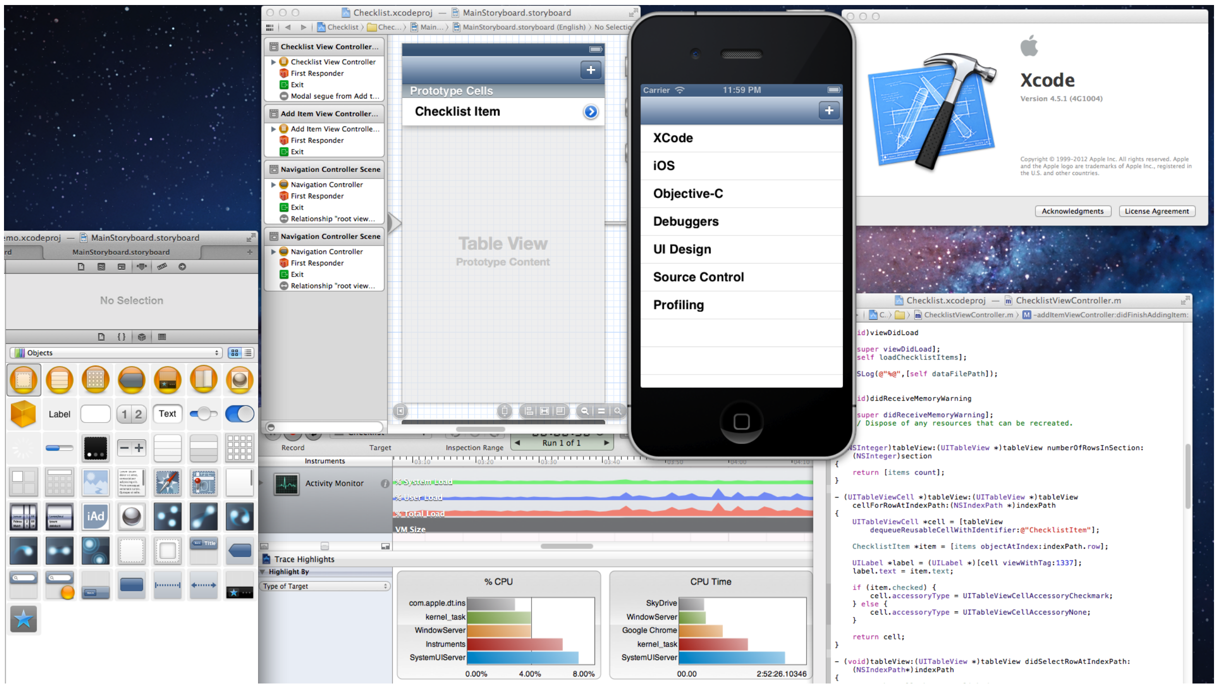
school of computing, informatics, & decision systems engineering

iOS Programming using Swift
Lab Manual

Christophe Faucon, Ryan Dougherty, Janaka Balasooriya



Contents

Chapter 1

Acknowledgments

The authors would like to thank Ali Rafiq, Xiaoxiao Wang, and Alexander Nou for their contributions to the labs.

Chapter 2

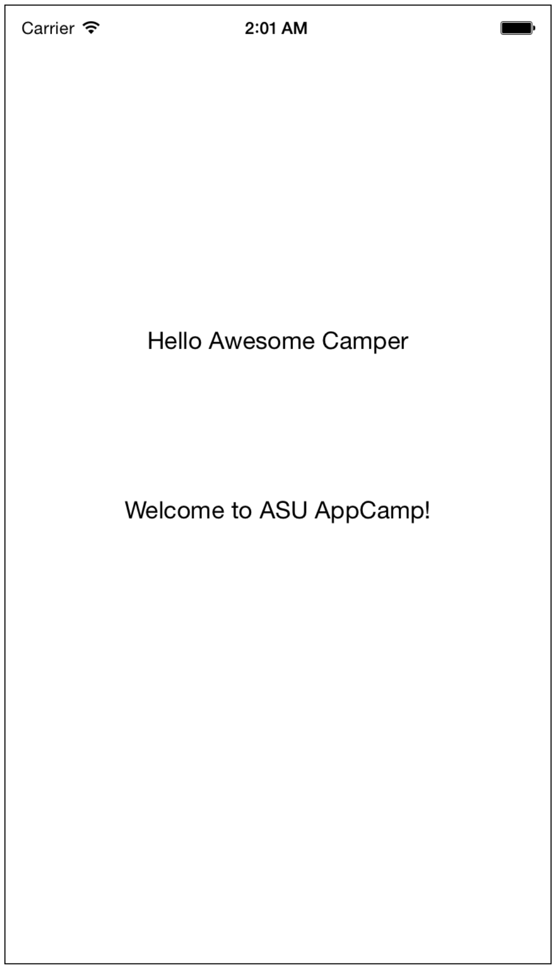
Introduction to Swift and Xcode

2.1 Our First Xcode Adventure!

2.1.1 Goals of this Lab!

- Get familiar with Xcode
- Learn to create a new project
- Learn how to use label views to display information

At the end of the application it should look like the image below:



Hello Awesome Camper

Welcome to ASU AppCamp!

2.1.2 Starting Our First Project

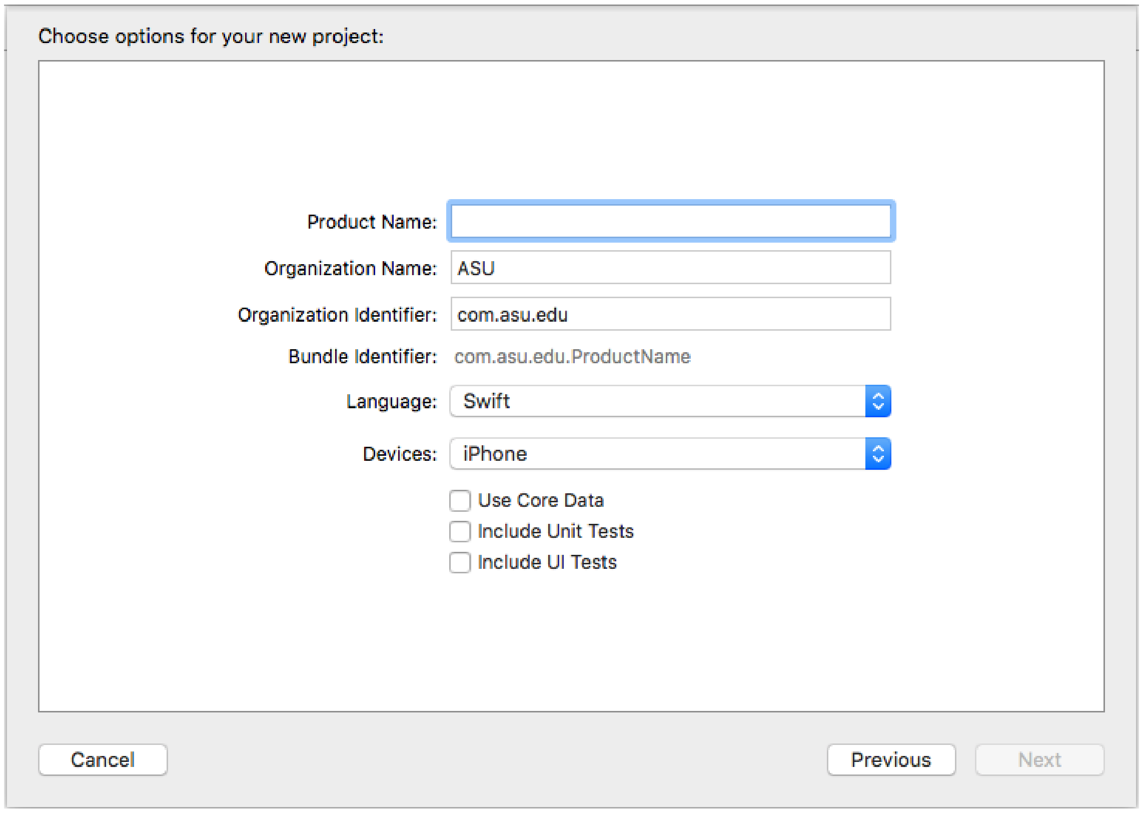
To get started, we'll first need to open Xcode and start a project. You can find Xcode by pressing the search (magnifying glass) icon in the upper-right corner of the screen and searching in spotlight, or you can navigate to it in Finder (Applications → Xcode).

- In the "Welcome to Xcode" screen, select "Create a New Xcode Project".
- Since our App will have one screen we want to create an:
- "iOS" → "Application" → "Single-View Application" → "Next"
- We need to give the App a name, like "Lab1", as well as provide a company identifier. iPhone Apps use the "reverse domain name" style for identifying authors; so our company identifier is `edu.asu`.

- For the "Device Family" select "iPhone", so that the App is iPhone only (not also iPad).

- You can also set a "Organization Identifier" for your project. Set the Organization Identifier to "HelloWorld".

- You will want to select "Swift" as the Language → "Next".


Choose options for your new project:

Product Name:
Organization Name: ASU
Organization Identifier: com.asu.edu
Bundle Identifier: com.asu.edu.ProductName
Language: Swift
Devices: iPhone
Use Core Data
Include Unit Tests
Include UI Tests

- You will be prompted to select a location to save your project folder; a location on the Desktop will be fine, or you're welcome to use the Documents directory.

- Later we will use the git repository; for this project we won't use it though.

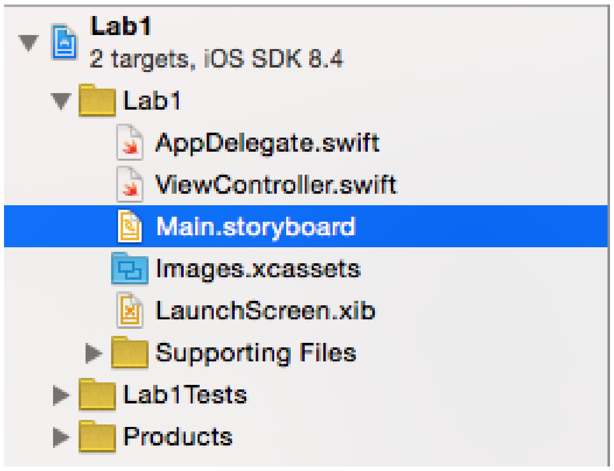
From here, you can press the "Run" button in the upper-left corner to launch the App in the iPhone Simulator. Huzzah! We have an application that runs. Now as if that wasn't enough excitement for one day, let's get started with the app!

2.1.3 Completing The App

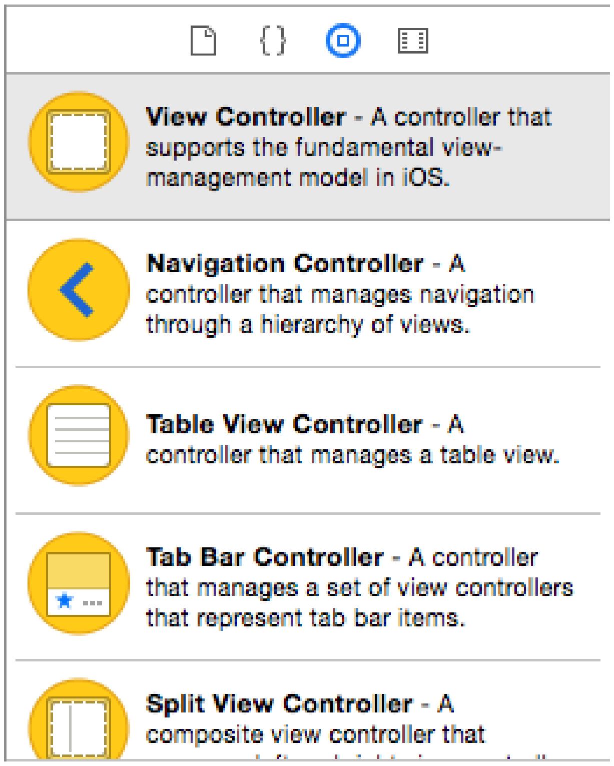
Working With The Storyboard

The next step is completing the storyboard. The storyboard is a visual representation of the program that you are making; it is a way to see what each of the screens will look like without running the program. What we are going to do here is add two labels, which are going to display a name and a greeting.

- From the Project Navigator click on the `Main.storyboard` file.



- In the bottom-right of the window you will see a cube icon; click on it and a list of user interface objects will appear:



View Controller - A controller that supports the fundamental view-management model in iOS.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Table View Controller - A controller that manages a table view.

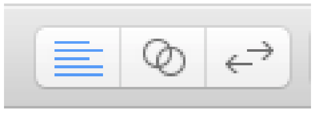
Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar items.

Split View Controller - A composite view controller that

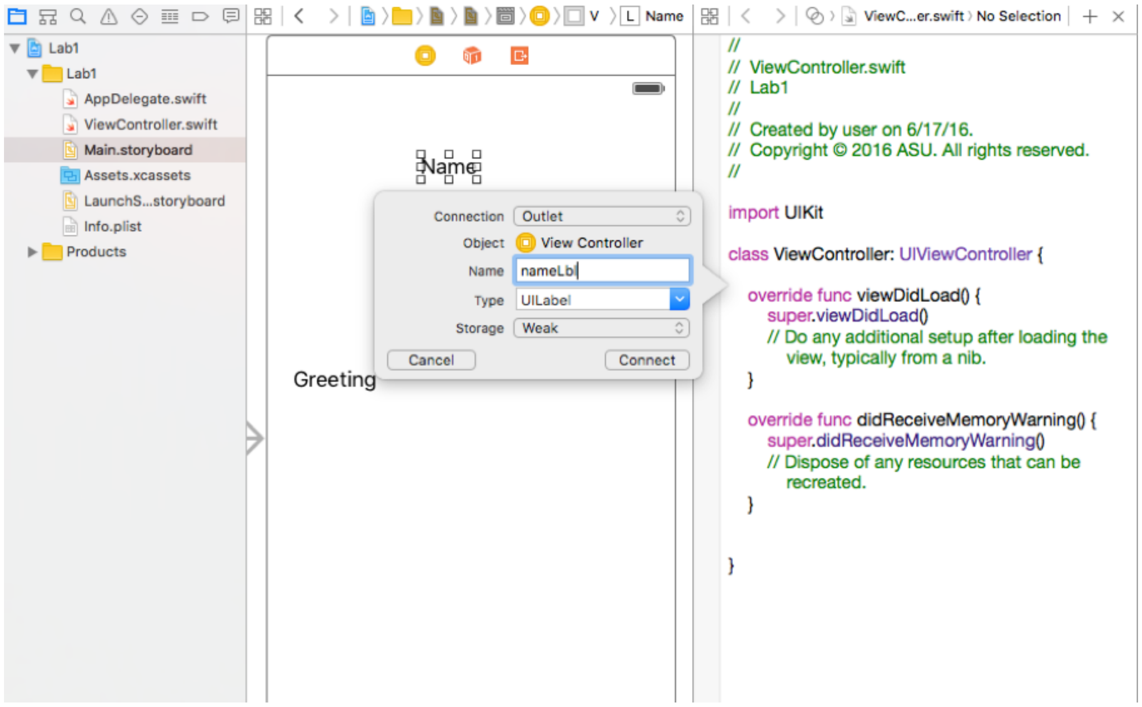
- From here drag out two `Labels`; to aid in your searching, there is a search bar at the very bottom. If you type into that search bar then the results will be dynamically filtered for you! After you drag them out, resize them (drag the left and right edges) so that they take almost the full width of the screen (you can use the blue snap-guides to help you).
- Next we are going to set the `Text` property of each label; the easy way to do this is to double-click on the label that you placed in the storyboard and replace the previous text. Set the text to `Name` in one of the labels, and `Greeting` in the other. If you run the app now you'll have some labels! We're almost there but we would like to be able to set the labels from the code.

To access the labels from the code we have to bind them to `IBOutlet` properties; you can think of `IBOutlet` properties as a phone number for accessing the label so that we can talk to it. Don't worry about the scary terminology though, setting properties up is a breeze!

First we will open the assistant editor, it's the double circle icon on the top right of the screen; make sure you're viewing the storyboard beforehand!



The assistant editor will allow us to have two files open at the same time; mind blowing, right? From here you will need to ctrl+click and drag from the label that says `Name` onto the source-code to the right between the `class` line and the `override func viewDidLoad()` line like in the image below. When you let go of the mouse a popover will appear; in the popover, set `Name` field to `nameLabel`. Do the same for the greeting label and name it `greetingLabel`. Once those are finished, switch back to the `Standard editor` (the button just left of the assistant editor).



Connection: Outlet
Object: View Controller
Name: nameLbl
Type: UILabel
Storage: Weak
Cancel
Connect

```
//
//  ViewController.swift
//  Lab1
//
//  Created by user on 6/17/16.
//  Copyright © 2016 ASU. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the
            view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be
            recreated.
    }


}
```

Completing the Code

Open your `HellowWorldViewController.swift` file. It should look familiar as we were just editing it with the assistant editor! Just for verification make sure that it looks something like below:

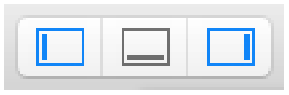
```
import UIKit

class ViewController: UIViewController {
        @IBOutlet weak var nameField: UILabel!
        @IBOutlet weak var greetingField: UILabel!

        override func viewDidLoad() {
                super.viewDidLoad()
        }
}
```

On the left of the properties there should be an open grey circle with a solid grey circle inside; this means that the `IBOutlet` property is connected to the storyboard properly (if you don't see this then flag one of us down and we can figure it out!). This file indicates how the code should act, and what it should do (in this case it will set the `Text` property of our labels! The code inside `viewDidLoad` won't exist when you first open the file, so welcome to your first experience writing code! Write the text from below so that everything matches; you can skip lines that start with `//`, which is a special symbol to tell the computer that a line has no commands for it.

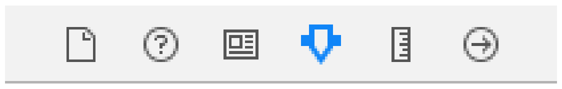
```
// this is where the magic happens! All active code appears here
override func viewDidLoad() {
        super.viewDidLoad()

        // replace "Awesome Camper" with your own name
        self.nameField.text = "Hello Awesome Camper"
        self.greetingField.text = "Welcome to ASU AppCamp!"
}
```

Polish and Swagger

The last step is going to be to personalize this application and make it your own (in addition to putting your name on it). To do this let's return to the storyboard. Ensure that the right-hand side view is open (in the top-right of Xcode make sure the view selector looks like the image below:



Now click on one of the labels and on the right-hand side view select the attributes inspector (it is the icon just below view),



You can use this to set things like text formatting. Let's change the alignment to center, and then change the font color to something of your choosing.

2.1.4 Congrats!

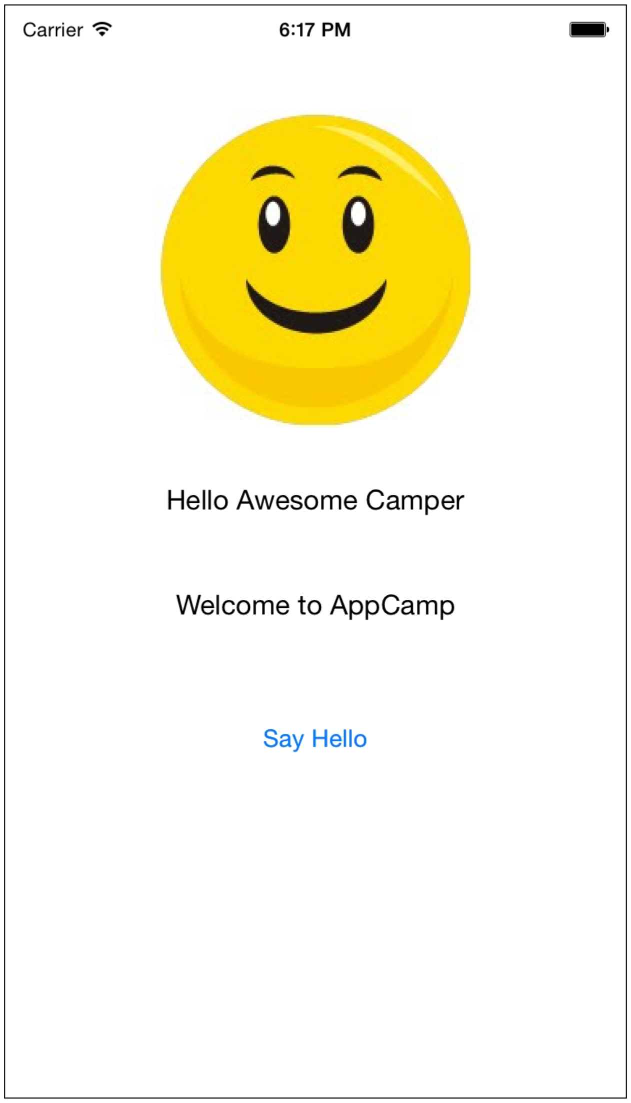
Congratulations, you finished your first iPhone app!

2.2 Actions and Smilies

2.2.1 Goals of this Lab!

- Understand how to detect and respond to user actions
- Utilize `Buttons` and `ImageView` objects
- Understand `IBActions` and their relations to the above

The final project should look like the following:

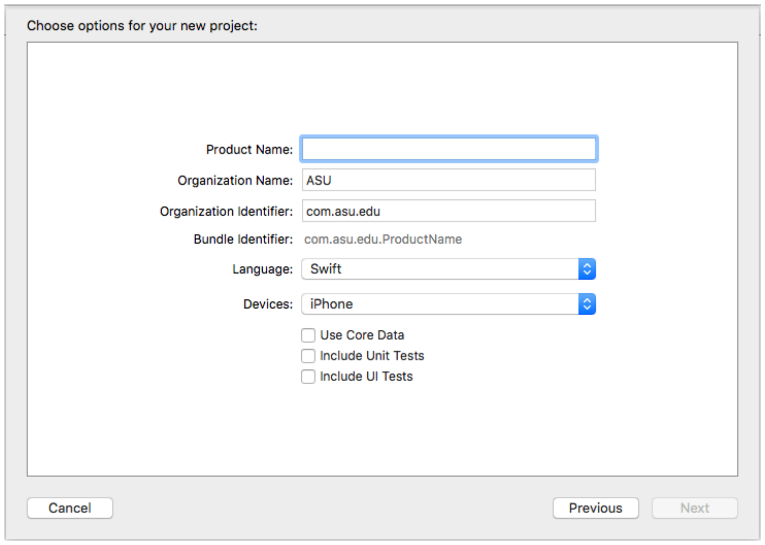


2.2.2 Starting Our Second Project

To get started, we'll first need to open Xcode and start a project. You can find Xcode by pressing the search (magnifying glass) icon in the upper-right corner of the screen and searching in spotlight, or you can navigate to it in Finder (Applications → Xcode).

- In the "Welcome to Xcode" screen, select "Create a New Xcode Project".
- Since our App will have one screen we want to create an:

- "iOS" → "Application" → "Single-View Application" → "Next"
- We need to give the App a name, like "Lab2", as well as provide a company identifier. iPhone Apps use the "reverse domain name" style for identifying authors; so our company identifier is `asu.edu`.
- For the "Device Family" select "iPhone", so that the App is iPhone only (not also iPad).
- You can also set a "Organization Identifier" for your project. Set the Organization Identifier to "CSE".
- You will want to select "Swift" → "Next"


Choose options for your new project:

Product Name:
Organization Name: ASU
Organization Identifier: com.asu.edu
Bundle Identifier: com.asu.edu.ProductName
Language: Swift
Devices: iPhone
Use Core Data
Include Unit Tests
Include UI Tests

- You will be prompted to select a location to save your project folder, a location on the desktop will be fine, or you're welcome to use the Documents directory.
- Later we will use the git repository, for this project we won't use it though.

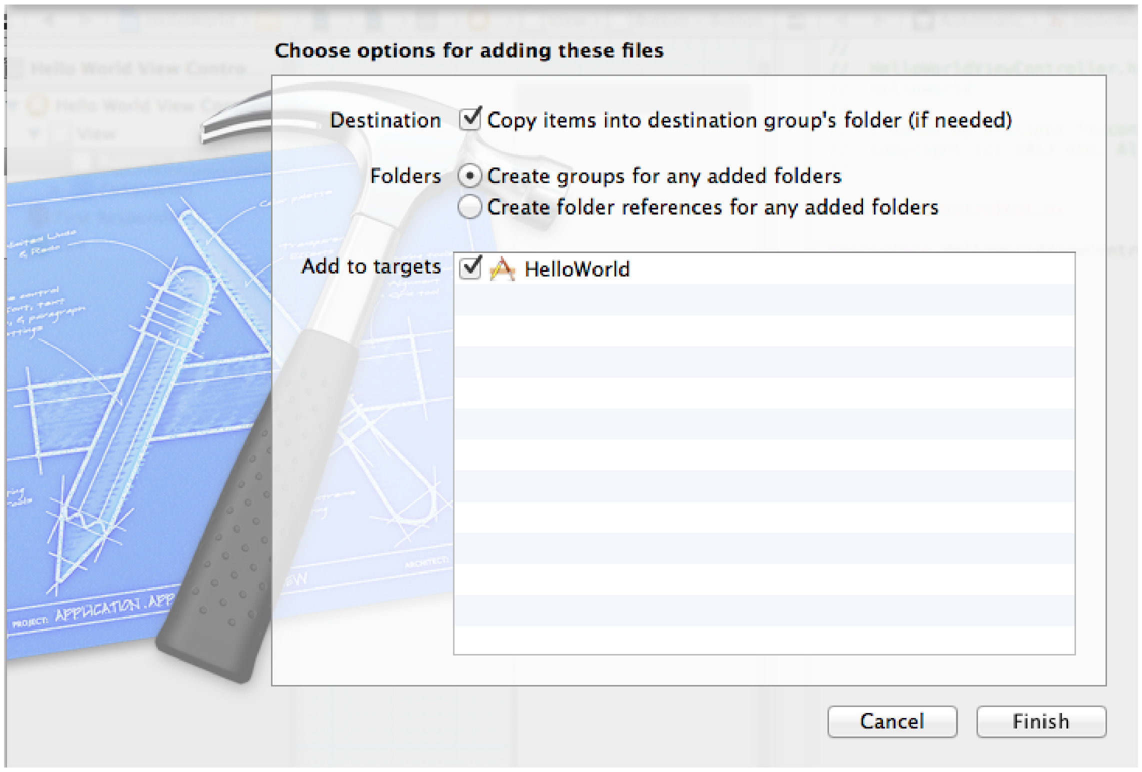
From here, you can press the "Run" button in the upper-left corner to launch the App in the iPhone Simulator. Huzzah! We have an application that runs. Now as if that wasn't enough excitement for one day let's get started with the app!

2.2.3 Completing The App

Importing Supporting Files

First we will have to download the supporting files from the camp web page. To unzip this file click on the Finder icon located on the left of your dock (bottom of your screen, it is a little smiley face). Select on the "Downloads" location on the left hand side and then double click on the zip file you downloaded. This will unzip the file and create a new folder named "Supporting Files".

Open the folder then click and drag the contents onto your Xcode project. When prompted select "Destination: Copy items into destination group's folder", keep folders the same and ensure `Add to targets: <ProjectName>` is checked. Ensure that you add the files to target and that you copy the sources into the destination group's folder.


Choose options for adding these files

Destination ☑ Copy items into destination group's folder (if needed)

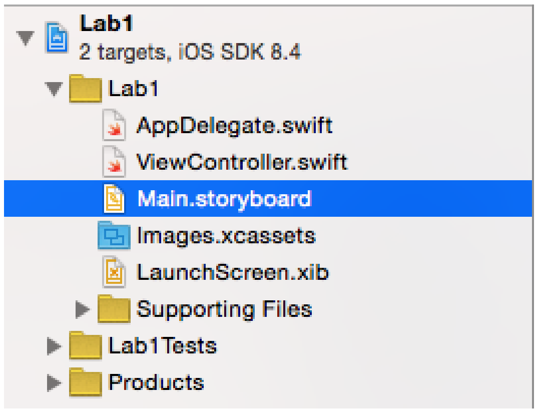
Folders ● Create groups for any added folders
○ Create folder references for any added folders

Add to targets ☑ HelloWorld

Working With The Storyboard

The next step is going to be to complete the storyboard. The storyboard is a visual representation of the program that you are making, it is a way to see what each of the screens will look like without running the program. Our storyboard is going to have 2 labels, an image, and a button.

- From the Project Navigator click on your `Main.storyboard` file.



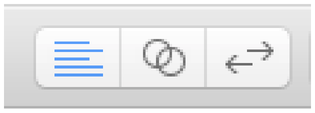
- In the bottom-right of the window you will see a cube icon; click on it and a list of user interface objects will appear.
- From here drag out 2 `Labels`, to aid in your searching there is a search bar at the very bottom , if you type into that search bar then the results will be dynamically filtered for you! After you drag them out resize them (drag from the left and right edges) so that they take the full width of the screen.
- Next we are going to set the `Text` property of each label, the easy way to do this is to double-click on the label and replace the previous text. Set the text to `Name` in one of the labels, and `Greeting` in the

other. If you run the app now you'll have some labels! We're getting closer, we just want to be able to set them from the code.

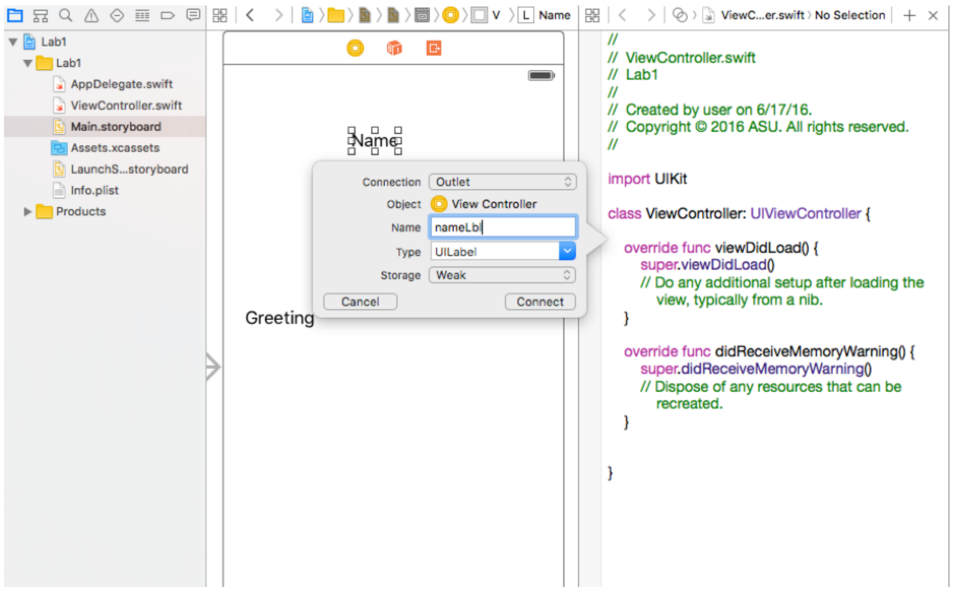
- Also drag out one `Image View` and one `Button`.
- Set the `Text` property of the button by double-clicking on it, change the new text to `Say Hello`.

To access the labels from the code we have to bind them to `IBOutlet properties`, you can think of `IBOutlet` properties as a kind of phone number for accessing the label so we can talk to it. Don't worry about the scary terminology though, setting properties up is a breeze!

First we will open the assistant editor, it's the tuxedo icon on the top right of the screen, make sure you're in the storyboard beforehand!



The assistant editor will allow us to have 2 files open at the same time; mind blowing, right? From here you will need to ctrl+click and drag from the label that says `Name` onto the source-code to the right, before `viewDidLoad()` line like in the image below. When you let go of the mouse a popover will appear, in the popover set `Name` field to `nameLabel`. Do the same greeting label and name it `greetingLabel`. Do the same thing for the `UIImageView` and name it `myPhoto`, but NOT for the button.

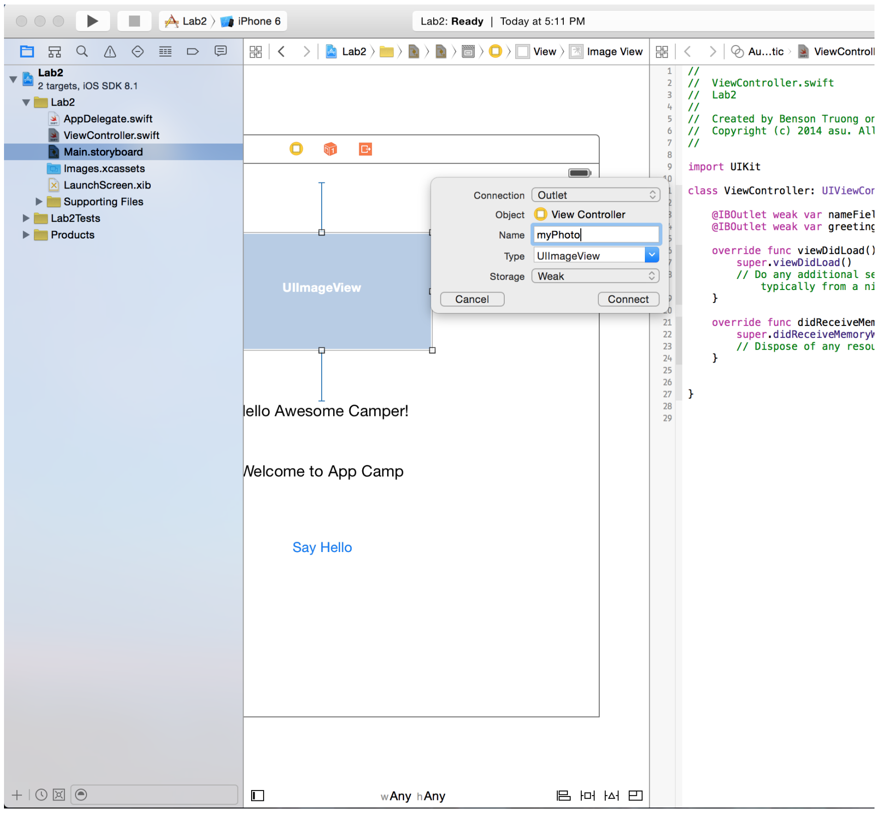


Connection: Outlet
Object: View Controller
Name: nameLbl
Type: UILabel
Storage: Weak
Cancel Connect

```
//
//  ViewController.swift
//  Lab1
//
//  Created by user on 6/17/16.
//  Copyright © 2016 ASU. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the
           view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be
           recreated.
    }

}
```

For the button we are going to use another type of connection, you may have noticed its name at the beginning of the lab... did you flip back to look at it? It's called an `IBAction`, and it conveys ACTION (but not adventure, unfortunately). Up until now everything has been an `IBOutlet`, essentially a way for the view controller code to talk to the object (label).

The `IBAction` goes the other way: it allows a button to communicate with the view controller... well, communicate might be exaggerating, about the only thing it can say is "HEY SOMEONE CLICKED ME!". As it turns out, that's exactly what we want though, so that's convenient. The first part of defining an `IBAction` is the same as an `IBOutlet`: ctrl+click and drag from the button onto the source-code right before the `viewDidLoad()` line. This time when the popover appears you will first change the `Connection` to type `Action`, and then set the name to `helloButton`.



Connection: Outlet
Object: View Controller
Name: myPhoto
Type: UIImageView
Storage: Weak
Cancel | Connect

```
//
//  ViewController.swift
//  Lab2
//
//  Created by Benson Truong or
//  Copyright (c) 2014 asu. All
//

import UIKit

class ViewController: UIViewCor

    @IBOutlet weak var nameFiel
    @IBOutlet weak var greeting

    override func viewDidLoad()
        super.viewDidLoad()
        // Do any additional se
            typically from a ni
    }

    override func didReceiveMem
        super.didReceiveMemoryW
        // Dispose of any resou
    }


}
```

Once those are finished switch back to the `Standard editor` (just left of the assistant editor)

Completing the Code

On the left of the properties there should be an open grey circle with a solid grey circle inside, this means that the `IBOutlet` or `IBAction` property is connected to the storyboard properly (if you don't see this then flag one of us down and we can figure it out!)

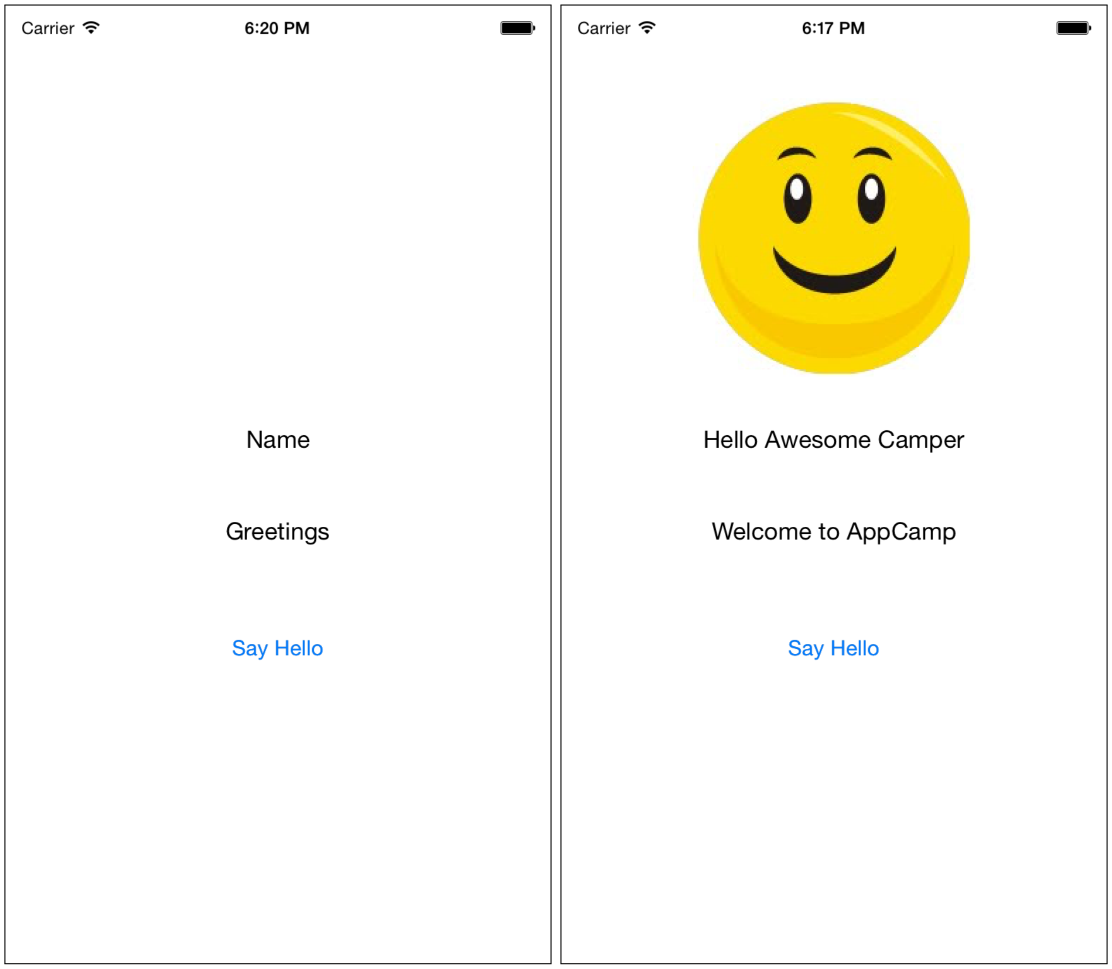
Next open your `ViewController.swift` file. This file indicates how the code should act, and what it should do (in this case it will set the `text` property of our labels! The code inside `sayHello` won't exist when you first open the file so you'll have to add it. Write the text from below so that everything matches, you can skip lines that start with //; that is a special symbol to tell the computer that a line has no commands for it.

```
class HelloWorldViewController: UIViewController {

    @IBOutlet weak var nameField: UILabel!
    @IBOutlet weak var greetingField: UILabel!
    @IBOutlet weak var myPhoto: UIImageView!

    @IBAction func helloButton(sender: AnyObject) {
        // Set the text of the Labels
        self.nameField.text = "Hello Awesome Camper"
        self.greetingField.text = "Welcome to AppCamp"

        // Smiles will be everywhere
        self.myPhoto.image = UIImage(named: "happy.jpg")
    }
}
```

Polish and Swagger

- Play around with the UI, and change the colors of the background and some of the fonts.
- If you're feeling adventurous, grab a new image from Google and replace the smiley face with that.
- If you're feeling really adventurous, try to hide the button when it is clicked. To do this you might need another property (`IBOutlet`) with a reference to the button. `UIButtons` also have a property called `hidden` which can allow a view (a UI element) to be hidden on the main screen.

Feel free to play around with things (resize the images, center the text, maybe if you're feeling adventurous grab a new image from google and switch it out)! Once you're ready, run the code and test it out:



2.2.4 Congrats!

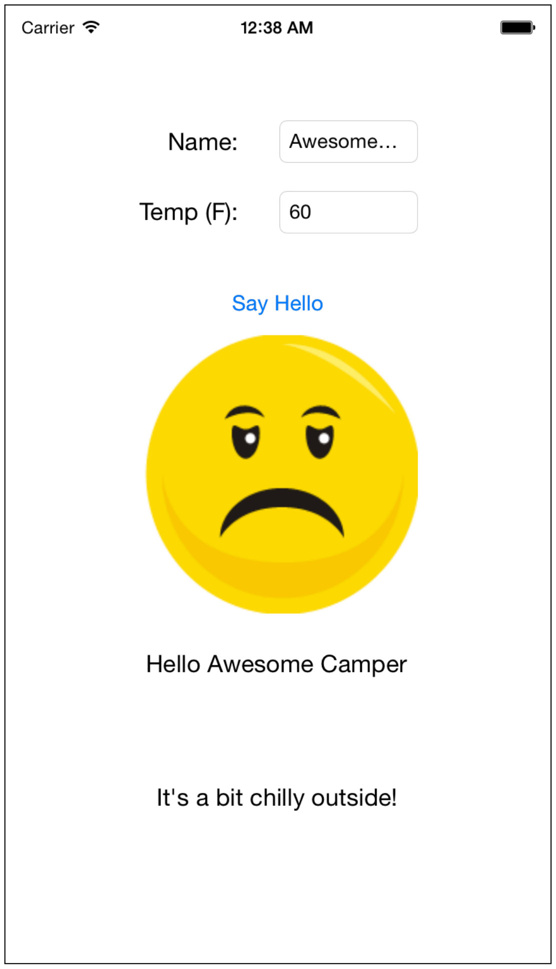
Congratulations, you finished Lab #2!

2.3 Conditionally Smiley

2.3.1 Goals of this Lab!

- Learn how to read data from the user
- Perform simple decisions using the `if-else` construct
- Collect input from the iPhone keyboard

In this application we will be making a more advanced version of the previous apps. We are going to collect user input then use it to determine whether the weather is comfortable. We will then greet the user and tell them how we feel about the weather. The final project should look like this:

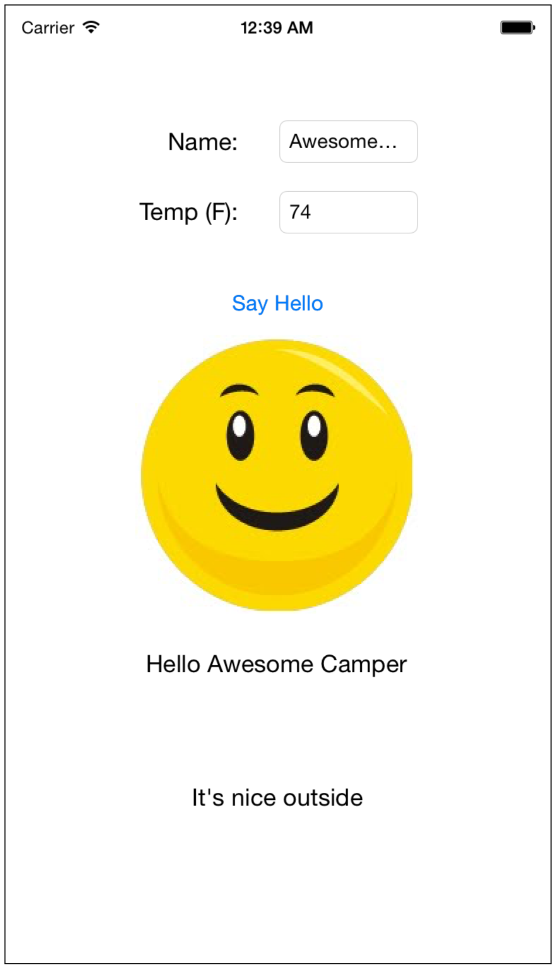


Name: Awesome...

Temp (F): 60

Hello Awesome Camper

It's a bit chilly outside!



Name: Awesome...

Temp (F): 74

Hello Awesome Camper

It's nice outside

2.3.2 Starting Our Third Project

To get started, we'll first need to open Xcode and start a project. You can find Xcode by pressing the search (magnifying glass) icon in the upper-right corner of the screen and searching in spotlight, or you can navigate to it in Finder (Applications → Xcode).

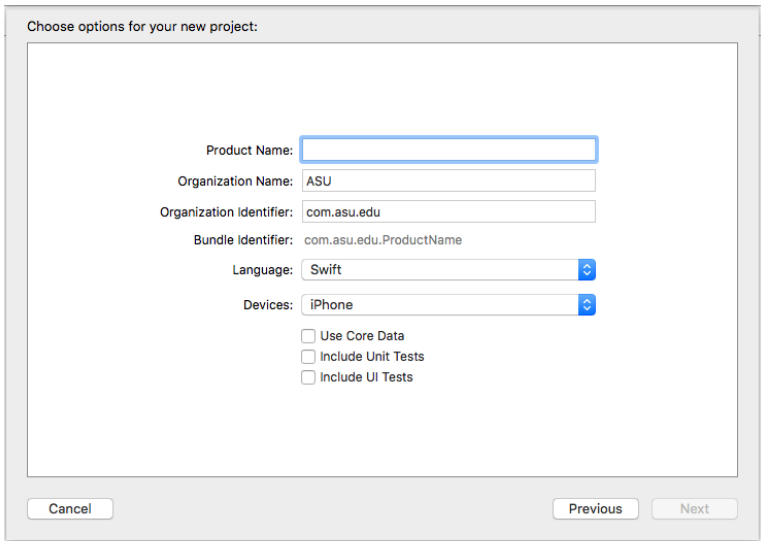
- In the "Welcome to Xcode" screen, select "Create a New Xcode Project".
- Since our App will have one screen we want to create an:
- "iOS" → "Application" → "Single-View Application" → "Next"

- We need to give the App a name, like "Lab3', as well as provide a company identifier. iPhone Apps use the "reverse domain name" style for identifying authors; so our company identifier is `asu.edu`.

- For the "Device Family" select "iPhone", so that the App is iPhone only (not also iPad).

- You can also set a "Organization Identifier" for your project. Set the Organization Identifier to "CSE".

- You will want to select "Swift" as the Language → "Next"


Choose options for your new project:

Product Name:
Organization Name: ASU
Organization Identifier: com.asu.edu
Bundle Identifier: com.asu.edu.ProductName
Language: Swift
Devices: iPhone
Use Core Data
Include Unit Tests
Include UI Tests

- You will be prompted to select a location to save your project folder, a location on the desktop will be fine, or you're welcome to use the Documents directory.

- Later we will use the git repository, for this project we won't use it though.

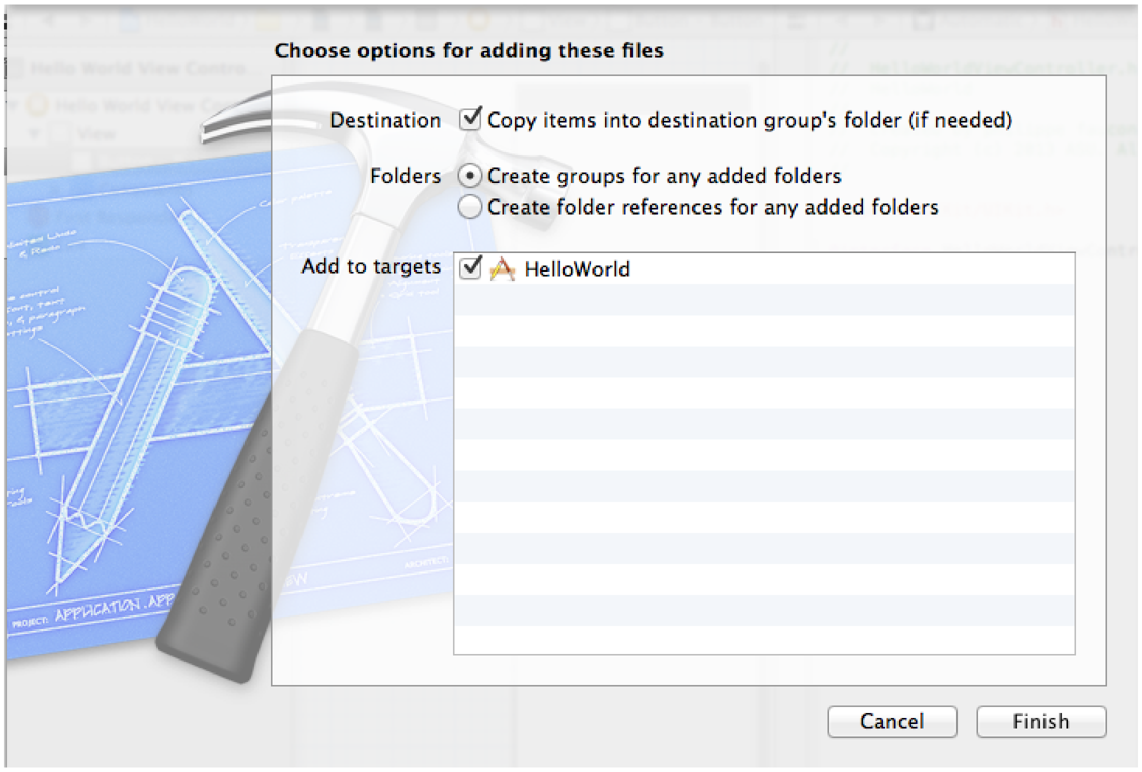
From here, you can press the "Run" button in the upper-left corner to launch the App in the iPhone Simulator. Huzzah! We have an application that runs. Now as if that wasn't enough excitement for one day let's get started with the app!

2.3.3 Completing The App

Importing Supporting Files

First we will have to download the supporting files from the camp web page. To unzip this file click on the Finder icon located on the left of your dock (bottom of your screen, it is a little smiley face). Select on the "Downloads" location on the left hand side and then double click on the zip file you downloaded. This will unzip the file and create a new folder named "Supporting Files".

Open the folder then click and drag the contents onto your Xcode project. When prompted select "Destination: Copy items into destination group's folder", keep folders the same and ensure `Add to targets: <ProjectName>` is checked. Ensure that you add the files to target and that you copy the sources into the destination group's folder.


Choose options for adding these files

Destination [✓] Copy items into destination group's folder (if needed)

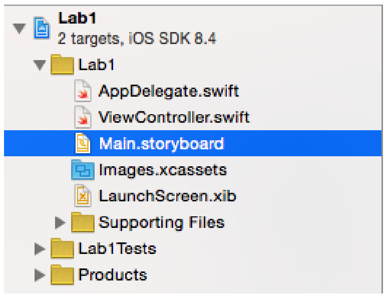
Folders [●] Create groups for any added folders
○ Create folder references for any added folders

Add to targets [✓] HelloWorld

Working With The Storyboard

The next step is going to be to complete the storyboard. The storyboard is a visual representation of the program that you are making, it is a way to see what each of the screens will look like without running the program. Our storyboard is going to have 2 `Labels`, an image (which will be used for two images), a `Button`, and 2 `Text Fields`.

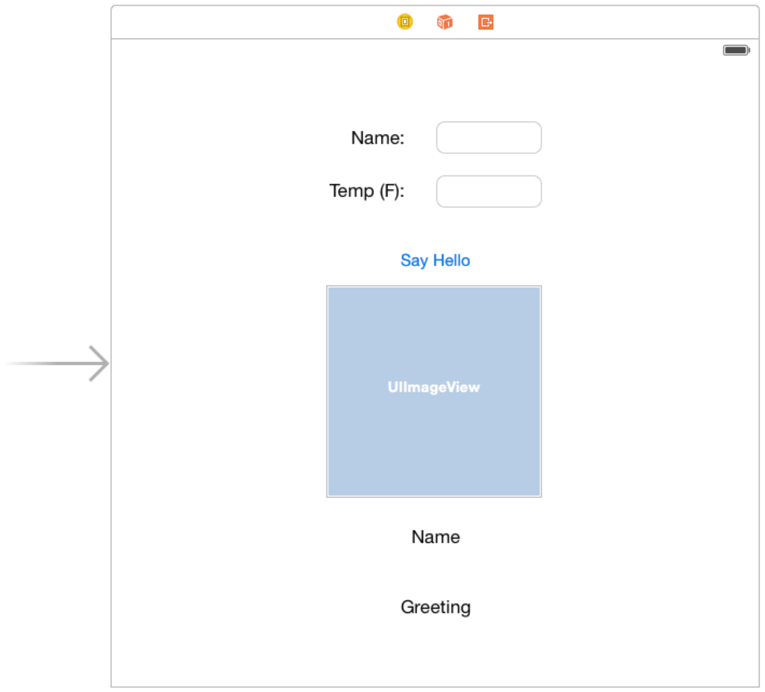
- From the Project Navigator click on your `Main.storyboard` file:



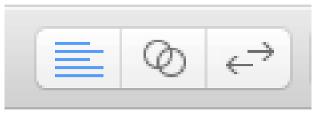
- In the bottom-right of the window you will see a cube icon; click on it and a list of user interface objects will appear.
- From here drag out 2 `Labels`; to aid in your searching, there is a search bar at the very bottom. If you type into that search bar then the results will be dynamically filtered for you! After you drag them out, resize them (drag from the left and right edges) so that they take the full width of the screen.
- Next we are going to set the `Text` property of each label, the easy way to do this is to double-click on the label and replace the previous text. Set the `text` to `Name` in one of the labels, and `Greeting` in the other. If you run the app now you'll have some labels! We're getting closer, we just want to be able to set them from the code.

- Next, drag out one `Image View` and one `Button`.
- Set the `Text` property of the button by double-clicking on it, change the new text to `Say Hello`.
- Finally drag out 2 more labels and 2 `Text Fields`, set the `Text` in the labels to `Name:` and `Temp(F):` then place them side by side.

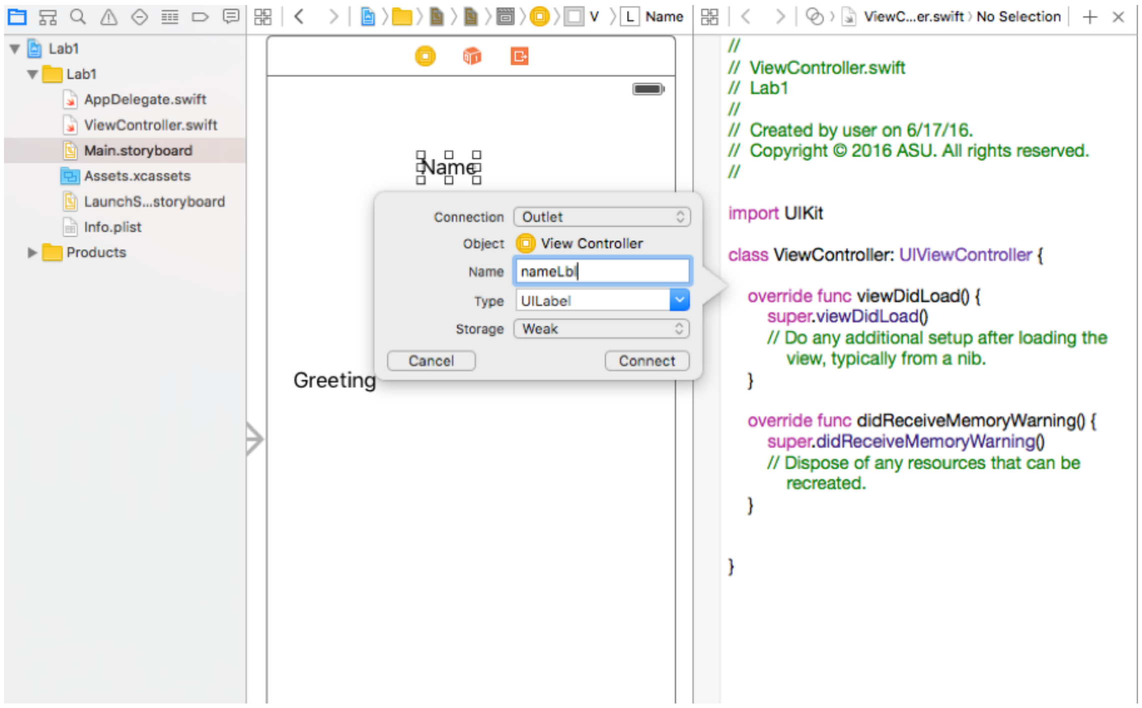
When you have finished these steps, your interface should look something like the image below:



We have a new view type that we are using. To access the `UITextField` views we will configure `IBOutlet properties` like we did previously. This is because we want to be able to ask the text field what the user has entered. You might be wondering if we can ask the text field to just tell us when a user enters something... the answer is yes of course, because programmers are awesome. The way `UITextFields` communicate back to a view controller (without being asked); this is a little more complicated than what buttons do though, so we will cover it later. For now, let's set up our `IBOutlets`:



The assistant editor, shown above, will allow us to have 2 files open at the same time; mind blowing, right? From here you will need to ctrl+click and drag from the label that says `Name` onto the source-code to right before the `viewDidLoad()` line, like in the image below. When you let go of the mouse a popover will appear, in the popover set `Name` field to be `nameLabel`. Do the same for the greeting label and name it `greetingLabel`. We don't need to attach the other labels to the backing code (the code won't change what they say). For the 2 `Text Fields`, name them `nameText` and `tempText`. Also, connect the `UIImageView` and name it `myPhoto`, but NOT the button.

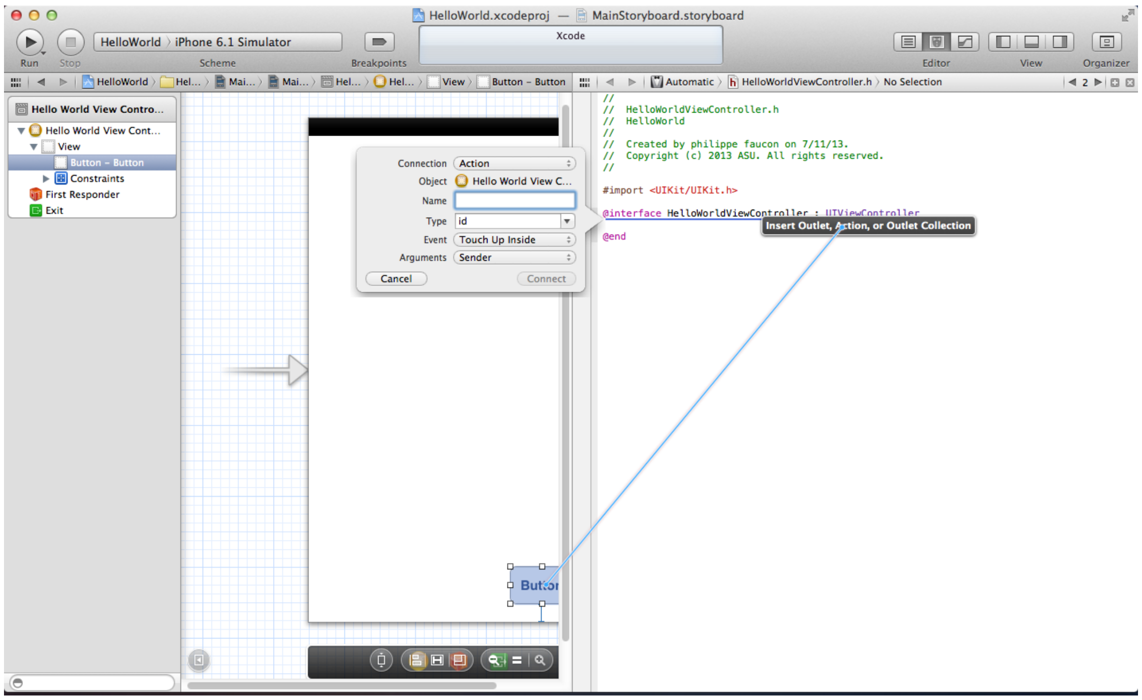


Connection: Outlet
Object: View Controller
Name: nameLbl
Type: UILabel
Storage: Weak

```
//
//  ViewController.swift
//  Lab1
//
//  Created by user on 6/17/16.
//  Copyright © 2016 ASU. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the
            view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be
            recreated.
    }


}
```

For the button we are going to use another type of connection, you may have noticed its name at the beginning of the lab... did you flip back to look at it? It's called an `IBAction`, and it conveys ACTION (but not adventure, unfortunately). Up until now everything has been an `IBOutlet`, essentially a way for the view controller code to talk to the object (label).

The `IBAction` goes the other way: it allows a button to communicate with the view controller... well, communicate might be exaggerating, about the only thing it can say is "HEY SOMEONE CLICKED ME!". As it turns out, that's exactly what we want though, so that's convenient. The first part of defining an `IBAction` is the same as an `IBOutlet`: ctrl+click and drag from the button onto the source-code right before the `viewDidLoad()` line. This time when the popover appears you will first change the `Connection` to type `Action`, and then set the name to `helloButton`.



```
//
//  HelloWorldViewController.h
//  HelloWorld
//
//  Created by philippe faucon on 7/11/13.
//  Copyright (c) 2013 ASU. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface HelloWorldViewController : UIViewController

@end
```

Connection: Action
Object: Hello World View C...
Name:
Type: id
Event: Touch Up Inside
Arguments: Sender

HelloWorld.xcodeproj — MainStoryboard.storyboard
Xcode
Run
Stop
HelloWorld 〉 iPhone 6.1 Simulator
Scheme
Breakpoints
Editor
View
Organizer
Automatic 〉 HelloWorldViewController.h 〉 No Selection
Hello World View Contro...
Hello World View Cont...
View
Button - Button
Constraints
First Responder
Exit
Cancel
Connect
Insert Outlet, Action, or Outlet Collection
Button

Once those are finished switch back to the `Standard editor` (just left of the assistant editor).

Completing the Code

On the left of the properties there should be an open grey circle with a solid grey circle inside, this means that the `IBOutlet` or `IBAction` property is connected to the storyboard properly (if you don't see this then flag one of us down and we can figure it out!)

Next open your `ViewController.swift` file. This file indicates how the code should act, and what it should do (in this case it will set the `text` property of our labels, and will set our image for us... conditionally! The code inside `sayHello` won't exist when you first open the file, so you'll have to add it. Write the text from below so that everything matches, you can skip lines that start with `//`, that is a special symbol to tell the computer that a line has no commands for it.

```
class ViewController: UIViewController {

    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var greetingLabel: UILabel!
    @IBOutlet weak var nameText: UITextField!
    @IBOutlet weak var tempText: UITextField!
    @IBOutlet weak var myPhoto: UIImageView!

    @IBAction func sayHello(sender: AnyObject) {
        // this says set the text in the name label to a new string
        //that is made by sticking together 2 strings.
        //Those 2 strings are "Hello " and whatever the user entered
        self.nameLabel.text = "Hello " + self.nameText.text

        // Check is the temperature less than 70?
        if Int(self.tempText.text!)! < 70 {
            self.greetingLabel.text = "It's a bit chilly outside!"
            self.myPhoto.image = UIImage(named: "sadFace.png")
        }
        // Check is it more than 95?
        else if Int(self.tempText.text!)! > 95 {
            self.greetingLabel.text = "It's a bit toasty outside!"
            self.myPhoto.image = UIImage(named: "sadFace.png")
        }

        // Any other possibility, in this case > 70 and < 95
        else {
            self.greetingLabel.text = "It's nice outside"
            self.myPhoto.image = UIImage(named: "happy.jpg")
        }
    }
}
```

Now take the application for a spin! Oh no, that ugly text field just won't leave us alone! We are done with it but it is still sticking around! The reason for this is that the text field has "first responder" status, which means it thinks it's all cool and can own the screen. We know that when a user presses our button that they are done entering text though. So we should close the keyboard when that happens; to do that, we can just send a message to the text field which tells it to go away. One problem we have is that we're not sure which view thinks it is first responder, so we just have to tell all of them to hide the keyboard. Add the code at the end of the `sayHello` method.

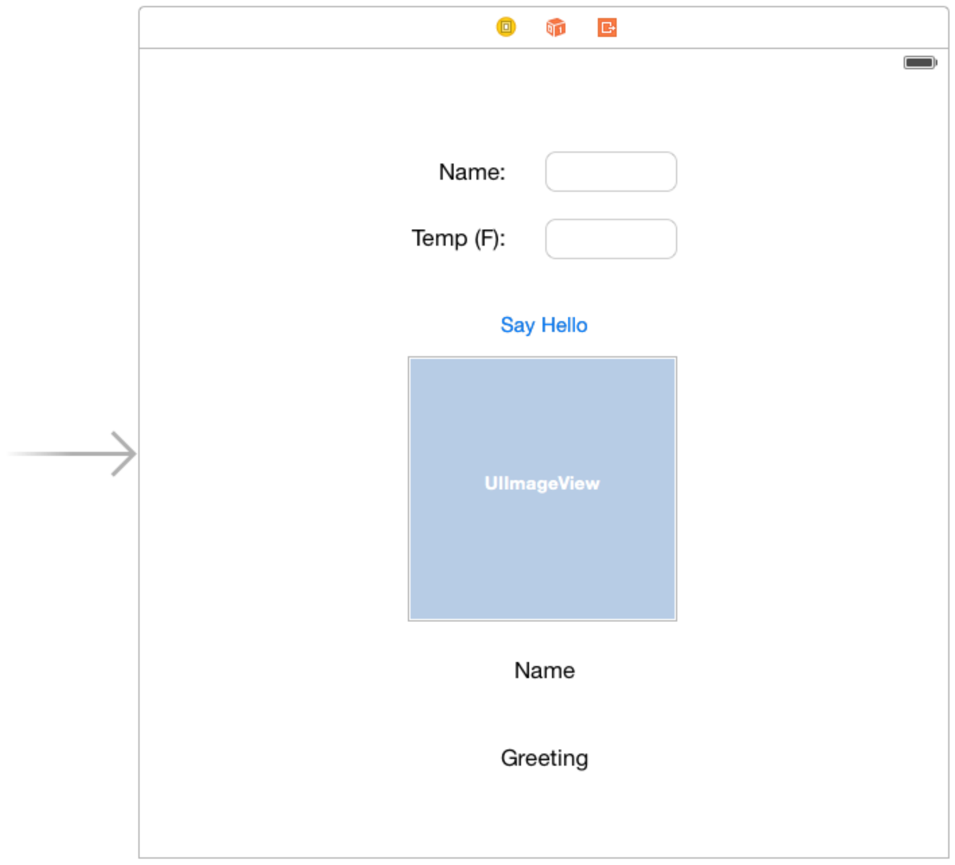
```
class ViewController: UIViewController {

    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var greetingLabel: UILabel!
    @IBOutlet weak var nameText: UITextField!
    @IBOutlet weak var tempText: UITextField!
    @IBOutlet weak var myPhoto: UIImageView!

    @IBAction func sayHello(sender: AnyObject) {
        // this says set the text in the name label to a new string
        //that is made by sticking together 2 strings.
        //Those 2 strings are "Hello " and whatever the user entered
        self.nameLabel.text = "Hello " + self.nameText.text

        // Check is the temperature less than 70?
        if Int(self.tempText.text!)! < 70 {
            self.greetingLabel.text = "It's a bit chilly outside!"
            self.myPhoto.image = UIImage(named: "sadFace.png")
        }

        // Check is it more than 95?
        else if Int(self.tempText.text!)! > 95 {
            self.greetingLabel.text = "It's a bit toasty outside!"
            self.myPhoto.image = UIImage(named: "sadFace.png")
        }

        // Any other possibility, in this case > 70 and < 95
        else {
            self.greetingLabel.text = "It's nice outside"
            self.myPhoto.image = UIImage(named: "happy.jpg")
        }
        // add me!
        self.nameText.resignFirstResponder()
        self.tempText.resignFirstResponder()
    }
}
```

Polish and Swagger

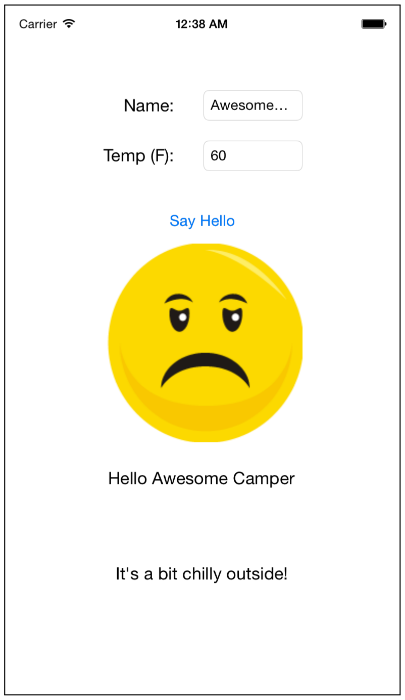
Try to add new temperature ranges; for example, if the temperature is less than 32, tell the user to be careful of ice, or if the temperature is above 120, welcome them to Arizona. Hint: try to use the `else if` tool we just learned, and keep in mind that the code is analyzed sequentially (this might be a problem because if something is less than 32 it's definitely less than 70).

Another fun question: what happens if you enter something that isn't a number into the temperature? Do you have any ideas why? Feel free to ask one of the helpers about it!

The final storyboard layout should look something like the image below:

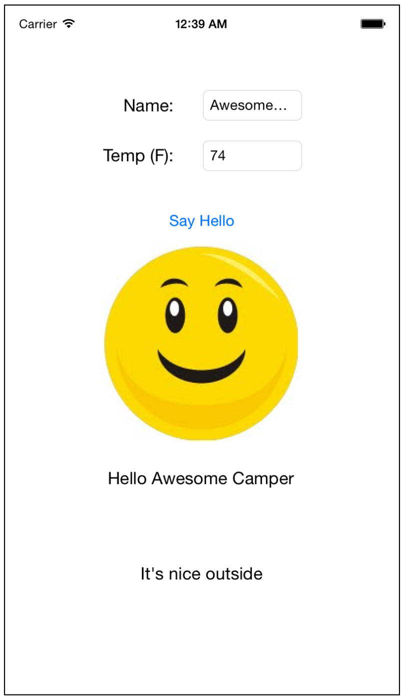


The final application, when you run it in the simulator, should look something like the images below:



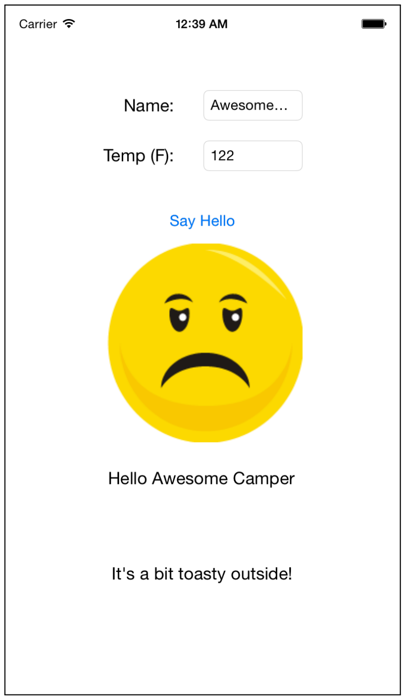
Hello Awesome Camper

It's a bit chilly outside!



Hello Awesome Camper

It's nice outside



Hello Awesome Camper

It's a bit toasty outside!

2.3.4 Congrats!

Congratulations, you finished lab 3!

Chapter 3

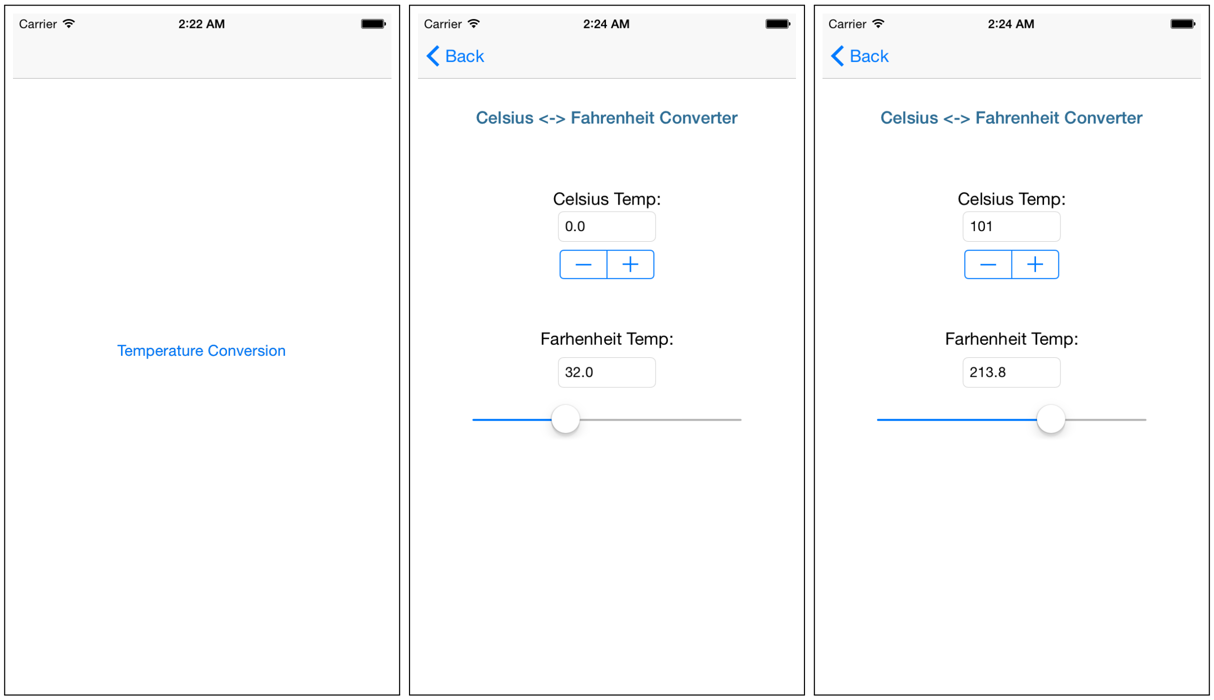
Segues and Multi-View Applications

3.1 Conversions

3.1.1 Content In this Lab!

- Developing multi-view applications using the storyboard
- Collecting data from the `Slider`, `Stepper`, and `Segment UI` objects
- Interaction and event handling with UI objects

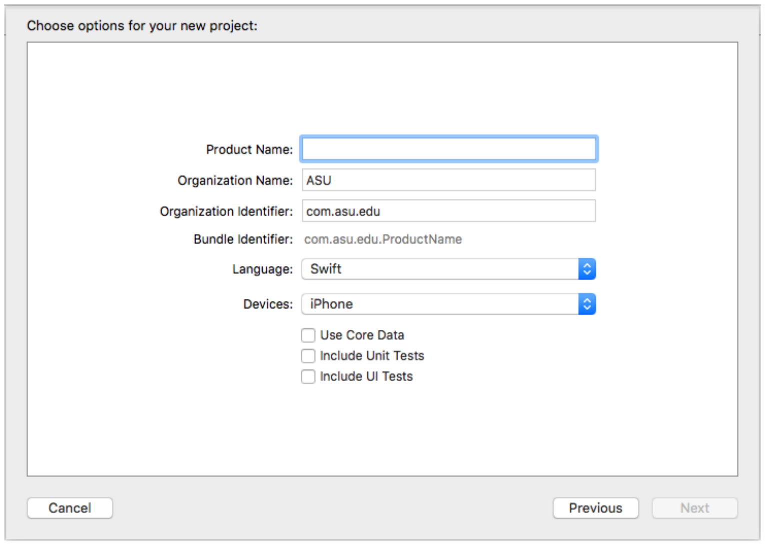
In this lab we will be developing a multi-view application that performs unit conversions. One view will perform temperature conversions (Celsius to Fahrenheit) and the other will convert distances (miles to kilometers). The first phase is required while the second phase is optional. The final project should look like this:



3.1.2 Starting Our Conversions Project

To get started, we'll first need to open Xcode and start a project. You can find Xcode by pressing the search (magnifying glass) icon in the upper-right corner of the screen and searching in spotlight, or you can navigate to it in Finder (Applications → Xcode).

- In the "Welcome to Xcode" screen, select "Create a New Xcode Project".
- Since our App will have one screen we want to create an:
- "iOS" → "Application" → "Single-View Application" → "Next"
- We need to give the App a name, like "Lab3', as well as provide a company identifier. iPhone Apps use the "reverse domain name" style for identifying authors; so our company identifier is `asu.edu`.
- For the "Device Family" select "iPhone", so that the App is iPhone only (not also iPad).
- You can also set a "Organization Identifier" for your project. Set the Organization Identifier to "CSE".
- You will want to select "Swift" as the Language" → "Next"


Choose options for your new project:

Product Name:
Organization Name: ASU
Organization Identifier: com.asu.edu
Bundle Identifier: com.asu.edu.ProductName
Language: Swift
Devices: iPhone
Use Core Data
Include Unit Tests
Include UI Tests

- You will be prompted to select a location to save your project folder, a location on the desktop will be fine, or you're welcome to use the Documents directory.
- Later we will use the git repository, for this project we won't use it though.

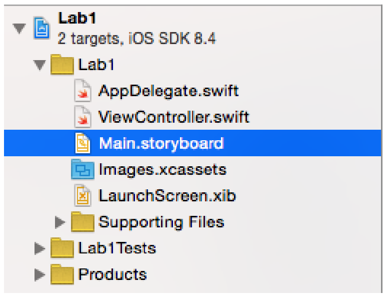
From here, you can press the "Run" button in the upper-left corner to launch the App in the iPhone Simulator. Huzzah! We have an application that runs. Now as if that wasn't enough excitement for one day let's get started with the app!

3.1.3 Completing The App

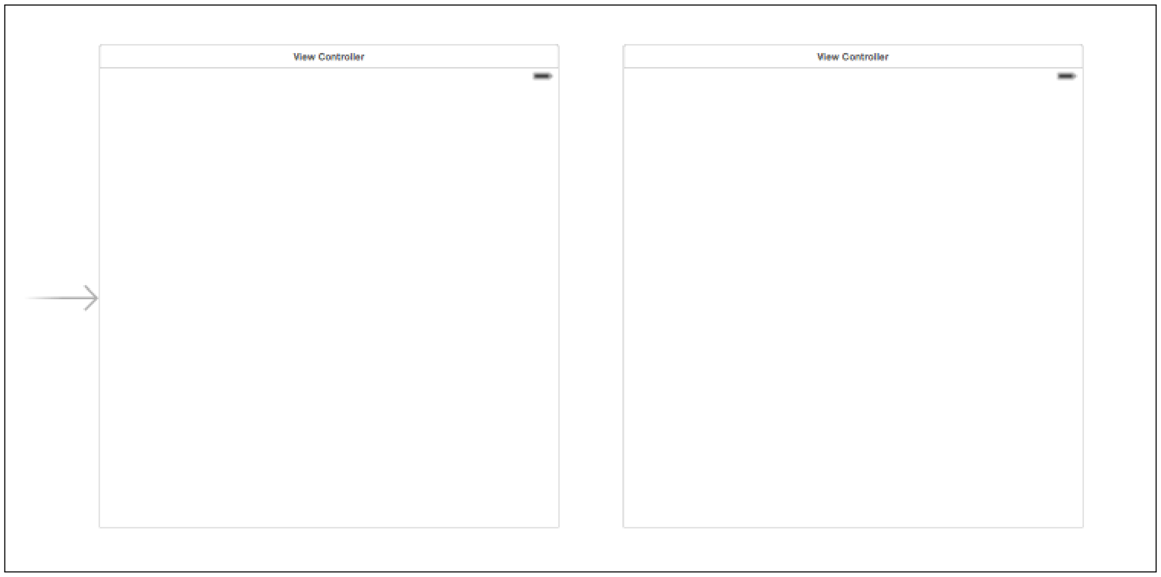
Setting Up the Interface

The next step is going to be to set up the interface on the storyboard. The storyboard is a visual representation of the program that you are making. It is a way to see what each of the screens will look like without running the program. In this lab we are going to be adding a few new things; most importantly, we will be adding new scenes with special view controllers.

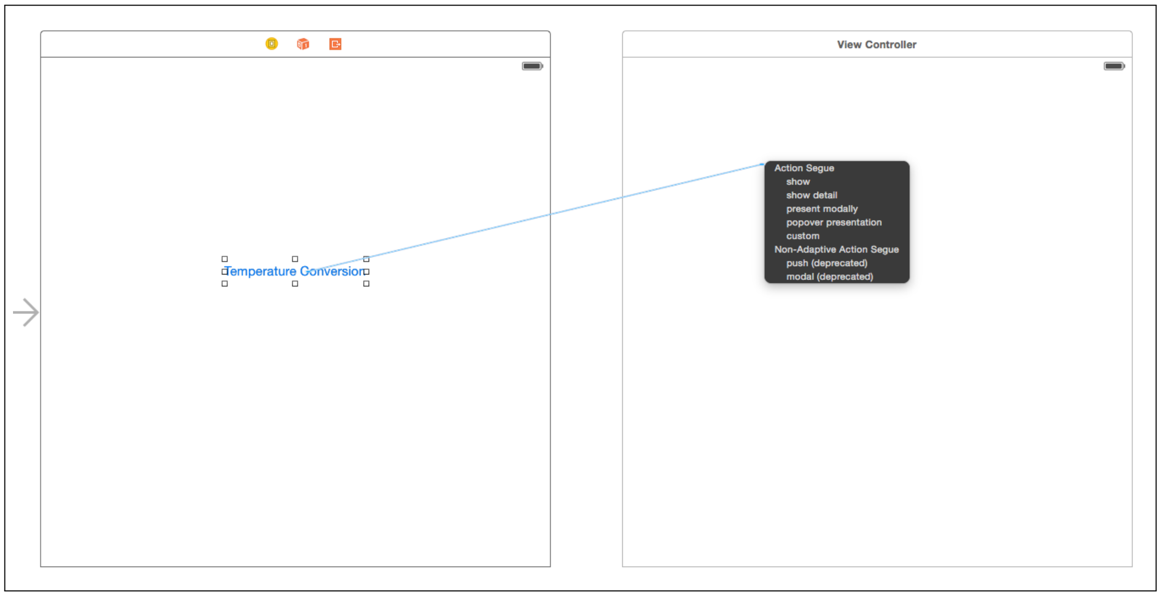
- From the Project Navigator click on the `Main.storyboard` file.



- In the bottom-right of the window you will see a cube icon; click on it and a list of user interface objects will appear.
- The first thing that we are going to do is to define our transition (like the first example image). To do this, let's drag out a `Rounded Rect Button`, change the `Title` property to `Temperature Conversion`.
- Next add one new `View Controller` from the object library. Unlike the views that we previously added, our new view controller object does not go inside of the other one, it will go beside it. Once it has been dragged out, it should look like the image below:

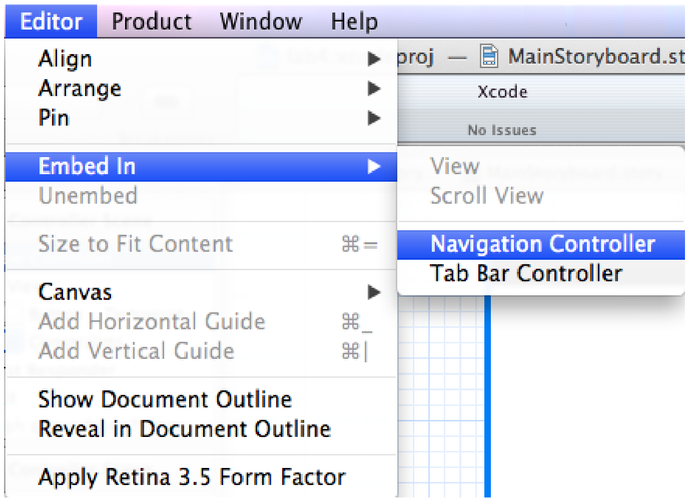


- Next we are going to define a `push segue` between the two views. This tells the program to show the next view controller as the main screen. We define a segue from a button by ctrl+click and dragging from the button to the next view controller, and then selecting `Push` like in the image below:

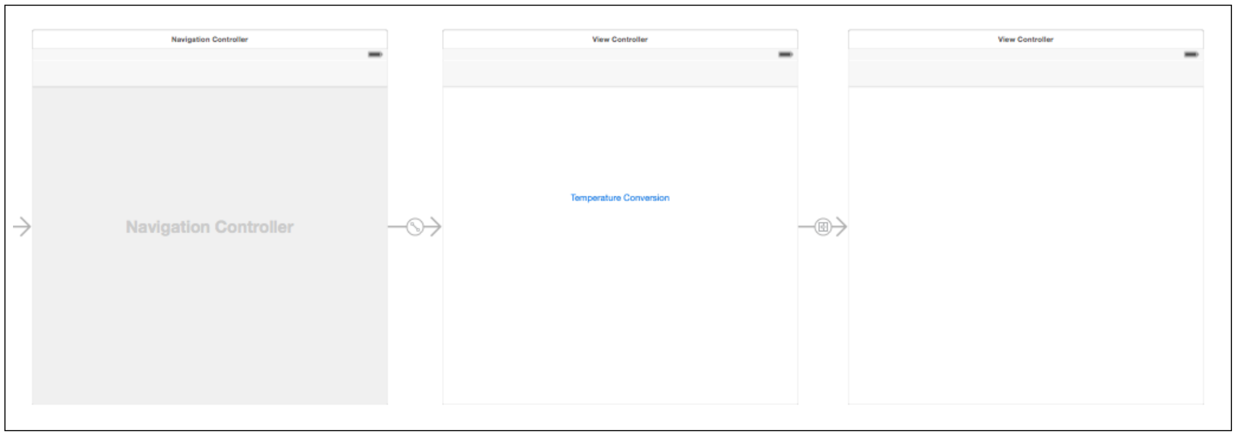


- If you try to run the program now, the button will not take you to the next screen, and that is for one very important reason. The segue describes how to transport the application from one view to another, but it does not contain a way to perform the change. What we need now is called a `Navigation Controller`, and it will allow us to navigate between views.

  To add a navigation controller, select your first view controller (the one with the button), then at the top of the screen, select Editor → Embed In → Navigation Controller, as shown below:



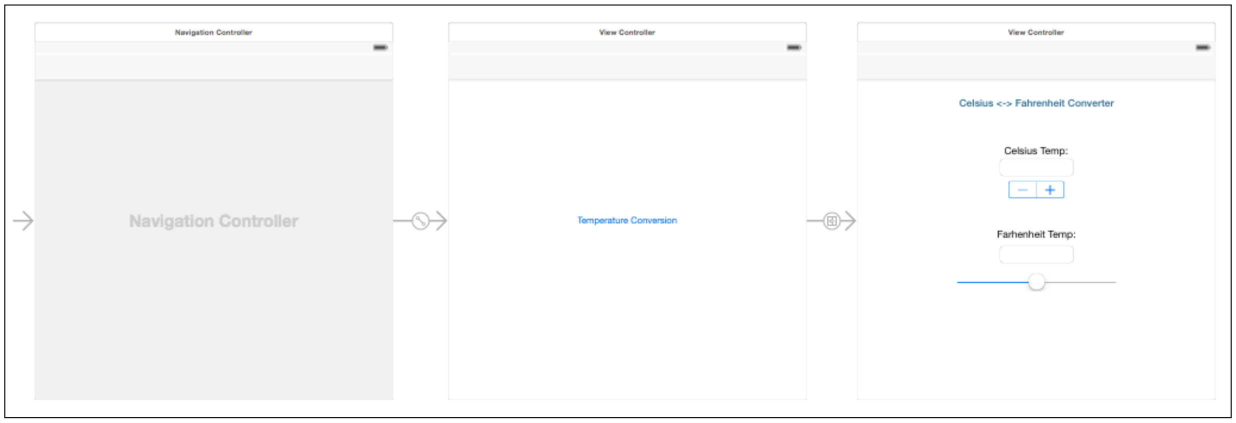
- Adding the Navigation Controller should add a View Controller-looking thing to the left of your first view controller, and all your view controllers should have a gray bar at the top now. To help show the differences between the views, set the color on your first view controller to something new (as a refresher you need to select the view (not the view controller) and set the `Background` property in the `Attributes Inspector`). Now test out your application and it should let you change between views; if not, then look back through the steps in case you missed something.



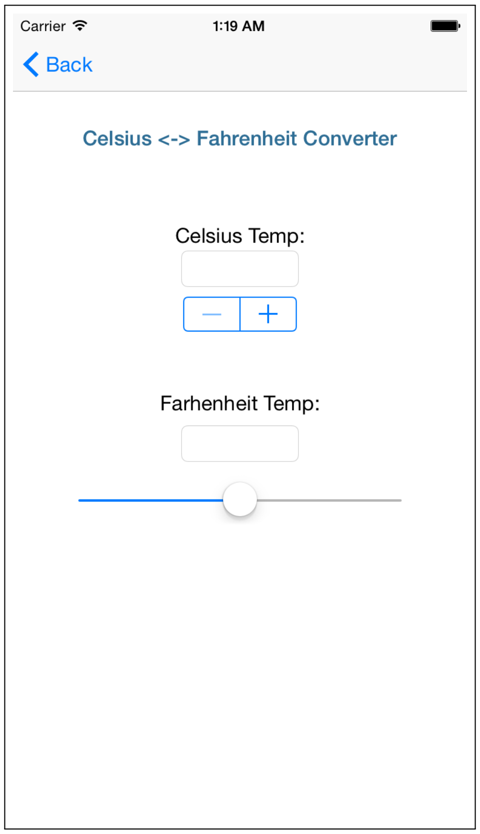
Ok, now we need to set up our new view (not the one connected directly to the navigation controller, the one we dragged out from the object library). First let's add all of our components, we're going to need:

- 1 `Slider` object
- 2 `Text Field` objects
- 1 `Stepper` object
- 3 `Labels` (2 as a description for the text fields, and 1 as an explanation for the page)

Lay them out so they look like the view below. Feel free to change the text fonts (but try not to spend too much time customizing, we have a lot of cool things to cover!).



When you have finished these steps your interface should look something like the image below:



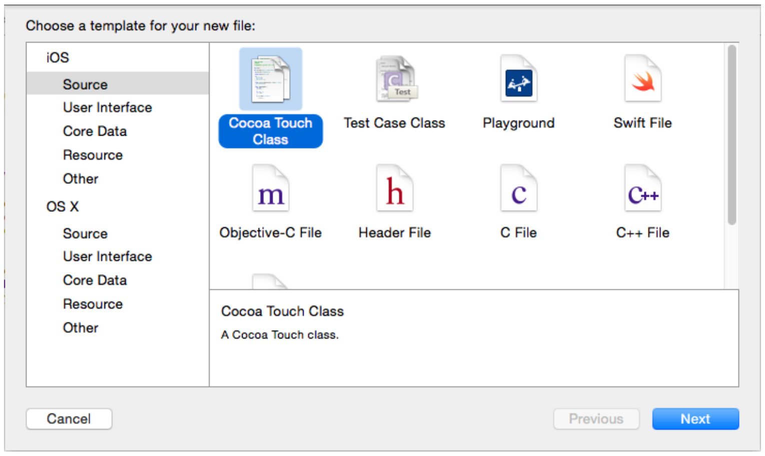
In this lab we are going to use two types of communication for our `UITextField`, `UISlider`, `UIStepper`, and `UITextField`. The first is the `IBOutlet` type, so that we are able to send data from our code to the views. The second is a new type called *delegation*, a way for the view to communicate to the code that an event has happened. Essentially we will be using the delegation to find out when values change in the text fields, and we will still use `IBActions` to read the slider and stepper's values. In Xcode we can set up delegation using only the code, or using the code and the Interface Builder; for now we will use the hybrid approach.

Creating and Configuring a New View Controller

Before we are able to modify our code we need to create the code, and associate it with our storyboard:

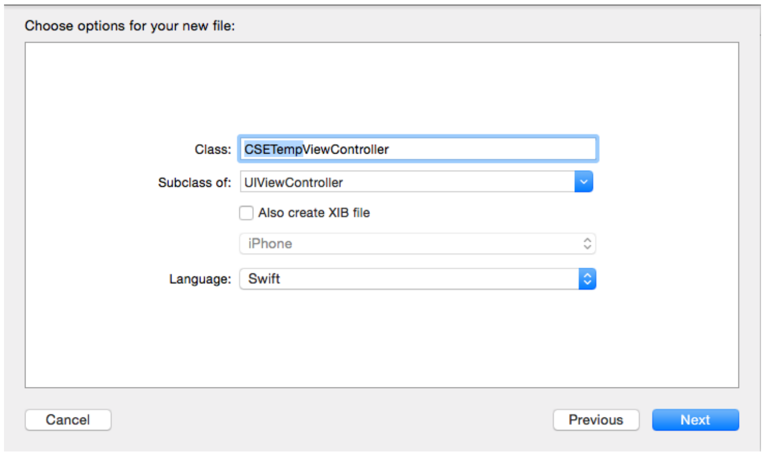
- To create new view controller source files select from the top menu `File` → `New` → `File`

- In the window that appears select `Cocoa Touch Class` and then press `next`



Cocoa Touch Class
A Cocoa Touch class.

- Set the `Subclass of` field to be `UIViewController` and then enter the `Class` field as `CSETempViewController` and press `next`.



Class: CSETempViewController
Subclass of: UIViewController
Also create XIB file
iPhone
Language: Swift

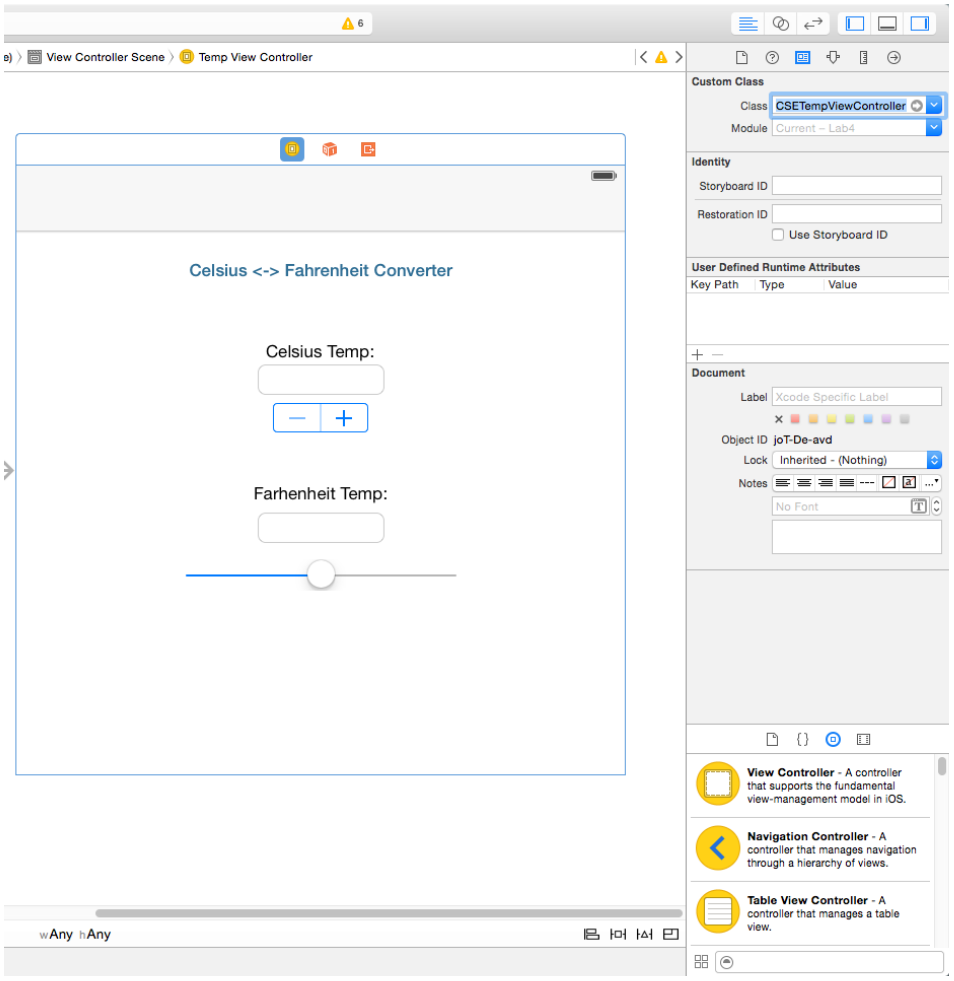
- Ensure that the checkbox beside `Targets` is checked, this makes sure that Xcode builds the new file as part of our app, and then press `Create`.

Now we have our backing View Controller! If Xcode changed what you're looking at switch back to the storyboard for the next subsection, we're not quite done with it.

Making Connections

At this point we should have our interface completed and view controller files. The next thing that we will need to do is connect the views to the view controllers, and then write the code to control them.

The first thing that we need to do is to tell Xcode that the new `View Controller` object we put on our storyboard relates to the `CSETempViewController` that we created just a moment ago. To do this select the `View Controller` object from our storyboard (tap on the top near where the batter is), once it is selected the entire scene should have a blue frame. Now go to the `Identity Inspector` and set the `Class` property to `CSETempViewController` (it should auto-complete as you type):



Class: CSETempViewController
Module: Current – Lab4
Object ID: joT-De-avd
Lock: Inherited - (Nothing)

View Controller - A controller that supports the fundamental view-management model in iOS.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Table View Controller - A controller that manages a table view.

Now let's set up the `IBOutlets` and actions for each of our objects. Use the same procedure as before with the assistant editor and dragging. Remember that for the slider and stepper you will need BOTH an `IBAction` and an `IBOutlet`. If you have forgotten the protocol for creating outlets and actions look in the labs from before. Set the `IBOutlet` names for the `UITextFields` to `cTempText` and `fTempText`, the stepper to `cTempStepper`, and the slider to `fTempSlider`. For the actions, name them `sliderChanged` and `stepperChanged`.

Completing the Code

Open your `CSETempViewController.swift` file. It should look familiar, we were just editing it with the assistant editor! Just for verification make sure that it looks something like below:

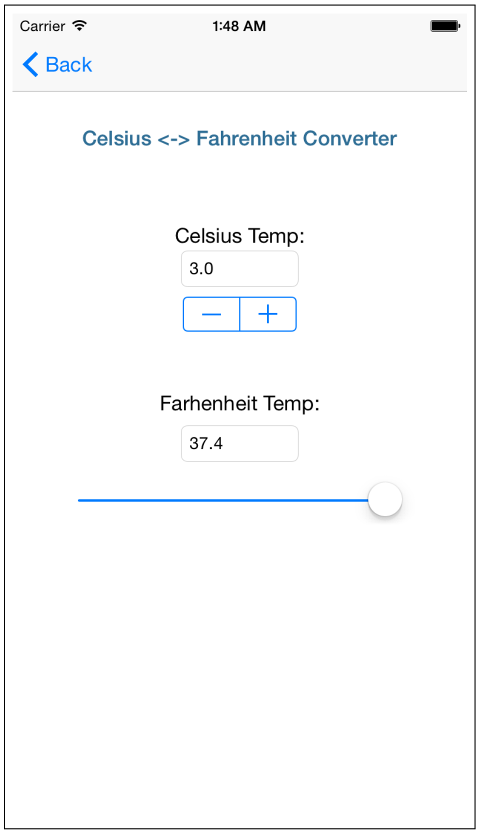
```
import UIKit

class CSETempViewController: UIViewController {

    @IBOutlet weak var cTempText: UITextField!
    @IBOutlet weak var fTempText: UITextField!
    @IBOutlet weak var cTempStepper: UIStepper!
    @IBOutlet weak var fTempSlider: UISlider!

    @IBAction func stepperChanged(sender: AnyObject) {
    }
    @IBAction func sliderChanged(sender: AnyObject) {
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }

}
```

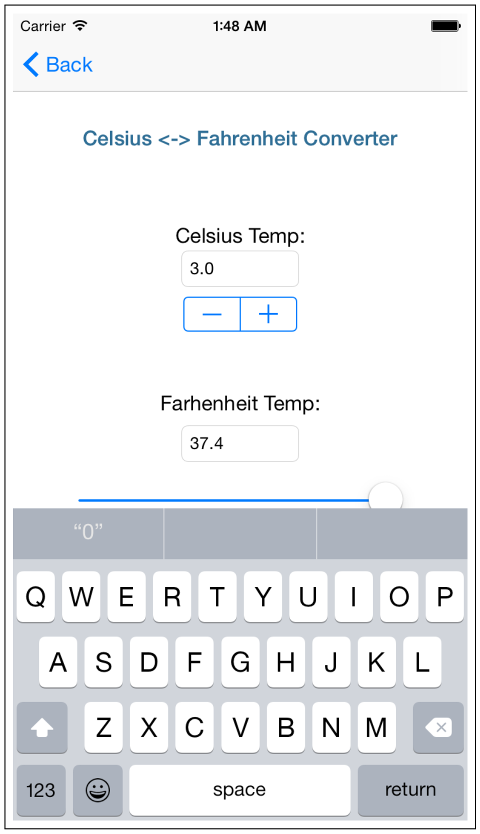
On the left of the properties there should be an open grey circle with a solid grey circle inside; this means that the IBOutlet or IBAction property is connected to the storyboard properly (if you don't see this then flag one of us down and we can figure it out!).

Add the getCelsius and getFahrenheit methods, and edit the code in for stepperChanged and sliderChanged (they should already exist in the file, but they will be empty).

```
class CSETempViewController: UIViewController {

    @IBOutlet weak var cTempText: UITextField!
    @IBOutlet weak var fTempText: UITextField!
    @IBOutlet weak var cTempStepper: UIStepper!
    @IBOutlet weak var fTempSlider: UISlider!

    // when someone changes the stepper value we get the
    // phone number of the stepper that changed
    @IBAction func stepperChanged(sender: UIStepper) {
        let cTemp = Double(sender.value)

        // create a new string which contains a "float"
        self.cTempText.text = String(format: "%.1f", cTemp)

        //use our converter
        let fTemp = getFahrenheit(cTemp)

        //set the slider and fahrenheit text
        self.fTempSlider.value = Float(fTemp)
        self.fTempText.text = String(format: "%.1f", fTemp)
    }

    @IBAction func sliderChanged(sender: UISlider) {
        let fTemp = Double(sender.value)
        self.fTempText.text = String(format: "%.1f", fTemp)

        //use our converter
        let cTemp = getCelsius(fTemp)

        //set the stepper and celsius text
        self.cTempStepper.value = Double(cTemp)
        self.cTempText.text = String(format: "%.1f", cTemp)
    }

    // these are helper methods which will transform
    // our celsius temp to fahrenheit, and vice versa
    func getCelsius(fahrenheit : Double) -> Double {
        //order of operations is important here!
        return (fahrenheit - 32) * 5/9
    }

    func getFahrenheit(celsius : Double) -> Double {
        //order of operations is important here!
        return celsius * 9/5 + 32
    }

}
```

Now take the application for a spin! One thing you should notice immediately is that the slider is currently completely useless, this is because by default the values of a slider range from 0-1, so let's fix that. The next things are that the stepper also has lame boundaries, it won't let us have negative temperatures, and the text fields won't go away.





Round 2!

Let's return to the storyboard and fix the boundary issues first. Select the stepper and then go to the `Attributes Inspector`, set the `Minimum` to -100, and the `Maximum` to 200. You can also change the step size if you want but 1 is reasonable. Next select the slider and in the `Attributes Inspector` set the `Minimum` to -150, and the `Maximum` to 400. Now if we run it again we can use the whole range!

Next return to your `CSETempViewController.swift` file and add the following method. This method is called automatically when a touch is in our view (not in the keyboard) and will hide the keyboard.

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
        self.view.endEditing(true)
        super.touchesBegan(touches, withEvent: event)
}
```

3.1.4 Polish and Swagger

At this point the app is finished, play around with it and make sure everything is good. If there is a little extra time play around with the ranges and possibly look into the `Text Field` delegate information, see if there is anything that interests you there. If you blazed through this and have a lot of extra time implement the distance conversions, most of that new view controller would be identical except the conversion methods. Here are some quick conversions:

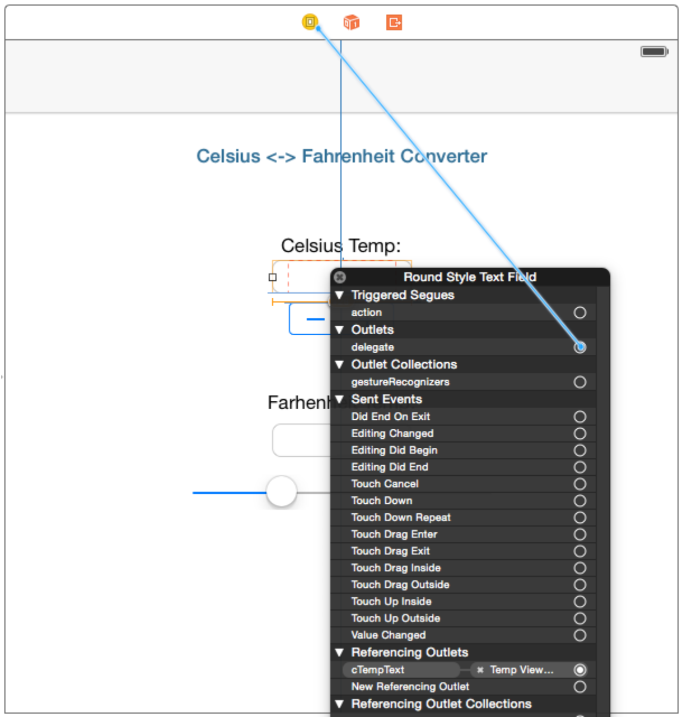
- 1ft = 0.305 meters
- 1 mile = 1.61 kilometers

Delegatin' the Text Field side

This is an extra subsection, and is conceptually a bit difficult; if you skip or don't understand this it is ok, we'll cover it in more detail later. One thing we want to be able to do is read input from our text fields without a button so that we can use them as more than labels. To do this we will utilize a tool called delegation. To set this up, go back to the storyboard, select your temperature conversion view controller and open the `Assistant Editor`. On the right-hand side (where your `.swift` file is), we're going to edit the `class` line (and only that line), add the `UITextFieldDelegate` component:

```
class CSETempViewController: UIViewController, UITextFieldDelegate
```

This announces that our view controller is able to act as a delegate for a `UITextField`. The next step will be telling the text field that we are going to be its delegate (Xcode now knows that we are capable of being a delegate, but we haven't told the text field yet). To do this ctrl+click (no drag) on the text field, then go to `Outlets` → `delegate` and drag from the circle on the right to the top of the view onto the yellow icon as shown below. Be sure to do the same for the other text field.

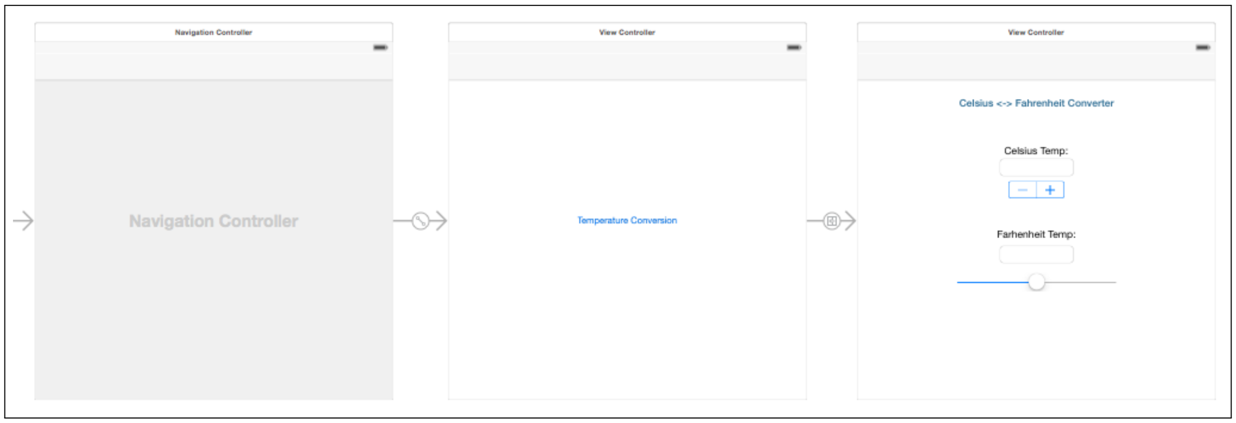


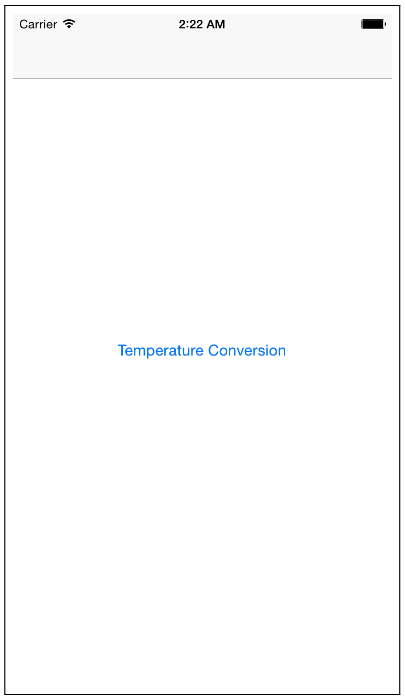
Great, now Xcode knows the text field can talk to us, and the text field knows it should talk to us, the last step is find out what to do when it does talk to us! Delegation is much less structured and much more powerful than IBAction, so it would be silly to add everything that you could possible react to from a delegate. As a result Xcode adds nothing, so we'll have to add these methods ourself. For now we'll just tell you which methods we're using, but if you want to see all of the available text field methods you can go into `CSETempViewController.swift` and then option+click on the text we added (`UITextFieldDelegate`), then select the `UITextFieldDelegate` Protocol Reference.

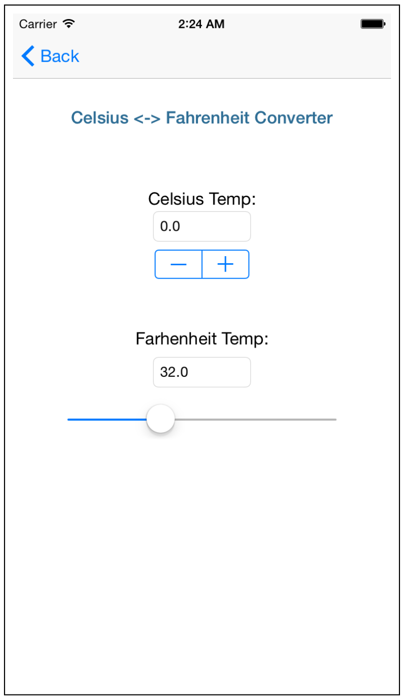
One other thing we need to do is to change the keyboard so the user has numbers instead of letters; to do this select the text field and in the `Attributes Inspector` change the `Keyboard` type from `Default` to `Numbers and Punctuation`. Add the code below, this code will respond to delegation by setting the relevant fields, and hiding the keyboard, and it will be called when the user presses `Return`.

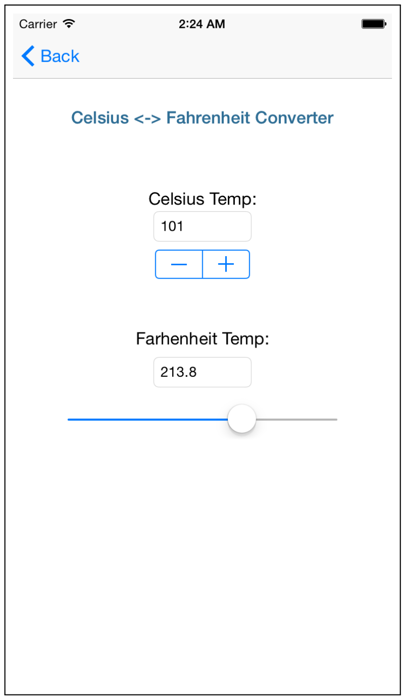
```
func textFieldShouldReturn(textField: UITextField) -> Bool {
        // first we need to know which textField we've got (since both
        // of them will talk to us in this method)
        if(textField == self.cTempText) {
                // get the temp from the text field and set the stepper
                let cTemp = NSString(string: textField.text).doubleValue
                self.cTempStepper.value = cTemp

                //use our converter
                let fTemp = getFahrenheit(cTemp)

                //set the slider and fahrenheit text
                self.fTempSlider.value = Float(fTemp)
                self.fTempText.text = String(format: "%.1f", fTemp)
        }
        else {
                // get the temp from the text field and set the slider
                let fTemp = NSString(string: textField.text).doubleValue
                self.fTempSlider.value = Float(fTemp)

                //use our converter
                let cTemp = getCelsius(fTemp)

                //set the stepper and celsius text
                self.cTempStepper.value = Double(cTemp)
                self.cTempText.text = String(format: "%.1f", cTemp)
        }

        textField.resignFirstResponder()
        return true
}
```

3.1.5 Fin!









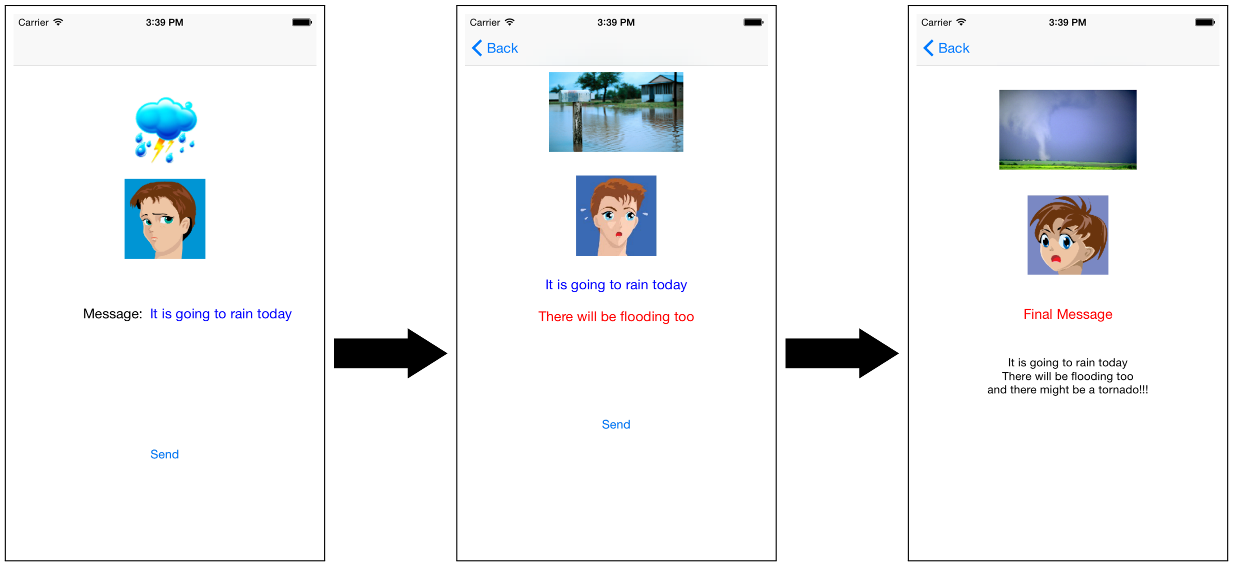
Congratulations, you finished the Conversions lab!

3.2 Segues and Weather

3.2.1 Goals of this Lab!

- Learn more about string building
- Learn about passing data between views

During this lab, we will develop a weather forecast message propagation scenario. Once the app is completed, it will have three views as shown below. The second view gets the message from the first view and adds its own version to the message and passes the message to the third view. The third view gets the message from the second view and appends its own version to it and displays. The final project should look like this:


Carrier
3:39 PM
Message: It is going to rain today
Send
Back
It is going to rain today
There will be flooding too
Send
Final Message
It is going to rain today
There will be flooding too
and there might be a tornado!!!

3.2.2 Phase I

In this phase we are going to create 2 view controllers where data will be passed from one to the other, and the second view controller will present the data that it was passed. The way this works is as follows:

- The sending view controller (`FirstViewController` here) will use a method called `prepareForSegue`. This method will be called automatically when a segue is in progress.
- In the `prepareForSegue` method, the sending view controller will set a property in the file of the receiving view controller.
- The receiving view controller (`SecondViewController` here) will then use its `viewDidLoad` method to take the data and present it to the user.

Once we start going through the steps it will make more sense, but try to keep the overarching goal in mind when writing the code.

Steps

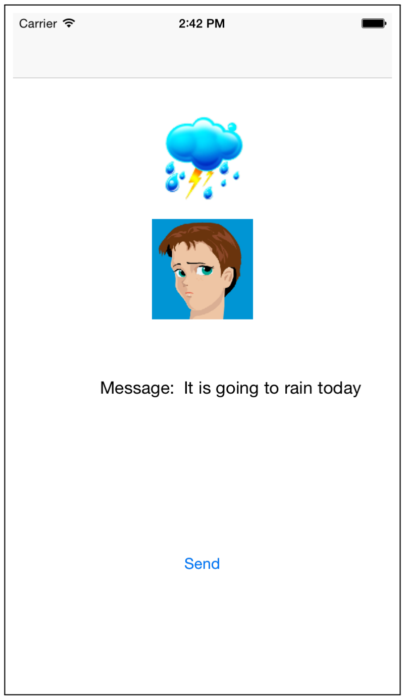
1. Create a new app called "Segues and Weather", with a first view controller named `FirstViewController` (make the `Organization Identifier` "CSE"), and embed the first view controller in a navigation controller.
2. Add the following objects to the *main view* controller as shown below.
   (a) Add two `Image Views`, a `Label`, `Text Field`, and a `Button`.

(b) Add the rain face and rain images as images as shown below.

(c) Change the `Label` text to be "It is going to rain today".

(d) Change the `title` of the `Button` to `Send`.

(e) Add an `IBOutlet` for the label in `FirstViewController.swift`.



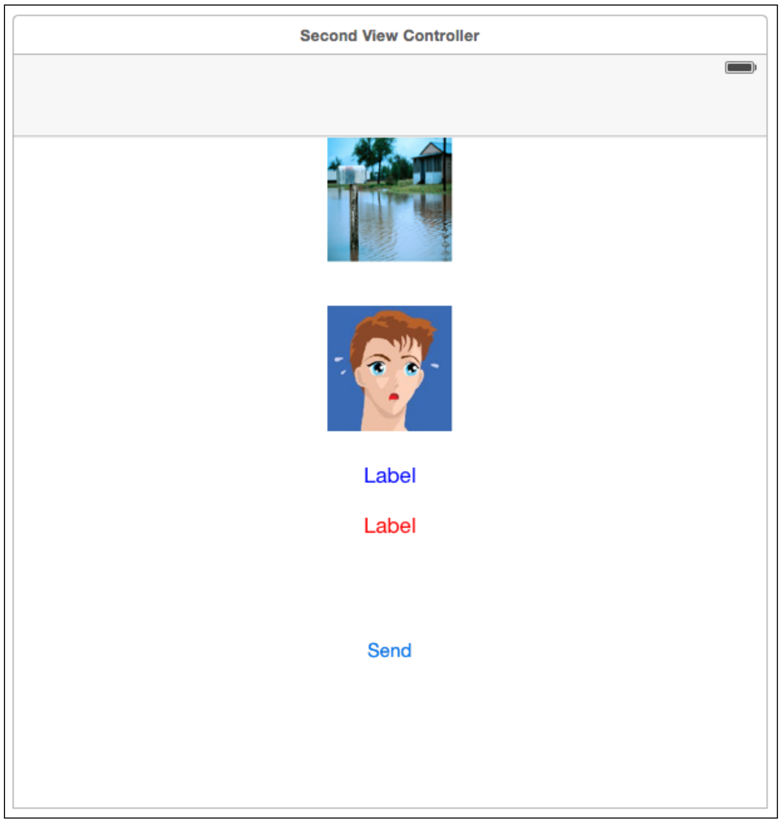
After adding outlets, your `FirstViewController.swift` file should look like the following:

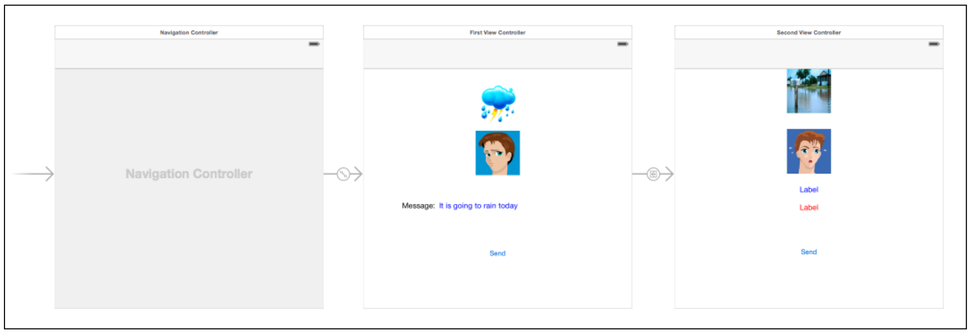
```
import UIKit

class FirstViewController: UIViewController {

    @IBOutlet weak var firstMessage: UILabel!

}
```

3. Add another scene with a view controller named `SecondViewController`. You'll need to drag a view controller out of the storyboard, create the backing code, and connect it (look at the resource guides on creating a new scene for more information). Configure the new scene with:

(a) Add two `Image Views`, two `Labels`, and a `Button`.

(b) Add the flood face and flood images as images as shown below.





(c) You likely skipped over reading the resource guide for setting up a new scene, you should go look at it.

(d) Now, create `IBOutlets` for the labels and the button, and the necessary properties to receive the message from the first view through segue.

After adding the property, your `SecondViewController.swift` should look like the listing below:

```
import UIKit

class SecondViewController: UIViewController {
    var message = String()
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

4. Now we need to create a segue from the first view to the second view. Select the `Send` button in the first view and ctrl→drag to the second view to create a segue. Select `Show` as the segue method.

5. Select the segue (the circle in between the arrow) and click the `Attributes Inspector` to change the identifier to `main`.

6. Add the following method to send data from the first view (main) to the second view. This method gets the message listed in the label of the main view and sends to the second view. Since the `prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)` method is referring to the second view, we need to include the second view controller file using the following command just before the end of the `FirstViewController.swift` file:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == "main" {
                let destination = segue.destinationViewController
                    as SecondViewController
                destination.message = self.firstMessage.text!
        }
}
```

After adding the prepare for segue method the `FirstViewController.swift` file should look like the code below:

```
import UIKit

class FirstViewController: UIViewController {

    @IBOutlet weak var firstMessage: UILabel!

    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == "main" {
            let destination = segue.destinationViewController as SecondViewController
            destination.message = self.firstMessage.text!
        }
    }
}
```

We are almost there... we need to add one last thing. After the second message is received we want to add "there will be flooding too" at the end of the original message and have it display its own message. As these messages need to be shown when we load the second view, we need to modify `viewDidLoad()` to add content to the second view as it is loaded. Edit the `SecondViewController.swift` file's `viewDidLoad()`:

```
override func viewDidLoad() {
        super.viewDidLoad()
        self.fromFirst.text = message
        self.secondMessage.text = "There will be flooding too"
}
```

7. Run the program and observe the output. It should show the first view with the message "It is going to rain today" and after the button is clicked the second view should include the text "there will be flooding too".

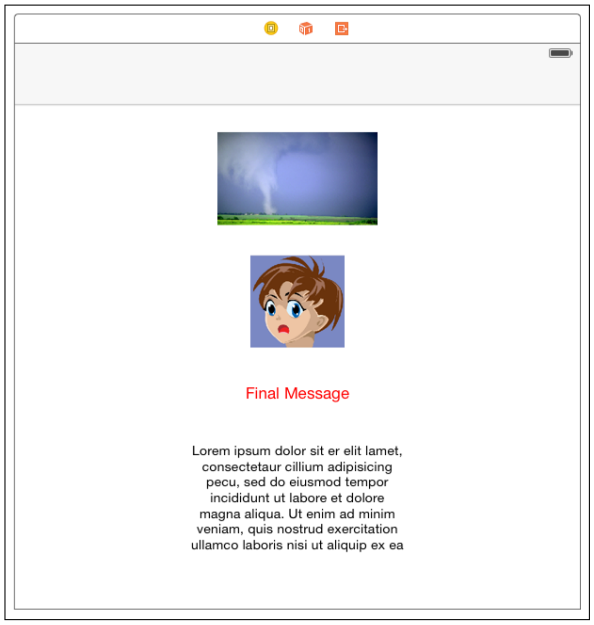
Congratulations! You are done with Part I of the Segues and Weather lab!

3.2.3 Phase II

During this phase we will add a third view that gets the message from the second view and adds its own message, "and the possibility of tornadoes!"

Adding the UI

1. Add another view controller named `ThirdViewController` (check the resource guide for adding a new scene!).
2. Add the following objects to the *third* view controller as shown below.
   (a) Add two `Image Views`, a `Label`, and a `Text View` (this allows for multiple lines of text).
   (b) Add the tornado face and tornado images as shown below:



Final Message

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea

   (c) Now it's time to add code to handle our UI items. First add a new view controller file for the third view. Name it as `CSEThirdViewController`.
   (d) Now link the third view to `CSEThirdViewController` (use the `Identity Inspector`!).
   (e) Now create an `IBOutlet` for the `Text View` and the `Label`. In the `ThirdViewController.swift` Also, add a string variable to create the final message.

```
var final = String()
```

After adding the Outlets, your `ThirdViewController.swift` file should look like the following:

```
import UIKit

class ThirdViewController: UIViewController {

    @IBOutlet weak var finalMessage: UITextView!

    var final = String()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

Passing the Data

To pass the data between the view controllers we will need to create a segue, pass the data in the prepare for segue, and retrieve the data in the third view controller.

1. Create a `Show` segue from the second view controller to the third view controller.

2. Select the segue (the circle in between the arrow) and go to the `Attributes Inspector` to rename the identifier to `second`.

3. In the `SecondViewController.swift` file create a `prepareForSegue` method like the one below:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "second" {
        let destination = segue.destinationViewController as ThirdViewController

        // create a message which is both strings combined
        // then pass it off to the ThirdViewController
        let nextMessage = self.fromFirst.text! + "\n" + self.secondMessage.text!
        destination.final = nextMessage
    }
}
```

4. As a final step, when the third view receives the message it should now add "and there might be a tornado!" to the end of the last message and then display the new message. As these messages need to be shown when we load the third view, we need to modify `viewDidLoad()` of the `ThirdViewController.swift` file to add content to the view.

```
override func viewDidLoad() {
        super.viewDidLoad()
        let finalString = "\nand there might be a tornado!!!"
        self.finalMessage.text = self.final + finalString
}
```

5. Run the program and observe the output, it should show the first view with the message "It is going to rain today" and then go to the second view as the `Send` button is clicked. When the `Send` button is then clicked on the second view, it should take us to the third and final view.

3.2.4 Polish and Swagger

If you have some extra time try to see if you can modify the application to add in a textfield so that you can pass your own messages between the view controllers. If you have lots of time, try to make a choose your own adventure book!

Congratulations!
Now you know how to send data from one view to another!

Chapter 4

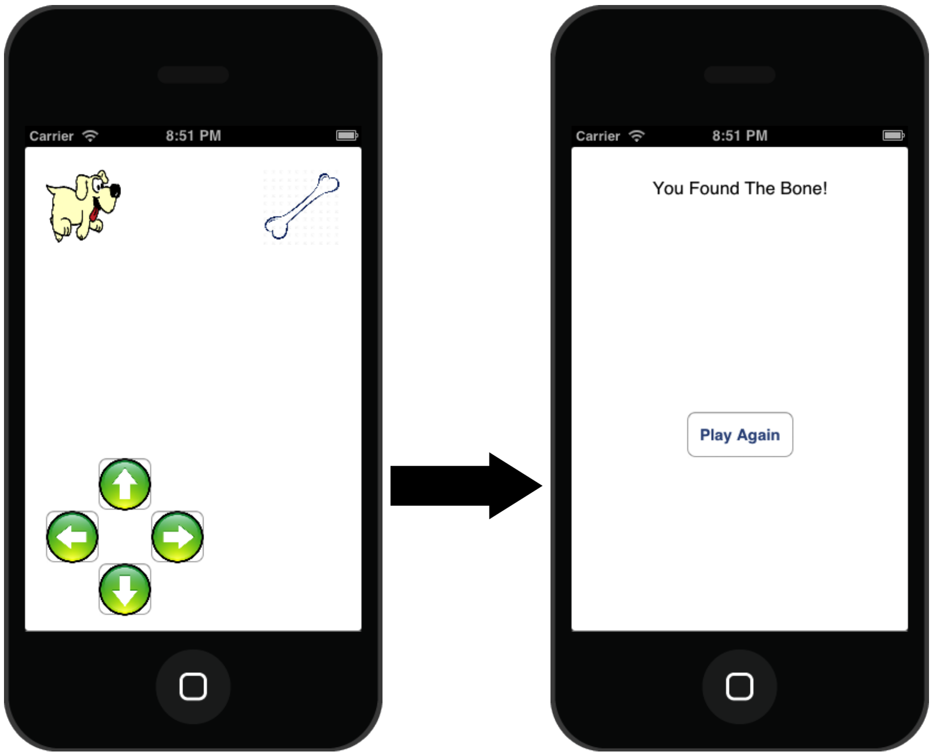
Views, Drawing, and Images

4.1 Animation, Drawing, and Controllers with Swift

4.1.1 Goals of this Lab!

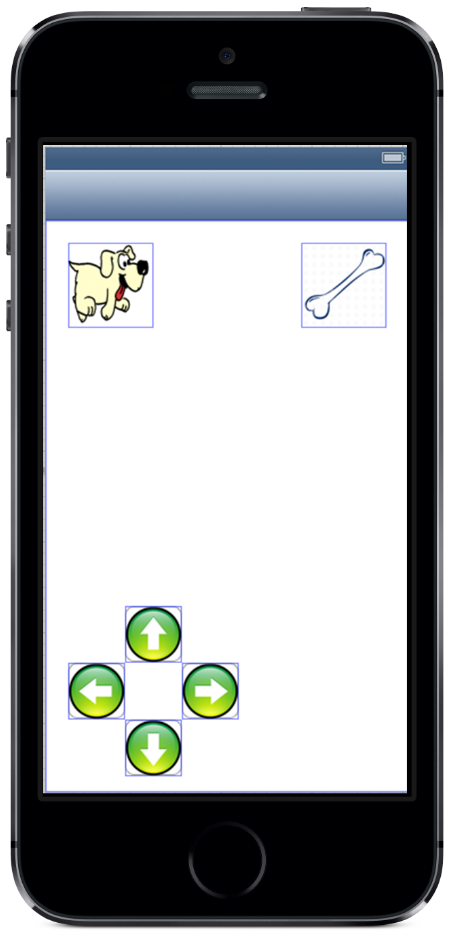
- Learn about views,
- Learn about moving views,
- Reiterate how to pass data between views,
- Learn to use the `NSTimer` class,
- Extra subsection (which probably everyone will reach) which introduces arrays and scaling a view.

In this example we will be creating a simple game about a dog who must search for his bone. The control scheme will be game-boy style (screen up top, controls down below), however we can help you change the movement to swipe gestures after the next lab. The final project should look like this:



4.1.2 Phase I

Phase I will focus on the setup of the basic UI and movement, essentially placing all the images and allowing them to move around. The end of phase I should look like like the image below:



Steps

1. Create a new app called `Lab6` but this time make sure you enable git repositories (after the first set of options, when you are choosing the directory for you app).

2. Import the supporting files (they are located on the camp site).

3. Add the following objects to the storyboard, on the *main view* controller:

   (a) Add one `View`, NOT a View Controller, and resize it to take the top half of the screen.

   (b) Add two `Image Views` *in* the new view. Set the `Image` property (from within the `Attributes Inspector`) to the name of the dog image on one image and to that of the bone on the other.

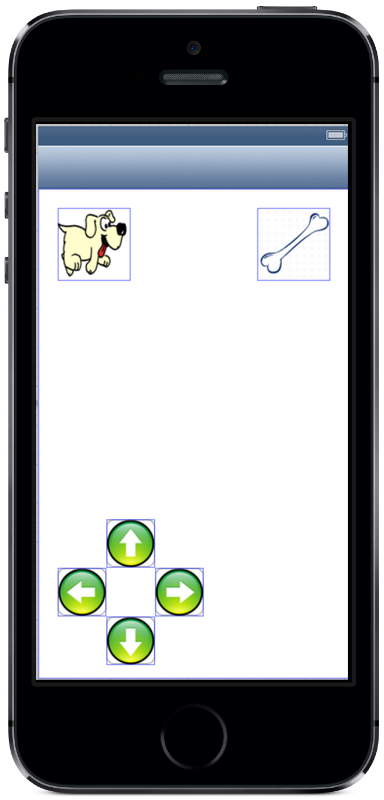
   (c) Add four `Buttons` at the bottom of the screen, NOT in the new view. Set the `Image` property of the buttons to up, left, down, and right (the remaining images from the supporting files).

   (d) Select the `button_left` button in the storyboard and in the `Size Inspector` (just right of the attributes Inspector) and set the following values:

   - `X` = 20
   - `Y` = 390
   - `Width` = 50
   - `Height` = 50

(e) Set the remaining buttons with the same height and width but with X and Y values that place them in a more "square" orientation (you can move them by hand or by setting the X and Y values, if you change the X and Y values directly then changes of about the button size should but good).

After adding that your storyboard should look like below:



Now we need to connect our view objects with the backing code, so lets open up the assistant editor. Connect the images to the backing code using the following property names:

- `ImageView` with the dog: `dogImage`
- `ImageView` with the bone: `boneImage`

Next connect the button objects to the backing code with the following `IBActions`, essentially compass directions:

- Up button: `moveNorth`
- Right button: `moveEast`
- Down button: `moveSouth`
- Left button: `moveWest`

After adding Outlets your CSEViewController interface should look like the following:

```
class GameViewController: UIViewController {

    // This code should have been added by the storyboard automatically
```

```
        @IBOutlet var dogImage:UIImageView!;
        @IBOutlet var boneImage:UIImageView!;
        @IBAction func moveNorth(sender: AnyObject) {
        }

        @IBAction func moveWest(sender: AnyObject) {
        }

        @IBAction func moveSouth(sender: AnyObject) {
        }

        @IBAction func moveEast(sender: AnyObject) {
        }

    }
```

4. Now that we have our UI and interface defined, let's set up our backing code so we can move around. Lets modify the `IBAction` methods to move our images around:

```
    @IBAction func moveNorth(sender: AnyObject) {
        var position = self.dogImage.frame
        position.origin.y = position.origin.y - 10
        self.dogImage.frame = position
    }

    @IBAction func moveWest(sender: AnyObject) {
        var position = self.dogImage.frame
        position.origin.x = position.origin.x - 10
        self.dogImage.frame = position
    }

    @IBAction func moveSouth(sender: AnyObject) {
        var position = self.dogImage.frame
        position.origin.y = position.origin.y + 10
        self.dogImage.frame = position
    }

    @IBAction func moveEast(sender: AnyObject) {
        var position = self.dogImage.frame
        position.origin.x = position.origin.x + 10
        self.dogImage.frame = position
    }
```

Finished Phase I!

Great job on the first phase, your application should let the dog to move around now. Let's store this code for later by committing it to our git repository. To do this select *Source Control* → *Commit*. When prompted for a commit line, enter something like "phase 1 commit, the images moooove!!!"; you can add your own data as well, and maybe put an explanation in your own words of what has been done so far. Making a git commit ensures that we have a snapshot of our project as it exists at this point, so we can always look back on the project.

4.1.3 Phase II

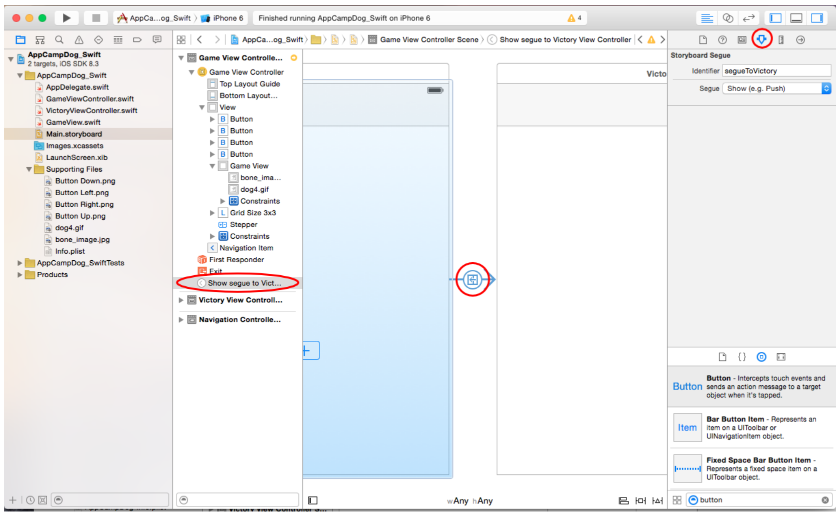
The next improvement that we are going to make is to add a success screen and some slick animations.

Feature 1 - create a new view to display a victory message

Create a new View Controller named `VictoryViewController` which we will use to show a message. If you've forgotten how to do this take a look at the resource guides.

Next we will define a segue between the two views; to do this, ctrl+click and drag from one of the view controllers to the other (to get the view controller drag form the top bar), like the image below.

Next create a segue between the two view controllers. To do this you'll need to ctrl-drag from the `View Controller` icon (yellow box) on the black bar below the main layout. In the popover select `Show` as the segue type. Name the segue by going selecting the segue, opening the `Attributes Inspector` and changing the `Identifier` to `segueToVictory`.


Storyboard Segue
Identifier segueToVictory
Segue Show (e.g. Push)

Now we have directions, and a place to go, we just need something to take us, and a time when we want to go! Let's define a navigation controller to take us to the next view. On the storyboard select our original view controller and embed it in a navigation controller. As a refresher:

1. In your storyboard select your first view controller (the one with an arrow into it from nowhere).

2. in the top menu select Editor → Embed In → Navigation Controller.

Now we will be able to move, but that means the old navigation controller is stealing some of our screen real estate! Normally this is a really convenient feature, so users can navigate "back" but since we want to have the whole screen to ourselves (and we provide all the necessary navigation features) let's hide that navigation controller. Add this `viewWillAppear` method, it will be called automatically as part of the view lifecycle when your application loads. We also need to activate the segue when we are ready to go; just for demonstration, let's do both together.

```swift
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    self.navigationController?.navigationBarHidden=true

    // immediately transfer to the second view
    self.performSegueWithIdentifier("segueToVictory", sender: self)
}
```

Awesome, now we can go forward, but our second view is lame, and we get stuck there! Return to the storyboard and add a label and button to our `VictoryViewController`, set the label text to something like "congrats!", and set the button text to `Play Again`. Connect the `playAgain` button to the backing code using an `IBAction`, and make the method name `playAgain`. Now in `VictoryViewController` add the following code:

```
@IBAction func playAgain(sender: AnyObject) {
    self.navigationController?.popViewControllerAnimated(true)
}
```

You may want to make a git commit here if everything is working, definitely test that everything is working though!

Feature 2 - Move the images in an animated fashion and detect the end of game

The next thing we are going to do is to move the images using animations; animations will give the application a smoother and more professional feel.

First we need to remove the `performSegue` line from our `DogProgramViewController` in the `viewWillAppear` method. That way we can access the first view again. Next we will need to update our movement by using 2 methods, one of which we will create:

1. `animateWithDuration`: allows us to send a number of changes to a view and the iOS system will automatically animate them for you (which is aaawwwwwwwesome).

2. `viewIntersectsView`: simplifies collision detection between 2 views, it returns `true` or `false` so we can use it in the `if()` construct that we've used before.

Let's define `IntersectsWithAnotherView` in our view controller now:

```
func viewIntersectsView(view:UIView, otherView: UIView) -> Bool {
    return CGRectIntersectsRect(view.frame, otherView.frame)
}
```

You'll need to update *each* of your movement methods with the following code (before and after are provided for simplicity):

before:

```
@IBAction func moveNorth(sender: AnyObject) {
    var position = self.dogImage.frame
    position.origin.y = position.origin.y - 10
    self.dogImage.frame = position
}
```

after:

```
@IBAction func moveNorth(sender: AnyObject) {
    var position = self.dogImage.frame
    position.origin.y = position.origin.y - 10
    UIView.animateWithDuration( 0.3, animations: { () -> Void in
        self.dogImage.frame = position
    })

    if(viewIntersectsView(dogImage,boneImage)){
        self.performSegueWithIdentifier("segueToVictory", sender: self)
    }
}
```

Finished Phase II

Great job on the second phase! Test it to make sure that it works; your application should let the dog move around in an animated fashion, and our application should switch views once the dog finds the bone. Let's store this code for later by committing it to our git repository. To do this select Source Control → Commit. When prompted for a commit line enter something like "phase 2 commit, the game is afoot!", you should add your own comment as well, maybe put an explanation in your own words of what has been done so far.

If you're feeling adventurous play around with the distances moved and see what works best for you. Also try to modify the application so that the dog cannot leave the frame, to check for that you can get the frame size with `self.view.frame` (you'll need the size of the dog, and her origin point to find out if you're going off screen).

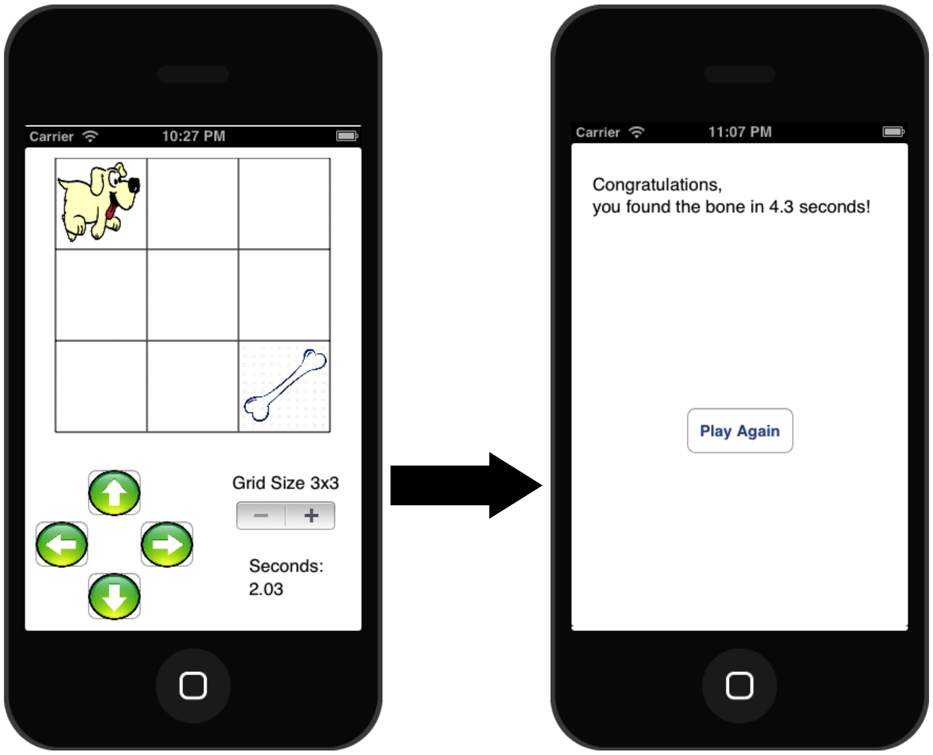
Congratulations!
Now you know how to work with views and draw!

4.2 Animation, Drawing, and Controllers with Swift - Part 2

4.2.1 Goals of this Lab!

- Reiterate how to pass data between views
- Learn to use the `NSTimer` class
- Introduce arrays and scaling a view programmatically

In the first half of this example created a simple game about a dog who must search for his bone. In this subsection we will be extending it to see how we can handle different view sizes and different game sizes. The final project should look like this:



4.2.2 Phase III

In this phase we will add a timer to allow us to find out how long a user takes to reach the bone, and this time will be shown in the next scene.

UI Updates

First on your main view controller add 2 labels. One should say `Seconds:` and the other will be set with the number of seconds. Connect the label with the number of seconds to the backing code with an outlet, name it `secondsElapsed`. Then a variable for the `NSTimer`, the following 2 lines should now be in your view controller:

```
var timer: NSTimer?
@IBOutlet weak var secondsElapsed: UILabel!
```

Adding a Timer

Next in the main view controller add the following code. Your `viewWillAppear` method should already exist with the navigation bar hiding code, but you'll need to add the Timer scheduler and the Timer response function.

```
override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        self.navigationController?.navigationBarHidden=true

        self.secondsElapsed.text = "0.0"
        self.timer = NSTimer.scheduledTimerWithTimeInterval(0.01, target: self,
            selector: "timerFired:", userInfo: nil, repeats: true)
}

//we only have one timer, so we don't care who called always increment the time
func timerFired() {
    self.secondsElapsed.text = String(format: "%.2f", Float(secondsElapsed.text!)!+0.01)
}
```

Next we are going to intercept the segue so that we can pass in the number of seconds to be displayed. We will need to define a new property in `VictoryViewController` which can hold the value that we are going to set, and a property to access the label. If you haven't connected the label to the backing code do that now (don't worry, it wasn't in the instructions). In addition to that connection add the code below:

```
var seconds: Double = 0.0
```

Next in our `VictoryViewController` file we're going to add modify the `viewDidLoad` method to set our label. We are going to expand the message we send to something like "Congratulations, you found the bone in 2 seconds!" where 2 is the seconds property that is passed over. To do this add the following code (or modify if the method exists):

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.victoryLabel.text = String(
        format: "Congrats,\nyou found the bone in %.1f seconds", self.seconds)
}
```

Passing the Data!

Finally once that is set up let's return to our main view controller and write our `prepareForSegue` method. This method will get called automatically when we switch between views and we can use it either to pause our view, or pass data to the next view. In this case we will do both!

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    // Invalidate the timer since we don't need it anymore
    self.timer?.invalidate()

    if segue.identifier == "segueToVictory" {
        let dest = segue.destinationViewController as! VictoryViewController
        dest.seconds = Double(self.secondsElapsed.text!)!
    }
}
```

End Phase III

Test it out now, you might need to do some resizing, since the labels might not be long enough. If you have a problem with it not using multiple lines just select the label, and in the `Attributes Inspector` change the `Lines` property to 2, or however many lines you need. Notice how when we select `Play Again` our timer is reset; in the next phase we are going to reset everything.

If you are using git now would be a good time to commit, just a reminder we've added the Timer, `prepareForSegue`, and we now print the time that it took to find the bone.

4.2.3 Phase IV

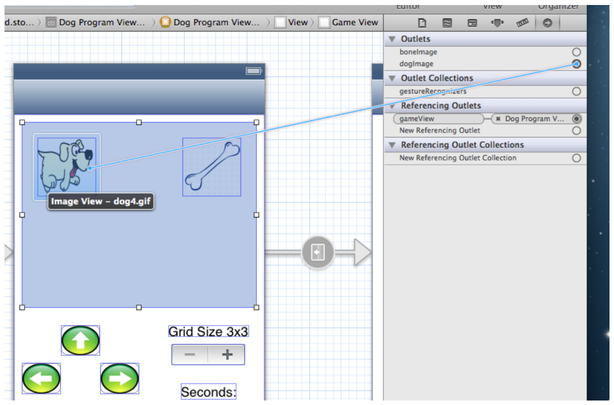
This will be the final phase and it will be a doozy! In this phase we are going to handle the actual drawing and we will be aligning our image movements with the drawn grid. We will also be adding a new view and connecting it with that view we made so long ago!

Up until now we have only created custom view controllers, but actually you can write custom code to manage any object. In this part we are going to be writing a custom view, which is capable of handling drawing and updating when drawing needs to occur.

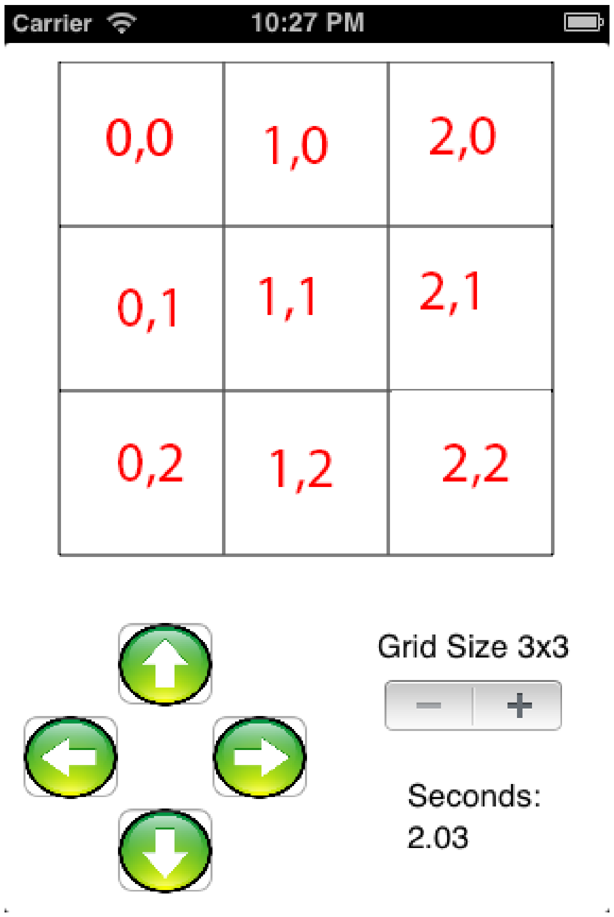
The first thing we will do is to create a new view; to do this create a new file that is a subclass of `UIView`, NOT `UIViewController` (this time for realsies, no typo). Name the new view `GameView`. In the source code for this new view add the following code. For subviews, the storyboard does not allow the easy connections so we will have to write the properties ourselves.

```swift
class GameView: UIView {

    // Allow us to access the image views
    @IBOutlet weak var boneImage: UIImageView?
    @IBOutlet weak var dogImage: UIImageView?

    // Current state of the game
    var dogLocation: CGPoint?
    var boneLocation: CGPoint?

    // Number of squares per row
    var numberOfCells: Int?

    // Maintain our grid shape
    var x_gap: Int?
    var y_gap: Int?
    var cellPointWidth: Int?

    func setInitialCoordinates() {

    }
}
```

Now let's return to the storyboard so that we can ensure that our views are connected properly. First select the view that holds the Bone and Dog images. After that, go to the `Identity Inspector` and set the `Class` to `GameView`. Now still with the view selected go to the `Connections Editor` (right-most icon on the line with `Identity Inspector`). From there you should have 2 outlets defined; left-click and drag from the circle on the right of the `boneImage` to the image of the bone.



The next major step in our `GameView` will be adding code so that we can re-define our space in terms of cells instead of points.



The first thing we want to do is to implement an `init` method with a `coder` parameter in `GameView`. This method is called automatically when our view is created by the storyboard. What we want to do here is to set the number of cells that will be in our grid. By using a property the number of cells can be flexible so that we can change that number later (hint hint!):

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    self.numberOfCells = 3
}
```

The next thing is to define a special view method called `drawRect:`; this method is called automatically when the view should update what it is displaying on the screen. We will override this method and use it to draw what we need to draw (the game squares, the dog, and the bone). Look in your `GameView` file; if there is no `drawRect` then add one; otherwise just copy the contents into the existing method.

```
override func drawRect(rect: CGRect) {
    super.drawRect(rect)
    self.drawGrid(rect)
}
```

The next step will of course be to define the `drawGrid` method (you should have an error in your code mentioning that Xcode isn't sure what `drawGrid` is).

Define `drawGrid` as follows, yes all of the following is 1 method (sorry for such a long method, this will be a test of your copying abilities!):

```
func drawGrid(rect: CGRect) {

    // How big is our view in total?
    let width = Int(rect.size.width)
    let height = Int(rect.size.height)

    // since we are making a square grid of whole squares we need to
    // take the smaller of the 2 dimensions and the larger will have a gap
    let gridWidth = min(width, height)

    // based on the dimensions how big will the gap be?
    // this will help us center our grid
    self.x_gap = width - gridWidth;
    self.y_gap = height - gridWidth;

    // what is the size of the width/height of each cell?
    self.cellPointWidth = gridWidth/self.numberOfCells!;

    // Because we might have an uneven division there may be a few points
    // that aren't used, this will create spaces after our grid
    // to solve this problem we have to subtract out the rounding errors
    let emptyPoints = gridWidth - cellPointWidth!*self.numberOfCells!;

    // lines are drawn from one point to another
    // the x_endpoint should be the right side of the
    let x_endpoint = gridWidth+x_gap!/2 - emptyPoints;
    let y_endpoint = gridWidth+y_gap!/2 - emptyPoints;

    // Set the color in the current graphics context for future draw operations
    UIColor.blackColor().setStroke()

    // Create our drawing path
    let drawingPath = UIBezierPath()

    // Define a grid
    // first the vertical lines
    for i in (x_gap!/2).stride(to: width, by: cellPointWidth!) {
        // essentially lift your pen from the paper, then set it here
        drawingPath.moveToPoint(CGPointMake(CGFloat(i), CGFloat(y_gap!/2)));
        // now drag the pen from where it was to this new point
        drawingPath.addLineToPoint(CGPointMake(CGFloat(i), CGFloat(y_endpoint)));
    }

    for i in (y_gap!/2).stride(to: height, by: cellPointWidth!) {
        // essentially lift your pen from the paper, then set it here
        drawingPath.moveToPoint(CGPointMake(CGFloat(x_gap!/2), CGFloat(i)));
        // now drag the pen from where it was to this new point
        drawingPath.addLineToPoint(CGPointMake(CGFloat(x_endpoint), CGFloat(i)))
    }

    // Now actually draw the grid
    drawingPath.stroke()
}
```

After you get all of that in test it to make sure that it works, you should see a cool grid. If you don't see a grid then flag one of us down and we can help you out. If you get nothing then the view might not be attached properly. If parts of the line aren't drawn, then you might just be missing a command.

Setting and moving our images

At this point we have our `GameView` which draws our game grid, and is able to access the dog and bone images but doesn't modify them. We also have our controller moving the images directly. What we will be doing is instead of using the controller to move the images, we will tell the view where we want the pieces logically, and we'll let the view figure out how to draw them.

We are going to begin our control methodology by defining custom setters. A setter is a special method that is used to set a property value; in our case we are also going to use it to redraw the screen. This will make sure that any time someone gives us a new position for either of the images we get them redrawn in the correct location. The next thing we will do is to use our view to manipulate the dog and bone images. Modify the variables of your `GameView` file:

```
var dogLocation: CGPoint? {
    didSet {
        self.setNeedsDisplay()
    }
}
var boneLocation: CGPoint? {
    didSet {
        self.setNeedsDisplay()
    }
}
```

Next we are going to move the dog and bone position updating to be in this view. Essentially our view controller will tell the view where the images should be, and the view will place them there, alternatively we can tell the view to move right, and the view can do it. Let's define an `updateImages` method which will set our frame positions:

```
func updateImages() {
    // Get a copy of the image frames
    var dogFrame = self.dogImage!.frame
    var boneFrame = self.boneImage!.frame

    // set the sizes
    let value = CGFloat(self.cellPointWidth!-4)
    dogFrame.size.width = value
    dogFrame.size.height = value
    boneFrame.size.width = value
    boneFrame.size.height = value

    // set the "origin" or "top-left corner" to be 2 points from
    // the grid edge, because our image is 4 points less than the
    // full grid size this should leave a 2 point empty frame
    // around our image
    dogFrame.origin.x = 2 + CGFloat(self.x_gap!/2) +
        self.dogLocation!.x*CGFloat(self.cellPointWidth!)
    dogFrame.origin.y = 2 + CGFloat(self.y_gap!/2) +
        self.dogLocation!.y*CGFloat(self.cellPointWidth!)
    boneFrame.origin.x = 2 + CGFloat(self.x_gap!/2) +
        self.boneLocation!.x*CGFloat(self.cellPointWidth!)
    boneFrame.origin.y = 2 + CGFloat(self.y_gap!/2) +
        self.boneLocation!.y*CGFloat(self.cellPointWidth!)

    // update the frames with our modified versions
    self.dogImage!.frame = dogFrame
    self.boneImage!.frame = boneFrame
}
```

Also update the *drawRect* method to cause an animated image update using the new method:

```
override func drawRect(rect: CGRect) {
    super.drawRect(rect)
    self.drawGrid(rect)

    UIView.animateWithDuration(0.75, animations: {() -> Void in
        self.updateImages()
    })
}
```

Now if you run it something interesting happens. The bone and the dog get drawn onto the same square! The reason for that is that all of our values get initialized to 0; and the top-left happens to be (0,0). We're going to need some way to set our initial positions by adding the following method:

```
func setInitialCoordinates() {
    self.dogLocation = CGPointMake(0, 0)
    self.boneLocation = CGPointMake(2, 2)
    self.updateImages()
}
```

Now update the *viewWillAppear* method in your *DogProgramViewController* to call our new method.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    self.navigationController?.navigationBarHidden = true
    self.secondsElapsed.text = "0.0"
    self.timer = NSTimer.scheduledTimerWithTimeInterval(0.01, target: self,
        selector: #selector(ViewController.timerFired), userInfo: nil, repeats: true)

    // initialize the game
    self.gameView.setInitialCoordinates()
}
```

Awesome, now our images get put into the correct starting locations! The last step is going to be switching our movement from the main view controller into *GameView*; luckily this is fairly easy since we've already handled redrawing in any space.

Take a breather here!

There has been a whole lot of code added up until now, take a few minutes and look it over, read the comments provided (if you read them before then awesome job, you should read them again anyways!). Try to make sure you understand basically what is going on, if you have any questions feel free to ask one of us. Don't worry if you don't understand this, it is complicated stuff!

Ok, time to finish it!

All we need to do now is instead of setting a new frame for the image we just give the `GameView` a new coordinate for the image, and the `GameView` will handle setting the frame. We will also do some basic error checking to ensure that our images don't get drawn off screen. Update your direction button responders to something like the following (also delete previous code):

```
@IBAction func moveNorth(sender: AnyObject) {
    let dogLocation = self.gameView.dogLocation
    if dogLocation!.y - 1 < 0 {
        print("Error: trying to go out of bounds north")
        return
    }
    else {
        self.gameView.dogLocation = CGPointMake(self.gameView.dogLocation!.x,
            self.gameView.dogLocation!.y)
    }
    if CGPointEqualToPoint(self.gameView.dogLocation!, self.gameView.boneLocation!) {
        self.performSegueWithIdentifier("segueToVictory", sender: self)
    }
}
```

4.2.4 Finished Phase IV!

Awesome job. This will be one of the tougher labs but hopefully it was enjoyable and you were able to get something out of it!

Congratulations!
Now you know how to transform pixel values into structured drawings using arrays

Chapter 5

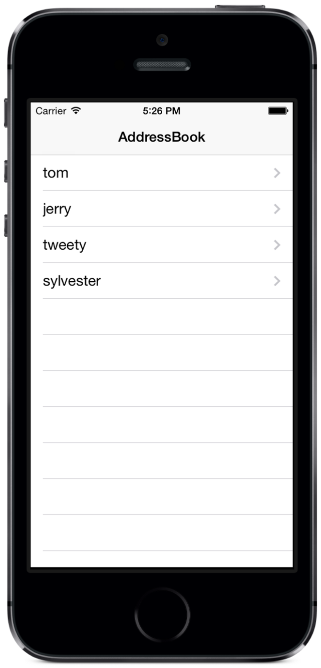
Table Views, SpriteKit, and MapKit

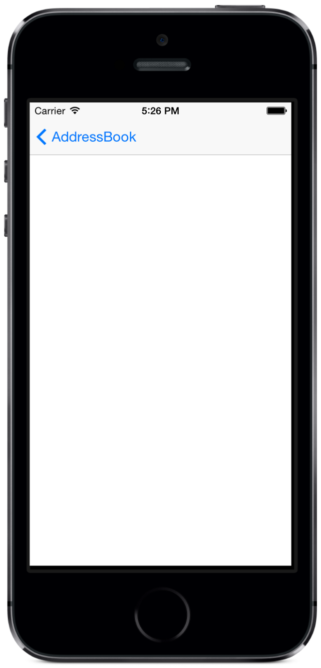
5.1 Address Book - Phase I

5.1.1 Goals of this Lab!

- Understand how to work with table views

The result of phase I should look like the images below:





5.1.2 Step 1

Create a storyboard based single view application. Delete the original view controller in the storyboard, and then add a `Table View Controller` to the scene. Then embed it in a navigation controller.

5.1.3 Step 2

First in your view controller file, change the superclass to be `UITableViewController` instead of `UIViewController`. Then change the class of the table view controller in the storyboard to be the name of the `.swift` file you created when you made the project. Let's put the data into to the table view. We need an `Array` to hold the names and an outlet to the table view. Let's add these:

```
var names: Array<String> = []
@IBOutlet var addressBook: UITableView!
```

5.1.4 Step 3

Now we are ready to add names to the names array:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.names = ["tom", "jerry", "tweety", "sylvester"]
}
```

5.1.5 Step 4

Before displaying table data let's customize the table cell. Name the cell as `addressCell` in the `Attributes Inspector`. Change the table cell `Accessory` to `Disclosure Indicator`. That will tell the app user that this cell should be selectable.

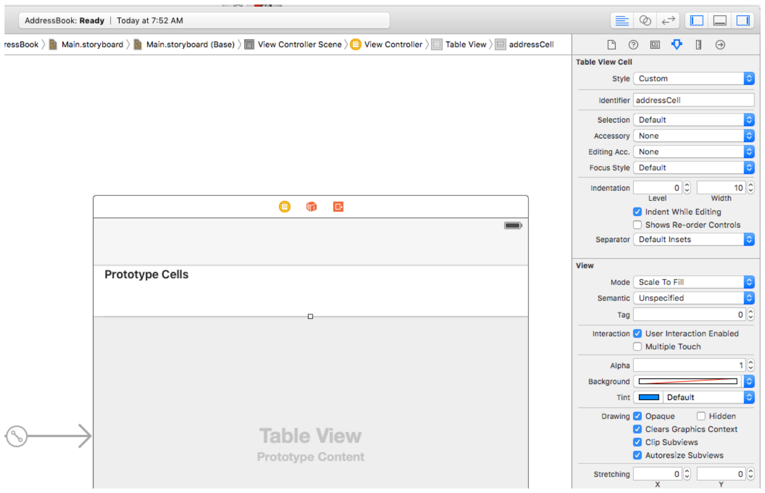

AddressBook: Ready | Today at 7:52 AM
ressBook > Main.storyboard > Main.storyboard (Base) > View Controller Scene > View Controller > Table View > addressCell
Prototype Cells
Table View
Prototype Content

Table View Cell
Style: Custom
Identifier: addressCell
Selection: Default
Accessory: None
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View
Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled ☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque ☐ Hidden ☑ Clears Graphics Context ☑ Clip Subviews ☑ Autoresize Subviews
Stretching: 0 (X) 0 (Y)

5.1.6 Step 5

The table view will ask the program code for the cells that it should be presenting. To allow the tableview to function we will need to add the two methods below to our view controller:

```swift
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return self.names.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath
    indexPath: NSIndexPath) -> UITableViewCell {
    var cell = self.addressBook.dequeueReusableCellWithIdentifier("addressCell")
    if (cell == nil) {
        cell = UITableViewCell(style: UITableViewCellStyle.Default,
            reuseIdentifier: "addressCell")
    }
    cell?.textLabel!.text = self.names[indexPath.row]
    return cell!
}
```

5.1.7 Step 7

Our objective is to display address book details for each name. When a user selects a name we will navigate into the new detail view and show more information about that person. The first step toward this goal is another view controller with a segue defined to it. Add another view controller and create a segue from the table cell to the new view controller.

5.1.8 Step 8

Now run the app and see how it works. You should see the UI screen as shown earlier. Congratulations, you have completed Phase I of creating an AddressBook!

5.2 Address Book - Phase II

5.2.1 Step 1

Create some new arrays: `photos`, `addresses`, and `emails`, that will hold relevant information about each person. Now let's populate our new arrays with some dummy data so we can see what it looks like. This should be done In the `CSEViewController.swift`:

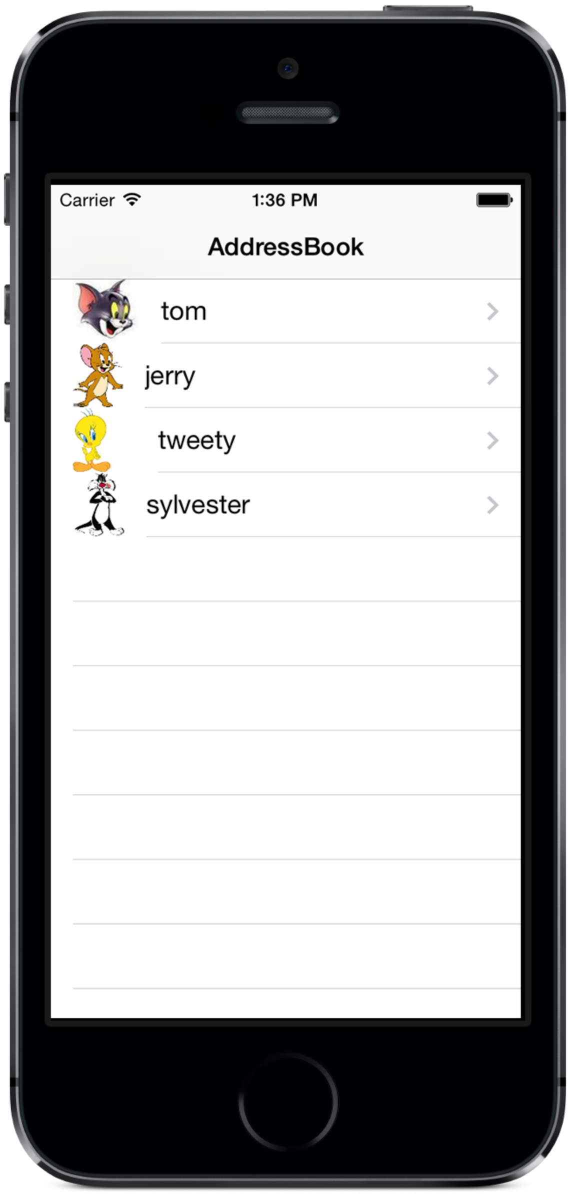
```
override func viewDidLoad() {
    super.viewDidLoad()
    self.names = ["tom", "jerry",
                  "tweety", "sylvester"]
    self.photos = ["tom.jpg", "jerry.jpg",
                   "tweety.jpg", "sylvester.jpg"]
    self.emails = ["tom.the.cat@acme.com",
                   "jerry.the.mouse@acme.com",
                   "tweety.the.bird@acme.com",
                   "sylvester.the.cat@acme.com"]
    self.addresses = ["699 S Mill Ave Tempe AZ 85281",
                      "7001 E Williams Field Rd, Mesa, AZ 85212",
                      "411 N Central Ave, Phoenix, AZ 85004",
                      "1475 N Scottsdale Rd, Scottsdale, AZ 85257"]
}
```

5.2.2 Step 2

Next let's modify the `cellForRowAtIndexPath` method so that we will show an image, in addition to the name that we are already showing. Make sure to add the photos from the supporting files for this lab:

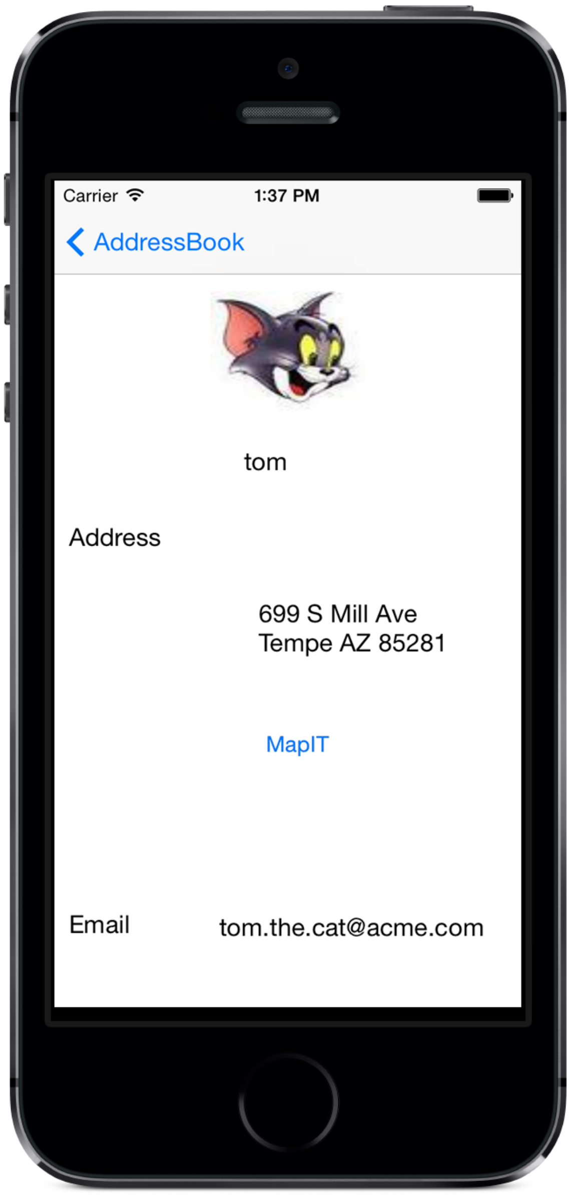
```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
    indexPath: NSIndexPath) -> UITableViewCell {
    var cell = self.addressBook.dequeueReusableCellWithIdentifier("addressCell")
    if (cell == nil) {
        cell = UITableViewCell(style: UITableViewCellStyle.Default,
            reuseIdentifier: "addressCell")
    }
    cell?.textLabel!.text = self.names[indexPath.row]

    // Add this line below!
    cell?.imageView!.image = UIImage(named: self.photos[indexPath.row])
    return cell!
}
```

This is what we should see if we run the app now:



5.2.3 Step 3

Now we are ready to implement the detail view. Add a file that is a subclass of type `UIViewController` and name it `CSEDetailViewController` as shown below, then link it to the backing code. If you're not sure how to do this you should check the resource guides. We will need some outlets linked from the UI to the program code: an image view and 3 labels (name, address, and email). You should also add 2 more labels to specify where the address and e-mail will appear, but feel free to design the UI however you want. This is a possible layout of the outlets (except for `MapIt`, which we will add in the next phase):



5.2.4 Step 4

We are going to need receiving properties set up in `CSEDetailViewController`:

```
var selectedAddress: String = ""
var selectedEmail: String = ""
var selectedName: String = ""
var selectedImage: String = ""
```

5.2.5 Step 5

Now, in the first view controller add the `prepareForSegue` method to the `CSEViewController` view controller to pass data to the detail view:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "detailView" {
        let indexPath = self.addressBook.indexPathForSelectedRow
        let dest = segue.destinationViewController as! CSEDetailViewController
        let index = indexPath!.row
        dest.selectedName = self.names[index]
        dest.selectedEmail = self.emails[index]
        dest.selectedAddress = self.addresses[index]
        dest.selectedImage = self.photos[index]
    }
}
```

Question for you: how does the `prepareForSegue` know which row was selected in the table view? Take a minute and look through the code above to figure that out.

5.2.6 Step 6

Now our data will be passed from the main view controller to `CSEDetailViewController`. In this file, let's update the `viewDidLoad` method to present that data to the user:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.name.text = self.selectedName
    self.address.text = self.selectedAddress
    self.email.text = self.selectedEmail
    self.photo.image = UIImage(named: self.selectedImage)
}
```

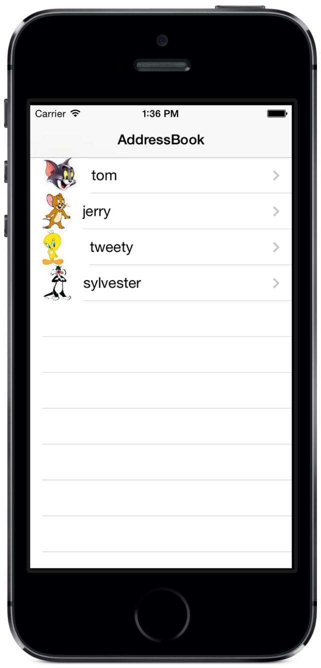
5.2.7 Step 7

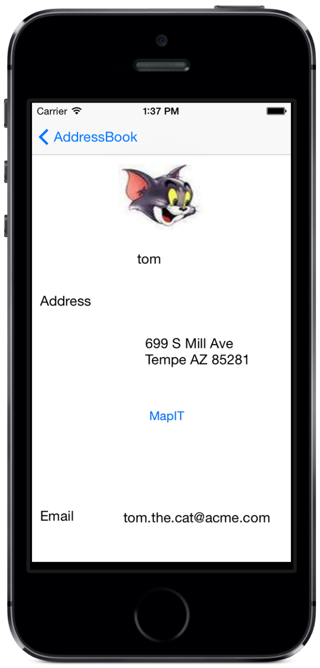
Give the app a run and you should see the information from the selected index appearing on the second screen. Congratulations! You have completed Phase II of AddressBook!

5.3 Address Book - Phase III

5.3.1 Goals of this Lab!

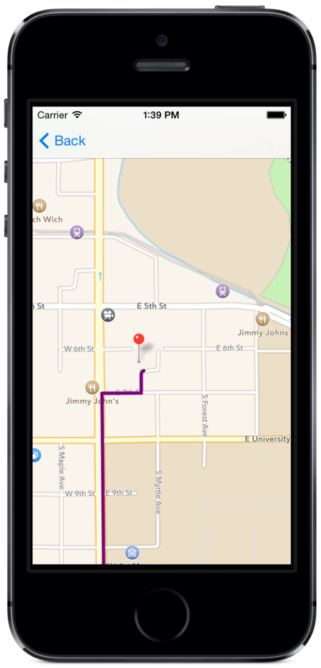
During this lab we will learn about how to integrate maps into our projects. This will including displaying addresses, and even finding and drawing a driving route between locations! The final result of phase III should look like the images below:





Address 699 S Mill Ave
Tempe AZ 85281

Email tom.the.cat@acme.com



5.3.2 Updating the UI

New View Controller

In our storyboard drag out a new view controller. This will be where we have our map, and our routing. Next we'll need a way to get to this new view controller; add a `Button` on `CSEDetailViewController` and change the text to be `MapIt`. The idea is that this new button will allow us to transition to the map view, and show the `address` label visually. We will also do some route mapping to see how we can draw routes onto the map.

Next we will need to create a new view controller to back this scene. Create a new `UIViewController` subclass named `CSEMapViewController` and link with the new view as shown below. Make sure to add the `MapKit` and `CoreLocation` frameworks by `import`ing them in the `CSEMapViewController` class.

Because I'm Mappy

On this new view controller drag out a new `Map View` (from the objects). We'll need to connect this `Map View` to the backing code with a `property` (let's call it `mapView`).

Segue to the Map

Now we need a way to get to our new View Controller; define a segue from the `MapIt` button to the `CSEMapViewController`, and name the segue `map`. Since we are going to want to transfer data between the view controllers, we'll also need to create a `prepareForSegue` method in the initiating view controller's Program Code (in `CSEDetailViewController`).

We are going to transfer the address data from the CSEDetailViewController file into a property named address in the CSEMapViewController. Once the properties are set up create the prepareForSegue:sender: method in the CSEDetailViewController file with the following code:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "map" {
        let dest = segue.destinationViewController as! CSEMapViewController
        dest.address = self.selectedAddress
    }
}
```

The address data that we are sending will be used by a super-cool feature called a geocoder. The Geocoder will translate that address into a latitude and longitude that the map knows how to use.

5.3.3 Mapping it Out

Next we need to react to the address that we are being passed to our CSEMapViewController. To do this we will define a viewDidAppear: method. Why not use viewDidLoad or viewWillAppear? The reason for this is some of the mapping shenanigans; we can't load things onto the map until it is finished loading, and the map begins loading asynchronously around when our viewDidLoad is called, so that is out. viewWillAppear MIGHT be called after the map is loaded, but failure is not an option! Instead we add our drawings in the viewDidAppear, by which point the map is guaranteed to have initialized.

Our goal with these new methods is that we need to be able to take the address that we are provided, find the latitude and longitude of the related address, and mark the location on a map. This first code block will handle geocoding the address to get the latitude and longitude. Add this code to the CSEMapViewController file:

```
override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)
    self.processAddress(self.address)
}

func processAddress(address: String) {
    let geocoder = CLGeocoder()

    // find the latitude and longitude of our address
    // *** This is Asynchronous ***
    geocoder.geocodeAddressString(address, completionHandler:
        {(placemarks, error) -> Void in
        // did our address provide any hits?
        if placemarks?.count > 0 {
         // what is the first (most likely) location that we found?
                let placemark = placemarks![0] as CLPlacemark
                let lat = placemark.location?.coordinate.latitude
                let lon = placemark.location?.coordinate.longitude

                // center our map there
                self.centerMapAt(lat!, lon: lon!)
            }
      }
}

func centerMapAt(lat: CLLocationDegrees, lon: CLLocationDegrees) {
    // when we define a rectangle we need a point and a width/height
    // mapping rectangles use the center point instead of top-left though
    let center = CLLocationCoordinate2DMake(lat, lon)
    let span = MKCoordinateSpanMake(0.01, 0.01)

    // transform our rectangle pieces into a real rectangle!
    let visibleRegion = MKCoordinateRegionMake(center, span)

    // tell the map view to move to this new location and animate the movement
    self.mapView.setRegion(visibleRegion, animated: true)
}
```

Stick a Pin In It

Next we will use the latitude and longitude to place a pin on the map (called an annotation). We'll be creating a new method as shown below, and we also need to call this method so that it is used. It is used in the same way as `centerMapAtLat:andLon:`, perhaps you could call it just below there?

```
func setAnnotationAt(lat: CLLocationDegrees, lon: CLLocationDegrees) {
        let center = CLLocationCoordinate2DMake(lat, lon)
        let annotation = MKPointAnnotation()
        annotation.coordinate = center

        // set these to custom data, possibly send an additional property
        annotation.title = "generic title"
        annotation.subtitle = "generic subtitle"

        self.mapView.addAnnotation(annotation)
}
```

Take Me There

Finally we will add in some code that will draw a static route between the address of whoever we have chosen from the address book and the latitude and longitude of Tempe, AZ. In this code we are using a pre-defined latitude and longitude, but we could use the `geocoder` to locate another address if we wanted to. This subsection is a little tricky, so we'll handle it in 2 steps. First we'll handle adding the overlay by updating our `processAddress` method and creating a `findDirectionsFrom:to:` method:

```
func processAddress(address: String) {
    let geocoder = CLGeocoder()
    geocoder.geocodeAddressString(address, completionHandler:
        {(placemarks, error) -> Void in
        if placemarks?.count > 0 {
            let placemark = placemarks![0] as CLPlacemark
            let lat = placemark.location?.coordinate.latitude
            let lon = placemark.location?.coordinate.longitude

            // center our map there
            self.centerMapAt(lat!, lon: lon!)

            // drop an annotation there
            self.setAnnotationAt(lat!, lon: lon!)

            // -- Add the code below --

            // Create a route from Tempe, AZ to wherever the address is
            // Lat and Lon from Tempe, AZ
            let fromLat = 33.3919224 as CLLocationDegrees
            let fromLon = -111.9281011 as CLLocationDegrees
            let fromCoordinate = CLLocationCoordinate2DMake(fromLat, fromLon)

            // change our inputs to MKPlacemarks
            let toPlacemark = MKPlacemark(placemark: placemark)
            let fromPlacemark = MKPlacemark(coordinate: fromCoordinate,
                addressDictionary: nil)

            // Make map items out of the placemarks
            let fromItem = MKMapItem(placemark: fromPlacemark)
            let toItem = MKMapItem(placemark: toPlacemark)

            self.findDirections(fromItem, to: toItem)
        }
    })
}

func findDirections(from: MKMapItem, to: MKMapItem) {
    let request = MKDirectionsRequest()
    request.source = from
    request.destination = to
    request.requestsAlternateRoutes = true

    let directions = MKDirections(request: request)
    directions.calculateDirectionsWithCompletionHandler({(response, error) -> Void in
        if (error != nil) {
            print("Error!")
        }
        else {
            let route = response!.routes[0] as MKRoute
            self.mapView.addOverlay(route.polyline)
        }
    })
}
```

So what is all that code doing? The new code in `processAddress` is going to create `MKMapItem`s which are needed for the `MKDirectionsRequest` in `findDirectionsFrom:to`. It does this by wrapping the placemark we got from the geocoder (think changing a Rectangle object into a Square object, the data isn't really different, only the representation has changed). The second `MKMapItem` is generated from a hard-coded latitude and longitude (the lat/lon of Tempe, Arizona according to Google Maps).

That part was pretty straight-forward, but this next part is a little different from how we have been coding so far. When we want to draw on a map we do so on something called an `overlay`. We are not allowed to draw on the map directly, so we are basically putting a semi-transparent covering over it, but the `MapView` doesn't know how to make that coloring, so we have a bit of back and forth.

The process for adding an overlay in code goes like this:

- **Us**: Hey Map View, It would cool if you drew this polyline (a bunch of small line segments) when they need to appear on the map.
- **Map View**: No problemo brosky, I'll definitely do that.

Once the user sees part of a map where an overlay should appear there is an exchange like this:

- **Map View**: Hey delegate, this overlay should appear on the line.
- **Delegate(us**: Cool, let me take that data and build/configure a renderer and I'll give it back to you.

- *And then you do that, like a boss.*

That whole interchange happens in these few lines:

```
func mapView(mapView: MKMapView,
    rendererForOverlay overlay: MKOverlay) -> MKOverlayRenderer {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.lineWidth = 5.0
    renderer.strokeColor = UIColor.purpleColor()
    return renderer
}
```

If you don't see the polyline you might want to verify that you set up the delegation in the storyboard. If you didn't then you'll see a whole lot of nothing!

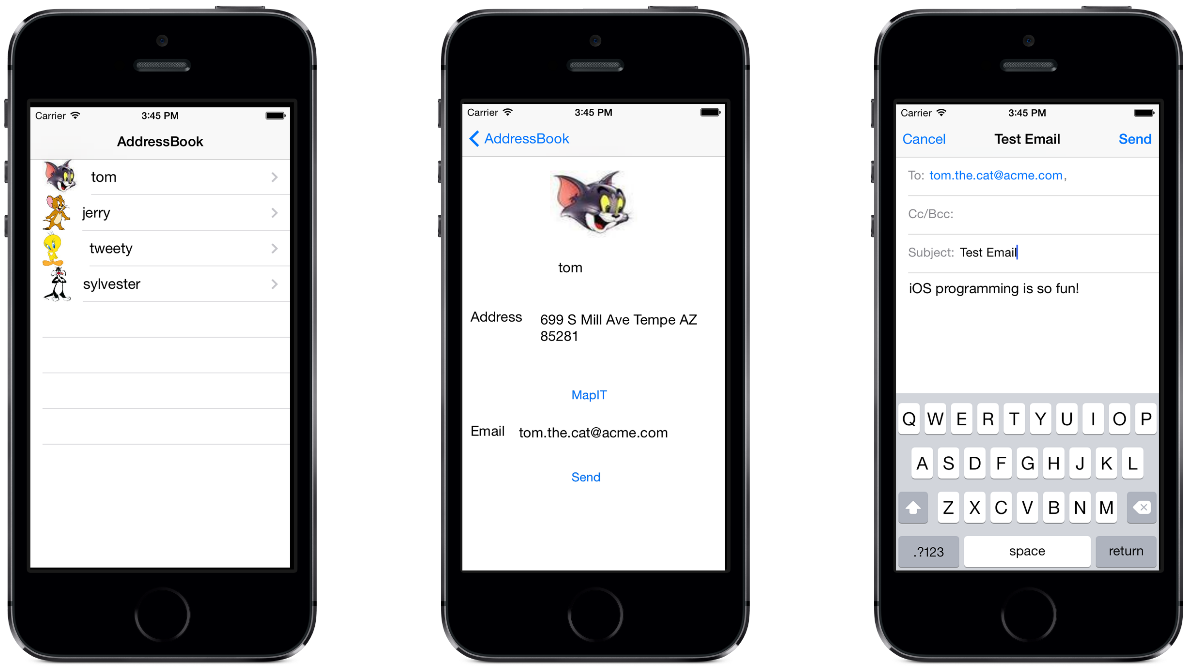
5.3.4 Congrats!

Congratulations, you will no longer get lost looking for cool places to go!

5.4 Address Book - Phase IV

5.4.1 Goals of this Lab!

During this lab we will add the final component to our address book, email! The final result of phase IV should look like the images below:



5.4.2 Have At It!

Update the UI

In the storyboard add a new `Send` button to `CSEDetailViewController`. This new button will be used to trigger sending an e-mail. Link the button to the backing code with an IBAction named `email`. We also need to import the relevant email code as well as set ourselves to be a delegate of creating emails:

```
import MessageUI

class CSEDetailViewController: UIViewController, MFMailComposeViewControllerDelegate {
```

We'll complete the method as shown below:

```
@IBAction func email(sender: AnyObject) {
    // Email subject
    let emailTitle = "Test Email"
    // Email content
    let messageBody = "iOS programming is so fun!"
    // To address
    let toRecipients = [self.selectedEmail] // needs to be an Array

    let mc = MFMailComposeViewController()
    mc.mailComposeDelegate = self
    mc.setSubject(emailTitle)
    mc.setMessageBody(messageBody, isHTML: false)
    mc.setToRecipients(toRecipients)

    // Present mail view controller on screen
    self.presentViewController(mc, animated: true, completion: nil)
}
```

I'm Stuck!

If you try to run the code now you'll see that the mail interface never closes, that's no good at all! What we need to do is implement a `delegate` method coming from the `MFMailComposeViewController` that will tell us when it is done. We can react to that delegation with the method below:

```
func mailComposeController(controller: MFMailComposeViewController,
    didFinishWithResult result: MFMailComposeResult, error: NSError?) {
    // Close the mail interface
    self.dismissViewControllerAnimated(true, completion: nil)
}
```

5.4.3 Congrats!

Congratulations, you've got mail!

Chapter 6

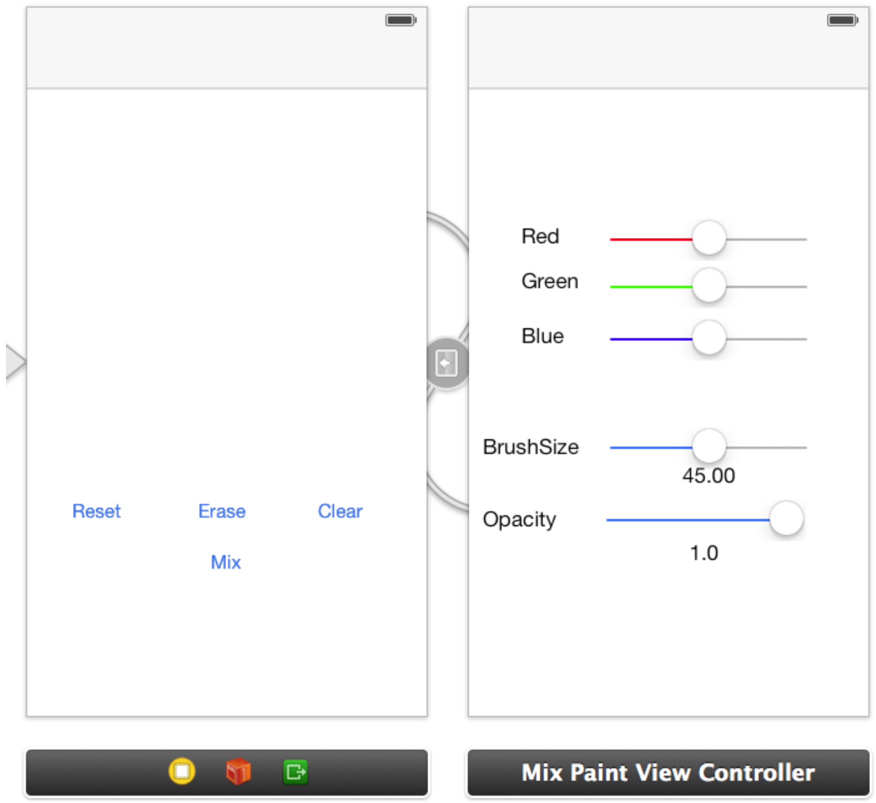
User Touches and Gestures

6.1 Finger Painting - Responding to User Touches

6.1.1 Goals of this Lab!

- Develop a finger painting app
- Implementing 2 view controllers
- Creating a custom `UIView` capable of handling the drawing.

The final product should look like this:



6.1.2 Getting Started

Create New Project with the following settings:

- iOS → Application → Single View Application → Next
- Name = `FingerPainting`
- Class Prefix = `Paint`
- You won't need a git repository, but you're welcome to add one.

6.1.3 Phase I - Do you View What I View?

In this subsection we will be creating a custom `UIView` subclass (whoop whoop!). This view will directly respond to the user touches so that we can easily paint a picture on our screen.

Updating the UI

Since we are making a custom `UIView` we will be doing a lot of the work from the code side meaning that there really isn't much to change in our UI just yet. Look at that, made it through a subsection in only 2 sentences and with no work!

Connecting the PaintView to Program Code

We are going to need to create a new backing class for our view and then connect it in the storyboard:

- Create a new Swift file named `PaintView` and set it as a `subclass` of `UIView`.
- Now go to the storyboard and click in the middle of the view controller, or on the `View` object in the `Document Outline`
- Go to identity inspector, and the class should say `UIView`; change it to `PaintView`.

Writing the Program Code

Now we have our view and it is connected to the backing code but it's totally empty! It's time to implement all the drawing functionality that we are so excited about! In `PaintView` update the code to be the following:

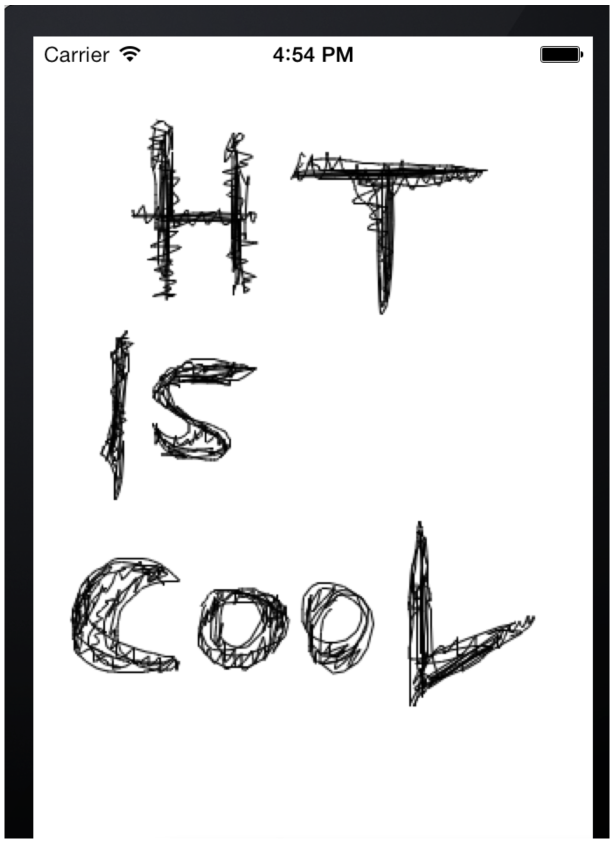
```
import UIKit

class PaintView: UIView {

    var color: UIColor?
    var tempDrawImage: UIImageView? = UIImageView()
    var lastTouch: CGPoint?

}
```

In this next subsection of code we will implement essentially every other function. Our application will track the movement of a users finger across the screen, and every so often we will draw a line segment from where a users finger was up to where it is now. As long as we update frequently enough the drawing should appear smooth.

In the `PaintView` file add these methods:

```
// this method is called automatically when a view is created by the storyboard
// Almost all this code is standard, it has to be here and doesnt customize anything
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    self.configure()
}

// this is a helper method to allow us to set whatever default settings
//      we would like to use
// if you'd like you can even override the .image property to that
//      we have an image at startup
func configure() {
    self.brushColor = UIColor.blackColor()
}

// when a user puts a finger on the screen this method is called automatically
override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
    let touch = touches.first
    self.lastTouch = touch?.locationInView(self)
}

// when a user who has previously put a finger on the screen moves that finger
// this method is called, we will use it to draw the new line segment
override func touchesMoved(touches: Set<UITouch>, withEvent event: UIEvent?) {

    //where did the user's touch(finger) move to?
    let touch = touches.first
    let currentPoint = touch?.locationInView(self)

    //open a graphics context with the old image
    UIGraphicsBeginImageContext(self.frame.size)
    self.tempDrawImage!.image?.drawInRect(CGRectMake(0, 0,
        self.frame.size.width, self.frame.size.height))

    //draw a line from the last touch to the current touch
    CGContextMoveToPoint(UIGraphicsGetCurrentContext(),
        (self.lastTouch?.x)!, (self.lastTouch?.y)!)
    CGContextAddLineToPoint(UIGraphicsGetCurrentContext(),
        (currentPoint?.x)!, (currentPoint?.y)!)

    //configure our drawing options
    CGContextSetLineCap(UIGraphicsGetCurrentContext(), .Round)
    CGContextSetStrokeColorWithColor(UIGraphicsGetCurrentContext(),
        self.brushColor?.CGColor)

    //draw the actual line
    CGContextStrokePath(UIGraphicsGetCurrentContext())

    //store the updated image so that we can draw it more easily
    self.tempDrawImage!.image = UIGraphicsGetImageFromCurrentImageContext()
    self.tempDrawImage!.alpha = 0.5
    UIGraphicsEndImageContext()

    //update the last touch, and refresh the image as seen on the screen
    self.lastTouch = currentPoint
    self.setNeedsDisplay()
}



// this method is called whenever a view needs to display itself.
// this happens automatically, and also whenever we call self.setNeedsDisplay().
override func drawRect(rect: CGRect) {
    UIGraphicsGetCurrentContext()
    self.tempDrawImage!.image?.drawInRect(CGRectMake(0, 0,
        self.frame.size.width, self.frame.size.height))
}
```

Whew, that was a lot of code! Take a few minutes and look back at the code and try to understand what it is doing. At the end of phase I you should be able to draw a picture with a single color and line thickness.

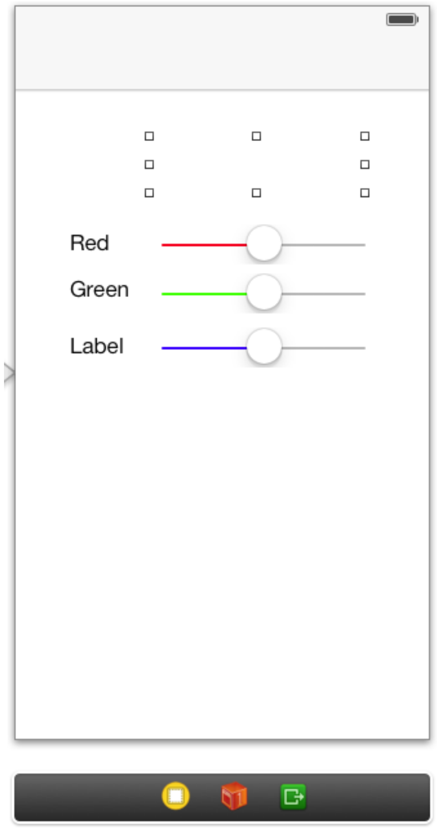


6.1.4 Phase II - Mixing Colors 101

Update The UI

In this subsection we are going to create a new `View Controller` that will allow us to easily mix colors so we can change our brush color. First we will update our UI:

- In the `Storyboard` drag out a new `View Controller`, and name it `MixPaintViewController`.
- On the `PaintViewController` in the storyboard create a new button named `Mix` and create a segue to `MixPaintViewController` from it. Name the new segue `segueToMix`.
- Embed the `PaintViewController` in a `Navigation Controller`.
- On the new `MixPaintViewController` drag out 3 sliders, 3 labels, and 1 button.
- Select the red slider and navigate to the `Attributes Inspector`:
  - Select `min track tint` and select `other`.
  - Select the second tab on the popup and select RGB from the drop down menu.
  - Set red to 255 and others to 0.
- Do the same for the other sliders (for the green slider, green should be set to 255).
- After these steps, the storyboard should look similar to the one below:



Connecting The UI to the Backing Code

- Connect the sliders to the backing code with `IBOutlets` named `redSlider`, `greenSlider`, and `blueSlider`.
- Create actions for the sliders, they can all be connected to a single method called `updateColor`.
- Create an `IBOutlet` for `paintColorButton`.

Writing the Program Code

Now that we've updated our UI let's write the code that will make this color changing work! In MixPaintViewController update the methods with the code below:

```
@IBAction func updateColor(sender: AnyObject) {
    let red = self.redSlider.value
    let green = self.greenSlider.value
    let blue = self.blueSlider.value

    let newColor = UIColor(colorLiteralRed: red, green: green, blue: blue, alpha: 1)
    self.paintColorButton.backgroundColor = newColor

    self.paintColorButton.setNeedsDisplay()
}

override func viewDidLoad() {
    super.viewDidLoad()
    let newColor = UIColor(colorLiteralRed: 0.5, green: 0.5, blue: 0.5, alpha: 1)
    self.paintColorButton.backgroundColor = newColor
}
```

Now you should be able to use sliders to change the color of what you what to draw. This color is represented by the color of the button.

Part 2 - Paint With All the Colors of the Wind

Now we can mix any color and we can see what it looks like, let's actually use it! We previously use the delegation and data source protocols from table views to allow us to pass data, now we are going to create our own delegation protocol so that we can pass the new color back to our view!

- Update your `MixPaintViewController` to include the following protocol and variable based off of that protocol:

  ```
  @objc protocol MixPaintViewControllerDelegate {
      optional func mixPaint(aMixer: AnyObject, aColor: UIColor)
  }

  class MixPaintViewController: UIViewController {

      var delegate: MixPaintViewControllerDelegate?

      // everything else is the same

  }
  ```

- From the storyboard connect the `PaintView` to the `PaintViewController` with an `IBOutlet` named `paintView`.
- Add this code in the `PaintViewController` file:

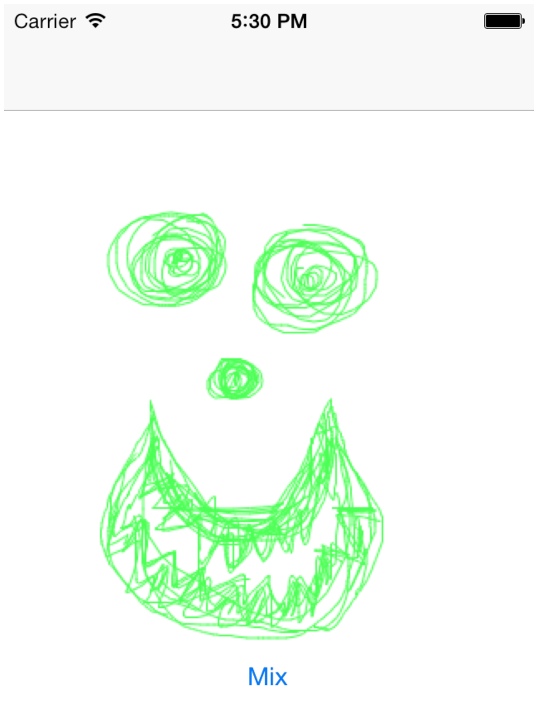
  ```
  func mixPaint(aMixer: AnyObject, aColor: UIColor) {
      self.paintView.brushColor = aColor
  }

  override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
      if segue.identifier == "segueToMix" {
          let dest = segue.destinationViewController as! MixPaintViewController
          dest.delegate = self
      }
  }
  ```

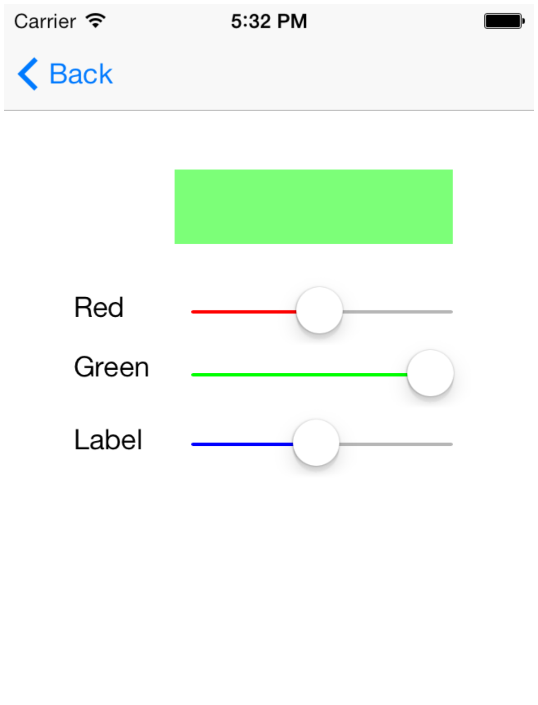
- Now we have the pipeline, a protocol to follow, and someone to follow it. We just need someone to pass on messages, in this case it will be the `MixPaintViewController`. Update the `updateColors` method so that it will tell its delegate when the brush color changes:

  ```
  @IBAction func updateColor(sender: AnyObject) {
      let red = self.redSlider.value
      let green = self.greenSlider.value
      let blue = self.blueSlider.value

      let newColor = UIColor(colorLiteralRed: red, green: green, blue: blue, alpha: 1)
      self.paintColorButton.backgroundColor = newColor
  ```

```
        self.paintColorButton.setNeedsDisplay()

        if self.delegate?.mixPaint != nil {
            self.delegate?.mixPaint!(self, aColor: self.paintColorButton.backgroundColor!)
        }
    }
```

Now you should be able to mix color and draw the color you mixed. Run your app and try it. In the end of phase II it should look like this:





6.1.5 Phase III - Cleaning Up

In this phase we will implement some miscellaneous features, including features to clear up the screen, and the ability to change the brush thickness.

Part 1 - Implementing Reset, Eraser, and Clear Functionalities

Now we will be implementing three more features to increase the flexibility of our program. We will implement a `Reset` button, an `Eraser` and a `Clear` button.

- Add three buttons on to your `PaintViewController`.
- Name them `Reset`, `Erase`, and `Clear`.
- Go to storyboard and open `Assistant Editor`.
- In the `PaintView.m` connect the buttons with `IBActions`, and define the methods as follows (note, you may need to type these methods in and then link them, instead of having the storyboard automatically generate the method prototypes):

```
// resets painter to black colored brush
@IBAction func resetButton(sender: AnyObject) {
    self.configure()
    self.setNeedsDisplay()
}
// clears the canvas
@IBAction func clearButton(sender: AnyObject) {
    let eraseColor = UIColor(colorLiteralRed: 1.0, green: 1.0, blue: 1.0, alpha: 1)
    self.brushColor = eraseColor
}
//paints white color to give impression of an eraser
@IBAction func eraseButton(sender: AnyObject) {
    self.tempDrawImage = UIImageView()
    self.setNeedsDisplay()
}
```

- Now you should be able to `Reset` brush color back to black, `Erase` your painting and `Clear` the canvas.
- Check to make sure that everything works, and then push onwards!

Part 2 - Implementing Brush Thickness and Brush Opacity

- In `PaintView` make two new variables to store the brush thickness and opacity:

```
var brushThickness: Double = 0.0
var opacity: Double = 0.0
```

- In `PaintView` update the `configure` method with the code below:

```
func configure() {
    self.brushColor = UIColor.blackColor()
    self.brushThickness = 15
    self.opacity = 1
}
```

- In the `touchesMoved` method before this line:

```
CGContextSetLineCap(UIGraphicsGetCurrentContext(), .Round)
```

  Add the following:

```
CGContextSetLineWidth(UIGraphicsGetCurrentContext(), CGFloat(self.brushThickness))
CGContextSetAlpha(UIGraphicsGetCurrentContext(), CGFloat(self.opacity))
```

- In the storyboard on the `MixPaintViewController` add two new two new sliders and four labels.
- Connect the two new sliders as actions to the `updateColor` method that we've been using.
- Name two of the labels "Brush Size" and "Opacity" and place them on the side of sliders.
- The other 2 should be connected to `MixPaintViewController` as `IBOutlets` showing the slider values.
- Change the min value to 5, max value to 85, and current to 45 for the brush slider in the `Attributes Inspector`.
- Select the opacity slider and go to `Attributes Inspector` and change `current` to 1.

Updating the Backing Code

Now we have our UI updated so that it can tell the `MixPaintViewController` what the user wants, but we need to update our protocol to pass that extra information back to the `PaintView`.

In `MixPaintViewController` update our protocol to:

```
@objc protocol MixPaintViewControllerDelegate {
    optional func mixPaint(aMixer: AnyObject, aColor: UIColor)
    optional func mixPaintBrushThickness(aMixer: AnyObject, brushThickness:Double)
    optional func mixPaintOpacity(aMixer: AnyObject, opacity: Double)
}
```

In your `MixPaintViewController` file update the `updateColor:` method to:

```
@IBAction func updateColor(sender: AnyObject) {
    // ... everything above should stay, just update the bottom

    if self.delegate?.mixPaint != nil {
        self.delegate?.mixPaint!(self, aColor: self.paintColorButton.backgroundColor!)
    }

    if self.delegate?.mixPaintBrushThickness != nil {
        self.delegate?.mixPaintBrushThickness!(self, brushThickness: Double(brushSlider.value))
    }

    if self.delegate?.mixPaintOpacity != nil {
        self.delegate?.mixPaintOpacity!(self, opacity: Double(opacitySlider.value))
    }
}
```

Finally we will need to take use these new delegation methods to pass information on to the `PaintView`. In your `PaintViewController` file add these new methods:

```
func mixPaintBrushThickness(aMixer: AnyObject, brushThickness: Double) {
    self.paintView.brushThickness = brushThickness
}

func mixPaintOpacity(aMixer: AnyObject, opacity: Double) {
    self.paintView.opacity = opacity
}
```

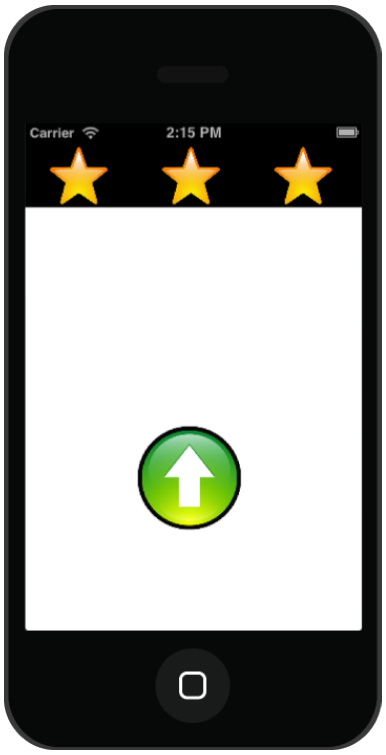
Congrats! You have completed the painting lab!

6.2 Gestures With a Secret Handshake

6.2.1 Goals of this Lab!

- Using gestures to enable a secret handshake lock for our app.
- Defining your own secret handshake and displaying a message when it is entered successfully.

The end result should look something like this:



6.2.2 Part 1

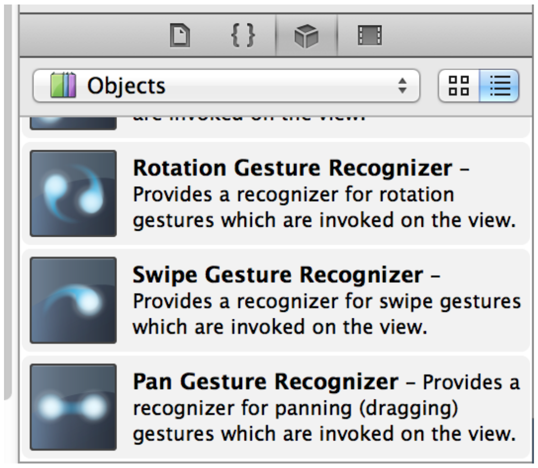
In this part we will just be doing the basic setup which will recognize a single gesture.

Steps

1. Create a new program with the name `SecretHandshake` and implement the same settings we have been using.
2. We will now want to add a new method that will recognize and handle swiping. To do this first open up your `SecretHandshakeViewController` file and add the following method. Once we do this we can manipulate it for future responses:

```
func handleSwipes(sender: UISwipeGestureRecognizer) {

    if (sender.direction == .Up) {
        print("Up swipe!")
    }
    else {
        print("Not up swipe!")
    }
}
```

3. The following are the steps required to add a gesture to your storyboard:
   (a) Drag a swipe gesture recognizer onto the main view from your Object Library.

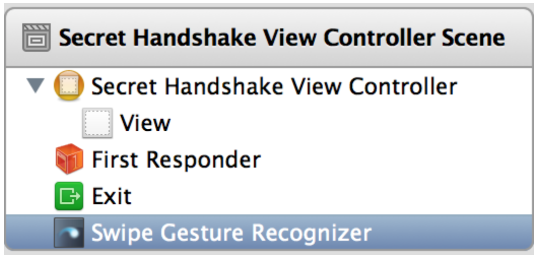


Rotation Gesture Recognizer – Provides a recognizer for rotation gestures which are invoked on the view.

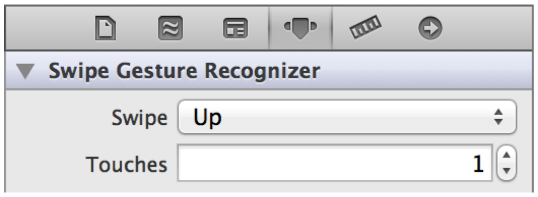
Swipe Gesture Recognizer – Provides a recognizer for swipe gestures which are invoked on the view.

Pan Gesture Recognizer – Provides a recognizer for panning (dragging) gestures which are invoked on the view.

   (b) In the Document outline select the gesture recognizer


Secret Handshake View Controller Scene
Secret Handshake View Controller
View
First Responder
Exit
Swipe Gesture Recognizer

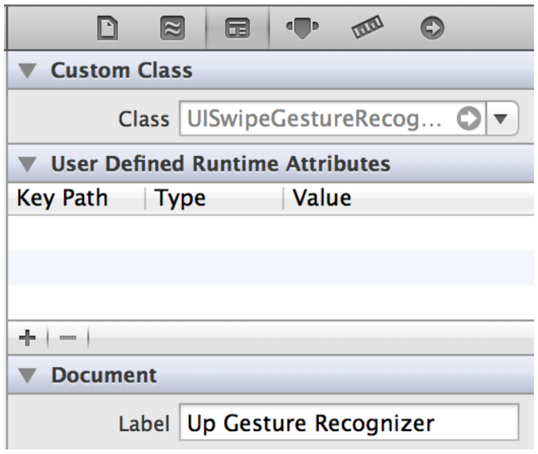
   (c) In the Attributes Inspector change Swipe: to Up



Swipe: Up

Touches: 1

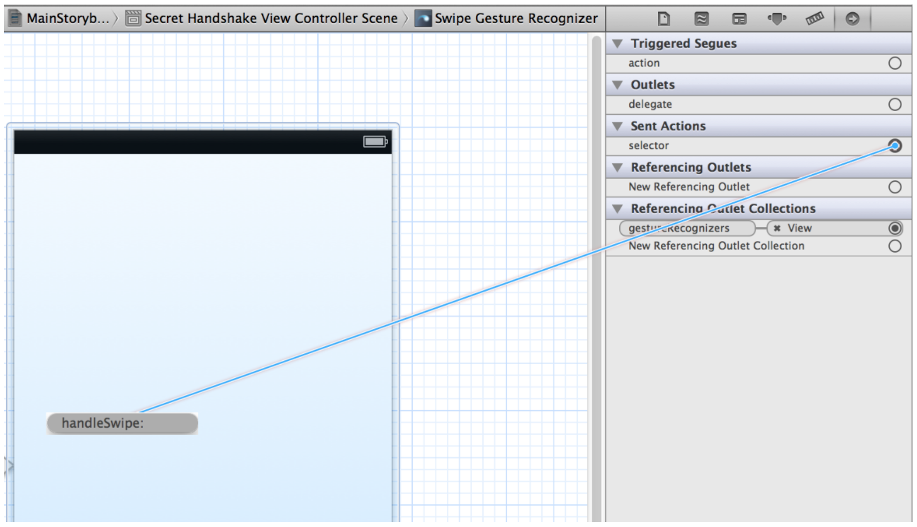
   (d) In the Identity Inspector change Document Label to "Up Gesture Recognizer"



Class: UISwipeGestureRecog...

Label: Up Gesture Recognizer

   (e) In the Connections Inspector drag from "Sent Actions" $\rightarrow$ "Selector" to the controller and select "handleSwipe:"



4. At this point you should be able to swipe up and you will see a message appear for the direction you moved ("Up swipe!"). A swipe is a quick movement of the finger on the screen and if you don't see anything happen when swiping up try swiping left, right, and down just in case.

   - Debugging: if it works on one of the other swipe directions then the `Attribute Inspector` step was probably skipped. If it doesn't work in any direction then the `Selector` might not be attached properly. If neither of those solve your problem raise your hand and we will be over to help resolve the issue.
   - Verify that the previous step works properly before continuing forward. Our next step will change the debug message ("Up swipe!") to be a visible icon on the screen.

5. Add the resource files for this project. There should be four directional arrows, a check_mark, and an x_mark.

   - Note: to make this next step work we are going to need to create an image view inside of our scene. Once we have this image view, our code will be able to move it around and change the image shown. With that in mind we can use the one image view to do a wide variety of tasks!

6. Now you will want to open up the storyboard and do the following:

   (a) Drag out a new Image View from the Object Library at the bottom right of the screen.

   (b) Open the assistant editor and then right-click → drag from the new image view onto the code shown. In the pop-up set the name to be `imageView` and then press connect.

   The image view can now be accessed from your code; the next step will be to place an image and animate it.

7. Now we will be placing the method that is necessary for animating the image view. Inside of your `SecretHandshakeViewController` file insert the following method:

```
func drawImageForGestureRecognizer(imageName: String, atPoint: CGPoint) {
    self.imageView.image = UIImage(named: imageName)
    self.imageView.center = atPoint
    self.imageView.alpha = 1.0
}
```

8. Make sure to also update your `handleSwipes` method that we added earlier as follows:

```
@IBAction func handleSwipes(sender: UISwipeGestureRecognizer) {

    var image = "Incorrect.png"
    let startLocation = sender.locationInView(self.view)
    var endLocation = startLocation

    if sender.direction == .Up {
        image = "Up.png"
        endLocation.y -= 220.0
    }
    else {
        print("Not up swipe!")
    }

    self.drawImageForGestureRecognizer(image, atPoint: startLocation)

    UIView.animateWithDuration(0.5, animations: {
        self.imageView.alpha = 0.0
        self.imageView.center = endLocation
    });
}
```

9. Once everything is working commit your project to git!

6.2.3 Part 2

In this part we are going to recognize all swipes and we will implement some simple images to allow our users to know that their swipes have been recognized!

Steps

1. The first thing that we will need to do is add more gesture recognizers. Repeat the steps that are listed above for the Up gesture three more times; once for left, right, and down. (Make sure they are all linked to the same `handleSwipes` method from earlier!).

2. Now that you have done that you will need to also update the `handleSwipes` method as shown below:

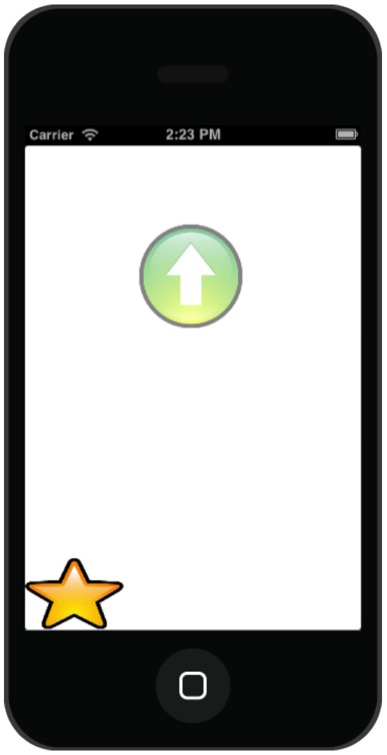
```
if sender.direction == .Up {
    image = "Up.png"
    endLocation.y -= 220.0
}
else if sender.direction == .Down {
    image = "Down.png"
    endLocation.y += 220.0
}
else if sender.direction == .Left {
    image = "Left.png"
    endLocation.x -= 220.0
}
else if sender.direction == .Right {
    image = "Right.png"
    endLocation.x += 220.0
}
else {
    print("Not up swipe!")
}
```

Note: We need the extra conditions so that we can match all directional possibilities!

3. Try running the code now and you should be able to see an arrow in each direction that you swipe. If you get the X icon then one of the swiped directions isn't configured properly. Commit everything to your git once it is working as intended!

6.2.4 Part 3

In this part we will implement the logic that will be able to store and match the secret handshake that the user enters. If it is entered correctly then we will set an icon indicating a successful match and if a gesture is entered incorrectly then we will place a different icon. After this part your app should look similar to this:



Steps

1. First we will need a new `ImageView` that can show the current state of the matching. To do this open the storyboard and do the following:
   (a) Drag out a new Image View from the Object Library at the bottom right.
   (b) Open the assistant editor at the top right and then use a right click→drag from the new image view onto the code show. In the pop-up set the name to be `matchView` and then press connect.
2. In `SecretHandshakeViewController` add the following lines just inside the class definition:

   ```
   var secretHandshake = Array<String>()
   var currentStep = 0
   ```

   Note: these new variables will be used to store our secret handshake and also how close we are to matching the handshake.
3. Still inside this file we want to define the following method:

```
func updateMatchStatus() {
    if currentStep == 0 {
        // No successful matches
        self.matchView.image = UIImage(named: "Incorrect.png")
    }
    else if currentStep < secretHandshake.count {
        // At least 1 gesture matched, but there are more...
        self.matchView.image = UIImage(named: "Matching.png")
    }
    else {
        // Matched all the gestures!
        self.matchView.image = UIImage(named: "Correct.png")
        currentStep = 0
    }
}
```

4. Our next step will be to update the `handleSwipes` method one final time. Replace the contents so that it looks like below:

```
@IBAction func handleSwipes(sender: UISwipeGestureRecognizer) {

    var image = "Incorrect.png"

    let startLocation = sender.locationInView(self.view)

    var endLocation = startLocation

    if sender.direction == .Up {
        image = "Up.png"
        endLocation.y -= 220.0
        print(secretHandshake)
        if secretHandshake[currentStep] == "UP" {
            currentStep += 1
        } else {
            currentStep = 0
        }
    }
    else if sender.direction == .Down {
        image = "Down.png"
        endLocation.y += 220.0
        if secretHandshake[currentStep] == "DOWN" {
            currentStep += 1
        } else {
            currentStep = 0
        }
    }
    else if sender.direction == .Left {
        image = "Left.png"
        endLocation.x -= 220.0
        if secretHandshake[currentStep] == "LEFT" {
            currentStep += 1
        } else {
            currentStep = 0
        }
    }
```

```
        else if sender.direction == .Right {
            image = "Right.png"
            endLocation.x += 220.0
            if secretHandshake[currentStep] == "RIGHT" {
                currentStep += 1
            } else {
                currentStep = 0
            }
        }
        else {
            print("Not up swipe!")
        }

        self.drawImageForGestureRecognizer(image, atPoint: startLocation)

        UIView.animateWithDuration(0.5, animations: {
            self.imageView.alpha = 0.0
            self.imageView.center = endLocation
        });

        self.updateMatchStatus()
    }
```

5. This final step is where all the magic happens in our program. Up until now we've written code to manage our handshake once we've told the computer what it is, now we will define what our handshake is. Inside the file modify the `viewDidLoad` method with the following:

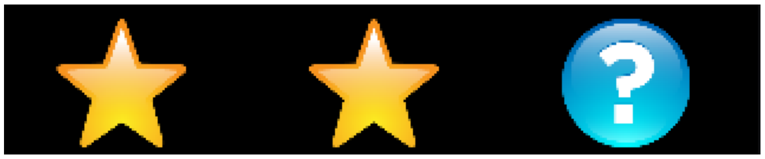
```
    override func viewDidLoad() {
        super.viewDidLoad()
        self.secretHandshake = ["LEFT", "LEFT", "UP"]
    }
```

6. Test out the one listed above to confirm everything works and then create your own secret handshake! Remember that the commands needed have to match what is in the gesture recognizer. (They are UP, DOWN, LEFT, and RIGHT)

7. Once everything is working commit your project to git!

6.2.5 Part 4

The previous part of the lab allowed you to see if each individual guess was correct, but there was no overall progress of the handshake shown. With this final part of the lab we will add a box at the top of the window and it will track our progression towards the secret handshake (shown below).



Steps

1. The first thing we want to do is add the new file that will be used for the current progress.

   - Click on File → New → File.

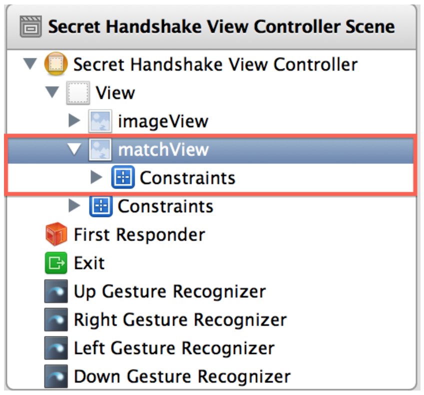
- Confirm Cocoa Touch is selected on the left under iOS and then make sure Cocoa Touch class is selected.
- Click next and then name the class `HandshakeMatches` with `UIView` selected for `Subclass of`.
- Save the file to the project folder and create!

2. It is smart now to remove some information from the previous parts to save memory and make it easier to understand. (It also won't show multiple match status updates) Below are the things that should be `removed` from the code for ease:

   (a) The entire method `updateMatchStatus`.

   (b) The call `self.updateMatchStatus()` at the bottom of the `handleSwipe` method inside of the same class.

   (c) The `ImageView matchView` should be removed from the storyboard.



Secret Handshake View Controller Scene
- Secret Handshake View Controller
  - View
    - imageView
    - matchView
      - Constraints
    - Constraints
  - First Responder
  - Exit
  - Up Gesture Recognizer
  - Right Gesture Recognizer
  - Left Gesture Recognizer
  - Down Gesture Recognizer

   (d) Now we want to add the appropriate properties to our new file `HandshakeMatches`. Below are the properties that should be added:

```
var incorrectImage = UIImage()
var matchingImage = UIImage()
var correctImage = UIImage()

var imageViews = Array<UIImage>()

var numberOfGestures = 0
var correctGuesses = 0

var midMargin = 0
var leftMargin = 0
var minImageSize = CGSize()
```

Note: these properties are all the images that will be utilized for our progress bar. We also create an array of image views (3 pictures) that are managed by the bar. The next two properties are used to determine how many gestures there are and how many are correct. The last three properties adjust the margins of the black box and the guess images.

3. Once the header file has been edited to implement the images we will manipulate the `HandshakeMatches` file to add some more information. We need to add a new method that is going to be called, this method explains to the computer what the variables actually represent. Add the following method to this file:

```
func baseInit() {
    self.incorrectImage = UIImage(named: "Incorrect.png")!
    self.correctImage = UIImage(named: "Correct.png")!
```

```
        self.matchingImage = UIImage(named: "Matching.png")!
        self.midMargin = 5
        self.leftMargin = 0
        self.minImageSize = CGSizeMake(5, 5)
        self.numberOfGestures = 5
        print("Handshake matches initialized!")
    }
```

4. The next thing we are going to do is make sure we call our `baseInit` method. So that being said, change this method to look like below:

```
override init(frame: CGRect) {
    super.init(frame: frame)
    self.baseInit()
}
```

5. We want to add another method to the `HandshakeMatches` file that is generally used for initializing information with a decoder, but it is still important to add in your code in case it is necessary for future usage (and is required in Swift):

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    self.baseInit()
}
```

6. Now we will edit the `drawRect` method that is present in this file and also uncomment it by deleting the `/*` and the `*/`. Include the following information so your `drawRect` looks like below:

```
override func drawRect(rect: CGRect) {
    self.refresh()
    let desiredImageWidth = (self.frame.size.width -
        CGFloat(self.leftMargin*2) - CGFloat(self.midMargin*self.imageViews.count))
        / CGFloat(self.imageViews.count)
    let imageWidth = max(self.minImageSize.width, desiredImageWidth)
    let imageHeight = max(self.minImageSize.height, self.frame.size.height)
    for i in 0...self.imageViews.count-1 {
        let imageView = self.imageViews[i]
        let imageFrame = CGRectMake(CGFloat(self.leftMargin) +
            CGFloat(i)*(CGFloat(self.midMargin)+imageWidth), 0, imageWidth, imageHeight)
        imageView.frame = imageFrame
    }
}
```

7. Once that is finished there are quite a few more methods that need to be added for usability of the application. Below is the method (by modifying the `numberOfGestures` variable) that sets how many gestures need to be done correctly to move on, and if the number of gestures is not at the max (3 in this case), it will reset to 0.

```
var numberOfGestures: Int = 0 {
    didSet {
        for imageView in self.imageViews {
            imageView.removeFromSuperview()
        }
        self.imageViews.removeAll()

        // Add new image views
        for i in 0...numberOfGestures {
            let imageView = UIImageView()
            imageView.contentMode = UIViewContentMode.ScaleAspectFit
            self.imageViews.append(imageView)
            self.addSubview(imageView)
        }

        self.setNeedsLayout()
        self.setNeedsDisplay()
    }
}
```

8. We now need to refresh the information and check whether it is correct or incorrect and adjust the image as appropriate.

```
func refresh() {
    for i in 0...self.imageViews.count-1 {
        let imageView = self.imageViews[i]
        if self.correctGuesses >= i+1 {
            imageView.image = self.correctImage
        }
        else {
            imageView.image = self.matchingImage
        }
    }
}
```

9. Now the box is actually created and the sub-images are created inside of it. Below is the next method that should be added.

```
override func layoutSubviews() {
    super.layoutSubviews()
    let desiredImageWidth = (self.frame.size.width -
        CGFloat(self.leftMargin*2) - CGFloat(self.midMargin*self.imageViews.count))
        / CGFloat(self.imageViews.count)
    let imageWidth = max(self.minImageSize.width, desiredImageWidth)
    let imageHeight = max(self.minImageSize.height, self.frame.size.height)
    for i in 0...self.imageViews.count-1 {
        let imageView = self.imageViews[i]
        let imageFrame = CGRectMake(CGFloat(self.leftMargin) +
            CGFloat(i)*(CGFloat(self.midMargin)+imageWidth), 0, imageWidth, imageHeight)
        imageView.frame = imageFrame
    }
}
```

10. The last three methods that are added to `HandshakeMatches` are below. It notates if it was a correct or incorrect match. And then also how many guesses are correct so far. Note that we are modifying the declaration for `correctGuesses`.

```
func notateIncorrectMatch() {
    self.correctGuesses = 0
    self.setNeedsDisplay()
}

func notateCorrectMatch() {
    self.correctGuesses += 1
    self.setNeedsDisplay()
}
var correctGuesses: Int = 0 {
    didSet {
        self.refresh()
        self.setNeedsDisplay()
    }
}
```

11. Now you do not have to worry about the `HandshakeMatches` files anymore, but we do need our view controller to reference these files. Add the below variable:

    ```
    @IBOutlet weak var handshakeMatcher: HandshakeMatches!
    ```

    Note: this is important to add because we need to be able to use the `HandshakeMatches` files inside of our view controller, but we have no reference to the files. So, we create an outlet of the file itself.

12. In this file you will want to add a new method which is similar to our `viewDidLoad` as in it gets called at the beginning of the application. Below is the method that gets added:

    ```
    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(animated)
        self.handshakeMatcher = HandshakeMatches(frame: CGRectMake(0, 0,
            self.view.bounds.size.width, 60))
        self.view.addSubview(self.handshakeMatcher)
        self.handshakeMatcher?.numberOfGestures = self.secretHandshake.count
    }
    ```

    Note: this is the actual call that sets the boundaries for the black box.

13. The very last thing that needs to be done is call `handshakeMatcher` and tell it how many correct guesses there have been. Inside of our `handleSwipe` method, at the very end, we want to add the below statement which will make the appropriate call:

    ```
    self.handshakeMatcher.correctGuesses = currentStep
    ```

14. Run the program and see if it works! It will now show you when you've completed each part of the handshake. But it isn't quite finished yet in the sense that once you get 3 guesses correct, you can not do anything else! We have one more part to this lab which implements what we've learn in a couple previous labs.

15. We are going to be adding a new view controller which is our "congratulations" screen, and then we need to add a navigation controller to go back to the first view controller. Here are the steps that need to be completed so our application allows for replay, which was outlined in a previous lab:

    (a) Return to the storyboard and drag out a new view controller and put this on to the right of the view controller that already exists. Convention says that the views should flow from a master on the left to later views on the right. On this new view controller add a label and a button. Change the label text to something like "great job!", and change the button text to "Play Again!". We will use this button to return to the previous view and reinitialize the positions.

    (b) The next thing we will want to do is to add a new class to back this view controller. (Add a new class with the same steps from earlier except you want to name it `VictoryViewController` and have it be a subclass of `UIViewController`, NOT `UIView`!)

    (c) After this select the new view controller and go into your `Identity Inspector` and make its class be `VictoryViewController`. This will connect the new view controller with your new class.

    (d) Once you have set and connect the backing class make sure to connect the button to the header file (also make sure it is an action)! Name it something simple like `playAgain`.

(e) Then we need to define a `Push Segue` between the two views. This just defines the ability of one view controller to cause another view controller to be the active view controller. To implement the segue right-click and drag from the top bar of one view controller anywhere onto another view controller.

(f) Embed `SecretHandshakeViewController` in a Navigation Controller. After doing so, ensure the code `self.navigationController?.setNavigationBarHidden(false, animated: false)` is inserted into the `viewDidLoad` method in your `SecretHandshakeViewController` file.

(g) Select the segue (arrow in-between the two view controllers) and go to the `Attributes Inspector` and then name your new segue `segueToVictory` where it says `Identifier`. We will call this from the code when the user gets three correct guesses. Before we update that, let's write out `VictoryViewController.swift`: you should edit the `playAgain` method to look like the following code:

```
@IBAction func playAgain(sender: AnyObject) {
    self.navigationController?.popViewControllerAnimated(true)
}
```

(h) The very last thing we need to do is check if the three gestures are correct, which should unlock the secret handshake and move to the next view. We also have to reset the number of guessed gestures. Go back into your `SecretHandshakeViewController` file and go to the bottom of the `handleSwipe` method. Right after we call `handshakeMatcher` we will add the following lines:

```
if currentStep == 3 {
    currentStep = 0
    self.performSegueWithIdentifier("segueToVictory", sender: self)
}
```

16. Now everything should work and allow you to replay your secret handshake. Don't forget to commit to git once you have finished! If your program does not work, please raise your hand and we will be over to assist you!

Congratulations!

You have finished your Secret Handshake lab!

Chapter 7

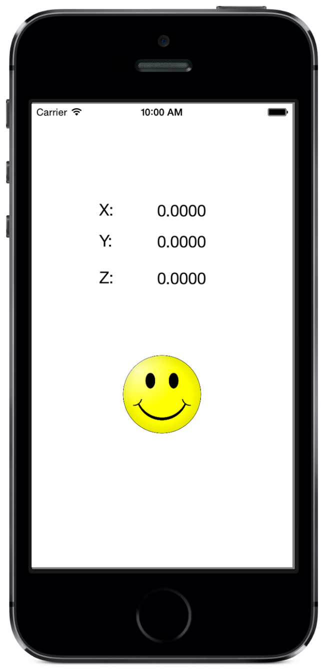
Accelerometer and Timer

7.1 Fun With the Accelerometer

7.1.1 Goals of this Lab!

- Using the accelerometer.
- Using `NSTimer`.
- Intro to update-based coding.

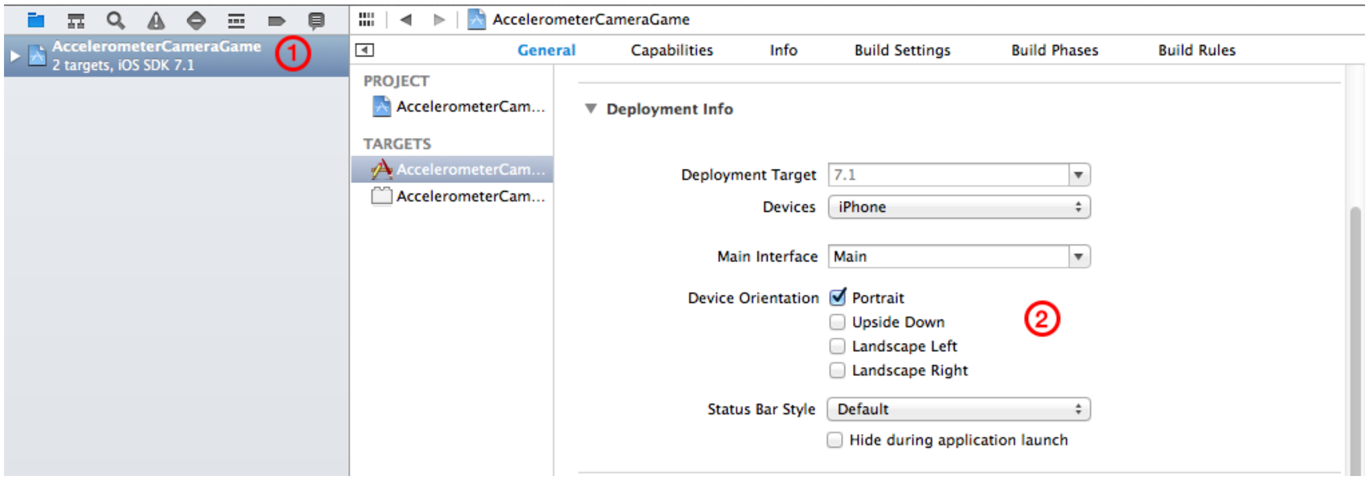
The end result should look something like this:



7.1.2 Getting Started

The first thing we'll need to do is create our project! Create a new program with the name `AccelerometerGame` and implement the same settings we have been using.

In the project settings disable all of the orientations except for portrait (if the screen rotates and the axes change then it will cause problems for us). To do this select your project settings (image below, #1), then under `General` settings, in the `Deployment Info` area deselect `Device Orientations` other than "Portrait", like in the screenshot below:


AcceleratorCameraGame
2 targets, iOS SDK 7.1

General | Capabilities | Info | Build Settings | Build Phases | Build Rules

PROJECT
AccelerometerCam...

TARGETS
AccelerometerCam...
AccelerometerCam...

Deployment Info

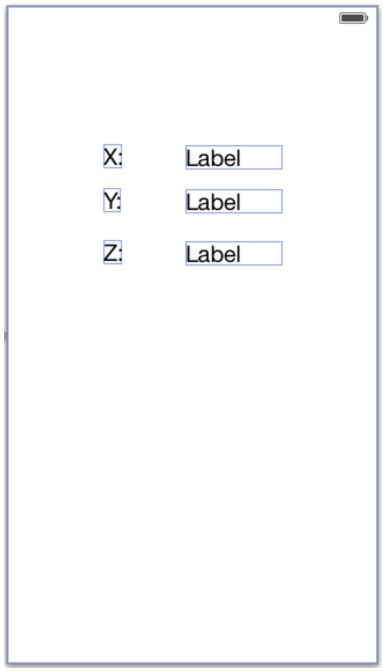
Deployment Target: 7.1
Devices: iPhone
Main Interface: Main
Device Orientation: [✓ Portrait] Upside Down, Landscape Left, Landscape Right
Status Bar Style: Default
Hide during application launch

7.1.3 What Will the Accelerometer Give Us?

It's always good to know what tools you will be using before you get started. In this case we will be utilizing an accelerometer, which can detect acceleration in 3 dimensions, 'x', 'y' and 'z'. Now here is an important question: if our device is not accelerating relative to us (we don't see it moving in any way) what will the accelerometer tell us? To answer this question, and the question of what the x, y, and z axes are all about let's perform a coding experiment!

Build the UI

Here we will build the layout. It will be a very simple application, we just need to drag out 6 `labels`. 3 of the labels will say "x:","y:", and "z:" with static text, the other 3 will be used to update the readings in the respective directions. You can lay them out like the image below:



Connect to the Backing Code

Next we will want to connect the 3 unnamed labels to the backing code. Open up the `Assistant Editor` and connect these (ctrl+click and drag). Create 3 properties that we can use to change the values of the labels. The names that we use are `xlabel`, `ylabel`, and `zlabel`. In this example we connected our properties to the `AccelerometerGameViewController` file.

Using the Accelerometer

Now that we have our interface configured we just need to retrieve the actual values and put them where we want them! The accelerometer utilizes a framework named `Core Motion` so we will need to import this framework, and we will also need to add a few properties to for the `CoreMotionManager`, and an `NSTimer`. In `AccelerometerGameViewController` update your imports to the following:

```
import CoreMotion //add this

class ViewController: UIViewController {

    @IBOutlet weak var xLabel: UILabel!
    @IBOutlet weak var yLabel: UILabel!
    @IBOutlet weak var zLabel: UILabel!

    //add the properties below
    var accelerometerUpdate = NSTimer();
    var motionManager = CMMotionManager();

}
```

Next we will need to create an instance of the `MotionManager`, and our timer. We will do this configuration in the `viewWillAppear:`, which will cause fewer changes down the road. In your `AccelerometerGameViewController` file create the `viewWillAppear:` method as below (the name should auto-complete):

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    self.motionManager = CMMotionManager()
    self.motionManager.accelerometerUpdateInterval = 0.1 //0.1 second interval between updates
    self.motionManager.startAccelerometerUpdates()

    self.accelerometerUpdate = NSTimer.scheduledTimerWithTimeInterval(0.1,
        target: self, selector: #selector(ViewController.update(_:)),
        userInfo: nil, repeats: true);

    print("Finished appearance")
}
```

Now that we have a timer calling our `update:` method we need to define what it should do. The main goal is going to be changing the values of the labels so they reflect the reading from the accelerometer so we can figure out what the x, y, and z axes are. In our update method we would like to print the current accelerometer values to the screen, let's define it now:

```
func update(timer: NSTimer) {

    //get the current accelerometer data
    if let accelerometerData = self.motionManager.accelerometerData {
      //print the accelerations on the x, y, and z axis in the respective labels
      self.xLabel.text = String(format: "%0.4f", accelerometerData.acceleration.x)
      self.yLabel.text = String(format: "%0.4f", accelerometerData.acceleration.y)
      self.zLabel.text = String(format: "%0.4f", accelerometerData.acceleration.z)

      // Will add more here

    }
}
```

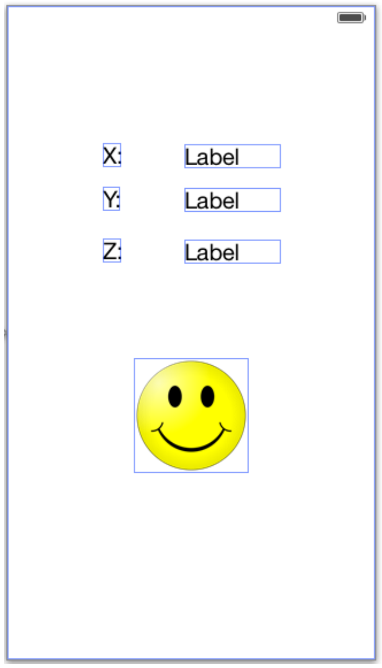
Give the application a whirl and see how it runs! From here we can answer our questions from before, do we get an accelerometer reading when the device isn't changing position or velocity (relative to us)? What do the x,y, and z directions mean to the device? More immediately importantly, how can we use the x, y, and z directions to make something "fall" on the screen?

7.1.4 Let's Get Our Game On!

Ok, so reading the accelerometer is great and fine, but we want to USE the accelerometer!

Update the UI

Let's first make an object that we can allow to fall; from the resources pack add the smiley-face image to the project (or get an image from the web). On the `storyboard` drag out a `Button`. In the `Attributes Inspector` set its `Image` property to be the image that you included. Most likely the image will be HUGE, so go into the `Size Inspector` (ruler icon) next, and change the `width` and `height` to be around 100. Your storyboard should look something like the image below:



Connect the Backing Code

Connect the button to the backing code with a new `property` (not an action) named "imageButton".

We Like to Move it, Move it

Now that we have our image we would like it to respond to gravity. As some of you who have taken physics may know there are 3 main features that we are going to have to consider:

- Position (where is the image)
- Velocity (how fast is the image moving)
- Acceleration (how fast is the velocity changing)

We know the position, and we can read the acceleration, but how can we get the velocity? Acceleration is the change in velocity over time, so we can use the acceleration readings to calculate a velocity, and we can use the velocity to change the position! If this doesn't make sense to you just flag down one of the helpers and we'd be happy to explain in more detail.

To manage our velocity we are going to create 2 new float `properties` named `xVelocity` and `yVelocity` in our class:

```
class ViewController: UIViewController {

// ... everything above can stay

  var xVelocity : CGFloat = 0;
  var yVelocity : CGFloat = 0;

}
```

Next we will want to make sure that we update our velocity and position whenever our acceleration is updated. One simple solution for this is to put all of our update code into the update method we have defined already. We will need to add 2 new features:

- Update the velocity with the accelerometer reading, and
- Update the position with the calculated velocity.

Add the new code to the `update:` method like below:

```
func update(timer: NSTimer) {

    //get the current accelerometer data
    if let accelerometerData = self.motionManager.accelerometerData {

      //print the accelerations on the x, y, and z axis in the respective labels
      self.xLabel.text = String(format: "%0.4f", accelerometerData.acceleration.x)
      self.yLabel.text = String(format: "%0.4f", accelerometerData.acceleration.y)
      self.zLabel.text = String(format: "%0.4f", accelerometerData.acceleration.z)

      //update the velocity
      self.xVelocity = self.xVelocity + CGFloat(accelerometerData.acceleration.x * 0.1)
      self.yVelocity = self.yVelocity - CGFloat(accelerometerData.acceleration.y * 0.1)

      //update the position
      let center = self.imageButton.center
      let destx = center.x + self.xVelocity * 0.1
      let desty = center.y + self.yVelocity * 0.1

      //set the updated position
      let dest = CGPointMake(destx, desty)
```

```
        self.imageButton.center = dest
    }
}
```

Give the code a test-run and you should be able to see your smiley flying around the screen! Oh noooo, it's difficult to recover him if he falls off the screen, maybe we can build a box to ensure that he doesn't fall off the screen!

7.1.5 Setting Boundaries

The next important step is going to be defining our boundaries, we can pick any arbitrary boundary but some will definitely be easier to enforce. We could allow the image to fall slightly off the screen, or perhaps to almost reach the corner of the screen (but fall short by a few points). On the other hand if we make the boundary exactly the screen boundary we already have a bounding rectangle defined for us!

Yup, That's a Wall

The first step is going to be detecting a collision, and being able to respond to it. Thankfully this is super easy for us! In the `update:` method add the new collision detection code at the bottom:

```
func update(timer: NSTimer) {
{
    // ... everything above should remain

    // check for collisions, is the image completely contained within our view?
    // if not then it must have left our view (off screen)
    if !(CGRectContainsRect(self.view.frame, self.imageButton.frame)) {
            print("Bump")
    }
}
```

Give that code a run, you should see a steady stream of `Bump!` messages coming as soon as our image begins to exit the scene. Now let's modify the code to ensure that our boundary behaves like one!

You Shall Not Pass!

We want to make sure that our wall is behaving like a wall, and we'll need some code to achieve that. What does that entail? We know there is a collision, but we're not sure which wall we collided with, so we should figure that out. We'll also need to modify our velocity, and maybe our position (to give it a real "bounce" feeling). That sounds like a lot of comparisons and code, we'll need a good place to put it.

The obvious choice is to add the code to our update: method, since it is already detecting the collision anyways, right? This is an important design decision though; software engineering practices say that every method should have one job (see: not two), so we want to ensure that our update method stays simple, let's instead make a `enforceBoundary` method that will handle the collisions. Define this new method, and take a look at what each step is doing. In particular, try to imagine each case you might encounter (what happens our image leaves the boundaries slightly? What happens if our image is WAY out of the boundaries? What happens if our image remains in the boundary?).

```
func enforceBoundary() {
    //find out where our image is trying to go
    let center = self.imageButton.center;
    var destx = center.x;
    var desty = center.y;

    //how big is our image?
    let xRadius = self.imageButton.frame.size.width/2;
    let yRadius = self.imageButton.frame.size.height/2;

    //are we trying to leave the "left" side?
    if (destx - xRadius < 0) {

      //make sure that the center of the image would be one radius distance
      //from the edge
      destx = self.imageButton.frame.size.width/2;

      //invert the velocity to simulate a bounce
      self.xVelocity = -self.xVelocity;
    }

    if (destx + xRadius > self.view.frame.size.width) {
      destx = self.view.frame.size.width - xRadius
      self.xVelocity = -self.xVelocity
    }

    if (desty - yRadius < 0) {
      desty = self.imageButton.frame.size.height/2;
      self.yVelocity = -self.yVelocity
    }

    if (desty + yRadius > self.view.frame.size.height) {
      desty = self.view.frame.size.height - yRadius
      self.yVelocity = -self.yVelocity
    }

    //reconstruct the image location and move the image to that point
    let dest = CGPointMake(destx, desty)
    self.imageButton.center = dest
}
```

One really cool feature about this code is that it will only have an effect if our image is outside of the boundaries. So we don't need to do any checking in the `update:` method, we just need to call this method. As a final step let's modify the `update:` method to account for this change:

```
func update(timer: NSTimer) {

    //get the current accelerometer data
    if let accelerometerData = self.motionManager.accelerometerData {

      //print the accelerations on the x, y, and z axis in the respective labels
      self.xLabel.text = String(format: "%0.4f", accelerometerData.acceleration.x)
      self.yLabel.text = String(format: "%0.4f", accelerometerData.acceleration.y)
      self.zLabel.text = String(format: "%0.4f", accelerometerData.acceleration.z)

      //update the velocity
      self.xVelocity = self.xVelocity + CGFloat(accelerometerData.acceleration.x * 0.1)
      self.yVelocity = self.yVelocity - CGFloat(accelerometerData.acceleration.y * 0.1)

      //update the position
      let center = self.imageButton.center
      let destx = center.x + self.xVelocity * 0.1
      let desty = center.y + self.yVelocity * 0.1
```

```
        //set the updated position
        let dest = CGPointMake(destx, desty)
        self.imageButton.center = dest

        // check for collisions, update if necessary
        self.enforceBoundary();
    }
}
```

7.1.6 Polish and Swagger

Great job on completing this lab! If you have time and want to build on this solution to do even more cool things here are a few ideas for activities:

- The image has a very clunky bounce. If you notice, sometimes when the image hits the edge of the screen it seems that the first position update isn't quite right. Essentially the image is trying to move to a position like -3, and we put it at 0. If it truly bounced it would probably be at a location closer to +3, can you make a modification to achieve this?
- Real objects typically undergo forces like friction, or elasticity in collisions. Can you modify the code so that our image loses some fraction of its velocity at every time step, and also loses some velocity when it bounces? What happens when the friction in the "x" direction doesn't match the friction in the "y" direction?
- Our image (if you used the smiley) is a circle. But our collision region is a square, can you modify the code so that collisions are only detected on the actual image?
- Can you modify the bounding box so that the image is more or less restricted?
- Can you build a small maze where the image could possibly leave but only if you find the right path? This might be easier to do by adding extra elements onto the screen
- What happens when you modify the frequency of the `NSTimer`? Can you counteract this feature so that no matter how frequently you are called you will always move approximately the same amount?

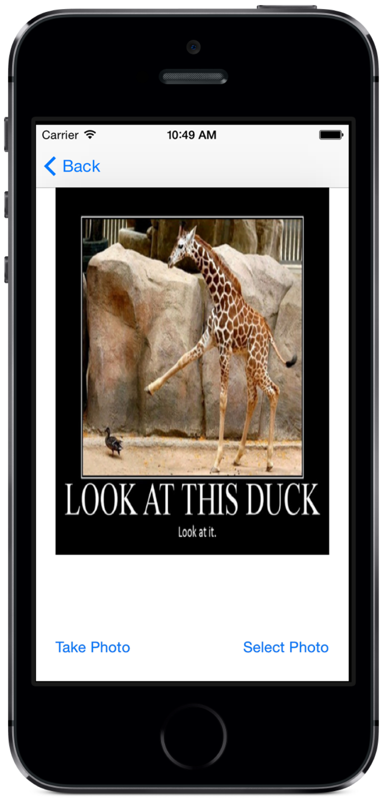
Chapter 8

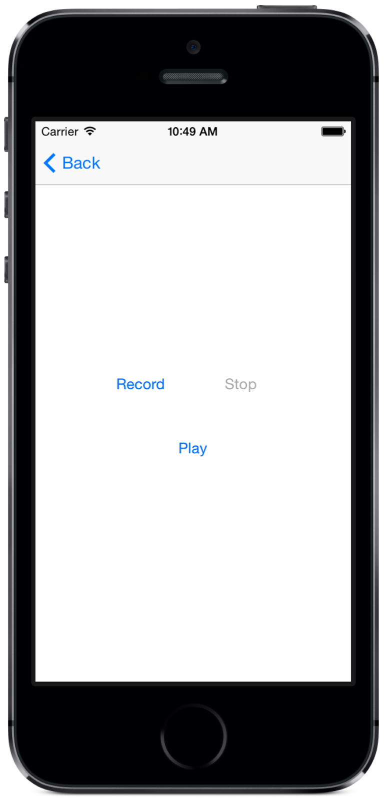
Camera, Microphone, and the Web

8.1 Using the Camera and Microphone

8.1.1 Goals of this Lab!

In this app we will be learning about the `UIImagePickerController`, and using it to take pictures. We will also learn about audio sessions and using them to record and play back sounds. This lab will build on the accelerometer lab that was completed previously. The end result of the lab will be the following screens:





8.1.2 Getting started

Because we are building upon the accelerometer application we don't need to use Xcode to create a new project. To begin, make a copy of the accelerometer lab (the entire folder that contains the `.xcodeproj` file). We are going to be editing this copy of the accelerometer code.

8.1.3 Changing the Image

Smiley-faces are super cool and all, but wouldn't it be great if we could take pictures of whatever we want and use them instead? That's what we'll be doing in this subsection, and we will make use of the `UIImagePickerControllerDelegate`.

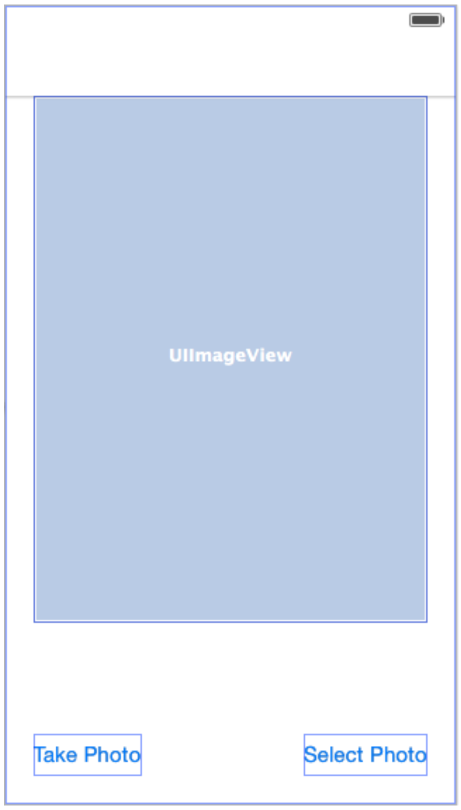
Add a View Controller

The first thing we want is a new view controller that can handle the picture taking, and a way to get there. In the `storyboard` drag out a new `View Controller` object. On this new view controller add:

- 1 `Image View`
- 2 `Buttons`

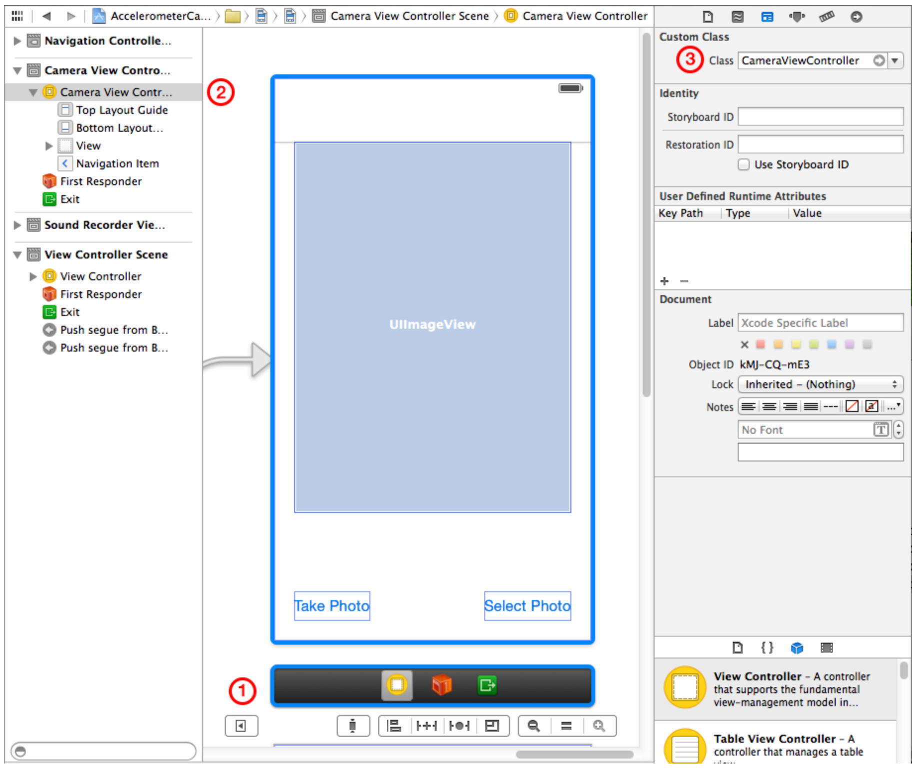
The buttons will be used to determine where we are getting the photo from, one will take a photo from the camera, and another will use the camera roll (the photos already on the device). You can name the buttons however you feel makes the most sense, but we like the names `Take Photo` and `Select Photo`.

Once the photo buttons are created you'll need to select the view controller, and in the `Attributes Inspector` select the `Is Initial View Controller` checkbox. This will ensure that we default to this scene, and we can test that everything is working later on. After this step your storyboard should have a new view controller that looks like this:



Connect the ViewController

Now we will need some code that can control the storyboard view controller. Add a new file named `CameraViewController` that is a subclass of `UIViewController`. Select the `CameraViewController` and then change the `Class` in the `Identity Inspector`. If you don't see the `Document Outline` tap the small play button in the bottom left corner of the storyboard.



Next we will want to define a few outlets so that the code can interact with the user interface. We're going to define some outlets and actions, so let's open the `Assistant Editor`. We are going to need:

- 1 action from the `Take Photo` button named `takePhoto`
- 1 action from the `Select Photo` button named `selectPhoto`
- 1 property for the `UIImageView` named `imageView`

Remember that when you ctrl+click and drag a button default definition in the popover may be for a property, but we want an action.

Add the Code

In the `CameraViewController` file we are going to have to complete the `takePhoto` and `selectPhoto` methods like below. These methods should have been added when you linked the storyboard view controllers, so you should just be adding the content:

```
// Take photo action from the storyboard.
@IBAction func takePhoto(sender: UIButton) {
    let camera = UIImagePickerControllerSourceType.Camera
    if UIImagePickerController.isSourceTypeAvailable(camera) {
        let picker = UIImagePickerController()
        picker.delegate = self
        picker.allowsEditing = true
        picker.sourceType = UIImagePickerControllerSourceType.Camera

        presentViewController(picker, animated: true, completion: nil)
    }
    else {
        print("Oh noes, the camera doesn't work on the simulator!")
    }
}

// Select photo action from the storyboard.
@IBAction func selectPhoto(sender: UIButton) {
    let savedPhotos = UIImagePickerControllerSourceType.SavedPhotosAlbum
    if UIImagePickerController.isSourceTypeAvailable(savedPhotos) {
        let picker = UIImagePickerController()
        picker.delegate = self
        picker.allowsEditing = true
        picker.sourceType = UIImagePickerControllerSourceType.SavedPhotosAlbum

        presentViewController(picker, animated:true, completion: nil)
    }
}
```

This will cause a `UIPickerView` to use the screen modally, and once it has completed it will tell its delegate (us) that it is finished. To handle this information we will need to implement the protocol, and the two delegate methods that will be called.

To tell the compiler that we intend to support the two protocols that the `UIImagePickerController` wants to use, we need to change a few things. To make this change update the definition of the class line to be the following:

```
class CameraViewController:
    UIViewController, UIImagePickerControllerDelegate, UINavigationControllerDelegate {
```

Then implement the required methods in `CameraViewController`:

```
// Call this function when an image picker operation is done.
func imagePickerController(picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    let chosenImage = info[UIImagePickerControllerEditedImage] as? UIImage
    self.imageView.image = chosenImage
    picker.dismissViewControllerAnimated(true, completion: nil)

    // State model update.
    let model = StateModel.sharedInstance
    model.image = chosenImage
    print(model.image)
}

// Call this function when an image picker operation is cancelled.
func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    picker.dismissViewControllerAnimated(true, completion: nil)
}
```

One last step that we will want to do is to ensure that our navigation bar is returned once we have everything linked back up. To do this we'll have to add the line below to our `viewDidLoad` method as shown below:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.navigationController?.setNavigationBarHidden(false, animated: true)
}
```

Test it Out

At this point we should be able to use the `UIImagePicker` to choose an image either from the camera roll, or from the active camera. Test these features to make sure that both of them work before moving on... But first, let's take a selfie!

8.1.4 Sound Off, One Two

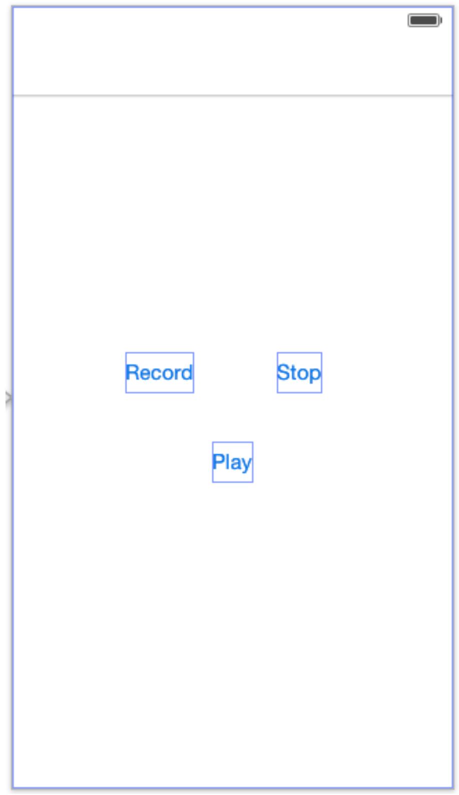
In this subsection we will be integrating the `AVFoundation` framework into our application so that we can play and record arbitrary sounds. To accomplish this feat we will be creating a new view controller, and a new model class that can handle the storage and playing of sounds.

Add a View Controller

We're going to create a new view controller that can handle recording and playing audio. In the storyboard drag out a new `View Controller` for us to use. On this new view controller add:

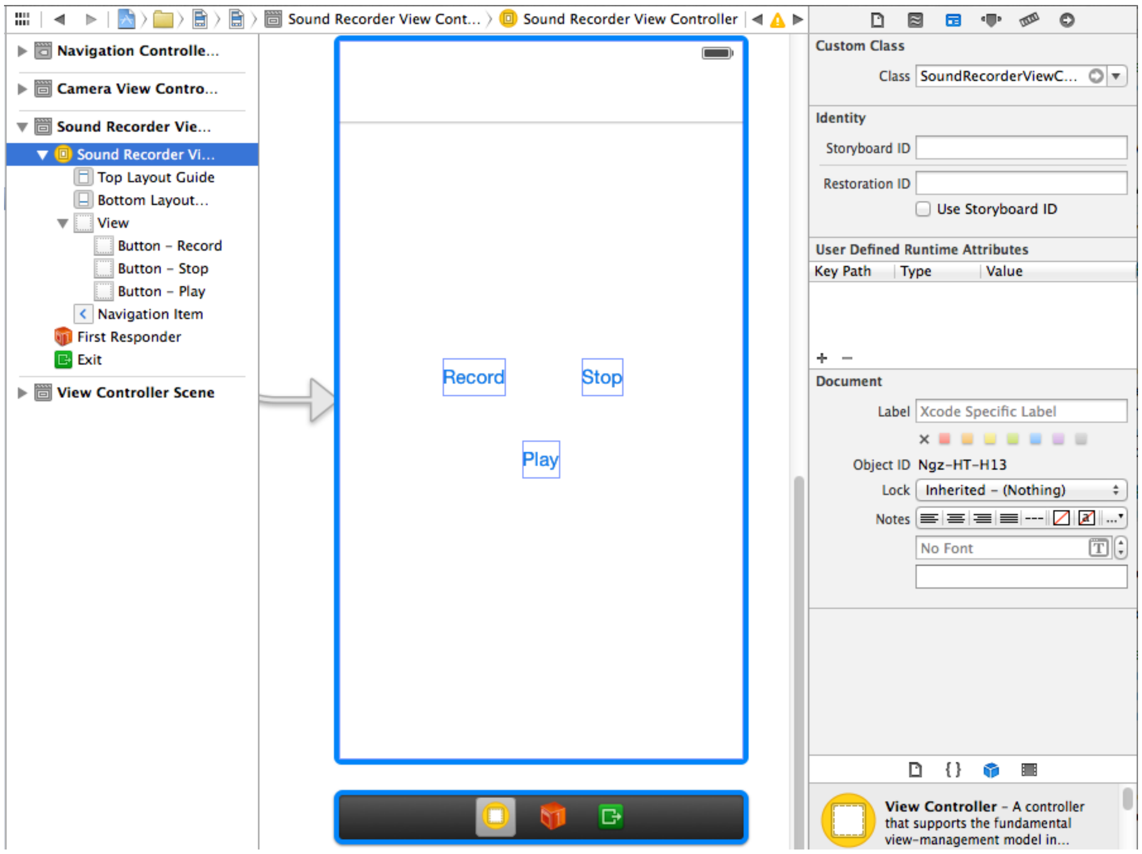
- 3 `Buttons`

The buttons will be used to start a recording, finalize a recording, and to play back a recording. Again you can name the buttons however you see fit, but we used the names `Record`, `Stop`, and `Play`. We will also want to start on this view controller so we can work on this code, in the `Attributes Inspector` select the `Is Initial View Controller` checkbox. After this step your storyboard should have a new view controller that looks like this:



Connect the View Controller

Now we will need some code that can allow the view controller to react to events. You know the drill! Add a new file named `SoundRecorderController` that is a subclass of `UIViewController`. In the storyboard make sure that you assign the code to back the view controller like we have done before (and in the image below):



Next we are going to create a property and an action for each button (the properties will allow us to modify the button behavior during recording). Open the `Assistant Editor` and make the following connections:

- 1 action from the `Record` button named `recordPauseTapped`.
- 1 action from the `Stop` button named `stopTapped`.
- 1 action from the `Play` button named `playTapped`.
- 3 properties named `recordPauseButton`, `stopButton`, and `playButton`.

Add the Code

If you run the program now you should see our new view controller with 3 buttons that currently do nothing. Now we're going to make our app sing!

Configuring the Audio Session

Before doing any more coding we are going to finish up our file by declaring our support for some delegation protocols (which we will talk about later), and some properties to help us with audio recording and playback. In the `SoundRecorderViewController` file add the following code:

```
import UIKit
import AVFoundation

class SoundRecorderViewController: UIViewController,
    AVAudioRecorderDelegate, AVAudioPlayerDelegate {
```

```
    @IBOutlet weak var recordPauseButton: UIButton!
    @IBOutlet weak var stopButton: UIButton!
    @IBOutlet weak var playButton: UIButton!

    var recorder: AVAudioRecorder!
    var player: AVAudioPlayer!

}
```

The `AVAudioRecorder` class allows us to record sound in iOS very easily. To use it we only have to set up a few things:

- Specify a sound file URL (where will our sound be saved),
- Set up an audio session (what sound activities will we be doing?), and
- Configure the `AVAudioRecorder` with the format we want to record and the sampling rate.

To handle this initialization we are going to use the `viewDidLoad` method. Since the `viewDidLoad` method is called before the scene is shown on screen we can assume that we'll be ready to record by the time the user is able to ask us to record. Update the `viewDidLoad` method in `SoundRecorderViewController` with the following, also add the `getAudioURL` helper method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.navigationController?.setNavigationBarHidden(false, animated: true)

    // Disable Stop/Play buttons when application launches.
    self.stopButton.enabled = false
    self.playButton.enabled = false

    // Set up audio session.
    let session = AVAudioSession.sharedInstance()
    do {
        try session.setCategory(AVAudioSessionCategoryPlayAndRecord)
    }
    catch {
        print("Failed to initialize recording session.")
    }

    // Define the recorder setting.
    let recordSetting = [
        AVFormatIDKey: Int(kAudioFormatMPEG4AAC),
        AVSampleRateKey: 44100.0,
        AVNumberOfChannelsKey: 2 as NSNumber,
    ]

    // Set the audio file.
    let outputFileURL = getAudioURL()

    // Initialize and prepare the recorder.
    do {
        recorder = try AVAudioRecorder(URL: outputFileURL, settings: recordSetting)
        recorder.delegate = self
        recorder.meteringEnabled = true
        recorder.prepareToRecord()
    }
    catch {
        print("Failed to initialize the recorder.")
    }

}

// Where are we going to save the recording?
func getAudioURL() -> NSURL {

    let pathComponents = NSSearchPathForDirectoriesInDomains(.DocumentDirectory,
        .UserDomainMask, true) as [NSString]
    let audioFileName = pathComponents[0].stringByAppendingPathComponent("MyAudioMemo.m4a")
    return NSURL(fileURLWithPath: audioFileName)
}
```

Great, so that was an excellent copy paste adventure, but what does that all do? First we do some basic UI configuration; We make sure the navigation bar is showing, and we disable everything except the record button (we can play if we have no recording). Next we set up an `AVAudioSession`, this allows us to tell the device that we intend to take control of the audio, and specifically that we intend to play audio, and to record audio.

We then create an `AVAudioRecorder` tell it how and where to record (what audio compression format do we want, and how many samples per second would we like, where does the recording go), the recorder will connect to the session without our interaction (so cool!). There is a lot more to audio recording if you want to play around with it, but if you just want to capture audio then you don't need any more configuration than this.

One of the settings that we give the recorder is the location we want to save our recording (`getAudioURL`). Here we'll do some hand waving and say that on the device there is a home directory (like in your "My Documents" folder), we are asking the system for the folder and in it we are creating a path to a file named `MyAudioMemo.m4a`. The Audio Recorder will write to this URL whether or not something is there, so be careful not to accidentally overwrite files!

Implementing the Record/Pause Button

We have completed the audio preparation. Let's move on to implement the action method of the `Record` button. Before we dive into the code, let's explain how the `Record` button works. When user taps the `Record` button, the app will start recording and the button text will be changed to `Pause`. If user taps the `Pause` button, the app will pause the audio recording until the `Record` is tapped again.

The audio recording will only be stopped (and saved) when user taps the `Stop` button. The way that the recording is "paused" or "stopped" is totally hidden from us, we don't need to know exactly what the system is doing. Edit the `recordPauseTapped:` method with the following code:

```
@IBAction func recordPauseTapped(sender: UIButton) {

    // Stop the audio player before recording.

    if self.player != nil{
        if self.player.playing {
            self.player.stop()
        }
    }

    if !self.recorder.recording {

        let session = AVAudioSession.sharedInstance()
        do {
            try session.setActive(true)
        }
        catch {
            print("Fail to start recording session.")
        }

        // Start recording.
        self.recorder.record()
        self.recordPauseButton.setTitle("Pause", forState: .Normal)
    }
    else {

        // Pause recording.
        self.recorder.pause()
        self.recordPauseButton.setTitle("Record", forState: .Normal)
    }

    self.playButton.enabled = true
    self.stopButton.enabled = true

}
```

In the above code, we first check if the audio player is playing. If it is playing we stop it by using the `stop` method. Since this button is used for both starting and pausing a recording we must next do a check to see which behavior we should exhibit. If the recorder is not recording then a user tapping on this button will want to start recording. If the recorder has already started recording then the user probably wants to pause a recording.

Implementing the Stop Behavior

For the `Stop` button, we simply call the `stop` method on the recorder, this tells our recorder that this recording is completed, and it should be written to the disk (storage on the phone). We also set the `audioSession` to inactive so that when the next app wants to use the session it knows that no one else is using it.

```
@IBAction func stopTapped(sender: UIButton) {

    self.recorder.stop()

    let audioSession = AVAudioSession.sharedInstance()
    do {
        try audioSession.setActive(false)
    }
    catch {
        print("Failed to complete the recording.")
    }
}
```

You can make use of the `AVAudioRecorderDelegate` protocol to handle audio interruptions (say, a phone call during audio recording) and the completion of recording. In this case we will only hook into the completion of a recording, the `audioRecorderDidFinishRecording:` method is called after the the `self.recorder.stop()` line has finished processing.

In this case we control both the recorder and the view that is controlling the recorder, so we already know when the recording is completed, but in many cases this isn't true. Soon we'll see a case where one object is playing a sound (through the session), and a second object is triggering the first object to play. Add the following code:

```
func audioRecorderDidFinishRecording(recorder: AVAudioRecorder, successfully flag: Bool) {

    self.recordPauseButton.setTitle("Record", forState: .Normal)
    self.playButton.enabled = true
    self.stopButton.enabled = false

    // State model update.
    let model = StateModel.sharedInstance
    model.setRecURL(self.getAudioURL())

}
```

This changes the title of the play/record button back to `Record`, since we are no longer recording. It also enables the play button (since we have something to play back), and disables the stop button (since we no longer have anything to stop).

Playing it Back

Finally, it comes to the implementation of the `Play` button for audio playback using `AVAudioPlayer`. In `SoundRecorderViewController`, edit the `playTapped:` method using the following code:

```
@IBAction func playTapped(sender: UIButton) {

    if !self.recorder.recording {
        do {
            self.player = try AVAudioPlayer(contentsOfURL: self.getAudioURL())
            self.player.delegate = self
            self.player.play()
        }
        catch {
            print("Failed to initialize the player.")
        }
    }
}
```

Here we are creating an audio player and telling it to play audio based on the URL that we recorded to. This is where using methods is very effective, if we change the location that we are recording to in one location then it will change everywhere, leading to fewer errors in our code.

We can also implement some delegate methods from `AVAudioPlayer`, like `AVAudioRecorder`. These methods inform us that the recording has finished playing back. You can implement the method below to see how we can use the callback to pop up a simple alert when the playback has completed:

```
func audioPlayerDidFinishPlaying(player: AVAudioPlayer, successfully flag: Bool) {
    let alert = UIAlertController(title: "Done",
        message: "Recording completed successfully!",
        preferredStyle: UIAlertControllerStyle.Alert)
    alert.addAction(UIAlertAction(title: "Ok",
        style: UIAlertActionStyle.Default, handler: nil))
    self.presentViewController(alert, animated: true, completion: nil)
}
```

Test it out

You can test audio recording and playback using a physical device or software simulator. If you test the app using actual device (e.g., iPhone), the audio being recorded comes from the device connected by the built-in microphone or headset microphone. On the other hand, if you test the app by using the Simulator, the audio comes from the system's default audio input device as set in System Preferences.

So go ahead to compile and run the app! Tap the `Record` button to start recording. Say something, tap the `Stop` button and then select the `Play` button to listen to the playback.

8.1.5 Bringing it All Together

In this subsection we are going to implement a new `Model` singleton that is going to help us integrate all of the pieces and bring our simple game together. In particular we would like to allow the user to choose their own image, which we can select from the `CameraViewController`; and choose their own sound to play when a "bounce" occurs, which we can select from the `SoundRecorderViewController`!

Creating Connections

The first step to making out view controllers work together is creating segues between them, and adding a navigation controller. To do this, embed the `accelerometerViewController` in a `Navigation Controller`, then set that `Navigation Controller` as the `initial View Controller`. In our case we used the image-Button as a link to the `Camera View Controller`, and added a new button that would take us to the `Microphone View Controller`. You're welcome to think up your own way of linking the views though.

Creating the State Model

Here we are going to create our state model singleton, which will facilitate communication between our other components. A singleton is an object of which only one can exist, this is particularly useful when talking about managing configurations, or taking input from the user. We will use it to store the last sound recording, and the last updated image.

First, create a new file which will be a subclass of `NSObject` named `StateModel`. We are going to create a few properties that will store information regarding the current configuration of the application. To do this add the following code:

```
import UIKit
import AVFoundation

class StateModel: NSObject, AVAudioPlayerDelegate {

    // add these to store the image we are using, and the sound recording/playback tool
    var image: UIImage!
    var recordingURL: NSURL!
    var player: AVAudioPlayer!

    // a special variable that is used to retrieve the one instance of our StateModel
    static let sharedInstance = StateModel()

}
```

Next we will update the code in other methods; first we will add the code that handles making our class a singleton:

```
// We can still have a regular init method,
//    that will get called the first time the Singleton is used.
override private init() {
    super.init()

    let session = AVAudioSession.sharedInstance()
    do {
        try session.setCategory(AVAudioSessionCategoryPlayAndRecord)
    }
    catch {
        print("Failed to initialize session.")
    }
}
```

Next we want to include a custom setter method. This method is called whenever someone changes the value of one of our our properties:

```
// This is a setter to set the URL for the player
func setRecURL(recordingURL: NSURL) {
    self.recordingURL = recordingURL
    do {
        self.player = try AVAudioPlayer(contentsOfURL: recordingURL)
    }
    catch {
        print("Failed to find the recording!")
    }
    self.player.delegate = self
}
```

"How does this singleton thing work?" you ask. That's a great question! The idea of a singleton is that it is an object where we only want to have a single instance of it in existence. "Why would we want that?", you ask. You are great with these questions! "Why am I great with questions?", you might ask. "Stop it :-|", I would reply. As for why we want a single instance, think about the speakers on a phone, there is only one hardware controller for the speakers, so if we have lots of objects fighting for control of the hardware then we have a recipe for disaster. Alternatively we might have some data that our entire application would like to share (like a recording, or an image), and we don't want every component of our system to talk to every other component (what happens if we want to remove one component later?). Instead all of the components can talk to one central component, and then in the future if we don't want to have the camera we can simply remove the camera.

Updating the Camera View Controller

Next we will need to update the Camera View Controller. Specifically we need to tell the `CameraViewController` file that there is a thing called a `StateModel`, and we would like to be able to communicate with it. Since we are using Swift, then our `CameraViewController` already knows that this `StateModel` exists!

Then to use the `StateModel` we will just set the image that the `StateModel` is holding every time we change the image that we are holding. Modify the `imagePickerController:didFinishPickingMediaWithInfo:` method:

```
// Call this function when an image picker operation is done.
func imagePickerController(picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    let chosenImage = info[UIImagePickerControllerEditedImage] as? UIImage
    self.imageView.image = chosenImage
    picker.dismissViewControllerAnimated(true, completion: nil)

    // State model update.
    let model = StateModel.sharedInstance
    model.image = chosenImage
    print(model.image)
}
```

Updating the Sound Recorder View Controller

In the SoundRecorderViewController file we will also need to inform the StateModel about new recordings. We can do that using the delegate method audioRecorderDidFinishRecording:successfully::

```
func audioRecorderDidFinishRecording(recorder: AVAudioRecorder, successfully flag: Bool) {
    self.recordPauseButton.setTitle("Record", forState: .Normal)
    self.playButton.enabled = true
    self.stopButton.enabled = false

    // State model update.
    let model = StateModel.sharedInstance
    model.setRecURL(self.getAudioURL())
}
```

Updating the Game View Controller

Just setting this information doesn't make any change that is noticeable by the user. To see a change in the game performance we will need to use this new information!

We will need to make a few changes in AccelerometerGameViewController. First we are going to ensure that any time we are about to appear on screen we update to the most recent image. Update the viewWillAppear method with the following code:

```
override func viewWillAppear(animated: Bool) {

    super.viewWillAppear(animated)
    self.navigationController?.setNavigationBarHidden(true, animated: true)

    //load up our image
    let model = StateModel.sharedInstance
    if model.image != nil {
        self.imageButton.setImage(model.image, forState: .Normal)
    }

    //create an accelerometer manager
    self.motionManager = CMMotionManager()
    self.motionManager.accelerometerUpdateInterval = 0.1
    self.motionManager.startAccelerometerUpdates()

    //reset velocity to nil so our image starts out at a normal speed
    self.xVelocity = 0
    self.yVelocity = 0

    //attach the timer
    self.accelerometerUpdate = NSTimer.scheduledTimerWithTimeInterval(0.1,
        target: self, selector: #selector(update), userInfo: nil, repeats: true)

    //set up the default audio player
    let audioFilePath = NSBundle.mainBundle().pathForResource("Boing", ofType: "wav")
    if audioFilePath != nil {
        let audioFileUrl = NSURL.fileURLWithPath(audioFilePath!)
        do {
            audioPlayer = try AVAudioPlayer(contentsOfURL: audioFileUrl)
            audioPlayer.prepareToPlay()
        }
        catch {
            print("can't create audio player")
        }
    }
}
```

We also want to make sure that we pause any time the game scene isn't up. The easiest way to do this is to simply remove the timer when we are leaving the main screen. In the `AccelerometerGameViewController` file let's hook into the lifecycle method named `viewWillDisappear`:

```
override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated)
    self.accelerometerUpdate.invalidate()
}
```

This code will check if the `StateModel` has an image, if it doesn't have any image then we won't load from it. If the `StateModel` does have an image then we will replace our old image with whatever it is using. The final change is cause our recording to play whenever we bounce into a wall. Because we have our code designed so nicely we know exactly where we can put that code...into the `bumpedWall` method! We'll need to add a variable that can tell us if we bumped one of the walls in our `enforceBoundary` method:

```
func enforceBoundary() {

    var bump = false

    // find out where our image is trying to go
    let center = self.imageButton.center
    var destx = center.x
    var desty = center.y

    //how big is our image?
    let xRadius = self.imageButton.frame.size.width / 2
    let yRadius = self.imageButton.frame.size.height / 2

    //are we trying to leave the "left" side?
    if destx - xRadius < 0 {

        // make sure that the center of the image would be one radius distance
        // from the edge
        destx = self.imageButton.frame.size.width / 2

        //invert the velocity to simulate a bounce
        self.xVelocity = -self.xVelocity
        bump = true
    }

    if destx + xRadius > self.view.frame.size.width {

        destx = self.view.frame.size.width - xRadius
        self.xVelocity = -self.xVelocity
        bump = true
    }

    if desty - yRadius < 0 {

        desty = self.imageButton.frame.size.height / 2
```

```
            self.yVelocity = -self.yVelocity
            bump = true
        }

        if desty + yRadius > self.view.frame.size.height {

            desty = self.view.frame.size.height - yRadius
            self.yVelocity = -self.yVelocity
            bump = true
        }

        //reconstruct the image location and move the image to that point
        let dest = CGPointMake(destx, desty)
        self.imageButton.center = dest

        if(bump) {

            // when the sound is stored
            if let player = StateModel.sharedInstance.player {
                player.play()
            }
            // else play the dafault sound
            else {
                if audioPlayer != nil {
                    audioPlayer.play()
                }
            }
        }
    }
```

8.1.6 Polish and Swagger

Congratulations on finishing the activity! Now we can take pictures, make audio recordings, and even store integrate them into existing projects (we'll be reusing some of this code in the next activity!).

If you have time and want to build on this solution to do even more cool things here are a few ideas for activities:

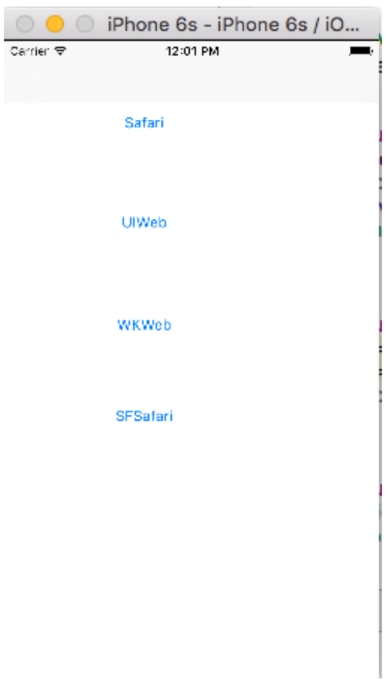
- By default the `UIImagePicker` uses the rear camera, can you switch it to use the front camera? You may want to look into the `cameraDevice` property (check the documentation)!
- If you analyze the microphone recording code you'll notice that we use a URL that refers to the file system on the phone. This recording is actually stored on the phone, and you can go to the url to extract the recording (you can use `print()` to print the url). The picture on the other hand is not saved, can you modify the code so that it is saved?
- We enable access to the saved photos album and to taking photos directly. There is a way to access the photo library, can you find it in the documentation?
- This one will take quite a bit of time, but can you replace the image with an animated gif? Perhaps a PSY, Gangnam Styling with gravity?

8.2 Web Access

8.2.1 Goals of this Lab!

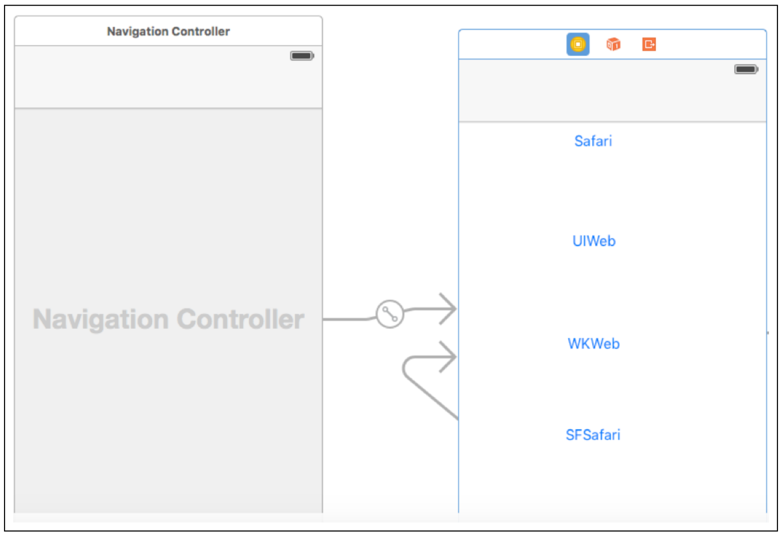
- Accessing the web with `WKWebView`, `UIWebView`, `Mobile Safari`, and `SafariWebViewController`.

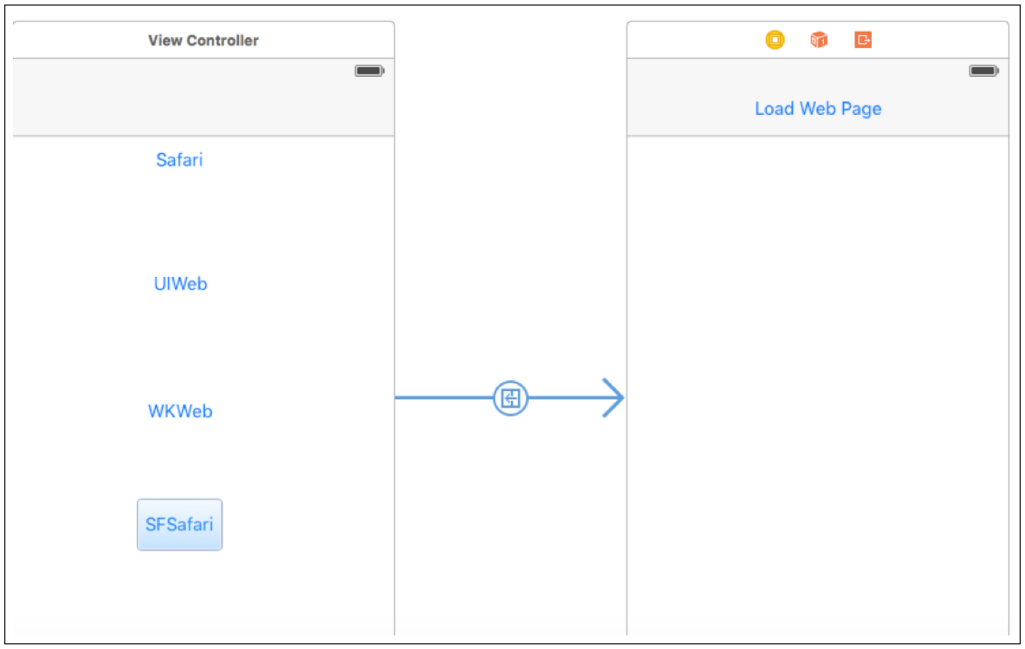
The end result should look something like this:

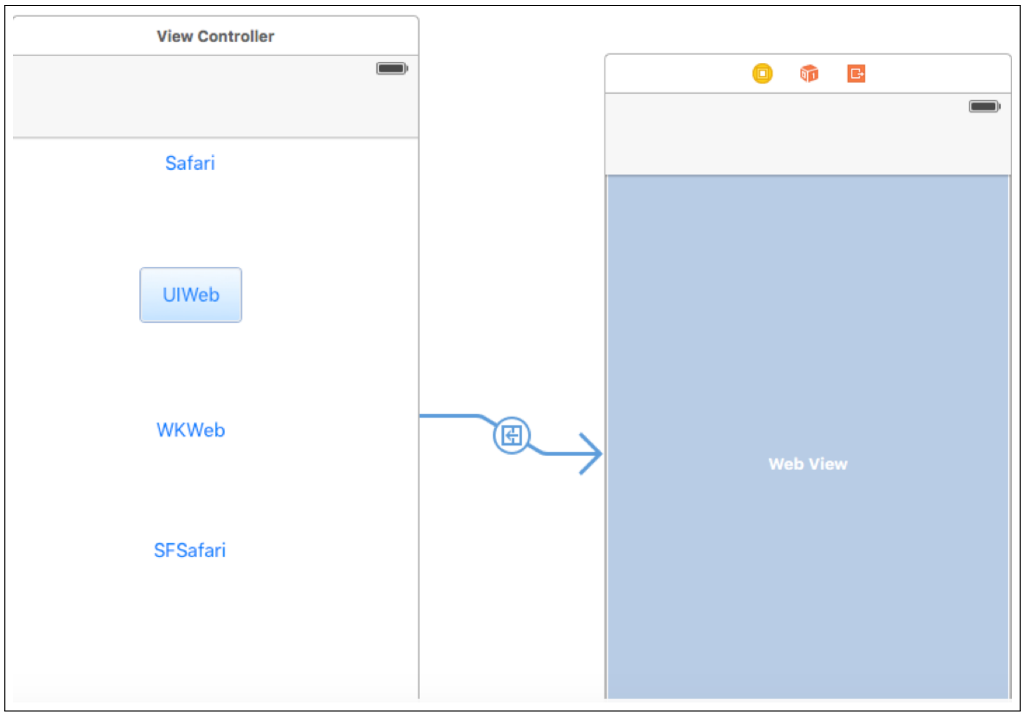


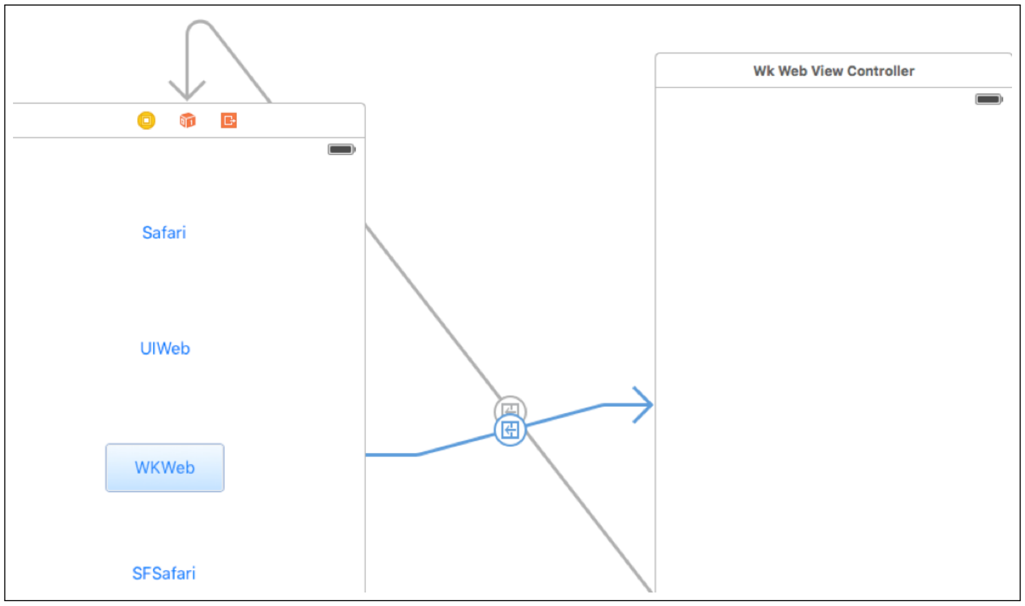
8.2.2 Getting started

Create a project called `WebLab` and then design the UI similar to the screenshot shown below. Create a segue from each button to appropriate view as shown, which is also shown below.




View Controller
Safari
UIWeb
WKWeb
SFSafari
Load Web Page


View Controller
Safari
UIWeb
WKWeb
SFSafari
Web View



8.2.3 Creating Files

Create three Swift files:

- `SafariWebViewController`,
- `WKWebViewController`,
- `UIWebViewController`.

For each of the segues to new view controllers you created above, change each of their classes to be one of the three classes you just created (as you have done before). Now we will implement each of these files.

8.2.4 `SafariWebViewController`

To use `SafariWebViewController`, we need to import the necessary code:

```
import SafariServices
```

We also need to make sure that we mark ourselves as the delegate for handling `SFSafariViewController`:

```
class SafariWebViewController: UIViewController, SFSafariViewControllerDelegate {
```

As in one of the earlier images, create an `IBAction` to the class called `load`, and have the sender's type be renamed to `UIButton`. We want to load a given URL (either hard-coded into the application, or supplied by the user). Add the following code to the `load` method:

```
@IBAction func load(sender: UIButton) {
    let urlString = "https://www.hackingwithswift.com"
    if let url = NSURL(string: urlString) {
        let vc = SFSafariViewController(URL: url)
        vc.delegate = self
        presentViewController(vc, animated: true, completion: nil)
    }
}
```

We also want to dismiss this view controller whenever the user wants to be done with it:

```
func safariViewControllerDidFinish(controller: SFSafariViewController) {
    dismissViewControllerAnimated(true, completion: nil)
}
```

Now we want to make sure that if the page does not load for whatever reason (either the network is down, or the user tapped cancel before the page loaded, or the page sent bad contents), that we dismiss this view controller:

```
func safariViewController(controller: SFSafariViewController,
    didCompleteInitialLoad didLoadSuccessfully: Bool) {
    if didLoadSuccessfully == false {
        print("Page did not load!")
        controller.dismissViewControllerAnimated(true, completion: nil)
    }
}
```

8.2.5 WKWebViewController

For those who don't know, the `WK` in the name refers to `WebKit`, which is the main engine behind the Safari browser on both iOS and Mac platforms. To use `WKWebViewController`, we need to import the necessary code:

```
import WebKit
```

Drag out two `Bar Button Items` into this view controller (preferably at the top where they can be placed and are easily accessible), and create `IBActions` for them, which correspond to "back" and "forward" as in any other browser (we will name them `back` and `forward`). We will have an instance of `WKWebView` which is a subclass of `UIView`:

```
var webView: WKWebView?
```

On loading the view controller, we want to set our own `view` to be `webView` in `viewDidLoad`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.webView = WKWebView()
    self.view = self.webView!
}
```

We only want to load the relevant webpage when the view has finished appearing, so add the following code to `viewDidAppear` (you might have to add the method declaration yourself):

```
override func viewDidAppear(animated: Bool) {
    var url = NSURL(string:"http://google.com/")
    var req = NSURLRequest(URL:url!)
    self.webView!.loadRequest(req)
}
```

Now we just call the relevant methods for going back and forward in the web browser's history:

```
@IBAction func forward(sender: UIBarButtonItem) {
    webView?.goForward();
}
@IBAction func back(sender: UIBarButtonItem) {
    webView?.goBack()
}
```

8.2.6 UIWebViewController

For `UIWebViewController`, we will be using something called an `UIActivityIndicatorView`, which is awesome! It shows when the network is being used, and we can turn it on or off depending on what is happening! We will use it to show when the webpage is loading, and dismiss it when it is done loading.

Drag out an `UIActivityIndicatorView`, a `UIWebView`, and a `UITextField`, which will be set as `IBOutlets` in the backing code. The purpose of the text field is for the user to input the URL of his/her favorite webpage. Also, add a button somewhere named `load` (as an IBAction) that will load the page inside the url:

```
@IBOutlet weak var activityProgress: UIActivityIndicatorView!
@IBOutlet weak var webView: UIWebView!
@IBOutlet weak var url: UITextField!
```

We want to say that we are the delegate of handling the web view (as before):

```
class UIWebViewController: UIViewController, UIWebViewDelegate {
```

On loading the view controller, we want to notify that we are the delegate for handling loading, etc.; also, it would be nice if the user could scroll on the web page, and that we can scale the page to fit the screen (instead of being too zoomed out or in):

```
override func viewDidLoad() {
     super.viewDidLoad()
     self.webView.delegate = self
     self.webView.scrollView.scrollEnabled = true
     self.webView.scalesPageToFit = true
}
```

When we want to load the page (when the button is pressed), we want to initiate an `NSURLRequest` which will fetch the contents of the page. We also want to start the `UIActivityIndicatorView` that shows to the user when the page is being loaded:

```
@IBAction func load(sender: UIButton) {
    let webpage: String = self.url.text!

    let url = NSURL(string: webpage)
    let request = NSURLRequest(URL: url!)

    activityProgress.hidesWhenStopped = true
    activityProgress.startAnimating()
    webView.loadRequest(request)
}
```

As always, you should add a few buttons that will refresh the page, go back or forward, or stop loading. We connect them as `IBActions` and link them to the backing code as follows:

```
@IBAction func refresh(sender: UIButton) {
    webView.reload()
}

@IBAction func back(sender: UIButton) {
    webView.goBack()
}

@IBAction func forward(sender: UIButton) {
    webView.goForward()
}
```

```
@IBAction func stop(sender: UIButton) {
    webView.stopLoading()
}
```

Now the final thing we want is that the indicator view still keeps animating even though the page has finished loading! We need to stop it by calling `stopAnimating` when the page finishes loading:

```
func webViewDidFinishLoad(webView: UIWebView) {
    activityProgress.stopAnimating()
}
```

And that's it!

8.2.7 Safari

What about that `Safari` button? We can actually load the Safari browser directly from our application! The way to do this is with `UIApplication` and calling `sharedApplication`, which is essentially a system-wide library for you to use (your app cannot call other non-native apps).

```
@IBAction func openSafari(sender: UIButton) {
    let url = NSURL(string: "http://www.google.com")

    UIApplication.sharedApplication().openURL(url!)
}
```

8.2.8 Polish and Swagger

At this point the application is done, congratulations! You've implemented some really cool features in SpriteKit! If you've got some extra time here are some things that might be interesting to explore on your own:

- Can you store the favorites of a user, or have a user add his/her own favorite sites?
- What other applications/libraries that are system-wide can you call? (Hint: check the documentation!)
- Which one of the methods of accessing web pages was the quickest? Why do you think so?
- What is the advantage or disadvantage of each of the methods?