iOS Dev Accelerator Week 1 Day 5

- Swift Access Control & Encapsulation
- Week Wrap up
- Extra Credit features

Access Control

- Access Control is a universal programming concept that allows developers to restrict access to parts of their code from code in other source code files or modules
- It allows you to hide implementation details of your code, and specify a designated interface through which your code should be interacted with

Access Control in Swift

- In swift, you can assign specific access levels to individual types (classes, structures, enumerations)
- You can also assign specific access levels to properties, methods, inits, and subscripts of those types.
- You can also restrict protocols, global constants/variables/functions (not very common)

Access Control in Swift

- Swift provides default access control levels to all of your code, which usually work well for your typical app projects
- If you are writing a regular, single-target app, you don’t actually need to specify access controls at all if you don’t want to.
- But its still good to know how it works!

Swift Modules and Source Files

- Swift’s access control model is based around on the concept of modules and source files
- A **module** is a single unit of code distribution. Like an app or a framework.
- A **source file** is a single Swift file within a module. It is a single file within an app or framework

Access Levels

- Swift provides 3 different access levels for your code:
- **Public** : enables entities to be used within any source file from their defining module (app or framework), and also in a source file from another module that imports the defining module.
- **Internal** : enables entities to be used within any source file from their defining module, but NOT in any source file outside of that module. This is the default access level, and it works great for single target apps!
- **Private** : restricts use of an entity to its own defining source file. Use this to hide implementation details of a specific piece of functionality.

Encapsulation

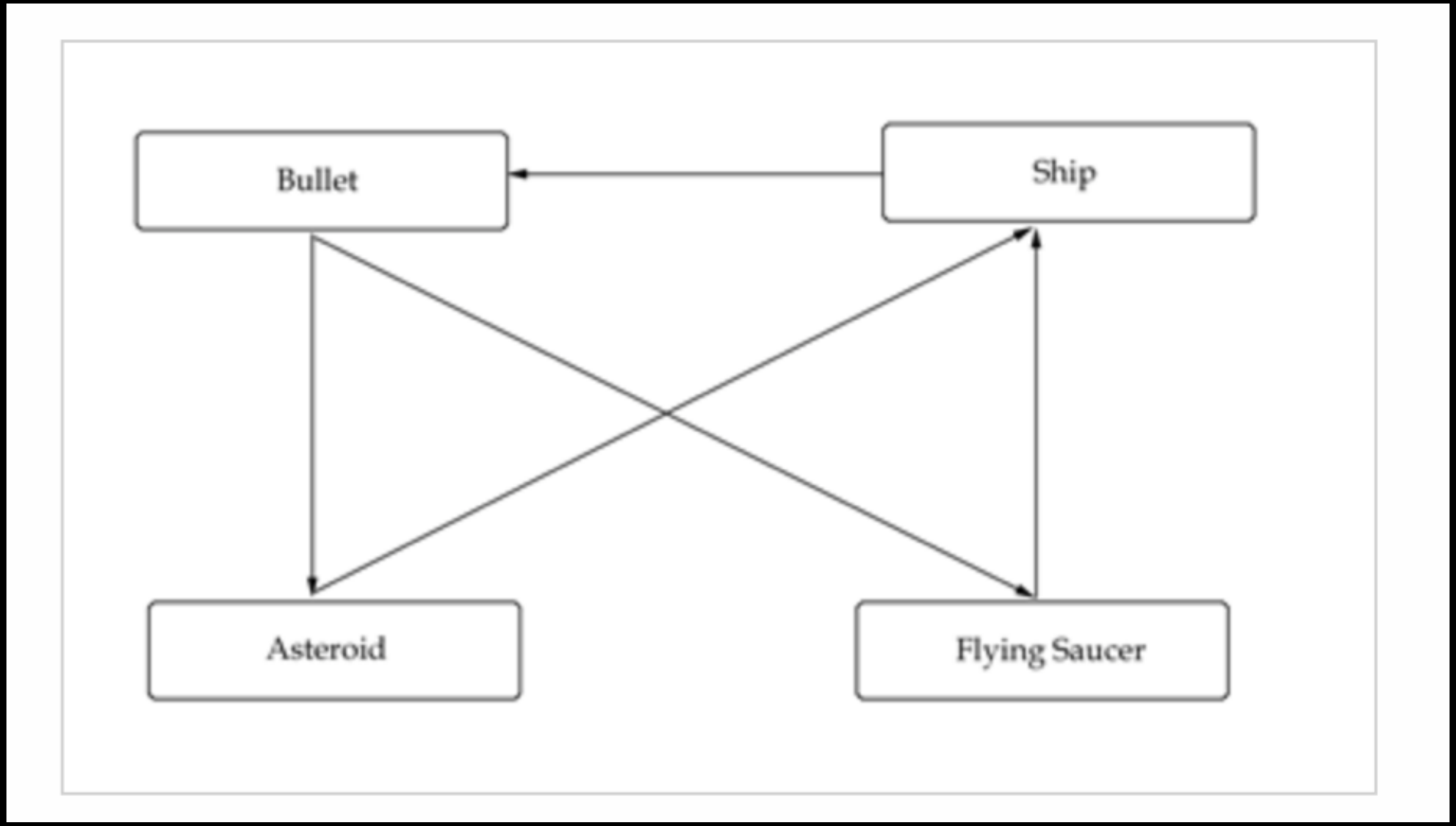
- In computer science and object oriented programming, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.
- Usually only the object's own methods can directly inspect or manipulate its data.
- When a programming language has features to help you with encapsulation, it is usually referred to as access control.
- In Objective-C, the separation of .h and .m does most of the work for us.

Encapsulation Benefits

- Encapsulation provides a number of benefits:
  - **Hides complexity:** Think of a car. A car has super complex internals. But we as users only interface with simple controls, like the wheel and ignition. Imagine if you had to actually know exactly how a car works to use it. We can consider those wheel and ignition controls its public interface.
  - **It helps us achieve loose coupling:** Encapsulation helps us decouple modules that comprise our app, allowing them to be developed, used, tested, and understood in isolation.
- Lets look at examples of these concepts.

Tight Coupling

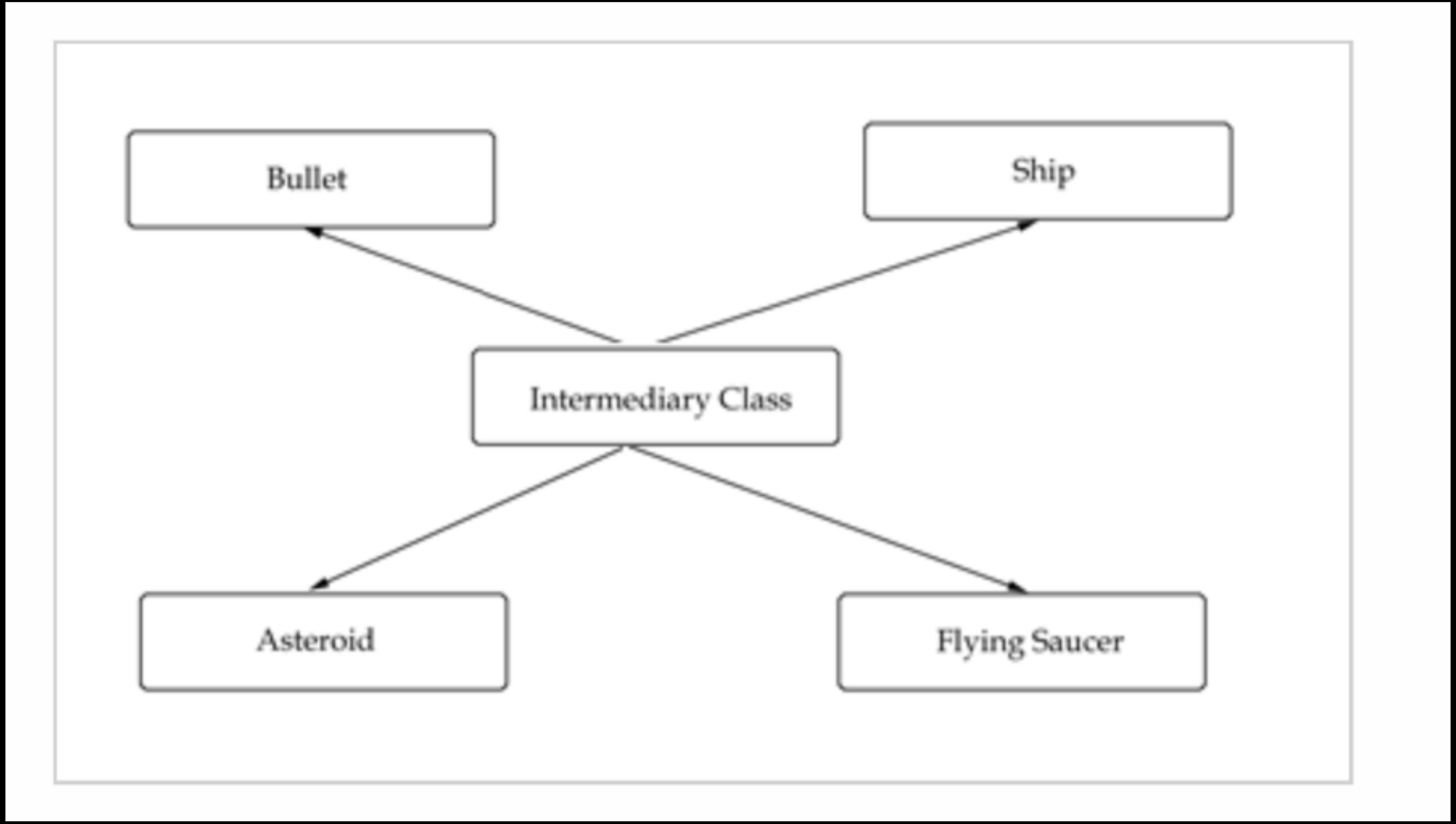
- Lets take a look at a game like Asteroids for example
- Asteroids has 4 main classes : Bullet, Ship, Asteroid, and Enemy
- With tight coupling, the ship would fire a bullet, keep track and update the bullets location, check if it hit an asteroid, and if it did destroy the asteroid, or or if it hit an enemey, destroy an enemy.



The issue with this is if later on down the road, for example, if we change how bullets work, or change the bullet to a lazer, we will have to update all the other classes appropriately.

Loose Coupling

- Instead lets design a system where the ship fires a bullet, and then doesn't have to worry about it after that point.
- We will let another class, maybe a class called GameLogic, handle that stuff
- With this GameLogic class able to work as an intermediary class, our diagram looks much cleaner:



Now if we change how bullets work, we only have to update our intermediary class. Bullet and Ship are loosely coupled.

Demo

Week 1 Patterns

- Crucial patterns for any language: Single Responsibility Pattern, Model View Controller, Lazy Loading, Concurrency, Callbacks, Encapsulation
- Crucial patterns for iOS: Delegation, Singletons,

Week 1 Frameworks

- Objective-C System frameworks: Foundation, UIKit
- Cocoa Frameworks: Social, Accounts

Week 1 UI Techniques

- Storyboards
- Nibs/Xibs
- Autolayout
- Auto sizing cells
- Labels, ImageViews, Buttons
- Table Views

Week 1 Important Classes

- NSOperationQueue
- UINavigationController

Week 1 Swift features

- Optional Binding
- Type methods
- [weak self]
- Closure Expressions
- Switch statements

Extra Credit Features

Infinite scrolling

- As the user approaches the bottom of the list of tweets, fetch another batch of tweets to plug in to the bottom
- So the user should never be able to actually reach the bottom of the table view
- Check how close the indexPath.row is to the end of the tweets array in cellForRow or willDisplayCell
- Use the max_id parameter to tell twitter you want tweets older than a certain tweet id

Pull to refresh

- Implement pull to refresh, just like in the twitter app (and many other apps)
- Check out the UIRefreshControl class
- Use the since_id parameter to tell twitter you want tweets newer than a certain tweet’s id

Show Retweeters

- If a tweet is a retweet, show the profile image of the original author.
- When selecting a retweeted tweet, allow the user to see the original author in the tweet info VC as well
- When you click on the original author, go to their tweets
- In theory this should allow you to infinitely be pushing view controllers onto the stack of different people's twitter accounts.