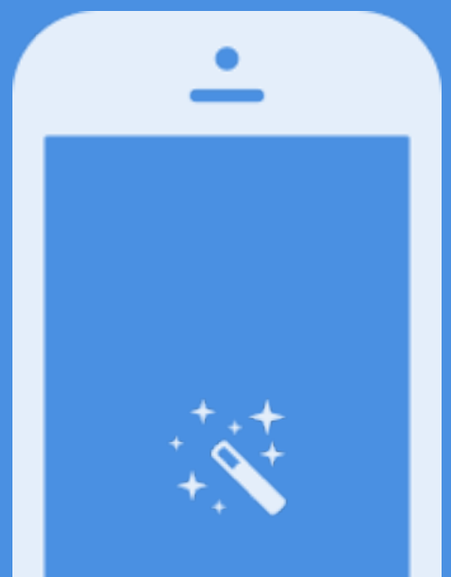
iOS Crash Course

Session Two

Janum Trivedi
iOSCrashCour.se



Session 1 Recap

- Last class, we focused on:
  - Class overview (i.e., what we'll learn)
  - Getting started with Xcode, Interface Builder, iOS Simulator
  - Creating a "Hello World" app

Session 2

- Today, we're focusing on the Objective-C language:
  - Structure, syntax, and semantics
  - Similarities and differences to other common languages
- We will be writing Objective-C code today

Follow Up

- Many of you asked me after class about an Xcode warning: “No matching code signing identify found”
  - This is okay– it’s just Xcode saying you’re not part of the Apple Developer Program
  - Has no effect on your ability to write or run programs

Follow Up

- Also, I mentioned checking out the Apple Student Developer program; this is actually something I have to apply for on my end

- Updates soon

Questions

Follow Up

- These slides for this lecture are online!
- www.github.com/iOSCrashCourse
- **or** www.iOSCrashCour.se
- So meta

What is Objective-C?

- Main programming language used for OS X and iOS development
- General purpose, object-oriented, statically typed

What is Objective-C?

- Objective-C is not simply *inspired by* or *modeled after* the C programming language, it *is* C with object-oriented features

- Thus the “Objective” in “Objective-C”

Why Objective-C?

- Why does Objective-C exist?
- Back in the 80s, C existed, but C is not OOP, and there was a great desire for it to be
- Options: write a new language that incorporates some C features and departs elsewhere
  - ex. C++, C#

Why Objective-C?

- Objective-C took a different approach
- Decided to build *on top of* C
- We call Objective-C a **strict superset** of C
- i.e., you could copy-paste plain C code from the 1970s and it would run and compile

Why is being a “strict superset” important?

- I am **not** saying you should be writing low-level C everywhere in your app

- Instead, it means **basic programming features** like:

  - if statements, for/while loops, switches, breaks, returns, ints, floats, bools, and basic operators

- **are the same as C (and thus, often C++ too)**

Example: Conditionals

- C: `if (true) { //do something }`
- C++: `if (true) { //do something }`
- Obj-C: `if (true) { //do something }`

Example: Loops

- C: (for int i = 0; i < n; i++) { }
- C++: (for int i = 0; i < n; i++) { }
- Obj-C: (for int i = 0; i < n; i++) { }

Example: int variables

- C: `int myVariable = 5;`
- C++: `int myVariable = 5;`
- Obj-C: `int myVariable = 5;`

Why is being a “strict superset” important?

- It also means that much of Objective-C’s syntax is meant to differentiate the C from the Objective-C
- This is why we’ll see square brackets, dots, and @ signs *everywhere*

Questions

Going through an Objective-C Program

- Now that we have some background on Objective-C, let's look at a real Objective-C program
- This time, we'll be focusing on the code, not the Interface Builder

Objects in Objective-C

- Now that we've gone over primitive C variables (and NSLog), let's talk about **object** variables

- Unlike primitive variables (i.e., int, double, char, bool, etc.) where all you need is the **type and the name** of the variable, objects are declared with **explicit pointers**

Objects in Objective-C

- Even strings (i.e., **`NSString`**s) require pointers to declare

- Why? Objects are large and not stored locally on the stack like primitives

Declaring a string

- **NSString*** someString;

Type

Pointer

Variable Name

Declaring other objects

- `NSString* someString;`
- `NSArray* someArray;`
- `NSDate* someDate;`
- `NSNumber* someNumber;`

Initializing a string

- We saw how to declare an **`NSString`**, but how do we assign it a value?

- **`NSString*`** `deviceName = @“iPhone”;`

- Note the type, then the pointer, then the variable name

- We set deviceName equal to a **string literal**

Initializing a string

- `NSString* deviceName = @“iPhone”;`
- Note the type, then the pointer, then the variable name
- We set deviceName equal to a **string literal**
- Using the @ sign with double quotes (“”) is a shorthand way of creating strings in Objective-C

Printing an `NSString`

- **`NSLog`**`(@“I am %i years old”, myAge);`
  works with integer variables only

- We can use the `%@` “placeholder” for **objects**
- **`NSLog`**`(@“My name is %@”, myName);`
- This is called “string interpolation”

Calling Methods on Objects

- Because Objective-C is *Object-Oriented*, we can ***tell*** objects to ***do*** things

- ex. We could *tell* an NSString to return an uppercased version of itself, or *tell* it to replace certain words with another word

  - ex., @“Hello” -> @“HELLO”

Calling Methods on Objects

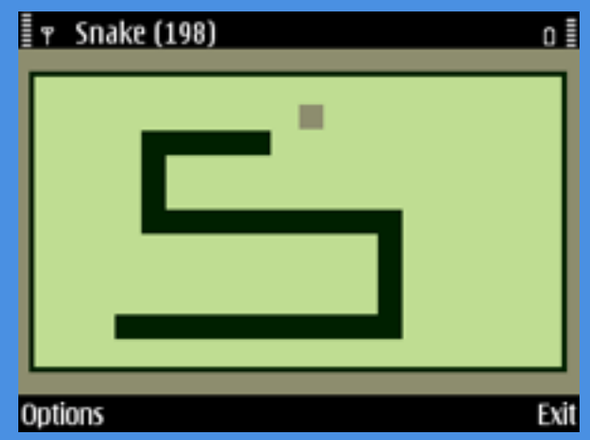
- This means **objects** have **methods** (and primitives don't, by the way)

- A method is roughly the same as a "function", except methods belong to objects

- We could say "*class* `NSString` has an uppercase *method*" that we could call

Calling Methods on Objects

- For example, say we were creating a "snake" game:



- We would have a Snake object (an instance of the `Snake` class)

- Some methods we could call on our snake object could be `pause`, `speedUp`, `slowDown`, etc.

Calling Methods on Objects

- So what is the syntax to call methods in Objective-C? We use our **square brackets**

- Using `NSString` as our example:
  - `NSString* str = @“Hey everyone!”;`
  - `str = [str uppercaseString];`
  - `[str uppercaseString]` returns a new NSString (@“HEY EVERYONE!”)

Calling Methods on Objects

- `NSString* str = @“Hey everyone!”;`
- `str = [str uppercaseString];`

- `[variableName methodName];`
- The general structure is the open bracket, the **name of our variable**, then the **name of the method we are calling**, followed by the closing bracket and semi-colon

Recap so Far

- History and overview of Objective-C
- Basic structure of an Objective-C code file
- C compatibility (if, for/while, int/float, bool, etc.)
- NSLog and string interpolation (%i, %@, etc.)
- String literals with NSString
- Difference in declaring primitive variables vs. objects
- How we can call methods on objects to do or return something

Questions

5 Minute Break

Object Instantiation

- NSStrings *are* objects, and we *did* learn how to declare and instantiate them, but NSString we used the shorthand for string literals (i.e., `foo = @“Bar”;`)

- So how would we create an object variable that *doesn’t* have a shortcut?

Object Instantiation: Arrays

- Refresher: arrays are a data structure in programming that contain an ordered list of elements (like C++ arrays or vectors)

- In an array, we can add elements, remove elements, re-order elements, etc.

- And we can access elements by their index (or which “place” they have in the array

Object Instantiation: Arrays

- C++: `int myArray[100];` (some capacity)
- Objective-C: Enter `NSArray` and its counterpart, `NSMutableArray`
- We'll focus on mostly on `NSMutableArray` because it is a type of `NSArray` that let's us change, or *mutate*, the array whenever we want

Object Instantiation: Arrays

- First, let's show the array declaration:
  - `NSMutableArray* myArray;`
- Remember, this looks similar to how we declared NSString variables:
  - `NSString* myStr;`

Object Instantiation: Arrays

- We *declared* our array (i.e., told the compiler that there *exists* some array variable called myArray, but now we need to initialize it
- Otherwise, myArray points to garbage memory
- ```
  NSMutableArray* myArray =
  [NSMutableArray alloc] init];
  ```
  ```

Object Instantiation: Arrays

- **`NSMutableArray* arr = [[NSMutableArray alloc] init];`**

- This is a *nested* method call. First, [NSMutableArray alloc] returns some allocated memory, then we call init on that memory to set some default values to the class's instance variables

- If this is unclear (i.e., haven't taken 280+ don't worry too much about what's going on under the hood. Experience before knowledge

Object Instantiation: Arrays

- We'll also see custom initializers with Foundation classes with parameters
  - Xcode's autocomplete will show you some of those
  - …alloc] initWithArray
  - …alloc] initWithFile, etc.

Object Instantiation: Generic

- Remember this structure is the same for all objects in Objective-C:
- **`Class* myObject = [[Class alloc] init];`**
- Even literals like NSString's @"" syntax calls alloc/init in the background
- NSString* str = [[NSString alloc] init];
- NSString* str = @"";

NSMutableArray Methods

- Let's look at some of the methods that NSMutableArray has

- To **add** an object:
  - `[myArray addObject:someObject];`
- To **access** an object at some index i:
  - `[myArray objectAtIndex:i];`
- To **remove** an object at some index i:
  - `[myArray removeObjectAtIndex:i];`

Questions

Exercises

1. FizzBuzz

- Quick exercise in our Objective-C command-line project to use NSString and NSMutableArray
- Problem: iterate from 1 to 100. If the number is divisible by 15, print “Fizzbuzz!”. If it is divisible by 3, print “Fizz”. And if it is divisible by 5, print “Buzz”.

1. FizzBuzz

- Problem: iterate from 1 to 100. If the number is divisible by 15, print “Fizzbuzz!”. If it is divisible by 3, print “Fizz”. And if it is divisible by 5, print “Buzz”.

- **Hint:** To iterate, use normal C/C++ loops, and our printing/loggin requires NSLog with placeholder %i

1. FizzBuzz Solution

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        // 1. Plain FizzBuzz
        for (int i = 1; i < 100; i++)
        {
            if (i % 15 == 0) {
                NSLog(@"Fizzbuzz!");
            }
            else if (i % 3 == 0) {
                NSLog(@"Fizz");
            }
            else if (i % 5 == 0) {
                NSLog(@"Buzz");
            }
        }

    }
    return 0;
}
```

2. FizzBuzz w/ arrays

- In our conditionals, instead of only `NSLog`ging "Fizz" or "Buzz", create an NSString with that value, and add it to an array. Finally, iterate through the array and print all the strings out.

- Hint: You'll need to declare an `NSMutableArray` *before* iteration, and remember `[myArray addObject:someObject];`

- Also, use `[myArray count]` to get length

2. FizzBuzz w/ arrays Solution

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        // 2. FizzBuzz with NSString and NSMutableArray
        NSMutableArray* messages = [[NSMutableArray alloc] init];

        for (int i = 1; i < 100; i++)
        {
            if (i % 15 == 0) {
                NSString* fizzbuzz = @"FizzBuzz!";
                [messages addObject:fizzbuzz];
            }
            else if (i % 3 == 0) {
                NSString* fizz = @"Fizz";
                [messages addObject:fizz];
            }
            else if (i % 5 == 0) {
                NSString* buzz = @"Buzz";
                [messages addObject:buzz];
            }
        }

        for (int i = 0; i < [messages count]; i++)
        {
            NSLog(@" %@ ", [messages objectAtIndex:i]);
        }

    }
    return 0;
}
```

3. FizzBuzz w/ arrays and fast enumeration

- There's a nicer way of iterating through all the elements in array. There are still times you would want plain C loops, but Objective-C supports **fast enumeration**

- **FE** is a cleaner, faster, and more concise syntax to go through (or enumerate) a collection

- You don't need to know the length of the array or keep track of the index. And it's fast.

3. FizzBuzz w/ arrays and fast enumeration

- Here's the syntax for fast enumeration over an NSArray/NSMutableArray:

```
for (NSString* str in array)
{
    // Do something
}
```

- "For every NSString object in this array, do something". Rewrite the last loop using fast enumeration.

3. FizzBuzz w/ arrays and fast enumeration Solution

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        // 3. FizzBuzz with NSString, NSMutableArray, and fast enumeration
        NSMutableArray* messages = [[NSMutableArray alloc] init];

        for (int i = 1; i < 100; i++)
        {
            if (i % 15 == 0) {
                NSString* fizzbuzz = @"FizzBuzz!";
                [messages addObject:fizzbuzz];
            }
            else if (i % 3 == 0) {
                NSString* fizz = @"Fizz";
                [messages addObject:fizz];
            }
            else if (i % 5 == 0) {
                NSString* buzz = @"Buzz";
                [messages addObject:buzz];
            }
        }

        for (NSString* str in messages)
        {
            NSLog(@" %@ ", str);
        }

    }
    return 0;
}

```