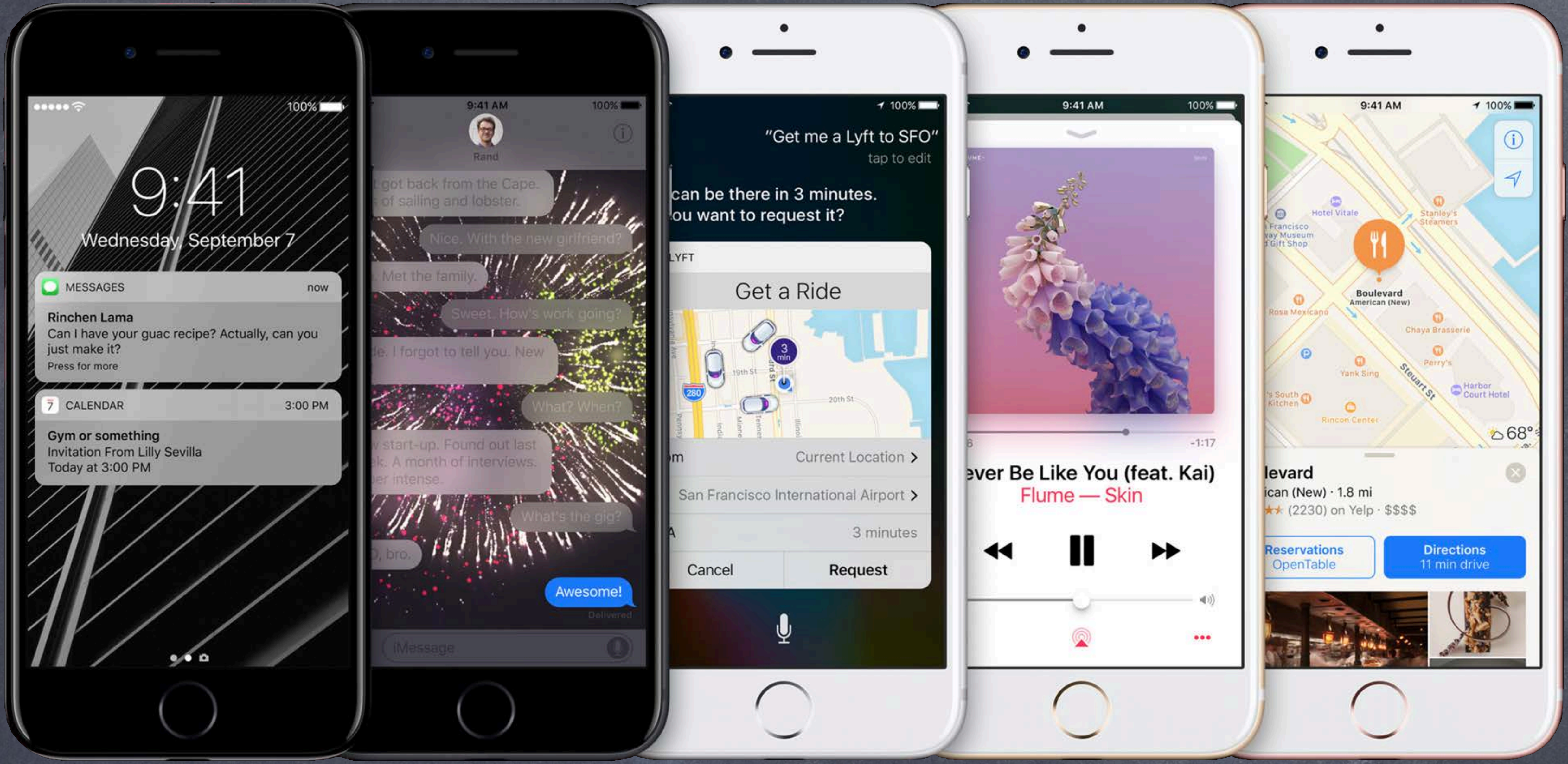

Stanford CS193p

Developing Applications for iOS
Winter 2017

Today

- Multiple MVCs
  - Demo: Emotions in FaceIt
- View Controller Lifecycle
  - Tracking what happens to an MVC over time
  - Demo: VCL in FaceIt
- Time Permitting
  - Memory Management (especially vis-a-vis closures)

Demo

- Emotions in FaceIt
  - This is all best understood via demonstration
  - We will create a new Emotions MVC
  - The Emotions will be displayed segueing to the Face MVC
  - We'll put the MVCs into navigation controllers inside split view controllers
  - That way, it will work on both iPad and iPhone devices

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 all 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 viewWillTransition(
    to size: CGSize,
    with coordinator: 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

Coming Up

- Now, a Demo …

  Let's plop some `print` statements into the View Controller Lifecycle methods in FaceIt

  Then we can watch as Face and Emotions MVCs go through their lifecycle

- Time Permitting

  Memory Management (especially vis-a-vis closures)

- Wednesday

  Extensions, Protocols, Delegation
  `UIScrollView`

- Friday

  Instruments (Performance Analysis Tool)

- Next Week

  Multithreading

  Table View

Memory Management

- **Automatic Reference Counting**

  Reference types (`classes`) are stored in the heap.

  How does the system know when to reclaim the memory for these from the heap?

  It "counts references" to each of them and when there are zero references, they get tossed.

  This is done automatically.

  It is known as "Automatic Reference Counting" and it is NOT garbage collection.

- **Influencing ARC**

  You can influence ARC by how you declare a reference-type var with these keywords …

  `strong`

  `weak`

  `unowned`

Memory Management

- strong

  strong is "normal" reference counting
  As long as anyone, anywhere has a strong pointer to an instance, it will stay in the heap

- weak

  weak means "if no one else is interested in this, then neither am I, set me to nil in that case"
  Because it has to be nil-able, weak only applies to Optional pointers to reference types
  A weak pointer will NEVER keep an object in the heap
  Great example: outlets (strongly held by the view hierarchy, so outlets can be weak)

- unowned

  unowned means "don't reference count this; crash if I'm wrong"
  This is very rarely used
  Usually only to break memory cycles between objects (more on that in a moment)

Closures

- Capturing

  Closures are stored in the heap as well (i.e. they are reference types).
  They can be put in Arrays, Dictionarys, etc. They are a first-class type in Swift.

  What is more, they "capture" variables they use from the surrounding code into the heap too.
  Those captured variables need to stay in the heap as long as the closure stays in the heap.
  This can create a memory cycle …

Closures

- Example

  Imagine we added public API to allow a unaryOperation to be added to the CalculatorBrain

  func addUnaryOperation(symbol: String, operation: (Double) -> Double)

  This method would do nothing more than add a unaryOperation to our Dictionary of enum

  Now let's imagine a View Controller was to add the operation "green square root".

  This operation will do square root, but it will also turn the display green.

```
addUnaryOperation("✅", operation: { (x: Double) -> Double in
    display.textColor = UIColor.green
    return sqrt(x)
})
```

Closures

- Example

  Imagine we added public API to allow a `unaryOperation` to be added to the `CalculatorBrain`

  `func addUnaryOperation(symbol: String, operation: (Double) -> Double)`

  This method would do nothing more than add a `unaryOperation` to our `Dictionary` of `enum`

  Now let's imagine a View Controller was to add the operation "green square root".

  This operation will do square root, but it will also turn the display green.

```
addUnaryOperation("✅") { (x: Double) -> Double in
    display.textColor = UIColor.green
    return sqrt(x)
}
```

Closures

- Example

  Imagine we added public API to allow a unaryOperation to be added to the CalculatorBrain

  func addUnaryOperation(symbol: String, operation: (Double) -> Double)

  This method would do nothing more than add a unaryOperation to our Dictionary of enum

  Now let's imagine a View Controller was to add the operation "green square root".

  This operation will do square root, but it will also turn the display green.

```
addUnaryOperation("✅") { (x: Double) -> Double in
    display.textColor = UIColor.green
    return sqrt(x)
}
```

Closures

- Example

Imagine we added public API to allow a `unaryOperation` to be added to the `CalculatorBrain`

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

This method would do nothing more than add a `unaryOperation` to our `Dictionary` of `enum`

Now let's imagine a View Controller was to add the operation "green square root".
This operation will do square root, but it will also turn the display green.

```
addUnaryOperation("✅") {
    display.textColor = UIColor.green
    return sqrt($0)
}
```

But this will not compile.

Closures

- Example

  Imagine we added public API to allow a unaryOperation to be added to the CalculatorBrain

  func addUnaryOperation(symbol: String, operation: (Double) -> Double)

  This method would do nothing more than add a unaryOperation to our Dictionary of enum

  Now let's imagine a View Controller was to add the operation "green square root".

  This operation will do square root, but it will also turn the display green.

```
addUnaryOperation("✅") {
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

  Swift forces you to put self. here to remind you that self will get captured!

  The Model and the Controller now point to each other through the closure.

  And thus neither can ever leave the heap. This is called a memory cycle.

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“✅”) {
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“✅”) {  [ <special variable declarations> ] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“✅”) {  [ me = self ] in
    me.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“✅”) {  [ unowned me = self ] in
    me.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("✅") {  [ unowned self = self ] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“✅”) {  [ unowned self ] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("✅") {  [ weak self ] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("✅") {  [ weak self ] in
    self?.display.textColor = UIColor.green
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("✅") {  [ weak weakSelf = self ] in
    weakSelf?.display.textColor = UIColor.green
    return sqrt($0)
}
```

Demo

- Green Square Root

  Let's do what we just talked about and see it in action in our Calculator