iOS Foundations II
Day 6

- Review
- Autolayout
- Optionals
- Text Fields if we have time

Auto Layout is Important

- With all these damn screen sizes and orientations now days, autolayout is the only way to go for properly laying out your interface.
- Apple chose to support iPhone 4s with iOS 9, so now we gotta deal with that screen size for another year. Great.
- Auto Layout is extremely flexible and feature rich, Apple has been working on it for a long time. You can even animate autolayout changes.

Autolayout

- A constraint-based layout system for making user interfaces.
- AutoLayout works by you bossing it around.
- Specifically, you can tell it 2 things about every view in your interface:
    1. Size - How big the view is going to be
    2. Location - Where the object is going to be located in its super view
- Once told ‘the rules’, autolayout will enforce the rules you setup.
- These rules are setup using constraints.

Auto Layout Fact #1

- **Auto Layout is a constraint-based layout system. Constraints are the fundamental building block of Auto Layout**
- Constraints express rules for the layout of elements in your interface.
- When all of your constraints are considered, there should only be ONE possible layout. If there is more than one, Auto Layout complains because how the heck is going to know which one to pick.

Auto Layout Fact #2

- **Constraints are always attached to attribute(s).**
- An attribute is one of leading (aka left), trailing (aka right), top, bottom, width, height, centerX, centerY, and baseline.
- So every view has each one of these attributes.

Auto Layout Fact #3

- **Constraints take the form of the mathematical equation y = mx+b.**
- That translates to firstItem.attribute = secondItem.attribute * multiplier + constant.
  - Constant : the physical size or offset, in points, of the constraint
  - Multiplier: Often times its kept at its default of 1. But it can be useful to help create ratios. Like the firstItems width attribute should be equal to the secondItem's width attribute multiplied by 0.5.

Auto Layout Fact #4

- **Most constraints have both a firstItem and secondItem in their equation, but width and height constraints often only have firstItem.**
- For example, if you wanted to make a view have a width of 40 points. You would create a constraint with this equation:
  - view.width = 0*1.0 + 40
  - theres a 0 where normally there would be a secondItem.

Auto Layout Fact #5

- **Constraints are just objects. The class is called NSLayoutConstraint.**
- Every UIView has a method called constraints() that returns an array of all constraints held by that view. It not very common to use this though.
- Later in the course we will create an app without any storyboards and nib/xibs. And we will get plenty of practice creating constraints in code.
- You can even create outlets to constraints from the storyboard.

Optionals

Optionals

- You use optionals in situations where a value may be absent.
- An optional says:
  - There is a value and it equals x

    OR

  - There isn’t a value at all.
- The concept of optionals does not exist in Objective-C or even C!

Marking an Optional

- A question mark at the end of a type indicates that the value it contains is optional, meaning it might contain a value or it might be empty:

```
class Person {

    var firstName : String?
```

Swift is strict!

- Swift does not allow you to leave properties in an undetermined state.
- They must be:
  - given a default value

    OR
  - a value set in the initializer

    OR
  - or marked as optional

Accessing a value inside an optional

To access the value inside of an optional variable, you must force unwrap it with the exclamation mark:

```
// return the first and last names in a string
func returnFullName() -> String {
    return "\(self.firstName!) \(self.lastName)"
}
```

You can also use a question mark to use whats called optional chaining.

Implicitly Unwrapped

Instead of marking an optional with a ?, using an exclamation point ! will mark it as implicit unwrapped:

```
class Person {

    var firstName : String!
```

This makes it so you don't have to unwrap this optional to access its value.
So basically you are saying “I’m making this an optional, but this thing will always have a value when I need to access it”

Optional Binding

- You can use **optional binding** to find out wether an optional contains a value, and if so, to make that value available as a temporary constant or variable that is unwrapped.
- The syntax of an optional binding:

```
if let constantName = someOptional {
    statements
}
```

using optional binding

```
func printTitleValue(value : String?) {

    if let title = value {
        println(title)
    }
}
```

not using optional binding

```
func printTitleValue(value : String?) {

    if value != nil {
        let title = value!
        println(title)
    }
}
```

Downcasting + Optional Binding

- It is very common to combine optional binding and downcasting
- Downcasting is used whenever a constant or variable of a certain type may actually refer to an instance of a subclass behind the scenes.
- When you think this is the case, you can use downcasting to attempt to cast the variable or constant to the subclass.
- There are two forms of down casting:
  - optional form: as?
  - forced form : as!

Demo