Stanford CS193p

Developing Applications for iOS
Fall 2017-18

Today

- Multithreading
  - How to handle long-running/blocking activity (like accessing the network)
  - Cassini Demo
- Autolayout
  - Review
  - Size Classes
  - Concentration Demo

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 <u>immediately</u> 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!

Autolayout

- You've seen a lot of Autolayout already …
  - Using the dashed blue lines to try to tell Xcode what you intend
  - Reset to Suggested Constraints in lower right corner if blue lines were good enough
  - Ctrl-dragging to the edges and to other views
  - Size Inspector to look at and edit simple details of the constraints on a view
  - Clicking on a constraint directly in the storyboard and inspecting it
  - The "pin" menu in the lower right corner (there's an "arrange" button there too)
  - The Document Outline is an awesome place to view/edit and resolve problem constraints
- Mastering Autolayout requires experience
  - There's no substitute. You just have to do it to learn it well.
- Autolayout can be done from code as well
  - Search for the words "anchor" and "auto layout" in the `UIView` documentation.

Autolayout

- All that is not always enough

  Sometimes the geometry changes so drastically that simple autolayout cannot cope.
  You actually need to reposition the views entirely to make them fit properly.

- Concentration

  For example, what if we had 20 buttons in Concentration?
  It would be a lot nicer in tight vertical environments to go 5 across and 4 down.
  But in regular vertical environments, we'd be much better off with 5 down and 4 across.
  No combination of "pinning to the edges or other views" is going to accomplish this.

- Solution? We can vary our UI based on its "size class"

  A view controller's current size class says what sort of room it has to lay out in.
  It's not an exact number or dimension, it's just either "compact" or "regular" width or height.

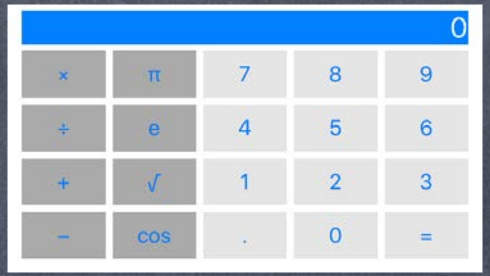
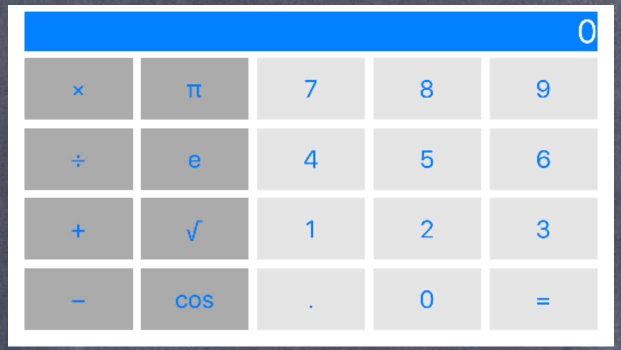
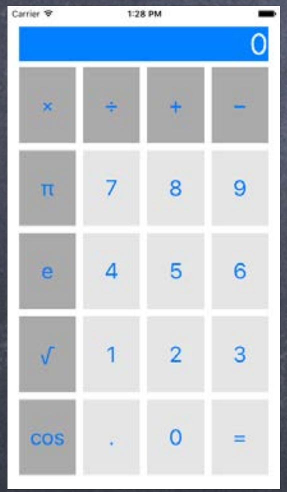
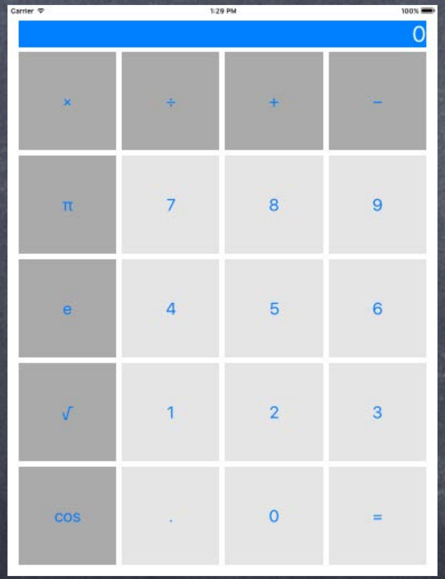
Autolayout

- iPhone
  - All iPhones in portrait are compact in width and regular in height
  - All non-Plus iPhones in landscape are compact in both dimensions
- iPhone Plus
  - iPhone Plus is also compact in width and regular in height in portrait
  - But in landscape, the iPhone Plus is only compact in height (i.e. it's regular in width)
- iPad
  - Always regular in both dimensions
  - But depending on the environment an MVC is in, it might be compact (e.g. split view master)

Size Classes

	Compact Width	Regular Width
Compact Height	iPhones (non-Plus) in Landscape	iPhone Plus in Landscape
Regular Height	iPhones in Portrait or Split View Master	iPads Portrait or Landscape

Size Classes

	Compact Width	Regular Width
Compact Height		
Regular Height		

Autolayout

What can we do based on our size class?

You can vary many properties in `UIView` …
Fonts, background color, `isHidden`, even whether a view is installed in the view hierarchy.
Most importantly though, you can also include or exclude any constraint based on size class.
By doing this, we can rearrange our UI completely differently in different size class situations.
InterfaceBuilder has full support for doing this.

Using Size Class information

You can find out your current “size class” in code using this method in `UIViewController` …

```
let myHorizSizeClass: UIUserInterfaceSizeClass = traitCollection.horizontalSizeClass
```

The return value is an `enum` … either `.compact` or `.regular` (or `.unspecified`).

However, it is rare to look at our size class in code.
Again, we’re almost always going to do this in InterfaceBuilder.

Demo Code

Download the Cassini demo code from today's lecture.