Bootstrapping iOS Application Development

Day 1

Chris Zelenak

12/02/2013

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 Single View based application in XCode called Lab1

File > New > Project

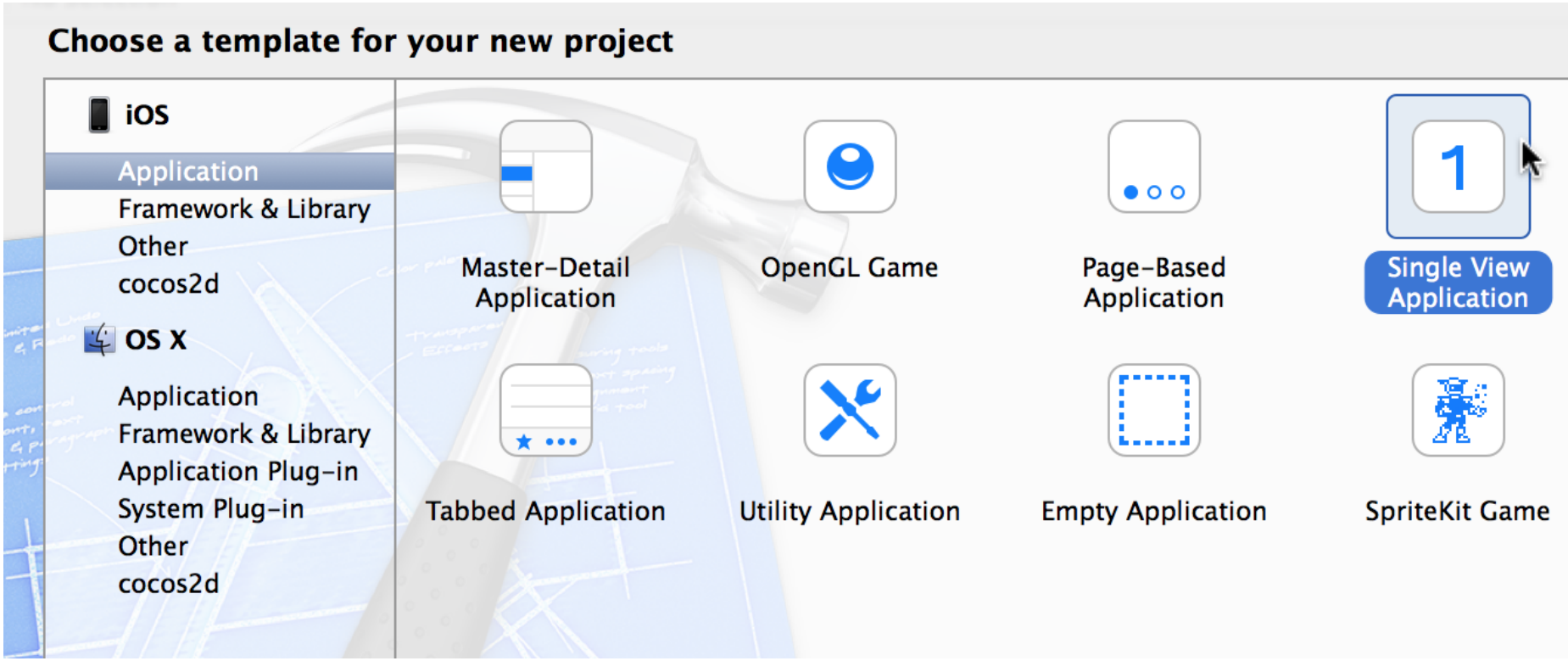


Figure : New Projects

Open Main.storyboard in Interface Builder

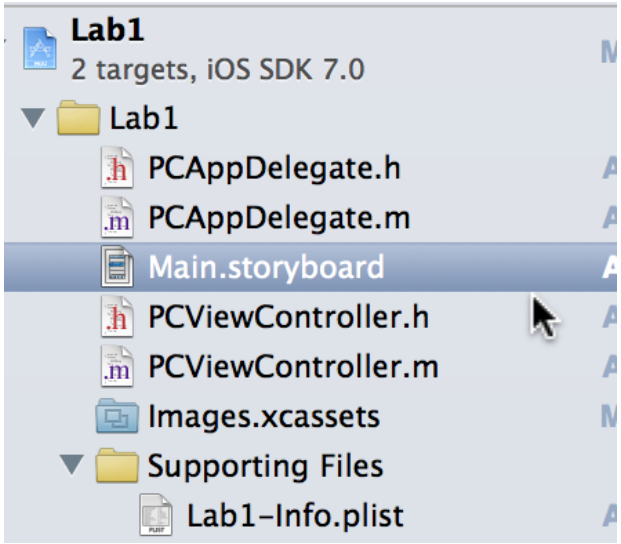


Figure : Main Storyboard

Add a button with the text “Hello iPhone”

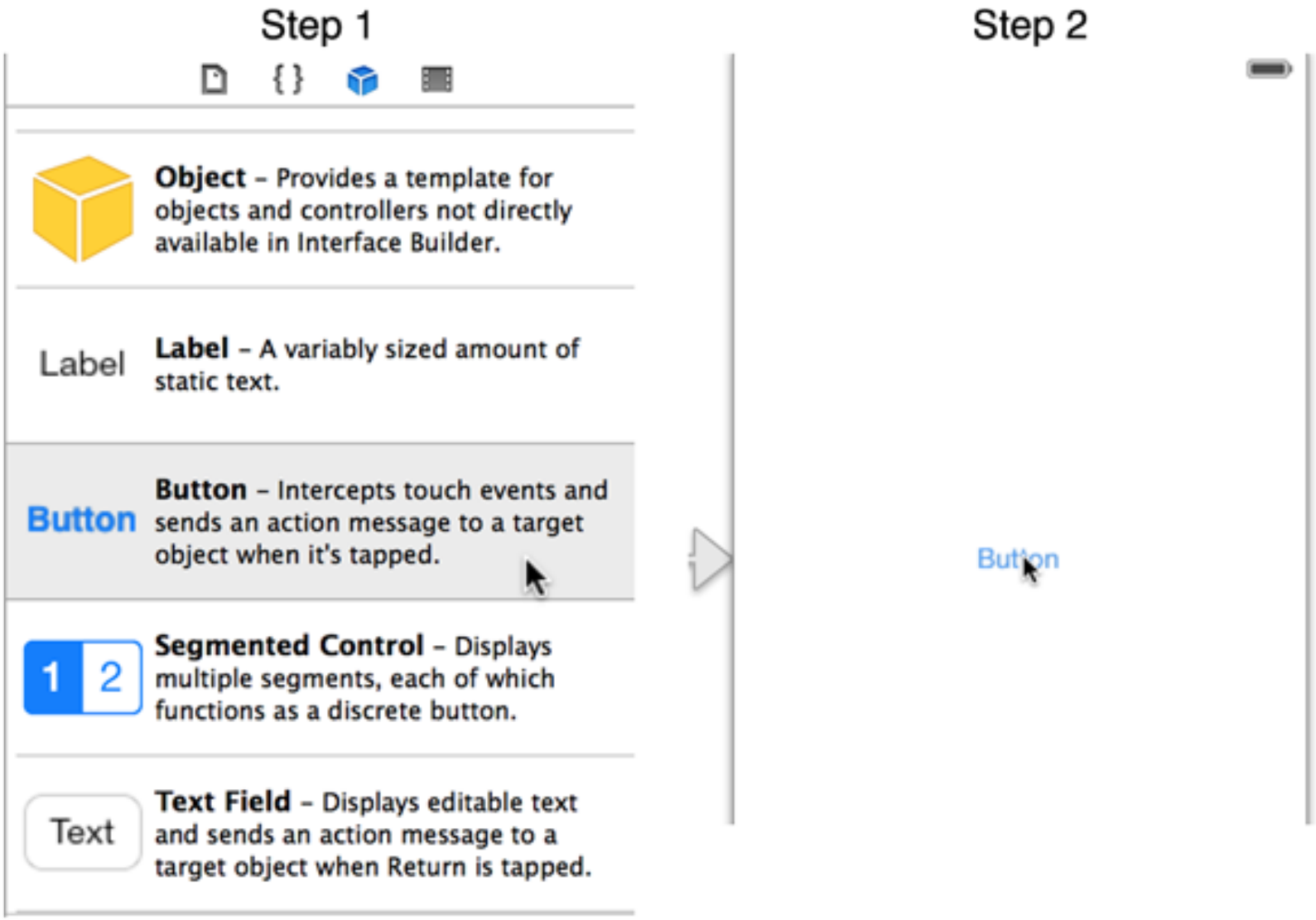


Figure : Drag Button to Storyboard

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.

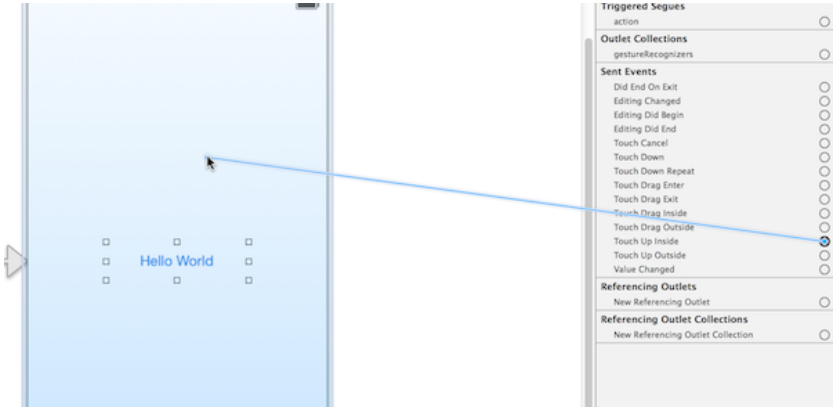


Figure : Connect Touch Up Inside event to IBAction

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

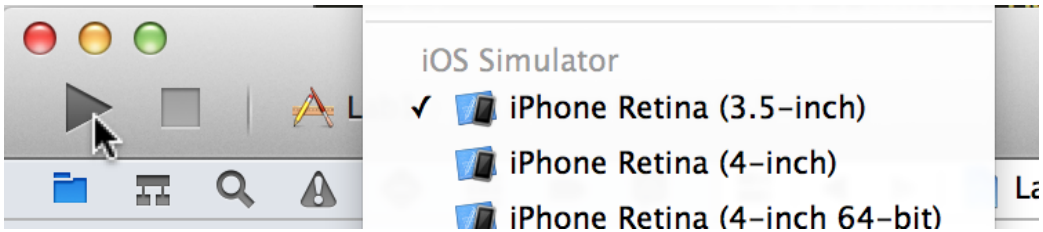


Figure : Launch Simulator

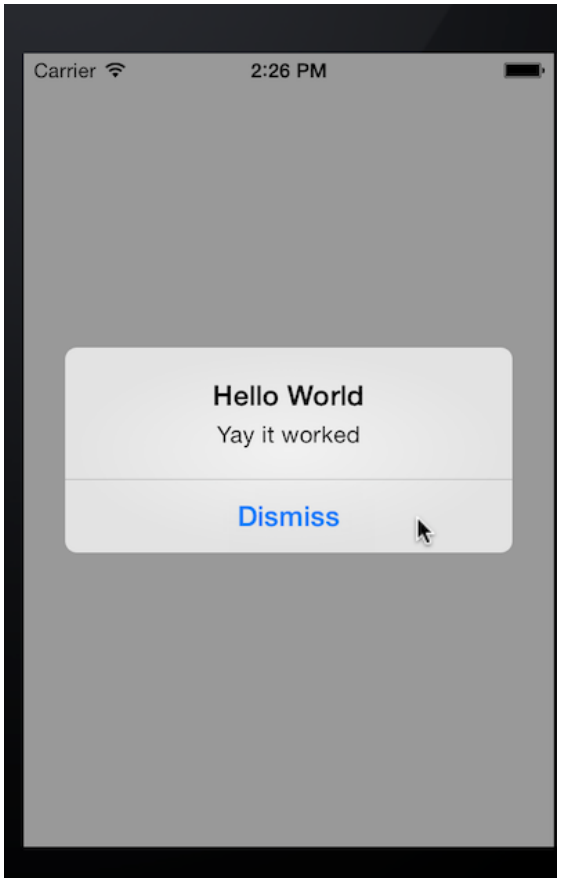


Figure : It Worked!

1 Objective C

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 dynamicly 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.

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 pointer to a read-only char array
```

These are items you'll commonly encouter 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
```

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 };
```

```
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;
```

```
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
```

```c
#include <stdio.h>
#define FOO 1
#define IS_ZERO(n) (n == 0)

int main(){
  printf("FOO is %i\n", FOO);
  if(IS_ZERO(0)) {
    printf("This code is very useful.\n");
  }
}
```

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.

```
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 %@ was case insensitive equal", anExampleString);
11  }
```

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.

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   }
```

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.

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
```

```
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 %@ and %f",
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 %@ and %f",
14            mo.instanceVariable,
15            mo.numericVariable);
```

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

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 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     }
```

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   }
```

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
```

```
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    }
```

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
```

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

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.

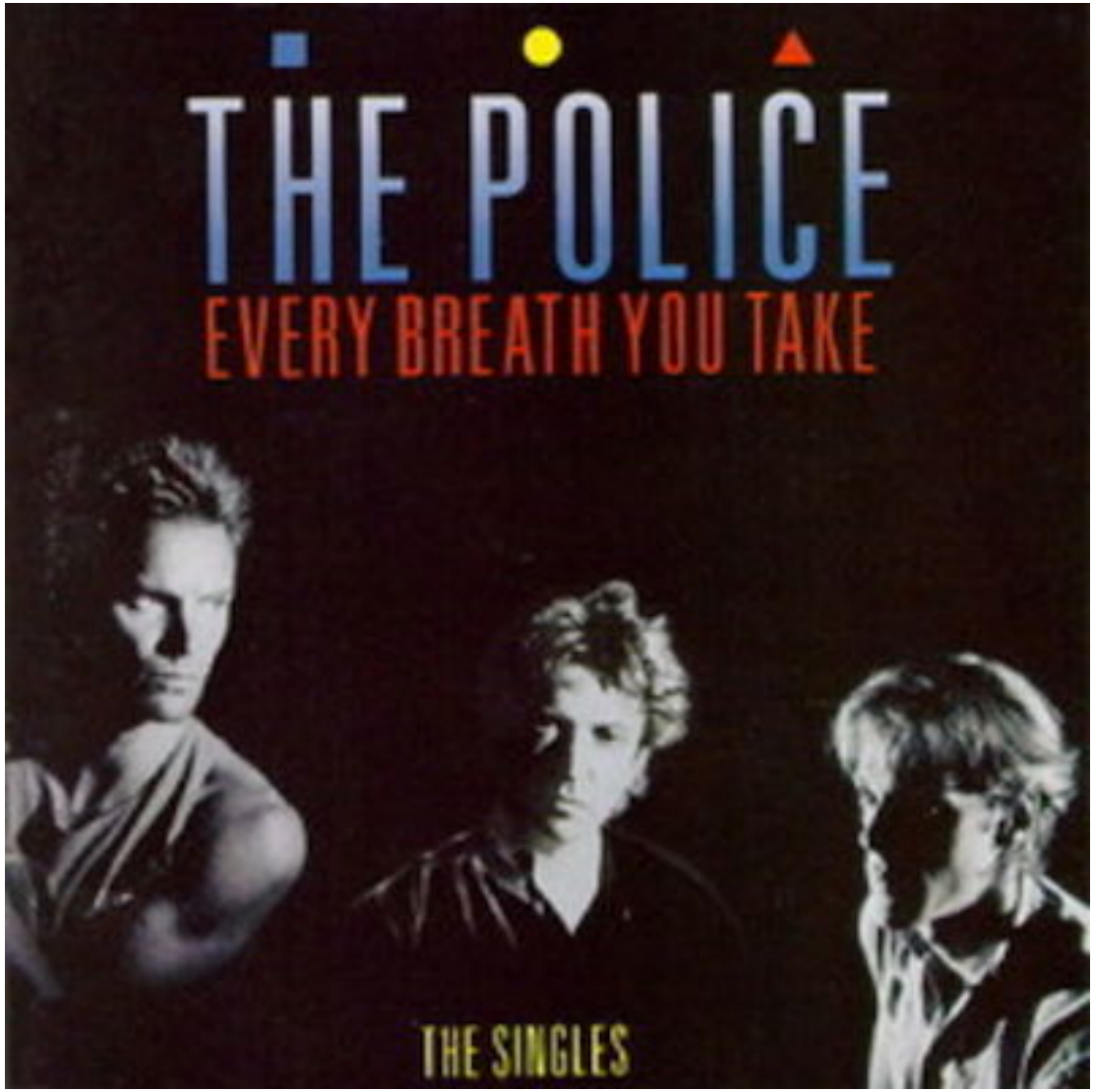


Figure : Memory Management circa 1983

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).

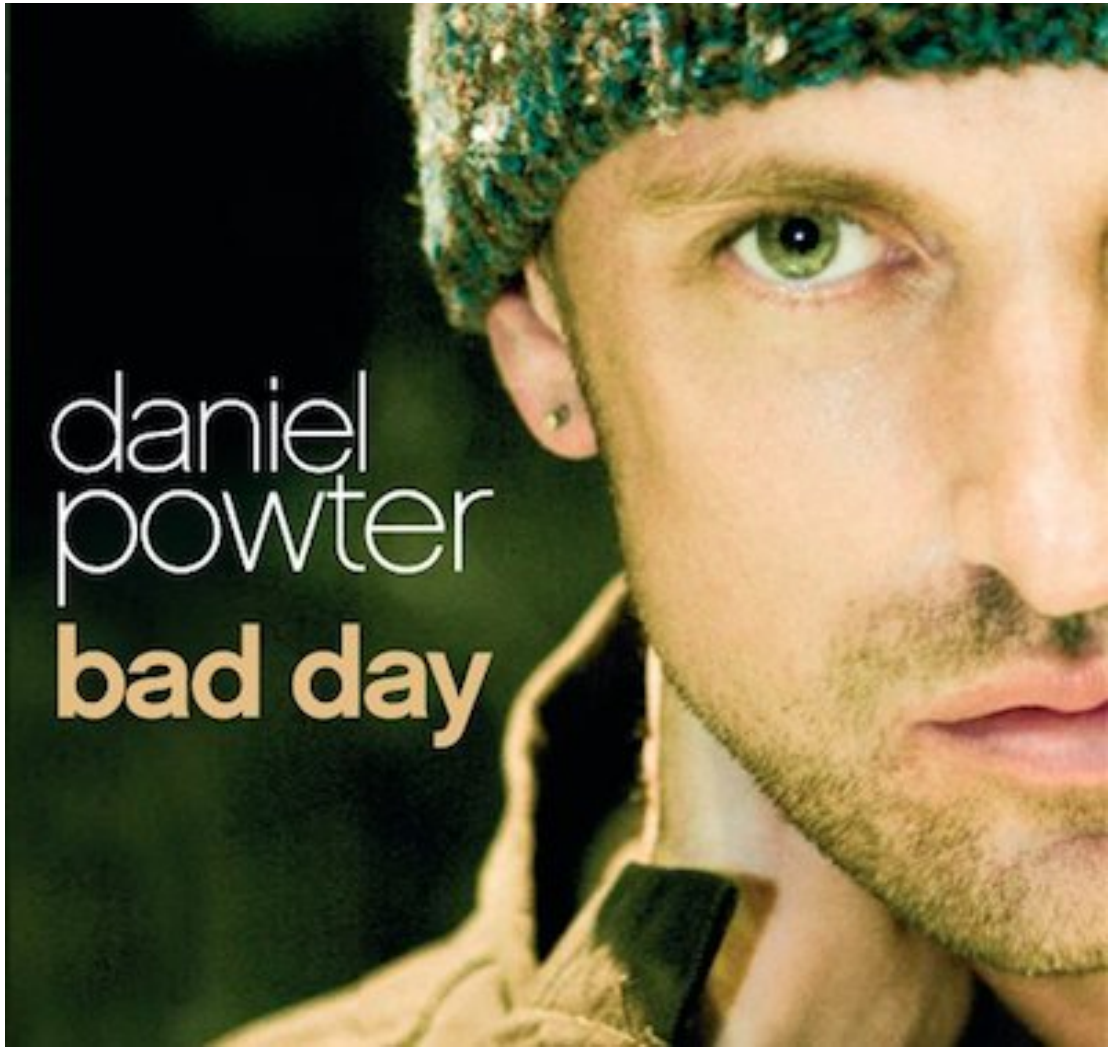


Figure : Memory Management circa 2006 - Apple adds Garbage Collection to Objective C

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.

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 Layout Conventions

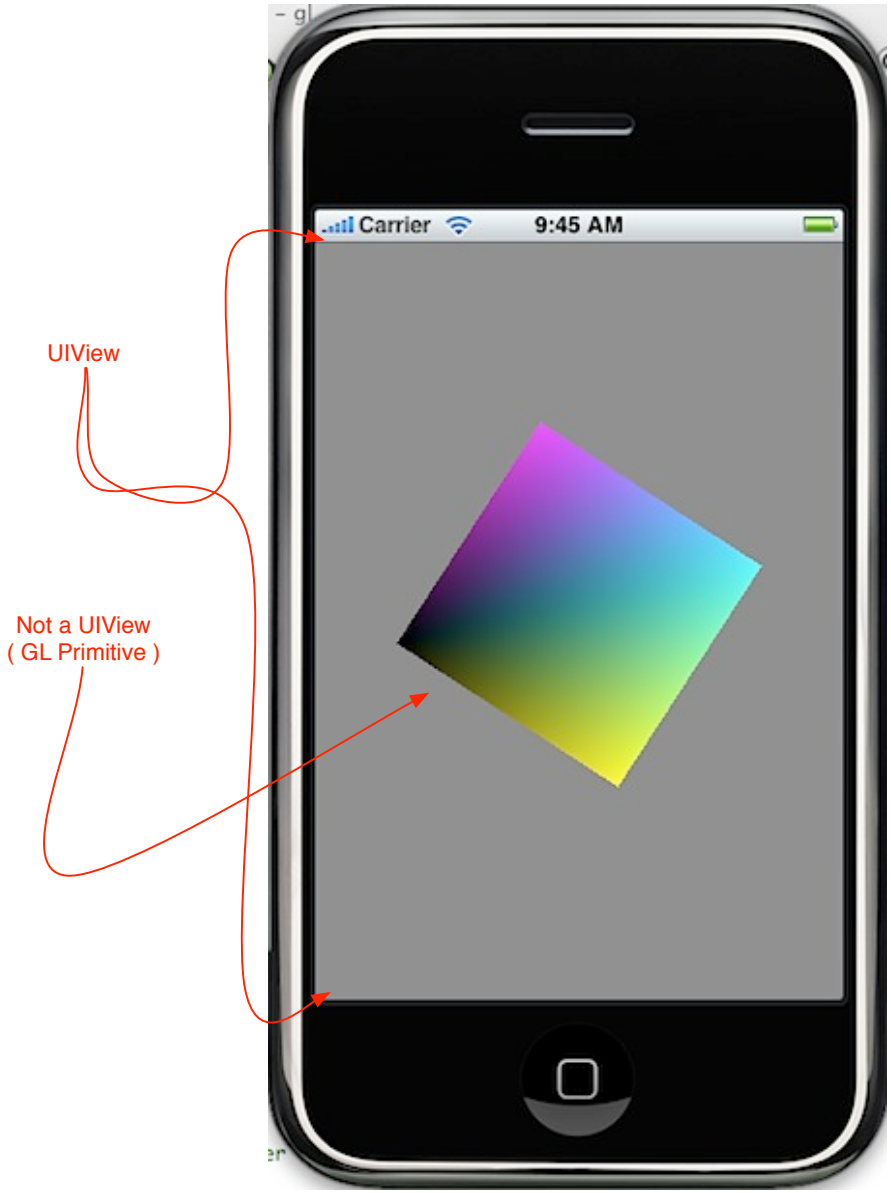
Apple strongly encourages you to adhere to Model View Controller pattern

Figure : Model, View and Controller

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).


UIView
1:09 PM
CNN
Top Stories
World
U.S.
Politics
Crime
Breaking news
Flight 1549; Airplane crash-lands into Hudson River
How a 'perfect storm' led to the economic crisis
How long does Obama have to fix the economy?
My CNN
Video
iReport


UIView
Not a UIView
(GL Primitive)
Carrier
9:45 AM

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

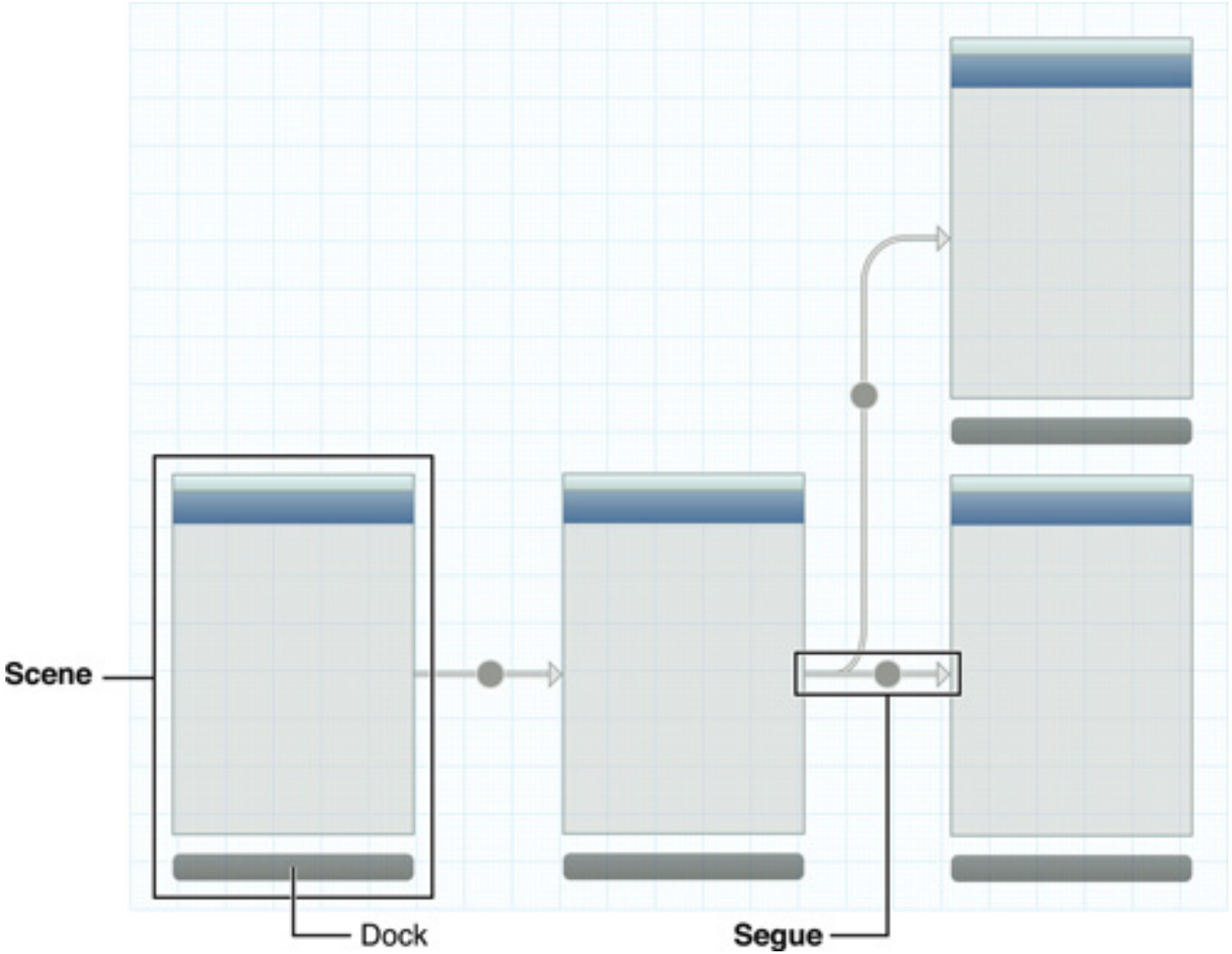


Figure : The Storyboard designer

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.

Create a new “Single View” project

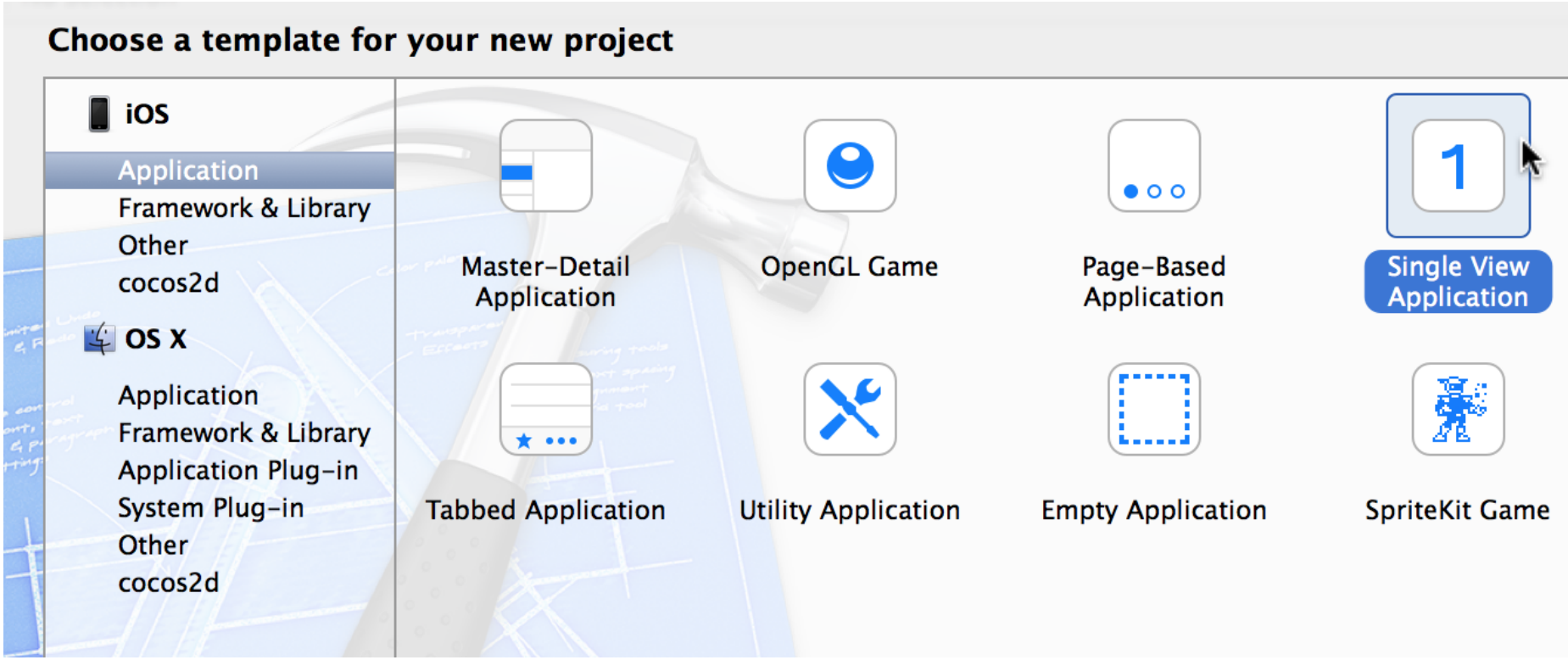


Figure : Create a new project

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

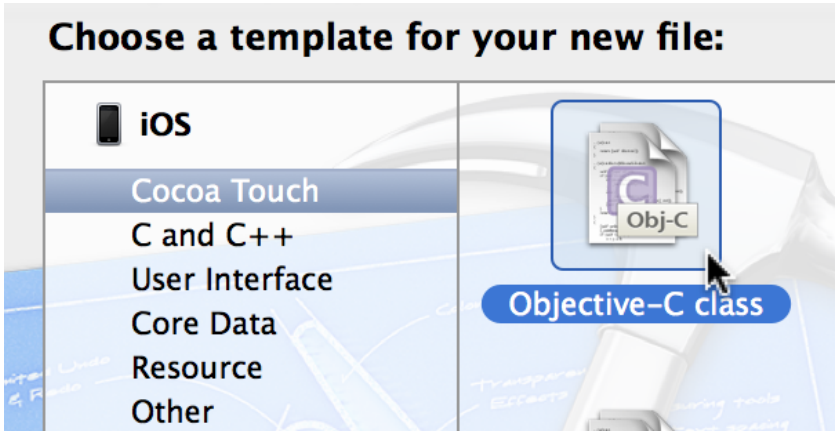


Figure : New Objective C class

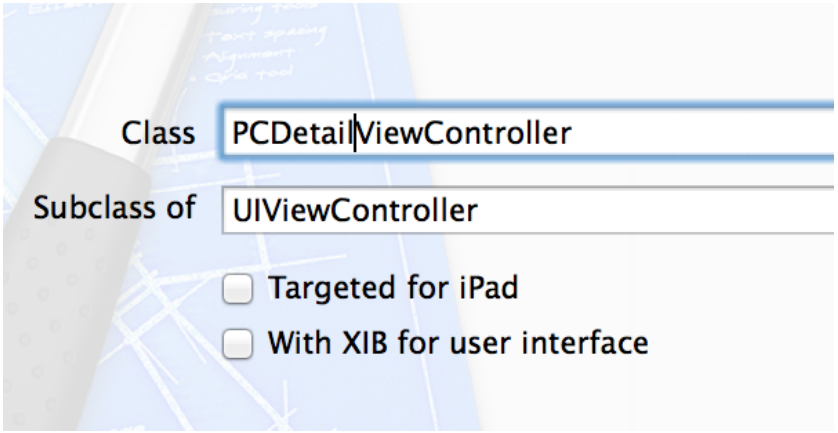


Figure : Subclassing UIViewController

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

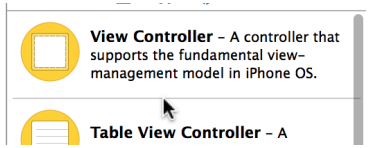


Figure : The UIViewController object in the Library

Figure : Drop UIViewController on Storyboard

Set the subclass of the UIViewController to “DetailViewController”.

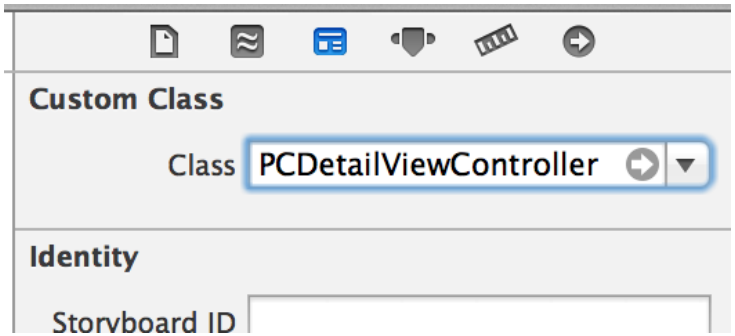


Figure : Setting the UIViewController subclass

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

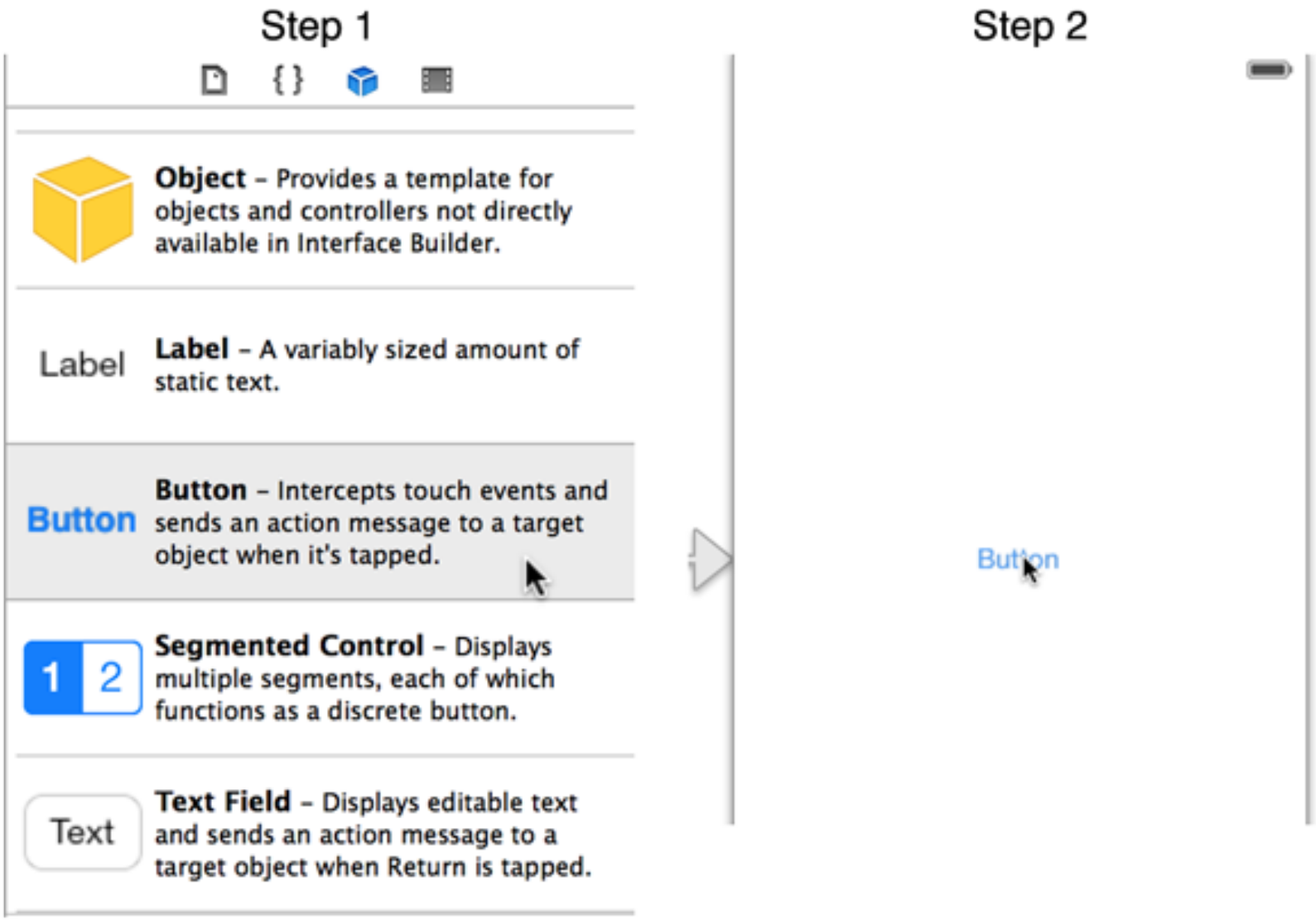


Figure : Drag a button to the scene

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.

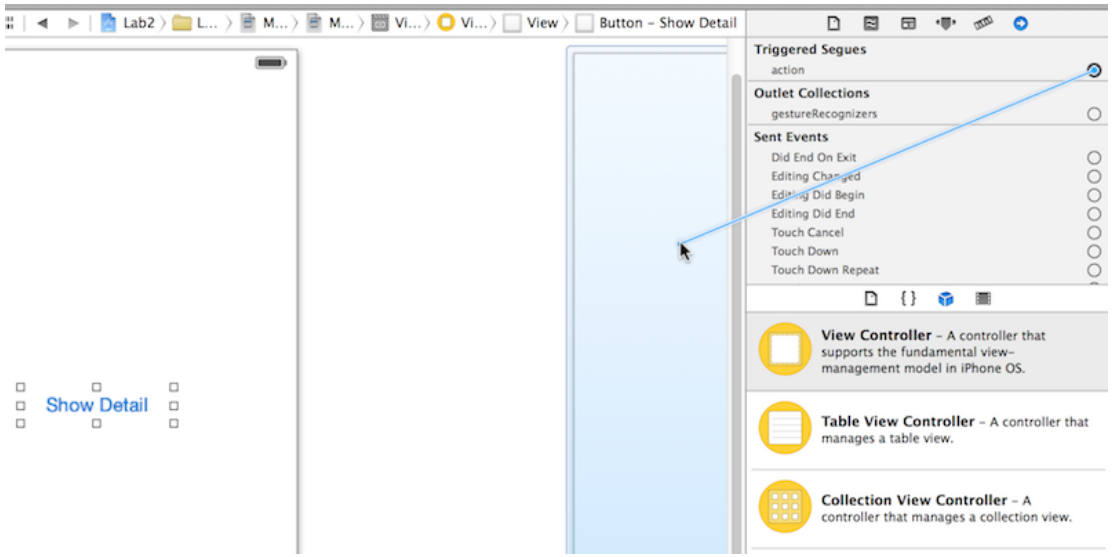


Figure : Connect the action to the Detail View Controller

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.

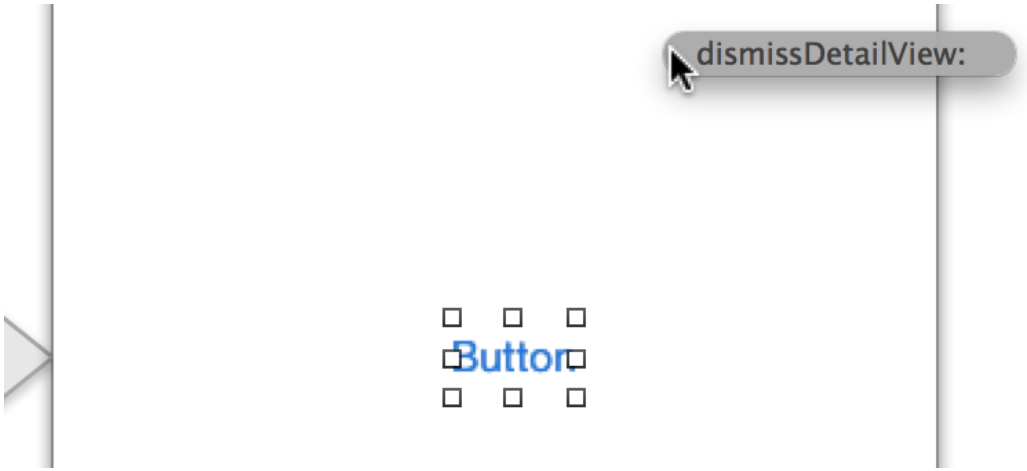


Figure : Connect the action to the dismiss selector

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 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.

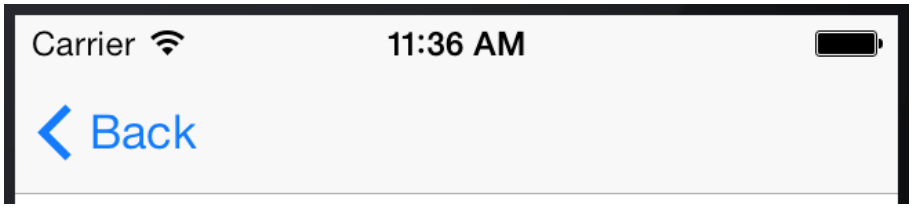
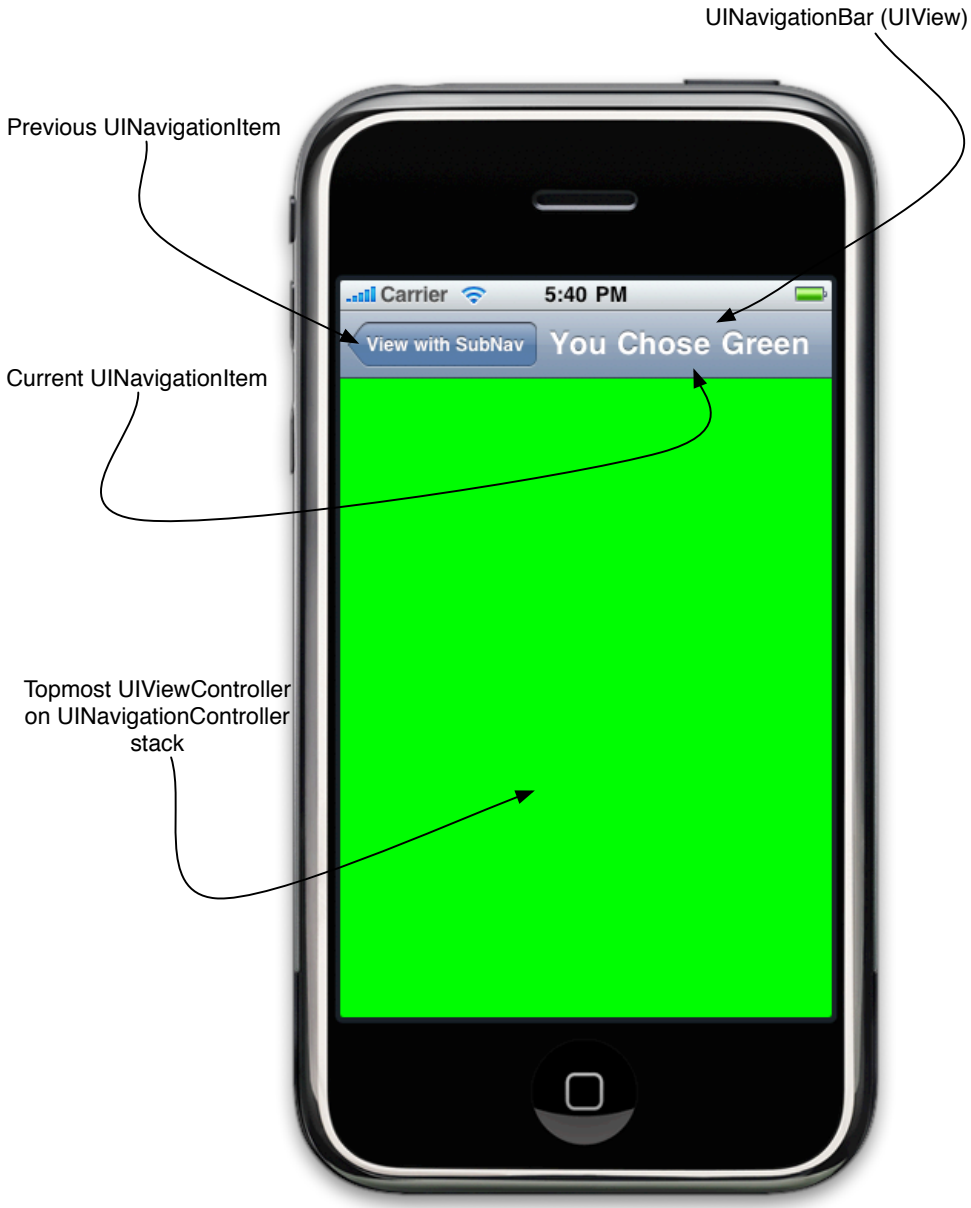


Figure : UINavigationController at work


UINavigationBar (UIView)
Previous UINavigationItem
Current UINavigationItem
Topmost UIViewController on UINavigationController stack
Carrier
5:40 PM
View with SubNav
You Chose Green

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.

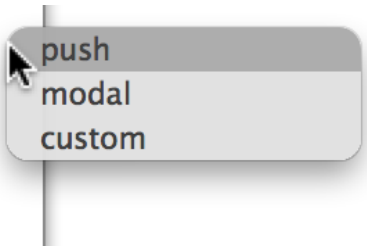


Figure : Selecting the push segue

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 UIViewControllers on the stack manually.

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

Create a new “Single View” project

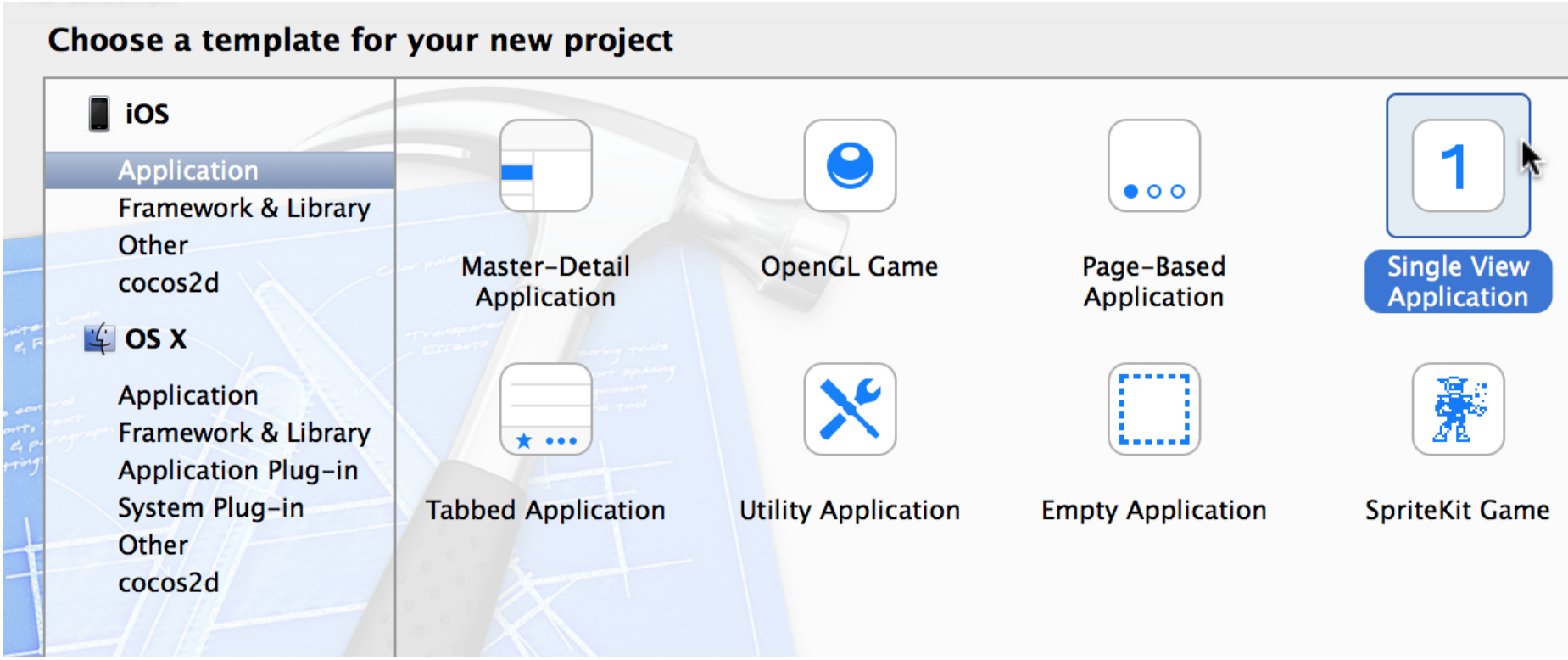


Figure : Create a new project

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

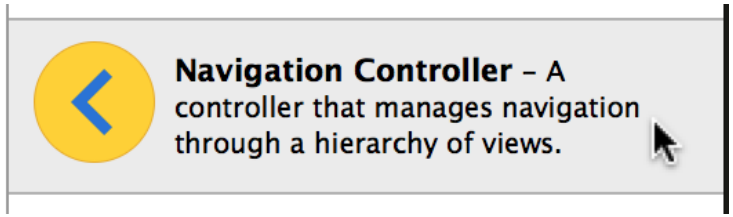


Figure : Select UINavigationController in the Object library

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.

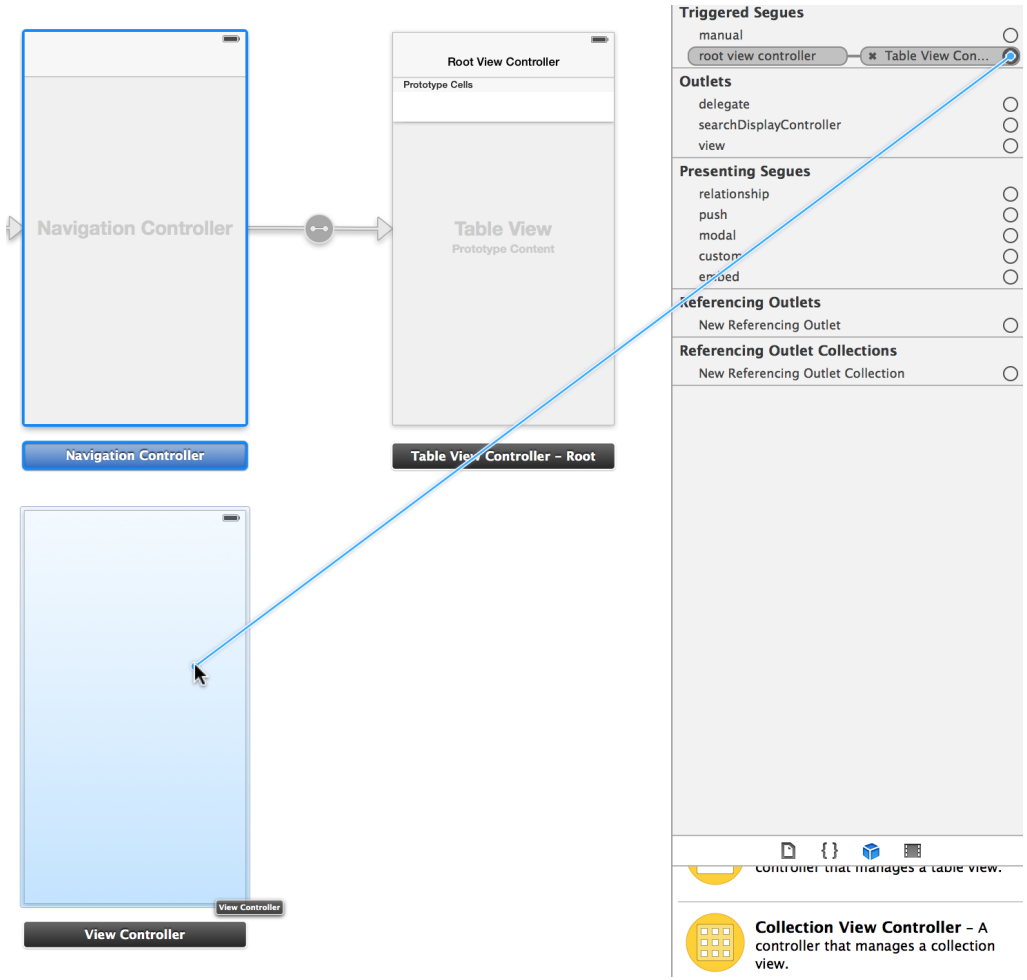


Figure : Change the root view controller outlet

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

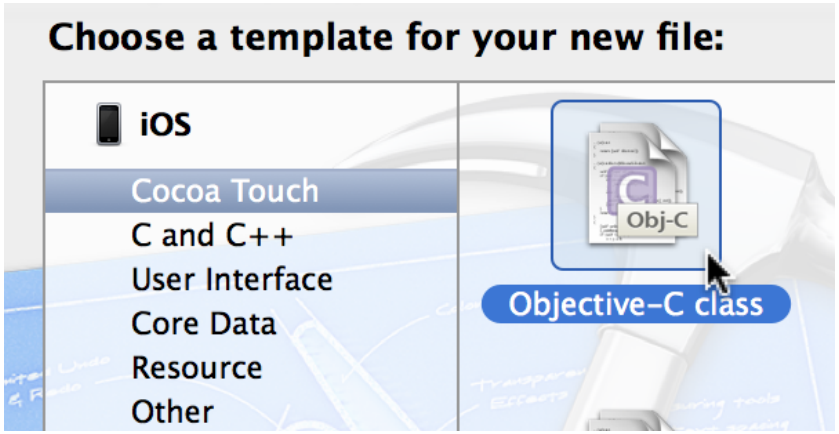


Figure : New Objective C class

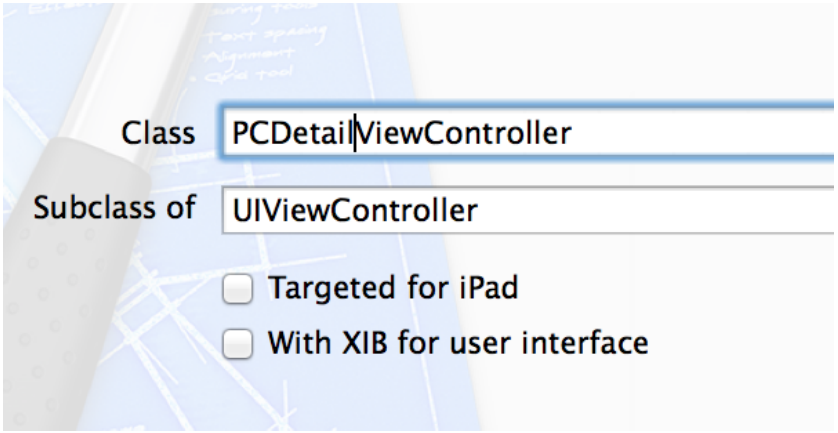


Figure : Subclassing UIViewController

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

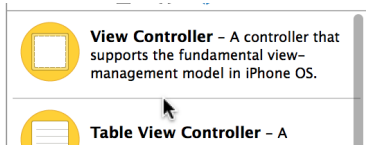


Figure : The UIViewController object in the Library

Figure : Drop UIViewController on Storyboard

Set the subclass of the UIViewController to “DetailViewController”.

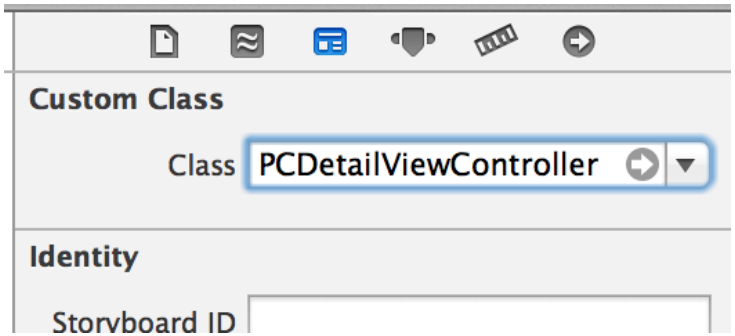


Figure : Setting the UIViewController subclass

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

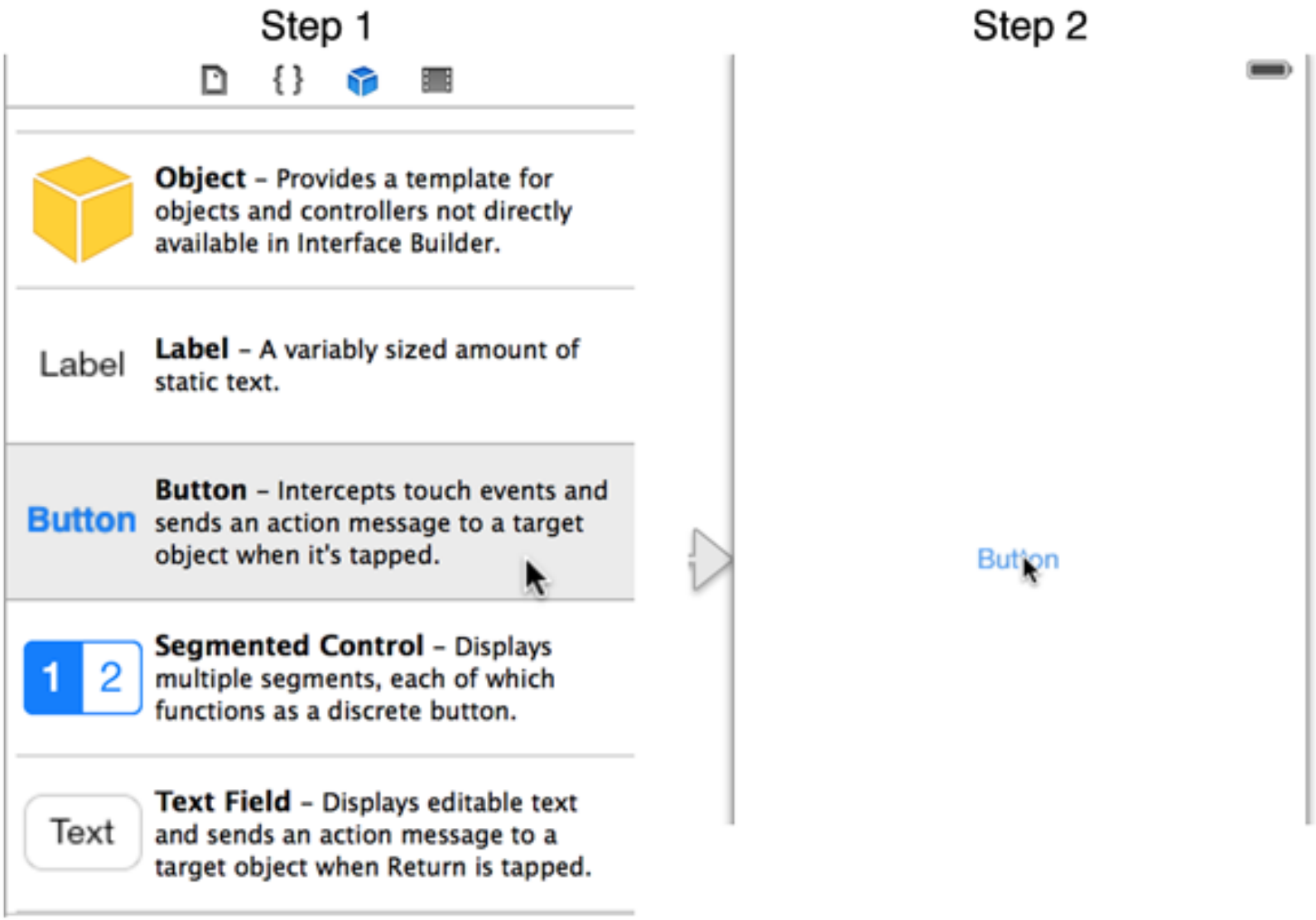


Figure : Drag a button to the scene

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.

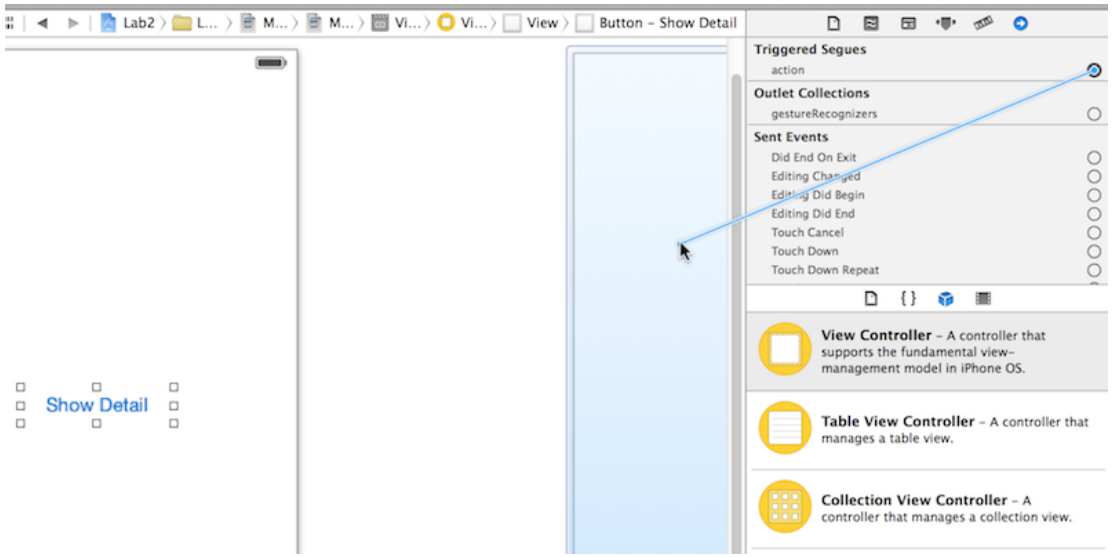


Figure : Connect the action to the Detail View Controller

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

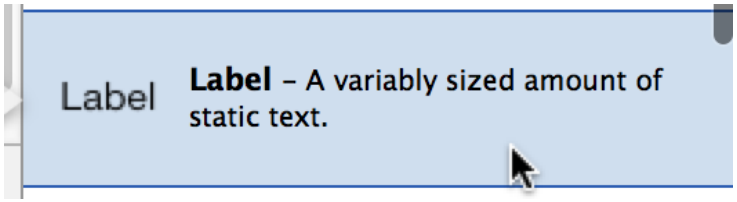


Figure : Select and place the Label control

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

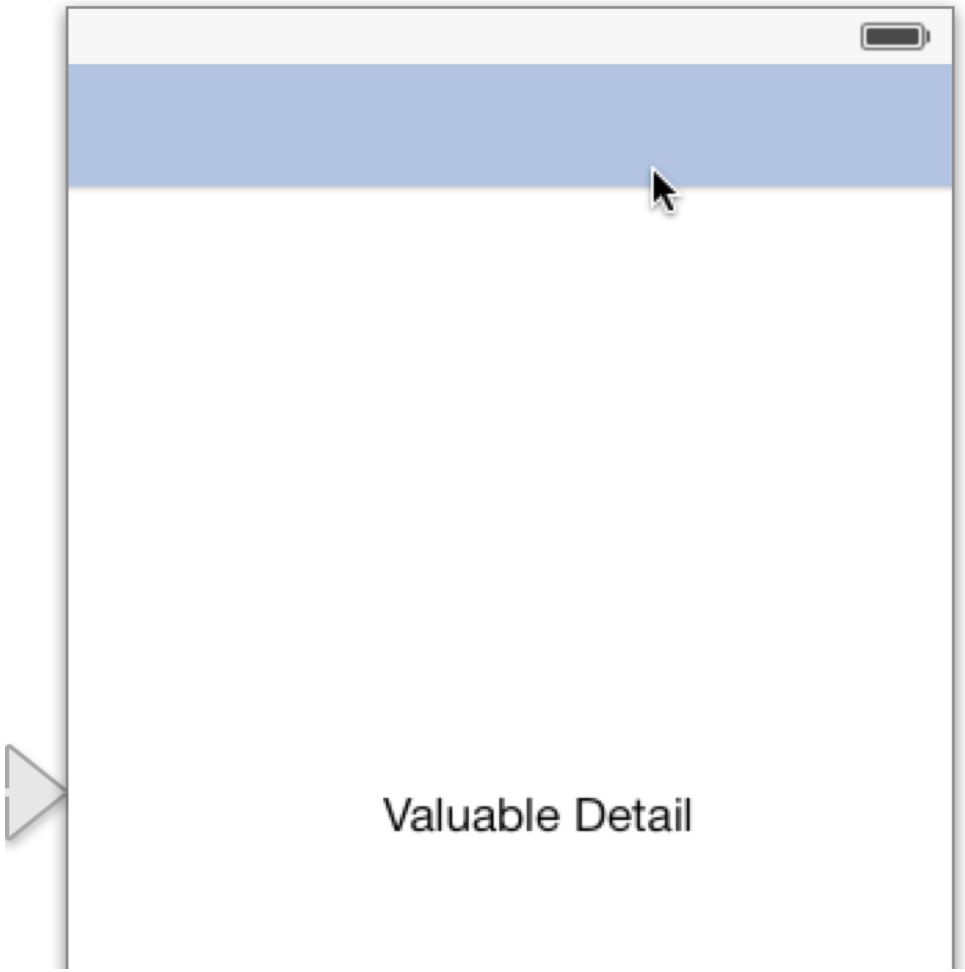


Figure : Select UINavigationItem in controller

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.

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.

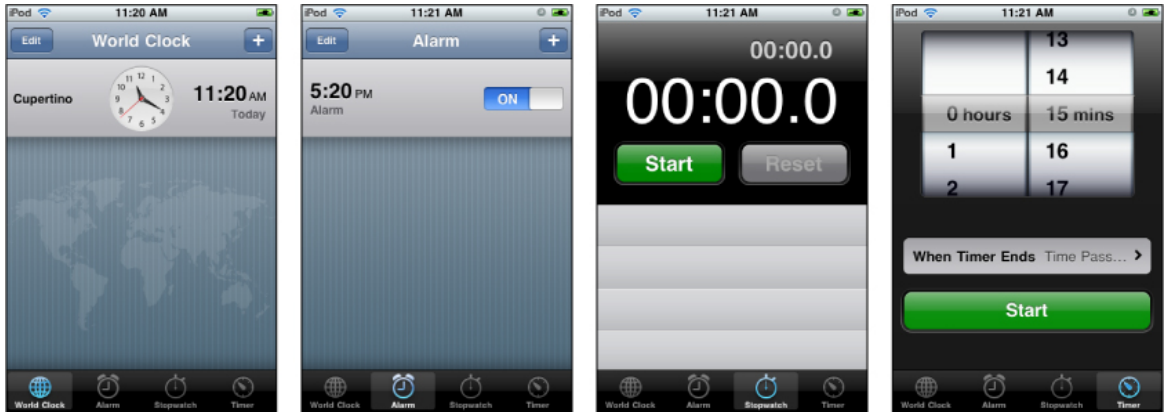


Figure : UITabBar examples

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

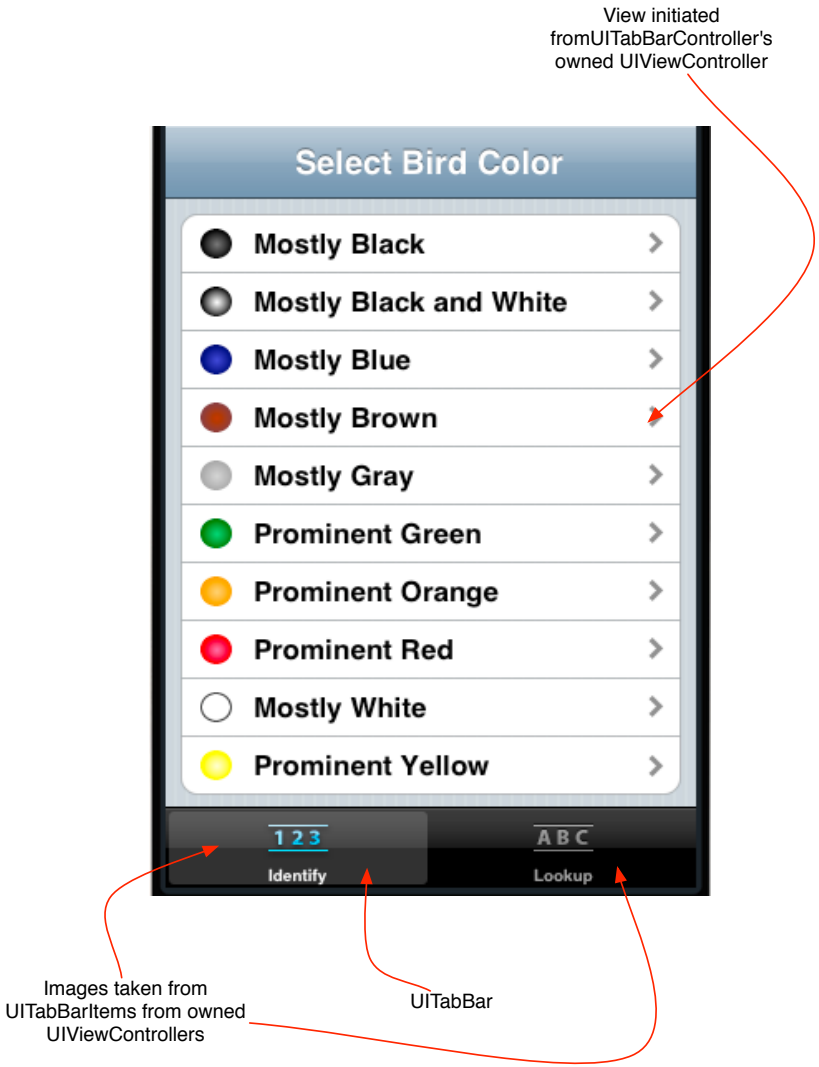


Figure : UITabBarController example

Figure : Sir Not Appearing In This Class

UITableViews 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.

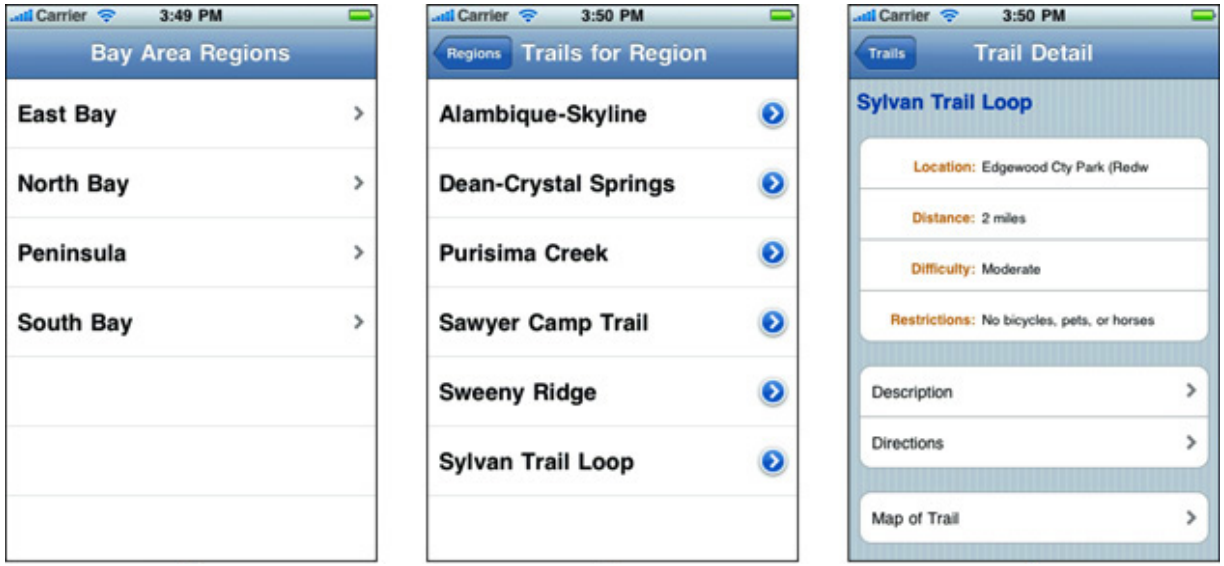


Figure : UITableView examples

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.

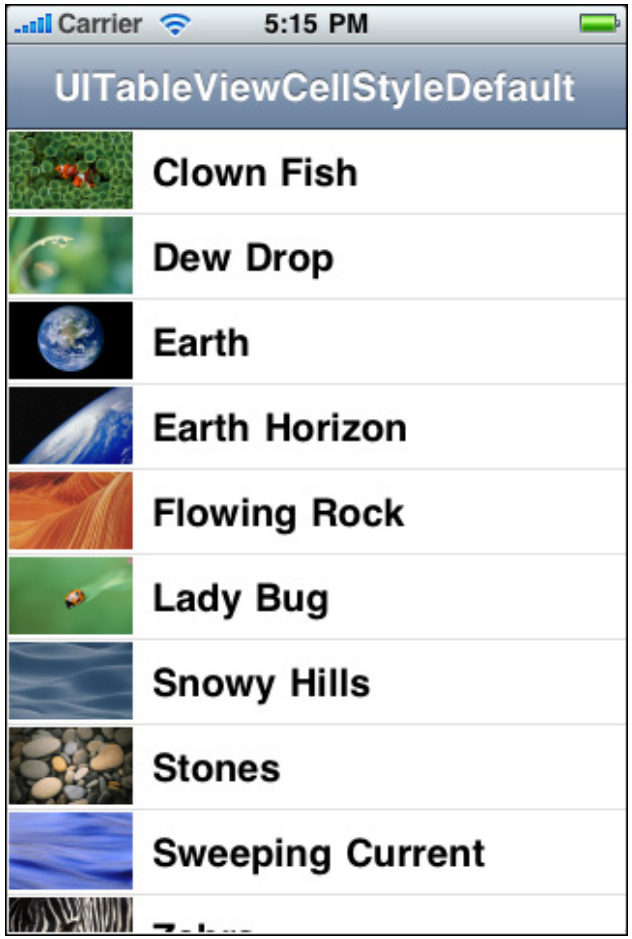
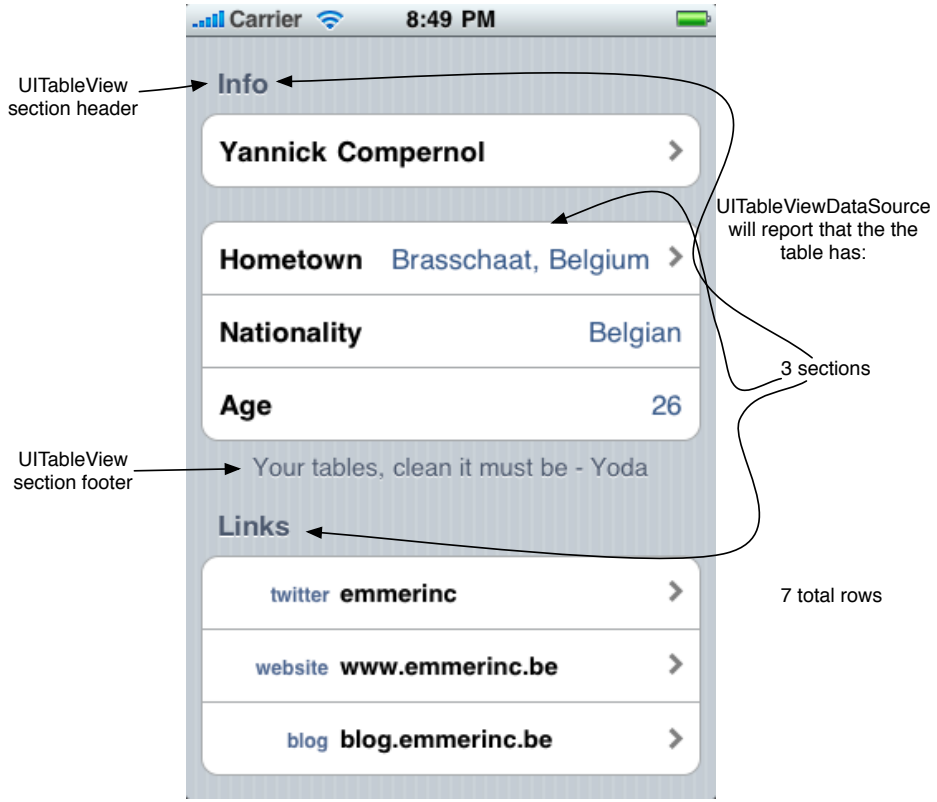


Figure : UITableViewCell default style

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.



Create a new “Single View” project

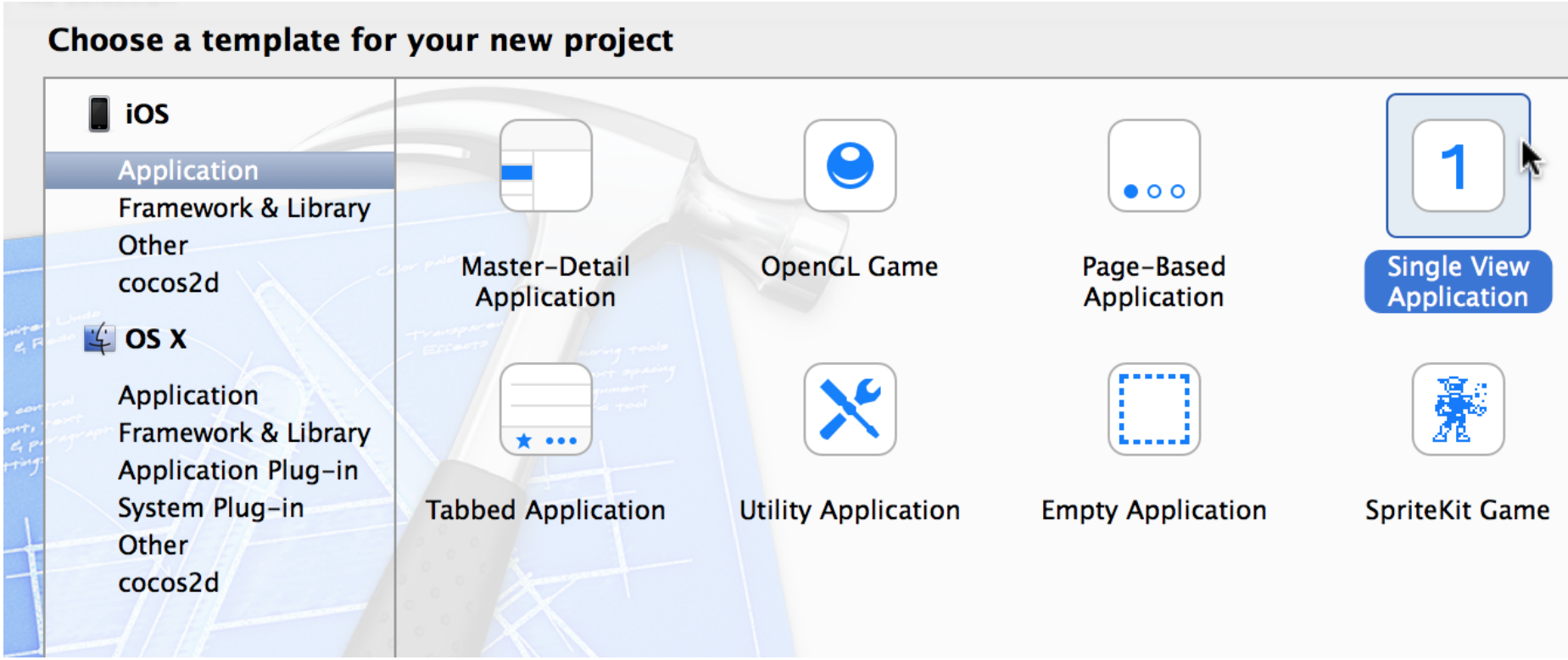


Figure : Create a new project

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

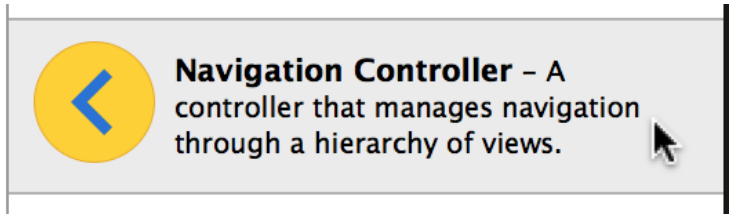


Figure : Select UINavigationController in the Object library

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”.

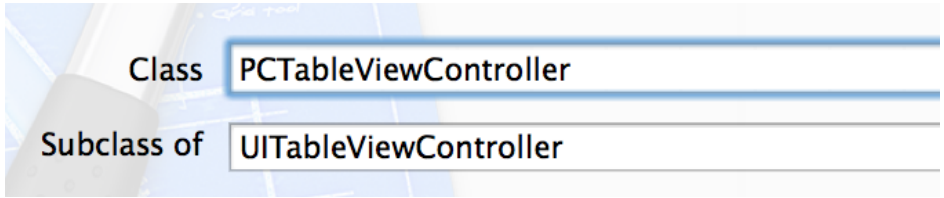


Figure : Add a new subclass of UITableViewController

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

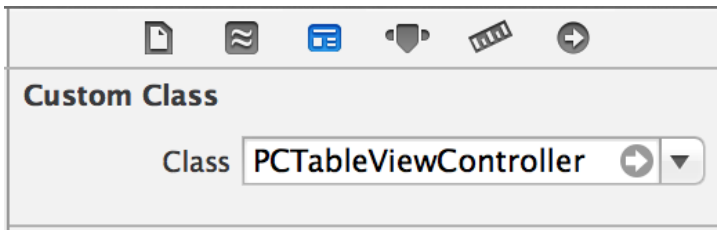


Figure : Change subclass to new UITableViewController

Add the following code to PCTableViewController.h:

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

Add the following code to PCTableViewController.m:

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

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”.

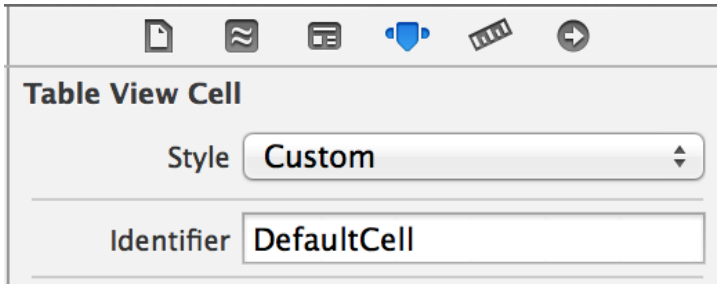


Figure : Change Reuse Identifier

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.

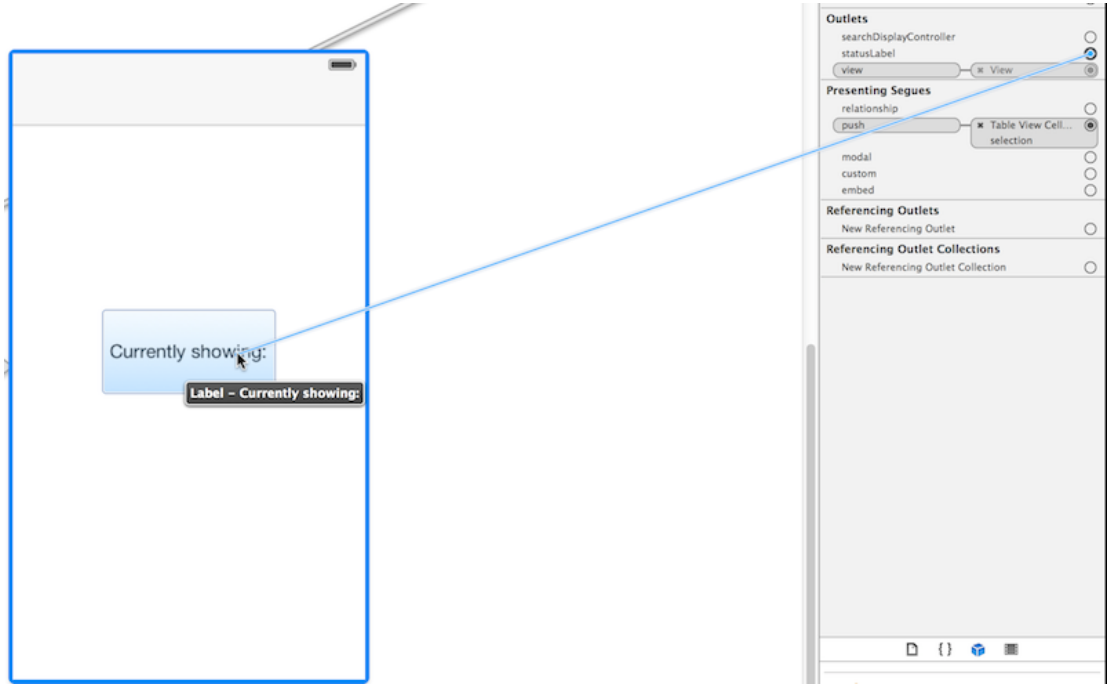


Figure : Connect the UILabel to the nameLabel outlet

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”.

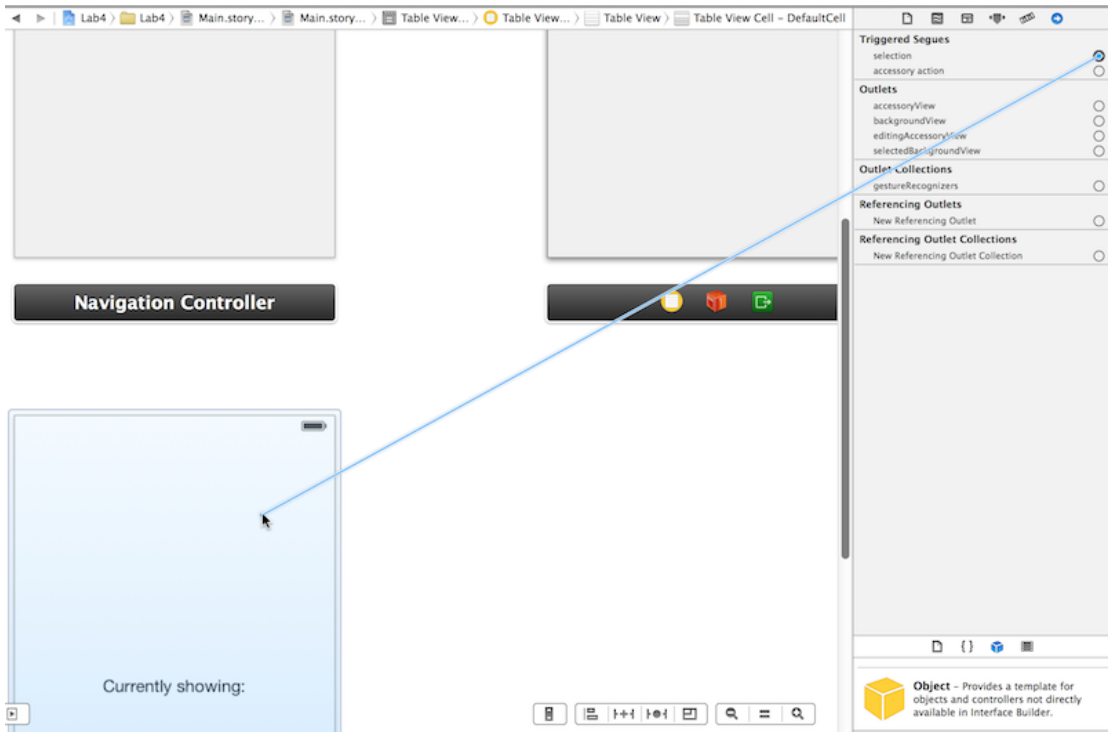


Figure : Connect the selection segue of the UITableViewCell

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.

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.

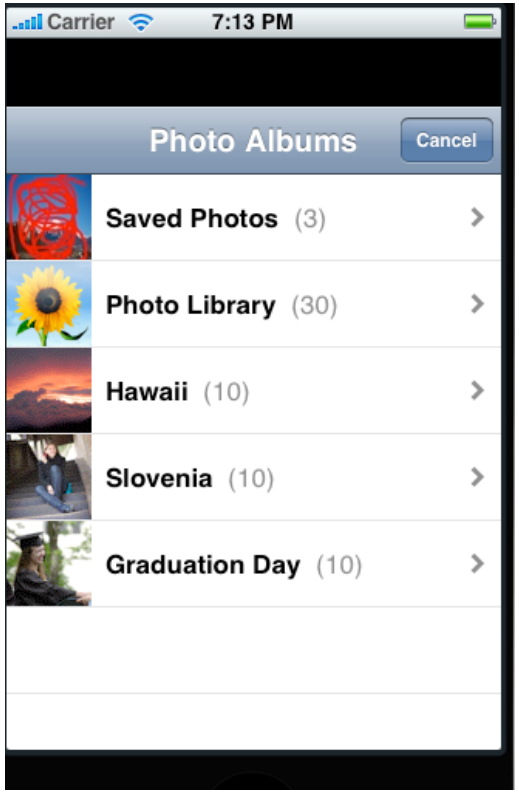
Objective C formal protocols are typically how a delegate can indicate at compile time that it provides certain functionality. See UITableViewDelegate and UITableViewDataSource

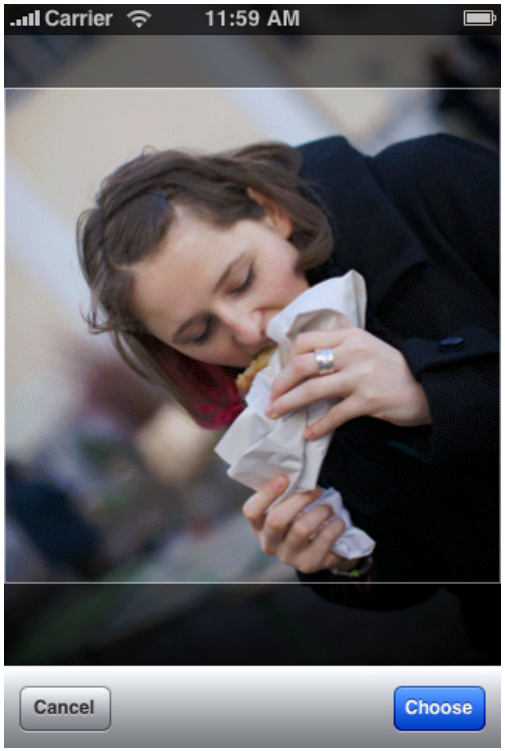
The delegate pattern is used widely throughout Apple frameworks.

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

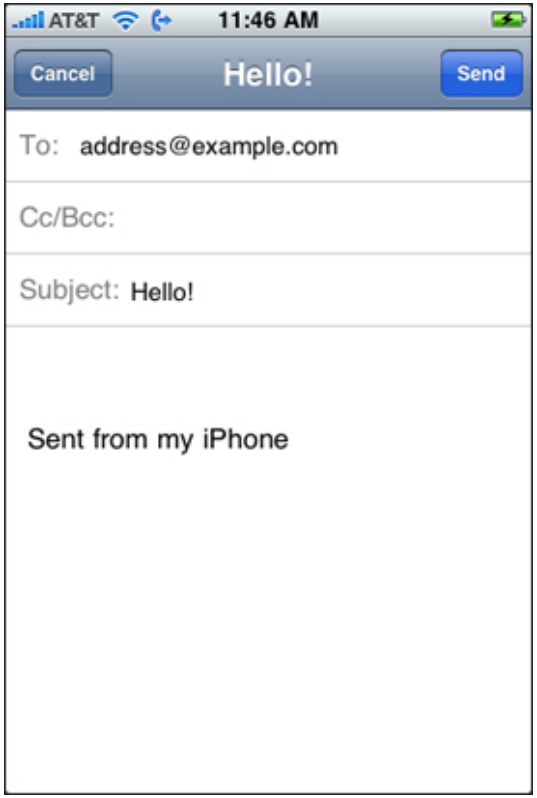
Figure : Delegate usage throughout the documentation

Not all view controllers fit within the “display in UINavigationController” or “display in UITabBarController” model.


Carrier
7:13 PM
Photo Albums
Cancel
Saved Photos (3)
Photo Library (30)
Hawaii (10)
Slovenia (10)
Graduation Day (10)


Carrier
11:59 AM
Cancel
Choose

MFMailComposeViewController



You can present a view controller modally by using the `presentModalViewController:animated:` message

```
1   -(IBAction) buttonClicked:(id) sender {
2     MyViewController * mvc = [[MyViewController alloc]
3                                       initWithNibName:@"TheNib"
4                                       bundle:nil];
5     [self presentModalViewController:mvc animated:YES];
6     [mvc release];
7   }
```

You can specify the transition style with which the view controller appears via the UIViewController message

```
setModalTransitionStyle:(UIModalTransitionStyle)style
```

1 XCode and Interface Builder

XCode and Interface Builder combined provide a development, debugging and interface development tool for you to develop your iPhone apps. It can integrate with SVN, CVS or Perforce (and soon, Git!), it can run automated tests on your code, provide inline documentation assist, code completion and a lot more. It also provides a developer level interface to all the devices you will be debugging your code on; you can manage devices and device profiles through XCode as necessary.

XCode provides a number of application templates for starting an app. Most basic types of applications can get a fast start by using one of these app templates.

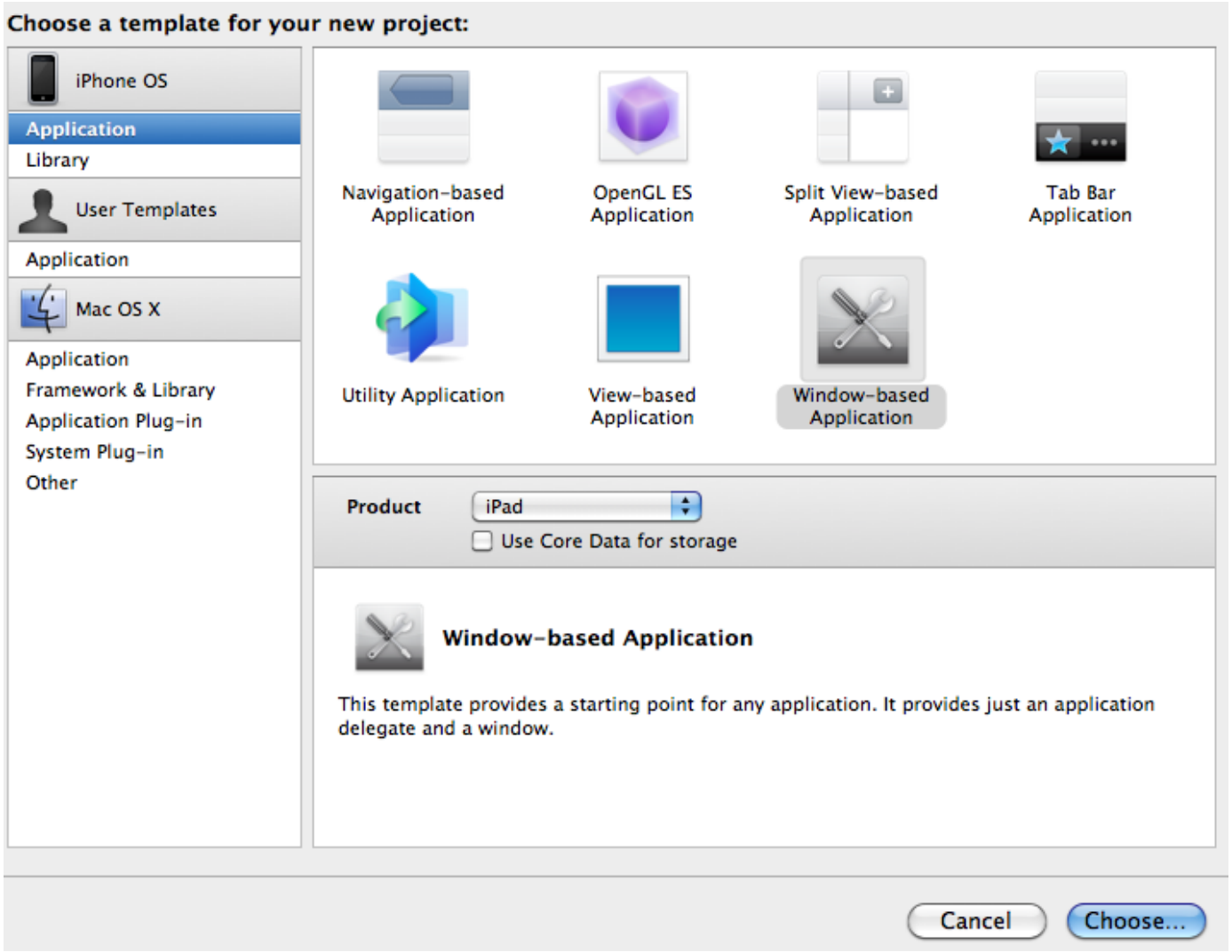
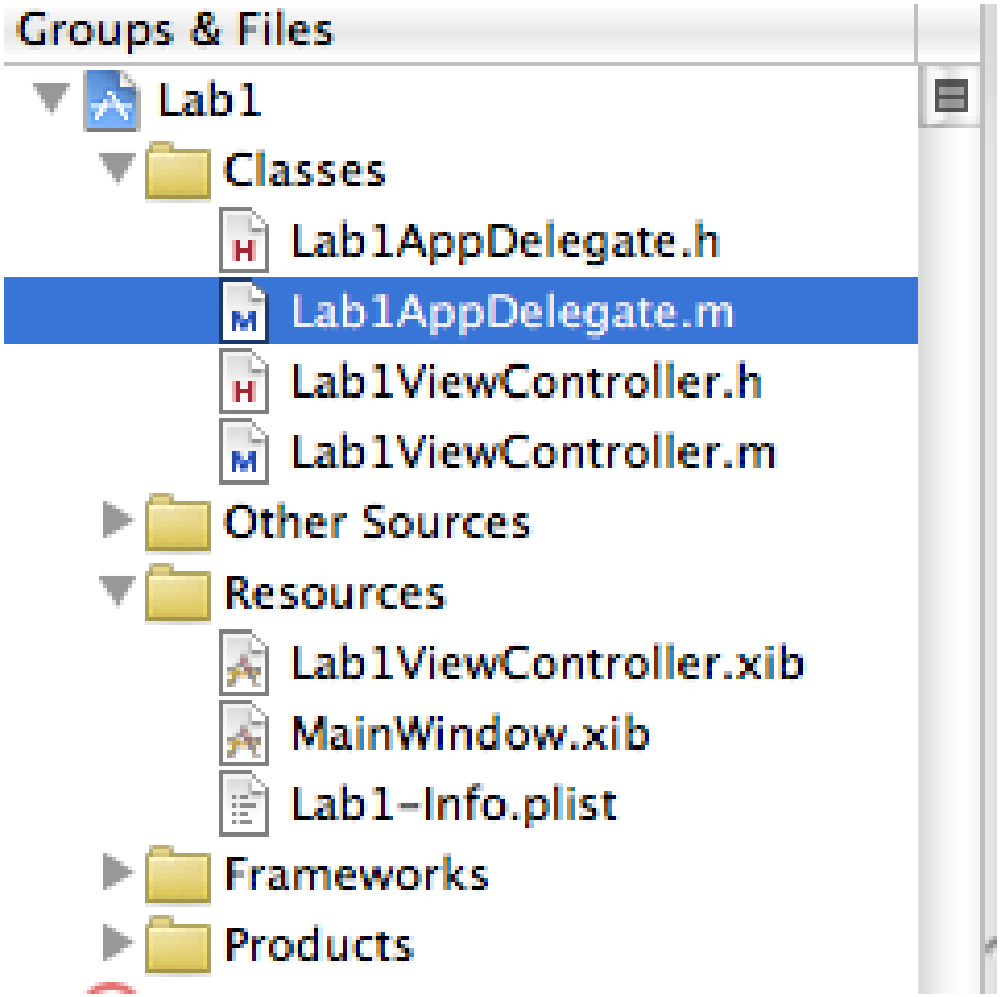


Figure : XCode New Project window

Once you've selected the type of app you want to create, your project will be created with an application delegate and usually at least one view controller that will manage what goes on the screen. It will also have a .xib file for your application, and for each view controller in the app, that is used by Interface Builder to layout the screen contents.



Double clicking on a .xib file will open it in Interface Builder, where you can drag components from the Interface Builder toolbox into the view. You can easily change the orientation of objects.

Help - Developer Documentation is an invaluable resource for researching Cocoa internals.

Learn hotkeys! (CMD+SHIFT+D and CMD+0 will buy you hours)

Make XCode fit you (Single window interface, XCode themes, application templates)

XCode plugins (Code Pilot
`http://mac.brothersoft.com/code-pilot-for-xcode.html`,
Accessorizer
`http://www.kevincallahan.org/software/accessorizer.html`)