Still busy!

- Multiple MVCs
  - Split View Controllers
  - Navigation Controllers
  - Tab Bar Controllers
- Segues
- Popovers
- VC Lifecycle

... and once again plenty of demos!

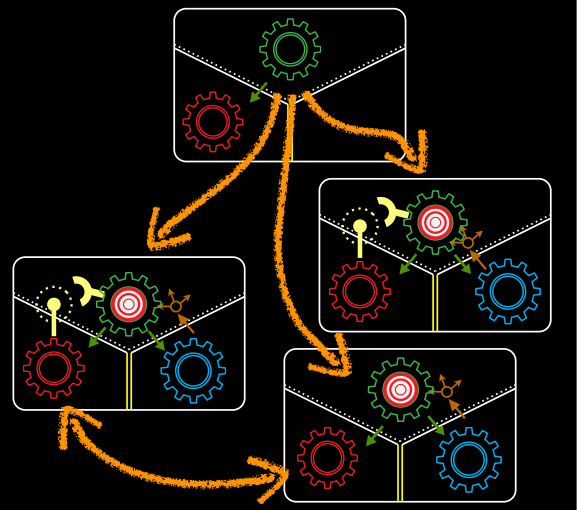
Multiple Model View Controllers

Multiple MVCs

- **You've probably got a pretty good handle on the basics of this**
  - Your Controller in an MVC grouping is always a subclass of UIViewController
  - It manages a View (made up of subviews that you usually have some outlets/actions to/from)
  - It is the liaison between that View and the Model (which is UI-independent)
- **How do we expand our application to use multiple MVCs?**
  - We need infrastructure to manage them all
  - iOS5 introduced storyboards in conjunction with "controllers of controllers" so that it is all a matter of connecting MVCs graphically in Interface Builder!

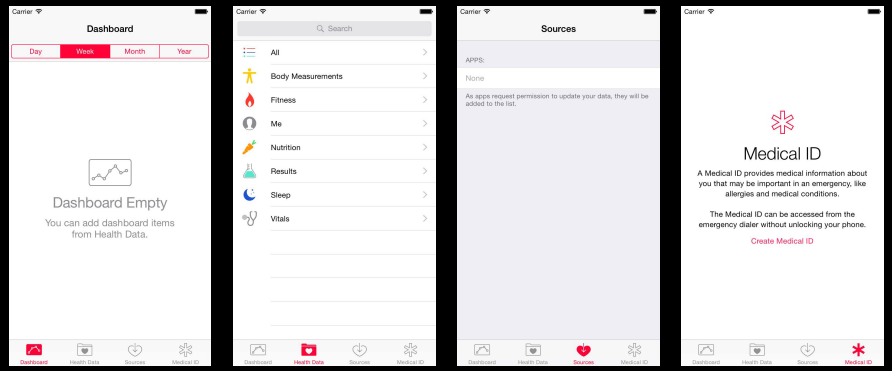
Multiple MVCs

- **Combine MVCs to to build more complex Apps**
- **iOS provides some Controllers whose View is other MVCs**
- **Examples:**
  - UITabBarController
  - UISplitViewController
  - UINavigationController



Multiple MVCs: UITabBarController

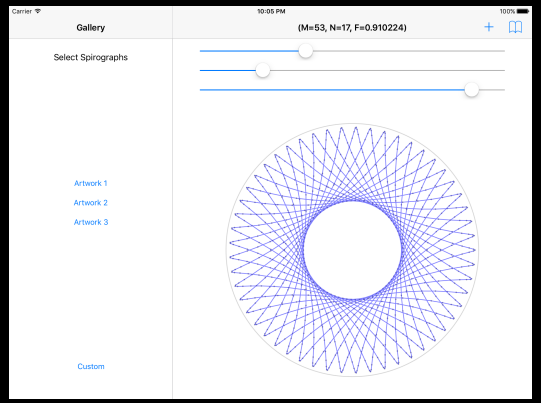
- Let the user choose between different MVCs



Multiple MVCs: UISplitViewController

- Display two MVCs side-by-side

Master



Detail

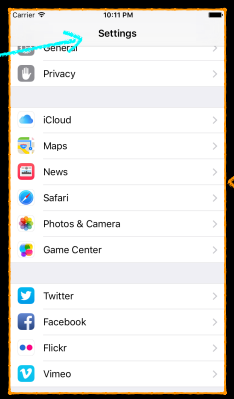
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)

This top area is drawn by the UINavigationController

But the contents of the top area (like the title or any buttons on the right) are determined by the MVC currently displayed (in this case, the "Settings" MVC)

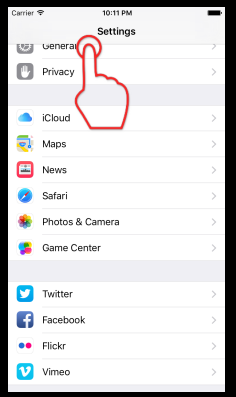
Each MVC communicates these contents via its UIViewController's navigationItem property



"Settings" MVC

Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)

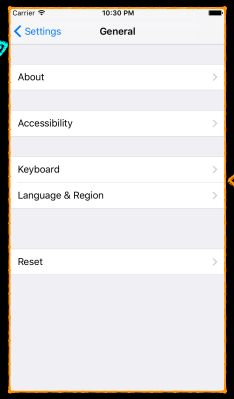


Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)

Note: "back" button has appeared

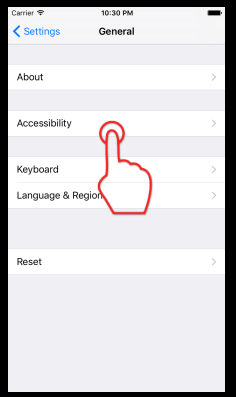
Placed here automatically by the UINavigationController



"General" MVC

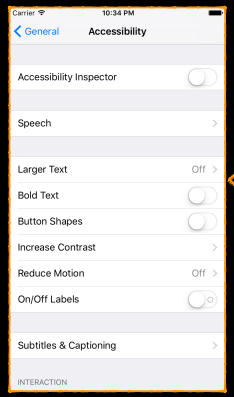
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



Multiple MVCs: UINavigationController

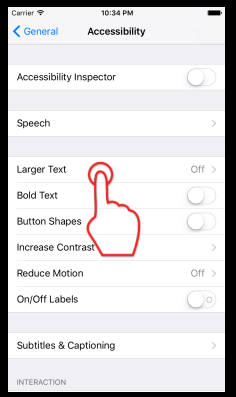
- Let the user push and pop MVCs off of a stack (like a stack of cards)



"Accessibility" MVC

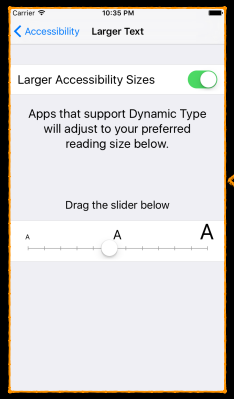
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



Multiple MVCs: UINavigationController

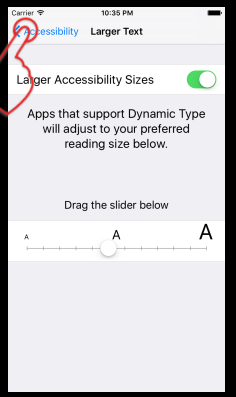
- Let the user push and pop MVCs off of a stack (like a stack of cards)



"Larger Text" MVC

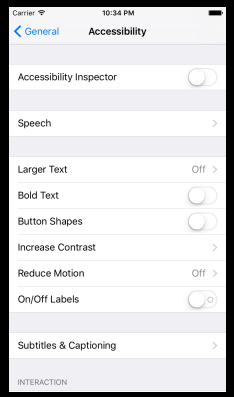
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



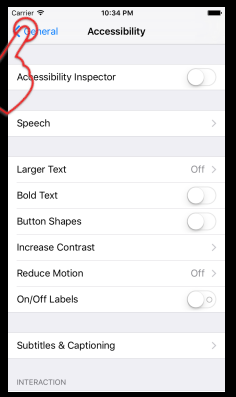
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



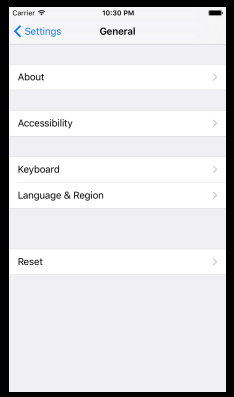
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



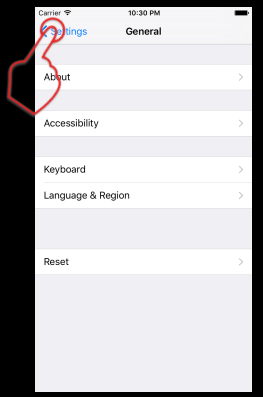
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



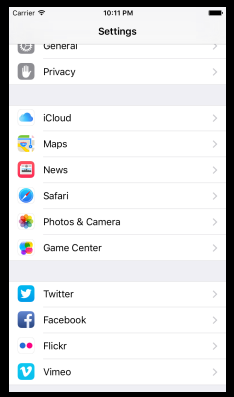
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)



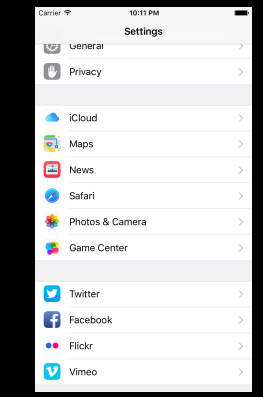
Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)

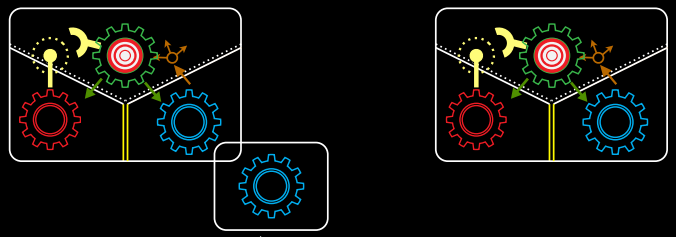


Multiple MVCs: UINavigationController

- Let the user push and pop MVCs off of a stack (like a stack of cards)

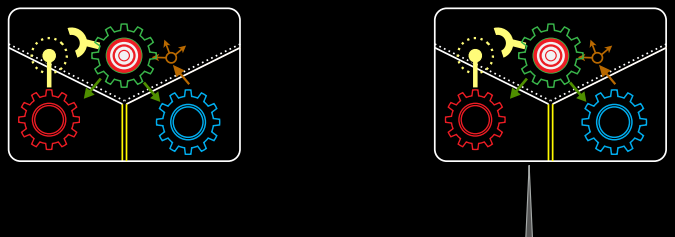


Multiple MVCs: UINavigationController



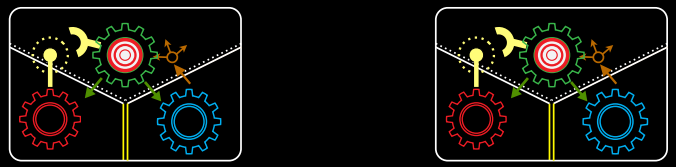
Need additional features without cluttering UI and putting them in single MVC

Multiple MVCs: UINavigationController



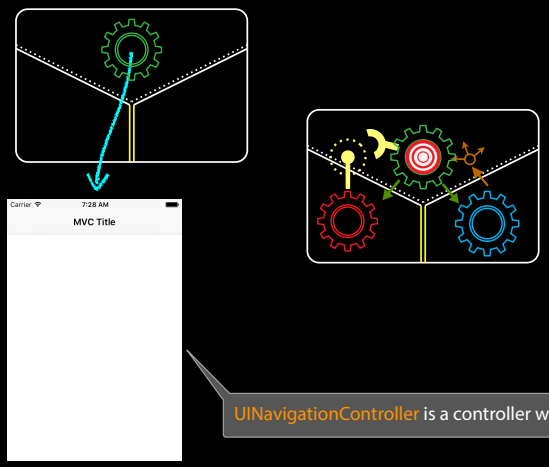
Solution: create a new MVC to encapsulate these features

Multiple MVCs: UINavigationController



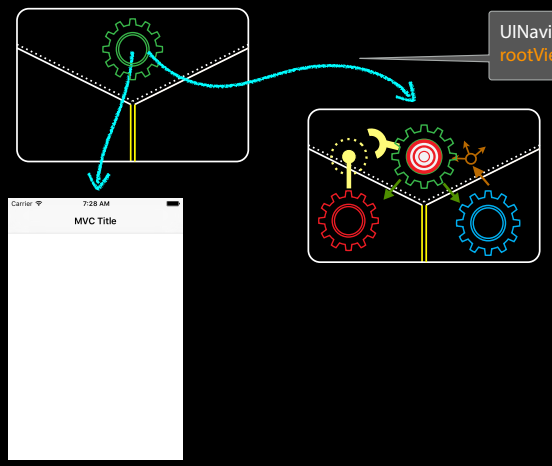
Solution: we can use a UINavigationController to share the same screen

Multiple MVCs: UINavigationController



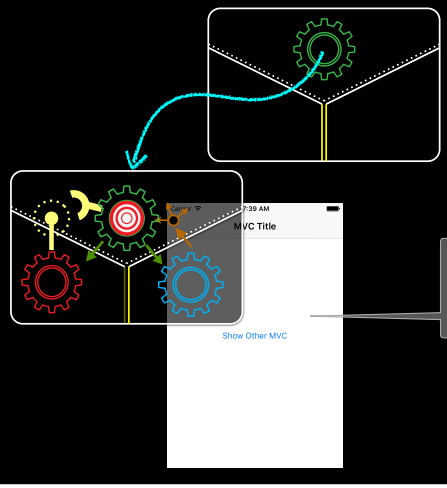
UINavigationController is a controller whose view looks like this

Multiple MVCs: UINavigationController



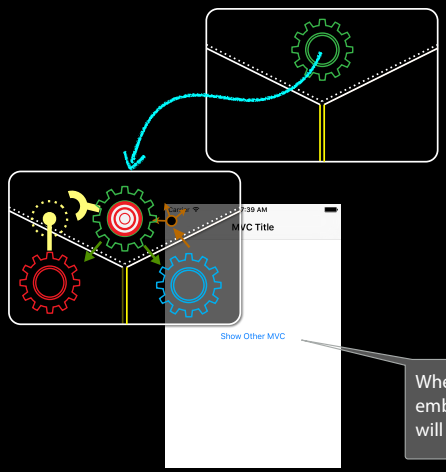
UINavigationController is special as it allows to se its rootViewController outlet property to another MVC

Multiple MVCs: UINavigationController



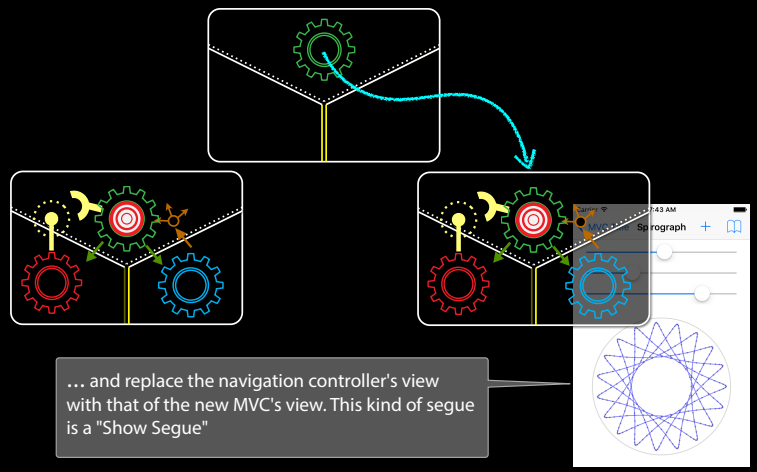
UINavigationController is special as it embeds other MVC's view inside its own view

Multiple MVCs: UINavigationController



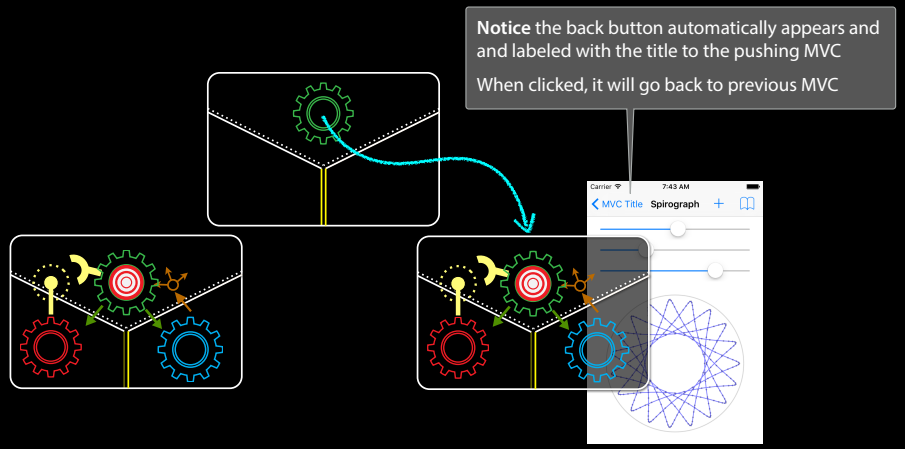
When a UI widget in the embedded view is triggered, it will segue to create a new MVC…

Multiple MVCs: UINavigationController

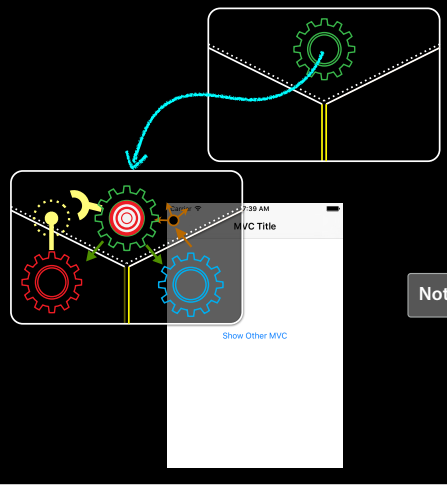


… and replace the navigation controller's view with that of the new MVC's view. This kind of segue is a "Show Segue"

Multiple MVCs: UINavigationController



Multiple MVCs: UINavigationController



Note: the other MVC is completely gone

Accessing the sub-MVCs

- **You can get the sub-MVCs via the viewControllers property**

```
var viewControllers: [UIViewController] { get set } // optional if not embedded
```

  - for a tab bar, they are in order, left to right, in the array
  - for a split view, [0] is the master and [1] is the detail
  - for a navigation controller, [0] is the root and the rest are in order on the stack
  - even though this is settable, usually setting happens via storyboard, segues, or other
  - for example, navigation controller's push and pop methods
- **How do you get ahold of the UITabBarController, UISplitViewController or UINavigationController?**
  - Every UIViewController knows the Split View, Tab Bar or Navigation Controller it is currently in
  - These are UIViewController properties …

```
var tabBarController: UITabBarController? { get }
var splitViewController: UISplitViewController? { get }
var navigationController: UINavigationController? { get }
```

  - So, for example, to get the detail of the split view controller you are in ...

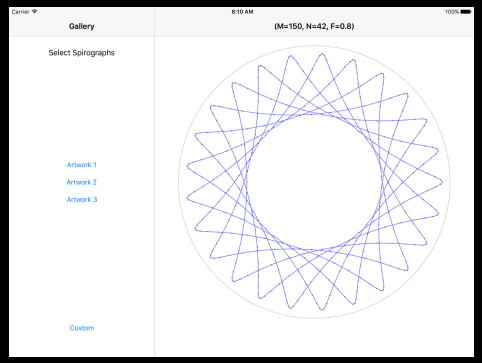
```
if let detailVC: UIViewController = splitViewController?.viewControllers[1] { ... }
```

Segues

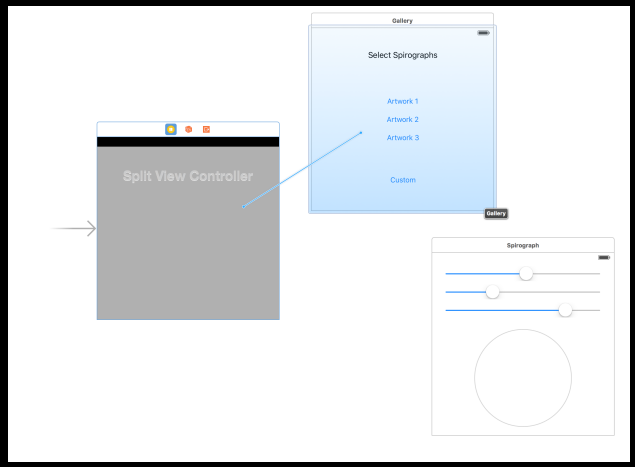
Connecting Multiple MVCs

- **How do we wire up multiple MVCs?**
  - Let's consider a spirograph artwork app with a menu to select spirograph artworks
  - How do we hook them up to be the two sides of a split view?

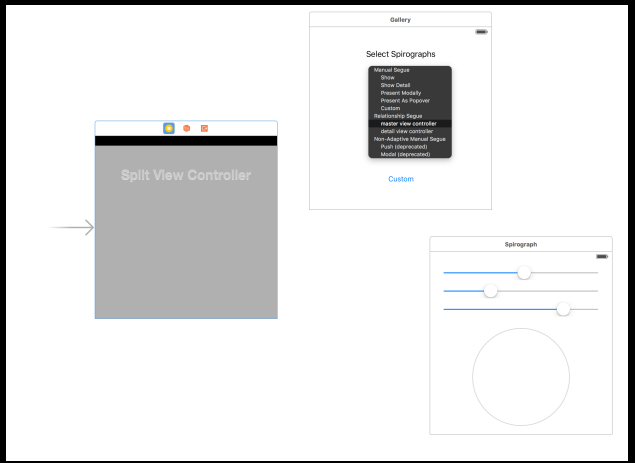
- **Solution:**
  - drag out a split view controller from the object library (and delete all the extra view controllers it brings with it)
  - CTRL-drag from the split view controller to the master and detail MVCs



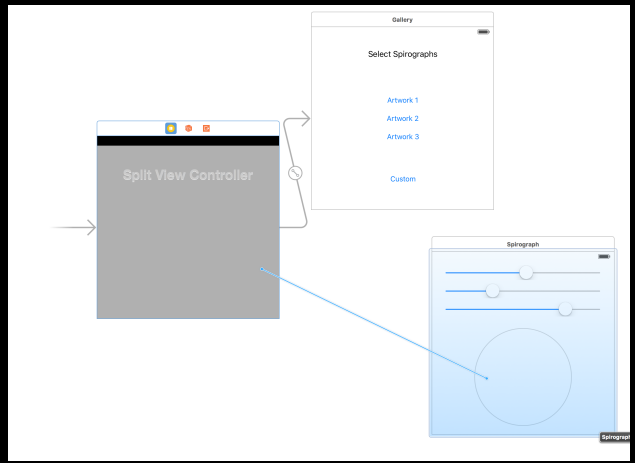
Connecting Multiple MVCs



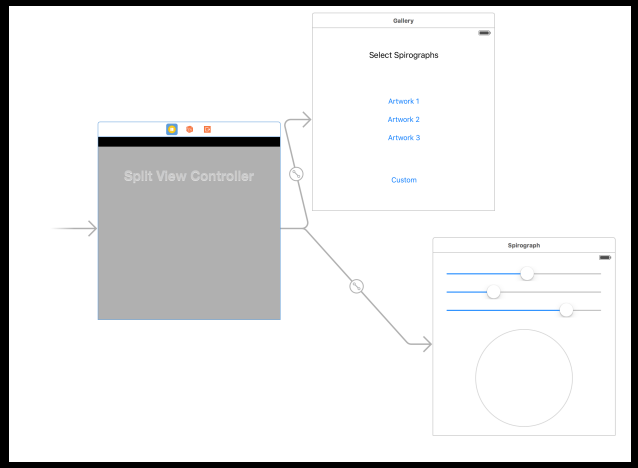
Connecting Multiple MVCs



Connecting Multiple MVCs

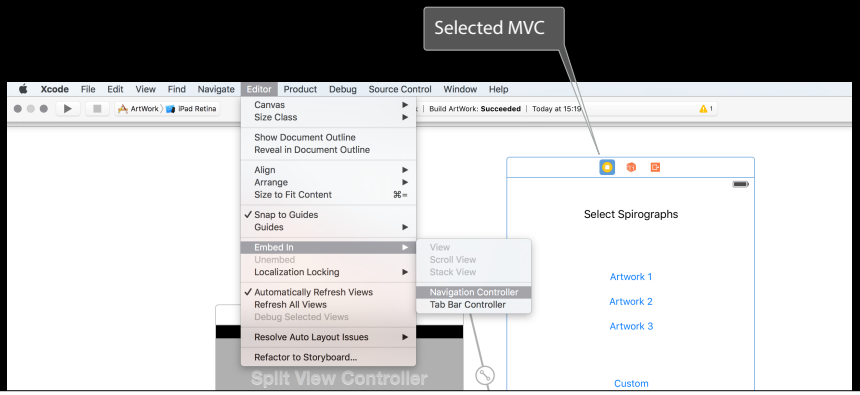


Connecting Multiple MVCs



Connecting Multiple MVCs

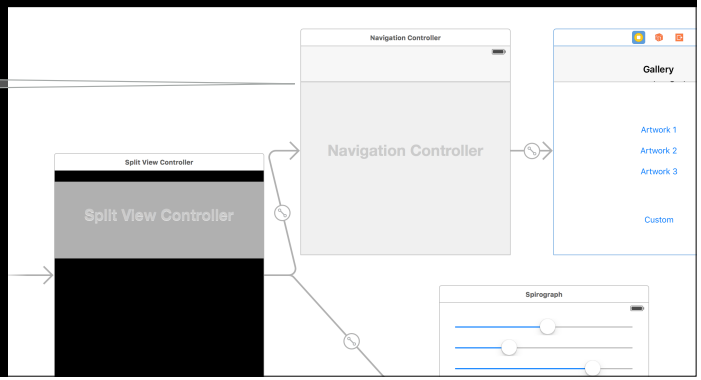
- **Split view can only do its thing properly on iPad**
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In



Connecting Multiple MVCs

- **Split view can only do its thing properly on iPad**
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In

MVC is now part of the View of the navigation controller (it's the rootViewController)

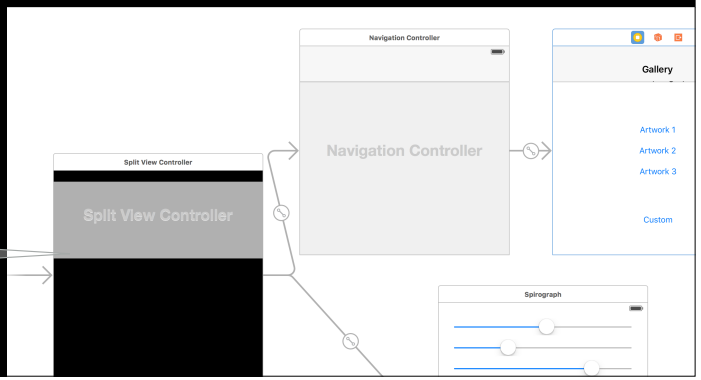


Connecting Multiple MVCs

- **Split view can only do its thing properly on iPad**
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In

UINavigationController is part of the View of this UISplitViewController

(it's the Master, viewControllers[0])

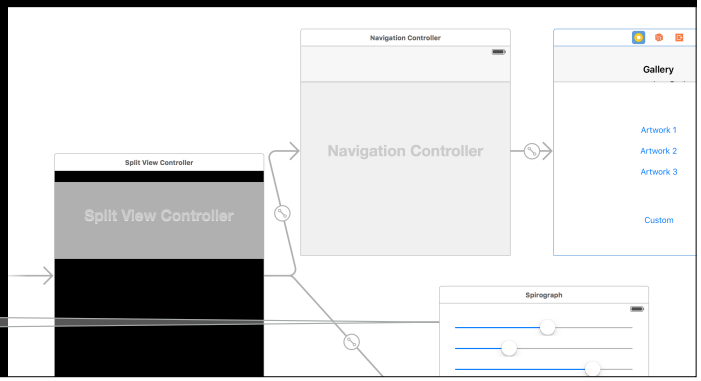


Connecting Multiple MVCs

- **Split view can only do its thing properly on iPad**
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In

You can embed this MVC in a navigation controller too (e.g. to provide a title)

The Detail of the UISplitViewController would now be a UINavigationController (so you'd have to get the UINavigationController's rootViewController if you wanted to talk to the MVC inside)



Segues

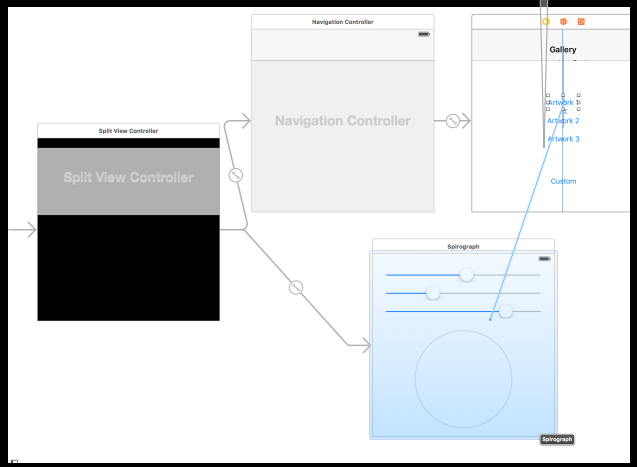
- **We've built up our Controllers of Controllers, now what?**
  - Now we need to make it so that one MVC can cause another to appear
  - We call that a "segue"
- **Kinds of segues (they will adapt to their environment)**
  - Show Segue (will push in a Navigation Controller, else Modal)
  - Show Detail Segue (will show in Detail of a Split View or will push in a Navigation Controller)
  - Modal Segue (take over the entire screen while the MVC is up)
  - Popover Segue (make the MVC appear in a little popover window)
- **Segues always create a new instance of an MVC**
  - This is important to understand
  - The Detail of a Split View will get replaced with a new instance of that MVC
  - When you segue in a Navigation Controller it will not segue to some old instance, it'll be new

Segues

- **How do we make these segues happen?**
  - Ctrl-drag in a storyboard from an instigator (like a button) to the MVC to segue to
  - Can be done in code as well

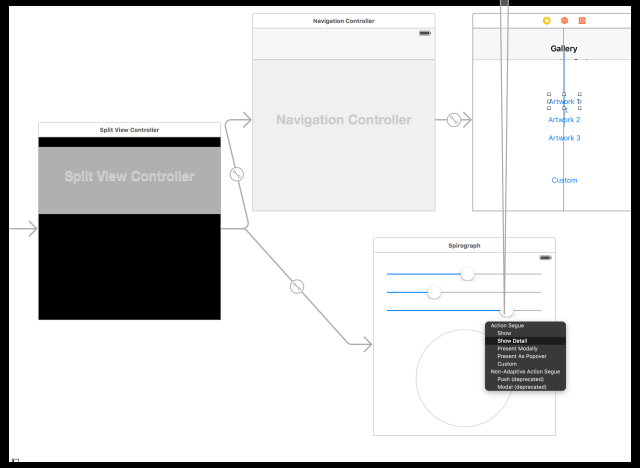
Segues

Ctrl-drag from the button that causes the spirograph to appear to the detail MVC



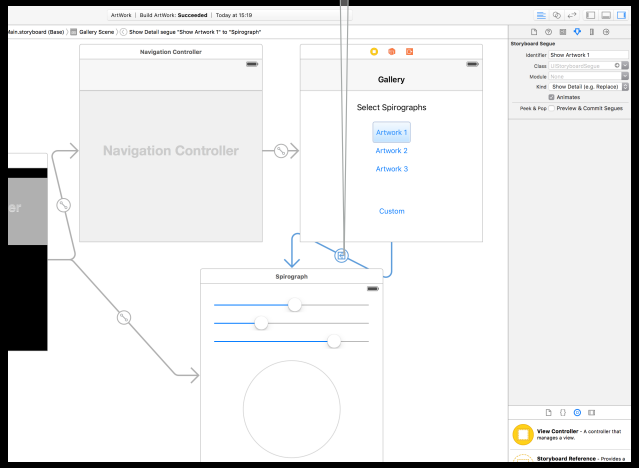
Segues

Select the kind of segue you want. Usually Show or Show Detail.



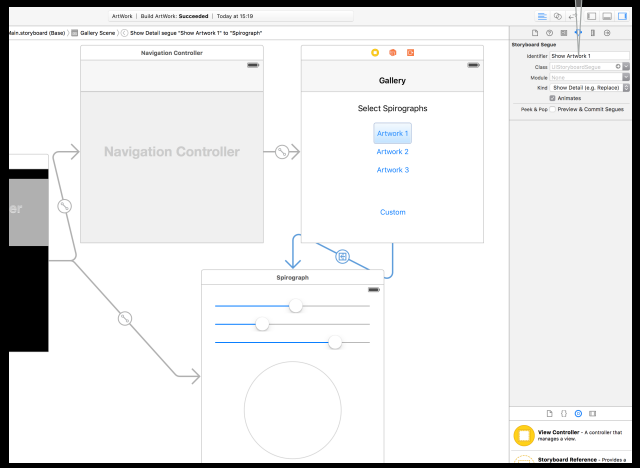
Segues

Now click on the segue and open the Attributes Inspector



Segues

Give the segue a unique identifier here. It should describe what the segue does



Segues

- **What's that identifier all about?**
  - You would need it to invoke this segue from code using this UIViewController method

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

  - (but we almost never do this because we set usually ctrl-drag from the instigator)
  - The sender can be whatever you want (you'll see where it shows up in a moment)
  - You can ctrl-drag from the Controller itself to another Controller if you're segueing via code (because in that case, you'll be specifying the sender above)
- **More important use of the identifier: preparing for a segue**
  - When a segue happens, the View Controller containing the instigator gets a chance to prepare the destination View Controller to be segued to
  - Usually this means setting up the segued-to MVC's Model and display characteristics
  - Remember that the MVC segued to is always a fresh instance (never a reused one)

Preparing for a Segue

- **The method that is called in the instigator's Controller**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

Preparing for a Segue

- **The method that is called in the instigator's Controller**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- **The segue passed in contains important information about this segue:**
  - the identifier from the storyboard
  - the Controller of the MVC you are segueing to (which was just created for you)

Preparing for a Segue

- **The method that is called in the instigator's Controller**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- **The sender is either**
  - the instigating object from a storyboard (e.g. a UIButton) or
  - the sender you provided if you invoked the segue manually in code

Preparing for a Segue

- **The method that is called in the instigator's Controller**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- **Here is the identifier from the storyboard (it can be nil, so be sure to check for that case)**
  - Your Controller might support preparing for lots of different segues from different instigators
  - So this identifier is how you'll know which one you're preparing for

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- For this example, we'll assume we entered "Show Artwork 1" in the Attributes Inspector when we had the segue selected in the storyboard

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- Here we are looking at the Controller of the MVC we're segueing to
- It is AnyObject, so we must cast it to the Controller we (should) know it to be

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- This is where the actual preparation of the segued-to MVC occurs
- Hopefully the MVC has a clear public API that it wants you to use to prepare it
- Once the MVC is prepared, it should run on its own power (only using delegation to talk back)

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Show Artwork 1":
            if let vc = segue.destinationViewController as? SpirographVC {
                vc.spirographModel = ...
                vc.callMethodToSetItUp(...)
            }
        default: break
        }
    }
}
```

- It is crucial to understand that this preparation is happening BEFORE outlets get set!
- It is a very common bug to prepare an MVC thinking its outlets are set.

Preventing Segues

- **You can prevent a segue from happening too**
  - Just implement this in your UIViewController …

```
func shouldPerformSegue(withIdentifier: String?, sender: Any?) -> Bool
```

  - The identifier is the one in the storyboard
  - The sender is the instigating object (e.g. the button that is causing the segue)

Demo

Spirograph Artwork

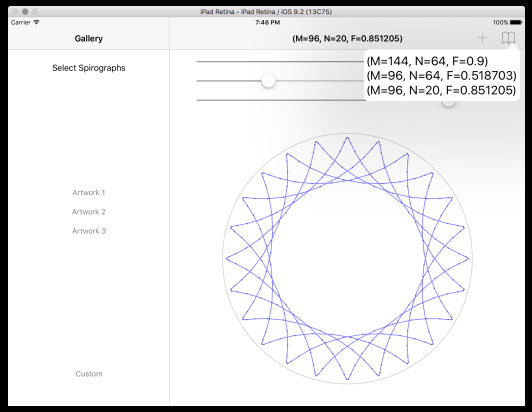
Demo

- **Create Spirograph Gallery MVC**
- **Show spirographs by segueing to the Spirograph MVC**
- **Embed MVC in navigation controllers inside a split view controller**
- **Run app on both iPad and iPhone**

Popovers

Popover

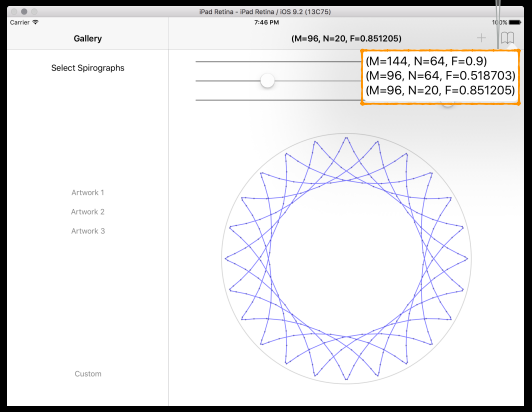
- Popovers pop an entire MVC over the rest of the screen



Popover

- Popovers pop an entire MVC over the rest of the screen

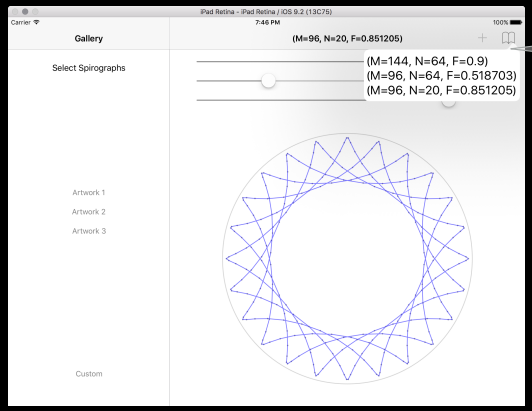
A text view MVC



Popover

- Popovers pop an entire MVC over the rest of the screen

Popover's arrow pointing to what caused it to appear

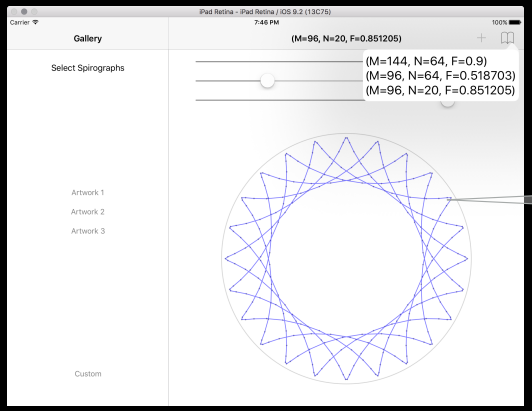


Popover

- Popovers pop an entire MVC over the rest of the screen

The gryed out area here is inactive.

Touching in it will dismiss the popover



Popover

- **Popovers are not quite the same as these other combiners**
  - Tab Bar, Split View and Navigation Controllers are UIViewControllers, popovers are not
- **Seguing to a popover works almost exactly the same though**
  - You still ctrl-drag, you still have an identifier, you still get to prepare
- **Things to note when preparing for a popover segue**
  - All segues are managed via a UIPresentationController (but we're not going to cover that)
  - But we are going to talk about a popover's UIPresentationController
  - It can tell you what caused the popover to appear (a bar button item or just a rectangle)
  - And it can let you control how the popover is presented
  - For example, you can control what direction the popover's arrow is allowed to point
  - Or you can control how a popover adapts to different sizes classes (e.g. iPad vs iPhone)

Popover Prepare

- **Here's a prepareForSegue that prepares for a Popover segue**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Do Something in a Popover Segue":
            if let vc = segue.destinationViewController as? MyController {
                if let ppc = vc.popoverPresentationController {
                    ppc.permittedArrowDirections = UIPopoverArrowDirection.Any
                    ppc.delegate = self
                }
                // more preparation here
            }
        default: break
        }
    }
}
```

  - One thing is different ; we are retrieving the popover's presentation controller

Popover Prepare

- **Here's a prepareForSegue that prepares for a Popover segue**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Do Something in a Popover Segue":
            if let vc = segue.destinationViewController as? MyController {
                if let ppc = vc.popoverPresentationController {
                    ppc.permittedArrowDirections = UIPopoverArrowDirection.Any
                    ppc.delegate = self
                }
                // more preparation here
            }
        default: break
        }
    }
}
```

  - We can use it to set some properties that will control how the popover pops up

Popover Prepare

- **Here's a prepareForSegue that prepares for a Popover segue**

```
func prepare(for: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "Do Something in a Popover Segue":
            if let vc = segue.destinationViewController as? MyController {
                if let ppc = vc.popoverPresentationController {
                    ppc.permittedArrowDirections = UIPopoverArrowDirection.Any
                    ppc.delegate = self
                }
                // more preparation here
            }
        default: break
        }
    }
}
```

  - And we can control the presentation by setting ourself (the Controller) as the delegate

Popover Presentation Controller

- **What can we control as the presentation controller's delegate?**
  - One very interesting thing is how a popover "adapts" to different sizes
  - By default, it will present on compact sizes Modally (i.e. take over the whole screen)
  - But the delegate can control this "adaptation" behavior, either by preventing it …

```
func adaptivePresentationStyle(for: UIPresentationController)
    -> UIModalPresentationStyle
{
    return UIModalPresentationStyle.none // don't adapt (default is .FullScreen)
}
```

… or by allowing the full screen presentation to happen, but modifying the MVC that is put up …

```
func presentationController(UIPresentationController,
    viewControllerForAdaptivePresentationStyle: UIModalPresentatinoStyle)
    -> UIViewController?
{
   // return a UIViewController to use (e.g. wrap a Navigation Controller around your MVC)
}
```

Popover Size

- **Important Popover Issue: Size**
  - A popover will be made pretty large unless someone tells it otherwise.
  - The MVC being presented knows best what it's "preferred" size inside a popover would be.
  - It expresses that via this property in itself (i.e. in the Controller of the MVC being presented) …

```
var preferredContentSize: CGSize
```

  - The MVC is not guaranteed to be that size, but the system will try its best

Demo

Spirograph Artwork

Demo

- **Add popover to show list of favourite spirographs**

View Controller Lifecycle

View Controller Lifecycle

- **View Controllers have a Lifecycle**
  - A sequence of messages is sent to a View Controller as it progresses through its "lifetime"
  - Why does this matter?
  - You very commonly override these methods to do certain work
- **The start of the lifecycle …**
  - Creation
  - MVCs are most often instantiated out of a storyboard
  - There are ways to do it in code (rare) as well
- **What then?**
  - Preparation if being segued to
  - Outlet setting
  - Appearing and disappearing
  - Geometry changes
  - Low-memory situations

View Controller Lifecycle

- **After instantiation and outlet-setting, viewDidLoad is called**
  - This is an exceptionally good place to put a lot of setup code
  - It's better than an init because your outlets are all set up by the time this is called.

```
override func viewDidLoad() {
    super.viewDidLoad() // always let super have a chance in lifecycle methods
    // do some setup of my MVC
}
```

  - One thing you may well want to do here is update your UI from your Model
  - Because now you know all of your outlets are set

  - But be careful because the geometry of your view (its bounds) is not set yet!
  - At this point, you can't be sure you're on an iPhone 5-sized screen or an iPad
  - So do not initialize things that are geometry-dependent here

View Controller Lifecycle

- **Just before your view appears on screen, you get notified**

```
func viewWillAppear(animated: Bool) // animated is whether you are appearing over time
```

  - Your view will only get "loaded" once, but it might appear and disappear a lot.
  - So don't put something in this method that really wants to be in viewDidLoad.
  - Otherwise, you might be doing something over and over unnecessarily.
  - Do something here if things your display is changing while your MVC is off-screen.
  - You could use this to optimize performance by waiting until this method is called (as opposed to viewDidLoad) to kick off an expensive operation (probably in another thread).
  - Your view's geometry is set here, but there are other places to react to geometry.
- **There is a "did" version of this as well**

```
func viewDidAppear(animated: Bool)
```

View Controller Lifecycle

- **And you get notified when you will disappear off screen too**
  - This is where you put "remember what's going on" and cleanup code.

```
override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated) //call super in all the viewWill/Did... methods
    // do some clean up now that we've been removed from the screen
    // but be careful not to do anything time-consuming here, or app will be sluggish
    // maybe even kick off a thread to do stuff here
}
```

- **There is a "did" version of this too**

```
override func viewWillDisappear(animated: Bool)
```

View Controller Lifecycle

- **Change of Geometry**
  - Most of the time this will be automatically handled with Autolayout
  - But you can get involved in geometry changes directly with these methods …

```
func viewWillLayoutSubviews()
func viewDidLayoutSubviews()
```

  - They are called any time a view's frame changed and its subviews were thus re-layed out.
  - For example, autorotation
- **You can reset the frames of your subviews here or set other geometry-related properties**
- **Between "will" and "did", autolayout will happen**
- **These methods might be called more often than expected (e.g. for pre- and post- animation arrangement, etc.)**
- **So don't do anything in here that can't properly (and efficiently) be done repeatedly**

View Controller Lifecycle

- **Interface Orientation and Autorotation**
  - Usually, the UI changes shape when the user rotates the device between portrait/landscape
  - You can control which orientations your app supports in the Settings of your project
  - Almost always, your UI just responds naturally to rotation with autolayout
  - But if you, for example, want to participate in the rotation animation, you can use this method …

```
func viewWillTransition(to: CGSize, with: UIViewControllerTransitionCoordinator)
```

  - The coordinator provides a method to animate alongside the rotation animation

View Controller Lifecycle

- **In low-memory situations, didReceiveMemoryWarning gets called …**
  - This rarely happens, but well-designed code with big-ticket memory uses might anticipate it
  - **Examples:** images or sounds
  - Anything "big" that is not currently in use and can be recreated relatively easily should probably be released (by setting any pointers to it to nil)

View Controller Lifecycle

- **awakeFromNib**
  - This method is sent to all objects that come out of a storyboard (including your Controller)
  - Happens before outlets are set! (i.e. before the MVC is "loaded")
  - Put code somewhere else if at all possible (e.g. viewDidLoad or viewWillAppear).
- **Summary**
  - Instantiated (from storyboard usually)
  - awakeFromNib
  - segue preparation happens
  - outlets get set
  - viewDidLoad
  - These pairs will be called each time your Controller's view goes on/off screen … viewWillAppear & viewDidAppear, viewWillDisappear & viewDidDisappear
  - These "geometry changed" methods might be called at any time after viewDidLoad ... viewWillLayoutSubviews (then autolayout happens, then…) viewDidLayoutSubviews
  - If memory gets low, you might get ... didReceiveMemoryWarning

Demo

Spirograph Artwork

Demo

- **View Controller Lifecycle**