Урок 1

Применение паттернов «наблюдатель», «одиночка», «делегат»

Изучаем реализацию самых популярных паттернов в iOS. Знакомимся с NotificationCenter.

Паттерн – это описание алгоритма или подхода в решении конкретной задачи. Можно сказать, что паттерны – это подарок опытных программистов начинающим. И хоть справочники паттернов бывают написаны сложным языком, они все же полезны, как и сами готовые алгоритмы, представленные в них.

Singleton – «одиночка»

«Одиночка» (singleton) – это паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

К единственному экземпляру класса можно обратиться из любой точки приложения.

*Рассмотрим пример с классом **Session,** предназначенном для хранения информации об актуальном пользователе приложения. Здесь хранится его имя, id, токен для доступа к серверу и баланс счета. После авторизации получаем данные о пользователе, создаем экземпляр класса **Session** и устанавливаем информацию в его свойства.*

*Далее нужно отобразить информацию о пользователе на экране личного кабинета. Если создать новый экземпляр класса **Session**, в нем не будет сведений, записанных в предыдущем. Значит, нужен единственный экземпляр – тот, который содержит информацию о пользователе. Так как он в приложении тоже один, экземпляр класса для других данных не требуется.*

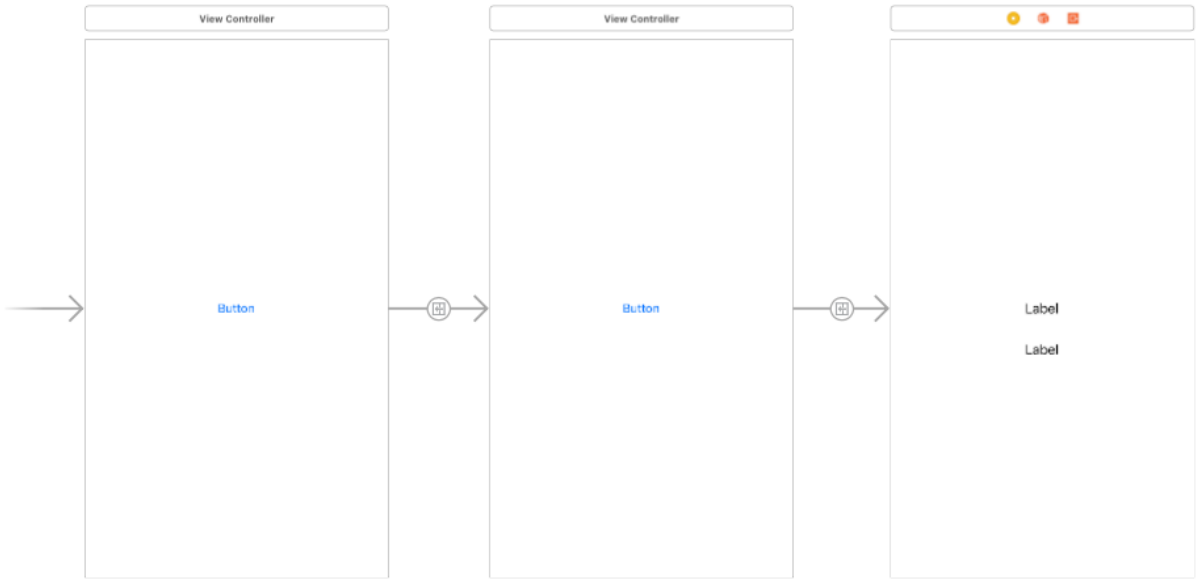
Из этого следует две проблемы:

1. Мы можем создавать несколько экземпляров класса, хотя они нам не нужны**.**
2. Необходимо передавать экземпляр с данными между экранами и сервисами приложения.

Первую проблему можно решить просто: пообещаем себе, что не будем создавать больше одного экземпляра класса. А передавать его – и только его! – в каждый сервис будем в конструкторе.

Но обещания, особенно данные самому себе, – вещь зыбкая, а перетаскивать класс через все приложение просто неудобно. Более надежным и эргономичным решением является **singleton** – это класс (в нашем случае класс **Session)**, создавать объекты которого запрещено на уровне языка. Также мы сможем получить к нему доступ в любой точке приложения и в любой момент.

Создадим небольшое приложение-пример. Оно будет содержать три экрана.



На первых двух будут кнопки перехода на следующий экран, а на третьем – два **UILabel**: для отображения ФИО и баланса.

На первом экране мы будем записывать данные о пользователе, а на последнем отображать.

У первого контроллера есть класс, по умолчанию созданный **xcode** – **ViewController**. Для второго контроллера, существующего для усложнения примера, класс не нужен. Для третьего создадим класс **LastViewController** и протянем в него **IBOutlet** для лейблов.

```
@IBOutlet weak var nameView: UILabel!
@IBOutlet weak var moneyView: UILabel!
```

Создадим класс **Session** с тремя свойствами.

```
class Session {
    var fio = ""
    var id = 0
    var money = 0
}
```

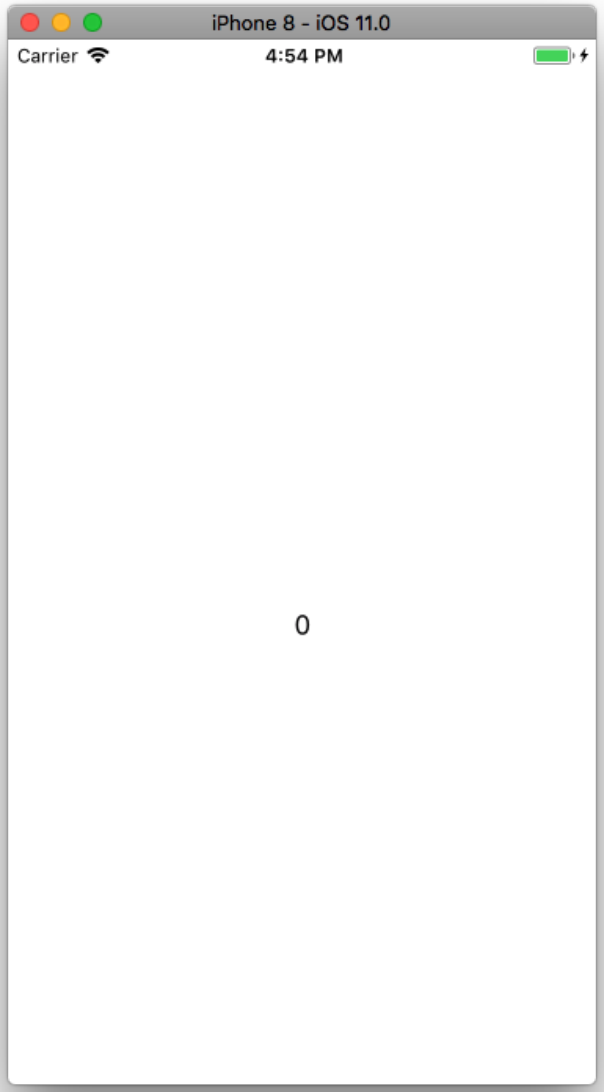
Создадим его экземпляр и заполним данными в первом контроллере.

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

        let session = Session()
        session.fio = "Иванов Иван Иванович"
        session.id = 1
        session.money = 9999

    }
```

Теперь откроем третий контроллер, создадим экземпляр, прочитаем его свойства и выведем информацию на экран.

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

        let session = Session()
        nameView.text = session.fio
        moneyView.text = String(describing: session.money)

    }
```

Запускаем, нажимаем кнопки для перехода и видим, что информации на экране нет. Это закономерно: ведь объект, созданный на третьем контроллере, никак не связан с объектом на первом. А в нем самом информации нет.



Переделаем наш класс в singleton. Сначала сделаем конструктор приватным – это запретит создание экземпляров класса.

```
private init(){}
```

Затем создадим в классе статическую константу с экземпляром этого же класса. Таким образом, сам класс будет хранить в себе свой же единственный объект.

```
static let instance = Session()
```

Полный листинг класса:

```
class Session {

    static let instance = Session()

    private init(){}

    var fio = ""
    var id = 0
    var money = 0
}
```

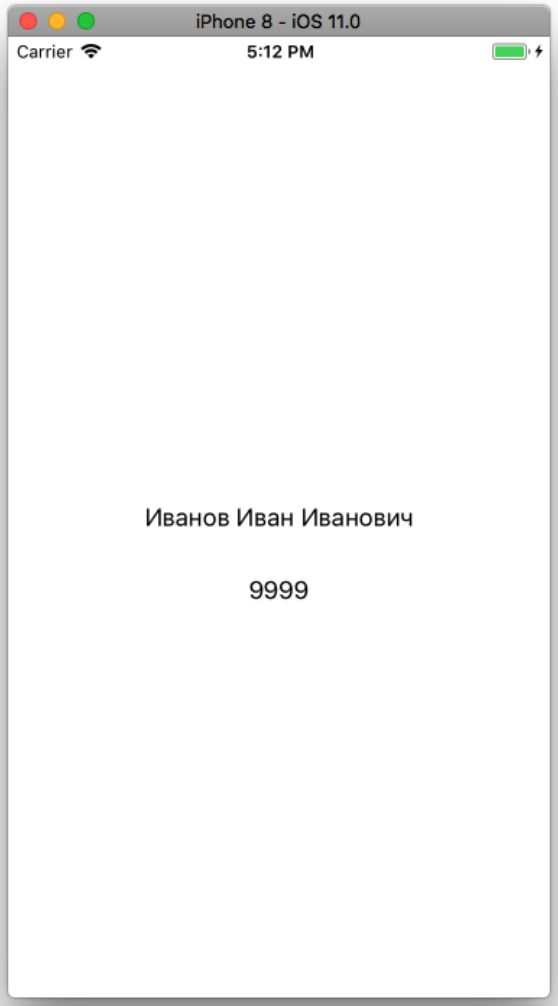
Теперь у нас в проекте ошибки: в двух контроллерах пытаемся создать экземпляры классов, а это уже невозможно. Исправляем эти строки кода на обращение к свойству **instance**. Начнем с первого контроллера.

```
let session = Session.instance
session.fio = "Иванов Иван Иванович"
session.id = 1
session.money = 9999
```

И повторим на последнем.

```
let session = Session.instance
nameView.text = session.fio
moneyView.text = String(describing: session.money)
```

Теперь на обоих контроллерах мы обращаемся к одному и тому же объекту. Запустим проект и посмотрим на результат.



Информация верная.

Синглтоны очень удобны при написании кода. Они позволяют не думать о передаче информации в разные участки приложения. Но этими возможностями надо пользоваться ответственно: думать, действительно ли класс должен существовать в единственном экземпляре. Кроме того, синглтон увеличивает вероятность ошибок в приложении. С ним слишком легко работать: можно вызывать его из любой строки кода, менять его значения, и со временем потерять понимание, кто положил в него данные и кто их читает. Это риск потери контроля над кодом.

Observer – «наблюдатель». NotificationCenter

Следующий паттерн – «наблюдатель» (**Observer**). В частности, он реализуется как стандартный паттерн в SDK **NotificationCenter**.

Представим два объекта — покупатель и магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю. Он может каждый день ходить в магазин, проверять

наличие товара, тратить на это время и злиться. Магазин может задать рассылку всем покупателям – и многие будут недовольны, так как товар специфический и не всем он нужен. Получается конфликт – либо один объект тратит ресурсы на периодические проверки, либо второй оповещает слишком широкий круг пользователей, тоже теряя в эффективности.

Другими словами, есть объект, в котором должно произойти событие, и есть множество других объектов, и некоторые из них желают узнать о его наступлении. Решается эта проблема просто: все желающие регистрируют свой интерес, подписываясь на это событие.

В коде это выглядит очень просто:

```
struct WeakBuyer {
    weak var buyer: Buyer?
}

class Shop {                        // массив, где хранятся все подписчики
    var listeners = [WeakBuyer]()
  func subscribe(buyer: Buyer) { // регистрация подписчиков
        let weakBuyer = WeakBuyer(buyer: buyer)
        listeners.append(weakBuyer)
    }
    func notify() {                 // уведомляем всех подписчиков
        listeners.forEach{ $0?.takeInfo() }
    }

}

class Buyer  {                      // покупатель
    func takeInfo() {               // метод, который реагирует на уведомления

    }
}
let shop = Shop()                   // магазин

let buyer1 = Buyer()                // три покупателя
let buyer2 = Buyer()
let buyer3 = Buyer()
shop.subscribe(buyer: buyer1)   // первый покупатель регистрируется для
получения уведомлений
shop.subscribe(buyer: buyer2)   //второй покупатель регистрируется для получения
уведомлений
```

Здесь класс магазина имеет массив, где хранится список подписчиков, и метод **subscribe** для их добавления. Метод **notify** вызывается для уведомления подписчиков. По факту, он просто проходит по массиву и вызывает у всех элементов их метод **takeInfo**, которым обладают «покупатели». В итоге мы создаем магазин с тремя покупателями, двоих из которых подписываем на событие, а третьего нет. В нужный момент метод **takeInfo** будет вызван только у первых двух.

Это один из способов решения похожих задач. Но в iOS есть **NotificationCenter** – это частный случай реализации паттерна «наблюдатель». Он необходим, чтобы отправлять уведомления из одной точки кода в любые другие.

Вы уже сталкивались с ним, когда на первом занятии подписывались на уведомления о появлении клавиатуры. Теперь поговорим, как направлять свои уведомления и подписываться на них.

Если создать свой экземпляр класса **NotificationCenter**, в нем будут работать только ваши уведомления, и придется думать, как доставить его в точку получения уведомлений. Поэтому проще воспользоваться центром по умолчанию.

```
NotificationCenter.default
```

В свойстве default центр доступен из любой точки приложения: можно отправить в него уведомление или подписаться на получение определенных сигналов.

Создадим небольшой тестовый проект. Он будет содержать два контроллера: **UIViewController** и **UINavigationController**. На первом будет кнопка перехода на второй, а на последнем – кнопка для отправки уведомления.

На втором контроллере будем отправлять уведомление, а первый будет подписываться на его получение и реагировать сменой цвета экрана.

Так как через центр могут отправляться любые уведомления, их необходимо различать. Для этого уведомлениям присваиваются имена, представленные простой структурой **Name,** описанной в классе **Notification**. Создадим уведомление для отправки.

```
let someNotification = Notification.Name("someNotification")
```

Теперь можем отправлять его в центр уведомлений – например, по нажатию на кнопку.

```
@IBAction func sendNotificationPressed(_ sender: Any) {
        let someNotification = Notification.Name("someNotification")
        NotificationCenter.default.post(name: someNotification, object: nil)
}
```

В этот момент центр найдет всех заинтересованных в получении уведомления подписчиков с именем **someNotification** и вызовет у них необходимый метод.

Подпишем первый контроллер на получение уведомлений. Сделаем это в методе **viewDidLoad**. Подписаться можно в любой момент, но в этом примере пусть контроллер получает уведомления с момента появления.

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

        let someNotification = Notification.Name("someNotification")
        NotificationCenter.default.addObserver(self, selector:
#selector(changeColor(notification:)), name: someNotification, object: nil)
    }
```

Разберем метод **addObserver**. Первый аргумент, который он принимает, – это объект-подписчик, в данном случае – сам контроллер. Второй аргумент – это селектор на метод подписываемого объекта. В нашем случае – метод **changeColor**. Третий аргумент – имя уведомления. Последний мы опустили, но там при желании можно указать объект, который отправляет уведомление. Такой же аргумент есть и у метода отправки уведомлений, так что можно организовать получение уведомлений только при совпадении объектов.

Метод изменения цвета выглядит так:

```
@objc func changeColor(notification: Notification) {
        view.backgroundColor = .red
}
```

Если уведомления перестали быть полезными, отписываемся от них в методе **deinit**.

```
deinit {
        let someNotification = Notification.Name("someNotification")
        NotificationCenter.default.removeObserver(self, name: someNotification,
object: nil)
    }
```

Здесь снова требуется имя уведомления. Отписка вызывается методом **removeObserver**. В качестве аргументов передаются объект, который прекращает подписку, имя уведомления и объект-маркер.

Запускаем проект: переходим на второй контроллер, нажимаем кнопку, возвращаемся на первый и видим, что цвет изменился.

Еще одна возможность **NotificationCenter** – передача данных вместе с уведомлением. Для этого существует специальный метод отправки уведомлений с информацией. Зададим таким образом цвет, в который будет окрашиваться первый экран.

```
NotificationCenter.default.post(name: someNotification, object: nil, userInfo:
["color": UIColor.blue])
```

Информация передается в аргументе **userInfo**. Это словарь типа **[AnyHashable : Any]**. Эти сведения мы можем получить в методе **changeColor**.

```
@objc func changeColor(notification: Notification) {
        let color = notification.userInfo?["color"] as? UIColor ?? UIColor.red
        view.backgroundColor = color
    }
```

Работать с центром уведомлений легко, при этом он позволяет передавать данные в любое место приложения и даже в нескольких направлениях одновременно. Главное – не увлекаться его использованием. Убедитесь, что уведомление действительно должно быть распространено по всему приложению, иначе лучше выбрать другой способ передачи информации.

Delegate – «делегат»

Паттерн «делегат» (**Delegate**) – один из самых популярных в iOS. Мы уже упоминали его при работе с таблицами и использовали для написания игры. Теперь рассмотрим его подробно.

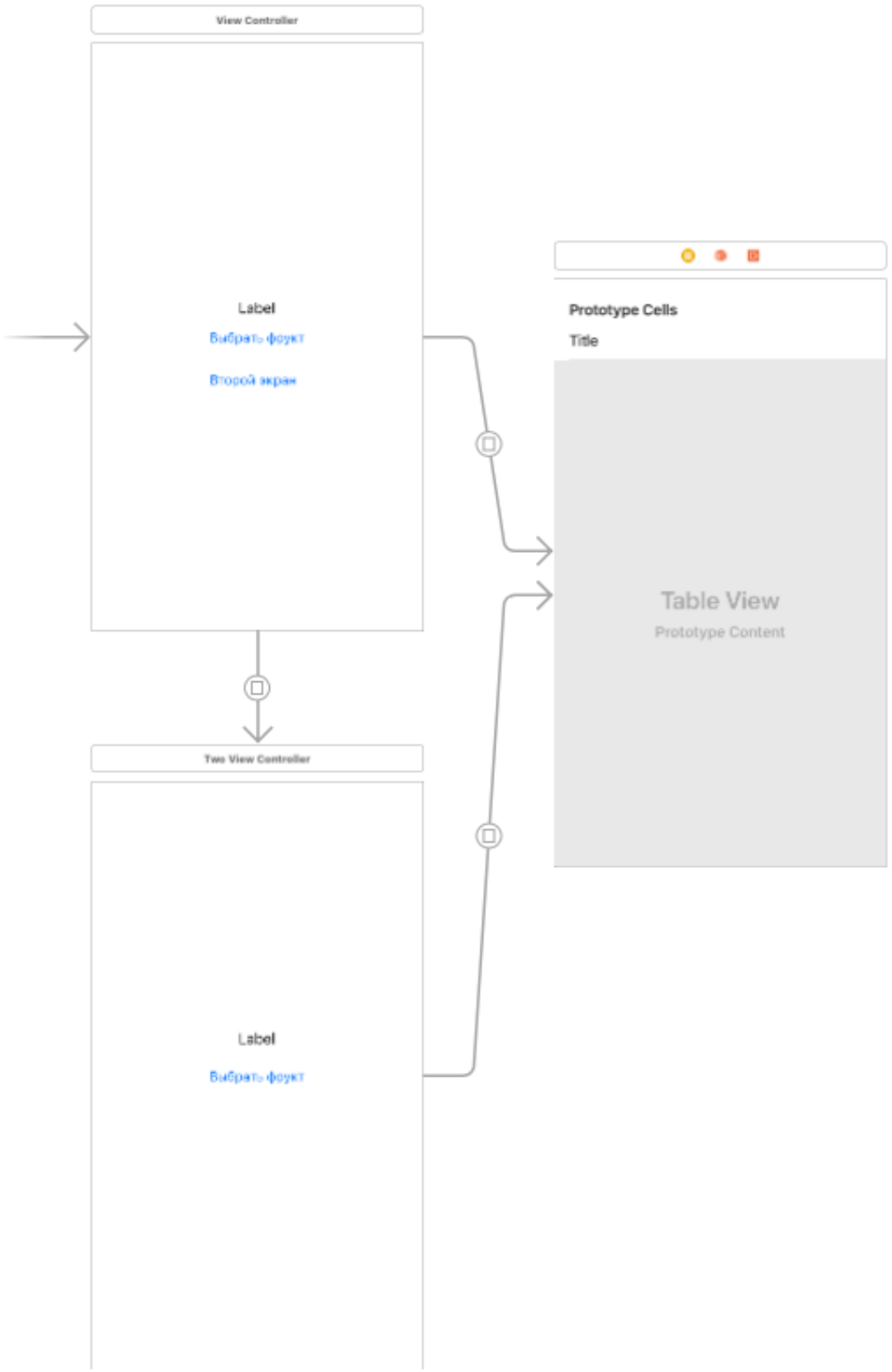
Делегат – это класс, выполняющий работу за других. Как секретарь, действующий по поручению и пользующийся делегированными полномочиями начальника.

Еще одно назначение делегата – вынести часть логики из класса. Как правило, это логика, которая часто меняется. Например, в стандартном элементе **UITextField** есть много методов, которые вызываются при редактировании текста. Но для каждого конкретного текстового поля их логика различается. Поэтому эти методы вынесены в класс «делегат», который мы и будем менять, если

потребуется изменить логику обработки текста. Иначе пришлось бы создавать «наследника» для каждого текстового поля и переопределять методы.

В UIKit много классов, имеющих делегаты. Чаще всего они пишутся при взаимодействии двух контроллеров. Раньше эта проблема решалась через **unwindSegue**, и при работе со **storyboard** этот вариант остается предпочтительным. Но он не единственный: можно использовать **delegate**.

Напишем простой пример с тремя контроллерами.



Первый контроллер отображает фрукт, который можно выбрать на экране с иконками. С него можно перейти на второй контроллер с тем же функционалом. Это упрощенный вариант распространенной ситуации, когда два разных экрана обращаются за выбором к третьему. В реальности первый контроллер мог бы быть формой регистрации, второй – формой оформления заказа, а третий – списком городов. Пользователь выбирал бы город и при регистрации, и при оформлении доставки.

Первый контроллер будет иметь класс **ViewController**, второй – **TwoViewController**, а третий – **AppleViewController**.

Напишем простой код отображения фруктов в таблице.

```
class AppleViewController: UITableViewController {

    let apple = ["Яблоко", "Персик", "Киви"]

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

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        // #warning Incomplete implementation, return the number of sections
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        // #warning Incomplete implementation, return the number of rows
        return apple.count
    }


    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "AppleCell",
for: indexPath)
        cell.textLabel?.text = apple[indexPath.row]
        return cell
    }

}
```

Добавим код нажатия на ячейку.

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
     print(apple[indexPath.row])
}
```

Пока мы просто выводим в консоль название фрукта. Третьему контроллеру не надо ничего делать с фруктами. Выбор важен либо первому, либо второму. Они и будут обрабатывать нажатие на ячейку. В этом поможет делегат.

Перейдем в первый контроллер и напишем метод выбор фрукта.

```
class ViewController: UIViewController {

    @IBOutlet weak var labelView: UILabel!

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

    func fruitDidSelect(_ fruit: String) {
        labelView.text = fruit
    }

}
```

Метод принимает строку и устанавливает ее в лейбл. Вызовем его на контроллере с фруктами.

```
class AppleViewController: UITableViewController {

    let apple = ["Яблоко", "Персик", "Киви"]

    weak var delegate: ViewController?
    <...>
    override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
        let fruit = apple[indexPath.row]
        delegate?.fruitDidSelect(fruit)
        dismiss(animated: true, completion: nil)
    }

}
```

Добавляем свойство для хранения ссылки на первый контроллер. При нажатии на ячейку вызываем метод **fruitDidSelect**. Здесь первый контроллер является делегатом контроллера с фруктами.

Чтобы пример заработал, осталось только установить при переходе значения для свойства делегата.

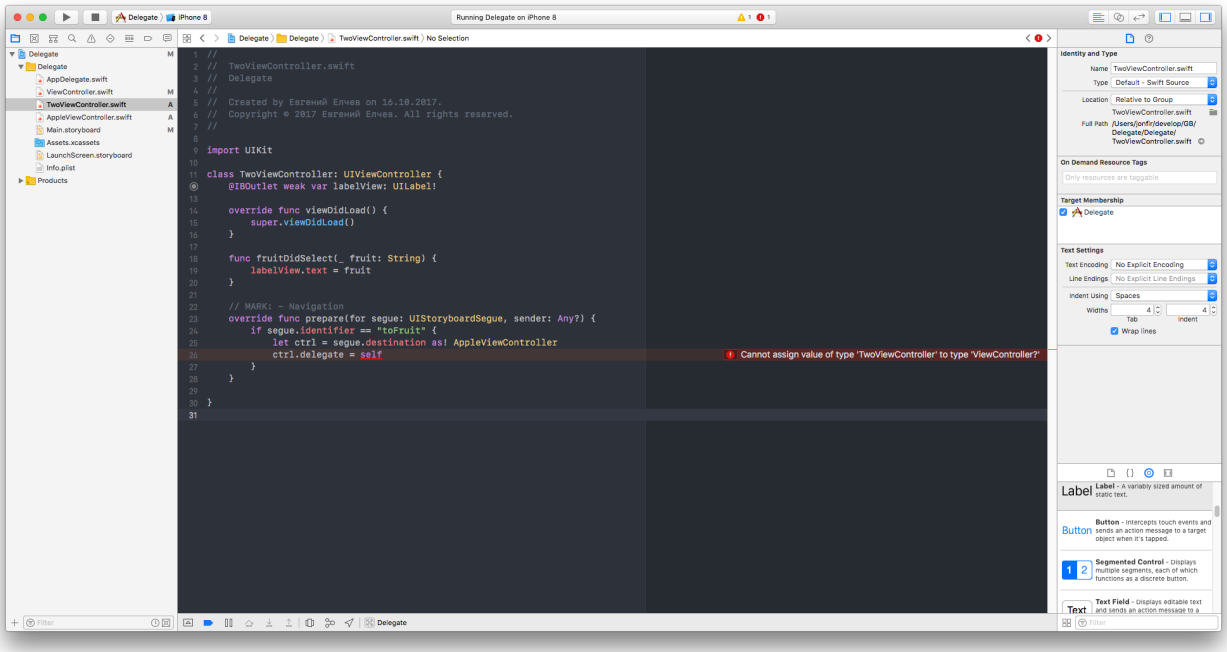
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "toFruit" {
            let ctrl = segue.destination as! AppleViewController
            ctrl.delegate = self
        }
    }
```

Запускаем проект с первого экрана и убеждаемся, что выбор фрукта работает.

Теперь повторяем все со вторым контроллером.

```
class TwoViewController: UIViewController {
    @IBOutlet weak var labelView: UILabel!

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

    func fruitDidSelect(_ fruit: String) {
        labelView.text = fruit
    }

    // MARK: - Navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
            let ctrl = segue.destination as! AppleViewController
            ctrl.delegate = self
    }

}
```

Но нас ожидает ошибка.



Проблема в том, что свойство делегата «фруктового» контроллера завязано на тип **ViewController** и не может работать со вторым контроллером типа **TwoViewController**. Здесь мы узнаем, что ***«делегаты» определяются протоколами***, и нужно написать свой.

```
protocol AppleViewControllerDelegate: class {
    func fruitDidSelect(_ fruit: String)
}
```

У делегата только один метод. Изменим тип свойства делегата.

```
weak var delegate: AppleViewControllerDelegate?
```

Имплементируем протоколы в контроллеры.

В первый:

```
extension ViewController: AppleViewControllerDelegate {

    func fruitDidSelect(_ fruit: String) {
        labelView.text = fruit
    }

}
```

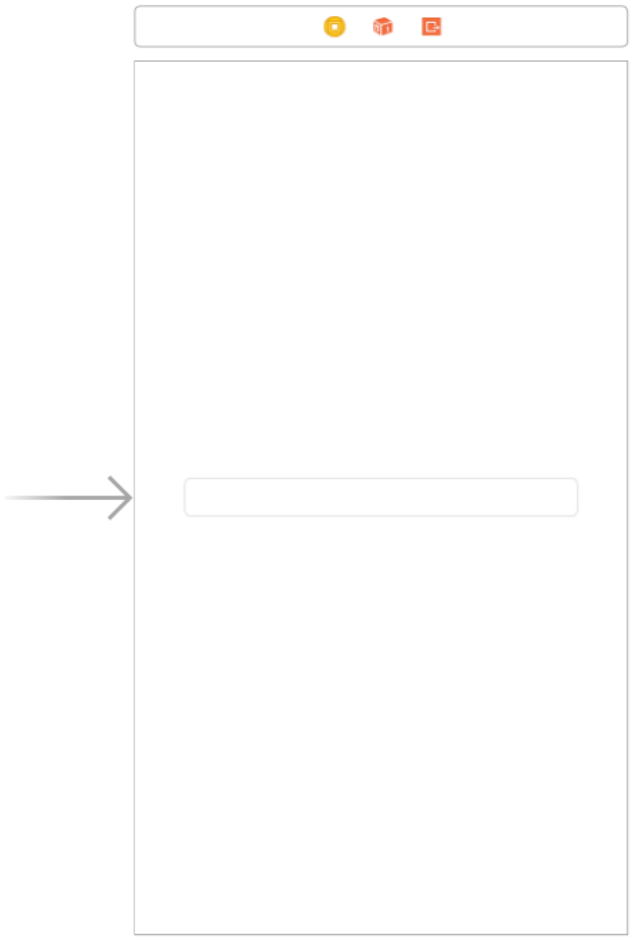
И во второй:

```
extension TwoViewController: AppleViewControllerDelegate {

    func fruitDidSelect(_ fruit: String) {
        labelView.text = fruit
    }

}
```

В результате оба контроллера могут выступать в роли делегата для **AppleViewController**.

При работе с готовыми классами UIKit часто встречаются классы с делегатами. Все они имеют протоколы, которые надо имплементировать какому-либо классу. После этого можно установить класс делегатом. Не идеальный, но самый простой способ – назначать делегатом UIViewController.

Рассмотрим еще один простой пример: контроллер с одним текстовым полем.



Для него создан **IBOutlet** в контроллер.

```
class ViewController: UIViewController {

    @IBOutlet weak var textView: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()
    }

}
```

Чтобы реагировать на действия с текстом в этом поле, нужно создать для него делегата. Им будет контроллер.

```
extension ViewController: UITextFieldDelegate {

}
```

Теперь надо выбрать необходимые методы. Ознакомиться со списком методов можно в документации к протоколу UITextFieldDelegate. Допустим, нам интересно «отлавливать» момент, когда пользователь выделяет поле, чтобы изменить текст. Для этого есть метод **textFieldDidBeginEditing**.

```
extension ViewController: UITextFieldDelegate {

    func textFieldDidBeginEditing(_ textField: UITextField) {
        print(textField.text)
    }
}
```

Осталось только установить контроллер в качестве делегата для поля.

```
@IBOutlet weak var textView: UITextField! {
        didSet {
            textView.delegate = self
        }
    }
```

Практическое задание

1. Добавить в проект синглтон для хранения данных о текущей сессии – Session
2. Добавить в него свойства:
    a. token: String – для хранения токена в VK,
    b. userId: Int – для хранения идентификатора пользователя ВК.

Дополнительные материалы

1. https://developer.apple.com/documentation/foundation/nsnotificationcenter

Используемая литература

Для подготовки данного методического пособия использовались следующие материалы:

1. https://developer.apple.com/documentation