

iOS DeCal : Lecture 9

Objective C

April 18, 2017

Announcements - 4/18

Make good progress on custom apps!

Remember! Jacobs Design Showcase is on May 4th if you want to show off what you've made.

Jacobs Design Showcase

Thursday, May 4th from 9:30 - 11:30am

If you are interested in presenting your final project, please let us know! (interest form on Piazza)

Jacobs Design Showcase



Thursday, May 4th, 2017 from 9:30 - 11:30am

Link to past showcase information: goo.gl/yaffUW

Fill out form on Piazza if you are interested in presenting

Overview

History and Philosophy

Syntax

Additional Stuff

History and Philosophy



vs



Why Bother?

- Most Companies still have thousands of lines of legacy Objective-C code
- Cocoa and Cocoa Touch Frameworks still in Objective-C (including ability to mix in C/C++)
- Objective-C is still just...better

Object-Oriented + C = Objective C

- Designed by Brad Cox and Tom Love at Stepstone
- Combined Smalltalk philosophy (OO, message passing, etc.) with C backwards compatibility
- Licensed by NeXT, later adopted by Apple

Pointers and Mem. Management

- YES, you must alloc memory for objects!
 - How much? Done for you
- YES, Objective-C has pointers
- no, you don't have to free allocated memory
 - Automatic Reference Counting (ARC)

Why not Garbage Collection?

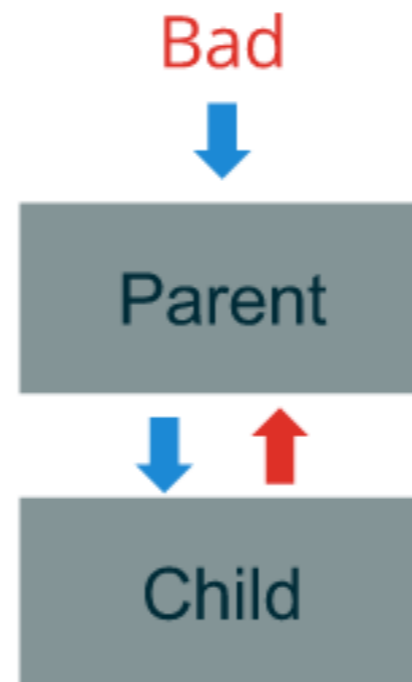
- What's wrong with Garbage Collection (GC)?
 - Stores object-dependency graph at run-time
 - Objects become candidates to free later
(unpredictable, slow, inefficient)
- Improve?
 - Make the compiler analyze code to determine when objects need to be released (freed) or retained (not freed) at compile-time

GC vs ARC

- **Garbage Collection (GC)?**
 - Stores object-dependency graph at run-time
 - Objects become candidates to free later
(unpredictable, slow, inefficient)
- **Automatic Reference Counting (ARC)**
 - Auto-generate release/retain code @ compile-time
 - Objects are freed IMMEDIATELY when not needed
 - Vulnerable to retain cycles

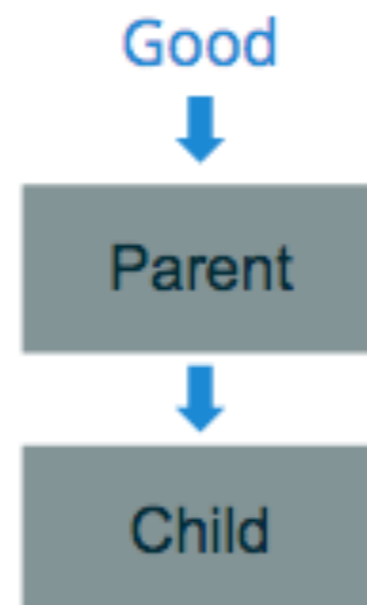
Retain Cycles

```
@class Child;  
@interface Parent : NSObject {  
    Child *child; //instance variables implicitly __strong  
}  
@end  
@interface Child : NSObject {  
    Parent *parent; //also implicitly __strong  
}  
@end
```



Retain Cycles

```
@class Child;  
@interface Parent : NSObject {  
    Child *child; //instance variables implicitly __strong  
}  
@end  
@interface Child : NSObject {  
    __weak Parent *parent; //doesn't increase ref count  
}  
@end
```



Syntax

Head the Implementation!

- Return of the header (.h) and main (.m) file
- **Header** file used to define class
- **Main** file is the implementation
- No more .swift

Header File - Example

```
#import <UIKit/UIKit.h>
@interface ViewController: UIViewController

@property (nonatomic, retain) UILabel *label;
-(NSString *)showString;

@end
```


Header File - General

```
#import <UIKit/UIKit.h>
@interface <ClassName>: <SuperClassName>

    //all properties and methods go here

@end
```

The Implementation File - Example

```
#import "ViewController.h"
@interface ViewController()
@end
@implementation ViewController
-(void)viewDidLoad {
    [super viewDidLoad];
    self.label = [[UILabel alloc] init];
}
-(NSString *)showString {
    return @"Obj-C >= Swift";
}
@end
```

The Implementation File - General

```
#import "ClassName.h"  
@interface <ClassName>()  
@end  
  
@implementation <ClassName>  
    //implement methods here  
@end
```

Properties

- @property - tag for creating instance variables
- **Automatically** generates getter/setter methods
 - i.e. [self setName: Steve]
- Several **Attributes**

Attributes

- atomic - less prone to multithreading (concurrency) errors
- nonatomic - prone to multithreading errors
- strong - want to own the object
- weak - you don't want control over object's lifetime
- readonly - doesn't create setter method
- copy - used for primitive data-types
- retain - pre-iOS 5 version of strong
- assign - pre-iOS 5 version of weak

Bracket Notation

- Methods and properties
 - Can also use dot notation but not recommended

Bracket Notation - Instantiating

```
UILabel *label = [[UILabel alloc] init];
```

Getters and Setters

- Compiler automatically synthesizes getters and setters

```
@property (nonatomic, strong) UILabel *ageLabel;  
...  
[self ageLabel]; //automatically generated  
[self setAgeLabel:someLabel]; //automatically generated
```


Dot Notation

- More similar to Swift
 - Less confusion when using both Swift / Obj-C
- Some Obj-C programmers still prefer getters/setters
 - Easier to distinguish between getter and setter
 - Able to use CMD + f
- Use when dealing with properties
 - Keeps it clean and concise

Dot vs. Bracket

```
@property (nonatomic, strong) UILabel *ageLabel;  
.....  
self.ageLabel = someLabel; //dot notation  
[self setAgeLabel:someLabel]; //bracket with setter
```

Functions

- **Plus (+)**
 - class method
- **Minus (-)**
 - instance method

Creating Functions

```
-(NSString *)showString {  
    return @"Obj-C >= Swift";  
}
```

```
-(NSString *)showString:(NSString *)extra {  
    return extra;  
}
```

```
-(NSString *)showString:(NSString *)extra withString:  
(NSString *)var {  
    return [NSString stringWithFormat:@"%@ %@", extra,  
var];  
}
```

Calling Functions

```
[self showString];
```

```
[self showString:@"Obj-C"];
```

```
[self showString:@"Obj-C" withString:@"Functions"];
```

```
//general method of calling  
[<Class> <Function in class>];
```

Strings

- There is no built-in class type String, must use NSString
- To use NSString, must prepend every string with @
- Printing to log is done with NSLog

Strings - Examples

```
//creating a string  
NSString* str = @"Hello world!";
```

```
//combining strings  
NSString* str = [NSString stringWithFormat:@"Hello %@",  
@"World!"];
```

```
//printing to log  
NSLog(@"Printing %@, @to log!");
```

Arrays

- There is no built-in class type Array
 - Use NSArray or NSMutableArray

```
//immutable  
NSArray *staff = @[@"Sony", @"Gene", @"Shawn", @"Allie",  
@"Lucy", @"Helena", @"Kyle"];
```

```
NSArray *staff1 = [NSArray arrayWithObjects:@"Sony",  
@"Gene", @"Shawn", @"Allie", @"Lucy", @"Helena",  
@"Kyle", nil];
```

```
//mutable  
NSMutableArray *hello = [[NSMutableArray alloc] init];  
[hello addObject:@"Hello"];
```


Check-In

UIView

```
//initWithFrame and CGRectMake  
UIView *testView = [[UIView alloc] initWithFrame:  
                    CGRectMake(0, 0, 200, 200)];
```

```
//addSubview is crucial!  
[self.view addSubview:testView];
```

UIView - Another example

```
//getting the size of our screen
CGSize *screen = [UIScreen mainScreen].bounds.size;

//let's apply this now (look at CGRectMake)
UIView *testView = [[UIView alloc] initWithFrame:
    CGRectMake(screen.width/2-100,
               screen.height/2-100, 200, 200)];
//addSubview is crucial!
[self.view addSubview:testView];
```

Initializers

```
#import "CustomLabel.h"
@implementation CustomLabel
• (id) custInit {

    self = [super initWithFrame:CGRect];
    //Design your UILabel here
    return self;
}
@end

CustomLabel *l = [[CustomLabel alloc] custInit];
```

Additional Notes

Objective-C + Swift

- Use a **bridging header**
- Add Obj-C Class (.m file)
- Add Bridging Header
 - Click 'Yes' when asked to configure bridging header
- Add Obj-C class header (.h file)
- Fill out Obj-C code (both .m and .h)
- Import class in Bridging Header

Using Swift with Objective-C

- Add Swift class (.swift file)
- Import Swift file as:
 - `#import "NameOfFile-Swift.h"`

Demo