iOS Dev Accelerator Week 6 Day 1

- Objective-C!!!

Objective-C History

- originally developed in the 1980's
- It was selected as the main language for NeXSTEP computers, for which OSX and iOS come from.
- It is considered a combination of smalltalk and C.

LLVM

- LLVM is a ‘compiler infrastructure’
- A compiler is just a computer program that transforms source code written in a high level programming language (swift, obj-c) into a lower level language (assembly or machine code)
- written in C++

Clang

- A compiler front end for C,C++,Objective-C, Objective C++, and Swift with LLVM as its backend.
- Works together with LLVM to provide you all the tools you need to develop Obj-C/ Swift apps with Xcode.
- The front end (like clang) of a compiler usually deals with scanning your input, parsing through it, and performing type checking to make sure everything is right.
- The back end (like LLVM) of a compiler optimizes your code and maps it to assembly code, which is then turned into machine code.

Objective C vs Swift

- explicit use of pointers
- .h and .m file
- properties and ivars
- call methods in brackets
- import your classes into other classes that need to use them.
- anything can be nil
- end a line with a semi colon

- pointers hidden
- one .swift file
- just properties
- dot notation
- all your swift classes know about your other swift classes
- optionals
- semi colons not required

.h and .m files

- the .h file, or header file, of a class is the public interface of the class.
- It defines all the public facing variables and methods of the class.
- the .m file, or implementation file, defines the implementation of the class. This is really just the inner workings of each method of the class.
- In addition, the m file can also declare a private interface for declaring private variables that are only used internally by the class.

.h example

```
@interface ViewController : UIViewController

@property (strong,nonatomic) NSString *myPublicString;

-(void)myPublicMethod;

@end
```

.m example

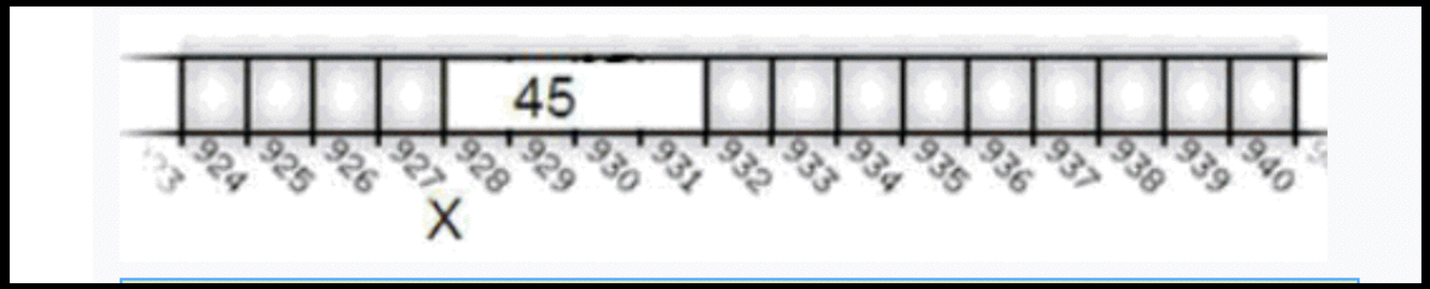
```objc
#import "ViewController.h"

@interface ViewController ()

{
    //instance variables
    NSString *_myName;
}

//properties
@property (strong,nonatomic) NSString *myName;
@property (strong,nonatomic) NSString *bro;

@end

@implementation ViewController

@synthesize bro = _bro;

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

The heap and the stack (same for swift and obj-c)

- The stack: a region of memory which contains storage for local variables. Every thread has a stack. When a function is called, a stack frame is pushed onto the stack, and all function-local data is stored there. When the function returns/ends, all that data is destroyed.
- The Heap: Everything else in memory. You have to request for memory to be allocated on the heap. If you aren't using garbage collection then you have to release that memory as well. This is where you store things that need to outlive the life of the function call. All objects are created in the heap.

Pointers? Huh?

- Variables in your app have four components:
  - Name
  - Location
  - Type
  - Current Value
- Now think of the memory of your iPhone as just a huge pile of bytes, each one next to the other. They all have an address number, like houses do.
- When you allocate a variable, the compiler reserves an exact amount of memory for it. So executing the statement int x = 45 will make the compiler reserve 4 bytes of memory to store the current value of x.

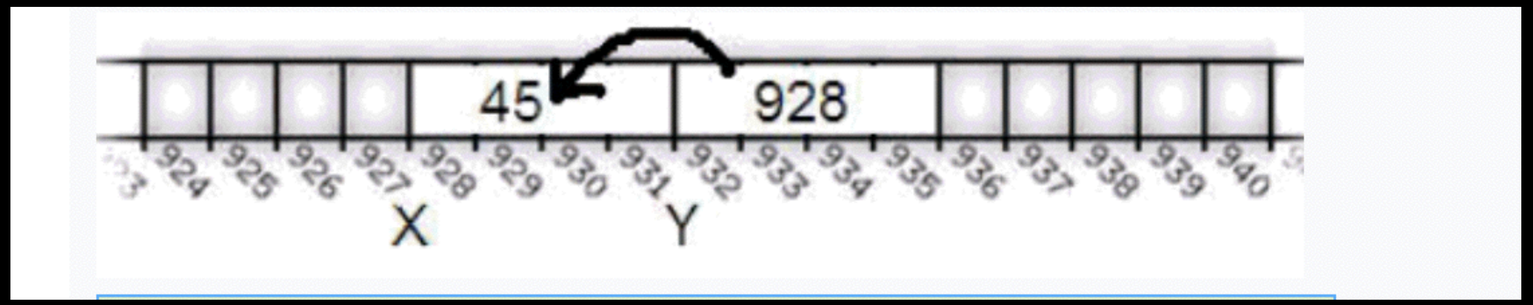


Pointers and their Address

- When you refer to the name of a variable, you are accessing or updating the value it is storing.
- You can use the & (address-of) in front of a variable name to get the variable's location.
- A variable that is able to store the address of another variable is called a pointer.
- Because the pointer variable is 'pointing' to the location of another variable.

Pointer example

- int x = 45;
- int* y = &x;
- So y's data type is int*. it points to another int type variables value.



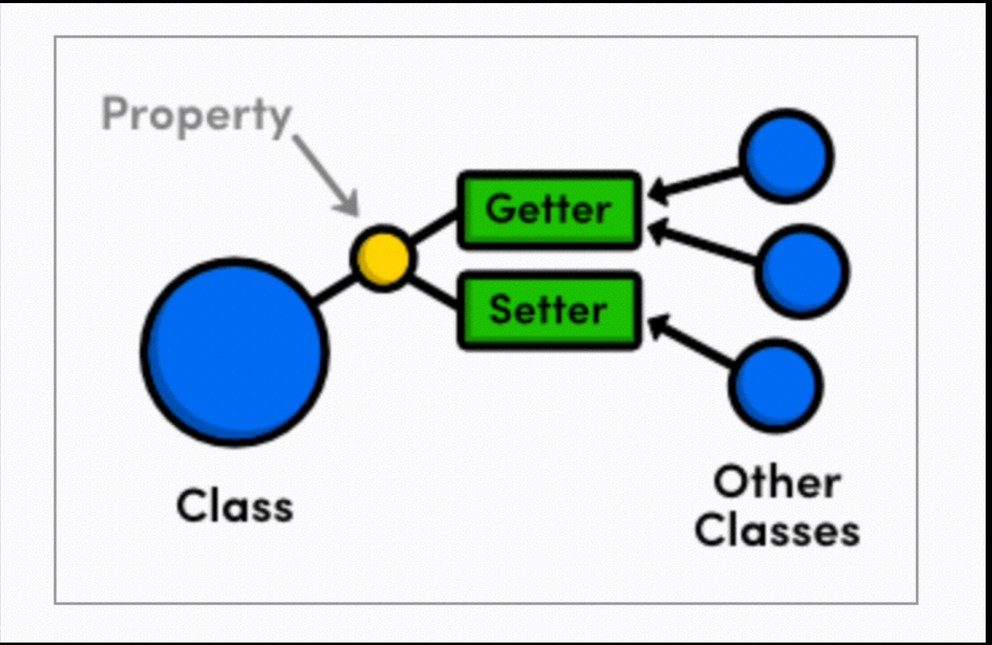
Pointers with Obj-C

- When you instantiate objects in Objective-C, you keep track of them with pointers, which are represented with the star:
- NSString *myString = @“Brad”;
- You only need to use the star when you initially create the pointer. You can access the objects at the address held by the pointer by just using the pointers name:
- NSLog(@“%@“,myString); //prints “Brad”
- NSLog (@“%p”,myString); //prints 0x102d72068
- You don't need to use the stars when creating primitive data types, because they aren't objects so they don't need pointers!
- int myInt = 31;
- If you do need to be passing primitives around as references, then you will use pointers, as shown in the prior slides. And use a * to deference those primitive pointers to access their values.
- int* y =&x;
- NSLog(@“%d”,*y)

Properties

- Properties in Objective-C look a lot of different than in Swift, but serve the exact same purpose.
- Example of a string property: @property (strong,nonatomic) NSString *myName;
- so the syntax is:
- @property (attributes) type *variableName;
- to use a property without self, you must use an underscore. This access's its underlying ivar.
- NSLog(@"%@",_myName)
- NSLog(@"%@",self.myName)

Setters and Getters



- Properties of an object are accessed via their setters and getters.
- self.title = @“Messages”; //using title’s setter
- NSString *myTitle = self.title; //using title’s getter

Setters and Getters

- You can override both the setter and getters of a property.
- So for our title property example:

```
-(void)setTitle:(NSString *)title {
    NSLog(@"someone is setting our title property!");
    _title = title;
}

-(NSString *)title {
    NSLog(@"Someone is getting our title propety!");
    return _title;
}
```

Instance Variables

- All properties are backed by an instance variable.
- The general format of an instance variable is an underscore and then the name of the variable.
- You can completely drop the use of properties and use instance variables only, but then you lose the power of setters and getters, and you lose the ability to call the instance variables with self.<nameOfProperty>. Apple says best practice is to use properties whenever you can, which is almost always.
- To declare an ivar (instance variable), add curly brackets under your @interface block:

```
@interface ViewController ()

{
    //instance variables
    NSString *_myName;
}

//properties
@property (strong,nonatomic) NSString *myName;
@property (strong,nonatomic) NSString *bro;

@end
```

@synthesize

- In older versions of Xcode, you had to manually synthesize your properties to generate the setters and getters. Now that is done for you by default under the hood. However, if you override both the setter and getter, you then have to synthesize the property. Here's what that looks like:

```
@implementation ViewController

@synthesize bro = _bro;
```

Calling methods

- In Objective-C, dot notation only works when calling or setting properties.
- For all other method calls, the syntax is:
- [recievingObject method];
- So for example, telling a tableview to reload its data looks like this:
- [self.tableView reloadData];
- In swift that looks like:
- self.tableView.reloadData()
- Methods with parameters don’t use parens like swift does:

```
UIViewController *detailVC = [[UIViewController alloc] init];
[self presentViewController:detailVC animated:true completion:nil];
```

Defining methods:

- Defining methods in your classes looks drastically different in Objective-C, but achieves the same result:

```
-(NSString *)doSomethingWithNumber:(NSNumber *)number {

    NSString *greeting = [NSString stringWithFormat:@"hello my number is %@",number];

    return greeting;
}
```

- so the format is: -(return type)nameOfMethodWithParameter:(ParameterType)parameterName {

}

Importing files

- You can #import any files or frameworks that your own classes need to know about and use.
- Frameworks are imported with <> brackets, while individuals files are imported with “”
- @import for frameworks
- @class forward class declaration

Everything can be nil!

- You can send messages to nil pointers and things wont crash.
- A zero value is returned in this case.
- This allows you to shorten your code:

```
// For example, this expression...
if (name != nil && [name isEqualToString:@"Steve"]) { ... }

// ...can be simplified to:
if ([name isEqualToString:@"steve"]) { ... }
```

Blocks

- Blocks are to objective-C as closures are to swift.
- The syntax to define a block uses a caret symbol ^:

```
^{
    NSLog(@"This is a block");
}
```

- blocks can be stored as variables, just like closures

```
double (^multiplyTwoValues)(double, double);
```

- here's the corresponding block:

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```