

# Bootstrapping iOS Application Development

Chris Zelenak

12/02/2013

- 1 Intro to class
- 2 Lab 1
- 3 Objective C
- 4 iPhone App Basics: UIViewControllers

- 1 Intro to class
- 2 Lab 1
- 3 Objective C
- 4 iPhone App Basics: UIViewControllers

Welcome to the 5-Day Bootstrapping iOS Application Development class. Over the course of these next 5 days, we're going to be reviewing Objective C, UIKit, Core Animation, Core Data, and a host of other technologies you may only know by name.

## 1 Lab 1

# Create a new iPhone Project

Create a new Single View based application in XCode called Lab1

File > New > Project

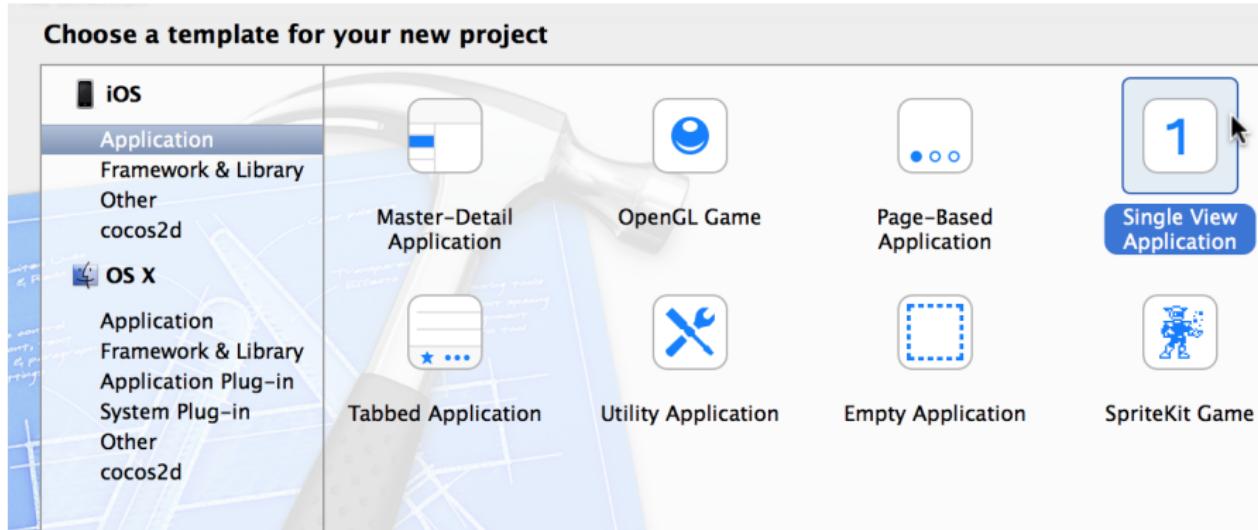


Figure : New Projects

Open Main.storyboard in Interface Builder



Figure : Main Storyboard

Add a button

Add a button with the text “Hello iPhone”

### Step 1



**Object** – Provides a template for objects and controllers not directly available in Interface Builder.

#### Label

**Label** – A variably sized amount of static text.

#### Button

**Button** – Intercepts touch events and sends an action message to a target object when it's tapped.



1

2

**Segmented Control** – Displays multiple segments, each of which functions as a discrete button.

#### Text

**Text Field** – Displays editable text and sends an action message to a target object when Return is tapped.

### Step 2



Button

Figure : Drag Button to Storyboard

Connect the button to the view controller

## Add an IBAction selector to the view controller

```
1 // In the ViewController.h file  
2 -(IBAction) helloWorldTapped:(id)sender;
```

## Make it open an alert view

```
1 // In the ViewController.m file
2 -(IBAction) helloWorldTapped:(id)sender {
3     [[[UIAlertView alloc] initWithTitle:@"Yay it worked"
4                                     message:@"Hello World"
5                                     delegate:nil
6                                     cancelButtonTitle:@"Dismiss"
7                                     otherButtonTitles: nil] show];
8 }
```

Connect the IBAction to the Button's Touch Up Inside action.

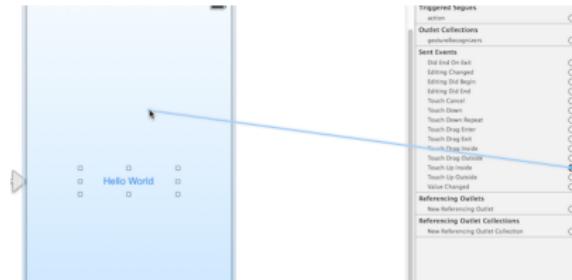


Figure : Connect Touch Up Inside event to IBAction

Build and run in simulator

Run the simulator with **⌘+R**, the **Product > Run** menu, or the Play button

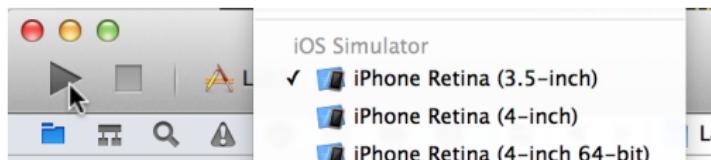


Figure : Launch Simulator

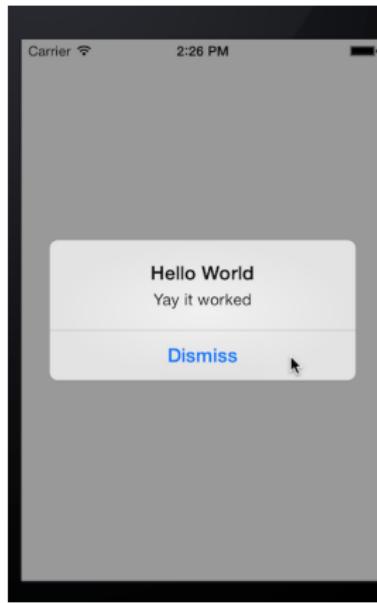


Figure : It Worked!

## 1 Objective C

# Overview

Objective C is a superset of C. It tacks on to C a message-passing object model similar to Smalltalk's (see Ruby), with support for reflection and limited runtime dynamic programming.

Objective C allows you to use weak and dynamically typed language features along with the static typing of C; it also allows you to take advantage of runtime method resolution, so that your object's implementation of certain messages may be determined at call-time rather than at compile-time.

The language is still physically identical to C in many ways, in that the bulk of your development will be alternating between header files (.h files) and implementation files (.m files). You will also be doing a lot of memory management, which may fill you with either fear or joy, depending on your experience with garbage collected languages.

You will also be dealing a lot with many other C-family familiars, such as enumerated integer constants (enums), structs, pass-by-reference v. pass-by-value semantics and pointers. You don't need to have a very strong grasp on these things immediately, but the more iPhone programming and Objective C programming you do, the more your skills will benefit from a firm grasp of C.

## Review of standard C elements

Variable declaration in C follows the form TYPE\_DESCRIPTION NAME,  
eg:

```
1 int i = -1; // just an integer
2 unsigned int k = 0; // an unsigned integer
3 const unsigned int j = 0; // a constant unsigned integer, no reassignment
4 const char c = 'c'; // a constant char
5 const char * s = "a constant string"; // a read only pointer to a char array
```

Figure : 001\_variables.c

These are items you'll commonly encounter while doing iPhone programming whose roots belong in the C family:

```
1  typedef enum {
2      NONE = 0,
3      SOME,
4      ALL
5  } HowMany;
6
7  HowMany i = SOME; // Use of enumerated constant
```

Figure : 002\_enumerations.c

Note the `typedef` in this example, which is creating a type alias to be used later.

```
1 // Declaring a point in space type
2 typedef struct {
3     float x;
4     float y;
5 } APointInSpace;
6
7 APointInSpace point;
8 point.x = 20.0;
9 point.y = 410.0;
10
11 // C99 style struct instantiation
12 APointInSpace point2 = { .x= 21.0, .y=32.0 };
```

Figure : 003\_typedef.c

```
1 #include <stddef.h>
2
3 // a constant byte array
4 char * constantCharArray = "Or in other words, a string";
5
6 // C99 style array instantiation
7 int aListOfNumbers[5] = { 1, 2, 3, 4, 5 };
8 int aListOfNumbers2[] = { 1, 2, 3, 4, 5 };
9
10 // C99 Variable length array declaration
11 int someLength = 5;
12 float aListOfFloats[someLength];
13
14 // pointer declaration to NULL
15 int * someIntegerPtr = NULL;
16 int someInteger = 5;
17
18 // address dereference
19 someIntegerPtr = &someInteger;
```

Figure : 004\_arrays\_and\_pointers.c

```
1 float aSimpleAdditionMethod(float a, float b){  
2     return a + b;  
3 }  
4  
5 float aSimpleSubtractionMethod(float a, float b){  
6     return a - b;  
7 }  
8  
9 int aFloatAdditionThatReturnsAnInt(float a, float b){  
10    return (int) (a + b);  
11 }  
12  
13 float (*anFnPtr)(float, float) = aSimpleAdditionMethod;  
14 anFnPtr(1, 1); // returns 2  
15 anFnPtr = aSimpleSubtractionMethod;  
16 anFnPtr(1, 1); // returns 0
```

Figure : 005\_functions.c

```
1 #include <stdio.h>
2 #define FOO 1
3 #define IS_ZERO(n) (n == 0)
4
5 int main(){
6     printf("FOO is %i\n", FOO);
7     if(IS_ZERO(0)) {
8         printf("This code is very useful.\n");
9     }
10 }
```

Figure : 006\_preprocessor.c

If you never learned C or it's been a long time, a fantastic book for you is "The C Programming Language". It doesn't cover many of the newer items in C99, but it is a thorough reference to the language fundamentals.

If you'd like to read more about the new features introduced by C99, read more at [C99Changes](#).

# Syntax

```
1 NSString * anExampleString = @"This is a pointer to an NSString object";
2 int strLen = [anExampleString length];
3
4 NSString * aSecondString = [anExampleString
5     stringByAppendingString:@" that had a second string appended to it"];
6
7 if([anExampleString
8     compare:@"this is a pointer to an nsstring object"
9     options:NSCaseInsensitiveCompare] == NSOrderedSame){
10    NSLog(@"The string %@", anExampleString);
11 }
```

Figure : 007\_basic\_objc\_example.m

# Calling functions

Given some Objective C object, like:

```
1 NSString * anObject = @"TheObject";
```

you can invoke functionality on that object via the form:

```
1 [anObject theMethodName];
```

Method names in Objective C follow the form

initialMethodNameAndArgument : theSecondArgument : theThirdArgument :

.

When you see references to “calling a method” or “calling a function” on an object in Objective C, it’s best to consider these synonymous with “dispatching a message”. Objective C objects respond to **messages**, and as you work more with Objective C, it will be more beneficial for you to perceive this as such.

nil, NULL

An object in Objective C may be nil; nil is a “special” object which may have any message at all sent to it, to which it will respond with a nil object.

```
1 NSString * anObject = nil;
2 if([anObject whoaThisProbablyDoesntExist] == nil){
3     NSLog(@"Ah well");
4 }
```

Figure : 008\_calling\_methods\_on\_nil.m

nil is the appropriate null object when dealing with pointers to Objective C classes; NULL, on the other hand, is the appropriate null value to use when dealing with pointers to all other types.

## Initializing objects

To initialize an Objective C object, you allocate its memory first with the `alloc` method, then initialize it with an `init` constructor.

```
1  [[SomeClass alloc] init]
```

Objects may have several different constructors. Each may require different arguments:

```
1  [[NSString alloc] initWithString:@"A test string"];
2  [[UIView alloc] initWithFrame:CGRectMake(0., 0., 20., 20.)];
```

# Basic Types

There are a host of useful classes in the Cocoa Touch framework that provide special functionality to you. Some of the more common objects you'll use in your programming are NSArray, NSDictionary, NSNumber and NSString. Each of these objects have mutable versions that allow you to modify them after instantiation.

The special type `id` refers to any valid `NSObject` or its subclass. You can use the `id` type when you're not sure what sort of object you may be given.

NSString is an enhanced, unicode aware string class that goes far beyond the simple byte-array behavior of C's byte-array strings.

```
1 NSString * s = @"This is an NSString";
2 NSMutableString * so = [[NSMutableString alloc]
3                           initWithString:@"This is a mutable string"];
4
5 [so appendString:s];
6
7 NSLog(@"The mutable string is %@", so);
```

Figure : 008\_nsstring\_example.m

NSNumber is a simple abstract wrapper around numeric values which allows you to automatically convert its held value to the appropriate form, as well as having an object-like representations of a numeric value.

```
1  NSNumber * n = @(20.0f);
2  double d = [n doubleValue];
3  int i = [n intValue];
4  NSNumber * b = @(YES);
5  NSLog(@"%@", @"Number is %@" , n, double value is %f, int value is %i", n, d, i);
6  NSLog(@"%@", @"Boolean number is %@" , b, bool value is %i", b, [b boolValue]);
```

Figure : 009\_nsnumber\_example.m

`NSDictionary` is a generic dictionary object that you can use to hold key/value associations. A key can be any `NSObject` that responds to `isEqual:` and `NSCopying`; in most cases, your keys will be either `NSString` or `NSNumber` objects.

```
1 NSDictionary * a = @{
2     @"Foo": @"The Foo string",
3     @"Bar": @"The Bar string"
4 };
5
6 NSMutableDictionary * b = [[NSMutableDictionary alloc] init];
7 [b setObject:@"Another string" forKey:@(20)];
8
9 NSLog(@"a's value for Foo is %@", [a objectForKey:@"Foo"],
10       @" and b's value for 20 is %@", [b objectForKey:@(20)]);
11
12 for(id key in a){
13     NSLog(@"The value for %@ is %@", key, [a objectForKey:key]);
14 }
15 }
```

Figure : 010\_nsdictionary\_example.m

NSArray is a simple way to collect NSObject inheriting classes into a sequential list. NSArray automatically retains each object added to it, and releases each object it holds once its own retain count has reached 0.

```
1 NSArray * a = @[@"Foo", @"bar", @"baz"];
2 NSMutableArray * b = [[NSMutableArray alloc] init];
3 [b addObject:@"Not foo"];
4 NSLog(@"The contents of a are %@", a, b);
5
6 for(id obj in a){
7     NSLog(@"The array contains %@", obj);
8 }
```

Figure : 011\_nsarray\_example.m

NSArray, NSDictionary, NSString and NSNumber may all be represented by the contents of a property list file ( Plist ). You can easily rebuild an NSArray or NSDictionary full of the property list contents by using the initialization method `initWithContentsOfFile:`

Key	Type	Value
▼ Root	Dictionary	(3 items)
Example Key	Number	2000
Some Other Key	String	I'm a value!
▼ A subarray of values	Array	(2 items)
Item 0	Number	1
Item 1	Boolean	<input type="checkbox"/>

Figure : Sample Property List

```

1 NSDictionary * example = [[NSDictionary alloc]
2                             initWithContentsOfFile:@"/path/to/sample.plist"]
3 NSLog(@"The subarray contents are %@", 
4       [example objectForKey:@"A subarray of values"]);

```

Figure : 012\_plist\_example.m

Apple also provides JSON serialize/deserialize capability in the NSJSONSerialization class. This class can decode NSString, NSNumber, NSArray and NSDictionary values much like property lists.

```
1  NSDictionary * dict = @ { @"foo": @"bar" };
2  NSData * jsonData = [NSJSONSerialization dataWithJSONObject:dict
3                                         options:0
4                                         error:nil];
5  if([jsonData writeToFile:@"./test.json" atomically:YES]){
6      NSData * inputJsonData = [NSData dataWithContentsOfFile:@"./test.json"];
7      id jsonDict = [NSJSONSerialization JSONObjectWithData:inputJsonData
8                                         options:0
9                                         error:&error];
10     NSLog(@"Foo's value is %@", [jsonDict objectForKey:@"foo"]);
11 }
```

Figure : 013\_json\_example.m

# Classes

Classes in Objective C are similar to their C++ cousins in that they come in two parts: interface declaration and implementation. The interface declaration portion of a class follows the form:

```
1 #import <Foundation/Foundation.h>
2 @interface MyNewClass : NSObject {
3     int privateVariable1;
4     NSString * privateVariable2;
5 }
6
7 +(int) someClassMethod;
8 -(void) doSomething;
9 -(NSString *) giveMeAStringOfLength:(int)length
10                      randomizeContents:(BOOL)randomize;
11 @end
```

Figure : 014\_objects.m part 1

The implementation of this class follows the form:

```
1 #import "MyNewClass.h"
2
3 @implementation MyNewClass
4
5 -(id) init {
6     self = [super init];
7     if(self){
8         privateVariable1 = 20;
9         privateVariable2 = [[NSString alloc] initWithString:@"Test"];
10    }
11    return self;
12 }
13
14 +(int) someClassMethod {
15     return 0;
16 }
```

Figure : 014\_objects.m part 2

```
1      -(void) doSomething {
2          NSLog(@"Something");
3      }
4
5      -(NSString *) giveMeAStringOfLength:(int)length
6                      randomizeContents:(BOOL)randomize {
7          return @"TODO: Make work";
8      }
9
10     @end
```

Figure : 014\_objects.m part 3

# Properties

Much of the interaction with Objective C objects equates to the usual getter/setter functionality commonly associated with Object Oriented Programming. To make it easier for programmers to declare this functionality, properties were introduced as a way to eschew all the boilerplate code and provide a common framework upon which to enhance Objective C objects.

```
1 @interface MyObject : NSObject {
2
3 }
4 @property(strong) NSString * instanceVariable;
5 @property float numericVariable;
6 @end
7
8 @implementation MyObject
9
10 @end
```

Figure : 015\_properties.m part 1

```
1 MyObject * mo = [[MyObject alloc] init];
2
3 // Bracket Syntax
4 [mo setInstanceVariable:@"Test"];
5 [mo setNumericVariable:20.0f];
6 NSLog(@"The values I set were %@", [
7     [mo instanceVariable],
8     [mo numericVariable]]);
9
10 // Dot Syntax
11 mo.instanceVariable = @"Test 2";
12 mo.numericVariable = 25.0f;
13 NSLog(@"The values I set were %@", [
14     mo.instanceVariable,
15     mo.numericVariable]);
```

Figure : 015\_properties.m part 2

You can specify the memory management semantics (strong, weak, copy), atomicity (locking behavior), and access level in the property declaration. You can also specify your own implementation of the getter and setter methods.

```
1 @property(nonatomic) NSString * privateName;
2 @property(weak, nonatomic, setter=myAgeSetter:) NSNumber * age;
```

Figure : 016\_advanced\_properties.m

Properties do not merely provide getter/setter functionality. The use of properties also implicitly adds Key-Value-Observing functionality to your class, letting you automatically monitor classes for change events and performing specific code in such cases. Read more in the SDK Documentation on properties and Key Value Observing.

# Selectors

Selectors in Objective C are a way to indicate a message as a variable. They allow you to dynamically send a message to an object, as well as to query an object to see if it responds to a given message.

```
1 SEL aSelector = @selector(length);
2 NSString * foo = @"Foo";
3 if([foo respondsToSelector:aSelector]){
4     NSLog(@"The length of foo is %i", [foo performSelector:aSelector]);
5 }
```

Figure : 017\_selectors.m

Selectors can be built from strings, and can refer to any Objective C message that is forwarded to an object.

```
1  NSString * anAllCapsString = @"THIS STRING SHOULD BE LOWERCASE";
2  NSString * theMessageToSend = @"lowercaseString";
3  SEL aSelector = NSSelectorFromString(theMessageToSend);
4  if([anAllCapsString respondsToSelector:aSelector]){
5      NSLog(@"All caps string (%@) converted: %@", 
6            anAllCapsString,
7            [anAllCapsString performSelector:aSelector]);
8 }
```

Figure : 018\_advanced\_selectors.m

# Protocols

Objective C provides only single-inheritance for its objects; to allow for situations where a class may provide functionality outside of its inheritance chain, the language provides Protocols, which are roughly analogous to interfaces in Java and C#. There are informal protocols (only referred to in documentation) and formal protocols, which are compiler checked. The majority of protocol usage in iOS programming is formal protocols.

```
1 @protocol CameraDevice
2
3 -(NSString *) manufacturerName;
4
5 @optional
6
7 -(int) flashStrength;
8
9 @end
10
11 @interface AnExpensiveCanonCamera : NSObject<CameraDevice>
12 @end
13
14 @interface TheCheapestCameraEver : NSObject<CameraDevice>
15 @end
```

Figure : 019\_protocols.m part 1

```
1 NSArray * cameras = @[
2     [[AnExpensiveCanonCamera alloc] init],
3     [[TheCheapestCameraEver alloc] init]
4 ];
5 for(NSObject<CameraDevice> * camera in cameras){
6     if([camera respondsToSelector:@selector(flashStrength)]){
7         NSLog(@"%@", camera has flash strength %i",
8                 [camera manufacturerName],
9                 [camera flashStrength]);
10    } else {
11        NSLog(@"%@", camera has no flash", [camera manufacturerName]);
12    }
13 }
```

Figure : 019\_protocols.m part 2

# Categories

Categories allow you to mix in new code to existing classes without having to change the original source code for those classes. Examples of categories include automatically adding special serialization rules to NSObject.

You “can’t” add instance variables to a class with categories; they are purely for adding new methods to a class, but cannot change the memory layout of a class after the fact.

```
1     @interface NSDictionary(AsJson)
2
3     -(NSString *) asJson;
4
5     @end
6
7     @implementation NSDictionary(AsJson)
8
9     -(NSString *) asJson {
10         NSData * jsonData = [NSJSONSerialization dataWithJSONObject:self
11                                         options:0
12                                         error:nil];
13         return [[NSString alloc] initWithData:jsonData
14                                         encoding:NSUTF8StringEncoding];
15     }
16
17     @end
```

Figure : 020\_categories.m part 1

```
1 #import "NSDictionary+AsJson.h"
2
3 void main(){
4     NSLog(@"%@", [@[ @{@"A string": @"String!", @"A number": @(20),
5                         @"An Array": @[@(1), @(2), @(3)]} asJson]);
6 }
7
8 }
```

Figure : 020\_categories.m part 2

# Memory management

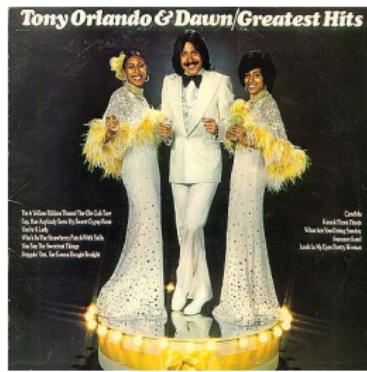


Figure : Memory Management circa 1973

C based languages have traditionally used `malloc` and `free` as the means by which memory was allocated on demand. This code looked something like:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int * anInteger = malloc(sizeof(int));
6     *anInteger = 10;
7     printf("Integer value is %i", *anInteger);
8     free(anInteger);
9     return 0;
10 }
```

`malloc` and `free` allocated memory on the heap; it was the programmer's responsibility to indicate when a particular item was ready to be released back to the operating system.

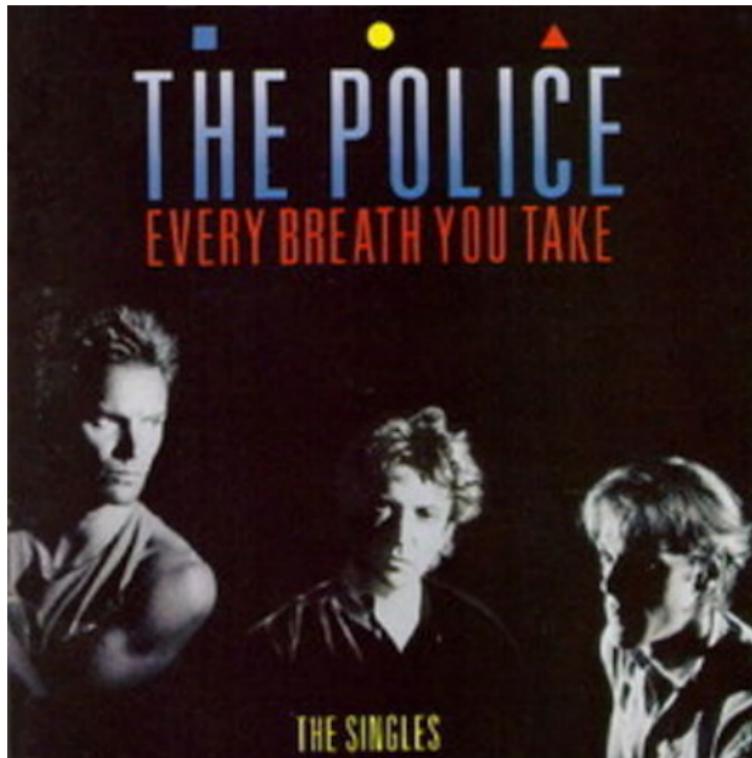


Figure : Memory Management circa 1983

Objective C used an “easier” means of managing the memory for its objects: reference counting.

Reference counting allowed Objective C programmers to write code that looked like this:

```
1  NSString * c = [[NSString alloc] initWithString:@"This is a string"];
2  NSLog(@"String is %@", c);
3  [someOtherObject setString:c];
4  [c release];
```

Each object would respond to two methods: `retain` and `release`. `Retain` would increment an object's retain count, and `release` would decrement. When an object's retain count reached 0, it was considered “deallocated”, and its memory on the heap would be available for use by other objects.

Objects could also be flagged as `autorelease`, which would send a `release` to an object at some point in the future. Typically this meant after one main `NSRunLoop` cycle (one UI loop).



Figure : Memory Management circa 2006 - Apple adds Garbage Collection to Objective C



**Figure :** Memory Management circa 2008 - The iOS SDK 2.0 and App Store launch. The iPhone does not support Garbage Collection due to performance concerns.



Figure : Memory Management circa 2011 - ARC

Apple introduces a new tool called ARC (*Automatic Reference Counting*). Improvements in static code analysis allow Apple to detect and automatically insert retain/release calls for most code.  
In other words, *most of the basic memory management is now done for you.*

What ARC *will do* for you:

- Let you avoid calling `retain` or `release` on Objective C objects

What ARC **will not do** for you:

- Prevent cyclic memory ownership
- Allow you to ignore `malloc` or `free` when using C-based apis
- Allow you to totally ignore memory management

```
1 NSString * s = [[NSString alloc] initWithString:@"Some string"];  
2 NSLog(@"My string is %@", s);  
3 // That's it.
```

```
1  NSString * s = [[NSString alloc] initWithString:@"Some string"];
2  [someOtherObject setString:s];
3
4  // and in the someOtherObject's implementation
5
6  @interface SomeOtherObject
7
8  @property(strong,nonatomic) NSString * string;
9
10 @end
```

Usage of ARC is by default on all new iOS projects. You can choose to disable it and manually manage memory. You can also still manually manage memory created through `malloc` and `free` if necessary.

More learning

Objective C is a deep language. There are lots of resources on the web and your computer where you can learn more. The best is easily the XCode documentation, but there are blogs out there that can provide invaluable help:

- Mike Ash's NSBlog
- objc.io
- NSHipster
- Apple Developer Forums

## 1 iPhone App Basics: UIViewControllers

MVC, as applied to an app

Apple strongly encourages you to adhere to Model View Controller pattern



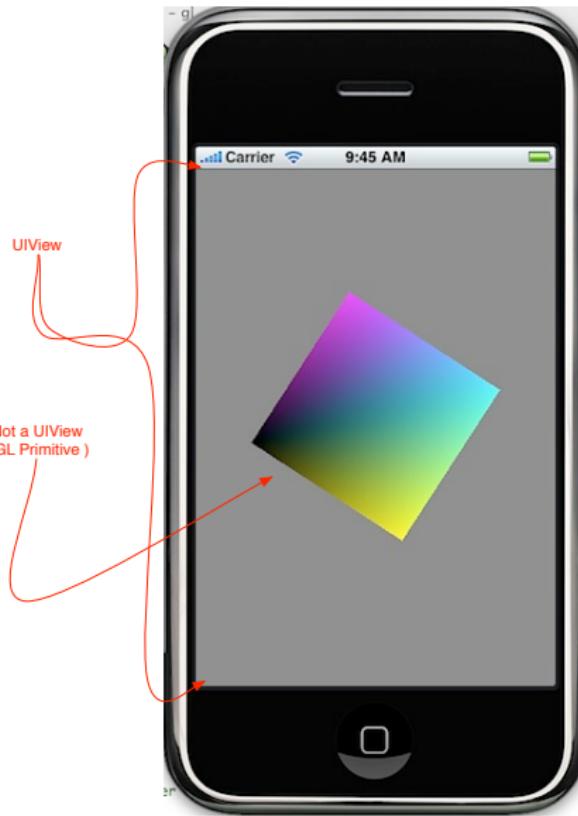
Figure : Model, View and Controller

Application  
⇒ *Window* ⇒ *ViewController(s)* ⇒ *View(s) + Models*

On iOS, a UIApplication ( your app ) typically has one UIWindow, which is the primary UIView upon which everything else displays. A UIView is “just” a rectangle upon which you draw things.

Almost everything visual is a UIView (except the things that aren't).





UIViewControllers are objects that manage what the views on screen are currently doing; typically where you write event handling code, user interaction code, and interact with your models.

There is no “official” model object in this pattern. Your business objects are the models, however you decide to implement them.

# Storyboards

Storyboards let you visualize the interaction between UIViewControllers. You can design both the UIView layout and the interaction between UIViewControllers using the Storyboard designer.

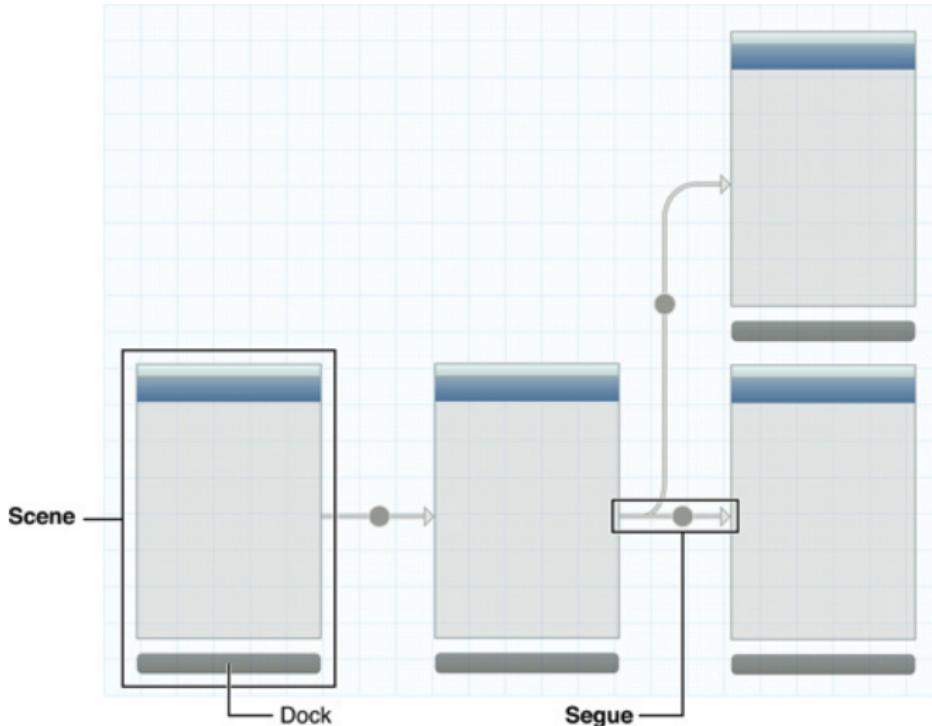


Figure : The Storyboard designer

Storyboards are comprised of **scenes** and **segues**. A **scene** represents a UIViewController's presentation. A **segue** is the transition between **scenes**, and carry information about the transition origin and destination, as well as the nature of the transition.

# Lab 2

# Create a new “Single View” project

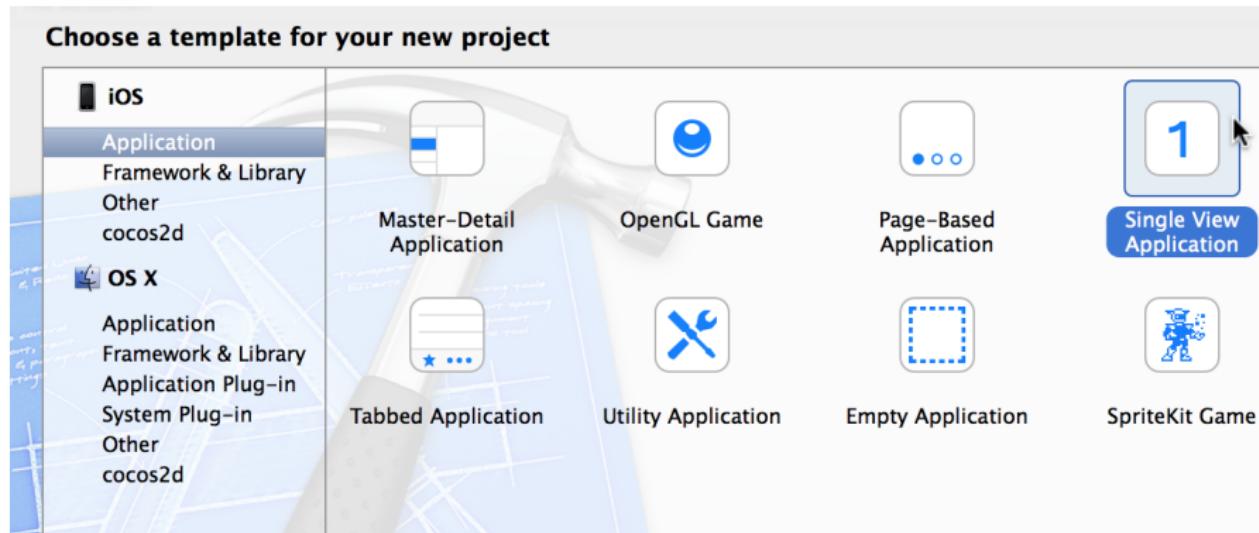


Figure : Create a new project

Add a new NSObject subclass of “UIViewController” to the project named “DetailViewController”

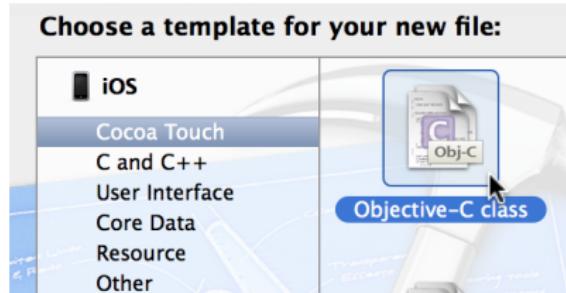


Figure : New Objective C class



Figure : Subclassing UIViewController

Add a “UIViewController” object from the Object library on to your storyboard.

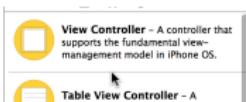


Figure : The UIViewController object in the Library

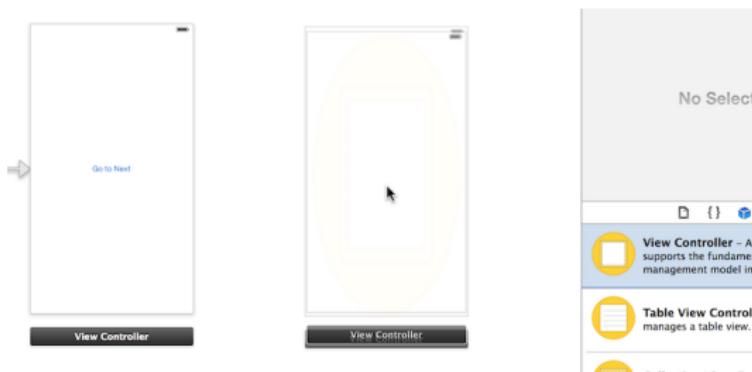


Figure : Drop UIViewController on Storyboard

Set the subclass of the UIViewController to “DetailViewController”.

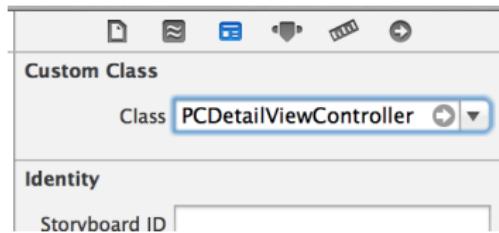


Figure : Setting the UIViewController subclass

Double click on the first UIViewController in the storyboard and add a button with the text “Show Detail” to its scene.

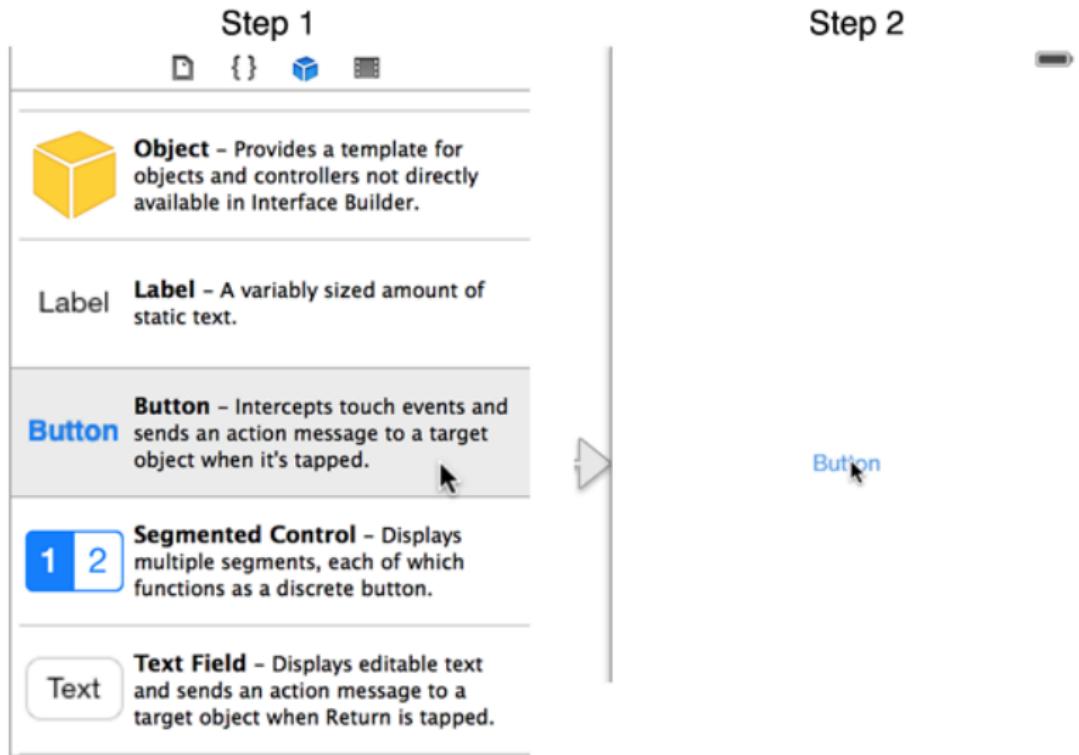


Figure : Drag a button to the scene

Select the button, and view its outlets in the inspector pane. Drag from the action outlet under Triggered Segues to the “Detail View Controller” scene in your storyboard. Choose “Modal” when asked what sort of segue style will be used.

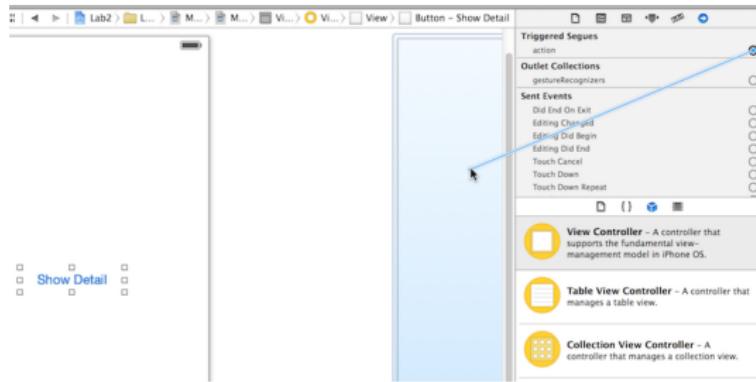


Figure : Connect the action to the Detail View Controller

Double click on the Detail View Controller and add a button to its scene with the text “Close”.

Add the following code to DetailViewController.h

```
1 -(IBAction) dismissDetailView:(id) sender;
```

Add the following code to DetailViewController.m

```
1 -(IBAction) dismissDetailView:(id) sender {  
2     [self dismissViewControllerAnimated:YES completion:nil];  
3 }
```

In your Storyboard, select the “Close” button you added to the Detail View Controller, and connect its “Touch Up Inside” outlet to the Detail View Controller. Select the “dismissDetailView:” selector when prompted which selector to connect to.

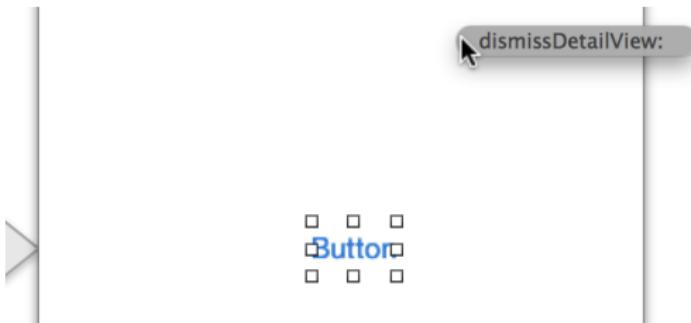


Figure : Connect the action to the dismiss selector

Run the project!



Figure : Yay, it works!

*One more thing:* Add another button on the Detail View controller that opens another View Controller as a modal, and this new view controller has a button on it that when tapped opens an alert box that says “Hello World”.

# UINavigationController

`UINavigationController` is a stack based manager of view controllers that the user can navigate through. The tiny left-facing back arrow present in most iOS apps is its most obvious characteristic.

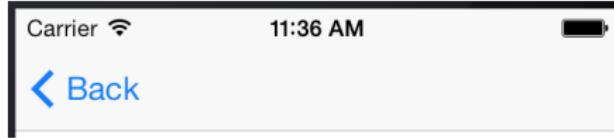
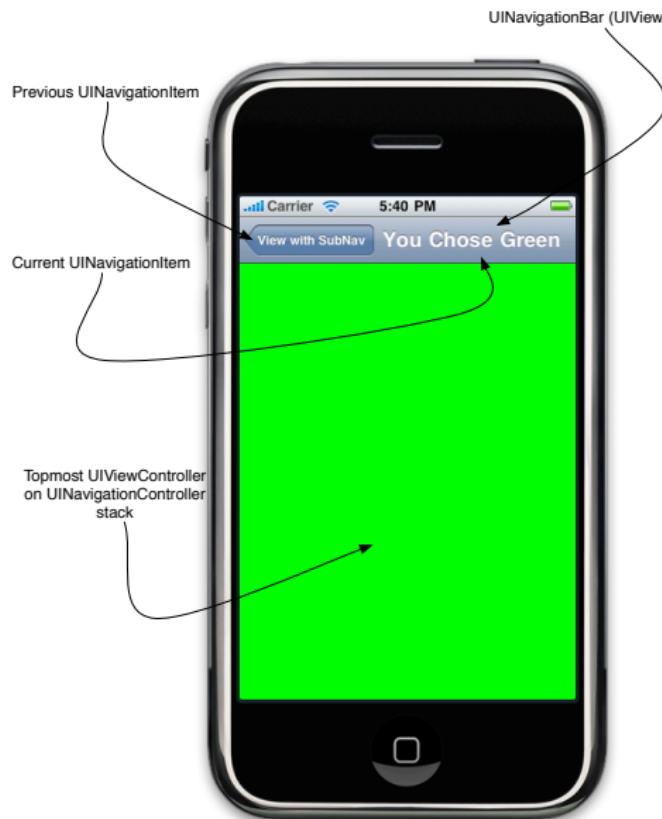


Figure : `UINavigationController` at work



Every UIViewController managed by a UINavigationController has a reference to that UINavigationController in [self navigationController].

```
1  -(void) viewDidLoad {
2      [super viewDidLoad];
3      NSLog(@"There are currently %i controllers in the navigation stack",
4          self.navigationController.childViewControllers.count);
5 }
```

You can add another UIViewController to the navigation stack by using the push segue when connecting view controllers in Storyboard.

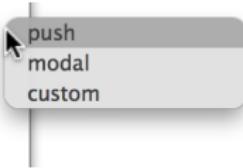


Figure : Selecting the push segue

*Note:* This segue will only work if you've already set up a UINavigationController, it will not automatically create a UINavigationController for you.

You can also push `UIViewController`s on the stack manually.

```
1 MyViewController * controller = [[MyViewController alloc] initWithNibName:nil  
2                                         bundle:nil];  
3 [[self navigationController] pushViewController:controller animated:YES];
```

# Lab 3

## Create a new “Single View” project

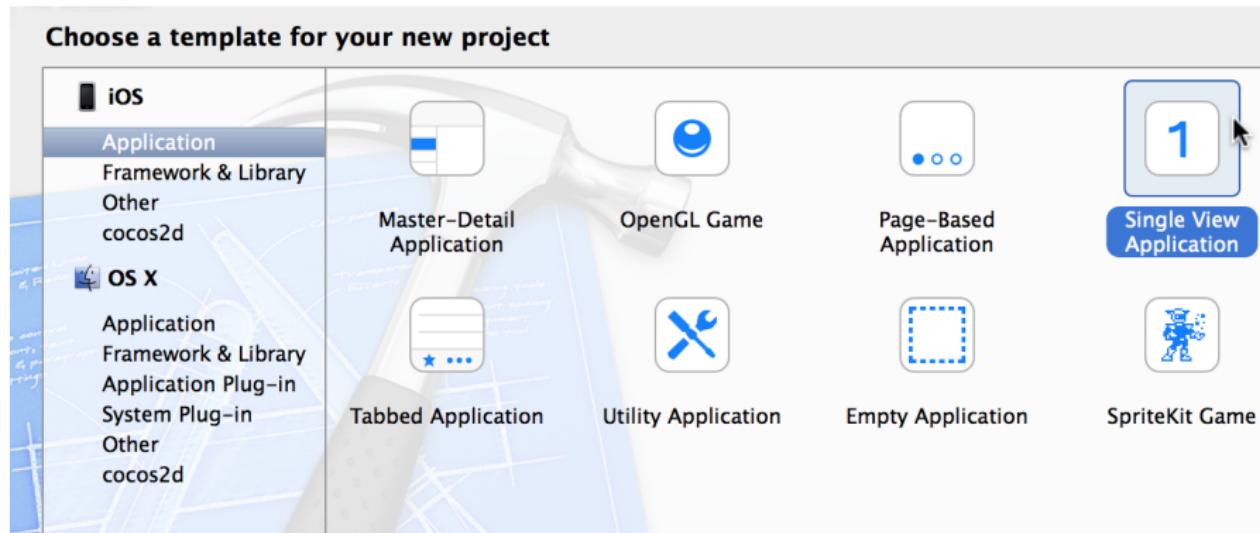


Figure : Create a new project

Open the project's storyboard and add a new “UINavigationController” to the storyboard.

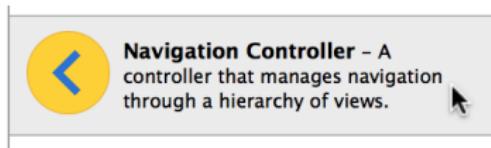


Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Drag the “root view controller” outlet from the UINavigationController to the default view controller in the storyboard.

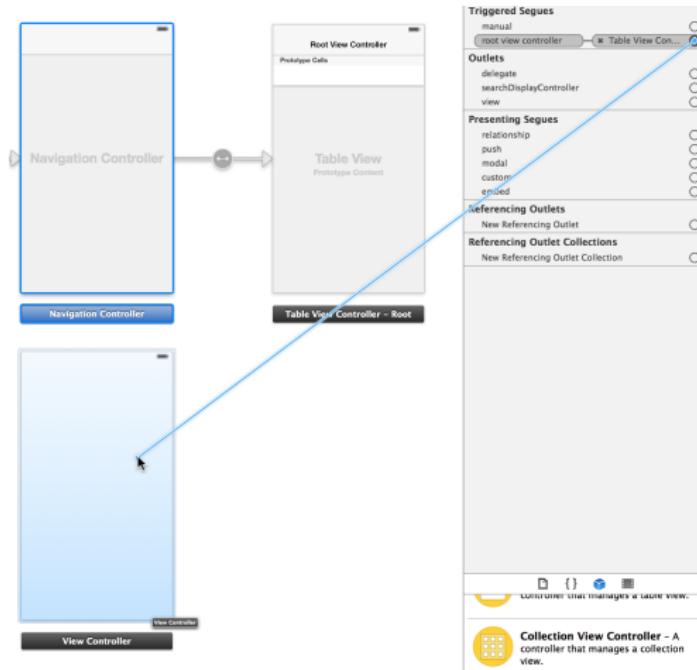


Figure : Change the root view controller outlet

Add a new NSObject subclass of “UIViewController” to the project named “DetailViewController”

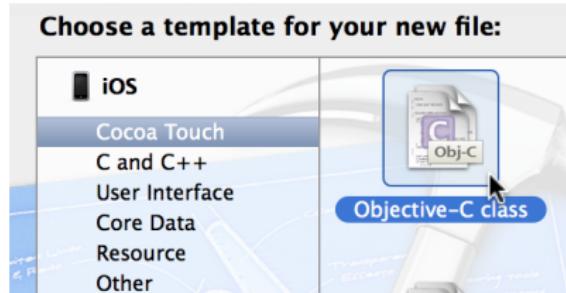


Figure : New Objective C class



Figure : Subclassing UIViewController

Add a “UIViewController” object from the Object library on to your storyboard.

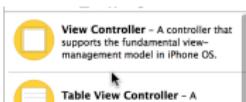


Figure : The UIViewController object in the Library

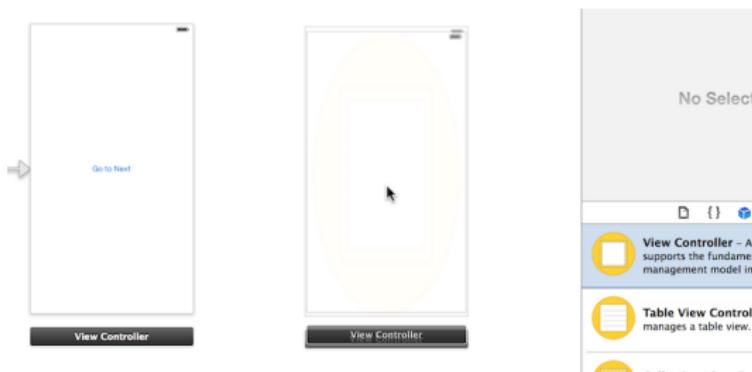


Figure : Drop UIViewController on Storyboard

Set the subclass of the UIViewController to “DetailViewController”.

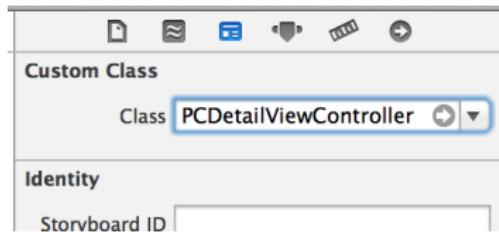


Figure : Setting the UIViewController subclass

Double click on the first UIViewController in the storyboard and add a button with the text “Show Detail” to its scene.

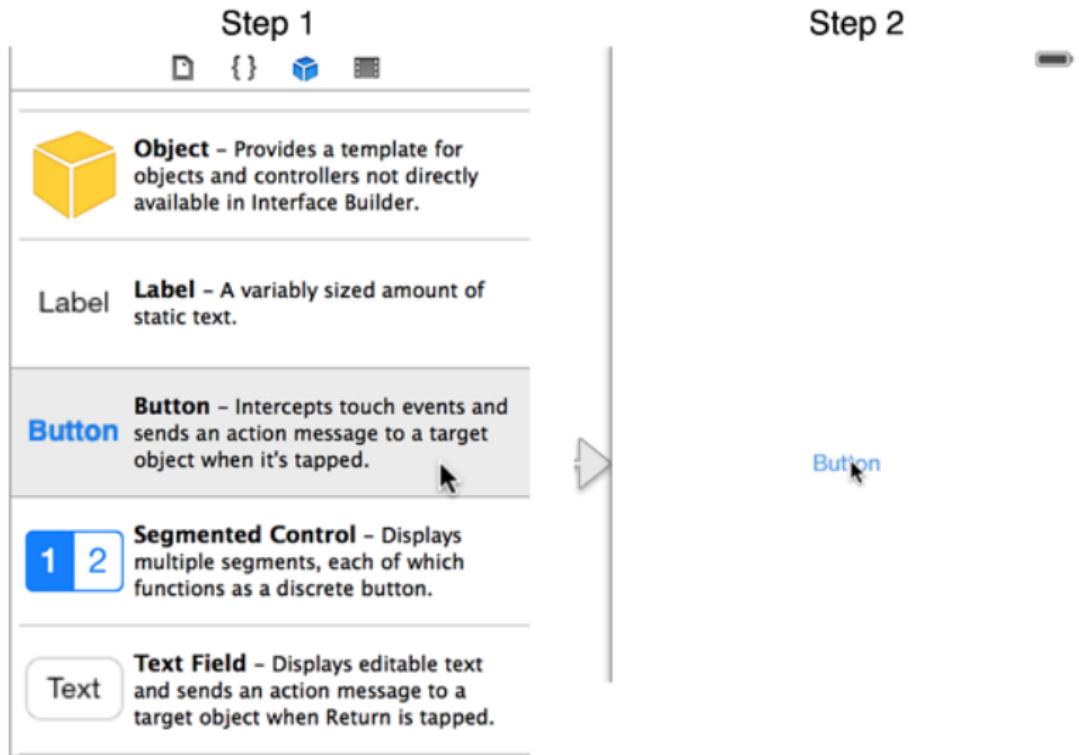


Figure : Drag a button to the scene

Select the button, and view its outlets in the inspector pane. Drag from the action outlet under Triggered Segues to the “Detail View Controller” scene in your storyboard. Choose “Push” when asked what sort of segue style will be used.

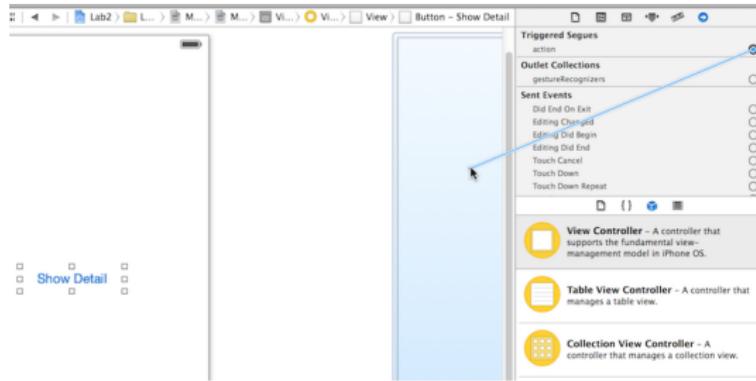
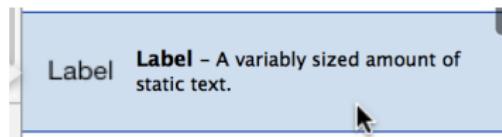


Figure : Connect the action to the Detail View Controller

Double click on the Detail View Controller and add a label to its scene with the text “Valuable Detail”.



**Figure :** Select and place the Label control

Select the navigation bar in the detail view controller, and change its title attribute to “My Detail View”.

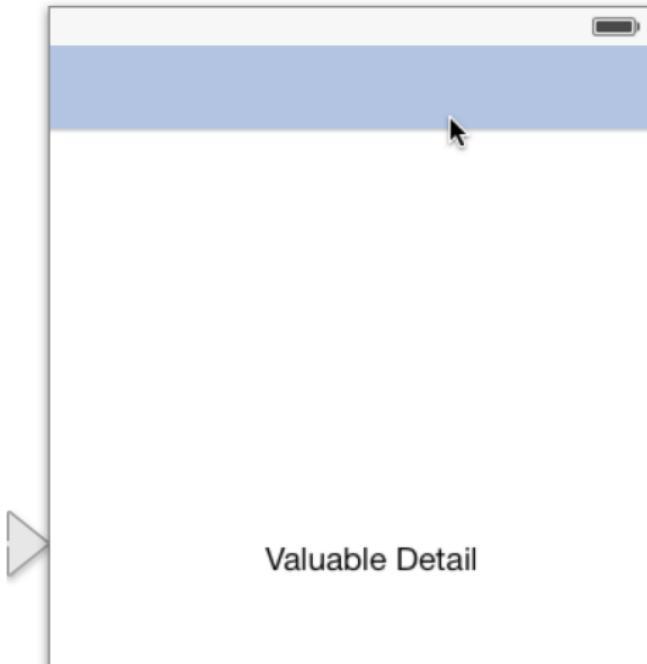


Figure : Select UINavigationItem in controller

Run the app!



Figure : Put it on the App Store!

*One more thing:* Add a button on your Detail View Controller that opens up a new View Controller that shows an image.

# UITabBarController

A UITabBar is the control that sits at the bottom of a screen, and allows you to switch between different views based on the button you click.



Figure : UITabBar examples

A UITabBarController swaps in UIViewControllers assigned to it when users click on the associated button

Each UIViewController has a tabBarItem property that UITabBarController uses to populate its UITabBar

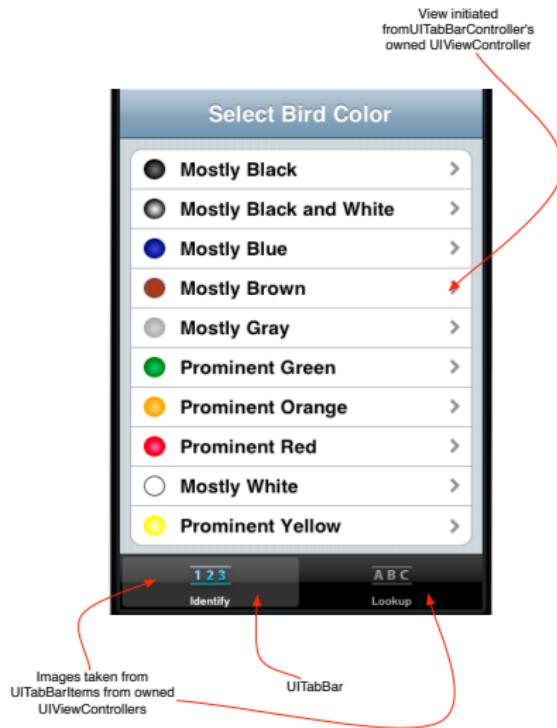


Figure : UITabBarController example



Figure : Sir Not Appearing In This Class

# UITableViewController

UITableViewController make up the majority of navigational aids in most iOS apps. If you see a vertical list of selectable items, it is almost certainly a UITableView.

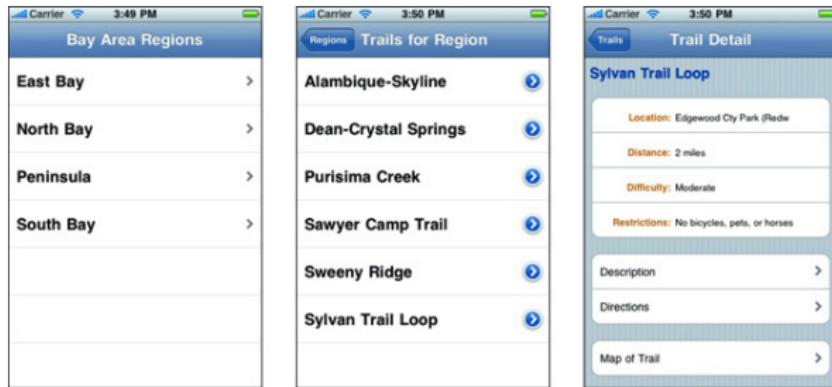


Figure : UITableView examples

A UITableView presents data in two dimensions: **sections** and **rows**.  
For any section **N**, there may be **M** rows of information to present.

Each cell in a UITableView is represented by a UITableViewCell. A UITableViewCell in its default implementation allows for a title label, a left aligned image view, and an optional accessory view that is right aligned.

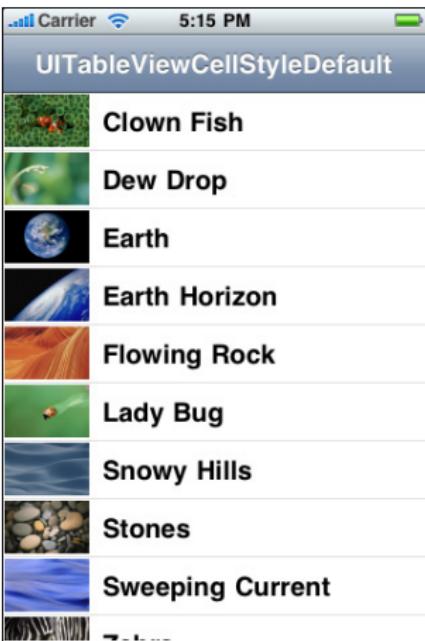
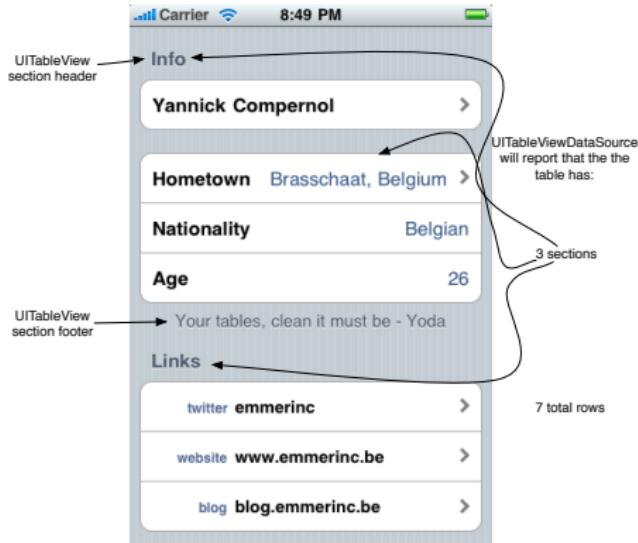


Figure : UITableViewCellStyle default style

A UITableView has special code that allows it to only create as many UITableViewCells as are currently on the screen, as opposed to the number of rows in your source data. This allows you to easily support paging through many thousands of items w/ negligible performance impact.

A UITableViewController has many pre-filled methods to make interacting with UITableViews easier. It is a subclass of UIViewController, and is merely a convenience to the programmer.



# Lab 4

# Create a new “Single View” project

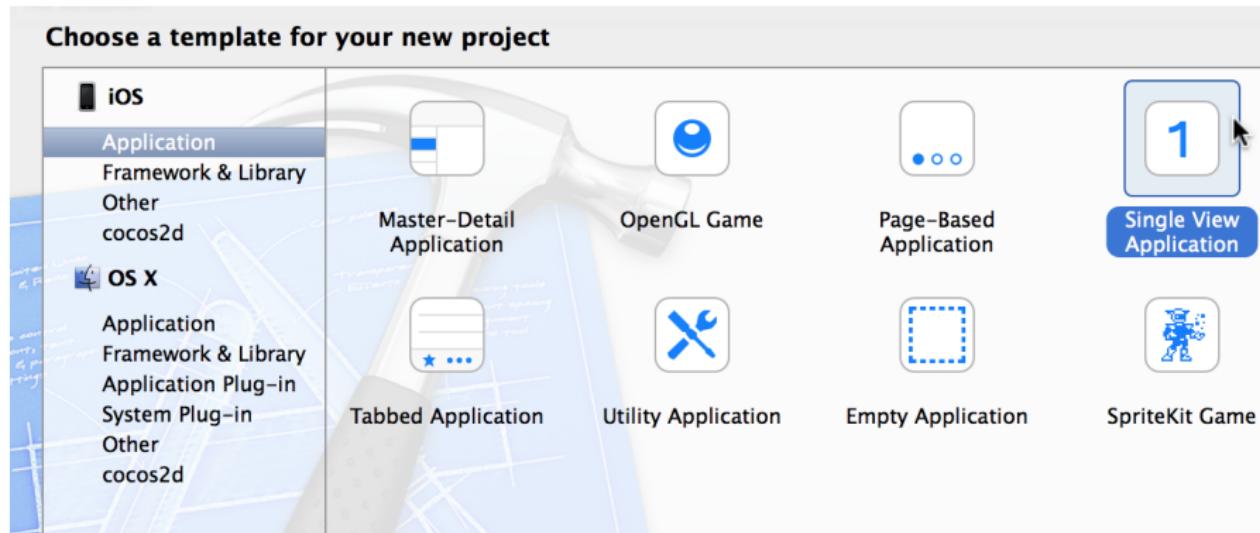


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

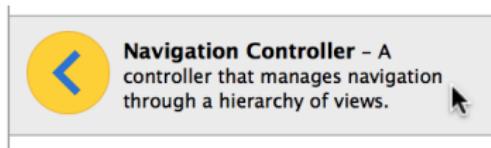


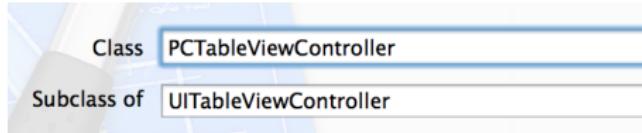
Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.



**Figure :** Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

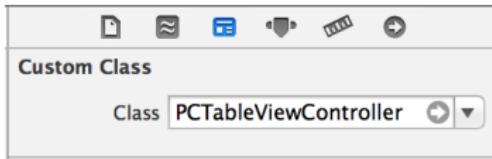


Figure : Change subclass to new UITableViewController

Add the following code to PCTableViewController.h:

```
1   @property NSArray * items;
```

Add the following code to PCTableViewController.m:

```
1   -(void) viewDidLoad {
2       [super viewDidLoad];
3       self.items = @[@"One", @"Two", @"Three"];
4 }
```

Change the method `numberOfSectionsInTableView: (UITableView *) tableView` in `PCTableViewController.m` to return 1.

Change the method `tableView:(UITableView *) tableView  
cellForRowAtIndexPath:(NSIndexPath *) path` to return  
`self.items.count`.

Change the method `tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath` to:

```
1  UITableViewCell *cell = [tableView
2      dequeueReusableCellWithIdentifier:@"DefaultCell"
3                      forIndexPath:indexPath];
4
5  cell.textLabel.text = [self.items objectAtIndex:indexPath.row];
6
7  return cell;
```

Double click on the table view controller in Storyboard and select the prototype UITableViewCell. Change its reuse identifier to “DefaultCell”.

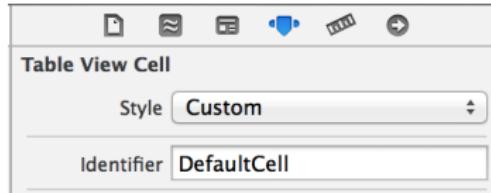


Figure : Change Reuse Identifier

In the PCViewController.h, add the following code.

```
1  @property NSString * name;
2  @property IBOutlet UILabel * nameLabel;
```

In Storyboard, drag a UILabel to PCViewController and connect it to the nameLabel outlet.

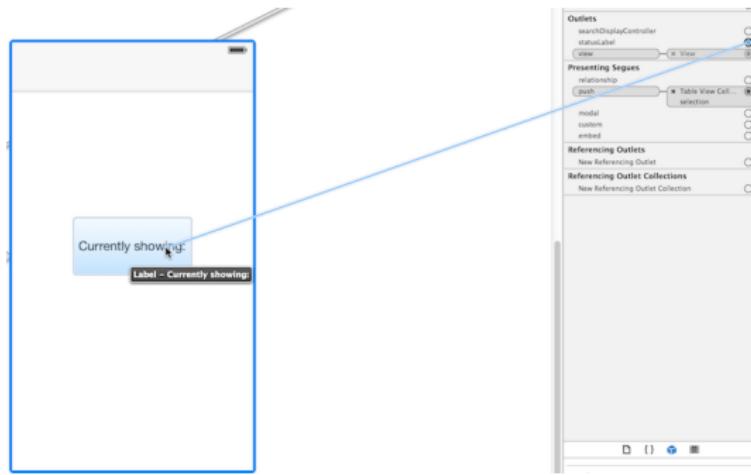


Figure : Connect the UILabel to the nameLabel outlet

In PCViewController.m, add the following code:

```
1  -(void) viewDidLoad {
2      [super viewDidLoad];
3      self.nameLabel.text = [NSString stringWithFormat:@"Name: %@", self.name];
4 }
```

In Storyboard, select the prototype cell in the table view controller and connect its “selection” triggered segue to the PCViewController and choose “Push”.

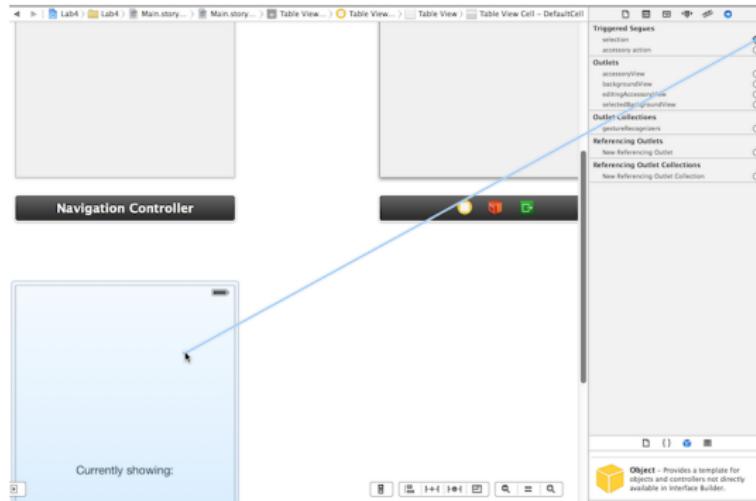


Figure : Connect the selection segue of the UITableViewCell

In PCTableViewController.m, uncomment the method  
prepareForSegue:sender: and add the following code:

```
1 PCViewController * c = (PCViewController *)[segue destinationViewController];  
2 c.name = [self.items objectAtIndex:self.tableView indexPathForSelectedRow.row
```

Add the following line to the top of the file:

```
1 #import "PCViewController.h"
```

Run it!



Figure : Time to get pumped

*One more thing:* Add a JSON file to your project that is an array of items. Deserialize it in your UITableViewController, and use the information from the JSON file to populate your UITableView and UIViewController.

## An Aside: The Delegate Pattern

You may have noticed that UITableViewController had odd methods like – `(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section`. Why does the UITableViewController need to have a UITableView passed to its `numberOfRowsInSection` method?

The delegate pattern allows for a weak binding between two objects, such that a “delegate” can be queried for configuration information and special behavior from an acting object.

Objective C formal protocols are typically how a delegate can indicate at compile time that it provides certain functionality. See `UITableViewDelegate` and `UITableViewDataSource`.

The delegate pattern is used widely throughout Apple frameworks.

```
netshade@shade [11:50:31] [/Applications/Xcode.app/Contents/Developer/Documentation  
-> % grep -ri 'delegate' ./* | wc -l  
2446
```

Figure : Delegate usage throughout the documentation

## Review Concepts: UIViewController Basics

Build an iOS app that:

- Has a UINavigationController
- Loads a UITableViewController as its first scene
- Populates the UITableViewController rows names taken from a JSON file in the local file system
- Populates the UITableViewController rows with images specified from the JSON file
- When a table row is selected, loads a detail view controller that presents the name and image that the user selected

## NSURL, NSURLRequest, NSMutableURLRequest AND NSURLConnection

NSURL is meant to only represent a single resource location

```
1 NSURL * theUrl = [NSURL URLWithString:@"http://bootstrapping-ios.com/"];
```

NSURL can be allocated to represent either a filesystem location, or a web resource

```
1 NSURL * aWebUrl = [NSURL URLWithString:@"http://news.ycombinator.org/"];  
2 NSURL * aFileUrl = [NSURL fileURLWithPath:[[NSBundle mainBundle] pathForResource:
```

NSURLRequest and NSMutableURLRequest represent specific web resources that you'd like to initiate a connection to; NSURLRequest should be used for simple GET HTTP requests, while NSMutableURLRequest can be used for more complex HTTP requests (POST, PUT, file uploads).

`NSMutableURLRequest` allows you to set HTTP headers, the HTTP method used, and the request body.

```
1 NSURLRequest * request = [NSURLRequest
2     requestWithURL:[NSURL URLWithString:@"http://bootstrapping-ios.com/"]];
3
4 NSMutableURLRequest * customRequest = [NSMutableURLRequest
5     requestWithURL:[NSURL URLWithString:@"http://bootstrapping-ios.com/"]];
6 [customRequest setHTTPMethod:@"POST"];
7 [customRequest setValue:@"Some-Token"
8     forHTTPHeaderField:@"X-Request-Token"];
9 [customRequest setValue:@"application/x-www-form-encoded"
10    forHTTPHeaderField:@"Content-Type"];
11 [customRequest setHTTPBody:[@"some-form=values"
12        dataUsingEncoding:NSUTF8StringEncoding]];
```

NSURLConnection can send data both synchronously or asynchronously. Asynchronous response information is passed back to the connection delegate via the NSURLConnectionDelegate, NSURLConnectionDataDelegate and NSURLConnectionDownloadDelegate protocols.

NSURLConnection initiates the download and returns the NSHTTPURLResponse which contains the body of the response, as well as http status code and response headers.

```
1  NSURLRequest * request = [NSURLRequest
2      requestWithURL:[NSURL URLWithString:@"http://bootstrapping-ios.com/"]];
3
4  NSError * error;
5  NSHTTPURLResponse * response;
6  NSData * syncResponse = [NSURLConnection sendSynchronousRequest:request
7      returningResponse:&response
8      error:&error];
9
10 NSURLConnection * asyncConnection = [[NSURLConnection alloc]
11     initWithRequest:request
12     delegate:self
13     startImmediately:YES];
```

NSURLSession adds conveniences for downloading content in the background.

# Lab 5

# Create a new “Single View” project

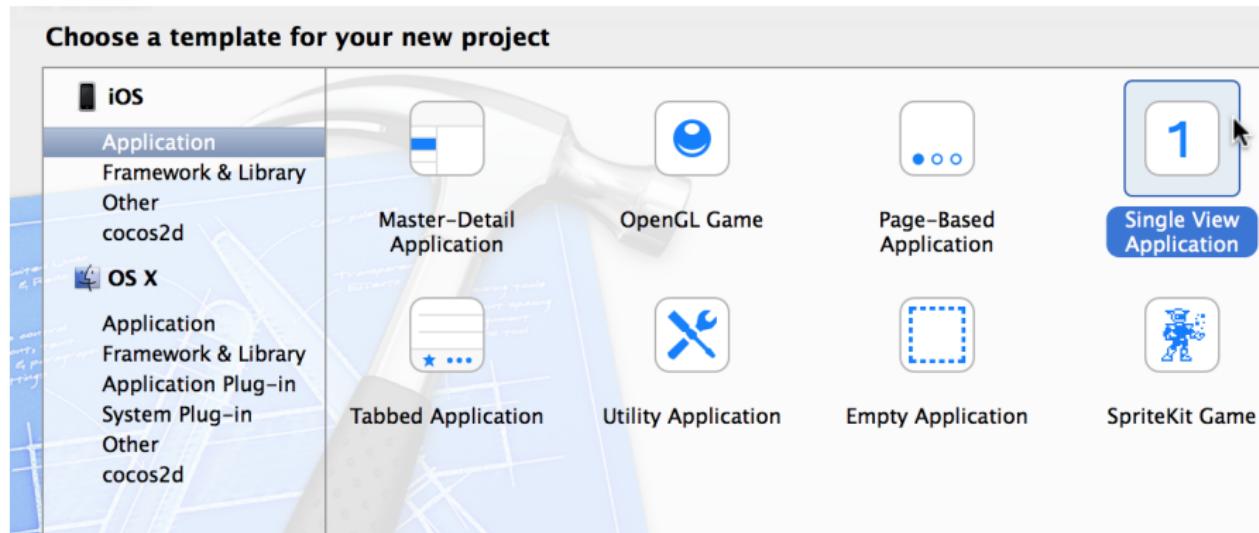


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

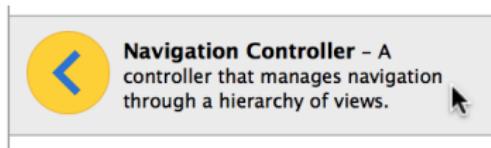


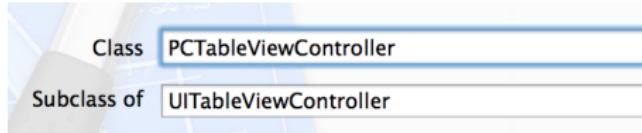
Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.



**Figure :** Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

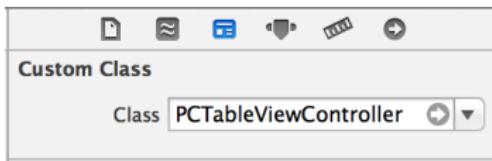


Figure : Change subclass to new UITableViewController

In the viewDidLoad method of your UITableViewController, initialize a request to the URL:

```
http://bootstrapping-ios.com/echo?names=one&\  
names=two&\  
names=three
```

Using NSURLConnection, asynchronously request the above JSON, deserialize it into an NSArray, and then reload your tableview.

You will need to have your view controller adopt specific methods from the `NSURLConnectionDelegate` and `NSURLConnectionDataDelegate` protocols. You will also need to store the request data in something like `NSMutableData`.

Run the app!



Figure : Basically an expert

*One more thing:* Change your request to a POST request  
(NSMutableURLRequest and setHTTPMethod:) with  
application/x-www-form-urlencoded content type  
(setValue:forHTTPHeaderField:), and instead of sending a  
querystring in the URL, send the key value pairs in the HTTP body  
(setHTTPBody:).

*And another thing:* Select your UITableViewController and enable “Refreshing” in its attributes. A UIRefreshControl will be added to your UITableViewController.

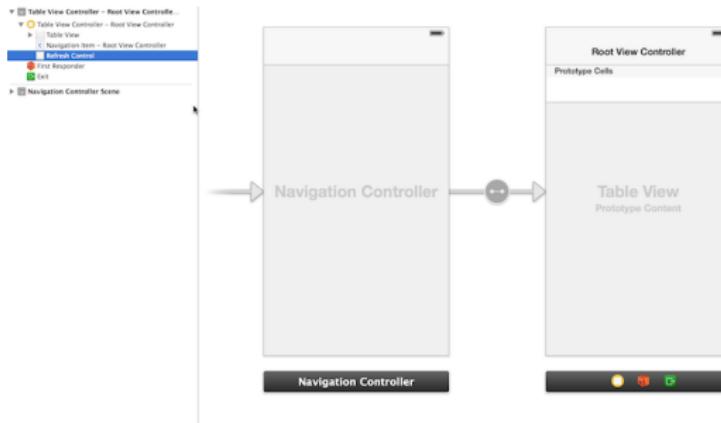


Figure : Select the refresh control

Connect the UIRefreshControl's Value Changed outlet to an IBAction method on your UITableViewController that triggers a new download. Tell the refresh control to change its appearance through [self.refreshControl beginRefreshing] and [self.refreshControl endRefreshing].

## NSTimer, NSRunLoop, NSThread and the Event Loop

Cocoa represents application threads using the `NSThread` object. Each `NSThread` has an `NSRunLoop` associated with it, which is responsible for managing IO and timer events.

The main application thread (the UI thread) is one of a set of default threads created for you. Any operations that occur on this main thread can cause the user interface to become non-responsive; it is important that you defer long running processes to outside this thread.

To perform an action at some later point, or on an interval, you can register an NSTimer with the current NSRunLoop. An NSTimer will invoke a selector you specify on a schedule you control.

```
1 [NSTimer scheduledTimerWithTimeInterval:10.0
2     target:self
3     selector:@selector(runEveryTenSeconds:)
4     userInfo:nil
5     repeats:YES];
6
7 -(void) runEveryTenSeconds:(NSTimer *) {
8     NSLog(@"Timer fired");
9 }
```

These timer actions occur in the thread they were scheduled from, so be cautious about scheduling expensive operations.

# Lab 6

# Create a new “Single View” project

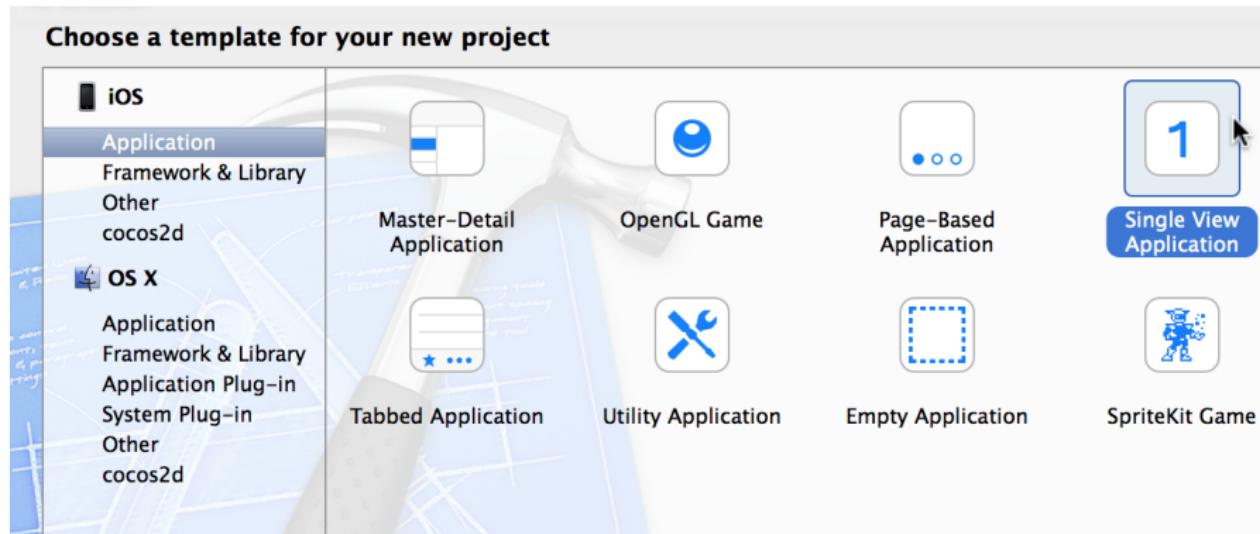


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

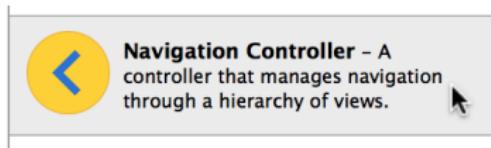


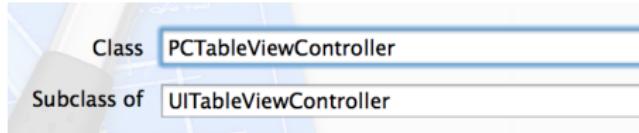
Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.



**Figure :** Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

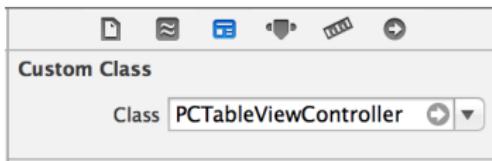


Figure : Change subclass to new UITableViewController

In the `viewDidLoad` method of your `UITableViewController`, initialize a request to the URL:

```
1 NSTimeInterval since1970 = [[NSDate date] timeIntervalSince1970];
2 NSURL * theUrl = [NSURL URLWithString:
3     [NSString
4         stringWithFormat:@"http://bootstrapping-ios.com/\echo?names=one-%f\
5             &names=two-%f\
6             &names=three-%f",
7         since1970, since1970, since1970]];
```

Using `NSURLConnection`, asynchronously request the above JSON, deserialize it into an `NSArray`, and then reload your tableview.

Create an NSTimer and schedule refreshes of the data every 10 seconds.

```
1  timer = [NSTimer scheduledTimerWithTimeInterval:5.0
2      target:self
3      selector:@selector(doRefresh:)
4      userInfo:nil
5      repeats:YES];
```

Add a method called dealloc that cancels the timer:

```
1  -(void) dealloc {
2      [timer invalidate];
3      timer = nil;
4  }
```

Run the app!



Figure : Yay!

*One more thing:* Add a call to sleep() (`man 3 sleep`) in your timer that intentionally delays response time of the refresh. Observe how UI suffers as the timer action takes longer to execute.

## Installing on the iOS Device

So far we've been running everything on the simulator. In order to run code on our device, we need to do two things:

- Have a valid code signing certificate
- Have a valid provisioning profile

A code signing certificate allows us to prove we are the trusted origin of our code. A provisioning profile creates the association between the certificate and the devices that it's allowed to run on.

You will need a separate certificate and profile for:

- Development deployment
- AdHoc deployment
- AppStore deployment
- Enterprise deployment

XCode will generally create the files you need, but you may manually create (and debug) the requisite files through:

- The XCode Organizer

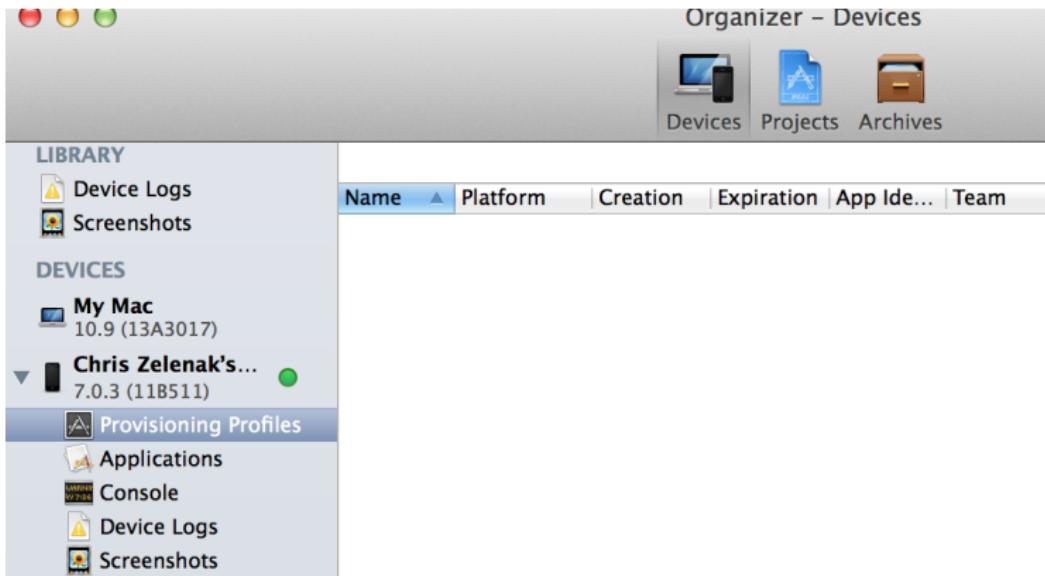


Figure : XCode Organizer

## ■ iTunes DevCenter

<https://developer.apple.com/account/overview.action>

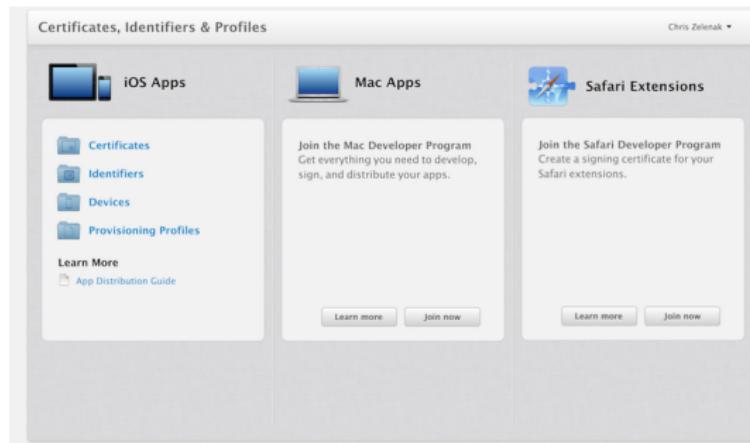


Figure : DevCenter

# Lab 7

Open up “Lab 6” in XCode.

Connect your iPhone to your machine.

Select the iOS device in the “Destination” dropdown.

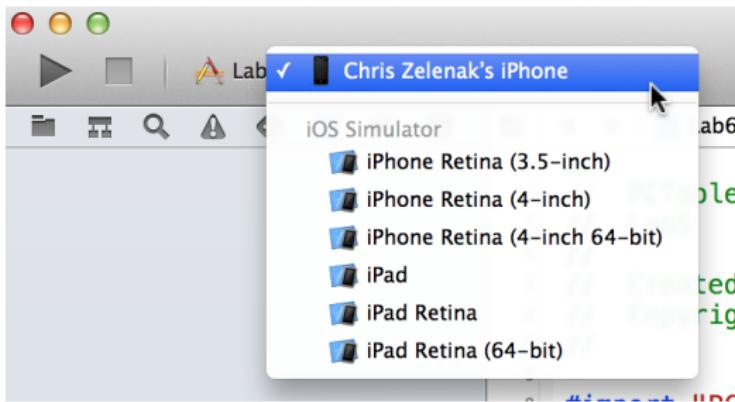


Figure : Select your iOS device

Run the app. If you've never actually ran a developer app on your device before, XCode should automatically create the files you need to sign code for the device.

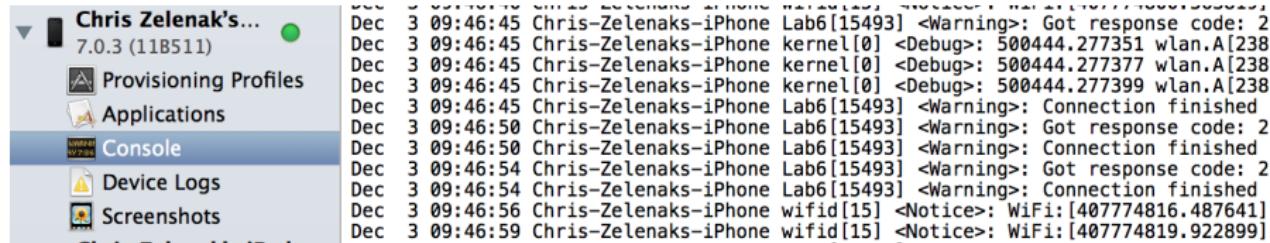
Alternatively, you can manually set the device to be a developer device on your account by adding it to the member center in XCode Organizer.



Figure : Adding to the Member Center manually

While the app is running on your device, you can:

- Access Console output in Organizer



The screenshot shows the Xcode Organizer interface. On the left, there is a sidebar with the following items:

- Chris Zelenak's... (7.0.3 (11B511))
- Provisioning Profiles
- Applications
- Console** (highlighted)
- Device Logs
- Screenshots

On the right, the main pane displays the console output for the application "Chris-Zelenaks-iPhone". The log entries are as follows:

```
Dec 3 09:46:45 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Got response code: 2
Dec 3 09:46:45 Chris-Zelenaks-iPhone kernel[0] <Debug>: 500444.277351 wlan.A[238]
Dec 3 09:46:45 Chris-Zelenaks-iPhone kernel[0] <Debug>: 500444.277377 wlan.A[238]
Dec 3 09:46:45 Chris-Zelenaks-iPhone kernel[0] <Debug>: 500444.277399 wlan.A[238]
Dec 3 09:46:45 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Connection finished
Dec 3 09:46:50 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Got response code: 2
Dec 3 09:46:50 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Connection finished
Dec 3 09:46:54 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Got response code: 2
Dec 3 09:46:54 Chris-Zelenaks-iPhone Lab6[15493] <Warning>: Connection finished
Dec 3 09:46:56 Chris-Zelenaks-iPhone wifid[15] <Notice>: WiFi:[407774816.487641]
Dec 3 09:46:59 Chris-Zelenaks-iPhone wifid[15] <Notice>: WiFi:[407774819.922899]
```

Figure : Watching console output

## ■ Set breakpoints and inspect memory in XCode



Figure : Inspecting breakpoints

## ■ Interactively query memory at breakpoints with LLDB

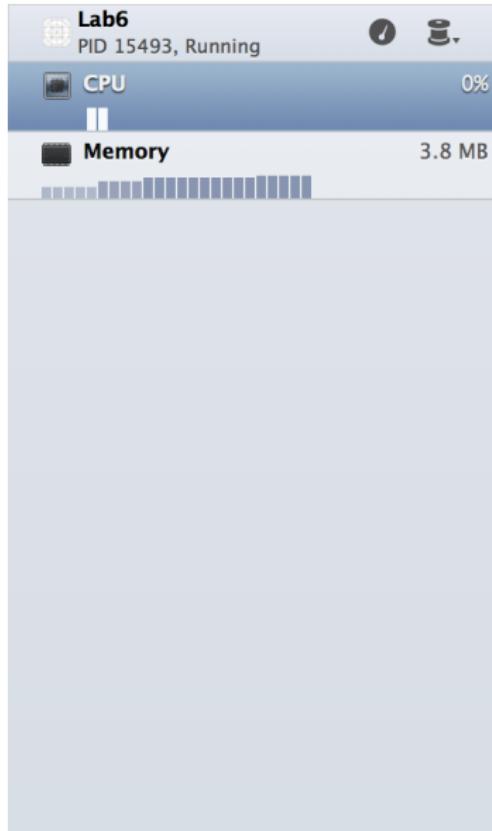
```
bleViewController refresh]
(lldb) po self
<PCTableViewController: 0x1654d160>
(lldb) po self.tableView
<UITableView: 0x168ad600; frame = (0 0; 320 568); clipsToBounds
= YES; opaque = NO; autoresizingMask = W+H; gestureRecognizers =
<NSArray: 0x1662b600>; layer = <CALayer: 0x16557620>;
contentOffset: {0, -64}>
(lldb) po [NSData data]
<
(lldb) help
The following is a list of built-in, permanent debugger
commands:

-regexp-attach    -- Attach to a process id if in decimal,
otherwise treat the
                  argument as a process name to attach to.
-regexp-break     -- Set a breakpoint using a regular expression
to specify the
                  location, where <linenum> is in decimal and
<address> is
                  in hex.
-regexp-bt        -- Show a backtrace. An optional argument is
All Output ♦
```



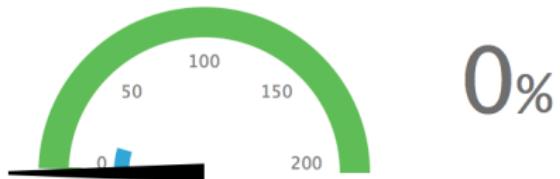
Figure : Querying objects and running code via LLDB

## ■ Observe basic system metrics



### CPU

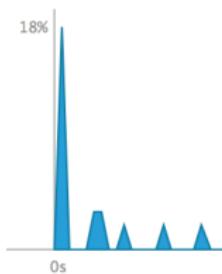
Percentage Utilized



0%

### Utilization over Time

Duration: 24 sec  
High: 18%  
Low: 0%



Threads

# Custom Views

Views are descendants of `UIView`, and have several properties that allow for some basic customization. Some commonly used attributes are:

- `.frame`, of type `CGRect`
- `.backgroundColor` of type `UIColor`
- `.alpha` of type `CGFloat`
- `.transform` of type `CGAffineTransform`
- `.hidden` of type `BOOL`

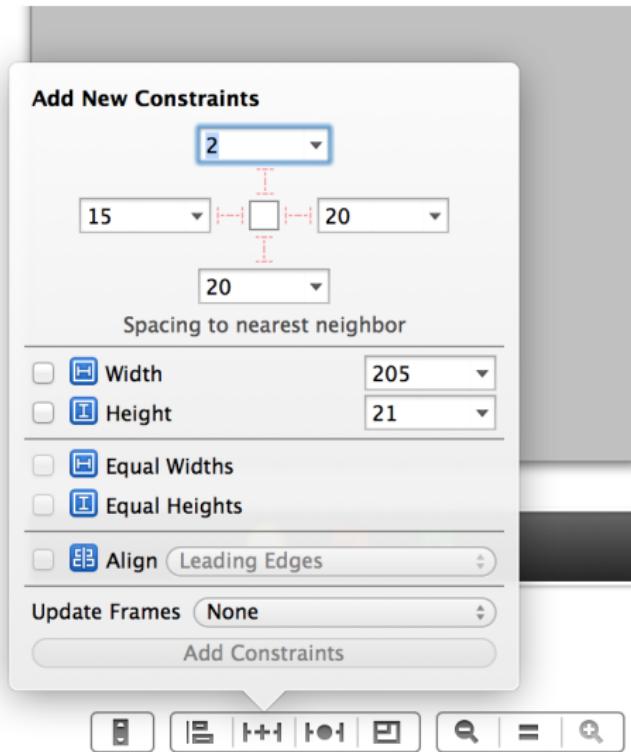
UIView can be subclassed and customized, just like any other view. A UIView may contain other UIViews to **compose** a higher order control. They can also draw directly to the screen if you need to manually draw certain types of graphics using an API called Core Graphics.

Any UIViewController has a `view` attribute which represents the top level view it is currently displaying. You can programmatically add other views to this using methods like `addSubview:`, or you can modify the `view` in Storyboard as we have been doing.

Views may have a tag associated with them, an integer identifier that allows specific views to be requested with the `viewWithTag:` method of `UIView`.

```
1  UIView * v = [[UIView alloc] initWithFrame:CGRectMake(0., 0., 20., 20.)];
2  v.tag = 100;
3
4  [self.view addSubview:v];
5
6  UIView * theView = [self.view viewWithTag:100];
```

UIViews may also specify certain constraints with respect to their sibling views. This is called Auto Layout, and allows for UIViews to change their appearance w/ respect to other views.



Additionally, UIViews can have application wide settings applied to them using UIAppearance objects, which set defaults for object appearance:

```
1  [[UINavigationBar appearance] setBarTintColor:[UIColor blackColor]];
2  [[UITableViewCell appearanceWhenContainedIn:[UITableView class], nil]
3   setBackgroundColor:[UIColor purpleColor]];
```

# Lab 8

# Create a new “Single View” project

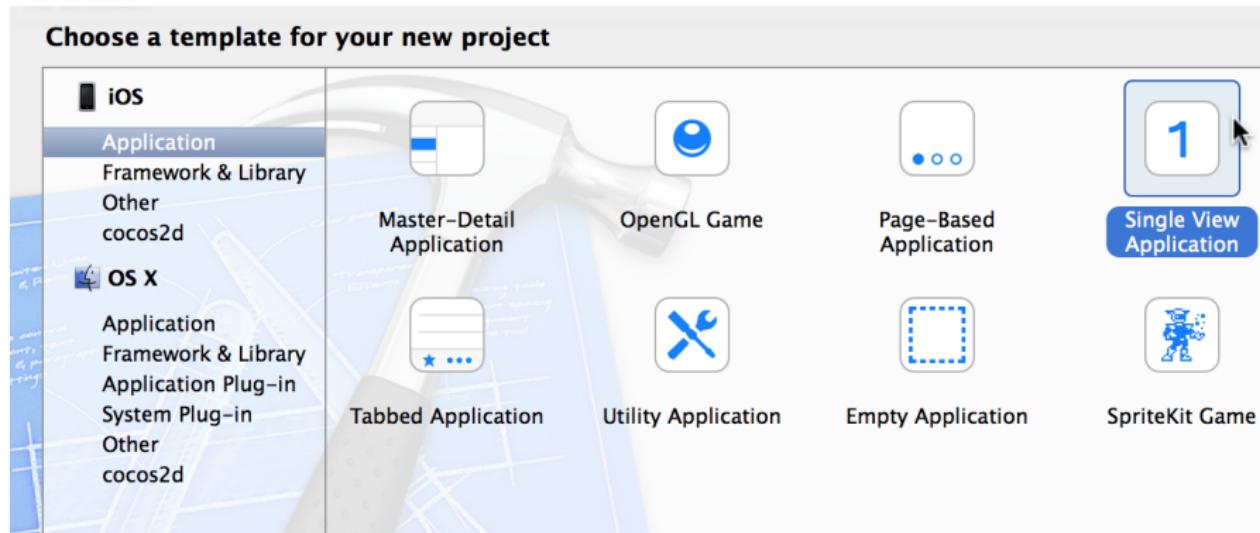


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

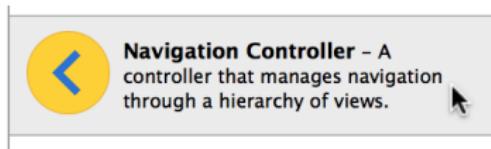


Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.

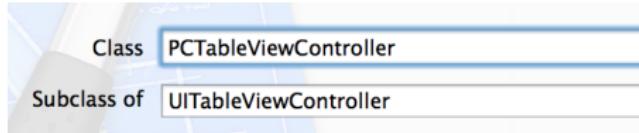


Figure : Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

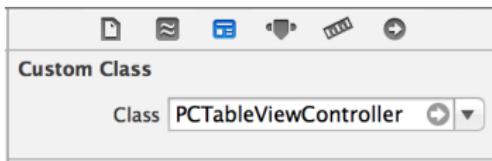


Figure : Change subclass to new UITableViewController

Populate your UITableView with data. :)

In Storyboard, add the following items to the prototype cell in your UITableView:

- A UIImageView (tag 100)
- A UILabel that is the “Title” label (tag 101)
- A UILabel that is the “Subtitle” label (tag 102)

Add an alignment constraint to the “Title” and “Subtitle” such that they keep a strict right margin with respect to their superview.

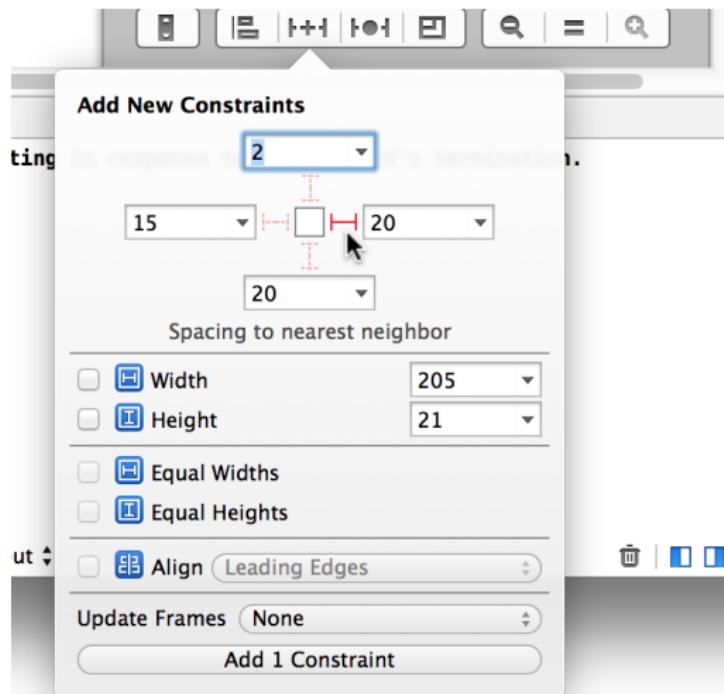


Figure : Set right margin

In the `tableView:cellForRowAtIndexPath:` method of your table view controller, retrieve the 3 views based on their tag, and set their values respectively.

```
1  UIImageView * imageView = [cell viewWithTag:100];
2  UILabel * title = [cell viewWithTag:101];
3  UILabel * subtitle = [cell viewWithTag:102];
4  title.text = @"Title"
5  subtitle.text = @"image"
6  imageView.image = anImage;
```

Run the app!



Figure : Put it on the App Store!

*One more thing:* Using the `UIAppearance` protocol, set the default font and text color of any `UILabel` object when your app starts up.

## Review Concepts: Integrating With Remote Resources

Build an iOS app that:

- Uses a UITableViewController to display main content
- Loads the content for the UITableViewController from a remote JSON file
- Loads a detail view when a row in the UITableViewController is tapped on
- Customize the default appearance of UILabels using the UIAppearance class
- Has a custom UITableViewCell to display the JSON data

# Revisiting Some Basics

Instantiating an Objective C object looks like:

```
1 NSString * s = [[NSString alloc] init];
```

where `alloc` generates the space for the class, and `init` may be thought of as the constructor.

Each class also has the concept of a destructor. The destructor method is named `dealloc`, and is called for you when your object is removed from memory.

```
1  -(void) dealloc {  
2      fclose(myFileHandle);  
3      [myDatabaseConnection close];  
4  }
```

Use the `dealloc` method to teardown resources used only by your object, as well as remove any global references to your object that will no longer be valid when it leaves memory.

Each Objective C class can have a number of instance variables, and a number of properties. **Instance variables** are variables available internally to a given instance of an Objective C object; **properties** are the public facing API of a given object.

```
1      @interface MyClass : NSObject {
2          int anInstanceVariable;
3          NSString * anotherInstanceVariable;
4      }
5
6
7      @property NSNumber * thisNumberIsAvailableToOtherClasses;
8
9      @end
```

You can modify instance variable scope if you want the instance variables themselves to be visible outside the instance. The default scope is `@protected`, but you can also use `@public` and `@private`.

```
1  @interface MyClass : NSObject {
2      @public
3      int anInstanceVariable;
4
5      @private
6      NSString * anotherInstanceVariable;
7      NSNumber * someNumber;
8
9      @protected
10     NSDictionary * aDictionary;
11 }
12
13 @end
```

Objective C objects are “just” structs, so if you intend to access a public instance variable, you can do so by using the -> operator.

```
1 MyClass * c = [[MyClass alloc] init];
2 NSLog(@"Some int: %i", c->anInstanceVariable);
```

The -> operator looks up the member variable of the struct pointed to by a pointer.

The delegate pattern allows for an object to notify its delegate that some event has occurred, or to query its delegate for more information. If we were to implement this pattern ourselves, it would look like this:

```
1      @protocol SimpleURLDownloader
2
3      -(void) downloadFinishedWithString:(NSString *) content;
4
5      @end
6
7
8      @interface SimpleURLDownloader : NSObject
9
10     @property(weak) id<SimpleURLDownloaderDelegate> delegate;
11
12     @end
```

```
1 @implementation SimpleURLDownloader  
2  
3 -(void) downloadUrl:(NSURL *) url {  
4     NSString * urlResponse = MagicallyDownloadData();  
5     [self.delegate downloadFinishedWithString:urlResponse];  
6 }  
7  
8 @end
```

```
1 -(void) viewDidLoad {
2     SimpleURLDownloader * dl = [[SimpleURLDownloader alloc] init];
3     dl.delegate = self;
4     [dl downloadUrl:[NSURL URLWithString:@"http://google.com/"]];
5 }
6
7 -(void) downloadFinishedWithString:(NSString *) content {
8     NSLog(@"Just downloaded google, %@", content);
9 }
```

# iOS App Structure

An iOS app shares common ancestry with Mac OS apps; they are “bundles” (NSBundle) that can contain many resources. So far, we’ve used the [NSBundle mainBundle] accessor to grab a reference to the bundle that contains our app; this reference allows us to query for information included with our app.

```
1  [[NSBundle mainBundle] pathForResource:@"file" ofType:@"json"] ;  
2  [[NSBundle mainBundle] classNamed:@"MyOwnClass"] ;
```

You can inspect the physical layout of an iOS app by opening its contents as seen by the simulator:

▼	SEE442FE-9A85-...-7C62F141DBCC	Today, 5:33 AM
►	Documents	Nov 30, 2013, 10:14 AM
►	Lab4	Dec 2, 2013, 6:33 AM
►	Library	Nov 30, 2013, 10:14 AM
►	tmp	Nov 30, 2013, 10:14 AM

Figure : App contents

These apps live in `~/Library/Application Support/iPhone Simulator/<Simulator Version>`.

Along with an NSBundle that is associated with our app, there is a UIApplication. While the NSBundle refers to primarily filesystem items associated with our app, the UIApplication is our reference to the currently running instance.

```
1  [[UIApplication sharedApplication] setStatusBarHidden:YES  
2  withAnimation:UIStatusBarAnimationFade];
```

A UIApplication is created automatically for you when your application is started up. When XCode creates a project for you, it will also automatically create a UIApplication delegate (UIApplicationDelegate protocol) that responds to UIApplication events.

```
1 - (BOOL)application:(UIApplication *)application  
2     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
```

The methods provided by UIApplicationDelegate are meant to help you handle the transition between application states: active, inactive, background and terminated.

Your application will usually be rejected if you don't suspend operation correctly in cases where a phone call is received or the app is backgrounded.

Your application has access to a number of sandboxed directories. While you don't have write access to anything in the bundle root, you can write to the Documents directory, the Caches directory and the tmp directory.

Directory paths should be looked up via specific helpers:

```
1 NSArray * docDirs = NSSearchPathForDirectoriesInDomains(
2                                     NSDocumentsDirectory,
3                                     NSUserDomainMask,
4                                     YES);
5 NSString * actualDir = [docDirs objectAtIndex:0];
6
7 NSString * tmpDir = NSTemporaryDirectory();
```

Your application may be rejected if you store anything other than user generated content in the Documents directory. Furthermore, the tmp and cache directories have “undefined” clearing schedules, so you cannot guarantee that files placed there will stay there for any length of time.

You can use traditional C file access functions (`fopen`, `fread`) to manage file access, or you can use convenience functions provided by the `NSFileManager`, `NSInputStream` and `NSOutputStream` classes.

Classes like `NSData` and `NSString` also have convenience methods like `initWithContentsOfFile:` and `writeToFile:atomically:` to make dealing with the filesystem easier.

# Lab 9

# Create a new “Single View” project

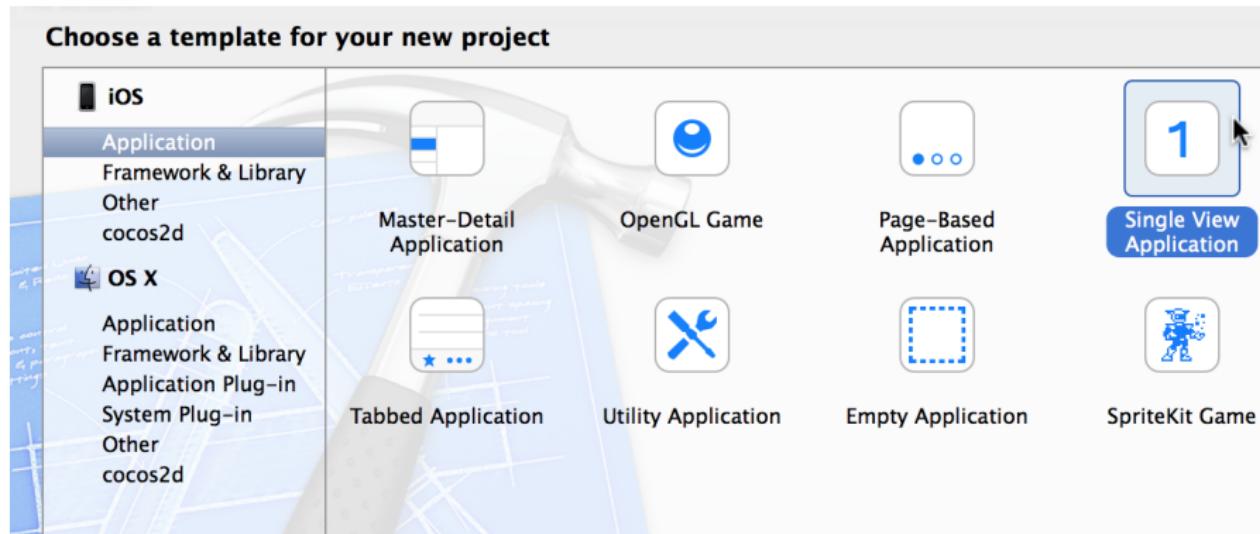


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

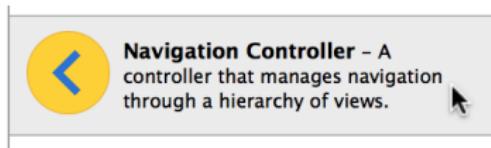


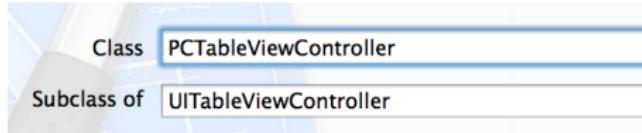
Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



**Figure :** Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.



**Figure :** Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

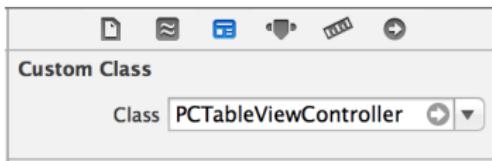


Figure : Change subclass to new UITableViewController

In the `viewDidLoad` method of your `UITableViewController`, initialize a request to the URL:

```
http://www.bootstrapping-ios.com/echoWait?names=one&\  
names=two&\  
names=three
```

Using the `writeToFile:atomically:` method of `NSData`, write the response of the URL to the `NSCachesDirectory` directory. In your `viewDidLoad` method, check for the existence of this file before attempting to perform the network request, and if it exists, load it instead of the network version.



Figure : Put it on the App Store!

*One more thing:* Find the cache JSON file in your simulator filesystem.

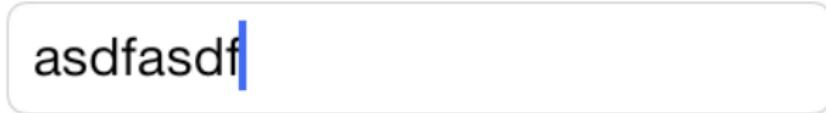
# User Input

iOS user input typically consists of the following two views:

- UITextField: A single line piece of editable text
- UITextView: A multi line text editor

These views form the foundation of most user text input, and their simplest use is plain, unadorned text.

Title



asdfasdfs

Figure : Simple UITextField

However, they can also display attributed strings (`NSAttributedString`), strings which specify special formatting information like bold text, font, files attached to character ranges and more.

UITextViews and UITextFields also support custom keyboards. You can specify a custom keyboard (`inputView`) or keyboard accessory (`inputAccessoryView`) which you can use to add special formatting instructions to your user input.



Figure : Custom keyboard accessory

# Lab 10

# Create a new “Single View” project

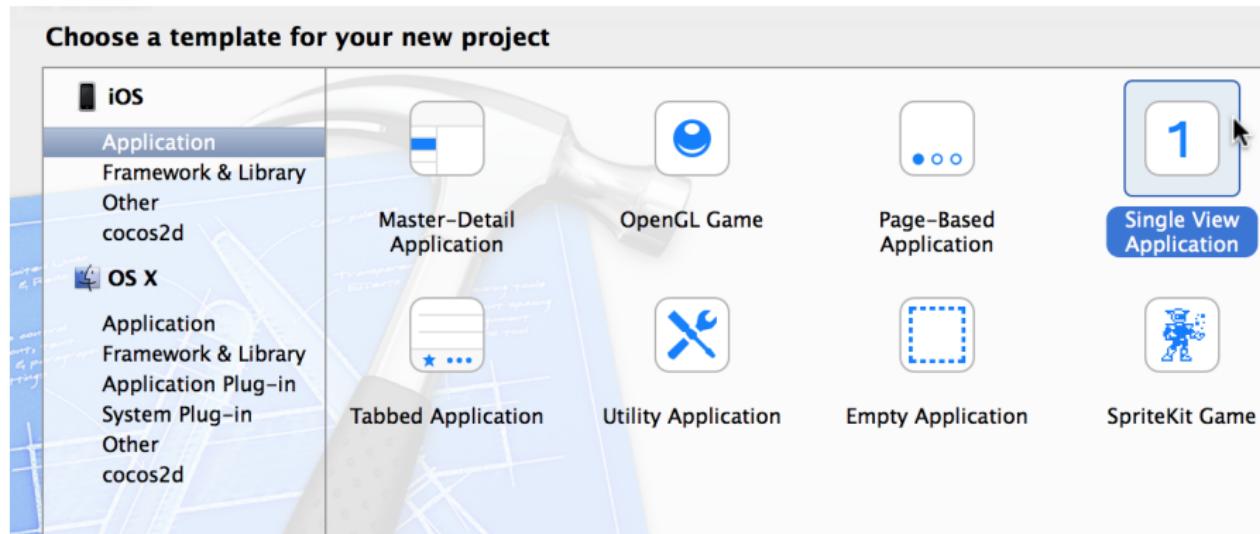


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

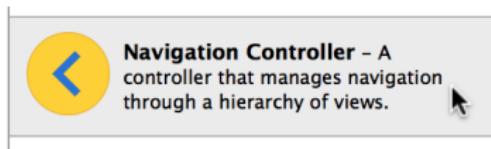


Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



Figure : Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.

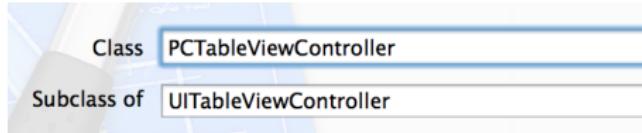


Figure : Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

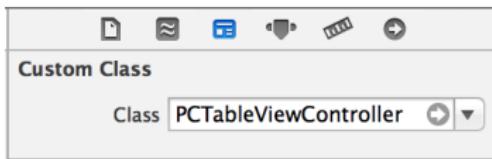
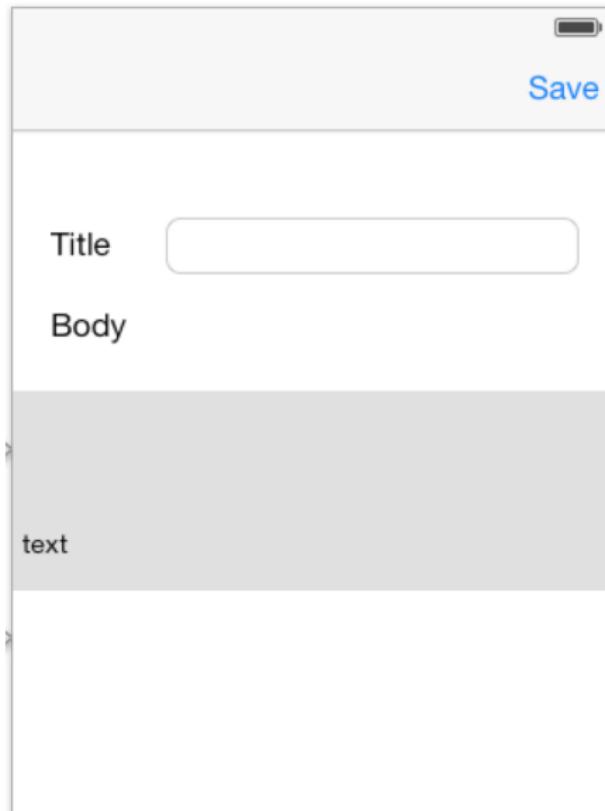


Figure : Change subclass to new UITableViewController

Using the `NSFileManager` object, query the `NSDocumentsDirectory` for the list of files it contains (`contentsOfDirectoryAtPath:error:`). Use this list to populate your tableview.

Create a detail view that contains a UITextField for a title, and a UITextView for a body. Connect these via IBOutlets to the view controller for this detail view.



Add an NSString property called path to the Detail View Controller.

Add a “Bar Button Item” to the right hand side of the navigation bar of your table view controller. Change it’s text to “New”.

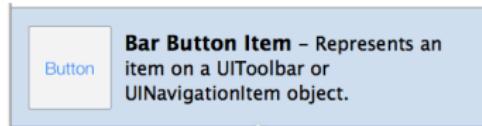


Figure : Bar Button Item

Connect the action triggered segue of this outlet to the detail view controller.

Connect the selection triggered segue of the table view controller cell to the detail view controller. Select the segue itself and change its identifier to “ExistingItem”.

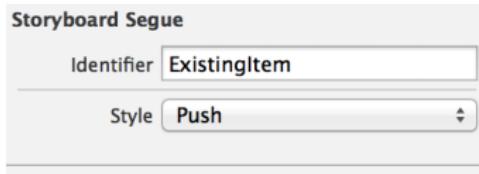


Figure : Change Segue Identifier

In the `prepareForSegue:sender:` method of the Table View Controller, check the segue to see if its identifier is equal to “ExistingItem” (`isEqualToString:`). If the identifier is equal, set the `path` property of the detail view controller equal to the path of the file that the user has selected.

Add a “Bar Button Item” to the right hand side of the detail view controller and change its text to “Save”. Connect it to an IBAction on the Detail View Controller called doSave:.

In the `viewDidLoad` method of the Detail View Controller, check to see if the `path` property has a value. If it does, set the title text field's `text` property equal to the filename in the `path`, and the body text view's `text` property equal to the contents of that file.

In the doSave: method of the Detail View Controller, check to see if the title property has a string length > 0. If it does, write the contents of the body view to a path equal to the documents directory + the name in the title view.

Add code to the table view controller to reload the table whenever the view goes from a hidden to an appearing state `viewDidAppear:`.



Figure : Put it on the App Store!

*One more thing:* Serialize the content as a JSON hash instead of a flat file. Put the title and body in the JSON file, and extract it as such when you open the file. Use string methods like  
`stringByAppendingPathComponent:`,  
`stringByAppendingPathExtension:` and  
`stringReplacingOccurrencesOfString:withString:` to format your pathnames correctly.

*One smaller thing:* When the user saves the content, tell the current navigation controller that you want it to pop the currently visible view controller off its view stack. `popViewControllerAnimated:..`

*One other thing:* Adjust the UITextView to the keyboard appearance  
*Thanks Apple!*: Apple Tech-Note on Fixing UIScrollView and Auto-Layout

We'll adjust the keyboard appearance by adding a auto layout constraint between the UITextView and its superview. Then we'll add an IBOutlet of type NSLayoutConstraint to our detail view controller, connect it with the height constraint we created, and adjust its constant value at keyboard appearance time.

## Review Concepts: Advanced Filesystem

- Build a UITableView app that loads remote content from this URL:

```
http://api.flickr.com/services/feeds/photos_public.gne?  
format=json&nojsoncallback=1&tags=indianapolis
```

- Download each image specified in the JSON, and cache it in the caches directory.
- Load the images from the filesystem when displaying them in the UITableView

# Core Data

Core Data is a way to map objects and object relationships into some persistent store. It also provides the ability to fetch data subsets from the store using “Fetch Requests”, expressions similar in function to SQL statements.

Core Data is not SQL.

Core Data can be backed by XML (Mac OS only), binary blobs on the filesystem, or SQLite.

Core Data is not SQL.

Core Data has four components that make up most of its operations:

- An object model (`NSManagedObjectModel`) which describes a schema.

Top-level components

The screenshot shows the Xcode interface with a project named "Books". In the center, the "Books.xcdatamodeld" editor is open. A vertical red line separates the sidebar from the main content area. The main content area is titled "Top-level components". Inside, there are four sections: "ENTITIES" (listing Author, Book, Publisher), "FETCH REQUESTS", "CONFIGURATIONS" (listing Default), and "Relationships". To the right of these sections, detailed entity definitions are shown.

Books.xcodeproj

Books: Ready | Today at 2:50 PM

Books

Books

BooksAppDelegate.h

BooksAppDelegate.m

MainMenu.xib

Books.xcdatamodeld

Books 2.xcdatamodel

Books.xcdatamodel

Supporting Files

Frameworks

Products

Entities

E Author

E Book

E Publisher

FETCH REQUESTS

CONFIGURATIONS

C Default

Attributes

Entity	Attribute
Author	S firstName
Author	S lastName
Book	S subTitle
Book	S title

+ -

Relationships

Entity	Relationship
Book	M authors
Author	M books

- A persistent store coordinator (`NSPersistentStoreCoordinator`) which manages operations with the persistent store.  
This object is the one responsible for writing to disk or memory.

- A managed object context (`NSManagedObjectContext`), which translates object operations into commands to be issued to the `NSPersistentStoreCoordinator`.  
This is the object which sees the most use in Core Data, as it will be responsible for handling the `save:` and `delete` calls.

- The translation between the persistent store and the logical object (`NSEntityDescription`) which can be used to generate new persistent instances of an object (  
`insertNewObjectForEntityForName:inManagedObjectContext:`  
 ) as well as query what properties belong to a given Entity.

Core Data supports automatic migration when it's able to infer how to do so correctly. Simple additions and removals are possible; complex migrations may require more complex manual steps using tools like migration identifiers or custom migrations.

You can request automatic migration from the `NSPersistentStore` coordinator by initializing it with the options

`NSMigratePersistentStoresAutomaticallyOption` and  
`NSInferMappingModelAutomaticallyOption` set to `@(YES)` in the options dictionary.

Core Data also supports the concept of “Fetch Requests”, which provide ways to query the persistent store for a set of objects. If you are using the SQLite store, these will be translated into a roughly equivalent SQL, otherwise the entire object graph and index will be inspected.

You can create precomputed fetch requests in the data model, and retrieve them later by name, or create fetch requests from strings you specify at runtime.

```
1  NSIndexPath * indexPath = [NSIndexPath indexPathForItem:0 inSection:0];
2
3  NSArray * objects = [self.fetchedResultsController.fetchedObjects
4    objectAtIndex:indexPath.row];
5
6  Person * person = objects[indexPath.row];
7
8  NSLog(@"Name: %@", person.name);
9  NSLog(@"Age: %d", person.age);
```

```
1  NSIndexPath * indexPath = [NSIndexPath indexPathForItem:0 inSection:0];
2
3  NSArray * objects = [self.fetchedResultsController.fetchedObjects
4    objectAtIndex:indexPath.row];
5
6  Person * person = objects[indexPath.row];
7
8  NSLog(@"Name: %@", person.name);
9  NSLog(@"Age: %d");
```

```
1  NSIndexPath * indexPath = [NSIndexPath indexPathForItem:0 inSection:0];
2
3  NSArray * objects = [self.fetchedResultsController.fetchedObjects
4    objectAtIndex:indexPath.row];
5
6  Person * person = objects[indexPath.row];
7
8  NSLog(@"Name: %@", person.name);
9  NSLog(@"Age: %d");
10
11  NSString * string = @"John Doe";
12  NSNumber * number = @21;
13  NSArray * array = @[@"IBM"];
14
15  NSLog(@"%@", string);
16  NSLog(@"%@", number);
17  NSLog(@"%@", array);
18
19  NSString * entityName = @"Person";
20  NSString * searchKey = @"name";
21  NSString * searchValue = @"John";
22
23  NSError * error = nil;
24
25  NSFetchRequest * preMadeRequest = [aManagedObjectModel fetchRequestByName:@""];
26  NSFetchRequest * preMadeRequestWArgs = [aManagedObjectModel fetchRequestByName:@""];
27
28  NSString * searchString = @"search string";
29  NSInteger limit = 200;
30
31  preMadeRequestWArgs.predicate = [NSPredicate predicateWithFormat:@"name = %@", searchValue];
32  preMadeRequestWArgs.fetchLimit = limit;
33
34  adHocRequest = [[NSFetchRequest alloc] initWithEntityName:entityName];
35  [adHocRequest setPredicate:
36      [NSPredicate predicateWithFormat:@"%@(age > 21) AND company IN %@", @[@"IBM"], @array]];
37
38  NSArray * results = [objectContext executeFetchRequest:adHocRequest error:&error];
```

Predicates (`NSPredicate`) are the language used to describe how a fetch request should be performed. You can test for membership in a set, general equality, range inclusion, string matching and more.

The [Predicate Programming Guide](#) in the iOS documentation offers a very detailed review of the full functionality provided by predicates.

The typical Core Data setup process looks like:

```
1  NSString * docDir = NSSearchPathForDirectoriesInDomains
2      (NSDocumentDirectory, NSUserDomainMask, YES)[0];
3  NSString * dbPath = [docDir
4      stringByAppendingPathComponent:@"example.sqlite"];
5  self.objectModel = [NSManagedObjectModel mergedModelFromBundles:nil];
6  self.coordinator = [[NSPersistentStoreCoordinator alloc]
7      initWithManagedObjectModel:self.objectModel];
8  [self.coordinator
9      addPersistentStoreWithType:NSSQLiteStoreType
10     configuration:nil
11     URL:[NSURL fileURLWithPath:dbPath]
12     options:nil
13     error:nil];
14 self.mainContext = [[NSManagedObjectContext alloc] init];
15 self.mainContext.persistentStoreCoordinator = self.coordinator;
```

`NSManagedObjectContext` is considered **NOT** thread safe; Apple recommends allocating a separate `NSManagedObjectContext` per thread if you must access Core Data concurrently.

The final result of a Core Data query is a grouping of NSManagedObjects. NSManagedObjects are the logical objects that you create when interacting with Core Data.

The properties of an NSManagedObject are what are reflected in the persistent store, and are marked as `@dynamic` in their implementation to reflect that their behavior is fulfilled at runtime.

```
1  @interface AManagedObject: NSManagedObject
2  @property(strong, nonatomic) NSString * name;
3  @end
4
5  @implementation AManagedObject
6  @dynamic name;
7  @end
```

NSManagedObjects have their properties represented by NSStrings, NSNumbers, NSDate and NSData. You can also specify special properties that are transformed using a value transformer (default is NSKeyedArchiver and NSKeyedUnarchiver) or have transient values generated at access time.

You can also specify object relationships like one-to-one, one-to-many and many-to-many. They can be unidirectional or bidirectional, and Core Data will attempt to maintain object graph consistency.

In order to initialize a new instance of a model, you need to tell the `NSEntityDescription` for a given model class to insert it using a given object context:

```
1  MyModel * m = [NSEntityDescription
2      insertNewObjectForEntityForName:@"MyModel"
3      inManagedObjectContext:myObjectContext];
```

Changes to a model are queued and recorded based on your interaction with their properties. Once you've changed their values, a call to `[myObjectContext save:&error]` will attempt to commit all your changes.

```
1  MyModel * m = self.results[0];
2  m.name = @"New Name";
3
4  MyModel * o = self.results[1];
5  m.name = @"New Name 2";
6
7  [myContext save:nil];
```

A managed object context can also delete objects using the `deleteObject:` method.

# Lab 11

# Create a new “Single View” project

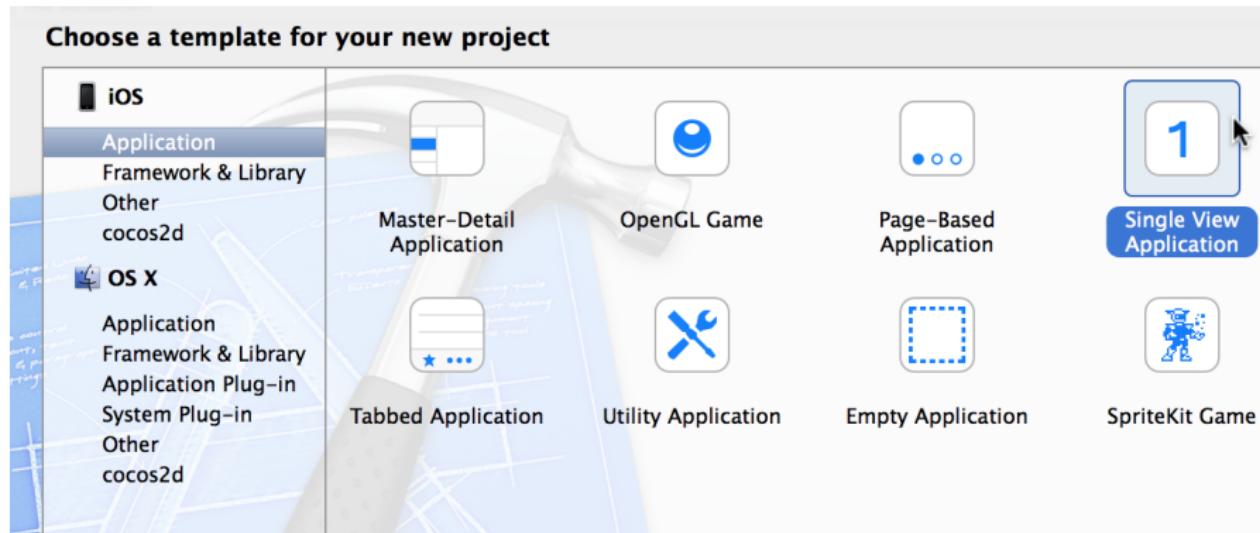


Figure : Create a new project

Open the project's storyboard and add a new "UINavigationController" to the storyboard.

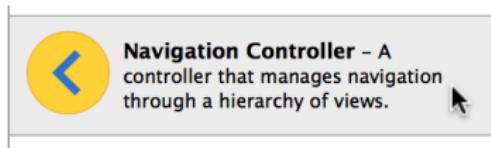


Figure : Select UINavigationController in the Object library

Drag the storyboard's "initial scene" arrow from the default view controller scene to the UINavigationController you placed.



Figure : Drag the initial scene arrow to the UINavigationController

Add a new subclass of NSObject to your project that is a subclass of “UITableViewController”.

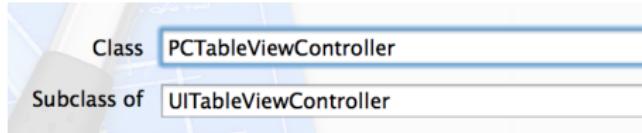


Figure : Add a new subclass of UITableViewController

Change the subclass of the UITableViewController connected to your UINavigationController in Storyboard to the UITableViewController you created.

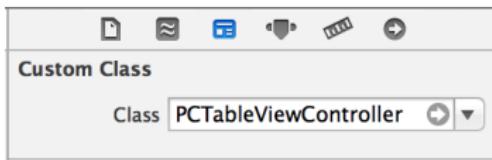


Figure : Change subclass to new UITableViewController

Add a new Core Data model to your project.

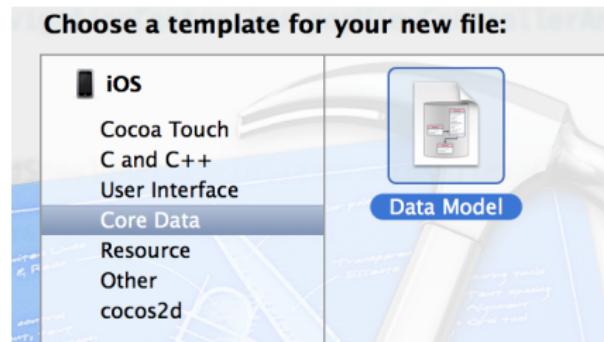


Figure : Create the Core Data model

Add a new Entity called “Person”, and add the following attributes to “Person”:

- Name (String)
- Birthdate (Date)
- Vegetarian (Boolean)
- Salary (Float)

Add a new “NSManagedObject” class to your project, and have it map to the “Person” entity you created.

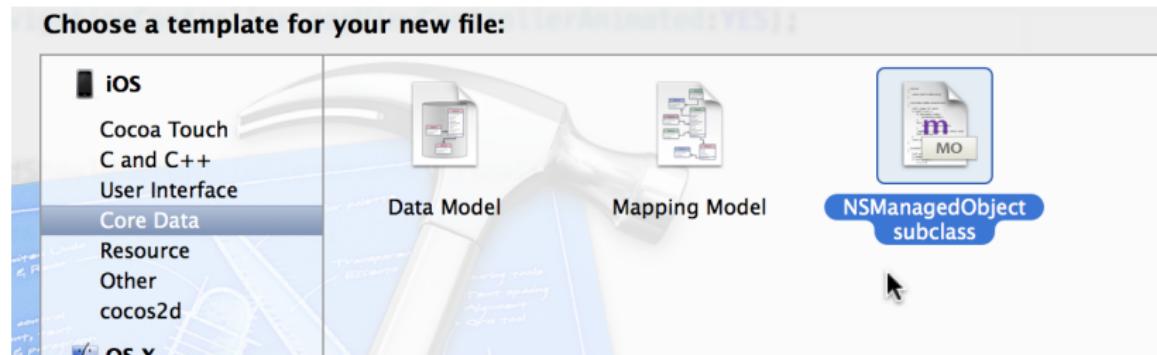


Figure : Add the NSManagedObject subclass

Create a new detail view controller, and add a UITextField (name), UISlider (salary), UISwitch (vegetarian) and UIDatePicker (birthdate) to its view.

Connect each of these controls via IBOutlets to the detail view controller.

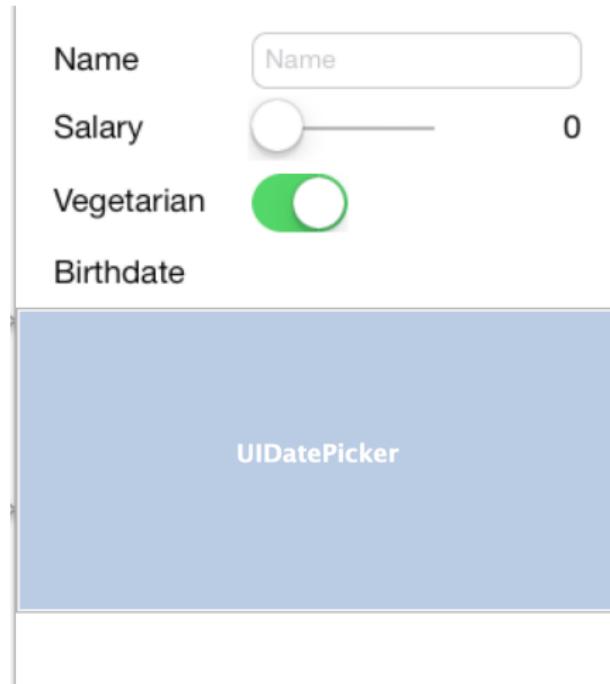


Figure 4 Example of detail view

Add a property to the detail view controller for a Person object.

Add a “Save” Bar Button Item to the right hand side of the detail view controller’s UINavigationItem.

Connect the selected segue from the UITableView of your table view controller to the detail view controller. Label the segue as “ExistingItem”.

Add a “New” Bar Button Item to the right hand side of the table view controller’s UINavigationItem. Connect it’s triggered segue outlet to the detail view controller.

Initialize your Core Data connection in your application delegate and assign the NSManagedObjectContext to a property.

In your table view controller, create a method that will returns an NSArray of all current Core Data objects:

```
1 PCAAppDelegate * appDelegate = (PCAAppDelegate *)[[UIApplication sharedApplication]
2     delegate];
3 NSFetchedResultsController * fetchedResultsController = [[NSFetchedResultsController alloc]
4     initWithFetchRequest:fetchRequest managedObjectContext:appDelegate.managedObjectContext
5     sectionNameKeyPath:nil
6     cacheName:nil];
```

You will need to access your application delegate in order to get a reference to the NSManagedObjectContext.

Change the `prepareForSegue:sender:` method of your tableview such that if the segue it is being called for has the identifier "ExistingItem", the currently selected object is passed to the detail view controller.

Change your detail view controller such that each property of the Person object is reflected by the controls in viewDidLoad.

**Note** that you will want to check if the Person object is nil before doing this. It should be assumed if Person is nil when viewDidLoad is called, then you intend to create a new person.

Create an IBAction called doSave: on your detail view controller that has code similar to the following:

```
1 -(IBAction) doSave:(id) sender {
2     Person * p = self.subject;
3     PCAppDelegate * d = (PCAppDelegate *)[[UIApplication sharedApplication]
4         delegate];
5     NSManagedObjectContext * mainContext = [d mainContext];
6     if(!p){
7         p = [NSEntityDescription
8             insertNewObjectForEntityForName:@"Person"
9             inManagedObjectContext:mainContext];
10    }
11    // sets properties from views
12    [mainContext save:nil];
13 }
```

Connect the “Save” button on the detail view controller to this method.

Change your table view controller such that it reloads its data whenever `viewWillAppear:` is called on it.



Figure : Halfway there :)

UISearchBar and UISearchDisplayController work in concert to provide an easy search results display on top of your own controls.

UISearchBar handles the user input, while UISearchDisplayController controls its own UITableView to show you specific results.

Adding the “Search Bar and Search Display Controller” to your table view controller makes it a table view delegate and datasource of the Search Display Controller as well as the table view it controls.

Drag a “Search Bar and Search Display Controller” to your table view controller in Storyboard.



**Figure :** The Search Bar and Search Display Controller

Modify your table view controller code and add an NSArray property that will hold your search results, as well as a boolean flag to indicate you are currently searching.

```
1 @property NSArray * searchResults;  
2 @property BOOL isSearching;
```

Also add the UISearchBarDelegate and UISearchDisplayDelegate protocols to your table view controller.

In the `tableView:cellForRowAtIndexPath:` and  
`tableView:numberOfRowsInSection:`, check if the `tableView` passed  
to the method ==  
`self.searchDisplayController.searchResultsTableView`.  
If it does, use the `searchResults` array to get the value you need, as  
opposed to the array of all results.

- Add the method `searchBar:textDidChange:`, and fill the `searchResults` array with the results of an `NSFetchRequest` that used that string as a search term.
- Add the method  
`searchDisplayController:didShowSearchResultsTableView:` and  
`searchDisplayController:didHideSearchResultsTableView:`, and turn the `isSearching` attribute to true when the results are shown, and false when they are hidden.
- Change your `prepareForSegue:identifier:` method to get the currently selected row from  
`self.searchDisplayController.searchResultsTableView` as opposed to `self.tableView` if `self.isSearching` is true.

Mostly there



Figure : Put it on the App Store!

*One more thing:* Add a button to the detail view to make it delete the record.

# Blocks, Grand Central Dispatch, Threading

Apple introduced blocks with Objective C as a way to make certain types of programming easier.

Blocks are functionally very similar to Ruby blocks or JavaScript function objects.

```
1 int multiplier = 7;
2 int (^myBlock)(int) = ^(int num) {
3     return num * multiplier;
4 };
5 assert(myBlock(3)==21);
```

```
1 int multiplier = 7;  
2 int (^myBlock)(int) = ^(int num) {  
3     return num * multiplier;  
4 };
```

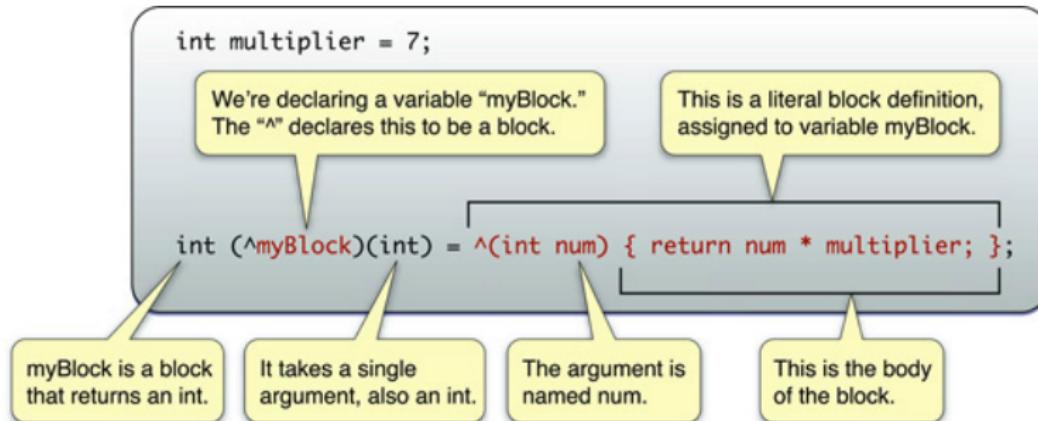


Figure : Blocks explained

Blocks may carry references to the stack and heap based on what variables were referred in their body. This can cause unpredictable memory effects; in order to help guide ARC, you can use variable qualifiers like `_weak` and `_strong` to indicate the type of ownership a variable you would like to use in a block should have.

```
1  __weak MyClass * weakSelf = self;
2  void (^weakRefBlock)() = ^{
3      if(!weakSelf) return;
4      return [weakSelf tryMethod];
5  };
```

Blocks may be stored in ivars, passed as method arguments, and generally treated as just another C-type.

See: <http://fuckingblocksyntax.com/>.

Grand Central Dispatch was introduced along with blocks with the primary goal of making concurrency easy.

GCD adds the concept of serial and concurrent queues that process blocks on available threads.

**Serial** queues process blocks one-by-one in the order that they are received. Only one serial queue is created for you (the main queue), but you can create others with `dispatch_queue_create`.

**Concurrent** queues process an undefined number of blocks concurrently. Four concurrent queues with different priority levels are created for you (HIGH, DEFAULT, LOW, BACKGROUND). You cannot create concurrent queues.

GCD also has several concurrency structures like semaphores (`dispatch_semaphore_t`), groups (`dispatch_group_t`, for batching) and barriers (`dispatch_barrier_t`, for concurrent synchronization).

GCD also provides `select()` like functionality that can trigger blocks whenever an IO object is ready to be read from / written to.  
`(dispatch_source_create)`

A common pattern you'll see in concurrent code is performing a double dispatch, wherein code is executed in a background thread, and then the results submitted to the UI on the main thread:

```
1  dispatch_async(dispatch_get_global_queue(
2      DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
3      NSString * result = PerformAnExpensiveOperation();
4      dispatch_async(dispatch_get_main_queue(), ^{
5          self.resultsLabel.text = result;
6      });
7  })
```

# Lab 12

Create a table view controller and hook it up. :)

In your viewDidLoad method, add code that does the following (replacing the comments with actual code):

```
1  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
2      // synchronously request data from a json url
3      // decode it
4      dispatch_async(dispatch_get_main_queue(), ^{
5          // assign the decoded results to a property
6          // reload the table view
7      });
8  });
```

Run the app!



Figure : Put it on the App Store!