



Stanford CS193p

Developing Applications for iOS
Fall 2017-18



CS193p
Fall 2017-18

STANFORD UNIVERSITY

School of Engineering

Developing Apps for iOS CS193P Fall 2017:

Разработка iOS 11 приложений с помощью Swift

Лекция 9: View Controller Lifecycle and Scroll View. (“Жизненный цикл” View Controller и Scroll View.)

October 23, 2017

Профессор Пол Хэгартி (Paul Hegarty)

[Независимая, неавторизованная транскрипция ©
2017 Paul Hegarty]

НАЧАЛО ЛЕКЦИИ

[----- 5 -ая минута лекции -----](#)
[----- 10 -ая минута лекции -----](#)
[----- 15 -ая минута лекции -----](#)
[----- 20 -ая минута лекции -----](#)

----- 25 -ая минута лекции -----
----- 30 -ая минута лекции -----
----- 35 -ая минута лекции -----
----- 40 -ая минута лекции -----
----- 45 -ая минута лекции -----
----- 50 -ая минута лекции -----
----- 55 -ая минута лекции -----
----- 60 -ая минута лекции -----
----- 65 -ая минута лекции -----
----- 70 -ая минута лекции -----
----- 74 -ая минута лекции -----
----- КОНЕЦ ЛЕКЦИИ 9 -----

Добро пожаловать на Лекцию 9. Да, это курс Стэнфорда **CS193P** Осень 2017 (**Fall 2017**)

[“Разработка iOS приложений”](#) ([запасная ссылка](#)).

Today

- ➊ **View Controller Lifecycle**
Keeping track of what's happening in your Controller as it goes through its lifetime
- ➋ **ScrollView**
A UIView that lets you scroll around and zoom on other UIViews

Сегодня

- “Жизненный цикл” View Controller

Отслеживает, что происходит в вашем Controller на протяжении его “жизни”.

- Scroll View

UIView, который позволяет прокручивать и масштабировать другие UIViews.

Сегодня у нас две основные темы. Мы будем говорить о “жизненном цикле” **View Controller**, который “оживает” вместе с **MVC**, затем делает то, что он должен делать, и, в итоге, “умирает”.

Затем мы поговорим об **UIView**, который называется **ScrollView** и который позволяет просматривать очень большие **UIViews** путем их прокрутки, увеличения масштаба и т.д..

Итак, “жизненный цикл” **View Controller**.

Все **View Controllers** имеют “жизненный цикл”, который, по существу, помечается некоторой последовательностью сообщений, методами, вызываемыми **UIViewController**.

Почему нас так интересует “жизненный цикл” **View Controller**?

Потому что мы очень часто хотим делать определенные вещи в различных ключевых точках “жизненного цикла” **View Controller**.

View Controller Lifecycle

- View Controllers have a “Lifecycle”

A sequence of messages is sent to a View Controller as it progresses through its “lifetime”.

- Why does this matter?

You very commonly override these methods to do certain work.

- The start of the lifecycle ...

Creation.

MVCs are most often instantiated out of a storyboard (as you’ve seen).

There are ways to do it in code (rare) as well which we will cover later in the quarter.

- What then?

Preparation if being segued to.

Outlet setting.

Appearing and disappearing.

Geometry changes.

Low-memory situations.



CS193p
Fall 2017-18

Жизненный цикл View Controller

- У View Controllers есть “жизненный цикл”

Система посылает View Controller последовательность сообщений по мере прохождения им этапов “жизненного цикла”.

- Почему это имеет значение?

Обычно вы переопределяете эти методы, чтобы выполнить определенную работу.

- Начало “жизненного цикла”...

Создание.

MVCs чаще всего создаются автоматически из storyboard (как вы уже это видели).

Существуют также способы создания MVC в коде, которые мы рассмотрим позже в этом курсе.

- Что затем?

Подготовка в случае, если к вам кто-то “переезжает” (segue).

Установка outlets.

Появление на экране (Appearing) и исчезновение (Disappearing) с экрана.

Изменения геометрии.

Ситуации недостатка “памяти”.

Начинается “жизненный цикл” View Controller с создания (**Creation**) **UIViewController**. В 99.99% случаев View Controller инициализируется, “приходя” в “жизнь” со **storyboard**. Вы вытягиваете что-то из Палитры Объектов на **storyboard**, размещаете там свои **views**. Вы знаете, как создавать **MVC**. Ваш “жизненный цикл” начинается, когда запускается приложение и кто-то “переезжает” на вас с помощью **Segue**, или когда вы являетесь первым View Controller на вашей **storyboard**.

У iOS также есть некоторый **API**, который мы еще не изучали, но он может вручить вам **View Controller**, на который вы впоследствии можете “переехать” (**segue**) или как-то его использовать. Например, есть **View Controller**, который позволяет вам получить изображение с помощью фотокамеры. Он может быть также получен в коде, а не со **storyboard**.

Но как только **View Controller** создан, то что с ним происходит?

Вы уже многое знаете о “жизненном цикле” **View Controller**.

Конечно, первая вещь, которая происходит сразу же после того как **View Controller** создан, это подготовка View Controller на случай, если к нему кто-то “переезжает” (**segue**).

Затем, после этого происходит очень важный шаг - установка Outlets, ваших кнопок **UIButtons** и им подобных вещей, они становятся подсоединенными.

Затем ваш **View Controller** появляется на экране и, возможно, исчезает с экрана. Представьте себе **Split View** в портретном режиме, когда **Master** исчезает, и вы можете заставить его обратно “скользить” на экран и также “скольжением” уходить обратно.

После мере того, как все это происходит, у вас может меняться геометрия. Прежде всего это происходит, когда у вас вращается устройство и вы переходите от “высокого” и “узкого” к “короткому” и “широкому” прямоугольнику. То есть происходит изменение “геометрии”. Это также может происходить и по другим причинам. И опять же в случае со **Split View**, как вы знаете, ваш **Master** иногда может быть очень “высоким” в портретном режиме и гораздо “короче”, когда он находится слева в ландшафтном режиме, а другой **Detail MVC** находится справа, но в портретном режиме он занимает полностью весь экран. В ландшафтном режиме **Master**, и **Detail** занимают только часть экрана. Так что “геометрия” меняется в зависимости от ситуации, в которой находится ваш **View Controller**.

И, наконец, последняя, но менее значимая ситуация нехватки памяти, в которой ваш **View Controller** могут попросить освободить некоторую память, и мы поговорим об этом через секунду.

Давайте поговорим обо всех методах, которые посылаются вашему **View Controller** при наступлении всех этих событий.

Вы уже знаете один такой метод - это **viewDidLoad**, я упоминал о нем в вашем домашнем Задании.

View Controller Lifecycle

Primary Setup

You already know about `viewDidLoad` ...

```
override func viewDidLoad() {  
    super.viewDidLoad() // always let super have a chance in lifecycle methods  
    // do the primary setup of my MVC here  
    // good time to update my View using my Model, for example, because my outlets are set  
}
```

Do not do geometry-related setup here! Your bounds are not yet set!

Жизненный цикл ViewController

- Основные установки

Вы уже знаете о viewDidLoad ...

```
override func viewDidLoad () {  
    super.viewDidLoad() //всегда давайте super шанс в ваших методах  
    //“жизненного” цикла  
    // выполняйте основные вашего MVC здесь  
    // подходящее время для модификации View, используя Модель.  
    // Например, потому что мои Outlets уже установлены  
}
```

Не делайте здесь никаких установок, связанных с “геометрией”!

Ваши границы bounds еще не установлены.

Это реально очень мощное место для выполнения инициализации, потому что вы уже “подготовлены” и все ваши **Outlets** уже установлены, так что теперь вы реально можете преуспеть. Это прекрасное место для инициализации, и мы обычно размещаем большую часть кода инициализации в методе **viewDidLoad** за одним гигантским ИСКЛЮЧЕНИЕМ. Оно представлено на слайде красным цветом, и вы не должны упустить его. Это ГЕОМЕТРИЯ.

Когда выполняется метод **viewDidLoad**, ваши границы **bounds** пока еще не определены, так что не размещайте в нем вещи, связанные с размером экрана, потому что вы еще не знаете, на какого типа устройстве вы находитесь. Это очень ВАЖНО. Очень распространенная ошибка связана с размещением в **viewDidLoad** кода, использующего “геометрию”, а потом вы будете удивляться, почему при запуске на другом устройстве ваш код больше не работает. Не размещайте здесь НИКАКОЙ “ГЕОМЕТРИИ”, через мгновение я покажу вам, где нужно размещать код, связанный с изменением “геометрии”.

Итак, мы с вами поговорили о замечательном методе **viewDidLoad**.

Следующий метод - **viewWillAppear**.

View Controller Lifecycle

⌚ Will Appear

This method will be sent just before your MVC appears (or re-appears) on screen ...

```
override func viewWillAppear(_ animated: Bool) {  
    super.viewWillAppear(animated)  
    // catch my View up to date with what went on while I was off-screen  
}
```

Note that this method can be called repeatedly (vs. viewDidLoad which is only called once).

Жизненный цикл ViewController

- Will Appear (ПЕРЕД появлением на экране)

Этот метод будет послан непосредственно перед тем, как MVC появляется (или появляется заново) на экране...

```
override func viewWillAppeар(_ animated: Bool) {  
    super.viewWillAppeар(animated)  
    // синхронизирую свой View с тем, что произошло, пока я отсутствовал на экране.  
}
```

Заметьте, что этот метод может вызываться много раз (в противоположность методу `viewDidLoad`, который вызывается лишь один раз).

Между прочим, заметьте, что во всех этих методах язываю `super.viewWillAppeар` или `super.viewDidLoad`, мы всегда это делаем, не забывайте об этом. Как-то на днях в одном из демонстрационных примеров я заметил, что забыл это сделать, так что в сегодняшнем демонстрационном примере я это исправлю. Кстати, я заметил, что в своих домашних Заданиях многие из вас незывают `super.viewDidLoad`. Конечно, мы еще не говорили о “жизненном цикле” **View Controller**, так что я не виню вас в этом. Но с данного момента не забывайте это делать с `super`, дайте `superclass` шанс выяснить, появится ли он на экране или уже загружен или еще что-то.

Итак, это метод `viewWillAppeар`, что мы обычно делаем в методе `viewWillAppeар`?

Метод `viewWillAppeар` вызывается именно тогда, как это следует из его названия, то есть непосредственно ПЕРЕД появлением **View Controller** на экране. И именно здесь вы можете “наверстать” те изменения в окружающем Мире, которые произошли пока ваш **View Controller** отсутствовал на экране. Если же это первое появление **View Controller** на экране, то вы можете здесь “схватить” текущее состояние окружающего Мира. Это самое подходящее место для загрузки в ваш View всей необходимой информации из вашей **Модели**, особенно, если **Модель** могла измениться как, например, в случае сетевой базы данных в качестве **Модели**.

----- 5-ая минута лекции -----

К этому времени другие люди могли что-то отредактировать или изменить в ней. Это очень удобный метод для обновления вашего View в соответствие с **Моделью**, потому что пока ваш View не появился на экране, вы не хотите, чтобы ваш **View Controller** делал какую-то большую работу. Вы хотите, чтобы он типа “сидел тихонечко”, но как только наступает время возвращаться на экран, то он должен сделать небольшие настройки в методе `viewWillAppeар`.

Другой особенностью метода `viewWillAppeар`, конечно же, является возможность повторных вызовов, потому что ваш **View Controller** может уходить с экрана и возвращаться вновь на экран, уходить и возвращаться. В то время как метод `viewDidLoad` вызывается всего один раз в течение “жизненного цикла” вашего **View Controller**. Он вызывается единственный раз после того, как он “подготовлен” (**prepared**) и его **Outlets** установлены. Он вызывается, и на этом все.

В противоположность методу `viewDidLoad` метод `viewWillAppeар`, очевидно, вызывается всякий раз, когда **View Controller** возвращается на экран.

Существует также метод `viewDidAppear`. Он вызывается ПОСЛЕ того, как **View Controller** появился на экране.

View Controller Lifecycle

• Did Appear

You also find out after your MVC has finished appearing on screen ...

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    // maybe start a timer or an animation or start observing something (e.g. GPS position)?  
}
```

This is also a good place to start something expensive (e.g. network fetch) going.

Why kick off expensive things here instead of in viewDidLoad?

Because we know we're on screen so it won't be a waste.

By "expensive" we usually mean "time consuming" but could also mean battery or storage.

We must never block our UI from user interaction (thus background fetching, etc.).

Our UI might need to come up incomplete and later fill in when expensive operation is done.

We use "spinning wheels" and such to let the user know we're fetching something expensive.



CS193p
Fall 2017-18

Жизненный цикл ViewController

• Did Appear (ПОСЛЕ появления на экране)

Этот метод будет послан после того, как ваш MVC завершит свое появление на экране...

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
  
    // возможно стартовать таймер или анимацию или  
    // начать наблюдать что-то ( например, позицию GPS)?  
}
```

Это также хорошее место для старта чего-нибудь затратного (например, выборки из сети).

Почему лучше стартовать затратные вещи именно здесь, а не в viewDidLoad?

Потому что мы знаем, что мы уже на экране и это не будет сделано впустую.

Под "затратностью" мы обычно понимаем "отнимающее много времени", но под этим можно также понимать батарею или память.

Мы НИКОГДА Не должны блокировать наш UI для взаимодействия с пользователем (именно поэтому осуществляется фоновая выборка и т.д.).

Наш UI может появиться на экране не полностью укомплектованным, но позже они заполнится, как только затратная операция будет закончена.

Мы используем "вращающееся колесико", чтобы пользователь знал, что мы делаем некоторую затратную выборку.

Что мы можем делать в методе `viewDidAppear`? Вы этом методе слишком поздно выполнять такие вещи, как обновление вашего View согласно Модели, так как все уже находится на экране. Вы же не

хотите, чтобы на экране появлялись что-то неправильное, затем срабатывал бы **viewDidAppear** и вы вычищали бы все, что появилось неправильное.

Но методе **viewDidAppear** - это прекрасное место для таких вещей, как начало анимации, потому что вы не можете это делать в **viewWillAppear**, когда вы еще не на экране. Вы можете здесь стартовать таймер, который что-то делает на экране. Можете начать наблюдение (**observing**) чего-то в Мире наподобие **GPS** местоположения или, может быть, позиции гироскопа в вашем **iPhone**. Все эти вещи вы можете сделать сразу же, появившись на экране. Вот для чего хорош метод **viewDidAppear**.

Еще одна вещь, которую вы можете делать в **viewDidAppear**, это, возможно, запуск какой-то очень затратной процедуры. Вы не хотите запускать такие вещи в методе **viewDidLoad**, потому что когда выполняется **viewDidLoad**, вы не можете гарантировать, что ваш **View Controller** появится на экране. Поэтому **viewDidLoad** - не самое подходящее место для этого, даже **viewWillAppear**, как это не удивительно, может быть вызван и все таки не появится на экране. Но находясь в методе **viewDidAppear**, вы точно знаете, что вы уже на экране, так что имеет смысл делать что-то затратное.

Что может быть этим “затратным”?

Допустим, вы хотите загрузить гигантское изображение из интернета, в большинстве случаев, если вы хотите выбрать что-то из интернета, это будет достаточно затратно по времени. Потому что на **iPhone** у вас может быть только сотовая связь (**cellular connection**), и это может быть плохая связь из-за того, что вы находитесь где-то на деревенской дороге, на которой едва ли можно вообще получить сотовую связь. Это действительно может быть достаточно затратно. По той же самой причине мы обычно выполняем такие затратные вещи в фоновом (**background**) режиме. И в Среду я буду говорить о том, как разместить определенные вещи в фоновом (**background**) режиме.

Потому что абсолютно главная цель, первостепенное требование, которому вы ДОЛЖНЫ следовать в любом **iOS** приложение - НИКОГДА НЕ БЛОКИРОВАТЬ пользовательский интерфейс (**UI**). Пользователи всегда должны иметь возможность касаться каких-то элементов на экране, например, выполнять жест **swipe**. Если вы касаетесь чего-то на экране или выполняете жест **swipe**, а ваше приложение “заморожено”, пользователи никогда не будут использовать ваше приложение, уж поверьте мне. Это чрезвычайно важно. Способ, каким мы будем этого добиваться, заключается в том, что мы размещаем все, что могло бы блокировать наш **UI** типа ожидания чего-то из интернета, за пределами **main queue**, так мы называем главный поток **main thread**, мы размещаем это в фоновых (**background**) процессах. Мы будем говорить об этом в Среду, но метод **viewDidAppear** - это прекрасное место для того, чтобы запускать такие затратные вещи, потому что вы не потеряете напрасно свои сотовые данные, полученные при выборке изображения, если вы реально не появитесь на экране. Однако это означает, что запустив выборку данных в фоновом (**background**) режиме, вы должны так сконструировать свой **UI**, чтобы он работал и в случае, когда затратные данные еще не вернулись из фонового режима. Понимаете почему? Эта затратная вещь может потребовать 10 минут или вообще не вернуться из-за плохой работы сети, например. Вам необходимо вставить либо место заменитель, либо использовать “вращающееся колесико” с

надпись “Идет загрузка ...” или разместить некоторую анимацию, которая показывала бы пользователю, что я выбираю что-то из интернета, я работаю над этим, но при этом ваш **UI** должен все еще оставаться в состоянии быстро реагировать на действия пользователя, **UI** должен быть “отзывчивым”. Если вы находитесь в **Navigation Controller**, то у пользователя должна оставаться возможность кликнуть на кнопке “**Back**” и вернуться назад. Или кликнуть на закладке внизу и перейти к другой закладке или делать то, что они хотят. Конструирование подобного рода **UIs** требует небольшой структурной перестройки. До сих пор вы не делали ничего подобного. В большинстве случаев вы мыслили линейно: для того, чтобы разместить изображение (**image**) на экране, я сначала получаю это изображение, а затем размещу его на экране. Вы не можете так мыслить в случае использования выборки изображения в фоновом режиме. Вы должны мыслить следующим образом: если я хочу разместить изображение на экране, то я размещу на экране что-то вместо этого изображения, которое еще не загрузилось, с таким маленьким “вращающимся колесиком”, при этом пользователь может делать все, что угодно. Затем позже, когда изображение будет доставлено, именно тогда я обновлю мой **UI**, чтобы показать “прибывшее” изображение. Вы как бы мыслите в другом измерении, в другом временном измерении.

Существует также метод **viewWillDisappear**, он вызывается непосредственно ПЕРЕД тем, как **View** исчезнет с экрана. Это прекрасное место для UNDO того, что вы делали в **viewDidAppear**. Если вы стартовали там таймер или анимацию или начали следить за **GPS** или делать что-то подобное, то метод **viewWillDisappear** - это прекрасное место для того, чтобы перестать это делать, потому что вы знаете, что через мгновение вы исчезните. Но когда вы вернетесь на экран, вы включите все это снова в методе **viewDidAppear**.

View Controller Lifecycle

- Will Disappear

Your MVC is still on screen, but it's about to go off screen.
Maybe the user hit “back” in a **UINavigationController**?
Or they switched to another tab in a **UITabBarController**?
`override func viewWillDisappear(_ animated: Bool) {
 super.viewWillDisappear(animated)
 // often you undo what you did in viewDidAppear
 // for example, stop a timer that you started there or stop observing something
}`

Жизненный цикл ViewController

- Will Disappear (ПЕРЕД уходом с экрана)

Ваш MVC все еще на экране, но он через мгновение покинет экран.

Возможно, пользователь кликнул “Back”, находясь в **UINavigationController**?

Или он переключился на другую закладку (tab) в **UITabBarController**?

```
override func viewWillDisappear(_ animated: Bool) {  
    super.viewWillDisappear(animated)  
    // часто вы делаете “Undo” того, что делали в viewDidAppear
```

```
// например, останавливаете таймер, который там стартовал или  
// останавливаете отслеживание чего-нибудь  
}
```

Фактически, эти два метода - **viewDidAppear** и **viewWillDisappear** - рассматриваются как зеркальное отображение друг друга. Это своего рода DO и UNDO некоторых действий, которые включаются и выключаются при появлении View на экране и при уходе его с экрана. Существует также метод **viewDidDisappear**, он вызывается ПОСЛЕ того, как вы полностью исчезли. Обычно не так много можно сделать в этом методе. Может быть, "почистить" ваш **MVC**, возможно сохранив некоторое его состояние.

View Controller Lifecycle

• Did Disappear

Your MVC went off screen.

Somewhat rare to do something here, but occasionally you might want to "clean up" your MVC.

For example, you could save some state or release some large, recreatable resource.

```
override func viewDidDisappear(_ animated: Bool) {  
    super.viewDidDisappear(animated)  
    // clean up MVC  
}
```

Жизненный цикл ViewController

• Did Disappear (ПОСЛЕ ухода с экрана)

Ваш MVC ушел с экрана.

Очень редко что-то можно сделать здесь, но во всяком случае вы, может быть, захотите "почистить" ваш MVC.

Например, вы могли бы сохранить некоторое состояние или освободить некоторый большой, но воссоздаваемый ресурс.

```
override func viewDidDisappear(_ animated: Bool) {  
    super.viewDidDisappear(animated)  
    // чистка MVC  
}
```

Метод **viewDidDisappear** очень редко используется.

----- 10-ая минута лекции -----

Итак, я сказал вам, что в методе **viewDidLoad** вы не можете работать с "геометрией".

Где же вы работаете с "геометрией"?

Вы можете подумать, что это можно делать в методе **viewWillAppear**, потому что я уже практически готов появиться на экране и моя "геометрия" полностью определена.

Но НЕТ. Вы можете так думать, но вы будете неправы. Вы также не можете это делать в методе **viewDidAppear**, потому что вы уже на экране и слишком поздно делать что-либо с "геометрией".

Место, в котором нужно работать с "геометрией" находится в двух методах:

viewWillLayoutSubviews() и **viewDidLayoutSubviews()**. Эти два метода посылаются вашему Controller, когда его View, то есть **self.view**, топовый view Controller, эта переменная **var view**. Когда этому **view**, непосредственно ПЕРЕД и непосредственно ПОСЛЕ посыпается метод **layoutSubviews**.

View Controller Lifecycle

Geometry

You get notified when your top-level **view**'s bounds change (or otherwise needs a re-layout).

In other words, when it receives **layoutSubviews**, you get to find out (before and after).

```
override func viewWillLayoutSubviews()  
override func viewDidLayoutSubviews()
```

Usually you don't need to do anything here because of Autolayout.

But if you do have geometry-related setup to do, this is the place to do it (not in **viewDidLoad!**).

These can be called often (just as **layoutSubviews()** in **UIView** can be called often).

Be prepared for that.

Don't do anything here that can't be properly (and efficiently) done repeatedly.

It doesn't always mean your **view**'s bounds actually changed.

Жизненный цикл ViewController

Геометрия

Вас уведомляют, если границы вашего топового **view** изменились (или нуждаются новом расположении).

Другими словами, если вы получаете **layoutSubviews**, то вы узнаете об этом (ДО или ПОСЛЕ).

```
override func viewWillLayoutSubviews ()
```

```
override func viewDidLayoutSubviews ()
```

Обычно нет необходимости что-то делать в этих методах из-за Autolayout.

Но если вы вынуждены делать некоторые “геометрически зависимые” установки, то это самое подходящее место для этого (а НЕ **viewDidLoad**)

Эти методы вызываются часто (также часто, как **layoutSubviews()** в **UIView**).

Будьте готовы к этому.

Не делайте здесь ничего, что не может быть сделано правильно (и эффективно) при повторном использовании.

Почему этому топовому **view** посыпается сообщение **layoutSubviews**?

По тем же самым причинам, что и другим **views**. Может быть, появились новые **subviews** или ушли какие-то уже находящиеся там **subviews** из вашего **view**. Это очень распространенная причина.

Или изменились границы **bounds** вашего **view**. Именно поэтому эти два метода являются хорошим местом для размещения кода при изменении “геометрии”. Потому что это всегда происходит, когда меняются границы **bounds** вашего топового **view**.

Обычно вы не реализуете эти методы.

Почему?

Потому что вы используете механизм **Autolayout**. Со всем его **CTRL**-перетягиванием, “пришпиливанием” **UI** элементов к краям, удержанием их в центре, фиксированием “коэффициента пропорциональности” (**aspect ratio**), со всем, что относится к использованию механизма **Autolayout**. И таким образом, вы автоматически вовлекаетесь в методы **layoutSubviews** вашего топового **view**. В этом случае нет необходимости что-то делать в вашем **Controller**. Но если вам действительно нужно делать что-то связанное с “геометрией” в вашем **Controller**, то, определенно, это то самое место для этого.

Нужно заметить, что эти два метода могут вызываться очень часто. Порой вы будете крайне удивлены: “Как же так? Ведь ничего не изменилось? Границы **bounds** не изменились, нет никаких изменений с **subviews**. Тогда почему эти методы вызываются?”

Системе разрешается размещать или проверять расположение **subviews** в любое время и в любом **view**. Может быть, она хочет выполнить анимацию типа имея расположение **subviews** в начале этой анимации и в конце, она хочет сделать “переворот” между ними. Границы **bounds** не изменились, но она хочет убедиться, где располагаются **subviews** или хочет переместить их в конечную точку анимации и т.д. По существу неважно почему, но системе разрешается вызывать эти методы в любое время, которое она сочтет приемлемым. Так как система может вызывать методы **layoutSubviews** в любое время, то два метода, представленные на слайде - **viewWillLayoutSubviews()** и **viewDidLayoutSubviews()** также могут вызываться в любое время. Вы должны это понимать, если хотите как-то реагировать на изменение границ **bounds**, вы должны использовать эти методы надлежащим образом и обеспечить их эффективную работу, даже если они непрестанно вызываются. Они могут вызываться два раза подряд при тех же самых границах **bounds**, например. Конечно, вы делаете что-то очень сложное и затратное, то вы, определенно, не захотите, чтобы это вызывалось два раза подряд.

Итак. Это “геометрия”.

Есть специальный случай, когда “геометрия” меняется и это автовращение (**autorotation**). Это происходит, когда ваше устройство переходит из ландшафтного режима в портретный и наоборот.

View Controller Lifecycle

⌚ Autorotation

When your device rotates, there's a lot going on.

Of course your `view`'s bounds change (and thus you'll get `viewWill/DidLayoutSubviews`).

But the resultant changes are also automatically animated.

You get to find out about that and even participate in the animation if you want ...

```
override func viewWillTransition(  
    to size: CGSize,  
    with coordinator: UIViewControllerTransitionCoordinator  
)
```

You join in using the coordinator's `animate(alongsideTransition:)` methods.

We don't have time to talk about how to do this, unfortunately!

Check the documentation 😕 .

Жизненный цикл View Controller

• Autorotation (автоматическое вращение устройства)

Когда устройство вращается, то многое что происходит.

Конечно, границы вашего `view` меняются (и таким образом, вы получаете `viewWill / DidLayoutSubviews`).

Результирующие изменения также автоматически анимируются.

Вас информируют об этом, и вы даже можете принять участие в анимации, если хотите...

```
override func viewWillTransition(  
    to size: CGSize,  
    with coordinator: UIViewControllerTransitionCoordinator  
)
```

Вы можете присоединиться к анимации, используя методами `coordinator animate (alongsideTransition:)`.

К сожалению, у меня нет времени говорить об этом!

Посмотрите документацию.

Когда это происходит, то, конечно, вы получаете сообщения `layoutSubviews`, потому что границы `bounds` вашего `view` меняются, и, соответственно, будут вызваны методы `viewWill / DidLayoutSubviews`. Но, кроме этого вы в придачу получаете совершенно бесплатно эту анимацию. iOS автоматически анимирует перемещение всех `subviews` из портретного `layoutSubviews` в ландшафтный `layoutSubviews`. Это делается для вас, и это замечательно, потому что экономит ваши усилия. Но если вы по какой-то причине хотите принять участие в этой анимации, то это можно сделать здесь.

Почему это может понадобиться?

Я не знаю, видели ли вы приложение **Calculator** на iPhone?

Если вы в портретном режиме, то кнопки в **Calculator** реально большие и их не так много для таких

операций, как умножение “**x**”, сложение “**+**”, вычитание “**-**”, равенство “**=**”. Но если вы переключаетесь в ландшафтный режим, то кнопки становятся маленькими и там уже присутствует полный набор операций: корень квадратный “**√**”, логарифм “**log₁₀**” и прочие.

Когда происходит вращение устройства, приложение **Calculator** вовлечено в процесс анимации, потому что оно хочет анимировать появление на экране этих дополнительных кнопок. Также изменяется само местоположение на экране от больших кнопок в середине до этих же кнопок, которые теперь будут находиться сбоку и появятся новые кнопки. Именно здесь вы можете вмешаться и принять участие в анимационном процессе.

Как вы будете использовать метод **viewWillTransition(to size:,with coordinator:)** ?

Координатор **coordinator**, который передается в этом методе, является анимационным координатором. У него есть метод с именем **animate(alongsideTransition:)**. Вы можете передать этому методу замыкание и выполнить с его помощью свою собственную анимацию. И эта анимация будет выполняться параллельно с анимацией, которую выполняет система при повороте устройства. Больше я об этом не собираюсь говорить. Вы просто должны знать, что это существует. В 90% случаев у вас не будет в этом никакой необходимости. В вашем текущем домашнем Задании № 4, если вы продвинулись так далеко, если - нет, то вы тем более должны заметить, что там есть анимация типа “сдачи карт”. Что произойдет, если вы начали “сдавать карту”, она летит через весь экран, а затем вы вращаете ваше устройство. Вы можете попробовать это с некоторой анимацией. Я специально указал эту ситуацию в вашем домашнем Задании № 4, чтобы вы не беспокоились об этом. На этой неделе вы не должны об этом беспокоиться, но вы можете спросить, а что нужно делать в этой ситуации?

----- 15-ая минута лекции -----

Вы должны понимать, что это может произойти и с этим нужно что-то делать, потому что, если вы “бросаете” карту через весь экран и из-за вращения начинает происходить эта анимация тех же самых свойств того же самого **view**, то, возможно, карта прилетит в неправильное место, потому что система переместит ее на новое место. Возможно, она начнет анимировать с текущего состояния и может закончить анимацию **views** абсолютно не в правильном месте. Вам нужно уметь настраивать свои анимации таким образом, что даже если она “подхватывается” чем-то еще, все равно она бы заканчивалась в правильном месте. Так что это может потребовать небольшой настройки, регулировки процесса анимации, как я это называю.

Но в Задании № 4 это не нужно делать, так как такая регулировка относится к продвинутым возможностям анимации.

И наконец, последний и наименее важный метод **didReceiveMemoryWarning**, связанный с нехваткой памяти. Интерес к этому методу вы будете проявлять только в том случае, если ваше приложение имеет дело с такими емкими по памяти объектами, как видео, изображения, огромные звуковые файлы, например, большие звуковые файлы с высоким разрешением. Вы знаете, что вполне возможно, что на **iOS** устройстве есть множество приложений, которые манипулируют с большими фотографиями или с чем-то подобным. И это теоретически может привести к нехватке памяти.

Большинство новых современных устройств имеют такую большую память, что такого не может

произойти никогда. Но если в вашем приложении есть “утечка памяти”, когда происходит потеря памяти больших изображений, то можно выйти на нехватку памяти. В любом случае, если это произойдет, то вам будет выслано сообщение **didReceiveMemoryWarning**:

View Controller Lifecycle

• Low Memory

It is rare, but occasionally your device will run low on memory.

This usually means a buildup of very large videos, images or maybe sounds.

If your app keeps strong pointers to these things in your heap, you might be able to help!

When a low-memory situation occurs, iOS will call this method in your Controller ...

```
override func didReceiveMemoryWarning() {  
    super.didReceiveMemoryWarning()  
    // stop pointing to any large-memory things (i.e. let them go from my heap)  
    //    that I am not currently using (e.g. displaying on screen or processing somehow)  
    //    and that I can recreate as needed (by refetching from network, for example)  
}
```

If your application persists in using an unfair amount of memory, you can get killed by iOS.

Жизненный цикл ViewController

• Недостаток “памяти”

Это происходит редко, но время от времени ваше устройство начинает получать сообщения о нехватке памяти.

Это обычно означает создание очень больших видео, изображений или музыки.

Если ваше приложение держит strong указатели на такие вещи в вашей “куче”, то вам можно помочь!

Если возникнет ситуация с нехваткой памяти, то iOS вызовет этот метод в вашем Controller...

```
override func didReceiveMemoryWarning () {  
    super.didReceiveMemoryWarning()  
    // прекратите указывать на любые объекты, занимающие большую память,  
    // (то есть позвольте им покинуть “кучу”) которые я в данный момент не использую  
    // (например, не показываю на экране и не обрабатываю)  
    // и которые я могу создать заново, если это необходимо (например, путем  
    // повторной выборки из интернета)  
}
```

Если ваше приложение упорствует в использовании недобросовестно большого количества памяти, то оно может быть “убито” iOS

Этот метод просит ваш **View Controller**: “Пожалуйста, убери что-нибудь из “кучи” (**heap**), что ты можешь достаточно легко создать это заново позже или когда это будет необходимо.” Это своего рода “очистка” памяти. Если ваше приложение имеет “утечку памяти” в “куче” и вы потребляете все больше и больше памяти, то **iOS** может послать вам это сообщение, но если у вас плохой код и вы

не знаете, как “почистить” память, то “утечка памяти” будет продолжаться, и тогда **iOS** может вас “убить”. Если вы продолжаете вести себя как “свинья” по отношению к памяти и “утечки памяти” продолжаются, то вы реально можете быть “убиты”.

Очень опасно быть плохим программистом, ваше приложение может быть убито, хотя этого практически никогда не происходит.

И последняя вещь возвращает нас в начало “жизненного цикла” **View Controller**, когда он пробуждается, приходя со **storyboard**. Строго говоря, метод **awakeFromNib** не является частью “жизненного цикла” **View Controller**, но я поместил его в этот раздел, потому что любой объект, который приходит со **storyboard**, все ваши **UIView**, все ваши **ViewController** получают сообщение **awakeFromNib**. Сообщение **awakeFromNib** посыпается **ViewController** очень рано, сразу после инициализации, но перед подготовкой, перед установкой **Outlets**. Я стараюсь подальше держаться от попытки использования этого метода до тех пор, пока мне действительно не будет нужно устанавливать что-то очень рано.

View Controller Lifecycle

• Waking up from an storyboard

This method is sent to all objects that come out of a storyboard (including your Controller) ...

```
override func awakeFromNib() {  
    super.awakeFromNib()  
    // can initialize stuff here, but it's VERY early  
    // it happens way before outlets are set and before you're prepared as part of a segue  
}
```

This is pretty much a place of last resort.

Use the other View Controller Lifecycle methods first if at all possible.

It's primarily for situations where code has to be executed VERY EARLY in the lifecycle.

Жизненный цикл ViewController

• “Пробуждение” со storyboard

Этот метод посыпается всем объектам, которые “приходят” из storyboard (включая ваш Controller)...

```
override func awakeFromNib() {  
    super.awakeFromNib()  
  
    // можно что-то инициализировать здесь, но это ОЧЕНЬ РАНО,  
    // это происходит до того, как все outlets установлены и перед тем, как вы будете  
    // подготовлены как часть “переезда”  
}
```

Это в большинстве случаев это место используется в крайних ситуациях.

Используйте сначала другие методы “жизненного цикла” View Controller, если это возможно.

В основном это для ситуаций, когда код должен выполняться ОЧЕНЬ РАНО в “жизненном

цикле”.

Мы использовали это в прошлом демонстрационном примере, потому что я хотел, чтобы мой **Master** в **Split View** был делегатом **Split View** как можно раньше, ПЕРЕД тем, как **Split View** стартовал бы наложение (**collapsing**) одних вещей поверх других. Я хотел сделать себя в методе **awakeFromNib** делегатом **Split View**. Но в нормальной ситуации мы не размещаем код в методе **awakeFromNib**. Я бы советовал сначала попытаться разместить его где-нибудь еще, в других методах, и только поняв, что вам совершенно необходимо сделать что-то ОЧЕНЬ РАНО, обращаться к методу **awakeFromNib**.

Конечно, это работает только для **MVCs**, которые приходят со **storyboard**, и мы именно так и поступаем практически всегда. Единственный **MVC**, который мы получаем из кода - это **Camera MVC**.

Вот краткое изложение основных этапов “жизненного цикла” **View Controller**.

View Controller Lifecycle

- Summary
 - Instantiated (from storyboard usually)
 - awakeFromNib (only if instantiated from a storyboard)
 - segue preparation happens
 - outlets get set
 - viewDidLoad
 - These pairs will be called each time your Controller's view goes on/off screen ...
 - viewWillAppear and viewDidAppear
 - viewWillDisappear and viewDidDisappear
 - These “geometry changed” methods might be called at any time after viewDidLoad ...
 - viewWillLayoutSubviews and viewDidLayoutSubviews
 - At any time, if memory gets low, you might get ...
 - didReceiveMemoryWarning

Жизненный цикл ViewController

- Итоги

Получаем экземпляр класса (обычно со storyboard)

awakeFromNib (только если получаем экземпляр класса со storyboard)

происходит подготовка “переезда” (**segue**)

outlets устанавливаются

viewDidLoad

Эти пары будут вызываться каждый раз, когда View вашего Controller будет “приходить” / “уходить” на / с экрана...

viewWillAppear и **viewDidAppear**

viewWillDisappear и **viewDidDisappear**

Эти “меняющие геометрию” методы вызываются в любое время после `viewDidLoad...`

`viewWillLayoutSubviews` и `viewDidLayoutSubviews`

В любое время, если память достигнет нижнего уровня, вы можете получить ...

`didReceiveMemoryWarning`

Вы получаете экземпляр **View Controller**, обычно со **storyboard**, но иногда вы можете запросить **View Controller** у **iOS**.

Затем вы получаете `awakeFromNib`, если вы приходите со **storyboard**.

Происходит подготовка “переезда” (**segue**) в случае, если другие **MVCs** готовят (**prepares**) вас, заметьте, что **Outlets** еще не установлены.

Затем устанавливаются **Outlets**, они “подвязываются” **iOS**.

Затем вызывается `viewDidLoad`.

Затем вы появляйтесь и исчезаете, появляетесь и исчезаете с экрана с сообщениями `viewWillAppear` и `viewDidAppear` при появлении на экране и с сообщениями `viewWillDisappear` и `viewDidDisappear` при исчезновении с экрана.

Между прочим, в любое время, когда меняется “геометрия”, вам могут быть посланы сообщения `viewWillLayoutSubviews` и `viewDidLayoutSubviews`. Возможно, между этими двумя вызовами может работать механизм **Autolayout**.

И в любое время, как только вы начнете использовать много памяти или даже все ваше устройство начинает использовать слишком много памяти, то вы получаете сообщение `didReceiveMemoryWarning`, в котором вы, надеюсь, “очистите” что-то, но часто вы не можете этого сделать из-за того, что в вашем приложении нет ничего реально большого.

Это “жизненный цикл” **View Controller**.

У меня есть небольшой демонстрационный пример, я не могу потратить слишком много времени на эту демонстрацию, потому что нам предстоит более подробный демонстрационный пример со **ScrollView**, но я покажу маленький кусочек кода, который вы можете “бросить” в ваше приложение.

Это **subclass** класса **UIViewController**, и вы можете просто изменить все ваши **View Controllers** так, чтобы они наследовали не от **UIViewController**, а от этого **subclass**, в котором, по существу, просто размещены предложения `print` для вывода на консоль всего, что связано с “жизненным циклом” **View Controller**.



Демонстрационный пример

- “Жизненный цикл” **View Controller**

Давайте поместим несколько предложений `print` в методы “жизненного цикла” приложения **Concentration**

Вы можете посмотреть на консоль и увидеть, что происходит в “жизненном цикле” **View Controller**. Я хочу с помощью этого **subclass** класса **UIViewController** показать вам сегодня пару интересных вещей. Мы будем работать с приложением **Concentration**.

----- 20-ая минута лекции -----

Между прочим, как раз здесь я сделал в прошлый раз ошибку в **awakeFromNib()**, не добавив **super.awakeFromNib()**. В методе **awakeFromNib()** необходимо также вызывать **super**, хотя, строго говоря, этот метод не является методом “жизненного цикла”, вы должны дать **super** шанс выполнить свою работу:

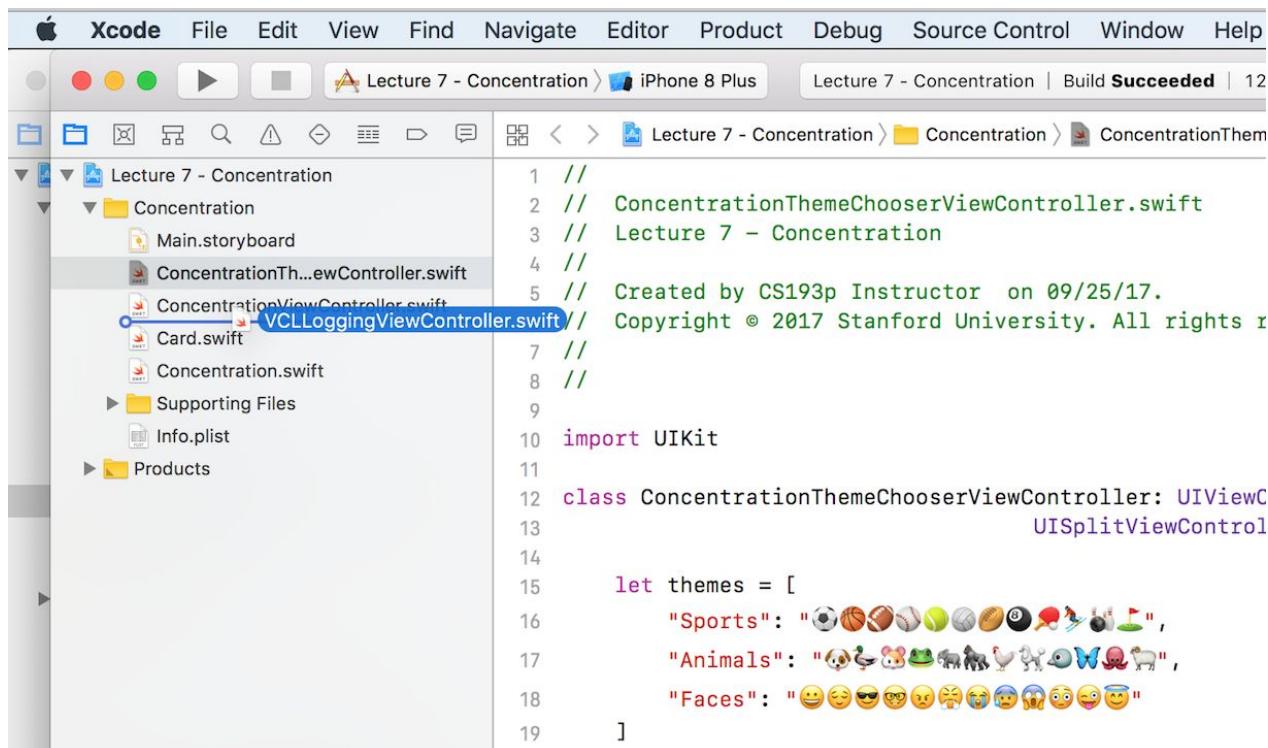
```
import UIKit

class ConcentrationThemeChooserViewController: UIViewController,
    UISplitViewControllerDelegate {

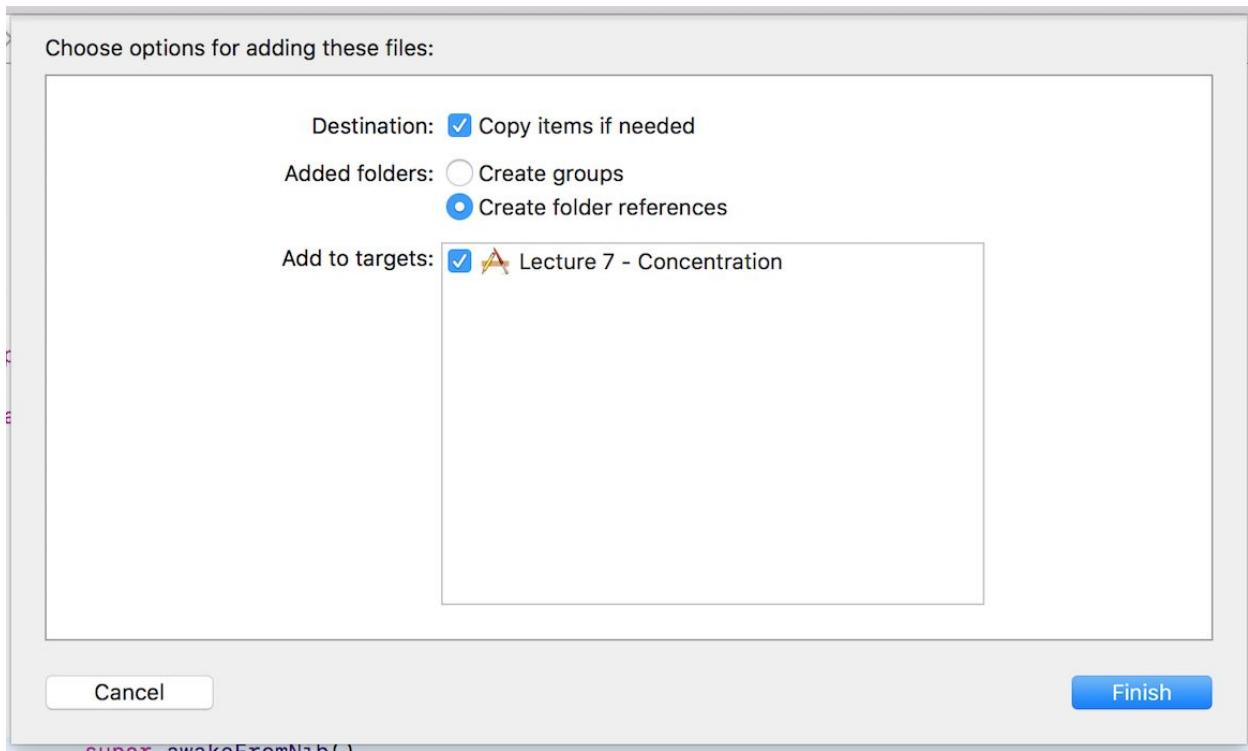
    let themes = [
        "Sports": "⚽🏀🏈⚾🎾🎱🏓🎱🎱🎱🎱🎱",
        "Animals": "🐶🦆🐹🐸🐊🐘🦒🐘🐘🐘🐘",
        "Faces": "😊😎☀️😍😘😭😢😱😱😱😱😱😱",
    ]

    override func awakeFromNib() {
        super.awakeFromNib()
        splitViewController?.delegate = self
    }
}
```

Где код, который я писал для того, чтобы это приложение работало нужным нам образом? Вот он, он может быть где угодно. Я перетягиваю его в мой проект **Concentration**:



Я копирую его:



Он выглядит в точности так, как вы о нем думали. Он является **subclass** класса **UIViewController**:

```
1 //  
2 //  VCLLoggingViewController.swift  
3 //  
4 //  Created by CS193p Instructor.  
5 //  Copyright © 2015–17 Stanford University. All ri  
6 //  
7  
8 import UIKit  
9  
10 class VCLLoggingViewController : UIViewController  
11 {  
12     private struct LogGlobals {  
13         var prefix = ""  
14         var instanceCounts = [String:Int]()  
15         var lastLogTime = Date()  
16         var indentationInterval: TimeInterval = 1  
17         var indentationString = "__"  
18     }  
}
```

И он распечатывает все **viewDidLoad**, **viewWillAppear**, **viewDidAppear**, **viewWillDisappear**, **viewDidDisappear**, **viewWillLayoutSubviews** и **viewDidLayoutSubviews**, даже **viewWillTransition** и **didReceiveMemoryWarning**. Он просто распечатывает, когда происходят эти события:

```

deinit {
    logVCL("left the heap")
}

override func awakeFromNib() {
    logVCL("awakeFromNib()")
}

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

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    logVCL("viewDidAppear(animated = \(animated))")
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    logVCL("viewDidDisappear(animated = \(animated))")
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    logVCL("viewWillDisappear(animated = \(animated))")
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    logVCL("viewDidDisappear(animated = \(animated))")
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    logVCL("didReceiveMemoryWarning()")
}

override func viewWillLayoutSubviews() {
    super.viewWillLayoutSubviews()
    logVCL("viewWillLayoutSubviews() bounds.size = \(view.bounds.size)")
}

override func viewDidLayoutSubviews() {
    super.viewDidLayoutSubviews()
    logVCL("viewDidLayoutSubviews() bounds.size = \(view.bounds.size)")
}

override func viewWillTransition(to size: CGSize, with coordinator: UIViewControllerTransitionCoordinator) {

```

Также в этом **subclass** есть очень крутая переменная **var** с именем **vclLoggingName**:

```

var vclLoggingName: String {
    return String(describing: type(of: self))
}

```

При выводе на консоль используется эта строка для идентификации **View Controller**. Или, если вы не выполняете **subclass**, то используется просто имя структуры, класса или того, в чем вы находитесь: **String (describing: type(of: self))**. Вот как мы это делаем.

В приложении **Concentration** я сделаю оба своих **View Controller subclasses** класса **UIViewController**. Один **View Controller** - это выбор темы **theme**, а другой - это игра **Concentration**. Начнем с **View Controller** для игры **Concentration**: вместо того, чтобы наследовать от **UIViewController**, он будет наследовать от **VCLLoggingViewController**, он будет наследовать способность печатать все методы “жизненного цикла” **View Controller**. Кроме того, я переопределяю (**override**) переменную **var** с именем **vclLoggingName**, чтобы она вернула “**Game**” и было бы более понятно, в каком **View Controller** мы находимся:

```
class ConcentrationViewController: VCLLoggingViewController {  
  
    override var vclLoggingName: String {  
        return "Game"  
    }  
}
```

Наш **View Controller** будет называться **Game** вместо **ConcentrationViewController**, что очень длинно.

Точно так же мы поступим по отношению к **ConcentrationThemeChooserViewController**.

Он будет наследовать от **VCLLoggingViewController** и мы изменим его имя на **ThemeChooser**:

```
class ConcentrationThemeChooserViewController: VCLLoggingViewController,  
    UISplitViewControllerDelegate {  
  
    override var vclLoggingName: String {  
        return "ThemeChooser"  
    }  
}
```

Это все, что нам необходимо сделать, даже нет необходимости в том, чтобы переопределять переменную **var** с именем **vclLoggingName**, если вы хотите оставить длинные имена.

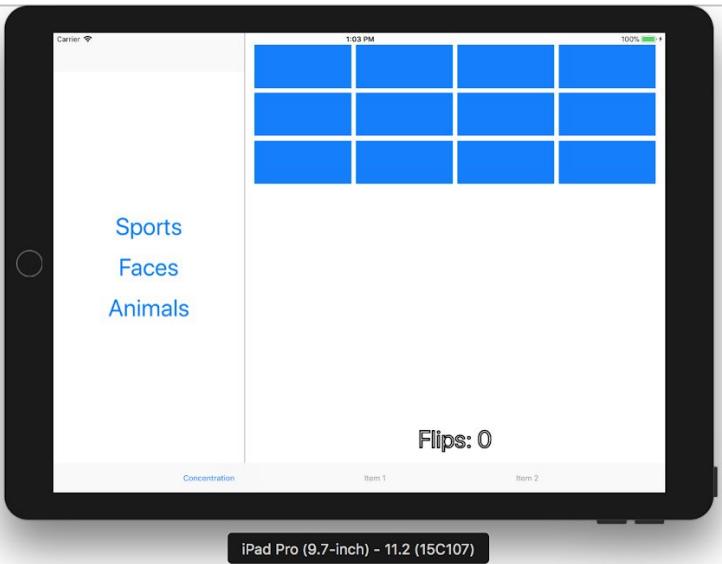
Давайте запустим это приложение на **iPad**, и посмотрим, что происходит. Давайте выдвинем консоль, чтобы было лучше видно, что происходит.

Это приложение игры **Concentration**, которое показано в ландшафтном режиме:

```

ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewWillAppear(animated = false)
ThemeChooser(1) viewWillAppear(animated = false)
Game(1) viewWillLayoutSubviews() bounds.size = (703.5, 768.0)
Game(1) viewDidLayoutSubviews() bounds.size = (703.5, 768.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidAppear(animated = false)
Game(1) viewDidAppear(animated = false)

```



Вы видите, что для обоих **View Controller** вызывается **init (coder:)**, что происходит, когда экземпляр класса создается со **storyboard**. Таким образом, оба **View Controller** инициализированы.

Затем оба вызывают **awakeFromNib ()**, видите?

Затем у обоих устанавливаются **Outlets** и все остальное.

Затем у обоих **View Controller** вызываются **viewDidLoad**.

Затем обоим сообщается, что они сейчас появятся на экране **viewWillAppear**.

Затем они получают сообщения о расположении своих **subViews**. Посмотрите на эти **viewWillLayoutSubviews** и **viewDidLayoutSubviews**. **ThemeChooser** получает их дважды, хотя его границы **bounds** не изменились.

Затем **ThemeChooser** получает **viewDidAppear**, и в конце **Game** получает **viewDidAppear**, и они оба появляются на экране.

Что произойдет, если поверну свое устройство в портретный режим?

Когда я ухожу в портретный режим, **ThemeChooser**, **Master**, получает **viewWillDisappear**, потому что он исчезает и больше не будет показываться на экране. Он также получает этот полный метод **viewWillTransition(to: (320.0, 1024.0), with: coordinator)**, потому что он повернулся и у нас есть возможность принять участие в анимации этого переворота, если мы этого хотим.

Конечно, **Game** получает новый размер. Видите? Это **768 x 1024**, так что **Game** посылаются сообщения **...LayoutSubviews**.

Затем **ThemeChooser** также по некоторым причинам получает свой новый размер **320 x 1024** и, соответственно, сообщения **...LayoutSubviews**.

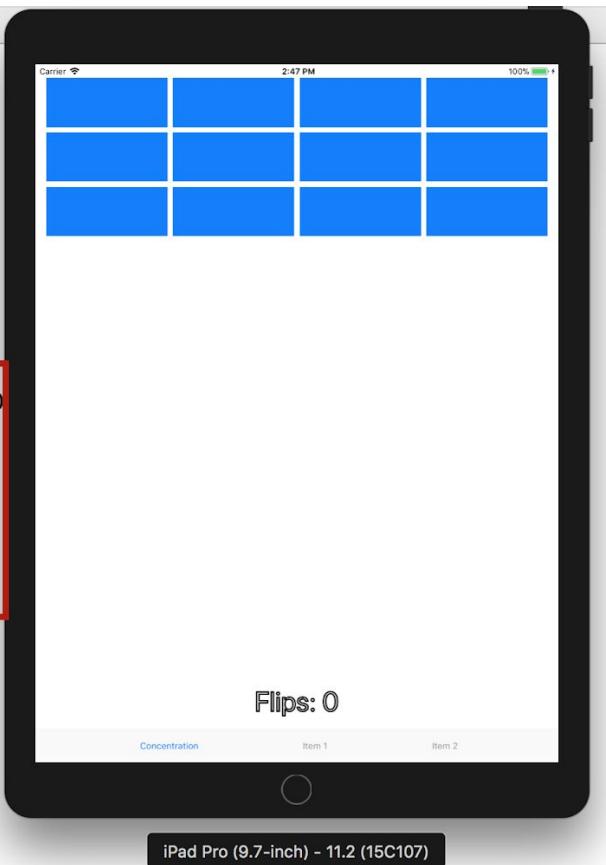
А затем он исчезает и получает сообщение **viewDidDisappear**.

```

ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewWillAppear(animated = false)
ThemeChooser(1) viewWillAppear(animated = false)
Game(1) viewWillLayoutSubviews() bounds.size = (703.5, 768.0)
Game(1) viewDidLayoutSubviews() bounds.size = (703.5, 768.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidAppear(animated = false)
Game(1) viewDidAppear(animated = false)

--ThemeChooser(1) viewWillDisappear(animated = false)
--ThemeChooser(1) viewWillTransition(to: (320.0, 1024.0), with: coordinator)
--Game(1) viewWillTransition(to: (768.0, 1024.0), with: coordinator)
--Game(1) viewWillLayoutSubviews() bounds.size = (768.0, 1024.0)
--Game(1) viewDidLayoutSubviews() bounds.size = (768.0, 1024.0)
--ThemeChooser(1) begin animate(alongsideTransition:completion:)
--Game(1) begin animate(alongsideTransition:completion:)
--ThemeChooser(1) end animate(alongsideTransition:completion:)
--Game(1) end animate(alongsideTransition:completion:)
--ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidDisappear(animated = false)

```



iPad Pro (9.7-inch) - 11.2 (15C107)

Смотрите, что произойдет, если я вытяну мой Master.

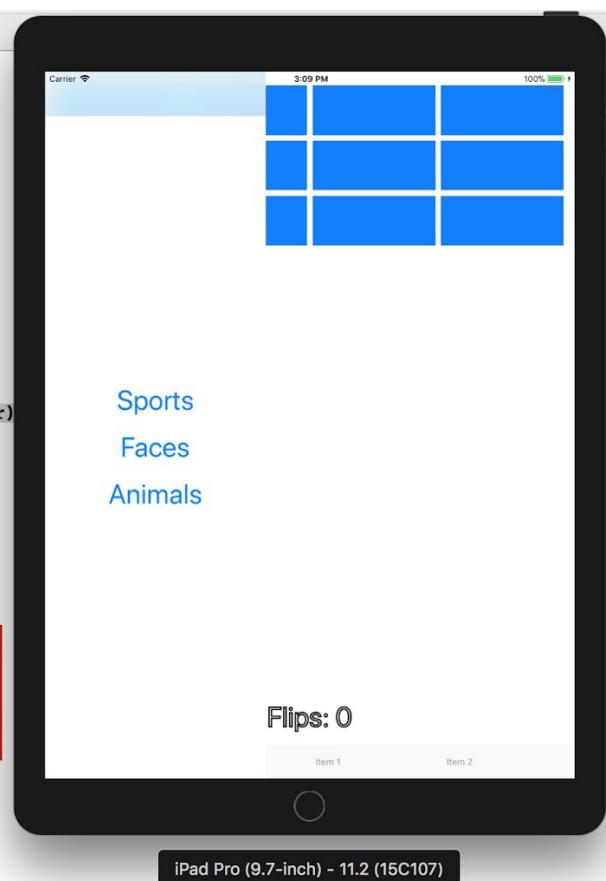
```

ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewWillAppear(animated = false)
ThemeChooser(1) viewWillAppear(animated = false)
Game(1) viewWillLayoutSubviews() bounds.size = (703.5, 768.0)
Game(1) viewDidLayoutSubviews() bounds.size = (703.5, 768.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 768.0)
ThemeChooser(1) viewDidAppear(animated = false)
Game(1) viewDidAppear(animated = false)

--ThemeChooser(1) viewWillDisappear(animated = false)
--ThemeChooser(1) viewWillTransition(to: (320.0, 1024.0), with: coordinator)
--Game(1) viewWillTransition(to: (768.0, 1024.0), with: coordinator)
--Game(1) viewWillLayoutSubviews() bounds.size = (768.0, 1024.0)
--Game(1) viewDidLayoutSubviews() bounds.size = (768.0, 1024.0)
--ThemeChooser(1) begin animate(alongsideTransition:completion:)
--Game(1) begin animate(alongsideTransition:completion:)
--ThemeChooser(1) end animate(alongsideTransition:completion:)
--Game(1) end animate(alongsideTransition:completion:)
--ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidDisappear(animated = false)

--ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewWillAppear(animated = false)
--ThemeChooser(1) viewDidAppear(animated = false)
--ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (320.0, 1024.0)
--ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (320.0, 1024.0)

```



iPad Pro (9.7-inch) - 11.2 (15C107)

ThemeChooser получает сообщения [...LayoutSubviews](#) и пару: [viewWillAppear](#) и [viewDidAppear](#).

Видите, как только что-то здесь происходит, вам говорят в точности, что произошло.

Сейчас я покажу вам одну очень интересную вещь.

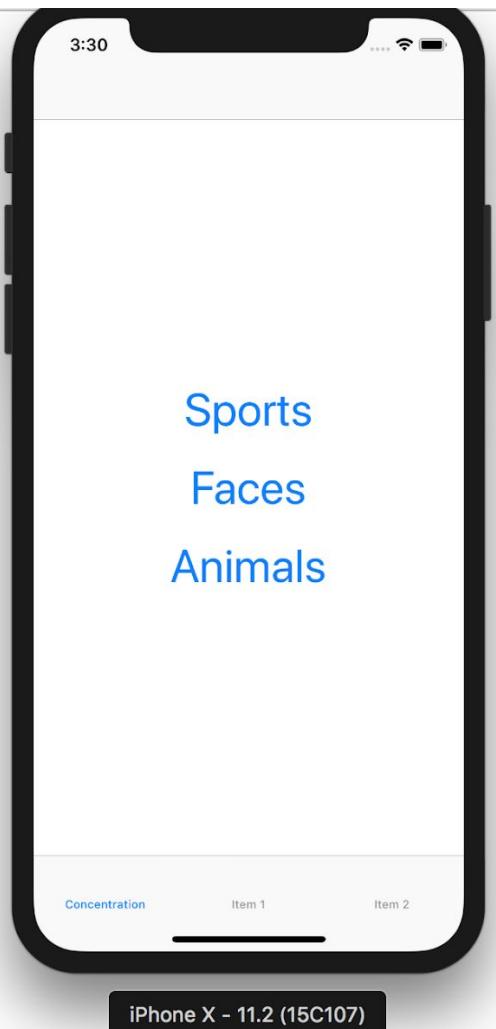
Идем в наш класс **ConcentrationThemePickerController**. О! Мы уже здесь!

Мы разместили очень много кода, чтобы предотвратить **Segue** (“переезд”). Но я сделаю так, что у нас всегда будет происходить **Segue** (“переезд”), для этого достаточно поставить комментарий на большую часть кода:

```
① @IBAction func chooseTheme(_ sender: Any) {
31 //     if let cvc = splitViewDetailConcentrationViewController {
32 //         if let themeName = (sender as? UIButton)?.currentTitle, let theme = themes[themeName] {
33 //             cvc.theme = theme
34 //         }
35 //     } else if let cvc = lastSeguedToConcentrationViewController {
36 //         if let themeName = (sender as? UIButton)?.currentTitle, let theme = themes[themeName] {
37 //             cvc.theme = theme
38 //         }
39 //         navigationController?.pushViewController(cvc, animated: true)
40 //     } else {
41 //         performSegue(withIdentifier: "Choose Theme", sender: sender as! UIButton)
42 //     }
43 }
```

Делая это, я создаю ситуацию, когда каждый раз при выборе темы **theme** для игры, я ВСЕГДА буду “ПЕРЕЕЗЖАТЬ” (**segues**). Я делаю так, потому что хочу вам показать, что происходит в “жизненном цикле” **View Controller**, когда вы “переезжаете”, потому это очень важно понимать. Я буду это делать на **iPhone**, просто так легче проследить, что происходит.

```
ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewDidAppear(animated = false)
Game(1) viewWillDisappear(animated = false)
Game(1) viewDidDisappear(animated = false)
ThemeChooser(1) viewDidAppear(animated = false)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidAppear(animated = false)
```



----- 25-я минута лекции -----

Это очень интересное начало, потому что вы видите, что мы создаем оба **View Controller**, мы

получаем `awakeFromNib()`, затем `viewDidLoad`.

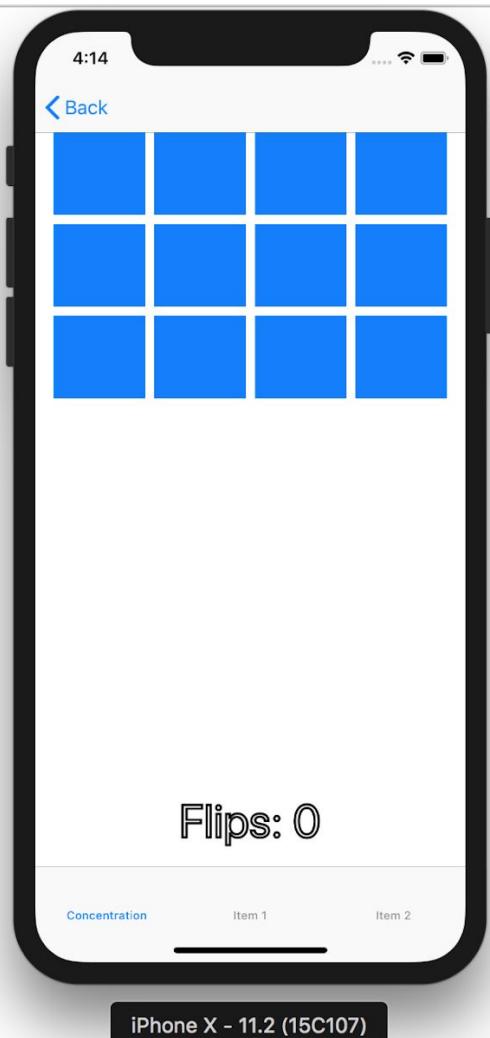
Затем **Game** получает `viewWillAppear`, но никогда не получит `viewDidAppear`, потому что мы этого добились, когда, помните, останавливали коллапс `secondaryViewController` так, чтобы он никогда не появлялся. Так что **Game** получает `viewWillAppear`, но никогда не получит `viewDidAppear`, потому что он действительно не появляется. И это еще одна причина ждать иногда до метода `viewDidAppear`, чтобы запустить что-то затратное.

После того, как **ThemeChooser** дважды получает пару сообщений `...LayoutSubviews`, он появляется на экране и получает `viewDidAppear`.

Теперь, если вы кликните на одной из тем **theme**, то создастся второй новый **MVC Game**, видите?

```
ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewWillAppear(animated = false)
Game(1) viewWillDisappear(animated = false)
Game(1) viewDidDisappear(animated = false)
ThemeChooser(1) viewWillAppear(animated = false)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidAppear(animated = false)

-- Game(2) init(coder:) - created via InterfaceBuilder
-- Game(2) awakeFromNib()
-- Game(1) left the heap
-- Game(2) viewDidLoad()
-- ThemeChooser(1) viewWillDisappear(animated = true)
-- Game(2) viewWillAppear(animated = true)
-- Game(2) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
-- Game(2) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
-- ThemeChooser(1) viewDidDisappear(animated = true)
-- Game(2) viewDidAppear(animated = true)
```



Это новый **MVC Game (2)**, **2** означает, что это новый экземпляр класса, и **2** показывает номер этого экземпляра.

Давайте посмотрим, что происходит с первым экземпляром **Game (1)**, который так и не появился?

Видите? Он покинул (**left**) “кучу” (**heap**). Именно это и происходит, когда вы “переезжаете” (**segue**). Вы получаете совершенно новый экземпляр **MVC Game (2)**, а старый **Game (1)** покидает “кучу” (**heap**).

Новый экземпляр **Game (2)** получает `viewDidLoad`, `viewWillAppear`, `viewDidAppear`.

Если я вернусь назад, кликнув на кнопке “Back”, **Game (2)** получит `viewWillDisappear` и `viewDidDisappear`, потому что **Game (2)** исчезает, а **ThemeChooser** появляется.

Если я опять “перееду” (**segue**), то я получу 3-ий экземпляр **MVC Game (3)**, а второй экземпляр

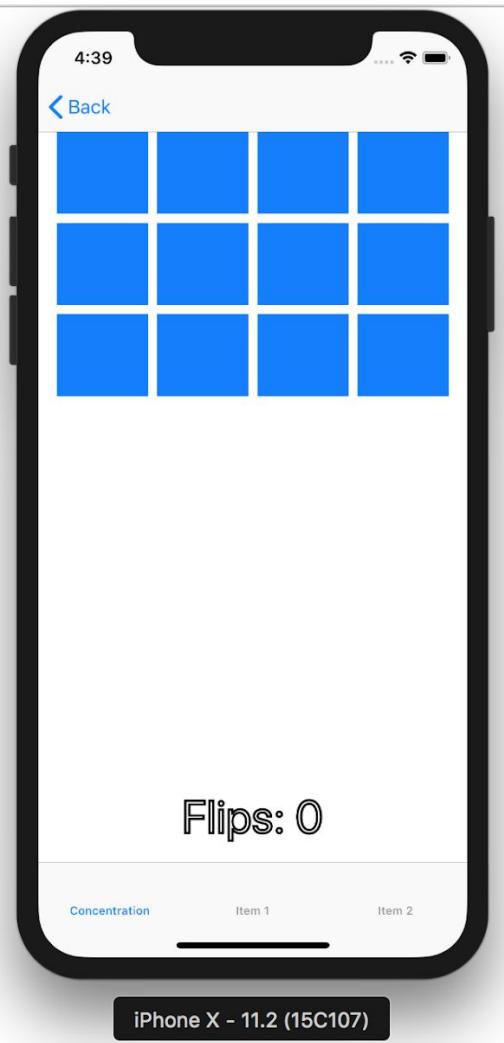
Game (2) покидает “кучу” (**heap**).

```
ThemeChooser(1) init(coder:) - created via InterfaceBuilder
Game(1) init(coder:) - created via InterfaceBuilder
Game(1) awakeFromNib()
ThemeChooser(1) awakeFromNib()
Game(1) viewDidLoad()
ThemeChooser(1) viewDidLoad()
Game(1) viewWillAppear(animated = false)
Game(1) viewWillDisappear(animated = false)
Game(1) viewWillDisappear(animated = false)
ThemeChooser(1) viewWillAppear(animated = false)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewDidAppear(animated = false)

__Game(2) init(coder:) - created via InterfaceBuilder
__Game(2) awakeFromNib()
__Game(1) left the heap
__Game(2) viewDidLoad()
__ThemeChooser(1) viewWillDisappear(animated = true)
__Game(2) viewWillAppear(animated = true)
__Game(2) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
__Game(2) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
__ThemeChooser(1) viewWillDisappear(animated = true)
__Game(2) viewDidAppear(animated = true)

____Game(2) viewWillDisappear(animated = true)
____ThemeChooser(1) viewWillAppear(animated = true)
____Game(2) viewWillDisappear(animated = true)
____ThemeChooser(1) viewDidAppear(animated = true)

Game(3) init(coder:) - created via InterfaceBuilder
Game(3) awakeFromNib()
Game(2) left the heap
Game(3) viewDidLoad()
ThemeChooser(1) viewWillDisappear(animated = true)
Game(3) viewWillAppear(animated = true)
Game(3) viewWillLayoutSubviews() bounds.size = (375.0, 812.0)
Game(3) viewDidLayoutSubviews() bounds.size = (375.0, 812.0)
ThemeChooser(1) viewWillDisappear(animated = true)
Game(3) viewDidAppear(animated = true)
```



Именно об этом я и хотел вам рассказать. Когда вы выполняете “переезд” (**segue**), всегда создается новый экземпляр, а старый будет выброшен из **Navigation Controller** и покинет “кучу” (**heap**). Все поняли?

Это все, что я хотел вам показать. Этот код вы можете использовать для выяснения того, что происходит в вашем приложении, если все работает не так, как вы ожидали.

Давайте вернемся к нашим слайдам и продолжим говорить о **ScrollView**.

Вы знаете, что такое **ScrollView**, правда? У вас на экране очень маленькая область, и вы хотите через эту маленькую область смотреть на что-то очень “большое”, в этом случае вы используете **ScrollView**. Вы можете прокручивать это “большое” с помощью ваших пальцев, вы можете собирать (разъединять) пальцы вместе и тем самым уменьшать (увеличивать) масштаб этого большого”. Я хочу, чтобы вы убедились, что **ScrollView** - это очень мощный **UIView**. Он работает практически в любых обстоятельствах.

Это моя любимая демонстрация, и то, что я вам буду показывать, - это запись, потому что трудно поверить, но это **iPhone 1**, но даже начиная с **iPhone 1**, уже тогда, **ScrollView** прекрасно делал свою работу.

UIScrollView



CS193p
Fall 2017-18

Смотрите, что будет происходить, если я это запущу.

Вы видите, что у меня есть **ScrollView** с горизонтальной прокруткой, и я могу делать эту горизонтальную прокрутку:



А внутри горизонтальной прокрутки есть вертикальная. Это совершенно нормально иметь

ScrollView внутри **ScrollView**.

На рисунке выше мы прокручиваем по горизонтали по **views**, связанным с котировкой акций. Затем мы достигаем чего-то и можем прокручивать по вертикали вверх и вниз:



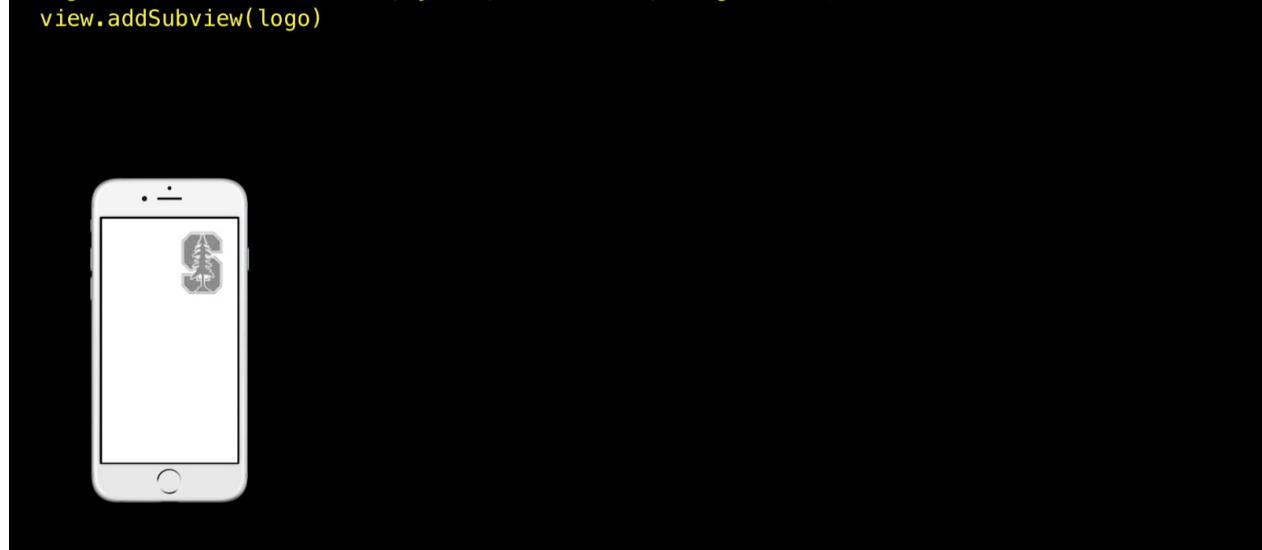
ScrollView знает, как “живь” внутри себя и знает, как “живь” в практически любой внешней среде, **ScrollView** – это реально потрясающий маленький класс.

Давайте поговорим о том, как **ScrollView** работает, его также очень легко использовать.

Вы уже знаете, как добавить **subviews** к обычному **UIView**.

Adding subviews to a normal UIView ...

```
logo.frame = CGRectMake(300, 50, 120, 180)
view.addSubview(logo)
```



У меня на слайде есть обычный **view** ТИПА **UIView**. Я просто создал **view logo** с логотипом Стэнфорда, установил его **frame**, добавил его как **subview** к топовому **view** и он появился на экране. Я уверен, что из Задания №3 вы прекрасно знакомы с тем, как это делается.

Насколько сильно это будет отличаться, если я буду это все делать не с обычным **UIView**, а со **ScrollView**? В случае со **ScrollView** громадное отличие состоит в том, что создается так называемый **contentSize**.

Определяя **contentSize**, мы задаем размер области, которую **scrollView** будет прокручивать. Область **contentSize** значительно больше размера маленького **scrollView**, через который мы

“рассматриваем” область **contentSize**. У области **contentSize** есть только размер **size**, начало координат **origin** у этой прямоугольной области ВСЕГДА **(0,0)**.

Adding subviews to a UIScrollView ...

```
scrollView.contentSize = CGSizeMake(width: 3000, height: 2000)
logo.frame = CGRectMake(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(logo)
```



CS193p
Fall 2017-18

Переменная **var contentSize**, которую вы видите на слайде, имеет решающее значение для работы **scrollView**. Это САМАЯ ВАЖНАЯ переменная **var**, может быть, в любом классе, повсюду. Эта переменная **contentSize** сообщает **scrollView**, насколько большим является пространство для его прокрутки.

И только ПОСЛЕ этого я добавляю **subview** как в случае с обычным **view: UIView**. Теперь я могу создать **frame** и добавить **subview** к **scrollView**, но вместо того, чтобы разместить его в “черной” области слева, оно размещается в области **contentSize**. Тот же самый **frame**, вы знаете, что для размещения **view** нужно задать **frame**. Но **frame** задается для размещения в **contentSize**.

Я могу разместить еще одну вещь в **scrollView**, может быть множество **subviews** в области **contentSize**, сколько угодно.

Adding subviews to a UIScrollView ...

```
scrollView.contentSize = CGSizeMake(width: 3000, height: 2000)
aerial.frame = CGRectMake(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```



CS193p
Fall 2017-18

Теперь у нас 2 **views**.

И что теперь делает **scrollView**? Он становится маленьким “окном” на контентную область **contentSize** ...

Adding subviews to a UIScrollView ...

```
scrollView.contentSize = CGSizeMake(width: 3000, height: 2000)  
aerial.frame = CGRectMake(x: 150, y: 200, width: 2500, height: 1600)  
scrollView.addSubview(aerial)
```

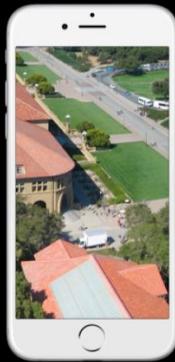


... позволяя прокручивать ее с помощью выполняемого пользователем жеста **pan**...

Scrolling in a UIScrollView ...



Scrolling in a UIScrollView ...

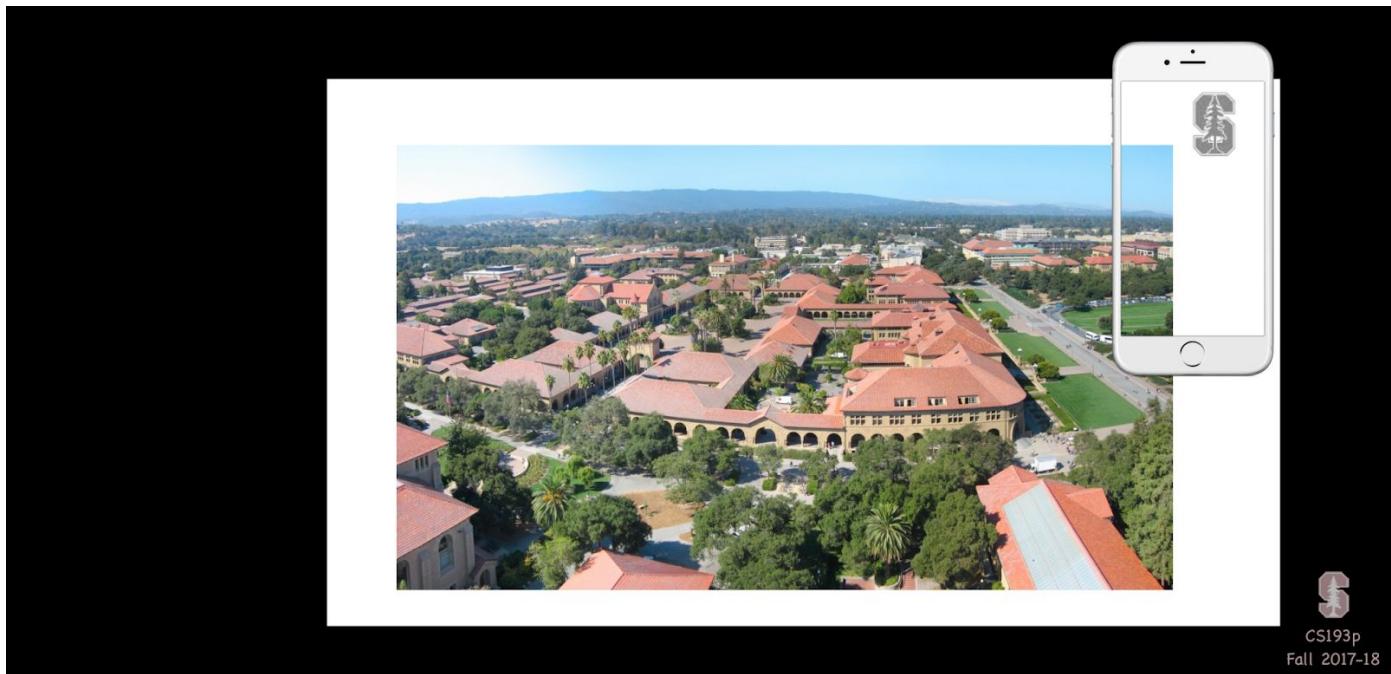


Scrolling in a UIScrollView ...



Scrolling in a UIScrollView ...





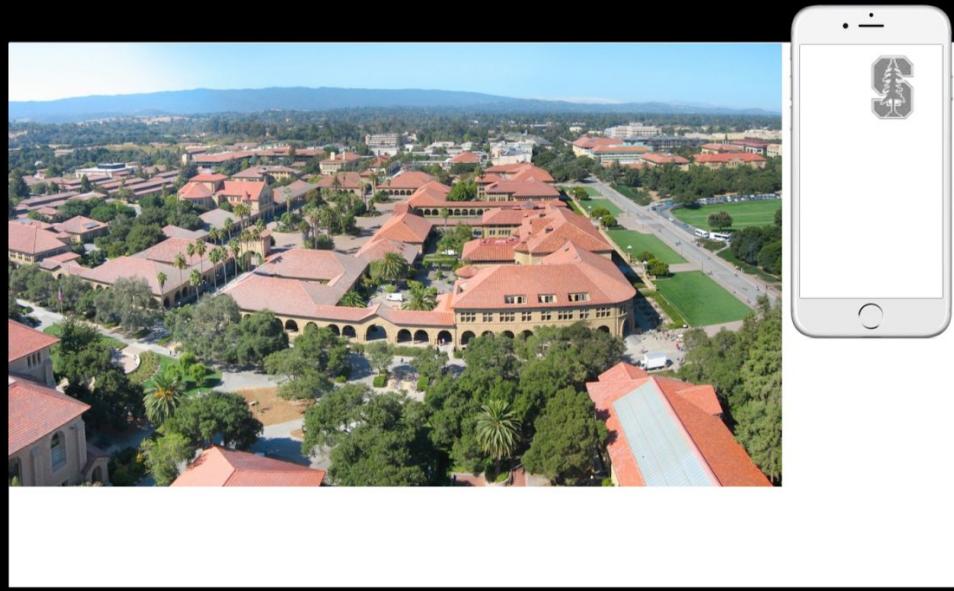
S
CS193p
Fall 2017-18

... и, конечно, вы можете изменять масштаб.

Вы можете задать другое позиционирование для **subviews** в контентной области **contentSize** точно также, как вы можете это делать для **subviews** в обычных **view**:

Positioning subviews in a UIScrollView ...

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
```



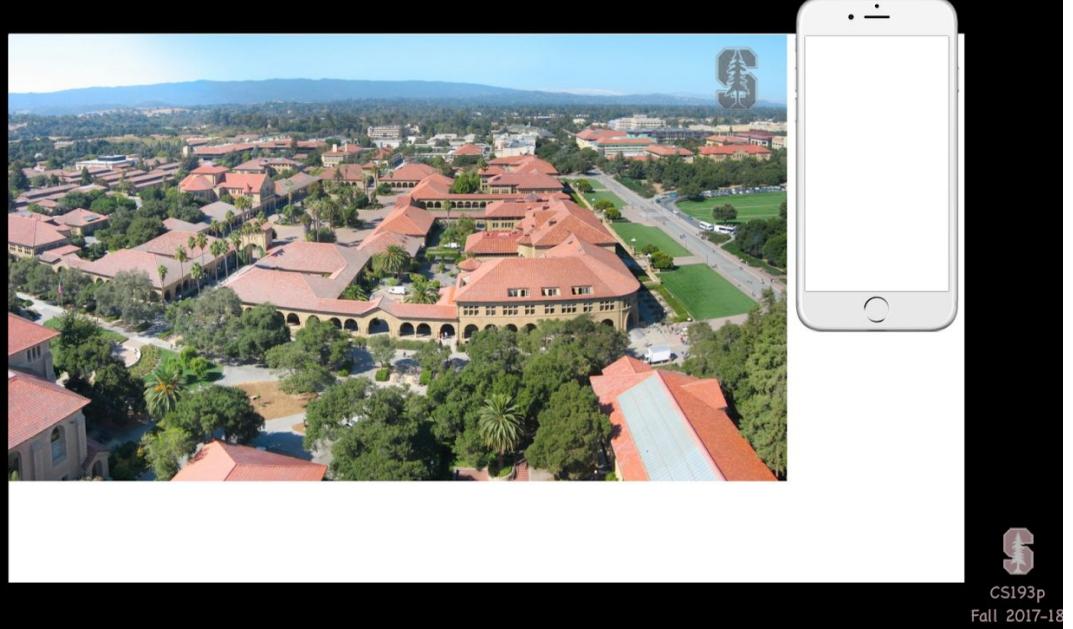
S
CS193p
Fall 2017-18

Я просто установил новый **frame** большому **aerial view**, и он позиционировался точно в левом верхнем углу.

А затем я установил другой **frame** для логотипа Стэнфорда **logo**, и он также занял новую позицию:

Positioning subviews in a UIScrollView ...

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)  
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
```



 CS193p
Fall 2017-18

Теперь наш **scrollView** показывает полностью белое пространство, которое образовалось по краям. Я не хочу, чтобы так было, поэтому я переустановлю размер области **contentSize**, чтобы она соответствовала **frame** второго **view** - **aerial**:

Positioning subviews in a UIScrollView ...

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)  
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)  
scrollView.contentSize = CGSize(width: 2500, height: 1600)
```



Теперь **scrollView** будет прокручивать новую область **contentSize**:

That's it!



That's it!



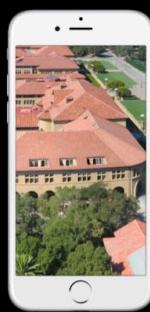
That's it!



That's it!



That's it!



Понятно?

----- 30-ая минута лекции -----

На этом все. Область **contentSize** - это ключевой момент **scrollView**.

Есть еще пара интересных вещей в **scrollView** типа переменной **var** с именем **contentOffset**.

Where in the content is the scroll view currently positioned?

let upperLeftOfVisible: CGPoint = scrollView.contentOffset
In the content area's coordinate system.

Где в контенте позиционирован текущий экран Scroll View?

В системе координат контентной области.

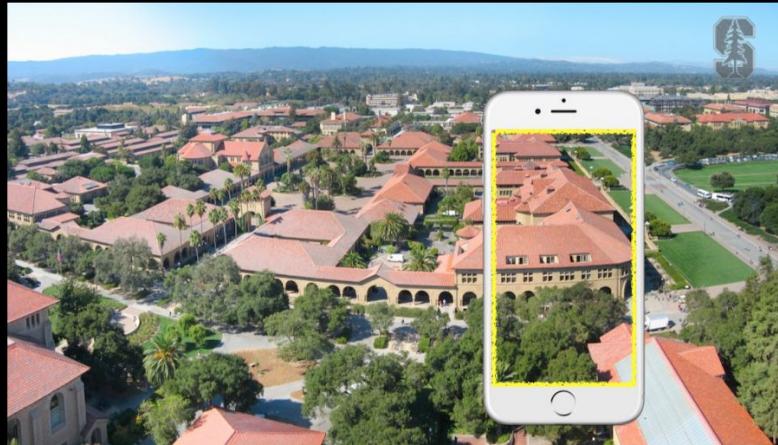
The diagram shows a white iPhone displaying a landscape view of a university campus. A green horizontal arrow labeled "contentOffset.x" points left from the center of the screen. A red vertical arrow labeled "contentOffset.y" points up from the center of the screen. The text "contentOffset.y" is above the red arrow, and "contentOffset.x" is below the green arrow. The entire diagram is overlaid on a larger image of the same Stanford campus scene.

Переменная **contentOffset** - это просто точка с координатами **x** и **y** положения верхнего левого угла той области, которую в данный момент показывает **scrollView** в системе координат контентной области. Так что в названии переменной **contentOffset** звучит в точности то, что она и делает - смещение (**offset**) в контентной области того, что показывается **scrollView**.

Вы можете также масштабировать то, что попало в область видимости **scrollView**. Но если вы будете масштабировать, то вам, определенно, очень интересно знать, на какой же прямоугольник в системе координат контентной области я в данный момент смотрю через **scrollView**?

What area in a subview is currently visible?

```
let visibleRect: CGRect = aerial.convert(scrollView.bounds, from: scrollView)
```



Why the `convertRect`? Because the `scrollView's bounds` are in the `scrollView's coordinate system`. And there might be zooming going on inside the `scrollView` too ...

CS193p
Fall 2017-18

Какая область в subview в настоящий момент видна?

Почему **convertRect**? Потому что границы **bounds** даны в системе координат **scrollView**.

Внутри **scrollView** может также осуществляться масштабирование...

Из-за того, что я могу масштабировать изображение внутри “окошка”, через которое я смотрю, то это реально может быть очень маленькая часть контентной области. И я использую здесь тот же самый метод, что вы возможно, использовали в своем домашнем Задании. Это методы **convert(_:to:)** или **convert(_:from:)** класса **UIView**, которые преобразуют работающие в **UIScrollView** точку и прямоугольник в другую систему координат.

Если мы напишем код ...

```
let visibleRect:CGRect = aerial.convert (scrollView.bounds , from: scrollView)
```

... то мы преобразуем прямоугольник **scrollView.bounds** из системы координат **scrollView** в систему координат, связанную с **aerial view**. В результате мы получим границы **bounds** нашего **scrollView** в системе координат **aerial view**.

Примечание переводчика. Этот **scrollView.bounds** - желтый прямоугольник, выделенный на слайде. Ведь **scrollView** - это обычное **view** и оно имеет границы **bounds**, это границы видимой в данный момент области в системе координат **scrollView**.

Вы должны преобразовать прямоугольник `scrollView.bounds` в прямоугольник интересующей вас области. В нашем случае это `aerial`.

Я это сделал с помощью кода:

```
let visibleRect: CGRect = aerial.convert(scrollView.bounds, from: scrollView)
```

Почему это может быть совсем не одно и то же?

По многим причинам.

Одна причина связана с тем, что мы используем в `scrollView` жест `pan` и этот прямоугольник всегда изменяется.

Вторая причина состоит в том, что мы можем изменять масштаб (`zoom in и out`), так что в этом случае они тоже будут отличаться. Если вы увеличиваете или уменьшаете масштаб на `scrollView`, то маленькая область может представлять как огромный кусок `aerial`, так и крошечный кусок `aerial`. Кто знает?

Поэтому мы используем этот метод `convert`.

Вот как мы находим прямоугольник в системе координат `aerial view`. Это обычный нормальный `convert`, в нем нет ничего специфического.

Как мы создаем `ScrollView`?

Мы, конечно, можем вытянуть его из Палитры Объектов в Области Утилит на storyboard.

Вы также можете выделить некоторый `view` на storyboard и пойти в меню **Editor -> Embed In -> ScrollView** и вставить этот `view` в `ScrollView` точно также, как вы вставляли `view` в **Navigation Controller**. В этом случае `ScrollView` “обернет” ваш `view`. К сожалению, он разместит рамку шириной в **20 points**, что очень мне не нравится. Я покажу вам это в сегодняшнем демонстрационном примере и покажу как избавиться от этого. Вы можете выбрать любой способ создания `ScrollView` или, конечно, вы можете создать его в коде. Это обычный `UIView` и у него также есть инициализатор `UIScrollView(frame:)`.

UIScrollView

• How do you create one?

Just like any other `UIView`. Drag out in a storyboard or use `UIScrollView(frame:)`.

Or select a `UIView` in your storyboard and choose “Embed In -> Scroll View” from Editor menu.

• To add your “too big” `UIView` in code using `addSubview` ...

```
if let image = UIImage(named: "bigimage.jpg") {
    let iv = UIImageView(image: image) // iv.frame.size will = image.size
    scrollView.addSubview(iv)
}
```

Add more subviews if you want.

All of the subviews' frames will be in the `UIScrollView`'s content area's coordinate system (that is, (0,0) in the upper left & width and height of `contentSize.width` & `.height`).

• Now don't forget to set the `contentSize`

Common bug is to do the above lines of code (or embed in Xcode) and forget to say:

```
scrollView.contentSize = imageView.frame.size (for example)
```



UIScrollView

- **Как вы создаете его?**

Как все другие UIView. Перетягиваем на storyboard из Палитры Объектов или используем UIScrollView (frame:).

Или выбираем любое другое UIView на storyboard и используем меню “Editor -> Embed In -> Scroll View”

- **Для добавления вашего “слишком большого” UIView в коде используем addSubviews...**

```
if let image = UIImage (named: "bigimage.jpg") {  
    let iv = UIImageView (image: image ) // iv.frame.size будет = image.size  
    scrollView.addSubview (iv)  
}
```

Добавьте еще **subviews**, если хотите.

frames всех subviews будут в системе координат контентной области UIScrollView (то есть (0,0) в верхнем левом углу и ширина и высота contentSize.width & .height).

- **Теперь не забудьте установить contentSize**

Очень распространенная ошибка - это выполнить 3 строки кода, приведенные выше (или вставить в Xcode) и забыть написать

```
scrollView.contentSize = imageView.frame.size (например)
```

Как только вы создали **ScrollView**, например, на storyboard и создали **Outlet** на него, вы можете просто использовать метод **addSubview** для добавления к нему **subviews**. Но это ничего не даст - ничего не появится на экране до тех пор, пока вы не установите **ScrollView contentSize**. Если вы не установите **contentSize**, то **ScrollView** будет прокручивать маленький прямоугольник, находящийся в левом верхнем углу и имеющий ширину **width** и высоту **height** равные **0**.

Если вы размещаете там **subview**, то, возможно, он выйдет за пределы области **contentSize** и будет показываться, но его нельзя будет прокручивать. Так что, если вы используете **ScrollView**, ваше изображение (**image**) появляется, или ваши **views** появляются, но вы не можете выполнять жесты **pan** и **pinch**, то, возможно, это из-за того, что **contentSize** равен **0**. Жесты **pan** и **pinch**, выполняются только в контентной области.

Вы можете прокручивать контент **ScrollView** программно, и, конечно, пользователь может прокручивать **ScrollView** с помощью пальцев. Но тоже самое можно сделать в коде с использованием **scrollRectToVisible**, определив прямоугольник в системе координат контентной области **contentSize**, и тогда **Scroll View** выполняет прокрутку так, чтобы показать как можно большую часть этого прямоугольника.

UIScrollView

• Scrolling programmatically

```
func scrollRectToVisible(CGRect, animated: Bool)
```

• Other things you can control in a scroll view

Whether scrolling is enabled.

Locking scroll direction to user's first "move".

The style of the scroll indicators (call `flashScrollIndicator`s when your scroll view appears).

Whether the actual content is "inset" from the content area (`contentInset` property).

UIScrollView

- Прокручивание программным путем

```
func scrollRectToVisible (CGRect, animated: Bool)
```

- Другие вещи, которыми вы можете управлять в Scroll View

Доступно ли прокручивание (enable).

Блокирование направления прокручивания при первом движении пользователя.

Стиль индикатора прокручивания (вызываем `flashScrollIndicator`s, когда ваш scroll view появляется).

Отступает ли действительный контент от контентной области (свойство `contentInset`).

Есть еще множество других вещей, которые вы можете делать со **ScrollView**, но у меня нет времени рассказывать о них. Например, когда **ScrollView** впервые появляется на экране, вы можете заставить "мигать" индикатор прокрутки. Два элемента по краям, которые показывают, где в данный момент выполняется прокрутка, могут "мигать", что устанавливается по умолчанию. Существует специальная переменная `var flashScrollIndicator`s, которую можно переключить в состояние `false`, если вы не хотите, чтобы происходило это "мигание".

ScrollView - супер сообразительный относительно **Safe Area**. Помните? Мы говорили о **Safe Area**, как о некотором месте в самом верху **iPhone X**? Или если вы находитесь в заголовке **Navigation Control** или на закладке в самом низу **TabBarController**, то в этих местах не безопасно рисовать. Однако **ScrollView** позволяет вам делать прокрутку даже если вы находитесь в этих небезопасных местах. Но если вы будете прокручивать все время вниз или все время в сторону, то он должен убедиться, что ваш контент не скрывается **Safe Area**. Вы увидите это в демонстрационном примере.

Так что **ScrollView** - супер сообразительный относительно **Safe Area**, потому что очень часто встречается такой случай, когда у вас есть **ScrollView**, и вы хотите получить полный экран для показа как можно большей части своего изображения. Но вы также знаете, что у вас есть несколько пикселей в левом верхнем углу, мы не хотим, чтобы наше изображение находилось по ними, это также касается маленькой черной полоски на самом верху **iPhone X**. Мы хотим, чтобы прокрутка нашего изображения не заходила так далеко.

А как насчет масштабирования (**zooming**)?

До сих пор я рассказывал вам только о том, как вы можете перемещаться по контентной области с помощью жеста **pan**, но вы также можете увеличить (**zoom in**) или уменьшить (**zoom out**) какой-то выбранный участок контентной области с помощью жеста **pinch**.

UIScrollView

• Zooming

All UIView's have a property (`transform`) which is an affine transform (`translate`, `scale`, `rotate`).

Scroll view simply modifies this transform when you zoom.

Zooming is also going to affect the scroll view's `contentSize` and `contentOffset`.

• Will not work without minimum/maximum zoom scale being set

```
scrollView.minimumZoomScale = 0.5 // 0.5 means half its normal size
```

```
scrollView.maximumZoomScale = 2.0 // 2.0 means twice its normal size
```

• Will not work without delegate method to specify view to zoom

```
func viewForZooming(in scrollView: UIScrollView) -> UIView
```

If your scroll view only has one subview, you return it here. More than one? Up to you.

• Zooming programmatically

```
var zoomScale: CGFloat  
func setZoomScale(CGFloat, animated: Bool)  
func zoom(to rect: CGRect, animated: Bool)
```



CS193p
Fall 2017-18

UIScrollView

• Масштабирование (**zooming**)

Все UIView имеют свойство **transform**, которое является аффинным преобразованием (`translate` - перемещение, `scale` - масштабирование, `rotate` - вращение).

Scroll View просто модифицирует это свойство **transform** при масштабировании.

Масштабирование (**zooming**) воздействует на `contentSize` и на `contentOffset`.

• Не будет работать без установки минимального / максимального масштаба при масштабировании

```
scrollView.minimumZoomScale = 0.5 // 0.5 означает половину нормального размера
```

```
scrollView.maximumZoomScale = 2.0 // 2.0 означает удвоение нормального размера
```

• Не будет работать без выполнения метода делегата по определению `view`, которое должно масштабироваться

```
func viewForZooming( in scrollView: UIScrollView) -> UIView
```

Если у Scroll View всего один subview, вы возвращаете его. Если больше одного? То выбор за вами.

• Масштабирование (**zooming**) программно

```
var zoomScale: CGFloat
```

```
func setZoomScale ( CGFloat, animated: Bool)
```

```
func zoom (to rect:CGRect, animated: Bool)
```

Одна вещь, с которой нужно быть внимательным при увеличении масштаба (**zoom in**), это его влияние на ваш **contentSize**. Если я увеличиваю масштаб (**zoom in**), чтобы сделать то, что рассматриваю, большего размера, контентная область также становится больше. Когда такое происходит, то люди ожидают, что их контентная область осталась прежнего размера, но это не так. **ScrollView** АВТОМАТИЧЕСКИ изменяет ее размер для вас, так что не о чем беспокоиться. Но если вы посмотрите на ваш **contentSize** при увеличении масштаба (**zoomed in**), то он станет больше или меньше при уменьшении масштаба (**zoom out**). То же относится, очевидно, и к **contentOffset**.

Как заставить работать масштабирование (**zooming**)?

Мы должны сделать две вещи. Не забудьте сделать эти две вещи. Без них масштабирование работать НЕ будет.

----- 35-ая минута лекции -----

Первая вещь - это установка минимального и максимального коэффициентов масштабирования: **minimumZoomScale** и **maximumZoomScale**. Эти свойства показывают, насколько вам разрешено уменьшать или увеличивать масштаб. По умолчанию **minimumZoomScale** и **maximumZoomScale** установлены в **1**, что означает “никакого масштабирования”. Вам нужно установить им какие-то значения. Если вы установите **minimumZoomScale** в **0.5**, то это означает уменьшение масштаба до половины нормального размера. Если вы установите **maximumZoomScale** в **2.0**, то вы можете максимально только удвоить нормальный размер. Вы можете решить, в каких пределах пользователю позволено увеличивать (**zoom in**) или уменьшать (**zoom out**) масштаб. По умолчанию **minimumZoomScale** и **maximumZoomScale** установлены в **1**, что означает “никакого масштабирования” и это должно иметь размер **identity**.

Масштабирование в **ScrollView** работает путем модификации свойства **transform**. Вы знаете, что у всех **UIViews** есть свойство **transform**, это аффинное преобразование, которое означает **translate** - перемещение, **scale** - масштабирование, **rotate** - вращение, которые включены в этот **transform**.

ScrollView использует **scale** часть преобразования **transform** для масштабирования **subviews**, мы должны будем сказать какие **subviews** будут подвергаться увеличению и уменьшению масштаба. Вот как работает **ScrollView**. Когда масштаб увеличивается или уменьшается, **ScrollView** не делает ничего, за исключением преобразования **transform** того **view**, на который вы указали. Устанавливая свойства **minimumZoomScale** и **maximumZoomScale**, мы говорим **ScrollView** насколько мы позволяем модифицировать это свойство **transform**. Если они равны **1.0**, то **transform** всегда будет **identity** матрицей и не будет никакого увеличения или уменьшения масштаба. Вам нужно дать этим свойствам какие-то значения, по крайней мере, одному из двух, а иначе вы не сможете масштабировать.

Вторая вещь, которую необходимо сделать, это определить **view**, которое трансформируется.

Потому что помните? Контекстной области **ScrollView** разрешается иметь множество **subviews**. У нас было **logo view** и у нас было **aerial view**. Когда я выполняю жест **pinch**, то какое из этих 2-х

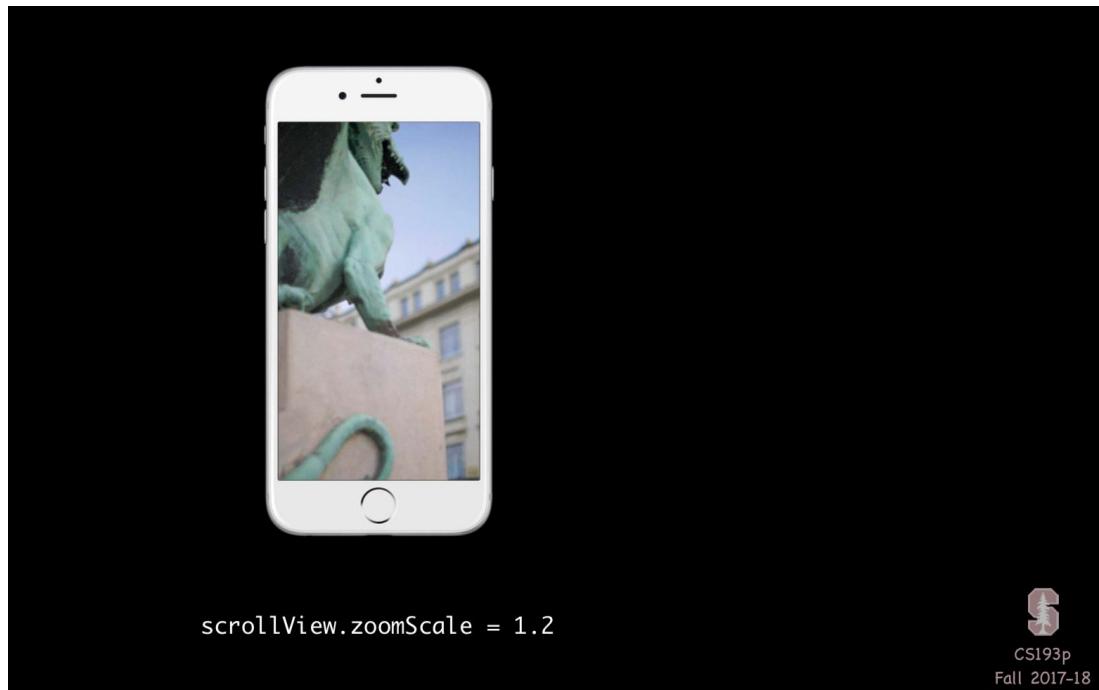
views должно масштабироваться?

Вы должны реализовать метод делегата **ScrollView**. Масштабирование не будет работать без делегирования. Метод делегата **delegate ScrollView** имеет имя **viewForZooming (in scrollView: UIScrollView)** и он просто возвращает **subview**, который вы собираетесь масштабировать.

Честно говоря, в нормальной ситуации, если у вас множество **subviews**, то вы можете разместить их внутри некоторого **view**, например, топового **view**, и именно это **view** вы будете масштабировать. И, конечно, если вы примените **transform** к этому **view**, то все его **subviews** также будут масштабироваться. Обычно, нет необходимости иметь два различных **views**, которые имеют **subviews** в контентной области **ScrollView**, и которые не являются **subviews** некоторого общего **view**. Это можно сделать, но обычно мы так не делаем.

Мы можем выполнять масштабирование программно в дополнение к **pinch** масштабированию. Вы можете установить коэффициент масштабирования **zoomScale** с помощью функции **setZoomScale** или масштабировать в прямоугольнике **zoom (to rect:)**.

Вот пример того, как это делается.



Здесь у меня **zoomScale 1.2**, это на 20% больше, чем **identity** матрица. Произошло небольшое увеличение масштаба (**zoom in**).

Опять вернемся к нормальному состоянию **zoomScale 1.0**, то есть к **identity** матрице.



```
scrollView.zoomScale = 1.0
```



Видите, масштаб немного уменьшился?

Возвращаемся к 1.2.



```
scrollView.zoomScale = 1.2
```



Видите? 1.2 означает, что я увеличиваю масштаб на 20%.

Но это не будет работать, если я не установлю значения свойствам **minimumZoomScale** и **maximumZoomScale**.

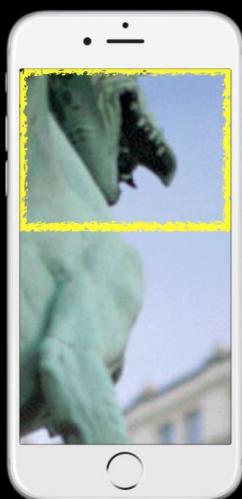
Еще круче увеличение масштаба внутри прямоугольника с помощью **zoom (to rect:)**:



```
zoom(to rect: CGRect, animated: Bool)
```



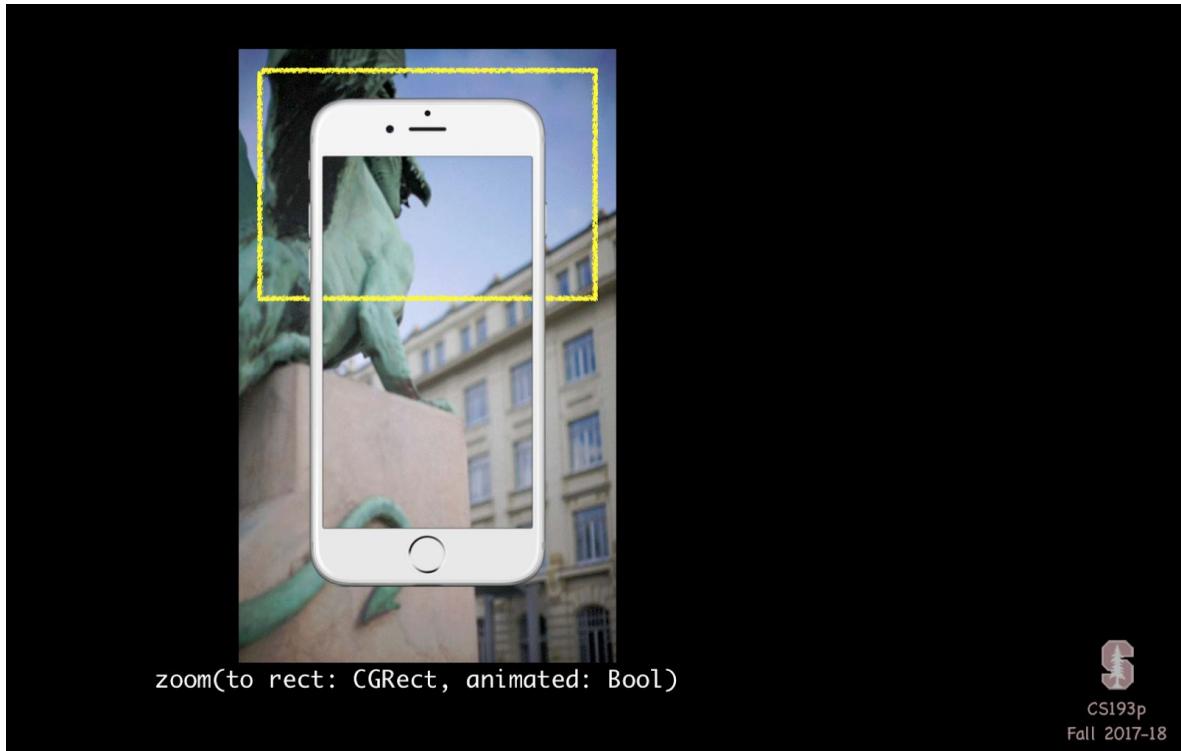
Если у вас маленький прямоугольник желтого цвета, и вы вызываете метод **zoom (to rect:)**, то желтый прямоугольник “растягивается” до достижения им края экрана насколько это возможно.



```
zoom(to rect: CGRect, animated: Bool)
```



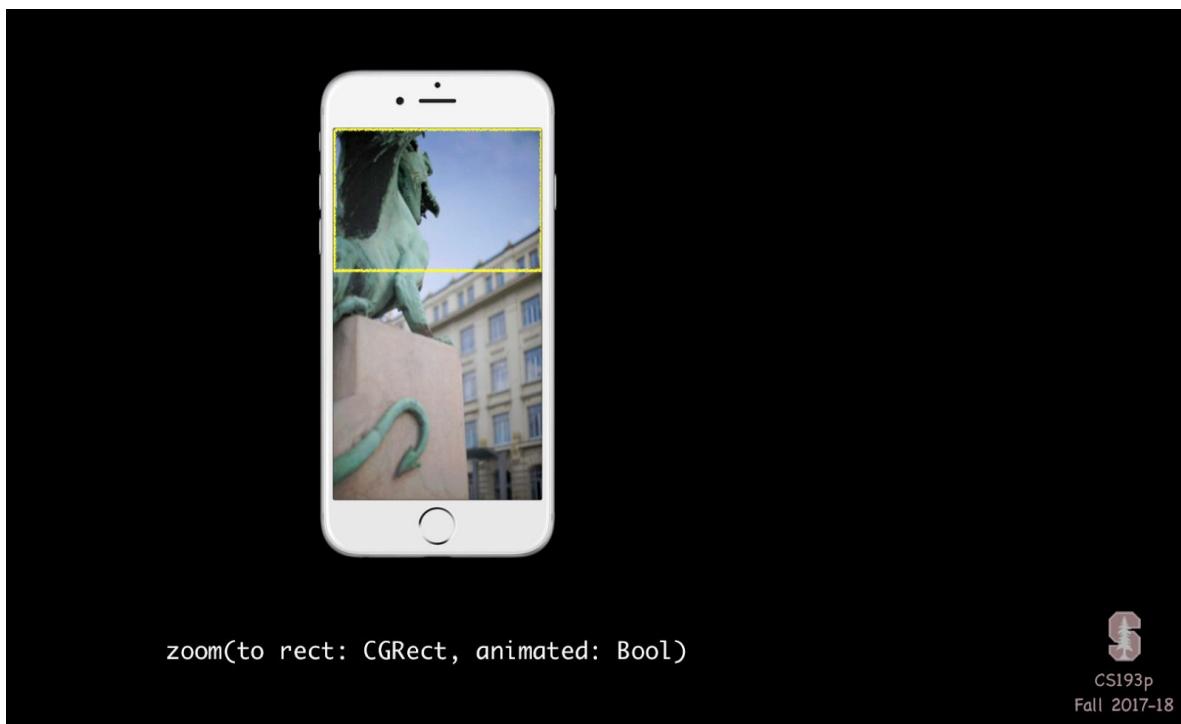
Или если у вас прямоугольник большего размера и выходит за пределы экрана...



`zoom(to rect: CGRect, animated: Bool)`


CS193p
Fall 2017-18

то применение метода **zoom (to rect:)** “сжимает” прямоугольник до того состояния, чтобы он мог уместиться на экране.



`zoom(to rect: CGRect, animated: Bool)`


CS193p
Fall 2017-18

У **ScrollView** множество методов делегата **delegate**, порядка 12 или 15. Я не буду говорить обо всех этих методах. Мы уже встречались с одним из них.

Это:

viewForZooming (in scrollView: UIScrollView)

Но есть еще один метод, я привел его в качестве примера, это:

scrollViewDidEndZooming (UIScrollView, withView: UIView, atScale: CGFloat)

Он посыпается вам, если пользователь выполняет жест **pinch**, а затем поднимает пальцы вверх от экрана. Помните? Выполнение жеста **pinch** приводит к изменению свойства **transform** при

масштабировании этого **view**, и все, ничего более не происходит. Когда вы отпускаете пальцы, что, если у вас есть возможность нарисовать (**drawing**) ваш **view** с большим разрешением, если масштаб увеличился (**zoomed in**)?

UIScrollView

- **Lots and lots of delegate methods!**

The scroll view will keep you up to date with what's going on.

- **Example: delegate method will notify you when zooming ends**

```
func scrollViewDidEndZooming(UIScrollView,  
                             with view: UIView, // from delegate method above  
                             atScale: CGFloat)
```

If you redraw your view at the new scale, be sure to reset the transform back to identity.

UIScrollView

- **Множество методов делегата!**

Scroll View будет держать вас в курсе дела того, что происходит.

- **Пример: метод делегата уведомит вас, когда закончится масштабирование**

```
func scrollViewDidEndZooming (UIScrollView,  
                             withView: UIView, // из предыдущего метода делегата  
                             atScale: CGFloat)
```

Если вы перерисовали ваш **view** в новом масштабе, убедитесь, что вы переустановили **transform** вашего **view** в значение **identity**.

Например, допустим, что ваш **view** рисует дугу (**arc**) с помощью **Core Graphics**. Рисование дуги сводится к размещению группы пикселей по кругу. Если масштаб увеличивается, то эта дуга выглядит как “нечто”, состоящее из маленьких блоков, то есть получается дуга с “зазубренными” краями. Когда вы получаете сообщение

scrollViewDidEndZooming (UIScrollView, withView: UIView, atScale: CGFloat)

... вы можете установить ваш **transform** назад в **identity** матрицу и выполнить рисование в прямоугольнике большего размера. В нашем случае рисования дуги, таким образом можно получить более “гладкую” дугу.

Это хороший способ рисования с более высоким разрешением после увеличения масштаба с помощью жеста **pinch**. Вы продолжаете выполнять масштабирование с помощью **transform** при выполнении жеста **pinch**, но как только пальцы поднимаются, вы заново рисуете **view** с более высоким разрешением, делая **view** более четким.

Coming Up

• Now

Cassini UIScrollView Demo (time permitting)

• Wednesday

Multithreaded Cassini

TextField, et. al.?

• Friday

Instruments (performance analysis tool)

• Next Week

Table View

Collection View

Drag and Drop

Что нас ждет

- Сейчас

Демонстрационный пример Cassini UIScrollView

- Среда

Многопоточный Cassini

TextField и т.д.?

- Пятница

Instruments (инструмент для анализа производительности)

- Следующая неделя

Таблица Table View

Коллекция views Collection View

Механизм Drag и Drop

Я покажу вам демонстрационный пример на тему **ScrollView**. Надеюсь, у меня будет достаточно времени, чтобы показать вам все, о чем я говорил.

Затем в Среду мы будем изучать многопоточность, то есть размещение некоторых вещей в фоновом (**background**) потоке. Мы обновим приложение, которое я напишу сегодня, с тем, чтобы получать изображение из интернета в фоновом (**background**) потоке. Я собираюсь это делать с помощью загрузки изображений в фоновом (**background**) потоке из интернета и сегодня начну показывать, как это делается, но на моем новом лэптопе пока интернет не работает, так что сегодня мы будем использовать локальные файлы для изображений, что вполне допустимо, но в Среду я буду показывать загрузку файлов с изображениями из интернета и я заставлю работать мой лэптоп. Потому что когда загрузка файлов производится из интернета, то это происходит очень медленно.

Сегодняшний демонстрационный пример называется **Cassini**, потому что изображения, которые мы будем загружать, получены с зонда **Cassini**, запущенного к планете Сатурн, и они очень большие. Большие и с очень-очень высоким разрешением, так что их загрузка с помощью даже такой супер быстрой Стэнфордской сети потребует нескольких секунд. Но мы не должны блокировать наш **UI** этой загрузкой.

В Среду я также расскажу о текстовом поле **TextField** и, возможно, о других небольших **UI** элементах, если позволит время.

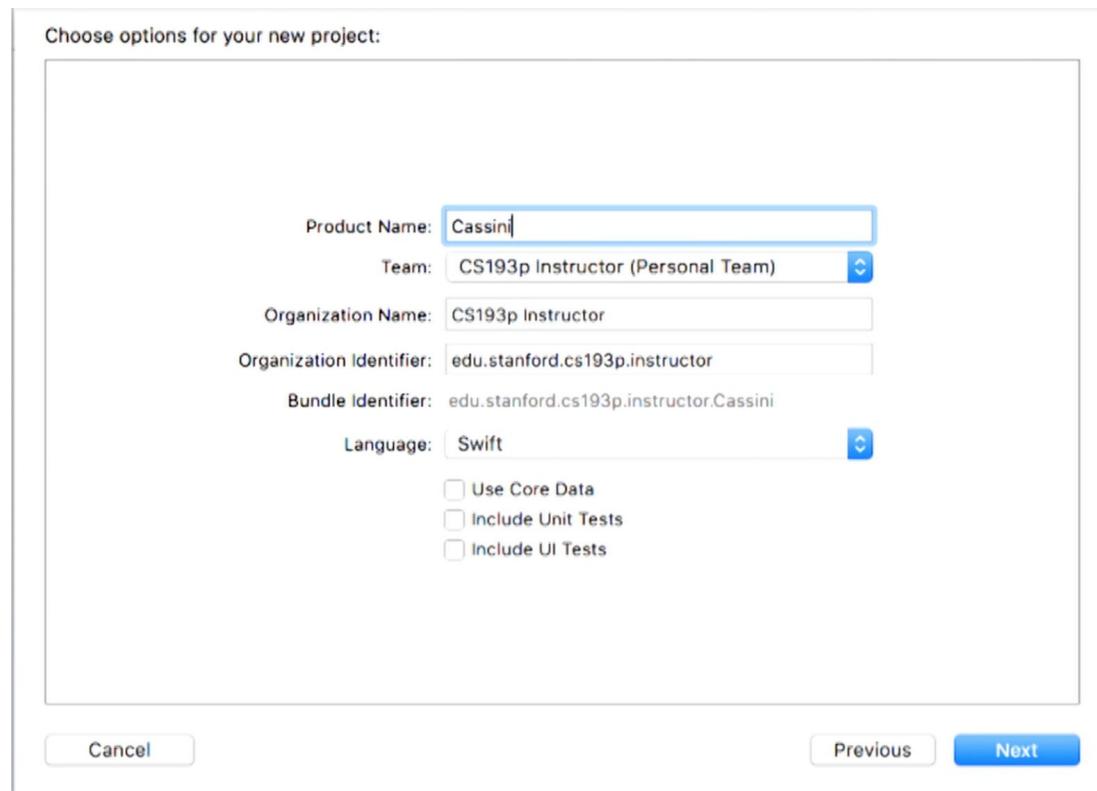
----- 40-ая минута лекции -----

В Пятницу у нас будет секция на этой неделе, она будет посвящена приложению с именем **Instruments**, которое используется для анализа производительности ваших приложений. Не пропустите эту секцию.

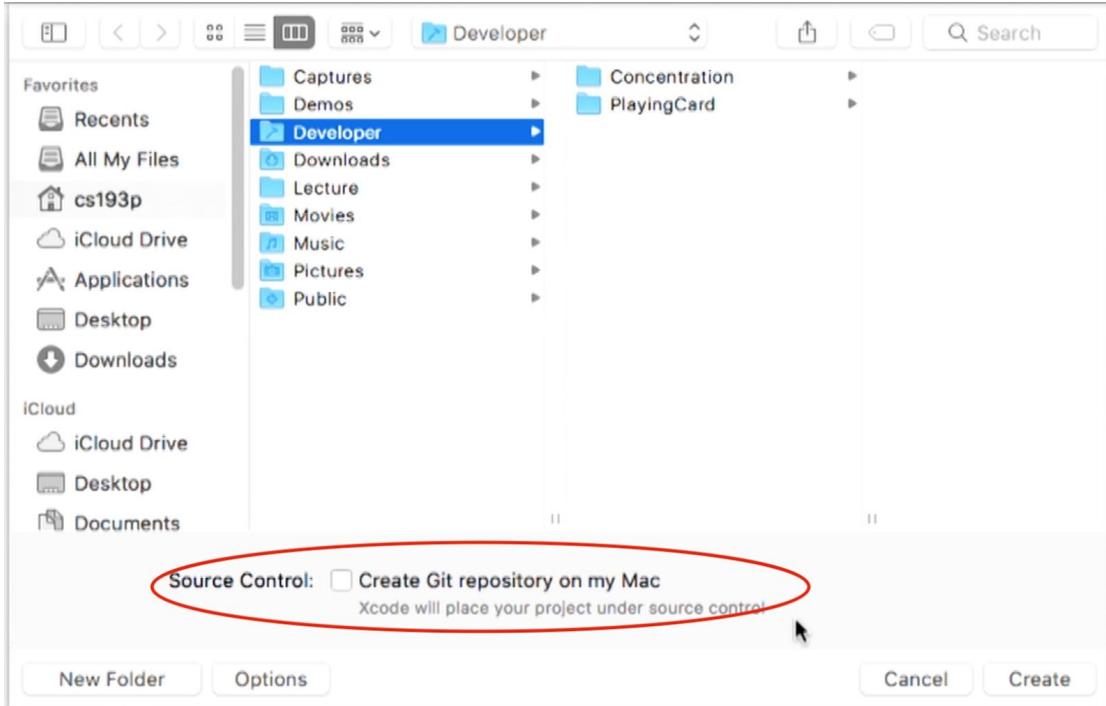
На следующей недели мы “погрузимся” в таблицы **Table View**, в коллекцию **views Collection View**, механизм **Drag & Drop**, которые являются более мощными, более сложными элементами **UI**, чем те, которые мы изучали до сих пор. Потому что они уже требуют хорошего понимания делегирования и подобных ему вещей.

Давайте приступим к демонстрационному примеру **Cassini**.

Я создаю новый проект для **Cassini**:

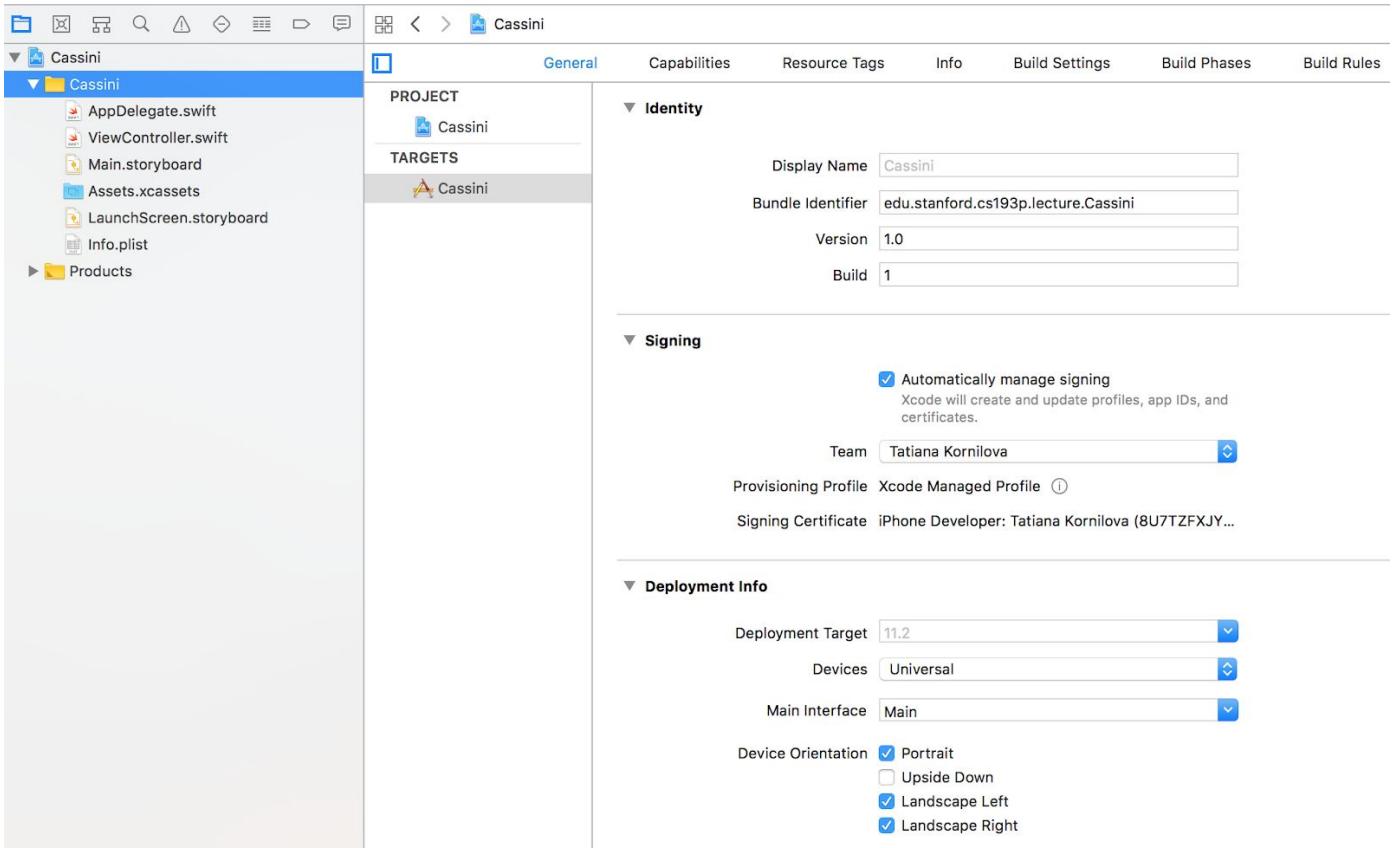


мы ничего специфического не будем делать:

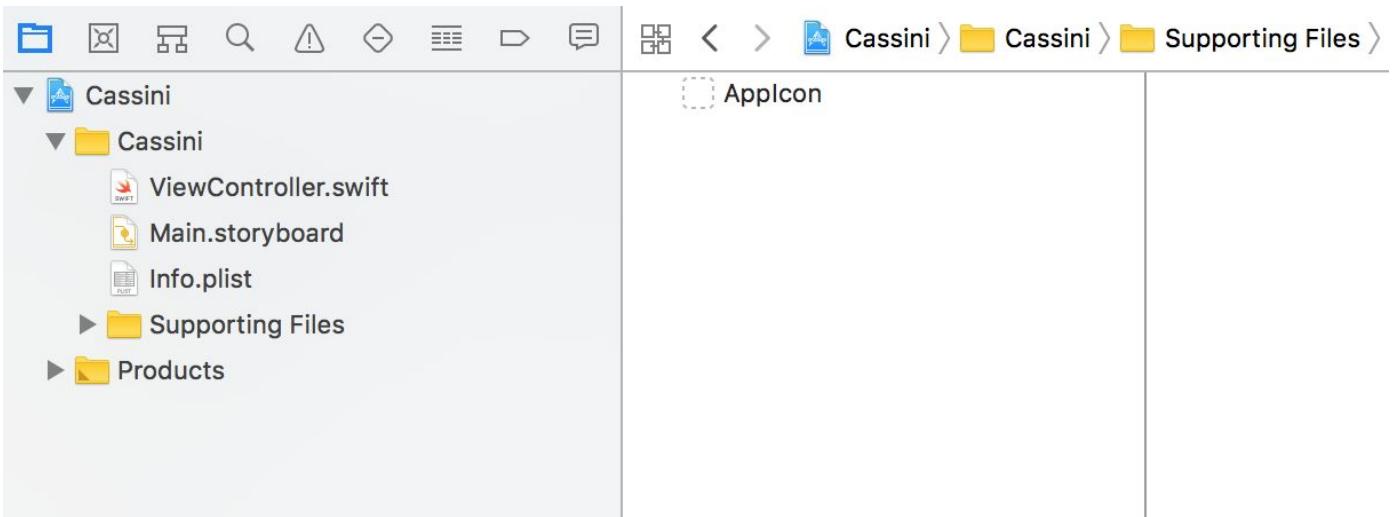


мы не будем задавать опцию “управление исходным кодом” (**Source Code Control**), которую вы изучили.

Это наш **Cassini**:



И я буду делать то, что я всегда делаю: я убираю файлы Assets.xcassets, AppDelegate.swift и LaunchScreen.storyboard в папку **Supporting Files**, чтобы они не смущали вас:

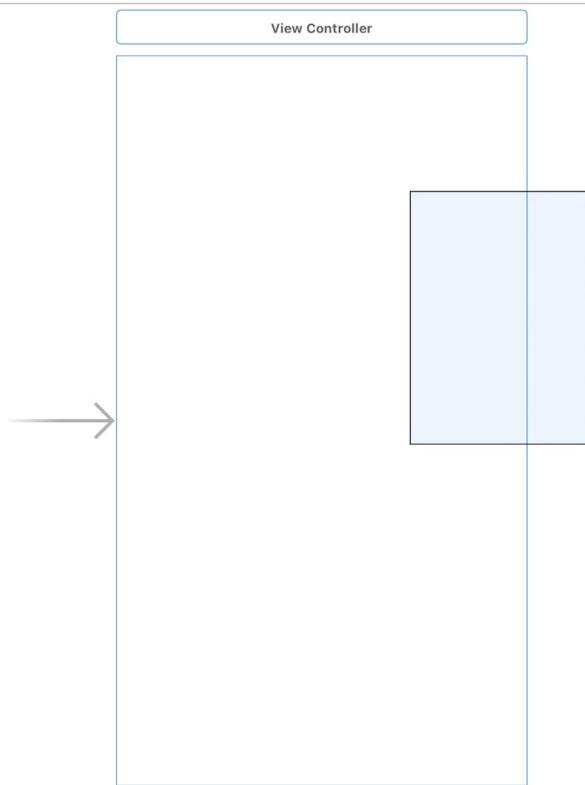


Я хочу сделать что-то интересное с этим приложением на этот раз.

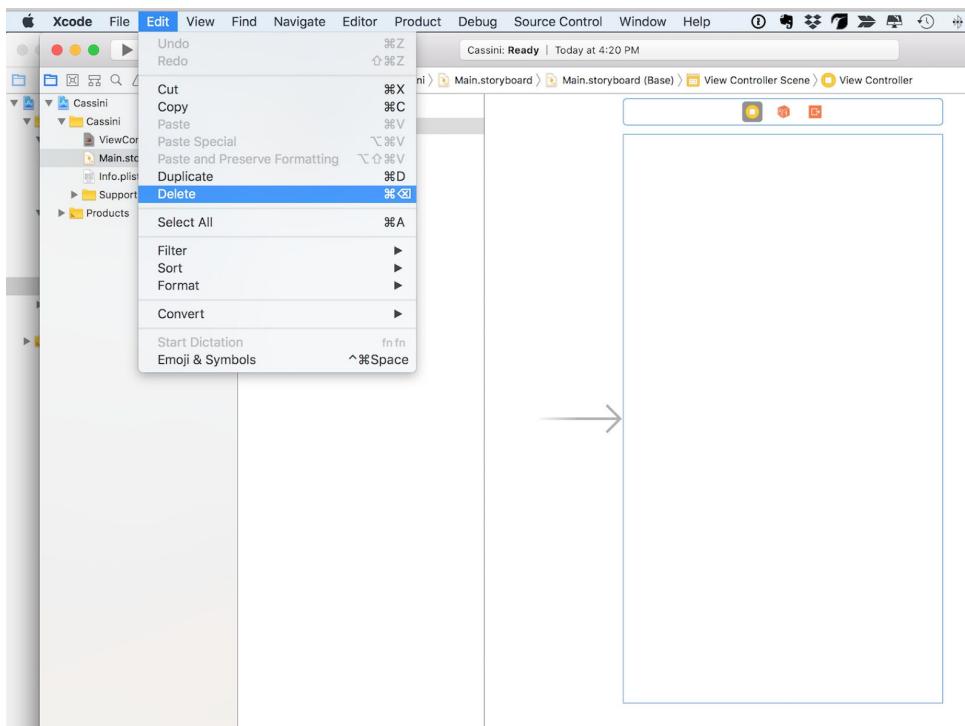
Мы всегда получаем из шаблона **Single View Application** обобщенный (generic) класс **ViewController**, обслуживающий наш экранный фрагмент:

```
1 // ViewController.swift
2 // Cassini
3 // Created by CS193p Instructor.
4 // Copyright © 2017 CS193p Instructor. All rights reserved.
5
6
7
8
9 import UIKit
10
11 class ViewController: UIViewController {
12
13     override func viewDidLoad() {
14         super.viewDidLoad()
15         // Do any additional setup after loading the view, ty
16     }
17
18     override func didReceiveMemoryWarning() {
19         super.didReceiveMemoryWarning()
20         // Dispose of any resources that can be recreated.
21     }
22 }
```

В большинстве случаев, которые у меня были до сих пор, я переименовывал **ViewController** в **ConcentrationViewController** или во что-то свое. Но есть и другой способ сделать то же самое, я иду на **storyboard**, и здесь есть экранный фрагмент **View Controller**, который я выделяю...



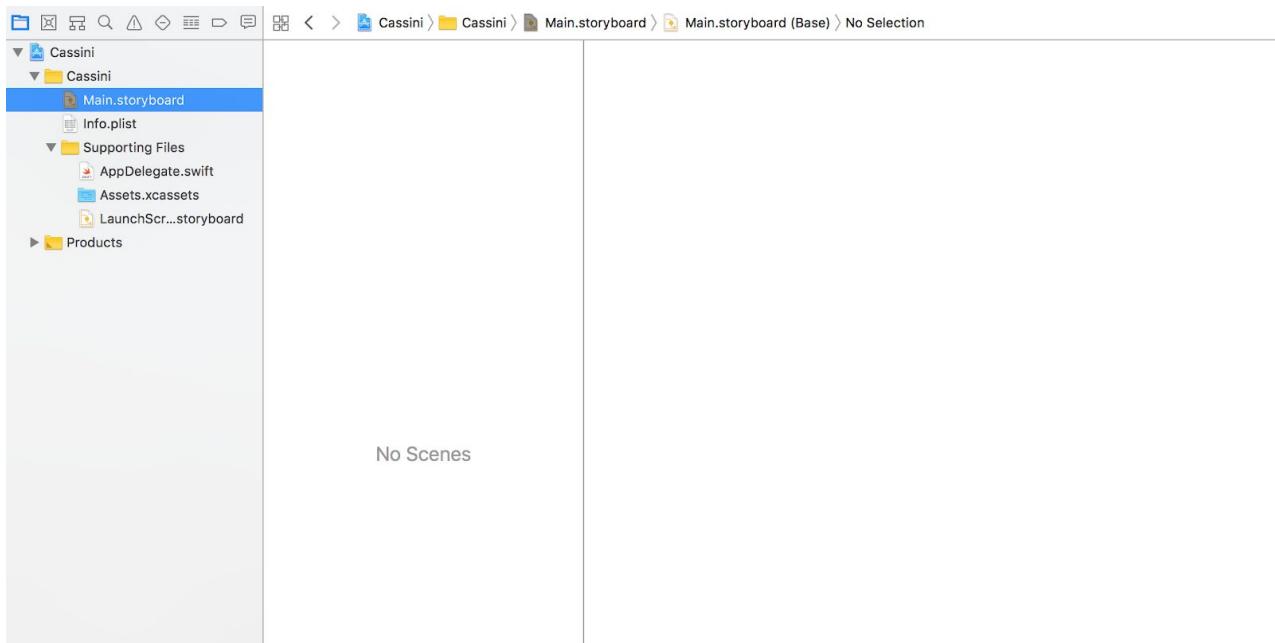
и удаляю:



Теперь у меня полностью пустая **storyboard**. На ней совершенно ничего нет.

Я также удаляю **ViewController (Move to Trash)**.

Теперь у меня абсолютно пустое приложение:



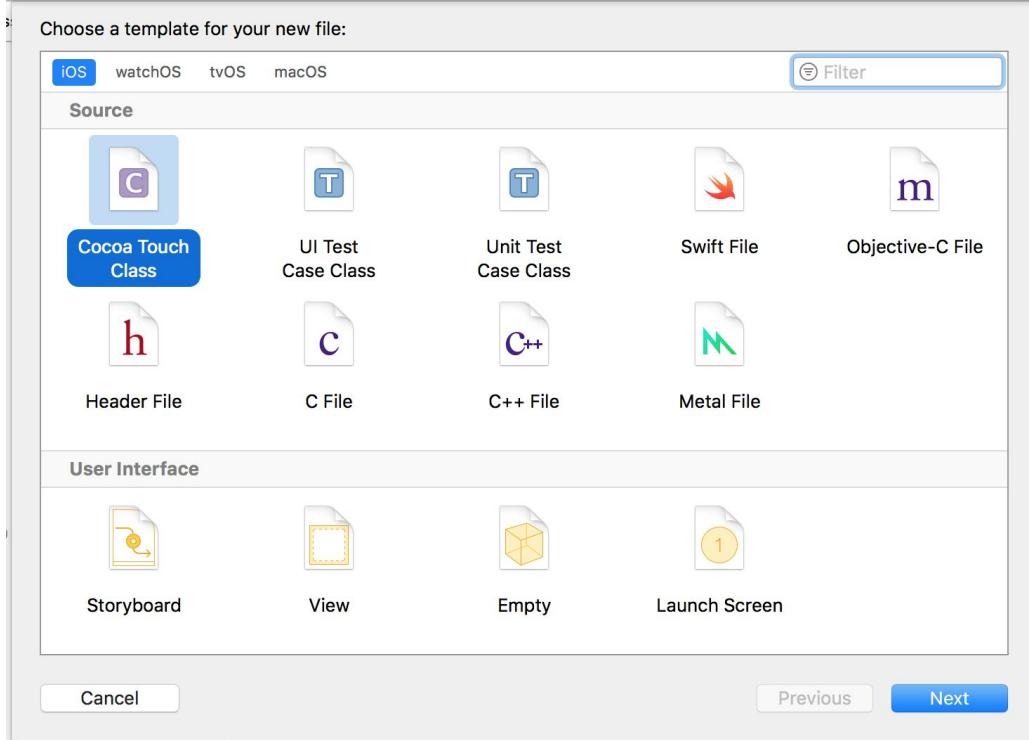
Единственная вещь, которая осталась, это папка **Supporting Files**, в которой также, по существу, ничего нет. Это совершенно нормально и это разрешено - не иметь ничего в своем приложении.

После того, как мы все это проделали, как мне опять создать приложение?

Я собираюсь создать приложение **Cassini**. Вначале я не собираюсь что-то делать с **Cassini**, у нас просто будет изображение (**image**), которое размещается в **ScrollView**. Вам будет разрешено выбрать изображение (**image**) и разместить его в **ScrollView**.

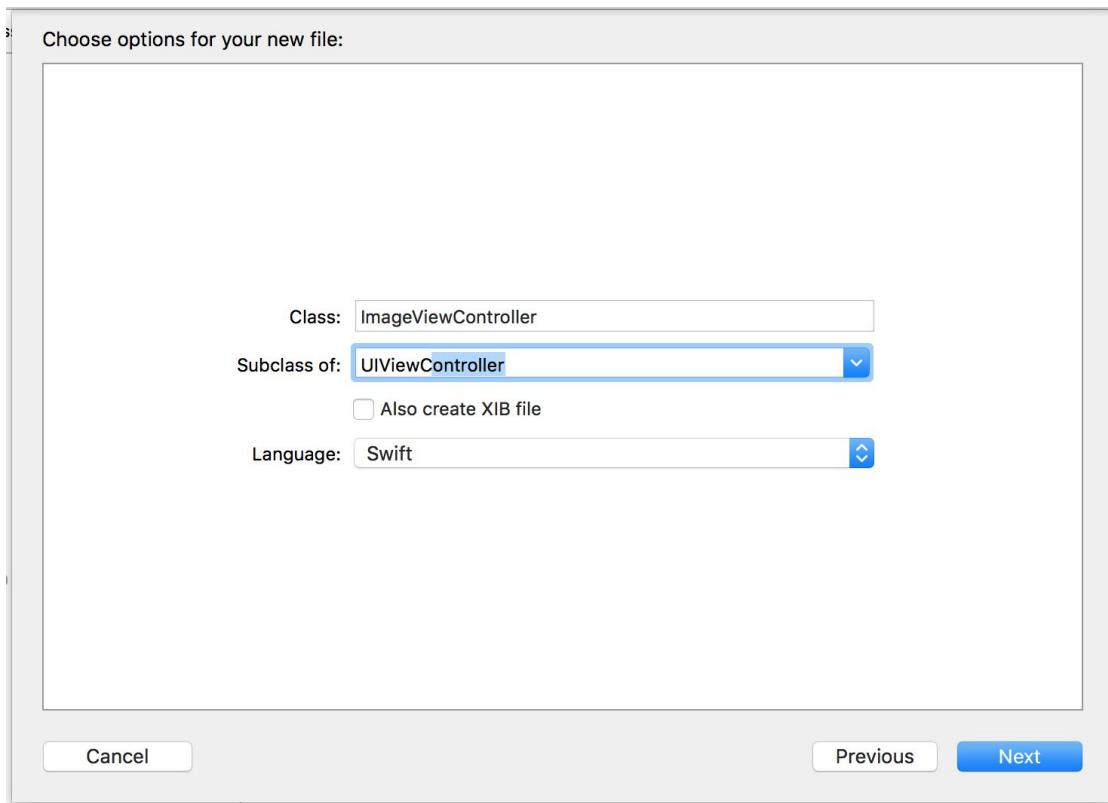
Я хочу создать мой собственный новый **ViewController**, который показывает изображение, и я назову его **ImageViewController**.

Давайте, как всегда, с помощью меню **New -> File -> New -> iOS Source -> Cocoa Touch Class** создадим новый класс:

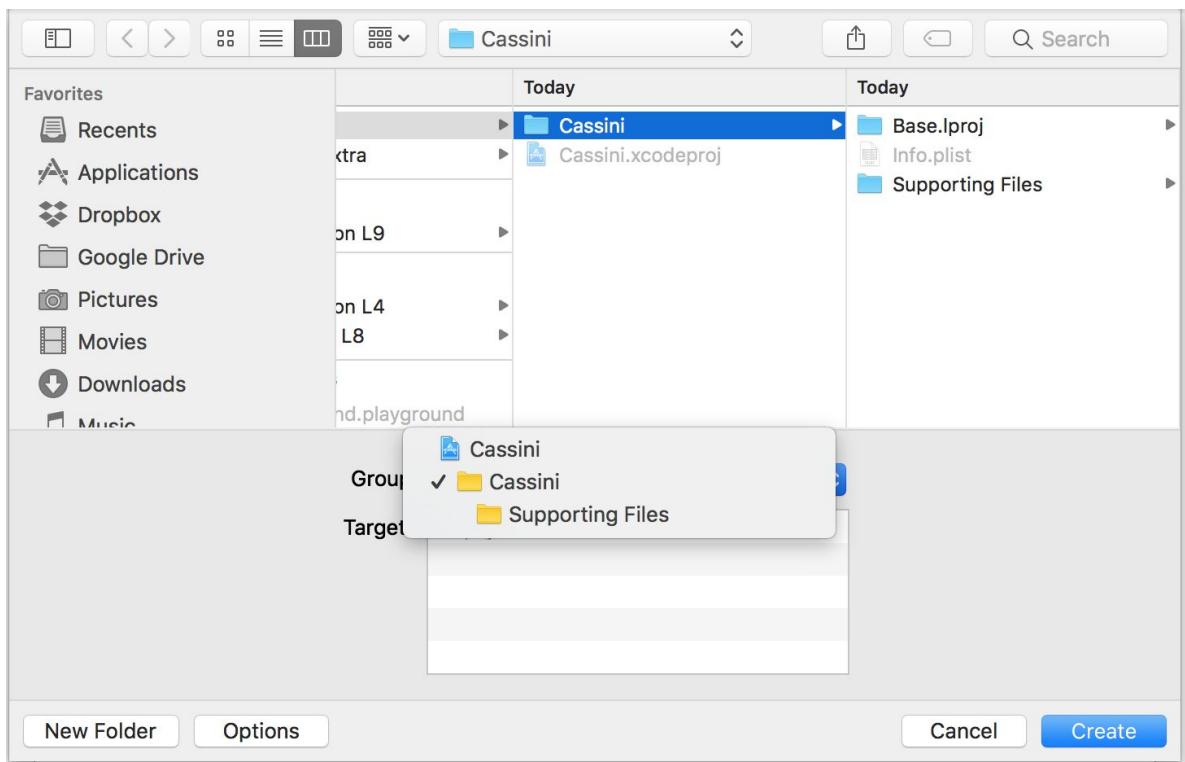


Мы назовем его **ImageViewController**, потому что именно это он и будет делать - показывать

изображение. Это будет **subclass UIViewController**:



Убеждаемся, что мы разместили его в правильном месте:



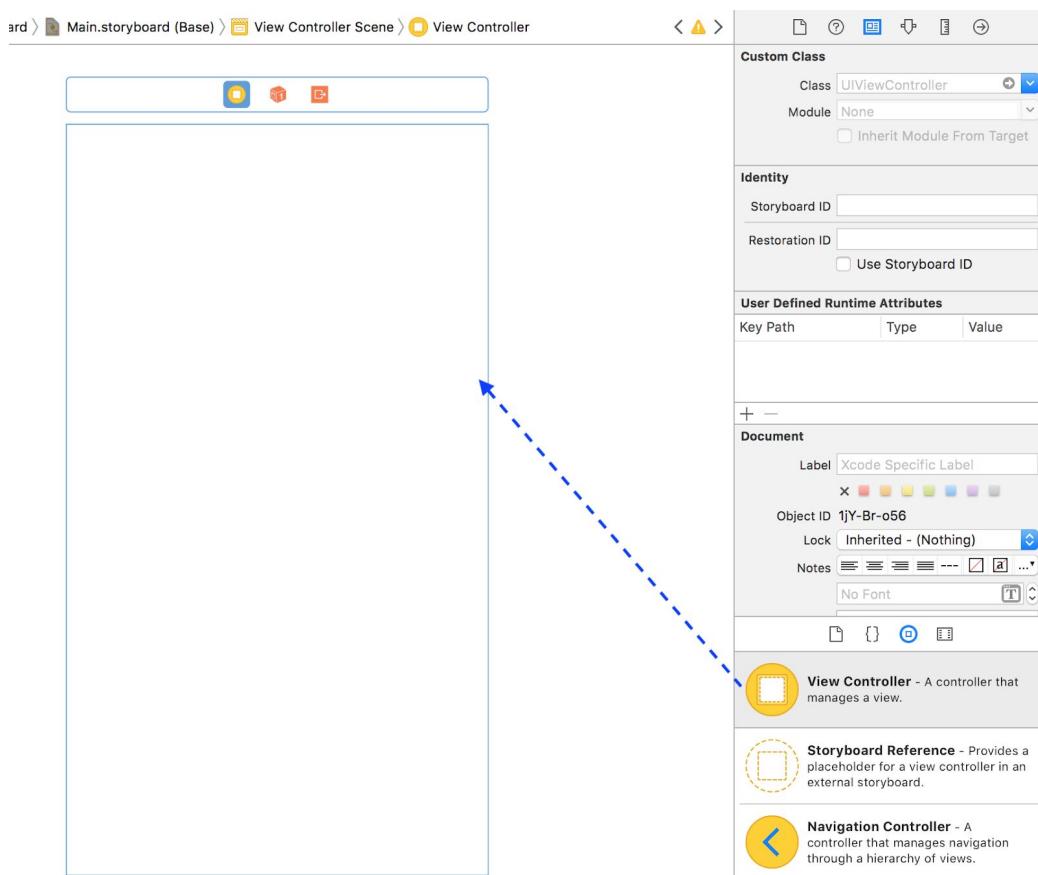
Итак, это наш **ImageView Controller**, в котором я удаляю весь код, пришедший с шаблоном:

```

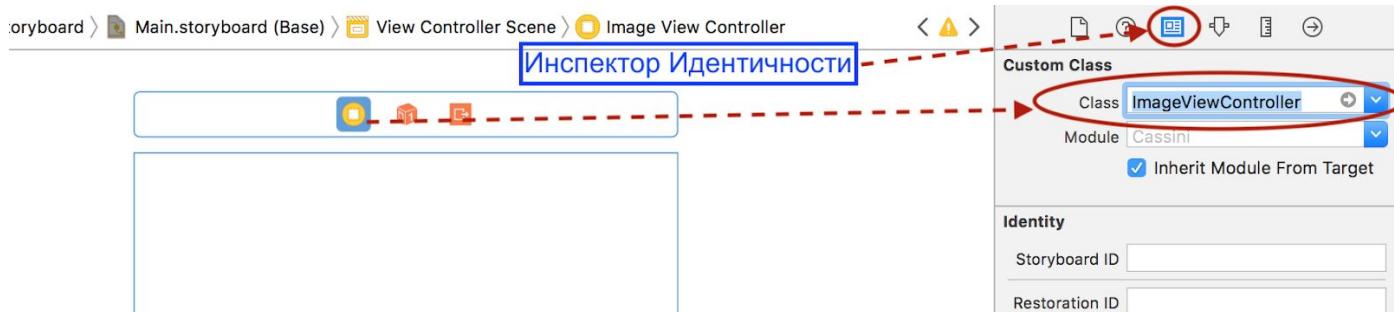
1 // 
2 //  ImageViewController.swift
3 //  Cassini
4 //
5 //  Created by CS193p Instructor.
6 //  Copyright © 2017 CS193p Instructor. All rights reserved.
7 //
8
9 import UIKit
10
11 class ImageViewController: UIViewController {
12
13 }

```

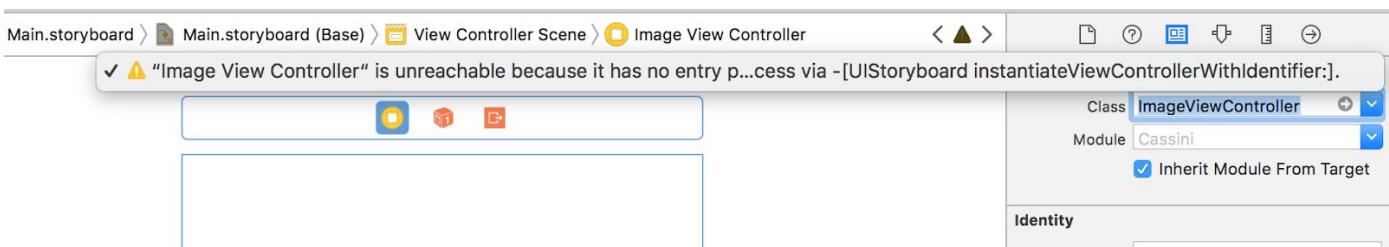
Я возвращаюсь на **storyboard**, которая абсолютно пуста, и перетягиваю на нее новый **View Controller**:



Затем я иду в Инспектор Идентичности (**Identity Inspector**) и изменяю пользовательский класс **Class** на **ImageViewController**:



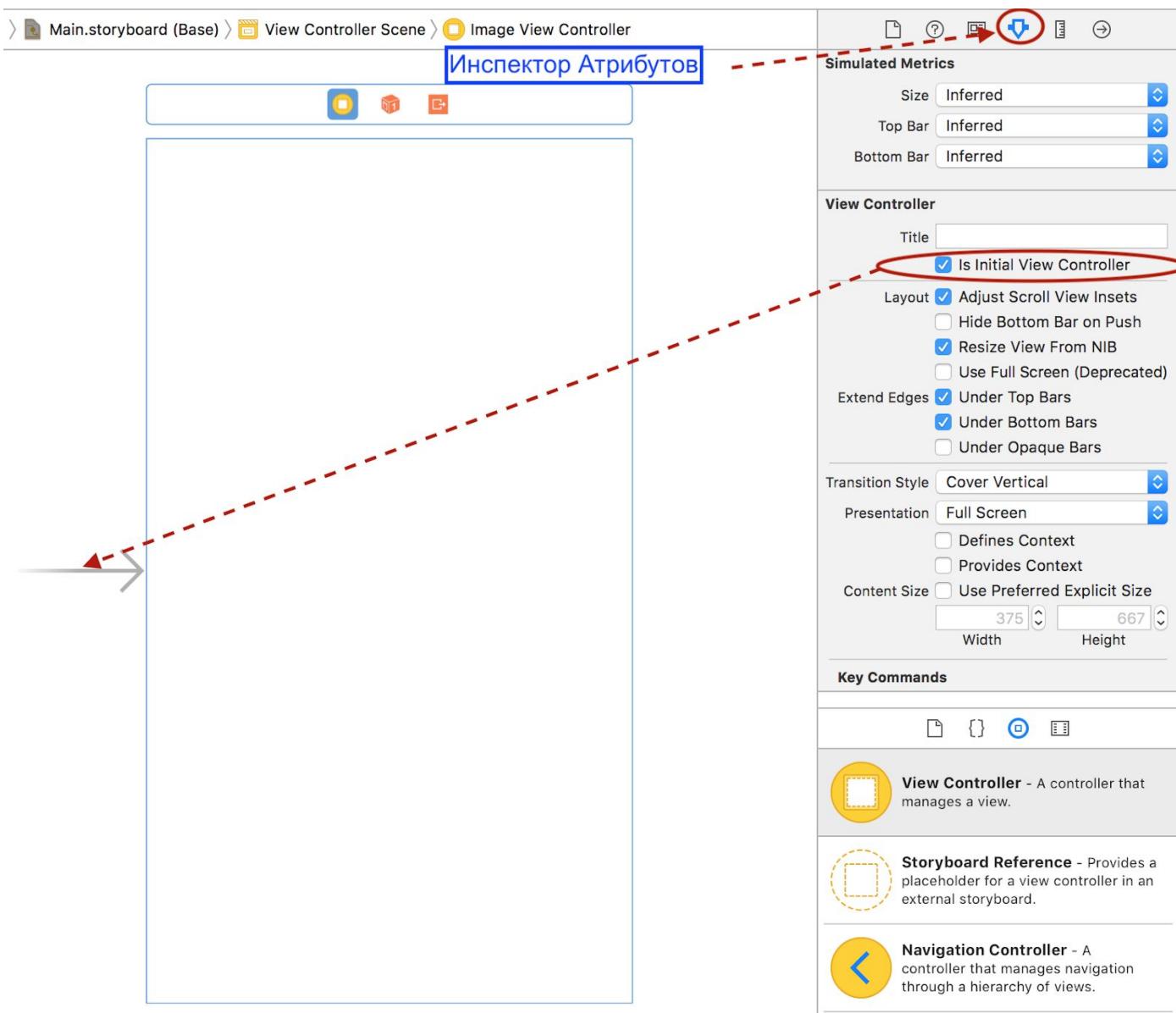
Я не сделал еще одну важную вещь, что и вызвало это предупреждение. Видите предупреждение вверху?



Нам говорят, что **ImageViewController** нельзя достичь. Потому что нет маленькой стрелки, которая указывает точку входа. Помните? Маленькую стрелочку, которую я перемещал в демонстрационном примере со **SplitViewController**?

У нас вообще нет этой маленькой стрелочки. Как нам получить ее?

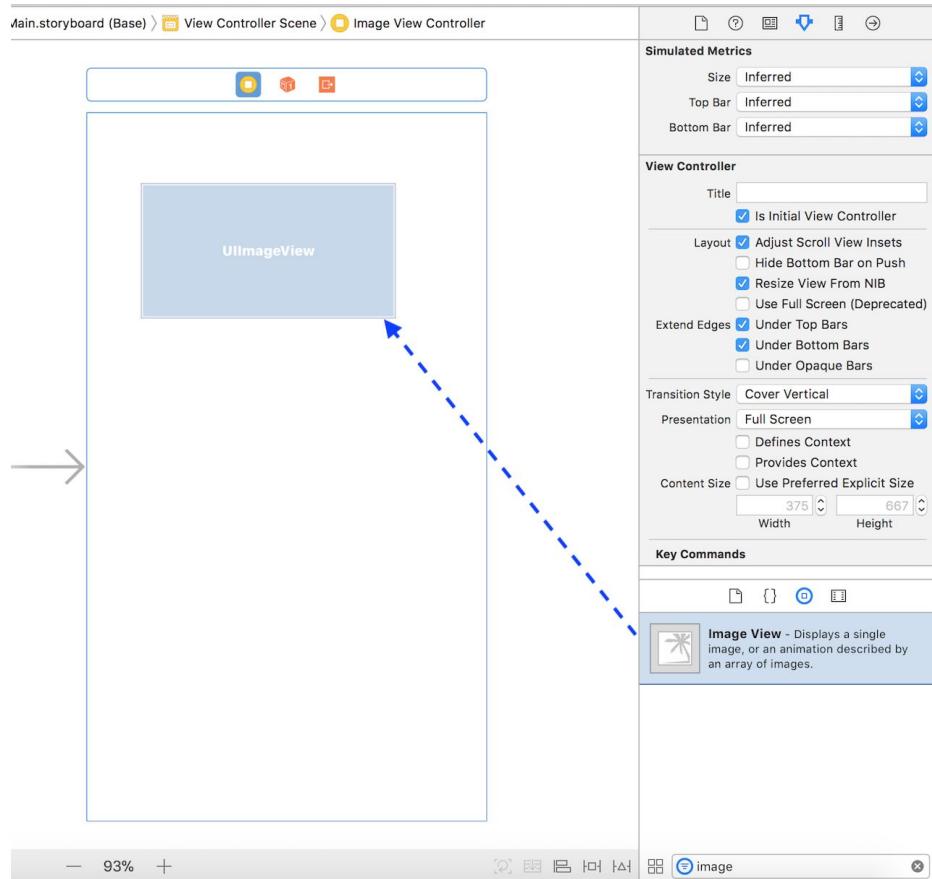
Для этого мне нужно проинспектировать мой **Image View Controller**, это обычный инспектор. И посмотрите! Здесь есть переключатель “**is initial View Controller**”, я его включаю, и - БУМ! - стрелочка появляется на нашем **Image View Controller**:



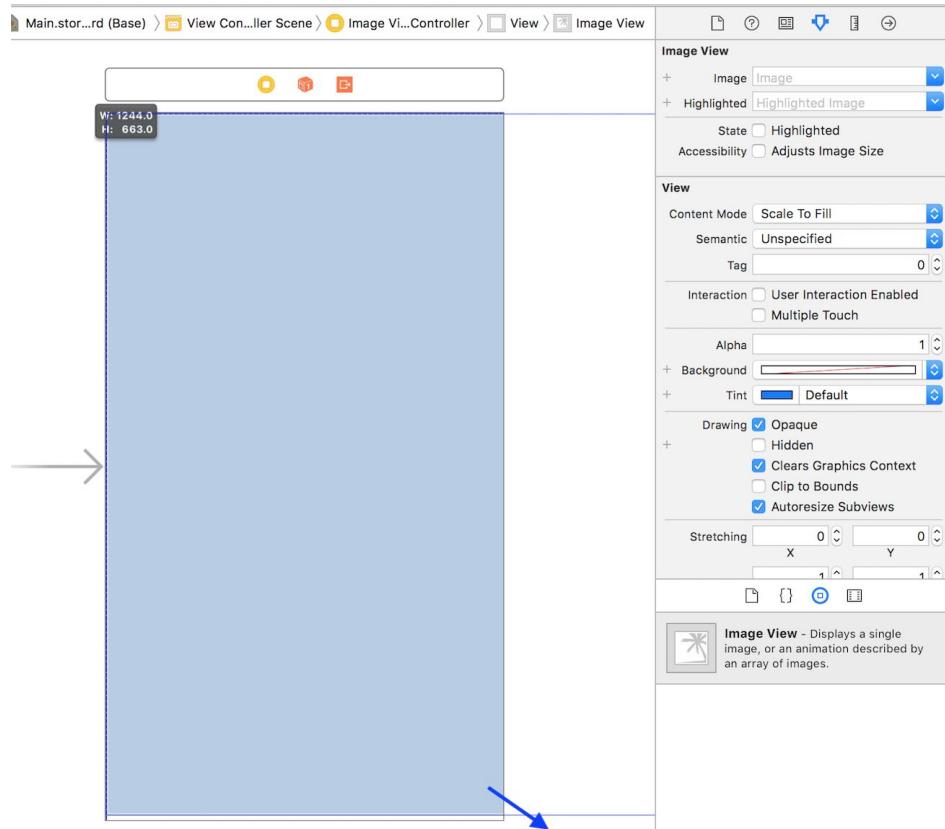
Давайте начнем с того, что у нас не будет **ScrollView**, у нас будет только **Image View**.

Мы вытягиваем **Image View** из Палитры Объектов. **Image View** напоминает метку **UILabel**, только

вместо текста в **Image View** можно разместить изображение (**image**).

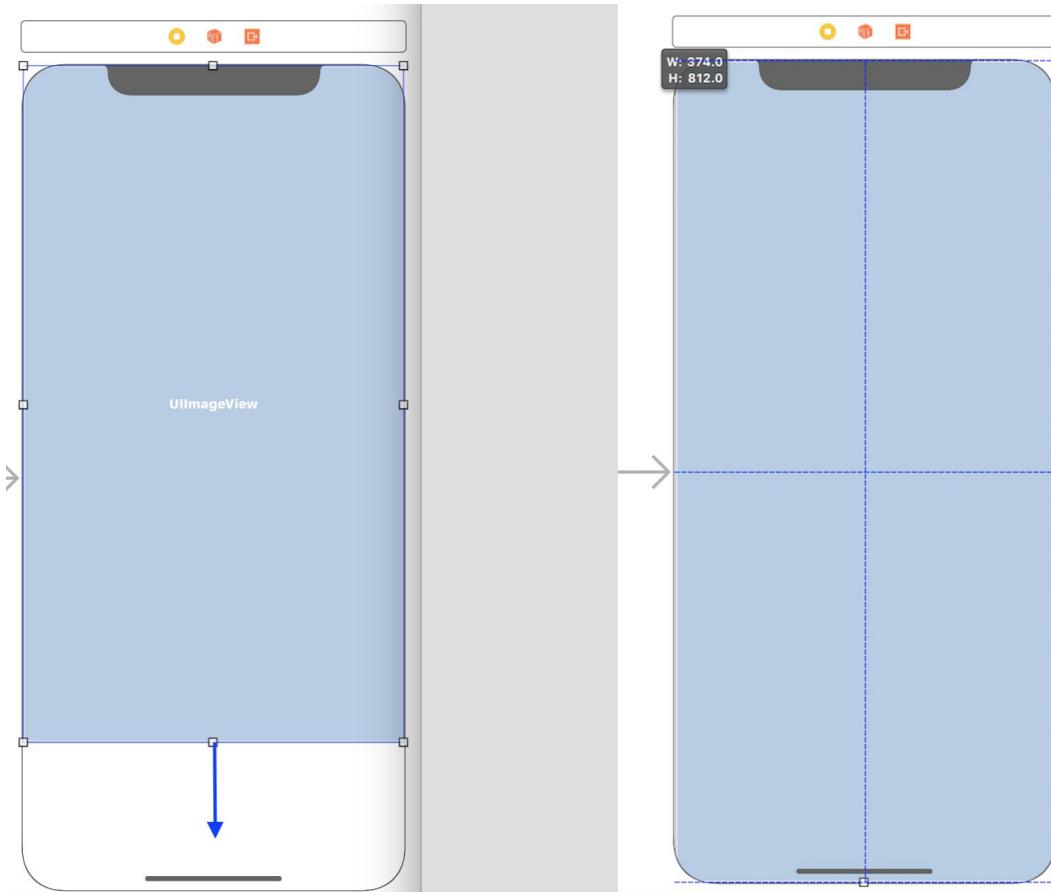


Мы разместим **Image View** по голубым линиям на полный экран от края до края :



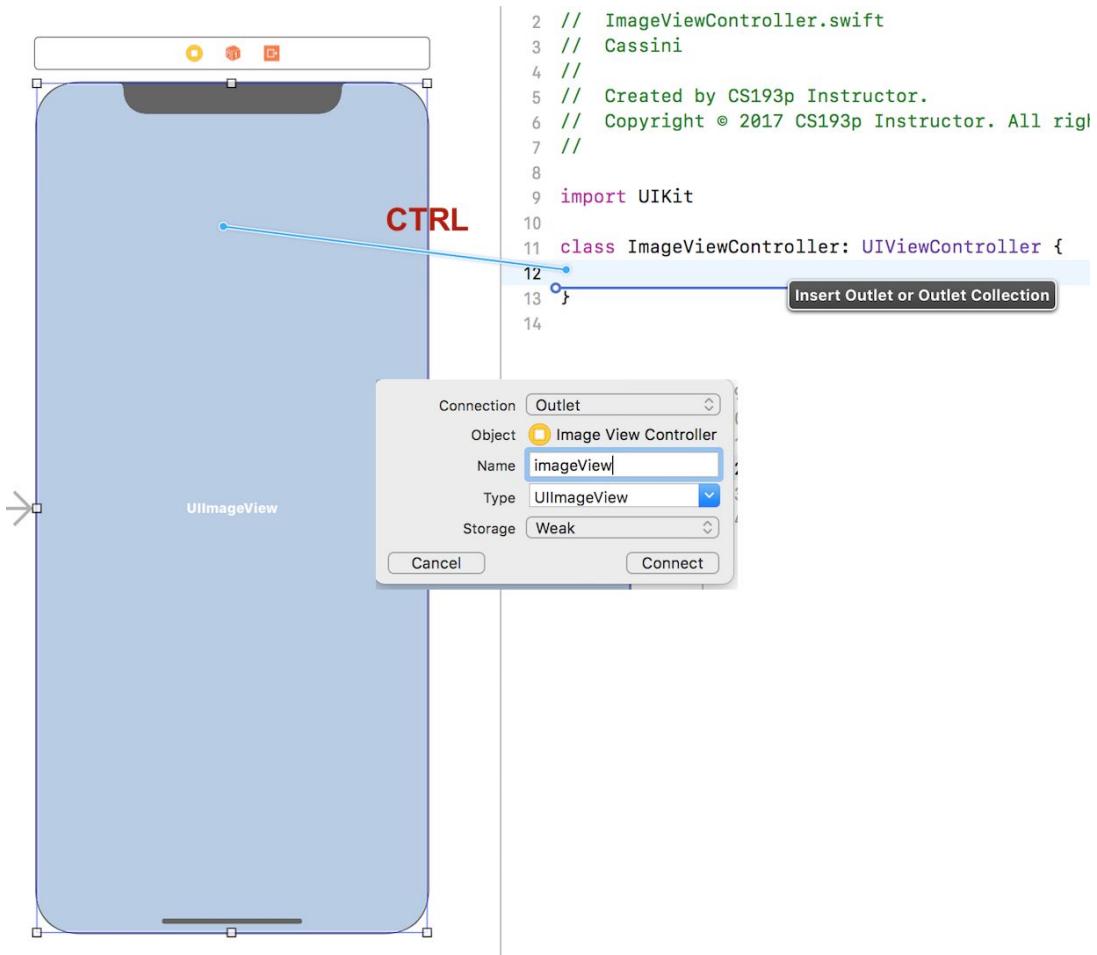
Давайте посмотрим, как это выглядит на **iPhone X**, это интересно, потому что там у нас есть вопросы, связанные с областью безопасности **Safe Area**.

Уменьшим немного масштаб, и я изменю размер **Image View** так, чтобы он покрывал весь экран, а не только **Safe Area**.



Видите черные участки, которые не входят в **Safe Area**?

Пока я здесь, я хочу создать **Outlet** к этому **Image View** в классе **ImageViewController** с помощью **CTRL**-перетягивания, чтобы с ним “разговаривать”:



Я назову этот Outlet **imageView**:

The screenshot shows the Xcode interface. On the left is the storyboard, which contains a single **UIImageView** element. On the right is the **ImageViewController.swift** code file. The code is as follows:

```
1 //  
2 //  ImageViewController.swift  
3 //  Cassini  
4 //  
5 //  Created by CS193p Instructor.  
6 //  Copyright © 2017 CS193p Instructor. All rights  
7 //  
8  
9 import UIKit  
10  
11 class ImageViewController: UIViewController {  
12     @IBOutlet weak var imageView: UIImageView!  
13 }  
14  
15
```

The line `@IBOutlet weak var imageView: UIImageView!` is highlighted with a red rectangle.

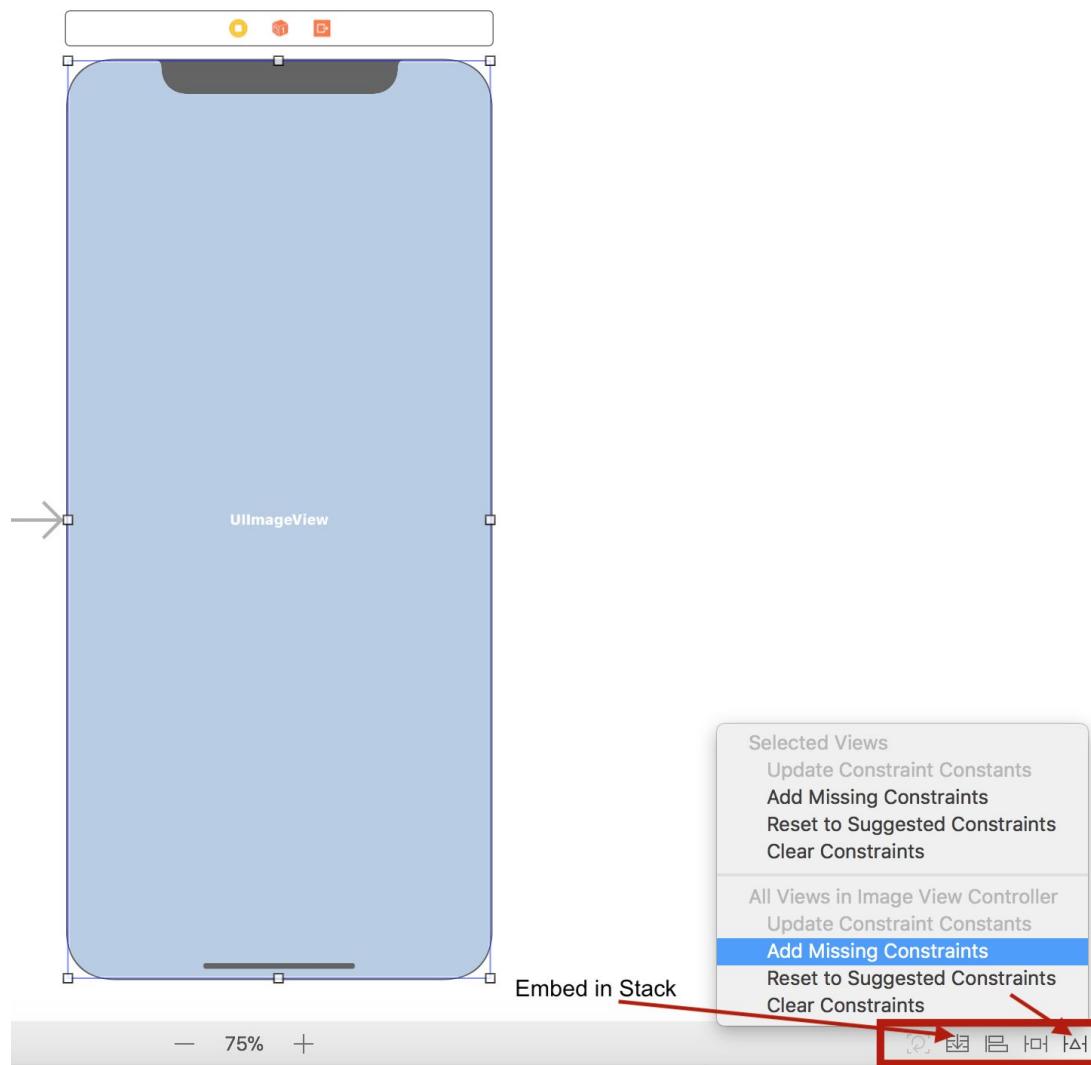
Я хочу, чтобы это большое изображение занимало весь экран до самых краев. Я не обращаю никакого внимания на **Safe Area**. У нас будет **ScrollView** и позже он нам поможет.

Это выглядит замечательно. Я совсем не против, что **Image View** покрывает эти черные области вверху и внизу. Меня все устраивает, я думаю, что мое изображение будет выглядеть очень круто. Как я могу сделать CTRL-перетягивание к краям?



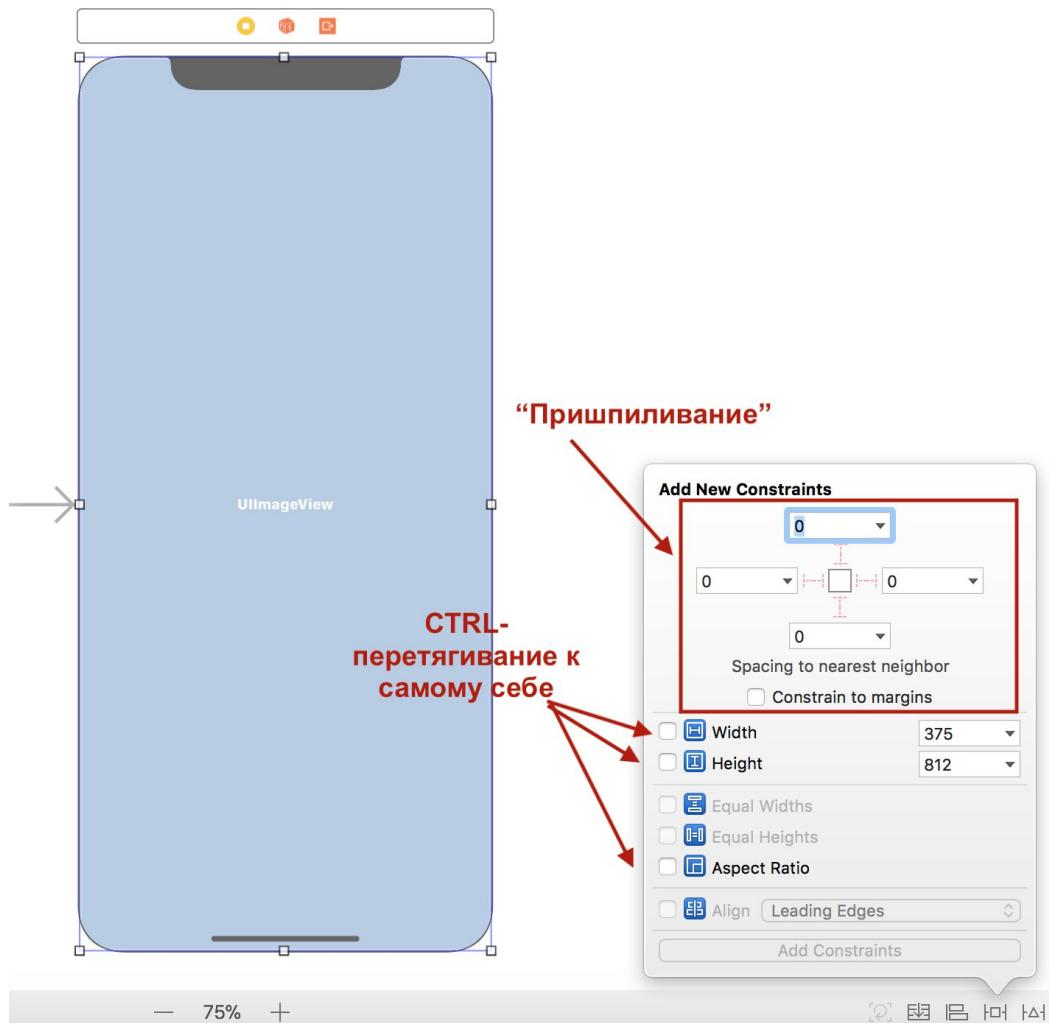
Я не могу достичь краев (**edges**), потому что мой **Image View** как раз располагается по краям, и я не могу их выбрать.

И опять, я покажу вам что-то новое в использовании механизма **Autolayout**, каждый раз по чуть-чуть. Мы начнем использовать кнопки, расположенные в правом нижнем углу и предназначенные для работы с механизмом **Autolayout**:



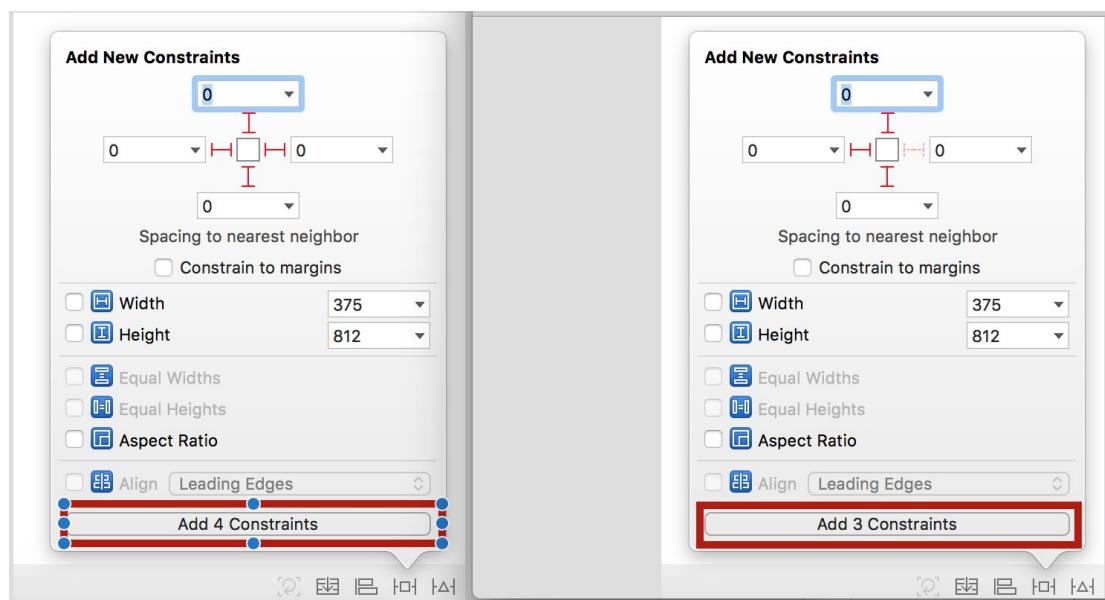
----- 45-ая минута лекции -----

Мы уже знаем одну кнопку - 4-ую справа - **Embed in Stack**, это вставка в **Stack View**, но мы не будем ее использовать. Мы уже видели 1-ую кнопку, которая предоставляет возможности убрать все ограничения системы **Autolayout** (**Clear Constraints**) или установить ограничения по “голубым линиям” (**Reset to Suggested Constraints**) и т.д. Но мы будем использовать другую кнопку, 2-ую справа. При наведении на нее “мыши” высвечивается название “**Add New Constraints**” (добавление новых ограничений). Смотрите, что произойдет, если я выберу мой **Image View** и кликну на ней, мне будет предложено установить некоторые ограничения для него:



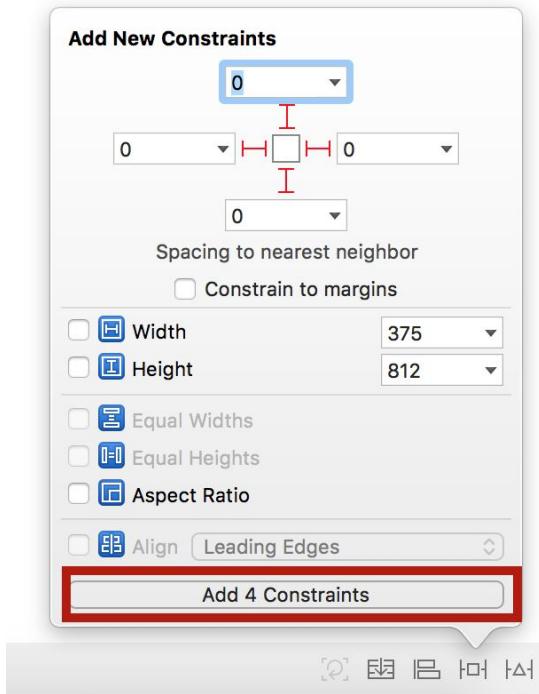
Некоторые ограничения - **Width**, **Height** и **Aspect Ratio** мы уже знаем и можем также задать с помощью **CTRL**-перетягивания к самому себе.

Но посмотрите на самую верхнюю часть. Это, по существу, “пришпиливание” к краям. Там имеются маленькие “лучики”, их включение будет добавлять ограничения. Видите, если я выберу все 4 “лучика”, то добавится 4 ограничения, если только 3 “лучика”, то добавится 3 ограничения.

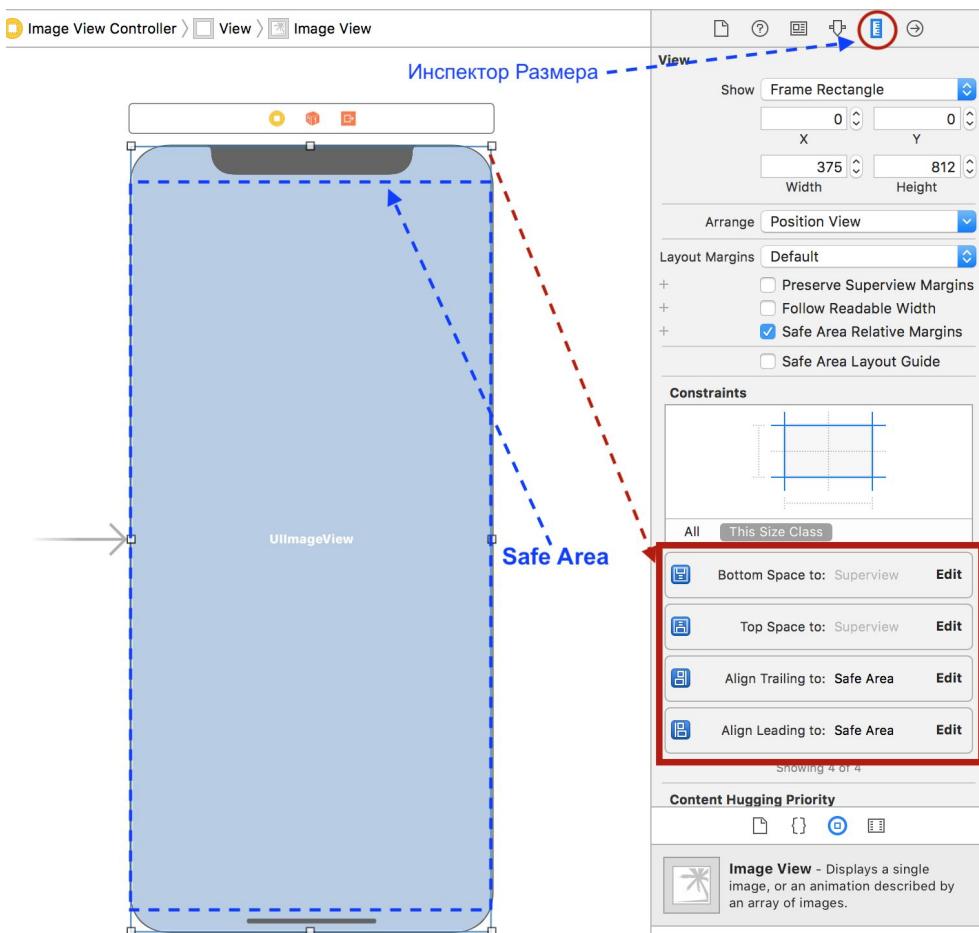


Я добавляю ограничения (**constraints**), которые “пришпиливают” выделенный мой **Image View** к тому, что называется “ближайший сосед” (**nearest neighbor**).

Вы видите “ближайшего соседа”? Давайте посмотрим на края и определим, что является “ближайшим соседом”? Может быть это края другого “ближайшего” **view**, может быть это края вашего **superview**, может быть это **Safe Area**? Давайте посмотрим, что произойдет, когда мы добавим эти 4 ограничения (**constraints**) к нашему **Image View**. Мы таким образом узнаем, кто является “ближайшим соседом” к нашим 4-м краям. Кликаем на кнопке “**Add 4 constraints**”:



Я иду в Инспектор Размера (**Size Inspector**) и смотрю на полученные ограничения (**constraints**):



Смотрите: для Верхней границы (**Top**) и Нижней границы (**Bottom**) “ближайшим соседом” является

Superview, то есть **self.view**, топовый View моего Controller. Но по бокам, справа и слева, - это **Safe Area**. Потому что на iPhone X **Safe Area** проходит чуть ниже черной области вверху, затем - вниз, затем чуть выше и не включая маленькую черную область внизу и затем вверх. Это и есть **Safe Area**.

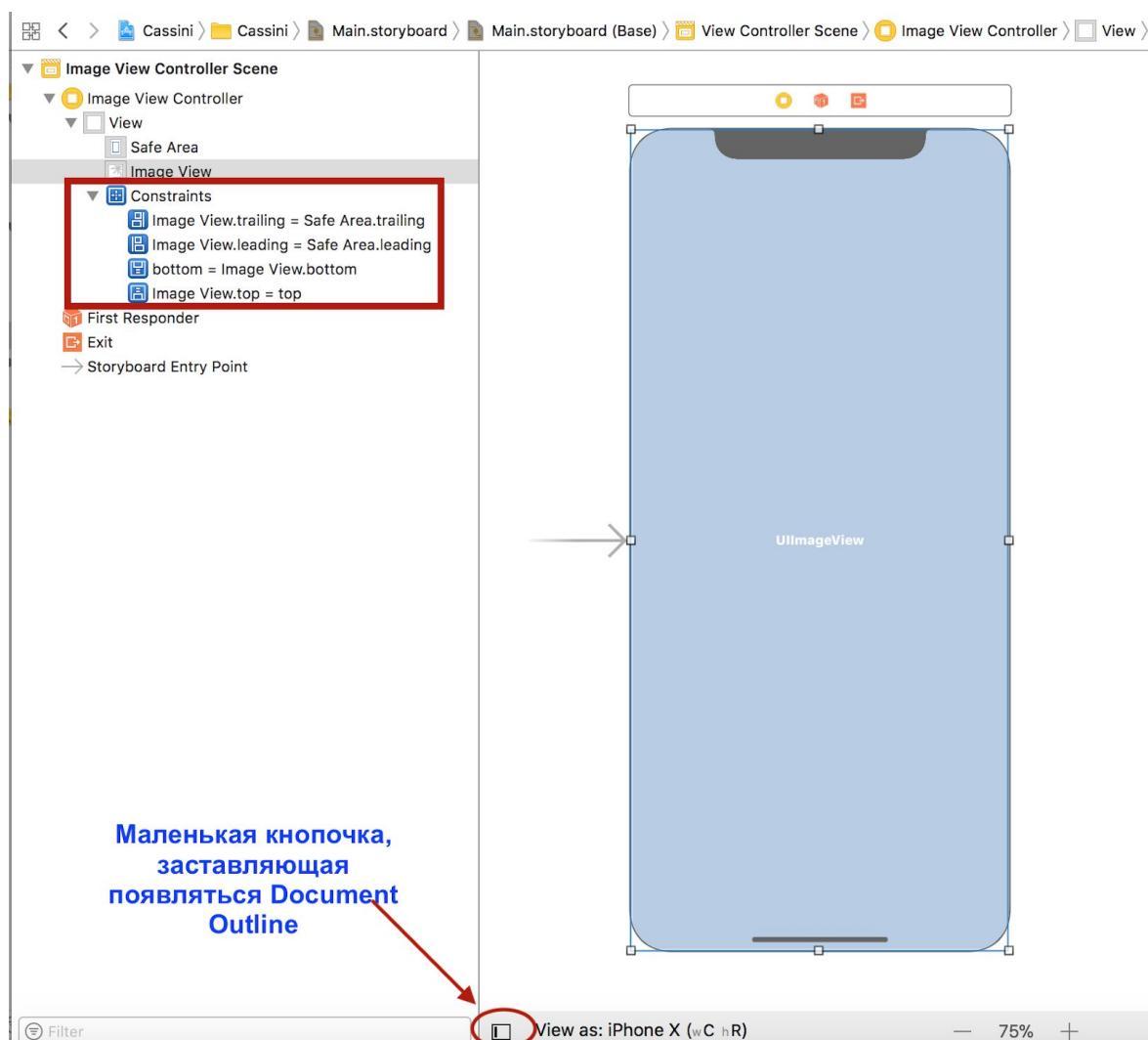
Очевидно, что оба края также близки к **Safe Area**, как и к **Superview**, но **Interface Builder** ВСЕГДА предпочитает края **Safe Area** краям **Superview**, если это одно и то же. Но вверху это не так, потому что Верхняя граница нашего **Image View** далека от Верхней границы **Safe Area**, она находится точно на Верхней границе **Superview**. Поэтому в ограничениях присутствует **Superview**. Видите в Инспекторе Размера?

Я не хочу, чтобы у меня в ограничениях фигурировала **Safe Area**, я хочу, чтобы это было на краях **Superview**. Как мне это исправить, как избавиться от **Safe Area** в ограничениях?

Мне нужно редактировать ограничения (**constraint**), но я не могу их даже выбрать, я их не вижу.

Я представлю вам способ, с помощью которого вы всегда сможете иметь доступ к ограничениям (**constraints**), и это **Схема UI (Document Outline)**.

Помните эту маленькую кнопочку внизу, о которой я говорил на прошлой Лекции при показе демонстрационного примера? Я говорил вам, что пока не стоит беспокоиться об этой кнопочке, но сейчас пришло время беспокоиться, потому что все, что показывается в моем View на storyboard, показывается в виде схемы, включая ограничения (**constraints**), в **Схеме UI (Document Outline)**:

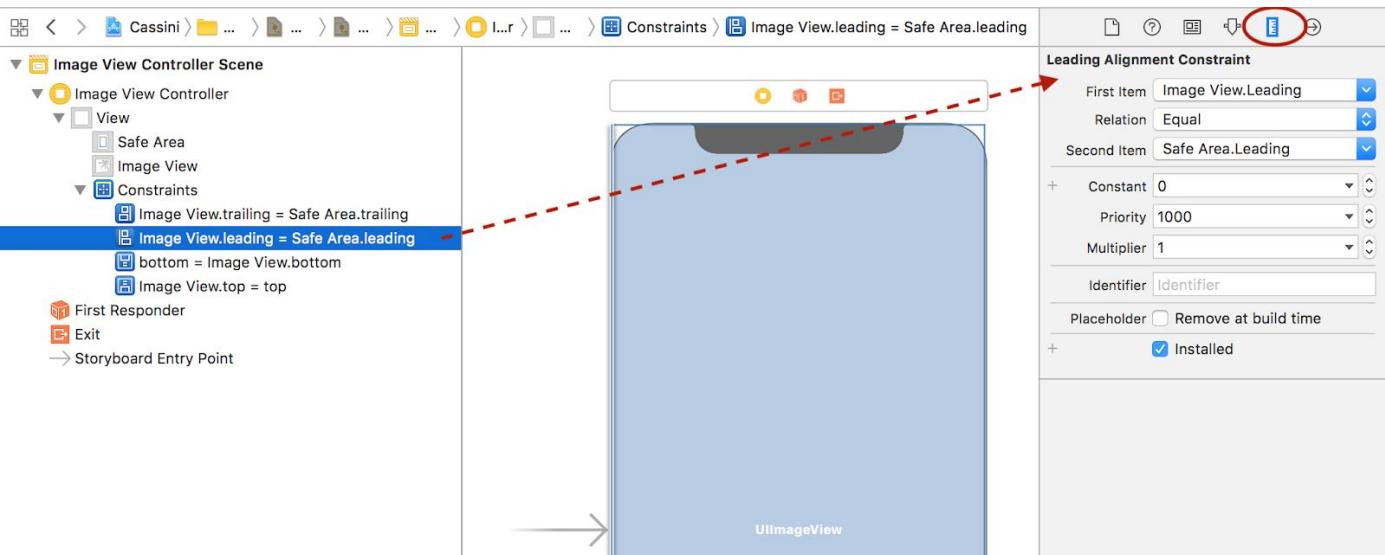


Видите? Ограничения выглядят очень знакомо. Это ограничения между моим **Image View** и

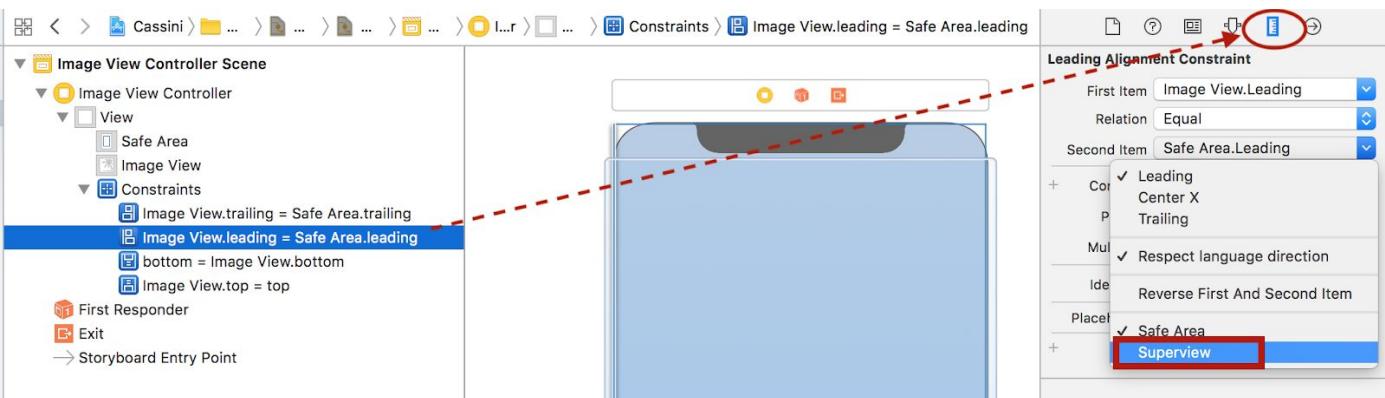
Superview. Два из этих ограничений “подцеплены” к **Safe Area**, а два других - к **Superview**.

Давайте кликнем на одном из ограничений и перейдем в нормальный Инспектор, Инспектор

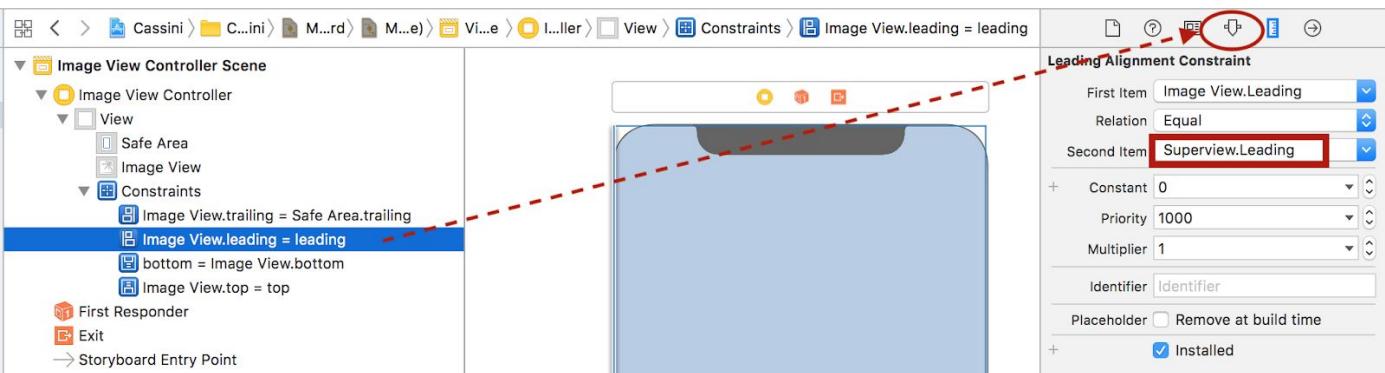
Размера, и в нем мы сможем редактировать это ограничение. Мы уже видели в предыдущих демонстрационных примерах, как нужно редактировать эти ограничения на примере **Multiplier**:



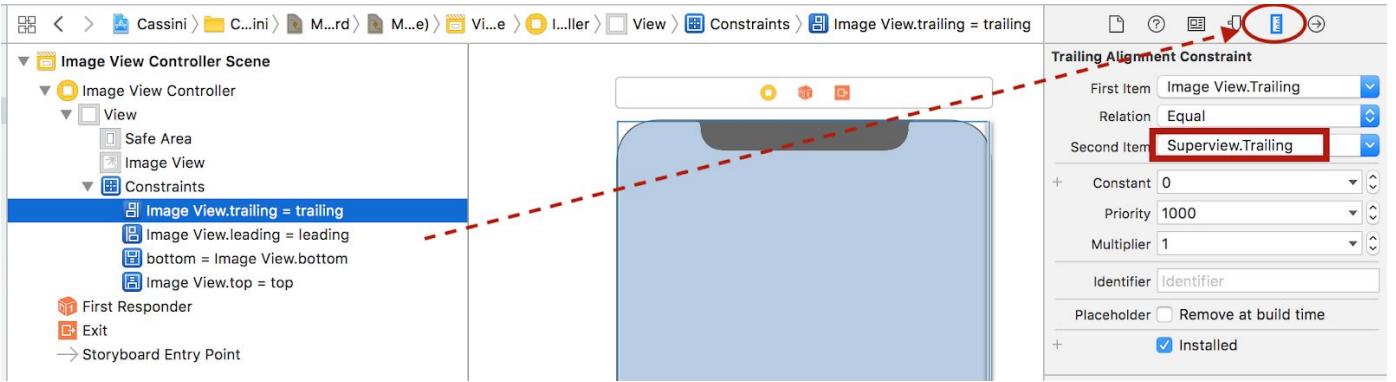
Сейчас мы будем редактировать взаимосвязь ограничения. Это ограничение направлено от (**From**) лидирующего края (**Leading**) **Image View** к лидирующему краю (**Leading**) **Safe Area** и установлено в **Equal**, то есть оно хочет быть тем же самым, но я не хочу, чтобы это был лидирующий край (**Leading**) **Safe Area**, я хочу, чтобы это был лидирующий край (**Leading**) **Superview**. Мы можем изменить **Safe Area** на **Superview**, просто редактируя ограничение (**constraint**) и изменения **Second Item**, к которому подсоединенено это ограничение:



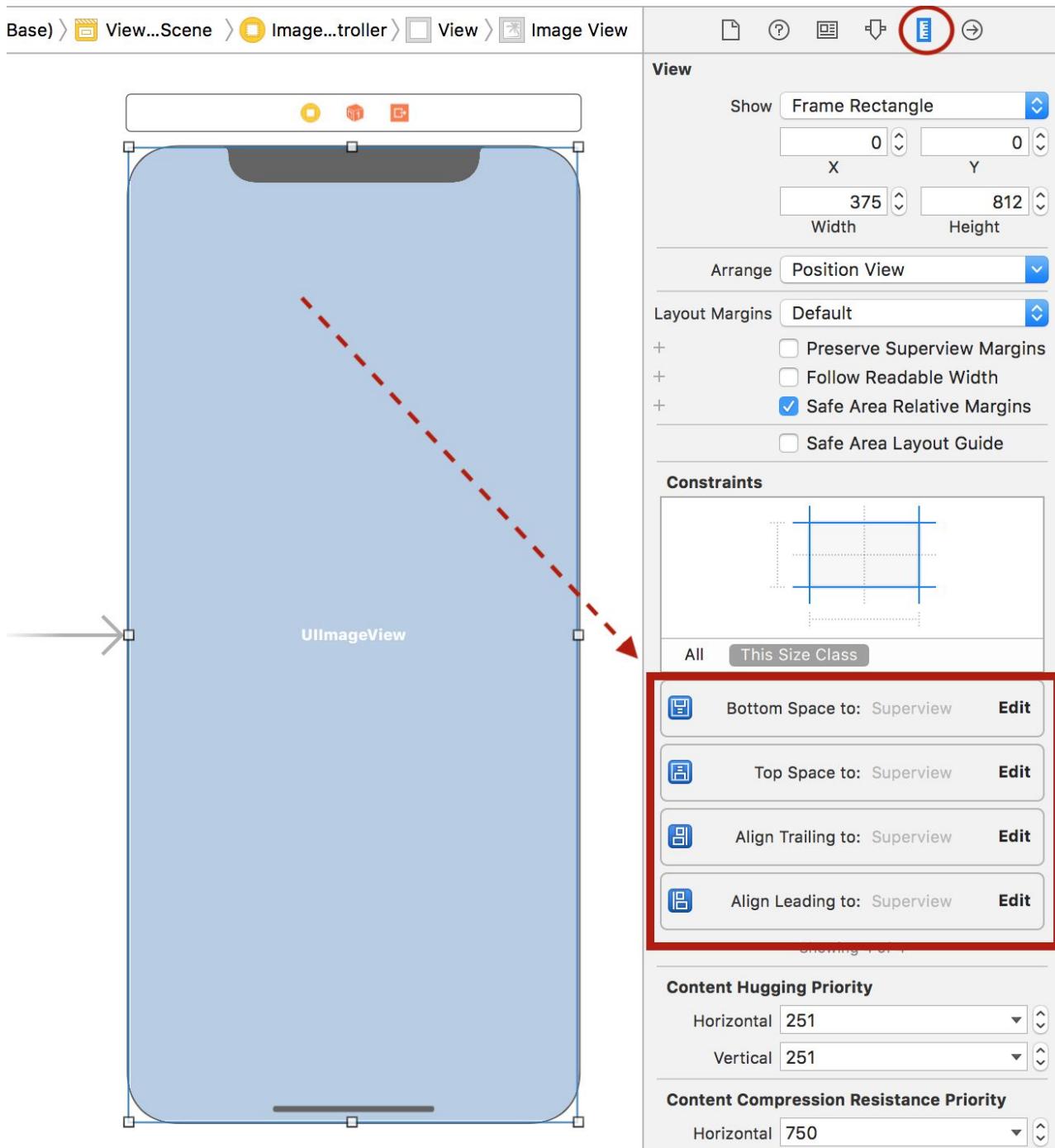
В результате получаем:



То же самое делаем с “хвостовым” (**Trailing**) ограничением: меняем **Safe Area** на **Superview**:



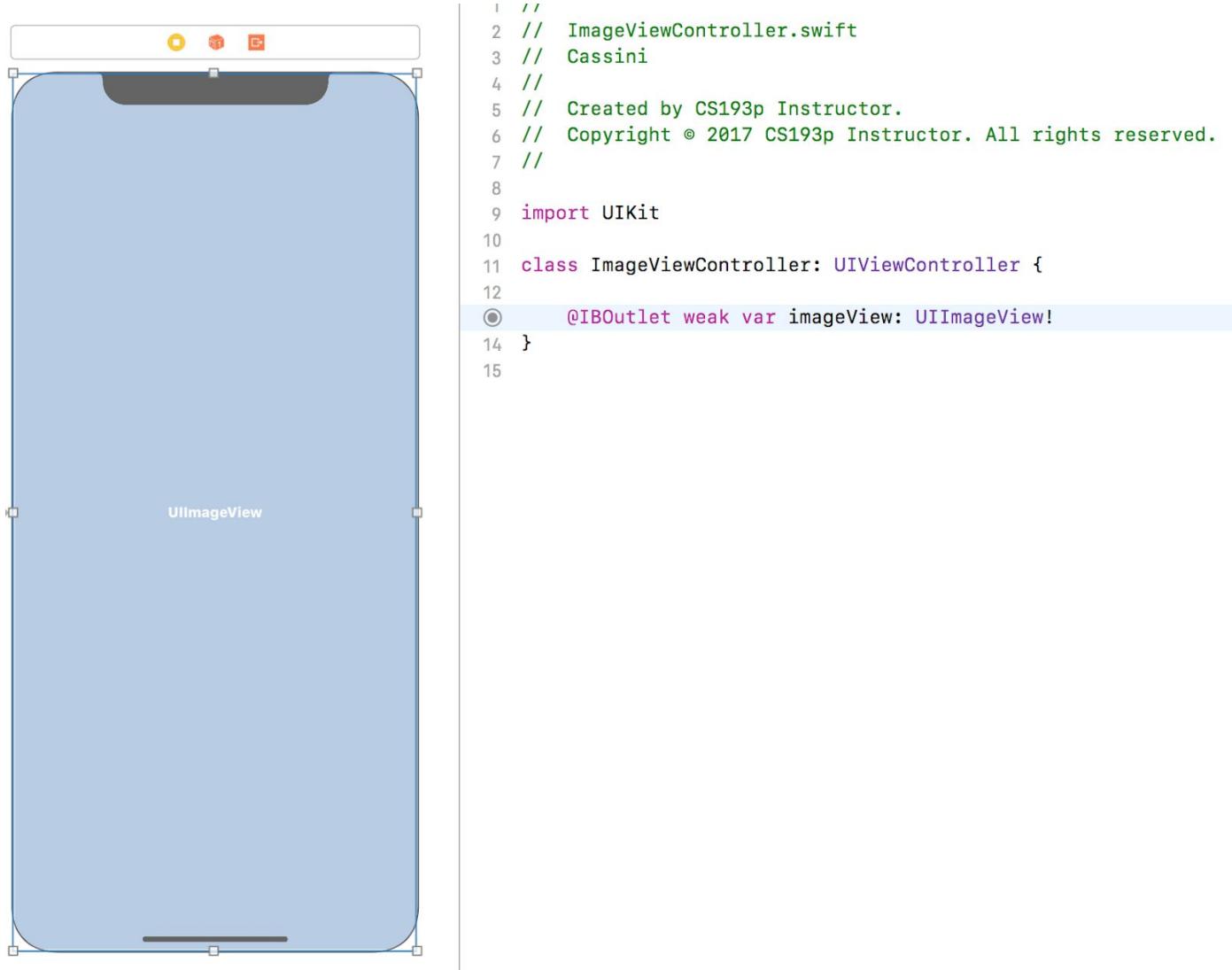
Теперь мой View заполняет все пространство, не беспокоясь ни о чем, ни о панели с заголовком в Navigation Controller, ни о закладках TabBarController:



Теперь Image View будет заполнять весь экран и находиться позади этих вещей, которые не

входят в **Safe Area** и на которых вы не можете рисовать. Но в данном случае я хочу именно этого. Через мгновение мы увидим еще некоторые возможности **Схемы UI (Document Outline)**, но в любом случае, это прекрасный способ доступа ко всему, что находится в моем View. Таким способом очень легко получить доступ ко всему.

Идем в наш код, и напишем некоторый код, который заставил бы этот **ImageViewController** выполнить свою работу.



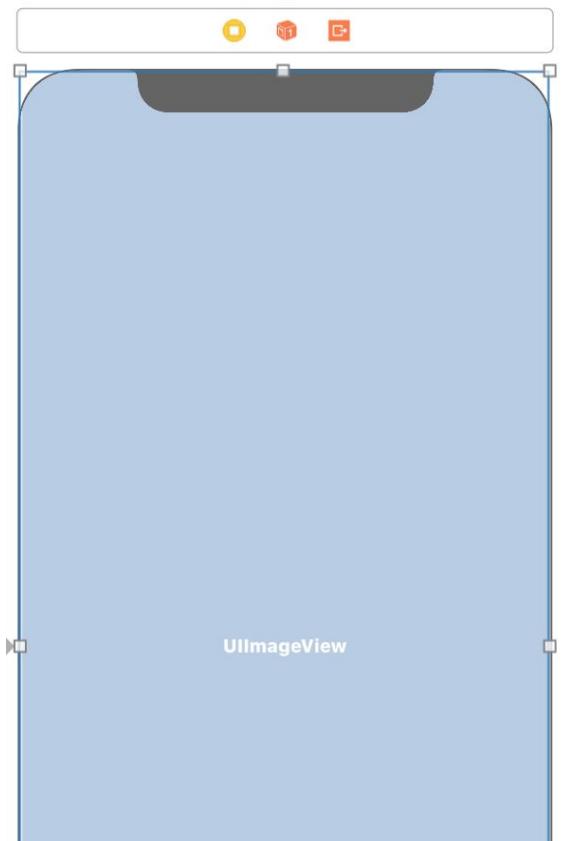
```
1 //  
2 //  ImageViewController.swift  
3 //  Cassini  
4 //  
5 //  Created by CS193p Instructor.  
6 //  Copyright © 2017 CS193p Instructor. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class ImageViewController: UIViewController {  
12  
13     @IBOutlet weak var imageView: UIImageView!  
14 }  
15
```

Итак, это **Controller MVC**.

Какая **Модель** у этого **MVC**?

----- 50-ая минута лекции -----

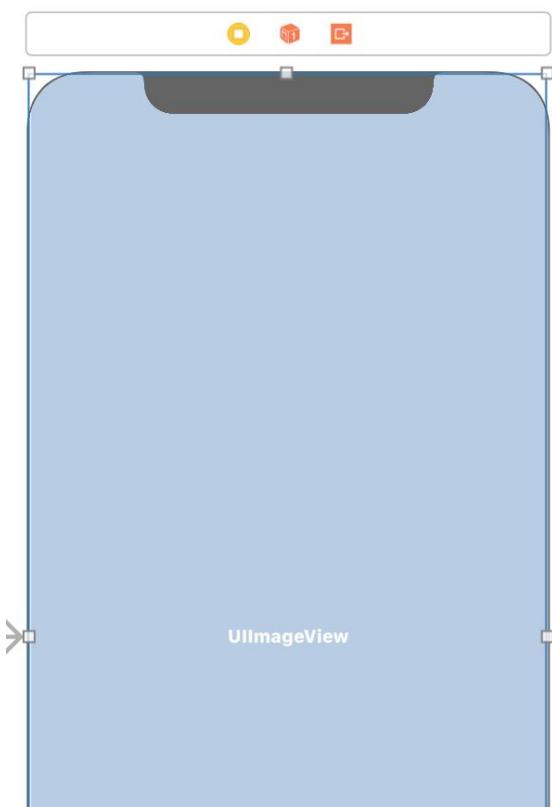
Моей **Моделью** будет переменная **imageURL**, имеющая ТИП класса с именем **URL**. Этот класс представляет **URL**, это может быть **URL** локального файла, это может быть **URL** в интернете:



```
2 // ImageViewController.swift
3 // Cassini
4 //
5 // Created by CS193p Instructor.
6 // Copyright © 2017 CS193p Instructor. All rights reserved.
7 //
8
9 import UIKit
10
11 class ImageViewController: UIViewController {
12
13     var imageURL: URL?
14
15     @IBOutlet weak var imageView: UIImageView!
16 }
17
```

Это моя **Модель**. Моя работа состоит в том, чтобы взять эту **Модель**, превратить ее в изображение и представить его в моем **View** с помощью **Image View**.

Если кто-то установит мою **Модель** из-вне, то я должен установить переменную **image** моего **Outlet** **imageView** в **nil**, потому что я должен заменить ее новым изображением, а затем выбрать новое изображение с помощью функции **fetchImage()**:



```
2 // ImageViewController.swift
3 // Cassini
4 //
5 // Created by CS193p Instructor.
6 // Copyright © 2017 CS193p Instructor. All rights reserved.
7 //
8
9 import UIKit
10
11 class ImageViewController: UIViewController {
12
13     var imageURL: URL? {
14         didSet {
15             imageView.image = nil
16             fetchImage()
17         }
18     }
19
20     @IBOutlet weak var imageView: UIImageView!
21
22     private func fetchImage() {
23
24     }
25 }
```

Причина, по которой мы сделали эту отдельную **private** функцию **func fetchImage()**, состоит в том,

что будет выборка через интернет. В Среду мы собираемся запустить выборку изображения из интернета как фоновую задачу, потому что выборка из интернета может заблокировать наш **UI**. Я смотрю немного вперед, и поэтому создал для выборки изображения отдельную функцию **fetchImage ()**, а не “бросил” код прямо в Наблюдателе **didSet{ }**.

Таким образом, если кто-то установит новый **imageURL**, мы очистим изображение **image**, которое у нас было, если вообще было, **image** - это просто переменная **var** в нашем **imageView**, которая содержит изображение. Затем мы выбираем новое изображение.

Я хочу напомнить одну вещь, которую я говорил вам при изучении “жизненного цикла” **View Controller**. Я сказал вам, что вы не должны делать что-то затратное наподобие этой выборки из интернета, до тех пор, пока это будет абсолютно необходимо. Я буду выбирать изображение в Наблюдателе **didSet{ }** только, если кто-то установит мою **Модель** и если я нахожусь на экране. Если кто-то установит мою **Модель**, а меня на экране нет, то я пока не буду делать выборку из интернета, я буду ждать до тех пор, пока я появлюсь на экране.

Как вы можете выразить в коде, что вы находитесь на экране?

Как вам узнать, что ваш **MVC** находится на экране?

Есть очень крутой способ это сделать. Я просто проверяю есть ли у моего **view** переменная **window**, которая не равна **nil**:

```
var imageURL: URL? {
    didSet {
        imageView.image = nil
        if view.window != nil {
            fetchImage()
        }
    }
}
```

Мы не говорили о переменной **window**, и я говорил вам в самом начале нашего курса, что нужно просто ее игнорировать, и в большинстве случаев вы так можете поступать и дальше, за исключением того случая, когда мы хотим узнать, находится ли наш **view** на экране, в этом случае мы используем переменную **var window**, которая является “окном”, в котором находится наш **view** и это обычно одно “окно”. Это замечательный и очень простой способ с помощью переменной **window** установить, находитесь ли вы на экране.

Если я нахожусь на экране, то я буду делать выборку изображения с помощью **fetchImage ()**.

А если я НЕ на экране?

В итоге, мне все равно необходимо выбрать изображение. Какое время является лучшим для этой операции? У кого-нибудь есть идея?

Да, **viewDidAppear**. Точно, давайте сделаем это в **viewDidAppear**.

Выполняем **super.viewDidAppear**.

В этой точке я знаю, что я уже на экране и внутри **viewDidAppear** я могу проверить наличие изображения в **imageView.image**. Если это изображение все еще остается равным **nil**, то я

выбираю его с помощью `fetchImage ()`:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    if imageView.image == nil {  
        fetchImage()  
    }  
}
```

Видите, как я использовал “жизненный цикл” **View Controller** для того, чтобы не делать ненужную работу?

Если выборка изображения `fetchImage ()` действительно работает в фоновом (**background**) режиме, то мне нужно предоставить пользователю некоторый индикатор, чтобы он видел, что я работаю над этим, потому что иначе пользователь скажет: “А где же мое изображение? Перейдя на этот **View Controller**, я не вижу никакого изображения здесь.” Поэтому я вынужден предоставить пользователю некоторую обратную связь.

Но так как в данный момент я блокирую пользовательский интерфейс (**UI**) до тех пор, пока не будет получено изображение из-за отсутствия многопоточности, я оставлю пока все как есть. Но мы увидим это позже, когда будем заниматься многопоточностью.

Итак, функция `fetchImage ()`, как мы будем делать выборку изображения?

Это действительно достаточно легко.

Мы просто смотрим не равен ли наш **URL url**, наша **Модель imageURL**, `nil`, потому что если он равен `nil`, мы ничего не сможем делать.

Потому что если кто-то установил новое изображение и установил нашу **Модель imageURL** в `nil`, то мы начнем выбирать изображение в `didSet {}` или в `viewDidAppear`, где мы установили изображение `imageView.image` в `nil`, если `imageURL == nil`, то мы получим пустое изображение, ничего не будет находиться в `imageView`.

Но если `imageURL` не равен `nil`, мы должны пойти в интернет, взять данные по этому **URL** и превратить их в изображение `image`:

```
private func fetchImage() {  
    if let url = imageURL { ⚠ Value 'url' was defined but never used; consider using _  
    }  
}
```

Как мы выходим в интернет и получаем данные из **URL**? Мы делаем это с помощью **Data** объектов. Помните? Я говорил вам о **Data** объекте, который представляет собой просто “сумку битов”. У него есть инициализатор, которому мы даем **URL**, а он возвращает нам “сумку битов” из этого **URL**. Для этого мы создадим локальную переменную, которую назовем `urlContents` и которая будет равна `Data(...)`, а дальше мы можем взглянуть на различные инициализаторы **Data**, у него их множество, так как множество различных ТИПОВ данных можно преобразовать в “сумку битов” **Data**:

```
Creates a data buffer with the contents of a URL.  
Data (buffer: UnsafeMutableBufferPointer<SourceType>)  
Data (bytes: Array<UInt8>)  
Data (bytes: ArraySlice<UInt8>)  
Data (bytes: UnsafeRawPointer, count: Int)  
Data (bytesNoCopy: UnsafeMutableRawPointer, count: Int, deallocator: Data.Deallocator)  
Data (capacity: Int)  
M Data (contentsOf: URL) throws  
M Data (contentsOf: URL, options: Data.ReadingOptions) throws  
33     let urlContents = Data( ⚠ Initialization of immutable value 'urlContents' was never used  
34  
35 } ✖ Expected expression in list context  
36 }
```

И одним из них **Data (contentOf: URL) throws** мы воспользуемся. Но что за ключевое слово находится в конце этого инициализатора?

Слово **throws**. Почему этот инициализатор может “выбрасывать” (**throws**) ошибки?

Этот метод “выходит” в интернет, и, соответственно, может быть множество причин, по которым он может “выбрасывать” **throws** ошибку: плохая связь, не отвечает сервер, там порядка 20 различных причин, которые могли бы “выбросить” ошибку.

----- 55-ая минута лекции -----

Если бы наш инициализатор никак не беспокоился об этих ошибках, то мы бы написали следующий код:

```
private func fetchImage() {  
    if let url = imageURL {  
        let urlContents = Data(contentsOf: url) 2⚠️✖ Call can throw  
    }  
}
```

Но так как наш инициализатор выбрасывает **throws** ошибки, то что мы должны написать перед этим инициализатором?

Правильно, ключевое слово **try**, потому что этот инициализатор может закончиться с ошибкой:

```
private func fetchImage() {  
    if let url = imageURL {  
        let urlContents = try Data(contentsOf: url) 2⚠️✖ Errors  
    }  
}
```

Мы могли бы “обернуть” инициализатор небольшим блоком **do {} catch let error {}** и обработать ошибки:

```

private func fetchImage() {
    if let url = imageURL {
        do {
            let urlContents = try Data(contentsOf: url) ⚠ Initialization
        } catch let error { ⚠ Immutable value 'error' was never used; consider
        }
    }
}

```

Это могут быть ошибки типа “Server time out”, “Network unavailable” и подобные им вещи. Но нам нет до них дела, это просто непрятательный **ImageViewController**, мы просто хотим разместить изображение на нашем **View**, если получится, и мы не будем беспокоиться об ошибках и использовать блок **do {} catch let error {}** для их обработки.

Вместо этого мы будем использовать эту локальную переменную **urlContents**, которая будет иметь ТИП **Data**, но я поменяю **try** на **try?**, а это означает, что инициализатор вернет **nil**, если произойдет ошибка. Теперь ТИП переменной **urlContents** будет **Data?**, то есть **Optional<Data>**, потому что наша попытка может привести к неудаче:

```

31     private func fetchImage() {
32         if let url = imageURL {
33             let urlContents = try? Data(contentsOf: url) ⚠ Initialization
Declaration let urlContents: Data?
Declared In ImageViewController.swift
37 }

```

Теперь я получил **urlContents** и могу написать код, который получит данные изображения **imageData** из этого **imageURL**. Затем из этих данных **imageData** я создам изображение. Данные **imageData** могут быть **JPEG** данными или чем-то подобным:

```

private func fetchImage() {
    if let url = imageURL {
        let urlContents = try? Data(contentsOf: url)
        if let imageData = urlContents {
            imageView.image = UIImage(data: imageData)
        }
    }
}

```

Изображение **imageView.image** я создаю с помощью инициализатора **UIImage**. Мы уже использовали инициализатор **UIImage (named:)**. Сейчас мы используем другой инициализатор - **UIImage (data:)**, мы даем ему данные изображения **imageData** ТИПА **Data**, он “заглядывает” в “сумку с битами” и распознает, выглядят ли они как JPEG файл или что-то похожее. Это здорово.

Примечание переводчика. Этот инициализатор для изображения **UIImage**, который берет, в

основном, “сырые” **JPEG** данные. Я получаю “сырые” **JPEG** или **PNG** данные в **imageData**, а затем из них создаю **UIImage**.

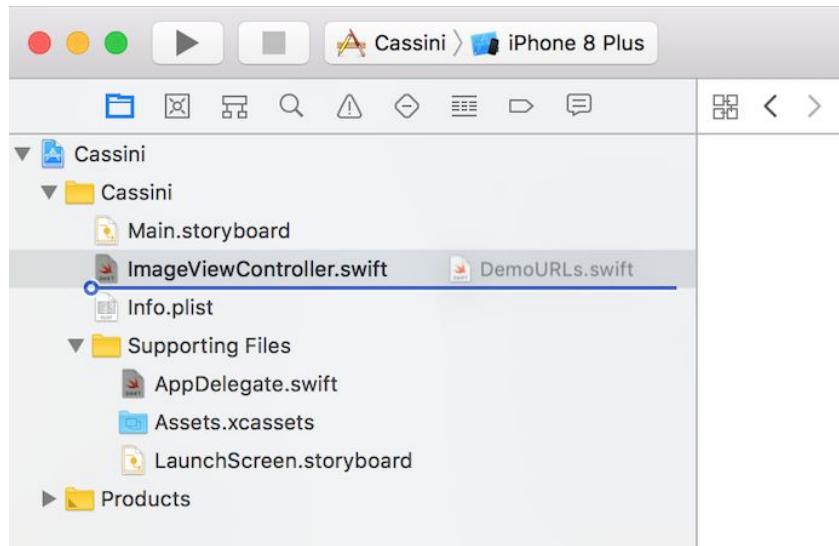
На этом все. Это все, что нам необходимо, чтобы показать здесь изображение.

Я сделаю еще одну вещь исключительно для демонстрационных целей в моем **viewDidLoad**.

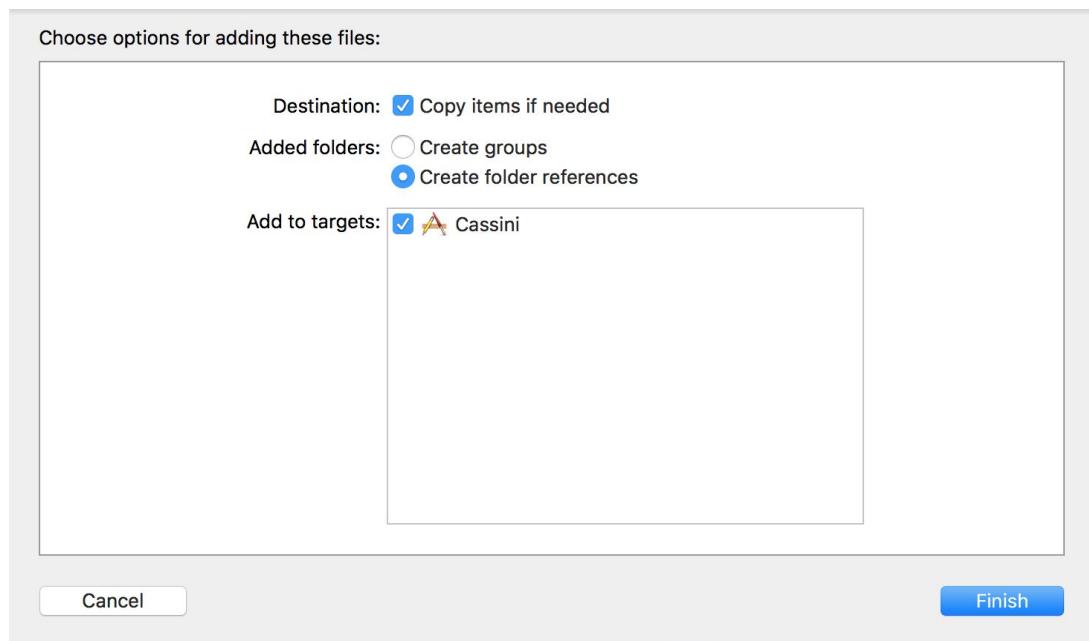
Я выполняю **super.viewDidLoad()**. А затем, если **imageURL** равен **nil** , я загружаю образец изображения (**sample image**):

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    if imageURL == nil {  
        imageURL =  
    }  
}
```

Для образца изображения (**sample image**) у нас будет специальный файл **demoURL.swift** с демонстрационными **URLs**, который я перетяну в свой проект:



Не забудьте опцию “**Copy items if needed**”:



Вот как выглядит этот [demoURL](#):

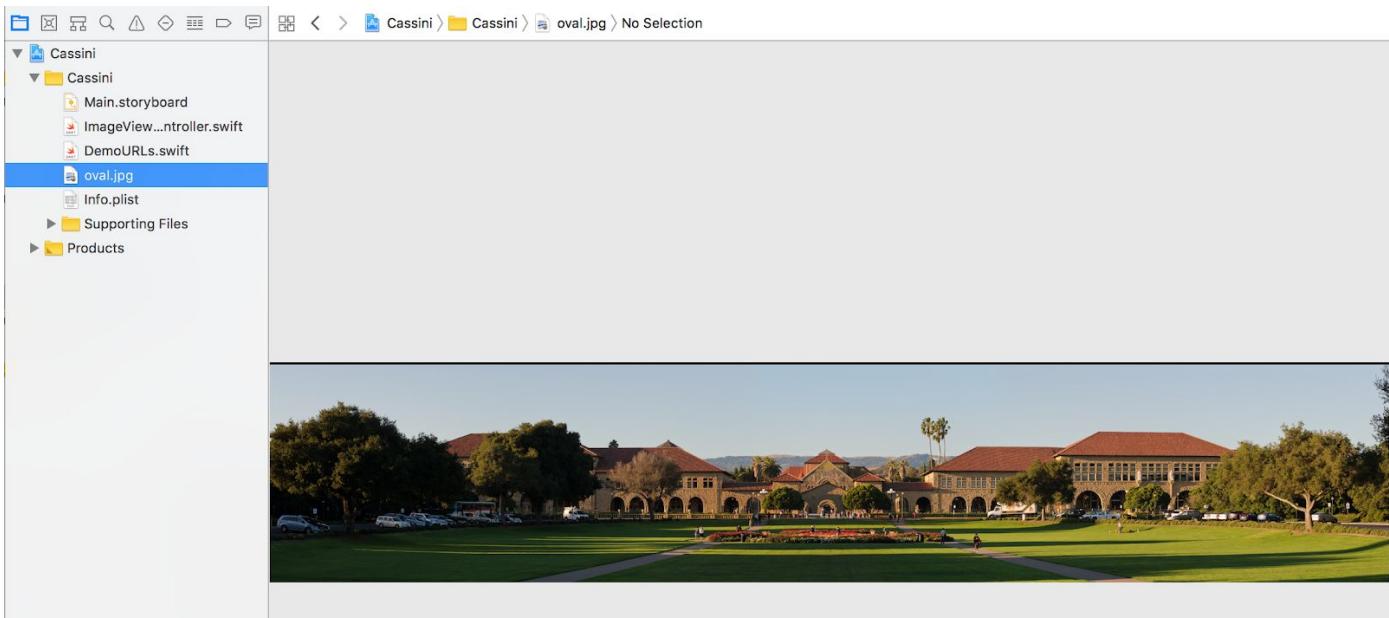
```
import Foundation

struct DemoURLs {
    static let stanford = Bundle.main.url(forResource: "oval", withExtension:"jpg")
    // static let stanford = URL(string: "https://upload.wikimedia.org/wikipedia/commons/5/55/Stanford_Oval_September_2013_panorama.jpg")

    static var NASA: Dictionary<String,URL> = {
        let NASAURLStrings = [
            "Cassini" : "https://www.jpl.nasa.gov/images/cassini/20090202/pia03883-full.jpg",
            "Earth" : "https://www.nasa.gov/sites/default/files/wave_earth_mosaic_3.jpg",
            "Saturn" : "https://www.nasa.gov/sites/default/files/saturn_collage.jpg"
        ]
        var urls = Dictionary<String,URL>()
        for (key, value) in NASAURLStrings {
            urls[key] = URL(string: value)
        }
        return urls
    }()
}
```

Это просто структура **struct** со **static** константами **let** и **static** переменными **var**. У меня есть **URL** изображения Стенфорда - **stanford**, это огромная фотография “Овала”, у меня также есть несколько изображений NASA, которые организованы в словарь **NASA** и которые мы будем показывать в демонстрационном примере в Среду.

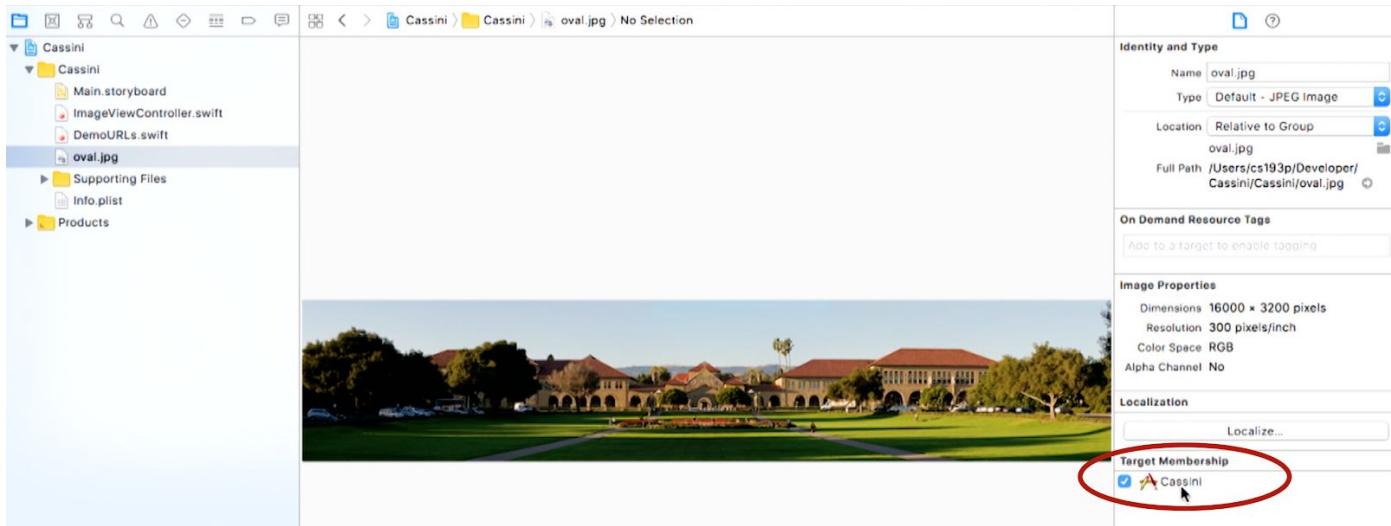
И опять, на моем лэптопе в данный момент нет интернета, так что я не могу считать из интернета изображение Стенфорда **stanford**, я вынужден заменить его локальным файлом, используя **Bundle.main.url** для получения локального файла. Я перетягиваю локальную версию “Овала” и просто копирую ее. Давайте посмотрим, как он выглядит:



Это очень большая фотография с очень высоким разрешением.

Я кликаю на кнопке **Target Membership**, чтобы убедиться, что, когда я буду устанавливать мое приложение на устройство, то этот файл должен быть включен, потому что я буду искать сейчас

локальную версию, но в будущем я получу этот файл из интернета. Просто так получилось, что я должен воспользоваться локальной версией, впрочем, сейчас это не имеет значения, так как пока у нас все равно нет многопоточности, так что все в порядке:

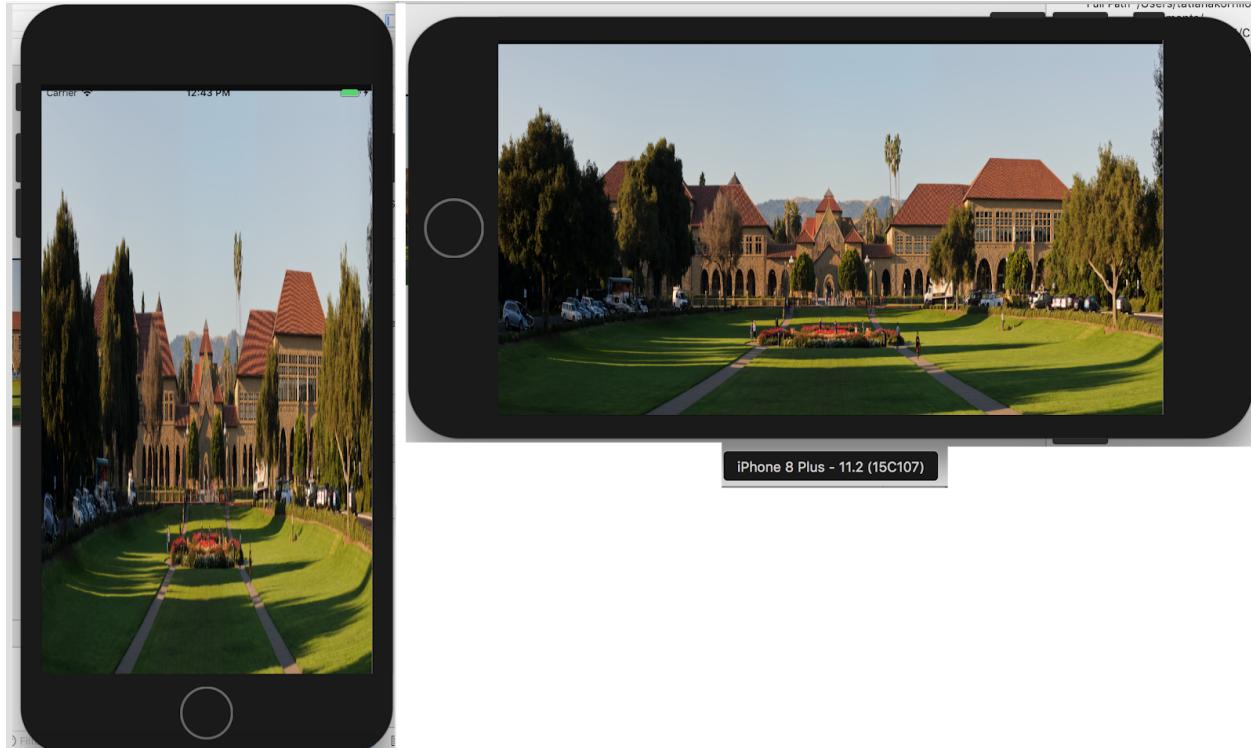


Я использую **stanford**, этот **URL** изображения Стэнфорда, в **viewDidLoad**, и там я устанавливаю мою Модель равной **DemoURLs.stanford**:

```
override func viewDidLoad() {
    super.viewDidLoad()
    if imageURL == nil {
        imageURL = DemoURLs.stanford
    }
}
```

Это просто **URL** для локального файла.

Давайте запустим приложение и посмотрим, что этот код сделал для нас:



В действительности я должен был заново запустить приложение на **iPhone X**. Но мы уже здесь, на **iPhone 8 Plus**.

Да, нам показывают изображение, но оно очень длинное и это делает его неправдоподобным. Если я переверну устройство в ландшафтный режим, то приложение - молодец, что заставляет изображение прилипнуть к краям **iPhone 8 Plus**, но все равно, оно остается неправдоподобным и не напоминает Stanford во всей его красе на этом изображении.

Нам необходимо это исправить. Как мы будем это делать?

Конечно, с помощью **ScrollView**. Если мы разместим **ScrollView**, то изображение может принять свой нормальный размер и мы сможем его прокручивать, увеличивать и уменьшать его масштаб. Это прекрасная возможность исправить наше приложение с помощью **ScrollView**.

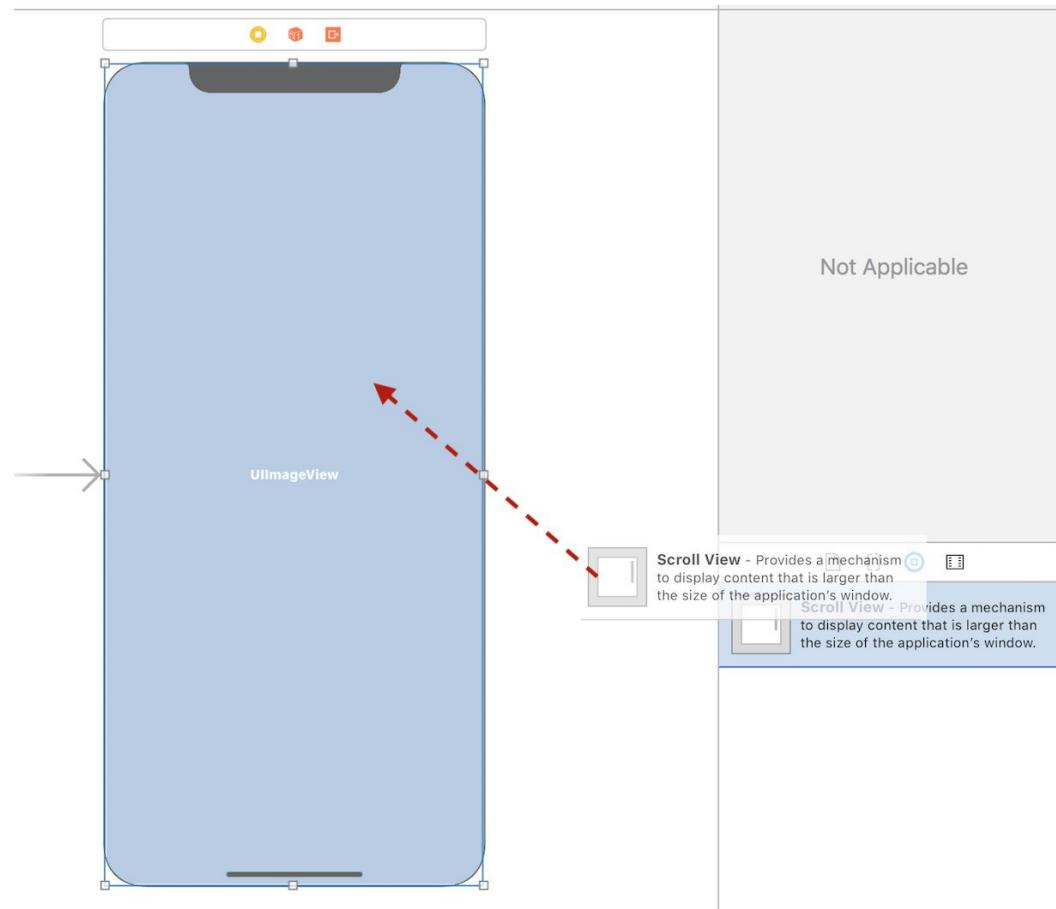
Я покажу вам оба способа размещения **ScrollView**: на **storyboard** и в коде.

Вы увидите, как это делается на **storyboard**, потому что это может быть хитроумно, вы увидите, что происходит с **contentsSize** и совсем прочим также и в коде.

Давайте начнем со **storyboard**.

Это наш **Image View**, и я собираюсь добавить **ScrollView**.

Я мог бы вытянуть **ScrollView** из Палитры Объектов, а затем начать вытягивать оттуда же **subviews** и размещать внутри **ScrollView**:



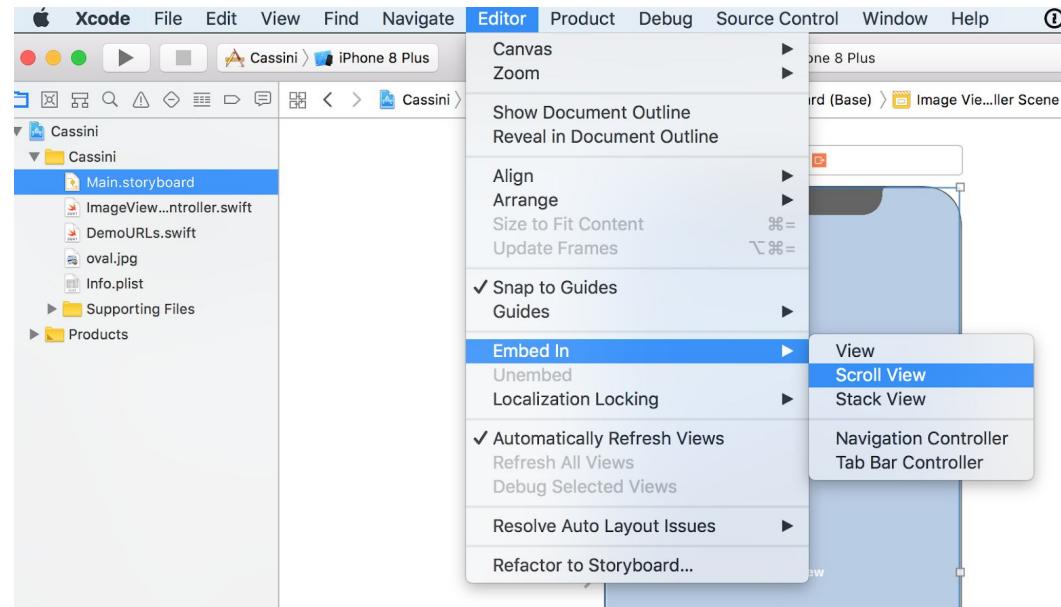
Между прочим, когда вы взаимодействуете с **subviews**, принадлежащими **ScrollView**, в **Interface Builder**, речь идет о контентной области. Так что **Interface Builder** даже не знает ничего о **subviews**, принадлежащих **ScrollView**, за исключением того, что они находятся в его контентной области. Например, если создаем некоторые ограничения (**constraints**) между **Image View** и **ScrollView**, это будут ограничения между **Image View** и контентной областью, этой большой белой

областью **ScrollView**. Это очень важно понимать, когда вы работаете со **ScrollView** в Interface Builder.

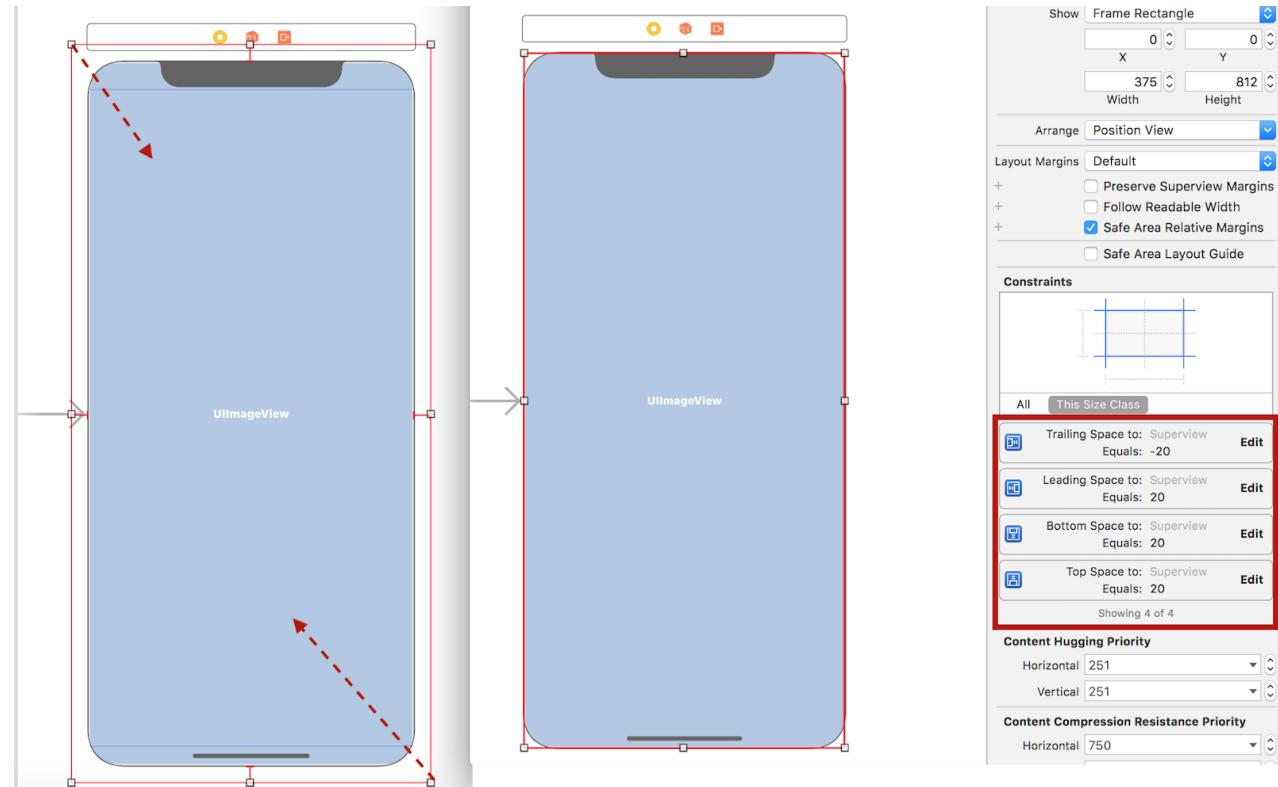
----- 60-ая минута лекции -----

Но я не буду вытягивать **ScrollView** из Палитры Объектов.

Вместо этого я использую меню **Editor** -> **Embed In** -> **ScrollView**, но сначала я выделю **Image View**, а затем использую меню **Editor** -> **Embed In** -> **ScrollView**:



Я говорил вам, что **ScrollView** появляется с рамкой в 20-pixel вокруг моего **Image View**, чего я, конечно, не хочу, я изменю размер **ScrollView** так, чтобы точно “подцепить” его к краям моего устройства:



Заметьте, что **ImageView** все еще смещен на 20 pixels, позже я собираюсь переместить ее.

Но пока я разместил **Image View** и **ScrollView** там, где я хотел. На самом деле не имеет значения, где действительно разместилось **Image View** с точки зрения своего размера и угла. Здесь важно то,

что **Image View** привязана к краям контентной области. Почему это так важно?

Не так важно, чтобы **Image View** имел правильный размер, но очень важно, чтобы контентная область имела правильный размер. Если я привяжу концы **Image View** к контентной области, и **Image View** изменит размер, контентная область также изменит размер. Нужно, чтобы контентная область всегда точно соответствовала размеру **Image View**, если вы хотите прокручивать полное изображение. В **Interface Builder**, а не в коде, вы с помощью ограничений (**constraints**) “подвешиваете” **Image View** к **ScrollView**. Фактически, это уже произошло, потому что, когда я вставлял (**embed in**) **Image View** в **ScrollView**, то **Image View** уже “привязался” к его **superview** таким образом, что удерживается на расстоянии 20 **points** от него со всех сторон, что мне абсолютно не нравится.

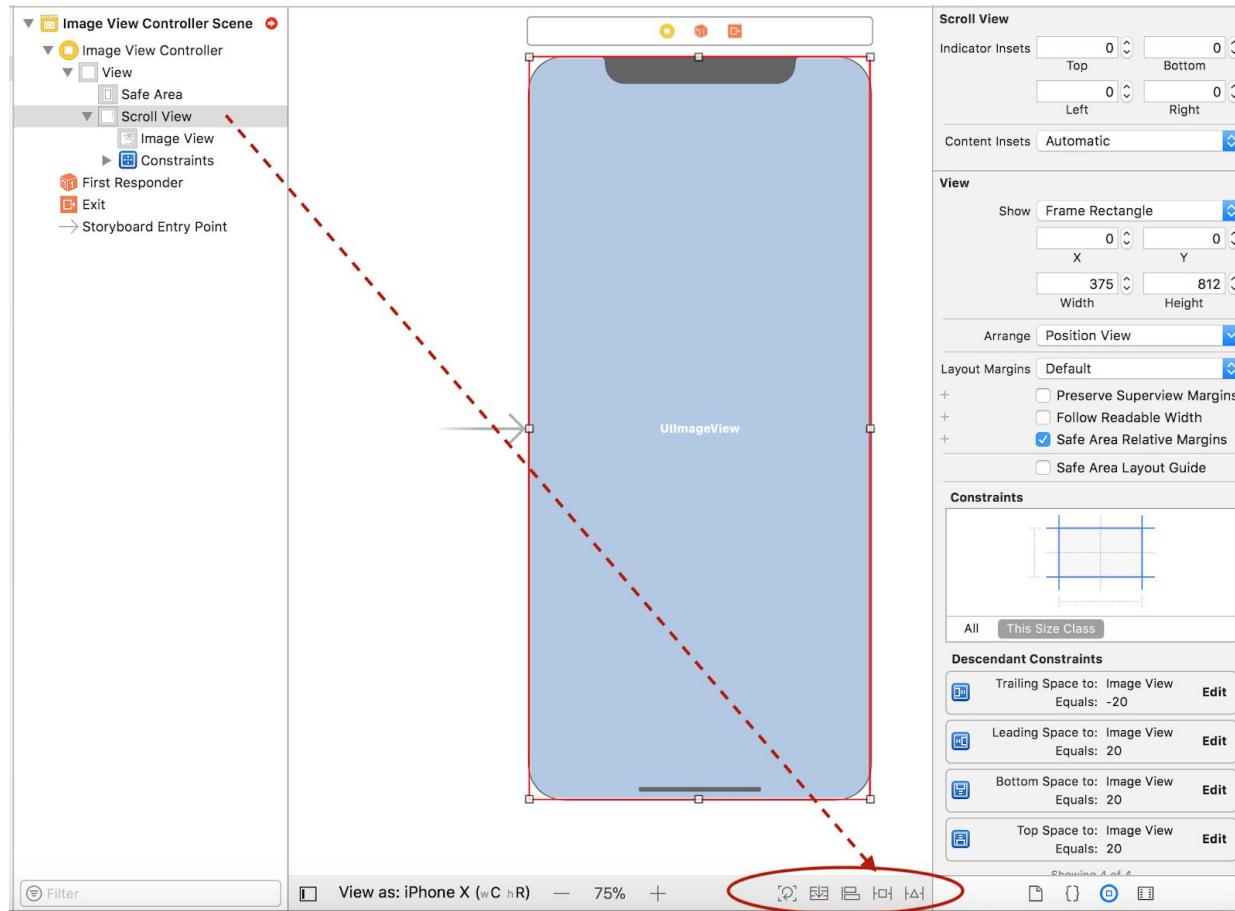
Но сам по себе **ScrollView** не привязан к краям моего экрана. Я изменил его размер, но у него нет никаких ограничений (**constraints**). Я привяжу **ScrollView** к краям моего экрана точно также, как и любые другие **views**.

Как мне выбрать **ScrollView**, между прочим? Когда я кликаю, то все время попадаю на **Image View**.

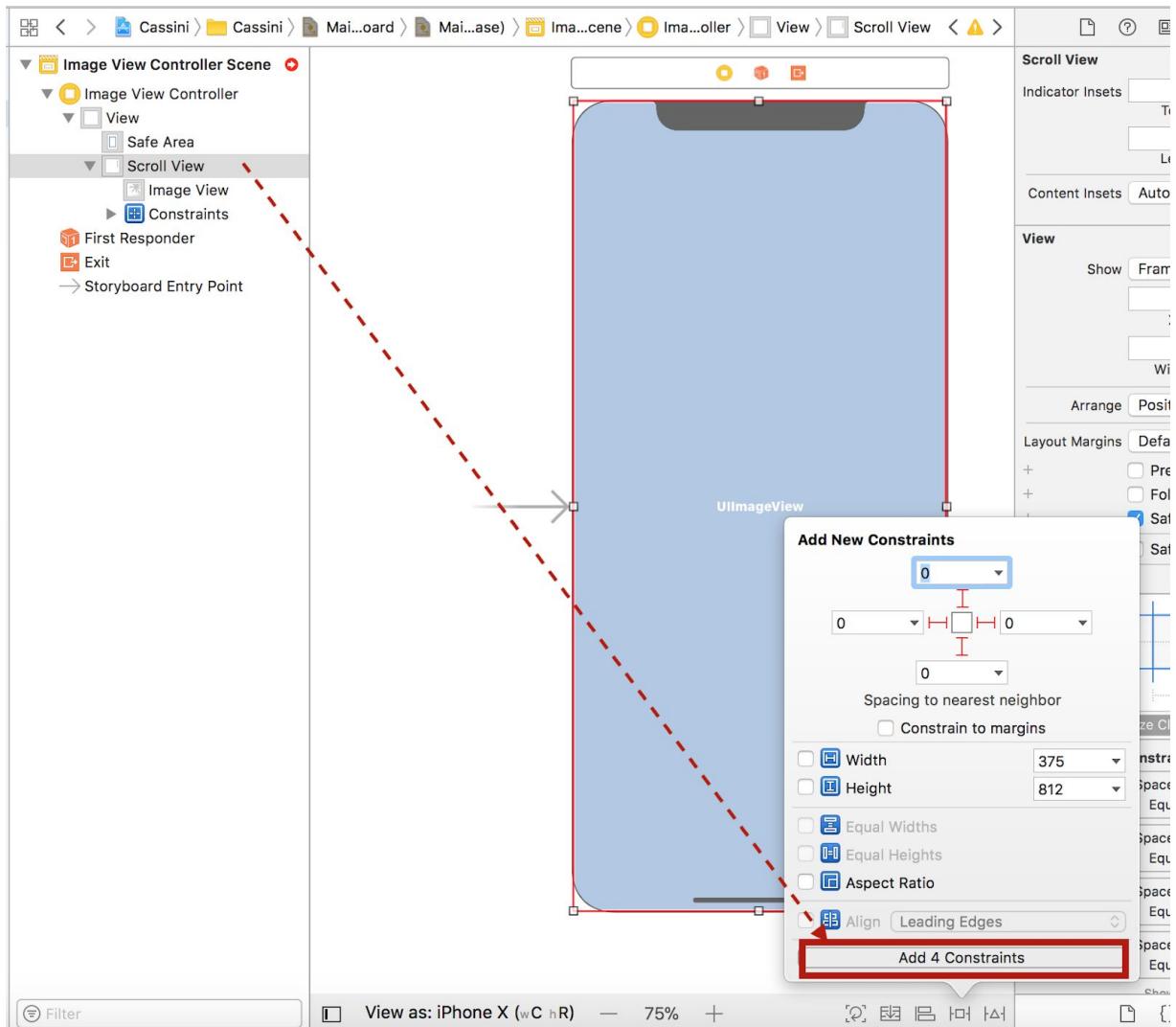
Кто-нибудь может указать мне способ выбора **ScrollView**?

Абсолютно правильно!

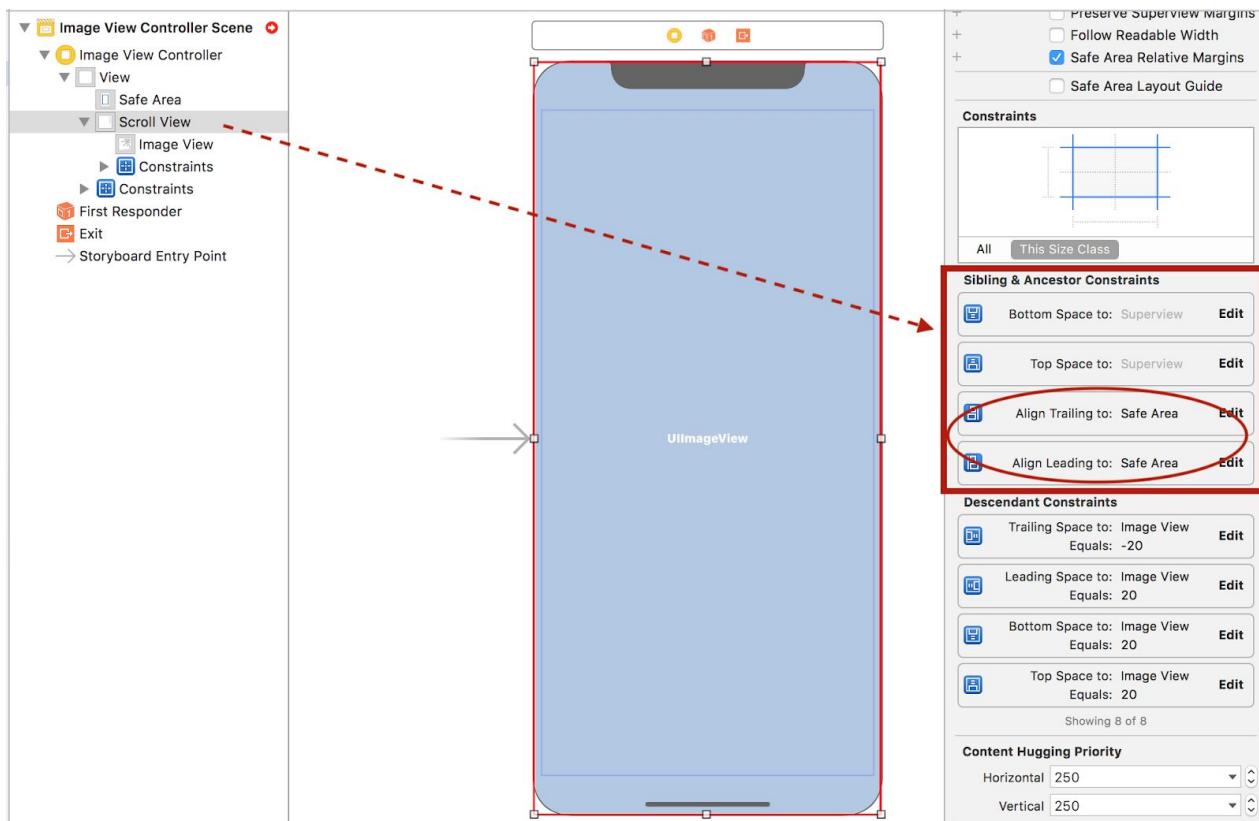
Схема UI (**Document outline**). Идем в **Document Outline** и выбираем там **ScrollView**:



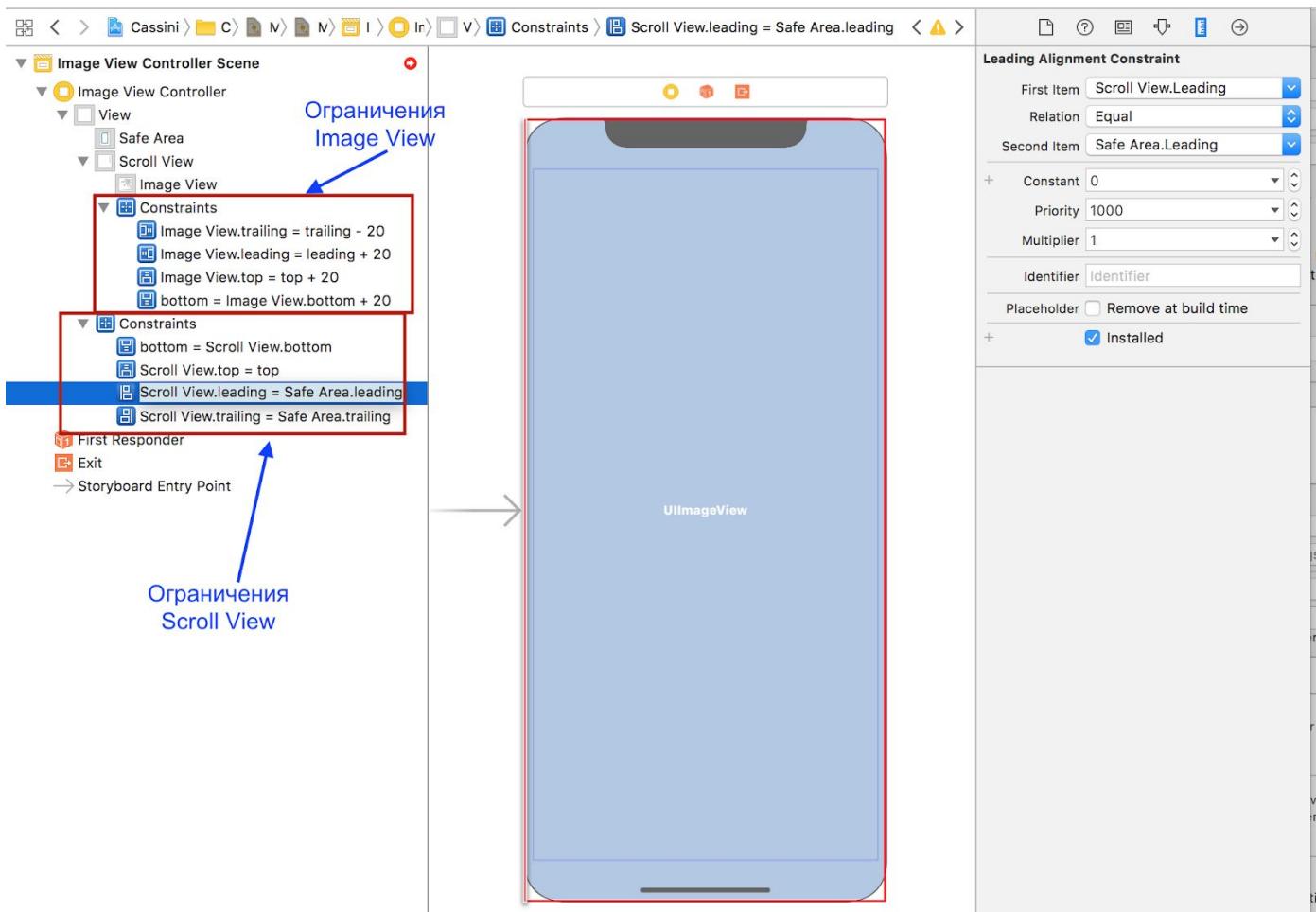
Действительно, очень легко выбрать что-то в **Document Outline**. Как только я выбрал **ScrollView**, мы можем воспользоваться кнопками, расположенными в правом нижнем углу для привязки **ScrollView** к краям “ближайшего соседа”. Мы выполняем абсолютно те же самые действия, что и при “привязки” **Image View** к краям “ближайшего соседа”:



Мы получаем те же самые проблемы с **Safe Area** для лидирующего (**Leading**) и хвостового (**Trailing**) зазоров:

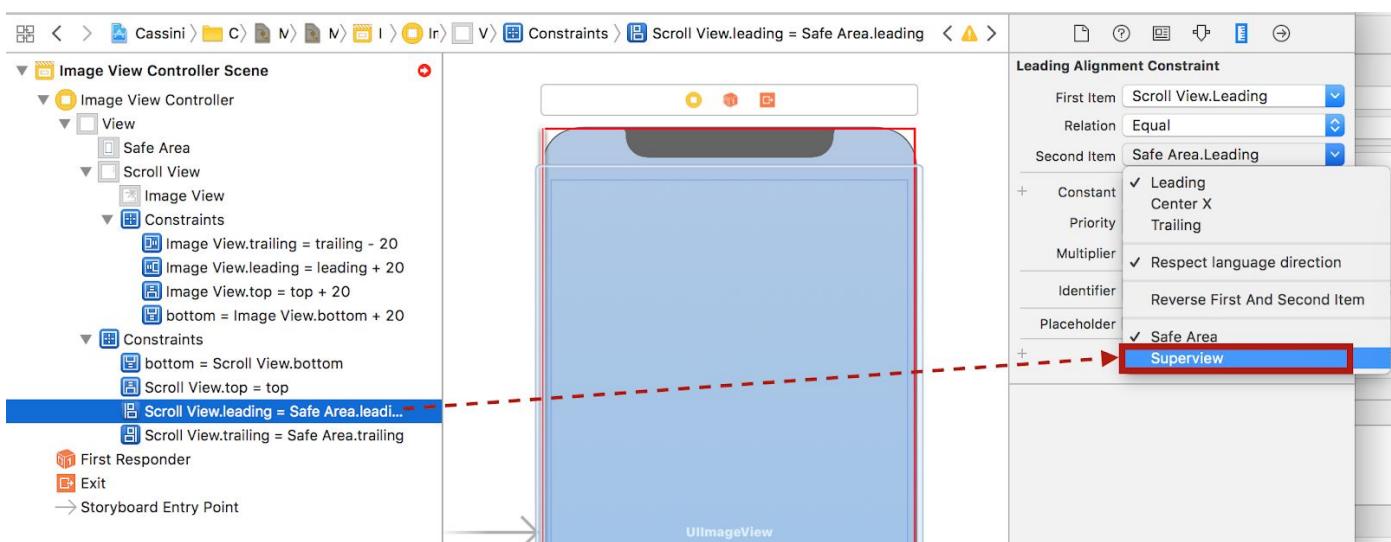


Теперь в Схеме UI (Document outline) мы смотрим на полученные ограничения (**constraints**), но будьте внимательны относительно отступа. Одни ограничения (**constraints**) относятся к **Image View**, а другие ограничения (**constraints**) относятся к **ScrollView**:

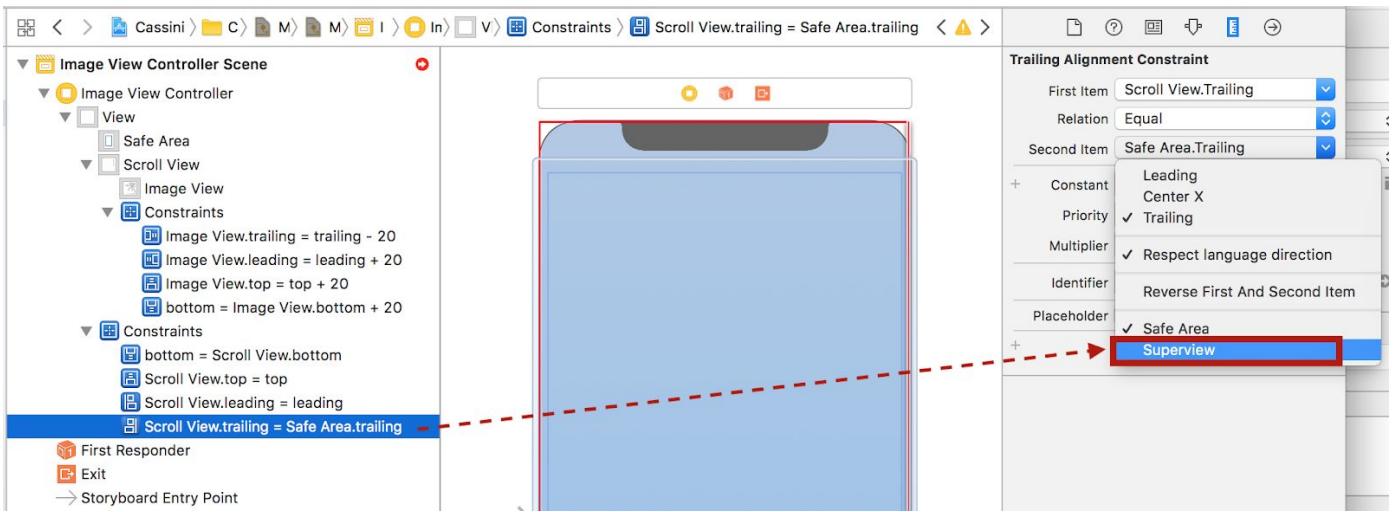


Видите? Отступ имеет значение.

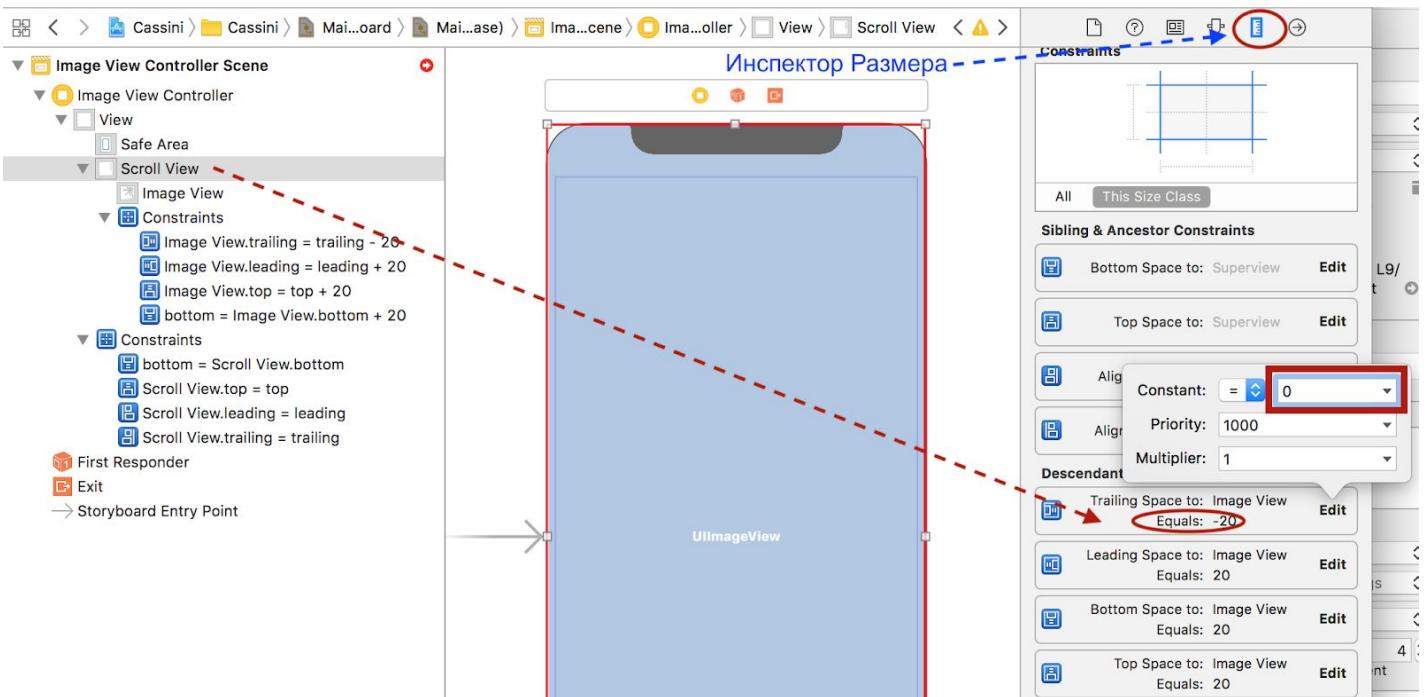
Я буду изменять **Safe Area** на **Superview** в лидирующем (**Leading**)...



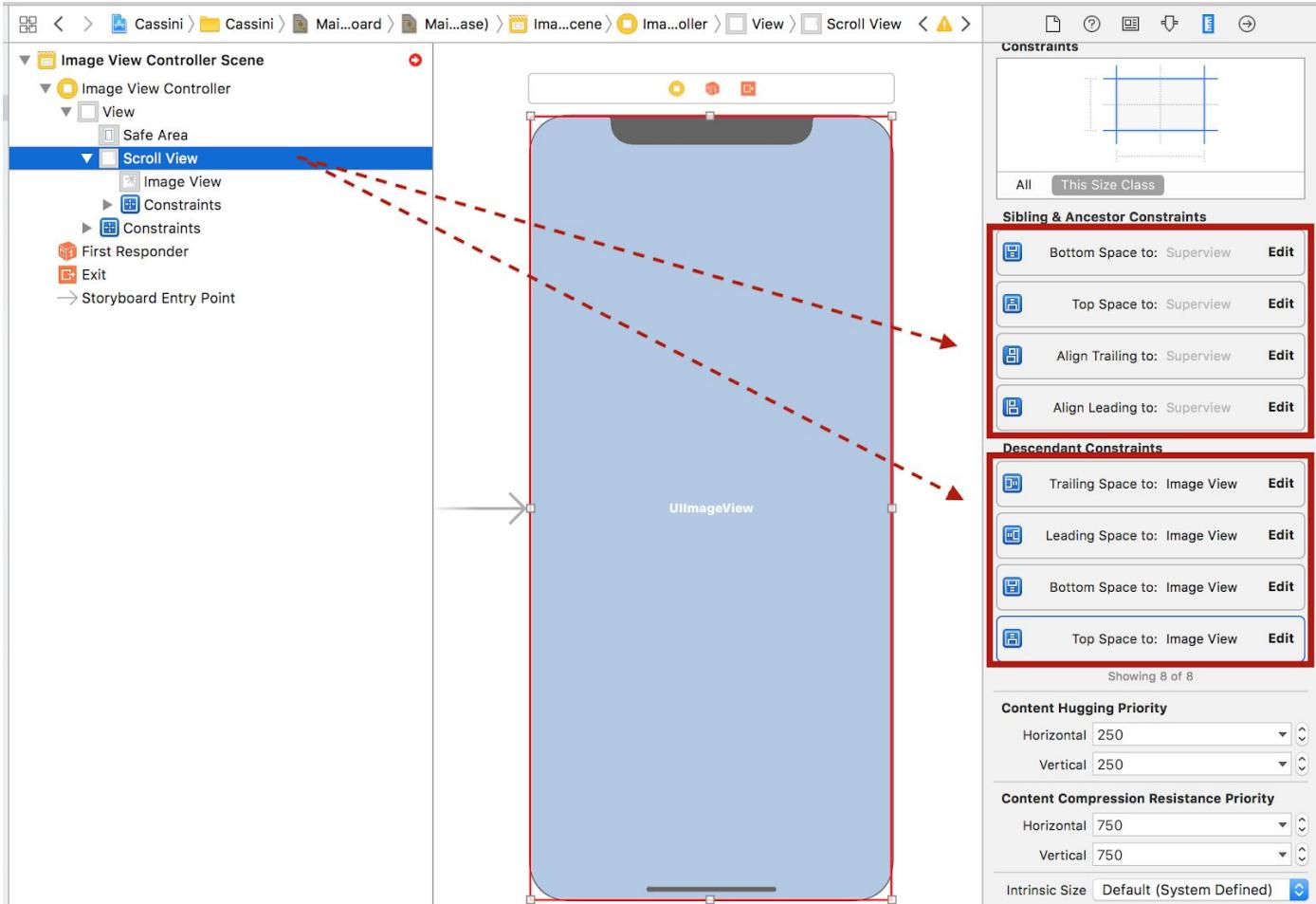
... и хвостовым (**Trailing**) ограничениях для **ScrollView**:



И, наконец, последняя вещь, которую я хочу сделать, это исправить эти **20 points**, которые я получил, когда вставлял мой **Image View** в **ScrollView**, и которые я не хочу иметь. Я могу отредактировать это прямо в Инспекторе Размера (**Size Inspector**), заменив все **20** на **0**:

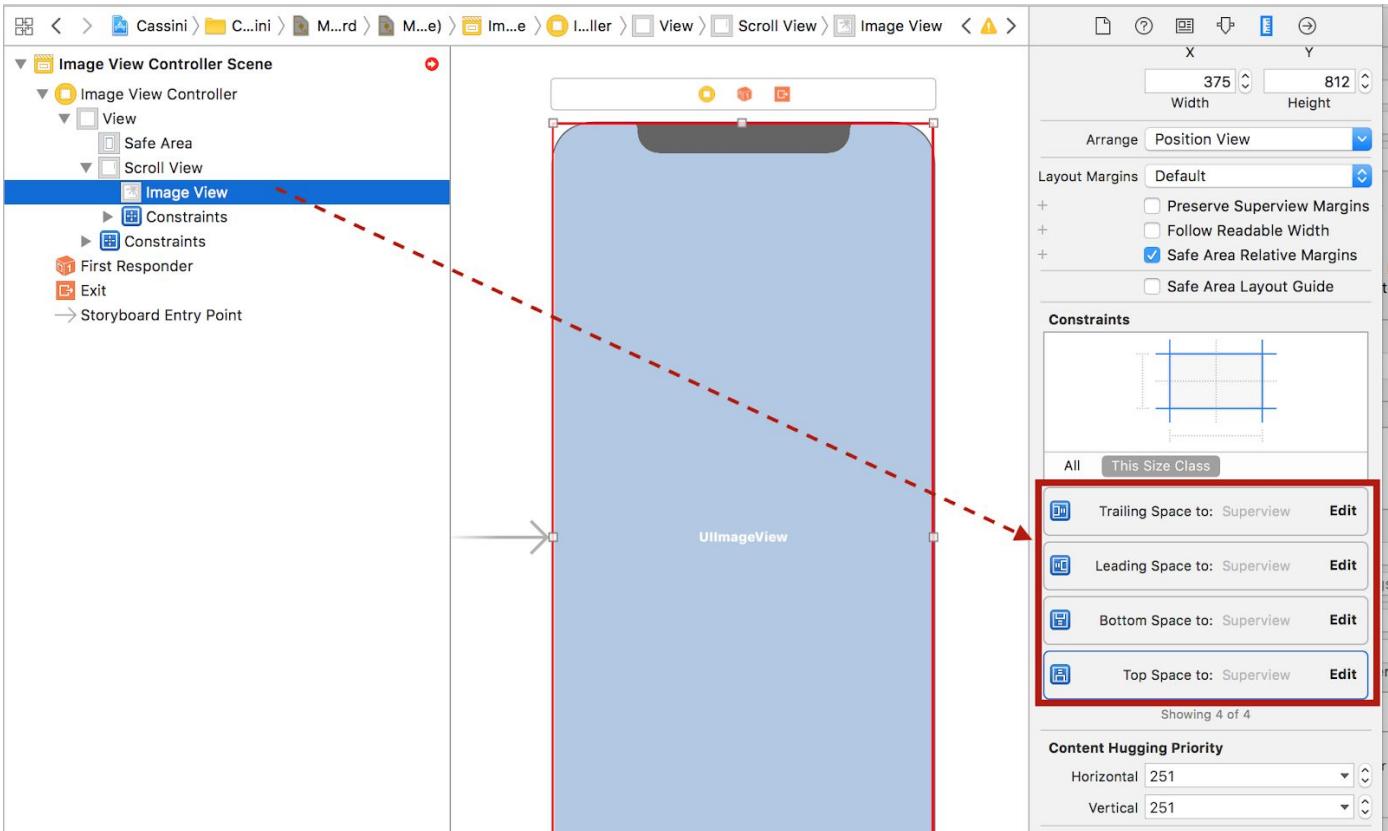


В результате мы получим прекрасные ограничения:



Видите? ScrollView “подцепляется” к своемуSuperview.

Вы видите, что ImageView также “подцепляется” к своемуSuperview...



... который в Interface Builder означает контентную область ScrollView.

Когда у вас есть subview, например, ImageView - это subview ScrollView и это показано тем, что

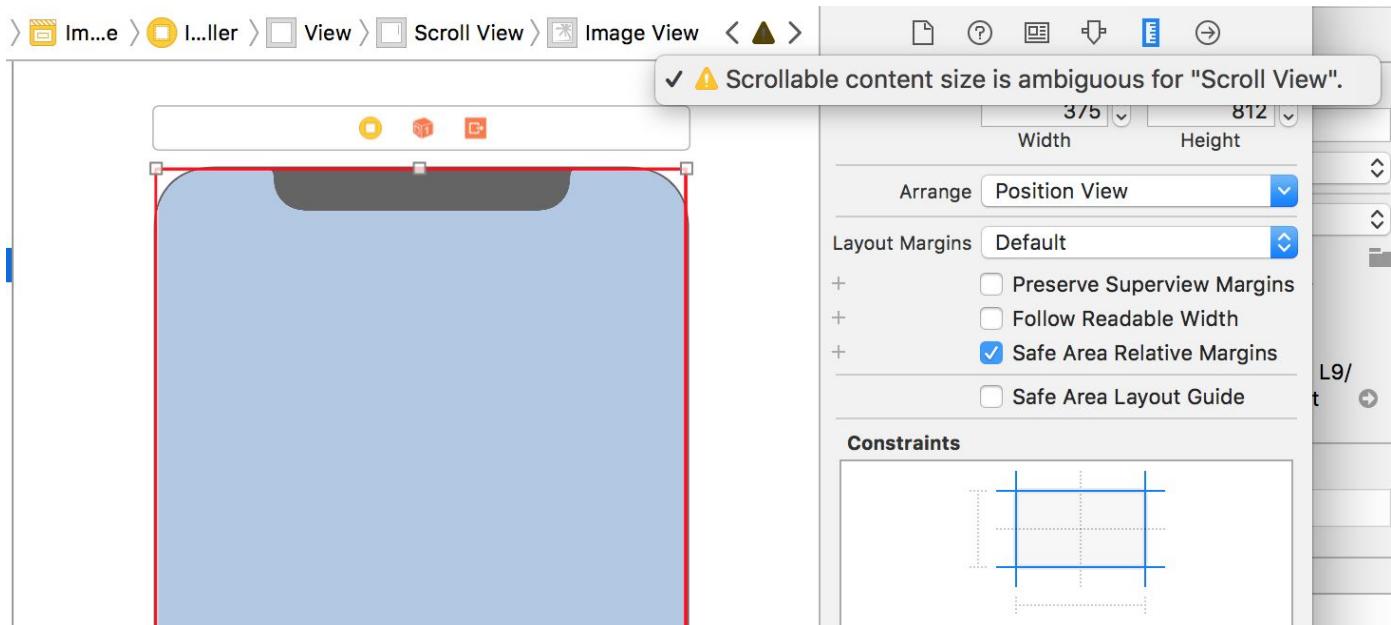
он имеет отступ, то это означает, что когда мы говорим о его **Superview Image View** как о **ScrollView**, то мы говорим о контентной области.

Почему у нас все еще красный цвет?

Мне кажется, что мы определили все ограничения (**constraints**) между этими **views**.

Почему красный цвет?

Давайте посмотрим, что скажет предупреждение, которое присутствует в виде желтого треугольника в верхней правой части экрана:



Это предупреждение говорит: “Размер прокручиваемого контента неоднозначен для **ScrollView**.”

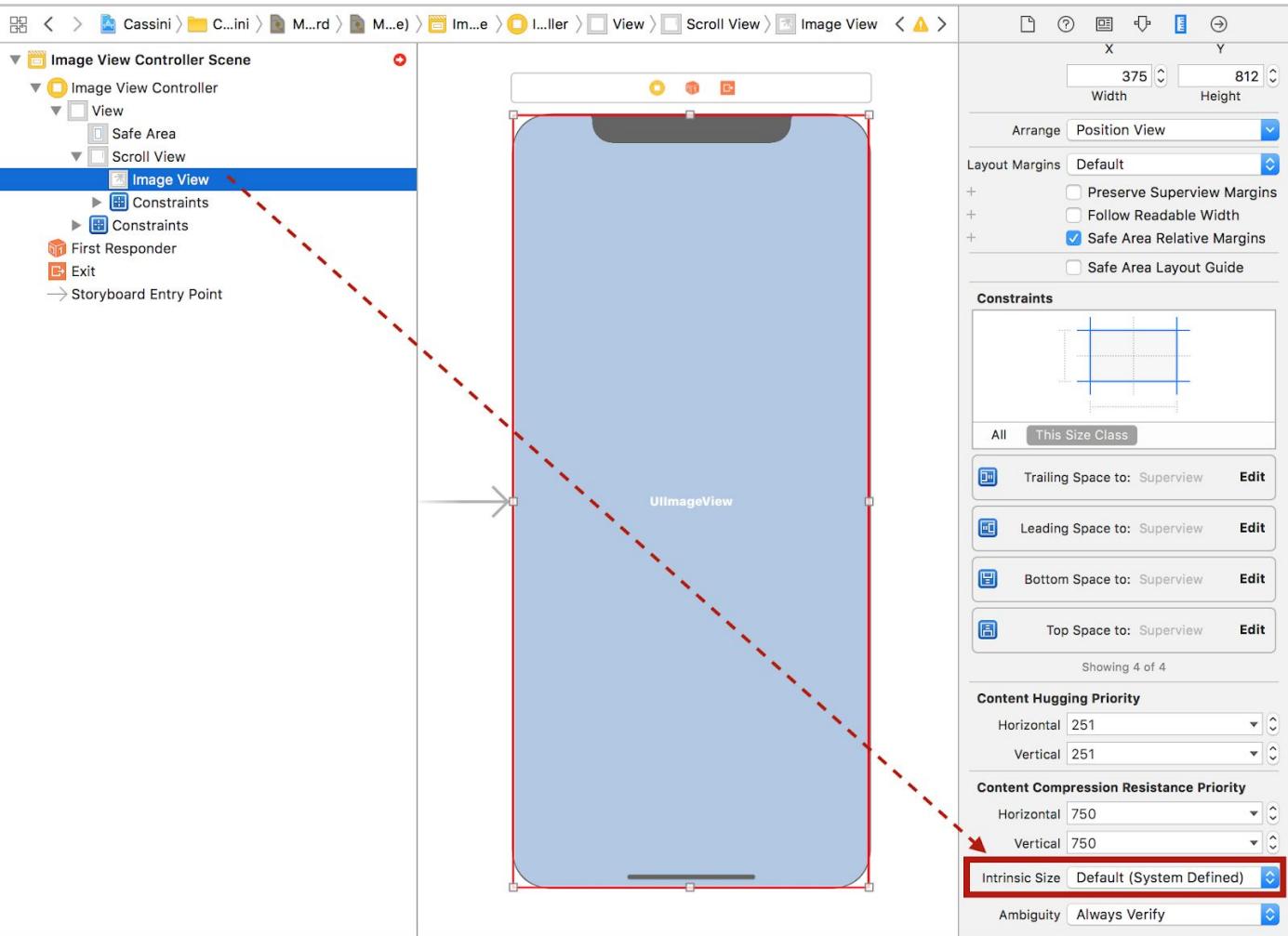
Оно говорит о том, что непонятно, насколько большой является контентная область (**content area**)?

Ответ заключается в том, что контентная область такая же большая, как и **Image View**, потому что края этих двух **views** привязаны друг к другу. Но **Interface Builder** не знает, как велико **Image View**. Почему? Потому что у **Image View** нет изображения (**image**). Этот **Image View** - пустой, у него нет изображения **image**, так что **Interface Builder**, похоже, не знает, насколько большим оно является, поэтому и сообщает в предупреждении: “Размер прокручиваемого контента неоднозначен для **ScrollView**.”

Это неразрешимая проблема, потому что мы не устанавливаем изображение **image** до тех пор, пока не запустим наш код на выполнение.

Как нам быть со всем этим в **Interface Builder**?

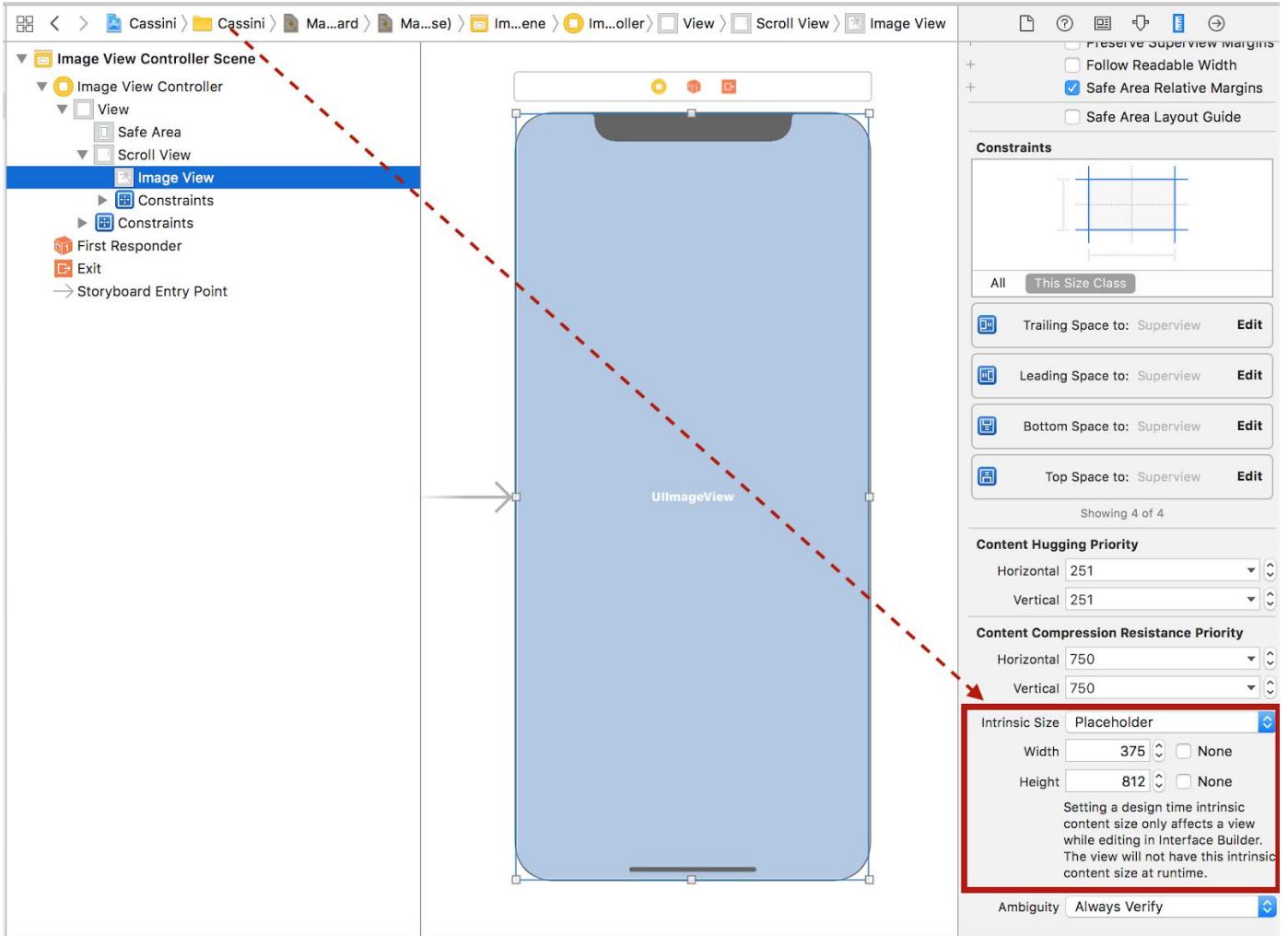
У **Interface Builder** есть реально крутая возможность: если вы инспектируете **Image View** в Инспекторе Размера, то в самом низу есть поле **Intrinsic Size** (собственный размер):



Intrinsic Size (собственный размер) - это некоторый размер, который “подгоняется” под свое содержимое. Например, для метки **UILabel Intrinsic Size** (собственный размер) - это минимальный размер, позволяющий вместить содержащийся в метке текст. Для **Image View Intrinsic Size** (собственный размер) - это минимальный размер, позволяющий “подогнать” естественный коэффициент пропорциональности (Aspect Ratio) и размер изображения.

В данный момент поле **Intrinsic Size** установлено в **Default** (значение по умолчанию), но вы можете установить его в **Placeholder** (местозаполнитель), который может иметь любой случайный размер, какой вы захотите, но он будет использоваться только в **Interface Builder**. Он просто будет “держать” это место в **Interface Builder**. В коде мы должны будем его установить.

----- 65-ая минута лекции -----



Видите? Мы избавились от красного цвета, “жалующегося” на ошибку, потому что теперь **Image View** имеет размер и контентная область также будет иметь размер 375 x 812, но только в **Interface Builder**. Как только мы запустим приложение, размер контентной области будет равен размеру **image View**, который зависит от размера находящегося в нем изображения, установленного нами. Всем понятно?

Я говорил вам, что это немного хитроумно, работать со **ScrollView** в **Interface Builder**. Вы должны немного привыкнуть к тому, что устанавливая ограничения (**constraints**) между **Image View** и **ScrollView**, вы говорите о контентной области, и это управляет именно контентной областью **ScrollView**. Давайте запустим приложение, потому что мы установили все ограничения (**constraints**), которые необходимы для работы **ScrollView**, нам ничего не нужно делать в коде.

Итак, запускаем приложение.

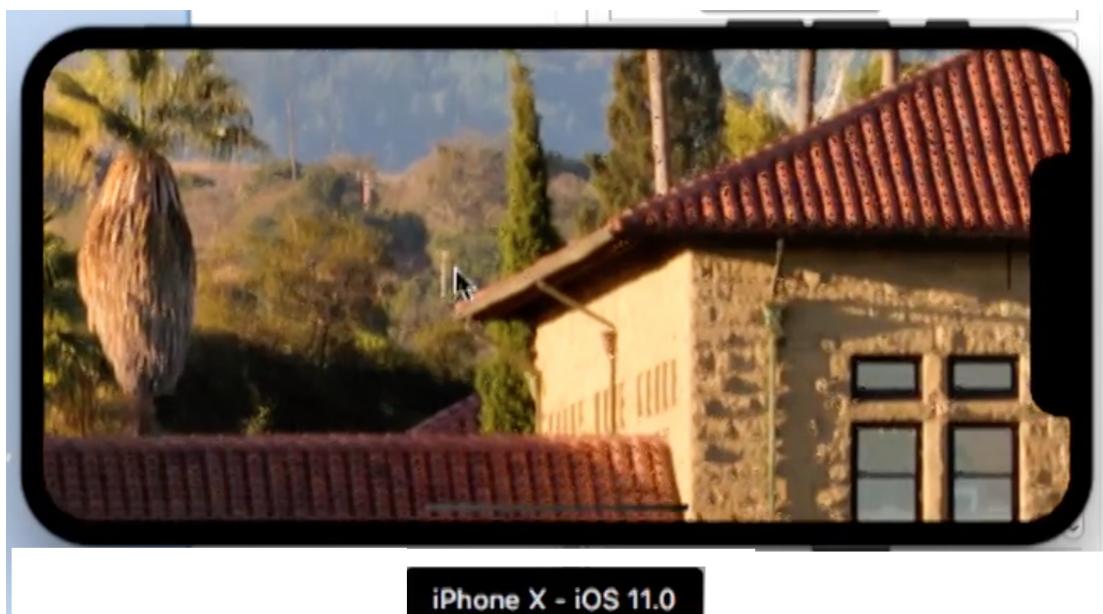
Я хочу запустить на **iPhone X**, это будет более интересно, потому что у него есть **Safe Area**, а на других **iPhones** ее нет.



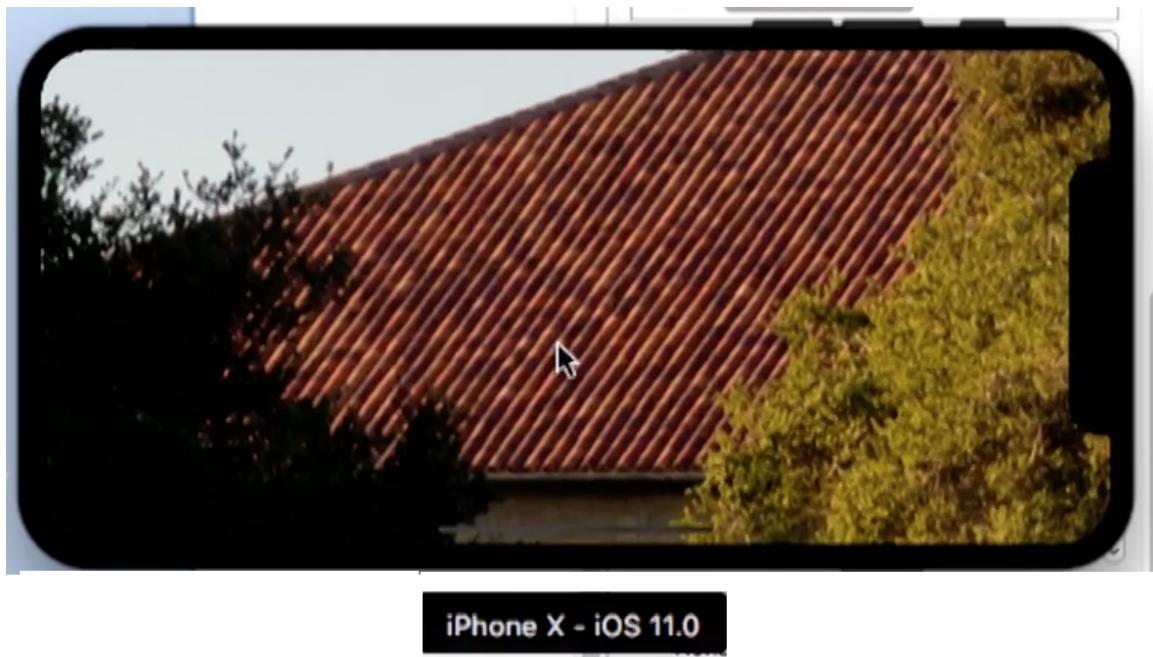
Смотрите - это наше изображение. Вы видите голубое небо на нашем изображении, прямо там, где находится время, индикатор работы сети и прочее. Я прокручиваю его и вижу деревья.
Здорово.

Заметьте, что если я прокручиваю вниз, то **ScrollView** выявляет **Safe Area** и автоматически располагается ниже. Это говорит о том, что **ScrollView** становится очень сообразительным, если дело касается **Safe Area**, даже если вы “привязали” его к краям **Superview**, он все равно знает, что существует **Safe Area**.

Мы можем пойти в ландшафтный режим...



... и прокручивать там. Мы можем найти то, что нас интересует:



На этом все. Это все, что нам требуется для **ScrollView**. Работать с ним очень легко.

А как насчет того, чтобы сделать это в коде?

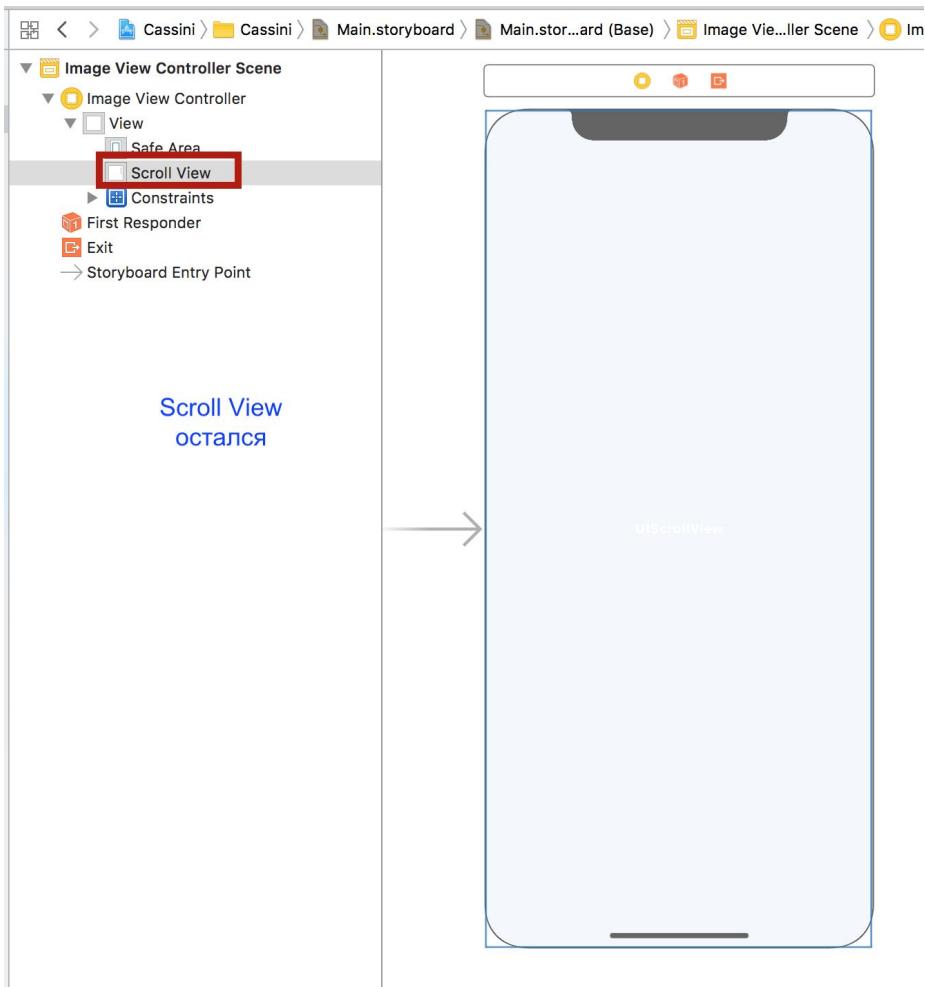
В некотором смысле в коде это делается легче, потому что у вас нет 20 pixel рамки и всего, что с этим связано.

Я собираюсь сделать все то же самое в коде.

Я выделяю мой **Image View**....



... и удаляю его:



Я удаляю только **Image View**, оставляя **ScrollView**, потому что мне необходимо “разговаривать” со **ScrollView** для того, чтобы установить размер его контентной области **contentSize**.

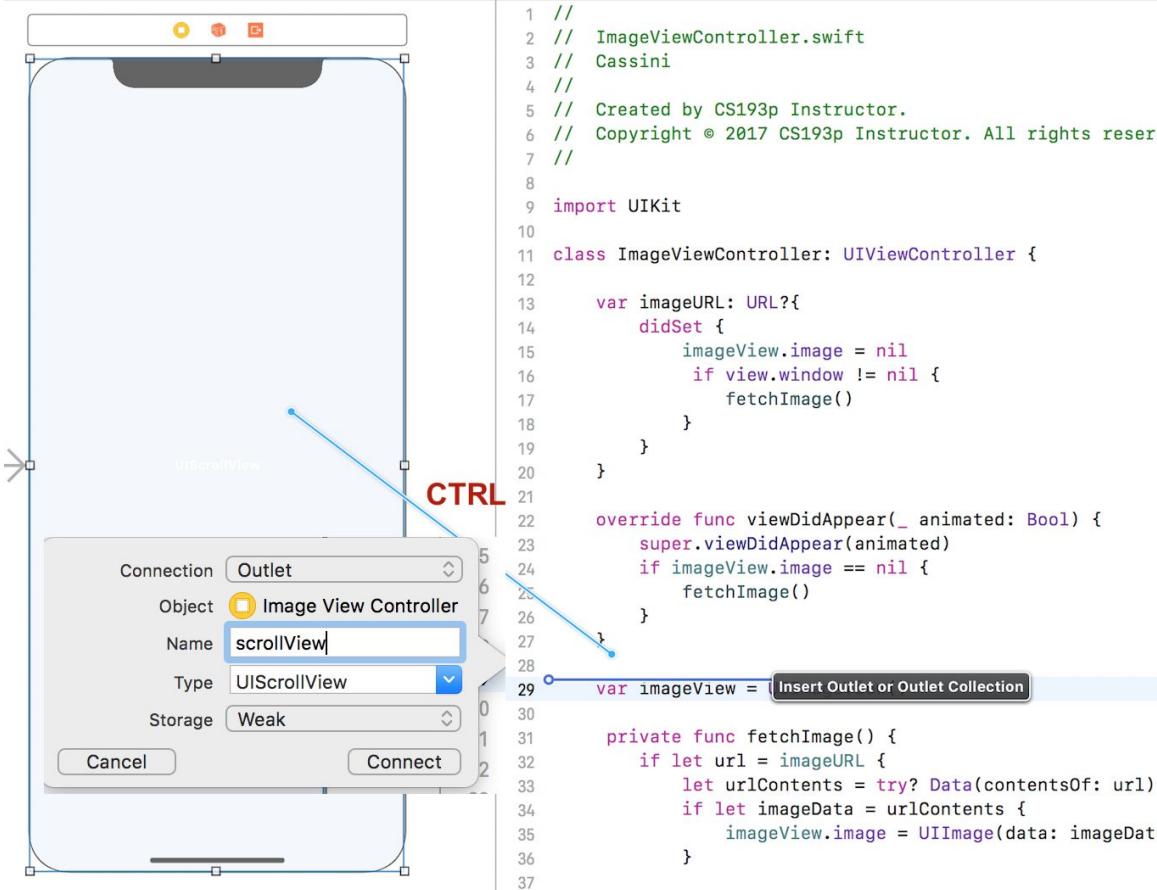
Давайте разместим **UI** и код одновременно на экране. Я собираюсь создать **Outlet** для этого **ScrollView**, чтобы я смог с ним разговаривать. Но мне больше не нужен **Outlet** для моего **Image View**, потому что я создам его в коде прямо сейчас:

imageView = UIImageView()

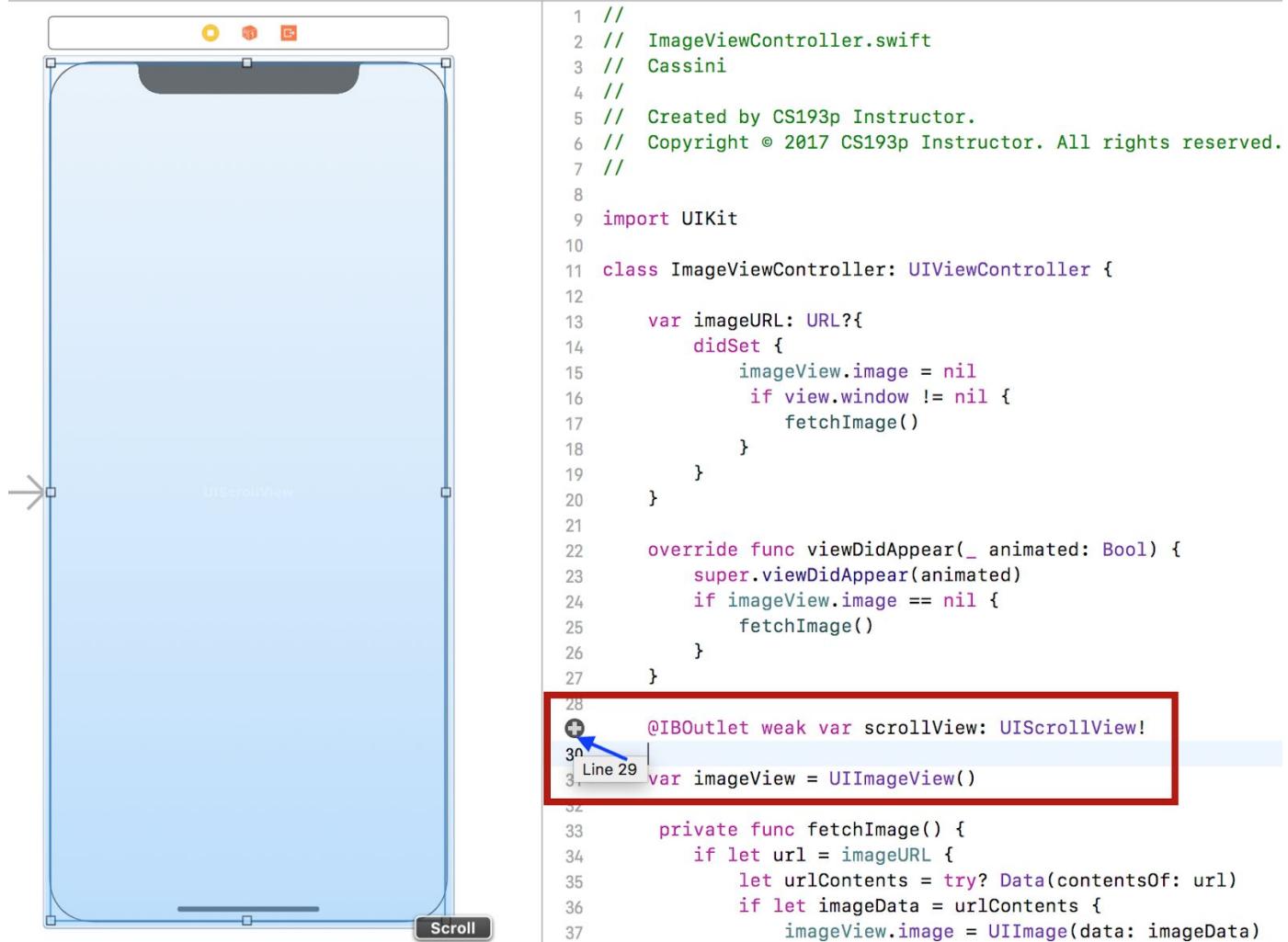
Это **imageView** с нулевым размером, но пока это нормально. Когда я буду устанавливать изображение **image** моему **imageView**, я изменю его размер.

Затем делаем **CTRL**-перетягивание, чтобы “подцепить” **ScrollView**.

Я назову мой новый **Outlet scrollview**:



Теперь у нас есть связь с обоими: **imageView** и **scrollView**.



Я хочу, чтобы **imageView** был контентной областью для **scrollView**.

Как только iOS “подцепит” **Outlet scrollView**, я попрошу **scrollView** добавить к нему в качестве **subview** мой **imageView**:

```
    @IBOutlet weak var scrollView: UIScrollView!{  
        didSet {  
            scrollView.addSubview(imageView)  
        }  
    }  
  
    var imageView = UIImageView()
```

Прекрасно. Контентная область имеет нулевой размер, так что мой **imageView** находится в левом верхнем углу с нулевым размером и поэтому полностью пустой.

Каждый раз, когда я устанавливаю изображение **image** в **imageView**, я делаю это в двух местах...

```
var imageURL: URL?{  
    didSet {  
        imageView.image = nil  
        if view.window != nil {  
            fetchImage()  
        }  
    }  
  
    override func viewDidAppear(_ animated: Bool) {  
        super.viewDidAppear(animated)  
        if imageView.image == nil {  
            fetchImage()  
        }  
    }  
  
    @IBOutlet weak var scrollView: UIScrollView!{  
        didSet {  
            scrollView.addSubview(imageView)  
        }  
    }  
  
    var imageView = UIImageView()  
  
    private func fetchImage() {  
        if let url = imageURL {  
            let urlContents = try? Data(contentsOf: url)  
            if let imageData = urlContents {  
                imageView.image = UIImage(data: imageData)  
            }  
        }  
    }  
}
```

... я должен попросить **imageView** “подогнать” свой размер **size** к размеру изображения:

```

private func fetchImage() {
    if let url = imageURL {
        let urlContents = try? Data(contentsOf: url)
        if let imageData = urlContents {
            imageView.image = UIImage(data: imageData)
            imageView.sizeToFit()
        }
    }
}

```

Метод **sizeToFit** означает “сделай себя равным размеру **Intrinsic Size**”. Теперь размер **imageView** наилучшим образом установлен и у него появился **frame**, который мы можем использовать для установки контентной области **contentSize** для **scrollView**:

```

private func fetchImage() {
    if let url = imageURL {
        let urlContents = try? Data(contentsOf: url)
        if let imageData = urlContents {
            imageView.image = UIImage(data: imageData)
            imageView.sizeToFit()
            scrollView.contentSize = imageView.frame.size
        }
    }
}

```

Если я этого не сделаю, то никакого прокручивания вообще не будет.

Понимаете почему? Потому что **contentSize** будет нулевым прямоугольником и в нем нечего будет прокручивать.

Мне нужно сделать то же самое там, где я устанавливаю мой **imageView.image** в **nil**:

```

class ImageViewController: UIViewController {

    var imageURL: URL?
    didSet {
        imageView.image = nil
        imageView.sizeToFit()
        scrollView.contentSize = imageView.frame.size
        if view.window != nil {
            fetchImage()
        }
    }
}

```

Я хочу опять вернуть **contentSize** в нулевой размер, и я просто скопировал предыдущий код и вставил его в нужное место. Но мы знаем, что это очень ПЛОХО. Нам всегда говорят: “Не делайте этого.” Но это демонстрационный пример, и я мог бы сослаться на то, что у меня нет времени на то, чтобы это исправить. Но я найду это время, потому что я хочу показать вам очень интересный способ исправления этой ситуации.

Я создам **private** переменную **var**, которую назову **image**, она будет иметь ТИП **UIImage** и она будет вычисляемой. Каждый раз, когда я хочу установить **set {}** эту переменную, и каждый раз, когда я хочу получить **get {}** эту переменную, я буду брать ее из **imageView**:

```
private var image: UIImage? {
    get {
        return imageView.image
    }
    set {
        imageView.image = newValue
    }
}
```

Это обычное вычисляемое свойство. Но самое замечательное то, что я могу разместить в этой вычисляемой переменной **image** мой предыдущий код, который я вынужден был разместить в двух местах:

```
private var image: UIImage? {
    get {
        return imageView.image
    }
    set {
        imageView.image = newValue
        imageView.sizeToFit()
        scrollView.contentSize = imageView.frame.size
    }
}
```

Теперь этот код не нужен в этих двух местах, и я могу заменить **imageView.image** просто на **image**. В одном месте...

```
var imageURL: URL? {
    didSet {
        imageView.image = nil
        imageView.sizeToFit()
        scrollView.contentSize = imageView.frame.size
        if view.window != nil {
            fetchImage()
        }
    }
}
```

... И во втором:

```

private func fetchImage() {
    if let url = imageURL {
        let urlContents = try? Data(contentsOf: url)
        if let imageData = urlContents {
            imageView.image = UIImage(data: imageData)
imageView.sizeToFit()
scrollView.contentSize = imageView.frame.size
        }
    }
}

```

----- 70-ая минута лекции -----

Это пример использования вычисляемой переменной **var**, потому что концептуально это изображение **image**, которое запоминается в **imageView**. Мне не нужно его дублировать, потому что я получаю его из **imageView**, и я устанавливаю его в **imageView**. Но я также могу дополнительно попутно выполнять необходимые настройки для него.

На этом все. Вы видите, что в коде также легко работать со **scrollView**.

Запускаем приложение. Это большой файл, даже при локальном размещении требуется некоторое ощущимое время для загрузки. Но вот мы его получили.



Это в точности то же самое, что было у нас перед этим, я могу также найти то, что мне нужно.
<http://bestkora.com> Разработка iOS приложений. CS193P Stanford Fall 2017 iOS 11 Swift 4

Давайте повернем наше устройство:



Но было бы реально здорово, если мы смогли увидеть всю картинку целиком. Это очень раздражает, когда вы можете рассматривать картинку только по частям. Конечно, это замечательно, что мы можем рассматривать изображение по частям, но я бы хотел иметь возможность его масштабировать и увидеть всю картинку целиком.

Итак, нам нужно МАСШТАБИРОВАНИЕ (**zooming**).

Какие две вещи нам нужны для масштабирования?

Нам нужно установить **minimumZoomScale** и **maximumZoomScale**.

Нам также нужно сказать с помощью делегата **delegate**, какое **view** мы хотим масштабировать.

Ясно, на каком **view** мы хотим заставить работать **transform**, это наш **ImageView**.

Давайте сделаем это.

Там, где мой **scrollView** “подцепляется” **iOS**, давайте зададим минимальный масштаб **minimumZoomScale** равным **1/25**, давая возможность нашему изображению уменьшаться до 1/25 своего нормального размера. А для максимального масштаба **maximumZoomScale** я, возможно, задам значение **1.0**.

```
① @IBOutlet weak var scrollView: UIScrollView! {
②     didSet {
③         scrollView.minimumZoomScale = 1/25
④         scrollView.maximumZoomScale = 1.0
⑤         scrollView.addSubview(imageView)
⑥     }
⑦ }
```

Почему я установил максимальный масштаб равный **1.0**?

Возможно, я не хочу реального увеличения масштаба потому, что может возникнуть битовая "зернистость" ("пикельность") изображения, поэтому я не хочу увеличения масштаба больше, чем **1.0**. Но зато мы можем уменьшать масштаб до **1/25**.

Итак, это часть 1. Часть 2 состоит в назначении себя **self** делегатом **delegate scrollView**:

```
① @IBOutlet weak var scrollView: UIScrollView!{
 ②     didSet {
 ③         scrollView.minimumZoomScale = 1/25
 ④         scrollView.maximumZoomScale = 1.0
 ⑤         scrollView.delegate = self
 ⑥     }
 ⑦ }
```

Cannot assign value of type 'ImageViewController' to type 'UIScrollViewDelegate'?
Insert ' as! UIScrollViewDelegate'

Fix

Эта строка генерирует ошибку, потому что **self** не является **UIScrollViewDelegate**.

Это легко исправить, если в самой верхней части кода заявить, что вы являетесь

UIScrollViewDelegate:

```
① import UIKit
②
③ class ImageViewController: UIViewController, UIScrollViewDelegate {
④
⑤     var imageURL: URL?
⑥     didSet {
⑦         image = nil
⑧         if view.window != nil {
⑨             fetchImage()
⑩         }
⑪     }
⑫ }
```

Итак, я подтвердил, что реализую все методы протокола **UIScrollViewDelegate**, и не реализовал ни один из них, потому что все они - **Objective-C** необязательные, то есть **optional** методы. Поэтому все ошибки исчезли.

Тем не менее масштабирование не начнет работать, если я не реализую метод **viewForZooming (in scrollView:)**:

```
func viewForZooming(in scrollView: UIScrollView) -> UIView? {
    return imageView
}
```

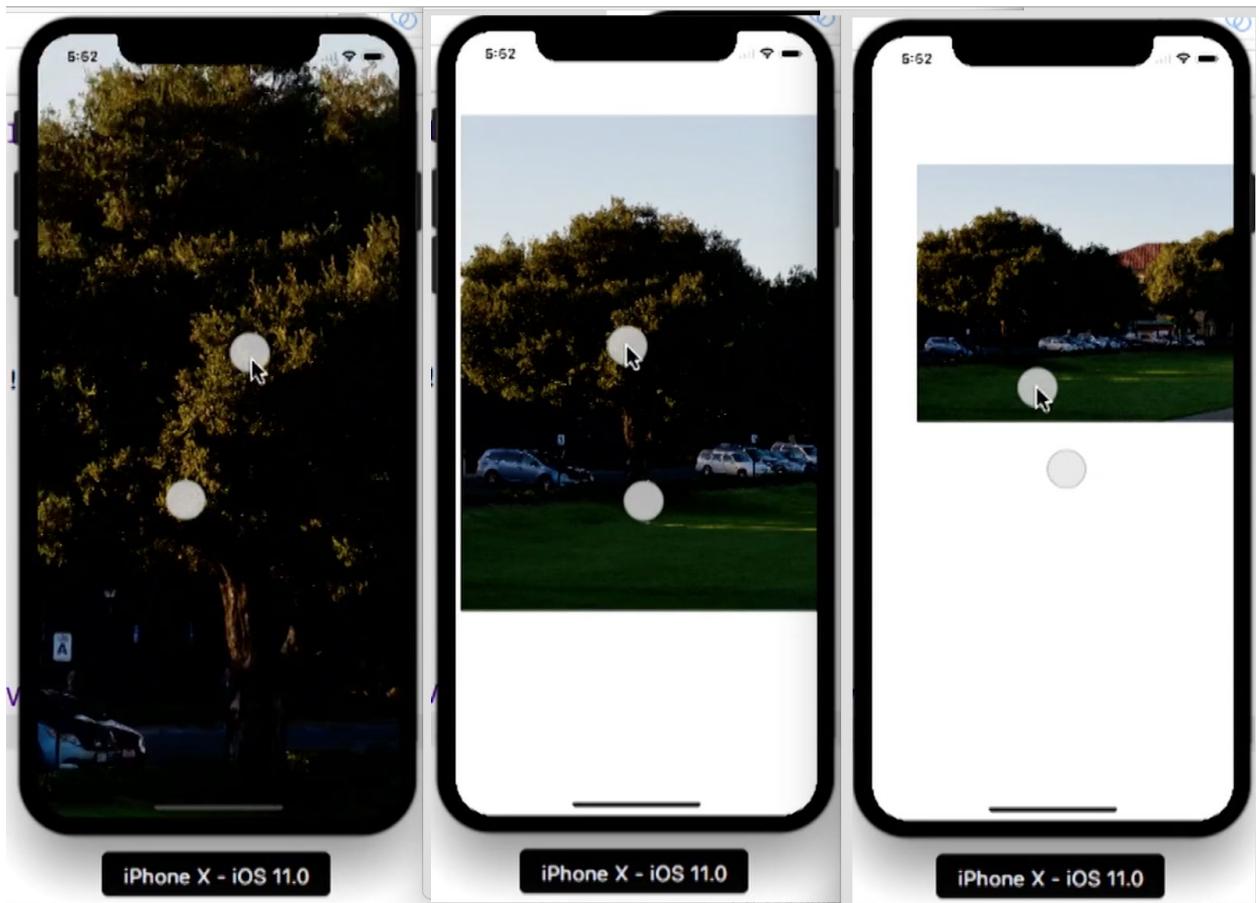
В этом методе нам просто необходимо вернуть **imageView**, потому что этот **view**, который является **subview** в контентной области и который я хочу трансформировать с помощью жеста **pinch**.

Давайте попробуем. Мы все еще можем прокручивать изображение, но теперь я могу менять масштаб. Помните, как мы выполняем жест **pinch** на симуляторе? Мы должны держать клавишу **option**, и тогда на экране появятся два серых маленьких кружочка, взаимным расположением

которых можно управлять с помощью жеста **pinch**:

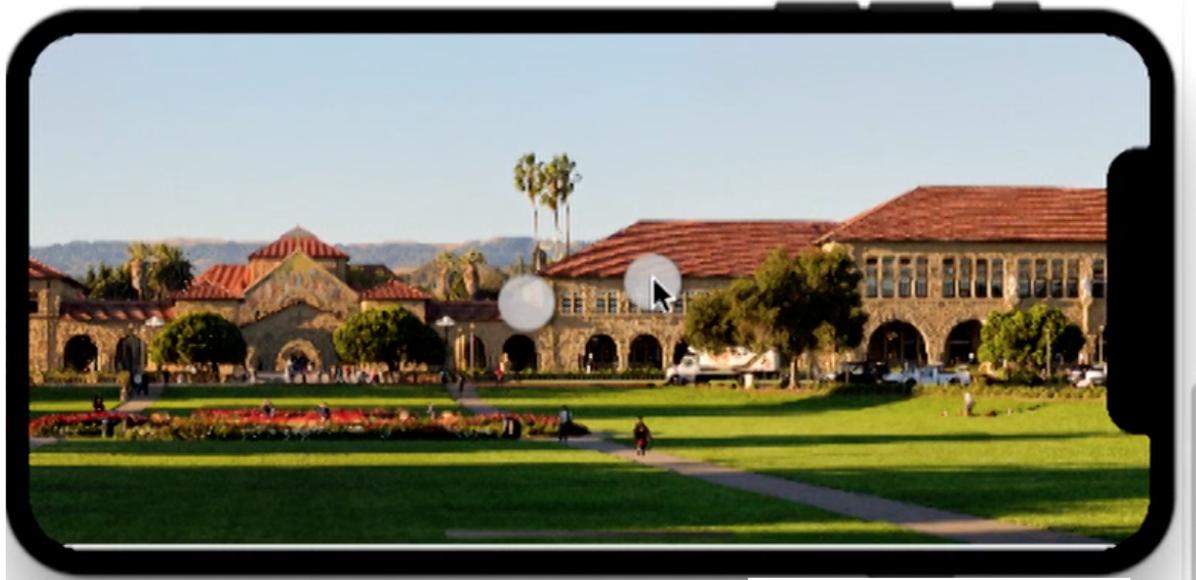


Я могу уменьшать масштаб. Возможно, изображение будет выглядеть лучше, если уменьшать масштаб. Это реально очень легко делать.

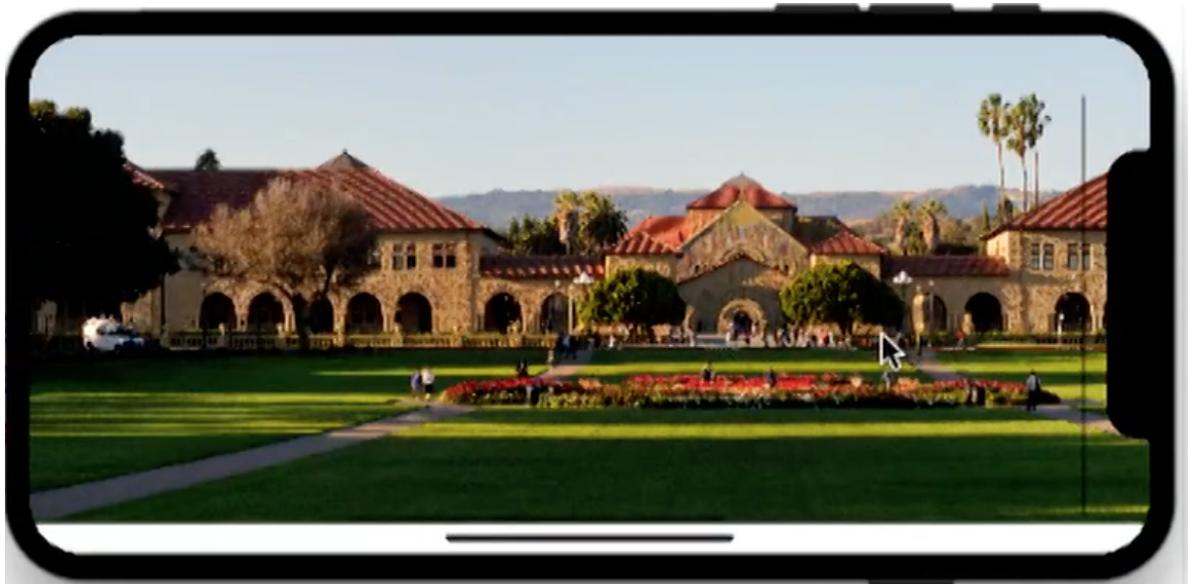


Я надеюсь это дало вам полное представление о **ScrollView** с точки зрения того, как добавлять **subviews**, управлять его **contentSize**, выполнять масштабирование (**zooming**) и использовать прочие его возможности. Все достаточно просто, понятно и легко в использовании.





iPhone X - iOS 11.0



iPhone X - iOS 11.0

В следующий раз мы получим все эти огромные изображения через интернет. Надеюсь, что я смогу заставить мой новый Mac работать через интернет, и это будет значительно медленнее. Но мы не хотим блокировать наш UI ожиданием завершения выборки изображения из интернета, мы хотим, чтобы наш UI был высоко интерактивным. Для этого нам необходимо использовать многопоточность (**multithreading**). Это будет основная тема в начале Лекции, которая состоится в Среду.

Увидимся.

----- 74-ая минута лекции -----

----- КОНЕЦ ЛЕКЦИИ 9-----

На НАЧАЛО ЛЕКЦИИ 9