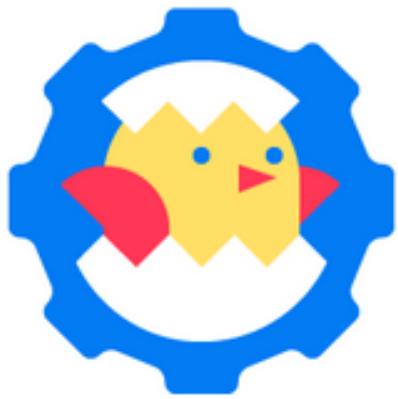


# Разработка под iOS. Начинаем

Часть 4

## View Controllers



# Оглавление

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Разработка под iOS. Начинаем</b>                       | <b>2</b> |
| 1.1      | Класс UIViewController . . . . .                          | 2        |
| 1.1.1    | Что такое UIViewController . . . . .                      | 2        |
| 1.1.2    | Перемещение между контроллерами . . . . .                 | 3        |
| 1.1.3    | Жизненный цикл класса UIViewController . . . . .          | 6        |
| 1.2      | Архитектура MVC . . . . .                                 | 10       |
| 1.2.1    | Шаблон MVC . . . . .                                      | 10       |
| 1.2.2    | MVC на примере приложения адресной книги . . . . .        | 12       |
| 1.3      | Класс UINavigationController . . . . .                    | 17       |
| 1.3.1    | Что такое UINavigationController . . . . .                | 17       |
| 1.3.2    | UINavigationController на практике . . . . .              | 21       |
| 1.3.3    | Другие полезные свойства UINavigationController . . . . . | 24       |
| 1.4      | Класс UITabBarController . . . . .                        | 28       |
| 1.4.1    | Что такое UITabBarController . . . . .                    | 28       |
| 1.4.2    | UITabBarController на практике . . . . .                  | 29       |
| 1.5      | Другие полезные контейнеры . . . . .                      | 32       |
| 1.6      | Что такое Storyboard . . . . .                            | 35       |
| 1.6.1    | Теория о Storyboard . . . . .                             | 35       |
| 1.6.2    | Storyboard на практике . . . . .                          | 36       |
| 1.7      | Класс UIScrollView . . . . .                              | 40       |
| 1.7.1    | Теория о UIScrollView . . . . .                           | 40       |
| 1.7.2    | Как настраивать UIScrollView . . . . .                    | 44       |
| 1.7.3    | Дополнительные возможности UIScrollView . . . . .         | 46       |
| 1.8      | Класс UITableView . . . . .                               | 48       |
| 1.8.1    | Принципы работы с классом UITableView . . . . .           | 48       |
| 1.8.2    | UITableView на практике . . . . .                         | 52       |
| 1.8.3    | Как работать с ячейками таблицы . . . . .                 | 54       |
| 1.8.4    | Другие важные особенности таблицы UITableView . . . . .   | 57       |
| 1.8.5    | Другие важные особенности таблицы UITableView . . . . .   | 62       |
| 1.9      | Собственные контейнер-контроллеры . . . . .               | 65       |

# Глава 1

## Разработка под iOS. Начинаем

### 1.1. Класс UIViewController

#### 1.1.1. Что такое UIViewController

На прошлых лекциях вы научились создавать интерфейсы, а сейчас разберемся, как из этих интерфейсов сделать полноценное приложение. Представим себе программу для iPhone, которую используете каждый день, например, контактную книгу. Эта программа содержит закладки, между которыми можно переключаться, или заходить вглубь на некоторых из них, например, чтобы посмотреть информацию о друге или о пропущенном звонке. По сути, это приложение состоит из отдельных статичных экранов. Каждый экран здесь выполняет конкретную функцию, и как правило, не влияет на другие экраны. Не трудно догадаться, что один такой экран состоит из знакомых графических элементов, и ещё какого-то кода, который позволяет этим элементам взаимодействовать с данными приложения. Можно сказать, что код делает так, чтобы элементы на одном экране обрабатывали то, что делает пользователь, а также взаимодействовали друг с другом. Такое разделение экранов приложения на графику и логику стало настолько распространённым, что Apple рекомендует всегда выделять логику управления в отдельный класс, и создала для этого UIViewController.

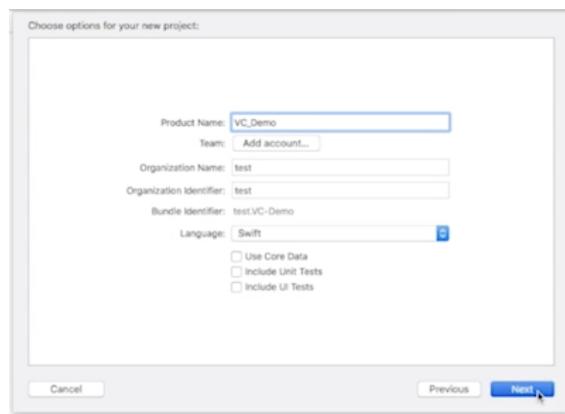
Теперь, чтобы добавить новый экран в приложение, мы создаём собственный класс, наследуем его от класса UIViewController, и пишем туда нужный код. Правда, иногда надо ещё настроить графику в редакторе интерфейсов.

Но зачем необходимо это наследование? Дело в том, что класс UIViewController тесно связан с самой операционной системой iOS. А функции в этом классе достаточно сильно упрощают жизнь. Например, помогают переключаться между экранами приложения, следить за жизненным циклом вью, или управлять переключением между ландшафтным и портретным режимами. И ещё многое чего другого. Также в iOS есть больше количества классов, реализующих уже готовые решения разных задач. Многие из них представляют полноценный экран и унаследо-

ваны от UIViewController. Например, UI-Image-Picker-Controller позволяет пользователю делать фотографии, или выбирать их из галереи телефона. И так, класс UIViewController позволяет разделить логику одного экрана от интерфейса и сделать удобными переходы между экранами. И ещё один бонус: наши классы с логикой могут быть переиспользованы в других местах где это потребуется.

### 1.1.2. Перемещение между контроллерами

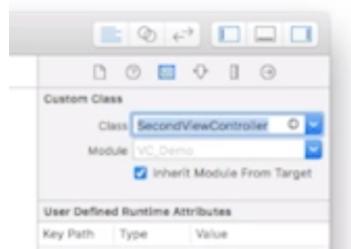
Теперь разберемся как перемещаться между экранами в приложении. Сейчас создадим новый проект. Для этого используем шаблон Single View App и дадим ему имя – VC\_Demo\_1. Галочки с Use Core Data, UI и UX тестов лучше снять чтобы Xcode не создавал не нужные файлы. Нажимаем Next и создаём проект.



Xcode уже создал первый вид контроллер. С него начнется выполнение программы. Графика этого контроллера настраивается в Main.storyboard. Добавим второй экран. Для этого создаём новый файл и выбираем тип Cocoa Touch Class. Здесь даём новому виду контроллеру понятное имя, и указываем сабкласс – SecondViewController.

Если галочка Also create XIB file выделена, то будет создан файл, в котором можно редактировать интерфейс нового контроллера. Проверим это. Заканчиваем создание виду контроллера.

Если всё сделали правильно, Xcode создаст 2 файла. В первом будет уже знакомый шаблон класса на языке Swift. Обратите внимание, что предок здесь – UIViewController. Второй – это интерфейсный файл, или Xib. Этот файл связан с нашим новым виду контроллером. Это просто проверить. Раскрываем список элементов в интерфейсе, иконка находится слева внизу. Выделяем File's owner и раскрываем в окне справа Identity Inspector.



Поле Class – это тот тип данных, который представляет выбранный элемент. Здесь должно быть имя нашего нового вью контроллера. Значение в поле можно изменить, и тогда интерфейс будет принадлежать другому вью контроллеру.

Теперь давайте добавим контент, чтобы отличить этот контроллер от остальных. Например, лейбл с текстом. Добавим необходимые констрайнты. Вью готова. Теперь нужно его показать вью контроллеру. Переайдём в стартовый вью контроллер и добавим туда действие:

```
@IBAction func showSecondViewController(_ sender: Any) {  
}
```

Напомню, что IBAction в начале функции делает её видимой в редакторе интерфейса. Итак, чтобы показать новый экран, нам надо создать его объект и вызвать специальный метод present. Создадим SecondViewController:

```
let secondViewController = SecondViewController(nibName: "SecondViewController",  
bundle: nil)
```

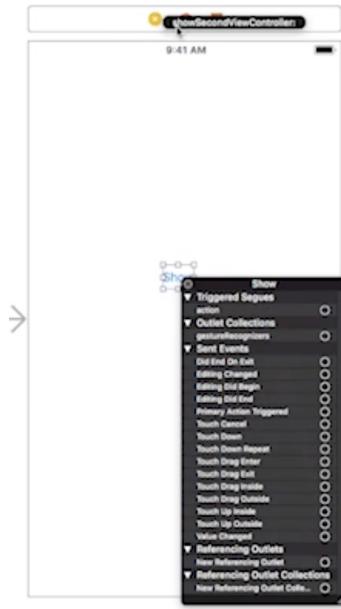
Nib name указывает имя файла, из которого будет загружаться интерфейс для вью контроллера. Пусть вас не смущает, что в проекте интерфейсные файлы имеют расширение XIB. При компиляции, Xcode перекодирует их в файлы с расширением NIB. Кстати, если имя XIB файла и вью контроллера совпадают, то этот параметр можно опустить, так же как и Bundle:

```
- let secondViewController = SecondViewController()
```

И теперь вызовем метод Present:

```
present(secondViewController, animated: true, completion: nil)
```

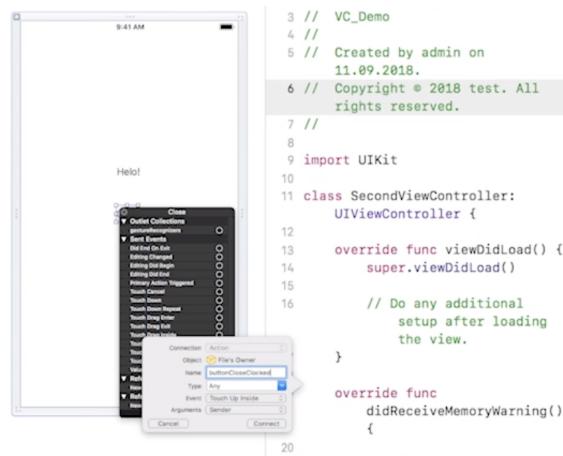
В качестве первого параметра передаём secondVlewController. Второй параметр – это animated. Он позволяет включать или выключать анимацию при появлении нового экрана. В completion можно передать замыкание, которое будет вызвано когда анимация закончится. Осталось добавить кнопку с кодом появления этого экрана. Открываем сториборд. Добавляем кнопку. Теперь связываем событие кнопки Touch Up Inside с нашим действием.



Собираем и запускаем проект. И нажмём на появившуюся кнопку. Отлично. Открылся новый экран. Как и хотели, он представлен классом SecondViewController. Теперь попробуем добавить кнопку, чтобы закрыть этот экран и вернуться назад в место старта.

Действия можно также связывать с использованием Assistant Editor. Он включается кнопкой сверху. Теперь в окне справа у нас появился экран с кодом. Обратите внимание, что Xcode автоматически выбирает нужный файл. Если этого не произошло, нужный файл можно выбрать в меню сверху.

Чтобы добавить действие, вызываем контекстное меню и перетаскиваем событие Touch Up Inside во вью контроллер. Назовём метод buttonCloseClicked.



Так получаем готовый для использования метод. Выходим из совместного режима и переключаемся к коду SecondViewController. Чтобы скрыть последний добавленный экран, вызовем dismiss:

```
dismiss(animated: true, completion: nil)
```

Запускаем проект. Теперь кнопка Close работает как надо. А мы узнали как создавать и показывать новые экраны.

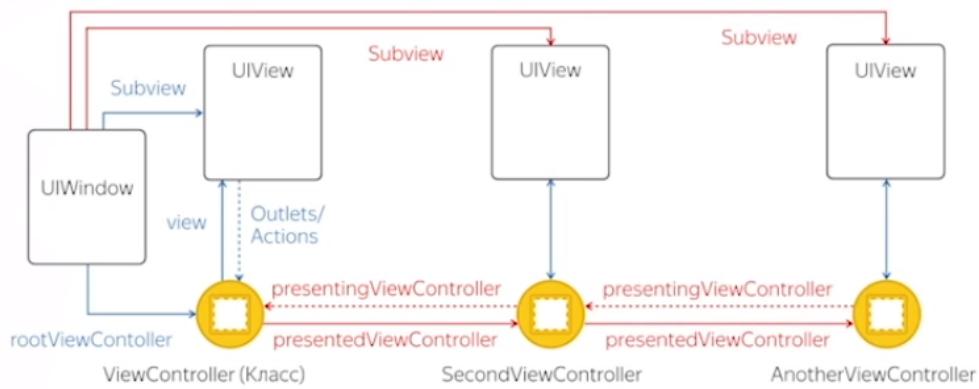
### 1.1.3. Жизненный цикл класса UIViewController

Давайте разберемся что всё-таки происходит внутри нашего приложения. Оно настроено таким образом, что использует Storyboard для запуска первого вью контроллеров. Пока не будем детально разбирать механизм работы Storyboard'ов, но нам важно знать, что при старте storyboard создаёт экземпляр класса UIWindow и делает его ключевым. Кстати, ссылку на объект Window можно найти в классе AppDelegate. Эта ссылка инициализируется автоматически после запуска приложения. Ещё storyboard механизм создаёт стартовый экран – это объект класса ViewController. Не надо путать его с базовым системным классом UIViewController. Просто так вышло, что в шаблоне Xcode у первого пользовательского экрана такое название – ViewController.

Теперь вспомним предыдущие занятия: чтобы вью стала видимой, мы должны добавить её в графическую иерархию, которая начинается с Window. Здесь интерфейс первого экрана будет добавлен как subview на window. А для ссылки на первый ViewController есть специальное свойство – rootViewController. Сюда можно присвоить любой ViewController, который захотите, и он станет корневым экраном. Теперь мы хотим показать новый экран. Перейдём к коду события showSecondViewController. Сначала создаём вью контроллер второго экрана. Он создаётся в паре со своим view, которая загружается из XIB файла. Потом вызываем метод present. В первом контроллере автоматически назначается ссылка presentedViewController на второй экран. А в ответ

создаётся слабая ссылка presentingViewController от второго к первому. Вью нового добавленного контроллера размещается на корневом window. Таким образом, второй вью контроллер становится видимым. При этом window может удалить первый экран из списка subviews, чтобы сэкономить ресурсы. Если после этого мы покажем третий вью контроллер, то он добавится аналогичным образом, так же как и второй.

```
@IBAction func showSecondViewController(_ sender: Any) {
    let secondViewController = SecondViewController()
    present(secondViewController, animated: true, completion: nil)
}
```



Перед тем как перейти к практике, давайте познакомимся ещё с одним термином. Это **UIViewController Life Cycle**, или жизненный цикл вью контроллера.

```
func viewDidLoad()
func viewWillAppear(_ animated: Bool)
func viewDidAppear(_ animated: Bool)
func viewWillDisappear(_ animated: Bool)
func viewDidDisappear(_ animated: Bool)
```

Фактически, это набор из пяти методов, используя которые, ваш класс сможет обрабатывать изменение состояния вью контроллера. Каждый из них вызывается в момент наступления того или иного события. И, чтобы подписаться на это событие, вам просто надо перегрузить соответствующий метод.

```

class CustomViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Ваш код
    }
    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        // Ваш код
    }
    ...
}

```

Итак, мы создаём виду контроллер. Первый метод в цепочке может быть либо один из методов init или init(nibName: bundle:), а если мы загружаем весь виду контроллер из XIB или из Storyboard, вызывается awakeFromNib. В этот момент виду контроллер находится в состоянии Unloaded, то есть его интерфейс пока ещё не загружен. Важно знать, что в этот момент все графические элементы интерфейса ещё не созданы, а ссылки на объекты, которые должны быть доступны через IBOutlet, ещё не назначены. При попытке обратиться к ним скорее всего произойдут ошибки и ваше приложение упадёт.

```

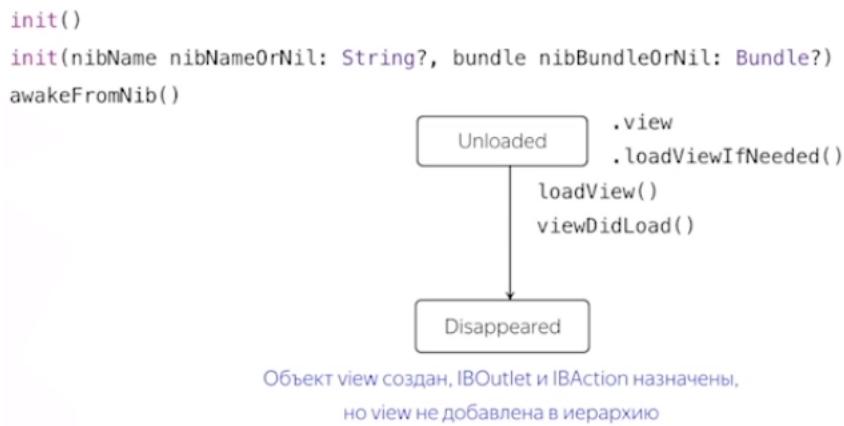
init()
init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: Bundle?)
awakeFromNib()

```

Unloaded

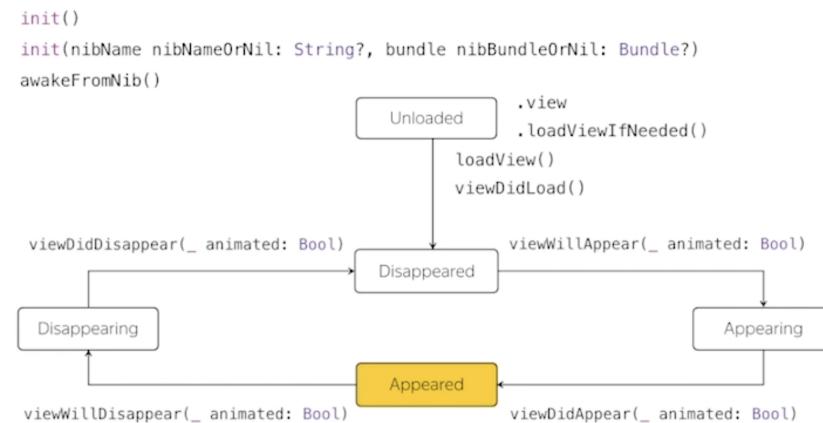
Объект view не создан, IBOutlet и IBAction не назначены

Следующий этап. Если кто-то попытается получить доступ к свойству viewController.view, или вызовет метод loadViewIfNeeded(), это инициирует процесс загрузки виду контроллера. Сначала механизм попытается вызвать метод loadView(). Мы так же можем переопределить его, чтобы изменить процесс загрузки. Но лучше пока этого не делать. По умолчанию этот метод загружает интерфейсный файл, создаёт все view, кнопки и прочие элементы, которые присутствуют в нём, потом назначает все связи, которые мы настроили через IBOutlet и IBAction. И после того, как всё будет загружено, вызывается метод viewDidLoad(). Это идеальное место для того, чтобы закончить настройку интерфейса: добавить и разместить все остальные элементы, которые не были добавлены в интерфейсном файле. Важно не забыть здесь вызвать super.viewDidLoad() для того, чтобы класс-предок нашего виду контроллера так же был проинициализирован.



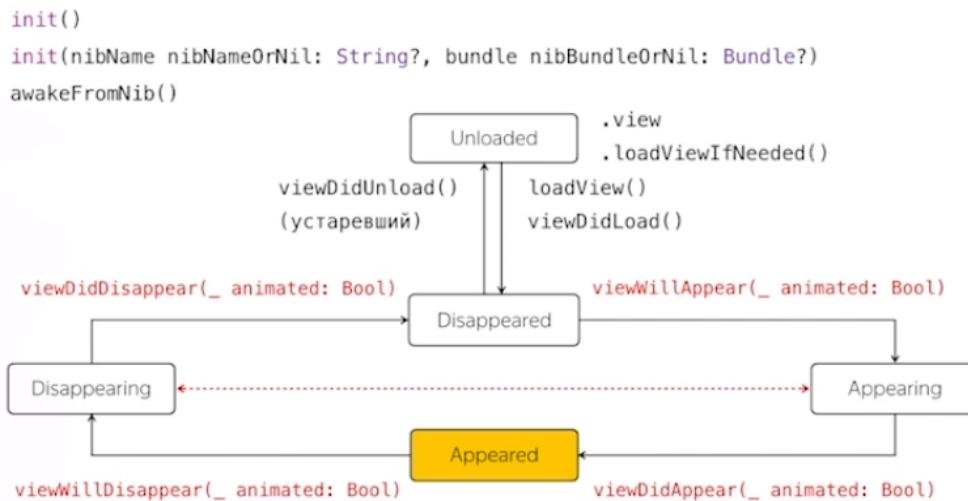
Следующее событие наступает, когда система пытается показать виду контроллер, но до того как view будет добавлена в графическую иерархию. Это событие viewWillAppears. В этот момент виду не видима на экране, но скоро будет. Здесь, как правило, настраивается весь контент, который будет виден пользователю. Например, текст на Label или на кнопках, или же цвета, которые мы не можем указать в Interface Builder. Важно не привязываться к размерам view в этот момент, потому что эти размеры ещё не финальные. Если вы правильно настроили констрайнты, то проблем возникнуть не должно.

В следующем состоянии appearing, как правило, происходит анимация, при которой view нашего контроллера становится видимой. По завершению этой анимации вызывается метод viewDidAppear. Теперь наш контроллер вошел в активное состояние и продолжит в нём находится до тех пор, пока не понадобится его спрятать.



Аналогично viewWillDisappear и viewDidDisappear будут вызваны, когда виду контроллер исчезает с экрана. Когда системе не хватает оперативной памяти, она может удалять некоторые

графические элементы, и вью контроллер перейдёт обратно в изначальное состояние Unloaded. Однако этот механизм на данный момент считается устаревшим и не используется на практике. И последний небольшой нюанс. Не всегда жизненный цикл вью контроллера изменяется в том направлении как показано на слайде. Иногда интерактивные жесты, которые должны открыть предыдущий экран, не заканчиваются до конца. Вы и сами можете это попробовать. Зайдите в какое-нибудь приложение, например, настройки. Потом зайдите вглубь на любой экран, а потом попытайтесь вернуться назад свайпом от левой границы экрана направо. Если на середине этого свайпа не отпускать палец, и опять повести его влево, то действие перехода будет отменено. В терминах View Controller Life Cycle это означает, что верхний экран начал исчезать, но не сделал это до конца. Поэтому для него будет вызван сначала метод viewWillDisappear, а затем сразу же viewWillAppear. Аналогичная ситуация будет и с нижним вью контроллером, только здесь сначала вызовется viewWillAppear, а затем viewWillDisappear, потому что он так и не был показан полностью. На нашей схеме это выглядит вот так.



Во время интерактивных действий вью контроллеры могут переходить из состояния `Appearing` в состояние `Disappearing`, и наоборот.

Более того, в iOS контроллеры вью могут менять свое состояние во время работы.

## 1.2. Архитектура MVC

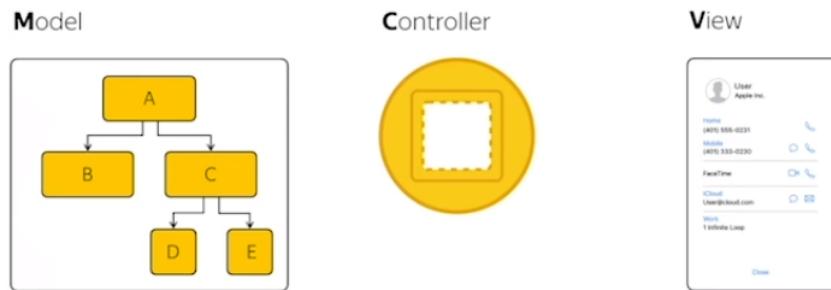
### 1.2.1. Шаблон MVC

Расскажу ещё немного теории про проектирование приложений. Для создания интерфейсов в iOS используется шаблон MVC. Это аббревиатура от имён основных его компонентов: Model,

View и Controller. По сути, это набор правил по которым организуется взаимодействие между классами в программе. Это, наверное, один из самых простых шаблонов, потому что он включает в себя всего три понятных компонента. За время своего существования, а это более 30 лет, он претерпел ряд изменений, и сегодня существует несколько его разновидностей. Мы рассмотрим ту, которую предлагает Apple, однако помните, что этот шаблон в своём базовом виде хорошо подходит для создания небольших проектов. Если хотите создавать крупные приложения, его надо модифицировать.

Первый компонент – это вью, оно же представление или графический интерфейс приложения. Это не только класс `UIView`, это целый набор знакомых вам графических элементов, таких как кнопки, текстовые поля, картинки, слайдеры и многие другие, объединенные вместе. Они не выполняют какую-то логику приложения. Основная их цель – выводить готовые данные на экран и взаимодействовать непосредственно с пользователем.

Второй компонент – это модель. Фактически, это данные приложения. В самом простом случае здесь может быть класс с полями, но в случае крупных приложений, это может быть целый слой, включающий десятки или сотни классов, реализующих бизнес логику, работающих с базами данных и сетью. Наверное, это самый неоднозначный и сложный компонент во всём шаблоне `model-view-controller`. С этим вы познакомитесь более подробно дальше, в лекции про архитектуру приложений. А пока давайте предположим, что модель – это просто класс с данными.

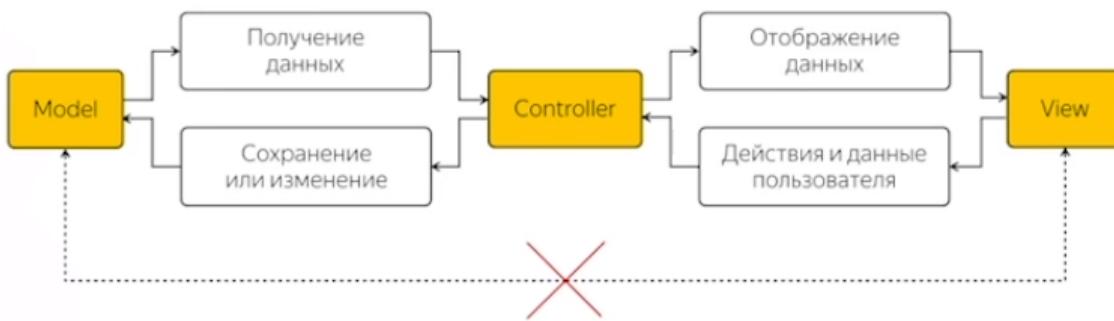


И, наконец, контроллер. Как вы уже догадались, он представлен классом `UIViewController`. Это посредник между данными приложения и графикой одного экрана. Его основная цель – отобразить имеющиеся данные пользователю. И, наоборот, превратить действия пользователя в данные приложения.

Вью контроллер может обрабатывать события, поступающие от вью, например, такие как нажатие кнопки или ввод текста. В ответ вью контроллер может давать команды интерфейсу. Например, показать картинку, которую только что загрузили из интернета. С другой стороны, вью контроллер взаимодействует с моделью. По сути, он может читать данные модели, анализировать их, и преобразовывать в такой вид, который понятен элементам интерфейса. А в обратную сторону контроллер может обновлять модель приложения новыми данными, полученными от пользователя.

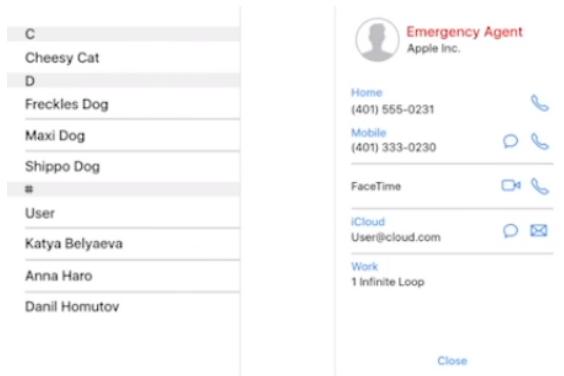
Так же есть ограничения, которые накладывает паттерн Model View Controller. Очень плохим тоном считается взаимодействовать напрямую с моделью из графических элементов. Все графические элементы должны быть максимально универсальными, чтобы их можно было переиспользовать в разных местах. И если ваш класс, унаследованный от UIView, получает данные напрямую из модели, то это означает, что вы свернули на очень опасный путь. Из-за этого в приложении всё чаще начнут появляться случайные баги, а код станет проще стереть и переписать заново, чем исправить. Поэтому данные всегда необходимо преобразовывать во виду контроллере, даже если эти преобразования кажутся очень простыми.

По той же причине и модель данных не должна ничего знать про виду. В идеальном случае, она вообще должна быть реализована в отдельном фреймворке.



### 1.2.2. MVC на примере приложения адресной книги

Вот ещё один пример. Допустим, вам необходимо сделать адресную книгу. Небольшое приложение, в котором 2 экрана. Первый экран – это список контактов. Здесь в таблице будут расположены имена и фамилии тех кто записан у вас в книге. Если нажимаем на какую-то фамилию, то открывается второй экран, который содержит детальное описание. Тут мы можем отредактировать поля, и вернуться назад. При этом все изменения должны сохраняться. А если мы меняем имя и фамилию, они должны поменяться на первом экране.



Итак, какие данные для работы такого приложения нам нужны? Было бы хорошо завести объект, хранящий информацию об одной записи в адресной книге. Пусть это будет структура Contact. Для такого рода данных намного проще работать со структурой, чем с классом из-за того, что поля структуры легко копировать. В копии Мы можем редактировать всё, что надо, и это никак не повлияет на исходный объект.

```
struct Contact {
    var id: String
    var name: String
    var phone: String
    var address: String
    var photo: UIImage?
    // другие поля
}

class AddressBook {
    var contacts: [Contact]
}
```

Пойдём дальше. Ещё нам необходима сама адресная книга. По сути это просто массив контактов. Для её описания мы используем класс, чтобы адресная книга была общей для обоих экранов. А класс как раз является ссылочными типом в языке Swift. Все изменения, которые мы сделаем в адресной книге на втором экране, будут доступны из первого.

Начнём собирать наше приложение с первого экрана. Как вы уже знаете он будет состоять из Вью и Контроллера. В качестве вью проще всего использовать таблицу – это класс UITableView. Однако, ни что не мешает вам использовать любой другой подход, например, связку классов UIStackView и UIScrollView, либо ещё что-то другое. Сейчас это сильно не важно, потому что нам нужно разобраться как работать с данными приложения, а не с их отображением. Создадим саму адресную книгу. Она будет выполнять роль модели данных. Теперь заполняем её контактами.

Пока мы не знаем как загружать хранимые данные. Поэтому просто запишем туда статические, например вот такие 3 контакта. На этом этапе уже созданы 3 компонента нашей архитектуры: Модель, Вью и Контроллер.

```
class ContactsViewController: UIViewController {
    let addressBook = AddressBook()

    override func viewDidLoad() {
        super.viewDidLoad()

        addressBook.contacts = [
            Contact(id: "1", name: "Chessy Cat", phone: "+8123 12345678",
                    address: "BS 19-45", photo: UIImage(named: "Photo 1")!),
            Contact(id: "2", name: "Katya Belyaeva", phone: "+712345 12345678",
                    address: "Oxford st. 1-23", photo: UIImage(named: "Photo 2")!),
            Contact(id: "3", name: "User", phone: "(401) 555-0231",
                    address: "1 Infinite Loop", photo: UIImage(named: "Photo 3")!)]
    }
}
```

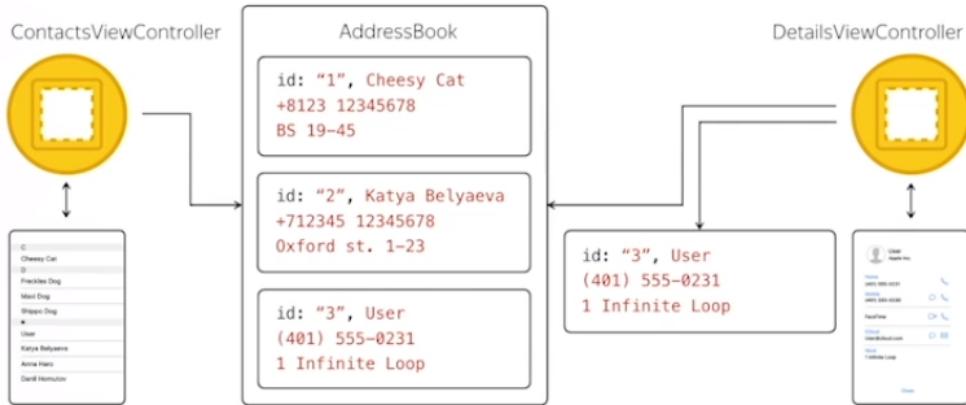
Движемся дальше. Представим что пользователь нажал на контакт и команда попала в метод didTapContact (at index). Получаем контакт из объекта адресной книги по индексу. Потом создаём DetailsViewController – он будет показывать данные выбранного контакта. Чтобы вью контроллер имел доступ к адресной книге, передадим в конструктор объект adressBook.

```
class ContactsViewController: UIViewController {
    let addressBook = AddressBook()
    // override func viewDidLoad() { ... }

    func didTapContact(at index: Int) {
        let contact = addressBook.contacts[index]
        let detailsViewController =
            DetailsViewController(addressBook: addressBook,
                                 contact: contact)
        present(detailsViewController, animated: true, completion: nil)
    }
}
```

И покажем DetailsViewController с использованием метода Present. Теперь у нас есть 2 вью контроллера, и оба они используют одну и ту же модель данных адресной книги. Плюс, так как

мы использовали структуру для контакта в книге, DetailsViewController ссылается на копию выбранного контакта.



Сейчас посмотрим как может быть устроен этот контроллер. Тут должна быть ссылка на адресную книгу и объект контакта, с которым мы будем работать. Добавим конструктор. Он просто принимает эти поля и сохраняет их внутри класса. Обратите внимание на то, что в конструкторе в любом случае необходимо вызвать `super.init(nibName:bundle:)`. Здесь мы используем стандартный конструктор класса `UIViewController`. Передадим в оба поля `nil` для того, чтобы видоизменение контроллер создался с параметрами по умолчанию.

```
class DetailsViewController: UIViewController {
    // @IBOutlet var nameTextField: UITextField! - И другие графические элементы

    let addressBook: AddressBook
    var contact: Contact

    init(addressBook: AddressBook, contact: Contact) {
        self.addressBook = addressBook
        self.contact = contact
        super.init(nibName: nil, bundle: nil)
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        // Тут показываем содержимое объекта contact
        // self.nameTextField.text = contact.name ... и так далее
    }
}
```

Настроить вид можно в методе `viewDidLoad`. Эту часть вы уже можете делать это самостоятельно, поэтому пойдем дальше. Добавим пару методов, которые будут изменять данные. Предположим, что они будут вызываться, когда пользователь что-то сделал в интерфейсе, и просто

обновят локальную структуру данных контакта. Так безопаснее, если мы, допустим, захотим отменить наши действия.

```
// MARK: - Обновление данных
extension DetailsViewController {
    func updateName(_ name: String) {
        contact.name = name
    }

    func updatePhone(_ phone: String) {
        contact.phone = phone
    }
}
```

Осталось добавить в DetailsViewController действие по кнопке Close. Тут всё не сложно. Ищем в массиве контактов индекс контакта с таким же ID что и наш. Если нашли, то просто меняем его на тот, с которым мы работали в контроллере. Если не нашли, то можем считать, что мы создали новый контакт, поэтому просто добавляем его в конец массива. И закрываем экран с использованием метода Dismiss.

```
class DetailsViewController: UIViewController {
    // ...
    @IBAction func saveAndClose() {
        if let index = addressBook.contacts.index(where: { $0.id == contact.id }) {
            // Обновляем существующий контакт
            addressBook.contacts[index] = contact
        } else {
            // Нет такого? – добавляем новый
            addressBook.contacts.append(contact)
        }
        dismiss(animated: true, completion: nil)
    }
}
```

Давайте ещё раз. После взаимодействия с пользователем обновились поля с именем и адресом в локальной модели. После нажатия кнопки Close мы сохранили локальные изменения в общую модель данных, то есть в адресную книгу. И закрыли DetailsViewController. Осталось дело за малым – обновить содержимое первого экрана – ContactsViewController. Это обновление можно

вызывать каждый раз при вызове метода `viewWillAppear`. Если обновление написано правильно, то новые данные просто покажутся пользователю.

```
class ContactsViewController: UIViewController {  
    // { .. }  
  
    override func viewWillAppear(_ animated: Bool) {  
        super.viewWillAppear(animated)  
        reloadAddressBook()  
    }  
  
    func reloadAddressBook() {  
        // Тут должна быть логика отображения списка контактов  
    }  
}
```

На этом всё. Мы разобрались как можно передавать данные между экранами в iOS.

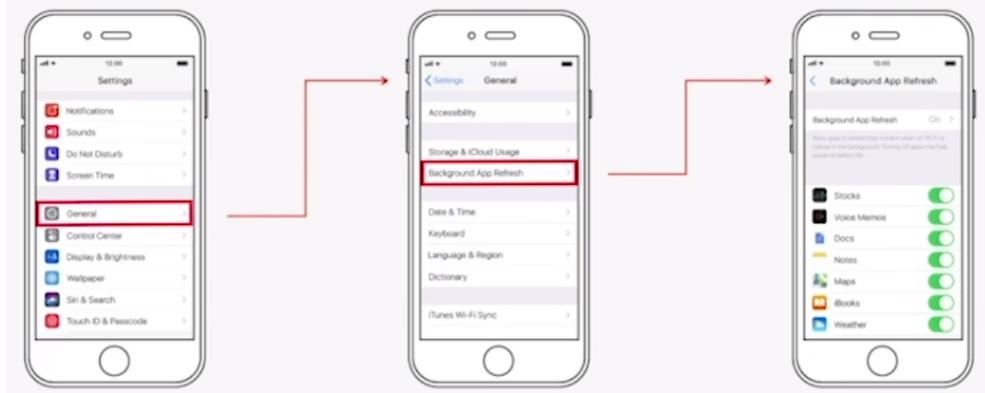
## 1.3. Класс UINavigationController

### 1.3.1. Что такое UINavigationController

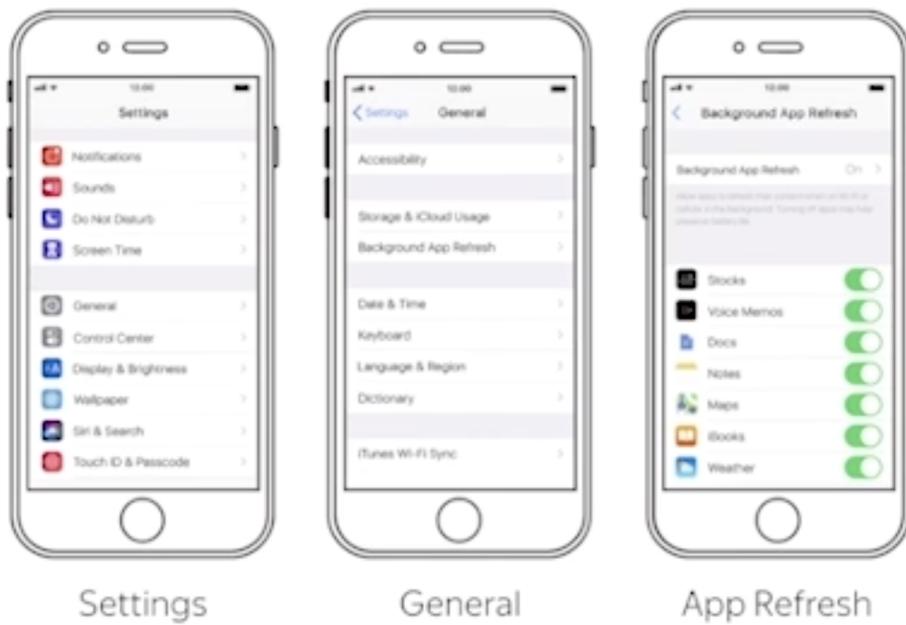
В iOS есть виду контроллеры, цель которых быть контейнерами для других контроллеров. Они также являются наследниками класса `UIViewController`. Но вместо того, чтобы отображать информацию для пользователя они позволяют удобно переключаться между экранами приложения. Контейнеры делают навигацию ещё проще, а главное, ваше приложение будет вести себя так как этого ожидают пользователи. Вы можете создавать свои собственные контейнеры, однако перед этим стоит убедиться, что ни один из существующих вам не подходит.

Начнём с наиболее используемого из них. Это `UINavigationController`. Так же как и метод `present`, этот виду контроллер позволяет переходить от одного виду контроллера к другому и возвращаться назад. Дополнительно он даёт намного больше функциональности и гибкости в настройках, чем `present`.

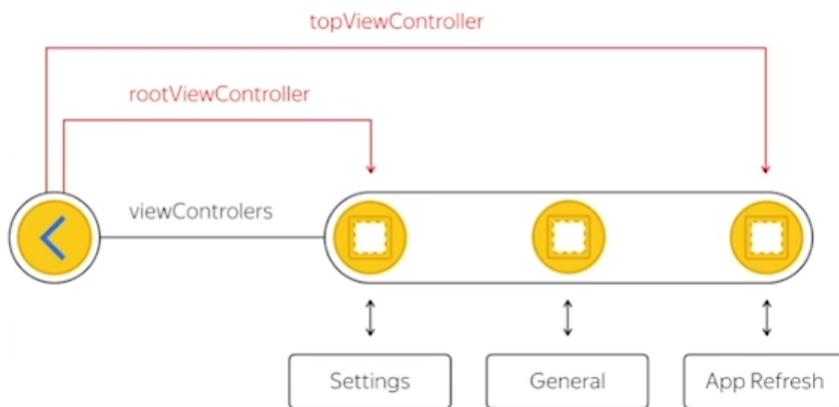
## UINavigationController



Посмотрим на приложение Настройки для iPhone. В его основе как раз лежит UINavigationController. При нажатии на пункт меню создаётся новый контроллер с содержимым выбранного пункта и добавляется в стэк navigation controller. Таким образом, мы можем заходить в глубь приложения, раскрывая всё новые и новые экраны. При чём количество вложений ограничено лишь объёмом оперативной памяти. В верхней части navigation controller размещена так называемая шапка – UINavigationController. Когда мы заходим в глубь приложения, на неё автоматически добавляется заголовок экрана и кнопка Back. Если кнопку back тапнуть, то navigation controller вернёт нас на предыдущий экран.



Все вью контроллеры, которые были добавлены в navigation controller, будут находиться в массиве `viewControllers`. Притом обязательно в том порядке, в котором они были добавлены. Последний добавленный будет в конце массива. На него существует отдельная ссылка – `topViewController`. Так же есть ссылка и на первый корневой вью контроллер. Его, как правило, нельзя убрать полностью, но можно заменить.



При работе с `UINavigationController` мы оперируем всего двумя командами. Первая – это `push`. Она добавляет новый вью контроллер в стэк и делаем его видимым. Для этого надо создать

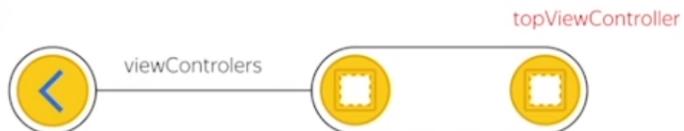
объект того экрана, который мы хотим добавить, а затем вызвать метод push для навигейшн контроллера. Обратите внимание, в каждом вью контроллере есть ссылка на навигейшн контроллер, в котором он находится. Если же navigation controller не создан, или вью контроллер с контентом добавлен неправильным образом, то в этой ссылке будет лежать nil.



```
// Создаем контроллер с контентом
let nextViewController = NextViewController()
// Помещаем его на верх стека UINavigationController
navigationController?.pushViewController(nextViewController, animated: true)
```

Если же мы хотим вернуться назад на один или несколько уровней, то надо воспользоваться командой popViewController. Она выталкивает верхний вью контроллер и удаляет ссылку на него.

## UINavigationController – удаление



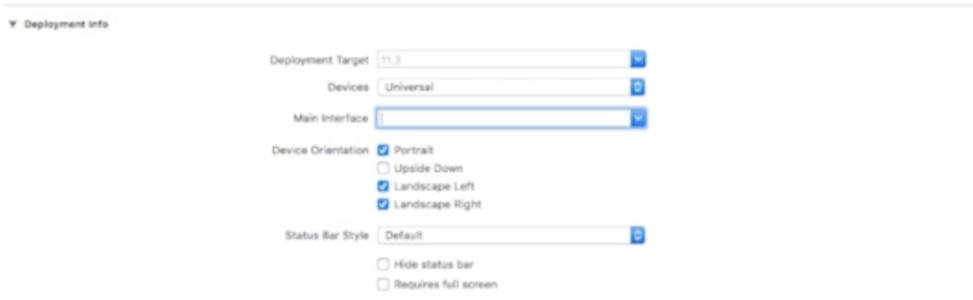
```
//Удаляем контроллер из стека UINavigationController
navigationController?.popViewControllerAnimated(true)
navigationController?.popToRootViewControllerAnimated(true)
navigationController?.popToViewController(someViewController, animated: true)
```

Этот метод работает точно так же, как кнопка back в левом верхнем углу, которая добавляется автоматически. Для того, чтобы вернуться на 2 и более экрана назад, существуют методы popToViewController и popToRootViewController. Можете использовать их в зависимости от ситуации.

### 1.3.2. UINavigationController на практике

Перейдём к проекту. Сейчас его отправной точкой является файл Main.storyboard. Он уже открыт в интерфейс билдере. Здесь расположен стартовый вид контроллер. В Storyboardе очень легко поменять его на UINavigationController, но так мы можем потерять всю суть происходящего. А про то, как использовать Storyboard, я расскажу вам немного позже.

Итак, для того, чтобы происходящее не казалось нам чистой магией, мы попробуем создать стартовый экран с использованием кода. Начнём с того что удалим Storyboard. Также необходимо удалить его имя в Application Settings. Выделяем проект и выбираем закладку General. И просто очищаем поле Main Interface.



И нам больше не нужен класс ViewController.

Перейдём к коду. Так как механизм Storyboard больше не работает, нам необходимо создавать всё вручную. Как вы помните, приложение начинает свою жизнь в методе `ApplicationDidFinishLaunchingWithOptions` в классе AppDelegate. Здесь есть ссылка на объект UIWindow. Когда мы удаляем Storyboard, главный window нам необходимо создать руками

```
window = UIWindow(frame: UIScreen.main.bounds)
```

Теперь добавим на него первый вид контроллер. Пусть это будет UINavigationController.

```
let navigationController = UINavigationController()
window?.rootViewController = navigationController
```

И ещё нужно вызвать команду для того, чтобы window стал видимым.

```
window?.makeKeyAndVisible()
```

Запустим проект.



Сейчас на симуляторе мы видим пустой экран, но вверху на нём можно заметить какой-то графический элемент. Это UINavigationBar – составная часть UINavigationController'a. Если он присутствует это значит UINavigationController был добавлен правильно.

Теперь давайте добавим контент. Создадим стартовый контроллер с контентом:

```
StartViewController
```

И немного изменим его интерфейс. Ещё можно добавить заголовок:

```
title = "First screen"
```

Этот заголовок автоматически попадёт в NavigationBar когда мы покажем вью контроллер. Сейчас разместим этот экран в UINavigationController'e. Перейдём в AppDelegate и создадим StartViewController:

```
let startViewController = StartViewController()
```

Можно использовать один из конструкторов UINavigationController'a:

```
let navigationController = UINavigationController(rootViewController:  
startViewController)
```

Этот конструктор принимает первый вью контроллер, который будет показан.

Соберём проект.



Отлично! А сейчас попробуем вызвать метод push и показать следующий экран. Для этого у нас уже есть созданный SecondViewController. Давайте добавим заголовок и сюда:

```
title = "Second screen"
```

В первом контроллере надо добавить кнопку (Push), по которой будем показывать второй экран. Осталось создать второй контроллер и передать его в метод push:

```
@IBAction func buttonPushClicked(_ sender: Any) {  
    let secondViewController = SecondViewController()  
    navigationController?.pushViewController(secondViewController, animated: true)  
}
```

Ещё раз соберём проект. По нажатию на кнопку push мы должны будем увидеть второй экран.



Да, кнопка теперь работает. Мы видим, что анимация перехода отличается от той, что была

при вызове метода Present. Обратите внимание, когда мы открыли второй экран, сверху автоматически добавилась кнопка Back. Если нажмём, она вернёт нас на стартовый экран. Теперь проверим, что будет, если нажать на кнопку Close. Как видим она не работает, потому что вызывает метод Dismiss. Этим методом можно скрывать только контроллеры, добавленные через метод Present. Используем в место него метод popViewController:

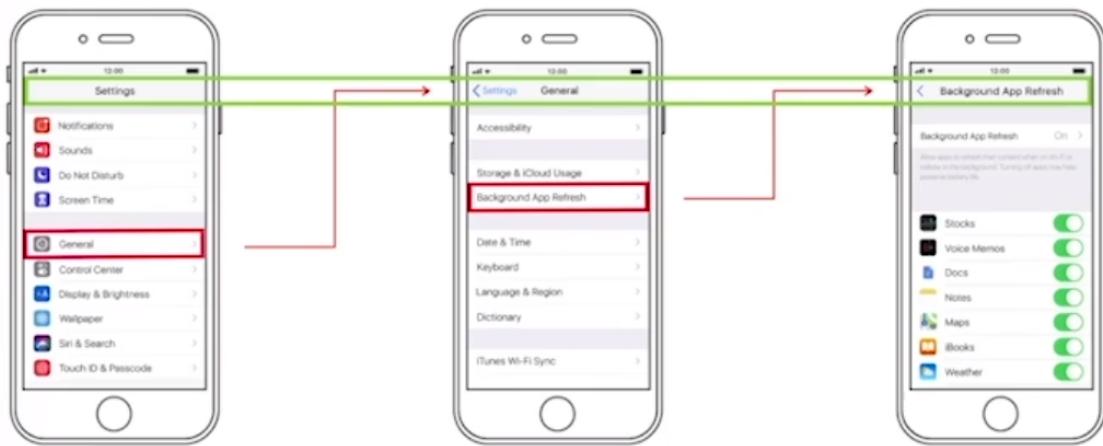
```
navigationController?.popViewController(animated: true)
```

Соберём проект и попробуем снова нажать на кнопку Close. Теперь всё работает как надо.

### 1.3.3. Другие полезные свойства UINavigationController

Важным компонентом навигейши контроллера является его навигейши бар. Он располагается сверху контроллера, и в некоторых случаях может перекрывать часть контента. Если нужно, мы можем изменить внешний вид навигейши бара: цвет его фона или стиль текста на кнопках.

## UINavigationController – Navigation Bar



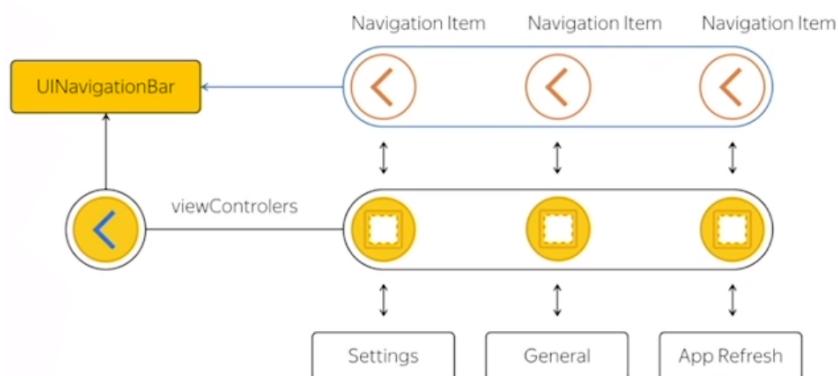
Доступ к нему можно получить через свойство navigationController.navigationBar. Мы также можем спрятать его, если захотим.

## UINavigationController – Navigation Bar

```
navigationController?.navigationBar  
navigationController?.navigationBar.backgroundColor = UIColor.red  
navigationController?.setNavigationBarHidden(true, animated: true)
```

Когда мы добавляем новые контроллеры, содержимое навигейшн бара изменяется в зависимости от того, какой контроллер показан. Сделано это следующим образом.

## UINavigationController – Navigation Item



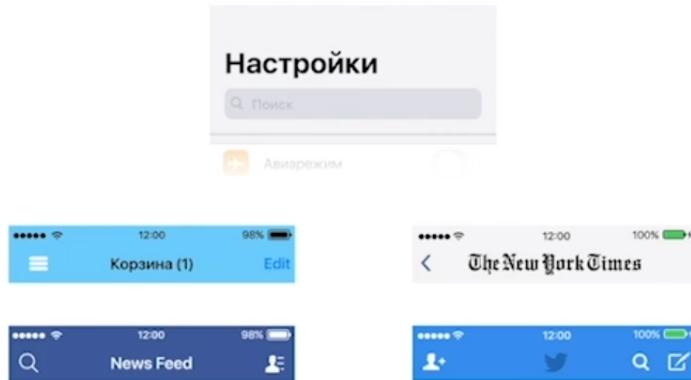
Каждый добавленный виду контроллер автоматически получает прикреплённый к нему объект `UINavigationItem`. У каждого он свой. Через `navigationItem` можно настраивать заголовок экрана и устанавливать дополнительные кнопки справа и слева от него. Можно настроить кнопку Back, которая будет показана на следующем добавленном экране.

## UINavigationBar/UINavigationItem – Компоненты



Начиная с iOS 11 навигейшн бар стал поддерживать крупные заголовки, и на него можно разместить строку поиска. Вот ещё несколько вариантов как может выглядеть navigation bar.

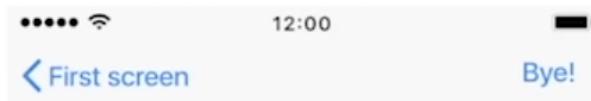
## UINavigationController – Navigation Item



Для того чтобы добавить кнопку, достаточно просто создать объект класса UIBarButtonItem, назначить ему селектор метода, который будет вызван при нажатии, и установить в поле navigation item. Здесь в примере это rightBarButtonItem.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Создаём кнопку, помещаем её в правую часть Navigation Bar
    // В качестве обработчика события устанавливаем объект 'self'
    // и селектор метода 'func closeClicked(_ sender: Any)'
    navigationItem.rightBarButtonItem =
        UIBarButtonItem(title: "Bye!", style: .plain,
                        target: self, action: #selector (closeClicked(_:)))
}

// Используем @objc чтобы можно было получить селектор на данный метод
@objc func closeClicked(_ sender: Any) {
    navigationController?.popViewController(animated: true)
}
```



Посмотрим на ещё один пример. Здесь мы создаём объект UIActivityIndicatorView. На самом деле, вместо него можно использовать любой класс унаследованный от UIView. Затем создаём UIBarButtonItem с использованием специального конструктора – initWithCustomView и передаём туда activity indicator.

Ещё UIBarButtonItem можно создать с использованием стандартных системных иконок. Их около двадцати штук. Вы можете сами поэкспериментировать с различными видами. Они находятся в типе UIBarButtonItemSystemItem.

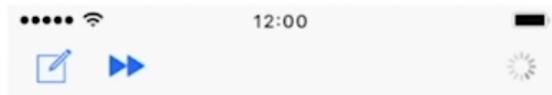
Для того, чтобы показать несколько элементов, можно использовать свойства – left или rightBarButtonItem. Они принимают массивы элементов. Но не стоит делать их слишком много. Это только усложнит интерфейс на небольших смартфонах. Как результат получаем такой навигейшн бар: с двумя кнопками слева и Активити индикатором справа.

```

let activityIndicator = UIActivityIndicatorView(activityIndicatorStyle: .gray)
activityIndicator.startAnimating()
let barButtonItemWithView = UIBarButtonItem(customView: activityIndicator)
navigationItem.rightBarButtonItem = barButtonItemWithView

let itemOne = UIBarButtonItem(barButtonSystemItem:
    .compose, target: nil, action: nil)
let itemTwo = UIBarButtonItem(barButtonSystemItem:
    .fastForward, target: nil, action: nil)
navigationItem.leftBarButtonItems = [itemOne, itemTwo]

```

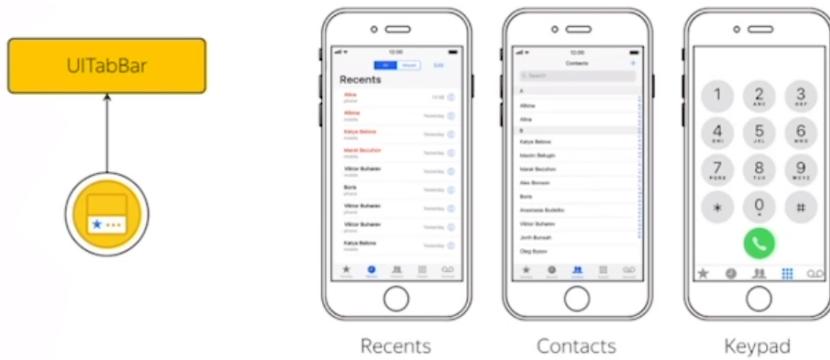


## 1.4. Класс UITabBarController

### 1.4.1. Что такое UITabBarController

Второй по популярности контейнер в iOS приложениях – это UITabBarController. Внешне он выглядит как панель закладок внизу экрана. А в каждой закладки находится UIViewController. Вернее, при нажатии на закладку, связанный с ней контроллер становится видимым. Притом в закладки можно класть ещё и контейнеры, например UINavigationController. Это хорошо видно в приложении для звонков в iPhone.

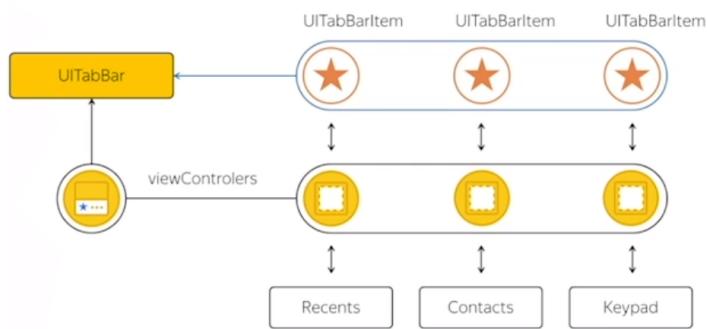
### UITabBarController



По устройству UITabBarController чем-то похож на UINavigationController. Здесь тоже есть мас-

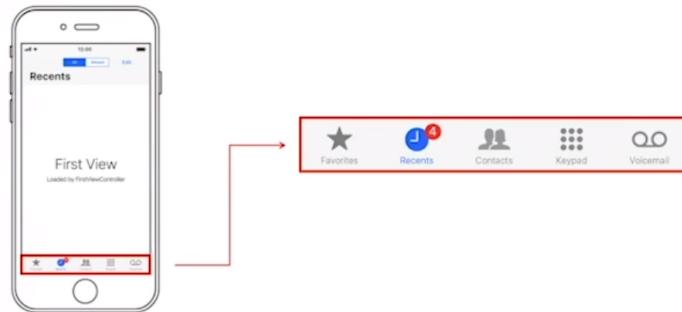
сив `viewControllers`, но он не меняется во время работы. Все контроллеры должны быть настроены перед отображением.

### UITabBarController



Для того, чтобы настроить иконки на таб баре, существует класс `UITabBarItem`. Так же как и в `NavigationController`, этот `tabBarItem` есть у каждого вью контроллера. Фактически, здесь можно настроить иконку и бейдж для каждой закладки.

### UITabBar

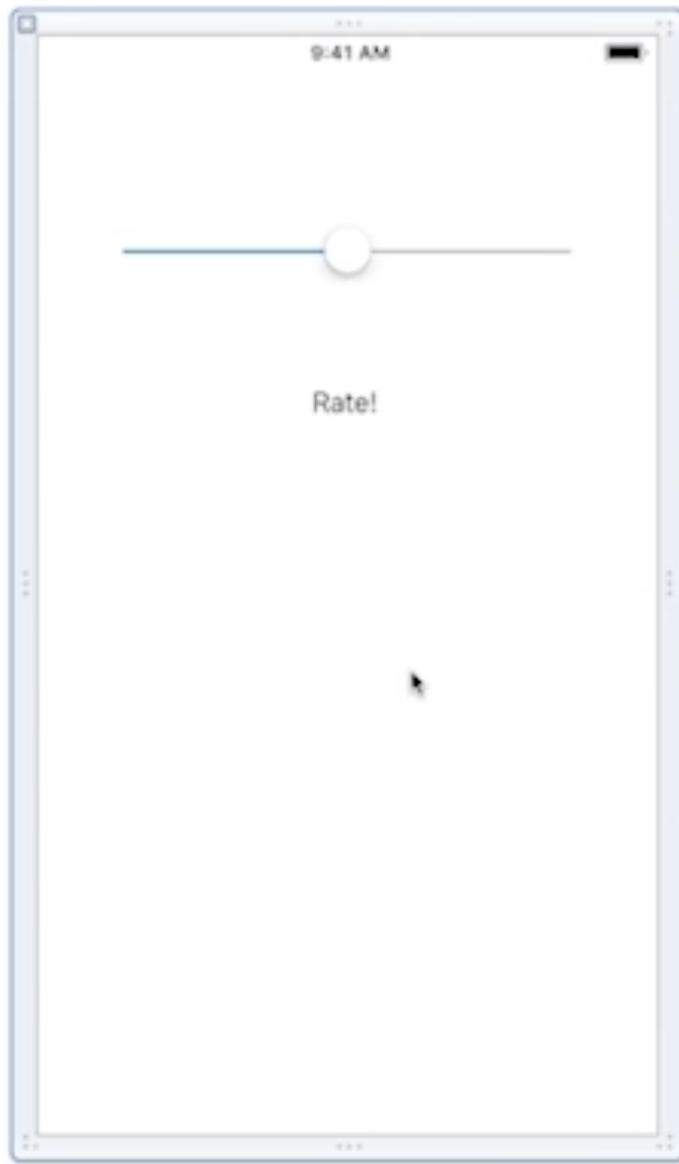


В иконке можно задать 2 состояния – нажатое и обычное. А бейдж – это красный кружок в углу загадки. Обычно он говорит о том, что в этой закладке есть что-то новое или важное. Цвет бэджа тоже можно менять.

#### 1.4.2. UITabBarController на практике

Сейчас попробуем добавить `TabBarController` в наш проект. На первой закладке мы оставим существующий `NavigationController` с экраном `StartViewController`. А на второй разместим экран рейтинга. Назовём его `RateViewController`.

Теперь настроим интерфейс. Нам нужен слайдер и подпись под ним, чтобы этот экран отличался от других (UILabel → Rate Me! + UISlider).



Теперь нужно настроить UITabBarController. Но сначала нам понадобятся иконки для его закладок. Самое время заглянуть в Assets Catalog. Это место, где обычно хранятся ресурсы вашей программы. Для таб бара нужно найти несколько иконок с прозрачном фоном. А потом перетащить сюда. Прозрачный фон здесь очень важен, потому что tab bar теряет всю информацию о

цвете, оставляя только непрозрачные части иконки. Лучше всего для этого подойдут файлы в формате PNG. Но можно использовать и векторные файлы PDF, если вы сможете их найти. Я уже добавил сюда пару иконок для закладок, которые планирую сделать.

Начнём с создания tab bar controller'a. Откроем AppDelegate. Сам tab bar controller создается так же, как и любой другой контроллер, с использованием конструктора:

```
let tabBarController = UITabBarController()
```

Теперь нам нужны контроллеры для закладок. StartViewController уже создан. Поэтому создаём RateViewController.

```
let rateViewController = RateViewController()
```

И теперь настраиваем UITabBarItem:

```
rateViewController.tabBarItem = UITabBarItem(title: "Rate", image: UIImage(named: «Star»))
```

Теперь вопрос – как установить правильно Tab Bar Item для первой закладки? Сам Tab Bar Controller ищет такие объекты только в массиве viewControllers. Так как на первой закладке мы хотим видеть контент, помещённый в Navigation controller, то настраивать tab bar item мы должны именно для него:

```
navigationController.tabBarItem = UITabBarItem(title: "Start", image: UIImage(named: "Pear"))
```

Теперь поместим оба контроллера в массив viewControllers:

```
tabBarController.viewControllers = [navigationController, rateViewController]
```

И последний шаг. Сейчас наш интерфейс должен строится от tab bar controller'a. Поэтому его необходимо сделать корневым:

```
window?.rootViewController = tabBarController
```

Всё готово, давайте соберём и запустим проект.



Итак, теперь у нас появились закладки. Обратите внимание, что если мы зайдём в глубь navigation controller'а, то при переключении закладок, мы вернемся на тот же открытый экран. Здесь есть ещё один любопытный момент. Когда мы используем UINavigationController внутри UITabBarController'a повторное нажатие на открытую закладку вернет нас на самый первый экран, аналогично действию popToRootViewController.

Также мы можем программно переключаться между закладками. Для этого создадим действие в SecondViewController:

```
@IBAction func showRateTab() {  
    tabBarController?.selectedIndex = 1  
}
```

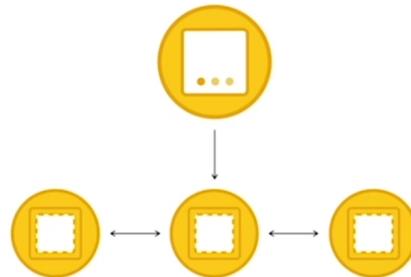
selectedIndex мы берём из массива viewControllers. Он должен соответствовать тому виду контроллеру, который вы хотите показать. Первый индекс в нашем случае соответствует закладке с RateViewController. Ссылка tabBarController будет доступна во всех контроллерах, добавленных в него. Включая даже те, которые находятся внутри navigation controller'ов. Теперь осталось добавить кнопку в редакторе интерфейса (UIButton → Show rate!). Теперь проверим, что получилось.

Это были основные свойства класса UITabBarController.

## 1.5. Другие полезные контейнеры

Рассмотрим ещё пару удобных контейнеров. Как оказалось, очень часто мы имеем дело с контентом, разбитым на отдельные страницы или слайды. Отличный пример – это галерея. Или приложение iBooks. Для этого используется класс UIPagerViewController.

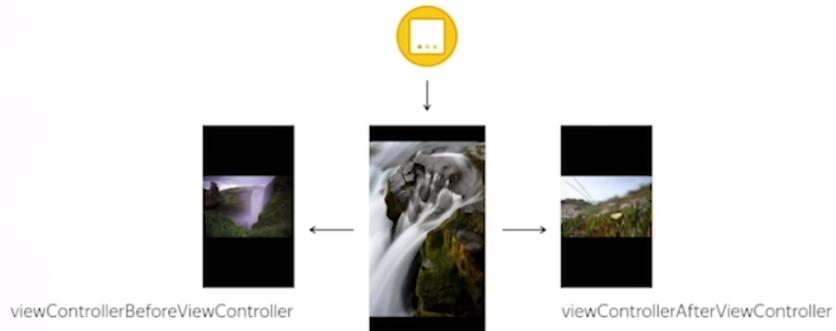
### UIPageViewController



Этот контейнер позволяет свайпом вправо или влево переходить между вью контроллерами с контентом. И у него есть два стиля. Первый стиль более формален. Он выглядит как составленные один к одному экраны. Это очень похоже на галерею телефона. А второй стиль больше похож на книжные страницы. С ними даже можно поиграться во время перелистывания.

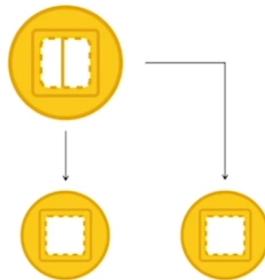
Как настраиваем? Сначала создаём UIPageViewController. Потом назначаем стартовый вью контроллер, и dataSource. Это протокол UIPageViewControllerDataSource. Как только пользователь начинает свайп, контроллер вызывает методы viewControllerBeforeViewController или viewControllerAfterViewController в зависимости от направления этого свайпа, и ожидает, что наш код вернёт следующий вью контроллер. Можно вернуть и nil, тогда это означает, что пользователь додолистал до конца.

### UIPageViewControllerDataSource



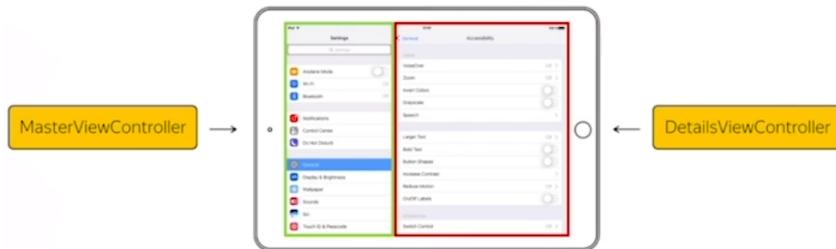
Второй контейнер используется в устройствах с большим экраном. Он может быть знаком по приложению настройки или почты в iPad. Это UISplitViewController.

### UISplitViewController



В нём всего 2 компонента – это MasterViewController и DetailsViewController. Притом их можно отображать или прятать независимо друг от друга, и конечно, указывать, сколько места будет занимать каждый из них. Настраивать так же просто как Tab Bar Controller. Создаём объект класса UISplitViewController. И используем свойство viewControllers. Максимум туда можно передавать два контроллера. Первый из них станет в позицию Master, а второй – в Details.

### UISplitViewController



Более подробно ознакомиться с этими контейнерами вы сможете самостоятельно в примерах и документации от Apple.

Итак, мы рассмотрели все основные контейнеры. UITabBarController позволяет создать набор закладок и переключаться между ними в произвольном порядке. UINavigationController позволяет раскрывать экраны один за одним и потом возвращаться назад. UISplitViewController позволяет делить экран на 2 независимые части, так называемые master и details. Помните, что он работает только для iPad или для айфонов с большим экраном. И, последний, это UIPageViewController. Предназначен для того, чтобы организовывать ваш контент в виде перелистываемых страниц. Используйте эти контейнеры, чтобы сделать ваше приложение удобным и понятным для пользователя.

## 1.6. Что такое Storyboard

### 1.6.1. Теория о Storyboard

Вы уже знакомы с двумя типами графических файлов в iOS. В Xib файлах обычно располагается по одному графическому элементу. Хотя может быть и больше. Притом какое-то время это был единственный способ работать с интерфейсом. Но разработчики Apple захотели сделать это ещё нагляднее и проще. И в 2011 представили новый тип интерфейсных файлов. Это Storyboard. Этот файл позволяет видеть структуру сразу всего приложения, вместе с переходами между экранами.

Итак, здесь есть несколько ключевых элементов. Первый – это сцена. Это вью контроллер с контентом или контейнер-контроллер. То есть то, что составляет интерфейс приложения. Сверху в каждой сцене расположен список его элементов. Этот список похож на тот, который мы видели в редакторе Xib файла. И с ним можно взаимодействовать подобным образом.



Второй ключевой элемент - это сегвей, или переход. Он представлен классом UIStoryboardSegue и может делать основные операции по переключению между экранами, такие как Present, Push, Popover. А если этого недостаточно, можно наследоваться от этого класса и реализовать своё собственное поведение перехода.



Итак, с помощью комбинации сцен иsegueев мы можем построить полноценное приложение.

### 1.6.2. Storyboard на практике

Будем разбираться со сторибордом на примере навигейши контроллера. Начнём с нового проекта и откроем Main.storyboard. Есть несколько способов как добавить навигейши контроллер на сцену. Мы пойдём длинным, но более детальным путём. Давайте найдём его в списке элементов и перетащим в сториборд.

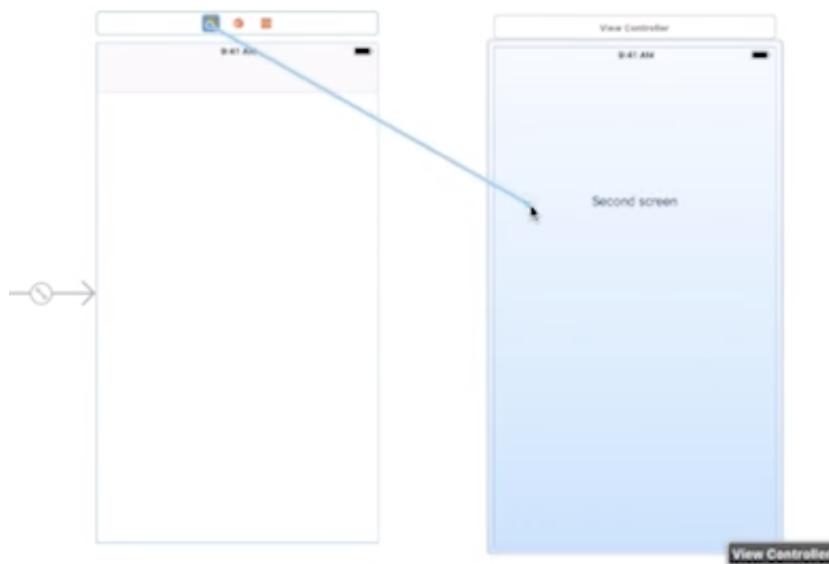


Навигейши контроллер создаётся сразу с рут виджетом контроллером. Но у нас уже есть свой рут

контроллер, поэтому мы переназначим его. Удаляем созданный за нас контроллер. Потом тянем мышкой от навигейши контроллера до нашего вью контроллера. Это можно делать либо правой клавишей мыши, либо левой, но с зажатой кнопкой control.

Мы хотим сделать этот контроллер корневым, поэтому в списке выбираем rootViewController. В каждом сториборде должна быть своя стартовая точка. То есть тот вью контроллер, который будет загружаться первым. В нашем случае это навигейши контроллер. Настраивается это в редакторе атрибутов. Мы должны выбрать навигейши контроллер и выделить галочку Is Initial View Controller. Сейчас добавляем следующий вью контроллер, на который будем делать переход. Просто перетягиваем его из панели объектов. Добавим лейбл, чтобы он как-то отличался от первого экрана.

Теперь нужно создать сегвей, чтобы обозначить переход из первого контроллера ко второму. Для начала выделяем первый вью контроллер, и тянем связь от иконки вью контроллера ко второму контроллеру. Напомню, что это делается с зажатой клавишей control. Здесь предлагают выбрать способ перехода. Для navigation controller это show.



Этот параметр соответствует методу Push, но в терминах Storyboard. Выделим получившийся сегвей и зайдём в его атрибуты. Здесь можно изменить класс, если вы захотите использовать нестандартный переход, или поменять тип в поле Kind. Но для того, чтобы вызвать сегвей, ему нужно задать идентификатор. Потом по этому идентификатору мы сможем обращаться к сегвею в коде. Давайте добавим – ShowSecondScreen. Теперь добавим кнопку на первый экран – Show Second Screen, и добавим действие – @IBAction func showButtonTapped().

Теперь перейдём к коду. Чтобы показать следующий экран нужно вызвать метод performSegue:

```
performSegue(withIdentifier:<String>, sender: <Any?>)
```

Первый параметр – это собственно идентификатор сегвея, который мы вписали в сториборде. Второй указывает на отправителя. По сути, это дополнительная информация, и может быть любым объектом.

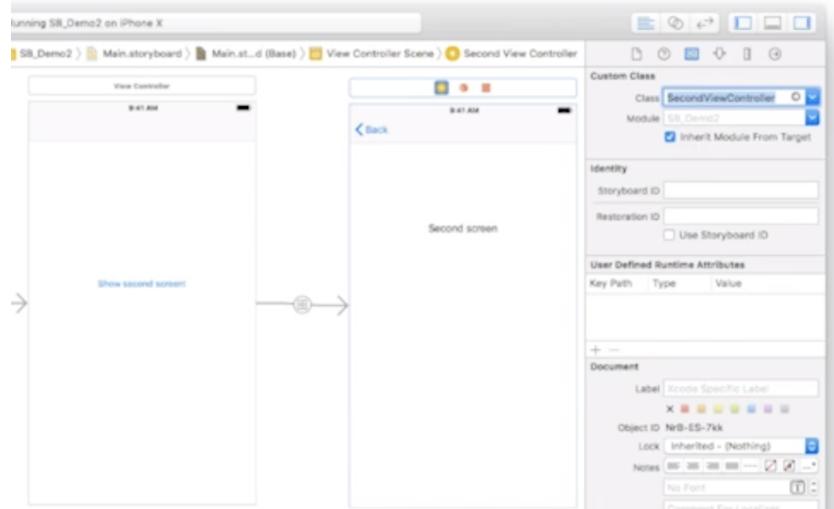
```
performSegue(withIdentifier:"ShowSecondScreen", sender: nil)
```

Давайте смотреть, что получилось. Отлично! Теперь мы попробуем передать данные во второй контроллер. Здесь отправной точкой станет метод prepare-for-segue:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) { }
```

Он вызывается сразу же после вызова performSegue. Первый параметр метода будет содержать идентификатор перехода и виду контроллер, который должен быть показан. Sender будет передаваться из метода performSegue. А пока создадим второй виду контроллер и настроим его на получение данных – SecondViewController.

Xib файл нам больше не нужен, потому что интерфейс контроллера будет в сториборде. Завершаем создание. Нужно назначить этот класс на сцену в сториборде. Выделяем виду контроллер и открываем Identity Inspector. В поле Class указываем нужный тип – SecondViewController.



Теперь создадим IBOutlet на лейбл – labelText.

Добавим свойство в контроллер:

```
var text: String?
```

И просто запишем его значение в лейбл:

```
override func viewDidLoad() {
    super.viewDidLoad()
   .textLabel.text = text
}
```

Итак, если мы что-то запишем в свойство `text`, оно сразу же попадёт на лейбл при загрузке вью контроллера. Вернёмся к методу `prepareForSegue`. Напомню, что этот метод будет вызван перед переходом на второй экран. Что мы знаем об этом переходе? То, что идентификатор его должен быть `ShowSecondScreen`, и что контроллер на который мы переходим – это `SecondViewController`. Можно записать это условие:

```
if let controller = segue.destination as? SecondViewController,
    segue.identifier == "ShowSecondScreen" {
}
```

И теперь мы можем передать в свойство `text` какие-нибудь данные:

```
controller.text = "Hello world!"
```

Сейчас, если мы нажмём на кнопку `show second view`, то должны будем увидеть `Hello World` на втором экране.

Ещё один нюанс, который вызывает много вопросов. Что, если нам нужно вернуться назад на предыдущий контроллер? И сделать это программно, то есть не по кнопке `Back`. В сториборде это реализуется тоже через сегвей, но весьма хитрым способом. Сначала мы должны добавить специальный `IBAction` во вью контроллер, в который мы хотим вернуться. Специальный он потому, что обязательно должен принимать сегвей как параметр:

```
@IBAction func unwindToMain(segue: UIStoryboardSegue) {}
```

Это обработчик возвратного перехода. Он ещё называется `unwind`. Здесь можно обработать событие перехода, если нужно. Но удалять этот метод нельзя, иначе возврат перестанет работать. Теперь создадим сам возвратный сегвей. Откроем сториборд, и добавим на второй экран кнопку `Close`. Здесь можно обойтись даже без кода. Просто привязываем любое действие к событию `Exit` на сцене, и выбираем тот метод, который добавили в первом контроллере.



Готово. Мы получили возвратный сегвей. Если нужно вызвать его программно, просто задайте идентификатор и используйте `performSegue`, как делали это для перехода вперёд.

На этом всё. Как видите, сториборд сильно упрощает создание пары экранов. Но в больших проектах сложно настраивать десятки сцен в одном сториборд файле. Обычно, либо разбивают всё приложение на несколько отдельных сторибордов, либо вовсе отказываются от них и используют XML файлы. Что касается небольших и учебный проектов, то сториборд подходит для них как нельзя лучше.

## 1.7. Класс UIScrollView

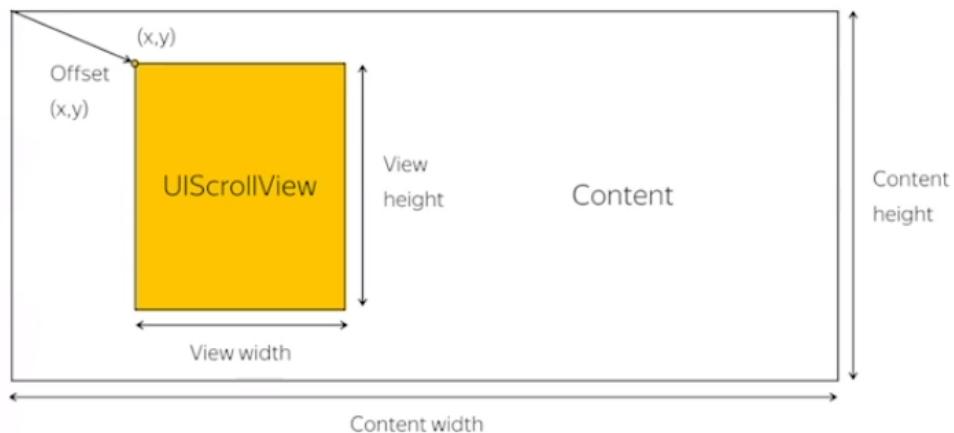
### 1.7.1. Теория о UIScrollView

Очень часто необходимо показать контент, который не влезит в границы экрана телефона. Это особенно актуально на мобильных платформах, где экраны обычно не очень большие. Например, разместить фотографии в длинный прокручивающийся список, или сделать меню похожее на настройки телефона. На помощь нам приходит класс `UIScrollView`. По сути, он как раз и предназначен для того чтобы хранить в себе контент, превышающий границы экрана. А пользователь может легко пролистывать его привычными жестами.

Итак, класс `UIScrollView` унаследован от `UIView`, поэтому его положением и размерами точно так же можно управлять через свойство `frame`, как в любом другом виду. Конечно, для этого можно использовать и `Autolayout`. Но Scroll View может также управлять границами собственного

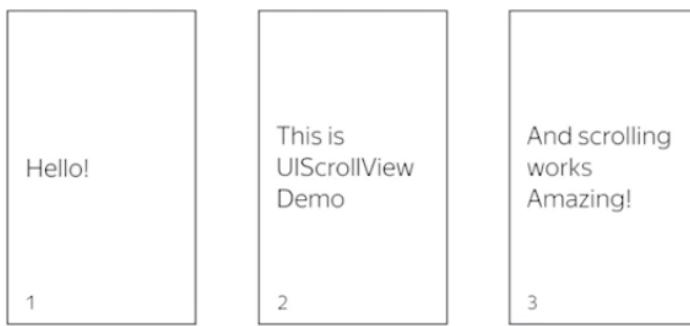
контента. Для этого есть свойства `contentSize` и `contentOffset`. `contentSize` говорит о том, какой размер области должен быть у контента, помещённого внутрь Scroll View. И если этот размер превышает размеры самого Scroll View хоть по какому-то из направлений, контент становится прокручиваемым, и пользователь может самостоятельно перемещаться внутри этой области. Второй параметр – `contentOffset`. Он изменяется, когда пользователь начинает перемещать контент внутри Scroll View. Математически он равен вектору от левого верхнего угла контента до угла самого Scroll View. Программно это структура `CGPoint`, где X будет изменяться если перемещать контент по горизонтали, а Y – по вертикали. Чем дальше пользователь перемещает контент, тем больше становятся эти значения. Конечно же, `contentOffset` можно изменять программно, чтобы пользователю можно было показать то, что сейчас нужно.

## UIScrollView



Настраивать ScrollView можно двумя способами – с применением Autolayout, и без него. Второй вариант кажется немного проще для понимания, поэтому начнём с него.

Допустим, нам нужно сделать экран приветствия для нашего приложения, и дизайнеры дали нам 3 готовых картинки, которые мы должны показать одну за одной. И было бы хорошо ещё показать, на какой из них сейчас находится пользователь. Картинки мы разместили в каталог ассетов.



Итак, создаём UIScrollView. Проще всего его создать в интерфейсном файле и добавить IBOutlet. Ещё нам понадобится массив объектов UIImageView. Это наши картинки на экране приветствия. Начинаем настраивать контент. В методе viewDidLoad берём название картинок из папки Assets и помещаем их в массив чтобы было проще работать. В цикле. Создаём картинку по имени. И сразу же создаём объект UIImageView. Всего их будет три, и они будут располагаться один за другим на ScrollView. Настраиваем contentMode для картинки. Нам нужно, чтобы картинка была полностью видна, а её пропорции оставались неизменными. За это отвечает режим scaleAspectFit. Добавляем ImageView на ScrollView. И в массив Images чтобы был доступ. Конец цикла.

## UIScrollView

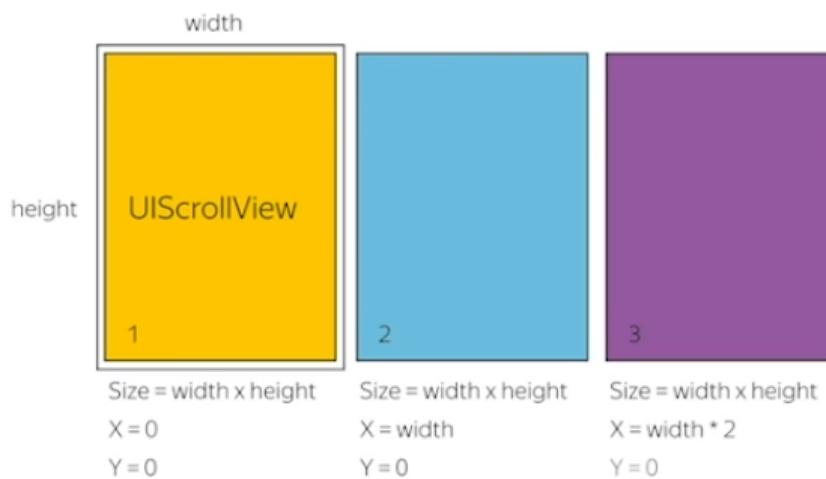
```
var imageViews = [UIImageView]()

override func viewDidLoad() {
    super.viewDidLoad()
    let imageNames = ["first", "second", "third"]
    for name in imageNames {
        let image = UIImage(named: name)
        let imageView = UIImageView(image: image)
        imageView.contentMode = .scaleAspectFit
        scrollView.addSubview(imageView)
        imageViews.append(imageView)
    }
}
```

Теперь картинки созданы, но их размеры ещё не заданы. Напомню, что этот код может быть запущен на любом устройстве под операционной системой iOS, а значит и размеры экранов могут быть совершенно разными. Также экран может быть развернут в ландшафтный режим.

Поэтому нам запрещено привязываться к каким-то статическим координатам. Всё должно рассчитываться на лету.

Итак, у нас есть scroll view с размерами по высоте и ширине. И наша цель правильно разместить внутри него 3 объекта UIImageView. С размерами каждого из них всё просто – они равны по ширине и по высоте scrollView. Осталось только определиться с координатами. Первая картинка находится в самом углу, для неё смещение по X и Y должно быть равно нулю. Вторая отстоит от угла Scroll View как раз на значение ширины этого самого scroll view. Поэтому X равен ширине, а Y – нулю. С третьей всё аналогично, только по оси X она отстоит на 2 ширины. Теперь это надо реализовать в коде.



Каждый раз, когда размеры вью обновляются, происходит вызов метода `viewDidLayoutSubviews`. Как раз в этот момент мы можем получить правильные размеры всех необходимых нам элементов. Более того, этот метод будет вызываться при повороте экрана.

И начнём писать код. Не забываем вызвать `super`. Проходим в цикле по картинками и их индексам. Размер картинки устанавливаем равным размеру scroll view. Координата X равна ширине scroll view умножить на индекс. Индексация здесь будет с нуля, поэтому первая картинка станет как раз в левый верхний угол контента. Координата Y всегда равна нулю. Заканчиваем цикл. Итак, положение картинок мы настроили, осталось настроить размер контента. Он равен ширине scrollView умноженной на количество картинок. А высота просто равна высоте.

## UIScrollView

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

    for (index, imageView) in imageViews.enumerated() {
        imageView.frame.size = scrollView.frame.size
        imageView.frame.origin.x = scrollView.frame.width * CGFloat(index)
        imageView.frame.origin.y = 0
    }
    let contentWidth = scrollView.frame.width * CGFloat(imageViews.count)
    scrollView.contentSize = CGSize(width: contentWidth,
                                    height: scrollView.frame.height)
}
```

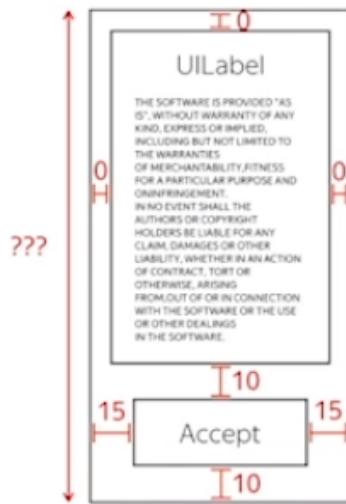
Завершаем метод и смотрим что получилось. Как видим, Scroll View позволяет пользователю перемещаться между картинками. Но ведёт он себя как-то странно. Нам бы хотелось чтобы scrollView мог останавливаться только на конкретной картинке, а не между ними. К счастью, это легко решается, если установить флаг isPagingEnabled в true, или включить соответствующую галочку в Interface Builder. Можно собрать проект и убедиться, что этот флаг работает.

### 1.7.2. Как настраивать UIScrollView

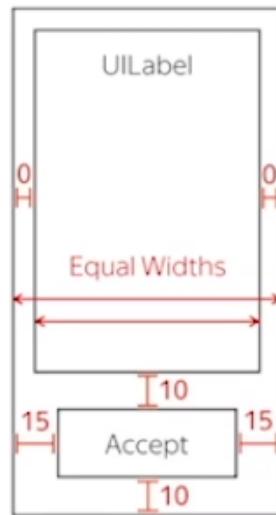
Теперь перейдём к настройке ScrollView через AutoLayout. Здесь основной особенностью является то, что ограничения, или констрайнты, должны быть привязаны к ScrollView во всех направлениях. Например, мы хотим показать большой блок текста, такой как пользовательское соглашение. А после текста нужно добавить кнопку принятия этого соглашения. Пока пользователь не прокрутит в самый низ, он не сможет нажать на кнопку. Вот как бы это выглядело при настройке Autolayout.

В качестве контейнера используем UIScrollView. На него помещаем UILabel с длинным текстом и кнопку. Теперь настроим констрайнты. Мы хотим, чтобы текст занимал всё пространство по ширине и растягивался вниз. Поэтому устанавливаем ограничения слева, сверху и справа между текстом и контейнером. Кнопка должна быть под текстом, поэтому необходимо зафиксировать расстояние между ними. И, чтобы кнопка растягивалась по ширине, ей так же необходимо установить ограничения слева и справа. Значение констрайнтов можно выбрать произвольными. Для этого примера выберем вот такие. Текст будет вплотную прилегать к контейнеру scroll view, а для кнопки зададим небольшие отступы. Остался самый важный нюанс, который отличает scroll view от остальных контейнеров. Для того, чтобы контент внутри можно было прокручивать, scroll view должен сам определить его contentSize. Если посмотреть на наш пример, то становится не понятно в каком месте снизу должен заканчиваться контент. Сразу за кнопкой, или ниже. И

на сколько ниже? Поэтому, когда вы размещаете элементы внутри ScrollView, их размеры и все отступы должны быть полностью определены. В том числе и снизу. Это решается добавлением ещё одного ограничения.



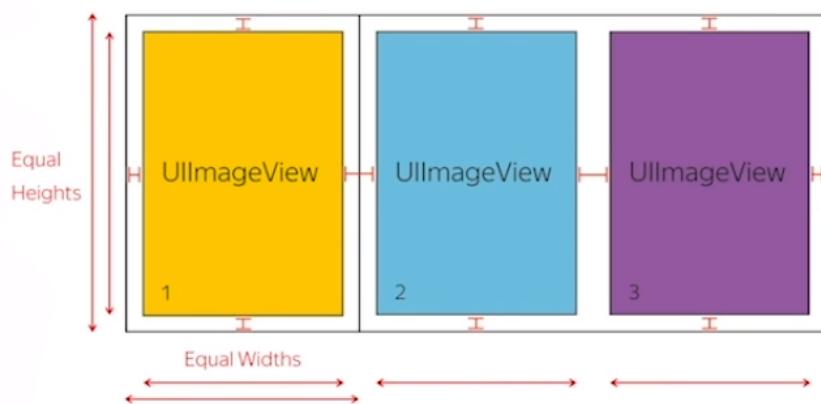
Аналогичная ситуация и с размером текста. На сколько длинным он должен быть? Вероятно, это зависит от его ширины. Но UIScrollView не задаёт никаких ограничений ни на ширину, ни на высоту элементов расположенных внутри. Поэтому здесь мы должны ограничить ширину сами. А потом Autolayout посчитает высоту за нас. Если подумать, то мы хотим чтобы ширина текста равнялась ширине scroll view. Как раз для этого существует ограничение Equal Width. Нужно выделить вместе UILabel и UIScrollView, а потом в настройках ограничений выбрать этот пункт.



Всю остальную магию scroll view делает за нас. Если текста будет слишком много, то автоматически появится возможность его прокручивать вниз до тех пор, пока мы не промотаем до самого нижнего элемента.

Сейчас мы немного разобрались в том, как управлять контентом в scroll view. Теперь давайте вернёмся к настройке экрана приветствия. Начнём с пустого scroll view. На него размещаем три UIImageView с нужными картинками. Для каждого из них задаём фиксированные расстояния со всех сторон до контейнера, а так же расстояния между собой. Как мы знаем, UIScrollView никак не ограничивает размеры собственного контента, поэтому нам его придётся устанавливать руками. Здесь всё достаточно просто. Нам необходимо, чтобы высота и ширина каждой картинки была равна ширине и высоте контейнера scroll view. Поэтому выделяем все картинки, а так же scroll view. Это удобно сделать в панели слева со списком всех графических элементов. И добавляем ограничения Equal Widths и Equal Heights.

## UIScrollView



На этом настройка закончена и можно проверять, что получилось. Вот так без единой строчки кода можно настроить достаточно сложный интерактивный экран.

### 1.7.3. Дополнительные возможности UIScrollView

В заключении стоит ещё немного добавить про некоторые полезные особенности ScrollView. Если вы обратили внимание, то при прокрутке к границе контента scroll view не останавливается как только достигнет края, а делает плавное замедление, или отскок, немного выходя за границы. Это регулируется свойством Bounce, а вернее тремя. Bounce on Scroll, в принципе, включает или отключает отскок.



Есть также отдельные свойства, для отскока по вертикали или горизонтали. Их можно настраивать в зависимости от того, в каком направлении вы собираетесь реализовать прокрутку контента.

Ещё одно свойство `contentInset` позволяет задавать дополнительные отступы со всех сторон. Оно принимает структуру `UIEdgeInsets`. Это можно использовать, чтобы реализовать, например, механизм Pull To Refresh. Он происходит, когда пользователь оттягивает scroll view чтобы обновить контент. И на то время, пока идёт запрос данных через интернет, мы увеличиваем `contentInset` чтобы значок с анимацией загрузки не прятался.

```
scrollView.contentInset = UIEdgeInsets  
(top: 80, left: 0, bottom: 0, right: 0)
```



Когда данные пришли, мы опять устанавливаем `contentInset` в исходное значение. В большинстве случаев это `UIEdgeInsets.zero`.

Есть пара свойств, которые отвечают за то, как будет реагировать контент внутри scroll view на жесты пользователя. Например, пользователь начал прокручивать экран, и попал в кнопку, на-

ходящуюся на Scroll View. Встаёт очевидный вопрос: что должно произойти? Прокрутка скроллью или всё-таки нажатие на кнопку? Вот несколько полезных свойств для этого.

```
scrollView.delaysContentTouches = false  
scrollView.canCancelContentTouches = true  
  
+  
  
override func touchesShouldCancel  
    (in view: UIView) -> Bool {  
    return true  
}
```



Если установить DelaysContentTouches в true, то scroll view будет немного задерживать все действия внутри своего контента. Задержка не очень большая. Где-то четверть секунды. Это сделано для того, чтобы точно определить чего хочет пользователь: прокрутить контент или тапнуть по кнопке. Второе свойство – canCancelContentTouches. Оно позволяет scroll view отменять даже длинные касания. Если пользователь попал по кнопке, задержался на ней, но потом снова начал прокрутку. Эти свойства при правильной настройке помогут вашему интерфейсу быть более отзывчивым и вести себя так как этого ожидают.

В завершении хочу рассказать ещё про одно интересное свойство Scroll View. Это Zoom. Он позволяет приближать и удалять контент с использованием жеста Pinch. Или щипок двумя пальцами, как многие называют. Похоже на то как можно увеличивать фотографии в галерее телефона. Активировать его не сложно, но придётся потратить некоторое время чтобы настроить взаимодействие с контентом так, как вам это нужно. Мы не будем вдаваться в подробности реализации, но важно знать что такой функционал доступен в iOS из коробки.

Мы рассмотрели класс ScrollView. Он используется в подавляющем большинстве iOS приложения, особенно там, где размеры контента больше размеров экрана телефона.

## 1.8. Класс UITableView

### 1.8.1. Принципы работы с классом UITableView

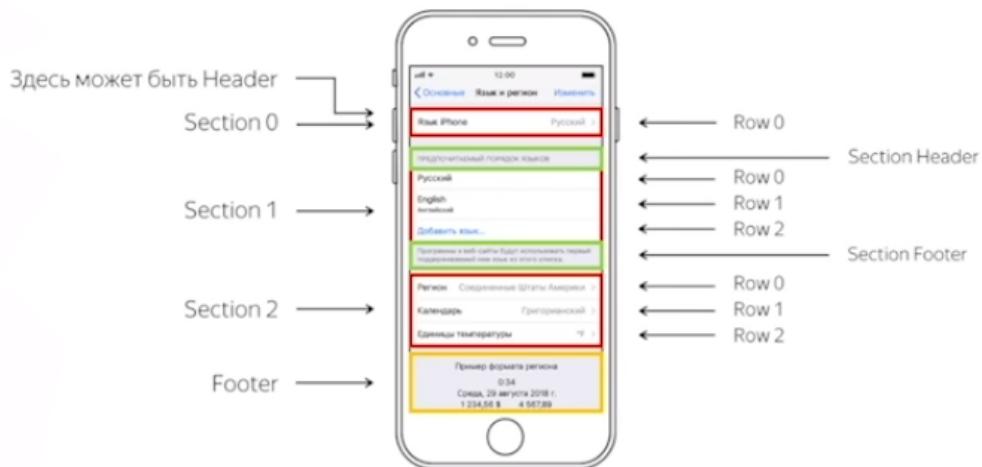
Возможно вы задумывались, как в iOS сделано приложение настроек? Или, допустим, лента новостей или адресная книга. Все они построены на одном и том же компоненте – это UITableView. И без него не обходится ни одна iOS программа. Разве что калькулятор, и то, когда мы заходит добавить туда историю, нам всё равно придётся воспользоваться TableView. Это один из

наиболее сложных компонентов в iOS, учитывая количество его настроек. Но он даёт огромные возможности для организации контента. По сути, любого контента, который можно прокручивать вертикально. Это может быть как простая адресная книга, так и намного более сложная лента новостей в приложении соцсети. Или список с заданиями, которые можно отмечать выполненными.

Стоит оговорить некоторые преимущества. Класс UITableView наследуется от UIScrollView. Это значит, что всё, что вы могли делать в scroll view, доступно и здесь. Помимо этого Table View берёт на себя ответственность за расположение контента. И даже если вы будете использовать Autolayout, в конечном итоге все элементы четко станут на свои места. Таблица старается не расходовать лишнюю память и оптимизирована таким образом, чтобы показывать только тот контент, который находится в видимой области. Это особенно важно, если вам нужно показать тысячи записей.

Итак, начнём со структуры TableView. Таблица использует так называемую двумерную адресацию. Здесь таблица небольшого размера и весь её контент помещается на экран. На первом уровне находятся секции. Тут их три. Как видим, нумерация секций начинается с нуля, и нулевая секция будет всегда сверху. В каждой секции могут присутствовать строки в любом количестве. Включая ноль. А в качестве содержимого строки может быть абсолютно всё что угодно. То есть любые вью и другие графические элементы, которые вы сможете придумать.

Нумерация строк так же начинается с нуля внутри каждой секции. Таким образом и получается двумерная адресация. И для того, чтобы точно указать строку, используется 2 индекса – индекс секции и порядковый индекс строки внутри этой секции. Так для индексации мы оперируем парой чисел.



В дополнение в каждой секции присутствуют верхний и нижний колонтитулы. Или, простыми словами, Header и Footer. Это не обязательные элементы таблицы, и если они не нужны, их

можно просто не задавать. Для настройки заголовков можно использовать простой текст, как на слайде, или же UIView с любым содержимым. И оставшиеся части, это так же Header и Footer, но для всей таблицы. И точно так же вы можете использовать вью с произвольным контентом. Строки в таблице настраиваются с помощью так называемого виртуального режима. Вместо того, чтобы устанавливать контент в таблицу, таблица наоборот, сама запрашивает его в тот момент, когда этот контент ей будет нужен. Здесь используется знакомый нам шаблон проектирования Delegate. Но вместо того, чтобы уведомлять о событиях, большинство методов делегата просят вернуть какие-то данные. Допустим, при старте таблица запрашивает у контроллера сначала количество секций. Потом количество строк для каждой секции. А потом размеры ячеек и наконец сами ячейки. В ответ таблица сообщает контроллеру о действиях пользователя.

## UITableView – Виртуальный режим



Методов в делегате таблицы достаточно много, поэтому их разбили на 2 части. Первая часть – это протокол UITableViewDataSource. Тут собраны методы, настраивающие контент таблицы. Есть методы, запрашивающие количество секций и количество строк, а так же ячейку для строки в тот момент, когда она будет нужна. Это 3 основных метода, использующихся повсеместно. Следующая группа методов позволяет вернуть текст для хедера и футтера в таблице. А ещё можно настроить оглавление, похожее на то, что использует адресная книга. А также различные методы для режима редактирования, о которых я расскажу позже.

```

public protocol UITableViewDataSource : NSObjectProtocol {
    public func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
    public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
    optional public func numberOfSections(in tableView: UITableView) -> Int
    optional public func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
    optional public func tableView(_ tableView: UITableView, titleForFooterInSection section: Int) -> String?
    optional public func sectionIndexTitles(for tableView: UITableView) -> [String]?
    optional public func tableView(_ tableView: UITableView,
                                 sectionForSectionIndexTitle title: String, at index: Int) -> Int
    optional public func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool
    optional public func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool
    optional public func tableView(_ tableView: UITableView,
                                 commit editingStyle: UITableViewCellEditingStyle,
                                 forRowAt indexPath: IndexPath)
    optional public func tableView(_ tableView: UITableView,
                                 moveRowAt sourceIndexPath: IndexPath,
                                 to destinationIndexPath: IndexPath)
}

```

Чуть больше методов находится во второй части делегата таблицы. Это протокол UITableView Delegate. Все методы здесь опциональные. То есть их можно реализовывать только тогда, когда вам понадобится их функционал. Конечно, все мы не запомним. Но я расскажу про самые важные из них.

```

public protocol UITableViewDelegate : UIScrollViewDelegate {
    // Variable height support
    optional public func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat
    optional public func tableView(_ tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat
    optional public func tableView(_ tableView: UITableView, heightForFooterInSection section: Int) -> CGFloat

    optional public func tableView(_ tableView: UITableView, estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat
    optional public func tableView(_ tableView: UITableView, estimatedHeightForHeaderInSection section: Int) -> CGFloat
    optional public func tableView(_ tableView: UITableView, estimatedHeightForFooterInSection section: Int) -> CGFloat

    // Section header & footer information. Views are preferred over title should you decide to provide both
    optional public func tableView(_ tableView: UITableView, viewForHeaderInSection section: Int) -> UIView?
    optional public func tableView(_ tableView: UITableView, viewForFooterInSection section: Int) -> UIView?

    // Selection
    optional public func tableView(_ tableView: UITableView, shouldHighlightRowAt indexPath: IndexPath) -> Bool
    optional public func tableView(_ tableView: UITableView, didHighlightRowAt indexPath: IndexPath)
    optional public func tableView(_ tableView: UITableView, didUnhighlightRowAt indexPath: IndexPath)

    // Editing
    optional public func tableView(_ tableView: UITableView,
                                 editingStyleForRowAt indexPath: IndexPath) -> UITableViewCellEditingStyle
    optional public func tableView(_ tableView: UITableView,
                                 titleForDeleteConfirmationButtonForRowAt indexPath: IndexPath) -> String?
    optional public func tableView(_ tableView: UITableView,
                                 editActionsForRowAt indexPath: IndexPath) -> [UITableViewRowAction]?
}

```

Здесь есть методы, отвечающие за настройку размеров элементов таблицы – строк, хедеров и футеров. Притом вторая группа возвращает лишь предварительные размеры, а реальные размеры элементов рассчитываются через механизм Autolayout.

Через эти методы таблица может запрашивать вью для хедеров и футеров секций. По сути, можно вернуть любой класс, унаследованный от UIView, и содержащий внутри другие элементы.

Следующая группа методов отвечает за выделение ячеек. Это самый простой способ понять что пользователь нажимает на какую-то ячейку. И ещё есть ещё небольшая группа методов для работы в режиме редактирования.

### 1.8.2. UITableView на практике

Теперь посмотрим как таблица работает на практике. Здесь уже создан новый проект SingleView App. Давайте сразу посмотрим на модель данных.

В качестве данных мы будем использовать список рецептов, разбитых по категориям. В структуре категории есть название и собственно список рецептов. Сам рецепт состоит из названия, массива ингредиентов и опциональной иконки. Дальше заполняем наш список рецептов по категориям. Их всего три, в каждой из которых до 20 рецептов. Обратите внимание, что фотографии есть не у всех рецептов. Давайте попробуем показать эти элементы в таблице. В редакторе интерфейса добавим UITableView. Теперь нужно установить delegate и dataSource для таблицы. Это можно сделать в закладке connections inspector. Если сейчас мы попробуем запустить проект, то получим ошибку. Это потому что мы не реализовали обязательные методы из протокола UITableViewDataSource. Сделаем это в отдельном расширении:

```
extension ViewController: UITableViewDataSource, UITableViewDelegate {  
}
```

Теперь Xcode подсказывает, что мы не реализовали все обязательные методы. Давайте воспользуемся этой подсказкой. XCode видит, какие методы мы должны реализовать, и может подставить их прототипы как раз туда куда нужно. Можем реализовывать их, но сначала было бы хорошо получить список рецептов в виде свойства.

```
var categories = RecipeCategory.allRecipes
```

Напомню, что таблица имеет двумерную структуру. Поэтому сначала нам надо вернуть количество секций. Оно будет равно количеству категорий:

```
func numberOfSections(in tableView: UITableView) -> Int {  
    return categories.count  
}
```

С количеством строк для каждой секции так же всё просто. Получаем нужную категорию, а затем получаем для неё количество рецептов:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return categories[section].recipes.count  
}
```

Теперь нужно вернуть настроенную ячейку в методе tableView - cellForRowAt - indexPath. Создаём объект UITableViewCell:

```
let cell = UITableViewCell(style: .subtitle, reuseIdentifier: nil)
```

Мы будем использовать стиль ячейки с подзаголовком, чтобы показать ингредиенты рецепта. Про остальные стили и про reuseIdentifier расскажу чуть позже. Теперь разместим контент. Получает нужный нам рецепт:

```
let recipe = categories[indexPath.section].recipes[indexPath.row]
```

И настраиваем ячейку:

```
cell?.textLabel?.text = recipe.title
cell?.detailTextLabel?.text = recipe.ingredients
```

Осталось вернуть эту ячейку:

```
return cell
```

В завершение было бы неплохо подписать секции, чтобы мы точно знали к какой категории относятся рецепты. Воспользуемся методом tableView:titleForHeaderInSection:

```
func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
    return categories[section].title
}
```

Теперь всё готово. Давайте запустим проект и посмотрим, что получилось.



Рецепты на месте, но списки ингредиентов не вмещаются полностью. Для начала изменим максимальное количество строк в текстовых полях:

```
cell?.textLabel?.numberOfLines = 0  
cell?.detailTextLabel?.numberOfLines = 0
```

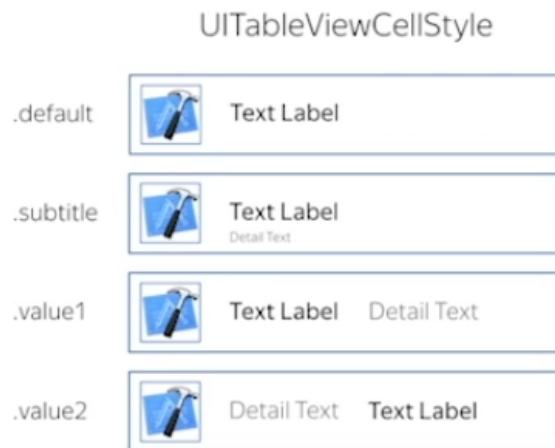
Параметр ноль означает, что количество строк может быть неограниченным. Смотрим что получилось. Ещё обратите внимание на то что заголовок секции прилипает к верху таблицы, когда мы прокручиваем контент. Это один из двух возможных стилей. А сам стиль можно поменять в редакторе интерфейса.

Итак, это базовые действия для работы с классом UITableView.

### 1.8.3. Как работать с ячейками таблицы

Ещё немного информации о настройке ячеек таблицы. Стандартные ячейки могут быть одного из нескольких стилей. Их всего 4, но в большинстве случаев этого достаточно. Обратите внимание что на всех стилях кроме Default, появляется второй элемент с текстом. Это detailsTextLabel. Обратите внимание как он располагается в сочетании с разными стилями.

## UITableView – Стандартные стили ячеек



Если нужно, вы можете настраивать его шрифт и цвет. А так же шрифт и цвет основного текста. Независимо от стиля вы можете воспользоваться свойством imageView для того чтобы установить иконку. Она располагается слева.

## UITableView – Стандартные стили ячеек

```
let cell = UITableViewCell(style: .subtitle, reuseIdentifier: nil)

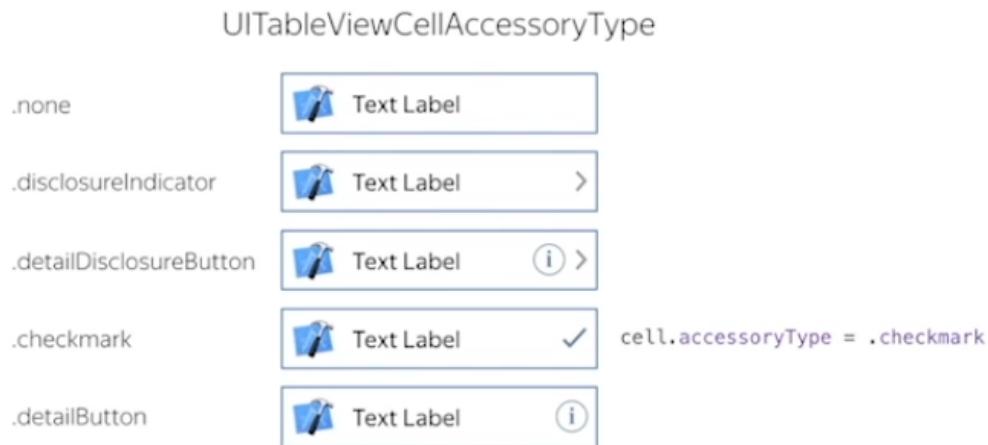
cell.textLabel?.text = "Text Label"
cell.detailTextLabel?.text = "Detail Test"

cell.textLabel?.textColor = .red
cell.detailTextLabel?.font = UIFont.systemFont(ofSize: 10)

cell.imageView?.image = UIImage(named: "Xcode icon")
```

Есть ещё одно интересное свойство. В правой части ячейки можно расположить одну из стандартных иконок.

## UITableView – Стандартные стили ячеек



Задать её можно через свойство accessoryType. Это свойство может принимать одно из пяти значений. По сути, этот элемент показывает пользователю чего ожидать от ячейки. Если на ячейке расположен, например, disclosureIndicator, то пользователь будет ожидать перехода на следующий экран. При использовании типа detailButton, мы получаем дополнительную кнопку на ячейке. Тап по ней можно обработать в методе accessoryButtonTappedForRowWithIndexPath. Этот метод принадлежит протоколу UITableViewDelegate.

## UITableView – Стандартные стили ячеек

```
cell.accessoryType = .detailButton

+
extension ViewController: UITableViewDataSource, UITableViewDelegate {

    func tableView(_ tableView: UITableView,
                  accessoryButtonTappedForRowWith indexPath: IndexPath) {
        let city = citiesFor(section: indexPath.section)[indexPath.row]
        print(city)

        // Do something else...
    }

    // Other methods...
}
```

Также можно менять цвета этих элементов через свойство tintColor. Теперь поговорим о производительности. Чем сложнее интерфейс, тем больше графических элементов присутствует на каждом экране. И тем дольше этот экран загружается в память. Особенно неприятно это выглядит когда пользователь листает красивую ленту новостей, а эта лента на мгновение замирает перед показом каждой следующей новости. Чтобы такого не происходило, приходится максимально облегчать загрузку каждой ячейки. А в идеальном случае эту загрузку вообще исключить.

Давайте разберём как работает UITableView. Здесь в рамочку выделена та область контента, которую пользователь видит. Когда он пролистывает контент, некоторые ячейки уходят за границы экрана. В то же время нам необходимо создавать новые ячейки с новым контентом чтобы показать ту область, в которую переместился пользователь. Идея здесь в том, что те ячейки, которые полностью скрылись из поля зрения, можно использовать повторно, но уже с новым контентом. Остаётся только один нюанс. Таблица может содержать ячейки совершенно разного вида. И, чтобы можно было извлечь ячейку именного того типа, который надо, применяется параметр reuseIdentifier. По сути, это строковый идентификатор группы ячеек, которые могут быть переиспользованы. И, когда нам нужно создать новую ячейку, мы можем попросить у таблицы поискать скрытую ячейку с нужным reuseIdentifier. Потом в ячейке меняем контент на нужный нам. И возвращаем эту ячейку обратно в таблицу уже как новую. Конечно, может так случиться что в таблице не будет ячейки с нужным reuseIdentifier. В этом случае просто создаём новую ячейку.

## Повторное использование ячеек



```
let cell = tableView.dequeueReusableCell(withIdentifier: "Orange", for: indexPath)
```

### 1.8.4. Другие важные особенности таблицы UITableView

Сейчас пройдёмся по остальным важным возможностям класса UITableView. Начнём с размеров ячеек. Сейчас, когда есть механизм Autolayout, работа с таблицей сильно упростилась. Но что если нам нужно четко задать размеры ячеек? Или комбинировать Autolayout с расчетными значениями? Для этого мы можем использовать метод tableView-heightForRowAt-indexPath. Когда точно знаем какой высоты должна быть ячейка мы просто возвращаем нужное значение. Если мы хотим чтобы Autolayout посчитал высоту за нас, мы возвращаем специальную константу UITableViewAutomaticDimension. Это знак для таблицы, что высота такой ячейки будет расчитываться сама. А до того как ячейка будет показана, таблица будет использовать метод tableView-estimatedHeightForRowAt-indexPath для предварительного расчета её размеров.

```
func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
    if indexPath.section == 0 {
        return 60
    } else {
        return UITableViewAutomaticDimension
    }
}

func tableView(_ tableView: UITableView,
              estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat {
    if indexPath.section == 0 {
        return 60
    } else {
        return 120
    }
}
```

В примере мы научились связывать таблицу со статическим контентом. Но что если контент со временем будет меняться. И таблица должна показывать эти изменения. В самом простом случае можно вызвать метод `reloadData`, и таблица полностью перезагрузит содержимое. Не всегда этот приём сработает, потому что иногда хочется анимировать например добавление новой записи. Ведь анимации всегда выглядят приятнее, чем простая перерисовка экрана. Есть несколько методов, позволяющих решать эту проблему. Как видите, в таблице есть методы для добавления, обновления и удаления целых групп ячеек. И тоже самое можно делать с секциями.

## UITableView

Полная перезагрузка таблицы:

```
tableView.reloadData()
```

Динамическое изменение строк:

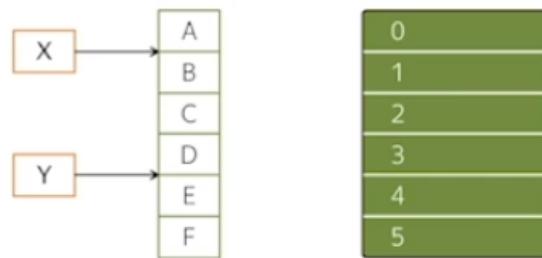
```
func insertRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)
func deleteRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)
func reloadRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)
func moveRow(at indexPath: IndexPath, to newIndexPath: IndexPath)
```

Динамическое изменение секций:

```
func insertSections(_ sections: IndexSet, with animation: UITableViewRowAnimation)
func deleteSections(_ sections: IndexSet, with animation: UITableViewRowAnimation)
func reloadSections(_ sections: IndexSet, with animation: UITableViewRowAnimation)
func moveSection(_ section: Int, toSection newSection: Int)
```

Посмотрим на примере. У нас есть массив моделей, которые отображаются в таблице. Допустим,

это массив рецептов из предыдущего примера.



В результате каких-то действий в массив добавилось два новых рецепта, и нам нужно динамически добавить их в таблицу. Получим индексы добавленных элементов в массиве. Это индексы один и пять. Чтобы таблица приняла эти изменения, мы должны вызвать метод `insertRows`, и передать туда индексы добавленных элементов. Здесь есть один важный момент. При динамическом обновлении контента таблицы все данные должны быть консистентны. То есть если вы вызвали команду на добавления двух строк, то метод `tableView-numberOfRowsInSection` должен вернуть на 2 строки больше чем возвращал в предыдущий раз. Поэтому правило здесь такое. Сначала обновляется модель, затем за одну команду мы должны внести все изменения модели на таблицу.



```
let indexPaths = [IndexPath(row: 1, section: 0),
                 IndexPath(row: 5, section: 0)]

tableView.insertRows(at: indexPaths, with: .fade)
```

Если мы хотим одновременно удалить и добавить ячейки, то все изменения можно поместить между методами `tableView beginUpdates` и `endUpdates`. Главное не напутать с индексами. Если вам такое обновление кажется слишком трудным, пользуйтесь методом `reloadData`. Он сильно облегчит задачу, но красивых анимаций не будет.

```
tableView.beginUpdates()

let indexPathsToAdd = [IndexPath(row: 1, section: 0),
                      IndexPath(row: 5, section: 0)]

let indexPathsToRemove = [IndexPath(row: 0, section: 0),
                         IndexPath(row: 3, section: 0)]

tableView.insertRows(at: indexPaths, with: .fade)
tableView.deleteRows(at: indexPaths, with: .fade)

tableView.endUpdates()
```

Теперь пару слов о выделении ячеек. Разработчики таблицы не хотели ограничиваться простым нажатием на ячейку, и они создали намного более универсальный механизм. Когда пользователь нажимает на ячейку, он её выделяет. И это событие вызывает соответствующий метод в делегате – tableViewDidSelectRowAtIndexPath. Это и есть нужное нам нажатие на ячейку. Здесь можем получить выбранные пользователем рецепт. И открыть контроллер, который показывает подробную информацию о рецепте. На самом деле в UITableViewDelegate есть очень много методов, которые позволяют настроить практический каждый аспект выделения ячеек. Просто попробуйте поэкспериментировать самостоятельно. И вы поймёте на сколько просто можно настроить любое поведение.

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let recipe = categories[indexPath.section].recipes[indexPath.row]
    // Показываем виджет контроллер с рецептом
    // showRecipeDetailsViewController(recipe)
}

func tableView(_ tableView: UITableView, shouldHighlightRowAt indexPath: IndexPath) -> Bool {
    // Нужно ли показывать нажатие на ячейку?
    return true
}

func tableView(_ tableView: UITableView, didHighlightRowAt indexPath: IndexPath) {
    // Обрабатываем графику нажатия на ячейку
}

func tableView(_ tableView: UITableView, didUnhighlightRowAt indexPath: IndexPath) {
    // Обрабатываем графику нажатия на ячейку
}
```

В самом классе UITableView тоже есть несколько свойств, отвечающий за выделение. Во-первых включать или отключать выделение. В том числе множественное выделение строк. То есть пользователь может выделять сразу несколько ячеек. А таблица при это будет хранить выделенные индексы в свойстве indexPathsForSelectedRows. Выделением так же можно управлять программно с помощью методов selectRowAtIndexPath и deselectRowAtIndexPath.

```
class UITableView {

    // Настройка выделений:
    var allowsSelection: Bool

    var allowsMultipleSelection: Bool

    // Индексы выделенных строк:
    var indexPathForSelectedRow: IndexPath? { get }

    var indexPathsForSelectedRows: [IndexPath]? { get }

    // Принудительное выделение строк:
    func selectRow(at indexPath: IndexPath?,
                  animated: Bool,
                  scrollPosition: UITableViewScrollPosition)

    func deselectRow(at indexPath: IndexPath, animated: Bool)
}
```

И весь этот механизм спроектирован так чтобы работать вместе с объектами ячеек. Поэтому в класс UITableViewCell были добавлены методы setSelected и setHighlighted. Они вызываются каждый раз, когда ячейка меняет своё выделение.

```

class RecipeTableViewCell: UITableViewCell {
    override func setSelected(_ selected: Bool, animated: Bool) {
        // Обрабатываем нажатие
        refresh()
    }
    override func setHighlighted(_ highlighted: Bool, animated: Bool) {
        // Обрабатываем выделение
        refresh()
    }

    func refresh() {
        if self.isSelected {
            // Ячейка выделена
        }
        if self.isHighlighted {
            // Ячейка нажата
        }
    }
}

```

Если мы хотим, чтобы ячейка как-то реагировала на выделение, переопределяем этот метод в своём классе-потомке и реализуем реакцию на выделение самостоятельно.

Итак, класс UITableView – это очень удобный компонент для того, чтобы показать практически любой контент. Мы разобрали только базовые свойства класса. Но этого уже достаточно для того, чтобы создавать полноценные приложения.

### 1.8.5. Другие важные особенности таблицы UITableView

Давайте разберемся как работает этот механизм на реальном примере. Допустим, создание каждой ячейки занимает достаточно много времени, и мы хотим этот процесс ускорить. Добавим константу, в которой будет лежать reuseIdentifier:

```
let cellIdentifier = "recipe cell"
```

Теперь попытаемся извлечь ненужную ячейку из таблицы. Это делается методом dequeueReusableCellReusableCell:

```
var cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier)
```

Если ячейка действительно нашлась, то можно переходить к её настройке. Но сначала обрабатаем случай, когда нужной ячейки не было:

```
if cell == nil {
    cell = UITableViewCell(style: .subtitle, reuseIdentifier: cellIdentifier)
}
```

Мы передаём идентификатор ячейки в параметр reuseIdentifier. Будьте очень внимательны, когда назначаете его. Иначе в будущем могут появится непредсказуемые ошибки. Также здесь можно разместить общие настройки ячейки:

```
cell.textLabel?.numberOfLines = 0  
cell.detailTextLabel?.numberOfLines = 0
```

Чтобы убедиться, что механизм работает, добавим вывод в лог. Будем выводить текст каждый раз, когда мы создали или переиспользовали ячейку:

```
if cell == nil {  
    cell = UITableViewCell(style: .subtitle, reuseIdentifier: cellIdentifier)  
    print("New cell")  
} else {  
    print("Cell reused")  
}
```

Остальной код оставляем как есть. Теперь запускаем проект. Посмотрим что вывелоось в консоль. Обратите внимание, здесь создалось столько ячеек, сколько помещается на экран. Когда мы начинаем прокручивать контент, создаётся максимум одна запасная ячейка, а все остальные начинают переиспользоваться.

Сейчас попробуем создать и настроить собственную ячейку для таблицы. Допустим, в списке рецептов нам нужно показать большую фотографию блюда. И немного поэкспериментировав со стандартной ячейкой UITableViewCell становится понятно, что иконку не выйдет сделать такого вида как хочется. Для этого создадим собственную ячейку. Добавляем в проект файл с шаблоном Cocoa Touch Class. Вводим имя класса, RecipeTableViewCell. Так как мы создаём ячейку для таблицы, то сабклассом должен быть UITableViewCell. Если название сабкласса написано правильно, то галочка Also Create Xib file станет активной. Не забудьте выделить её. Завершаем создание ячейки.

Переходим в графический файл. Здесь есть пара отличий от файла, созданного для вью контроллера. Во первых, тот объект, с которым мы будем настраивать связи, находится не в File's Owner. Это будет корневой объект в списке элементов. В этом можно убедиться если посмотреть его тип. Как видим, тут он совпадает с типом созданной ячейки. А если мы посмотрим на тип File's Owner, то здесь будет пусто. Это связано с тем, что вью контроллер лишь настраивался с помощью Xib файла, а ячейки обычно полностью создаются из этого файла без использования стандартного конструктора.

Итак, мы создаём ячейку для того, чтобы показать рецепт блюда с его фотографией. Нам нужно 2 лейбла для названия и списка ингредиентов и место куда мы разместим фотографию. Давайте создадим их. Теперь нужно настроить ограничения. Не забывайте добавить ограничения снизу и справа, чтобы ячейка смогла правило рассчитать собственный размеры.

Вспомним, что название и список ингредиентов могут быть слишком длинными. Чтобы не вылезать за экран, настроим максимальное количество строк для каждого лейбла. А также изменим стиль заголовка чтобы ячейка лучше смотрелась.

В картинке настроим правильное масштабирование контента. И включим флаг Clip To Bounce, чтобы контент не вылезал за пределы элемента.

Графика настроена. Теперь добавим аутлеты на лейблы и картинку:

```
iconImageView: UIImageView!
recipeTitleLabel: UILabel!
recipeIngredientsLabel: UILabel!
```

Можно переходить к коду. Для начала откроем класс ячейки – RecipeTableViewCell.swift. Здесь есть метод awakeFromNib. Обычно сюда размещают всю дополнительную настройку, если этого нельзя сделать через редактор интерфейсов. Метод чем-то похож на viewDidLoad из вью контроллера. Теперь используем созданную ячейку, чтобы показать все рецепты с картинками. Для этого вернёмся во вью контроллер.

Конечно, можно создать ячейку в методе cellForRowAtIndexPath так, как делали раньше. Но в table view есть более удобный способ. Мы регистрируем Xib файл с нужным ReuseIdentifier в таблице. И потом, когда хотим получить объект этой ячейки, используем метод dequeueReusableCell(withIdentifier:for:indexPath). Он сначала ищет ячейку, которую можно переиспользовать. Если не находит, то загружает зарегистрированный Xib файл. В итоге мы получим готовую для настройки ячейку, и нам не нужно создавать её руками. Начнём с регистрации. Находим метод viewDidLoad и регистрируем ячейку:

```
tableView.register(UINib(nibName: "RecipeTableViewCell", bundle: nil), forCellReuseIdentifier: "recipe")
```

Удаляем код в методе cellForRowAtIndexPath, он нам больше не нужен. Теперь получаем ячейку с нужным ReuseIdentifier:

```
let cell = tableView.dequeueReusableCell(withIdentifier: "recipe", for: indexPath)
```

Метод dequeueReusableCell возвращает объект типа UITableViewCell. Поэтому мы должны сделать преобразование к типу RecipeTableViewCell

```
let cell = tableView.dequeueReusableCell(withIdentifier: "recipe", for: indexPath) as! RecipeTableViewCell
```

Кстати, нам больше не нужна проверка на nil. Table View либо создаёт новую ячейку либо переиспользует существующую. Теперь получаем рецепт:

```
let recipe = categories[indexPath.section].recipes[indexPath.row]
```

И настраиваем поля в ячейке:

```
cell.recipeTitleLabel?.text = recipe.title
cell.recipeIngredientsLabel?.text = recipe.ingredients
cell.iconImageView?.image = recipe.photo
```

Возвращаем ячейку как результат метода:

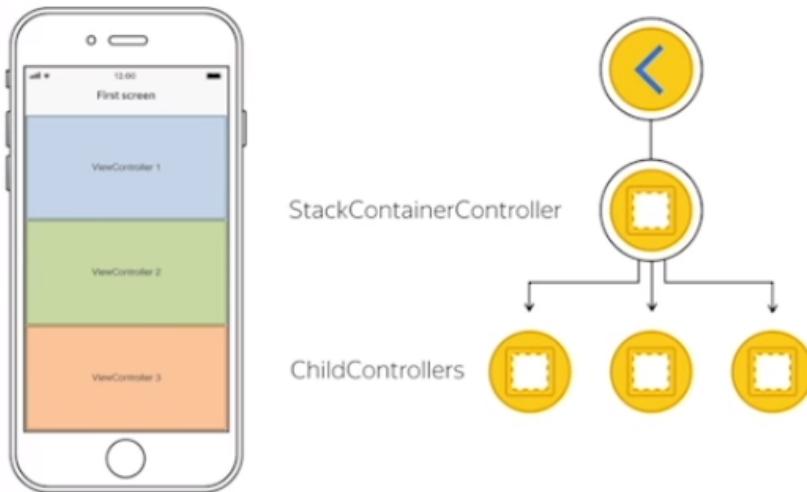
```
return cell
```

Готово, давайте посмотрим что получилось. Вот таким способом можно создавать и настраивать собственные ячейки для таблицы.

## 1.9. Собственные контейнер-контроллеры

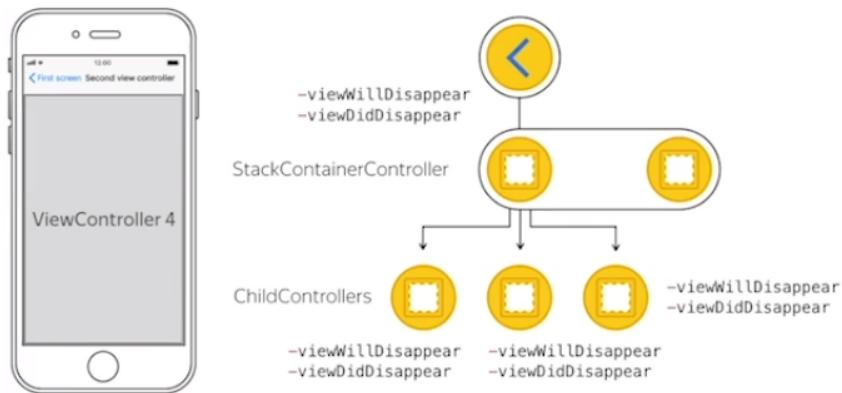
Давайте разберемся как сделать собственные контейнер-контроллеры. Как мы знаем, это виду контроллеры, которые могут размещать внутри себя другие контроллеры. При этом после добавления они образуют так называемую child-parent связь. То есть добавленный контроллер становится дочерним, а тот? в который добавили – становится родителем. Естественно, у родителя может быть больше чем один дочерний виду контроллеров.

Но зачем нужна эта child-parent иерархия? Представим вот такую схему.



В основе приложения у нас есть навигейшн контроллер. И на него добавлен контейнер контроллер, который мы сами создали. Назовём его `StackContainerController`. Он умеет хранить в себе несколько дочерних виду контроллеров один над другим. Как `UISplitViewController`, только количество дочерних контроллеров здесь может быть любым. А теперь мы делаем push и помещаем в главный навигейшн контроллер новый экран. При этом созданный нами контейнер

пропадает. Если вспомнить view controller life cycle, то в этот момент в первом вью контроллере должны вызываться методы `viewWillDisappear` и `viewDidDisappear`. И эти же методы должны вызываться для всех его дочерних вью контроллеров. Но без правильной настройки child-parent связи этого не произойдёт. Точно так же для дочерних контроллеров не будут вызваны методы `viewWillAppear` и `viewDidAppear`, когда мы вернёмся назад на первый экран.



То есть первое, что нужно помнить, когда вы создаёте собственный контейнер контроллер – это то, что вы должны обязательно настроить child-parent связь и правильно управлять ей. Больше никаких ограничений нет. Вью дочернего контроллера может располагаться где угодно и быть любого размера.