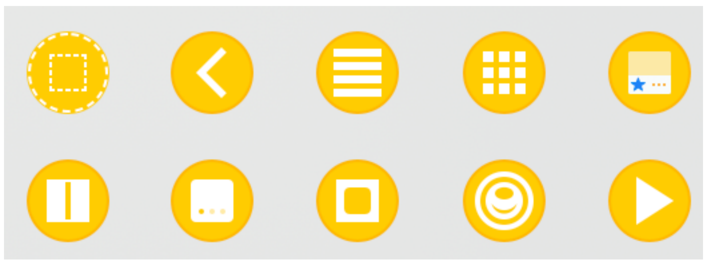
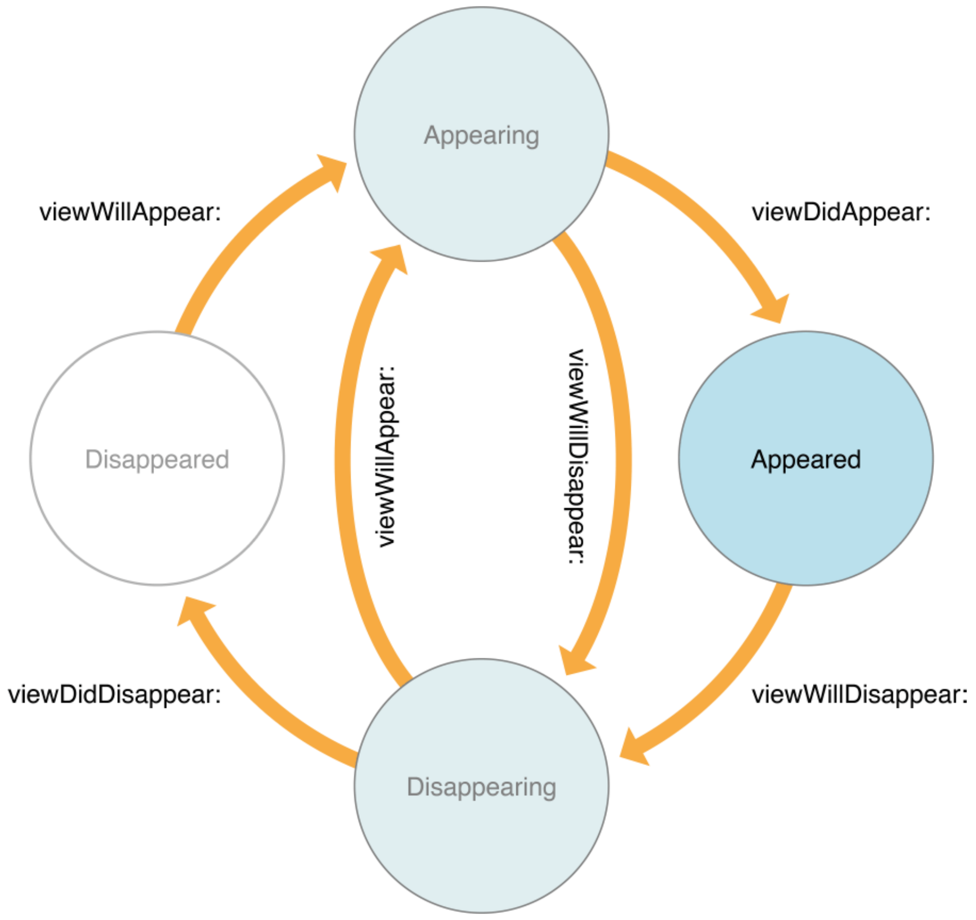
CHAPTER TWELVE : COMMUNICATIONS BETWEEN CONTROLLER AND VIEW





I. ROLE OF CONTROLLER IN MVC

The controller layer is the least reusable part of your app as it often involves your domain-specific rules. It should be no surprise that what makes sense in your app won't always make sense in someone else's. The controller will then use all the elements in your model layer to define the flow of information in your app.

Usually, you'll see classes from this layer deciding things like:

- What should I access first: the persistence or the network?
- How often should I refresh the app?
- What should the next screen be and in which circumstances?
- If the app goes to the background, what should I tidy up?
- The user tapped on a cell; what should I do next?

Think of the controller layer as the brain, or engine, of the app; it decides what happens next.

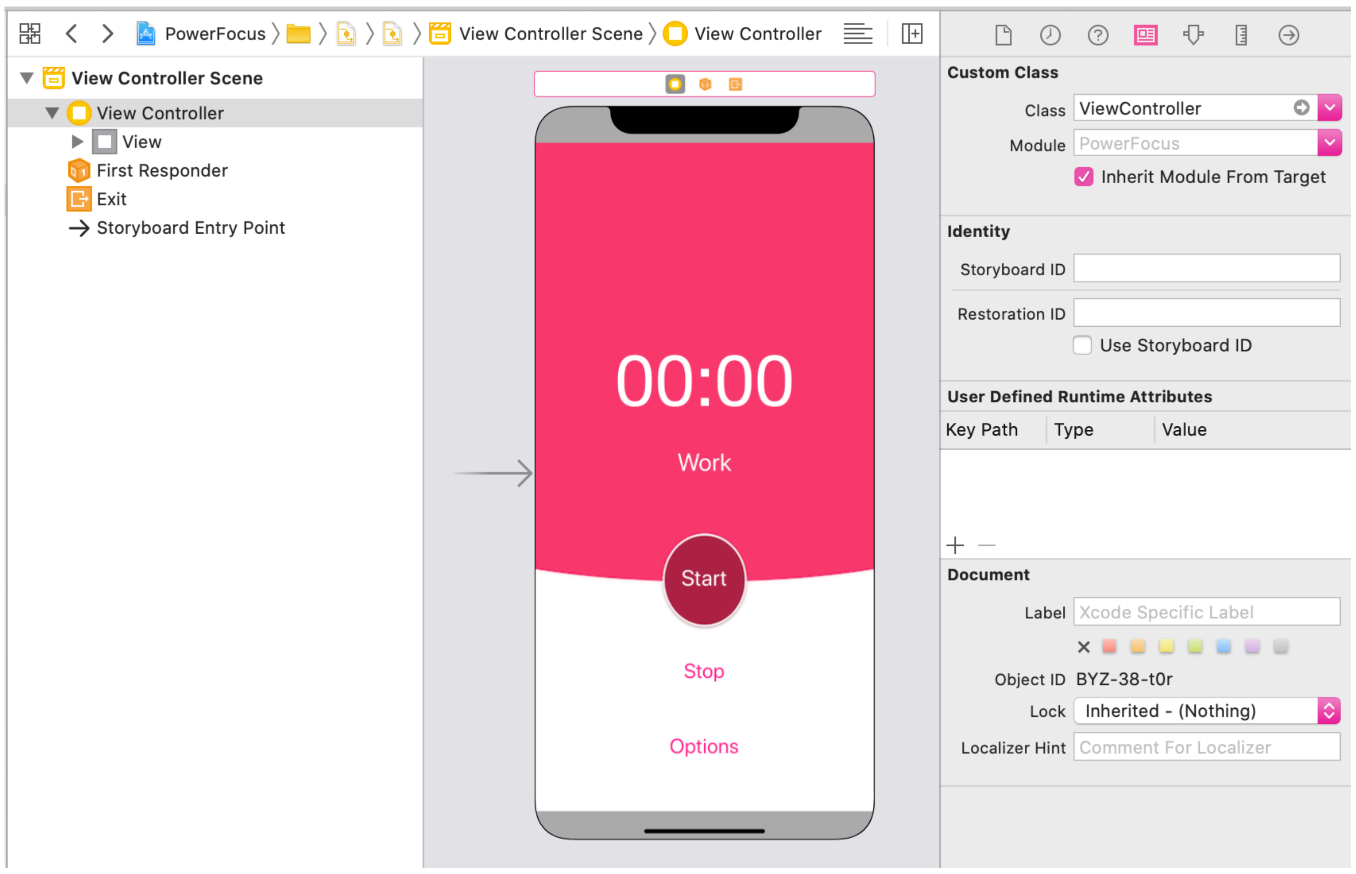
You likely won't do much testing in the controller layer as, for the most part, it's meant to kick things off, trigger data loads, handle UI interactions, mediate between UI and model, etc.

II. SIMPLE BINDING

How does the controller control the view? Since the controllers are Swift classes, we need a mechanism to synchronize the data changes between them. The data bindings are here to help. The idea of data binding is extremely intuitive and straightforward: the data in the view should always be the same as the bound attributes in the controller.

Here is a simple example to bind the timer label to an attribute in the controller ViewController.swift. Almost all elements in the storyboard can be bound. But since the binding is goes both ways, we need to make the model know the controller Swift file.

The first thing to check before creating any binding is the view controller's class in the identity inspector. Go to the **Inspectors > Identity Inspector** in the right pane. In the field "class", there will be all the classes that implement the UIViewController protocol. You can define your own customized controller by creating your own class that implements (directly or indirectly) the UIViewController protocol. But for now, we only need to work on the default ViewController.



Custom Class
Class ViewController
Module PowerFocus
☑ Inherit Module From Target

Identity
Storyboard ID
Restoration ID
☐ Use Storyboard ID

User Defined Runtime Attributes

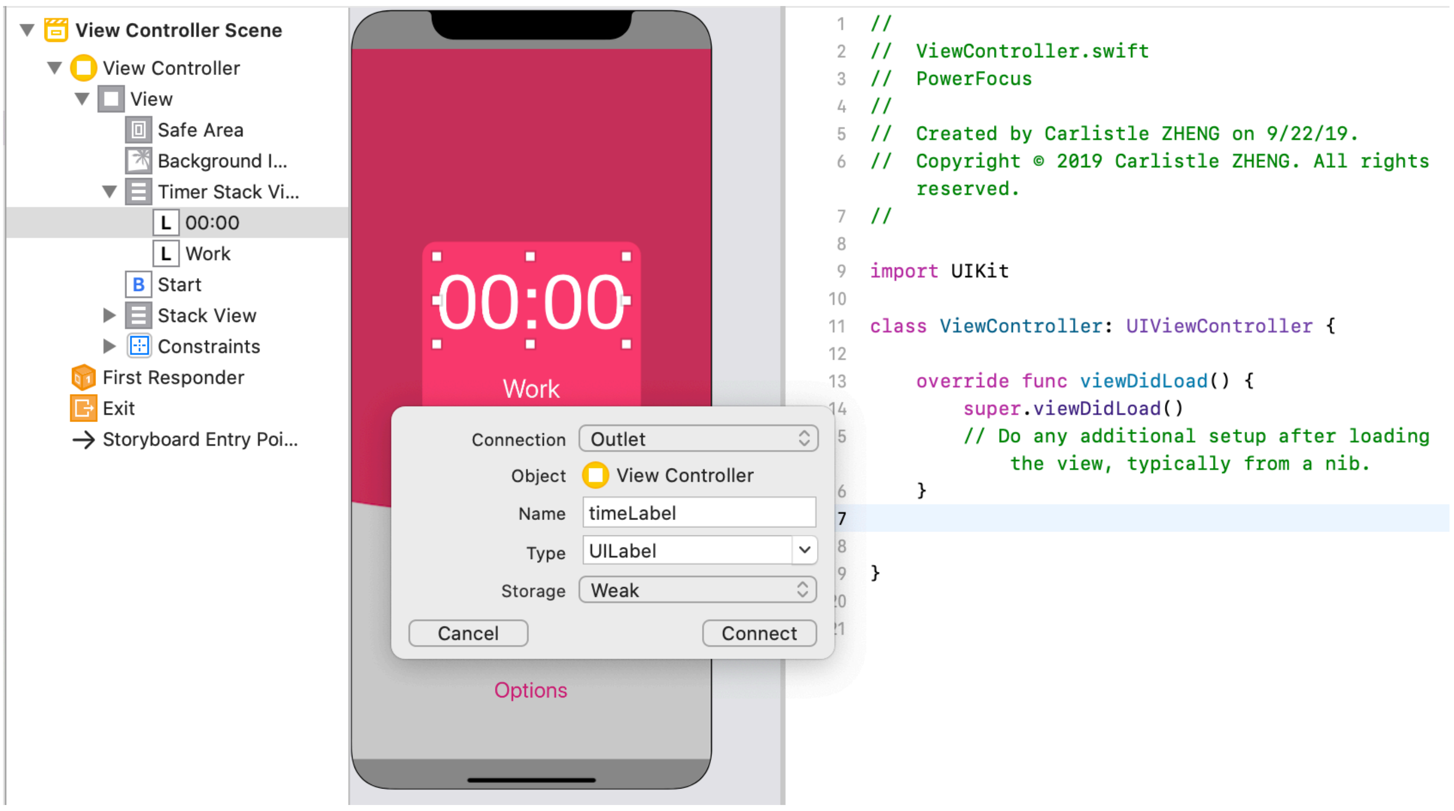
Key Path	Type	Value

Document
Label Xcode Specific Label
Object ID BYZ-38-t0r
Lock Inherited - (Nothing)
Localizer Hint Comment For Localizer

If Xcode can't find the associated controller, there won't be an arrow on the right. Click on the arrow, you will see the auto generated view controller you got when the project was created.

```swift
//
//  ViewController.swift
//  PowerFocus
//
//  Created by Carlistle ZHENG on 9/22/19.
//  Copyright © 2019 Carlistle ZHENG. 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.
    }


}


```

Open a second editor to show the storyboard and the ViewController.swift at the same time. Select the time label (00:00) from the view hierarchy and control drag it to the controller. Put your mouse inside the class but outside the viewDidLoad method, you will see a prompt: **Insert outlet or outlet collection**. A dialog box will pop out like this:



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

```swift
//
//  ViewController.swift
//  PowerFocus
//
//  Created by Carlistle ZHENG on 9/22/19.
//  Copyright © 2019 Carlistle ZHENG. 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.
    }


}
```

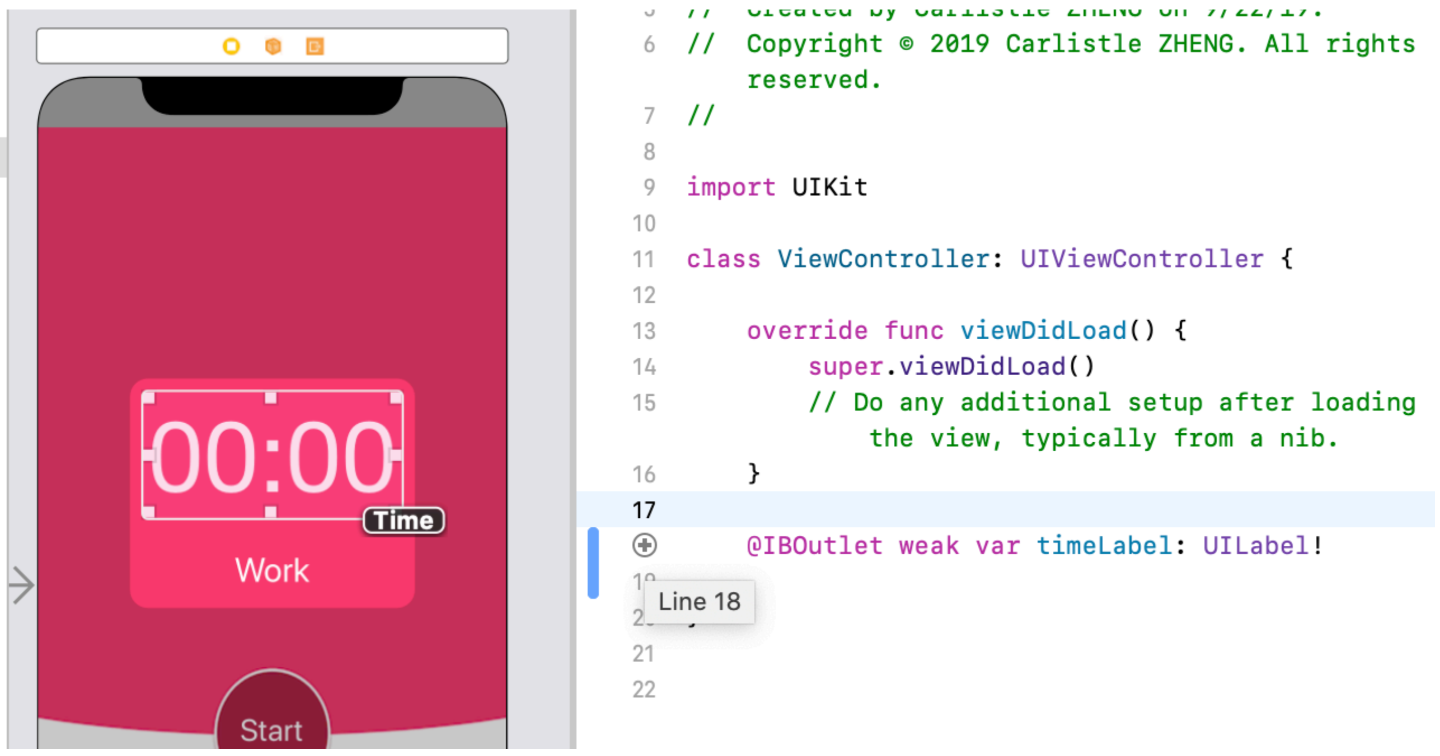
Since the binding is just for one label, the type of the connection is **Outlet** (if you choose the Outlet Collection type, you will get a collection with one element in it).

Enter the name **timeLabel** as the outlet's name, this will be the name of the outlet attribute for the class ViewController. The convention among iOS developers is to indicate the type of the referenced widget in the name of the outlet.

The type should be **UILabel** because we know the outlet's type for sure. But in some cases, you might need to choose the type **Any** if it can't be determined before the compilation. In that case, you need use the keyword **as?** to downcast it in the controller during runtime.

The option **weak** is for memory management. Swift's ARC (Automatic Reference Counting) mechanism only destroys an object when it doesn't have any reference from other objects. Our timeLabel outlet shouldn't be kept in the memory heap after the element in the storyboard is destroyed. If you have trouble understanding the memory management process, a simple version of this is that you almost never declare strong outlet because the widget shouldn't be kept in the memory heap on the controller's account.

Once the outlet is correctly created, hovering on the cross next to the line number will highlight the label widget in the storyboard.



```
//  Copyright © 2019 Carlistle ZHENG. 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.
    }

    @IBOutlet weak var timeLabel: UILabel!
```

In the viewDidLoad method, you can set the text to see the binding's effect. This method sets the label outlet's text field to "01:59" after the view is loaded.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
    timeLabel.text = "01:59"
}
```

If you launch the app again in the simulator, you will see that instead of displaying the default text defined in the storyboard, the text will be "01:59" because the label's text is updated in the viewDidLoad method.

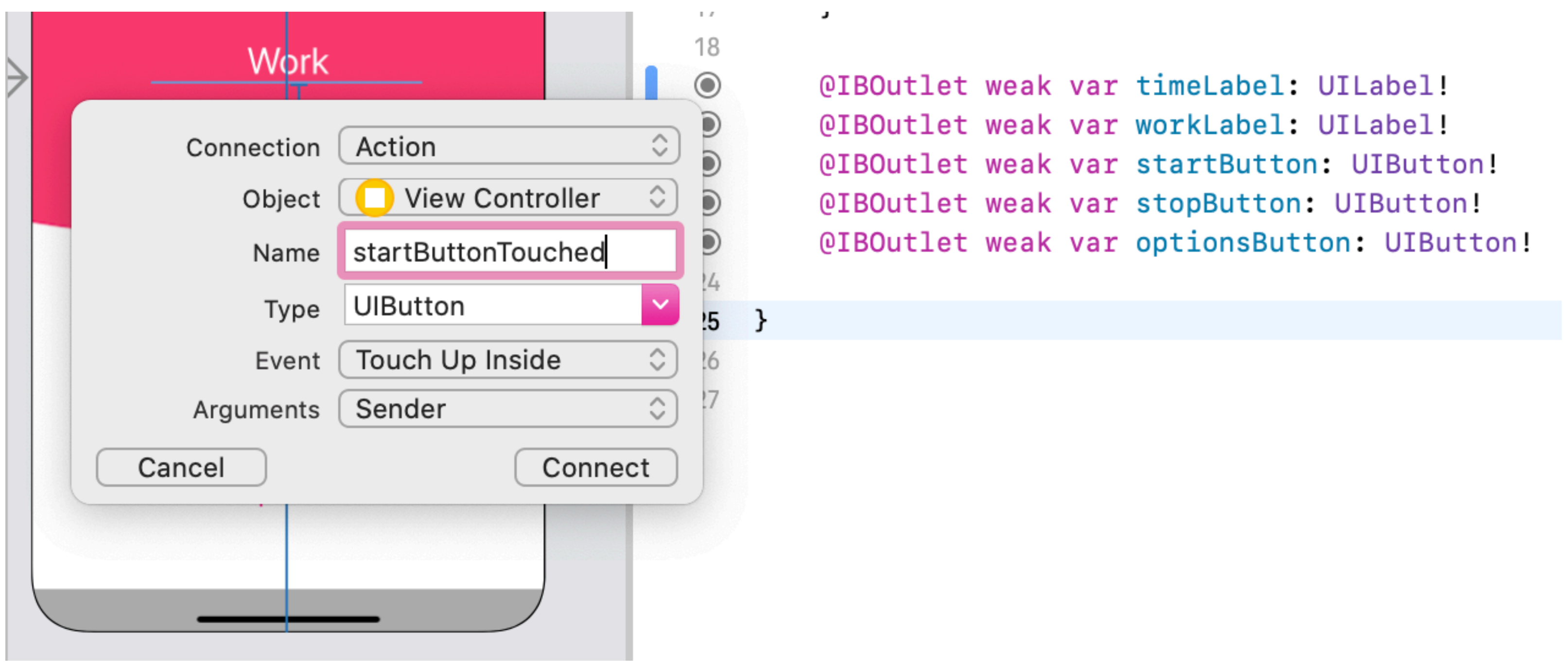
Just like what we did with the time label, now create outlets for the work label and the start, stop, and options button. Name them workLabel, startButton, stopButton, and optionsButton. When you control-drag from a button, the default prompt is to create an action. Because we want to access the button itself in the controller, choose "outlet" instead of "action". If you don't remember the difference between an outlet and an action, I suggest you go back to the first part of this book and have a quick revision.

You should have these following lines in the ViewController:

```
@IBOutlet weak var timeLabel: UILabel!
@IBOutlet weak var workLabel: UILabel!
@IBOutlet weak var startButton: UIButton!
@IBOutlet weak var stopButton: UIButton!
@IBOutlet weak var optionsButton: UIButton!
```

Being able to access UI elements from the ViewController class is not enough. A view controller's job also includes handling user interactions' events. A event can be "button touched", "slider value changed", "swipe gesture recognized", etc.

The simplest way to handle an event is to add an action to the view's view controller. Control-drag from the start button to the view controller. The connection type should be "Action" and the event should be "**Touch Up Inside**". Name the action "**startButtonTouched**". Most importantly, since we know for sure that the sender is a button, change the type from any to "**UIButton**".



Connection: Action
Object: View Controller
Name: startButtonTouched
Type: UIButton
Event: Touch Up Inside
Arguments: Sender

```
@IBOutlet weak var timeLabel: UILabel!
@IBOutlet weak var workLabel: UILabel!
@IBOutlet weak var startButton: UIButton!
@IBOutlet weak var stopButton: UIButton!
@IBOutlet weak var optionsButton: UIButton!

}
```

Click Connect and you will see an automatically generated method in the ViewController class. Create two more actions for the stop button and the options button and name the actions "stopButtonTouched" and "actionButtonTouched".

Just to make sure that they are correctly connected, add some print statement in the function and restart the simulator. Every time the button is touched, the print statement in the button's IBAction should print in the console.

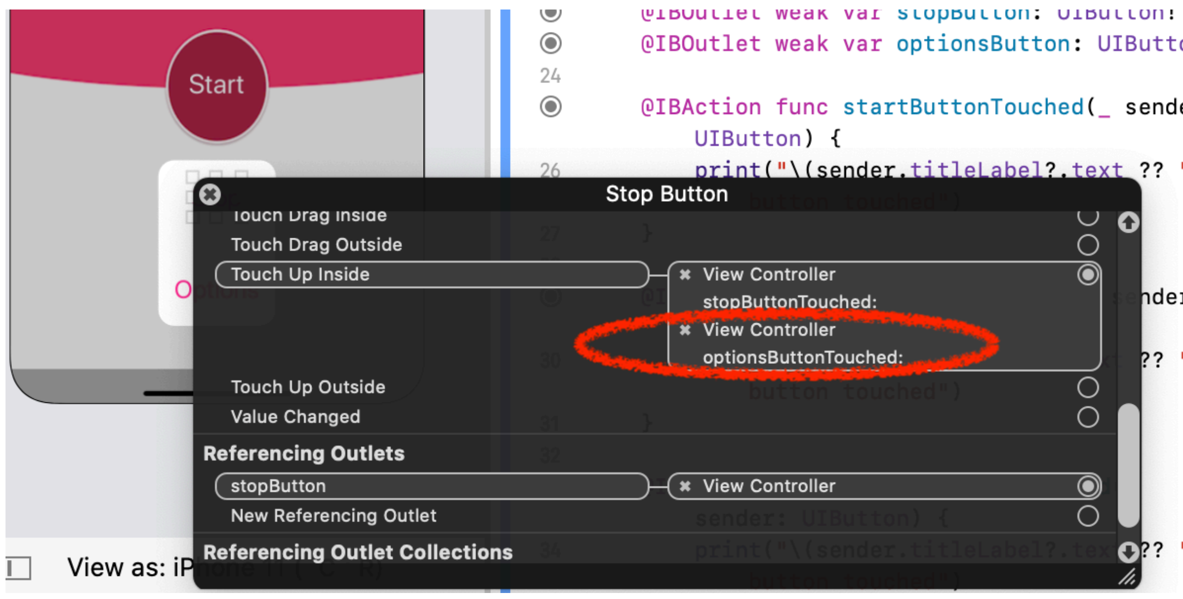
Your actions should look like this:

```
@IBAction func startButtonTouched(_ sender: UIButton) {
    print("\(sender.titleLabel?.text ?? "") button touched")
}

@IBAction func stopButtonTouched(_ sender: UIButton) {
    print("\(sender.titleLabel?.text ?? "") button touched")
}

@IBAction func optionsButtonTouched(_ sender: UIButton) {
    print("\(sender.titleLabel?.text ?? "") button touched")
}
```

When you have created some outlets and actions in a view controller, you should ALWAYS check that they are properly wired. This is a very common source of errors. Open the storyboard and the ViewController class side by side, all actions and outlets's circles (in the row of line number) should be filled. If it's not, the action or outlet is not connected to any storyboard element.

Another important thing to check is that there's no "double wired" actions (at least not intentionally). Right click on a button, you should see in "Send Event" **ONE** Touch Up Inside action and **ONE** "Referencing Outlets" to the View Controller.

In the following example, there are two actions linked to the touch up inside event of the stop button.

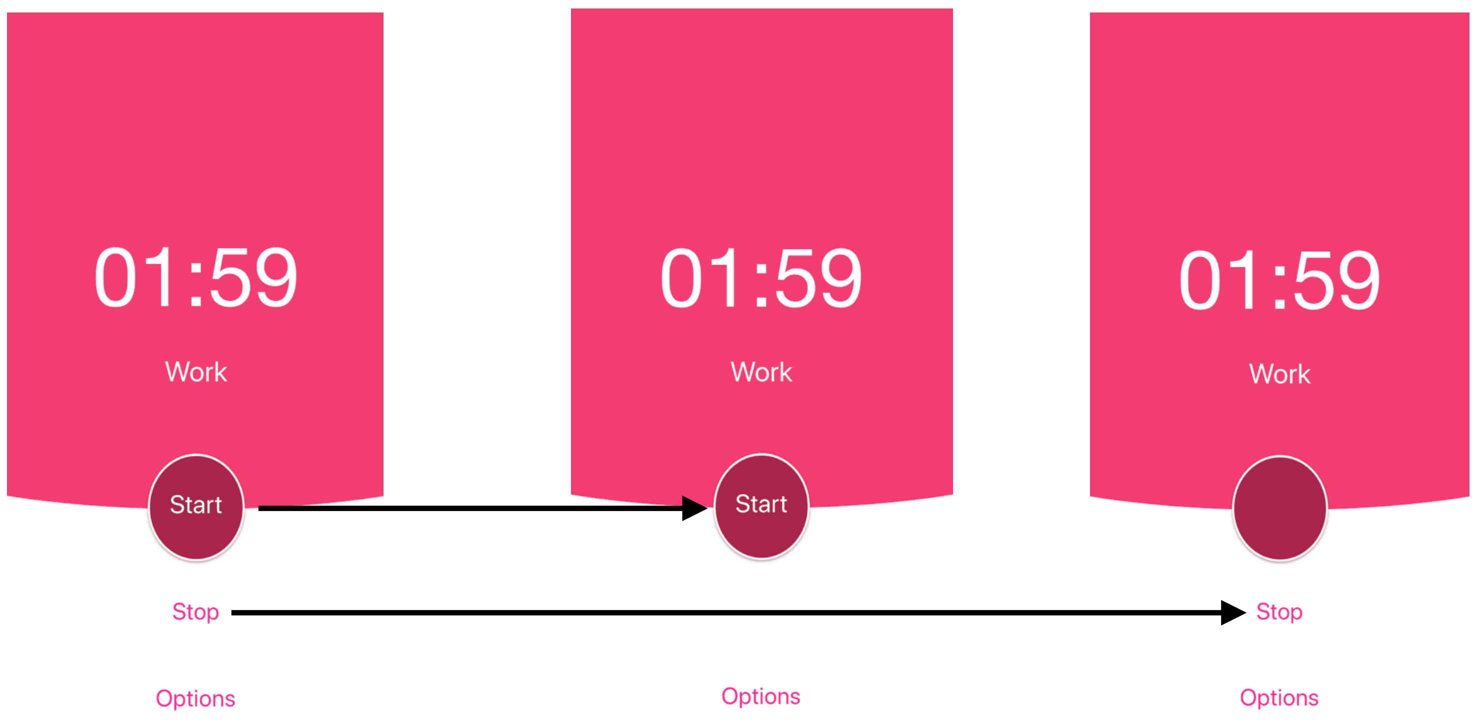


```
@IBOutlet weak var optionsButton: UIButton!

@IBAction func startButtonTouched(_ sender:
    UIButton) {
```

If you tap the button once, it will trigger both "stopButtonTouched" and "optionalButtonTouched". You will see "Stop button touched" and "Options button touched" printed in the console. Click the cross in the sent event list to delete extra actions.

EXERCISE

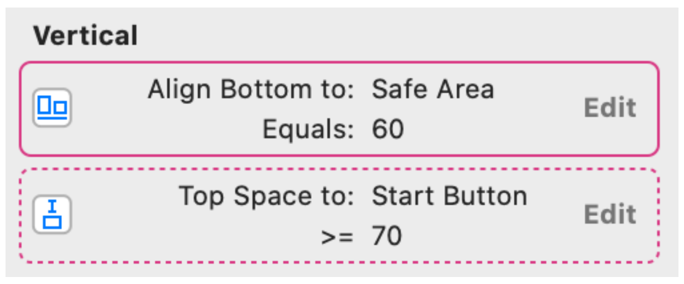
If you look closer to the current user interface, you will notice that displaying both "Start" button and "Stop" button doesn't make sense. When the timer is on, the start button should be hidden (or turn to a pause button). The stop button should only be visible when the timer is on.



Go to Help > Developer Documentation and look for a way to hide and show a button. For the start button, you can simply use the attribute UIButton.isHidden: Bool. You should set it to true or false when the view is loaded, the start button is touched, or the stop button is touched.

However, if you simply hide the stop button, the options button will move up because they share a stack view. Try to come up with a solution yourself and try it out. Here are some hints:

- You can set the stack view's distribution to align the bottom of the stack view.
- You can modify the stack view's constraints to make it align to the bottom in priority like this:

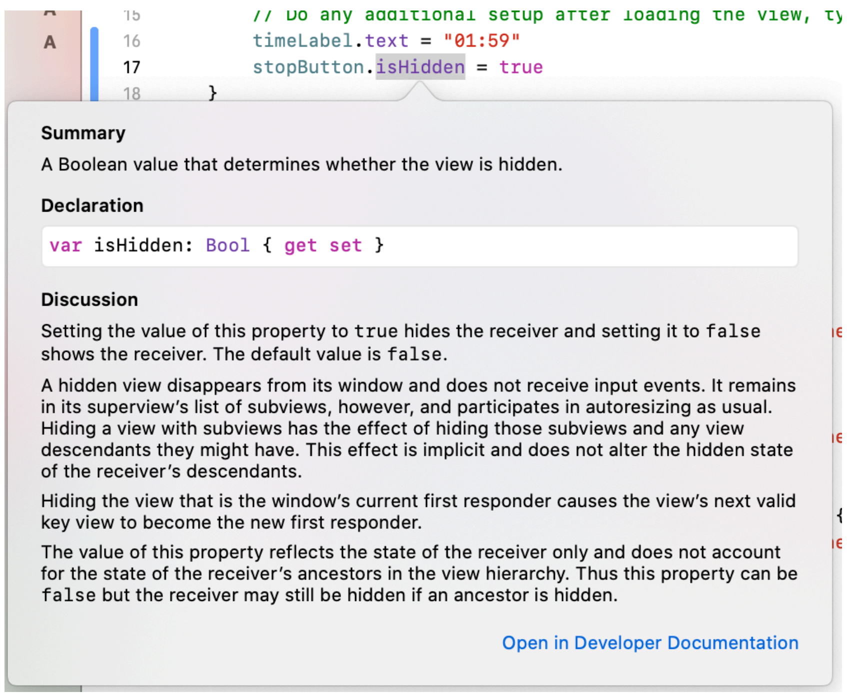


Vertical

Align Bottom to: Safe Area
Equals: 60
Edit

Top Space to: Start Button
>= 70
Edit

- You can set the title to an empty string using UIButton.setTitle(_ title: String?, for state: UIControl.state). The state should be **.normal**. Then you can enable or disable the button using UIButton.isEnabled: bool.



```
        // Do any additional setup after loading the view, ty
        timeLabel.text = "01:59"
        stopButton.isHidden = true
    }
```

Summary

A Boolean value that determines whether the view is hidden.

Declaration

```
var isHidden: Bool { get set }
```

Discussion

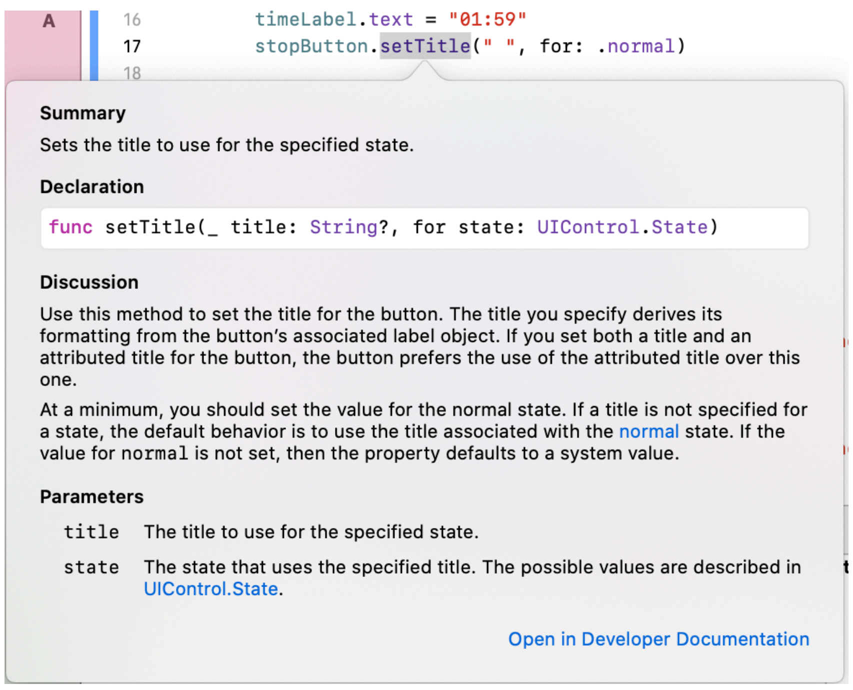
Setting the value of this property to `true` hides the receiver and setting it to `false` shows the receiver. The default value is `false`.

A hidden view disappears from its window and does not receive input events. It remains in its superview's list of subviews, however, and participates in autoresizing as usual. Hiding a view with subviews has the effect of hiding those subviews and any view descendants they might have. This effect is implicit and does not alter the hidden state of the receiver's descendants.

Hiding the view that is the window's current first responder causes the view's next valid key view to become the new first responder.

The value of this property reflects the state of the receiver only and does not account for the state of the receiver's ancestors in the view hierarchy. Thus this property can be `false` but the receiver may still be hidden if an ancestor is hidden.

Open in Developer Documentation



```
        timeLabel.text = "01:59"
        stopButton.setTitle(" ", for: .normal)
```

Summary

Sets the title to use for the specified state.

Declaration

```
func setTitle(_ title: String?, for state: UIControl.State)
```

Discussion

Use this method to set the title for the button. The title you specify derives its formatting from the button's associated label object. If you set both a title and an attributed title for the button, the button prefers the use of the attributed title over this one.

At a minimum, you should set the value for the normal state. If a title is not specified for a state, the default behavior is to use the title associated with the normal state. If the value for `normal` is not set, then the property defaults to a system value.

Parameters

`title`	The title to use for the specified state.
`state`	The state that uses the specified title. The possible values are described in UIControl.State.

Open in Developer Documentation

SUMMARY

In this chapter, we have worked on these topics:

- introduction to controller's roles
- View controller's outlets to storyboard
- View controller's actions to storyboard
- Handle simple user interactions

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

If you can't figure out the exercise on your own, you can check the answers at

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Twelve_Exercises.zip

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Twelve_PowerFocus.zip