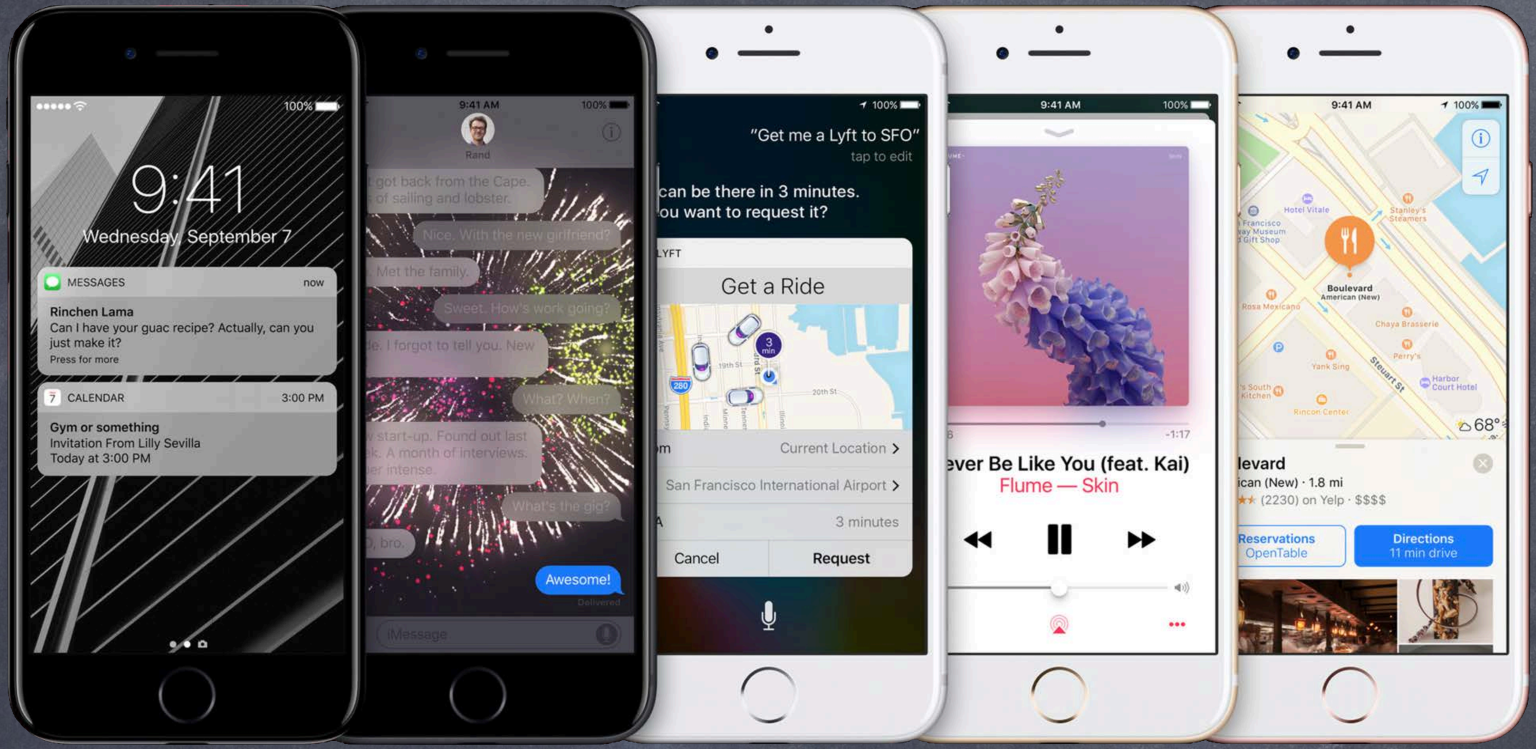

Stanford CS193p

Developing Applications for iOS
Winter 2017

Today

- Demo
  - Cassini Continued
  - Multiple MVC app which shows images related to Cassini space probe
- Multithreading
  - Keeping the UI responsive
  - Multithreaded Cassini Demo
  - Keeping the UI responsive while fetching Cassini images
  - Showing a “spinner” when the app is busy fetching something in the background
  - Bonus: How to cause split view to come up showing the master instead of the detail
- Text Field
  - Like `UILabel`, but editable text

Demo

- Cassini Continued
  - Multiple MVC to view some NASA images

Multithreading

- Queues

  Multithreading is mostly about "queues" in iOS.
  Functions (usually closures) are simply lined up in a queue (like at the movies!).
  Then those functions are pulled off the queue and executed on an associated thread(s).
  Queues can be "serial" (one closure a time) or "concurrent" (multiple threads servicing it).

- Main Queue

  There is a very special serial queue called the "main queue."
  All UI activity MUST occur on this queue and this queue only.
  And, conversely, non-UI activity that is at all time consuming must NOT occur on that queue.
  We do this because we want our UI to be highly responsive!
  And also because we want things that happen in the UI to happen predictably (serially).
  Functions are pulled off and worked on in the main queue only when it is "quiet".

- Global Queues

  For non-main-queue work, you're usually going to use a shared, global, concurrent queue.

Multithreading

- Getting a queue

  Getting the main queue (where all UI activity must occur).

  ```
  let mainQueue = DispatchQueue.main
  ```

  Getting a global, shared, concurrent "background" queue.

  This is almost always what you will use to get activity off the main queue.

  ```
  let backgroundQueue = DispatchQueue.global(qos: DispatchQoS)
  DispatchQoS.userInteractive // high priority, only do something short and quick
  DispatchQoS.userInitiated   // high priority, but might take a little bit of time
  DispatchQoS.background      // not directly initiated by user, so can run as slow as needed
  DispatchQoS.utility         // long-running background processes, low priority
  ```

Multithreading

- Putting a block of code on the queue

  Multithreading is simply the process of putting closures into these queues.
  There are two primary ways of putting a closure onto a queue.

  You can just plop a closure onto a queue and keep running on the current queue …

  ```
  queue.async { . . . }
  ```

  … or you can block the current queue waiting until the closure finishes on that other queue …

  ```
  queue.sync { . . . }
  ```

  We almost always do the former.

Multithreading

- Getting a non-global queue

  Very rarely you might need a queue other than main or global.

  Your own serial queue (use this only if you have multiple, serially dependent activities) …

  ```
  let serialQueue = DispatchQueue(label: “MySerialQ”)
  ```

  Your own concurrent queue (rare that you would do this versus global queues) …

  ```
  let concurrentQueue = DispatchQueue(label: “MyConcurrentQ”, attributes: .concurrent)
  ```

Multithreading

- We are only seeing the tip of the iceberg
  - There is a lot more to GCD (Grand Central Dispatch)
  - You can do locking, protect critical sections, readers and writers, synchronous dispatch, etc.
  - Check out the documentation if you are interested
- There is also another API to all of this
  - `OperationQueue` and `Operation`
  - Usually we use the `DispatchQueue` API, however.
  - This is because the "nesting" of dispatching reads very well in the code
  - But the `Operation` API is also quite useful (especially for more complicated multithreading)

Multithreading

- Multithreaded iOS API

  Quite a few places in iOS will do what they do off the main queue
  They might even afford you the opportunity to do something off the main queue
  iOS might ask you for a function (a closure, usually) that executes off the main thread
  Don't forget that if you want to do UI stuff there, you must dispatch back to the main queue!

Multithreading

- Example of a multithreaded iOS API

  This API lets you fetch the contents of an http URL into a `Data` off the main queue!

```
let session = URLSession(configuration: .default)
if let url = URL(string: “http://stanford.edu/...”) {
    let task = session.dataTask(with: url) { (data: Data?, response, error) in



    }
    task.resume()
}
```

Multithreading

- Example of a multithreaded iOS API

  This API lets you fetch the contents of an http URL into a Data off the main queue!

```
let session = URLSession(configuration: .default)
if let url = URL(string: “http://stanford.edu/...”) {
    let task = session.dataTask(with: url) { (data: Data?, response, error) in
        // I want to do UI things here
        // with the data of the download
        // can I?
    }
    task.resume()
}
```

  NO. That's because that code will be run off the main queue.
  How do we deal with this?
  One way is to use a variant of this API that lets you specify the queue to run on (main queue).
  Here's another way using GCD …

Multithreading

- Example of a multithreaded iOS API

  This API lets you fetch the contents of an http URL into a Data off the main queue!

```
let session = URLSession(configuration: .default)
if let url = URL(string: “http://stanford.edu/...”) {
    let task = session.dataTask(with: url) { (data: Data?, response, error) in
        DispatchQueue.main.async {
            // do UI stuff here
        }
    }
    task.resume()
}
```

  Now we can legally do UI stuff in there.
  That’s because the UI code you want to do has been dispatched back to the main queue.

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: “http://stanford.edu/...”) {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print(“did some stuff with the data, but UI part hasn’t happened yet”)
       }
g:     task.resume()
   }
h: print(“done firing off the request for the url’s contents”)
```

  Line **a** is obviously first.

Multithreading

- Timing

Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: "http://stanford.edu/...") {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print("did some stuff with the data, but UI part hasn't happened yet")
       }
g:     task.resume()
   }
h: print("done firing off the request for the url's contents")
```

Line b is next.
It returns immediately. It does nothing but create a dataTask and assign it to task.
Obviously its closure argument has yet to execute (it needs the data to be retrieved first).

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: "http://stanford.edu/...") {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print("did some stuff with the data, but UI part hasn't happened yet")
       }
g:     task.resume()
   }
h: print("done firing off the request for the url's contents")
```

  Line g happens immediately after line b. It also returns immediately.
  All it does is fire off the url fetch (to get the data) on some other (unknown) queue.
  The code on lines c, d, e and f will eventually execute on some other (unknown) queue.

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: “http://stanford.edu/...”) {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print(“did some stuff with the data, but UI part hasn’t happened yet”)
       }
g:     task.resume()
   }
h: print(“done firing off the request for the url’s contents”)
```

  Line h happens immediately after line g.

  The url fetching task has now begun on some other queue (executing on some other thread).

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: "http://stanford.edu/...") {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print("did some stuff with the data, but UI part hasn't happened yet")
       }
g:     task.resume()
   }
h: print("done firing off the request for the url's contents")
```

  The first four lines of code (a, b, g, h) all ran back-to-back with no delay.
  But line c will not get executed until sometime later (because it was waiting for the data).
  It could be moments after line g or it could be minutes (e.g., if over cellular).

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: “http://stanford.edu/...”) {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print(“did some stuff with the data, but UI part hasn’t happened yet”)
       }
g:     task.resume()
   }
h: print(“done firing off the request for the url’s contents”)
```

  Then line d gets executed.

  Since it is dispatching its closure to the main queue async, line d will return immediately.

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: “http://stanford.edu/...”) {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print(“did some stuff with the data, but UI part hasn’t happened yet”)
       }
g:     task.resume()
   }
h: print(“done firing off the request for the url’s contents”)
```

  Line f gets executed immediately after line d.
  Line e has not happened yet!
  Again, line d did nothing but asynchronously dispatch line e onto the (main) queue.

Multithreading

- Timing

  Let's look at when each of these lines of code executes …

```
a: if let url = URL(string: "http://stanford.edu/...") {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print("did some stuff with the data, but UI part hasn't happened yet")
       }
g:     task.resume()
   }
h: print("done firing off the request for the url's contents")
```

  Finally, sometime later, line e gets executed.
  Just like with line c, it's probably best to imagine this happens minutes after line g.
  What's going on in our program might have changed dramatically in that time.

Multithreading

- Timing

  Let’s look at when each of these lines of code executes …

```
a: if let url = URL(string: “http://stanford.edu/...”) {
b:     let task = session.dataTask(with: url) { (data: Data?, response, error) in
c:         // do something with the data
d:         DispatchQueue.main.async {
e:             // do UI stuff here
           }
f:         print(“did some stuff with the data, but UI part hasn’t happened yet”)
       }
g:     task.resume()
   }
h: print(“done firing off the request for the url’s contents”)
```

  Summary: `a b g h c d f e`

  This is the “most likely” order.

  It’s not impossible that line `e` could happen before line `f`, for example.

Demo

- Multithreaded Cassini
  - Let's get that URL network fetch off the main queue!

UITextField

- Like UILabel, but editable
  - Typing things in on an iPhone is secondary UI (keyboard is tiny).
  - More of a mainstream UI element on iPad.
  - Don't be fooled by your UI in the simulator (because you can use physical keyboard!).
  - You can set attributed text, text color, alignment, font, etc., just like a UILabel.
- Keyboard appears when UITextField becomes "first responder"
  - It will do this automatically when the user taps on it.
  - Or you can make it the first responder by sending it the `becomeFirstResponder` message.
  - To make the keyboard go away, send `resignFirstResponder` to the UITextField.
- Delegate can get involved with Return key, etc.
  - `func textFieldShouldReturn(sender: UITextField) -> Bool` // when "Return" is pressed
  - Oftentimes, you will `sender.resignFirstResponder()` in this method.
  - Returns <u>whether</u> to do normal processing when Return key is pressed (e.g. target/action).

UITextField

- Finding out when editing has ended

  Another delegate method …

  ```
  func textFieldDidEndEditing(sender: UITextField)
  ```

  Sent when the text field resigns being first responder.

- `UITextField` is a `UIControl`

  So you can also set up target/action to notify you when things change.

  Just like with a button, there are different `UIControlEvents` which can kick off an action.

  Right-click on a `UITextField` in a storyboard to see the options available.

Keyboard

- Controlling the appearance of the keyboard

  Remember, whether keyboard is showing is a function of whether its first responder.
  You can also control what kind of keyboard comes up.
  Set the properties defined in the `UITextInputTraits` protocol (`UITextField` implements).

```
var autocapitalizationType: UITextAutocapitalizationType  // words, sentences, etc.
var autocorrectionType: UITextAutocorrectionType          // .yes or .no
var returnKeyType: UIReturnKeyType                         // Go, Search, Google, Done, etc.
var isSecureTextEntry: Bool                                // for passwords, for example
var keyboardType: UIKeyboardType                           // ASCII, URL, PhonePad, etc.
```

- Other Keyboard functionality

  Keyboards can have accessory views that appear above the keyboard (custom toolbar, etc.).

```
var inputAccessoryView: UIView   // UITextField method
```

Keyboard

- The keyboard comes up over other views

  So you may need to adjust your view positioning (especially to keep the text field itself visible).
  You do this by reacting to the UIKeyboard{Will,Did}{Show,Hide} Notifications sent by UIWindow.
  We have not talked about what a Notification is yet, but it's pretty simple.
  You register a method to get called when a named "event" occurs like this …

```
NotificationCenter.default.addObserver(self,
                                       selector: #selector(theKeyboardAppeared(_:)),
                                           name: Notification.Name.UIKeyboardDidShow,
                                         object: view.window)
```

  The event here is Notification.Name.UIKeyboardDidShow.
  The object is the one who is causing the even to happen (our MVC's view's window).
  `func theKeyboardAppeared(_ notification: Notification)` will get called when it happens.
  The notification.userInfo is a Dictionary that will have details about the appearance.
  Almost always the reaction to the keyboard appearing over your text field is to scroll it visible.
  If the first responder is not in a scroll view, then position it so the keyboard never covers it.
  UITableViewController listens for this & scrolls table automatically if a row has a UITextField.

UITextField

- Other UITextField properties

```
var clearsOnBeginEditing: Bool
var adjustsFontSizeToFitWidth: Bool
var minimumFontSize: CGFloat       // always set this if you set adjustsFontSizeToFitWidth
var placeholder: String?           // drawn in gray when text field is empty
var background/disabledBackground: UIImage?
var defaultTextAttributes: [String:Any] // applies to entire text
```

- Other UITextField functionality

  UITextFields have a "left" and "right" overlays.

  You can control in detail the layout of the text field (border, left/right view, clear button).