iOS Dev Accelerator
Week 5 Day 1

- Objective-C Intro
- MapKit

Objective-C

- A subset of C
- An object-oriented language
- Uses ‘message passing’, inspired by SmallTalk
- Dynamically typed (with optional static typing, more on this later)

Swift

```
super.viewDidLoad()
```

Objective-C

```
[super viewDidLoad];
```

receiver — message

(This is Objective-C's message sending syntax. Its just a fancy way of saying you called a method on an object)

Swift

```
var things = [AnyObject]()
```

Objective-C

```
NSMutableArray *persons = [[NSMutableArray alloc] init];
```

main.m

- Entry Point for Objective-C application
- UIApplication is initialized with a reference to your App Delegate
- Just know this is here, you never have to actually touch this

```
#import <UIKit/UIKit.h>
#import "PASAppDelegate.h"

int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                                 NSStringFromClass([PASAppDelegate class]));
    }
}
```

.h and .m

- In Objective-C, a class's public interface is described separately from its internal implementation.
- So, every class you create in Objective-C has a separate .h and .m file.
- The .h file is referred to as the header file or sometimes the interface file.
- The .m file is referred to as a class's implementation file.
- Basically, properties and methods you want publicly accessible in your class, put it in your .h
- Everything you want to keep private, only put it in your .m
- Keep in mind, the .h will only contain declarations of methods, the implementation of those methods always goes in the .m, regardless of if you want to have them be public or not.

```objc
@interface Person : NSObject

- (NSString *)fullName;

@end
```

Header

```objc
@implementation Person

- (NSString *)fullName {
    // TODO: implement
    return nil;
}

@end
```

Implementation

@interface and @implementation

- @interface is used to declare a class interface
- @implementation is used to declare a class's implementation
- The .h file will always have an @interface block.
- The .m file will always have an @implementation block.
- A lot of the times you will see an @interface block in the .m as well. This allows the developer to declare private properties.

Demo

Method Prototypes

- Objective-C requires public methods to be defined in header files
- Have same signature as functions, but without a body

```
- (void)updateDataStore;

- (void)setPersons:(NSArray *)persons;

- (BOOL)shouldTerminateWithOptions:(id)options
```

Methods

- Instance method definitions prefaced with **-** sign.

```
- (NSString *)fullName
```

```
- (void)startAnimations
```

- Class (aka type) method definitions prefaced with **+** sign.

```
+ (UIApplication *)sharedApplication
```

```
+ (NSString *)formattedString
```

Methods

- Return type placed before method name, in parenthesis
- Method name follows return type
- A optional colon : follows the name, before any parameters
- Each parameter is named, prefaced by its type

```
- (NSDictionary *)parseJSON:(NSDictionary *)jsonDitionary usingParser:(CFParser *)parser
```

```
func parseJSON(jsonDictionary : NSDictionary, usingParser parser : CFParser) -> NSDictionary
```

Demo

Messages

Messages are just how you call methods on objects/classes in Objective-C

```
[NSUserDefaults standardUserDefaults]
```

```
[receiver message]
```

```
[NSArray arrayWithObject:@42]
```

```
[receiver message:argument]
```

Message Sending

- ObjC runtime keeps a list of method names and address locations
- The method name is the **`Selector`**
- When message is sent, ObjC checks if the receiver responds to the message (**`respondsToSelector:`**)
- All messages sent to **`nil`** are ignored, not an exception like in Swift!

Demo

Properties

- Defined using the `@property` keyword
- Defined in header file, within `@interface` blocks
- Modified using attributes, following keyword
- By default, properties are `(atomic, readwrite, assign)`
- Setters and Getters automatically generated

Anatomy of a property

- Creating a property in your class creates 3 things:
  - An ivar or instance variable: properties are always backed by an instance variable. This is the actual variable that will be storing the object for the property. By default the name of the ivar is underscore and then the name of the property.
  - A setter for the ivar (more on this in a bit)
  - A getter for the ivar (more on this in a bit)

Demo

Setters and Getters

- You access or set an object's properties via accessor methods
- These are commonly referred to as setters and getters
- By default, these accessor methods are created for you by the compiler, all you have to do is declare the property. You used to have to declare them yourself, but not anymore.
- These accessor methods follow specific conventions:
  - The method used to access the value (the getter) has the same name as the property
  - The method used to set the value (the setter) started with the word "set" and then uses the capitalized name of the property

Getters and Setters

```objc
@property (strong,nonatomic) NSString *name;
```

Property

```objc
NSString *myName = [self name];
```

Getter

```objc
[self setName:@"Brad"];
```

Setter

Demo

Property Modifiers

- Property modifiers specify (bolded means default):
  - Thread restrictions (nonatomic, **atomic**)
  - Access restrictions (**readwrite**, readonly)
  - Memory management (retain, copy, assign,weak, **strong**, unsafe_retained)
  - Can specify custom names for getter/setter
  - Most common pairing is (strong,nonatomic)
  - **Always explicitly declare your memory management, and thread restriction modifiers. This is best practices for Objectie-C.**

```
@property (nonatomic, strong) NSString *title;

@property (nonatomic, strong, readonly) NSArray *cars;

@property (atomic, assign, getter=isActive) BOOL *active;
```

Demo

Dot Notation

- Somewhat confusingly, Objective-C allows dot notation (just like Swift) when using your setters and getters on properties.
- Thats the only time you can use it!
- You can't use it to call methods like in Swift.
- So you will see a lot of self.*nameOfProperty* in some people's Objective-C code.
- Its a fiercely debated topic on the interwebs.
- I liked it before Swift, and I like it even more after Swift.

Dot Notation

```objc
@property (strong,nonatomic) NSString *name;
```

Property

```objc
NSString *myName = [self name];
//is the same as
NSString *myName1 = self.name;
```

Getter

```objc
self.name = @"Hello";
//is the same as
[self setName:@"Hello"];
```

Setter

A good rule of thumb: if you see dot notation used to the left of the equal sign, its the setter. To the right, its the getter.

Demo

Creating new instances

```
[[NSArray alloc] init]
```

- Call to **`alloc`** will allocate enough memory for the instance and its properties
- `alloc` returns an `id` type— could be anything
- Call the `init` will set suitable initial values for properties.

Implementing init

- Call [super init] and assign the result to self **(Best Practice)**
- If self is not nil setup initial state and return self
- return either id or instancetype

```
- (instancetype)init {
    if (self = [super init]) {
        // TODO: implement
    }

    return self;
}
```

dealloc

- Since ARC, you don’t need to explicitly release memory in dealloc
- But dealloc is still a good place to handle teardown operations
- Do not call super

```
- (void)dealloc {
    //
}
```

Creating custom inits

- Less structured than the initializers we created in Swift.
- There is no special init keyword, but best practices is to start with the word init
- You will still need to have self = [super init] in the first line
- And you need to declare in your .h if you want it to be used publicly, which most inits are

Creating custom inits

```objc
@interface Person : NSObject

@property (strong,nonatomic) NSString *name;

-(instancetype)initWithName:(NSString *)name;

@end
```

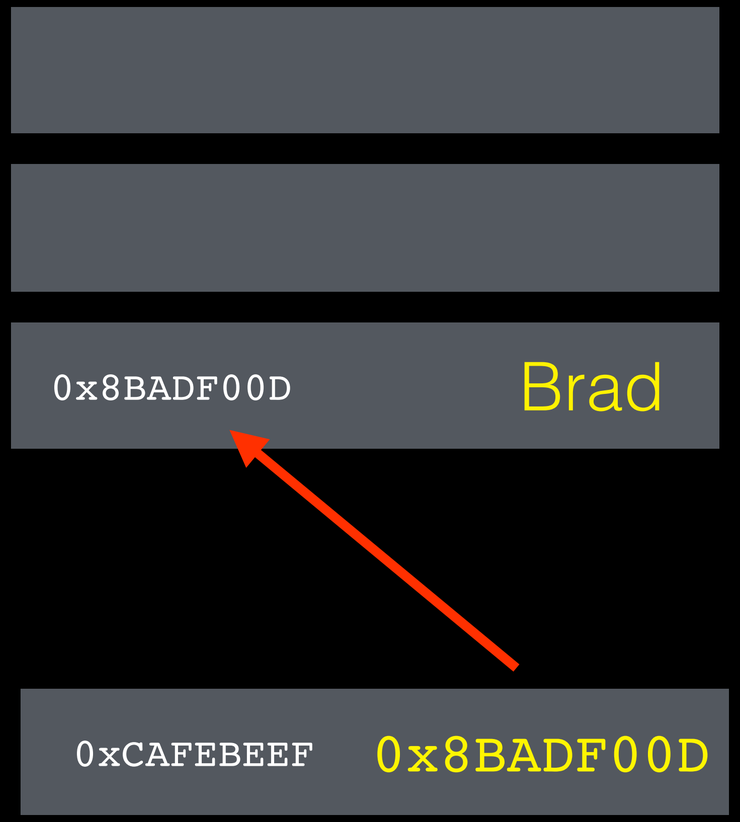
```objc
@implementation Person

-(instancetype)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        [self setName:name];
    }
    return self;
}

@end
```

Demo

Pointers

- Creating variables, or allocating objects, stores values in memory.
- The location of a value in memory is called the *memory address*
- *So a* pointer is just a variable with the memory address of *another* value in memory
- Thus, a pointer *points to* the location in memory where a value is stored.

Pointers




```
NSString *name = @“Brad”;
```

* is the **value at address** operator

Pointers

- Typically, you only need to declare a variable or property as a pointer once, with the * operator
- Once you have done that, Xcode knows that this variable or property is a pointer, and accessing it means accessing the underlying value it points to in memory.
- Most of the time, the only time you will use the & operator is when dealing with NSError objects, which we will see later on.

Demo

#import or #include

- #include comes from C
- #import was added to Objective-C
- Using #import ensures a file is only ever included once; No recursive imports
- Good rule of thumb: #import for ObjC, #include for C libraries

#import

- You need to avoid what is referred to as ‘circular imports’
- This happens when class A’s .h imports class B , and class B’s .h imports class A
- To avoid this, you can use @class forward declaration to let a .h know about a certain class, and then do the actual import in the .m file.

Demo

The @ sign

- Used to denote certain keywords:
  - `@interface`, `@implementation`, `@property`, `@protocol`, `@selector`, `@synchronized`, `@end`
- Used for `NSArray`, `NSDictionary`, and `NSNumber` shorthand
  - `@[view1, view2, view3], @{@“key”: @(42)}`
- Used for literal syntax of string values
  - `NSString *organization = @“Code Fellows”`

A note on Arrays and Dictionaries

- Arrays and Dictionaries in Objective-C are structurally the same as in Swift. Or really any other programming language.
- The difference is in the safety
- In Objective-C, there is no built in way to ensure that all objects in an array or dictionary are of a certain type, or they are all of the same type.
- So you need to be careful when using these data structures

Demo

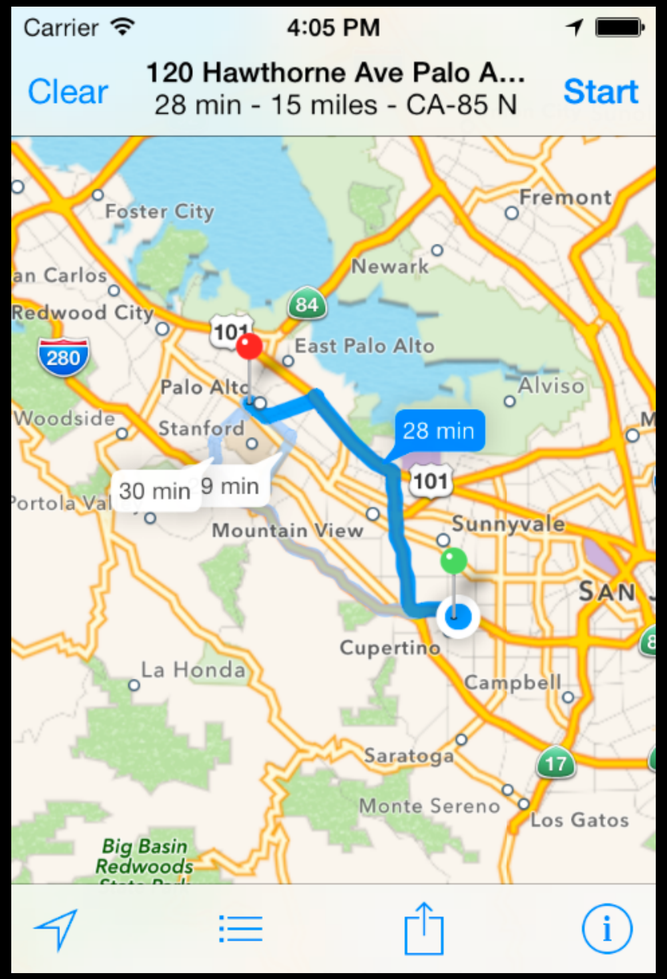
MapKit

Locations and Maps

- “Location-based information consists of two pieces: location services and maps”
- Location services are provided by Core location framework
- Maps are provided by the MapKit framework
- Both frameworks available on iOS and OSX

MapKit

- “The Map Kit framework lets you embed a fully functional map interface into your app”
- Provides many of the same features that the maps app on iOS and OSX does.
- Street level map info, satellite imagery, etc.
- Programmatically zoom, pan, and pinch, display 3d buildings, and show customized annotations, in addition to regular touch controls.



MapKit Info

- To enable Maps in your app, turn on the Maps capability in your Xcode project.
- Any map view in your app represents a flattened earth sphere.
- Uses a Mercator map projection to project the round earth to a flat view.
- Uses Prime Meridian as its central meridian.

MapKit data points

- Map Kit supports 3 basic ways to specify map data points:
  - **Map Coordinate**: is a lat/long pairing using the CLLocationCoordinate2D struct. Also used to specify areas with MKCoordinateSpan and MKCoordinateRegion.
  - **Map Point**: An x and y value on the map projection. Mainly used for custom overlays and designating their shape and location on your view.
  - **A Point**: Regular CGPoint, used in CGSize and CGRect.
- The data types you use is usually chosen for you by the API you are implementing. When saving locations in files or inside your app, coordinates are the most precise.

Getting Map view on screen

- “MKMapView is a self contained interface for presenting map data in your app”
- Displays map data, managers user interactions, and hosts custom content provided by your app.
- Never subclass MKMapView.
- Has a delegate object which the map view reports all relevant interactions to.
- Drag onto your storyboard, or programmatically initialize an instance of MKMapView.

MKMapView Specifics

- Keep in mind an MKMapView is still a view, so you can manipulate its size and position, and add subviews to it just like any other view.
- Subviews added to it will not scroll when the user pans the map, its like an overlay.
- If you want something to stay stationary relative to a specific map coordinate, you just use an annotation.
- By default a new map view is configured for user interaction and to display map data. Uses a 3D perspective by enabling pitch (not available on simulator!).
- Manipulate the pitch and rotation by creating an MKMapCamera Object (more on this later).
- Change its Type attribute to go from satellite to map data or a mix.
- You can limit user interaction by changing values in rotateEnabed, pitchEnabled, zoomEnabled, etc properties.
- Use the delegate to respond to user actions.

MKMapView Region

- “The region property of the MKMapView class controls the currently visible portion of the map”
- Typically when a map view is created, its region is set to the entire world.
- Change this by setting a new value to region, has a setRegion:animated method for smooth zooming.
- The properties type is MKCoordinateRegion struct, with two values:
  - center : CLLocationCoordinate2D
  - span : MKCoordinateSpan
- Span defines how much of the map at a given point should be visible. MKCoordinateSpan is measured in degrees, but you can create one with meters with MKCoordinateRegionMakeWithDistance.

MKMapView Zooming and Panning

- Zooming and panning can be done programmatically in addition to user interaction.
- To pan, but keep the same zoom level, change the value in the centerCoordinate property of the map view. Or call the map view's setCenterCoordinate:animated method to animate the change.
- To change the zoom level (and optionally pan the map), change the value in the region property of the map view. Or use the animated method.

MKMapView and User Location

- MKMapView comes with built in support for displaying the user's location.
- Just set the showsUserLocation property to true.
- Shows the user's location and adds an annotation on the users location.
- Interfacing with CoreLocation required to use this (more on CoreLocation tomorrow)

Demo