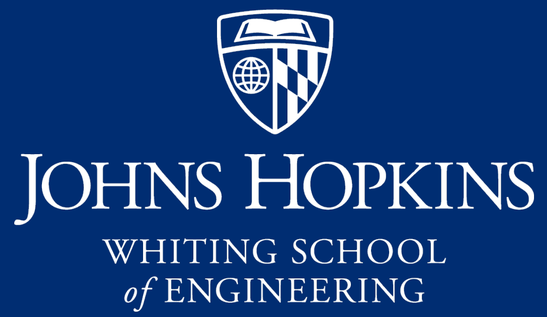
```
class ViewController
{
    @IBOutlet weak var tableView:UITableView!

    override func viewDidLoad()
    {
         super.viewDidLoad()
         tableView.delegate = self   //THIS!
         tableView.dataSource = self //THIS!
     }
}

extension ViewController: UITableViewDelegate,
UITableViewDataSource
{
    //protocol implementions here
}
```

UITableViewDataSource

- `tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int`
- `tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell`
- `numberOfSections(in tableView: UITableView) -> Int`
- `tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?`
- `tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool`
- `sectionIndexTitles(for tableView: UITableView) -> [String]?`
- `tableView(_ tableView: UITableView, sectionForSectionIndexTitle title: String, at index: Int) -> Int`
- `And more!`

UITableViewDelegate

- `func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath)`
- `tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat`
- `tableView(_ tableView: UITableView, estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat`
- `tableView(_ tableView: UITableView, viewForHeaderInSection section: Int) -> UIView?`
- `tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)`
- `And more!`



Using Segues to Change Screens

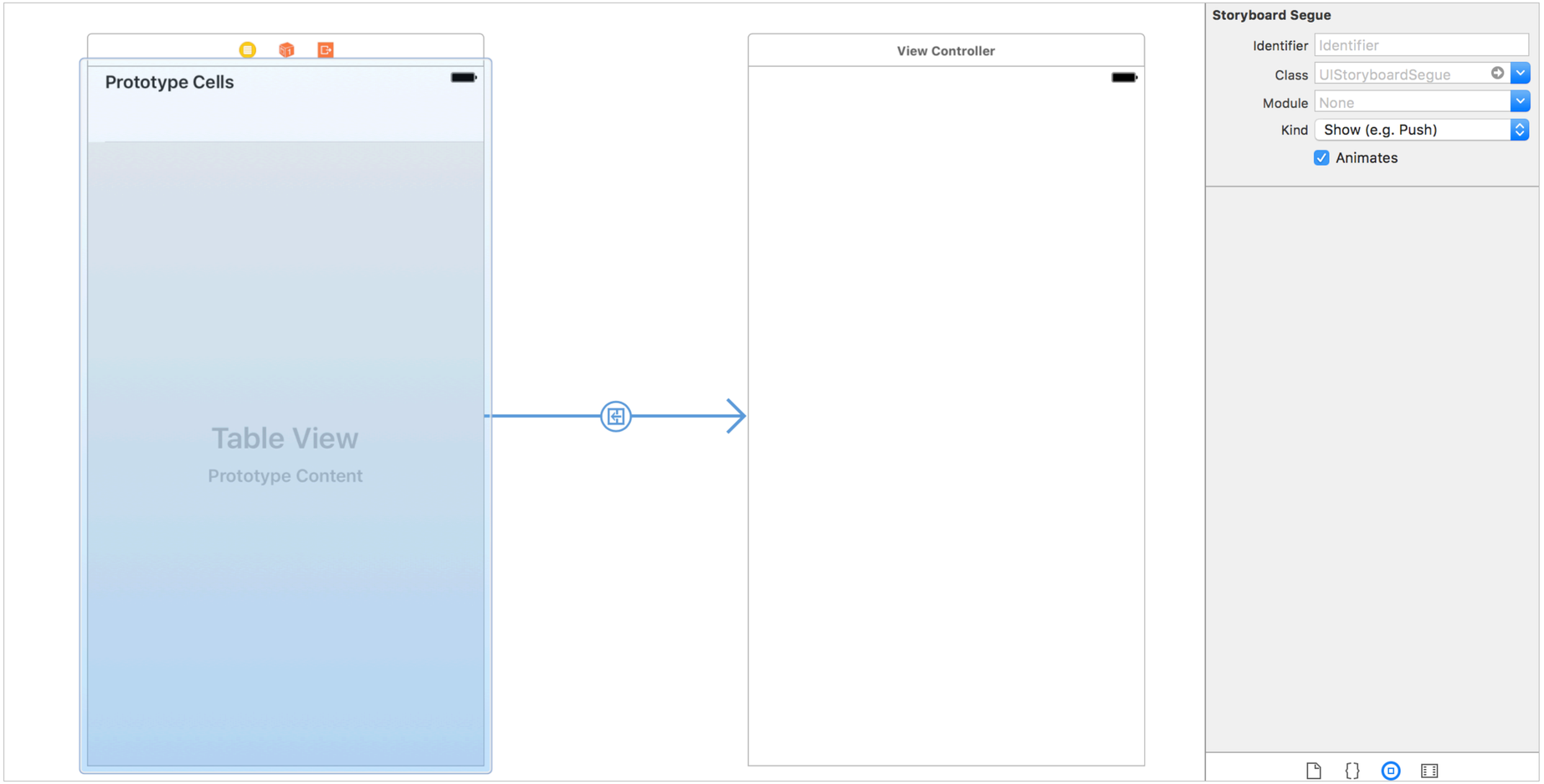
One Window, Many Screens

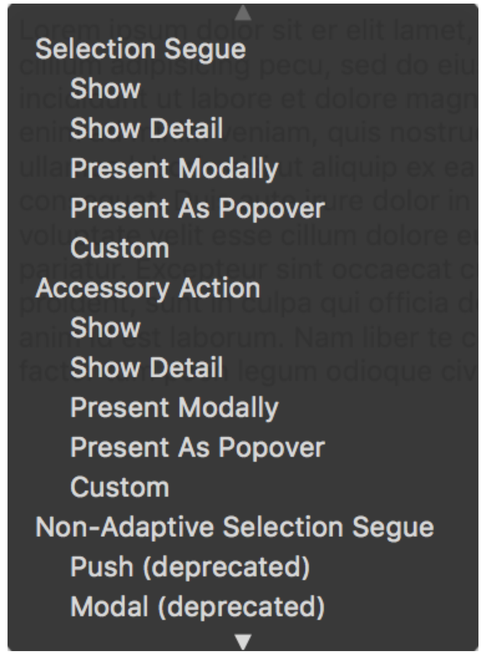
- While your app is confined to a window, the data presented in your app is separated into distinct screens
- Each screen is represented by a UIViewController (or one of its child classes)
- One way to transition between screens is to use a segue (UIStoryboardSegue)

UIStoryboardSegue

- The UIStoryboardSegue type represents a segue, which is a transition between 2 view controllers in your storyboard
- The start of the segue is a UIView based object such as a button or table view cell, or an action based object like a gesture recognizer. Then end of the segue is the view controller you want to present
- The new view controller can be presented in a few different ways – for now, we'll stick with the simple "show" style

Connecting with a UIStoryboardSegue



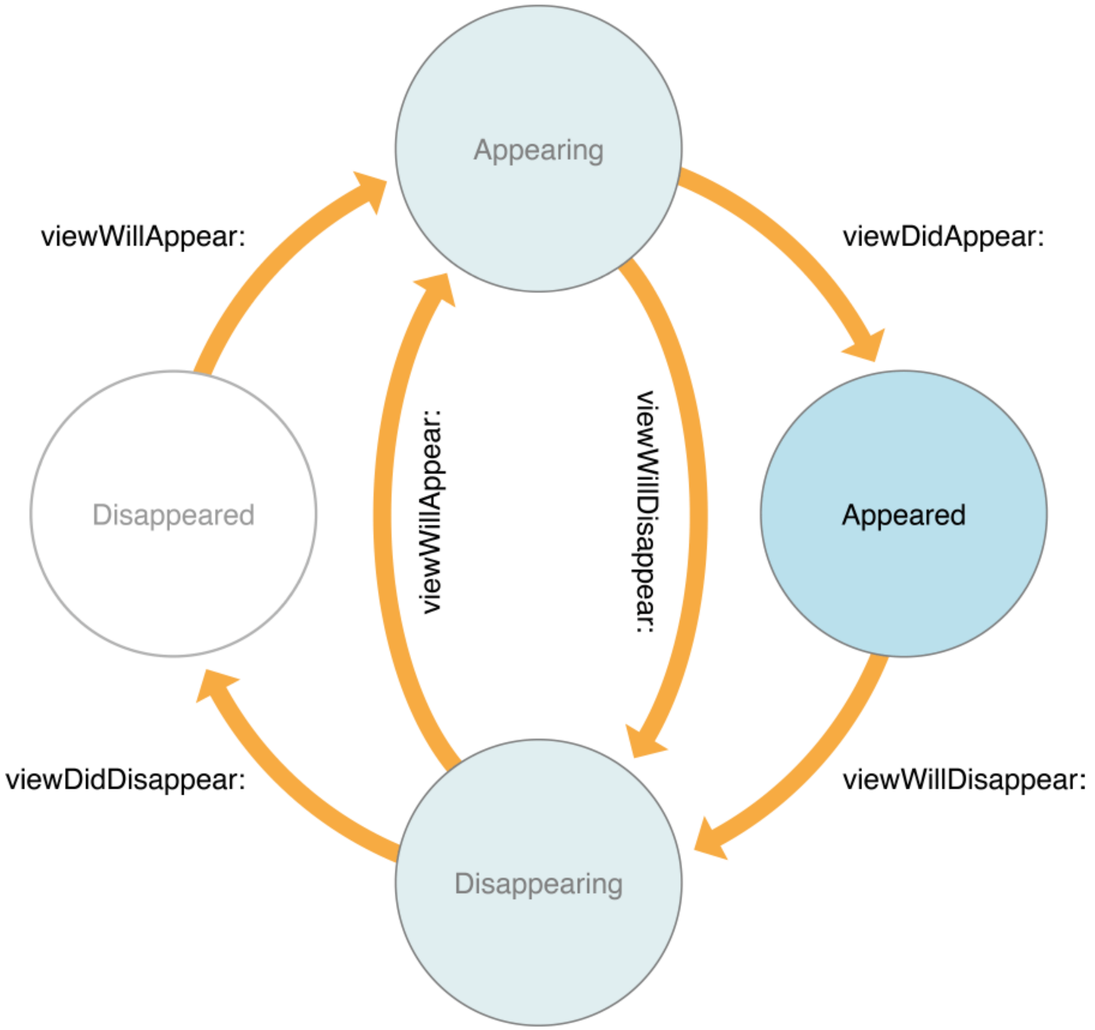


Sidebar: UIViewController Lifecycle

- UIViewControllers go through a series of method calls in order to initialize the class and render the elements on screen:

init(coder:)	When View Controllers created from Storyboards, this method is called (you'll need to override)
loadView	Creates the UIView for the view controller. Typically not implemented unless making entire view from scratch
viewDidLoad	Here, view has been created, and all outlets are assigned. You can set your UI before presentation (Called once)
viewWillAppear	Called each time before the view controller comes on screen. Can be used to update the UI between appearances
viewDidAppear	Called after the view has appeared on screen. Start animations, sounds, etc here.
viewWillDisappear	Called before the transition to the next view controller happens
viewDidDisappear	Called after the view controller gets removed from the screen; here you can turn off timers, sounds, etc.

Sidebar: Transitions between Appearance States



https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/

prepare(for segue:UIStoryboard, sender:Any?)

- Provides a reference to the segue used to transition away from the current UIViewController
- The UIStoryboardSegue object provides a reference to the destination view controller. With a cast, you can set properties on that view controller before it loads on screen
- Can check for a particular segue by querying the segue's `identifier` property

Unwind Segues

- Once you've transitioned to a new view controller via a segue, how do you get back to the original view controller?
- Unwind segues work by connection some actionable item – like a button – to the exit object of the view controller from which you are transitioning
- The view controller you are returning to needs to implement

```
@IBAction func myUnwindAction(unwindSegue: UIStoryboardSegue)
```

Presenting a segue programmatically

- Segues can be triggered programmatically instead of being tied to an immediate user interaction. UIViewController has a method called

```
performSegue(withIdentifier: String, sender: Any)
```

- This requires that the segue be given an identifier in its properties tab.

JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING