Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- View Controller Lifecycle
  - Tracking what is going on with your View Controller
  - Brief Demo
- Autolayout
  - Review
  - Size Classes
  - Demos

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 (as you've seen).
  - There are ways to do it in code (rare) as well which we may cover later in the quarter.
- 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 or ???.
  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 (again, we'll cover threads later)
}
```

- There is a "did" version of this too

```
func viewDidDisappear(animated: Bool)
```

View Controller Lifecycle

- Geometry changed?

  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 (more on this in a moment).

  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 you'd imagine
  (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

- 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 viewWillTransitionToSize(
    size: CGSize,
    withTransitionCoordinator: UIViewControllerTransitionCoordinator
)
```

  The `coordinator` provides a method to animate alongside the rotation animation
  We are not going to be talking about animation, though, for a couple of weeks
  So this is just something to put in the back of your mind (i.e. that it exists) for now

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 and 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`).

View Controller Lifecycle

- 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` and `viewDidAppear`

    `viewWillDisappear` and `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`

View Controller Lifecycle

- Demo

  Let's plop some `println` statements into the View Controller Lifecycle methods in Psychologist
  Then we can watch as Psychologist and Happiness MVCs go through their lifecycle

Autolayout

- You've seen a lot of Autolayout already
  - Using the dashed blue lines to try to tell Xcode what you intend
  - Ctrl-Dragging between views to create relationships (spacing, etc.)
  - The "Pin" and "Arrange" popovers in the lower right of the storyboard
  - Reset to Suggested Constraints (if the blue lines were enough to unambiguously set constraints)
  - Document Outline (see all constraints, resolve misplacements and even conflicts)
  - Size Inspector (look at (and edit!) the details of the constraints on the selected view)
  - Clicking on a constraint to select it then bring up Attributes Inspector (to edit its details)
- Mastering Autolayout requires experience
  - You just have to do it to learn it
- Autolayout can be done from code too
  - Though you're probably better off doing it in the storyboard wherever possible
  - The demo today will show a simple case of doing Autolayout from code

Autolayout

- ## What about rotation?
  Sometimes rotating changes the geometry so drastically that autolayout is not enough
  You actually need to reposition the views to make them fit properly
- ## Calculator
  For example, what if we had 20 buttons in a Calculator?
  It might be better in Landscape to have the buttons 5 across and 4 down
  Versus in Portrait have them 4 across and 5 down
- ## View Controllers might want this in other situations too
  For example, your MVC is the master of a side-by-side split view
  In that case, you'd want to draw just like a Portrait iPhone does
- ## The solution? Size Classes
  Your View Controller always exists in a certain "size class" environment for width and height
  Currently this is either Compact or Regular (i.e. not compact)

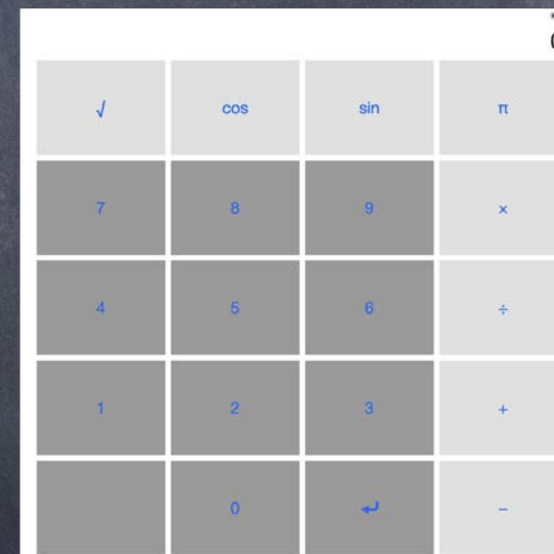
Autolayout

- iPhone 6+
  - The iPhone 6+ in Portrait orientation is Compact in width and Regular in height
  - In Landscape, it is Compact in height and Regular in width
- iPhone
  - Other iPhones in Portrait are also Compact in width and Regular in height
  - But in Landscape, non-6+ iPhones are treated as Compact in both dimensions
- iPad
  - Always Regular in both dimensions
  - An MVC that is the master in a side-by-side split view will be Compact width, Regular height
- Extensible
  - This whole concept is extensible to any "MVC's inside other MVC's" situation (not just split view)
  - An MVC can find out its size class environment via this method in `UIViewController` …

```
let mySizeClass: UIUserInterfaceSizeClass = self.traitCollection.horizontalSizeClass
```

The return value is an enum `.Compact` or `.Regular` (or `.Unspecified`).

Size Classes

	Compact	Regular
Compact		
Regular		

Vertical

Horizontal

Size Classes

	Compact	Any	Regular
Compact			
Any			
Regular			

Horizontal

Vertical

Demos

- ShowSizeClass
  A trivial app to take a look at how we react to size class differences in our storyboard
- Autolayout
  Let's pull it all together by building a UI that has some Autolayout challenges
  Including needing to do something different in different size classes
- A Challenge to You!
  After watching today, see if you can make your Calculator react to size class changes
  For example, change the number of rows and columns in different size class situations
  Or even show more operations in one size class or another (like Apple's Calculator app does)