INTRODUCTION TO IOS APP DEVELOPMENT

FIRST CLASS

Elliot Schrock

AGENDA

How this course will be taught

Motivation

Introduction to programming fundamentals

Tour of Xcode

Introduction to MVC

THIS COURSE

1 class = 1 instructor

Misunderstanding = our fault

Please please PLEASE ask questions

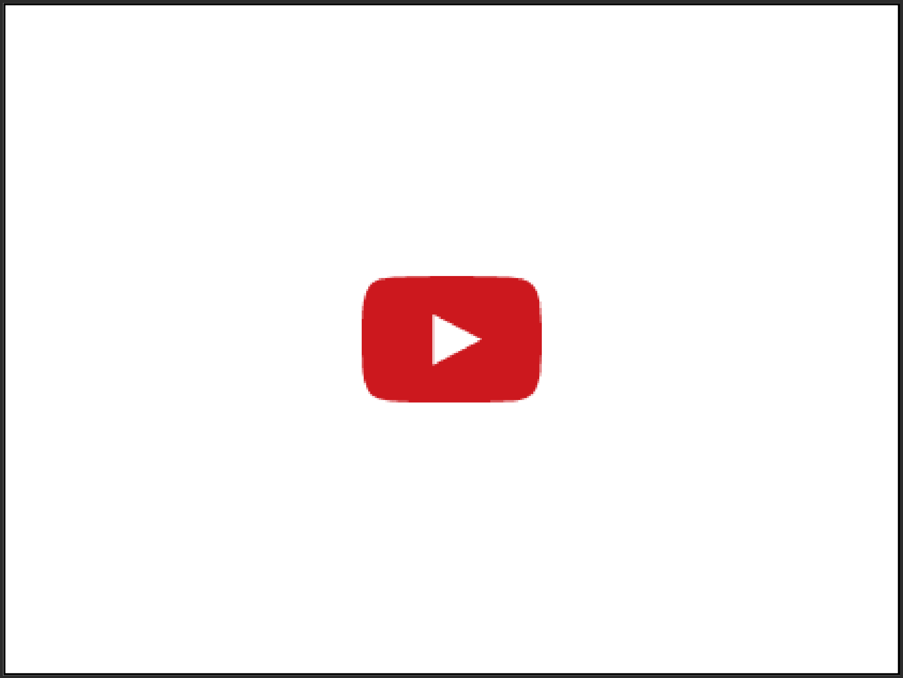
gdi.ios.anonymous.question@gmail.com
@GDI.Question!

My email: elliot.schrock@gmail.com

WHY SHOULD I CARE ABOUT PROGRAMMING?

It's MAGIC!

Like anything in the wizarding world, it takes precision



LIST OF COMMANDS

Each command is like a spell

Generally one command per line:

```
[someObject doSomething];
[anotherObject doSomethingElse];
someObject.someProperty = anotherProperty;
```

OBJECT ORIENTED PROGRAMMING (OOP)

Create 'objects' that do things

In OOP, (almost) everything is an object

Description of:

- What the object can do
- What other objects it owns

Write this description in its own file (kind of)

EXAMPLES

Dog

Pen

Arm

OBJECTS

Type of object = Class

The object itself = instance

PROPERTIES

Access a property using a 'dot'

```
arm.hand;
arm.elbow;
```

Can even chain them together:

```
arm.hand.pinkieFinger;
```

FUNCTIONS

Functions = actions

Tell an object to do something with brackets:

```
[hand open];
[hand close];
```

They can be given some number of things (called 'arguments'):

```
[student takeThisPen:pen];
```

```
[student putThisInkCartridge:cartridge intoThisPen:pen];
```

Or they can be combined with properties:

```
[arm.hand.pinkieFinger wiggle];
```

DEFINING A FUNCTION

```
- (void)open
{
    //do something
}
```

Functions with arguments:

```
- (void)takeThisPen:(Pen *)thisPen
{
    myPen = thisPen;
}
```

```
- (void)putInkCartridge:(InkCartridge *)cartridge intoPen:(Pen *)pen
{
    pen.cartridge = cartridge;
}
```

Functions can 'return' things:

```
- (Pen *)giveMeYourPen
{
    return myPen;
}
```

PUTTING IT ALL TOGETHER

```
- (Pen *)pen:(Pen *)pen withInkCatridge:(InkCartridge *)cartridge

{
    [pen open];
    pen.inkCartridge = cartridge;
    [pen close];
    return pen;
}
```

HAND.H

```
#import <Foundation/Foundation.h>
#import "Finger.h"

@interface Hand : NSObject
@property (nonatomic, strong) Finger *indexFinger;
@property (nonatomic, strong) Finger *middleFinger;
@property (nonatomic, strong) Finger *ringFinger;
@property (nonatomic, strong) Finger *pinkieFinger;

- (void)open;
- (void)close;
@end
```

ARM.H

```
#import <Foundation/Foundation.h>
#import "Hand.h"

@interface Arm : NSObject
@property (nonatomic, strong) Hand *hand;

- (void)extend;
- (void)retract;
@end
```

EXERCISE

```
#import <Foundation/Foundation.h>
#import "Hand.h"

@interface Arm : NSObject
@property (nonatomic, strong) Hand *hand;

- (void)extend;
- (void)retract;
@end
```

```
#import <Foundation/Foundation.h>
#import "Finger.h"

@interface Hand : NSObject
@property (nonatomic, strong) Finger *indexFinger;
@property (nonatomic, strong) Finger *middleFinger;
@property (nonatomic, strong) Finger *ringFinger;
@property (nonatomic, strong) Finger *pinkieFinger;

- (void)open;
- (void)close;
@end
```

EXERCISE FUNCTION

```
- (void)pickUpPen:(Arm *)arm
{

}
```

A SOLUTION

```
- (void)pickUpPen:(Arm *)arm
{
    [arm.hand open];
    [arm extend];
    [arm.hand close];
    [arm retract];
}
```

EXERCISE FUNCTION

```
- (void)putDownPen:(Arm *)arm
{

}
```

A SOLUTION

```
- (void)putDownPen:(Arm *)arm
{
    [arm extend];
    [arm.hand open];
    [arm retract];
    [arm.hand close];
}
```

BREAK TIME!

IMPLEMENTATION

Header (.h) = how others interact with an instance.

Implementation (.m) = what happens when they do.

Recall `pickUpPen`. In the header:

```
#import <Foundation/Foundation.h>

@interface Arm : NSObject
- (void)pickUpPen:(Arm *)arm;
@end
```

In the implementation:

```
#import "Arm.h"

@implementation Arm
- (void)pickUpPen:(Arm *)arm {
    [arm.hand open];
    [arm extend];
    [arm.hand close];
    [arm retract];
}
@end
```

SELF

The "self" keyword refers to the current instance

Example: rewrite of `pickUpPen:`

```
#import "Arm.h"

@implementation Arm
- (void)pickUpPen
{
    [self.hand open];
    [self extend];
    [self.hand close];
    [self retract];
}
@end
```

A SHORT DIGRESSION: MEMORY

Instances are stored in memory (RAM)

Each instance has an address in memory

That address is how the computer refers to the object

THE DENTIST EXAMPLE

I have a contact in my phone called 'Dentist'

Move to a new city, need a new dentist

Find a new one and replace the old dentist's phone number with the new one

```
- (void)setDentist:(Dentist *)newDentist
{
    self.dentist = newDentist;
}
```

VARIABLES

Create a variable like so:

```
Dentist *myDentist;
```

Right now, `dentist` is equal to `nil`

Usually we'll *instantiate* the object at the same time as we create the variable:

```
Dentist *myDentist = [[Dentist alloc] init];
```

PM me on Slack with how to create a variable of class `Arm` named `someonesArm`

```
Arm *someonesArm = [[Arm alloc] init];
```

PERSON

SCOPE

Variables have a shelf life

Scope of a variable is where it is valid

```
- (void)setDentist:(Dentist *)newDentist
{
    self.dentist = newDentist;
}
```

```
- (void)makeAppointmentWithDentist
{
    DentistAppointment *appointment = [[DentistAppointment alloc] init];
    appointment.dentist = self.dentist;
    appointment.patient = self;
    [appointment confirm];
}
```

Rule of thumb: only valid between the curly braces in which it was born

PRIMATIVES

`int` - integers

`BOOL` - boolean values (`YES` and `NO`)

`float` - decimal numbers

Don't need * because not an address

Math operations work as you'd expect:

```
    int myLuckyNumber = 5 + 4;
    myLuckyNumber = 86 - 77;
    myLuckyNumber = 3 * 3;
    myLuckyNumber = 81 / 9;
    myLuckyNumber = 103 % 47;
```

EXERCISE

Let's write a function that adds two integers together and returns the result.

```
- (int)add:(int)firstNumber to:(int)secondNumber
{
    return firstNumber + secondNumber;
}
```

Now you: write a function that subtracts one int from another and returns the result.

BOOLEANS

Recall that `BOOL` is a boolean value (either `YES` and `NO`)

```
int myLuckyNumber = 9;
if (myLuckyNumber == 7){
    //do something only if myLuckyNumber is 7
}
```

```
int myLuckyNumber = 9;
if (myLuckyNumber != 7){
    //do something only if myLuckyNumber is NOT 7
}
```

SUBCLASSING

Add functionality to a class by subclassing it (extending it)

It 'inherits' all the functions of its super class

Plus you can add your own functions

Suppose I have a class called `Automobile`

`Automobile` has wheels, a driver's seat, can brake, can accelerate...

```
#import "Automobile.h"

@interface Truck : Automobile
@property (nonatomic, strong) FlatBed *flatBed;

- (void)haul:(NSObject *)something;
@end
```

BREAK TIME!

REAL LIVE APP

https://github.com/schrockblock/boston-ios

Download zip, extract it

Run `pod install`

Open the workspace

HOMEWORK

https://github.com/schrockblock/gdi-homework-1

I accidentally deleted a class! Use Xcode to figure out which one, and re-create it so that the project runs.