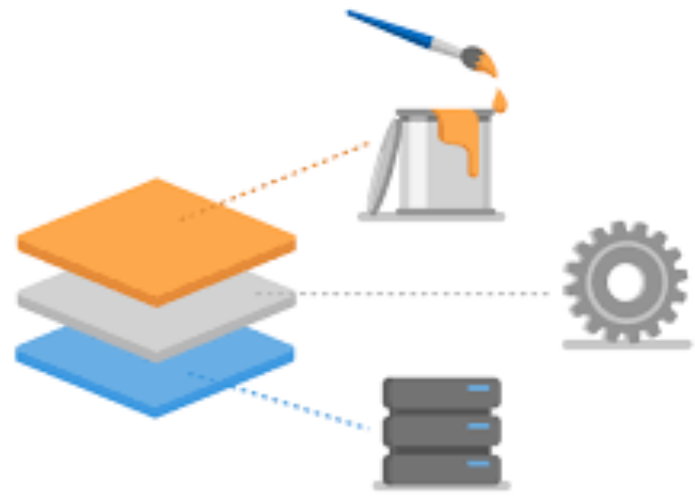
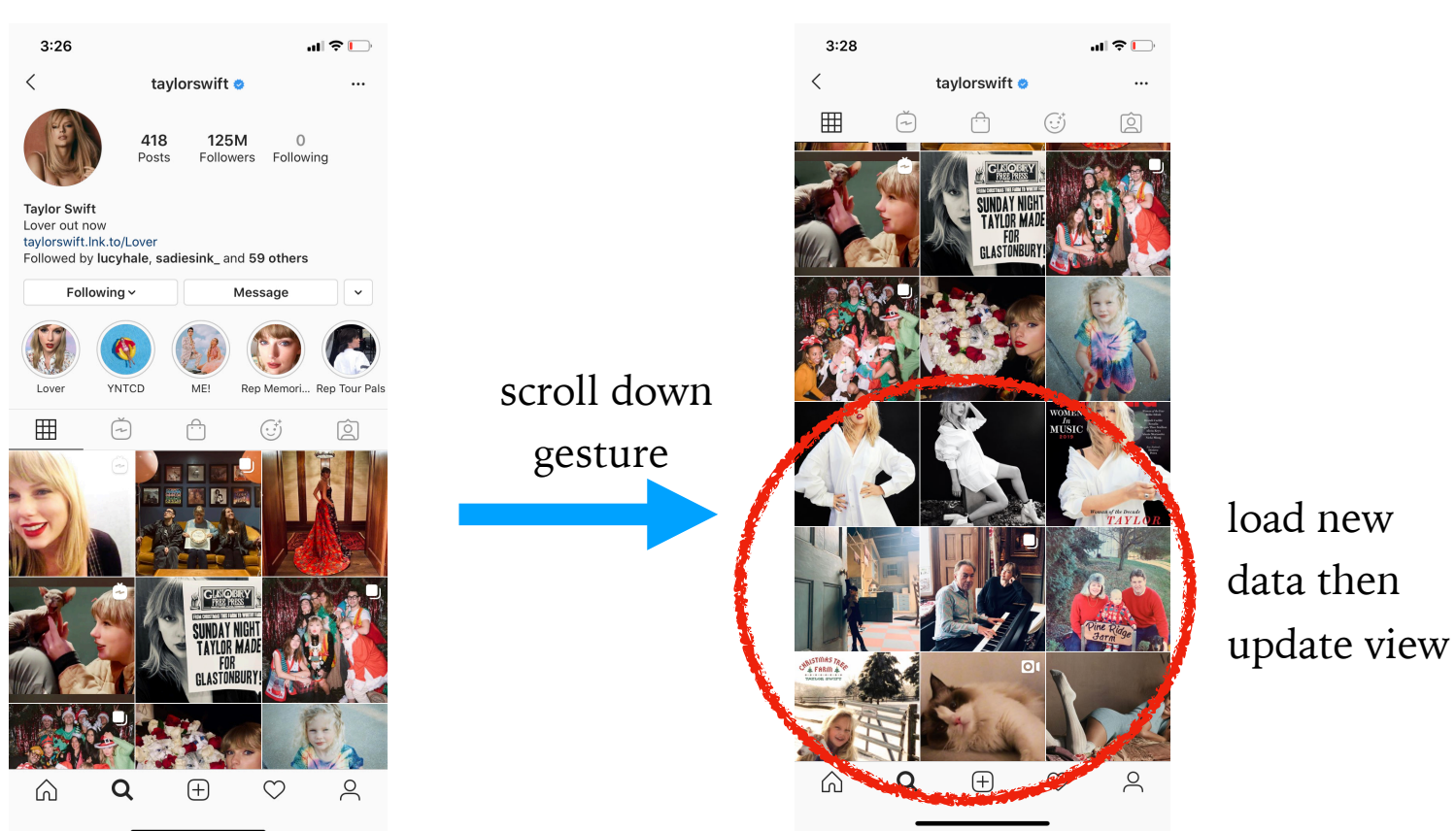
CHAPTER THIRTEEN : CONTROLLER BEHAVIORS



I. CONTROLLER BEHAVIOR AND MODEL UPDATE

The most important role of a controller is to control the behavior of the user interface. When an event leads to some changes in the UI or the data, it is up to the controller to decide what to do and what to update. An event can be a user interaction event, some network event (when the application is linked to a networked server), or a system event (iOS can send events for low battery, connection loss…)

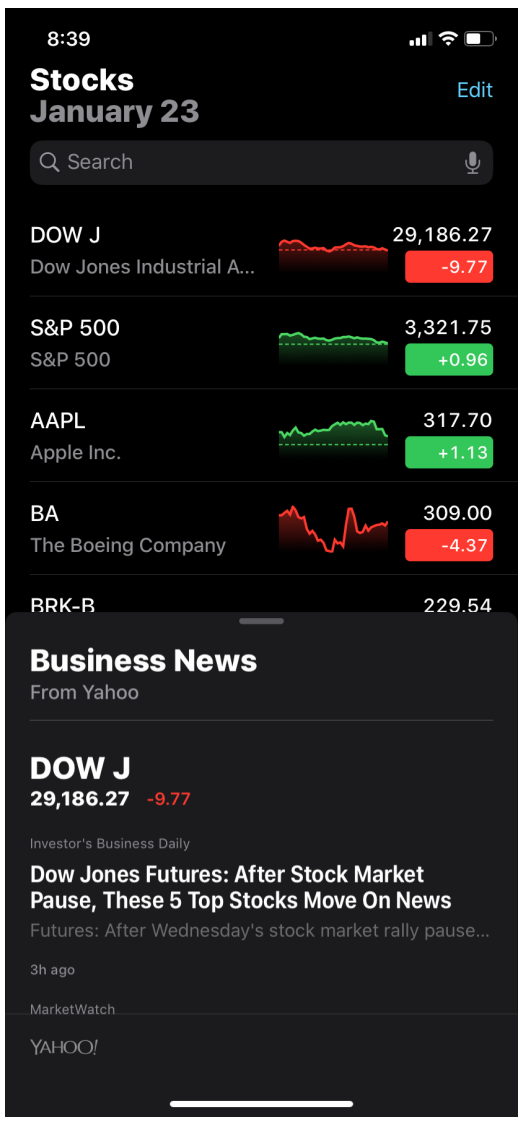
For example, when you scroll down in the Instagram app, the controller has to handle the scroll action. When a scroll action is detected, some HTTP requests are sent to get more data from the Instagram server. When the Instagram app receives some response packages, a



network event is triggered and the view controller processes the new data and display it in the UI.

The UI and its model should be in sync for most of the time, and it's their controller's job to update the UI when the model's values are changed. Since checking the model can be consuming in time and memory, we can't use simple dead loops to check if the model has changed.

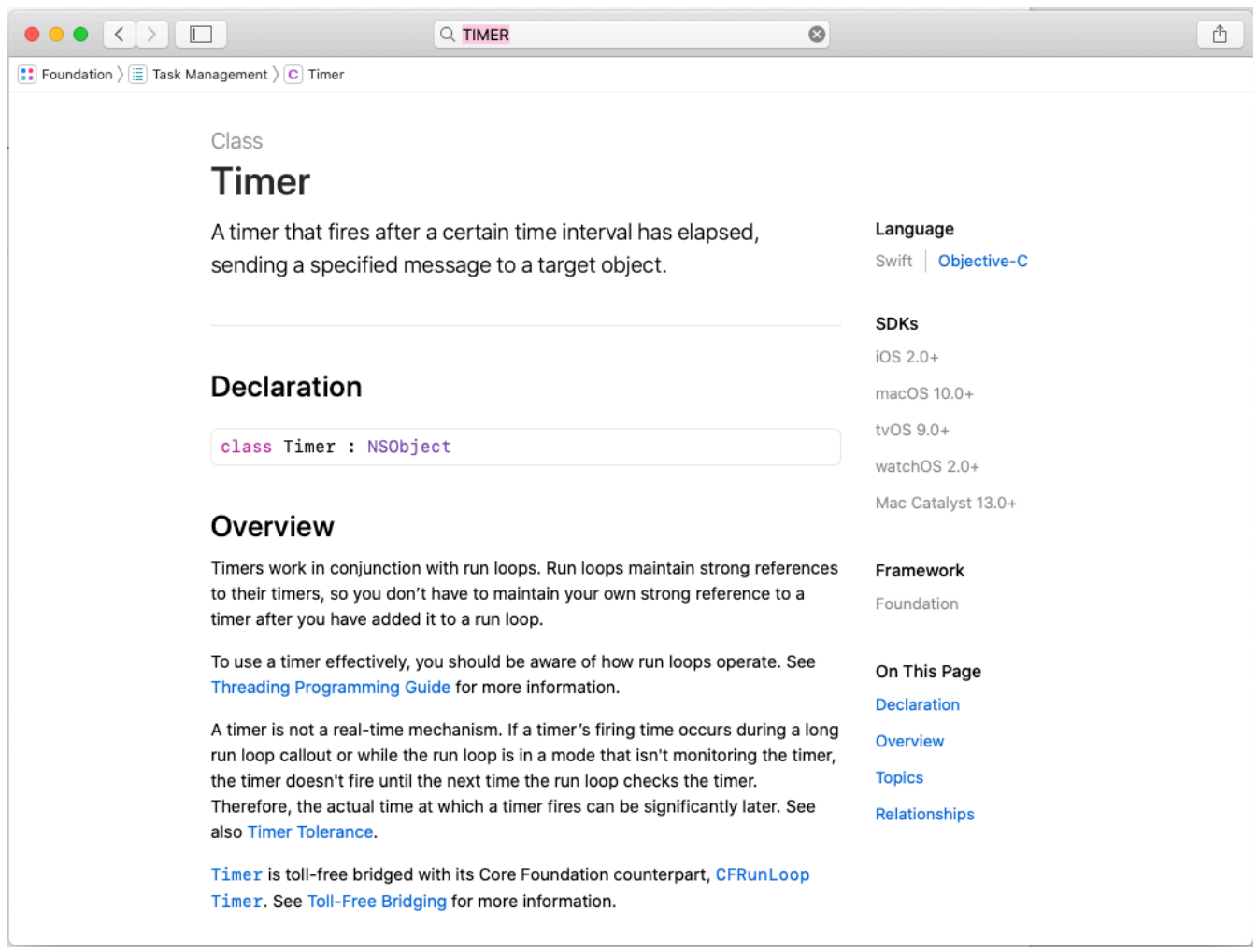
For example, open the Stocks app and check the stock prices. If you do nothing, the app will change the stock values several times per minute. When the model has some value changes (in this example, the stock value), the ViewController need to perform some actions to display the new data that it received. The ViewController of this app needs to change the values in stock value text labels, and it also need to add the new value to the trend diagram and repaint it.



11. POWERFOCOUS TIMER

In chapter ten, we created a timer model that can alternate between work and break sessions. But how can we know when to update the timer from the controller? While developing an iOS app, you might often need some features or functionalities that are useful and needed often. A good instinct is to check the developer documentation because Apple provides a large library for cases like this. These methods are well written, well tested, extremely stable, and well documented.

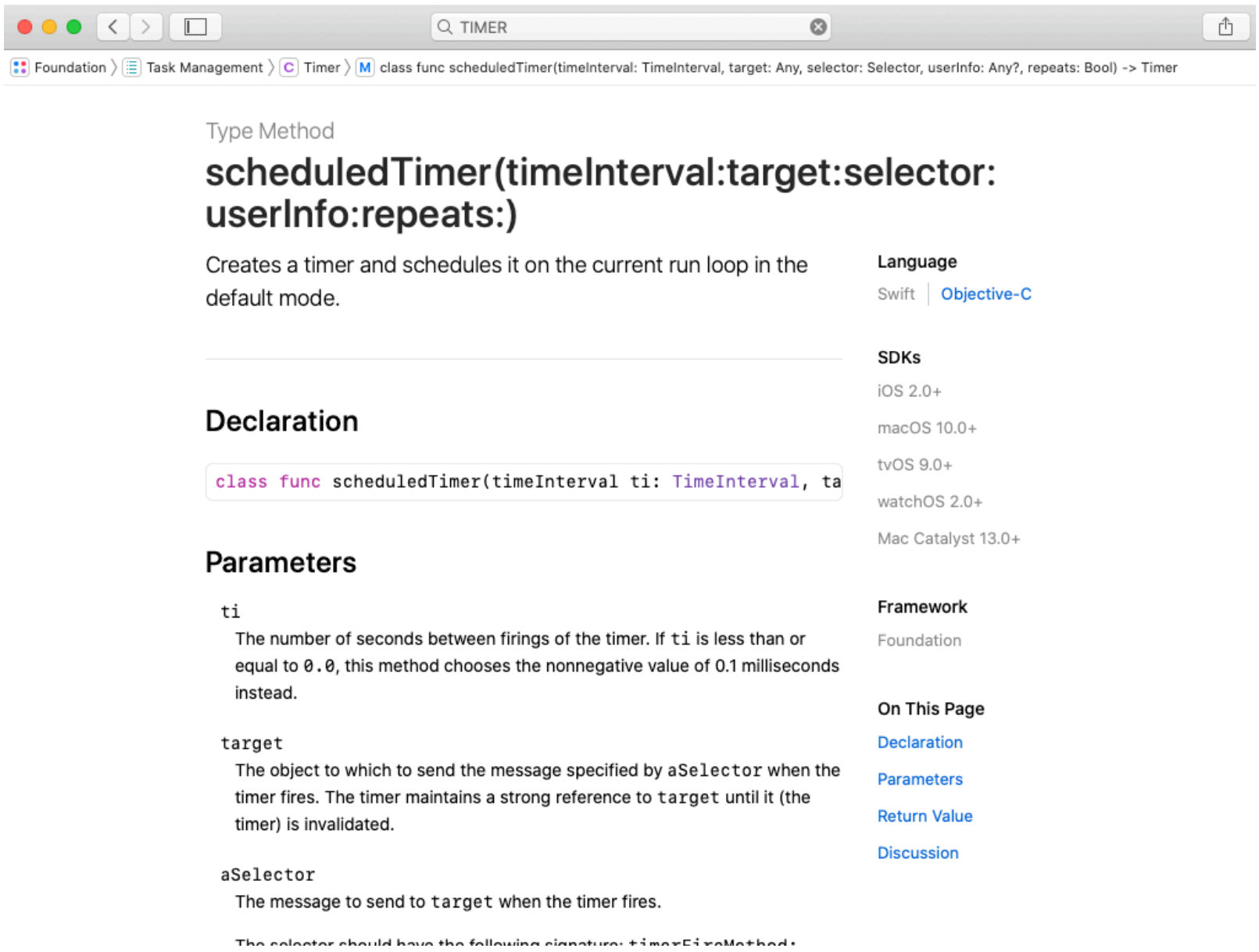
Open developer documentation by pressing ⌘⇧0 and search timer in the search bar, you will see Apple provides iOS developers with a class Timer in the **Foundation** package.



Great! We don't have to implement a timer on our own! Now it's time to figure out how to use the provided Timer class. If you don't want to read the entire documentation, at least read the overview section first to get a rough idea about what a timer does. Then go to the topic "creating a timer" and study the constructors of this class.

There are many ways to create a timer and sometimes it's pretty hard to choose the most suitable constructor. Here we need the timer to notify us to update the time left every second, so it's a repeated timer with the interval of one second. The best fit is the following

constructor. Now look more closely at its parameters, and think about how and where should we initialize the timer.



First, add an attribute *timer* to the ViewController class:

```
var timer: Timer?
```

In the method viewDidLoad, initialize the timer with Timer.scheduledTimer constructor like this:

```
timer = Timer.scheduledTimer(timeInterval: 0.05,
                             target: self,
                             selector: #selector(ViewController.decSec),
                             userInfo: nil,
                             repeats: true)
```

The timer is fired every 50 ms so the timeInterval equals to 0.05. The timer message should be sent to ViewController so the target is self. The selector is a function that gets to executed when the time is up, for now leave it to ViewController.decSec and we will implement this method later. Leave the userInfo to nil and the repeats to true because it's a repeated timer.

III. TIMER HANDLER AND VALIDATION

The handler of the timer gets called every second, it's the method that updates the UI according to the model.

In the ViewController class, add a private variable focusTimer of type FocusTimer:

```
// The timer model that keeps track of work/break cycles
private var focusTimer: FocusTimer!
```

In the method viewDidLoad, initialize the attribute with the following constructor:

```
    focusTimer = FocusTimer(25, 5, 15, 3, true)
```

When the start and stop buttons are clicked, we need to toggle the "active" status of the focusTimer. To test this out, add a simple print statement in the handler:

```
@objc func decSec() -> Void {
    print("decSec")
}
```

In the viewDidLoad method, the timer should be invalidated after it's created. When the start button is clicked, the timer should become active, and when the stop button is clicked, it should be invalidated. To invalidate a timer, go to the documentation and study the following method:

> func invalidate()
> Stops the timer from ever firing again and requests its removal from its run loop.

In the method stopButtonTouched, add the following statements:

```
if timer?.isValid ?? false {
    timer?.invalidate()
}
```

It stops the timer if the timer is not nil and valid by using the method invalidate. If the timer is invalid, we give the default value of false and do nothing. In the method viewDidLoad, simply invalidate the timer without testing. It's unnecessary to test because we are sure it is not nil and validated right after the constructor.

```
timer = Timer.scheduledTimer(timeInterval: 1,
                             target: self,
                             selector: #selector(ViewController.decSec),
                             userInfo: nil,
                             repeats: true)
timer?.invalidate()
```

In the method startButtonTouched, initialize the timer using Timer.scheduledTimer method. This validates the timer and starts calling the handler that we defined.

```
        timer = Timer.scheduledTimer(timeInterval: 1,
                                     target: self,
                                     selector: #selector(ViewController.decSec),
                                     userInfo: nil,
                                     repeats: true)
```

Now test it out, you shouldn't see any print after the view is loaded. Click the start button, you will see a decSec message gets printed every second. Click the stop button, there shouldn't be any new printed message in the console. If it's not the case, go back to the corresponding method and check the timer's initialization and validation.

Your ViewController class should look like this:

```
class ViewController: UIViewController {

    var timer: Timer?

    // The timer model that keeps track of work/break cycles
    private var focusTimer: FocusTimer!

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

        timer = Timer.scheduledTimer(timeInterval: 1,
                                     target: self,
                                     selector: #selector(ViewController.decSec),
                                     userInfo: nil,
                                     repeats: true)
        timer?.invalidate()

        focusTimer = FocusTimer(25, 5, 15, 3, true)
    }

    @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!

    @IBAction func startButtonTouched(_ sender: UIButton) {
        print("\(sender.titleLabel?.text ?? "") button touched")
        timer = Timer.scheduledTimer(timeInterval: 1,
                                     target: self,
                                     selector: #selector(ViewController.decSec),
                                     userInfo: nil,
                                     repeats: true)
    }

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

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

    @objc func decSec() -> Void {
        print("decSec")
    }
}
```

IV. UPDATE THE MODEL

In the previous section, we managed to call the handler at the right time and validate/invalidate the timer when the start/stop button is clicked. But how do we keep track the time in the controller?

A huge part of the controller's job is to communicate with the model. In our case, the model is the FocusTimer, a customized class we created to track the work/break cycles. In the handler that gets called every second when the timer is active, we should update the model and get the remaining time and the updated activity type (work/break/longbreak).

When the start button is clicked, it should start with a work activity and zero completed cycles. Open the FocusTimer class and see what method to use. If you can't find any method to do these things, there might be some problems in your FocusTimer design. If that is unfortunately the case, you can download the zip from the link at the end of the chapter twelve. Move the ViewController.swift in the zip to your project and replace your own ViewController.swift.

In the startButtonTouched method before the timer initializer, add these following lines:

```
startButton.isHidden = true
focusTimer.setActivity(to: .work)
focusTimer.setSecondLeft(to: focusTimer.workLength * 60)
```

First we hide the start button because we shouldn't allow user to click start when the timer is already on. Then we set the activity to ActivityType.work (the parameter .work uses Swift's type inference). The secondLeft is in second but the work length is in minute. We need to set the secondLeft to workLength * 60.

The view controller should update the view according to the model. So the next step is to complete the timer's handler method. For now we don't need to worry about the case where the time's up when the timer gets called. Let's go to the method @objc func decSec() -> Void.

Here this method needs to decrement the second and set the time label to the updated value. If the time drops to zero, we need to switch the view to break or long break theme. You might want to add some convenience methods to the FocusTimer class that you didn't think of during the modeling. For example, a great method to add is a method that decrements the second and returns if the current cycle is over.

```
func decrementSecond() -> Bool {
    if secondLeft == 0 {
```

```
            return true
        } else if secondLeft == 1 {
            secondLeft -= 1
            return true
        } else {
            secondLeft -= 1
            return false
        }
    }
```

If the secondLeft is greater than zero, we decrement the second by one second. If the updated value of secondLeft is zero, we return true to indicate that the time is up for the current activity.

In the decSec method, use the FocusTimer.decrementSecond like this:

```
    @objc func decSec() -> Void {
        if focusTimer.decrementSecond() {
            // TODO update the view when the activity has finished
            print("Time is up")
        } else {
            let sec = focusTimer.getSecondLeft()
            timeLabel.text = "\(sec / 60):\( sec % 60)"
        }
    }
```

By calling the focusTimer.decrementSecond method, the focusTimer gets updated and we collects the result if the activity has finished. If we get true, some view updates are needed, we will deal with this case later. If we get false, we simply format the timeLabel to the second left. We use the integer division and the modulus operator to get the number of minutes and the number of seconds. Then these two integers get injected into the formatted string using the \() operator.

Run the app and click the start button, the time label should be automatically decremented.

EXERCISE

In the current design, we can only start or stop the timer. In this exercise we would like to add a functionality of pause.

When the timer is on, the start button should become the pause button. When the pause button is on, the focusPower should stop decrementing. In the paused state, the pause button should become the start button. Clicking the button again will bring the timer back to active. In the paused state, you should also be able to stop the timer, which initializes the timer.

To implement the pause feature, you can add a flag to the ViewController class. When the pause flag is true, the handler desSec should be disabled. The label outlet workLabel should also be updated when the pause is enabled or disabled.

In the next chapter, I will provide you a solution to implement this feature along with other features. For now, just think of your own solution. This "design" part is one of the most important steps of the tasks of a software engineer. There is no the best solution, you can use your creativity and try to implement your design.

A small hint: the ViewController may need a enum to capture the status. There are 3 possible states:

```
enum TimerStatus {
    case stoped
    case paused
    case enabled
}
```

In the methods that use or change the status, just do a switch like this:

```
@IBAction func someFunc(someArg: Arg) {
    switch timerStatus {
    case .enabled:
        // Foo
    case .stoped:
        // Foo
    case .paused:
        // Foo
    }

}
```

SUMMARY

In this chapter, we have worked on these topics:

- Review the roles of view controllers
- Use provided class Timer and study its developer documentation
- Create timer handler
- Update model from view controller
- Add pause mechanism to timer

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/Chapteer_Thirteen_PowerFocus.zip

For reference, you can download the complete Xcode project from

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