**HooApps Lesson 1-iOS**

**Learning Objectives:**

→ Xcode Project Setup

→ Model-View-Controller Design Pattern

→ Using Interface Builder and Auto-Layout

→ Using the Simulator

→ Hooking up IBOutlet Target-Action Pairs

→ Learning the basics of how to write code

**1. Create a new Xcode Project**

→ On the left side of the pop-up, choose Application under iOS and select “Empty Application”

→ For Product Name, type “UVA Fun Facts”

→ For organization name, you may type anything. I'll use “hooapps”

→ For company identifier, you may type any string. It is important that this be unique for App Store purposes. I'd recommend com.hooapps.<yourname> or something even more specific.

→ For Class Prefix, choose UFF (for UVA Fun Facts).

(\***See** **Pro-Tip 1 for more on class prefixes\*)**

**2. Create a UFFHomeViewController**

→ Go to File → New File → Objective-C Class

→ For class, write “UFFHomeViewController”

→ Make it a subclass of UIViewController

→ Check the box that says “also create XIB file”

(\***See** **Pro-Tip 2 for more on XIB files\*)**

**(\*See Pro-Tip 3 for more on the Model-View-Controller Design Pattern\*)**

**3. Set the Application's Root View Controller**

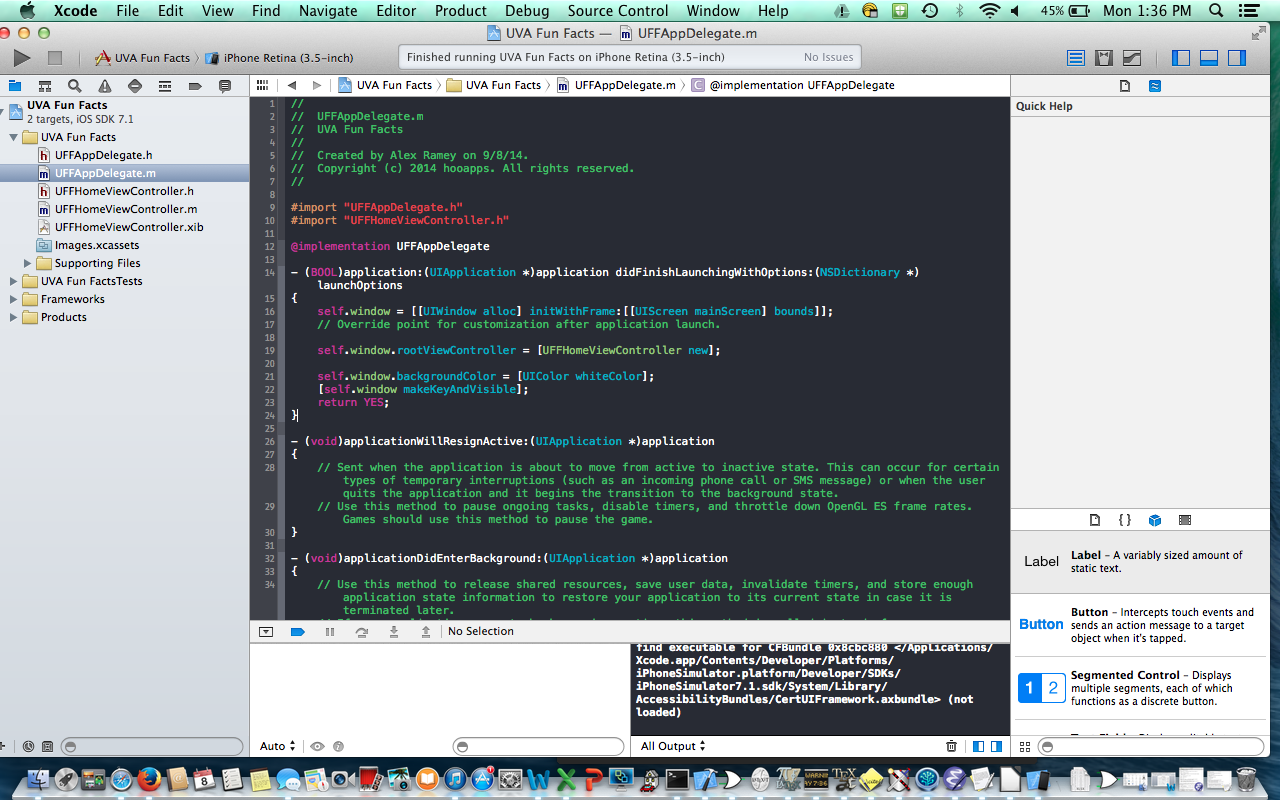
→ Click on the UFFAppDelegate.m file in the project navigator.

→ Add #import “UFFHomeViewController.h” to the top

→ Add

self.window.rootViewController = [UFFHomeViewController new];

to the application: didFinishLaunchWithOptions: method, just after the override point for customization after application launch. See below:



**4. Create the User Interface for UVA Fun Facts**

→ Select UFFHomeViewController.xib in the Project Navigator

→ If necessary, expand the Utilities drawer on the right side of Interface Builder by selecting the button in the top right corner.

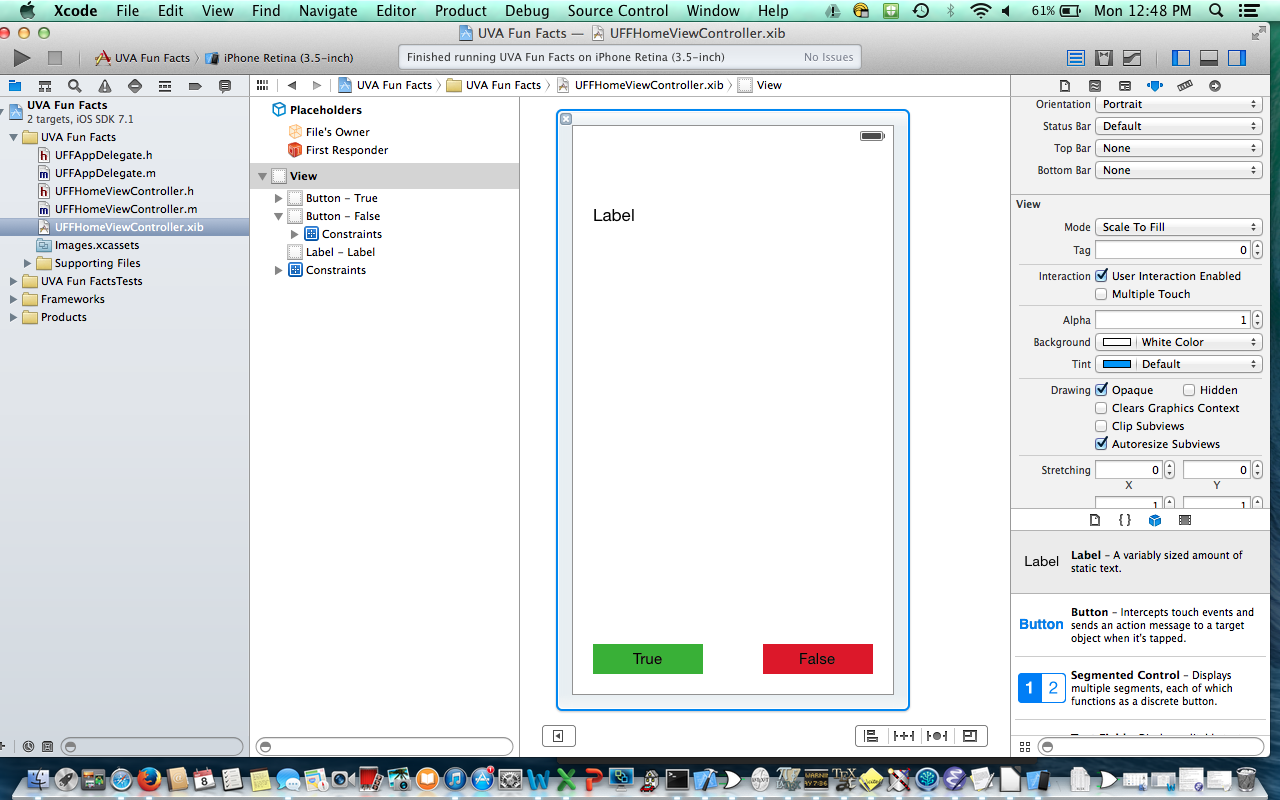
→ Find “Button” in the Object Library Tab of the window at the bottom of the Utilities drawer

→ Drag two buttons onto your “View” (the iPhone-sized, white rectangle in Interface Builder)

→ Drag a Label from the Object Library onto your “View”

→ Click on the first Button, and select the Attributes Inspector Tab of the window at the top of the Utilities drawer. Change the first button's text to “True” and scroll down to “Background” and set that to Green (RGB: 70, 177, 5). For the second Button, make its text “False” and its background color Red (RGB: 215, 30, 36). (RGB values can be specified on the second tab of the “Color” pop-up that presents itself when you click on Background Color.)

→ Move the views to roughly match the layout shown below:



**4. Add Constraints**

→ First, make sure your button heights are 30.0 and widths are 110.0. If you click on a button, you can change its shape by grabbing its corner and stretching/compressing it. You can also enter these dimension values in the Size Inspector, the tab to the right of the Attributes Inspector tab in the top of the utility window.

→ When moved toward the edges of the screen, margin guidelines should appear to help you create consistent margins. Make sure these buttons are equidistant from their respective sides of the screen, and align them so that their distance from the screen's bottom are the same.

→ Now, hold down **ctrl** and click in the middle of the True button and drag towards the left side of the screen. You should see a blue line appear. Release the click in the whitespace margin area between the screen's left edge and the True button. When the black box pops up, click “Leading Space to Container”.

→ Now, do the same thing for the False button, this time dragging toward the right side of the screen and selecting “Trailing Space to Container”.

→ Now, hold down **ctrl** again, and this time drag from the green button to the red button. When the black box appears, select Equal Widths. Do this again, and select Horizontal Spacing.

→ Now, for each button, hold down **ctrl** and drag to the whitespace between the bottom of the button and the bottom of the screen. Select “Vertical Spacing”.

→ Finally, for each button, hold down **ctrl** and drag up a little bit, forming a small vertical line that starts and ends within the button. Select “Height”.

→ For now, just leave the label near the top of the screen. Stretch it to take up the width of the screen (with a little bit of margin on either side) and set the following constraints between the label and its container view:

1) Leading Space to Container

2) Trailing Space to Container

3) Top Space to Container

→ Then select the label and, in the attributes inspector, change the value of the “Lines” property to 0. This means that this label may have an unlimited number of lines. Had we left it at 1 and set the label's text to be larger than the label, the text would get cut off. By setting this property to 0, we allow the label to vertically stretch to encompass all of the lines.

**(\*See Pro-Tip 4 for more on Auto-Layout\*)**

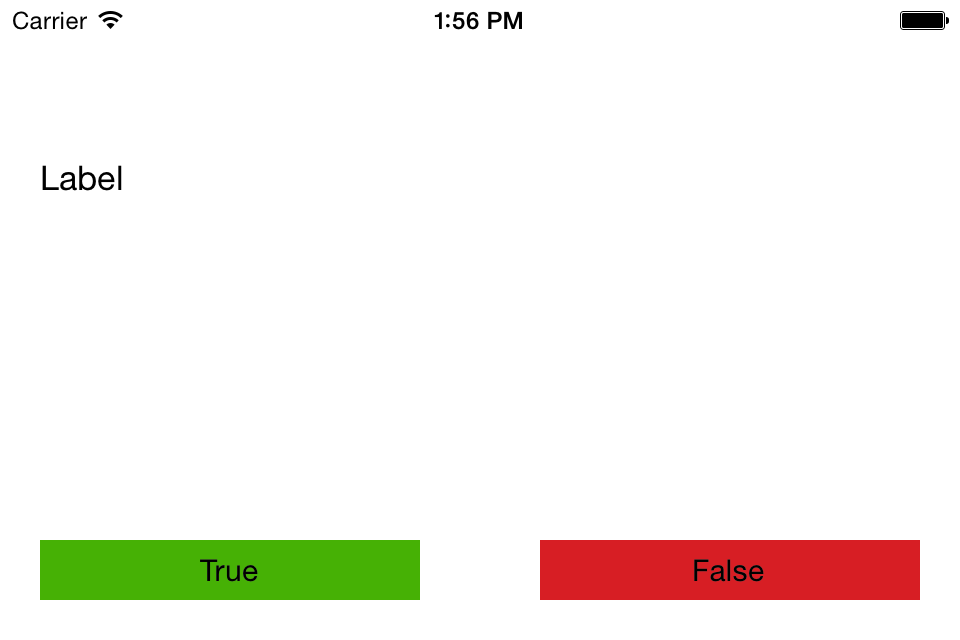
**5. Run Your Code on the Simulator**

→ At the top left of the screen, next to the current build target (which in this case should be UVA Fun Facts), make sure one of the simulators is selected instead of iOS Device. For the fastest load times, I recommend using iPhone Retina (3.5 inch), which simulates the screen size of the iPhone 4 and 4s.

→ Click run. When the app loads, you can rotate the simulator with the keys **cmd →** and **cmd ←.**

The true/false buttons should resize themselves as shown when the simulator rotates. This is why we use Auto-Layout.

**(\*See Pro-Tip 5 for more on supporting different screen orientations\*)**

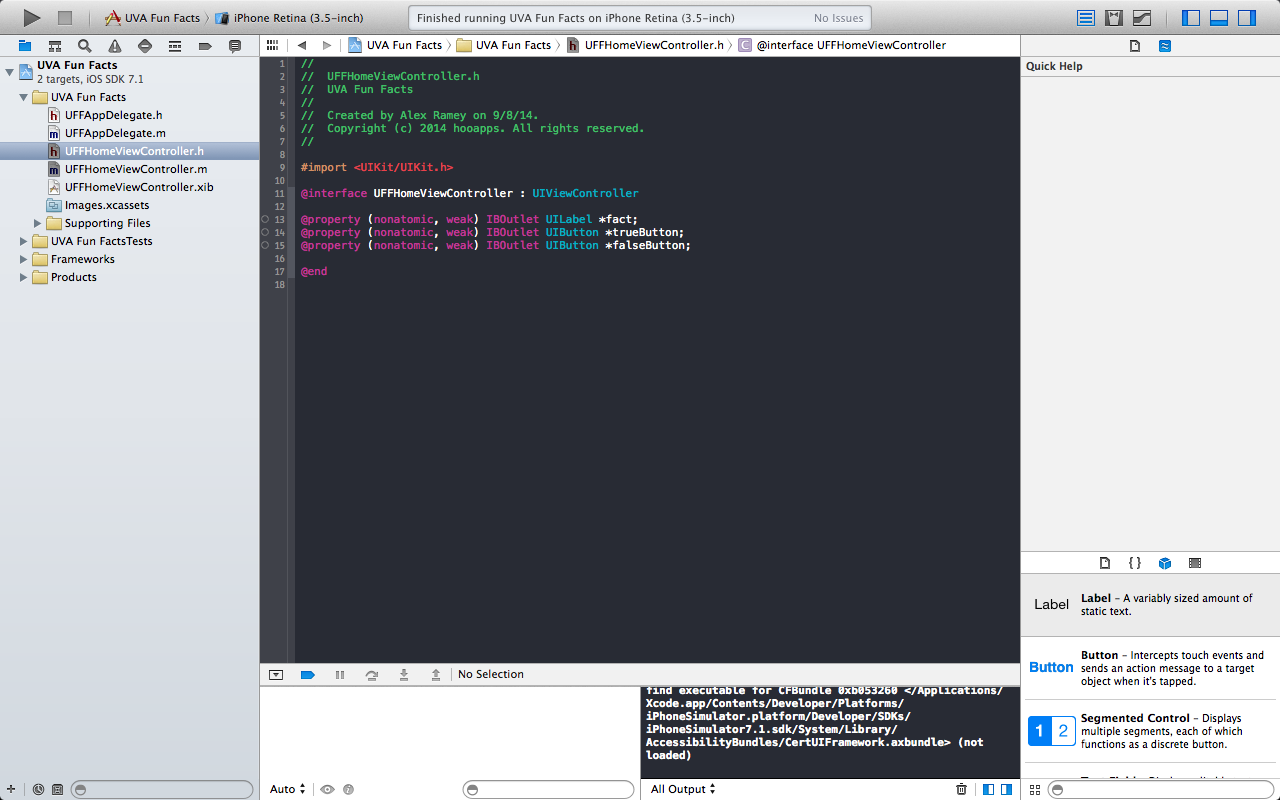
**6. Hook Up the Buttons/Labels to the UFFHomeViewController**

→ Add the following lines to UFFHomeViewController.h as shown:

@property (nonatomic, weak) IBOutlet UILabel \*fact;

@property (nonatomic, weak) IBOutlet UIButton \*trueButton;

@property (nonatomic, weak) IBOutlet UIButton \*falseButton;



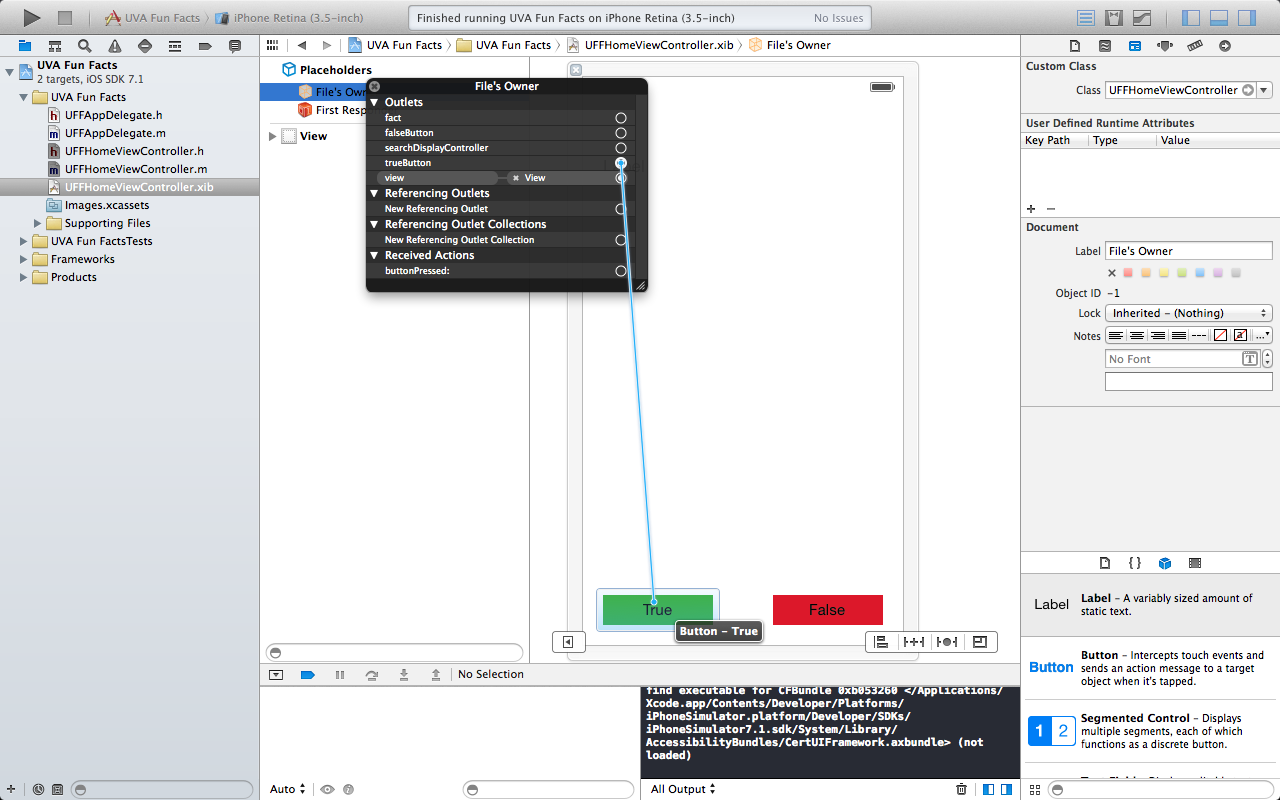
→ Notice the little open circles to the left of the @property designations.

→ Notice the fact that we used the IBOutlet keyword before declaring our labels and buttons. We did this because we are declaring an “Interface Builder Outlet”. Time to connect the code to the UI.

→ Open UFHomeViewController.xib

→ Click on File's Owner and verify that its class is UFFHomeViewController in the Identity Inspector tab at the top of the Utilities window.

→ Right click on File's Owner and hook up your label and two buttons as shown:



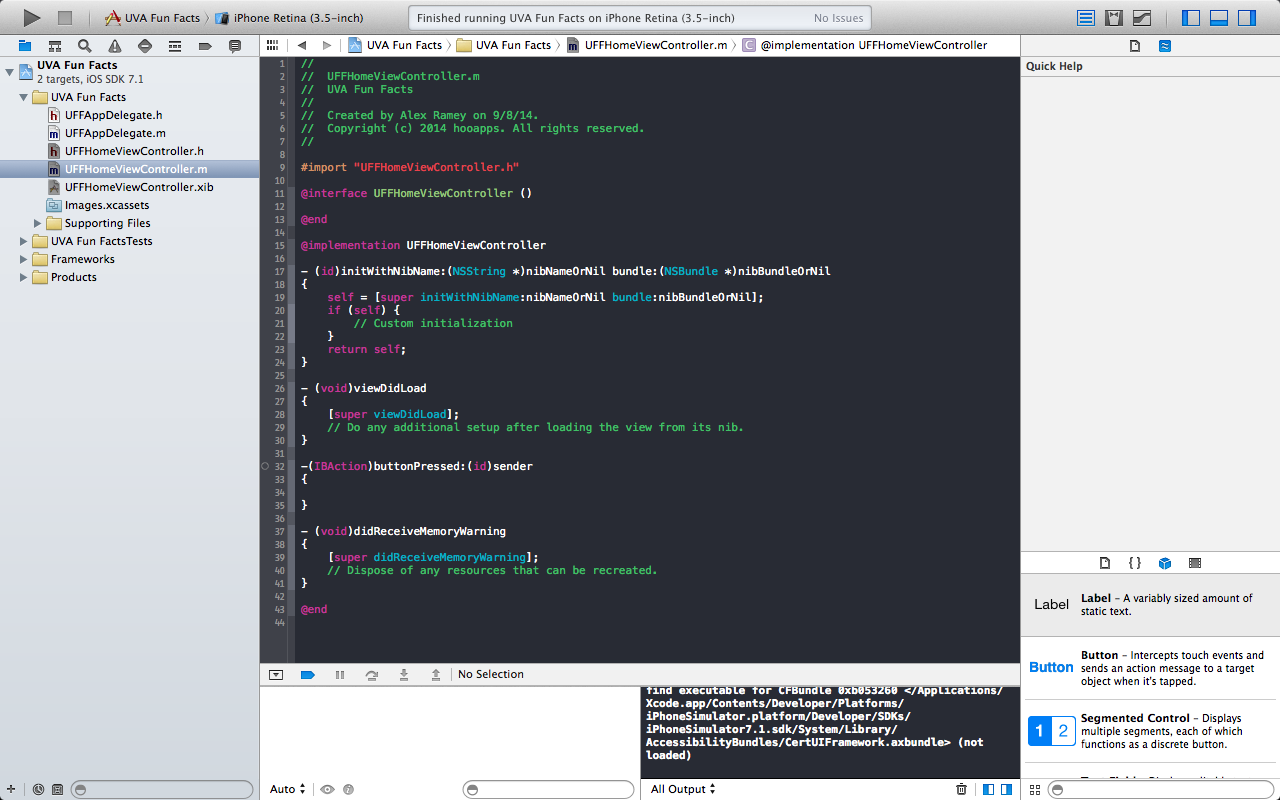
→ Now add the following method to UFFHomeViewController.m

-(IBAction)buttonPressed:(id)sender

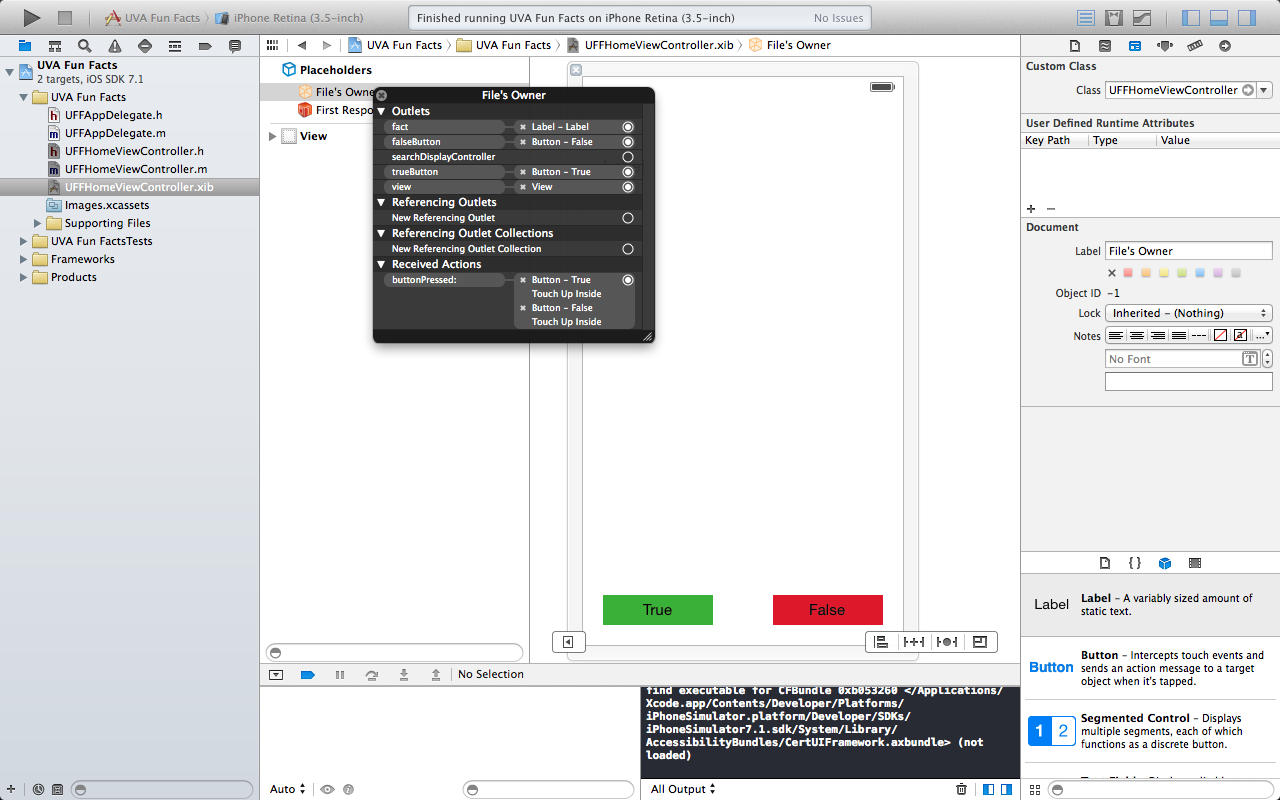
{

}

as shown:



→ Then return to Interface Builder and hook up both of the buttons to this method by right-clicking on File's Owner again and dragging from the “buttonPressed” method to the buttons. When the black box appears over each button, choose “Touch Up Inside”. This means the button will trigger that action when a user moves their finger off the screen from inside the button. When you're done, File's Owner should be hooked up as shown:

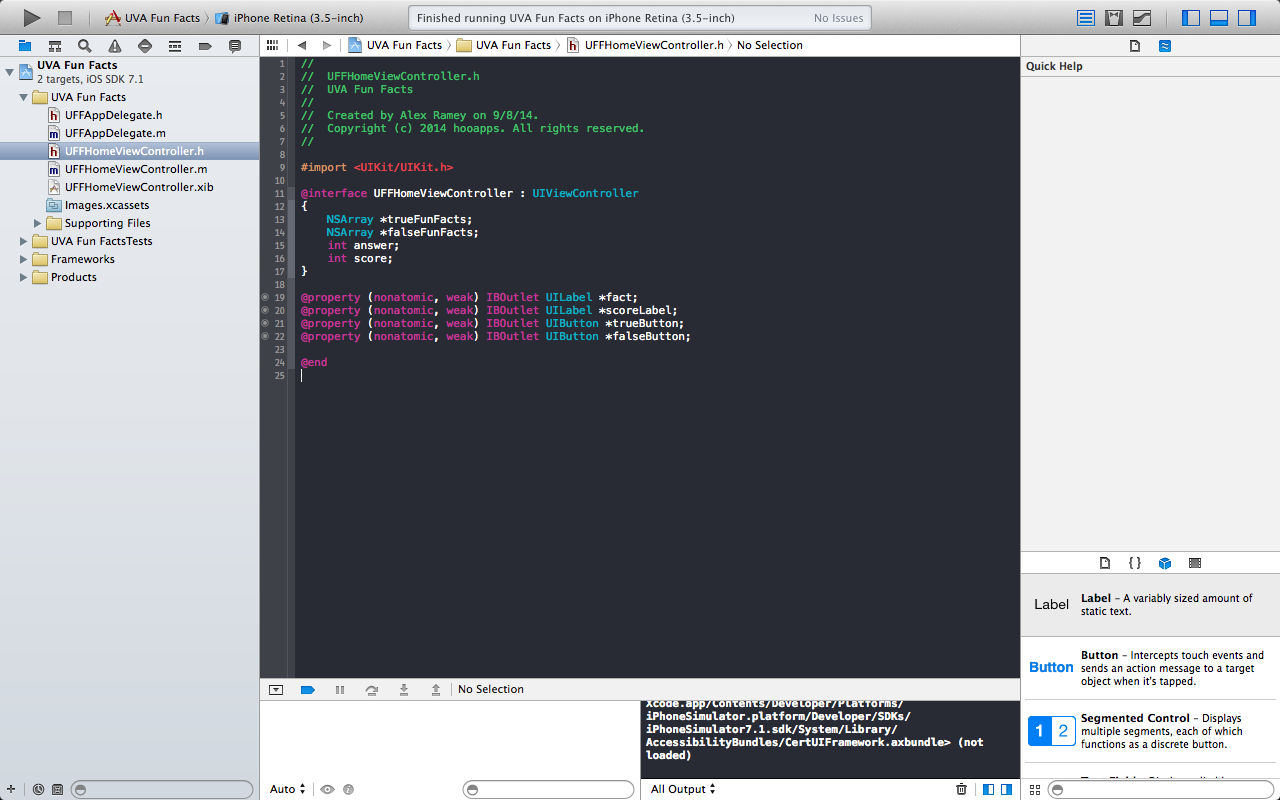


This is an illustration of a design pattern known as “Target-Action”. Basically, the Button now has a target (the UFFHomeViewController) on which to call the action buttonPressed whenever it gets tapped.

**7. Add some Meat and Make it Work**

**→** Apply what you've learned so far! Add 4 private fields and one more IBOutlet to your UFFHomeViewController.h file as shown below. Then add a score label to the top center of your XIB file in Interface Builder, and hook it up to the new scoreLabel IBOutlet you just created. Finally, apply AutoLayout to constrain the label.

Note: Fields held within the { } in the header file are private, meaning that other classes don't have visibility on them.



→ Add the following 3 lines to your UFFHomeViewController's -(id)initWithNibName: bundle: method inside the body of the if statement like so

: if (self) {

trueFunFacts = @[@"Dr. Seuss was denied admittance to UVA. Rumor has it the name for the fictional town of \"Whoville\" was a pun off the University's nickname, \"Hoos\".", @"The official mascot for UVA is the cavaliers, but they unofficially adopted the Wahoos, a fish that can drink its weight in water, as their second mascot. While it was originally shouted as an insult by an opposing school at a football game, the students at UVA instead adopted it as a name.", @"UVA was founded in 1819 by Thomas Jefferson, who was this nation's first secretary of state, second vice president, and third president.", @"UVA's undergraduate student body is 65% out of state", @"The founder of MIT was an engineering professor who came from UVA"];

falseFunFacts = @[@"In May 2005, there was a water slide that went down the middle of the UVA Lawn", @"The Rotunda has been under construction for 99.1 % of its life.", @"UVA was founded in 1831 by Christopher Columbus", @"UVA has put 15 monkeys on the moon.", @"UVA's current President is Chuck Norris, and he lives in the Steam Tunnels with the Ninja Turtles."];

score = 0;

}

Basically, you just initialized three of your private variables. This method is called once when the view controller is first created. In the AppDelegate, where we say [UFFHomeViewController new], that is actually shorthand for [[UFFHomeViewController alloc] init]. Sending the UFFHomeViewController class the message “alloc” returns a memory address with enough space to hold an instance of a UFFHomeViewController. Then sending that memory address the message “init” puts an object there, setting its fields to default values, and returns its address. For view controllers, init internally calls the initWithNibName: nibBundle: method, the one we just implemented above, which is called the “designated initializer” for UIViewControllers because it calls [super initWithNibName: nibBundle:] at the beginning. Calling the designated initializer of your superclass is always the first step when writing designated initializer methods in Objective-C land. It is absolutely required that all objects call some form of init on their superclass. This creates a chain known as the “initialization chain”. Note that in Swift, the call to super doesn't come first.

We also just declared trueFunFacts and falseFunFacts as “array literals”. Meaning we straight-up hard-coded them. An alternative would've been to create an NSMutableArray and then to add the strings one by one to it. NSArrays, like the ones we created, are immutable, meaning you can't add/remove objects once they've been assigned a value.

Notice inside the array literal the @ the comes before the “<string text>”. This is shorthand for creating an NSString object, assigning it that <string text> as its value, and then returning the memory address at which that NSString object resides. Arrays are basically lists of pointers in Objective-C. The value of a pointer is a memory address, where the object that it “points to” resides.

**(\*See ProTip #7 for a word about Apple Docs\*)**

→ Make sure your viewDidLoad method looks like this:

- (void)viewDidLoad

{

[super viewDidLoad];

// Do any additional setup after loading the view from its nib.

[self updateFactAndScore];

}

→ Add the following method:

-(void)updateFactAndScore

{

if (score == 10)

{

UIAlertView \*a = [[UIAlertView alloc] initWithTitle:@"You Win" message:@"Nice job!" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];

[a show];

score = 0;

}

int trueFact = arc4random\_uniform(2); //random selection of either 1 or 2

int random = arc4random\_uniform(5); //random selection of int that is in range 0 – 4 inclusive

if (trueFact == 0) //CS people tend to think of 0 as false and 1 as True

{

[\_fact setText:falseFunFacts[random]];

}

else

{

[\_fact setText:trueFunFacts[random]];

}

answer = trueFact;

[\_scoreLabel setText:[NSString stringWithFormat:@"Score: %d", score]];

//%d is a token that takes the place of an int in formatted strings

}

→ And implement the IBAction method as shown:

-(IBAction)buttonPressed:(id)sender

{

UIButton \*tappedButton = (UIButton \*)sender;

if ([tappedButton.titleLabel.text caseInsensitiveCompare:@"True"] == NSOrderedSame)

{

if (answer == 1)

{

score++;

}

else

{

score--;

}

}

else //False was tapped

{

(answer == 0) ? score++ : score—;

/\*PRO-TIP

This is the same as

if (answer == 0)

{

score ++;

}

else

{

score--;

}

\*/

}

[self updateFactAndScore];

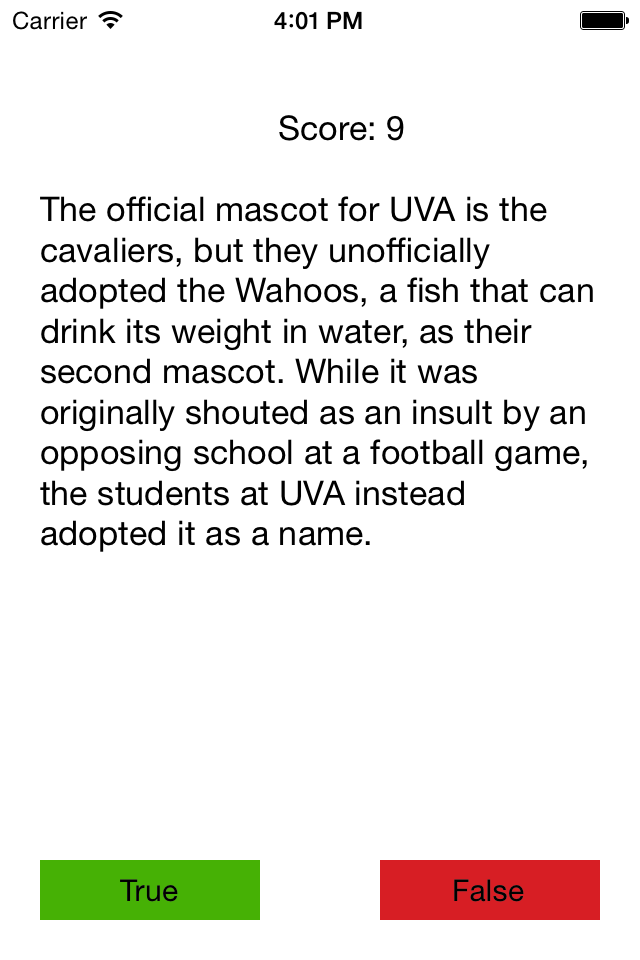
}

**8. A Word About What Just Happened**

Alright! Hit Run and see what happens. This app should be working. If not, come ask me for help.

When you're done, you should review the encouraged reading and give the challenges a shot.

Congratulations on taking your first step into iOS Programming.



**Pro-Tips**

**#1:** **Class Prefixes**

It is important to choose a class prefix that is either two or three capital letters and to stick to it for all of you classes throughout the project. This prevents namespace collisions with Apple's pre-defined classes. For instance, Apple has a class called NSArray that we'll commonly use. If we were to make an array class and just call it “Array”, and Apple had just called their array class “Array”, it would be unclear to the compiler (the thing that converts your objective-c code into the machine code that is run on the iPhone's processor) which class we were talking about when we used “Array” in our code. Apple, who defines the “best practices of iOS development”, uses prefixes to avoid these collisions, and we will too. By the way, NS is one of the common prefixes you'll see Apple's classes use. It stands for NeXTStep, a company at which Steve Jobs worked during his time away from Apple and which Apple later purchased. So, whenever you see the “NS” Prefix, you're using a class that likely has its roots in NeXTStep.

**#2:** **XIB Files**

XIB (pronounced “zib”) stands for “XML Interface Builder”. Interface Builder is the environment that Apple created in Xcode to enable easy editing of XIB files. If you click on a XIB file in the project navigator, you will see Interface Builder pop up. Under the hood, Interface Builder is editing the underlying XML code that represents that view object as you visually edit the view object in the interface builder. For our purposes, we will never need to dive into this underlying XML code. However, if you're curious and want to see what the underlying XML looks like, right click on a xib file in the project navigator and select “Open As” → “Source Code”. XML stands for “Extensible Markup Language” and is basically a language that allows you to express information in a nested node/tree format. XML is fairly simple to use, but learning it is beyond the scope of this lesson. These “XIB” files represent view objects, or, put simply, the screens that users will see when they use your app. When your app is compiled, these XIB files are converted to more compact, un-editable NIB files, and consequently the terms XIB and Nib are often used interchangeably.

**#3: A Word about Model-View-Controller (MVC)**

In iOS development, adopting the “Model-View-Controller” design pattern is considered good practice and recommended to maximize your code reusability and readability. For this project, we will be following this pattern closely so that you will begin to get a sense for it. In practical development, you will more frequently encounter the Model-View-Controller-Store (MVCS) design pattern, where “Store” objects manage, fetch, and updat data that either resides in the phone's memory or on a web service. “Store” objects will usually have methods that return a set of model objects that are built from this data. For more information about MVC, see section 2 of the encouraged reading chapter below.

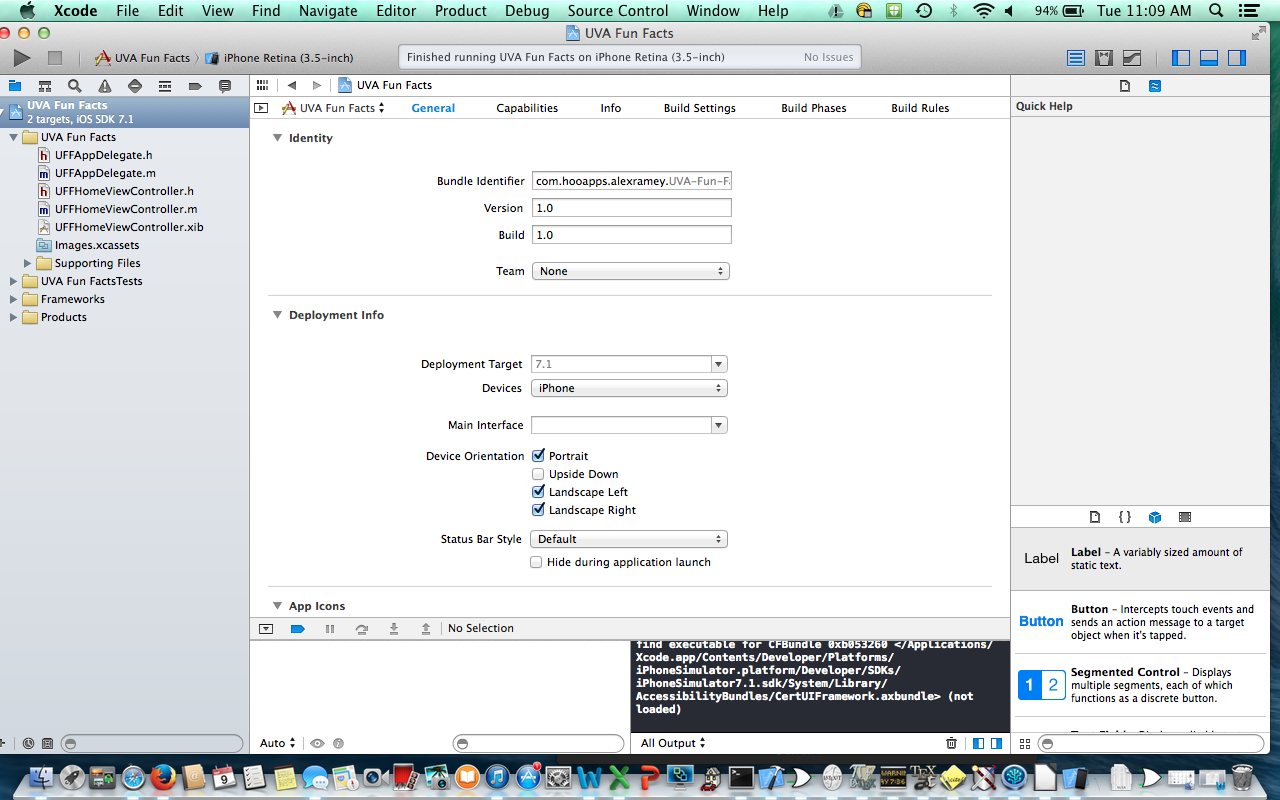
**#4: Auto-Layout**

Auto-Layout is essentially a mechanism by which you can specify certain constraints for each of your views. You are required to provide enough constraints such that Xcode can “solve” your view hierarchy and determine the placement and size of each of your views. The motivation for using Auto-Layout is creating views for multiple screen sizes/orientations. Ideally, with Auto-Layout, you can create the view once and it will be displayed appropriately on all screen sizes/orientations. By the way, Apple just announced iPhone 6 and iPhone 6 plus, which add two new screen sizes to the mix that you need to support. Auto-Layout is becoming more and more important.

For more information about Auto-Layout, I recommend this tutorial: <http://www.raywenderlich.com/50317/beginning-auto-layout-tutorial-in-ios-7-part-1>.

**#5: Supporting Different Screen Orientations**

In the real world, many apps don't bother to support two screen orientations. A lot of apps are either only landscape or only portrait. This is because not all apps have a user interface that is easily transferable to different screen orientations. Additionally, supporting multiple screen orientations increases development time and the cost of the project. If you wish to disable landscape mode, click on the UVA Fun Facts Target at the top of the Project Navigator, and, under the “General” Tab, uncheck the boxes next to “Landscape Left” and “Landscape Right”.

****

**#6: Property Specifiers and an Introduction to Memory Management**

@property (nonatomic, weak) → This means we are declaring a property. The “property specifiers” in the parentheses are characteristics of the property. When declaring a property, we always will use nonatomic. Don't worry about why for now. “Weak” means this property holds a weak reference to the object (view in this case) to which it points. The view will stay alive, or stay allocated, as long as there exists at least one thing that holds a strong reference to it. In this case, the view's superview will always hold this strong reference, and this view controller will hold a strong reference to this superview, so there is no need to declare this property strong. If the view were to get destroyed, this property would be set to point to “nil” (or memory address 0x00000000), which is a nice, “smart pointer” feature that prevents the “dangling pointer” scenario in which you dereference a pointer that points to a spot in memory where an object was deallocated and your program crashes because it tries to do something based on the assumption that an object resides where the pointer is pointing. Thanks to ARC (automatic reference counting), which Apple added in an earlier version of Xcode, you don't have to worry too much about memory management. In this case, nothing would break had you declared the property (nonatomic, strong), but declaring it weak is best practice because the strong is unnecessary.

Properties are readwrite by default, meaning external classes can “write” to them or “set” their values in addition to being able to read or “get” them. You may instead declare properties as readonly, meaning that external classes can “get” them but not “set” them. These other property specifiers are used frequently as well and would be worth doing a quick Google search on: copy, assign.

**#7: Apple Docs**

In UFFHomeViewController.m, hold down the option key and click on the initWithNibName: bundle: method. You should get a nice pop-up from the Apple Docs telling you all about the method. Then scroll down in the pop-up and click on UIViewController, next to Reference. This will bring up the Apple Docs for UIViewController, enabling you to see all of the methods and fields of UIViewController that are exposed to you.

I usually open the docs in Safari by googling iOS UIViewController or iOS UIScrollView or iOS whatever and clicking on the Apple Docs hit. Navigating these docs is the bread and butter of a developer.

**Challenges- Give these a shot if you have what it takes.**

Bronze: Make it so that facts are never repeated until all of them have been seen once.

Silver: Make it so that swiping left on the screen answers “true” and swiping right on the screen answers “false”

Gold: When the app loads, pull the facts down from a web-service. This is for experienced programmers only. Topics you'll encounter: NSURLSession, JSON, NSJSONSerialization.

URL: https://raw.githubusercontent.com/hoo-apps/iOS-Lessons/master/Lesson%201/HALesson1Gold.json

**Encouraged Reading:** Chapter 1 of the Big Nerd Ranch Guide to iOS Progamming 4th ed.

→Using Safari, go to UVA Library Virgo homepage, and search for Big Nerd Ranch: Guide to iOS Programming. You should be able to open the 4th edition in Safari Books Online.