



Stanford CS193p

Developing Applications for iOS
Winter 2017



CS193p
Winter 2017

S T A N F O R D U N I V E R S I T Y

S c h o o l o f E n g i n e e r i n g

Developing Apps for iOS CS193P Winter 2017:

Разработка iOS 10 приложений с помощью Swift

Л е к ц и я 1: Introduction to iOS 10, Xcode 8 and Swift 3. January 9, 2017

Профессор Пол Хэгертி (Paul Hegarty)

[Независимая, неавторизованная транскрипция ©
2017 Paul Hegarty

НАЧАЛО ЛЕКЦИИ

----- 5 -я минута лекции -----

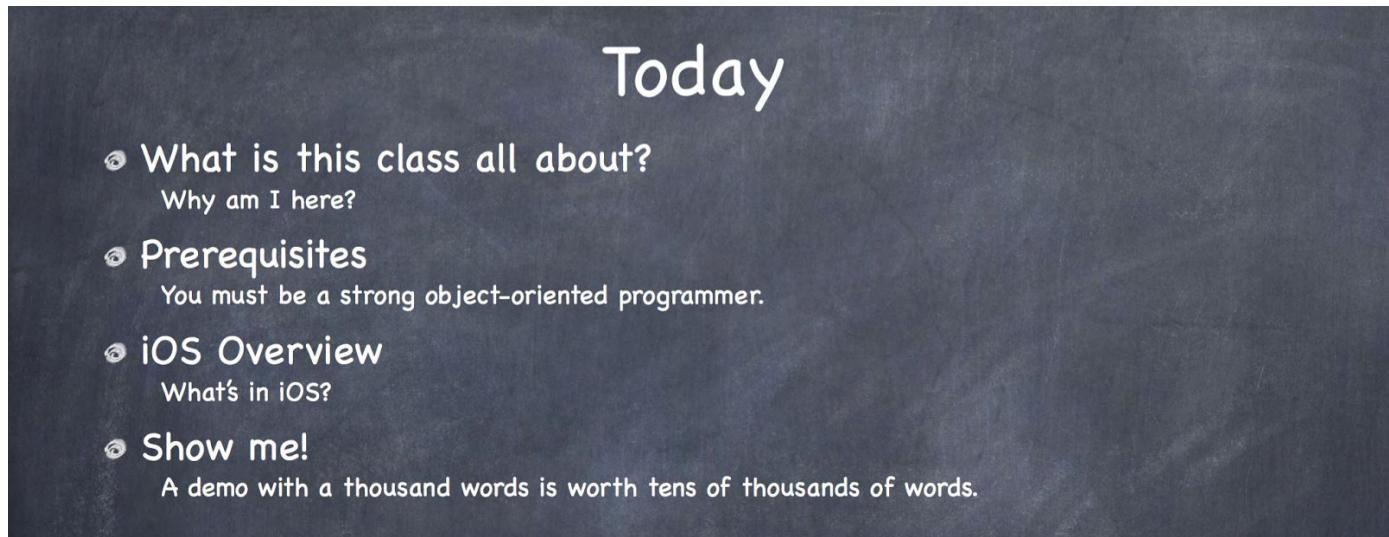
----- 10 -я минута лекции -----

----- 15 -я минута лекции -----

----- 20 -я 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 -----
----- 75 -ая минута лекции -----
----- 80 -ая минута лекции -----
----- Конец Лекции -----

Добро пожаловать на курс Стэнфорда **CS193P**. Это курс “Разработка iOS 10 приложений на Swift” **Winter 2017**, и сегодня мы поговорим о том, что собой представляет этот курс и каковы предварительные требования для его успешного прохождения. Очень быстро я расскажу о том, что собой представляет iOS. Затем я полностью погружусь в очень большой и продолжительный демонстрационный пример.



Сегодня

- **О чем этот курс?**
Почему я здесь?
- **Предварительные требования**
Вы должны быть уверенным объектно-ориентированным программистом.
- **Обзор iOS**
Что находится в iOS?
- **Покажи мне!**
Лучше один раз увидеть демонстрационный пример, чем сто раз услышать.

В этом демонстрационном примере вы достаточно быстро увидите, что такое реальная разработка

на iOS, и сможете решить, подходит вам это или нет.

Мы будем изучать, как создавать замечательные iOS приложения. Чем они так замечательны?

По ряду причин.

Во-первых, они могут располагаться в вашем кармане или рюкзаке, и вы можете показывать их своим друзьям на вашем iPhone. Большинство этих приложений настроены на работу в интернете и являются приложениями типа социальных сетей, так что использование их доставит вам много удовольствия. Кроме того, если вы решили превратить свое приложение в продукт, то сможете очень легко разместить его в **AppStore** для своих потребителей. Вам даже не нужно упаковывать его в коробку и размещать на полке какого-то магазина. Ваши потребители могут добраться до него очень быстро.

Вы также увидите, что в настоящее время можно очень легко и очень быстро разработать очень сложное iOS приложение. Так что вы практически мгновенно получаете удовлетворение от разработки iOS приложений. Для тех из вас, кто изучает Информатику (Computer Science) есть возможность увидеть в реальной жизни системы, построенные с использованием объектно-ориентированного программирования. И не только объектно-ориентированного программирования, но мы также будем использовать в этом курсе базы данных, графику, некоторые мультимедийные вещи, многопоточность, анимацию, работу с интернетом и многое другое. И вы увидите как все это функционирует вместе в реальной среде. Возможно, что вы уже прошли многие курсы по этим отдельным направлениям и у вас возникло ощущение оторванности изучаемого предмета от реального мира. На этом курсе вы увидите все это в действии. Поэтому этот курс - своего рода "синтез" других курсов для создания реальных приложений.

What will I learn in this course?

• How to build cool apps

Easy to build even very complex applications.

Result lives in your pocket or backpack!

Very easy to distribute your application through the AppStore.

Vibrant development community.

• Real-life Object-Oriented Programming

The heart of Cocoa Touch is 100% object-oriented.

Application of MVC design model.

Many computer science concepts applied in a commercial development platform:

Databases, Graphics, Multimedia, Multithreading, Animation, Networking, and much, much more!

Numerous students have gone on to sell products on the AppStore.



CS193p
Winter 2017

Что я узнаю на этом курсе?

- Как построить замечательные приложения

- Легко создать даже сложные приложения
 - Результат будет жить в вашем кармане или рюкзаке
 - Очень легко распространять приложение через AppStore
 - Очень отзывчивое сообщество разработчиков
- **Объектно-ориентированное программирование в реальной жизни**
 - Ядро Cocoa Touch является 100% объектно-ориентированным
 - Применение MVC шаблона конструирования
 - Представлены многие концепции информатики, применяемые в коммерческих платформах разработки: Базы данных, Графика, Мультимедиа, Многопоточность, Анимация, Сети и многое, многое другое!
 - Многие студенты смогут разместить приложения в AppStore для продажи

Предварительные требования в этом курсе действительно важны. Но они достаточно простые - вы должны иметь очень хорошие навыки объектно-ориентированного программирования.

Prerequisites

• Prior Coursework

Object-Oriented Programming experience mandatory.
CS106A&B (or X) required & CS107 or CS108 or CS110 also (at a minimum) required.
(or equivalent for non-Stanford undergrads)

Предварительные требования

- **Предварительные курсы**

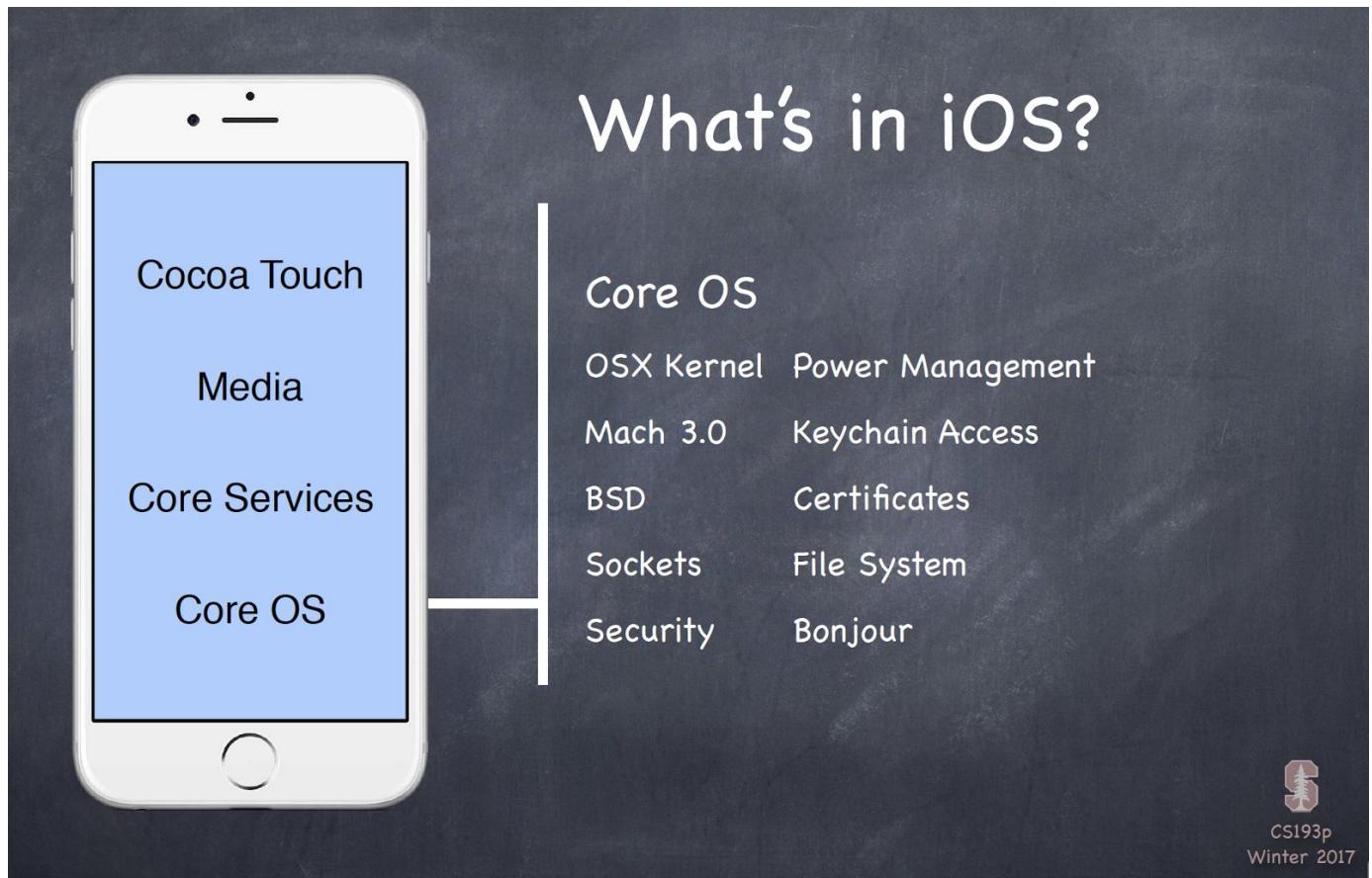
Опыт Объектно-Ориентированного программирования обязателен
Обязательны CS106A&B (или X) и CS107 или CS108 или CS110 желательны (или эквивалентные не Стэнфордским градациям)

Я не буду на этом курсе учить вас объектно-ориентированному программированию, я буду предполагать, что вы полностью и в полном объеме его знаете. И не только знаете, но и имеете некоторый опыт практического использования. Именно поэтому Стэнфордские курсы **CS106A&B** или **CS106X** - это обязательное предварительное требование. Вам абсолютно необходимо пройти оба этих курса, оба они являются объектно-ориентированные курсами и кроме того, вы получите на этих курсах небольшой опыт объектно-ориентированного программирования.

Затем я должен быть уверен, что у вас есть опыт программирования как такового или вы уже что-то программируете за пределами школы, или вы уже прошли курсы **CS107** или **CS108** или **CS110**. Если вы возьмете курс **CS108**, то вы будете прекрасно подготовлены для нашего курса, так как это курс объектно-ориентированного программирования. Если у вас есть возможность взять курс **CS108**, то пройдите сначала его, а затем уже наш курс, когда он будет предложен в

следующий раз.

Что содержится в **iOS**, каковы ее части? Их огромное количество и они не поддаются классификации, но я все же разделил их на 4 слоя.



Вот группы, на которые Apple часто подразделяет iOS. Всего 4 слоя:

- Core OS - очень близка к hardware,
- Core Services - объектно-ориентированная надстройка над Core OS,
- Media - потому что для устройства, особенно iPod с iPhone, да и для iPad, это важный раздел
- Cocoa Touch - слой пользовательского интерфейса (UI)

От API, близкого к **hardware**, наш путь лежит наверх, к API слоям, близким к пользователю, а точнее к пользовательскому интерфейсу (UI): кнопки (buttons), ползунки (sliders) и другие подобные вещи.

Но реально разделение iOS на слои реализуется не очень строго. И все из-за слоя **Media**. Часть элементов этого слоя может находиться в более низком слое, близком к **hardware**. А часть, например, видео, - в более высоком слое, близком к пользователю.

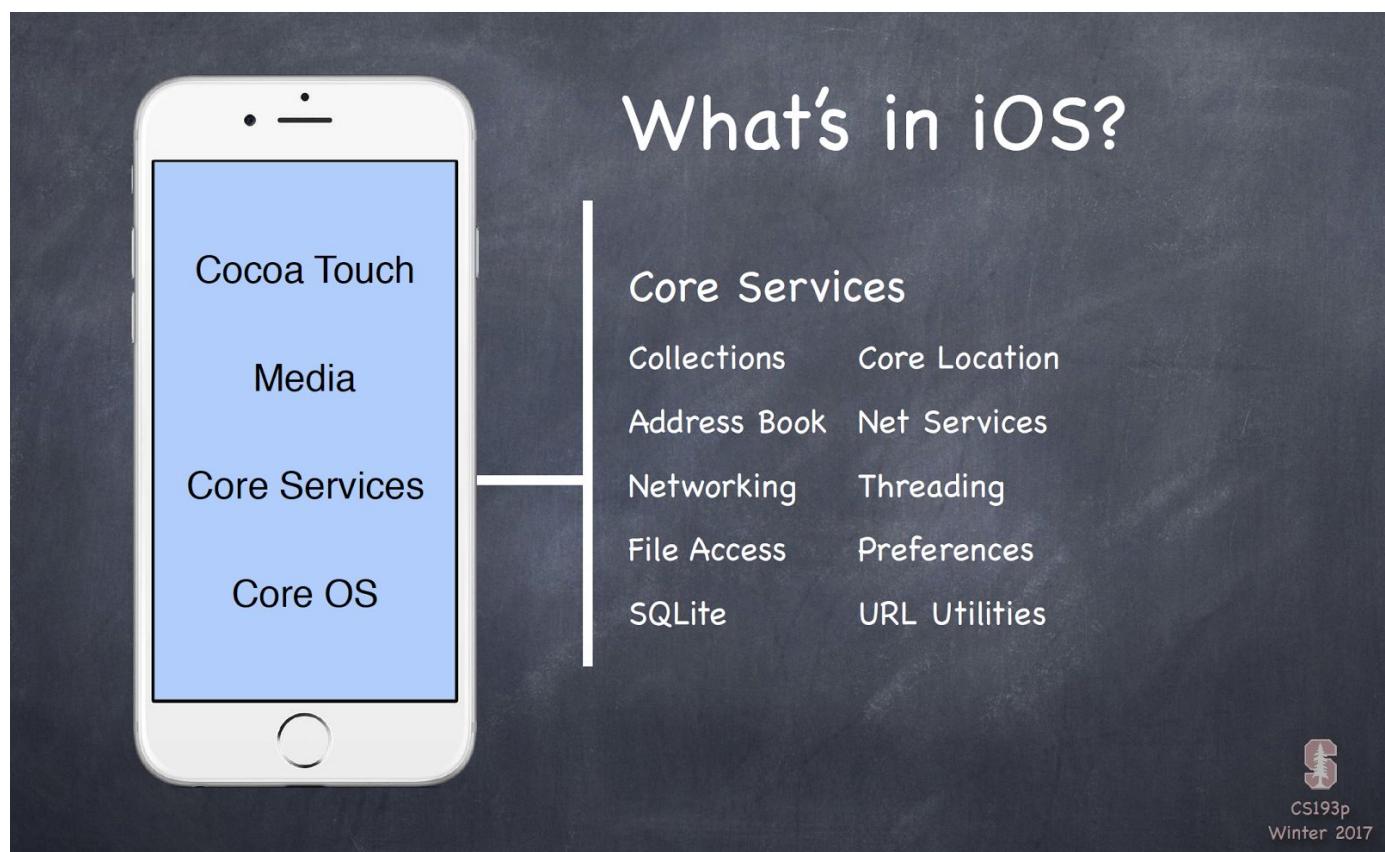
В любом случае, это дает только основное представление о том, из чего состоит iOS.

Давайте посмотрим из чего состоит каждый слой.

В слое **Core OS**, который ближе всех к **hardware**, находится ядро **UNIX**. Большинство людей даже не представляют, что их телефон работает под управлением операционной системы **UNIX**, да, это формат **BSD UNIX**, именуемый **Mach** - та же основная технология, что и в Mac OS X, но вы можете

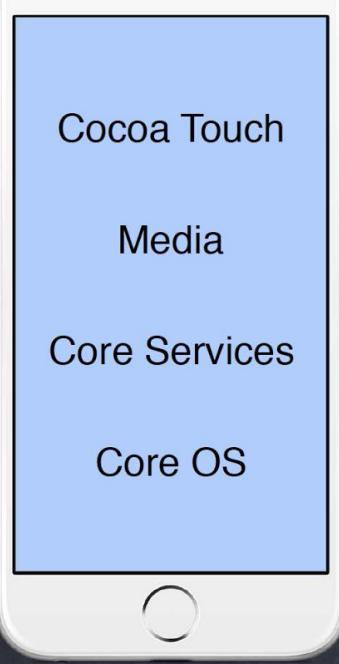
делать под ней то же, что и под **UNIX**: полная многозадачность, файловая система **Unix**, все, что необходимо. Большая часть **Core OS** такая же, как и в MacOS, но оптимизированная для мобильных устройств с их ограниченной батареей. Вы получаете всю ее мощь **UNIX** при разработке iOS приложений. Она включает в себя все, что показано на слайде. Но есть вещи, специфические для мобильных устройств - например, управление питанием (**Power Management**), что очень важно для мобильных устройств. Компонент **Keychain Access** делает недоступными некоторые вещи на iPhone, если им не разрешено и т.д. Но здесь **UNIX** присутствует во всем своем величии на самом нижнем уровне. На этом уровне программные **API** не являются объектно-ориентированными, сам **UNIX** написан на C, так что преимущественно это программирование на C. На этом уровне мы не будем программировать на этом курсе. Наш курс полностью ориентирован на объектно-ориентированное программирование.

Давайте поднимемся на один уровень абстракции выше, **Core Services**.



Иногда люди ссылаются на этот слой как на **Foundation**, но здесь есть много других вещей помимо **Foundation**. И есть объектно-ориентированный слой, выстроенный поверх всех этих вещей. Вы можете работать с "сетью" (networking), с многопоточностью (multithreading), с файловой системой, используя объектно-ориентированное **API**. Но в этом слое еще нет никаких элементов пользовательского интерфейса (UI). Это базовый слой и он ближе к **hardware**. Я буду учить вас программировать, используя возможности этого слоя, потому что вам необходимо делать то, о чем я упомянул выше. И мы будем много на нем программировать и изучать.

Следующий слой - **Multimedia**. Это огромный слой, который включает в себя 3D графику, различные аудио фреймворки, обработку изображений, видео и много всего другого.



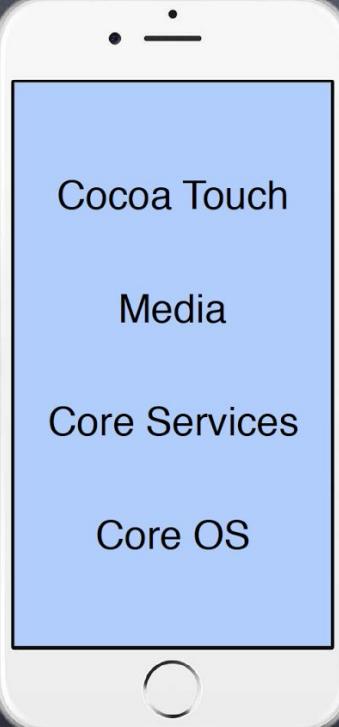
What's in iOS?

Media	
Core Audio	JPEG, PNG, TIFF
OpenAL	PDF
Audio Mixing	Quartz (2D)
Audio Recording	Core Animation
Video Playback	OpenGL ES

CS193p
Winter 2017

К сожалению, у меня нет времени останавливаться на этом слое подробно, хотя большая часть того, что делает iOS устройство, находится именно на этом уровне.

И, наконец, на топовом уровне находится слой, в котором мы будем проводить большую часть нашего времени, - это **Cocoa Touch**.



What's in iOS?

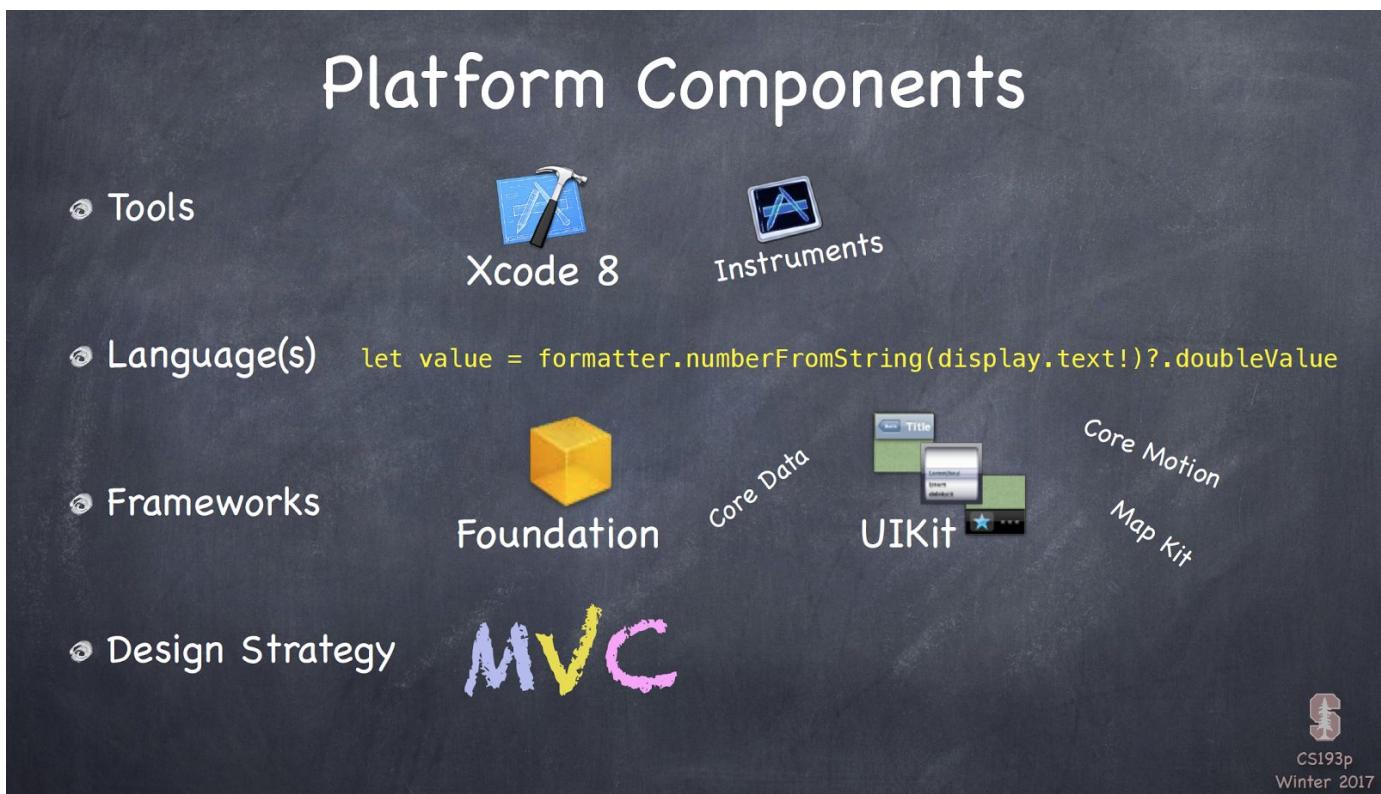
Cocoa Touch	
Multi-Touch	Alerts
Core Motion	Web View
View Hierarchy	Map Kit
Localization	Image Picker
Controls	Camera

CS193p
Winter 2017

Это объектно-ориентированные **APIs** для построения пользовательского интерфейса (**UI**). Именно на этом слое находятся кнопки (**Button**), текстовые поля (**TextField**) и другие элементы пользовательского интерфейса (**UI**), но есть и очень мощные объекты, такие, как “карта Мира” (**Map**). В слое **Cocoa Touch** объект “карта Мира” (**Map**) представляет собой целое **Maps** приложение на iOS устройстве, так что в своем приложении вы можете разместить “карту Мира” в любой прямоугольной области, вообще не выполняя никакой работы. Это очень мощный объект. Так что большую часть времени мы будем проводить на слое **Cocoa Touch**, создавая пользовательский интерфейс для наших приложений.

Это очень грубый обзор iOS. Невозможно рассказать об iOS за 2 минуты, но у вас уже создалось определенное представление о том, что мы будем делать на этом курсе.

Мы будем использовать все представленные ниже компоненты для выполнения разработки приложений:



Во-первых, это инструменты (**Tools**), у нас несколько инструментов, но реально **Xcode 8** - это то, что будет использоваться в 99% случаев на этом курсе. Это и редактор, и отладчик, у него есть управление версиями исходного кода, все это находится в **Xcode 8**. Есть еще небольшое дополнительное приложение **Instruments**, которое позволяет измерять производительность и много всего другого, но в основном мы будем работать в **Xcode 8**.

----- 5 -ая минута лекции -----

Во-вторых, я буду учить вас новому языку программирования. Все вы изучаете Информатику (Computer Science) и знаете, что изучение различных языков программирования - это очень ценный навык. Не потому, что есть необходимость использовать все языки программирования, возможно, некоторые из них вы вообще никогда не будете использовать. А из-за возможности увидеть, как разработчики языка программирования выбирают те или иные синтаксические конструкции и дают

в ваше распоряжение те или иные возможности; это очень ценно. Я думаю, у вас будет возможность увидеть это на этом курсе. Swift - это великолепный язык, он был изобретен всего два-три года назад и вобрал в себя все самое лучшее из других языков программирования. Я собираюсь провести для вас блицкриг по изучению этого языка программирования в течение первой пары недель.

Третьим элементом платформы являются **Frameworks**. iOS наполнено множеством frameworks и мы рассмотрим их. Основной framework - это **UIKit**. Он связан с пользовательским интерфейсом, это в основном библиотеки объектов, которые вы используете как строительные блоки для создания вашего приложения: кнопки, метки, текстовые поля и другие элементы **UI**. Фреймворк **Foundation** - это библиотека слоя **Core Services**, о котором я говорил, но есть еще множество других : **CoreMotion** - для отслеживания движений iOS устройства с помощью гироскопа и акселерометра, **CoreData** - для объектно-ориентированной базы данных, которую вы будете использовать в Задании 5. Я говорил вам о картах - это фреймворк **MapKit** для работы с картами. Я покажу вам как можно больше **frameworks**, но их слишком много, чтобы рассмотреть подробно в течение 10 недель.

И, наконец, последняя, но действительно очень важная часть платформы - это стратегия проектирования iOS приложений, которая называется **MVC (Model View Controller)**. Она используется также и на других платформах. Кто из вас когда-либо на какой-то платформе использовал **MVC** прежде? Приблизительно половина.

Так как только половина из вас знакома с **MVC**, то первую половину лекции в Среду я буду рассказывать о том, что представляет собой **MVC** и как применить эту методологию в разработке iOS приложений. Это очень важная вещь, и в 100 % случаев мы должны использовать **MVC** для разработки iOS приложений. Нет других способов это сделать. В противном случае вы будете “плыть против течения”, против самой сути iOS... И закончите полной неразберихой в своем приложении. Так что мы будем изучать методологию **MVC**.

Демонстрационный пример, который я вам сейчас буду показывать, - это Калькулятор (Calculator). Это замечательное приложение. У него очень простой пользовательский интерфейс (**UI**), но внутри имеется достаточно сложная вычислительная часть, достаточно насыщенная, чтобы показать вам, как работает **MVC**, а также различные возможности языка программирования. Но не настолько сложный, чтобы не сделать полный Калькулятор от начала до конца за две лекции.

Ниже представлен такой слайд, к которому вы, вернувшись после Лекции сегодня, должны убедиться, что вы все поняли в этом демонстрационном примере.

Demo

⌚ Calculator

All this stuff can be very abstract until you *see* it in action.

We'll start getting comfortable with Swift 3 and Xcode 8 by building something right away.

Two part demo starting today, finishing on Wednesday.

⌚ Today's topics in the demo ...

Creating a Project in Xcode 8

Building a UI

The iOS Simulator

print (outputting to the console using \() notation)

Defining a class in Swift, including how to specify instance variables and methods

Connecting properties (instance variables) from our Swift code to the UI (outlets)

Connecting UI elements to invoke methods in our Swift code (actions)

Accessing iOS documentation from our code

Optionals (?), unwrapping implicitly by declaring with !, and unwrapping explicitly with ! and if let)



CS193p
Winter 2017

Демонстрационный пример

● Калькулятор

Все кажется абстрактным до тех пор, пока вы не увидите это в действии.

Мы начнем осваивать Swift и Xcode 8 путем создания чего-нибудь прямо сейчас.

Демонстрационный пример состоит из 2-х частей. Сегодня - первая часть, а в Среду - окончательная вторая часть

● Темы сегодняшней части демонстрационного примера...

Создание проекта Project в Xcode 8

Построение пользовательского интерфейса (UI)

iOS симулятор

print (вывод на консоль, используя \() нотацию)

Определение класса в Swift, включая определение переменных экземпляра класса и методов

Связывание свойств (переменных экземпляра класса) в Swift коде с элементами пользовательского интерфейса UI (outlets)

Привязка элементов UI к методам в коде Swift (actions)

Доступ к iOS документации из кода

Optionals (?), неявное развертывание путем декларирования со знаком !, явное развертывание с помощью ! и if let)

Итак, я не собираюсь возвращаться к слайдам после демонстрационного примера, так что позвольте мне рассказать, что вас ждет в ближайшее время.

Coming Up

• Today

Reading Assignment 1 assigned (due on Wednesday next week)

• Wednesday

Calculator demo continued

MVC (Model View Controller) design pattern

Programming Assignment 1 assigned (also due on Wednesday next week)

• Friday

Using the Debugger in Xcode 8

• Monday

MLK Day - No Lecture

Что нас ожидает?

• Сегодня

Выдача Задания на чтение № 1 (выполнение к Среде на следующей неделе)

• Среда

Продолжение демонстрационного примера Calculator

Методология проектирования MVC (Model View Controller)

Выдача Задания на программирование № 1 (выполнение к следующей Среде)

• Пятница

Использование отладчика в Xcode 8

• Понедельник

Выходной - Лекции не будет

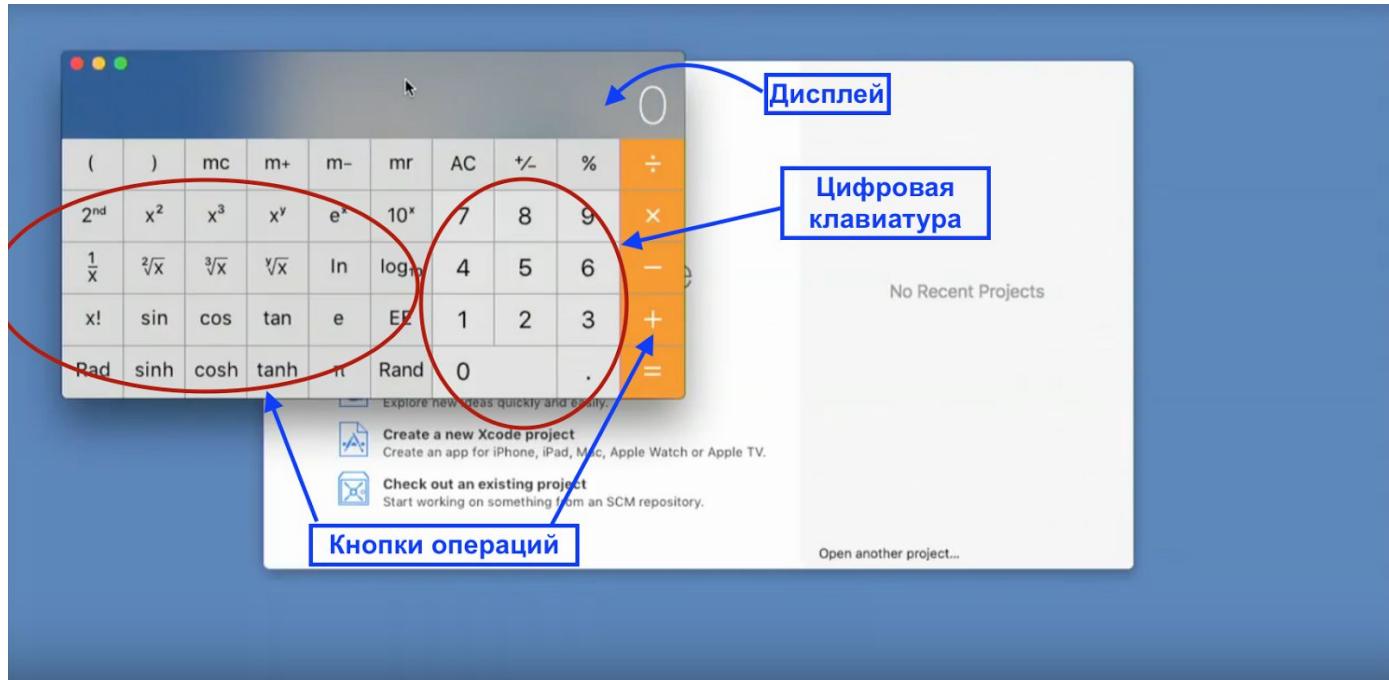
В Среду я продолжу демонстрационный пример, начатый сегодня. Но вначале я буду говорить об **MVC**, так как я буду применять MVC к построению iOS приложения Калькулятора. Вам будет выдано Задание на программирование № 1, которое нужно выполнить через неделю, в следующую Среду. Но это в значительной степени просто повторение того, что я буду делать сегодня и в Среду. У вас будет видео моего демонстрационного примера и вы можете смотреть его при выполнении Задания на программирование №1.

В Пятницу у вас необязательная секция, время и место будут объявлены дополнительно. На этом курсе каждую Пятницу, или почти каждую Пятницу, будут проводится необязательные секции, вы можете не ходить на них, если не хотите, но там всегда будет представляться очень нужная и ценная информация. Например, в ближайшую Пятницу будет рассматриваться отладчик (**debugger**), который очень нужен при написании кода. Если у вас нет опыта работы с отладчиком в Xcode 8, то вам следует посетить эту пятничную секцию.

Не забудьте, что в следующий Понедельник у нас праздник, так что Лекции не будет. Следующая

наша встреча после этой Среды состоится в Среду через неделю.

Давайте приступим к демонстрационному примеру. Как я и обещал, это будет Калькулятор.



Я показываю вам Калькулятор, который существует на Mac OS. Наш Калькулятор будет очень на него похож. В верхней части у нас будет дисплей, у нас будет цифровая клавиатура для ввода чисел и кнопки для операций. Вы можете набрать любое число, затем знак “**×**” умножения, затем “**8**” и знак равенства “**=**” для получения результата. Наш Калькулятор будет выглядеть не точно также, но похоже с учетом некоторых особенностей iOS устройств. Я вам обещаю, что все время, пока мы разрабатываем Калькулятор, мы будем проводить в **Xcode**. В настоящий момент **Xcode** - это приложение, которое вы можете получить, зайдя в **Mac AppStore** и загрузив его на ваш компьютер. Оно бесплатное. Когда вы первый раз запускаете Xcode, появляется следующий экран "заставки":



Все ваши проекты начинают аккумулироваться справа в серой области. У нас нет этого списка, потому что мы только что начали наш семестр. Но каждую неделю там будет появляться все больше и больше проектов.

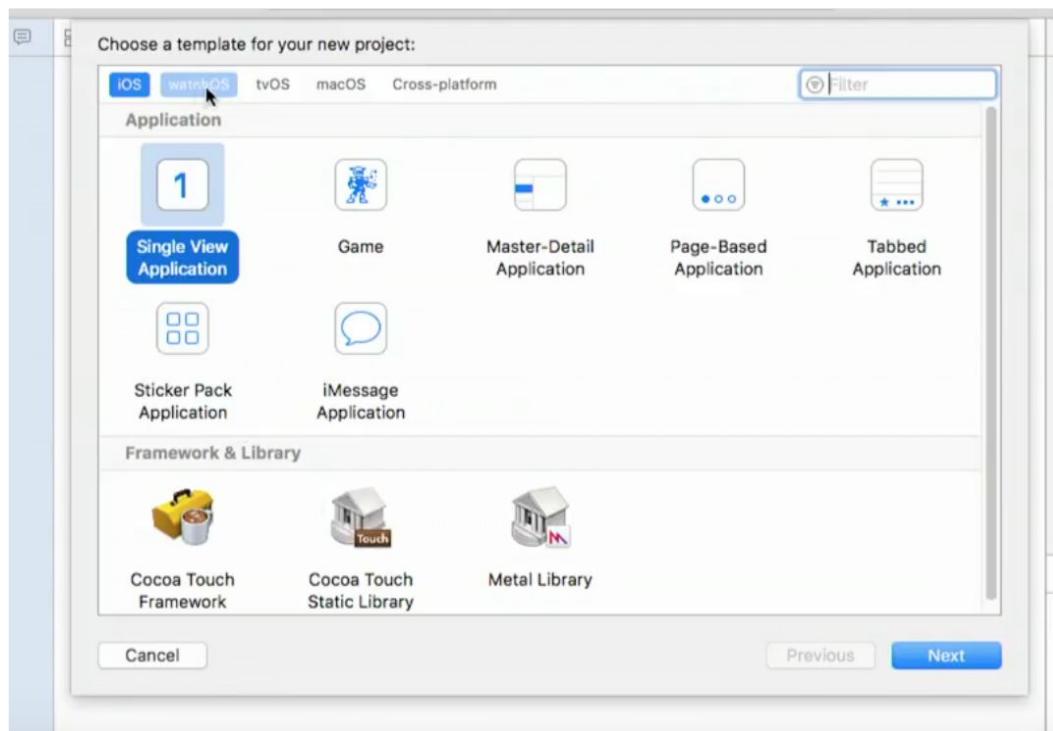
У вас есть три варианта попасть в **Xcode**.

Первый Вариант - вы можете использовать **playground**, и я покажу вам **playgrounds** в Среде. Playground дает вам возможность писать код без создания приложения. Вы можете вызывать любые **APIs**, размещать любые вещи на экране, рисовать и т.д. Это своего рода "игровая площадка" для **iOS** программирования.

Третий вариант, в самом низу списка, - вы можете сделать **Check out** существующего проекта в системе управления версиями исходного кода (Source Code Management). Мы еще не говорили о системе управления версиями кода проекта SCM, хотя, возможно, мы посвятим этой теме одну из наших пятничных секций.

Второй средний вариант создает новый проект в **Xcode**, новое приложение и именно этот вариант нам сейчас и нужен. Мы будем создавать приложение "с нуля". Большинство моих демонстрационных приложений предполагают создание приложения "с нуля". Потому что для вас этот способ будет легче для восприятия, ибо не будет ничего "волшебного", чтобы происходило "за кулисами", когда некоторые вещи появлялись бы неизвестно откуда. Кликаем на втором варианте. Вы видите **Xcode** хочет создать приложение для меня и задает мне некоторые вопросы. Он хочет знать какого типа приложение я хочу создать.

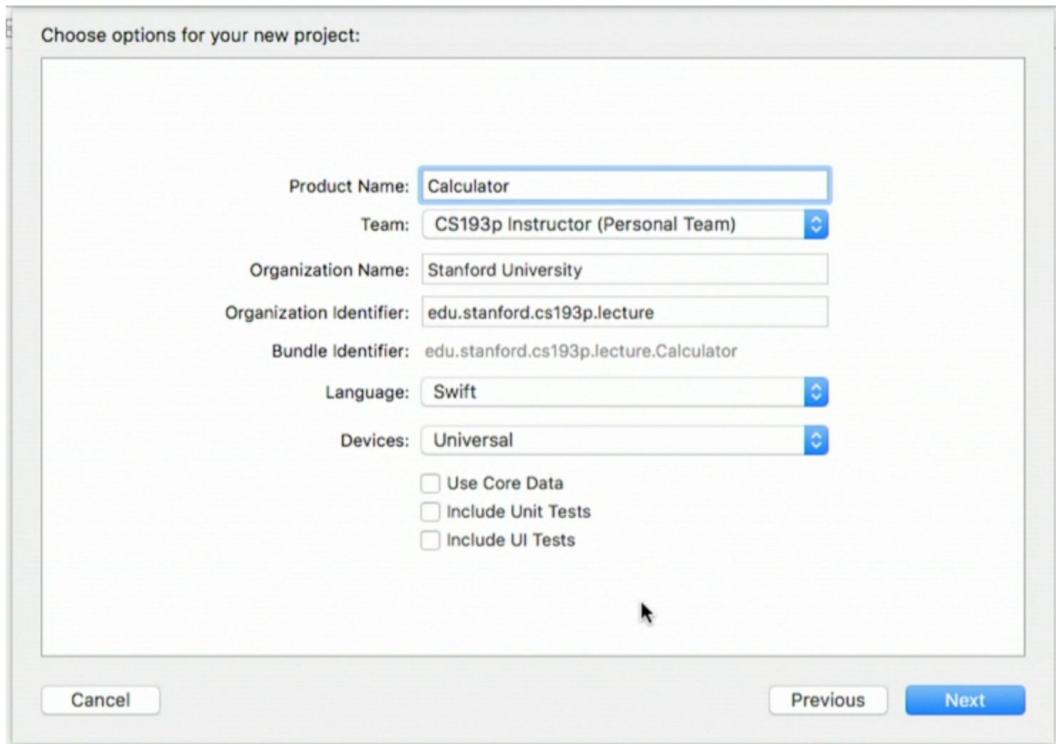
----- 10 -ая минута лекции -----



Я хочу создать **iOS** приложение. Убедитесь, что вы кликнули **iOS Application**. Не **watchOS**, не **tvOS** и не **macOS** приложение. Нам не нужно ни одно из вышеупомянутых приложений. Нам нужно iOS приложение.

Фактически, мы в большинстве случаев будем выбирать шаблон **Single View Application**. Это

простейший шаблон и дает наименьшее количество кода при старте, другие шаблоны имеют значительно больше готового кода. Я действительно хочу показать вам, как самим писать код вместо того, чтобы иметь уже готовый код в шаблоне. Поэтому мы выбираем шаблон **Single-View Application** и кликаем “**Next**” кнопку.



Теперь от нас еще хотят узнать несколько особенностей относительно нашего приложения и самое главное находится в самом верху - это имя приложения (**Product Name**). Приложение, которое мы будем создавать сегодня, - это Калькулятор. Поэтому я назову приложение **Calculator**.

Во второй строке нужно заполнить поле **Team** - это команда разработчиков, работающих над этим проектом. Команда может состоять из одного человека, которым являетесь вы. Вполне возможно, что при запуске **Xcode** не будет выпадающего списка в строке **Team**, а будет кнопка с заголовком “**Add Account**” или что-то типа “**Add Team**”. Вы кликаете на ней и все, что вам нужно сделать, это указать любой **AppleID**, который можно получить совершенно бесплатно. Вам придется пройти через диалог и вы добавите себя в качестве **Team**.

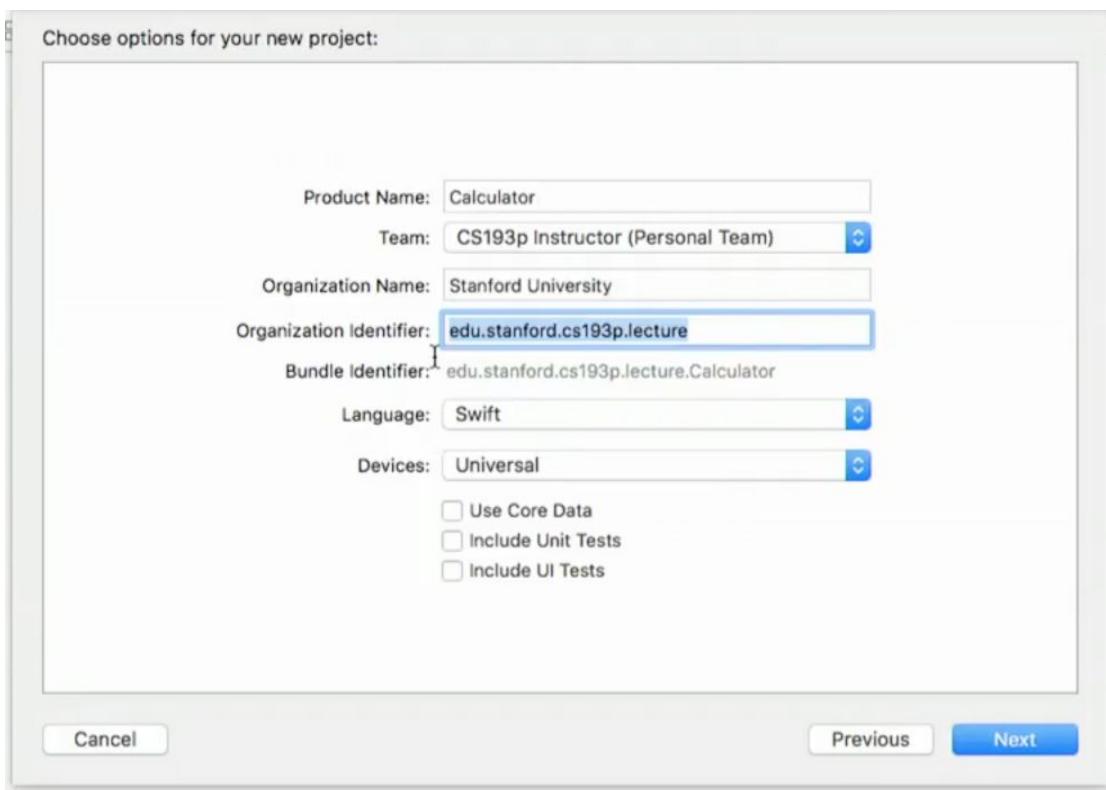
Есть еще имя организации (**Organization Name**), которое может быть любым.

Имя организации появляется только как copyright символ (права авторства) в верхней части вашего исходного кода и все.

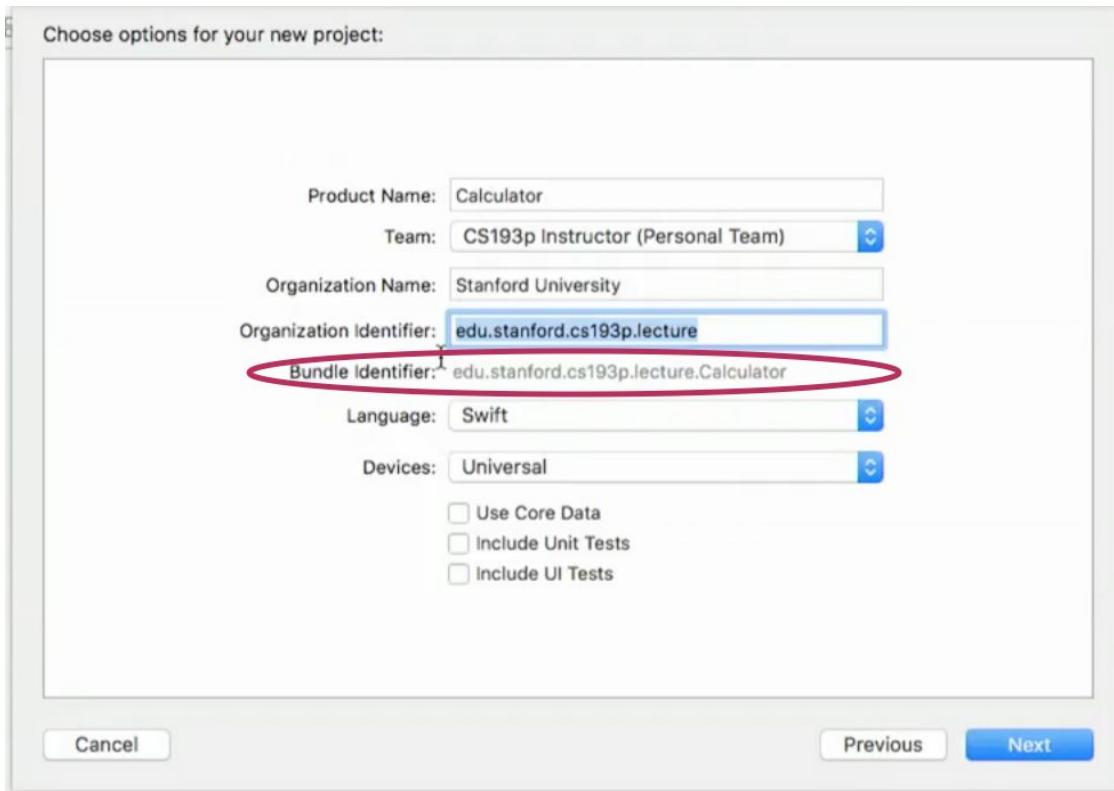
Но следующая четвертая строка имеет поле **Organization Identifier**, которое супер важно и представляет собой ВАШ уникальный идентификатор как разработчика.

Я настоятельно рекомендую использовать `edu.stanford.cs193p` + ваш SUNet ID. Если вы разместите это, то я почти наверняка гарантирую уникальность, если вы - студент Стэнфорда.

Если вы не являетесь студентом Стэнфорда и изучаете курс по iTunes U, то можете выбрать что-то уникально идентифицирующее вас. Хорошим стилем является использование здесь **обратной DNS** нотации, которая, надеюсь, также будет хорошо работать в ваших обстоятельствах.

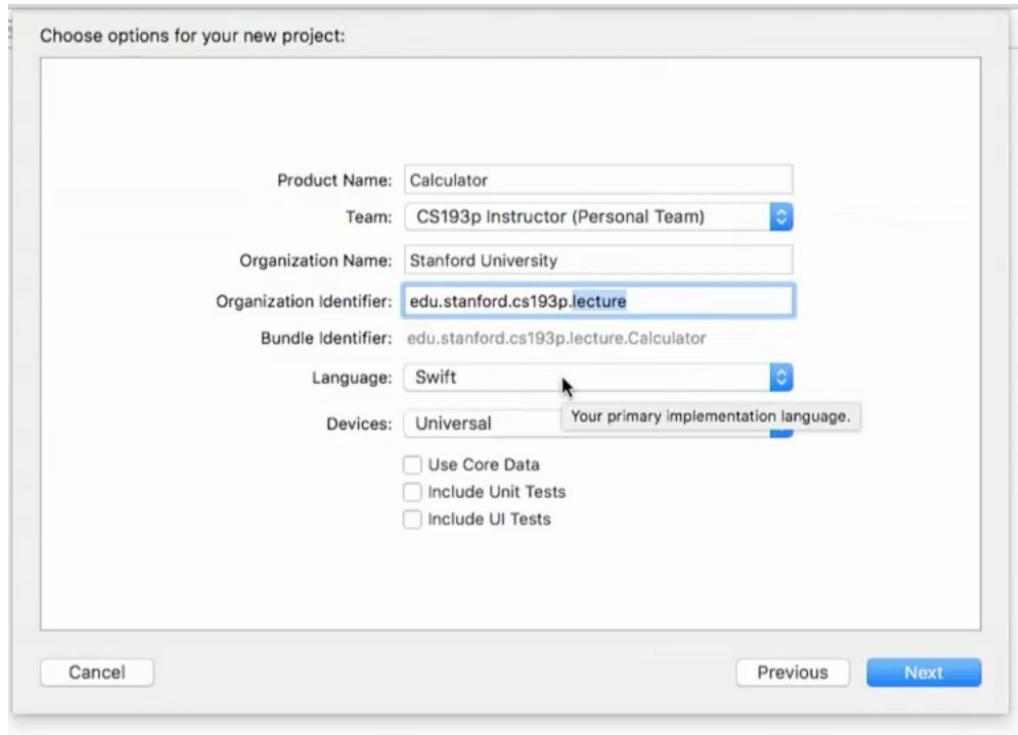


Путем комбинации уникального идентификатора разработчика и имени приложения создается уникальный идентификатор для вашего приложения, который автоматически размещается в поле **Bundle identifier**.

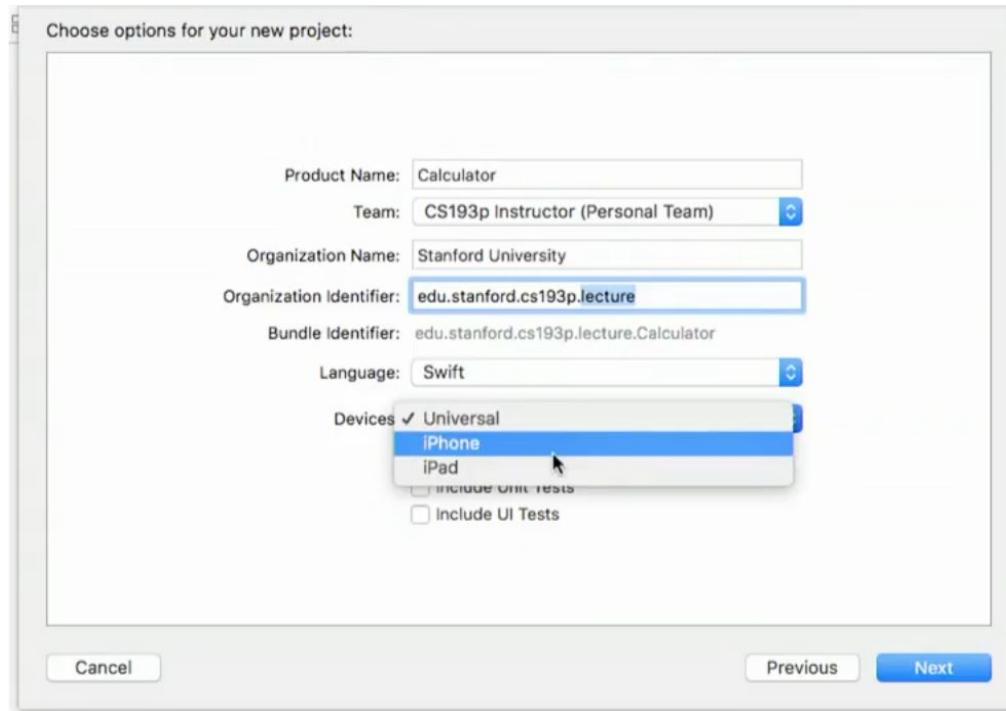


Затем вы должны выбрать язык, на котором будете программировать, и этим языком будет **Swift**. iOS изначально была написана на другом языке, который называется **Objective-C**. Оказывается, вы можете использовать **Objective-C** и **Swift** в одном и том же приложении. Они используют тот же самый базовый iOS **API**. Так что все, что вы изучите на этом курсе на **Swift**, вам пригодится, если вы решите продолжить изучать **Objective-C**. Потому что фактически эта та же самая кодовая база,

хотя и не тот же самый **API**. Swift был разработан так, чтобы быть максимально совместимым с **APIs** языка **Objective-C**. В последнюю пару лет был существенно улучшен и **Objective-C** с целью вобрать в себя некоторые продвинутые возможности, которые имеет **Swift**.

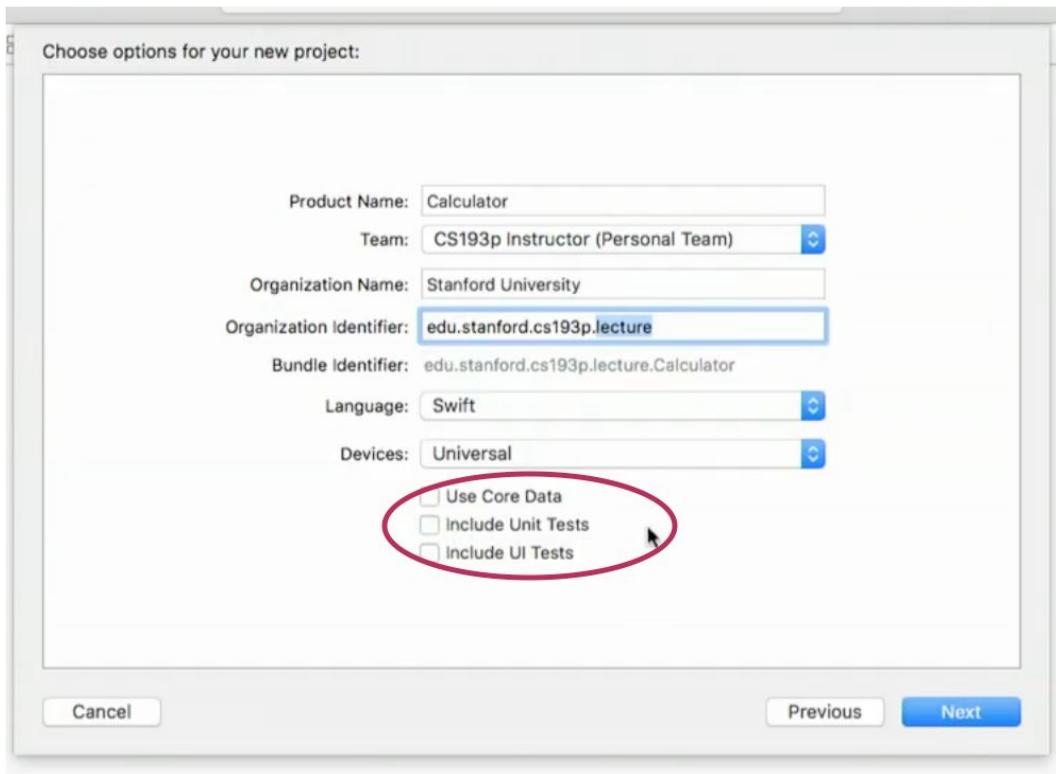


В поле **Device** мы должны указать, хотим ли мы создать калькулятор только для **iPhone**, или только для **iPad** или у нас будет универсальный (**Universal**) калькулятор, который будет работать как на iPad, так и на iPhone.



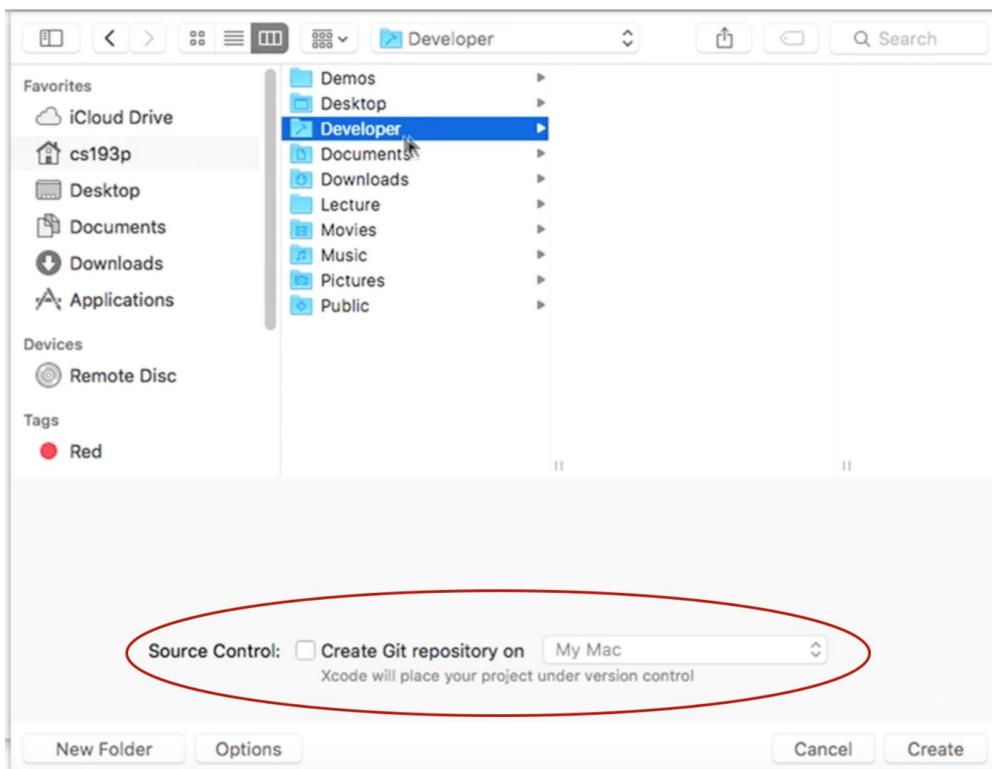
Приложение, которое мы будем создавать будет универсальным (**Universal**), то есть оно будет работать как на iPhone, так и на iPad. Первую пару недель оно будет работать только на iPhone, но в дальнейшем мы добавим также поддержку iPad.

Мы не будем в нашем приложении использовать базу данных Core Data, но она понадобиться нам в Задании 5.

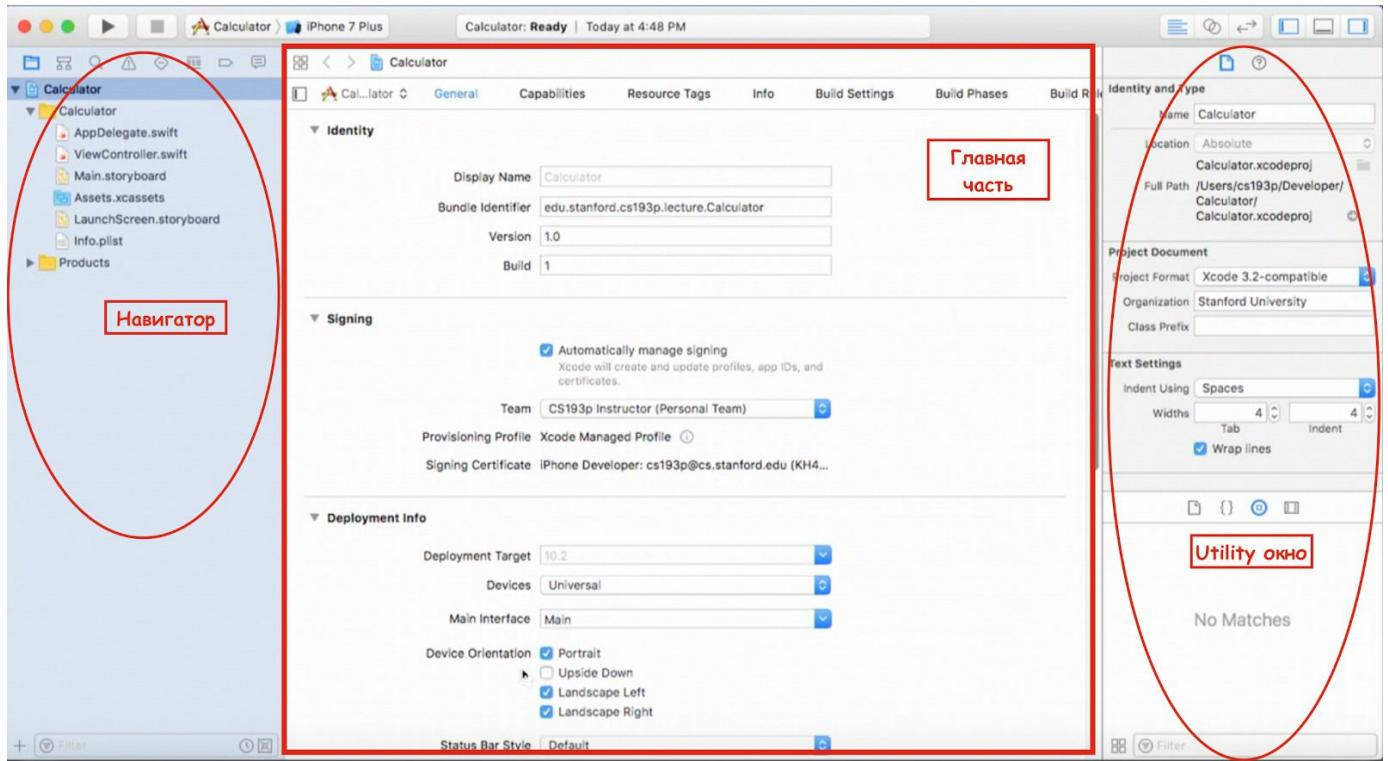


Тестирование супер важно. Я надеюсь, что у нас будет пятничная секция, посвященная тестированию, особенно тестированию пользовательского интерфейса (UI), это действительно великолепный фреймворк, надеюсь, что у нас будет шанс показать вам его.

Теперь я кликаю **Next**, и меня спрашивают, где я хочу разместить свой проект? Я размещаю мой проект в моей **Home** директории, в папке с именем **Developer**, и я настоятельно рекомендую вам делать то же самое. **Home /Developer** является каноническим местом для размещения приложений так что все ваши приложения можно посмотреть в одном месте.

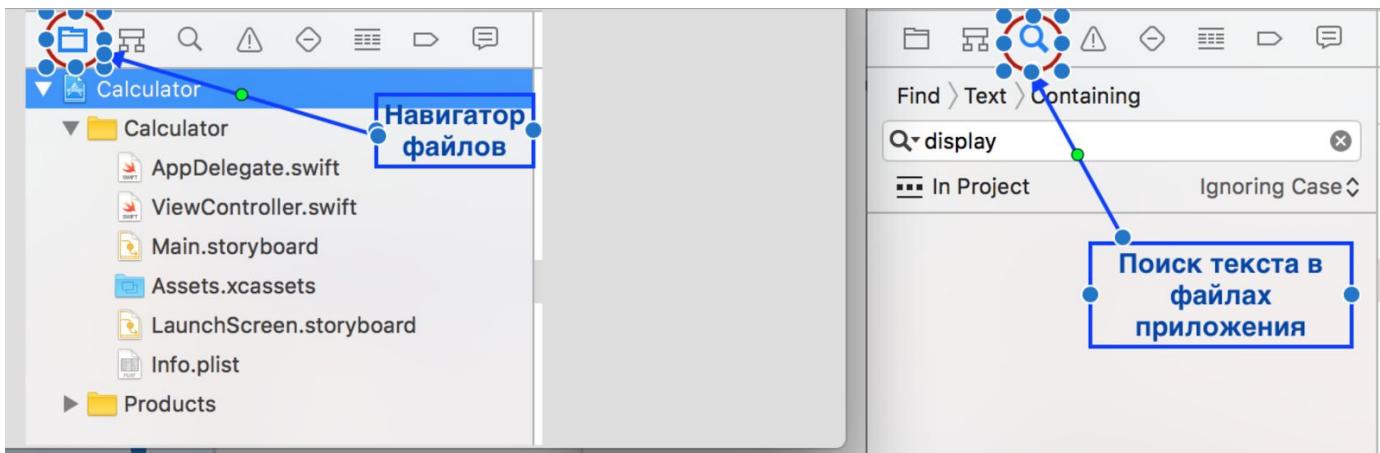


В нижней части можно указать вещи, связанные с управлением версиями исходного кода (**Source Control**), о котором я много говорил, и которое больше относится к работе команды разработчиков. Мы собираемся работать индивидуально, так что **Source Control** оставляем не помеченным. Нажимаем **Create** (Создать) для нашего первого приложения. Это наше первое iOS приложение.



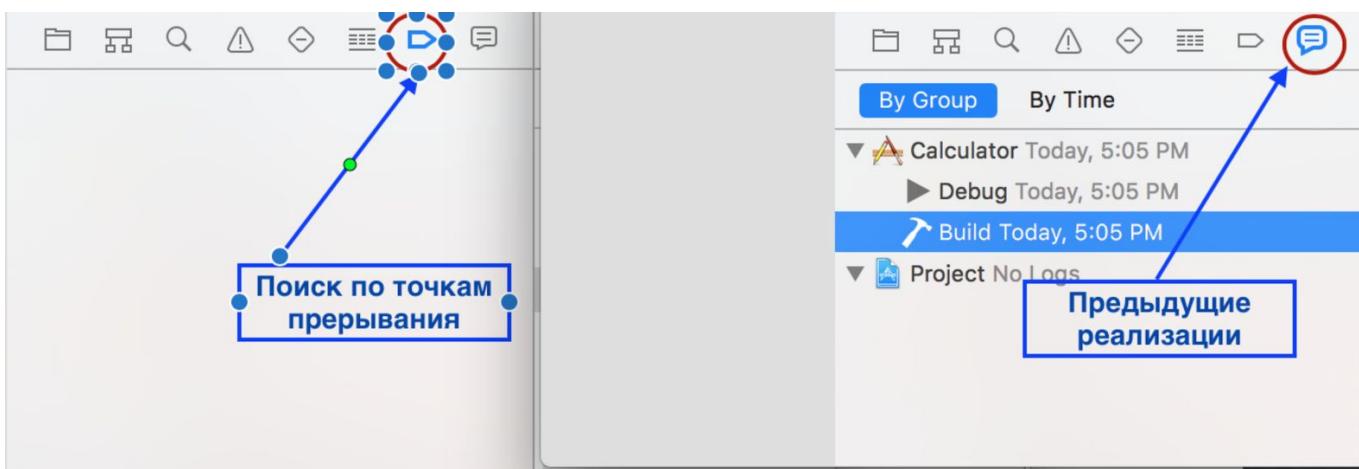
Давайте я расскажу немного об **Xcode** и объясню, как там все располагается, если вы еще не видели. Кто использовал раньше **Xcode** в других курсах или где-либо еще? Приблизительно половина. Для тех, кто еще не видел **Xcode** я дам небольшие пояснения относительно пользовательского интерфейса **Xcode**. Весь экран в **Xcode** разделен на 3 основные секции: средняя секция - основная, именно в ней мы будем редактировать исходный код и делать всю основную работу. Секция слева называется “Навигатор” (**Navigator**) и предназначена для навигации по вашему проекту. Вы можете работать с “Навигатором” в разных режимах. Основной режим — это показ файлов проекта. Вы можете организовывать файлы в различные папки, которые имеют чисто логическую цель и не совпадают с папками на диске, но это просто файлы в вашем приложении.

Вы можете осуществлять навигацию и путем поиска определенного текста в вашем приложении.



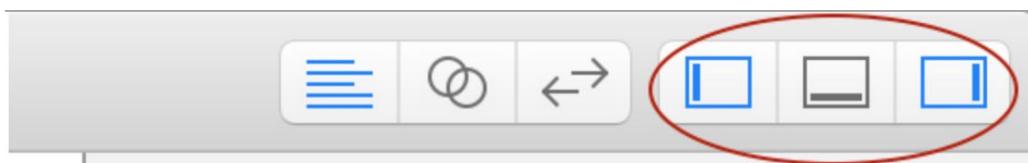
Вы можете осуществлять навигацию по точкам прерывания (**breakpoints**), если вы отлаживаете ваше приложение.

Можно вернуться назад и посмотреть на все ваши старые варианты приложения на предмет каких-либо ошибок или предупреждений, даже если вы их уже исправили, вы все равно можете на них взглянуть.



Таким образом, с помощью Навигатора вы можете следить за тем, что происходит в вашем приложении. По мере продвижения этого курса вы можете изучать различные способы навигации.

Справа в **Xcode** находится Область Утилит (**Utilities Area**). Мы будем подробно говорить об этой области через 5 минут, мы будем очень интенсивно ее использовать и вы узнаете все ее детали. Я хочу вам показать как вы можете управлять пространством, занимаемым каждой секцией. Если вы посмотрите на верхний правый угол, то увидите там эти кнопки, с помощью которых сможете показывать и скрывать правую и левую сторону пользовательского интерфейса **Xcode**.

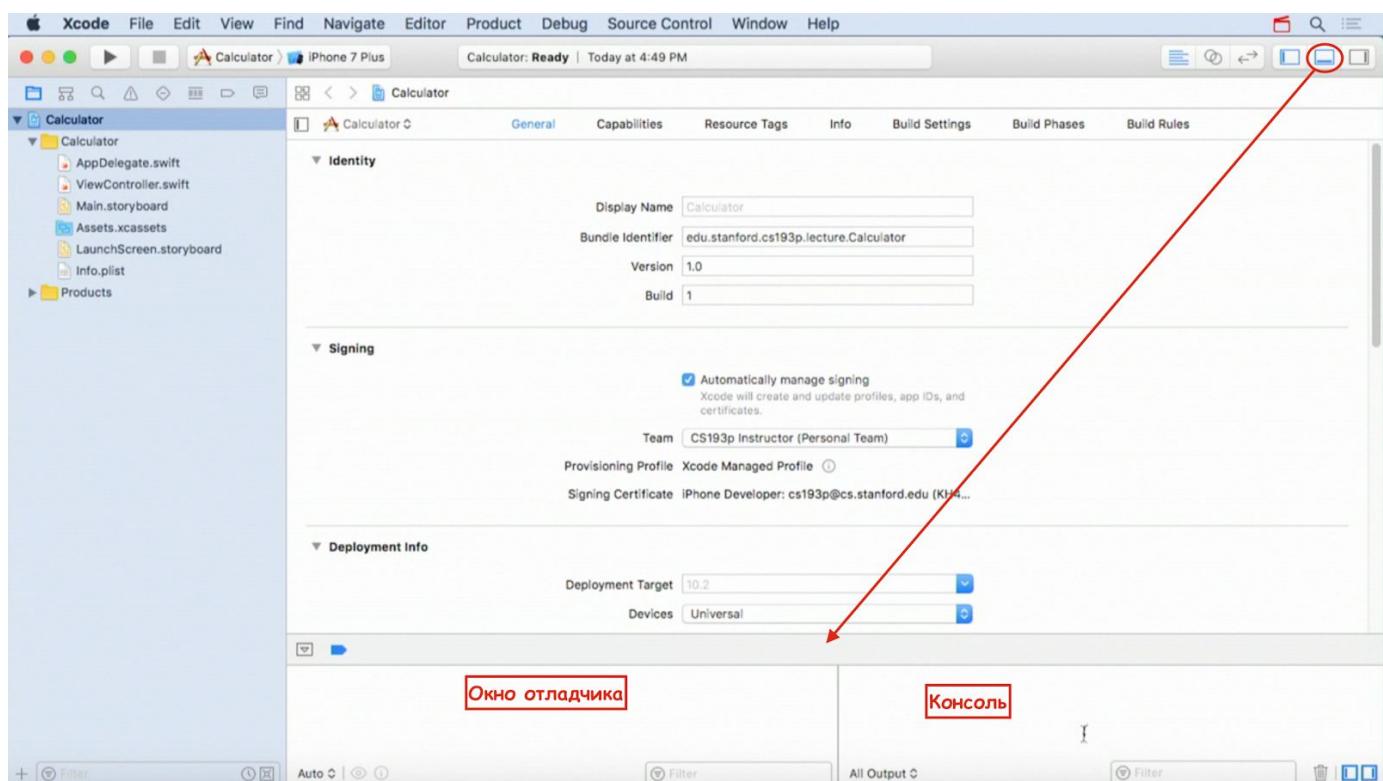


Самая правая кнопка управляет появлением и исчезновением Области Утилит (**Utilities Area**).

Самая левая кнопка управляет появлением и исчезновением "Навигатора".

Есть еще одна кнопка - в середине. Она отвечает за появлением и исчезновением некоторой области внизу, в которой слева находится "окно отладчика", а справа - консоль, куда выводится

печатать.



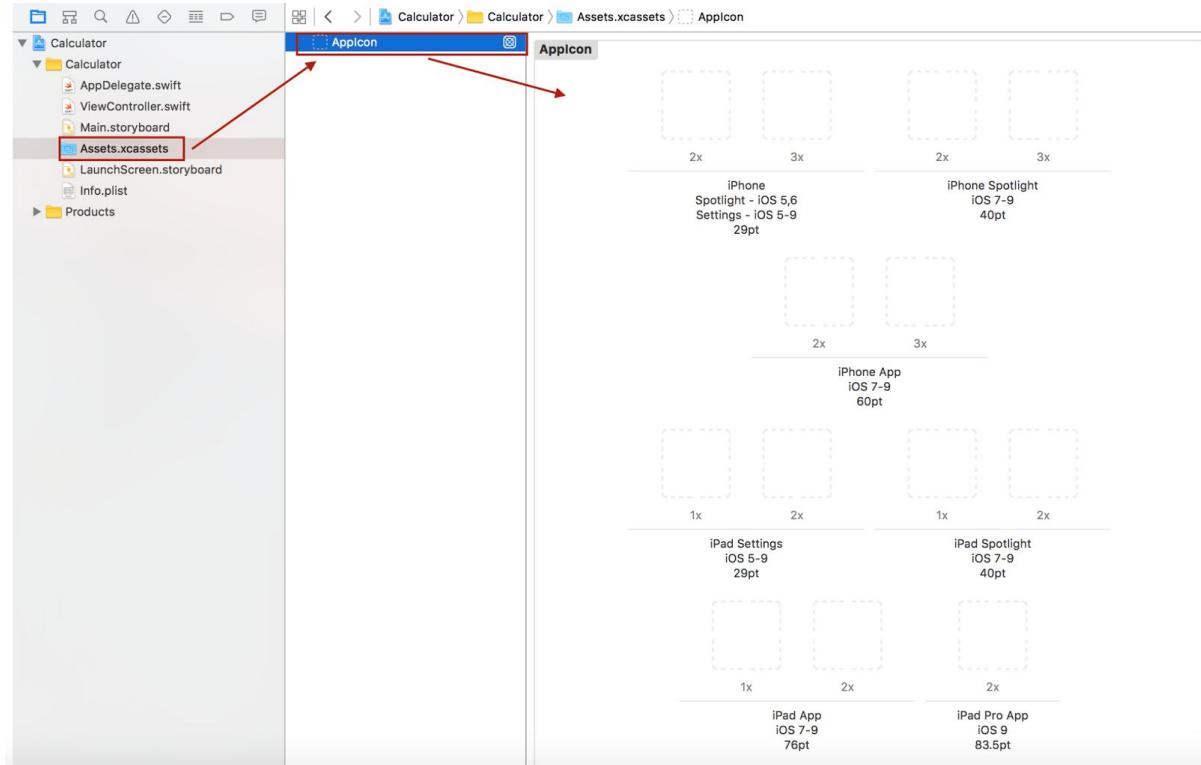
Вы все это увидите через секунду.

Давайте посмотрим на файлы в навигаторе.

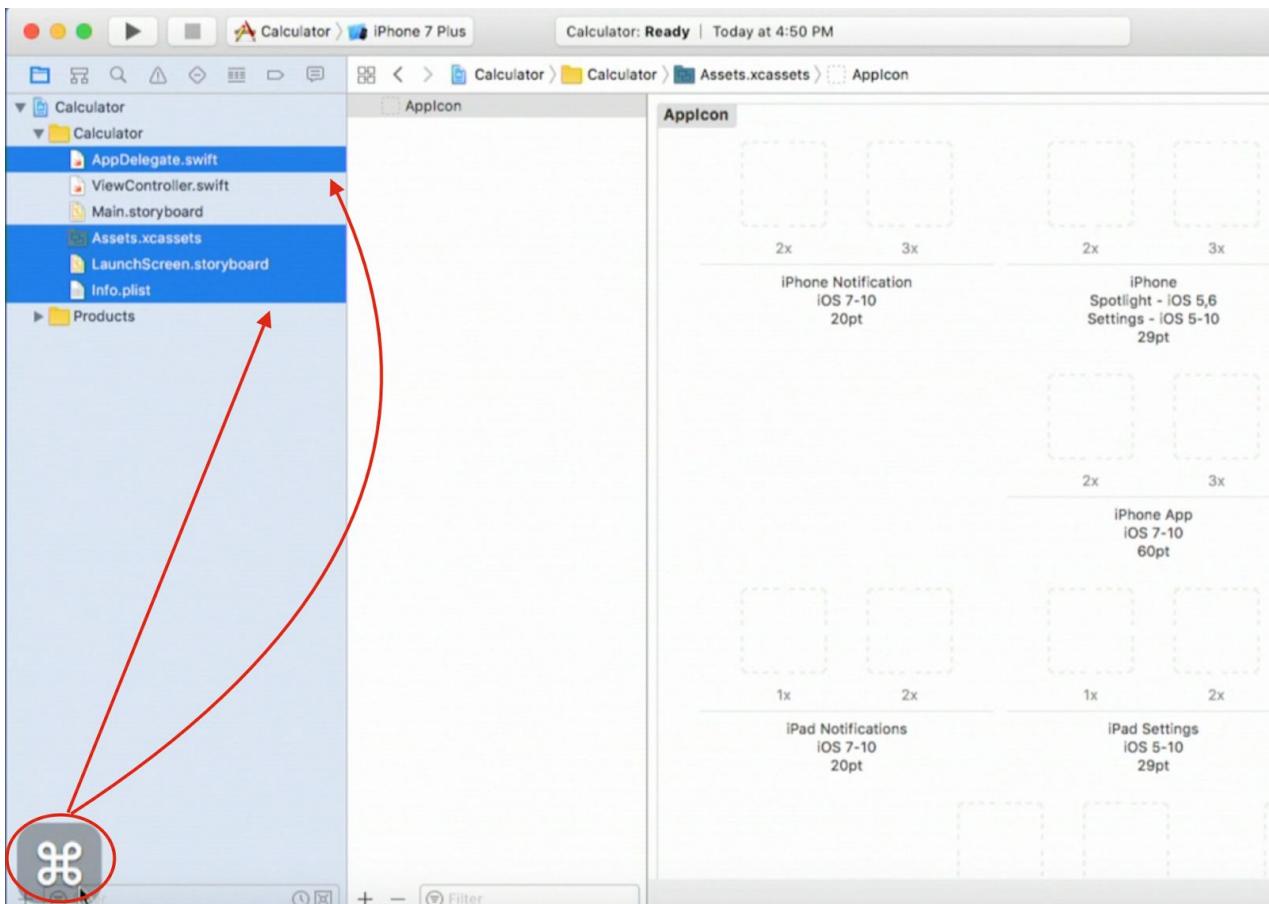
Этих файлов 6, но 4 из них - действительно просто вспомогательные файлы и мы даже не будем их видеть в **Calculator**. Мы взглянем на пару из них по мере того, как наш семестр будет двигаться к концу, но все равно это не главные файлы.

----- 15 -ая минута лекции -----

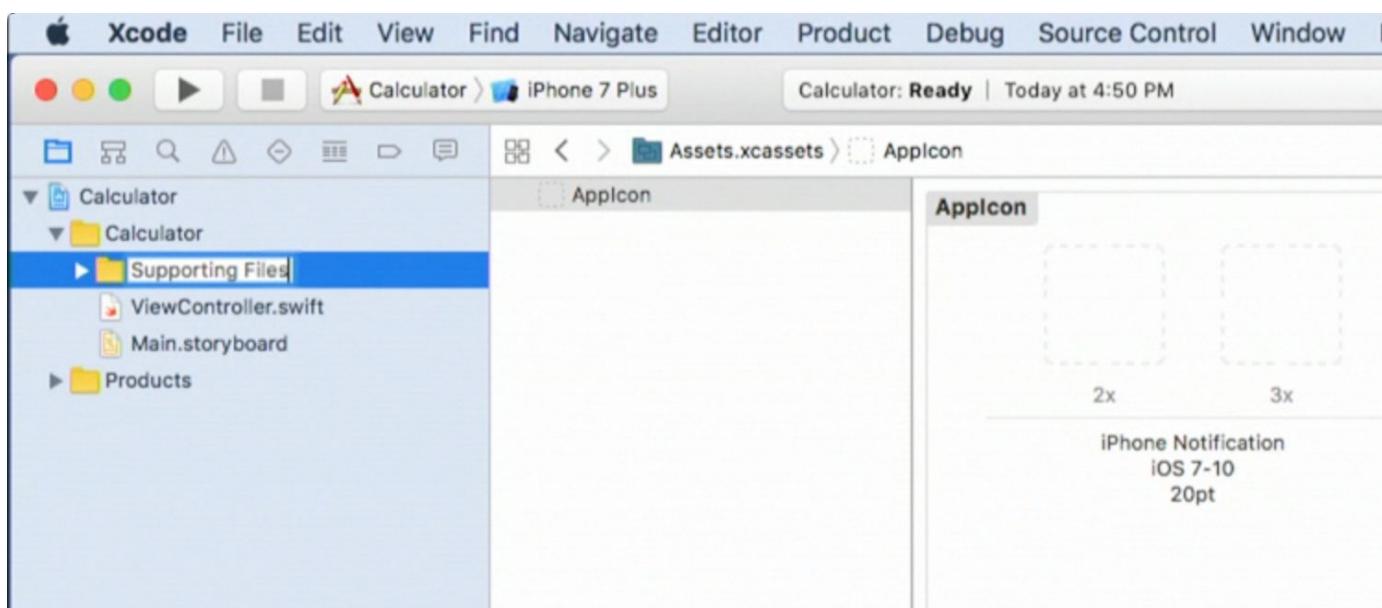
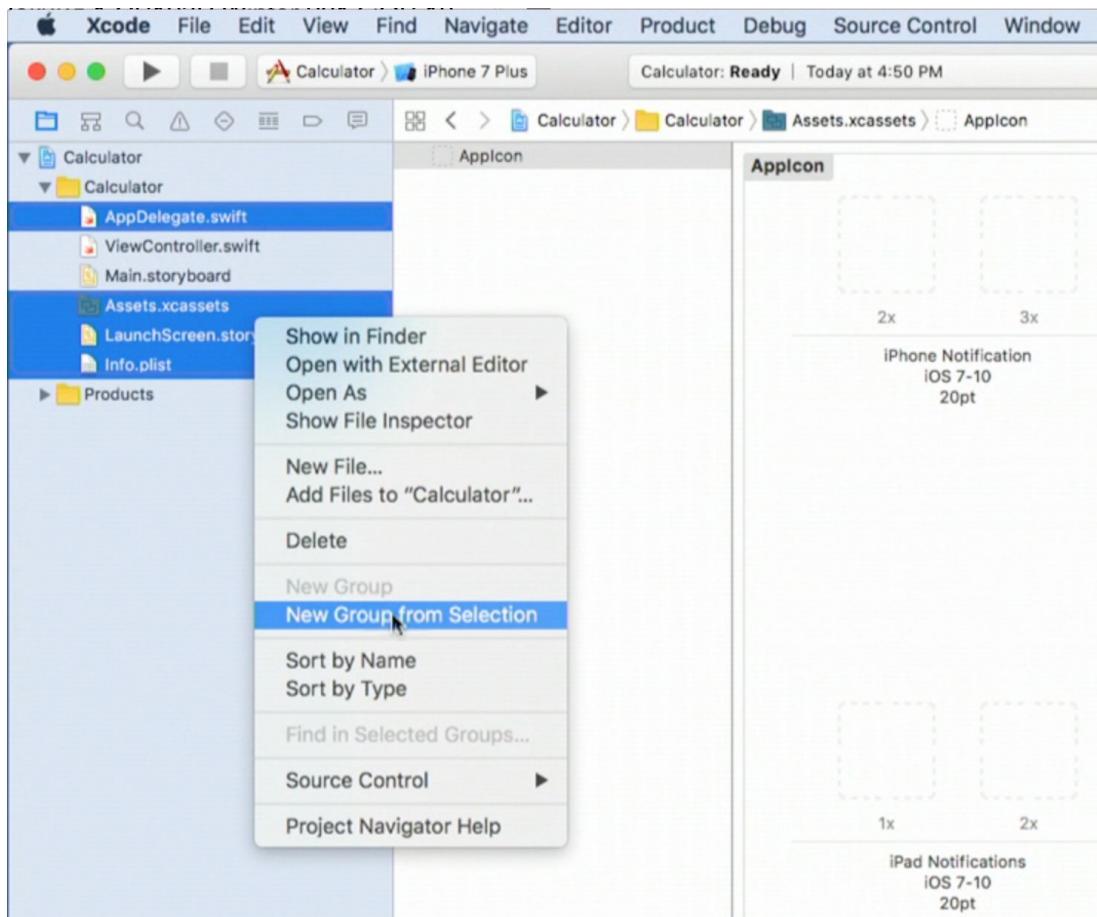
Например, одним из этих файлов является файл **Assets.xcassets**. Он включает все **media** содержимое приложения. Все видео, изображения, звуки, иконки и подобные этому вещи, которые могут быть вставлены в ваше приложение. Вот как выглядят иконки приложения **AppIcon**, которые я не установил и все эти «кармашки» пустые.



Итак, выделяем файлы `Assets.xcassetts`, `LaunchScreen. storyboard`, `Info.plist` и этот `AppDelegate.swift`. Между прочим, я выделяю их все с помощью нажатия клавиши **command**, которую вы видите в нижнем левом углу, потому что вы сможете увидеть любые командные клавиши, которые я нажимаю при выполнении определенных действий. Итак с помощью клавиши **command**, я выделяю 4 файла:



После выбора файлов, я кликаю правую клавишу мыши (**CTRL**-клик) и выбираю во всплывающем меню **New Group From Selection** для размещения выделенных файлов в отдельной папке:



Эту папку я назову "**Supporting Files**", потому что эти файлы действительно поддерживают мое приложение. Вы видите, что все они скрылись в этой папке, теперь мы не будем их видеть. Мы сфокусируемся на двух файлах, которые являются наиболее важными для сегодняшнего демонстрационного примера.

Файл *ViewController.swift* — это ваш первый взгляд на **Swift**.

```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

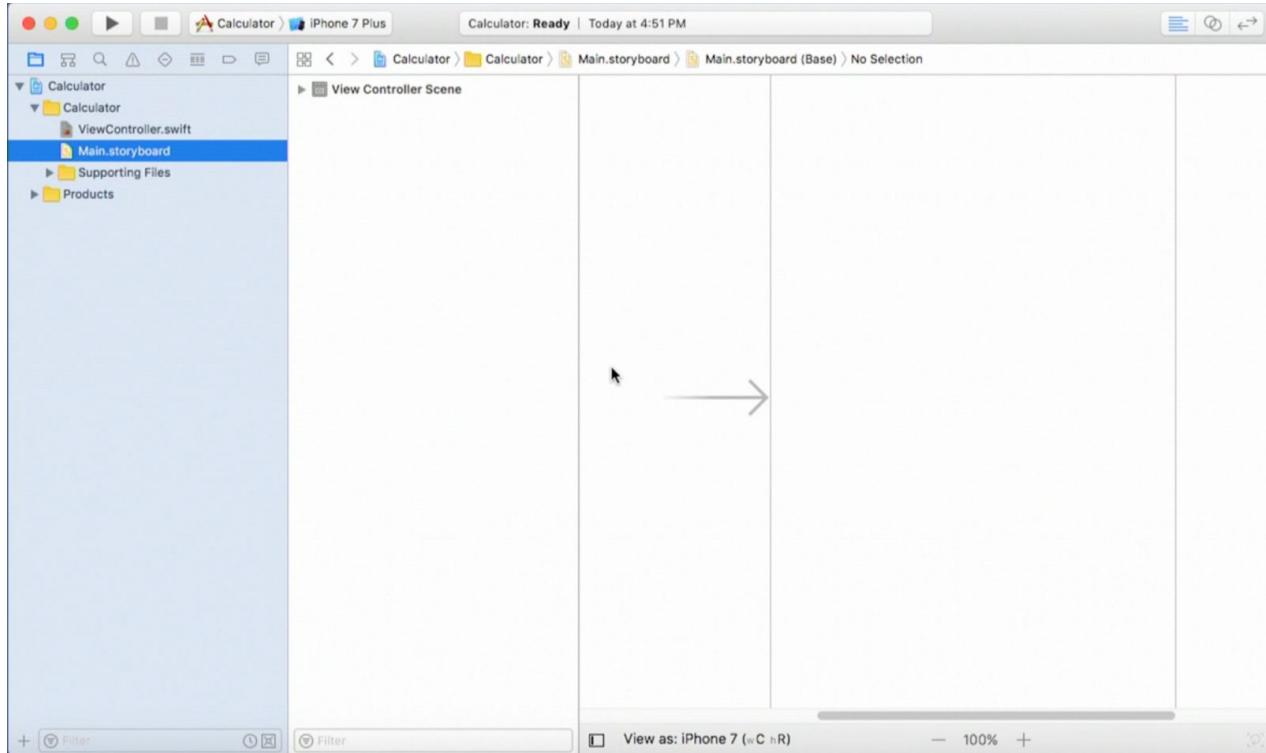
Этот класс получил из шаблона код для очень важных методов, и я удаляю этот код, хотя факт удаления не делает их менее важными.

```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

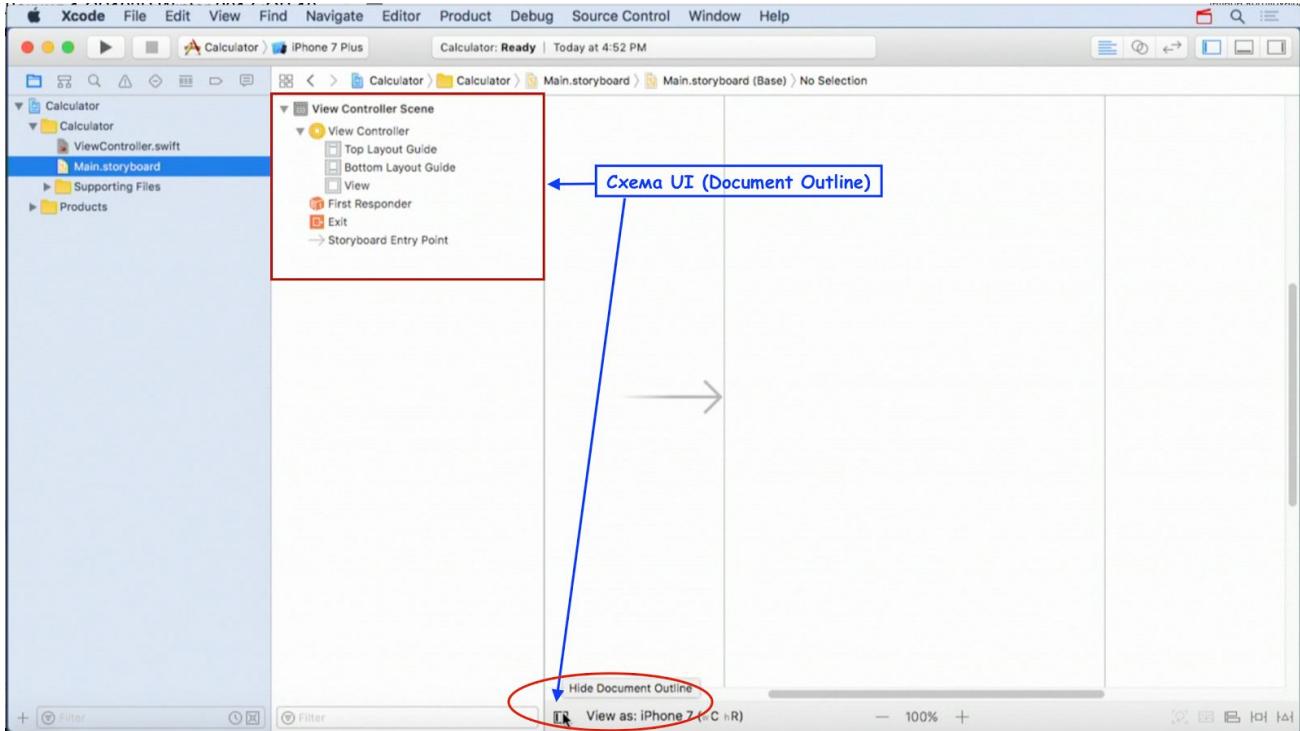
class ViewController: UIViewController {
```

Это означает, что мы не будем изучать их в ближайшие две недели, мы не будем их использовать в **Calculator**. Через мгновение я расскажу детально о **Swift** коде в этом классе, но вначале я сосредоточусь на файле *Main.storyboard*.

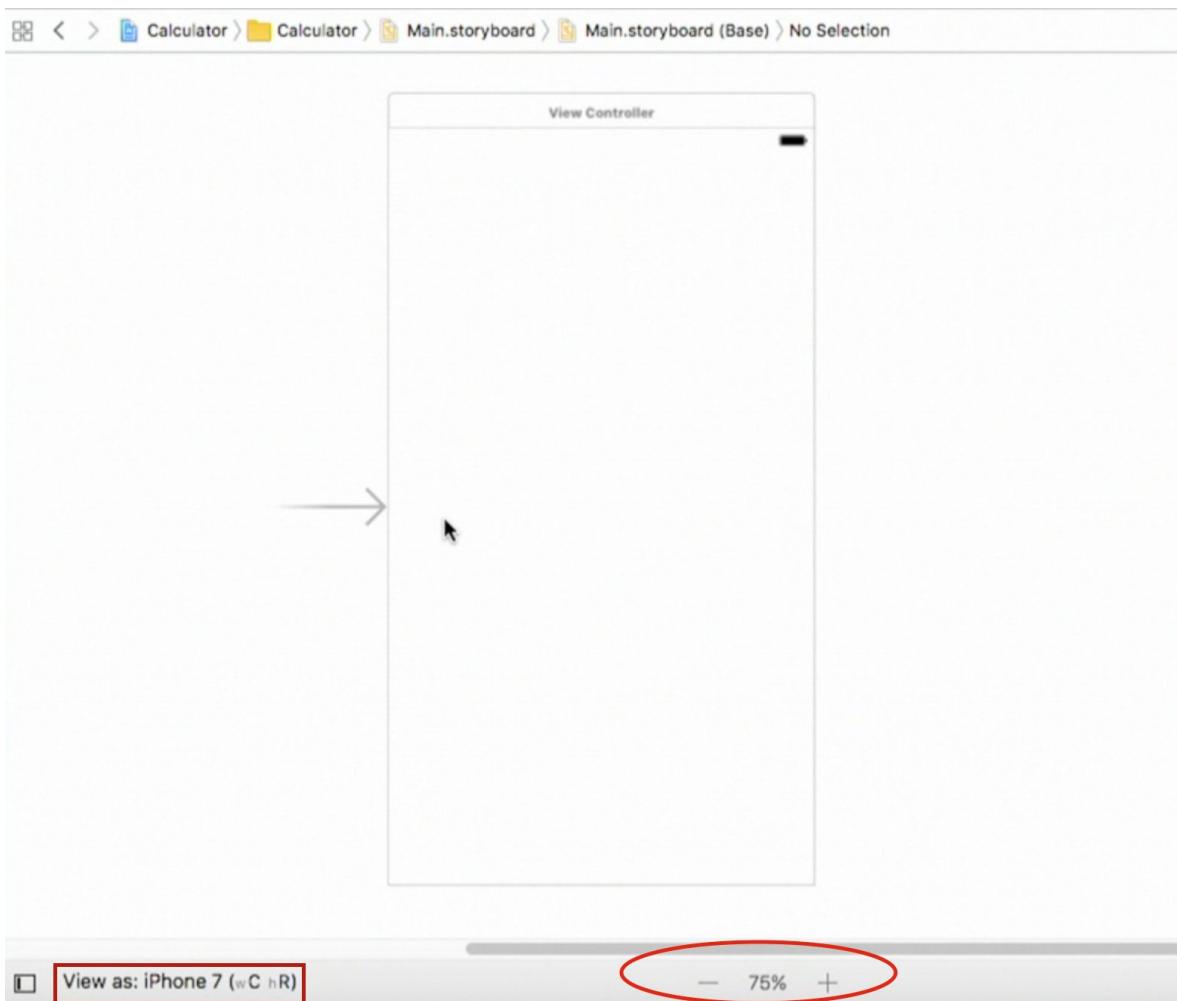


Это ваш пользовательский интерфейс (**UI**), **UI** Калькулятора. Вы, наверно, заметили одну особенность, когда я кликнул на файл *Main.storyboard*. В нем нет кода. Когда вы строите пользовательский интерфейс в **iOS** приложении в **Xcode**, то вам не нужно писать код. Вы будете строить его с помощью “мышки”. Мы будем перетаскивать с помощью “мышки” различные объекты **UI** туда, куда мы хотим, мы будем инспектировать эти объекты и устанавливать их свойства также с помощью “мышки”. Вот как мы будем строить пользовательский интерфейс. Код, который вы только что видели, управляет поведением UI. То есть тем что происходит, если вы нажмете на кнопку. Код управляет подобными вещами. Поведение **UI** управляется кодом, а реальное взаимное расположение **UI** объектов - всех этих кнопок, меток, текстовых полей - выполняется в графическом виде здесь, на storyboard.

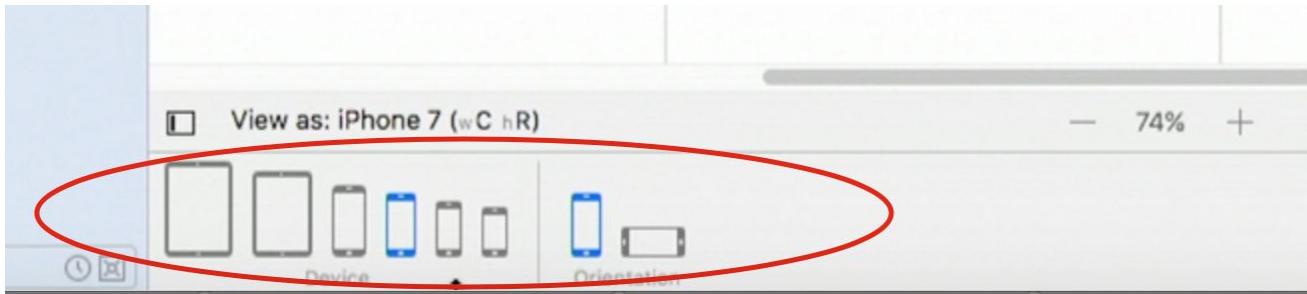
Слева от **UI** вы видите небольшую область, называемую Схема **UI** (**Document outline**). Внизу находится маленькая кнопочка, которая показывает и скрывает Схема **UI** (**Document outline**).



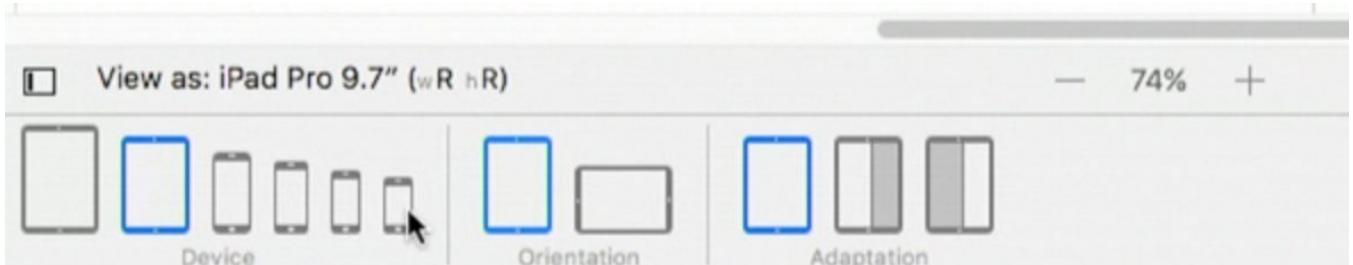
В схеме UI (**Document outline**) представлены все элементы пользовательского интерфейса, и это часто оказывается очень полезным для разработчика, нам она понадобится позже в этом семестре. А сейчас я ее скрою с помощью маленькой кнопки, расположенной внизу, для того, чтобы дать больше пространства для UI. Если я к тому же изменю немного масштаб нашего UI, то мы увидим что-то похожее на **iPhone**.



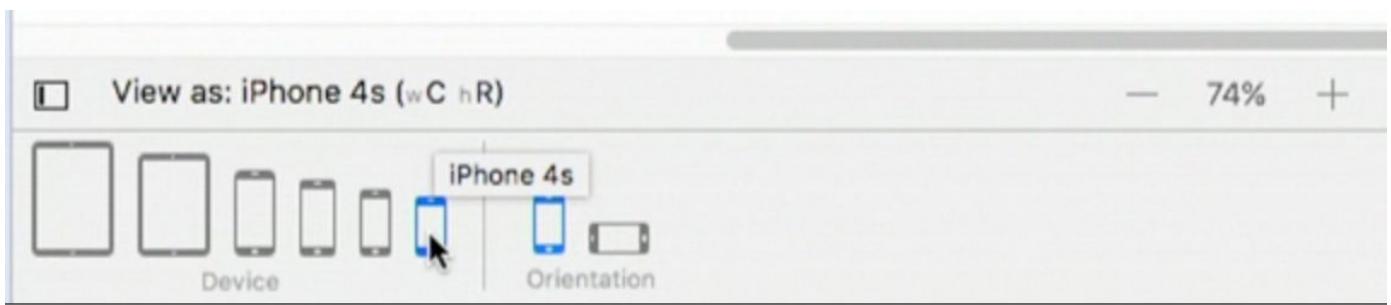
Это напоминает форму iPhone, и я могу, между прочим, удерживая клавишу Option и используя “колесико” “мыши” увеличивать и уменьшать масштаб UI (**zoom in** и **zoom out**). Это выглядит как iPhone 7. Вы видите внизу написано: View as iPhone 7 (выглядит как iPhone 7) и если вы кликните на этой надписи, то появятся все другие iOS устройства:



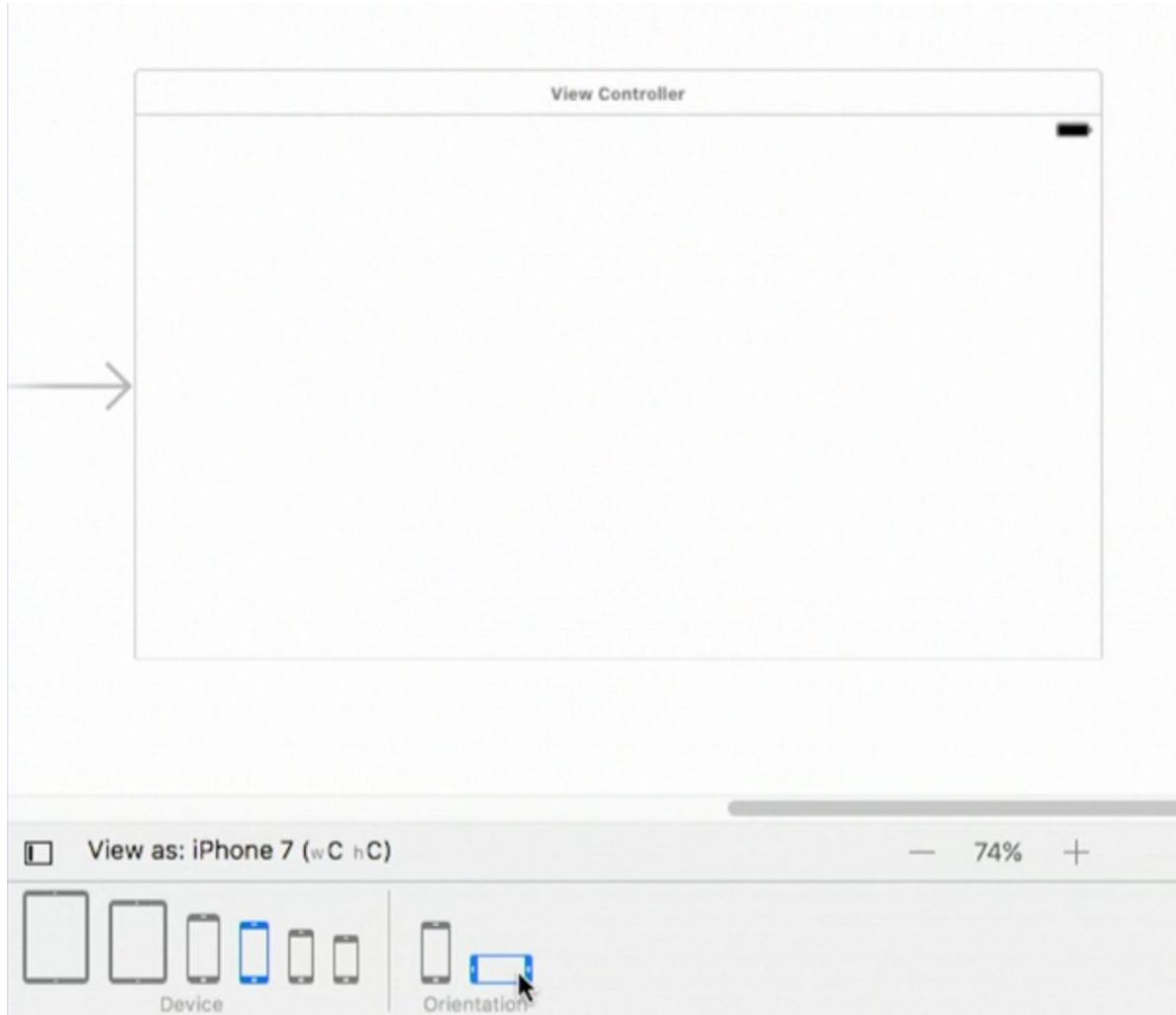
Вы можете выбрать, например, iPads:



Или старые iPhone 4s, которые реально выглядят очень маленькими:



И не только различные устройства, но вы можете также переключать ориентацию iOS устройства:



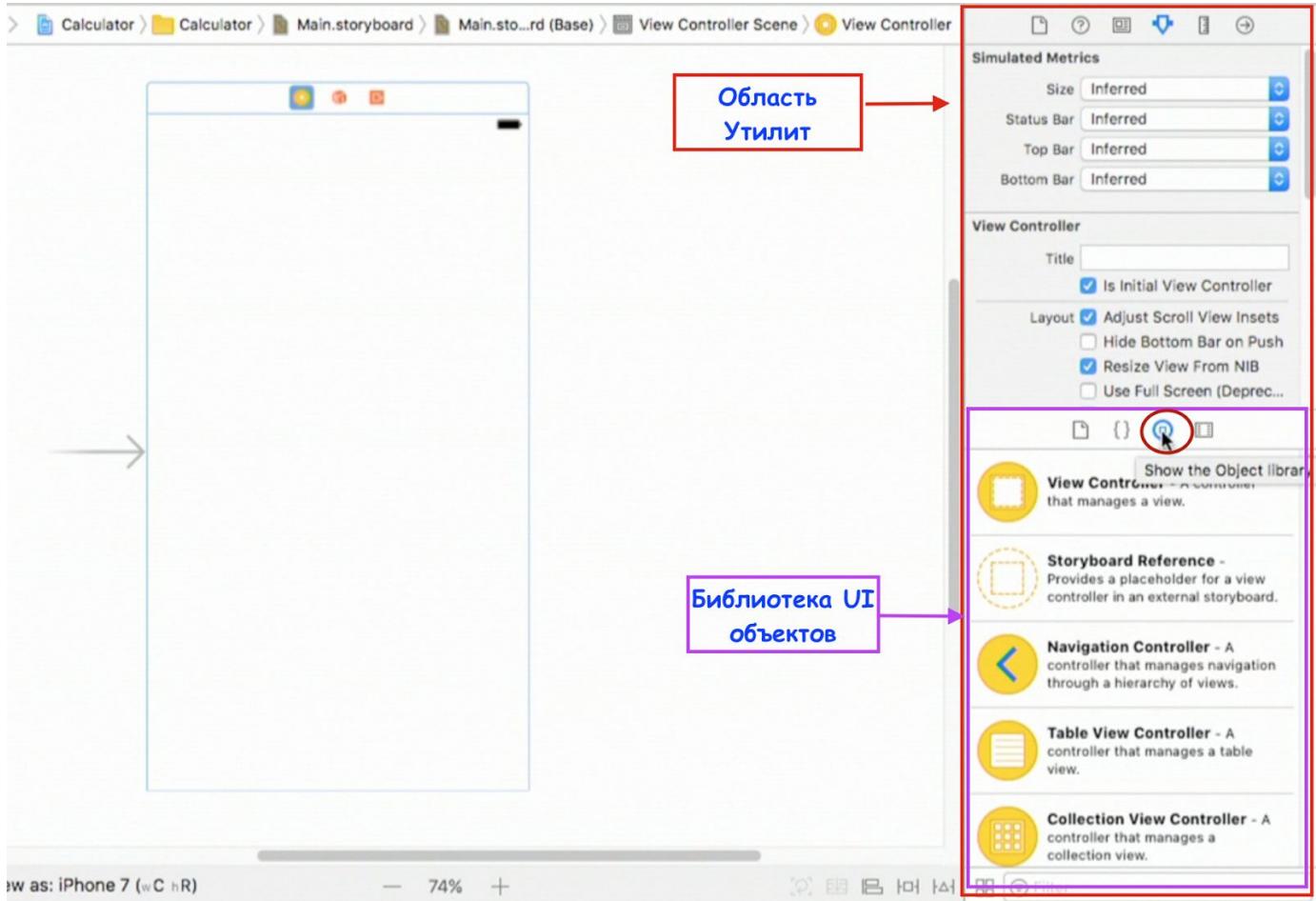
Когда вы разрабатываете iOS приложение, вы хотите, чтобы пользовательский интерфейс (**UI**) выглядел прекрасно на всех iOS устройствах. Вы не хотите писать множество конструкций **if ...** повсюду, чтобы заставить все их работать.

Перед вами находится целая система построения **UI**, которая называется **Interface Builder**. Это часть **Xcode**, называемая **Interface Builder**, с широкими функциональными возможностями ориентирована на однократное построение UI, который будет прекрасно работать на всех устройствах. Однако этим я не буду заниматься до конца следующей Лекции в Среду.

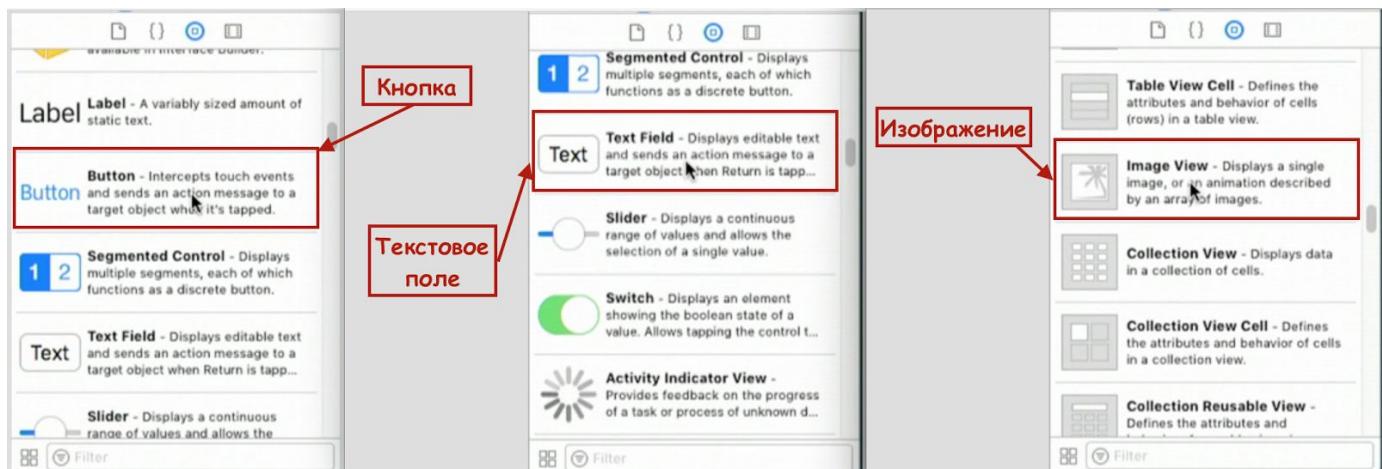
Сейчас мы будем создавать наш UI, который будет немного беспорядочным. Мы будем размещать кнопки там, где хотим, и наш UI реально не будет работать в ландшафтном режиме также хорошо, как в портретном. Он также не будет адаптироваться к маленьким или большим устройствам. Мы сейчас не будем придавать этому значение. Но я вам рассказал о том, как выполнить предварительный просмотр вашего **UI** (preview) при построении универсального (universal) **UI**, чтобы убедиться, что он работает на всех iOS устройствах.

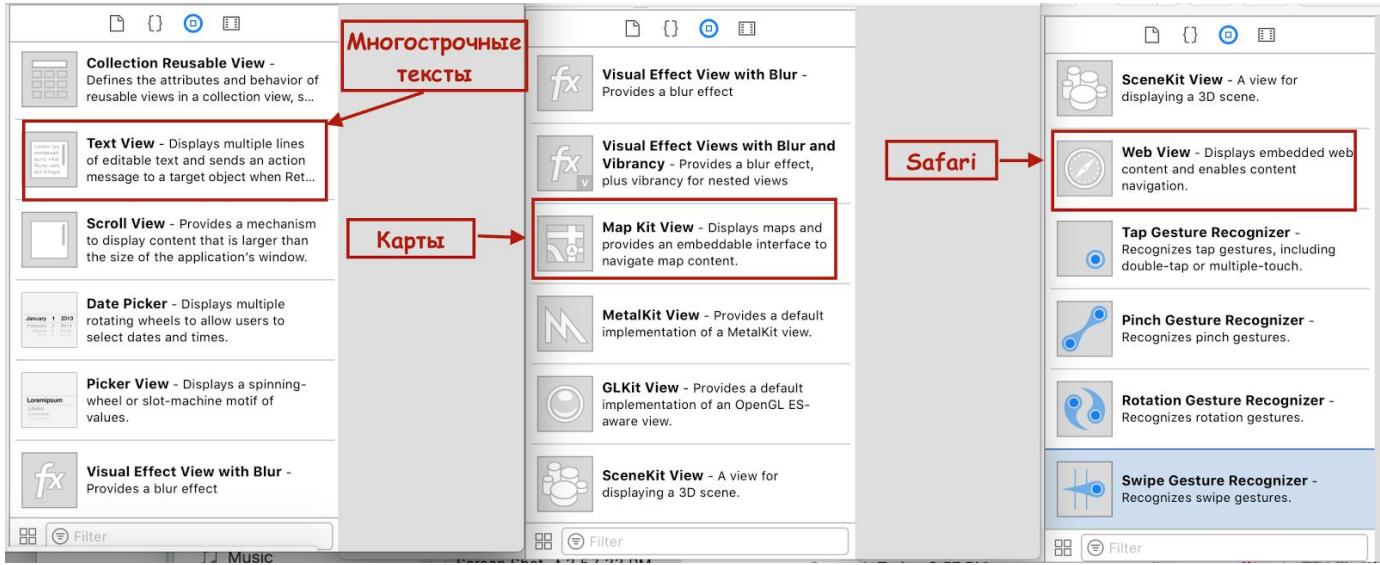
Итак, перед нами iPhone 7. Мы начинаем строить наш **UI**.

Что нам необходимо? Нам необходимы кнопки (**buttons**) и в верхней части экрана - дисплей. Давайте начнем с кнопок. Где мы возьмем эти кнопки? Как я и обещал, прямо сейчас поговорим об Области Утилит (**Utilities Area**). У этой области есть верхняя и нижняя части. В нижней части в разделе, называемом "Object Library" располагается библиотека iOS объектов, из которых создается **UI** приложения:



Здесь есть кнопки (**Button**), текстовые поля (**Text Field**), есть и более сложные элементы - изображения (**Image View**) и тексты (**Text View**), которые представляют собой многострочный редактируемый текст. Есть даже такие объекты как карта (**Map Kit View**), о которой я рассказывал, и даже целое Safari (**Web View**), которое можно разместить в небольшой прямоугольной области:



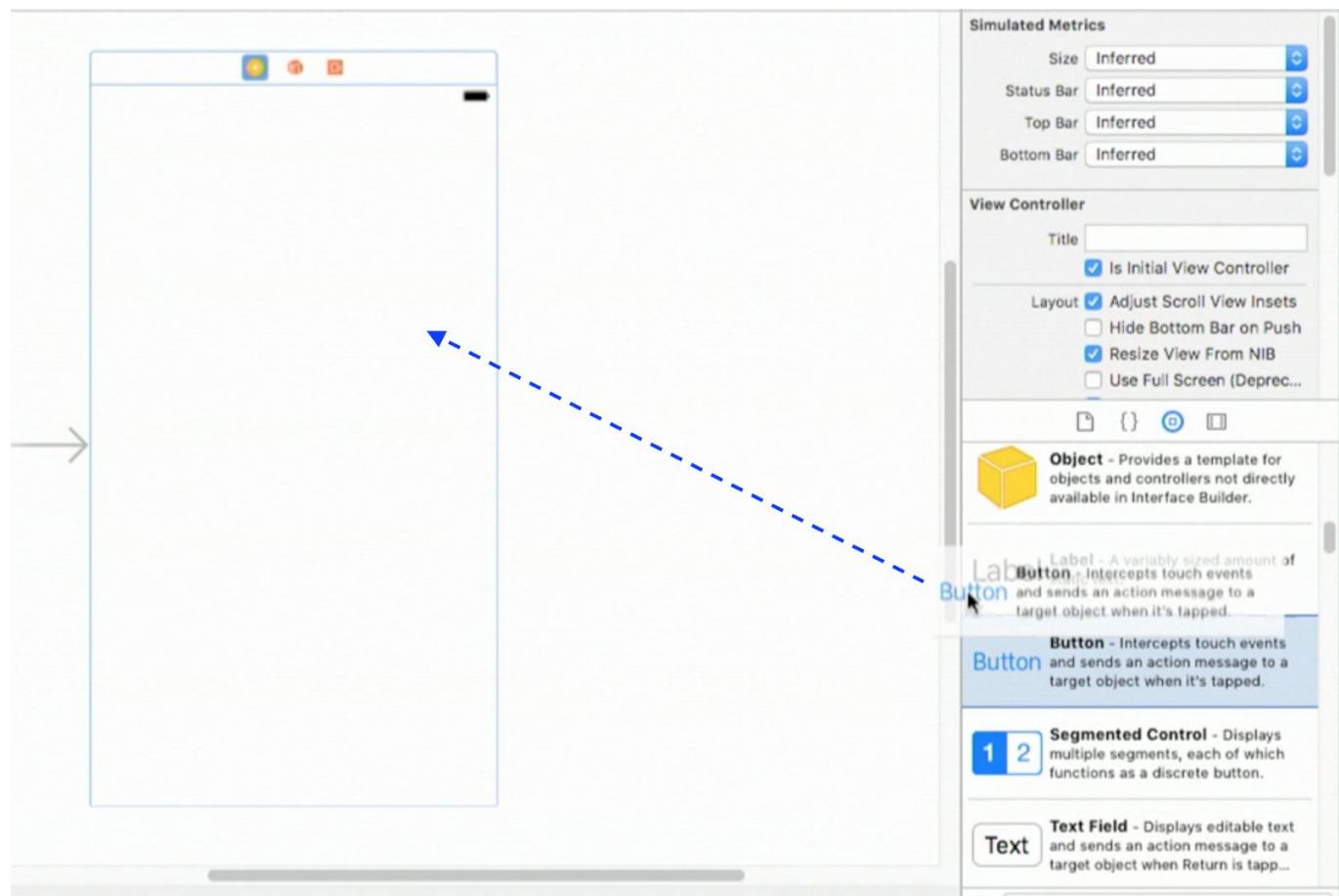


----- 20 -ая минута лекции -----

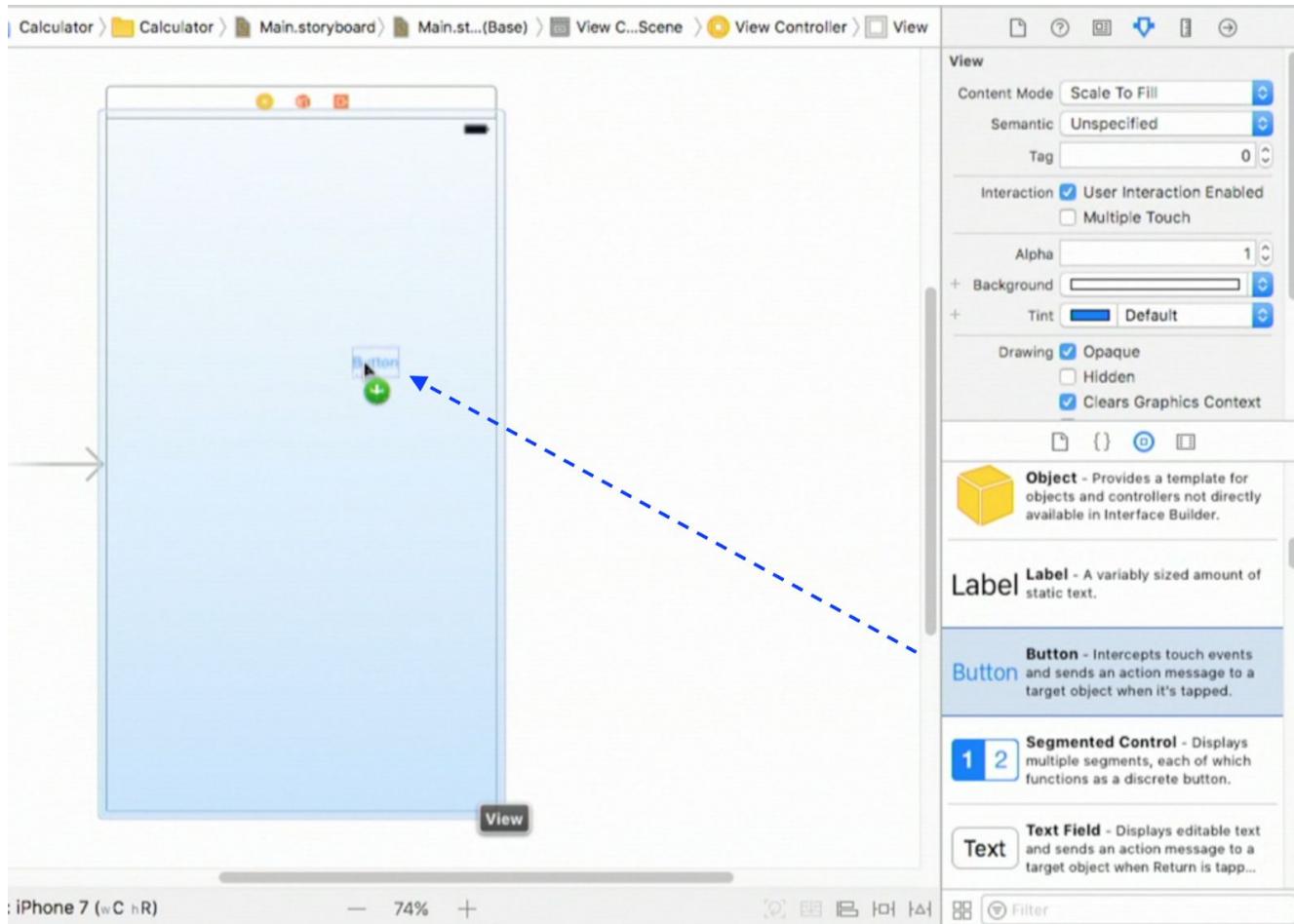
В библиотеке множество очень мощных объектов и большую часть мы рассмотрим на этом курсе, но там их реально очень много, так что останутся и не рассмотренные.

Начнем с простейшего - с кнопки **Button**. Я хочу перенести кнопку на мой UI.

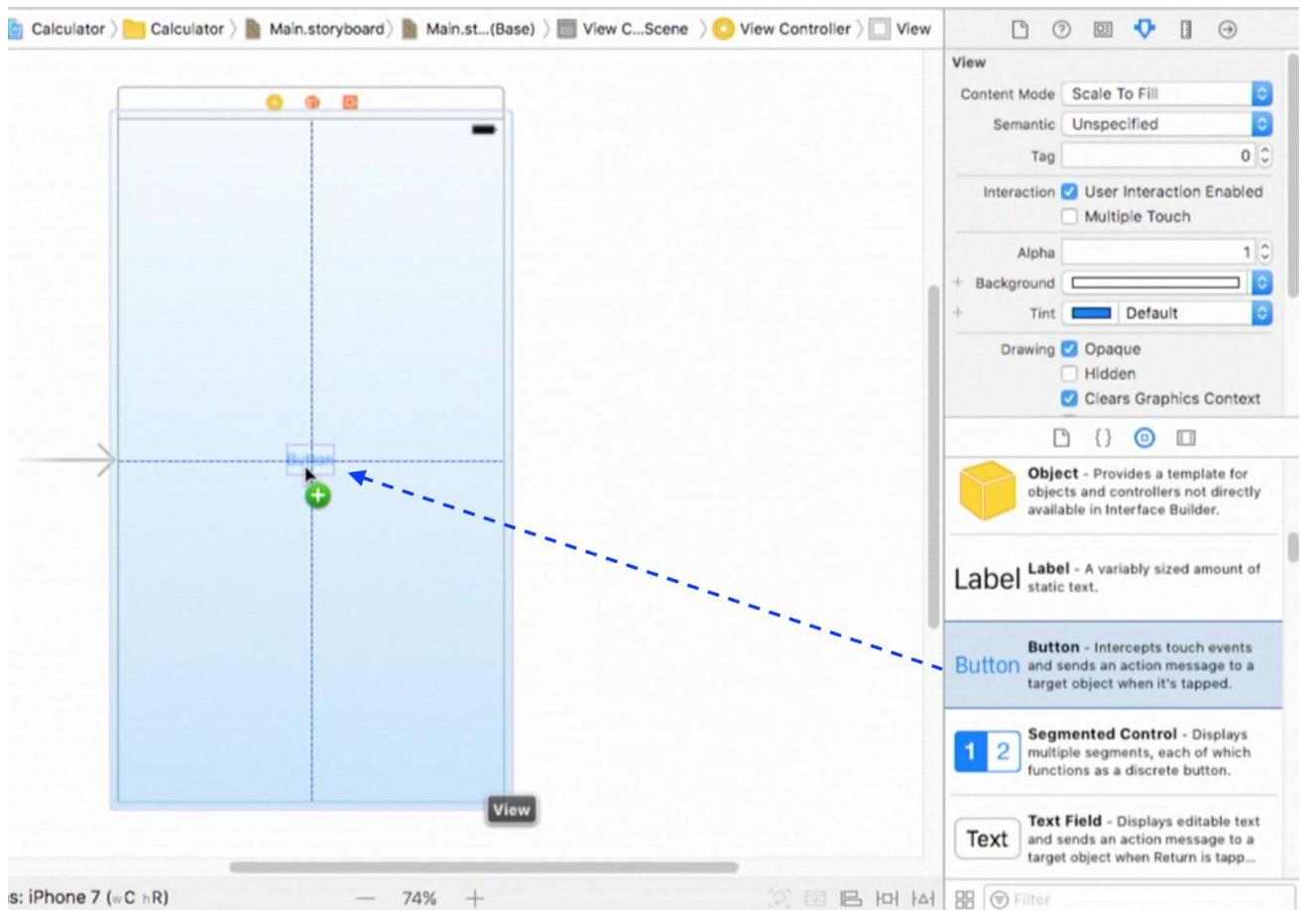
Я подцепляю "мышкой" кнопку



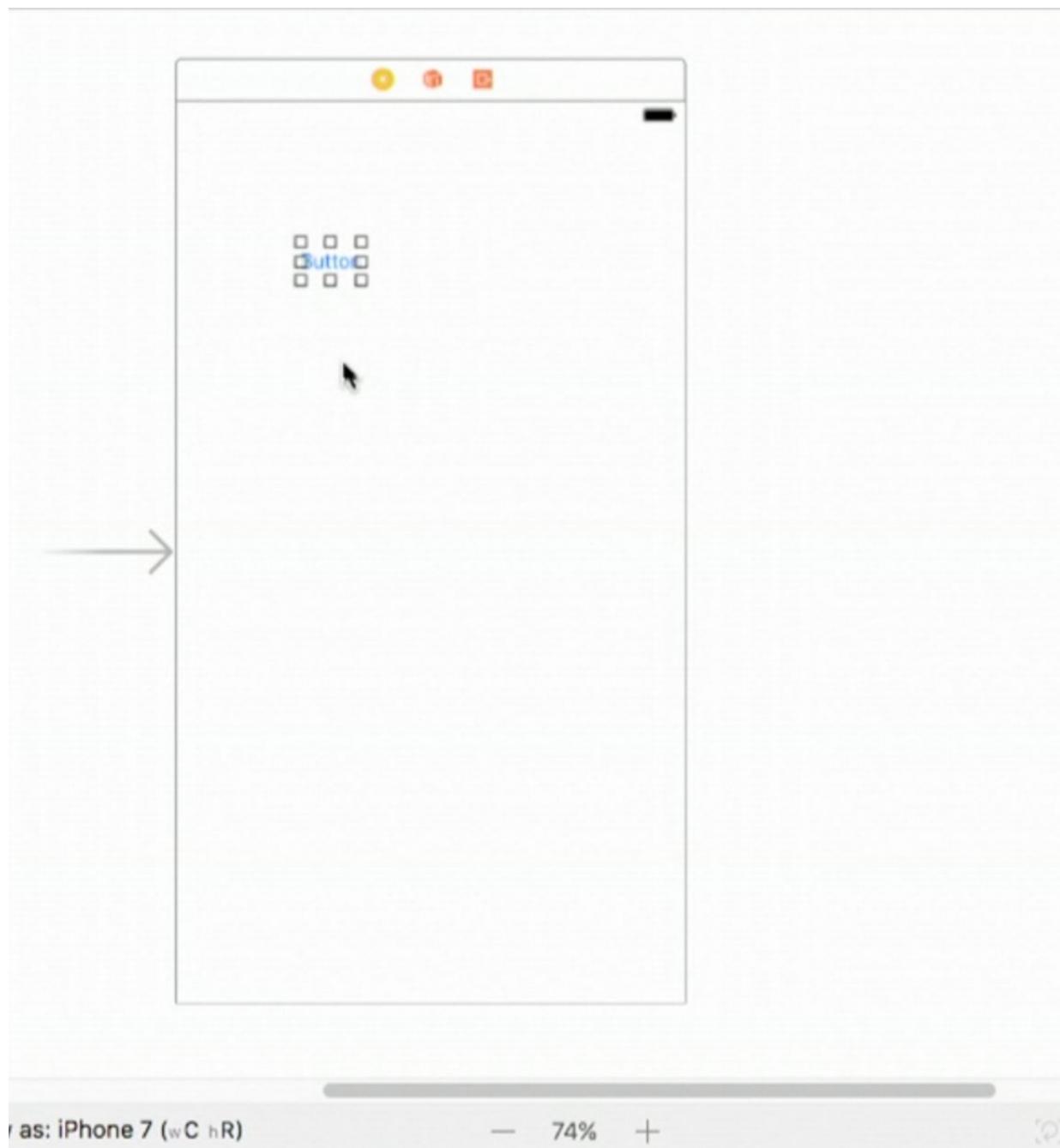
И перетягиваю ее на UI:



В процессе перемещения кнопки по экрану появляются пунктирные голубые линии, которые помогают мне найти правильное место размещения:

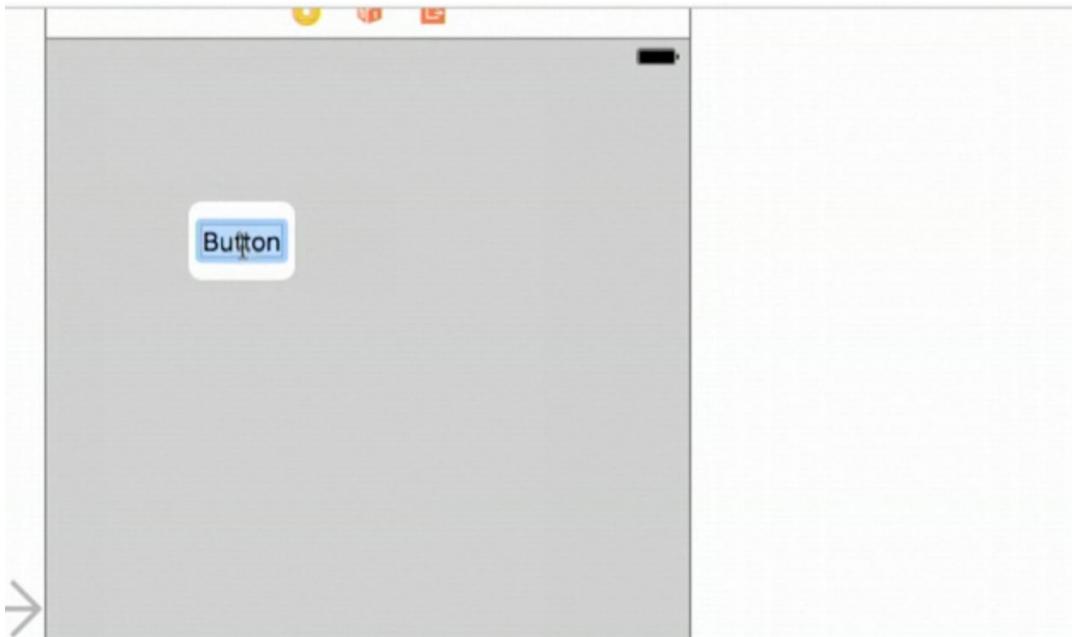


Видите? Прямо сейчас мы не будем обращать на это внимание, потому что я говорил вам, что сейчас мы не будем строить **UI**, который работает на всех устройствах. Но как только вы начнете думать о создании UI, работающего на всех устройствах, вы начнете очень интенсивно использовать направляющие голубые пунктирные линии. Например, в представленном выше случае направляющие голубые линии говорят о том, где находится центр (**center**) любого устройства. То есть мы можем разместить кнопку в центре любого устройства, независимо от размера экрана. Видите? Голубые линии помогают нам взаимодействовать с Interface Builder, вы как бы говорите ему, что хотите, чтобы кнопка располагалась точно посередине экрана. Но опять, сейчас мы на это не обращаем внимания, и я могу “бросить” кнопку прямо в середине или где-то еще.

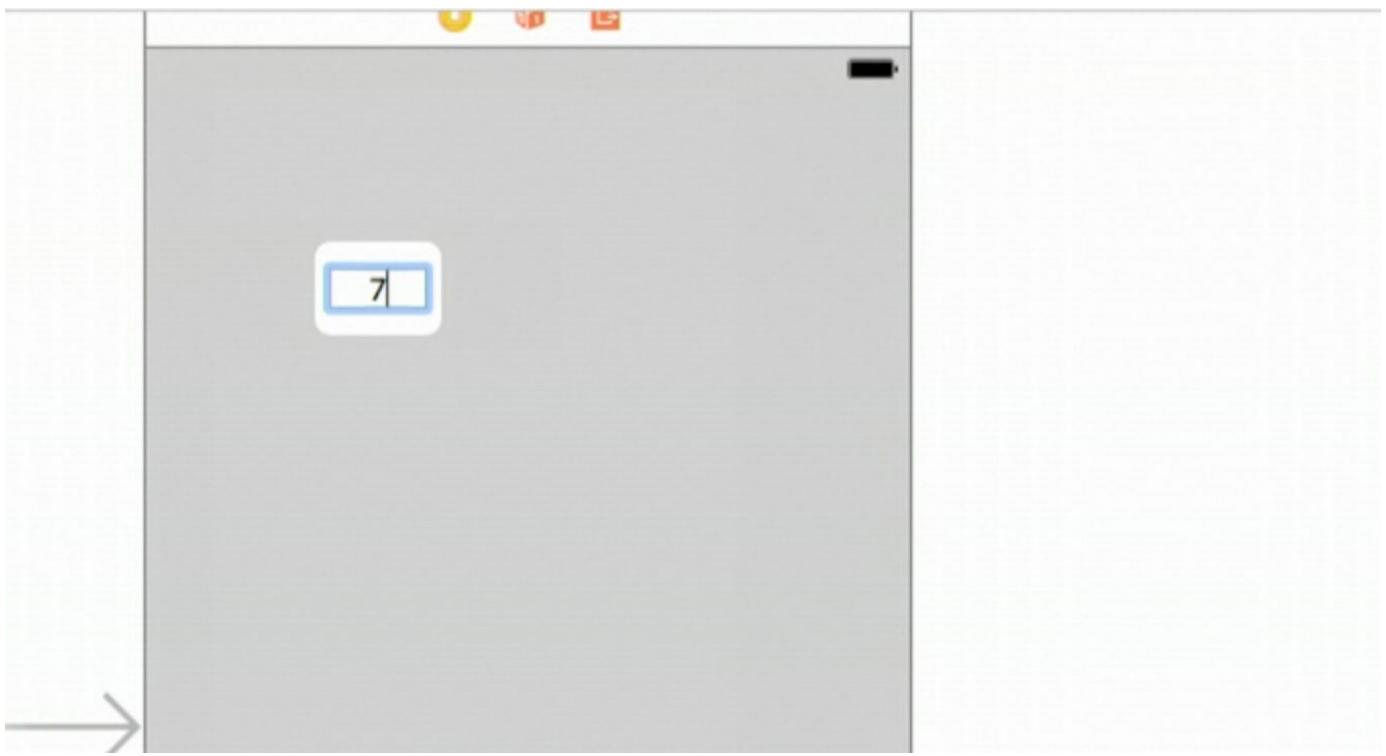


И все таки хочу повторить еще раз. Голубые линии - это очень важные “дорожные знаки”.

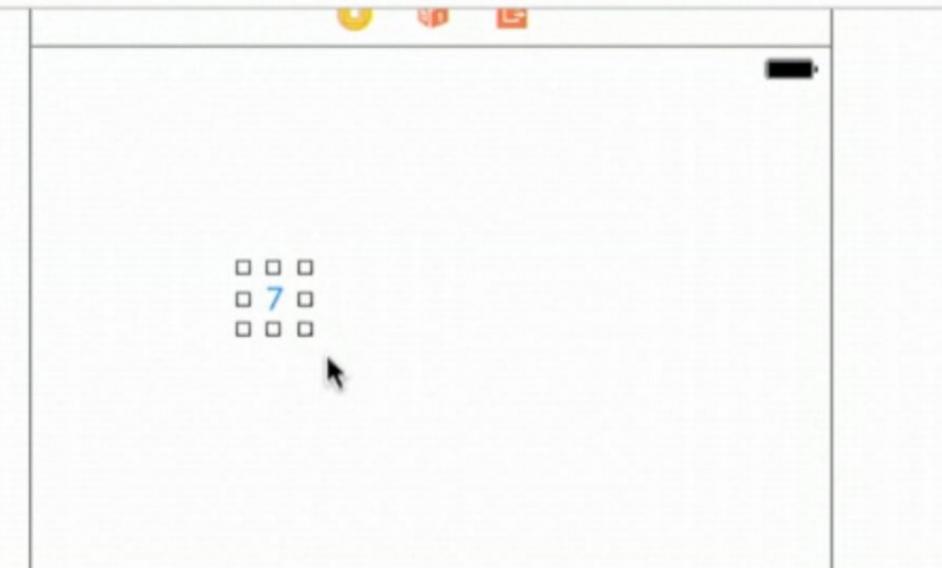
Итак, мы получили кнопку, но она слишком маленькая, да и на ней написано “Button”, а мы хотим, чтобы это была какая-нибудь цифра. Для того, чтобы изменить заголовок кнопки, мы дважды кликаем на ней



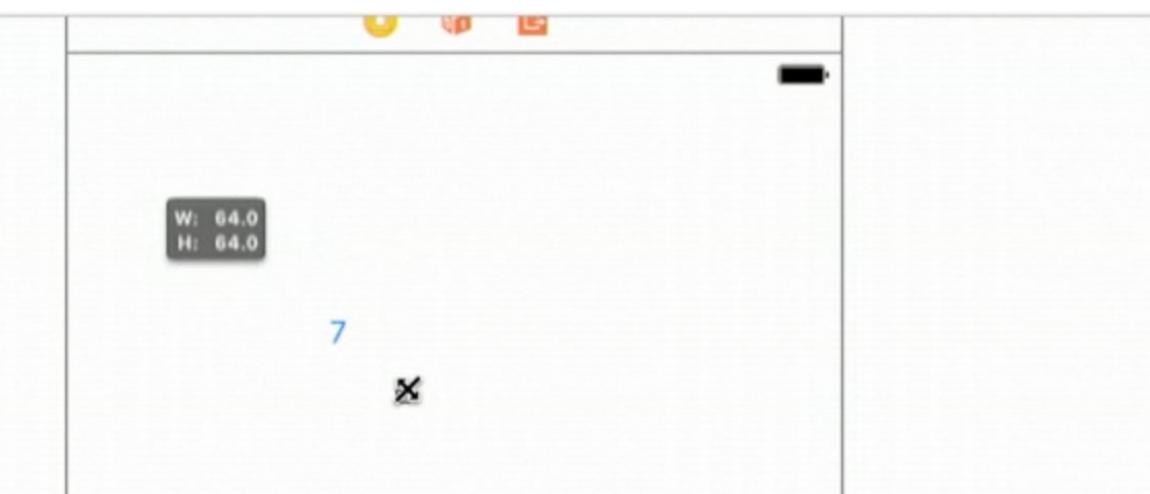
И набираем цифру “7”. Теперь это будет цифра 7 в моем Калькуляторе:



Я могу изменить размер этой кнопки. Если я ее выберу, то появятся маленькие “ручки”, за которые можно тянуть в разные стороны и добиться того размера, который вам нужен:



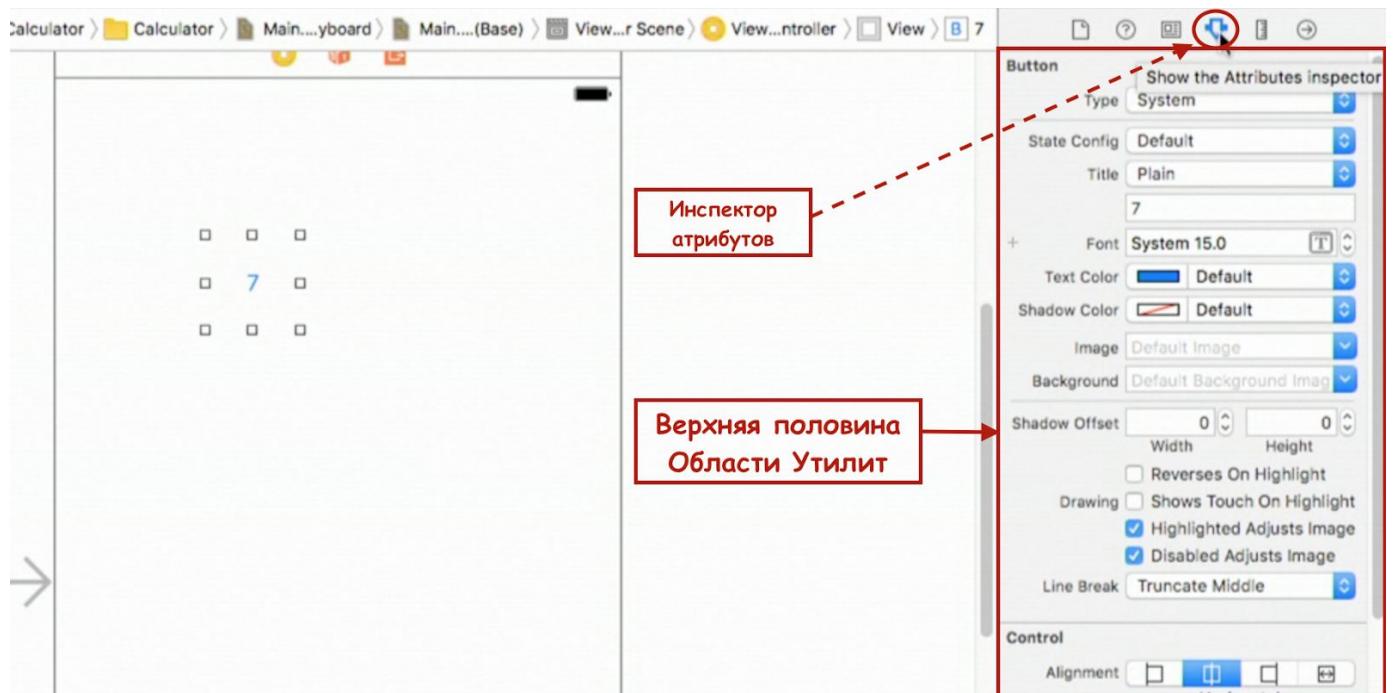
Мне нужен размер 64 x 64. Когда я буду тянуть маленькие “ручки”, на экране будет высвечиваться размер кнопки.



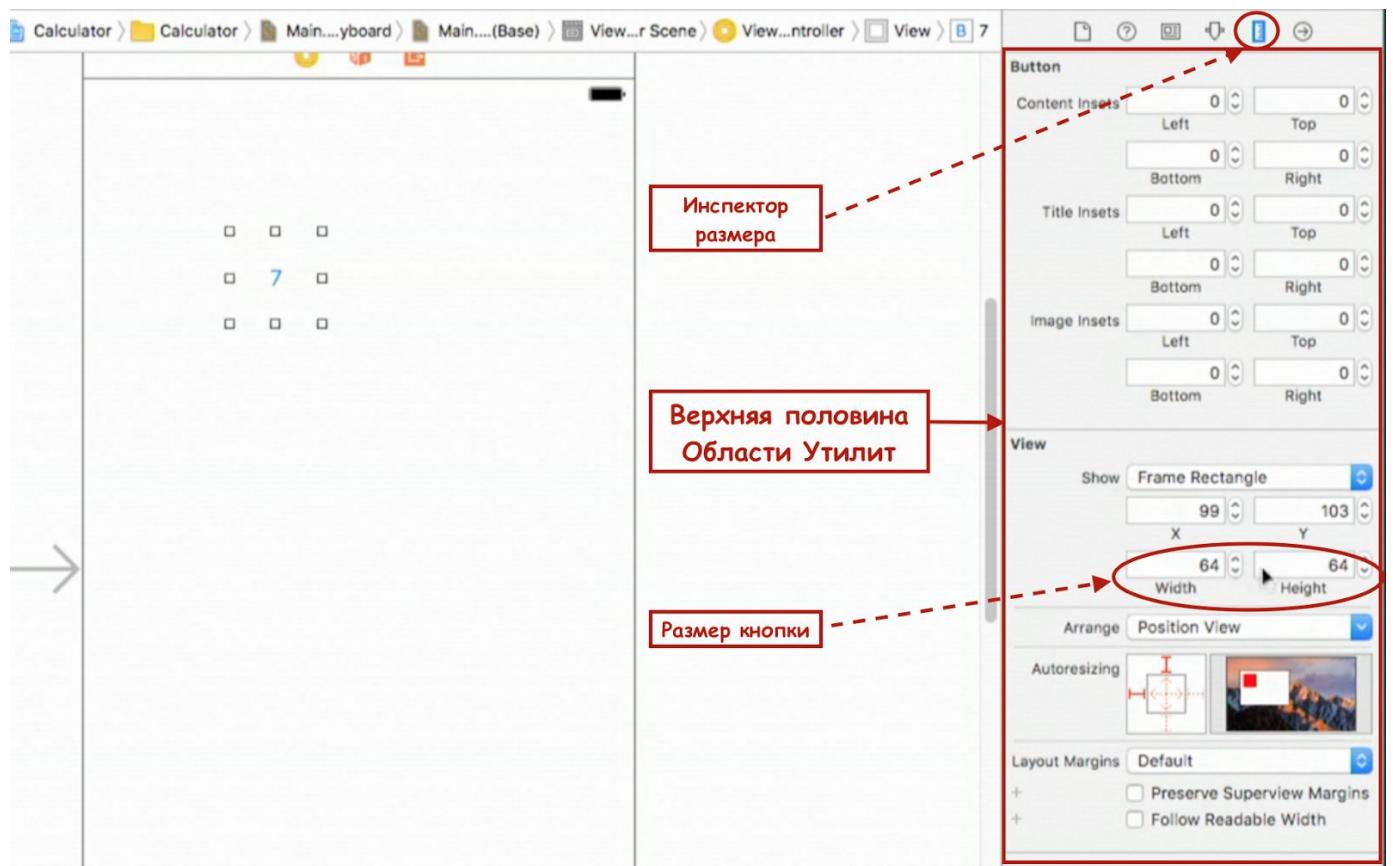
По большому счету конкретный размер не имеет значения, потому что на маленьких устройствах типа iPhone 4 кнопка будет сжиматься, а на больших - расширяться. Но мы выбрали какой-то разумный размер кнопки для iPhone 7.

Но шрифт написания заголовка кнопки “7” маловат. Нам нужно больше.

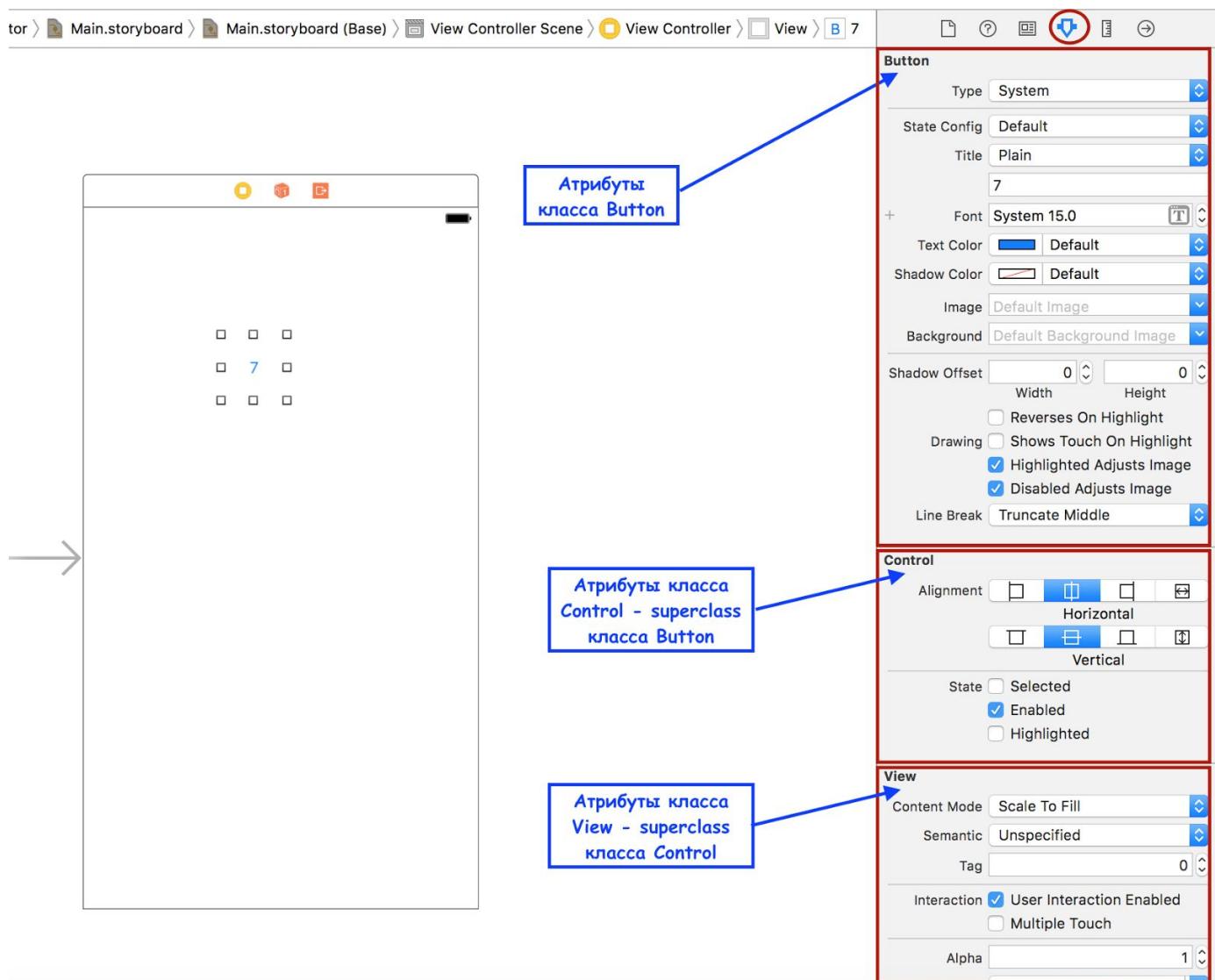
Когда мы хотим делать подобные настройки, мы должны использовать верхнюю половину Области Утилит. Я могу вообще избавиться от нижней половины Области Утилит, задвинув ее в самую нижнюю часть Области Утилит. В самом верху Области Утилит находится панель с маленькими кнопками, соответствующими различным инспекторам. Мы собираемся использовать Инспектор Атрибутов (**Attribute Inspector**):



Еще у нас есть Инспектор Размера (Size Inspector):

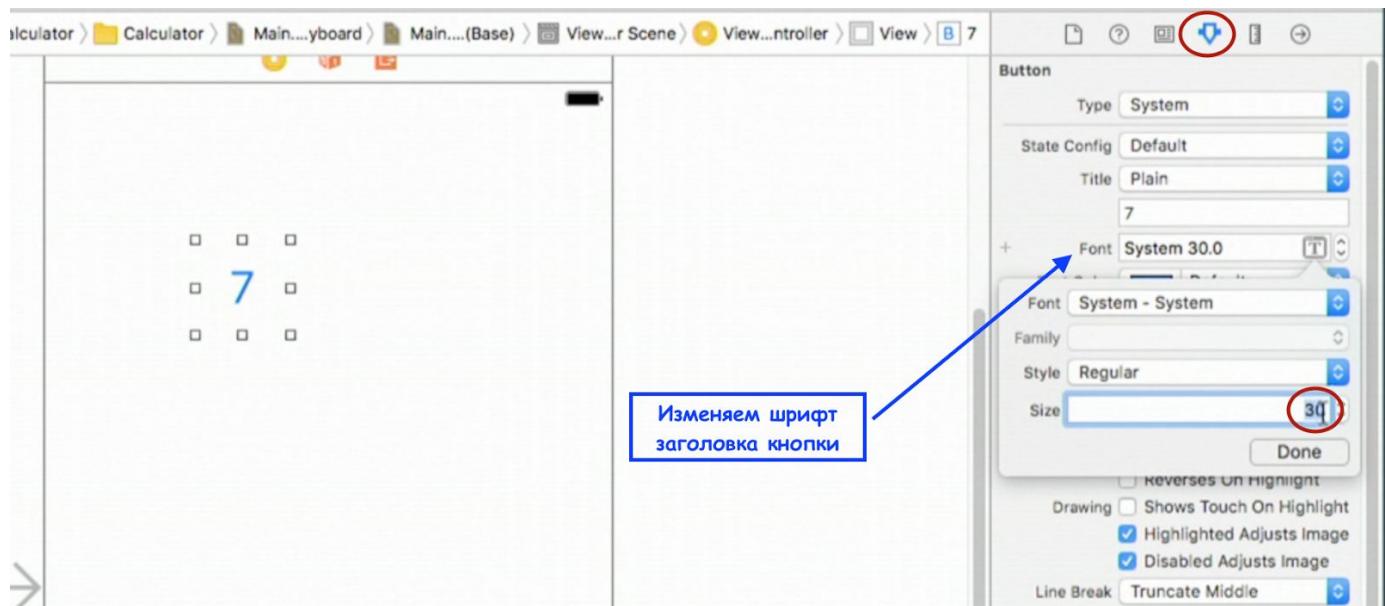


Вместо того, чтобы изменять размер кнопки ее растяжением или сжатием, можно просто напечатать размер. Например, вместо 64 x 64 напечатать какие-то другие числа.
Инспектор атрибутов имеет объектно-ориентированную природу:



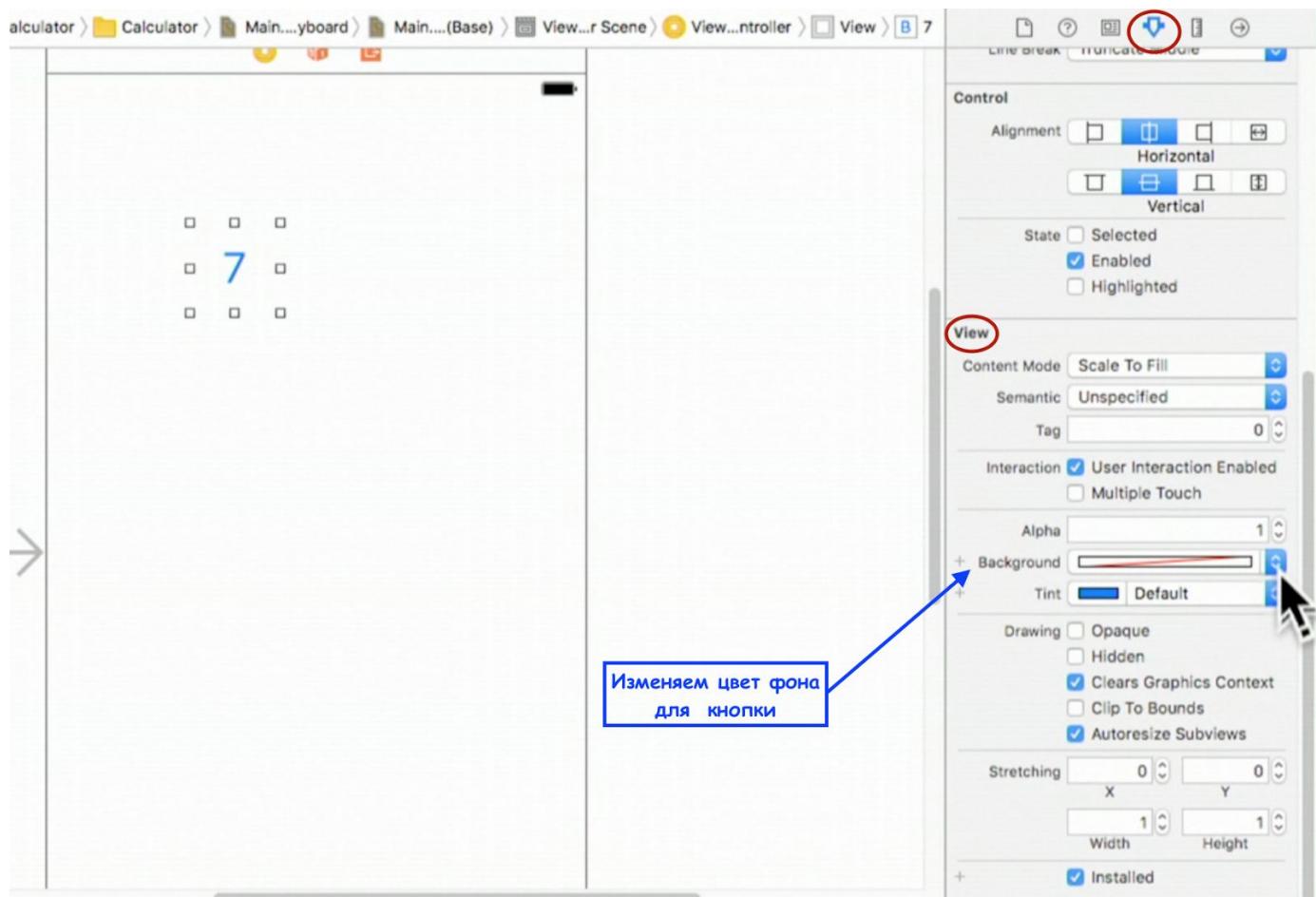
Здесь я могу устанавливать атрибуты того, что выбрано, то есть кнопки **Button**, но **Button** - это обычный объектно-ориентированный класс, который наследует от класса **Control**, и мы можем видеть атрибуты объекта **Control**, который является **superclass** класса **Button**. В свою очередь объект **Control**, наследует от объекта **View**, который является **superclass** **Control**. И мы можем видеть атрибуты объекта **View**. Таким образом, благодаря объектно-ориентированной природе Инспектора Атрибутов мы можем инспектировать атрибуты не только самого объекта, но и целой иерархии наследования.

Если я хочу изменить размер шрифта, то нахожу поле **Font**, кликаю и изменяю размер на 30.

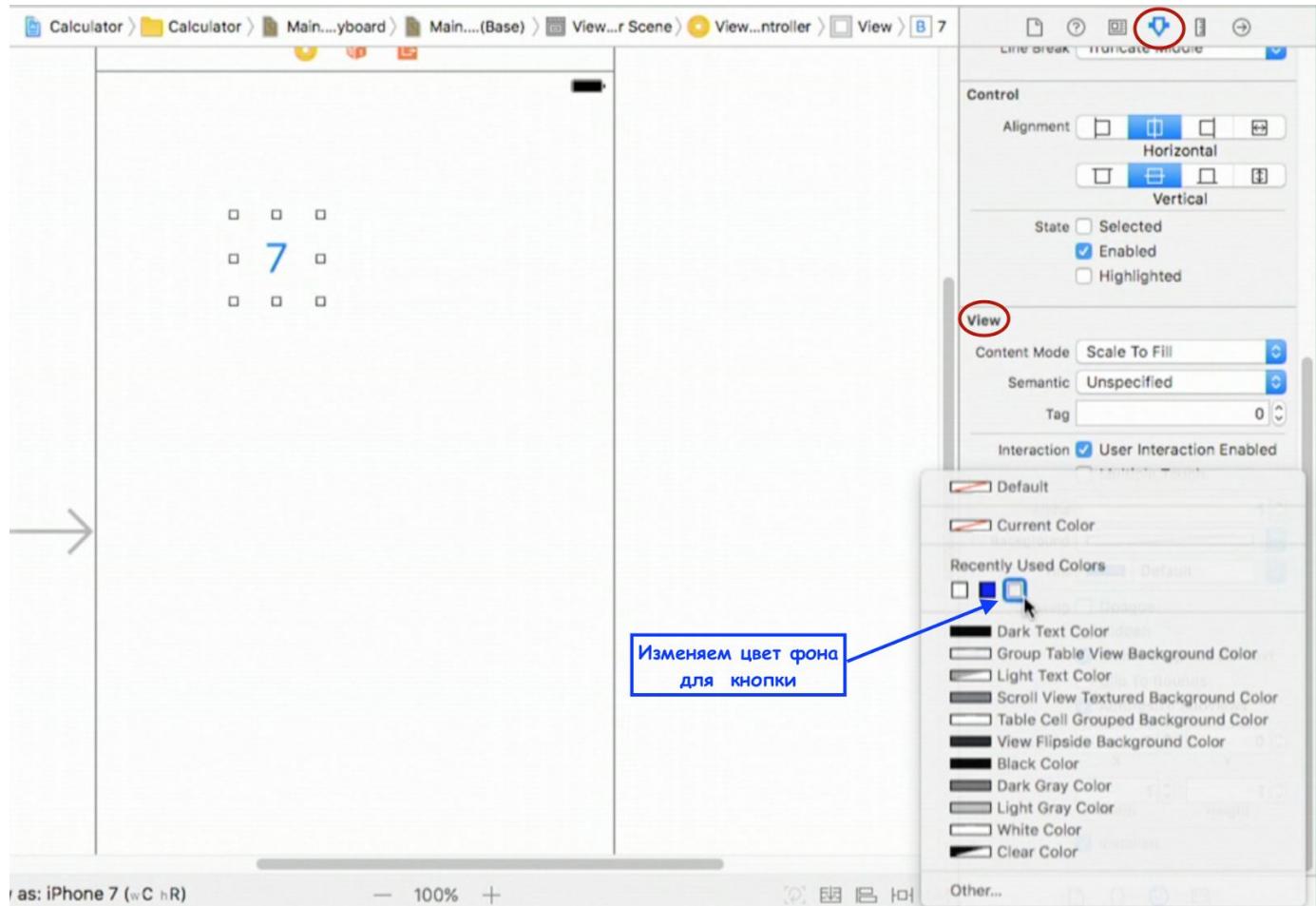


Выглядит хорошо.

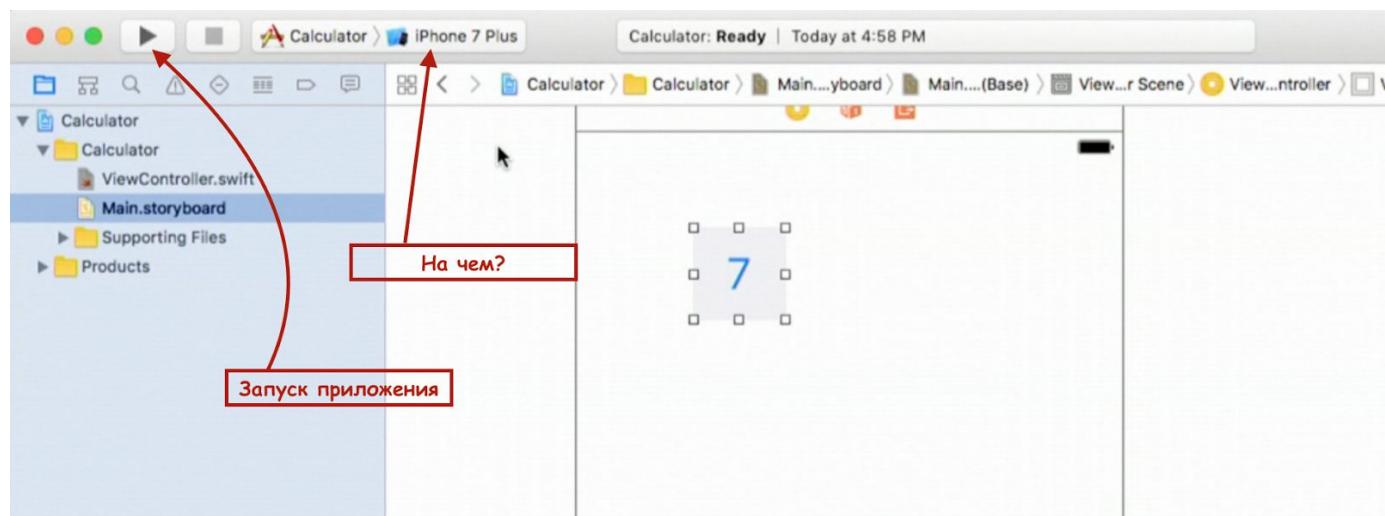
Теперь я хочу изменить цвет фона для моей кнопки, но я не нахожу поля **Background Color** в объекте **Button**. Оно есть только в объекте **View**:



Я кликаю на этом поле и выбираю один из предопределенных цветов, например, **Light Gray** (светло серый):

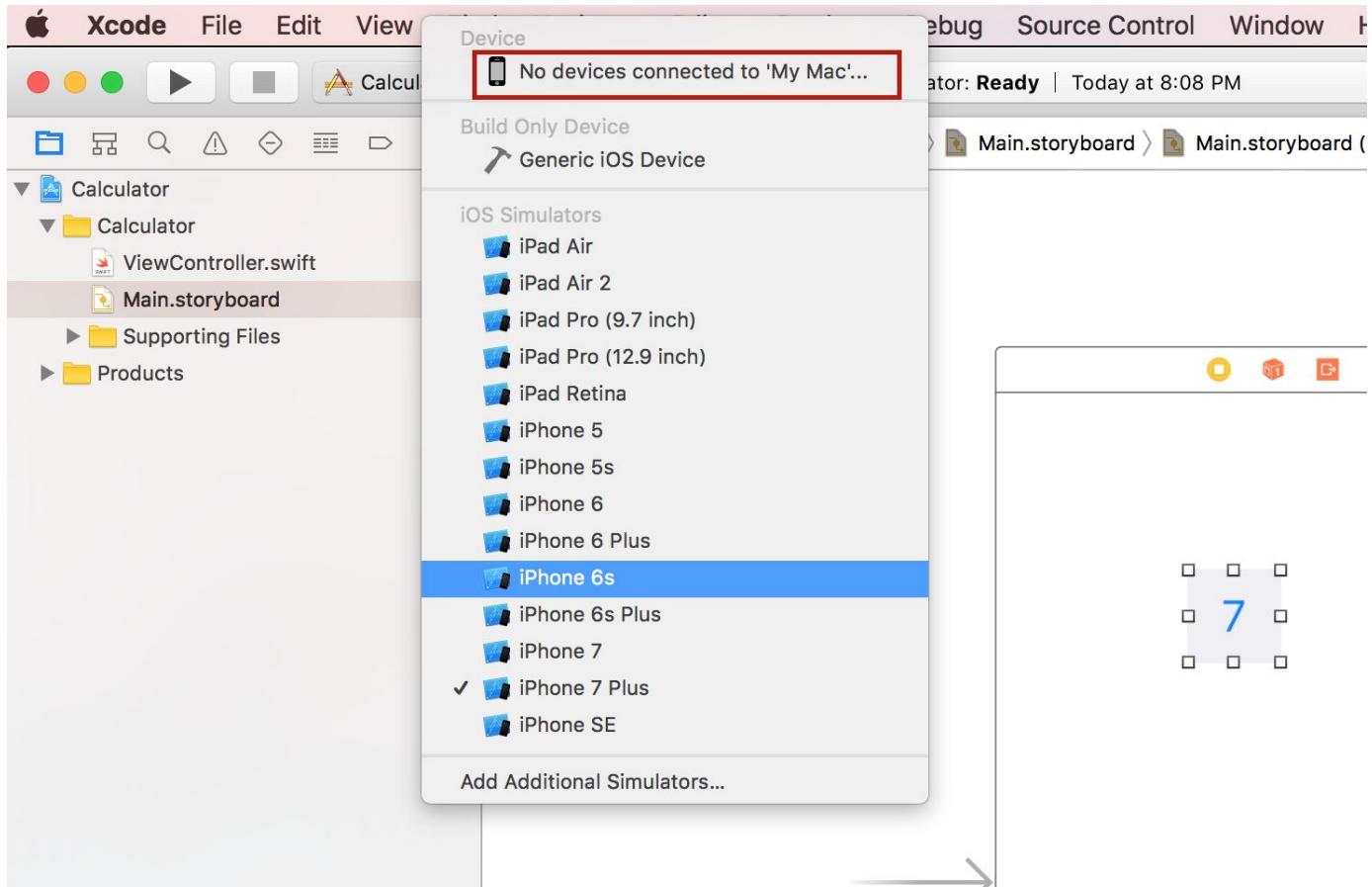


С серым цветом фона кнопка выглядит прекрасно:



Фактически, мы можем запустить наше приложение с одной кнопки и посмотреть, на что это похоже. Но сначала мы должны решить, где мы будем запускать нашу программу, потому что ее можно запустить на вашем реальном устройстве или вы можете запустить ее на симуляторе. Первые несколько недель мы будем использовать симулятор. К концу курса я попрошу вас запускать приложения на вашем реальном устройстве так, чтобы вы также привыкли и к этому. Кстати, выбор, где запускать приложение вы делаете в верхней левой части Xcode, там, где написано **Calculator iPhone 7 plus**.

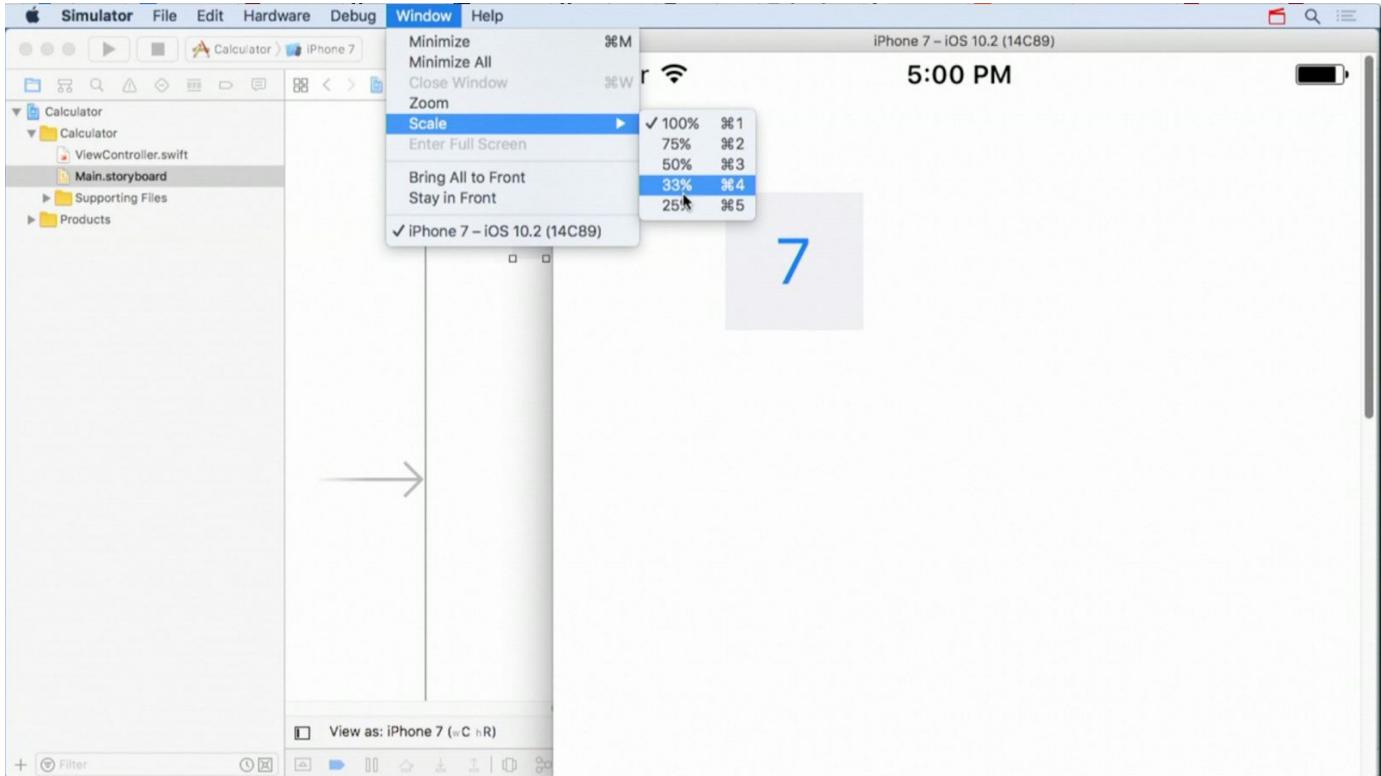
Вы кликаете на этом, и появляется целая куча симуляторов на выбор.



Здесь представлен список симуляторов всех устройств, на которых вы можете запустить iOS 10 приложение. В самой верхней строке вы можете выбрать реальное устройство, если оно подсоединенено к вашему Mac, то есть вы можете выбрать физическое устройство. Через пару недель я покажу вам, как это делается. Но сейчас просто выберем iPhone 7 и нажмем кнопку "Play" для запуска приложения:



Таким образом мы запустим наше приложение на симуляторе iPhone 7. Это полный симулятор iPhone, а не просто запуск приложения в отдельном окне.

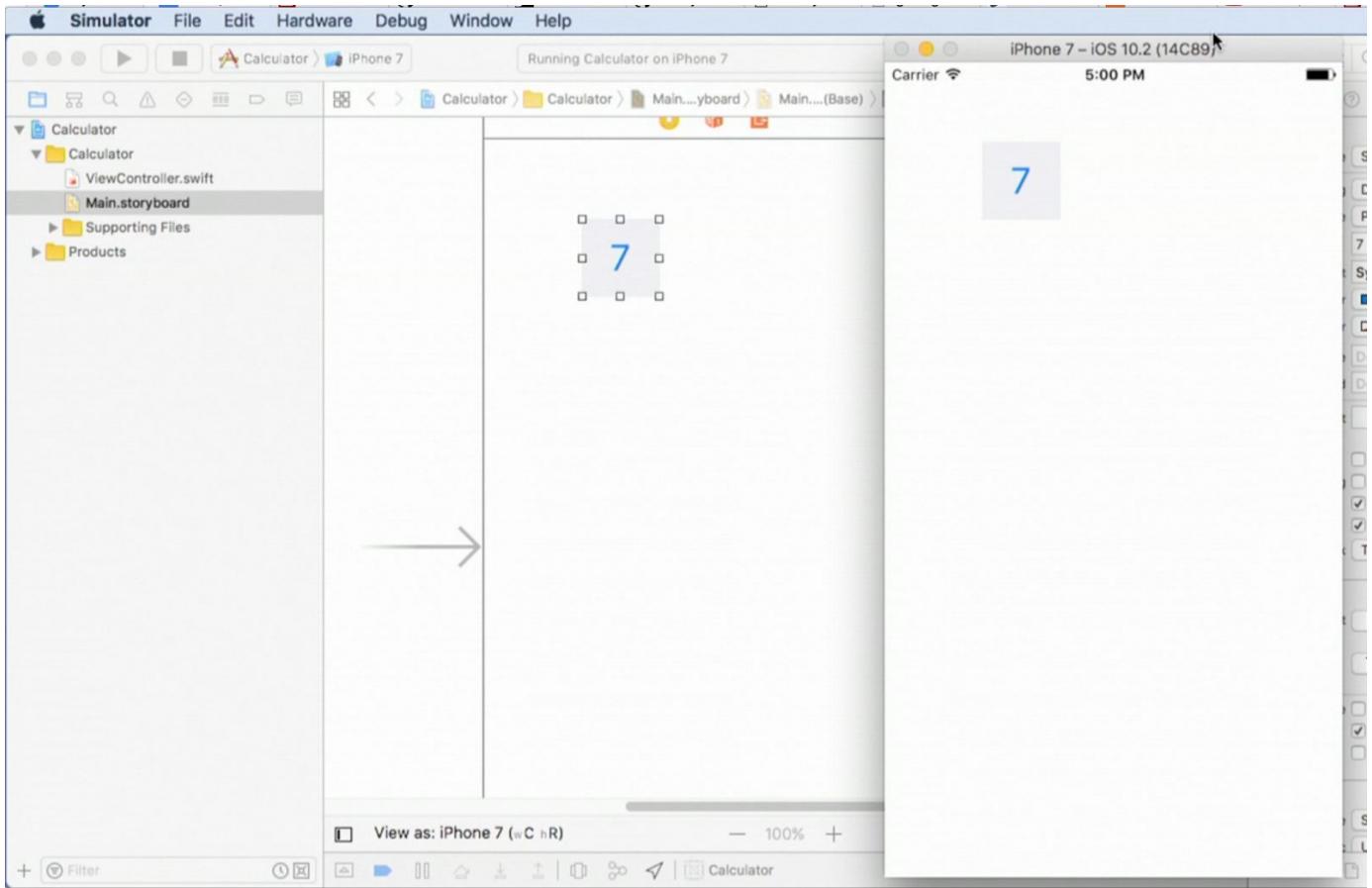


Мы получили слишком большое окно симулятора, вы видите, что масштаб сильно увеличен, так как iPhone 7 имеет очень высокое разрешение, а я запускаю на экране с маленьким разрешением, но мы можем уменьшить масштаб:



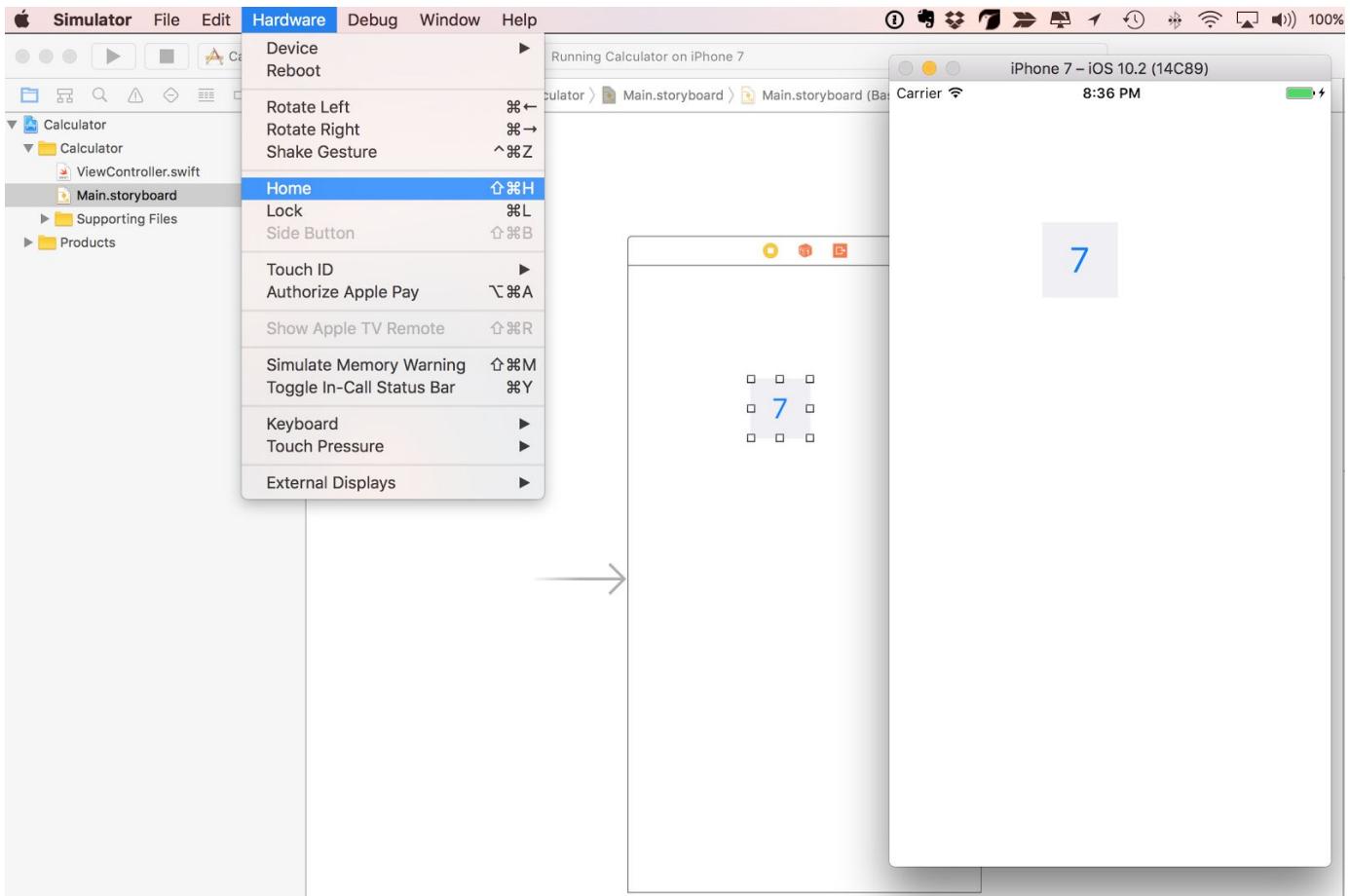
33% - слишком маленький, сделаем 50% размера iPhone 7.

----- 25 -я минута лекции -----

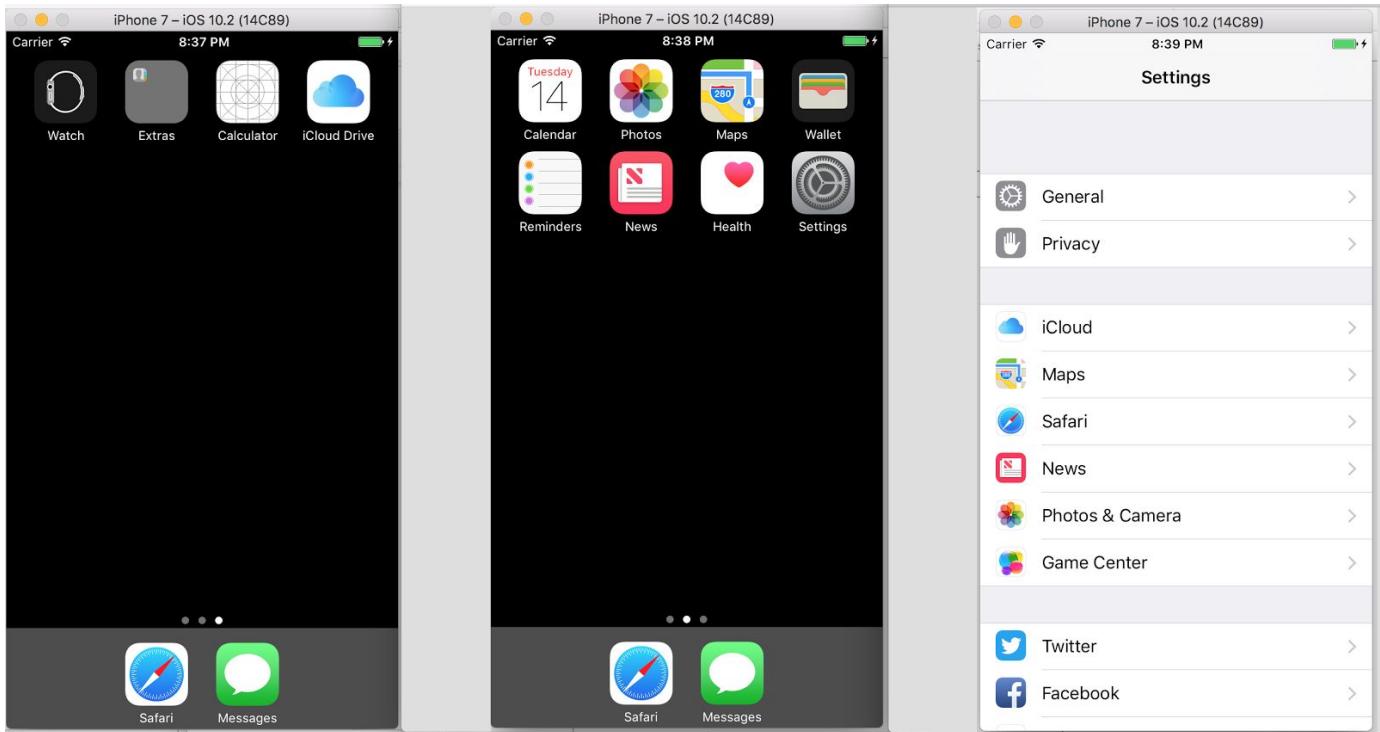


Это наша кнопка "7".

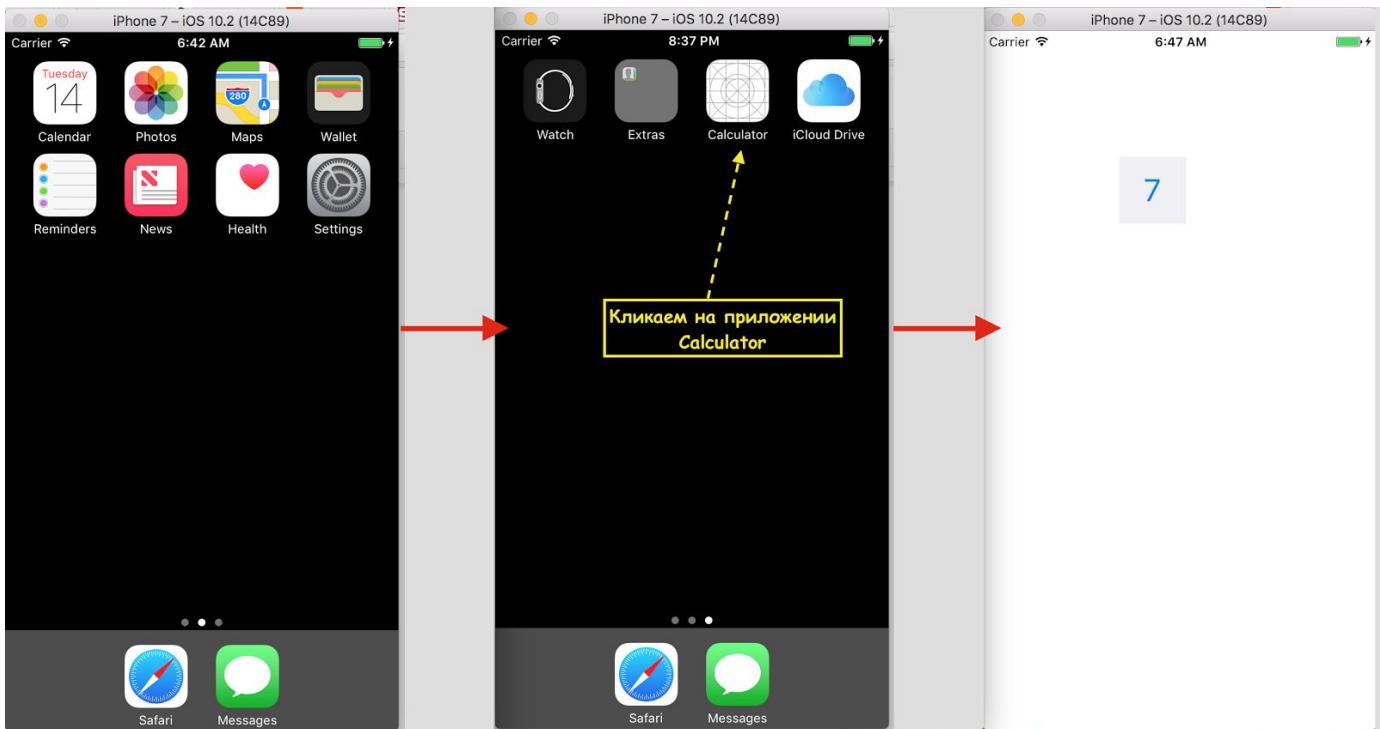
Мы можем нажать кнопку Home на iPhone 7 с помощью меню **Hardware > Home**.



И смотрите, что произойдет:



Симулятор выглядит абсолютно как iPhone. У вас даже есть Установки (**Settings**). Вы можете войти в Установки (**Settings**) и, например, авторизоваться в Twitter. Если вы используете опять клавишу Home (Cmd+Shift+H), то вернетесь обратно к приложениям. Выбираем только что созданное вами приложение **Calculator**, кликаем на нем и оказываемся внутри приложения.



Вы можете написать приложение, которое использует GPS. В этом случае вам нужно пойти в Установки (**Settings**) и разрешить GPS определять ваше местоположение. Вы сможете сделать это здесь, на симуляторе.

Итак, у нас есть кнопка “7”. Давайте кликнем на ней. Она мигает и по-видимому работает, но, конечно, ничего не делает. Мы же не сказали кнопке, что ей нужно делать, поэтому она и не делает

ничего. Давайте заставим ее что-то делать. Я говорил вам, что поведение UI описывается кодом.

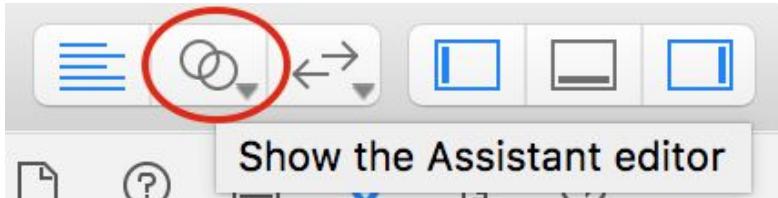
И здесь нам поможет *ViewController.swift*, в котором мы можем написать код, который будет

определять поведение кнопки.

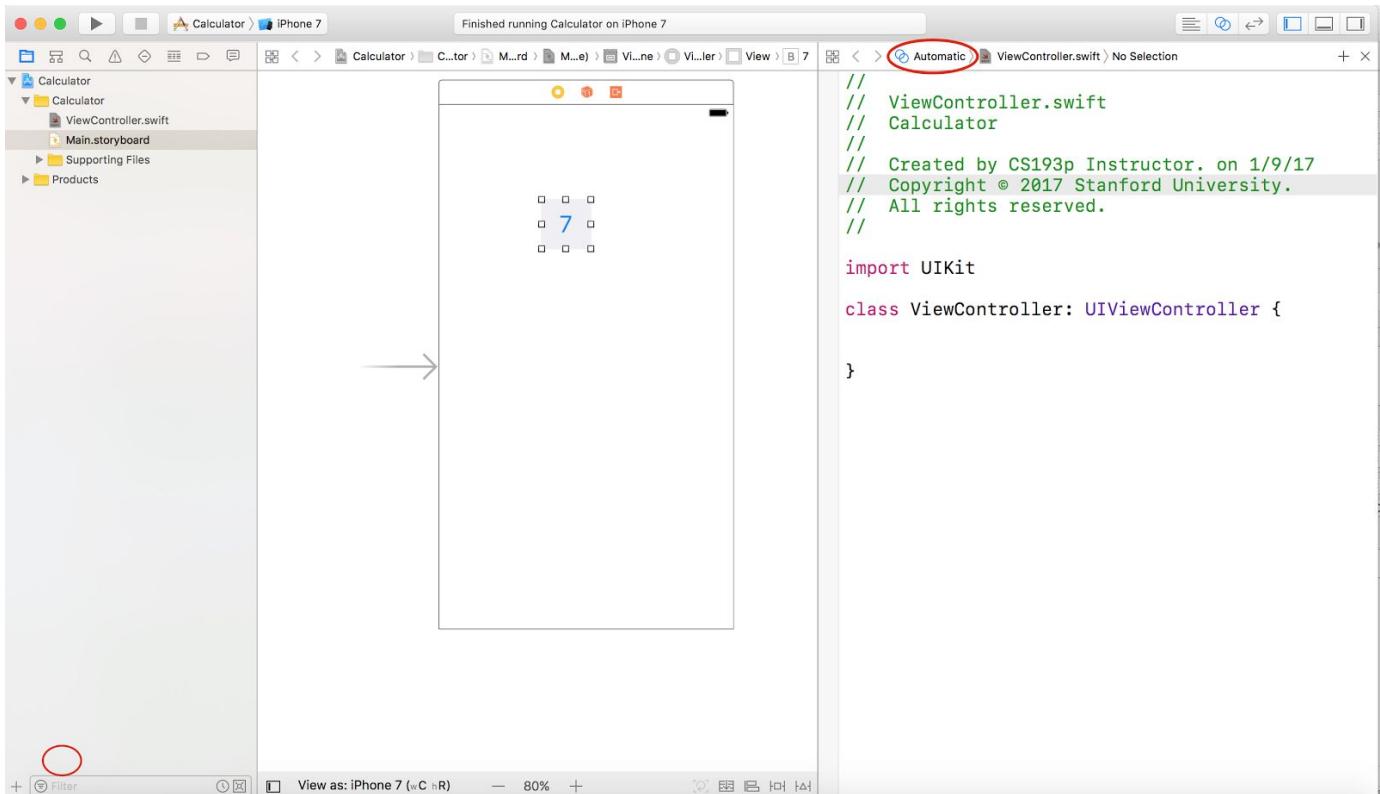
Но как нам подцепить UI на storyboard и код в *ViewController.swift*?

Для того, чтобы это сделать, нам нужно вначале расположить их на экране одновременно. И

сделаем это мы с помощью кнопки Ассистента Редактора (**Assistant Editor**) в правом верхнем углу экрана:



Я кликаю на ней, и мы получаем сразу два файла на экране. Если на одной стороне я показываю свой UI, то на другой стороне автоматически показывается код, потому что **Xcode** уже знает, что мне, возможно, может понадобиться.



Xcode поступает так, потому что видите? Вверху написано **Automatic**. Этот файл справа выбран автоматически. Вы можете кликнуть на **Automatic**. И заменить режим показа на **Manual** и выбрать тот файл, который вы хотите показать на этой стороне:

```
// Manual
// Automatic (1)
// Top Level Objects (1)
// Localizations
// Notification Payloads
// Preview (1)
// Created by CS193p Instructor. on 1/9/17
// Copyright © 2017 Stanford University.
// All rights reserved.

import UIKit

class ViewController: UIViewController {
```

Но в большинстве случаев вы захотите оставить режим **Automatic** и сохранить чувствительность правой части к выбору файла в левой части.

Давайте освободим больше пространства для кода и посмотрим на **Swift**, который вы видите сейчас впервые:

```
// Automatic > ViewController.swift > No Selection
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor. on 1/9/17
// Copyright © 2017 Stanford University.
// All rights reserved.

import UIKit

class ViewController: UIViewController {
```

Swift - действительно прекрасный язык, потому что он очень лаконичный и очень понятный, благодаря правильному выбору ключевых слов.

Слово **import** - это как **include** в большинстве языков, оно просто говорит о том, что я хочу

использовать фреймворк с именем **UIKit**.

Наш код будет управлять поведением пользовательского интерфейса (UI), поэтому, конечно, нам нужно использовать **UIKit**. Если мы будем писать объект, который похож на “внутренность” Калькулятора, которая не зависит от **UI**, то мы будем импортировать **Foundation** вместо **UIKit**. Помните? **Foundation** соответствует слою **Core Services**, который не связан с **UI**. Вы никогда не будете импортировать **UIKit** в одном из таких **НЕ-UI** классов. Вы все это увидите в Среду, когда мы будем применять **MVC**. Конечно, возможно, что мы захотим импортировать еще что-то, например, **MapKit**.

Здесь вы видите первую декларацию **Swift** класса:

```
import UIKit

class ViewController: UIViewController {  
}
```

Конечно, ключевое слово **class**. Затем имя этого класса. **ViewController** - это имя класса, слишком общее (**generic**) имя, не отражающее специфики того, что делает этот класс. Возможно, следует придумать более подходящее имя для класса, если у вас есть такая возможность, но в нашем случае мы получили такое имя из шаблона.

К сожалению, мы не можем переименовать этот класс, просто перепечатав имя, потому что он уже связан с **UI**. Позже я покажу вам, как это сделать корректно. А в данный момент вы “привязаны” к этому обобщенному имени. Затем следует двоеточие : и **UIViewController**. Это означает, что наш класс **ViewController** наследует от **UIViewController**. Это объектно-ориентированное программирование и это - наследование (**inheritance**). Повторяюсь, если вы сидите в этой комнате, значит вы знаете объектно-ориентированное программирование и вы знаете, что означает наследование. Swift - язык программирования с единичным наследованием. Вы можете наследовать только от одного класса и в нашем случае класс **ViewController** наследует только от **UIViewController**. Возможности класса **UIViewController** состоят в том, что он знает, как управлять (**control**) пользовательским интерфейсом (**UI**). Поэтому он и называется **UIViewController**, он знает как управлять View - это то, что находится слева, когда на экране были и код, и **UI**.

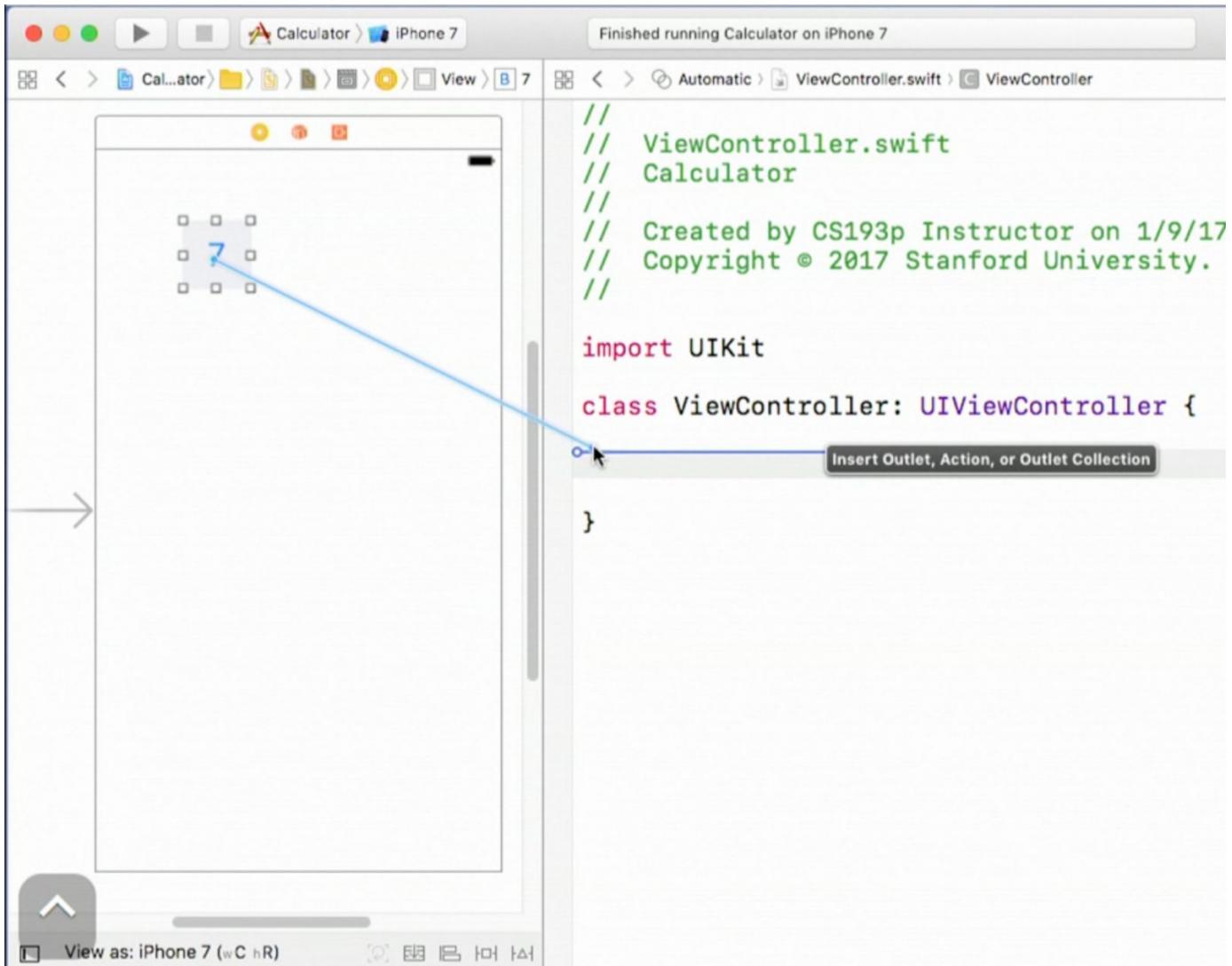
Наш класс **ViewController** наследует всю функциональность класса **UIViewController**, то есть способность управлять **UI**. Что совершенно замечательно, потому что именно этого мы и хотим добиться.

И затем в фигурных скобках мы хотим разместить все наши переменные экземпляра класса (instance variables) и методы (methods). Надеюсь, что все знают, что такое переменные экземпляра класса (instance variables)? Переменные экземпляра класса (instance variables) - это “хранилище” переменных внутри нашего класса, а методы - это просто функции внутри класса. Между прочим, в Swift мы называем Переменные экземпляра класса (instance variables) свойствами (properties). Поэтому если вы услышите от меня термин “свойства (properties)”,

имейте в виду, что я говорю о переменных экземпляра класса (**instance variables**). Ну а методы мы называем методами.

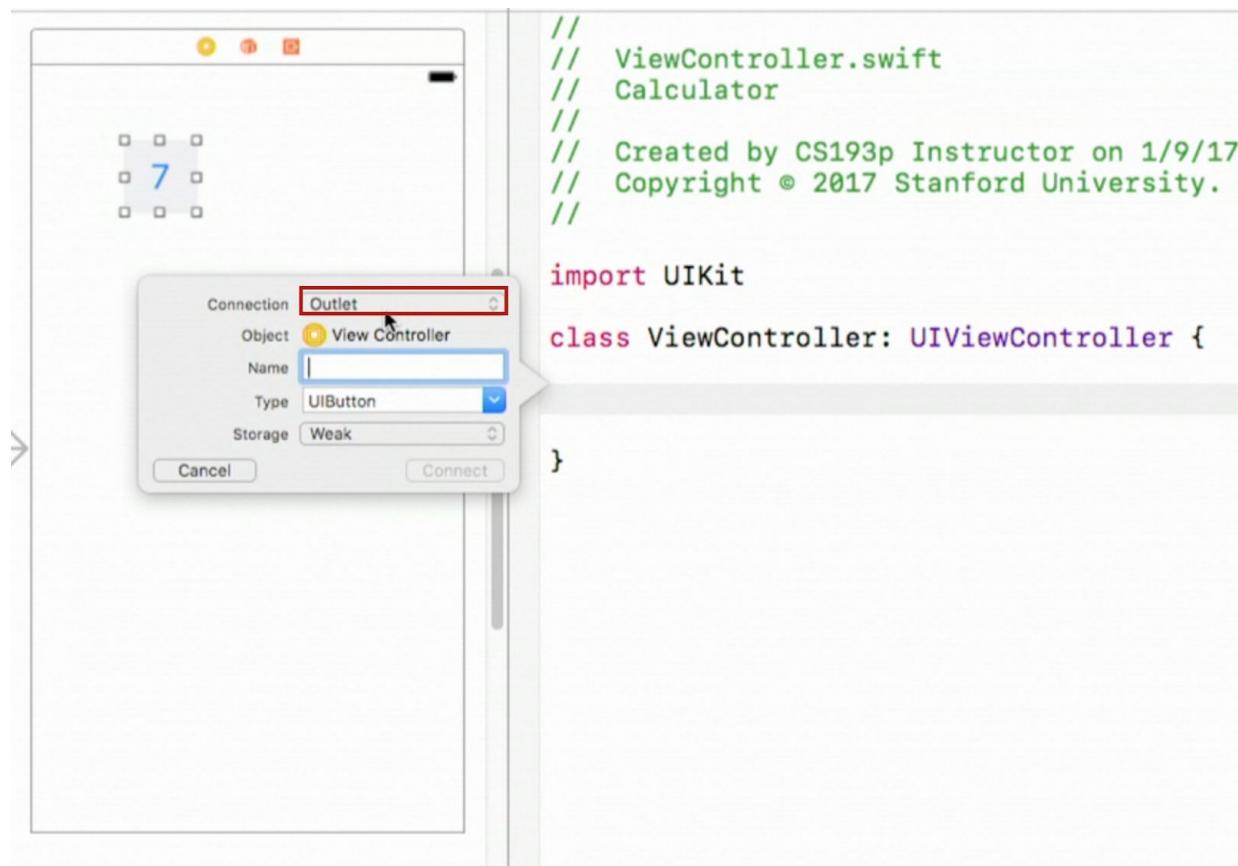
В действительности я хочу, чтобы при нажатии на кнопку у меня срабатывал метод моего класса. Я хочу, чтобы при нажатии на кнопку вызывался метод, и это было бы прекрасно, потому что в этом методе я могу разместить какой угодно код. А мне это и нужно.

Способ, каким мы будем это делать, несколько замысловатый. Я буду удерживать нажатой клавишу **CTRL** и буду тянуть узате "мышки" от моей кнопки в код, собираясь их соединить:



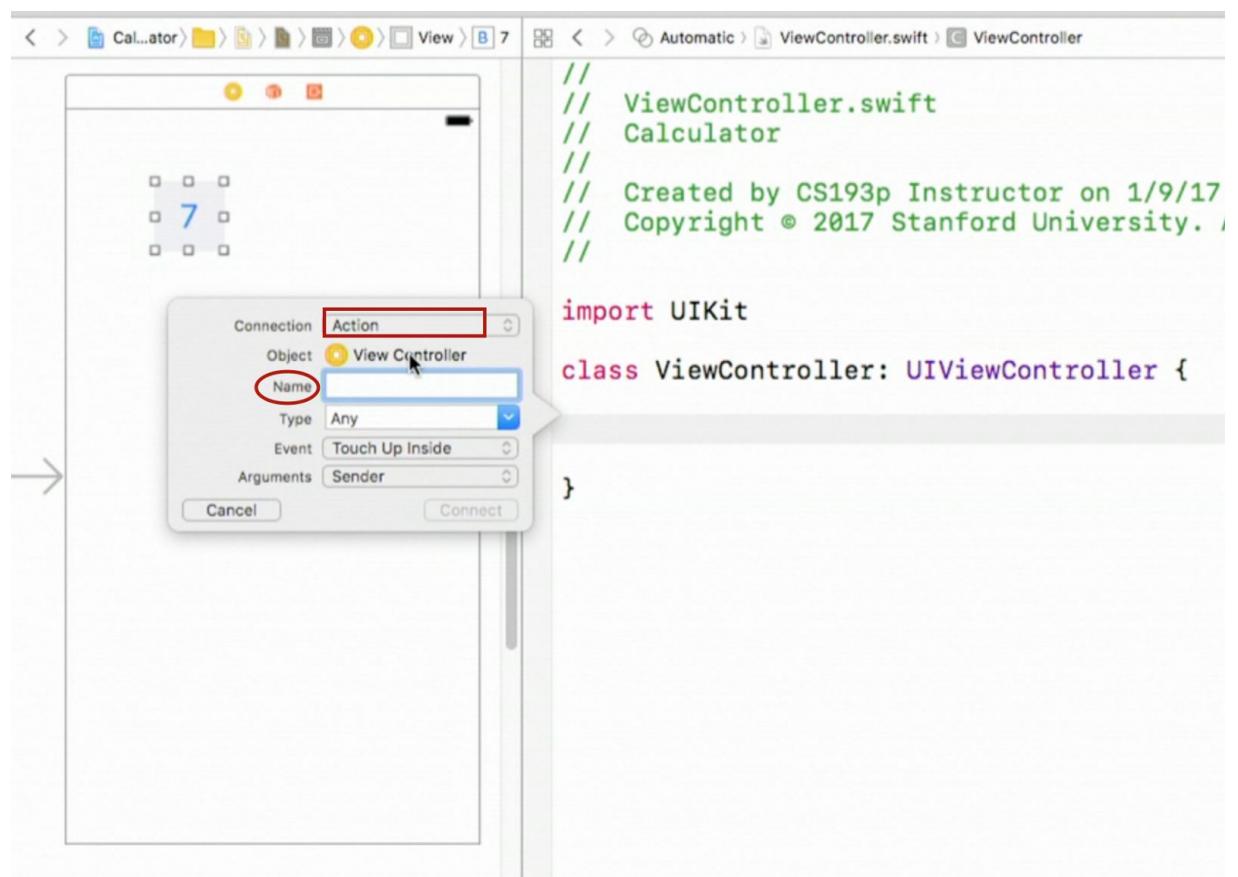
Вы видите на экране фиксируется, что нажата клавиша **CTRL**. И, когда я отпускаю клавишу **CTRL**, у вас появляется возможность осуществить взаимосвязь между вашим UI и кодом:

----- 30 -ая минута лекции -----

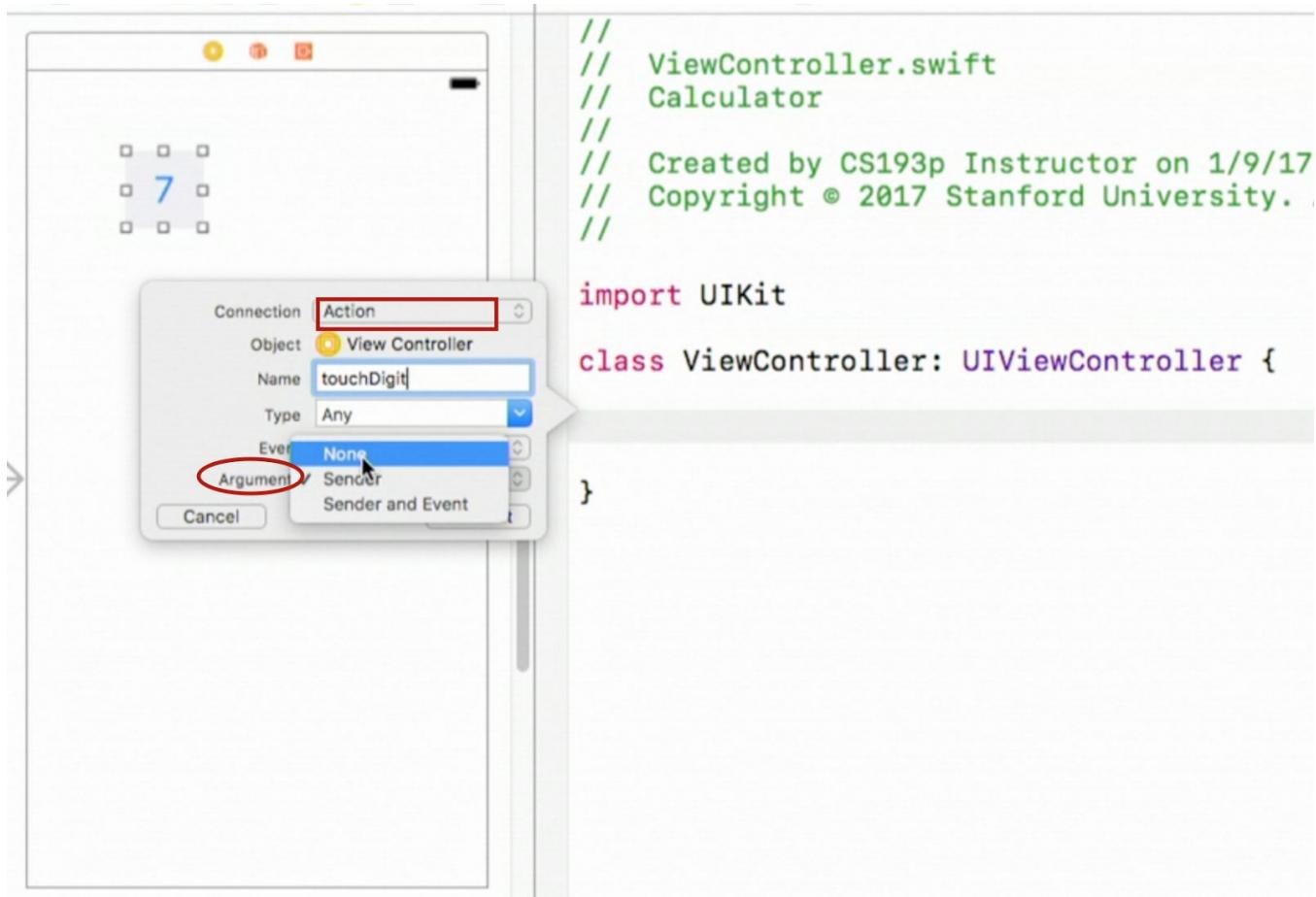


Нас спрашивают, какого типа взаимосвязь нам нужна? И у нас два выбора. Здесь представлена связь типа **Outlet**, что означает создание свойства или переменной экземпляра класса (**instance variable**), которая указывает на кнопку и я смогу “разговаривать” с кнопкой.

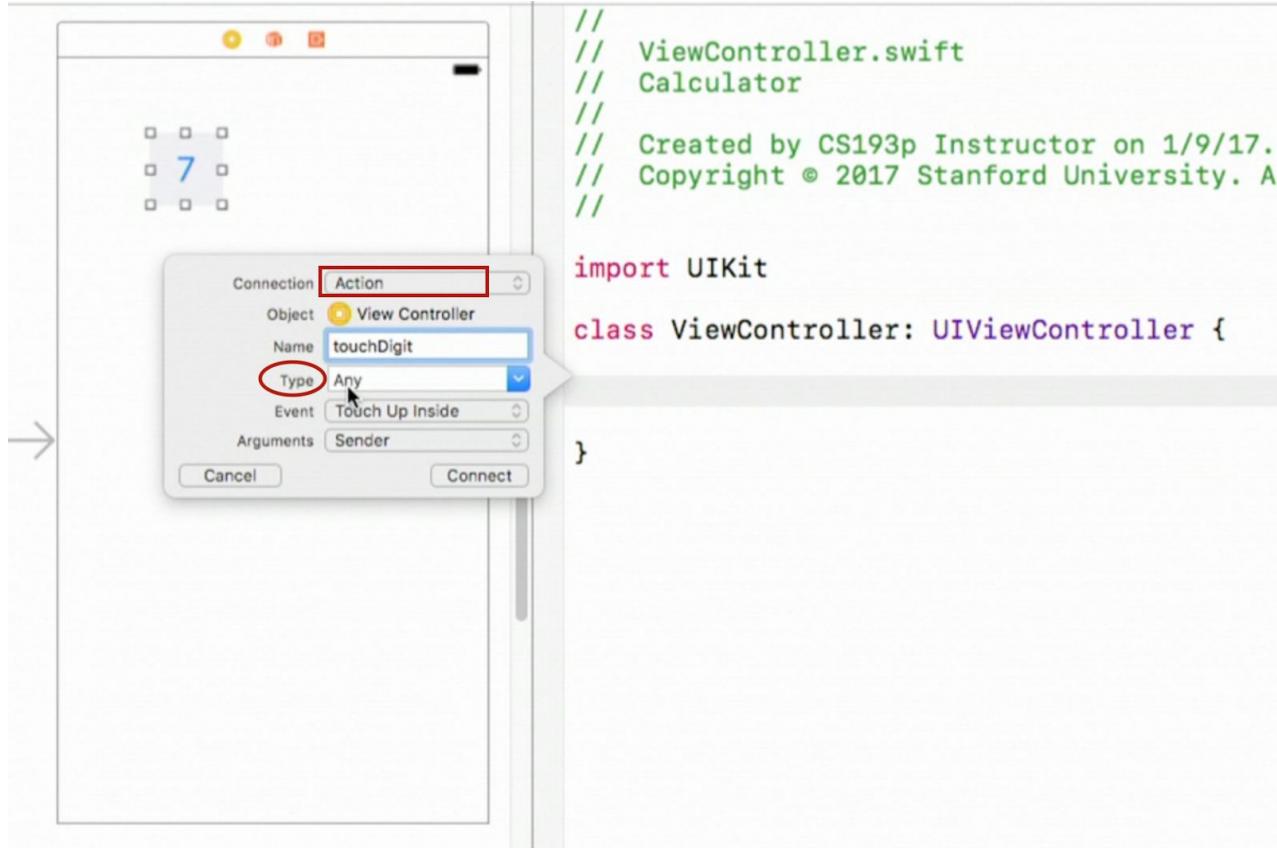
А еще есть связь типа **Action**, что означает создание метода, который вызывается при нажатии на эту кнопку.



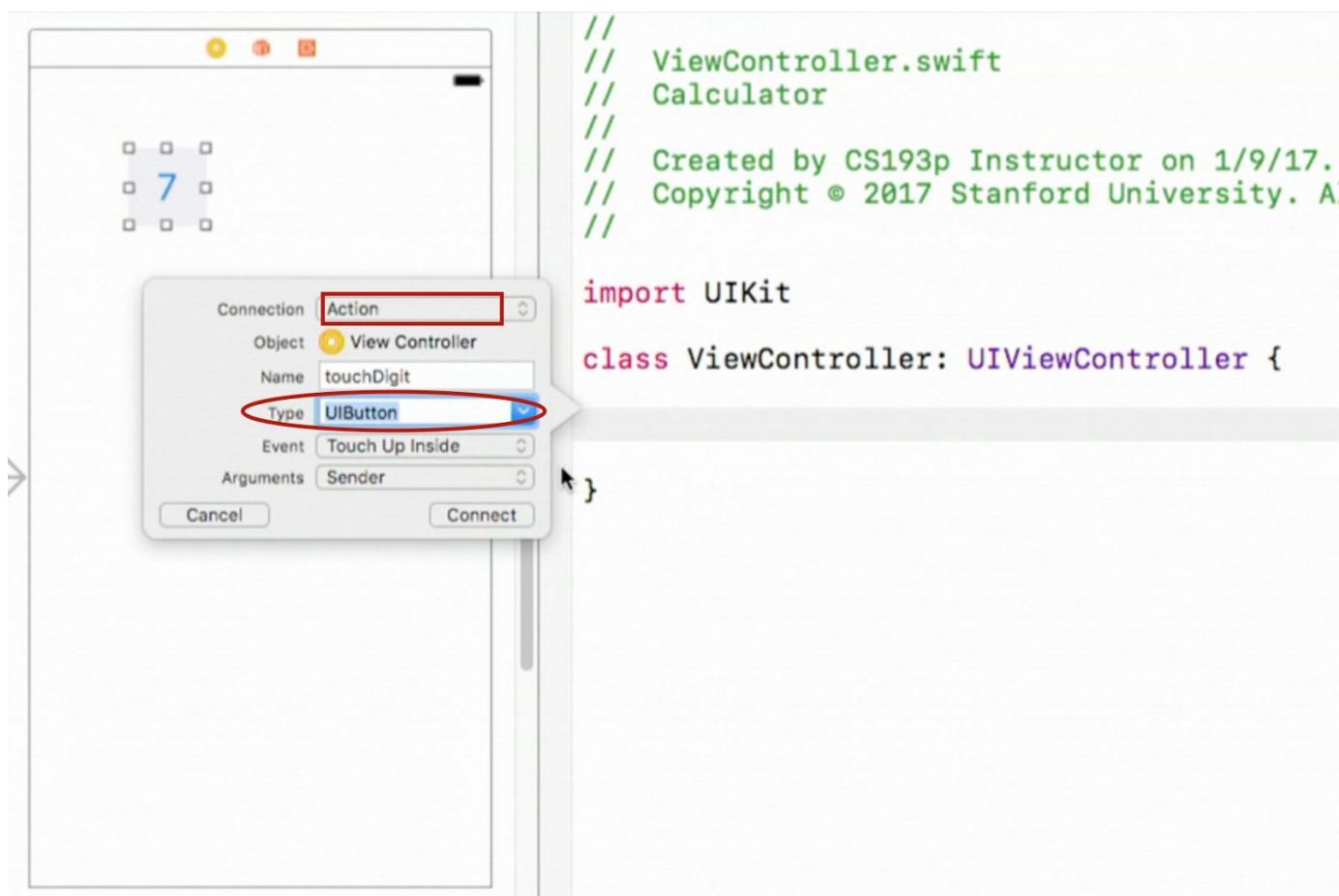
Всем понятно? В этом случае запрашивается имя метода в поле **Name** и я назову мой метод **touchDigit**.



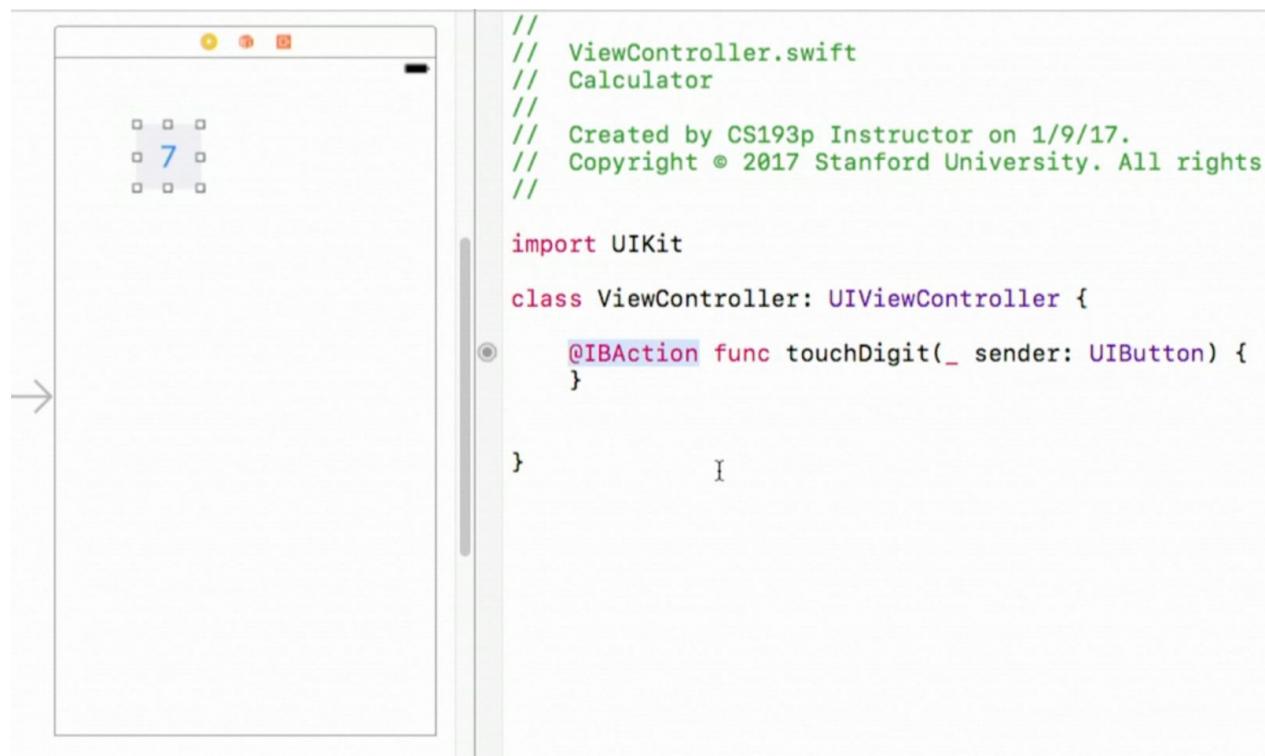
Метод может иметь аргументы. Он может вообще не иметь аргументов, а может иметь один аргумент, которым является кнопка, посылающая мне это сообщение. В нашем случае мне нужен аргумент, потому что я хочу иметь один метод для всего набора кнопок. Следовательно, я вынужден спрашивать кнопку, посылающую мне сообщение, кто ты такая? Какой у тебя заголовок? Ты кнопка “7”? Или ты кнопка “5”? Поэтому мне нужен **Sender**.



В поле **Type** стоит **Any**, это тип аргумента. Пора проснуться, если у вас дневной сон, и обязательно запомнить, что при выполнении Домашнего Задания вы не должны оставлять в этом поле **Any**. Потому что мы точно знаем тип объекта, посылающего нам это сообщение. Это **UIButton**, так что вам нужно заменить **Any** на **UIButton**.



Если вы пропустите этот шаг, то ваш код испортится, потому что типом аргумента sender будет **Any**, что практически всегда означает, что тип не определен. А это плохо. Так что убедитесь, что вы заменили **Any** на **UIButton**. Когда мы нажимаем “**Connect**”, то создается метод.



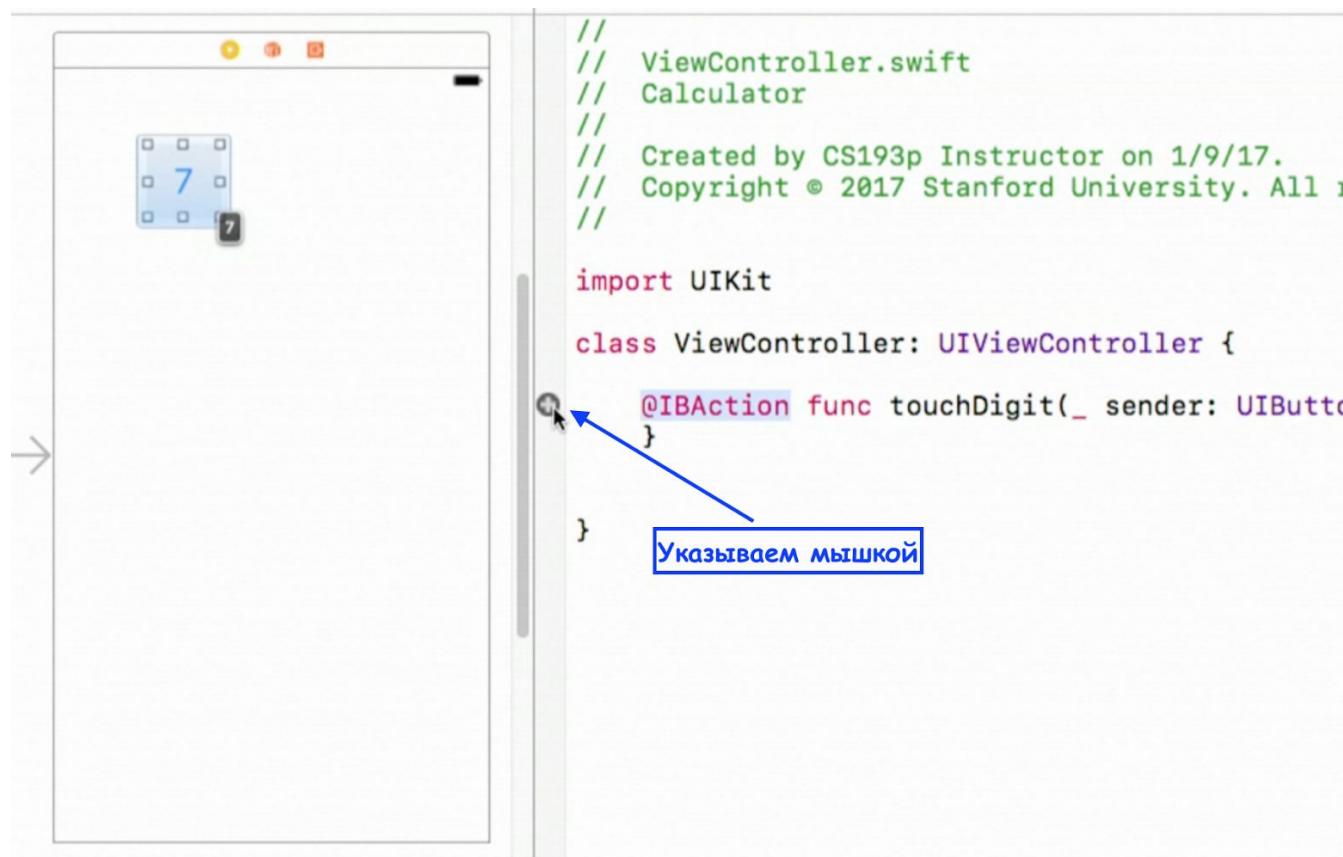
```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
    }
}
```

Вы получили первый **Swift** метод. Выражение **@IBAction** в действительности не является частью языка **Swift**. Это не часть метода. Просто **Xcode** разместил здесь что-то для себя. Слева от него появился маленький кружок. Если вы наведете на него “мышку”, не кликая, то смотрите, что происходит. Вам покажут к чему на **UI** «подцеплен» ваш метод.



```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

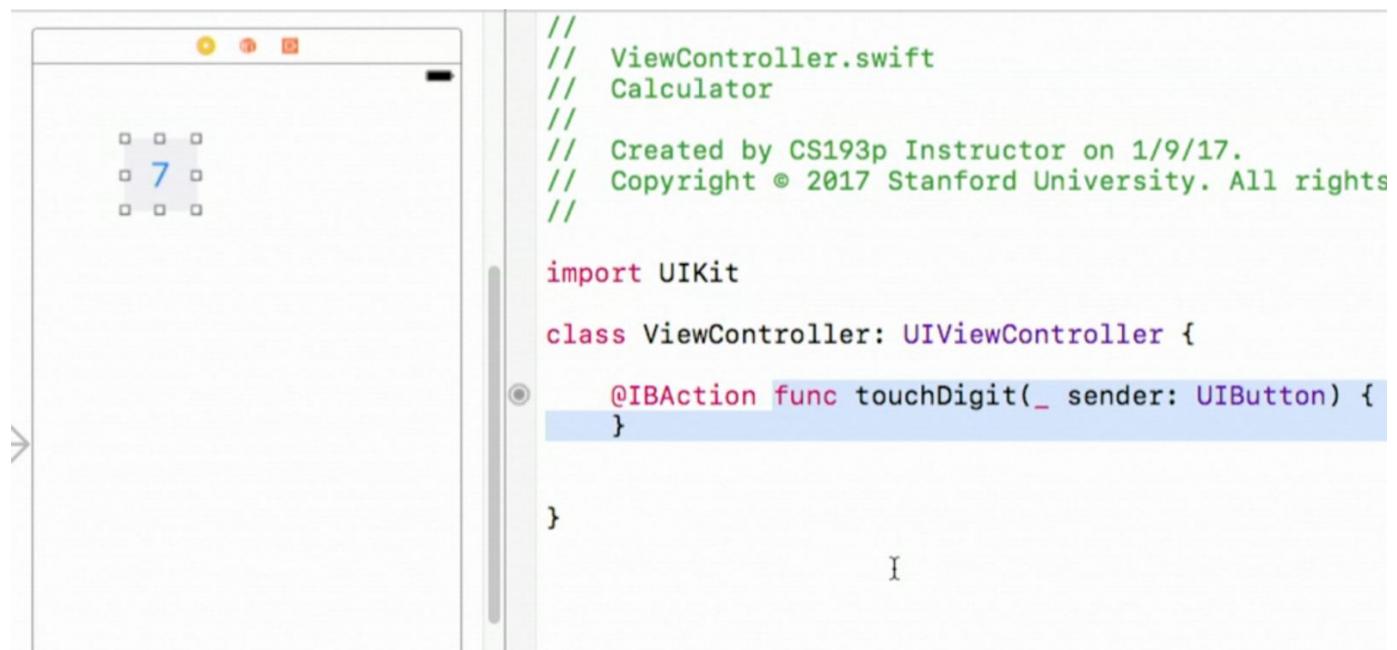
class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton)
}

```

Другими словами, выделяется элемент пользовательского интерфейса, который посыпает мне это сообщение. Так что это не элемент языка **Swift**, это принадлежность **Xcode**.

Оставшаяся часть - это **Swift** метод.



The screenshot shows the Xcode interface. On the left is a storyboard preview window showing a single button with the number '7' on it. To its right is a code editor window displaying a Swift file named 'ViewController.swift'. The code contains a class definition for 'ViewController' that includes an '@IBAction' method named 'touchDigit'.

```
// ViewController.swift
// Calculator
//
// Created by CS193P Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
    }
}
```

Давайте посмотрим на части метода **Swift**, на его синтаксис, чтобы лучше его понимать.

Я приведу здесь другой метод, не связанный с нашей задачей:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
            ...
        }
    }
}
```

Это пример метода **Swift**. Заметьте, что у него три параметра. Видите? Часть ": Double" - это тип параметра. В данном случае параметр имеет тип **Double** - число двойной точности с плавающей точкой. Второй параметр также имеет тип **Double**, а третий параметр имеет тип **UIColor**, это совершенно другой тип, описывающий цвет. Интересно, что каждый параметр имеет два имени: **from startX, to endX, using color**, у каждого параметра два имени. Зачем нам нужны эти два имени? Первое имя - это внешнее (**external**) имя параметра, а второе - внутреннее (**internal**) имя параметра.

Внутренние имена, такие, как **startX, endX** и **color** будут использовать в вашем коде внутри этого метода. Например, я могу написать:

```

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
    }

    func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
        distance = endX - startX
    }

}

```

Видите? Я использую **endX** и **startX** внутри этого метода. Это часть его реализации (**implementation**).

Внешние имена используются теми, кто вызывает этот метод. Например, я могу вызвать метод **drawHorizontalLine** из метода **touchDigit**:

The screenshot shows a portion of an Xcode editor window. The code being typed is:

```

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        drawH
    }

```

The cursor is at the end of "drawH". A completion dropdown menu is open, showing several suggestions:

- M** Void **drawHorizontalLine**(from: Double, to: Double, using: UIColor)
- f** Void **CTFontDrawGlyphs**(font: CTFont, glyphs: UnsafePointer<CGGlyph>)
- M** Void **decodeRestorableState!**(with: NSCoder)
- M** Void **decodeRestorableState**(with: NSCoder)
- V** SSLCipherSuite **SSL_DH_RSA_WITH_DES_CBC_SHA**
- V** SSLCipherSuite **SSL_DHE_RSA_WITH_DES_CBC_SHA**
- V** SSLCipherSuite **SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA**
- V** SSLCipherSuite **TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA**

Между прочим, **Xcode** любит печатать с целью подсказки. Я просто напечатаю "**drawH**" и нажимаю **Tab** и еще раз **Tab**, и **Xcode** не только заполняет мне полное имя метода, но и показывает все аргументы, по которым я могу передвигаться с помощью **Tab** и начинать их печатать.

```

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        drawHorizontalLine(from: Double, to: Double, using: UIColor)
    }

    func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
    }

}

```

Заметьте, что для параметров остались только внешние имена: **from**, **to** и **using**. Поэтому вы можете для них использовать значения - **from: 5.0, to: 8.5, using: UIColor.blue**

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        drawHorizontalLine(from: 5.0, to: 8.5, using: UIColor.blue)
    }

    func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
    }

}
```

Имена **from**, **to** и **using** - вот что используют вызывающие функции. Заметьте также, что все эти имена являются обязательными.

Вы не можете их опустить и написать так, как это принято в других языках:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        drawHorizontalLine(from: 5.0, to: 8.5, using: UIColor.blue)
        drawHorizontalLine(5.0, 8.5, .blue)
    }

    func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
    }

}
```

drawHorizontalLine(5.0, 8.5, .blue)

Так нельзя написать

Вы должны использовать внешние имена для функции **drawHorizontalLine**. Заметьте также, что имена внешних параметров подобраны так, чтобы фраза целиком правильно читалась по-английски: “Draw a horizontal line from 5.0 to 8.5 using blue” (Нарисовать горизонтальную линию от 5.0 до 8.5 используя голубой).

----- 35 -ая минута лекции -----

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        drawHorizontalLine(from: 5.0, to: 8.5, using: UIColor.blue)
    }

    func drawHorizontalLine(from startX: Double, to endX: Double, using color: UIColor) {
    }

}
```

Swift максимально стремится представить название метода в сочетании с внешними именами параметров как разговорный Английский, если это в его силах. Итак, это своего рода ускоренный курс. Вы будете читать обо всем этом в вашем Задании на

чтение, там об этом написано более подробно.

Между прочим, если ваша функция что-то возвращает, то это оформляется в виде “стрелочки”:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) -> String {
    }

}
```

В данном случае возвращается з типа **String**.

Однако вернемся к нашему первоначальному виду функции **touchDigit**:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
    }

}
```

Она кажется странной из-за двух вещей.

Во-первых, вместо внешнего имени параметра стоит знак “подчеркивания” “_”.

Означает ли это, что я могу вызвать функцию как **touchDigit (_:что-то)**? Нет, присутствие знака “подчеркивания” означает, что у параметра вообще нет внешнего (**external**) имени. Таким образом, вы можете вызвать функцию **touchDigit** так:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        touchDigit(someButton)
    }

}
```

В качестве аргумента используется объект **someButton** типа **UIButton**. В этом случае вам не нужно имя параметра. Если бы вместо знака “подчеркивания” “_” стояло имя **foo**, то вам пришлось бы указать его при вызове функции **touchDigit**:

```
import UIKit

class ViewController: UIViewController {
    @IBAction func touchDigit(foo: sender: UIButton) {
        touchDigit(foo: someButton)
    }
}
```

Но там нет **foo**, вместо него стоит знак “подчеркивания” “ ”, так что вам не нужно задавать имя параметра при обращении к функции **touchDigit**:

```
import UIKit

class ViewController: UIViewController {
    @IBAction func touchDigit(_ sender: UIButton) {
    }
}
```

Почему нам иногда не нужно внешнее имя первого параметра? Потому что иногда оно неявно указывается либо в имени функции, либо в типе аргумента, который вы предположительно передаете, и получается, что вы не нуждаетесь во внешнем имени первого параметра. Между прочим, знак “подчеркивания” “ ” практически никогда не используется ни для второго, ни для третьего или четвертого аргумента.

Время от времени используется для первого аргумента, но не всегда, только изредка. Но никогда ни для второго, ни для третьего, четвертого, пятого или далее следующего. НИКОГДА. В вашем Задании на чтение (reading assignment) я указываю вам на документ, в котором объясняется, как следует именовать аргументы, какие существуют правила, когда следует использовать знак “подчеркивания” “ ”, а когда - нет. Его нужно прочесть и он будет оставаться для вас главным до конца семестра, если вы хотите называться профессиональным iOS разработчиком. Он является ключом к пониманию этого.

Очевидно, что метод имеет ключевое слово **func**, потому что это функция в этом классе:

```
import UIKit

class ViewController: UIViewController {
    @IBAction func touchDigit(_ sender: UIButton) {
    }
}
```

Затем следует имя метода. Параметры заключаются в круглые скобки и могут разделяться запятой, если их более одного. У нас единственный параметр, который имеет тип **UIButton**, потому что это сообщение нам посыпает кнопка. И **sender** - это внутреннее имя, которое мы можем использовать внутри метода для доступа к кнопке.

Но прежде, чем начать это делать, давайте сделаем что-то очень простое, например, распечатаем на консоле текст "**touchDigit was called**":

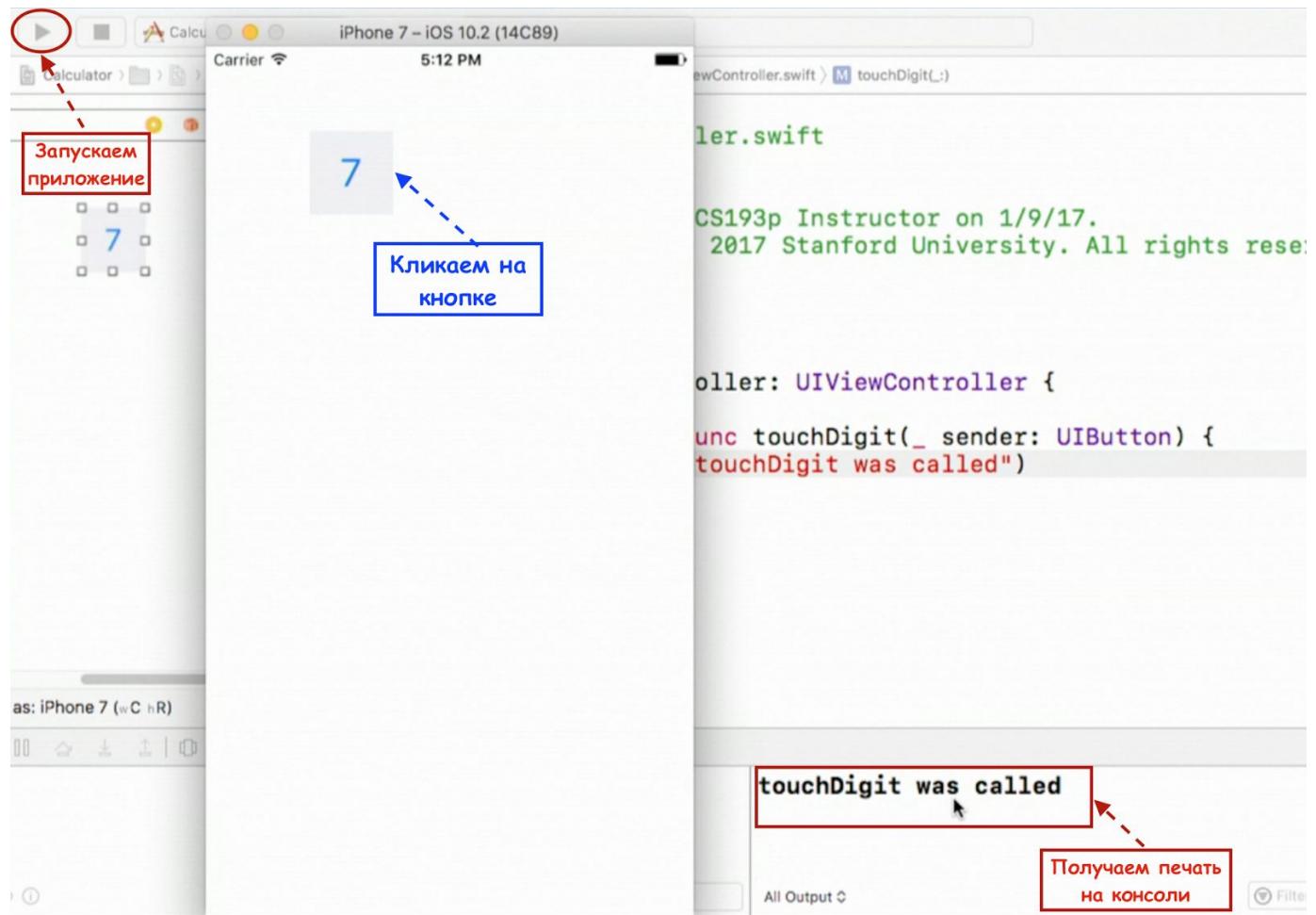
```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        print("touchDigit was called")
    }
}
```

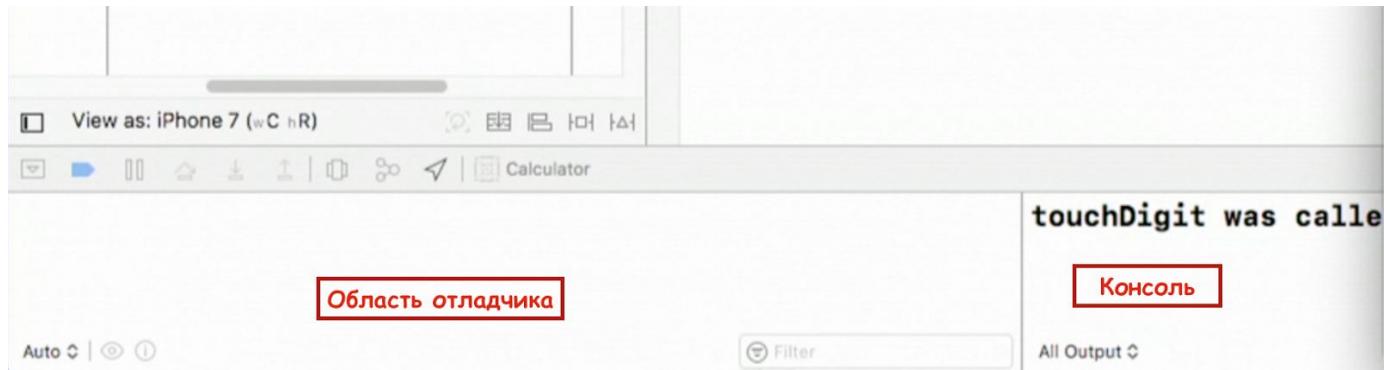
Как только мы нажимаем на кнопку, будет срабатывать этот метод и мы получим печать на консоли.

Давайте запустим приложение. Это наш UI на iPhone 7. Кликаем на кнопке, срабатывает метод **touchDigit** и текст появляется на консоли внизу:



Консоль делит пространство с отладчиком (debugger) в нижней части экрана **Xcode**: левая область

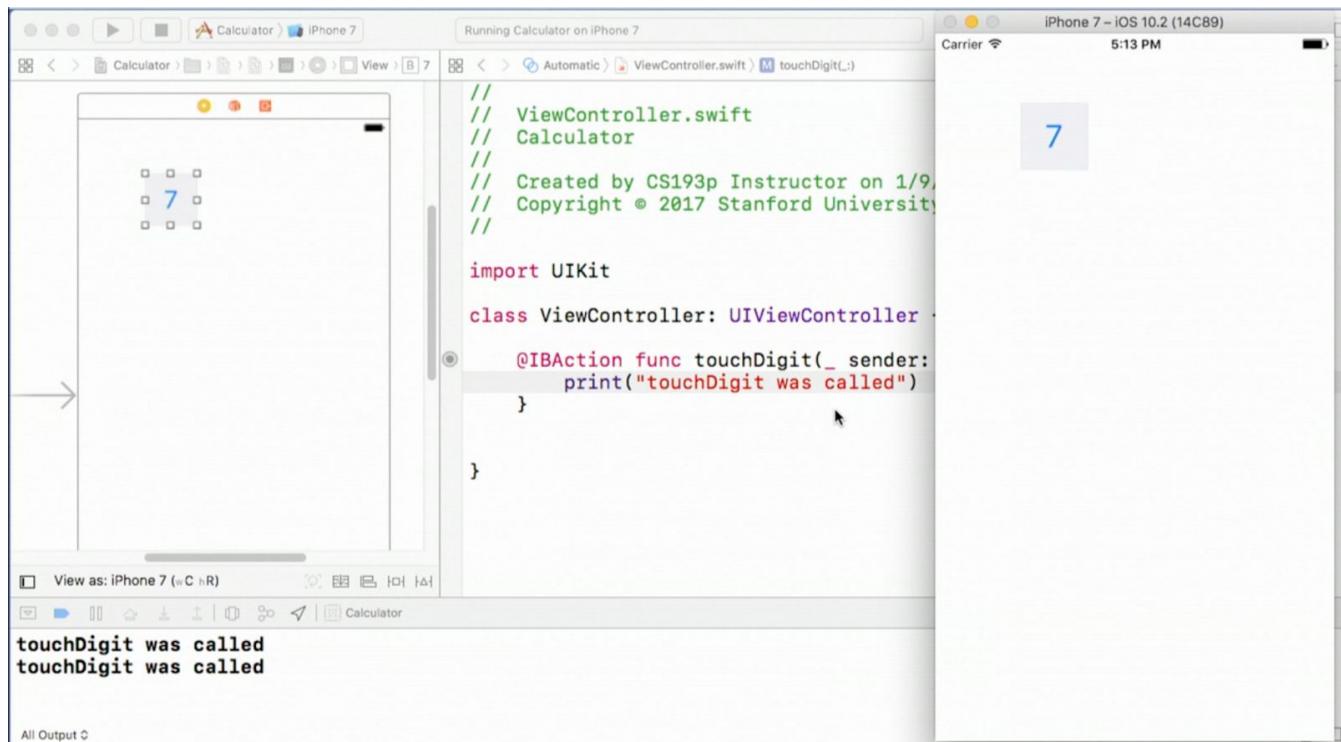
принадлежит отладчику, а правая - консоли.



Мы можем менять соотношение между этими областями, мы можем также полностью скрыть одну из областей с помощью маленьких кнопок, расположенных справа:



Мы оставим только консоль и вернемся к нашему симулятору. И каждый раз, когда мы нажимаем на кнопку "7", наш метод `touchDigit` распечатывает нам текст снова и снова.



Все понимают, что здесь происходит? Как мы "подцепили" метод `touchDigit` к кнопке? Реально очень просто.

Теперь у нас кнопка работает, но мы знаем, что нам нужно больше кнопок, нам нужна целая цифровая клавиатура. Я хочу, чтобы другие кнопки выглядели точно также, поэтому я буду

использовать “копирование и вставку”:

The screenshot shows the Xcode interface with two panes. On the left is the storyboard editor showing a 3x3 grid of buttons, each labeled with the number '7'. On the right is the code editor showing the ViewController.swift file. The code defines a class ViewController that implements the UITableViewDataSource protocol. It contains a method touchDigit(_:) which prints "touchDigit was called" to the console.

```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/
// Copyright © 2017 Stanford University
//

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        print("touchDigit was called")
    }
}
```

Я могу даже выбирать сразу 3 кнопки за раз

The screenshot shows the Xcode interface with two panes. On the left is the storyboard editor showing a 3x3 grid of buttons, with the top row of three buttons selected. On the right is the code editor showing the ViewController.swift file. The code defines a class ViewController that implements the UITableViewDataSource protocol. It contains a method touchDigit(_:) which prints "touchDigit was called" to the console.

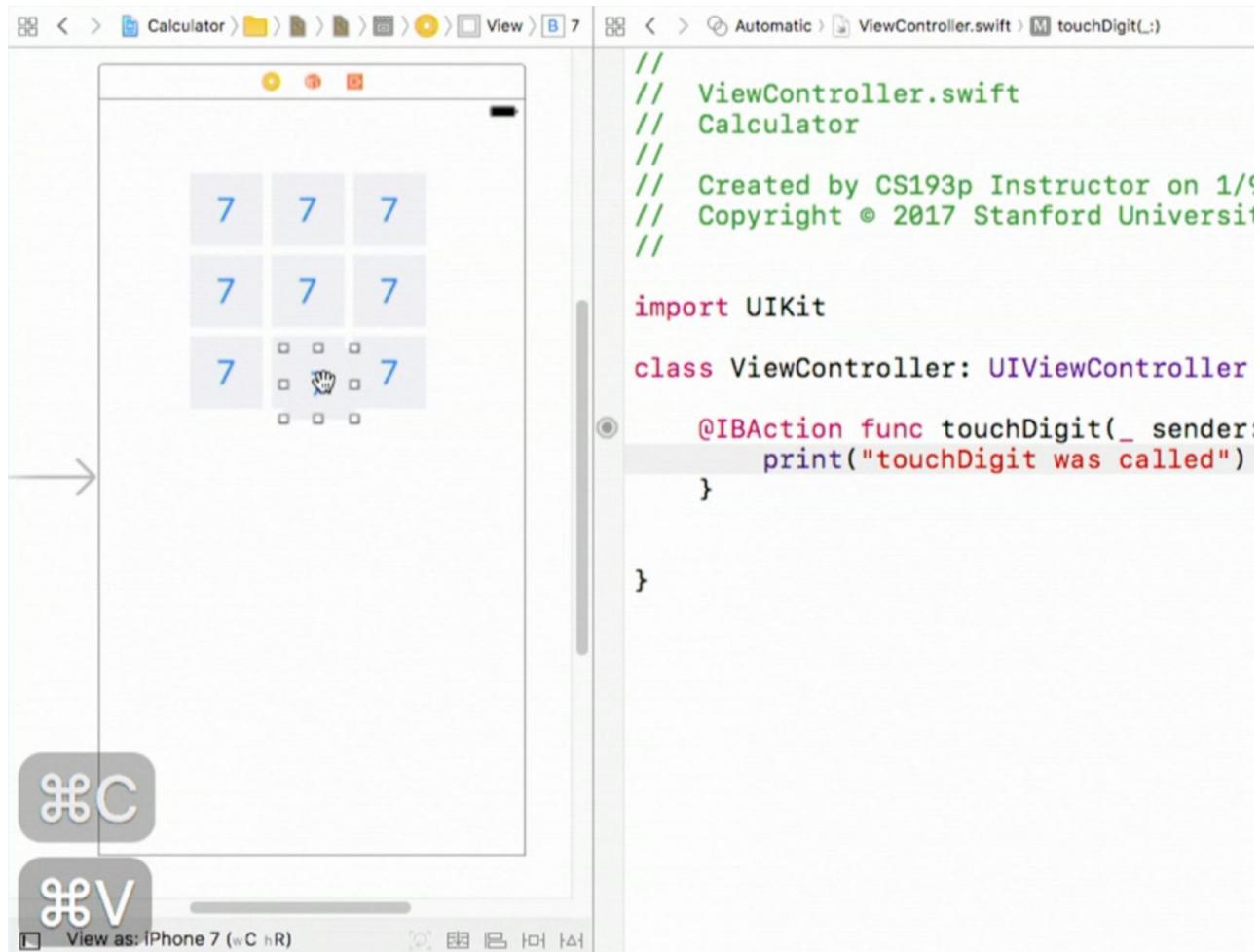
```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/
// Copyright © 2017 Stanford Universi
//

import UIKit

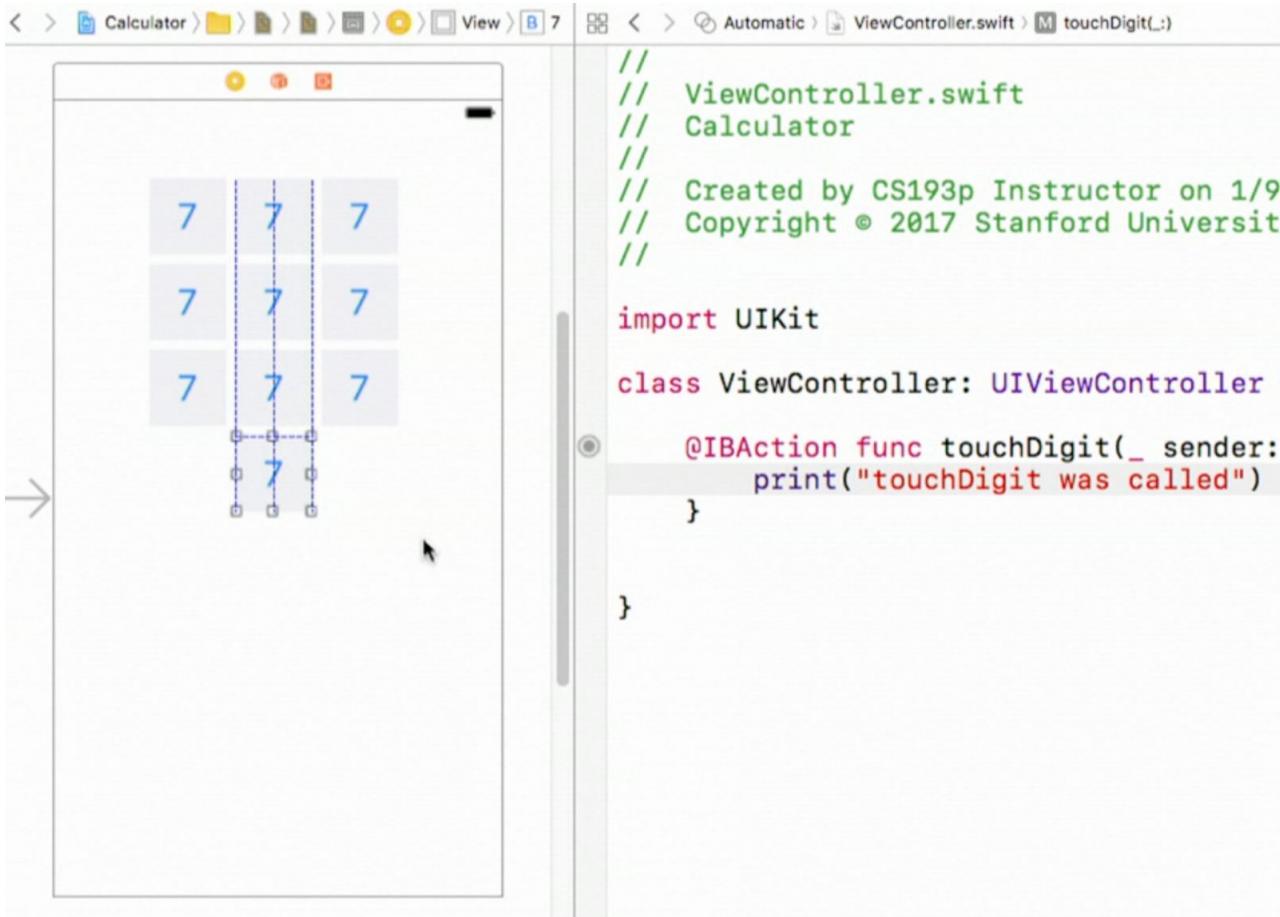
class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        print("touchDigit was called")
    }
}
```

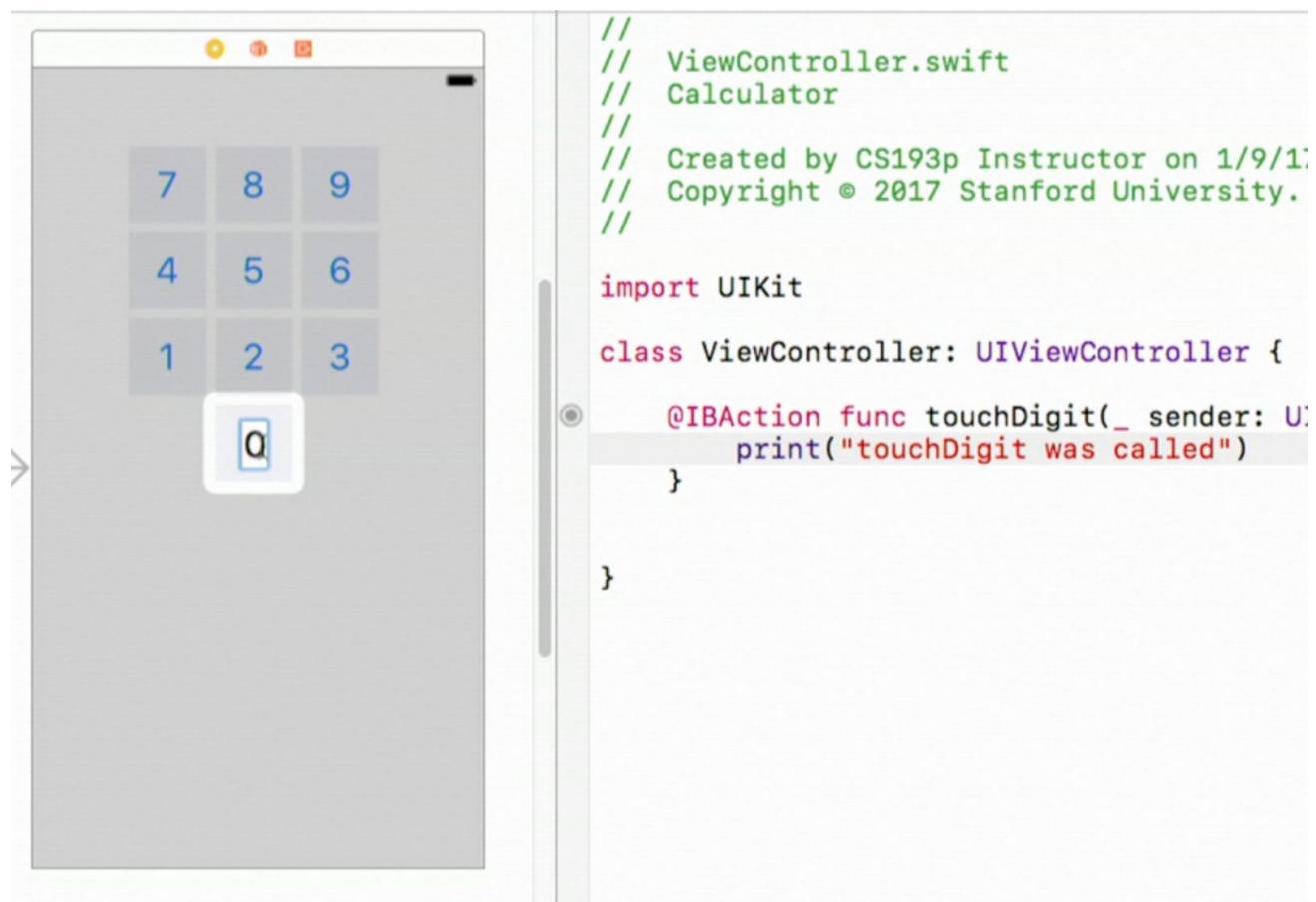
И ВСТАВИТЬ:



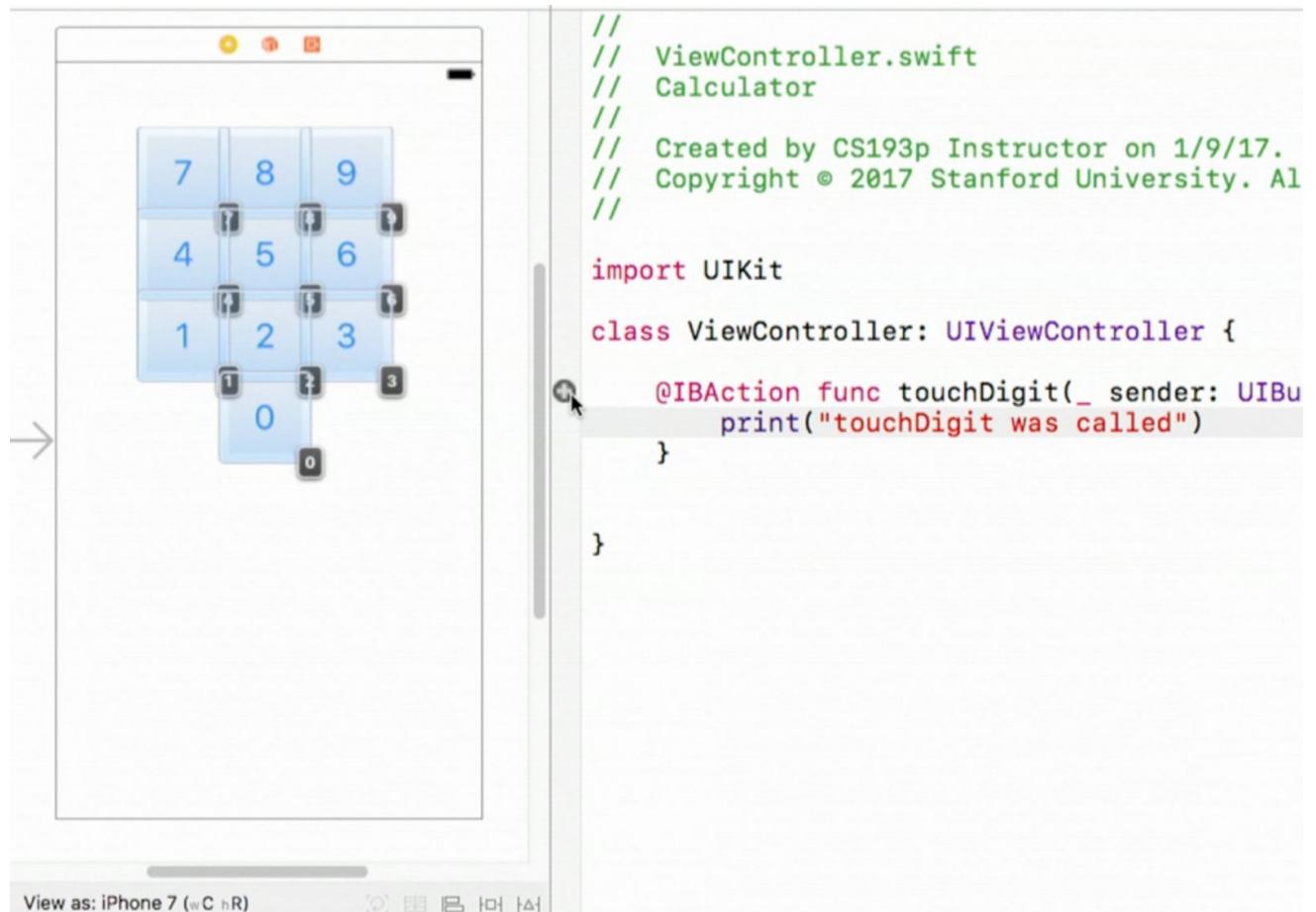
Заметьте, что я использую голубые пунктирные линии для их выравнивания. Голубые линии работают великолепно:



Теперь я просто переименую эти кнопки, дважды кликая на них:

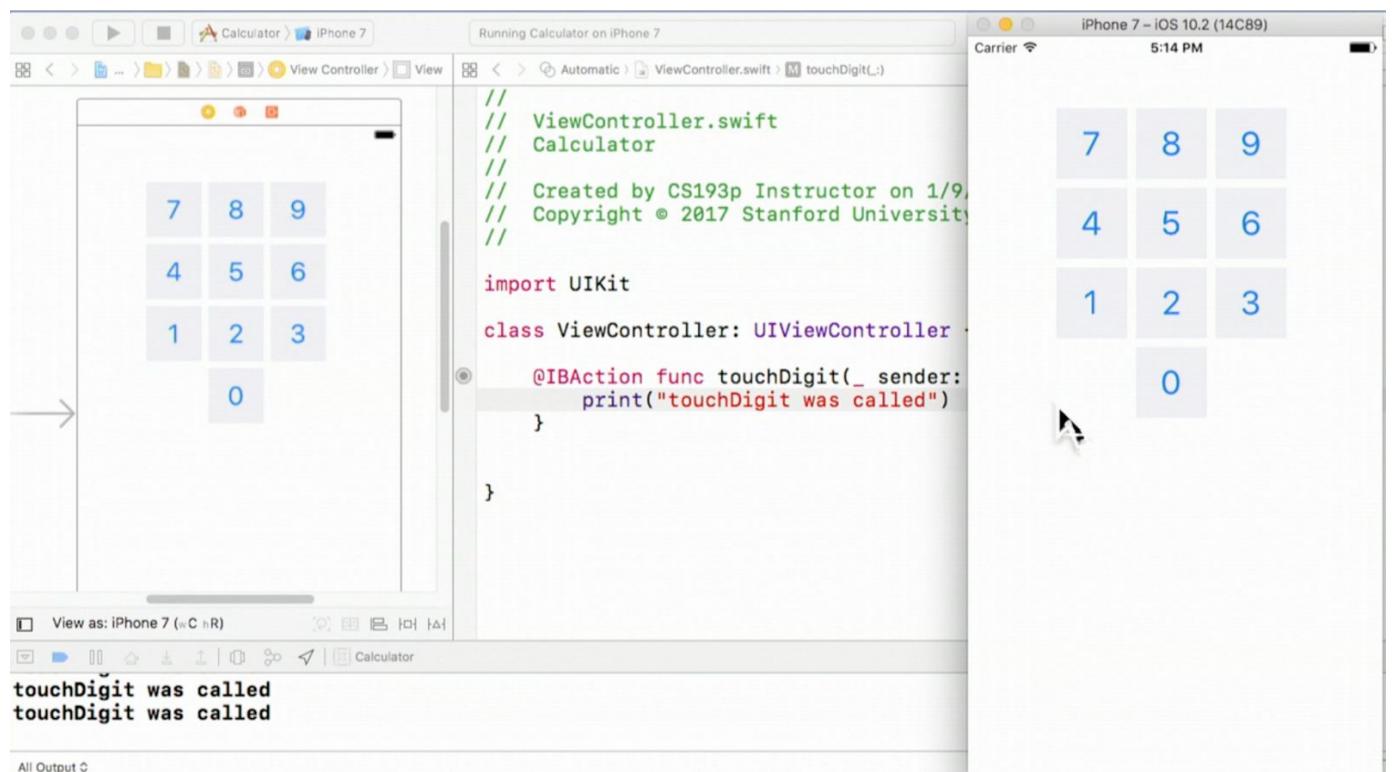


У меня получилась прекрасная клавиатура, и самое замечательное то, что метод `touchDigit` оказался “подцепленным” ко всем кнопкам.



Это произошло потому, что при копировании и вставке сохраняется связь с любым методом, у которого была связь с источником копирования. Это замечательно. Таким образом, все эти кнопки будут посыпать нам сообщение **touchDigit**.

Теперь. Если я запущу приложение и буду нажимать на любые кнопки - не только на **7**, но и на **5**, на **0** - то будет вызываться метод **touchDigit** и мы будем получать печать на консоли.



Это замечательно, но, конечно, мы хотим знать, какая кнопка послала нам это сообщение. И мы можем это узнать, потому что у нашего метода есть аргумент **sender**.

Давайте добавим локальную переменную:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit: String
        print("touchDigit was called")
    }

}
```

----- 40 -ая минута лекции -----

Вы впервые видите локальную переменную **var digit** в **Swift**. Ключевое слово **var** означает, что это локальная переменная. Ее имя - **digit**. Если вы хотите указать тип этой переменной, например, **String**, то вам нужно разместить после имени переменной тип : **String** точно также, как это сделано для аргумента **sender**.

Но в **Swift** мы обычно этого не делаем,

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit|
        print("touchDigit was called")
    }

}
```

потому что **Swift** - строго типизированный язык программирования, другими словами, вы везде должны определять тип. Он хочет знать типы всего, чего угодно. Он может “выводить” (**infer**) тип прямо из контекста и очень часто это делает. Это своего рода компромисс. Если у нас действительно сильно типизированный язык программирования, то вы вынуждены указывать тип абсолютно везде. Но совершенно прекрасно, когда компилятор может определить для вас какой тип имеет тот или иной элемент. Поэтому мы перестаем указывать тип, если это возможно.

При определении аргумента **sender** функции **touchDigit** мы не можем этого сделать, потому что мы должны знать, что ожидается на входе этого метода. Но для локальных переменных мы практически всегда можем не указывать тип.

Итак у нас есть локальная переменная **digit**, и я хочу установить ее равной чему-то. Чему она должна быть равна?

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit =
            print("touchDigit was called")
    }
}
```

Я хочу, чтобы она была равна заголовку (**title**) кнопки **sender**, которая послала мне сообщение **touchDigit**. Я хочу спросить у кнопки **sender**, какой у нее заголовок.

Как вы можете послать сообщение другому объекту в **Swift**?

Вы просто печатаете им объекта, которому хотите послать сообщение и добавляете точку “.” :

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.

        V Bool adjustsImageWhenDisabled
        V Bool adjustsImageWhenHighlighted
        M NSAttributedString? attributedTitle(for: UIControlState)
        M UIImage? backgroundImage(for: UIControlState)
        M CGRect backgroundRect(forBounds: CGRect)
        V UIButtonType buttonType
        V UIEdgeInsets contentEdgeInsets
        M CGRect contentRect(forBounds: CGRect)

    }
}
```

A Boolean value that determines whether the image changes when the button is disabled.

Это как в **Java** и во множестве других языков, просто объект и точка “.”, именно так вы посыпаете сообщение объекту. К сожалению, **Xcode** помогает мне и показывает мне все методы и свойства, которые кнопка (**button**) может выполнять и иметь. Посмотрите на этот огромный список, я нахожусь только на “F”, и могу прокручивать и прокручивать:

```

import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.
            UIView(forLastBaselineLayout)
            Any(forwardingTarget(for: Selector!))
            CGRect(frame)
            CGRect(frame(forAlignmentRect: CGRect))
            [UIGestureRecognizer... gestureRecognizers]
            Bool(gestureRecognizerShouldBegin(gestureR...
            Bool(hasAmbiguousLayout)
            Int(hash)
}

```

A Boolean value that determines whether the image changes when the button is disabled.

Здесь огромное множество методов. Как же всем этим можно управлять? Я хочу только получить заголовок (**title**) этой кнопки. Конечно, я мог бы взять документацию, начать читать ее, пытаясь найти то, что мне нужно и фактически я так и буду поступать. Но есть некоторая хитрость, которую нам предлагает **Xcode**. Я просто печатаю наименование того, что мне нужно, и смотрите, что будет происходить, если я просто напечатаю “**title**” - это не метод или свойство кнопки с именем **title**:

```

5   6 import UIKit
2   3 class ViewController: UIViewController {
0
    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.title
V     UILabel? titleLabel
M         String? title(for: UIControlState)
V         UIEdgeInsets titleEdgeInsets
M         UIColor? titleColor(for: UIControlState)
M         UIColor? titleShadowColor(for: UIControlState)
M         CGRect titleRect(forContentRect: CGRect)
V         String? currentTitle
V         UIColor currentTitleColor
}

```

A view that displays the value of the currentTitle property for a button.

Вы видите, что **Xcode** показал мне все методы и свойства, начинающиеся со слова “**title**” ИЛИ имеющие в имени слово “**title**” ИЛИ даже то, что имеет в имени “t”, “it”, “le”. **Xcode** делает все возможное, чтобы показать вам максимально подходящее тому, что вы напечатали.

Давайте посмотрим, может быть мы найдем то, что нам надо. Что нам даст заголовок кнопки? Как насчет **title (for: UIControlState)**? Выглядит подходяще и возвращает заголовок (**title**) для соответствующего состояния:

```
5   6
2   3
0
import UIKit

class ViewController: UIViewController {
    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.title
    }
}

V UILabel? titleLabel
M String? title(for: UIControlState)
V UIEdgeInsets titleEdgeInsets
M UIColor? titleColor(for: UIControlState)
M UIColor? titleShadowColor(for: UIControlState)
M CGRect titleRect(forContentRect: CGRect)
V String? currentTitle
V UIColor currentTitleColor
```

Возвращает заголовок, ассоциируемый с определенным состоянием

Returns the title associated with the specified state.

Однако я ничего не знаю относительно “состояния” **UIControlState**. Давайте продолжим поиск нужного метода. Может быть, **currentTitle**, который возвращает текущий заголовок кнопки, который в данный момент показан на экране:

```
5   6
2   3
0
import UIKit

class ViewController: UIViewController {
    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.title
    }
}

V UILabel? titleLabel
M String? title(for: UIControlState)
V UIEdgeInsets titleEdgeInsets
M UIColor? titleColor(for: UIControlState)
M UIColor? titleShadowColor(for: UIControlState)
M CGRect titleRect(forContentRect: CGRect)
V String? currentTitle
V UIColor currentTitleColor
```

Возвращает текущий заголовок, показываемый на кнопке

The current title that is displayed on the button.

Звучит в точности так, как мне нужно. Ура! Победа! Я выбираю этот метод двумя кликами.

```

import UIKit

class ViewController: UIViewController {

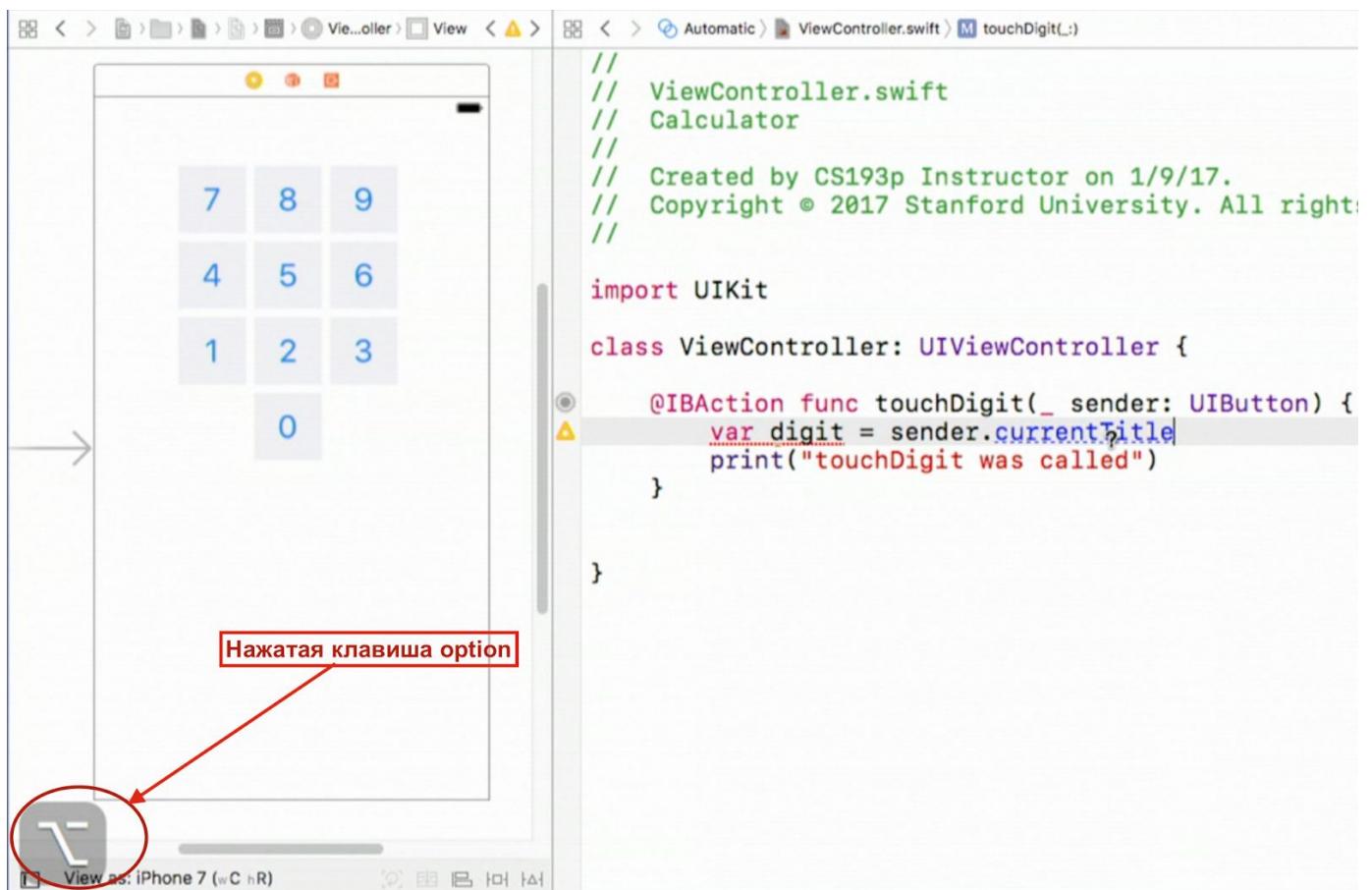
    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        print("touchDigit was called")
    }

}

```

Итак, мы посылаем сообщение `currentTitle` кнопке `sender`. Давайте узнаем побольше о сообщении `currentTitle`. Мы знаем, что это односторончный текст, показываемый в данный момент на кнопке.

Для того, чтобы узнать больше информации о `currentTitle`, нажимаем клавишу `option`. Видите на видео в левом нижнем углу показывается, какую клавишу я нажимаю? Когда я нажимаю эту клавишу и подвожу “мышку” к интересующему нас объекту, то появляется голубая пунктирная линия, подчеркивающая текст, со знаком “?” вопроса:



Если я кликну на слове с голубой пунктирной линией, то появится небольшое окошко с информацией:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        print("touchDigit was called!")
    }
}
```

Declaration var currentTitle: String? { get }

Description The current title that is displayed on the button. The value for this property is set automatically whenever the button state changes. For states that do not have a custom title string associated with them, this method returns the title that is currently displayed, which is typically the one associated with the normal state. The value may be nil.

Availability iOS (8.0 and later), tvOS (9.0 and later)

Declared In UIKit

More [Property Reference](#)

В этом окошке у меня есть детальное описание **currentTitle**, я узнаю, как декларируется **currentTitle**. Оказывается, **currentTitle** - это не функция **func**, а свойство **var**. И здесь мы впервые видим, как декларируется свойство (property) в **Swift**. Это переменная экземпляра класса (**instance variable**) класса **UIButton**.

Понятно, что это переменная, следовательно **var**, имя переменной - **currentTitle**. Тип переменной - **String?**? Может быть, кнопка вообще не уверена, что за тип у нее имеет заголовок? Я думаю, что кнопка-то знает тип своего текущего заголовка, а мы в настоящий момент можем предположить, что это строка **String**, а в дальнейшем увидим, если что-то не так. Далее следует маленький синтаксис **{ get }**, означающий что мы можем только “читать” текущий заголовок кнопки с помощью свойства **currentTitle**. Мы не можем устанавливать (“писать”) заголовок **currentTitle**.

Но я бы все-таки хотел знать, а как установить заголовок кнопки? Я должен посмотреть документацию. Как мы можем получить ее прямо сейчас? Для этого существует раздел “**More**” в этом маленьком справочном окошке:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        print("touchDigit was called!")
    }
}
```

Declaration var currentTitle: String? { get }

Description The current title that is displayed on the button. The value for this property is set automatically whenever the button state changes. For states that do not have a custom title string associated with them, this method returns the title that is currently displayed, which is typically the one associated with the normal state. The value may be nil.

Availability iOS (8.0 and later), tvOS (9.0 and later)

Declared In UIKit

More [Property Reference](#)

Если я кликну на ссылке в этом разделе, то получу документацию для **currentTitle**:

The screenshot shows the UIKit documentation for the `UIButton.currentTitle` property. On the left, there's a sidebar with links for Swift, Objective-C, and JavaScript. The main content area has a breadcrumb navigation bar: `UIKit` > `UIButton` > `currentTitle`. A red box highlights the `UIButton` link, with a red arrow pointing to a callout box containing the text: "Можем кликнуть на классе `UIButton` и получить документацию по этому классу". Below the breadcrumb, it says "Instance Property". The title `currentTitle` is shown with a bookmark icon. The description states: "The current title that is displayed on the button." To the right, under "Language", it lists Swift and Objective-C. Under "SDKs", it lists iOS 8.0+ and tvOS 9.0+. On the far right, there's a "On This Page" section with links for Declaration, Discussion, and See Also.

Замечательно в этой документации то, что я могу кликнуть на любом объекте : **String** или **UIButton** и получить документацию по соответствующему объекту.

Кликаем на **UIButton** и получаем документацию по классу **UIButton**.

The screenshot shows the UIKit documentation for the `UIButton` class. The sidebar on the left includes links for Swift, Objective-C, and JavaScript. The breadcrumb navigation bar shows `UIKit` > `UIButton`. Below the breadcrumb, it says "Class". The title `UIButton` is shown with a bookmark icon. The description states: "A UIButton object is a view that executes your custom code in response to user interactions." To the right, under "Language", it lists Swift and Objective-C. Under "SDKs", it lists iOS 2.0+ and tvOS 2.0+. On the far right, there's a "On This Page" section with links for Overview, Symbols, and Relationships. An oval highlights the "Overview" link, with a red arrow pointing to a callout box containing the text: "Настоятельно рекомендую ознакомиться с этим разделом". Below the "Overview" link, there's a section about button interactions and another callout box with the same text: "Настоятельно рекомендую ознакомиться с этим разделом". At the bottom, there's a "Figure 1" section with a "Button" button and a plus sign, followed by a callout box with the same text: "Настоятельно рекомендую ознакомиться с этим разделом". The final callout box at the bottom right also contains the same text: "Настоятельно рекомендую ознакомиться с этим разделом".

Здесь есть раздел Overview (Обзор). Он совершенно замечательный и я настоятельно рекомендую ознакомится с этим разделом для каждого класса, который вы будете использовать - потратьте 5-10 минут для прочтения этих разделов. Тогда вы точно будете знать, как эти классы работают.

Посмотрите, например, что у нас есть для класса [UIButton](#).

Здесь описывается как сконфигурировать внешний вид кнопки (**appearance**).

Configuring a Button's Appearance

A button's type defines its basic appearance and behavior. You specify the type of a button at creation time using the `init(type:)` method or in your storyboard file. After creating a button, you cannot change its type. The most commonly used button types are the Custom and System types, but use the other types when appropriate.

Note

To configure the appearance of all buttons in your app, use the appearance proxy object. The `UIButton` class implements the `appearance()` class method, which you can use to fetch the appearance proxy for all buttons in your app.

Button States

Buttons have five states that define their appearance: default, highlighted, focused, selected, and disabled. When you add a button to your interface, it is in the default state initially, which means the button is enabled and the user is not interacting with it. As the user interacts with the button, its state changes to the other values. For example, when the user taps a button with a title, the button moves to the highlighted state.

When configuring a button either programmatically or in Interface Builder, you specify attributes for each state separately. In Interface Builder, use the State Config control in the Attributes inspector to choose the appropriate state and then configure the other attributes. If you do not specify attributes for a particular state, the `UIButton` class provides a reasonable default behavior. For example, a disabled button is normally dimmed and does not display a highlight when tapped. Other properties of this class, such as the `adjustsImageWhenHighlighted` and `adjustsImageWhenDisabled` properties, let you alter the default behavior in specific cases.

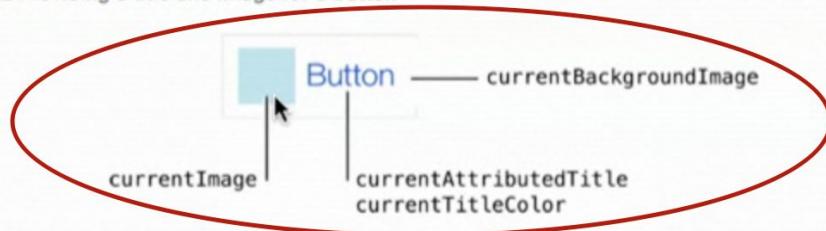
Этот раздел рассказывает о состояниях **states** кнопки. Помните? Мы рассматривали метод `title(for: UIControlState)`? И мы не знали, что означает состояние **state**. Здесь нам рассказывают об этом.

Нам рассказывают о том, из каких частей состоит кнопка. Мы можем поместить туда изображение (**image**) и, очевидно, текст (**text**). Также объясняется, что такие отступы (**edge insets**).

----- 45 -ая минута лекции -----

results in the appearance shown in [Figure 2](#). You can access the current content of a button using the indicated properties.

Figure 2 Providing a title and image for a button



When setting the content of a button, you must specify the title, image, and appearance attributes for each state separately. If you do not customize the content for a particular state, the button uses the values associated with the Default state and adds any appropriate customizations. For example, in the highlighted state, an image-based button draws a highlight on top of the default image if no custom image is provided.

Tint Color

You can specify a custom button tint using the `tintColor` property. This property sets the color of the button image and text. If you do not explicitly set a tint color, the button uses its superview's tint color.

Edge Insets

Use insets to add or remove space around the content in your custom or system buttons. You can specify separate insets for your button's title (`titleEdgeInsets`), image (`imageEdgeInsets`), and both the title and image together (`contentEdgeInsets`). When applied, insets affect the corresponding content rectangle of the button, which is used by the Auto Layout engine to determine the button's position.

Даже есть сведения обо всем, что есть в различных инспекторах **Interface Builder**:

Interface Builder Attributes

[Table 1](#) lists the core attributes that you configure for buttons in Interface Builder.

Table 1 Core attributes

Attribute	Description
Type	The button type. This attribute determines the default settings for many other button attributes. The value of this attribute cannot be changed at runtime, but you can access it using the <code>buttonType</code> property.
State Config	The state selector. After selecting a value in this control, changes to the button's attributes apply to the specified state.
Title	The button's title. You can specify a button's title as a plain string or attributed string.
(Title Font and Attributes)	The font and other attributes to apply to the button's title string. The specific configuration options depends on whether you specified a plain string or attributed string for the button's title. For a plain string, you can customize the font, text color, and shadow color. For an attributed string, you can specify alignment, text direction, indentation, hyphenation, and many other options.

Посмотрите, все детально разъясняется. Совершенно потрясающая вещь, прочтя которую, вы будете знать все эти классы. И, конечно, есть список методов. Например, тот, который мы искали - **setTitle**:

Symbols

Creating Buttons

```
init(type: UIButtonType)  
Creates and returns a new button of the specified type.
```

Configuring the Button Title

```
var titleLabel: UILabel?  
A view that displays the value of the currentTitle property for a button.
```

```
func title(for: UIControlState)  
Returns the title associated with the specified state.
```

```
func setTitle(String?, for: UIControlState)  
Sets the title to use for the specified state.
```

```
func attributedTitle(for: UIControlState)  
Returns the styled title associated with the specified state.
```

```
func setAttributedTitle(NSAttributedString?, for: UIControlState)  
Sets the styled title to use for the specified state.
```

```
func titleColor(for: UIControlState)
```

Если я кликну на этом методе, то получу его полное описание.

UIKit > UIButton > setTitle(_:for:)

Instance Method

setTitle(_:for:)

Sets the title to use for the specified state.

Language
Swift | Objective-C

SDKs

iOS 8.0+
tvOS 9.0+

On This Page

[Declaration](#)

[Parameters](#)

[Discussion](#)

[See Also](#)

Declaration

```
func setTitle(_ title: String?, for state: UIControlState)
```

Parameters

title	The title to use for the specified state.
--------------	---

state	The state that uses the specified title. The possible values are described in UIControlState .
--------------	--

Можно кликнуть на UIControlState и получить его описание

И даже можно посмотреть, что такое состояние **UIControlState** и кликнуть на нем для получения

информации о том, что это такое.

Constants

```
static var normal: UIControlState
The normal, or default state of a control—that is, enabled but neither selected nor highlighted.

static var highlighted: UIControlState
Highlighted state of a control. A control becomes highlighted when a touch event enters the control's bounds, and it loses that highlight when there is a touch-up event or when the touch event exits the control's bounds. You can retrieve and set this value through the isHighlighted property.

static var disabled: UIControlState
Disabled state of a control. User interactions with disabled control have no effect and the control draws itself with a dimmed appearance to reflect that it is disabled. You can retrieve and set this value through the isEnabled property.

static var selected: UIControlState
Selected state of a control. For many controls, this state has no effect on behavior or appearance. Some subclasses, like the UISegmentedControl class, use this state to change their appearance. You can retrieve and set this value through the isSelected property.

static var focused: UIControlState
Focused state of a control. In focus-based navigation systems, a control enters this state when it receives the focus. A focused control changes its appearance to indicate that it has focus, and this appearance differs from the appearance of the control when it is highlighted or selected. Further interactions with the control can result in it also becoming highlighted or selected.

static var application: UIControlState
Additional control-state flags available for application use.

static var reserved: UIControlState
```

Ну, конечно, это состояния **normal** (нормальное), **highlighted** (подсвеченная кнопка), **disabled** (неработоспособная кнопка), **selected** (выбранная кнопка) и т.д.. И в каждом из этих состояний у вас может быть свой собственный заголовок. Здорово!

Вы видите, как легко и быстро я перемещаюсь между различными типами и классами простым кликом по соответствующим ссылкам и это очень важно для того, чтобы быть профессиональным разработчиком iOS. Очень важно эффективно пользоваться документацией.

Возвращаемся к **currentTitle**, он имеет тип **String?** :



И я говорил вам, что **Swift** “выводит тип” (**infer**) из контекста. Давайте убедимся в этом и посмотрим, какой тип имеет локальная переменная **digit**.

```
@IBAction func touchDigit(_ sender: UIButton) {
    var digit = sender.currentTitle
    print('touchDigit was called')
}
```

Declaration var digit: String?
Declared In ViewController.swift

Она также имеет тип **String?**. Конечно, потому что я установил знак равенства тому, что имеет тип **String?**, и компилятор знает это.

Давайте пойдем дальше и распечатаем **digit**.

Для печати в некоторых языках вы можете использовать следующий синтаксис:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        printf("%s was touched", digit)
    }
}
```

В других языках можно использовать такой синтаксис для печати

К сожалению, вы не можете этого делать в **Swift**, там нет **%** формата и нет метода **printf**.

Вместо этого мы используем **print**, а вместо **%** мы используем “магические” круглые скобки с обратным слэшем **\()**. Внутри круглых скобок вы можете разместить строку и все, что может быть преобразовано в строку. Тогда это будет включено в распечатываемый текст. Очевидно, сама строка может быть преобразована в строку:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        print("\(digit) was touched")
    }
}
```

Синтаксис Swift

Этим способом можно вставлять строки в другие строки. Или вставлять более сложные объекты, которые знают как превратить себя в строку, в другие строки.

Заметим, что у нас есть предупреждение в виде желтого маленького треугольника.

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        var digit = sender.currentTitle
        print("\(digit) was touched")
    }
}
```

Если он желтого цвета, то это предупреждение и в этом случае ваше приложение будет

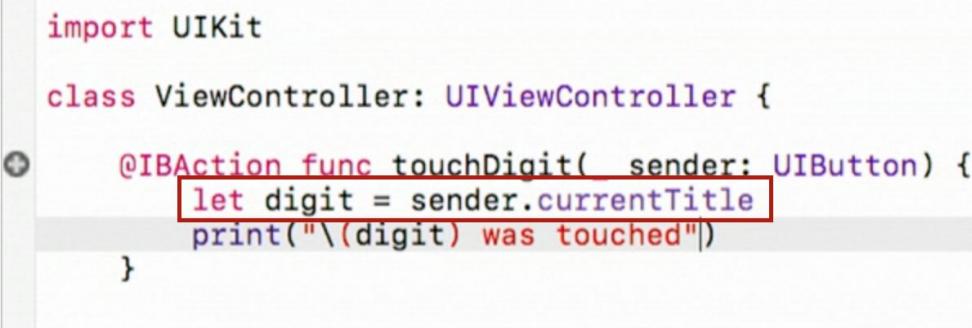
компилироваться и запускаться. Но если вы увидите предупреждение в своей Домашней Работе, то вы должны вернуться к этому коду и сделать исправления, иначе в этом курсе вы не сможете послать свое Домашнее Задание на проверку (submit). Никаких предупреждений не допускается. Вы меня слышите?

Могут быть красные пометки, в этом случае ваше приложение даже не будет компилироваться. Но у нас с вами желтый треугольник. Как мы можем определить, что это за предупреждение? Мы просто кликаем на желтом треугольнике.



И нам говорят, что переменная **digit** никогда не изменяется (was never mutated) и советуют заменить **var** на **let**, то есть сделать переменную константой.

Более того, нам предлагают сделать эту замену автоматически. Я кликаю на выделенной в подсказке строке и замена **var** на **let** происходит автоматически:



Предупреждение исчезло. Оно говорило о том, что **digit** никогда не изменяется, теперь **digit** действительно стала константой. Мы дали этой константе начальное значение и больше она не изменяется.

Когда вы декларируете константу, вам всегда нужно использовать **let**.

Почему нам нужно использовать другое ключевое слово для константы, отличное от **var**? Потому что константа - это не переменная, она не изменяется, это действительно константа. Ключевое слово **let** - прекрасное слово для читаемости кода: пусть **digit** равняется текущему заголовку **sender**. Прекрасно читается. Почему мы так беспокоимся о том, чтобы различать константы и переменные? Этому две причины. Одна заключается в том, что, если вы читаете чей-то код и видит ключевое слово **let**, то сразу понимаете, что это константа и нигде далее в коде она не будет изменена. И если кто-то попытается изменить ее, то компилятор генерирует ошибку. Таким образом, вы не сможете изменить что-то, что является константой.

Но другая более важная причина состоит в том, что вы сообщаете **Swift**, что это - константа, что вы намереваетесь ее использовать как константу. И позже, если вы попытаетесь модифицировать ее, даже если это массив **Array** или словарь **Dictionary**, то есть пытаетесь что-то разместить в массиве **Array** или добавить что-то в словарь **Dictionary**, то **Swift** будет знать о ваших намерениях

использовать их как константы и выдаст ошибку.

Это способ сказать **Swift** о ваших намерениях. Разница между неизменяемым (**immutable**)

массивом **Array** и массивом **Array**, в который вы можете добавлять элементы, - огромна!

Потому что массивы **Array** и словари **Dictionary** передаются в **Swift** путем копирования. Это очень необычный способ передачи по сравнению с другими языками программирования, в которых объекты “сидят” в “куче” (**heap**) и мы передаем только указатель (**pointer**) на них.

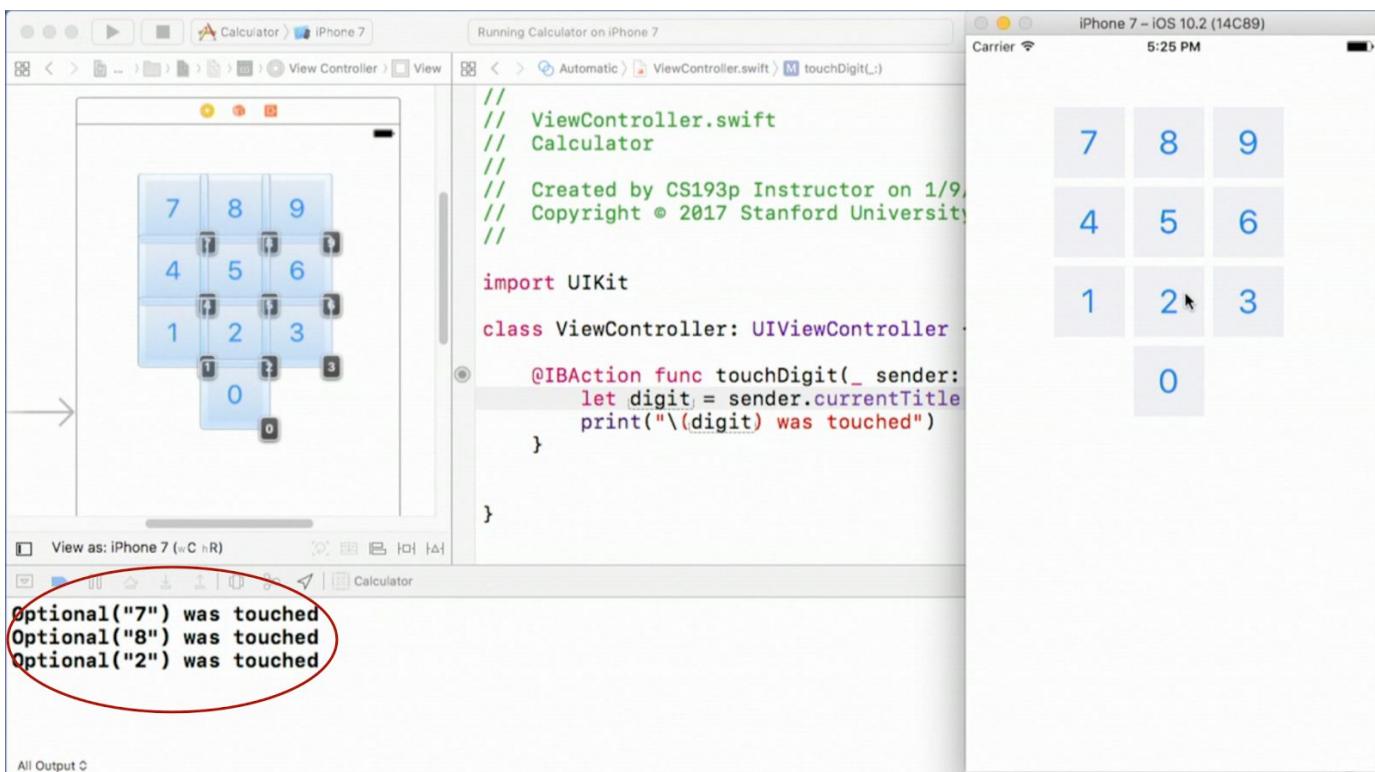
В **Swift** каждый раз при передачи **Array** и **Dictionary** некоторой функции, мы их копируем, это очень неэффективно, если они все - изменяемые. Потому что мы действительно должны их физически копировать, если кто-то их изменил. Но **Swift** знает, какие из них изменяемые (**mutable**), а какие - нет. Когда вы передаете неизменяемые (**immutable**) **Array** и **Dictionary**, то не происходит физического неэффективного копирования. Фактически, до тех пор, пока вы не присвоите **Array** и **Dictionary** изменяемой (**mutable**) переменной явно записав **var ... = Array / Dictionary**, у **Swift** нет причин беспокоиться на этот счет. Он выполняет “копирование при записи” (**copy on write**) только, когда происходят реальные изменения **Array** и **Dictionary**.

Итак, это что касается **var / let**, привыкайте к тому, чтобы всегда использовать **let** для констант.

----- 50 -ая минута лекции -----

Давайте запустим наше приложение и посмотрим, что будет происходить. Это должно работать.

Мы получаем текущий заголовок кнопки, на которую нажимаем. Нажимаем “7”, “8”, “2”:



Похоже, что работает и каждая кнопка знает свой заголовок. Но что такое **Optional (“7”)**? Это о чем?

Это потому что **digit** - это не **String**, а **String?**.

```
@IBAction func touchDigit(_ sender: UIButton) {
    var digit = sender.currentTitle
    print("\(digit) was called")
}

Declaration var digit: String?
Declared In ViewController.swift
```

Отличается ли тип **String?** от обычного **String**, вот в чем вопрос. Тип **String?** называется **Optional** типом. Это супер ВАЖНО!

Опять, те, у кого сейчас дневной сон, проснитесь! Это действительно супер ВАЖНО!

Очень небольшое количество языков программирования имеет эту концепцию. Это действительно крутая концепция, которую обязательно нужно понять, ибо она встречается повсюду во всех **iOS APIs**. Но требуется некоторое время, чтобы к ней привыкнуть.

Что собой представляет **Optional** ?

Optional — это просто тип. Этот тип может иметь только два значения: **set** (“установлен”) и **not set** (“не установлен”). Только два значения и больше никаких других. Тем не менее, если **Optional** находится в состоянии **set** (“установлен”), то оно может иметь ассоциированное значение (**associated value**). Это значение, которое держится в стороне, но при создании **Optional** и при декларировании **Optional** вы должны определить тип этого ассоциированного значения. В нашем случае ассоциированным значением является строка, ее тип **String**. Мы говорим о текущем заголовке кнопки. Поэтому, типом **digit** будет **Optional** строки или **String?**. Это означает **Optional**, у которого ассоциированное значение в состоянии **set** (“установлен”) имеет тип **String**.

Все это хорошо, но нам нужно как-то получить это ассоциированное значение (**associated value**).

Дайте мне ассоциированное значение, которое и является заголовком нашей кнопки.

Как я могу “вытянуть” его из **Optional** ?

Ответ заключается в том, что мы должны использовать восклицательный знак “!”:

```
import UIKit

class ViewController: UIViewController {

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        print("\(digit) was touched")
    }

}
```

Если мы разместим восклицательный знак “!” в конце **Optional**, то это даст нам ассоциированное значение при условии, что само **Optional**, находится в состоянии **set** (“установлен”).

Давайте теперь посмотрим на тип **digit**.

```
class ViewController: UIViewController {  
    @IBAction func touchDigit(_ sender: UIButton) {  
        let digit = sender.currentTitle!  
        print("(\(digit)) was touched")  
    }  
}
```

Declaration let digit: String
Declared In ViewController.swift

Это **String**. **Swift** умеет “выводить тип” из контекста, так что “развернув” (**unwrapped**) **Optional** в правой части, мы получаем **String** в левой части выражения.

Что будет, если мы применим восклицательный знак **!** к **Optional**, которое находится в состоянии **not set** (“не установлен”)? В этом состоянии у **Optional** нет ассоциированного значения. Что произойдет с моим приложением? Кто-нибудь попытается угадать?

Правильно, ваше приложение завершится аварийно. Я уверен, что некоторые из вас, кто более консервативен, подумают: “Восклицательный знак **!** ? Это - не для меня, я никогда его не буду использовать. Я не хочу, чтобы мое приложение завершалось аварийно. Это ужасно!”

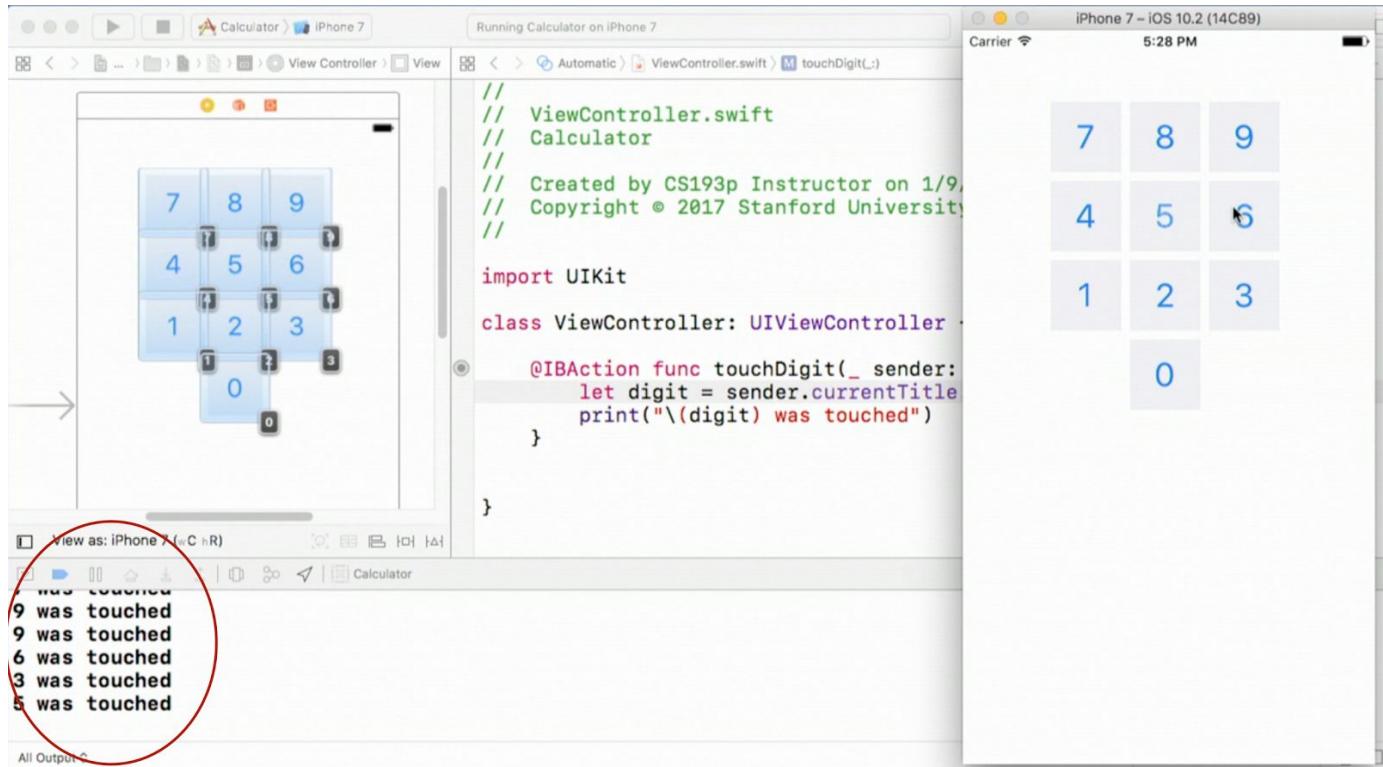
Но в действительности, аварийное завершение приложения на этапе разработки - это благо, потому что помогает быстро найти ошибки, вы оказываетесь прямо в отладчике и можете посмотреть, а что же произошло?

В нашем случае, когда мы говорим о заголовке (**title**), это означает, что заголовок на кнопке никогда не устанавливался, а метод **touchDigit** вызывается, что никогда не должно происходить.

Если это произойдет в приложении, которое я поставил своим пользователям, то они начнут жаловаться : “Это кнопка, на которой нет заголовка, я не знаю на что я нажимаю, когда использую такую кнопку.” Вы должны “отловить” такую ситуацию во время разработки приложения. Так что иногда аварийное завершение приложения - это хорошо. Но, конечно, я не хочу, чтобы каждый раз при “развертывании” **Optional** мое приложение аварийно завершалось, и я покажу вам небольшой прием для “развертывания” **Optional** и получения ассоциативного значения, который сначала тестирует **Optional** с тем, чтобы убедиться, что оно находится в состоянии **set** (“установлен”).

Но пока мы будем использовать восклицательный знак **!** .

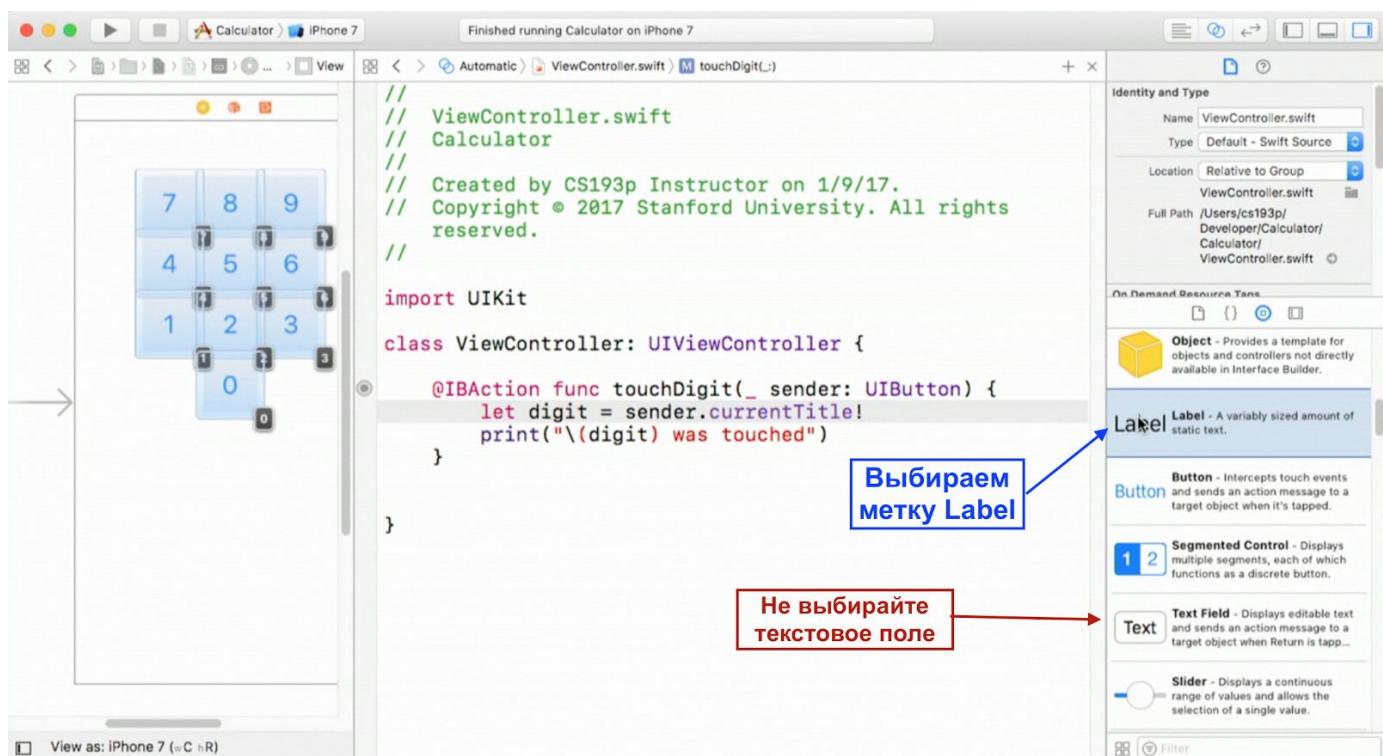
Давайте запустим приложение и посмотрим, на что это похоже. Оно должно работать. Но сейчас мы “развернули” **Optional** и получили ассоциативное значение. Мы знаем, что текущий заголовок **currentTitle** должен быть установлен для каждой кнопки и мы не будем беспокоиться об аварийном завершении приложения. Нажимаем “7”, “9”, “3”, “5”, “6”.



Теперь мы можем коллекционировать цифры, полученные от пользователя.

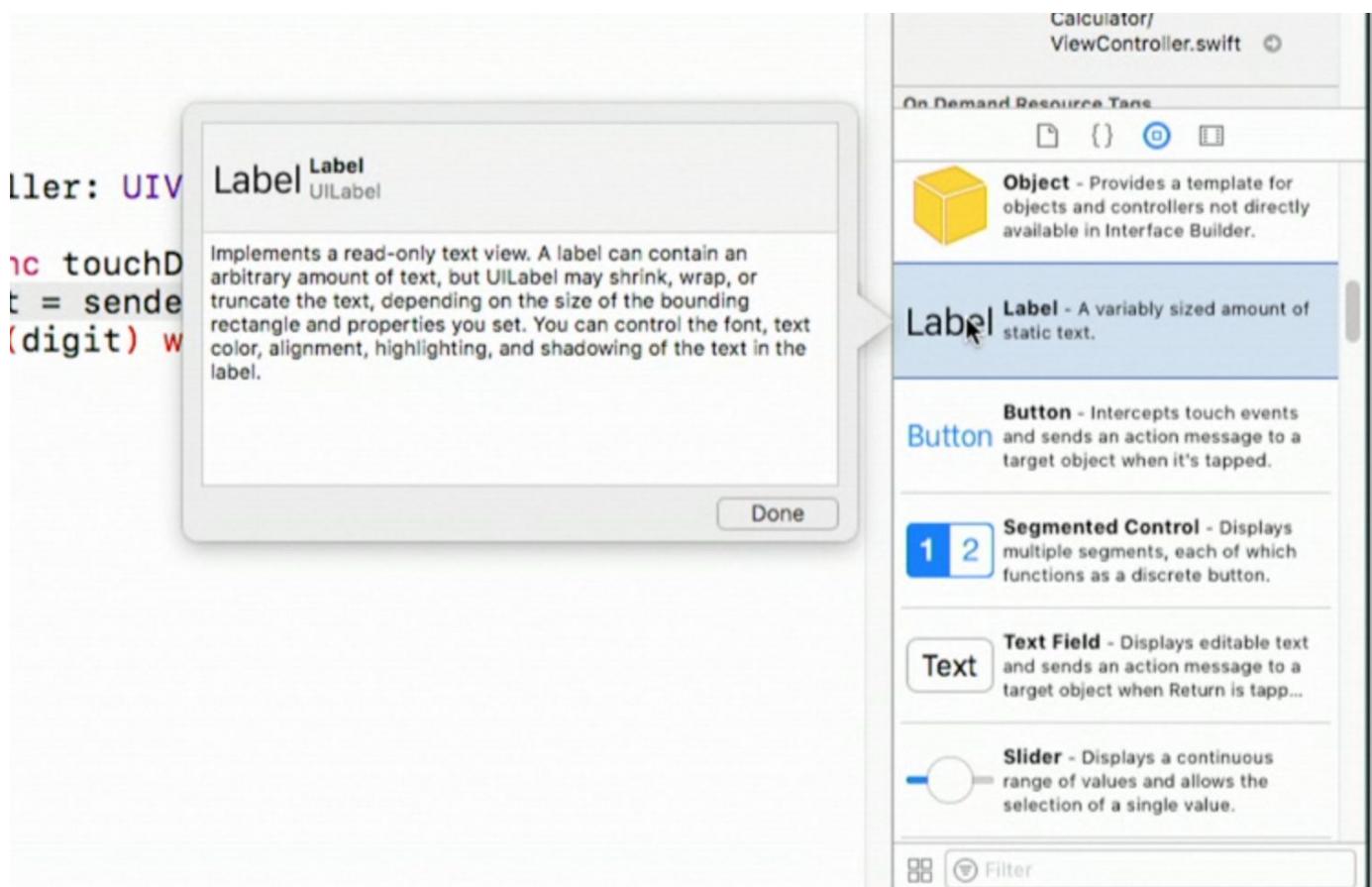
Давайте разместим их на дисплее. Нам нужно добавить дисплей в наш Калькулятор.

Мы возвращаем Область Утилит на экран и смотрим внимательно на Палитру Объектов с тем, чтобы выбрать что-то для дисплея. Будьте внимательны и не захватите текстовое поле **Text Field**, потому что это редактируемый текст, а в Калькуляторе мы не можем кликать на дисплее и редактировать его. Мы печатаем цифры на только что созданной цифровой клавиатуре с тем, чтобы их показывать на дисплее. Мы выберем метку **Label**.

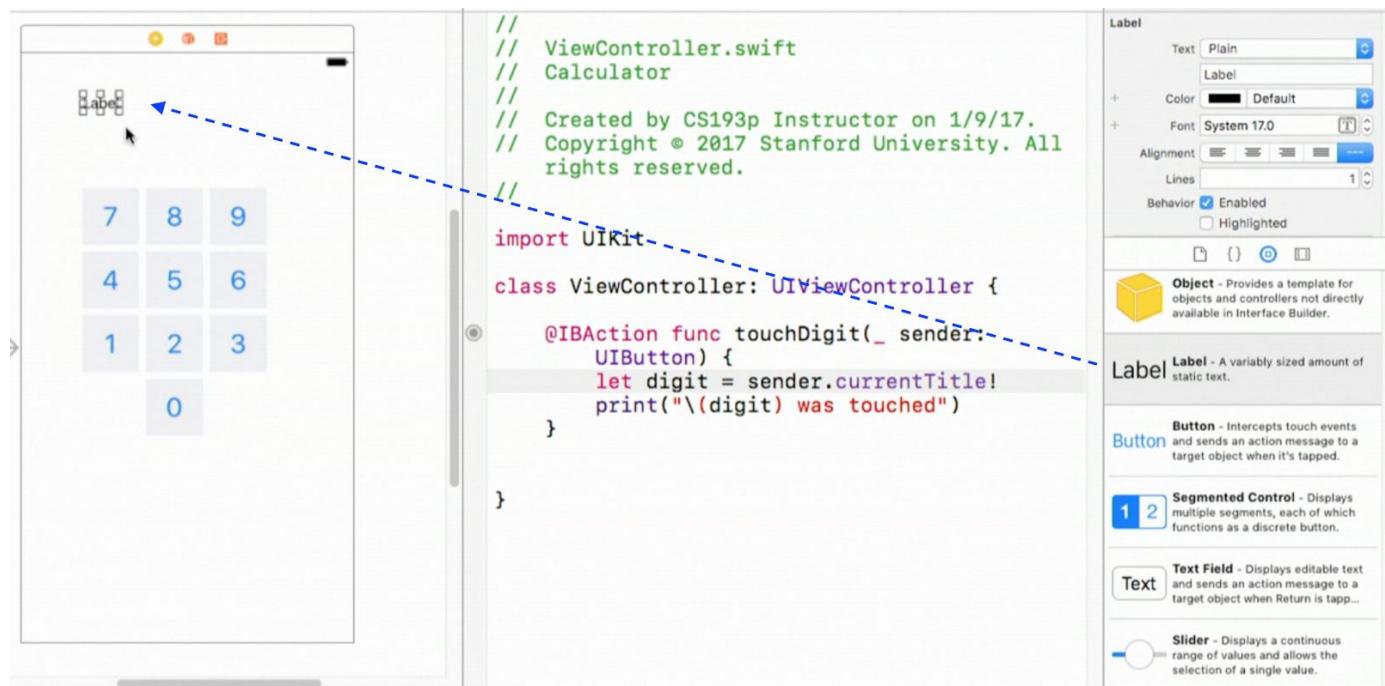


Между прочим, если вы кликаете на любом объекте в Палитре Объектов и оставляете на нем

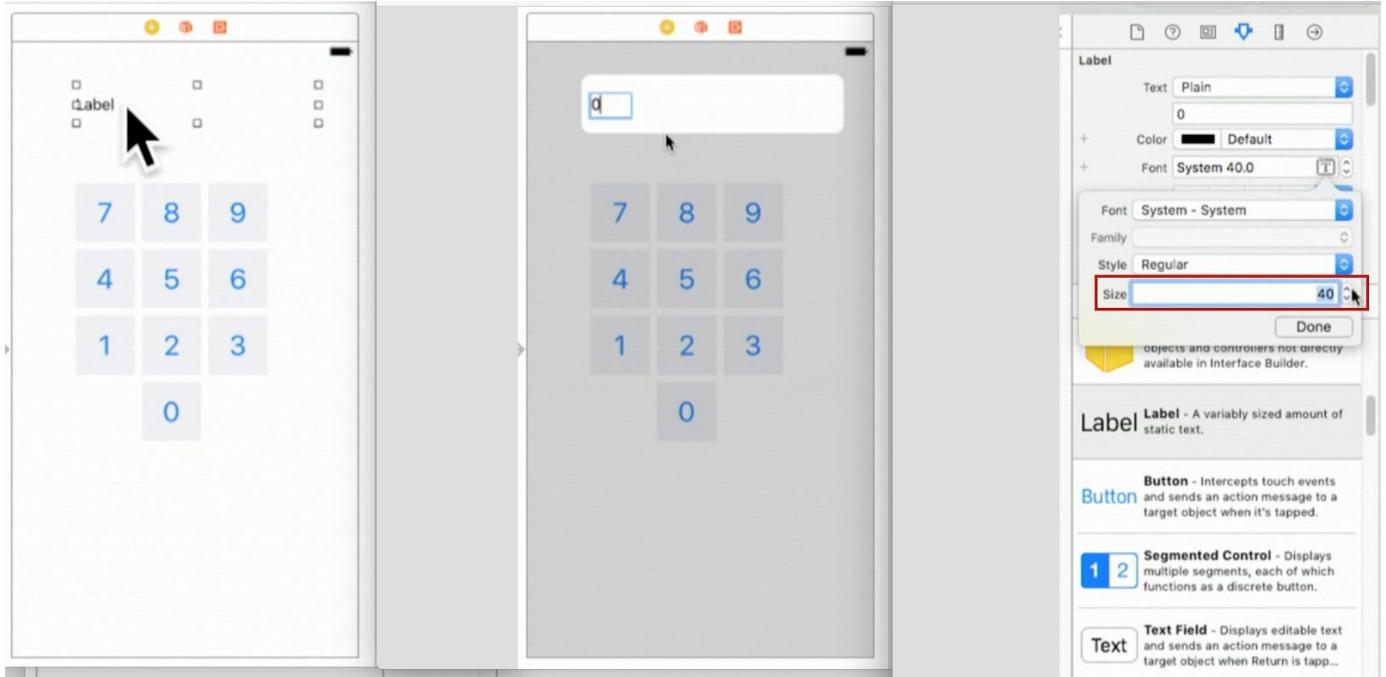
"мышку" в течение секунды, то получаете детальное объяснение функциональных возможностей объекта.



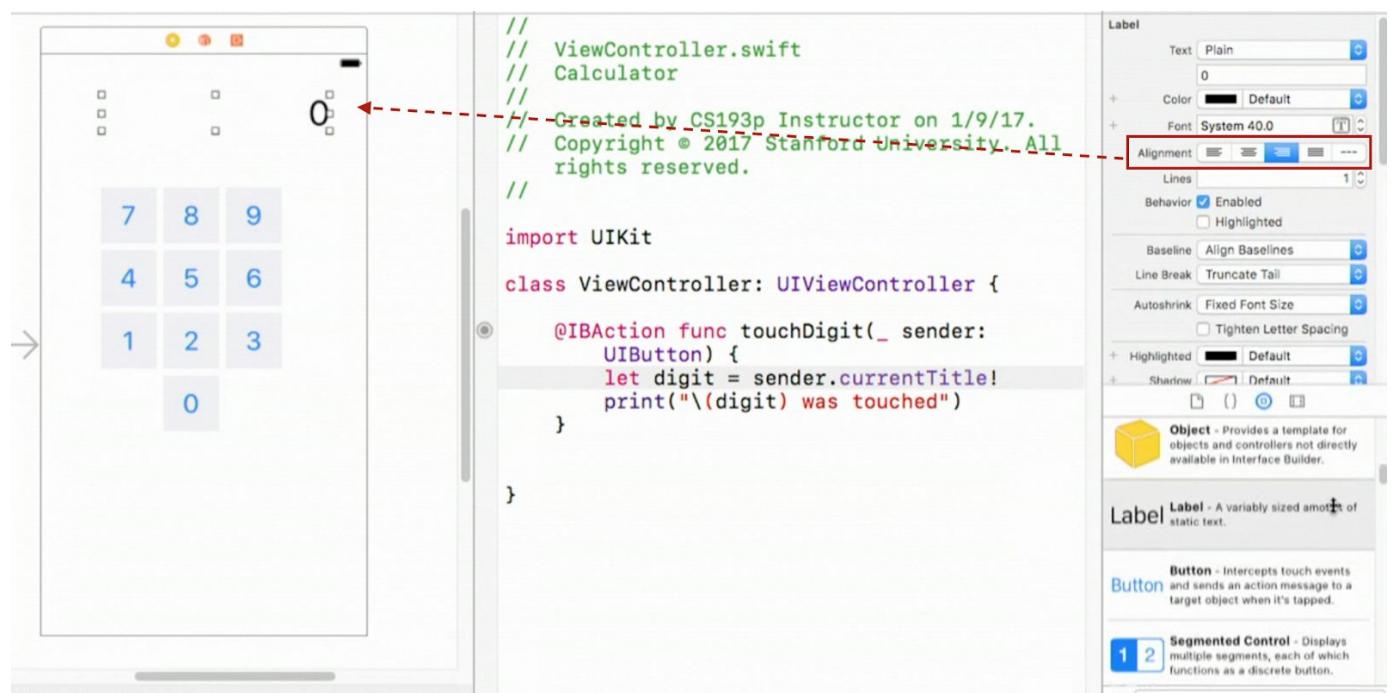
Я выбираю метку **Label** и перетягиваю ее на самый верх моего UI.



И делаю с ней то же самое, что и с первой кнопкой - изменяю размер, делаю больше шрифт, хочу чтобы на дисплее был **0** в самом начале:

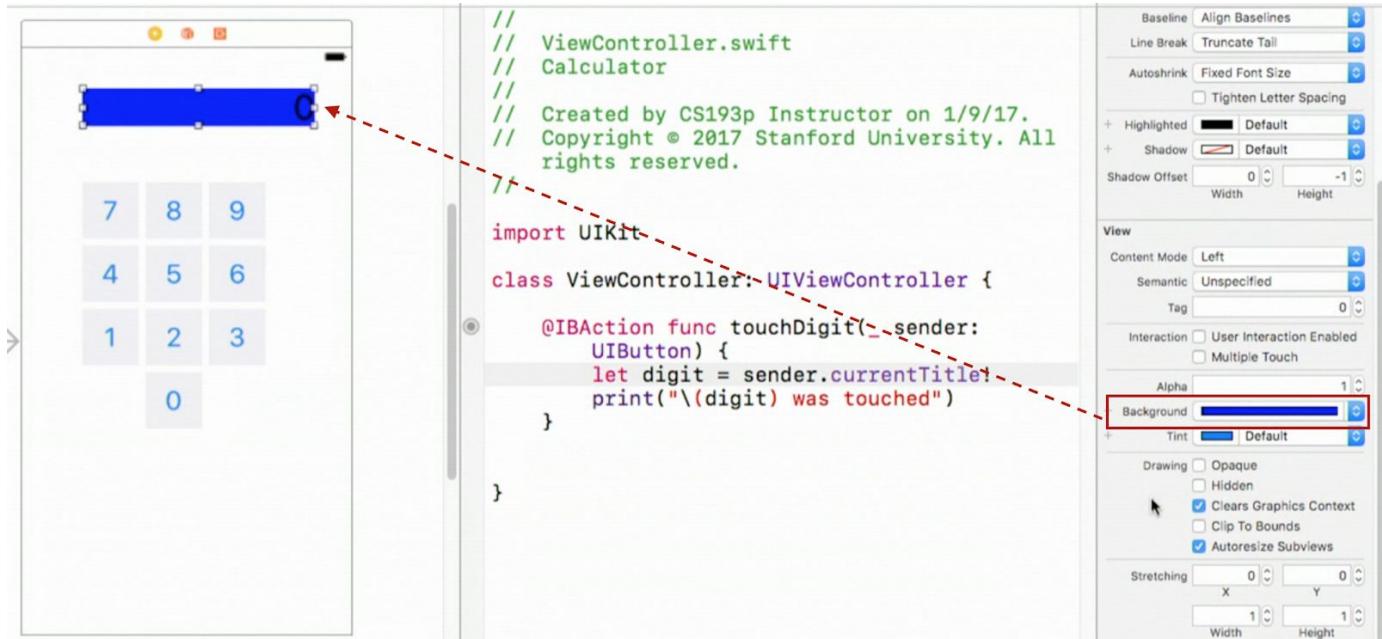


Текст на дисплее должен выравниваться вправо:

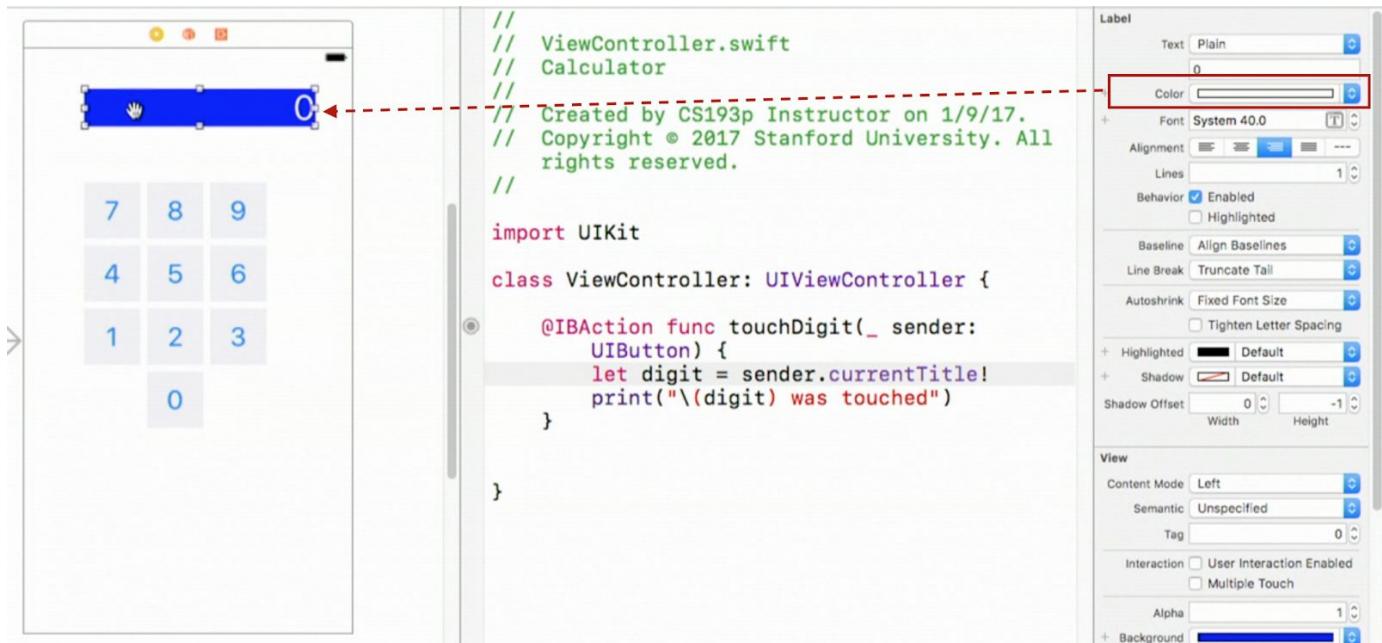


----- 55 -ая минута лекции -----

Может быть изменим цвета - сделаем цвет фона голубым:

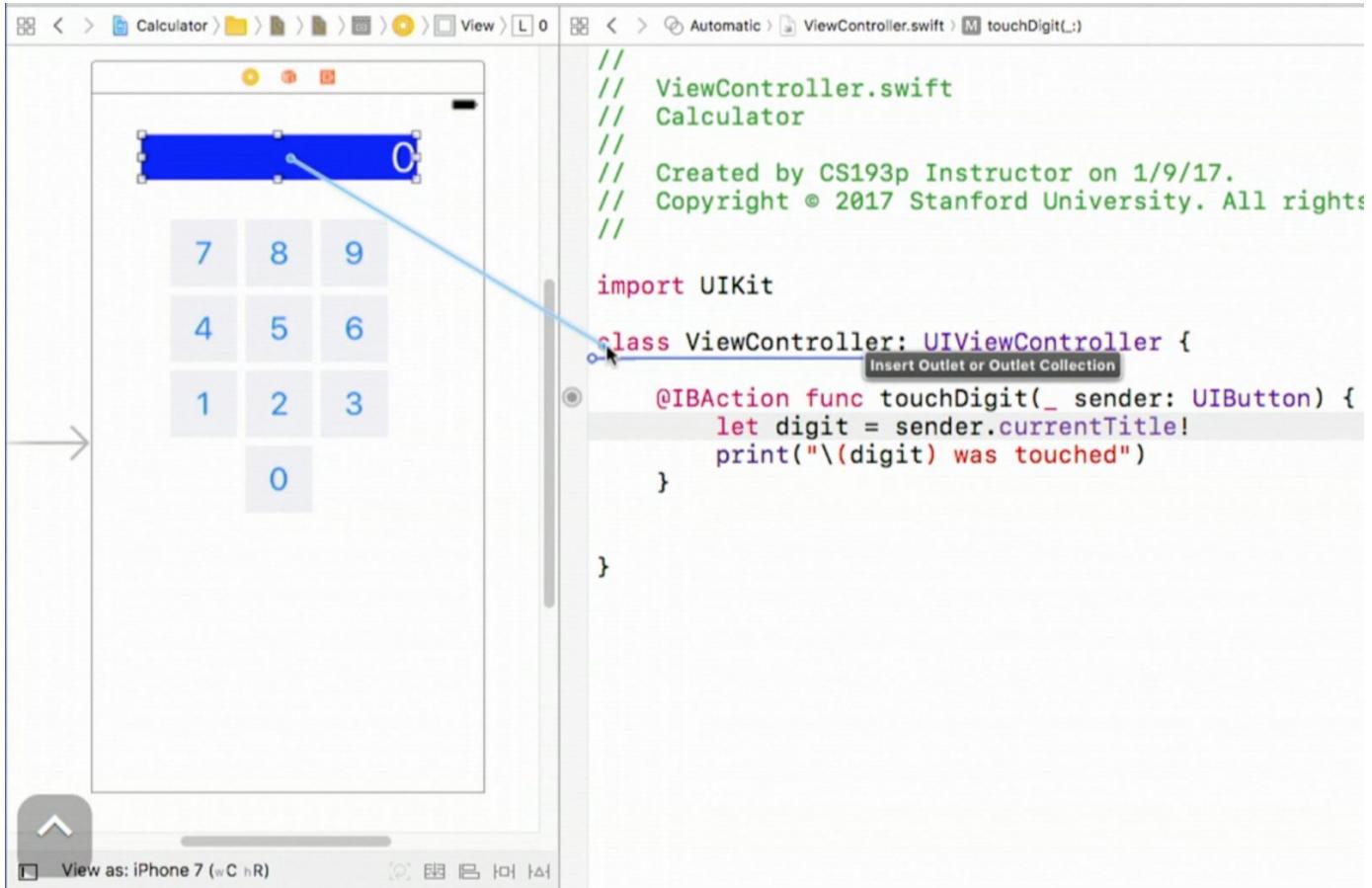


Я действительно не хочу, чтобы цвет текста был черным на голубом, я поменяю цвет текста на белый:

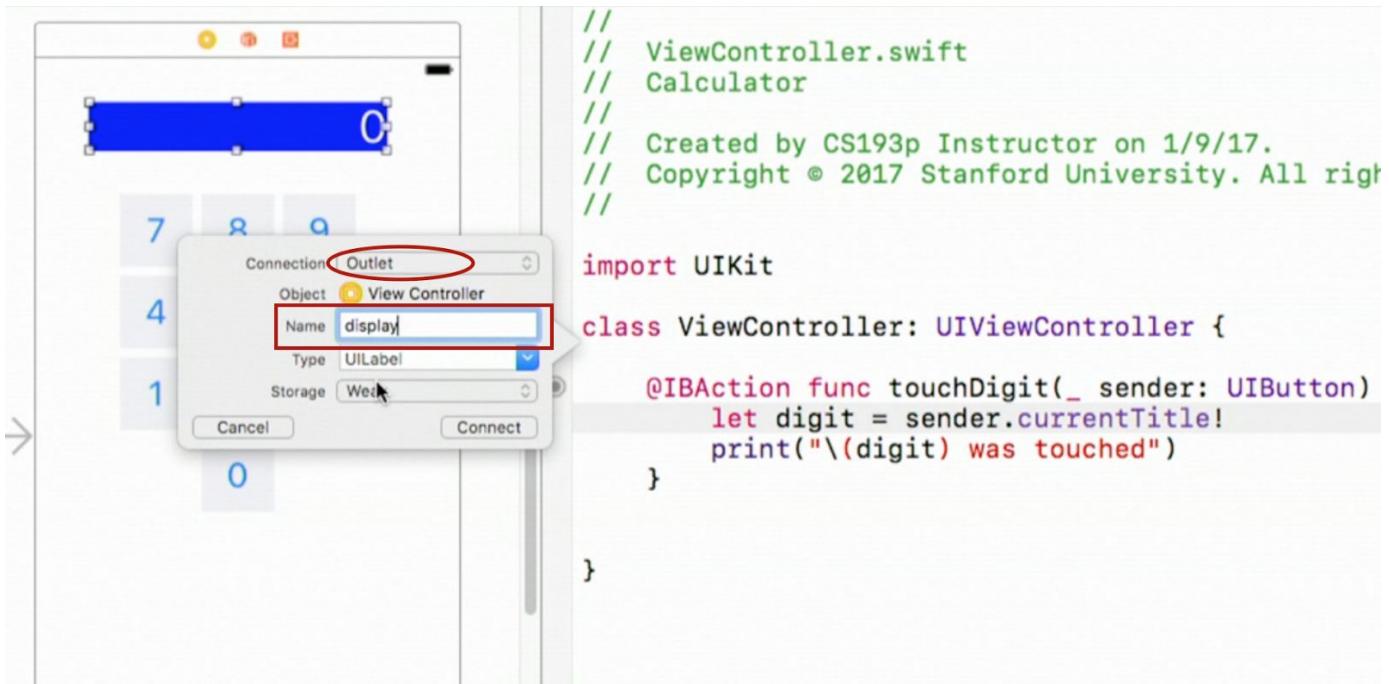


Наш дисплей выглядит вполне достойно на данный момент. Теперь нам надо сделать так, чтобы нажимая на кнопки, мы “разговаривали” с дисплеем и сообщали ему цифры, которые нажимает пользователь.

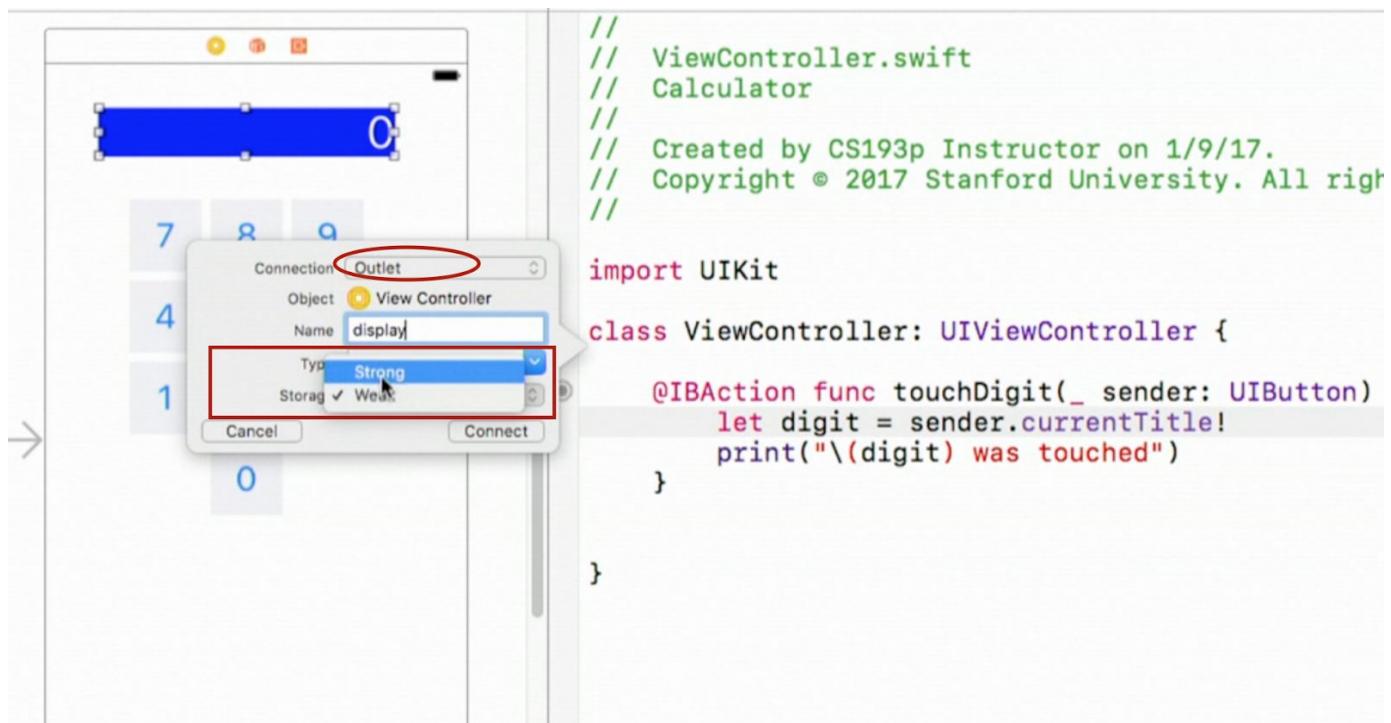
Как мы можем это сделать? Нам нужна связь между дисплеем и нашим кодом, но это связь другого типа, потому что мы не нажимаем на дисплее, как на кнопке, вследствие чего вызывается метод. Нам необходима переменная экземпляра или свойство (property), которое бы указывало на объект UI, и мы могли бы с ним “говорить” когда захотим, потому что нам нужно размещать на нем цифры. Механизм создания связи будет тот же. Мы нажимаем клавишу **CTRL** и тянем в направлении от метки в код:



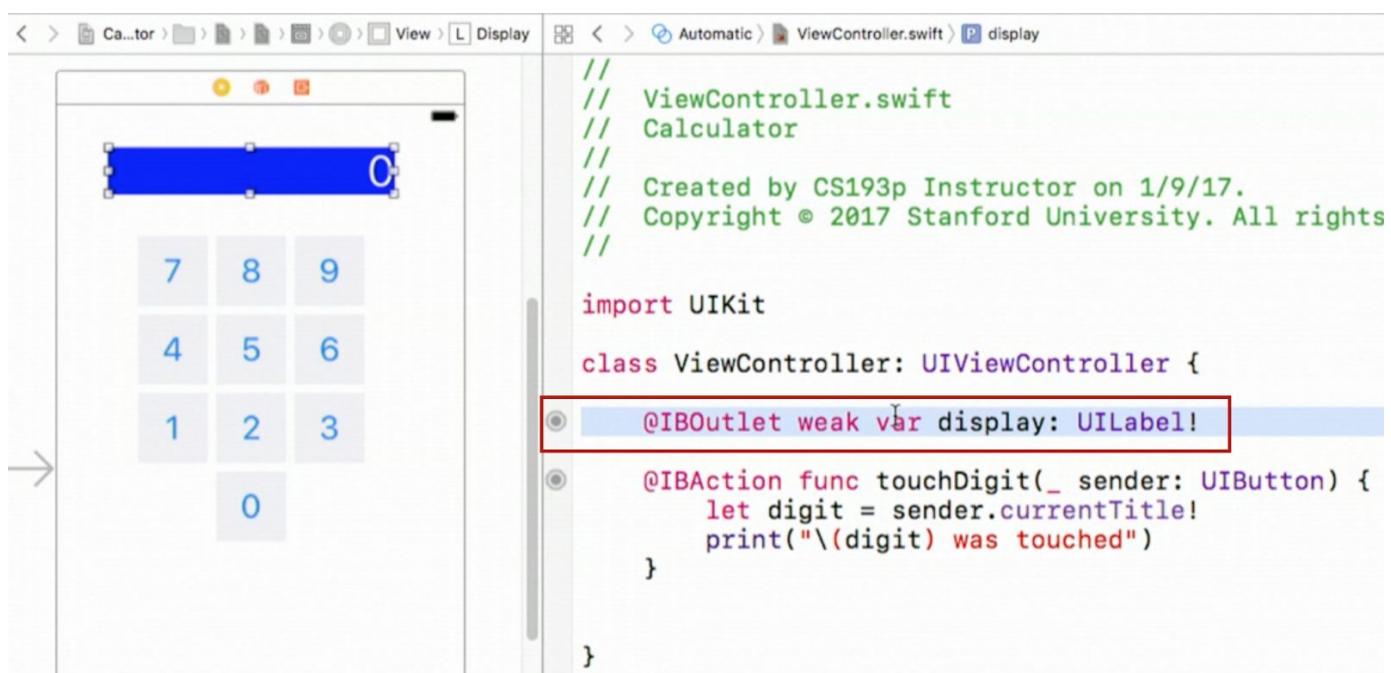
Отпускаем **CTRL**, тут же появляется диалоговое окошко, в котором на этот раз мы используем **Outlet**. **Outlet** означает свойство, которое указывает на элемент UI.



Мы называем наш **Outlet display**. Это наш дисплей. У меня уже есть тип - **UILabel**. В поле **Storage** - **weak** и **strong**, но пока не обращайте на это внимание. Я буду рассказывать об этом на следующей неделе:



Нажимаем “Connect” и получаем нашу первую переменную экземпляра класса в нашем классе.
Здорово!



И опять у нас есть некоторая часть **@IBOutlet**, которая принадлежит **Xcode**, кроме этого он размещает еще маленький кружочек здесь же. Если вы наведете мышку на этот кружочек, то увидите к чему на **UI** подсоединен этот **@IBOutlet**.

```
// ViewController.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University. All rights reserved.

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel!

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        print("\(digit) was touched")
    }
}
```

Ключевое слово **weak** мы игнорируем, а дальше идет декларирование свойства (**property**):

```
@IBOutlet weak var display: UILabel!
```

Конечно, оно начинается с **var** или может быть также **let**. Если вы хотите, чтобы ваша переменная экземпляра класса (**instance variable**) только устанавливалась в самом начале и потом никогда бы не изменялась, вы можете использовать **let**. Но это бывает крайне редко, хотя вы имеете право это делать. Обычно используется **var**. Имя переменной экземпляра класса - **display**.

"**: UILabel!**" - это тип. Можно догадаться, что здесь что-то делается с **Optionals**.

Поначалу присутствие восклицательного **!** знака может сбить вас с толку, ведь в нормальной ситуации он означает "развертывание" **Optional**. Очевидно, что мы не можем здесь что-то "разворачивать", мы здесь просто декларируем свойство. Здесь восклицательный **!** знак - это в точности то же самое, что и вопросительный **?** знак. Фактически я сейчас заменю восклицательный **!** знак на вопросительный **?** знак, а позже мы поменяем его обратно на восклицательный **!** знак, и вы увидите в чем разница.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        print("\(digit) was touched")
    }
}
```

Он означает тот же самый **Optional**, что и вопросительный **?** знак.

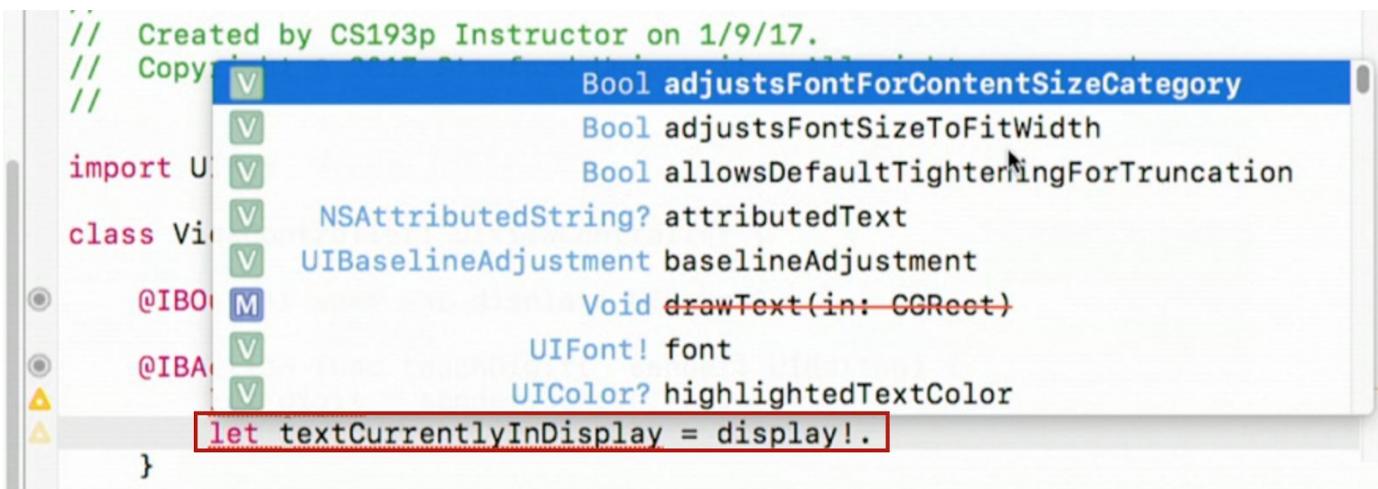
Но тип свойства **display** - это **Optional<UILabel>**.

Почему это **Optional<UILabel>**, а не просто **UILabel?** Почему вообще нужно, чтобы **Outlet display** был **Optional**? Потому что, когда ваш **UI** появляется в самом начале, **iOS** необходимо несколько наносекунд, чтобы "подцепить" все для вас. Поэтому в тот момент, когда **UI** появился впервые, **Outlets** не установлены, они являются **Optional** с состоянием **not set** ("не установлено"), затем **UI**

“подцепляется” для вас, и после этого они являются установленными навсегда. И это очень важно, что они устанавливаются навсегда (forever), и мы увидим, что дальше они имеют дело с восклицательным ! знаком, от которого я только что избавился. Но сейчас для лучшего понимания мы будем полагать, что **display** - это просто **Optional**, которое нам необходимо “разворачивать” (**unwrap**) всякий раз, когда мы его используем, это просто.

Теперь вместо печати на консоли цифр, давайте разместим их на дисплее **display**, и каждый раз при нажатии цифры, мы будем добавлять цифру в конец дисплея. Например, если у нас на дисплее - **56** и мы хотим добавить (append) **2**, то мы получим **562**, если еще добавляем **8**, то получаем **5268**. Мы хотим постоянно добавлять цифры в конец.

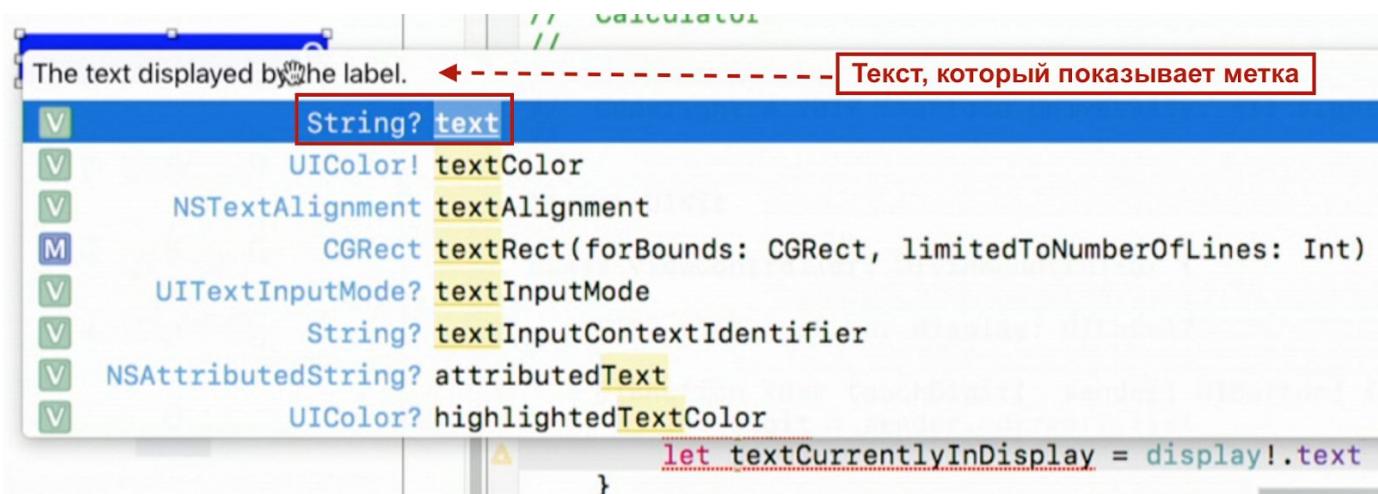
Нам нужен текст, который в данный момент находится на дисплее, чтобы добавить к нему цифру. Поэтому мне нужна еще одна локальная переменная, которая также будет константой с именем **textCurrentlyInDisplay** и я получу ее, посылая сообщение **display**, который я должен “развернуть” (**unwrap**) с помощью восклицательного ! знака, а потом поставить точку “.”:



```
// Created by CS193p Instructor on 1/9/17.  
// Copy  
//  
import UIKit  
  
class View : UIView {  
    @IBOutlet var display: UILabel!  
    @IBInspectable var font: UIFont! = UIFont.systemFont(ofSize: 16)  
    @IBInspectable var highlightedTextColor: UIColor? = nil  
    let textCurrentlyInDisplay = display!.  
}
```

The screenshot shows the Xcode interface with code completion. A tooltip "The text displayed by the label." is shown above the code. The variable `display` is selected in the completion dropdown, which lists various properties of the `UILabel` class. The property `text` is highlighted with a red box and has a red arrow pointing to the tooltip. The tooltip text is "Текст, который показывает метка".

Здесь представлены все сообщения, на которые реагирует метка **display!**, их сотни. Я применяю ту же самую хитрость, что и прежде. То есть просто напечатаю слово “text”, потому что я хочу извлечь текст из метки:



```
The text displayed by the label. ← ----- Текст, который показывает метка  
String? text  
UIColor! textColor  
NSTextAlignment textAlignment  
CGRect textRect(forBounds: CGRect, limitedToNumberOfLines: Int)  
UITextInputMode? textInputMode  
String? textInputContextIdentifier  
NSAttributedString? attributedText  
UIColor? highlightedTextColor  
let textCurrentlyInDisplay = display!.text
```

The screenshot shows the Xcode interface with code completion. A tooltip "The text displayed by the label." is shown above the code. The expression `display!.text` is selected in the completion dropdown, which lists various properties of the `UILabel` class. The property `text` is highlighted with a red box and has a red arrow pointing to the tooltip. The tooltip text is "Текст, который показывает метка".

Смотрите! Первая же строка дает нам текст, который показывает метка. Это **Optional <String>** - **String?**.

Прекрасно - я ее беру. Теперь у меня есть **text**, но он - **Optional**, поэтому я “разворачиваю” его с помощью восклицательного ! знака:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
    }
}
```

Итак, у меня есть **textCurrentlyInDisplay**. Если я **option**-кликну на ней, то мы увидим, что это **String**.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
    }
}
```

Declaration let textCurrentlyInDisplay: String
Declared In ViewController.swift

Теперь я могу просто установить текст на метке **display**:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    }
}
```

Я “разворачиваю” метку **display**, и беру его свойство **text**, которое устанавливаю в текст, который в данный момент показывает **display**, плюс цифра **digit**.

Заметьте, что когда я устанавливаю **Optional** равно чему-то, а свойство **text** для метки является **Optional**, то вам неизбежно “разворачивать” его и ставить восклицательный **!** знак вслед за **text**. Мы устанавливаем **Optional**, и компилятор знает, что **Optional** должно иметь ассоциированное значение типа **String**, которое и связывается с этим **Optional** свойством.

----- 60 -ая минута лекции -----

```
@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    let textCurrentlyInDisplay = display!.text!
    display!.text = textCurrentlyInDisplay + digit
}

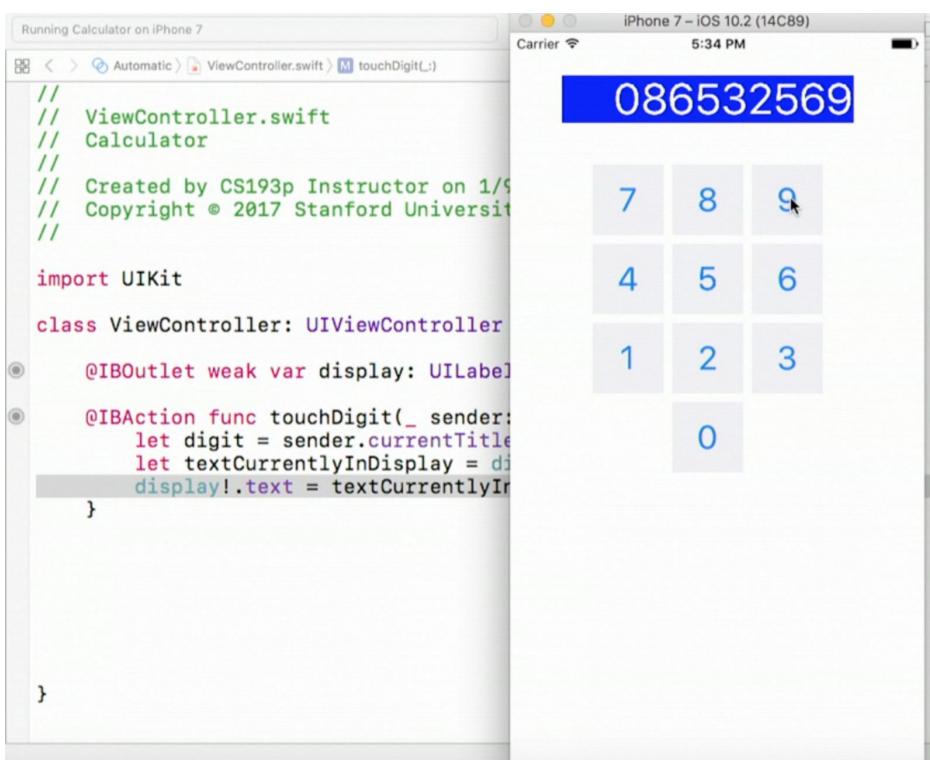
Declaration var text: String? { get set }
Description The text displayed by the label.
This string is nil by default.
In iOS 6 and later, assigning a new value to this property also replaces
the value of the attributedText property with the same text, albeit
without any inherent style attributes. Instead the label styles the new
string using the shadowColor, textAlignment, and other style-related
properties of the class.
Availability iOS (8.0 and later), tvOS (9.0 and later)
Declared In UIKit
More Property Reference
```

Кроме того, заметьте, что свойства **text** вслед за типом **String?** Приводится небольшой синтаксис **{get set}**, который означает возможность и чтения, и установки этого свойства в отличие текущего заголовка кнопки **currentTitle**, который можно только читать **{get}**, и нельзя устанавливать:

```
@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    let textCurrentlyInDisplay = display!.text!
    display!.text = textCurrentlyInDisplay + digit
}

Declaration var currentTitle: String? { get }
Description The current title that is displayed on the button.
The value for this property is set automatically whenever the button
state changes. For states that do not have a custom title string
associated with them, this method returns the title that is currently
displayed, which is typically the one associated with the normal state.
The value may be nil.
Availability iOS (8.0 and later), tvOS (9.0 and later)
Declared In UIKit
More Property Reference
```

Давайте запустим приложение и посмотрим, как это работает. Наш UI выглядит прекрасно, в том числе и дисплей. Давайте попробуем нажимать кнопки “8”, “6” и т. д.



Прекрасно, действительно добавляет наши цифры на дисплей, но впереди остается **0**, что, конечно, неправильно. Нуль не является частью того, что я печатаю на моей цифровой клавиатуре. Нуль появился потому, что он находился на дисплее с самого начала. Нуль не должен располагаться впереди всех цифр и это очень простая проблема. Она возникла потому, что мы не научили наш Калькулятор определять, когда пользователь находится в середине ввода числа, а когда он только что начал ввод. А кроме того, на дисплее может появится результат операции. Очевидно, что, когда мы печатаем, мы не хотим, чтобы нам что-то добавлялось со стороны - спасибо, не надо.

Поэтому мы должны **научить** наш “Умный Калькулятор” различать ситуацию, когда пользователь находится в середине печати, а когда - нет. И мы сделаем это с помощью еще одного свойства с именем **userIsInTheMiddleOfTyping**, которое будет иметь тип **Bool**:

```
import UIKit

❶ class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    var userIsInTheMiddleOfTyping: Bool

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    }
}
```

Теперь, что касается такого длинного имени. Возможно, мы могли бы назвать переменную **typing**. Вы знаете, что нужен компромисс между краткостью и понятностью. Краткость очень приветствуется, но понятность - более важна. Может быть достаточно было бы **isTyping**. Я все-таки предпочитаю взять сторону понятности. Так что я использовал это длинное имя. Другая причина, по которой я взял такое длинное имя заключается в том, что, напечатав это длинное имя один раз, мне больше ни разу не придется еще раз его печатать, потому что **Xcode** всегда будет дополнять то, что я начал печатать и помогать мне.

Теперь, когда я добавил такую прекрасную переменную **var**, она имеет тип **Bool**, а **Bool**, между прочим, - это то, что может быть **true** или **false**, я получил ошибку. Причем ошибку в строке, в которой я вообще ничего не делал:

```
import UIKit

❶ class ViewController: UIViewController {           ! Class 'ViewController' has no initializers

    @IBOutlet weak var display: UILabel?

    var userIsInTheMiddleOfTyping: Bool

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    }
}
```

Нам говорят, что у класса **ViewController** нет инициализаторов. Что за ерунда? Я ничего не делал

ни с каким **var**?

Дело вот в чем. В **Swift** все свойства (**properties**) должны быть инициализированы. Все до одного, никаких исключений. Есть два способа инициализировать ваши свойства в классе **class** или структуре **struct**.

Один способ - с помощью инициализаторов, это такой специальный метод с именем **init** (I-N-I-T). У него может быть любое количество аргументов, которые требуются классу, но внутри реализации инициализатора необходимо инициализировать все неинициализированные свойства (**properties**). Сегодня мы не будем говорить об инициализаторах. Я расскажу о них немного в следующую Среду. Так что мы не будем использовать инициализатор, потому что есть второй способ инициализации, который состоит просто в том, чтобы дать свойству значение (value). В нашем случае это **false**. Очевидно, что при старте приложения, пользователь не находится в середине ввода числа, он находится в начале, поэтому **false**.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    var userIsInTheMiddleOfTyping: Bool = false

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    }
}
```

Это избавило нас от ошибки.

Фактически, нам не нужно указывать тип : **Bool**. Понимаете почему? Потому что у нас есть значение **false**, которое может принадлежать только **Bool**. Таким образом, **Swift** может “выводить из контекста” (infer), что это должно быть **Bool**.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel?

    var userIsInTheMiddleOfTyping = false

    Declaration var userIsInTheMiddleOfTyping: Bool ) {
        Declared In ViewController.swift
            let textCurrentlyInDisplay = display!.text!
            display!.text = textCurrentlyInDisplay + digit
    }
}
```

И опять, если мы задаем свойствам значения, то их тип можно не указывать - он будет “выводится” из контекста.

А как насчет **display**?

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var display: UILabel? I

    var userIsInTheMiddleOfTyping = false

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    }
}
```

У этого свойства нет инициализатора. Почему же компилятор “не жалуется”? Как же так? Ведь это свойство не получает никакого значения, следовательно должен быть инициализатор?

Все из-за того, что оно - **Optional**. А с **Optionals** мы будем обращаться особым образом. Они всегда инициализируются автоматически со значением **nil**, а **nil** означает **not set** “не установлено”.

```
@IBOutlet weak var display: UILabel? = nil
```

В **Swift** **nil** имеет только этот смысл - оно означает, что **Optional** имеет значение **not set** “не установлено”.

Таким образом **Optional** автоматически, всегда получает значение при декларировании. Поэтому имеет смысл определять свойство как **Optional**, потому что его не нужно устанавливать в самом начале, оно может устанавливаться только тогда, когда вы этого захотите.

Вы можете установить **UILabel** во что-то конкретное с самого начала, это также возможно:

```
@IBOutlet weak var display: UILabel? = UILabel
```

UILabel **UILabel**

```
Int32 MNT_MULTILABEL
CFString! KCFURLLocalizedLabelKey
AccessibilityDelegate UIScrollViewAccessibilityDelegate
TimeInterval UIApplicationBackgroundFetchIntervalNever
TimeInterval UIApplicationBackgroundFetchIntervalMinimum
CFURL! CFBundleCopyAuxiliaryExecutableURL(bundle: CFBundle!, exec
CFOptionFlags kCFSocketAutomaticallyReenableDataCallBack
```

Но если вы не сделаете этого, то оно получит значение **not set** “не установлено”. Мы выберем именно этот вариант :

```
import UIKit

class ViewController: UIViewController {
    ...
    @IBOutlet weak var display: UILabel?
}

@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    print("\(digit) was touched")
}
```

Итак, у нас есть локальная переменная `userIsInTheMiddleOfTyping` и мы можем ее использовать в нашем коде. Если пользователь в середине печати числа, то мы используем весь этот код в методе `touchDigit`, а если нет, то мы просто устанавливаем `display!.text` равным введенной цифре `digit`, потому что мы только начали ввод нового числа и, естественно, мы переходим в состояние, когда мы в середине печати числа:

```
import UIKit

class ViewController: UIViewController {
    ...
    @IBOutlet weak var display: UILabel?

    var userIsInTheMiddleOfTyping = false

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        if userIsInTheMiddleOfTyping {
            let textCurrentlyInDisplay = display!.text!
            display!.text = textCurrentlyInDisplay + digit
        } else {
            display!.text = digit
            userIsInTheMiddleOfTyping = true
        }
    }
}
```

Все поняли?

Давайте убедимся, что мы сделали правильные вещи и запустим приложение. Посмотрим, решили ли мы проблему лидирующего нуля. Должно быть, потому что когда мы начинаем печатать, мы не находимся в середине набора числа, а набираем новое.

The screenshot shows the Xcode interface with two main panes. On the left, the code editor displays `ViewController.swift` for a calculator application. The code includes comments about the file being created by CS193p Instructor on 1/9/2017. It imports `UIKit` and defines a `ViewController` class that conforms to `UIViewController`. The class contains an `@IBOutlet weak var display: UILabel` and an `@IBAction func touchDigit(_ sender: UIButton)` method. This method checks if the user is currently in the middle of typing a number. If so, it updates the display's text to include the digit from the sender. Otherwise, it sets the display's text to the digit and marks the user as in the middle of typing. The right pane shows the iPhone 7 simulator running iOS 10.2 (14C89). The status bar indicates "Carrier" and "5:39 PM". The calculator's display shows "56986". Below the display is a 3x3 grid of buttons labeled 7, 8, 9 in the top row, 4, 5, 6 in the middle row, and 1, 2, 3 in the bottom row. The button labeled "3" has a cursor arrow pointing to it. A single button labeled "0" is located below the grid.

```
// Viewcontroller.swift
// Calculator
//
// Created by CS193p Instructor on 1/9/17.
// Copyright © 2017 Stanford University.
//

import UIKit

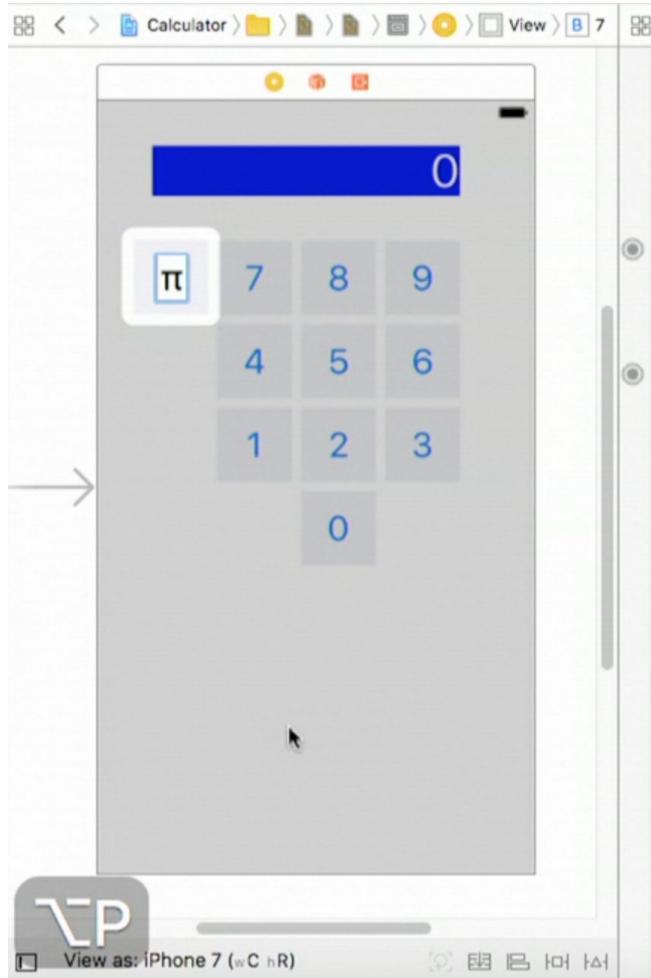
class ViewController: UIViewController {
    @IBOutlet weak var display: UILabel!
    var userIsInTheMiddleOfTyping = false

    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        if userIsInTheMiddleOfTyping {
            let textCurrentlyInDisplay = display!.text
            display!.text = textCurrentlyInDisplay + digit
        } else {
            display!.text = digit
            userIsInTheMiddleOfTyping = true
        }
    }
}
```

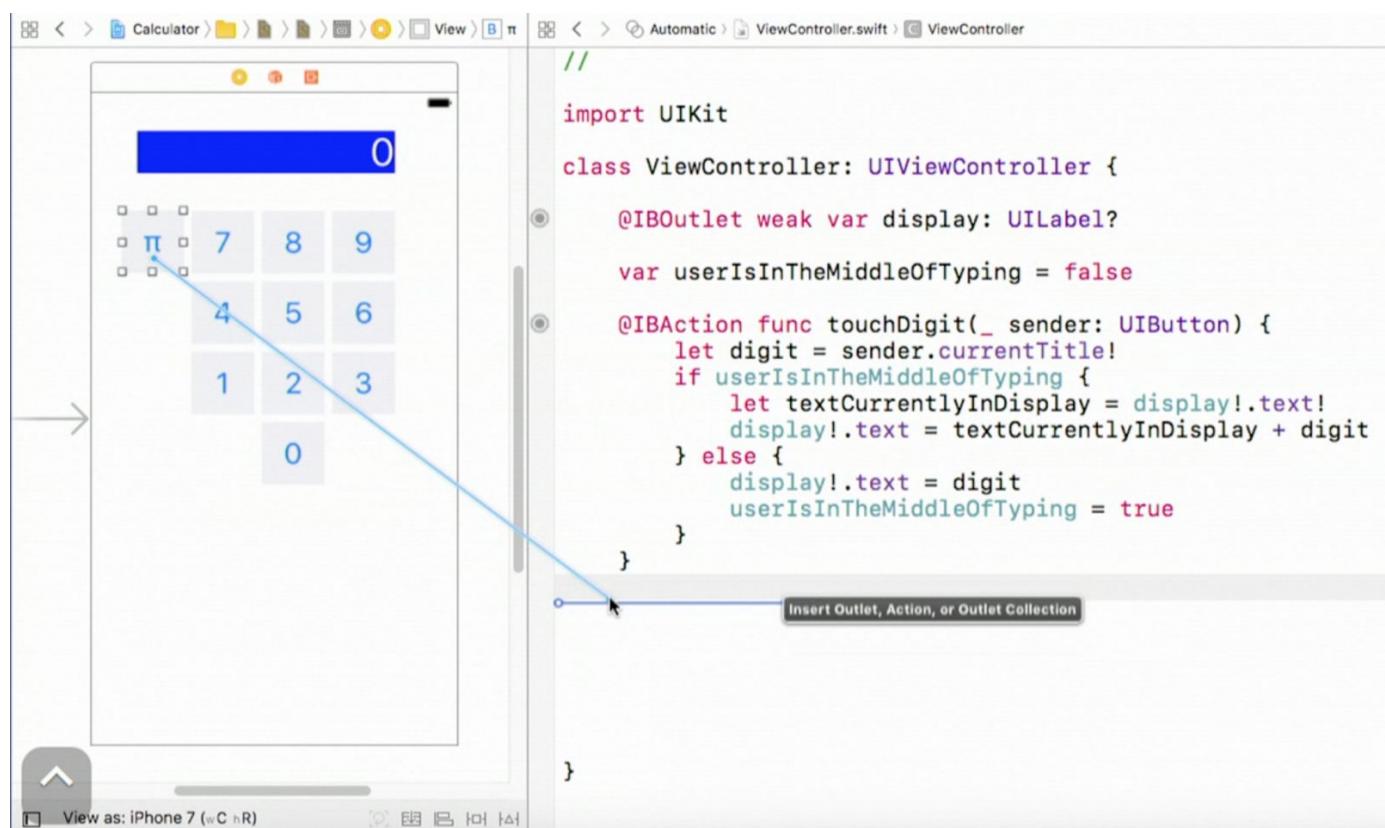
Начинаем набирать цифры “5”, “6”, “9”, “8” и т.д.. Да, проблема лидирующего нуля решена. Если же мы находимся в середине набора числа, то цифры добавляются. Так что все в порядке.

Следующим шагом в разработке Калькулятора разместим кнопки для некоторых операций. Мы уже умеем вводить числа, теперь займемся операциями.

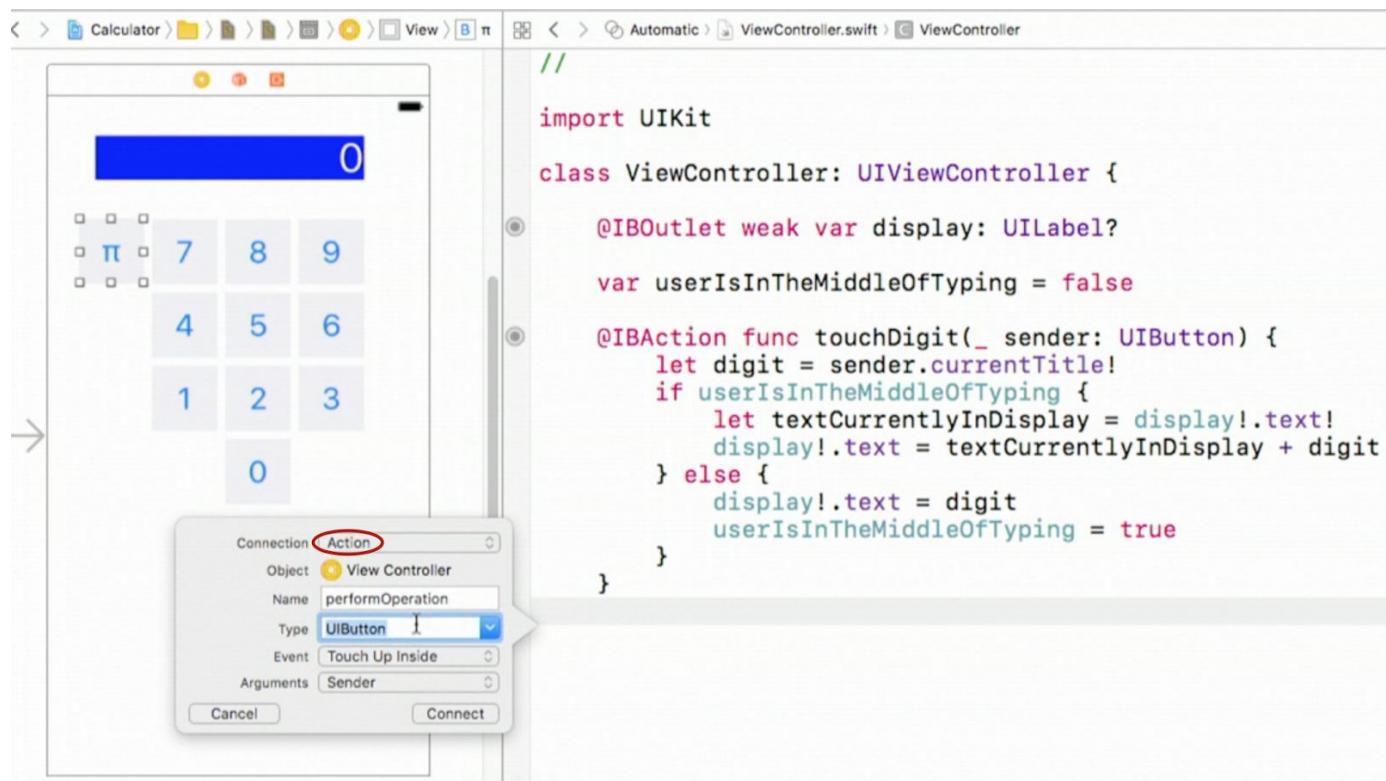
Сейчас я сделаю очень плохую вещь - скопирую кнопку “7” и просто напечатаю там простейшую операцию “ π ”, смысл которой заключается в размещении значения “ π ” на дисплее. Через секунду вы поймете, почему я поступил плохо.



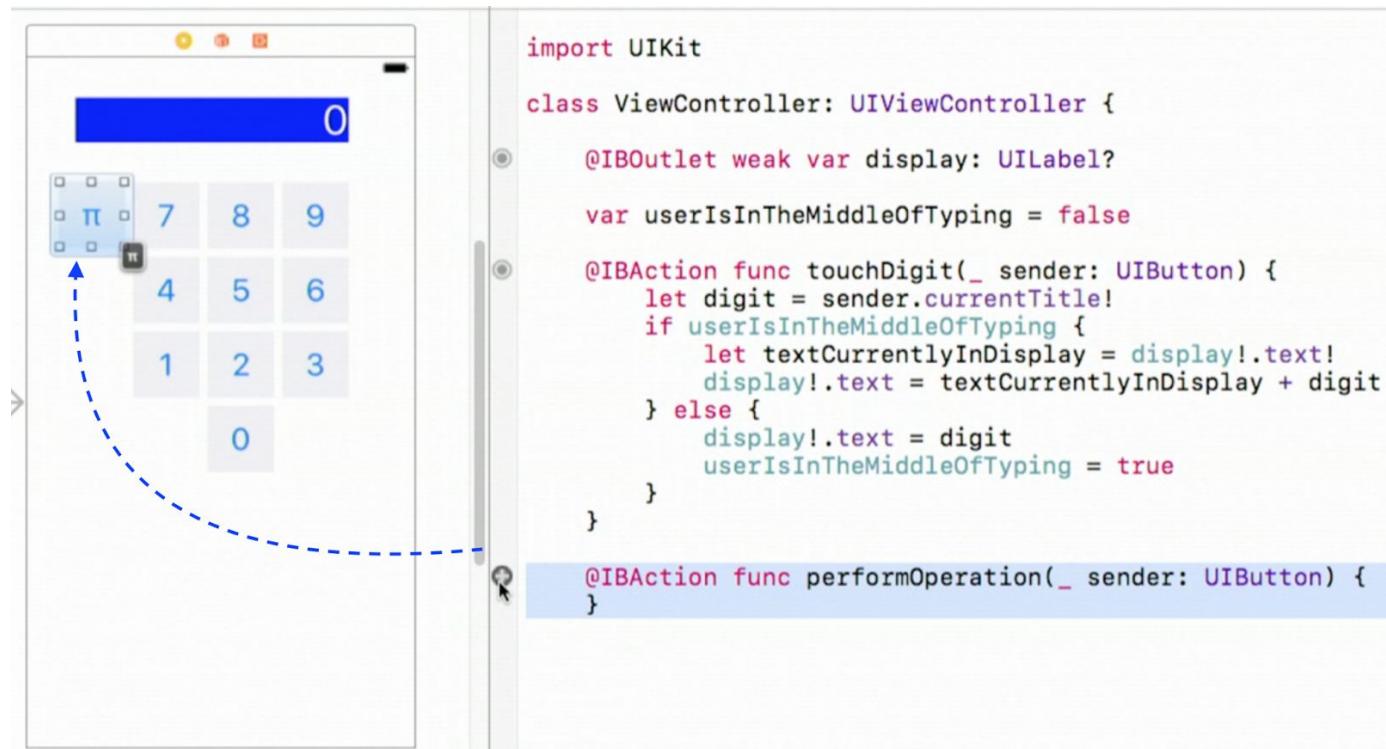
Я буду использовать **CTRL-перетягивание** в код, чтобы “подвязать” метод к кнопке с операцией “**π**”.



Это будет **Action**, а не **Outlet**. Это будет похоже на **touchDigit**. Я назову наш новый метод **performOperation**, потому что это именно то, что он будет делать: он будет выполнять операции.



Я должен убедиться, что переключил в поле **Type** тип на **UIButton**, не забудьте это сделать в вашей Домашней работе. И я нажимаю на кнопку “**Connect**”.



Теперь у меня есть новый метод, который “подцеплен” к кнопке “**π**”, это здорово. Что я буду делать в этом методе? То же самое, что и в **touchDigit**: я буду спрашивать кнопку, какую операцию она выполняет. Я буду использовать локальную переменную **mathematicalSymbol**, потому что на этих кнопках будут символы математических операций:

```
@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    if userIsInTheMiddleOfTyping {
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    } else {
        display!.text = digit
        userIsInTheMiddleOfTyping = true
    }
}

@IBAction func performOperation(_ sender: UIButton) {
    let mathematicalSymbol = sender.currentTitle!
}
```

Прекрасно, теперь я получил “π”, но я хочу показать вам, как и обещал, как сделать “развертку” **Optional** так, чтобы приложение не завершалось аварийно. Давайте решим, что, если у нас пустая кнопка, то есть кнопка, у которой не установлен заголовок, мы не будем делать ничего, не будем выполнять никаких операций, как будто бы на эту кнопку никто не нажимал.

Теперь посмотрите насколько важны **Optional** и какой мощный синтаксис имеется в **Swift** для этого типа: вопросительный знак ?, восклицательный знак !. Вам почти ничего не придется печатать при использовании. То, что я покажу относится к этому же разряду. Если вы хотите проверить **Optional** прежде, чем его “развернуть”, то вместо восклицательного ! знака в конце нужно разместить два символа вначале. Эти символы - **if**.

```
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle
}
```

Теперь прочтите это на английском языке: “If I can let mathematical symbol equal the sender's **currentTitle**, then” (Если допустить, что математический символ равен **currentTitle** отправителя, то ...).

Таким образом, если я смогу “развернуть” **Optional** и получить его ассоциированное значение, то я смогу разместить какой угодно код внутри этого **if**, при этом **mathematicalSymbol** будет строкой **String**, ассоциированным значением, то есть “развернутым” (**unwrapped**) **Optional**.

За пределами фигурных скобок **mathematicalSymbol** даже не определен и не имеет значения. Внутри фигурных скобок я мог бы написать:

```
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        if mathematicalSymbol == "π" {
        } else if mathematicalSymbol == "|"
    }
}
```

То есть бесконечные **if then else**, **if then else**, **if then else**, что является реально плохим кодом. Поэтому я буду использовать другой подход, другое маленько выражение. Я буду использовать

оператор **switch**. Множество языков программирования имеют оператор **switch**. Я буду переключаться по математическому символу **mathematicalSymbol**. Не все языки могут переключаться по строке. Некоторые могут и **Swift** в их числе.

Я могу написать :

```
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
        }
    }
}
```

В случае, если **mathematicalSymbol** равен “**π**”, тогда я хочу присвоить тексту дисплея значение, эквивалентное “**π**”, например, “3.1415926”. Совершенно ужасно печатать его как строку, но я вернулся сюда через секунду. Но это именно то, что я хочу сделать - я хочу установить **display** в “**π**”. Я получил ошибку. Что это за ошибка?

```
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
        }
    }
}
```

Эта ошибка сообщает нам, что оператор **switch** должен быть исчерпывающим (**exhaustive**), может быть стоит добавить предложение **default**. И это правда. В операторе **switch** вы должны представить каждый возможный случай (**case**). До тех пока пока мы не пройдем несколько следующих лекций, я предполагаю, что мы должны сделать что-то в следующем духе, пытаясь напечатать случаи для всевозможных строк:

```
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
            case "a":
            case "aa":
            case "aaa":
        }
    }
}
```

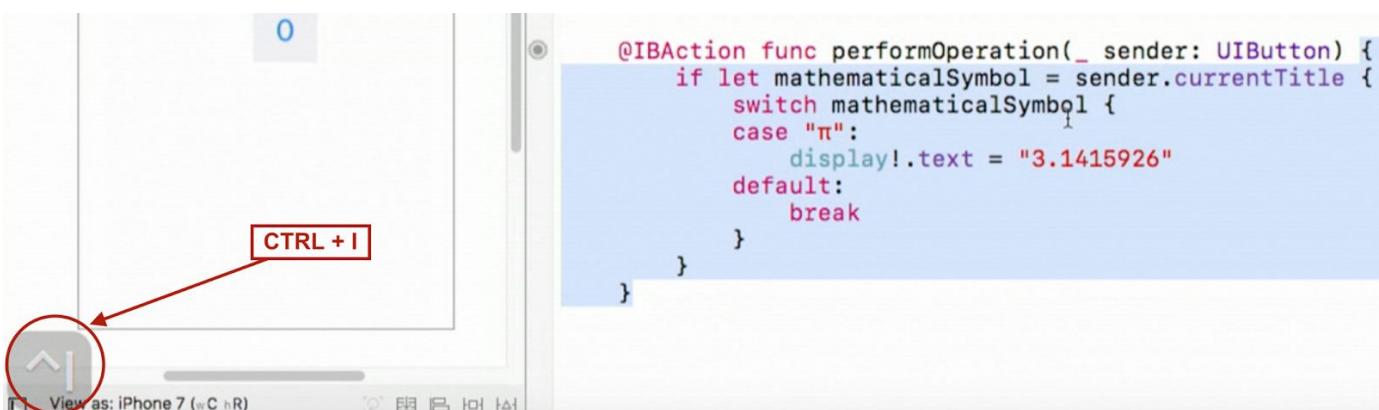
Но, к счастью, у нас есть случай (**case**) **default**, который означает “все остальные случаи”. Для этого обобщенного случая я использую команду **break** для выхода из **switch**.

```

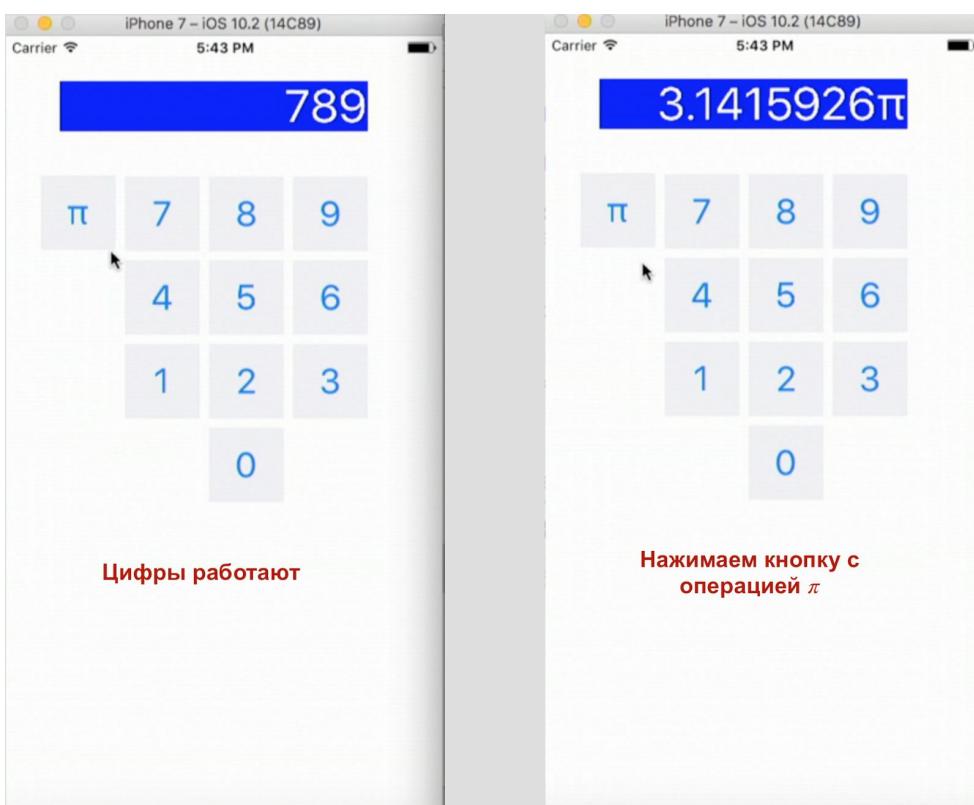
@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
            default:
                break
        }
    }
}

```

Заметьте, что отступы в коде для разных случаев несколько не согласованы. Должны быть одинаковые отступы для различных **case**, включая **default**. Реально крутая возможность состоит в выборе всего текста или даже целого файла и применении сочетания **Ctrl + I** для выравнивания отступов, это заново установит отступы абсолютно для всего. И я рекомендую это делать для всех ваших файлов с исходным кодом в вашей Домашней работе. Просто выделите все (**Select All**) и **Ctrl + I**:



Итак, мы получили операцию “ π ”, давайте посмотрим, как это работает. Цифры работают, затем нажимаем кнопку с операцией “ π ”:

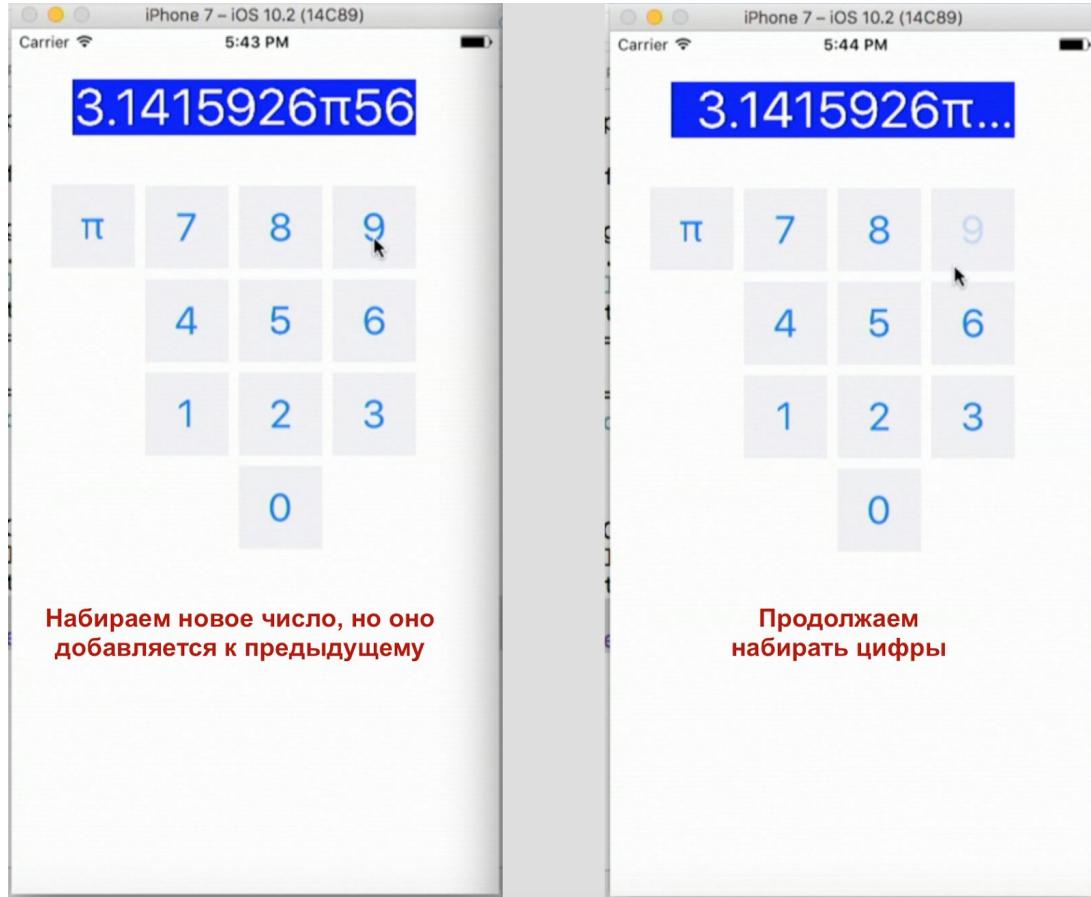


Что-то странное. Я получил значение “ π ”, которое я напечатал непосредственно в коде, но почему добавился символ операции “ π ”? Это кажется странным.

Давайте попробуем новое число. О! Не может быть! Когда я набираю новое число, оно добавляется в конец старого. Может потому, что здесь “ π ” присутствует?

Смотрите, что будет происходить дальше. Я продолжаю печатать цифры и у меня появляется многоточие и дисплей перестает воспринимать цифры.

Полная неразбериха.



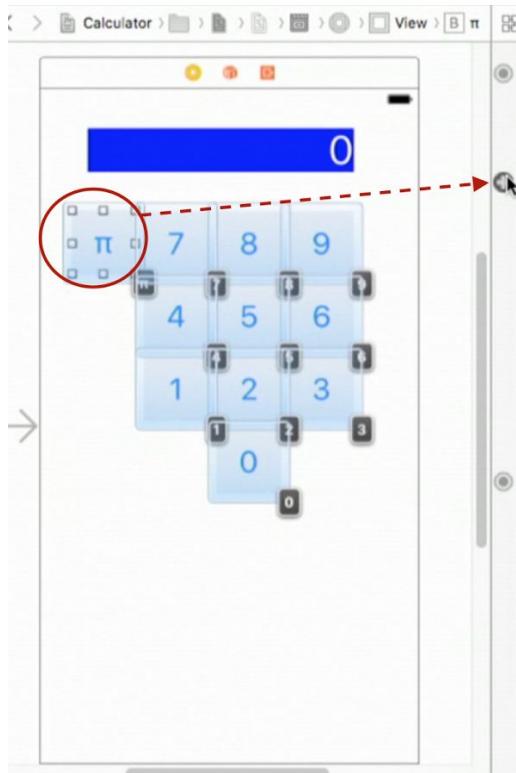
У нас с вами 3 больших проблемы. Это наличие “ π ” на дисплее. Второй факт, что цифры добавляются в конец, даже если печатается новое число. И третья проблема - многоточие. Как мы будем все их исправлять?

Давайте решим сразу все три проблемы и реально быстро.

Вначале проблема с “ π ” на дисплее в конце.

Давайте посмотрим, где я устанавливаю `display.text`? Я делаю это в трех местах, и единственное место, где происходит добавление - это метод `touchDigit`. Может быть кнопка с заголовком “ π ” подцепилась к этому методу?

----- 70 -ая минута лекции -----



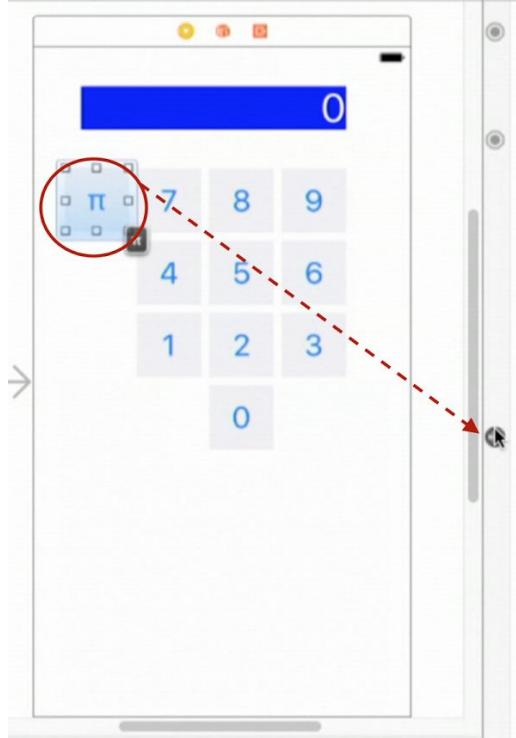
```
@IBOutlet weak var display: UILabel?

var userIsInTheMiddleOfTyping = false

@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    if userIsInTheMiddleOfTyping {
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    } else {
        display!.text = digit
        userIsInTheMiddleOfTyping = true
    }
}

@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
        case "π":
            display!.text = "3.1415926"
        default:
            break
        }
    }
}
```

Видите кнопку “π”? Она действительно “подцеплена” к методу **touchDigit** и выполняет его. Но она выполняет и метод **performOperation**:



```
@IBOutlet weak var display: UILabel?

var userIsInTheMiddleOfTyping = false

@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    if userIsInTheMiddleOfTyping {
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    } else {
        display!.text = digit
        userIsInTheMiddleOfTyping = true
    }
}

@IBAction func performOperation(_ sender: UIButton) {
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
        case "π":
            display!.text = "3.1415926"
        default:
            break
        }
    }
}
```

Похоже она выполняет оба метода.

Вначале она выполняет метод **performOperation** и размещает **3.1415926** на экране, а затем выполняет **touchDigit** и добавляет **digit**, которым в нашем случае является символом “π”. Это очень плохо. Как мы можем это исправить?

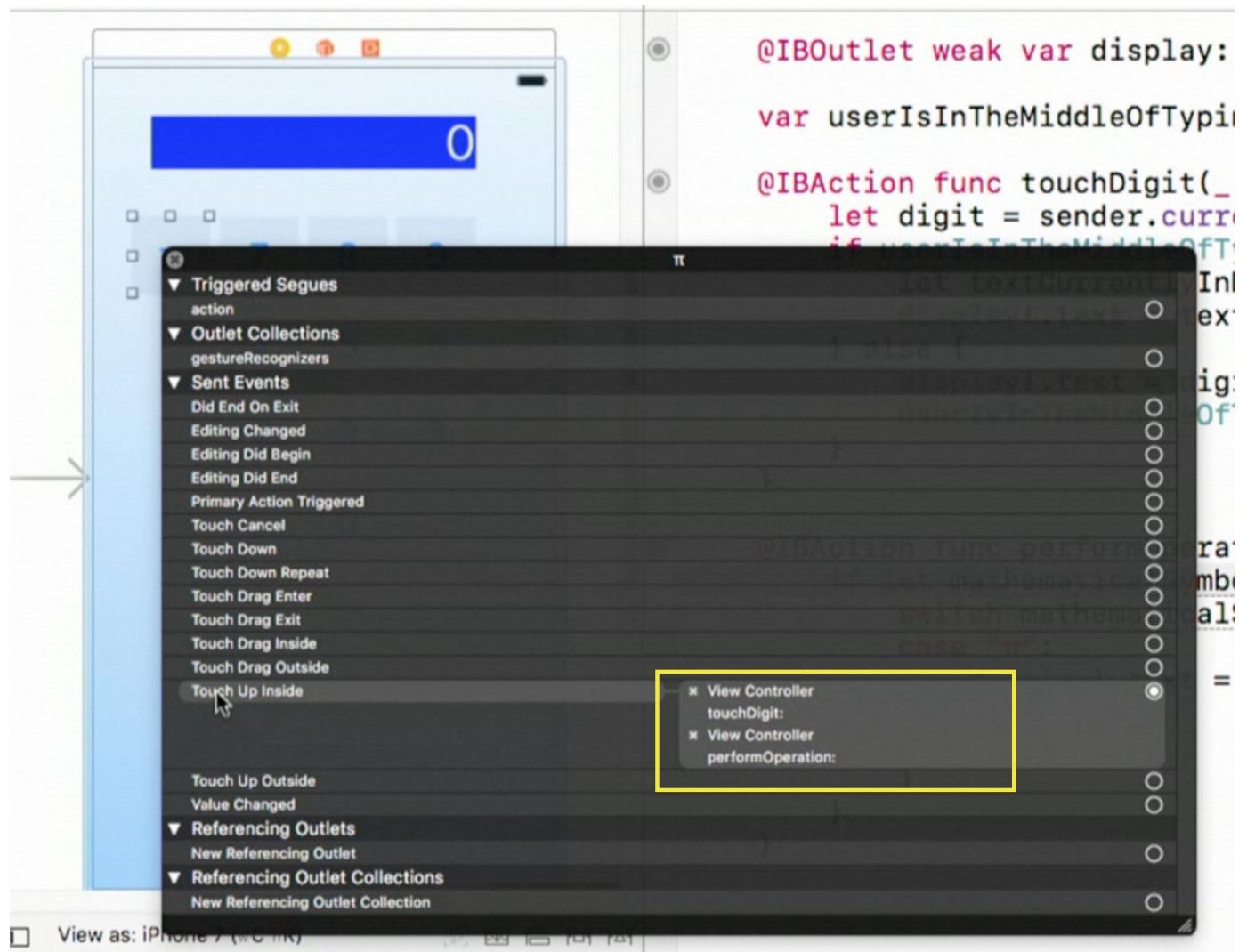
ВОПРОС: Как вы установили порядок выполнения методов **performOperation** и **touchDigit** ?

ОТВЕТ: Порядок не определен. Из моего опыта, это обычно алфавитный порядок, но официально он не определен. Вам не следует опираться на какой-то порядок. Но да, **performOperation**

выполняется перед **touchDigit**. Я не знаю почему. Но у вас никогда не будет “подцеплено” сразу два метода. Здесь это очевидная ошибка. И именно она вызывает все проблемы.

Как нам ее исправить?

У нас есть другой способ узнать, что подсоединенено к кнопке “**П**”, отличный от использования маленького кружочка. Он заключается в том, что если вы кликните правую кнопку мыши (или удерживая **CTRL**, просто кликнуть на мышке) на чем-нибудь из вашего **UI** (например, на кнопке “**П**”), то вы увидите все связи этого элемента **UI**.



Здесь представлены все переменные экземпляра класса, все методы и т.д.

Вы можете видеть, какие методы подсоединены к “**π**” для события **Touch Up Inside**, которое означает касание пальцем внутренней части кнопки и подъем пальца вверх. Такое событие посылает нам два сообщения : **touchDigit:** и **perfomOperation:**, но нам определенно не нужен метод **touchDigit:** для “**π**”.

Каким образом метод **touchDigit**: “привязался” к “π”? Потому что я скопировал и вставил кнопку “7”. Помните? Я говорил вам, что это плохая идея? Так вот именно поэтому она и плохая, сохраняет все “привязки”.

Давайте отсоединим метод `touchDigit:` от кнопки “π”. Для этого достаточно нажать на маленький крестик слева вверху `touchDigit:`.



Теперь связь `touchDigit:` ушла и осталась только `performOperation:`.



Эту проблему, связанную с размещение символа “π”, мы решили. Но другая проблема пока осталась.

Как насчет того, что я могу печатать дополнительные числа в конце “π”?

Здесь также простое решение.

Когда мы разрешаем добавлять цифры? Когда пользователь находится в середине ввода числа.

Как только я нажал “π”, я нахожусь в середине ввода числа? Нет, я просто нажал “π”, я не ввожу число. Фактически, каждый раз, когда мы вызываем метод `performOperation`, пользователь уходит из режима ввода числа и локальная переменная `userIsInTheMiddleOfTyping` равна `false`. Мы больше не находимся в середине ввода числа:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
            default:
                break
        }
    }
}
```

Что будет показывать `display`? Он будет показывать результат операции. Так что мы исправили и вторую ошибку.

А как насчет многоточия и усечения чисел?

Это очень умное исправление. В Инспекторе Атрибутов для нашего дисплея `display`, который является меткой `UILabel`, есть замечательная «фишка» с именем `Autoshrink`, благодаря которой метка `display` будет автоматически «сжиматься», например, не менее, чем до 9 point.



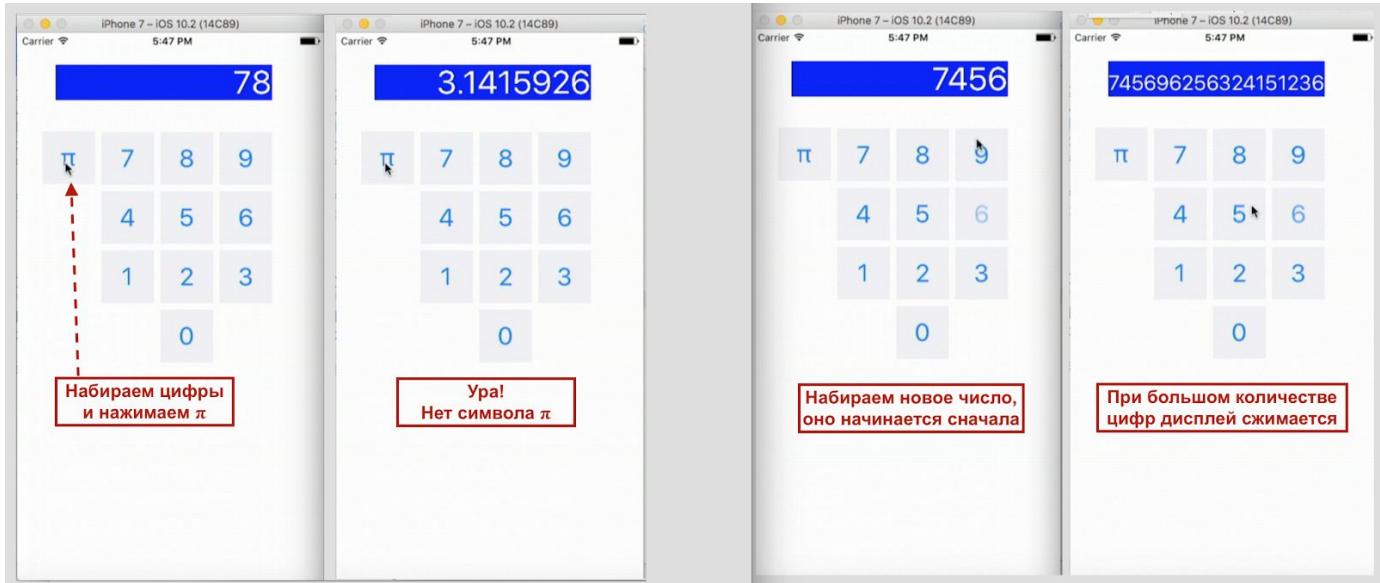
Это решит нашу проблему:



Теперь, когда мы получаем в `display` слишком много цифр, наш `display` сжимается вместо показа многоточия. Возможно, это не самое хорошее решение нашей проблемы, лучше было бы иметь Калькулятор, который показывает только определенное число цифр после десятичной точки. Это действительно было бы лучше и это будет вашим дополнительным пунктом в вашем Домашнем задании. Я желаю вам удачи в этом.

Итак, мы сделали исправления по всем трем проблемам, давайте посмотрим, что у нас получилось.

Набираем число и нажимаем “**π**”. На `display` мы видим только значение “**π**”, самого математического символа нет, он не добавляется в конец числа. Начинаем набор нового числа после ввода “**π**”, оно не добавляется в конец старого набора. Еще продолжаем набирать цифры и видим, что число на `display` сжимается, но мы не теряем введенные цифры.



Да, возможно, не лучшее решение, но я хотел показать вам **Autoshrink** (автосжатие).

Что следующее?

Мы с вами займемся маленькой проблемой, связанной с заданием значения "π" в методе

performOperation:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "3.1415926"
            default:
                break
        }
    }
}
```

Это выглядит ужасно и я хочу показать вам действительно крутую возможность в **Swift**, **Double.pi**,

что является числом двойной точности с плавающей точкой равным значению "π".

```
① @IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = Double.pi
            default:
                break
        }
    }
}
```

Но, конечно, я не могу вот так вот взять и присвоить **display!.text** значение **Double.pi**, потому что:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = Double.pi
            default:
                break
        }
    }
}
```

Нам говорят, что мы не можем присваивать тип **'Double' Optional** строке **'String?'** Потому что единственное значение, которое мы можем присвоить **Optional** строке помимо **nil**, должно иметь тип **String**, то есть мы таким образом устанавливаем ассоциированное значение.

Кто-нибудь может сказать мне, учитывая все сказанное на лекции, как мне преобразовать **Double.pi** в строку? Нет смелых?

Точно, обратный слэш и круглые скобки, это такая маленькая хитрость:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = "\(\Double.pi)"
            default:
                break
        }
    }
}
```

Помещаем **Double.pi** внутрь круглых скобок. **Doubles** могут преобразовываться в **String**, Победа! На самом деле это все-таки некрасиво и выглядит уродливо, потому что представленный выше механизм означает вставку объектов в строку, это не естественный способ преобразования, **Doubles** могут преобразовываться в **String**. Более естественный способ заключается в создании новой строки, вы уже видели синтаксис для создания нового объекта - новой структуры **struct** или нового класса **class**. Это имя класса или структуры и круглые скобки. Внутри этих круглых скобок помещается что-то, что класс может взять и превратить в один из своих экземпляров.

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String() "\(\Double.pi)"
            default:
                break
        }
    }
}
```

Класс может взять что-то в качестве аргументов. Помните? Я упоминал инициализаторы? Это аргументы для инициализатора, у вас может быть множество инициализаторов. У **String** их целая куча. Одним из них является инициализатор, который берет в качестве аргумента **Double**.

```

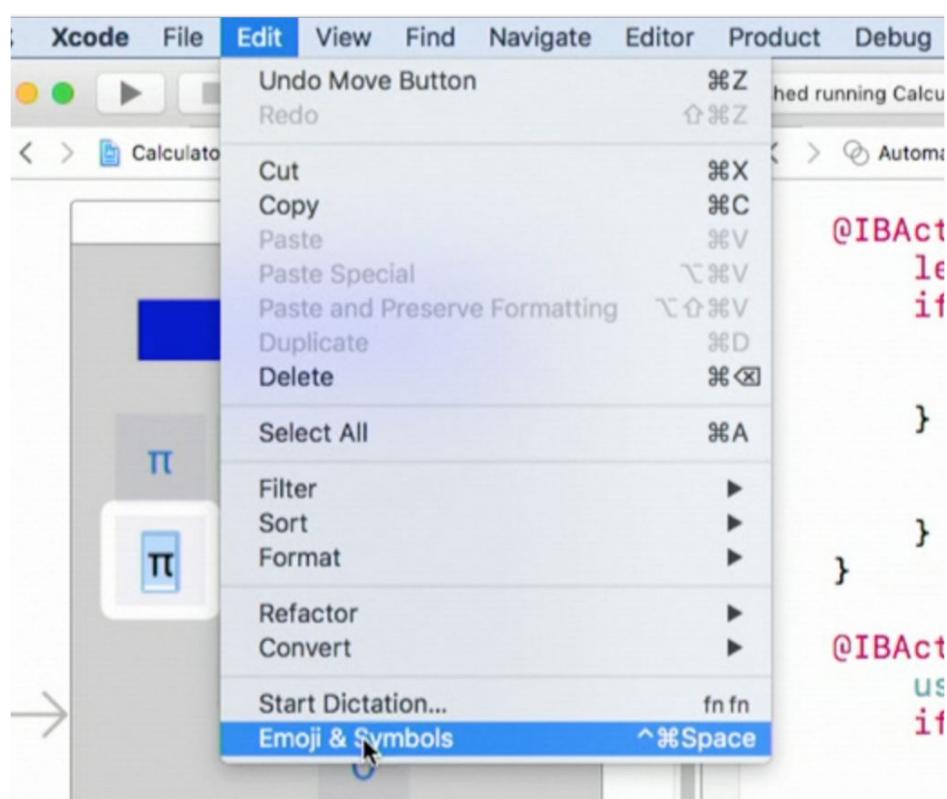
    @IBAction func performOperation(_ sender: UIButton) {
        userIsInTheMiddleOfTyping = false
        if let mathematicalSymbol = sender.currentTitle {
            switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            default:
                break
            }
        }
    }
}

```

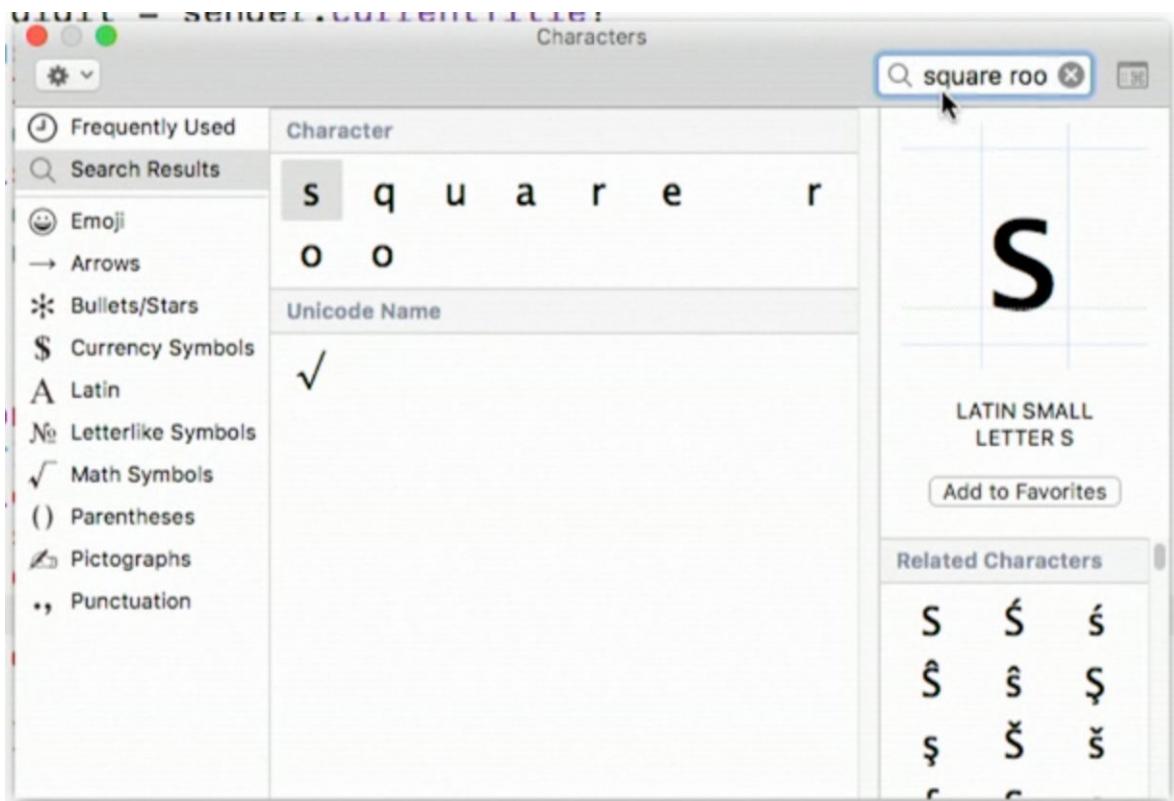
Так что этот способ будет естественным способом преобразования **Double** в **String**, он выглядит значительно эстетичнее в коде и более понятно, что же мы делаем.

Давайте добавим еще одну операцию. Я хочу добавить операцию взятия квадратного корня.

Копирую кнопку “**Π**” и заменяю символ “**Π**” на символ “ $\sqrt{}$ ” для квадратного корня, который я заимствую из меню **Edit /Emoji & Symbols**:

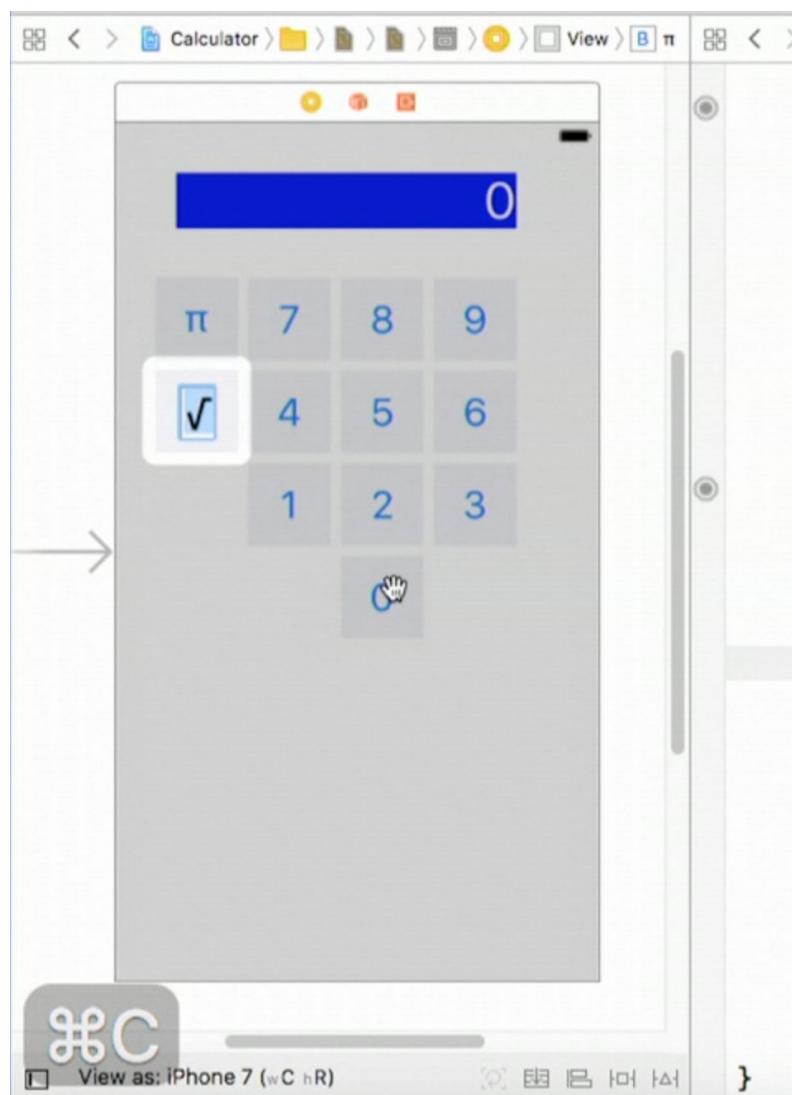


В полученном диалоговом окне вы можете осуществлять поиск, в нашем случае по тексту “**s q u a r e root**”:

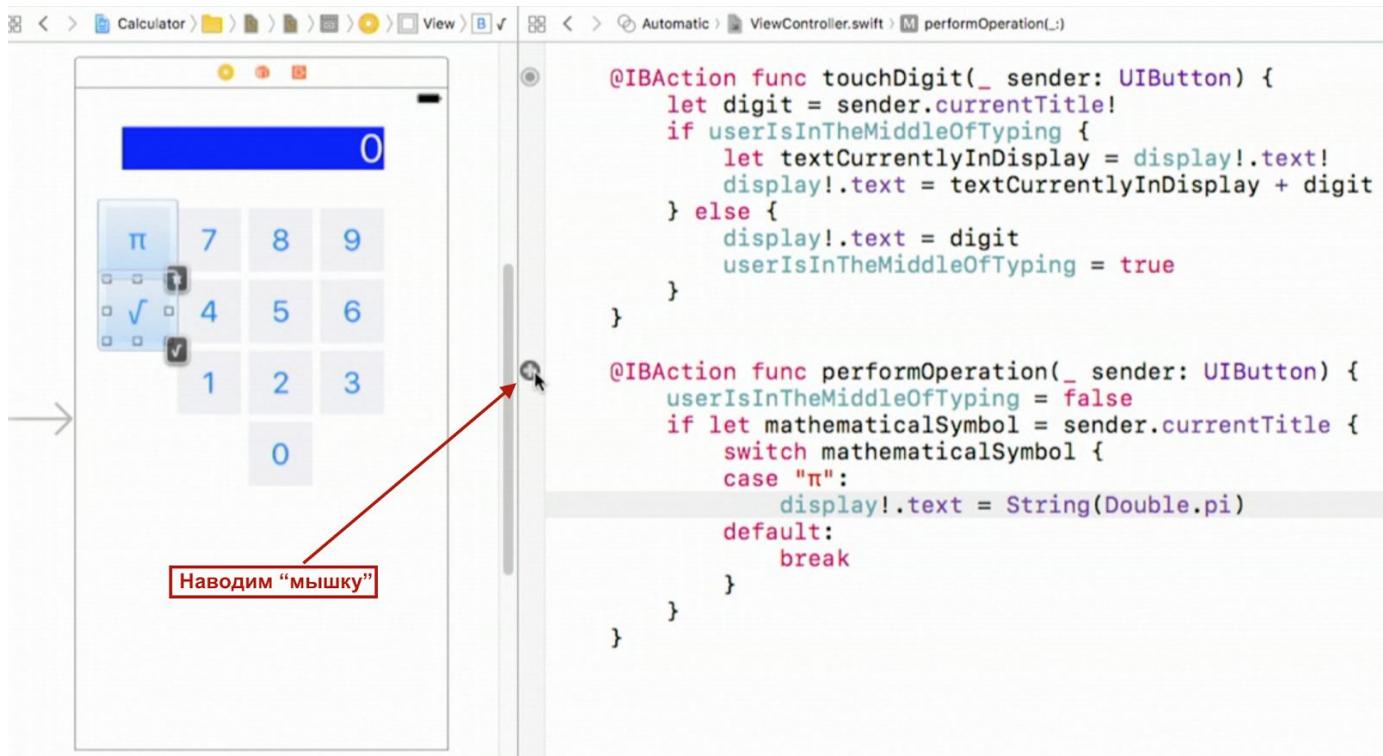


----- 75 -ая минута лекции -----

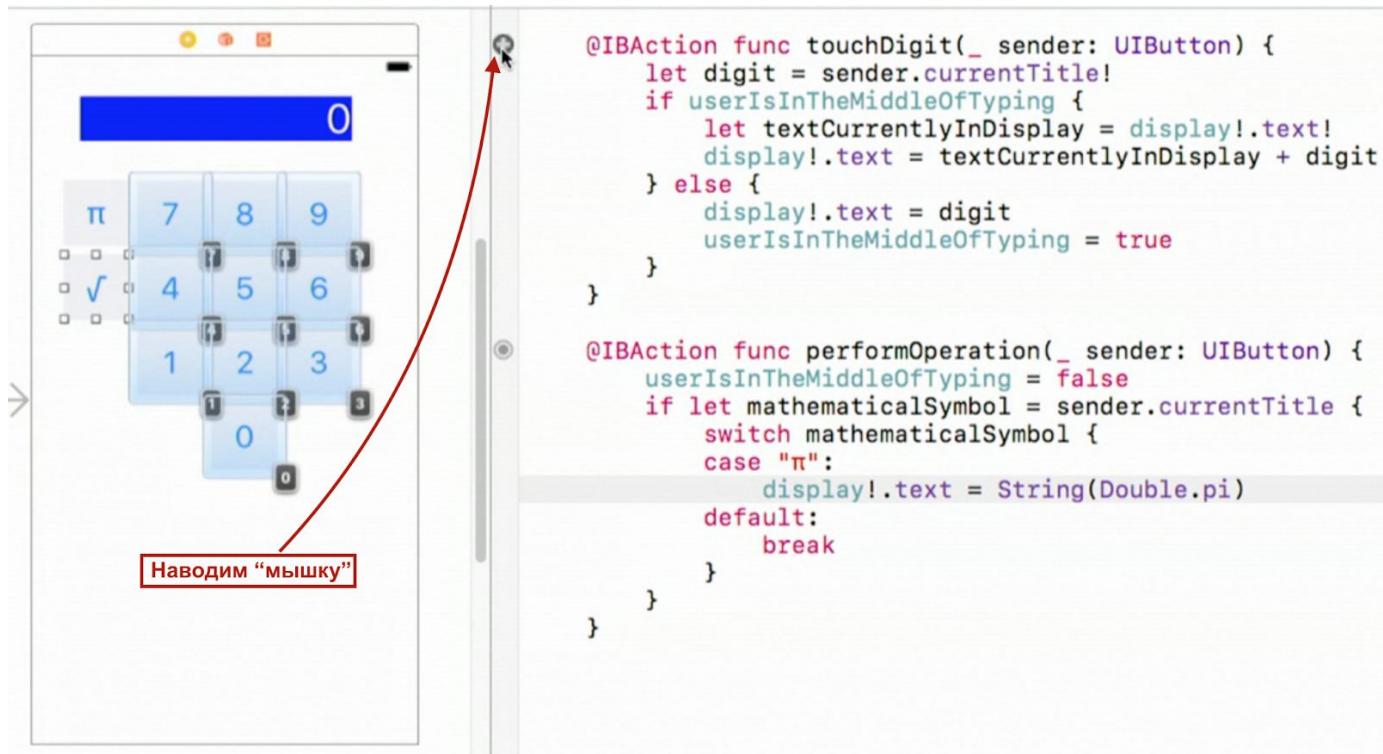
Помещаем на кнопку и копируем для использования в коде:



Убедимся, что эта кнопка подсоединенна к **performOperation**:



и не подсоединенна к **touchDigit**:



Все прекрасно. Нам нужно добавить **case** для квадратного корня в **performOperation**:

```
@IBAction func performOperation(_ sender: UIButton) {  
    userIsInTheMiddleOfTyping = false  
    if let mathematicalSymbol = sender.currentTitle {  
        switch mathematicalSymbol {  
            case "π":  
                display!.text = String(Double.pi)  
            case "√":  
                display!.text = sqrt()  
            default:  
                break  
        }  
    }  
}
```

Мы хотим, чтобы на дисплее `display!.text` показывалась величина корня квадратного от чего-то. От какого значения мы хотим взять корень квадратный? Мы хотим взять корень квадратный от того, что уже находится в настоящий момент на дисплее `display!.text`.

```
@IBAction func performOperation(_ sender: UIButton) {  
    userIsInTheMiddleOfTyping = false  
    if let mathematicalSymbol = sender.currentTitle {  
        switch mathematicalSymbol {  
            case "π":  
                display!.text = String(Double.pi)  
            case "√":  
                display!.text = sqrt(display!.text!)  
            default:  
                break  
        }  
    }  
}
```

Это будет работать? Нет, потому что `display!.text` - это строка `String`, а мы не можем извлекать квадратный корень из строки. И не только, функция `sqrt` вернет нам `Double` и мы не сможем разместить `Double` в строке `String`, нам придется делать то, что мы делали только что - превратить `Double` в строку `String`:

```
@IBAction func performOperation(_ sender: UIButton) {  
    userIsInTheMiddleOfTyping = false  
    if let mathematicalSymbol = sender.currentTitle {  
        switch mathematicalSymbol {  
            case "π":  
                display!.text = String(Double.pi)  
            case "√":  
                display!.text = String(sqrt(display!.text!))  
            default:  
                break  
        }  
    }  
}
```

Но аргументом функции `sqrt` все еще является `String`. Я вытащу выражение для аргумента функции `sqrt` в локальную переменную `operand`, которая пока все еще является `String`, а очень хочет быть `Double`:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            case "√":
                let operand = display!.text!
                display!.text = String(sqrt(operand))
            default:
                break
        }
    }
}
```

Как мы сможем преобразовать **operand** в **Double**? Я думаю тем же путем, каким мы делали обратное преобразование:

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            case "√":
                let operand = Double(display!.text!)
                display!.text = String(sqrt(operand))
            default:
                break
        }
    }
}
```

Будет ли это работать? Ответ - ДА. ДА? А если строкой будет "Hello"? Мы не сможем преобразовать эту строку в **Double**. Этот инициализатор **Double**, который берет строку **String** в качестве аргумента, возвращает **Optional** число с двойной точностью - **Double?**, потому что если вы даете ему строку, которую он не может преобразовать в **Double**, то он возвращает состояние **not set** ("не установлен"). Он вам сообщает, что не может этого сделать.

Если мы посмотрим на тип переменной **operand**, то она окажется **Double?**.

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            case "√":
                let operand = Double(display!.text!)
                display!.text = String(sqrt(operand))
            default:
                break
        }
    }
}
```

А почему, когда мы преобразовывали **Double** в **String** нам возвращалась не **Optional** строка?

String нет необходимости возвращать **Optional** строку **String?**, потому что она всегда может

преобразовать **Double** в **String**, но не всегда можно преобразовать строку **String** в **Double**.

Я собираюсь принудительно “развернуть” **Double?** - в конце ставлю восклицательный “!” знак, предполагая, что на моем дисплее **display** никогда не появится того, что нельзя будет преобразовать в **Double**. Может быть это плохое предположение, но на данный момент я буду предполагать именно это.

```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            case "√":
                let operand = Double(display!.text!)!
                display!.text = String(sqrt(operand))
            default:
                break
        }
    }
}
```

И опять используем **CTRL + I**, чтобы выровнять все отступы.

Теперь все в порядке. Переменная **operand** имеет тип **Double**, и я могу рассчитать квадратный корень.

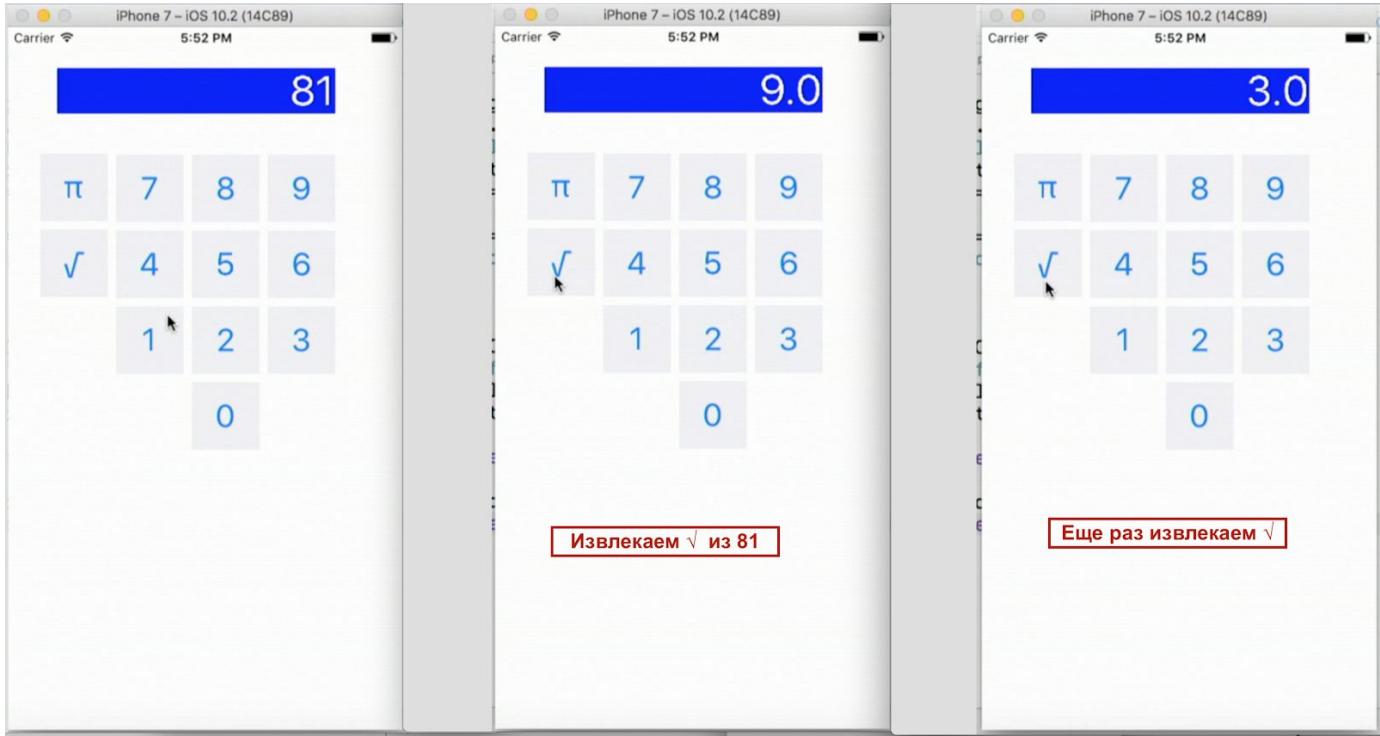
```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                display!.text = String(Double.pi)
            case "√":
                let operand = Double(display!.text!)!
                display!.text = String(sqrt(operand))
        }
    }
}
```

Declaration let operand: Double
Declared In ViewController.swift

Я преобразую результат в **String**.

Очень беспорядочный код, я собираюсь его исправить, но концептуально, надеюсь, вы поняли, что здесь происходит.

Запускаем приложение и извлекаем квадратный корень $\sqrt{81}$ из **81**, потом еще раз квадратный корень $\sqrt{\sqrt{81}}$.



Еще раз корень квадратный, теперь “ π ” и снова “ $\sqrt{\cdot}$ ” теперь уже из “ π ”. Все работает идеально.

Но наш код очень беспорядочный. Представьте, что вам нужно добавить **case** для еще одной операции и еще одной операции и т.д. И будем делать **Double (это), String (то)**, принудительно “разворачивать” с помощью восклицательного **!** знака почти всюду?

Слишком много восклицательных **!** знаков – это первое.

Второе – я устал от всех этих преобразований **String** в **Double** и **Double** в **String**. Я собираюсь исправить все это.

Давайте начнем с восклицательного **!** знака. Он появляется всякий раз, как я использую **display**. Я использую это сочетание **display!** по всему коду:

```

@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    if userIsInTheMiddleOfTyping {
        let textCurrentlyInDisplay = display!.text!
        display!.text = textCurrentlyInDisplay + digit
    } else {
        display!.text = digit
        userIsInTheMiddleOfTyping = true
    }
}

@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
        case "π":
            display!.text = String(Double.pi)
        case "√":
            let operand = Double(display!.text!)!
            display!.text = String(sqrt(operand))

        default:
            break
        }
    }
}

```

Хотя я знаю, что **display**, как только установится iOS в самом начале, уже никогда не вернется в <http://bestkora.com/> Разработка iOS приложений. CS193P Stanford Winter 2017 iOS 10 Swift 3

состояние **not set** (“не установлен”), он ВСЕГДА будет находиться в состоянии **set** (“установлен”). Давайте вернемся к вопросительному ? знаку при декларировании свойства **display**:

```
class ViewController: UIViewController {  
    @IBOutlet weak var display: UILabel? }
```

Так вот что означает восклицательный ! знак при декларировании **Optional**. Он означает: “Да, это **Optional**, то же, что и означает вопросительный ? знак, но везде, где вы будете использовать это свойство, я (восклицательный ! знак) буду автоматически его “разворачивать”.

```
class ViewController: UIViewController {  
    @IBOutlet weak var display: UILabel!
```

Теперь, когда **display** уже “развернут”, я могу везде убрать восклицательный ! знак в коде, и все будет работать, потому что **display** автоматически “разворачивается”:

```
class ViewController: UIViewController {  
    @IBOutlet weak var display: UILabel!  
  
    var userIsInTheMiddleOfTyping = false  
  
    @IBAction func touchDigit(_ sender: UIButton) {  
        let digit = sender.currentTitle!  
        if userIsInTheMiddleOfTyping {  
            let textCurrentlyInDisplay = display!.text!  
            display!.text = textCurrentlyInDisplay + digit  
        } else {  
            display!.text = digit  
            userIsInTheMiddleOfTyping = true  
        }  
    }  
}
```

Но если **display** находится в состоянии **not set** (“не установлен”), то приложение все равно завершится аварийно, потому что оно является НЕЯВНО “развернутым” (**implicitly unwrapping**). Именно поэтому тип **UILabel!** называется “неявно развернутым **Optional**” (**implicitly unwrapped Optional**). Уловили фразу? Неявно развернутое **Optional** (**implicitly unwrapped Optional**). Это означает, что мы повсюду в коде можем изменить **display!** с восклицательным знаком на **display** без восклицательного знака:

```

class ViewController: UIViewController {
    @IBOutlet weak var display: UILabel!
    var userIsInTheMiddleOfTyping = false
    @IBAction func touchDigit(_ sender: UIButton) {
        let digit = sender.currentTitle!
        if userIsInTheMiddleOfTyping {
            let textCurrentlyInDisplay = display.text!
            display.text = textCurrentlyInDisplay + digit
        } else {
            display.text = digit
            userIsInTheMiddleOfTyping = true
        }
    }
    @IBAction func performOperation(_ sender: UIButton) {
        userIsInTheMiddleOfTyping = false
        if let mathematicalSymbol = sender.currentTitle {
            switch mathematicalSymbol {
            case "π":
                display.text = String(Double.pi)
            case "√":
                let operand = Double(display.text!)
                display.text = String(sqrt(operand))
            default:
                break
            }
        }
    }
}

```

Это сделало наш код более понятным.

Но все еще мы вынуждены “разворачивать” `text`. Уж извините. Мы вынуждены делать это, так как `text` не является “неявно развернутым **Optional**” (**implicitly unwrapped Optional**).

А как насчет **Double** и **String**?

Представьте себе, что у вас есть переменная `var` с именем `displayValue` и она - **Double**. Эта переменная отслеживает все, что появляется на дисплее как **Double**:

```

@IBOutlet weak var display: UILabel!
var userIsInTheMiddleOfTyping = false
@IBAction func touchDigit(_ sender: UIButton) {
    let digit = sender.currentTitle!
    if userIsInTheMiddleOfTyping {
        let textCurrentlyInDisplay = display.text!
        display.text = textCurrentlyInDisplay + digit
    } else {
        display.text = digit
        userIsInTheMiddleOfTyping = true
    }
}
var displayValue: Double
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
        case "π":
            display.text = String(Double.pi)
        case "√":
            let operand = Double(display.text!)
            display.text = String(sqrt(operand))
        default:
            break
        }
    }
}

```

Потому что все, что у нас есть на дисплее, мы получаем как **String**, и устанавливать можем только как **String**. Было бы здорово, если бы у меня была переменная **var**, которая возвращала бы мне то, что находится на дисплее, но только как **Double**. Потому что я постоянно реально нуждаюсь в содержимом дисплея в форме **Double**, и мне также необходимо устанавливать на дисплее значения **Double**.

Как я могу это делать? У Swift есть реально крутая “фишка”, которая называется вычисляемые свойства (**computed properties**). Все, что мне нужно сделать для этого - разместить код после декларирования свойства и тогда мы сможем вычислить значение вместо того, чтобы хранить его:

```
var displayValue: Double {  
    |  
    |  
    |  
    |  
}
```

----- 80 -ая минута лекции -----

Например, свойство **userIsInTheMiddleOfTyping** - это хранимое свойство, у него где-то есть хранилище. А свойство **displayValue** - вычисляемое свойство, у которого вы можете вычислять как получение значения - **{get}** случай, так и установку значения - **{set}** случай:

```
@IBOutlet weak var display: UILabel!  
  
var userIsInTheMiddleOfTyping = false  
  
@IBAction func touchDigit(_ sender: UIButton) {  
    let digit = sender.currentTitle!  
    if userIsInTheMiddleOfTyping {  
        let textCurrentlyInDisplay = display.text!  
        display.text = textCurrentlyInDisplay + digit  
    } else {  
        display.text = digit  
        userIsInTheMiddleOfTyping = true  
    }  
}  
  
var displayValue: Double {  
    get {  
        |  
        |  
        |  
    }  
    set {  
        |  
    }  
}
```

Хранимое свойство

Вычисляемое свойство

У вас может быть некоторый код как для получения значения, так и для его установки. У нас нет необходимости где-то хранить значение **displayValue**. Мы собираемся получать его и устанавливать. Откуда мы будем брать значения для **displayValue** и что будем использовать для его установки?

Из текста **text** метки **display**.

Как будет выглядеть **{get}**?

```
var displayValue: Double {  
    get {  
        return Double(display.text!)!  
    }  
    set {  
        ...  
    }  
}
```

Мы просто возвращает **Double** текста **text** на дисплее **display**, вам не нужно “разворачивать” **display**, но вам придется “разворачивать” **text** и я должен принудительно “развернуть” все выражение - **Double (display.text!)!**.

Повторюсь, я предполагаю, что всегда строку на дисплее **display** можно интерпретировать как **Double**. Может быть, по ходу дела я изменю это предположение, но сейчас я буду придерживаться этого предположения.

Теперь рассмотрим случай **{set}**?

В случае **{set}** я установлю текст на дисплее **display**, **String** эквивалент тому значению, которое люди пытаются установить для **displayValue**.

В коде это будет выглядеть так:

```
var displayValue: Double {  
    get {  
        return Double(display.text!)!  
    }  
    set {  
        display.text = String(newValue)  
    }  
}
```

Declaration let newValue: Double
Declared In ViewController.swift

Допустим где-то в коде мы написали **displayValue = 5**. На дисплей **display** мы должны вывести превращенную в строку цифру “5”. Внутри **String ()** для случая **{set}** вычисляемого свойства (**computed property**) существует специальная переменная с именем **newValue**, причем **newValue** всегда имеет тип устанавливаемого значения, потому что это правая часть равенства (например, **displayValue = 5**) при установке:

```
var displayValue: Double {  
    get {  
        return Double(display.text!)!  
    }  
    set {  
        display.text = String(newValue)  
    }  
}
```

Declaration let newValue: Double
Declared In ViewController.swift

Здорово. Теперь, когда у нас есть переменная **displayValue** и она всегда отслеживает то, что находится на дисплее **display**, мы можем существенно улучшить наш код:

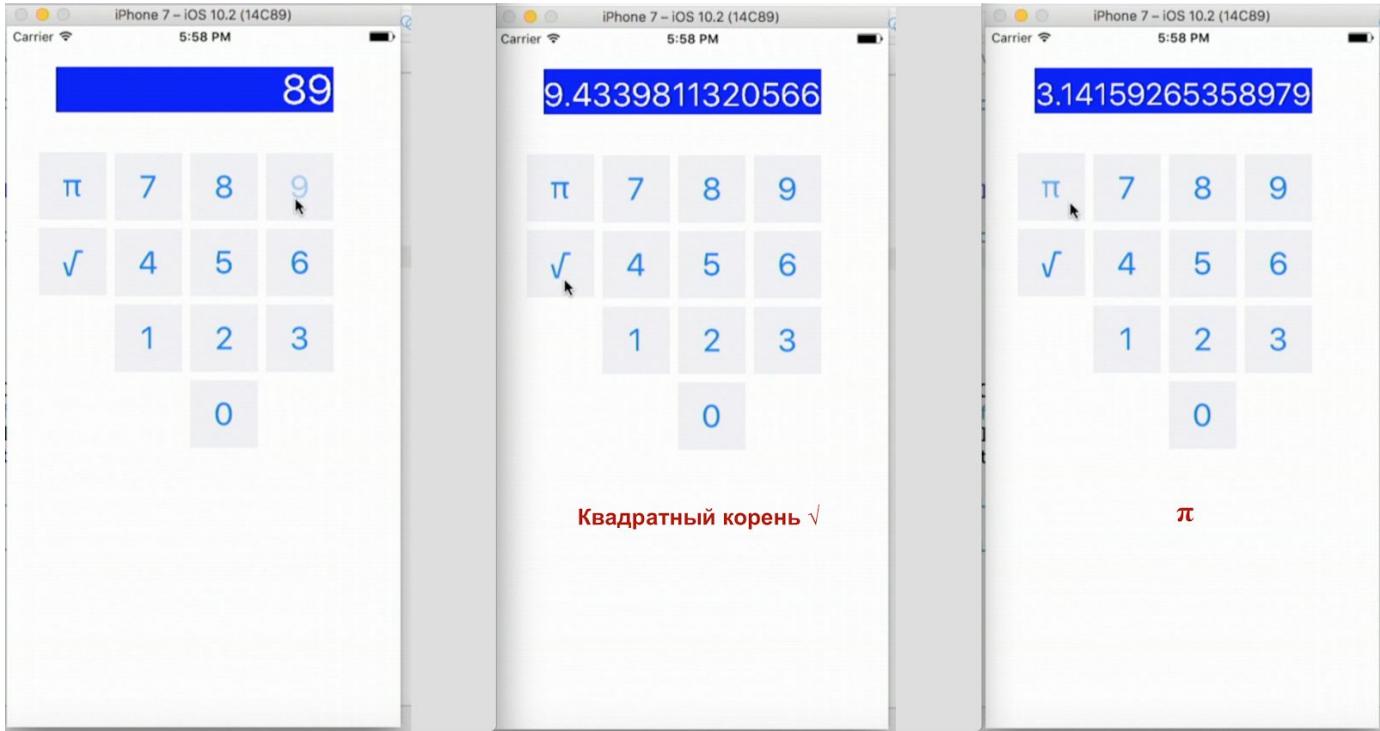
```
@IBAction func performOperation(_ sender: UIButton) {
    userIsInTheMiddleOfTyping = false
    if let mathematicalSymbol = sender.currentTitle {
        switch mathematicalSymbol {
            case "π":
                displayValue = Double.pi
            case "√":
                displayValue = sqrt(displayValue)
            default:
                break
        }
    }
}
```

Мы избавились от беспорядочного кода, и неожиданно стал очень компактным. Это минимум того, что вы можете напечатать, чтобы сказать, что вы хотите показать значение “ π ” на дисплее `display`. Когда я получаю `displayValue`, то выполняется код в `{get}`. Когда я устанавливаю `displayValue`, то выполняется код в `{set}`. Это называется вычисляемое свойство (**computed property**). Вы увидите его повсюду. Вы уже видели его. Например, `currentTitle` – это вычисляемое свойство (**computed property**), оно вычисляется классом `UIButton`:



Классом `UIButton` определяет текущий заголовок на кнопке и возвращает его. Это реализовано некоторым кодом, который и возвращает нужное значение. Как я это знаю? Потому что это **read-only** переменная `var`, которая использует вычисляемое свойство.

Давайте убедимся, что мы ничего не сломали с нашим прекрасным лаконичным кодом.



Набираю 89 - выглядит прекрасно, квадратный корень , “ π ”.

Все работает. Много материала для изучения, я знаю. Давайте все это проработаем.

В Среду вначале я расскажу об **MVC**, парадигме конструирования, а затем добавлю **MVC** в наш Калькулятор, кроме того, мы сделаем так, чтобы **UI** нашего Калькулятора работал на всех устройствах. Пока.

----- 84 -ая минута лекции -----

----- Конец лекции 1 -----

[В НАЧАЛО ЛЕКЦИИ](#)