VirginiaTech Invent the Future CS3714 iOS Mobile Software Development

Home | Course | Policies | Staff | Grading | Other

Tutorial: Temperature

Learning Objectives

1. How to create a User Interface (UI) with label, image view, and slider objects.
2. How to apply Design Patterns: Model-View-Controller (MVC), Delegation, and Target-Action.
3. How to reposition and resize UI objects under different device orientations and screen sizes.
4. iOS Application Life Cycle.
5. Introduction to Swift Programming Language.

Table of Contents

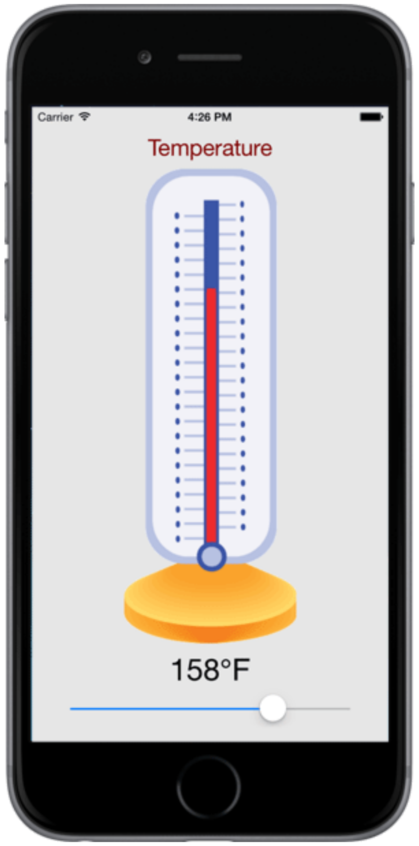
- Step 0: Earlier Phases of the Software Life Cycle
- Step 1: Application Functionality Specification
- Step 2: Application Design Specification based on the MVC Design Pattern
- Step 3: Creation of a New Project
- Step 4: The iOS Application Life Cycle
- Step 5: Delegation Design Pattern: AppDelegate
- Step 6: MVC Design Pattern: View – User Interface Development
- Step 7: Auto Layout of the UI Objects
- Step 8: Target-Action Design Pattern
- Step 9: MVC Design Pattern: Controller
- Step 10: Build and Run your application

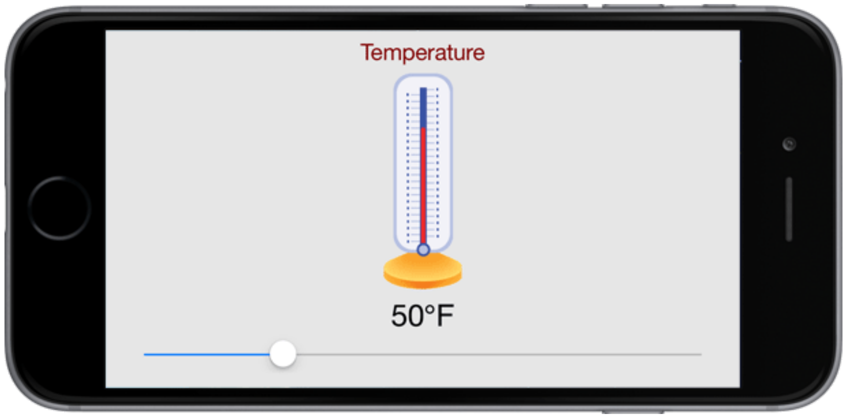
Step 0: Earlier Phases of the Software Life Cycle

We assume that earlier phases of the Software Life Cycle have been completed. See Dr. Balci's Software Life Cycle. The life cycle processes *Problem Formulation*, *Requirements Engineering*, *Architecting*, and *Design* are assumed to be completed. In this tutorial, we focus on the *Programming* process.

Step 1: Application Functionality Specification

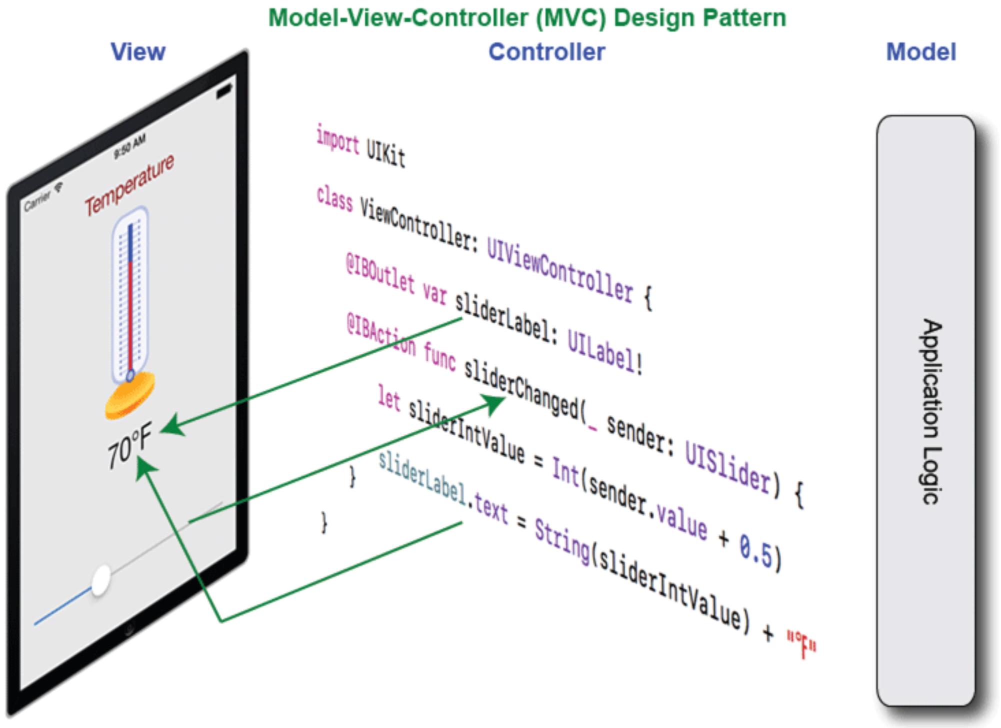
This is a **universal** iOS app that runs on all iOS devices for all screen sizes and resolutions under both **portrait** and **landscape** device orientations. The app displays a title **Temperature** and a thermometer image as shown below. The user taps the slider knob and moves it to the left or right to display an integer number between 0 (leftmost) and 212 (rightmost) above the slider as shown. The initial displayed value is set to 70. The user interface objects are repositioned and resized upon device's rotation to landscape orientation as shown on right. Note that, in landscape orientation, the thermometer image is shrunk to fit the vertical space and the slider object is stretched to the available horizontal space.





Step 2: Application Design Specification based on the MVC Design Pattern

iOS applications are commonly designed using the following Design Patterns: (1) **Model-View-Controller (MVC)**, (2) **Delegation**, and (3) **Target-Action**. Based on the desired functionality described above, we have the following MVC-based design specification for our application.



```
import UIKit
class ViewController: UIViewController {
  @IBOutlet var sliderLabel: UILabel!
  @IBAction func sliderChanged(_ sender: UISlider) {
    let sliderIntValue = Int(sender.value + 0.5)
    sliderLabel.text = String(sliderIntValue) + "°F"
  }
}
```

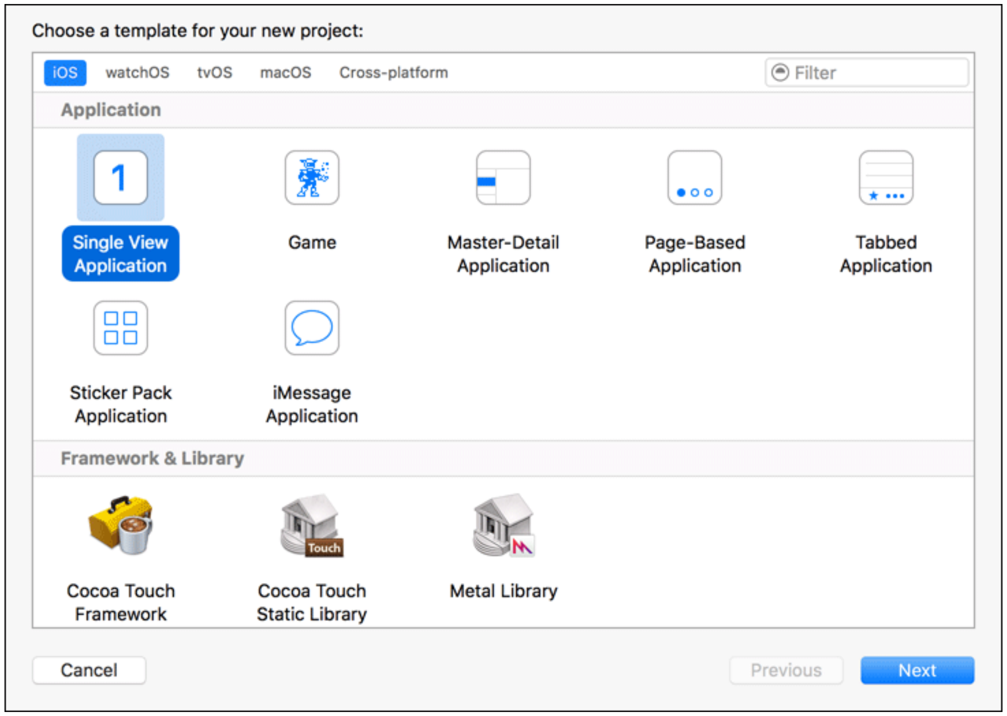
Our View Controller class, `ViewController`, inherits from the `UIViewController` class. By importing with `import UIKit`, we make all `UIKit` framework classes as available for use. See the UIKit Framework Classes Hierarchy and identify the `UIViewController`, `UILabel, UIImageView, and UISlider` classes used in our application.

- Every time the slider knob is moved by the user, the slider value is changed.
- Every time the slider value is changed, the **sliderChanged** method is invoked.
- Every time the **sliderChanged** method is invoked, the slider's new value is assigned to the slider label displaying the temperature value in Fahrenheit.

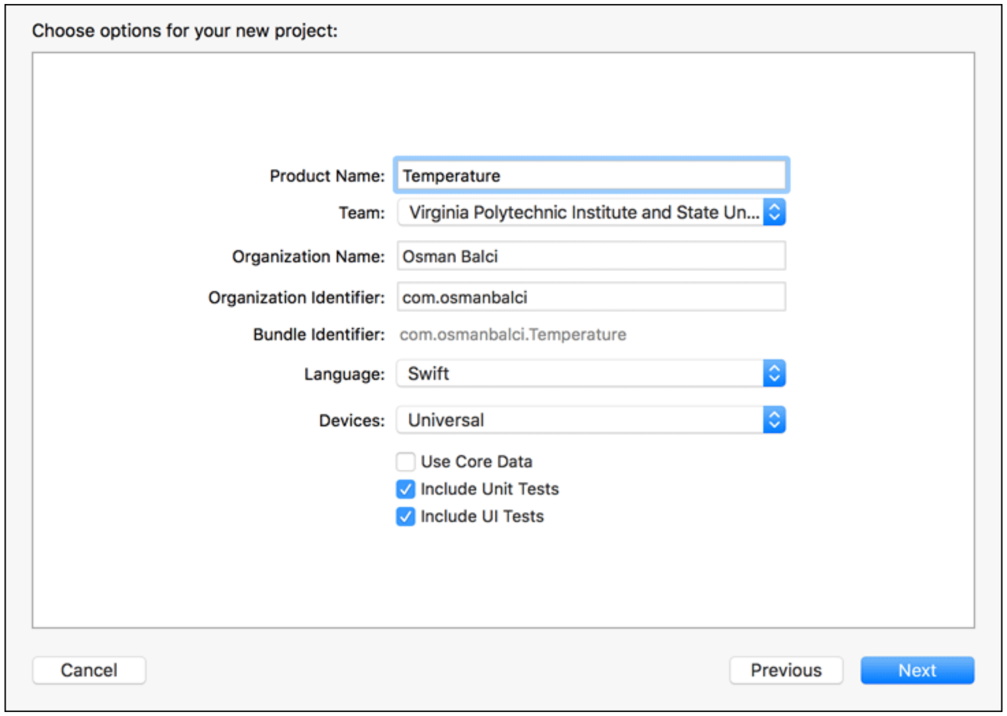
Step 3: Creation of a New Project

Project File Creation

1. Launch **Xcode 8.0** latest beta version.
2. Create a new project by selecting **File → New → Project...**
3. In the *Choose a template for your new project* dialog, select **iOS → Application → Single View Application** template as shown below. Click Next.



4. In the *Choose options for your new project* dialog as shown below:


Choose options for your new project:

Product Name: Temperature
Team: Virginia Polytechnic Institute and State Un...
Organization Name: Osman Balci
Organization Identifier: com.osmanbalci
Bundle Identifier: com.osmanbalci.Temperature
Language: Swift
Devices: Universal
☐ Use Core Data
☑ Include Unit Tests
☑ Include UI Tests

- Enter Product Name as **Temperature**.
- Select your **iOS Developer University Program** membership of the Team named **Virginia Polytechnic Institute and State Universit (Information Technology Acquisitions)**.
- Enter **Your Name** as the Organization Name to show that you are the developer of this app.
- Enter `com.yourname` as your company/organization unique identifier.
- The string `com.yourname.Temperature` becomes the Bundle ID for your application uniquely identifying it in the App Store for distribution.
- Select **Swift** as the programming language to use.
- Select **Universal** for the Devices, implying that your app will run on all iOS devices, i.e., iPhone, iPad, and iPod Touch. Click Next.
- In the File Browser window displayed,
  1. Select a location on your hard disk to store your project.
  2. **Uncheck Source Control** to disable version control. Click **Create** to create your project.

General Project Settings

1. Click the project name **Temperature** under the Project Navigator and select **Temperature** under **Targets** to display the **General** project settings.
2. Change **Deployment Target** to 9.0 so that devices running iOS version 9.0 and above can run the app.

Copy the Resources Needed into your Project

1. Download the resource files needed for the project: Temperature_files.zip.
2. Uncompress it.
3. Click `Assets.xcassets` under the Project Navigator to display the Assets Pane.
4. Drag and drop the following downloaded folders into the Assets Pane.

   - App Icons
   - LaunchImage.imageset
   - Thermometer.imageset

Follow the Instructions in the Previous HelloWorld Tutorial to Perform the Following

- Assigning App Icons in Imagesets
- Assigning Launch Images in Imageset

- Assigning Thermometer Images in Imageset (similar to above)
- Assigning App Icons from the Media Library
- Setting the Launch Image

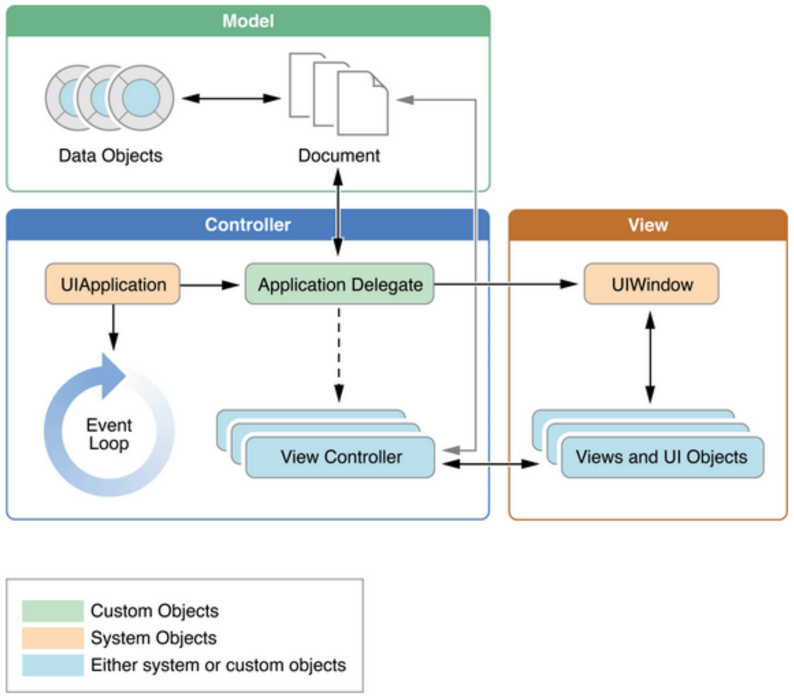
Step 4: The iOS Application Life Cycle

Please read **The App Life Cycle** in the App Programming Guide for iOS for further information. The following is taken from this source.

The Structure of an App

"During startup, the UIApplicationMain function sets up several key objects and starts the app running. At the heart of every iOS app is the UIApplication object, whose job is to facilitate the interactions between the system and other objects in the app. Figure 2-1 shows the objects commonly found in most apps.... iOS apps use a **model-view-controller** ~~architecture~~ design pattern. This pattern separates the app's data and business logic from the visual presentation of that data. This ~~architecture~~ design pattern is crucial to creating apps that can run on different devices with different screen sizes."

Figure 2-1 Key objects in an iOS app

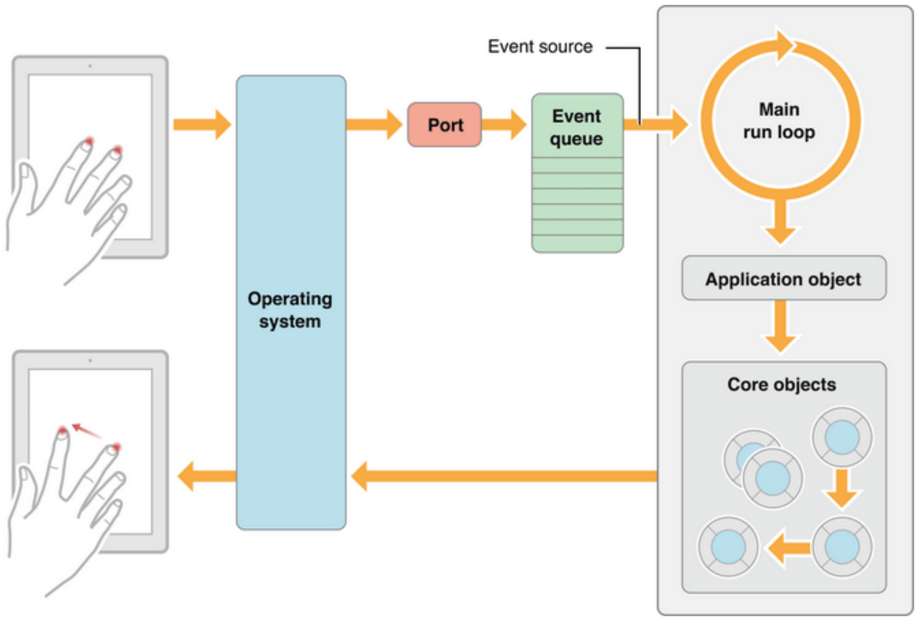


The Main Run Loop

"An app's **main run loop** processes all user-related events. The UIApplication object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces. As the name suggests, the main run loop executes on the app's main thread. This behavior ensures that user-related events are processed serially in the order in which they were received.

Figure 2-2 shows the architecture of the main run loop and how user events result in actions taken by your app. As the user interacts with a device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit. Events are queued internally by the app and dispatched one-by-one to the main run loop for execution. The UIApplication object is the first object to receive the event and make the decision about what needs to be done. A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred. Other events might take slightly different paths through various app objects."

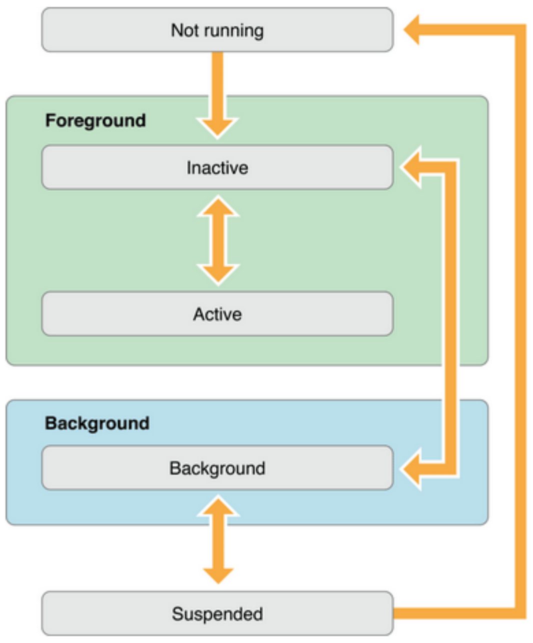
Figure 2-2 Processing events in the main run loop



Execution States for Apps

“At any given moment, your app is in one of the states listed in Table 2-3. The system moves your app from state to state in response to actions happening throughout the system. For example, when the user presses the Home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response. Figure 2-3 shows the paths that an app takes when moving from state to state.”

Figure 2-3 State changes in an iOS app



Step 5: Delegation Design Pattern: AppDelegate

The `UIApplication` is a class contained in the **UIKit framework** under the **Cocoa Touch Layer**. Go to the iOS Developer Library. Expand the **Cocoa Touch Layer** and click **UIKit** to display the list of classess contained in it.

The `UIApplication` is a gigantic class like an elephant. We do **not** want to subclass `UIApplication` and create another elephant just to override some of its methods. Doing so would be very costly for our limited resources. Instead, an object is designated as a delegate, called **AppDelegate**, representing the elephant. The **AppDelegate** object is called the **Delegate Object** and this design approach is called the **Delegation Design Pattern**.

The **AppDelegate** object is created to adopt the **UIApplicationDelegate** protocol. “The UIApplicationDelegate protocol defines methods that are called by the singleton `UIApplication` object in response to important events in the lifetime of an iOS app. The app delegate works alongside the app object to ensure your app interacts properly with the system and with other apps. Specifically, the methods of the app delegate give you a chance to respond to important changes.”

The `AppDelegate.swift` class file provided below is automatically created:

```
//
//  AppDelegate.swift
//  Temperature
//
//  Created by Osman Balci on 8/10/16.
//  Copyright © 2016 Osman Balci. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }
    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is terminated later.
        // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        // Called as part of the transition from the background to the active state; here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
    }

}
```

The statements of the `AppDelegate.swift` class file are explained below.

```
import UIKit
```

The import declaration (compiler directive) imports the UIKit framework so that all of the classes in the UIKit framework are accessible in this `AppDelegate` class.

```
@UIApplicationMain
```

The @UIApplicationMain attribute (annotation) defines the starting point for your application.

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

Declares a class called AppDelegate as inheriting from the UIResponder class and as conforming to the UIApplicationDelegate protocol. Once you conform to a protocol, you have the right to implement its methods.

```
var window: UIWindow?
```

"The UIWindow class defines an object known as a window that manages and coordinates the views an app displays on a device screen. Unless an app can display content on an external device screen, an app has only one window."

This statement declares an instance **var**iable (property), called window, as an object reference with **Optional** value pointing to an object instantiated from the UIWindow class. If the instantiation fails, no UIWindow object is created, and the window object reference points to `nil`. Swift does not allow assigning a `nil` value to a variable and triggers a compiler error. Therefore, we write a **question mark** after the type of the value to direct the compiler to **wrap up** the object reference value into a so called **Optional** value.

Optionals must be used in those situations where a value may be absent. "An optional value either contains a value or contains `nil` to indicate that the value is missing. **Write a question mark (?) after the type of a value to mark the value as optional.**" [Swift]

"You can use an `if` statement to find out whether an optional contains a value. If an optional does have a value, it evaluates to `true`; if it has no value at all, it evaluates to `false`. Once you're sure that the optional does contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional's name. The exclamation mark effectively says, *I know that this optional definitely has a value; please use it*. This is known as **forced unwrapping** of the optional's value." [Swift]

A method or a function contains procedural code. You use the func keyword to designate the code as either a method or a function. Read about **Functions** and **Methods** in The Swift Programming Language (Swift 3).

"Each function parameter has both an **argument label** and a **parameter name**. The argument label is used when calling the function; each argument is written in the function call with its argument label before it. The parameter name is used in the implementation of the function. By default, parameters use their parameter name as their argument label. ... **If you don't want an argument label for a parameter, write an underscore (_) instead of an explicit argument label for that parameter.**" [The Swift Programming Language (Swift 3)]

The following defines a function (protocol method) called "application" with two parameters: The first one's **argument label** is suppressed by using an underscore and its **parameter name** is "application". The second one's **argument label** is "didFinishLaunchingWithOptions" and its **parameter name** is "launchOptions". The function returns a value of type Boolean.

```
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }
```

The first parameter with no argument label is called "application", which is an object reference pointing to an object instantiated from the UIApplication class. Your application is instantiated from the UIApplication class as a singleton object, meaning as the only object, and the object reference of the newly created UIApplication object is stored into the object reference named "application".

The second parameter of the "application" method is named "launchOptions" with argument label "didFinishLaunchingWithOptions". This parameter is an object reference with **Optional** value pointing to a Dictionary object with KEY of type UIApplicationLaunchOptionsKey and VALUE of type Any.

The -> Bool indicates that the method returns only one value of type **boolean**. Note that a method can return multiple values in Swift.

The following UIApplicationDelegate Protocol methods are listed as empty and are optional to implement. They are listed in the class file to guide the developer.

```
    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when
the user quits the application and it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is
terminated later.
        // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        // Called as part of the transition from the background to the active state; here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
    }
```

Step 6: MVC Design Pattern: View – User Interface Development

The **Interface Builder (IB)** is used to create your application's **User Interface (UI)**. It enables you to drag and drop UI objects from the Object Library and graphically connect some UI objects to instance variables and instance methods specified in your code. We use the `.storyboard` file to use the IB to create the UI.

What is a **Storyboard**? Dictionary definitions:

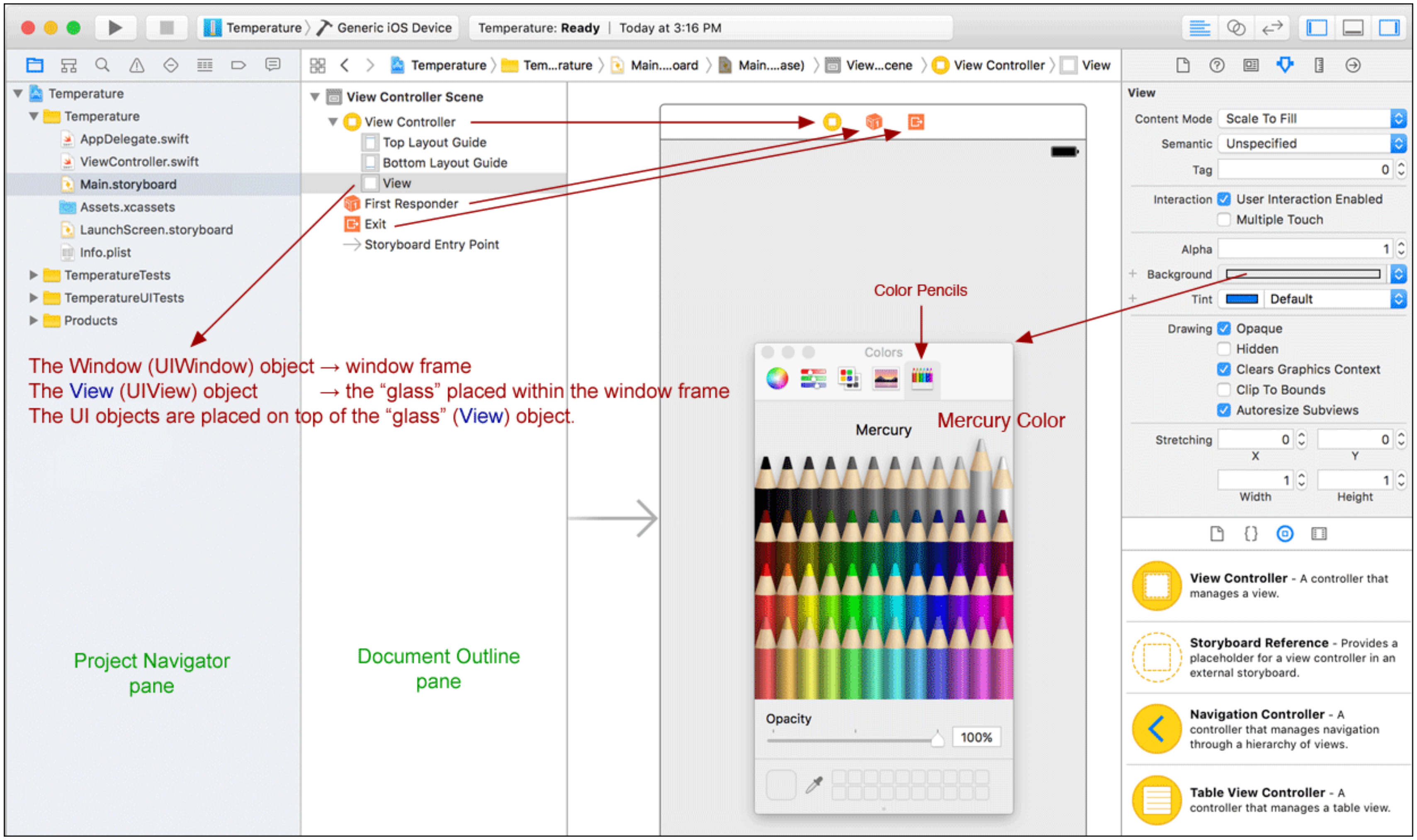
- “a panel or panels on which a sequence of sketches depict the significant changes of action and scene in a planned film, as for a movie, television show, or advertisement.”
- “a series of sketches or photographs showing the sequence of shots or images planned for a film”

We use the term **Storyboard** to refer to an iOS file containing

- an iOS app's navigation structure or logical flow (with horizontal and vertical decomposition of app's functionality),
- definitions of the **Views** (**Scenes**) making up the iOS app's UI,
- connection of a UI object to code for performing an action or defining an outlet,
- specification of a relationship between two Views (Scenes), and
- how a View (Scene) will segue into the next one.

Click the `Main.storyboard` file in the Project Navigator to display your app's only scene (view) as shown in the screenshot below.

1. Click inside the window frame or click the View (Scene) object in the Document Outline pane to select the **View** object instantiated from the UIView class.
2. Bring up the **Attributes inspector** as shown in the screenshot below.
3. Click the **Background** color well (rectangular area) to pop up the **Colors** window as shown below.
4. Select the **Pencils** option. Select the **Mercury** pencil as shown below to color the background with Mercury color.


Temperature
Generic iOS Device
Temperature: Ready | Today at 3:16 PM
Temperature > Tem...rature > Main....oard > Main....ase) > View...cene > View Controller > View
Temperature
Temperature
AppDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
TemperatureTests
TemperatureUITests
Products
View Controller Scene
View Controller
Top Layout Guide
Bottom Layout Guide
View
First Responder
Exit
Storyboard Entry Point
Colors
Mercury
Opacity
100%
Color Pencils
Mercury Color
Project Navigator pane
Document Outline pane
View
Width
Height
X
Y
View Controller - A controller that manages a view.
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Table View Controller - A controller that manages a table view.

The Window (UIWindow) object → window frame
The View (UIView) object → the “glass” placed within the window frame
The UI objects are placed on top of the “glass” (View) object.

Content 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 To Bounds, ✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1

It is important to understand the relationship between the Window and View objects. The Window object is instantiated from the UIWindow class and is referenced in the AppDelegate Swift file. "A UIWindow object provides the backdrop for your app's user interface and provides important event-handling behaviors." [UIKit Framework Reference] The Window object only determines the frame within which the UI is constructed. Inside the Window frame is empty. So, we place the View object instantiated from the UIView class within the Window frame so that we can build the UI on top of it. By anology, think of it as the Window object representing a picture frame, the View object representing a glass placed within the picture frame intended to hold the UI objects. Therefore, we always build our UI on top of the UIView (View) object.

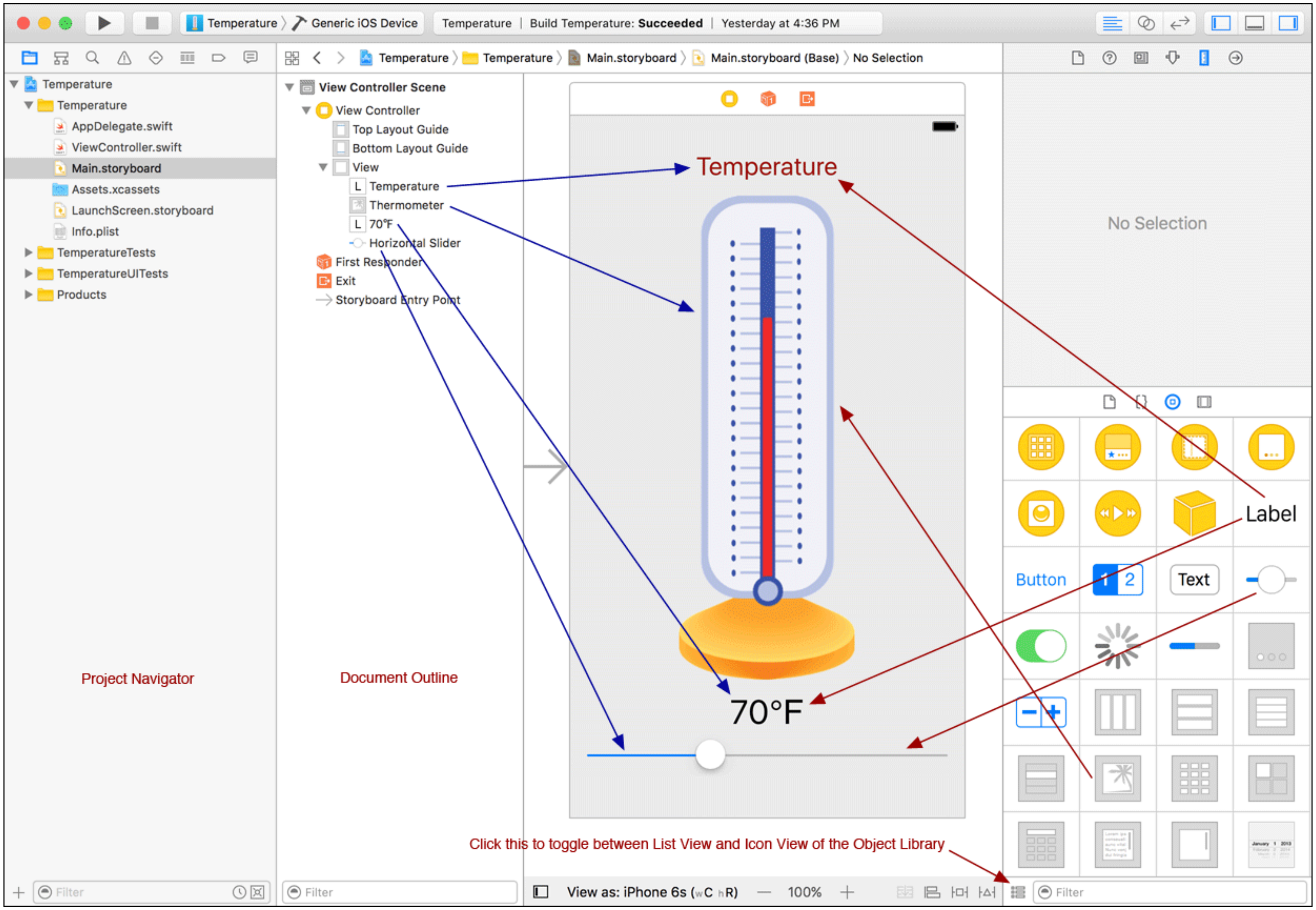
5. Drag a **Label** object from the Object library and drop it on the View canvas towards the top as shown in the screenshot below. This action corresponds to the instantiation (creation) of a **Label** object from the UILabel class at design time.

   - Select the label and bring up the Attributes inspector and make sure that "View as" is set to **iPhone 6S**.
     - change **Text** value from **Label** to **Temperature** and press the Return key for the change to be recorded.
     - change its **Color** to **Cayenne**
     - set its **Font** to **System 24.0**
     - set its **Alignment** to **Center**
   - Bring up the Size inspector and set its position and size to **x=87, y=28, w=201, and h=40**, with origin specified as the upper-left corner.

6. Drag and drop an **Image View** object from the Object library and position it on the View canvas right below the Temperature label as shown below. This action corresponds to the instantiation (creation) of an **Image View** object from the UIImageView class at design time.

While the newly created Image View object is selected and "View as" is set to **iPhone 6S**,

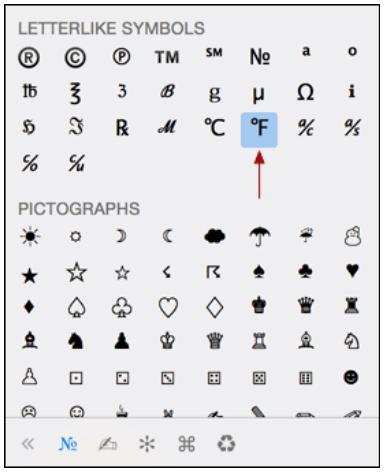
- show the Size Inspector and set **x=78, y=76, w=219, and h=460**, with origin specified as the upper-left corner.
- show the Attributes Inspector:
  - select `thermometer` from the Image pull-down menu
  - select **Aspect Fit** for the **View Content Mode** so that the image is not distorted.



7. Drag another **Label** object from the Object library and drop it on the View canvas right below the thermometer image as shown above. This action corresponds to the instantiation (creation) of a **Label** object from the UILabel class at design time.

   - Select the label and bring up the Attributes inspector and make sure that "View as" is set to **iPhone 6S**.

- enter the **Text** value as **70°F**. To enter **°F**, select **Edit → Emoji & Symbols**. Click **°F** from the **Letterlike Symbols** category as shown below.



- set its **Font** to **System 32.0**
- set its **Alignment** to **Center**

- Bring up the Size inspector and set **x=127, y=544, w=121, and h=45**, with origin specified as the upper-left corner.

8. Drag a **Slider** object from the Object library and drop it on the View canvas as shown in the screenshot above. This action corresponds to the instantiation (creation) of a **Slider** object from the UISlider class at design time.

   While the newly created Slider object is selected and "View as" is set to **iPhone 6S**,
   - Under the Attributes inspector, set its current **Value** to 70, **Minimum** value to 0, and **Maximum** value to 212.
   - Under the Size inspector, set **x=16, y=592, w=343, and h=31**, with origin specified as the upper-left corner.

Step 7: Auto Layout of the UI Objects

"Auto Layout defines your user interface using a series of constraints. Constraints typically represent a relationship between two views. Auto Layout then calculates the size and location of each view based on these constraints." [Auto Layout Guide]

We want all of the UI objects to be centered within the view in both portrait and landscape orientations in all screen sizes.

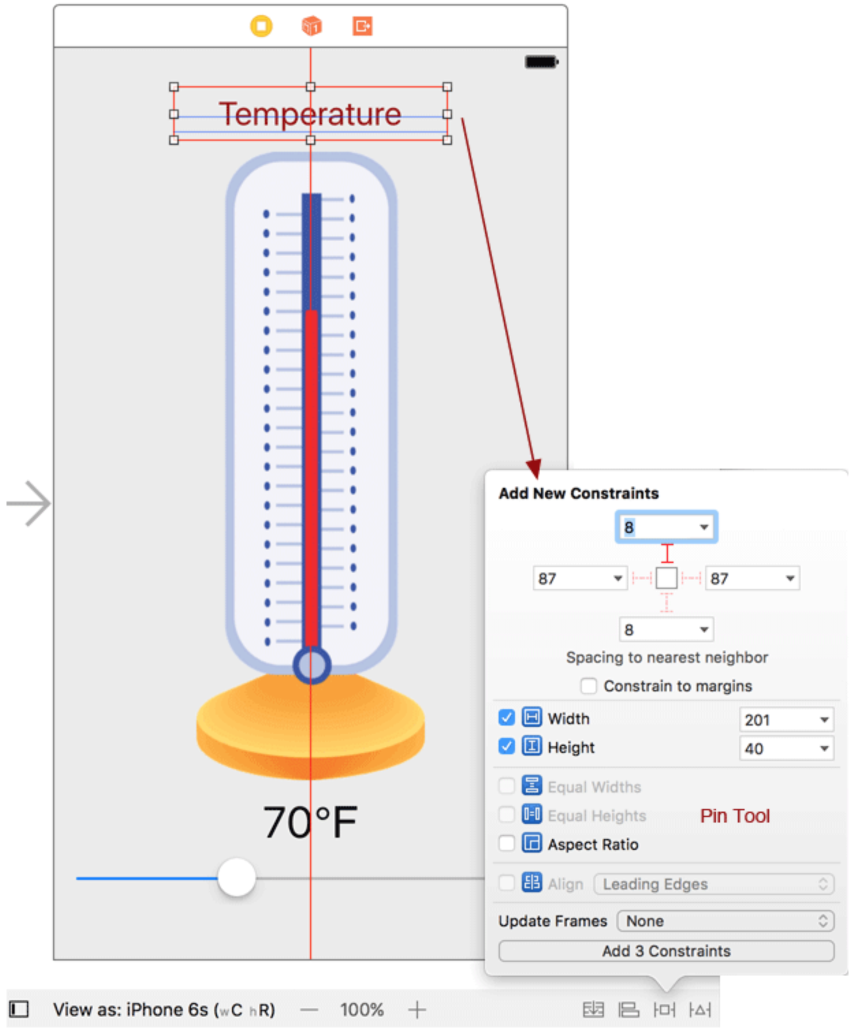
1. Within the View area, drag around all of the UI objects created to select them all. Or you can Command-click each UI object to select them all.
2. Click the **Align** tool to show it.
3. Select **Horizontally in Container**.
4. Click **Add 4 Constraints** to apply the constraints to the 4 UI objects.

Auto Layout of the Temperature Label Object

We want the Temperature heading label object to be positioned 8 pts below the Top Layout Guide in both portrait and landscape orientations and have fixed width and height.

1. Select the Temperature label object.
2. Click the **Pin** tool to show it.
3. In the *Add New Constraints* dialog, as shown in the following screenshot,

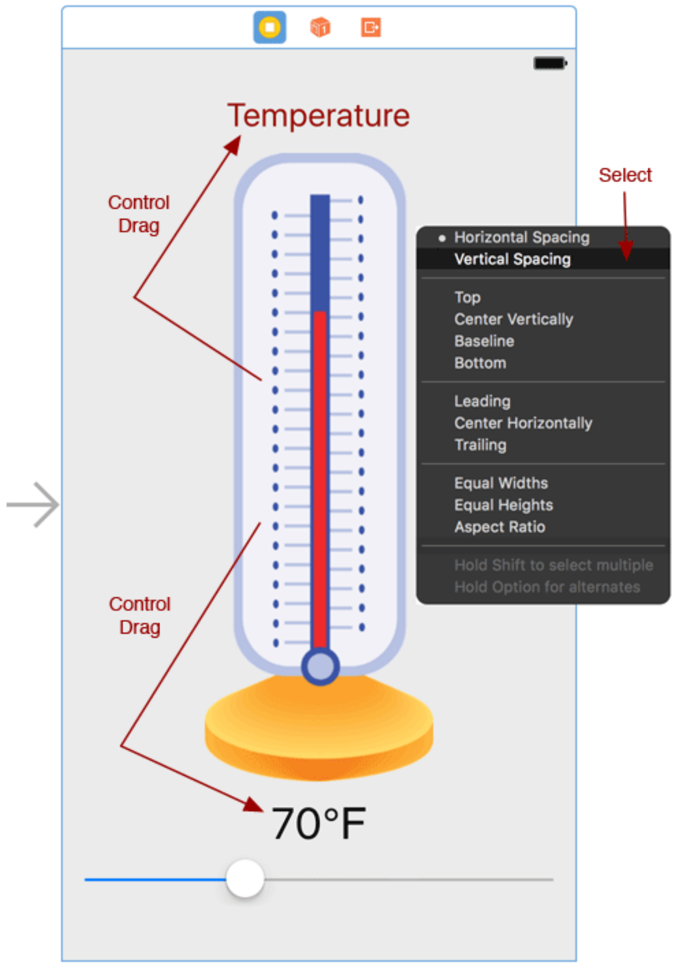
   - Uncheck *Constrain to margins*.
   - Click the top **I beam** to fix the top vertical space to 8 points as shown below.
   - Check the boxes for **Width** and **Height** to fix them to 201 and 40.
   - Click the **Add 3 Constraints** button to apply the constraints.



Auto Layout of the Thermometer Image View Object

The thermometer Image View object would be too high to fit within the landscape orientation. Therefore, we want it to vertically shrink to fit the available space.

1. Select the Thermometer Image View object.
2. **Control-drag** from it to the Temperature label object, let go the Control key and mouse, and select **Vertical Spacing** from the pop-up menu as shown in the screenshot below.
3. **Control-drag** from it to the **70°F** label object, let go the Control key and mouse, and select **Vertical Spacing** from the pop-up menu as shown in the screenshot below.

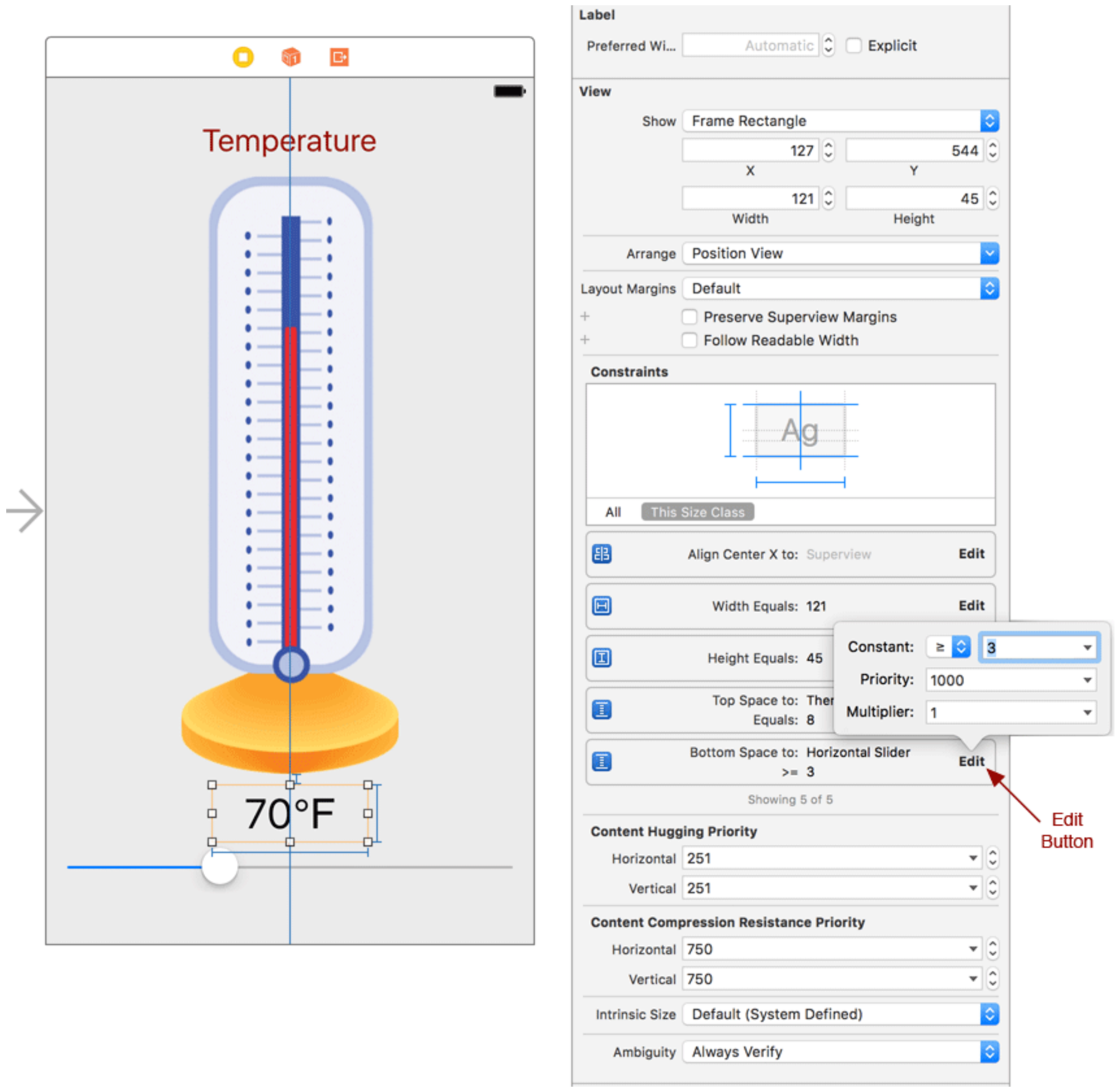


Auto Layout of the Temperature Value Display Label Object

1. Select the **70°F** label object as shown in the following screenshot.

2. Click the **Pin** tool to show it.
3. Check the boxes for **Width** and **Height** to fix them to 121 and 45.
4. Click the **Add 2 Constraints** button to apply the constraints.

5. **Control-drag** from the **70°F** label object to the **Horizontal Slider** object, let go the Control key and mouse, and select **Vertical Spacing** from the pop-up menu.
6. Bring up the **Size Inspector** and click the **Edit** button of the **Bottom Space to: Horizontal Slider** constraint to edit it.
7. In the pop-up menu, set the distance to be **Greater Than or Equal** to 3 pts as shown in the screenshot below. Press the Return key to record the change.



Label
Preferred Wi... Automatic Explicit

View
Show: Frame Rectangle
X: 127 Y: 544
Width: 121 Height: 45
Arrange: Position View
Layout Margins: Default
Preserve Superview Margins
Follow Readable Width

Constraints
All | This Size Class
Align Center X to: Superview — Edit
Width Equals: 121 — Edit
Height Equals: 45
Top Space to: The... Equals: 8
Bottom Space to: Horizontal Slider >= 3 — Edit
Showing 5 of 5

Constant: ≥ 3
Priority: 1000
Multiplier: 1

Content Hugging Priority
Horizontal: 251
Vertical: 251

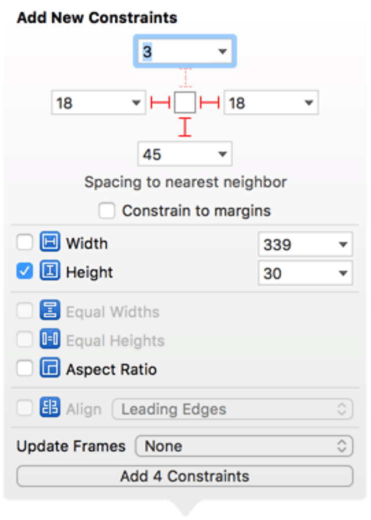
Content Compression Resistance Priority
Horizontal: 750
Vertical: 750

Intrinsic Size: Default (System Defined)
Ambiguity: Always Verify

Auto Layout of the Horizontal Slider Object

We want the Horizontal Slider object to **stretch** to the entire size of the screen by leaving 14 pts on the left and 14 pts on the right. We want its distance to the bottom edge of the screen to be fixed to 20 pts.

1. Select the Horizontal Slider object object.
2. Click the **Pin** tool to pop up the *Add New Constraints* dialog as shown in the following screenshot.
3. Uncheck *Constrain to margins*.
4. Click the left and right I-beams to fix the left and right distances to 18 pts each. This means that 18 pts will be left on each side of the slider and the slider image will be stretched alongside the screen of the iOS device.
5. Click the bottom I-beam to set the bottom distance to 45 pts as fixed distance.
6. Check the box for **Height** to fix it to 30.
7. Click the **Add 4 Constraints** button to apply the constraints.


Add New Constraints

3

18 — 18

45

Spacing to nearest neighbor

☐ Constrain to margins

☐ Width 339

✓ Height 30

☐ Equal Widths

☐ Equal Heights

☐ Aspect Ratio

☐ Align Leading Edges

Update Frames None

Add 4 Constraints

Incremental Development: Run your app under the iOS Simulator to test the auto layout constraints for all iOS devices

1. Run your app under the iOS Simulator for many different iOS devices, iPhone, iPad, and iPod Touch.
2. Under the **iOS Simulator**, select **Hardware → Rotate Left** or **Rotate Right** or press **Command-Left Arrow** or **Command-Right Arrow** repeatedly to see how auto layout works under different device orientations and screen sizes.

Step 8: Target-Action Design Pattern

Under the Target-Action design pattern, we graphically connect the **Slider Label** (70°F) and **Slider** objects to our `ViewController.swift` code.

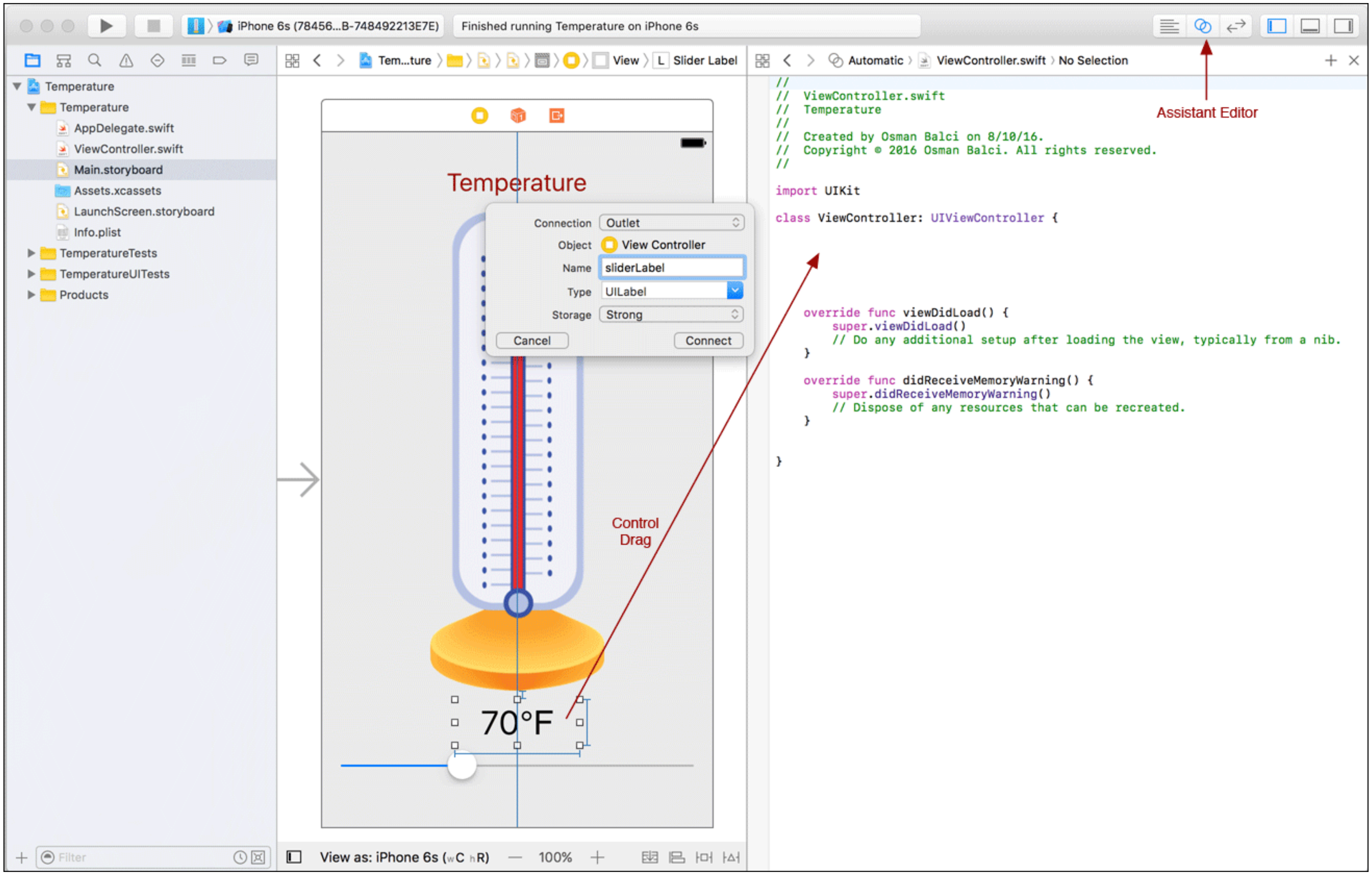
1. Click the `Main.storyboard` file in the Project Navigator to show it.
2. Click the **Assistant Editor** to show the `ViewController.swift` file on the right of the storyboard.
3. Control-drag from the **Slider Label** (70°F) object to the location in the `ViewController.swift` file as shown below. When you let go the mouse button, a dialog box shows up.

4. Select **Outlet** for the Connection.
5. Enter `sliderLabel` as the name of the connection.
6. Set class (Type) to **UILabel**.
7. Select **Strong** for Storage to direct the compiler to retain the value during the entire execution of the app.
8. Click **Connect** to insert the following into the code:

```
@IBOutlet var sliderLabel: UILabel!
```

Read the statement above as follows:

- Create a property (instance **var**iable), called `sliderLabel`, as an object reference pointing to the object instantiated from the UILabel class in the **Interface Builder (IB)**.
- Create the **Getter** and **Setter** methods for the `sliderLabel` property. (This is done behind the scenes internally in Xcode.)
- The @IBOutlet IB attribute indicates that this property will be visible as an **Outlet** in the **IB,** that is, in the **Storyboard** file. Note that the @IBOutlet attribute is not part of the Swift programming language. **An IB attribute is a declaration attribute used by the IB to synchronize with Xcode.**



Connection: Outlet
Object: View Controller
Name: sliderLabel
Type: UILabel
Storage: Strong

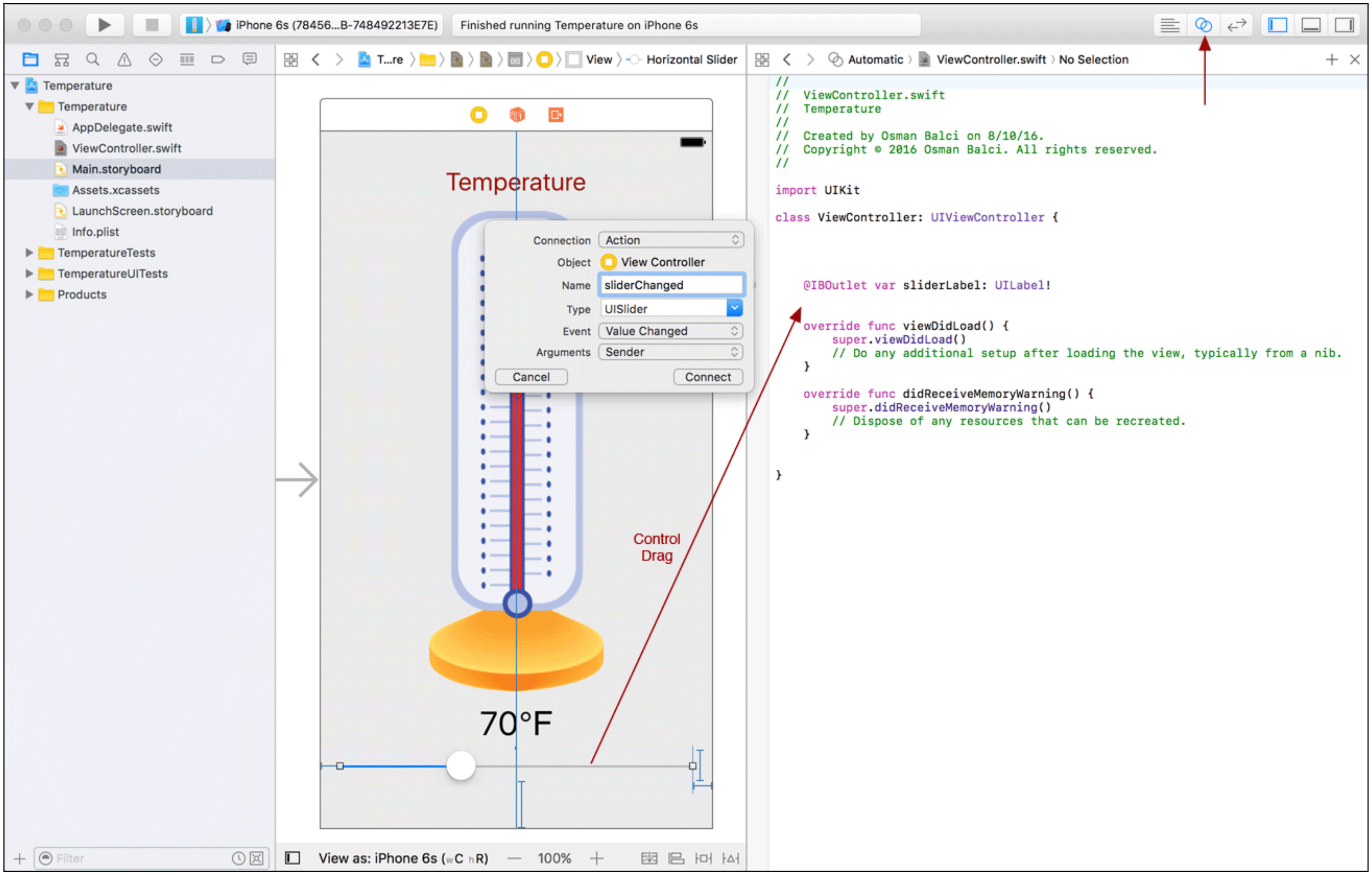
```
//
//  ViewController.swift
//  Temperature
//
//  Created by Osman Balci on 8/10/16.
//  Copyright © 2016 Osman Balci. All rights reserved.
//

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.
    }


}
```

1. Control-drag from the **Slider** object to the location in the `ViewController.swift` file as shown below. When you let go the mouse button, a dialog box shows up.
2. Select **Action** from the Connection pop-up menu.
3. Enter `sliderChanged` as the Name.
4. Set the class (Type) to **UISlider** to make it strongly typed.
5. Make sure that the *Event* type is **Value Changed** and *Arguments* is set to Sender.
6. Click **Connect** to insert the following into the code:

```
@IBAction func sliderChanged(_ sender: UISlider) {
}
```

Read the statement above as follows:

- Declare a method called sliderChanged with one parameter called sender, which specifies the object reference of the object instantiated from the UISlider class in the Interface Builder (IB). The **argument label** of the parameter is suppressed by using the underscore in front of the parameter name.
- The @IBAction IB attribute indicates that this method will be invoked from an IB object. Note that the @IBAction attribute is not part of the Swift programming language. **An IB attribute is a declaration attribute used by the IB to synchronize with Xcode.**



Connection	Action
Object	View Controller
Name	sliderChanged
Type	UISlider
Event	Value Changed
Arguments	Sender

Cancel Connect

```
//
//  ViewController.swift
//  Temperature
//
//  Created by Osman Balci on 8/10/16.
//  Copyright © 2016 Osman Balci. All rights reserved.
//

import UIKit

class ViewController: UIViewController {



    @IBOutlet var sliderLabel: UILabel!

    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.
    }


}
```

Step 9: MVC Design Pattern: Controller

Copy and paste the documented code given below. **Carefully study the code, understand what it is doing, and learn from it!**

`ViewController.swift` file:

```
//
//  ViewController.swift
```

```swift
//  Temperature
//
//  Created by Osman Balci on 8/10/16.
//  Copyright © 2016 Osman Balci. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    // Instance Variable
    @IBOutlet var sliderLabel: UILabel!

    // Instance Method
    @IBAction func sliderChanged(_ sender: UISlider) {

        /*
         The sender's value, i.e., slider's value, is a floating-point value.
         Floating-point values are always truncated when used to initialize a new integer value.
         Therefore, we add 0.5 to round the value to the nearest integer value.
         */

        let sliderIntValue = Int(sender.value + 0.5)    // Conversion from Float to Integer

        /*
         Slider's integer value is converted into a String value.
         The String value is assigned to be the slider's label text.
         */

        sliderLabel.text = String(sliderIntValue) + "°F"  // Conversion from Integer to String
    }

    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.
    }

}
```

Step 10: Build and Run your application

1. Run your app under the iOS Simulator for many different iOS devices, iPhone, iPad, and iPod Touch.
2. Under the **iOS Simulator**, select **Hardware → Rotate Left** or **Rotate Right** or press **Command-Left Arrow** or **Command-Right Arrow** repeatedly to see how auto layout works under different device orientations and screen sizes.
3. Move the Slider knob left and right to see how the temperature value changes under portrait and landscape device orientations.