Assignment II: Calculator Brain (Калькулятор с мозгами)

Цель:

Вы начнете это задание с включения в ваш калькулятор Calculator из Задания 1 всех изменений, сделанных на лекции (то есть CalculatorBrain). Это последнее задание, когда вам нужно печатать код с лекции, а не использовать copy / paste.

Теперь, когда мы добавили MVC Модель к нашему калькулятору Calculator, мы собираемся расширить его возможности. Мы усовершенствуем наш Calculator и позволим ему вводить “переменные” в стэк калькулятора. Кроме того, мы снабдим пользователя более понятным представлением того, что было введено в наш Calculator.

Обязательно посмотрите ниже раздел подсказок (**Hints**)!

Материалы

- У вас должно быть успешно выполненное Задание 1. Задание 2 выполняется на его основе.
- Вам необходимо посмотреть видео Лекции 3 и сделать такие же изменения с кодом вашего Задания 1. Это видео можно найти на **iTunes название “3. Applying MVC”**.

Обязательные пункты задания

1. Все изменения, сделанные с калькулятором Calculator на лекции, должны быть внесены в код вашего Задания 1. Сделайте полученный Calculator полностью функционирующим прежде, чем вы приступите к выполнению остальных обязательных заданий. И, как и в прошлый раз “печатайте” изменения в коде, а не используйте copy / paste.

2. Ничего не меняйте в non-private API в CalculatorBrain и продолжайте использовать **enum** в качестве основной внутренней структуры данных.

3. Ваш пользовательский интерфейс (UI) должен быть всегда синхронизирован с вашей Моделью (CalculatorBrain).

4. Дополнительное задание с прошлой недели превратить **displayValue** в **Double?** (то есть сделать его **Optional**, а не **Double**) теперь является обязательным. **displayValue** должен возвращать **nil**, если содержимое **display** не может интерпретироваться как **Double**. Установка **displayValue** в **nil** должна очищать **display**.

5. Наделите ваш CalculatorBrain способностью помещать “переменные” в его внутренний стэк. Сделайте это путем реализации следующего API в вашем CalculatorBrain ...

```
func pushOperand (symbol:String) -> Double?
var variableValues:Dictionary<String, Double>
```

   Это должно делать точно то, что говорят названия : первая функция “толкает” “переменную” во внутренний стэк CalculatorBrain (например, **pushOperand** (“x”) будет помещать переменную с именем x), а вторая переменная ввиде словаря позволит пользователю CalculatorBrain установить любое значение для любой переменной (например, brain.variableValues [“x”] = 35.0) . **pushOperand** должен вернуть результат **evaluate()** после помещения переменной в стэк (как pushOperand делает в других случаях).

6. Функция **evaluate()** должна использовать значение переменной (из словаря **variableValues**) всякий раз, когда “переменная” рассчитывается или вернуть **nil**, если при расчете не находится подходящего значения для этой переменной.

7. Реализуйте новую **read-only** (только **get**, нет **set**) переменную **var** для CalculatorBrain, чтобы описать содержимое “мозгов” как **String**...

`var` description: `String`

- Унарные операции должны быть показаны с использованием нотации "функция". Например, ввод **10 cos** должен отображаться в **description** как **cos (10)**
- Бинарные операции должны быть показаны с использованием инфиксной (infix) нотации. Например, ввод **3 ↲ 5 -** должен отображаться в **description** как **3 - 5**. Убедитесь, что вы получили правильный порядок чисел.
- Весь другой контент стэка (например, операнды, переменные, константы как **π** и так далее) должны быть показаны "как есть". Например **23.5** ⇒ **23.5**, **π** ⇒ **π** (не 3.1415!), "переменная" **x** ⇒ **x** (не значение) и т.д.
- Любые комбинации элементов стэка должны отображаться правильно. Например: **10 √ 3 +** ⇒ **√ (10) + 3**
  3 ↲ 5 + √ ⇒ **√ (3 + 5)**
  3 ↲ 5 √ + √ 6 ÷ ⇒ **√(3 + √ (5)) ÷ 6**
- Если есть пропущенные операнды, заменяем их на ?, например, **3 ↲ +** ⇒ **? + 3**
- Если в стэке несколько законченных выражений, разделяем их запятыми. Например, **3 ↲ 5 + √ π cos** ⇒ **√(3+5), cos(π)**. Выражения должны располагаться в историческом порядке с наиболее старыми вначале строки и с недавно **push / perform** в конце.
- Ваш **description** должен правильно представлять математическое выражение. Например, **3 ↲ 5 ↲ 4 + ×** **не должно** быть **3 × 5 + 4** оно **должно** быть **3 × (5 + 4)**. Другими словами, вы кое- где должны добавить круглые скобки вокруг бинарных операций. При этом постарайтесь минимизировать количество круглых скобок насколько это возможно (но результат должен быть математически корректным). Посмотрите дополнительное задание, если вы действительно хотите сделать это правильно.

8. Модифицируйте метку UILabel, которую вы добавили на прошлой неделе, чтобы она показывала **description** вашего CalculatorBrain. В конце разместите знак "=" и разместите метку UILabel так, чтобы **дисплей** выглядел как будто бы он является результатом этого "=" . Это знак **"**=" был дополнительным пунктом задания на прошлой неделе, но сейчас это обязательный пункт.

9. Добавьте две новых кнопки на цифровую клавиатуру вашего калькулятора Calculator: →**M** и **M**. Эти две кнопки будут соответственно устанавливать и получать переменную в CalculatorBrain, называемую M.
   - →**M** устанавливает значение переменной M в brain в текущее значение на

дисплее.

- →**M** не должна автоматичски выполнять ↲ (хотя она должна переустановить переменную "пользователь в середине печати числа")
- Нажатие **M** должно "толкать" (push) переменную M (не значение M) в CalculatorBrain
- Нажатие любой другой кнопки должно показывать результат вычисления brain (то есть результат evaluate()) на **дисплее.**
- →**M** и **M** являются механизмом Controller, а не механизмом Model (хотя они оба используют концепцию "переменных" Model).
- Это не выдающаяся кнопка "memory" на нашем калькуляторе, но она является хорошим инструментом для тестирования, правильно ли работает наша переменная функция, реализованная выше.
  Примеры …
  7 M + √ ⇒ description имеет вид **√(7+M)**, **display** - пустой, так как **M** не установлена
  9 →M ⇒ display показывает 4 (квадратный корень 16), **description** все еще показывает **√(7+M)**
  14 + ⇒ display показывает 18, **description** теперь **√(7+M)+14**

10 . Убедитесь, что кнопка **C** вашего Задания 1 работает правильно в этом задании.

11 . Когда вы нажимаете кнопку **C**, переменная M должна быть убрана из **variableValues Dictionary** в CalculatorBrain (не установлена в 0 или какое-то другое значение). Это позволит вам тестировать случай "неустановленной" переменной (потому что это заставит **evaluate ()** вернуть **nil**, и **дисплей** вашего калькулятора будет пустым, если **M** будет использоваться без →**M**).

12 . Ваш UI должен выглядеть хорошо на iPhone любого размера как в портретном, так и в ландшафтном режиме (не беспокойтесь относительно iPad до следующей недели). Это означает, что вы должны сделать правильные установки Autolayout (Разметки) и больше ничего.

Подсказки (Hints)

1. Рассмотрите возможность использования **Optional chaining** (Optional цепочек) при реализации **displayValue**.

2. Теперь, когда ваш **displayValue** стал лучше, убедитесь, что вы его правильно используете всюду в вашем **ViewController**.

3. Если вы на прошлой неделе реализовали **π** как простое “выталкивание” в стэк M_PI, вам, возможно, придется улучшить CalculatorBrain чтобы уметь “выталкивать” в стэк “константы” так, чтобы ваш **description** показывал **π**, а не 3.1415926...

4 . Возможно вы не захотите реализовывать **π** как переменную с именем **π** и значением 3.1415926… Почему нет? Потому что программисты, использующие ваш CalculatorBrain могут очистить все значения переменных (для своих собственных целей) и могут потом удивиться, что **π** не имеет значения.

5. Если вас все еще смущает, что делать с **π**, не заморачивайтесь. Это просто очередная **known Op**. Вы можете улучшать **Op** как вам нужно, чтобы оно воспринимало константы подобные **π**.

6. Когда вы очищаете ваш **display**, разместите “ “ (пробел) , не **nil** или “” (пустая строка), в противном случае ваша **UILabel** “сожмется” по вертикале, смещая весь ваш пользовательский интерфейс немного вверх, а когда появится текст - вниз.

7. Очевидно, что **evaluate ()** очень похожа по своей функциональности на новую переменную **description**, которую вас просят реализовать (они обе используют вспомогательные рекурсивные методы), так что воспринимайте это как руководство к действию.
8. Возможно, что часть “Если в стэке несколько законченных выражений, разделяем их запятыми” , которую нужно выполнить для **description,** лучше реализовать внутри get для **var description** (а не в его рекурсивном вспомогательнои методе).

9. Вы можете убрать весь код, связанный с **history** меткой (которая размещает все, что ввел пользователь в стэк) из Задания 1, так как теперь у вас новый API в CalculatorBrain, который будет показывать это в более удобной форме. Но может быть предложено очень элегантное решение, в котором находится единственный “узел” в вашем ViewController, через который и адаптируется эта **UILabel** (но это

просто подсказка, а не обязательное требование).

10. Вам следует реализовать **@IBAction** методы для **M** и → **M** с двумя - тремя строками кода каждый. Но это не обязательное задание, а просто подсказка.

11. Не забудьте подумать о вашем CalculatorBrain как о повторно используемом классе: будет более гибко разрешить использовать программистам public API для раздельного очищения brain стэка и значений переменных из brain (хотя ваша кнопка **C** делает оба этих действия).

12. Ваш UI всегда должен быть синхронизован с вашей Model (CalculatorBrain). Есть много "подводных камней" при реализации этого (например, **C**, **M**, → **M**, и т.д). Будьте внимательны.

13. Если ваш Autolayout (разметка) все перемешала, у вас есть конфликтующие ограничения (constraints) и т.д., то начните все сначала. С того, что уберите все ограничения на вашем scene (используйте кнопку в нижнем правом углу редактора storyboard), переместите ваши views туда, где они выглядят хорошо (используя голубые пунктирные линии) , установите ограничения (constraints) на ваши метки **UILabels** (с помощью **CTRL**-перетягивания к сторонам и к верхней границе контейнера и использования Инспектора Размера (Size Inspector) для их редактирования, если нужно), затем выделите (selecting) все кнопки **UIButtons** и (используя кнопку **Pin** в нижнем правом углу редактора storyboard) примените ограничения (constraints), которые вы хотите, чтобы исполнялись одновременно для всего, что вы выделили (разные размеры **Eqiale Sizes** , особенно зазоры **Spacing**). Наконец, откройте Схему UI (**Document Outline**) и посмотрите, есть ли какие-то предупреждения или ошибки в ваших ограничениях (constraints), и исправьте их, кликая на желтые и красные символы.

14. **Autolayout** - это все о **голубых пунктирных линиях**. Без них Xcode будет трудно понять, что вам нужно.

Что нужно изучать

Это частичный список концепций, в которых это задание увеличивает ваш опыт работы или продемонстрирует ваши знания.

- Optionals
- Closures
- enum
- switch
- Dictionary
- Tuples
- Autolayout
- Рекурсия (не iOS вещь, но нужно знать!)

Оценка (Evaluation)

Во всех заданиях требуется написание качественного кода, на основе которого строится приложение без ошибок и предупреждений (without warnings or errors), следовательно вы должны тестировать полученное приложение (application) до тех пор, пока оно не начнет функционировать правильно согласно поставленной задачи.

Приведем наиболее общие соображения, по которым задание может быть отклонено (marked down):

- Приложение не создается (Project does not build).
- Приложение не создается без предупреждений (Project does not build without warnings).
- Один или более пунктов в разделе **Обязательные пункты задания** не выполнены.

- Не понята фундаментальная концепция задания (A fundamental concept was not understood).
- Код - небрежный или тяжелый для чтения (например, нет отступов и т.д.).
- Ваше решение тяжело (или невозможно) прочитать и понять из-за отсутствия комментариев, из-за плохого наименования методов и переменных, из-за непонятной структуры и т.д.

Часто студенты спрашивают: “Сколько комментариев кода нужно сделать?” Ответ - ваш код должен легко и полностью быть понятным любому, кто его читает. Вы можете предполагать, что читатель знает SDK, но вам не следует предполагать, что он уже знает решение проблемы.

Дополнительные задания (Extra Credit)

Мы постарались создать Дополнительные задания так, чтобы они расширили ваши познания в том, что мы проходили на этой неделе. Попытка выполнения по крайней мере некоторых из них приветствуется в плане получения максимального эффекта от этого курса. Есть несколько подсказок для выполнения дополнительных заданий на следующей странице

1. Сделайте так, чтобы ваша **description** имела как можно меньше круглых скобок для бинарных операций.

2. Добавьте “Undo” к вашему калькулятору: в дополнительном пункте Задания 1 вы добавляли кнопку “backspace”, если пользователь ввел неверную цифру. Теперь мы говорим о комбинации “backspace” и реального “Undo” в единственной кнопке. Если пользователь находится в середине ввода числа, то это кнопка работает как “backspace”. Если пользователь не находится в середине ввода числа, то должно выполняться Undo последней вещи, которая была выполнена в CalculatorBrain.

3. Добавьте новый метод, **evaluateAndReportErrors()**. Он должен работать как **evaluate()** за исключением того, что если возникают проблемы с вычислением стэка (не установлены значения переменных или пропущены операнды, деление на 0, квадратный корень из отрицательного числа и т.д.), то вместо возвращения nil, он возвращает String с тем, что собой *представляет проблема* (если проблем много, то вы можете вернуть любую из них, но одну). Покажите все такие ошибки на **display** вашего калькулятора (вместо того, чтобы делать его пустым или показывать какие-то странные значения). Вы все равно должны реализовать **evaluate()** как определено в Обязательных пунктах задания, но если хотите, у вас может быть **evaluate() ,** возвращающий **nil**, если имели место любые ошибки (не просто “не установлено значение” или “недостаточно операндов”). Методы **push** и **perform** все еще возвращают **Double?** (на первый взгляд это бессмысленно, но мы хотим иметь возможность оценить Обязательные и Дополнительные пункты Задания раздельно).

Подсказки к дополнительным пунктам (Extra Credit Hints)

Здесь представлены некоторые ссображения, как подойти к решению к дополнительных пунктов Задания 2.

1. Убрать лишние круглые скобки
   a. Это потребует добавления концепции “приоритета (precedence) операций” в **Op** вашего CalculatorBrain
   b. Подобно **var** description в **Op** , **var** “precedence” могла бы возвращать значение по умолчанию для большинства **Ops**, но для бинарных операций какое-то специальное ассоциированное значение. Приоритет “переменных”, унарных операций, констант, операндов - один и тот же, то есть насколько возможно наивысший.
   c. Наверно, было бы уместно использовать **Int** для представления приоритета (наибольшее значение означает наибольший приоритет). В этом случае наивысший приоритет был бы **Int.max**.
   d. Приоритет влияет только на **description** стэка (другими словами , это дополнительная информация, которая нужна **description** чтобы знать, как оптимизировать описание стэка). Это не воздействует на то, как будет проходить *вычисление* (**evaluate()**) стэка. Приоритет *вычисления* определяется порядком расположения элементов в стэке.
2. **Undo**
   a. Посмотрите Дополнительный пунк в Задании 1, связанный с “backspace” и подсказки к нему, и проверьте, что он работает.
   b. **Backspace** / **Undo не должен** убирать установки **M** переменной. Это позволит пользователю использовать калькулятор для вычисления значения **M**, которое он хочет, затем сделать **Undo** и получить назад выражение, над которым он работал, используя **M**.
   c. Возможно вам придется добавить некоторый новый non-private API в ваш CalculatorBrain (хотя реализация его потребует одной- двух строк кода).
   d. Если вы уже имеете ясный код для этого пункта Задания, то я хочу вам сказать, что есть возможность выполнить этот пунк с помощью единственного метода в **ViewController** и полдюжины (12) строк кода или меньше. Если это потребует больших изменений, то попытайтесь понять почему и попытайтесь достичь этого в первую очередь с помощью реорганизации вашего кода.
3. Сообщения об ошибках

a. Какая структура данных прекрасно подходит для ситуации “или- или” и которую мы уже использовали?
b. Сообщение об ошибках не должно влиять на ваш **description**.
c. Необходимо, чтобы **evaluateAndReportErrors()** умела спрашивать операции **BinaryOperation** и **UnaryOperation,** какие ошибки (если они есть) будут генерироваться при передачи в них операнда (ов).
d. Один из способов сделать это - иметь ассоциированное значение для **BinaryOperation** и **UnaryOperation** , которое представляют собой *функцию*, анализирующую потенциальные аргументы и возвращающую соответствующее **String** сообщение об ошибке, если выполнение операции генерирует ошибку (или **nil** в противном случае). Есть очень много и других способов выполнить этот пункт, но это ведь секция подсказок в конце концов.
e. Большинство операций не могут сообщать о каких-то ошибках. Вам нужно облегчить создание таких операций. Если вы использовали подсказку, приведенную выше, то передача **nil** в качестве функции, тестирующей ошибки, будет наиболее простым способом сделать это. Вы можете сделать **Optional** целиком **тип функции**, размещая **тип функции** в круглых скобках и ставя знак **?** после круглых скобок. Например, **((Double, Double) -> String?)?** - это **Optional** функция, которая берет два **Double** и возвращает **Optional String?**.
f. Не забывайте использовать синтаксис “цепочек” **Optionals**. Например, вы можете использовать **Optional** функцию с именем **errorTest** следующим образом…

```
if let failureDescription = errorTest?(arguments) { }
```

… и здесь интересны два варианта:
первый - если она ошибочная, то есть сама функция **errorTest** -это **nil**,
второй - функция **errorTest** не **nil**, но она декларирована так, что возвращает **nil**. Это очень удобно.

g. В вашем **ViewController** вы, возможно, обнаружите, что реализация нового свойства **var displayResult** (которое обрабатывает как значения, так и ошибки) и реорганизация старой переменной **displayValue** в терминах этой новой переменной **var displayResult**, сделает ваш код более понятным. Или нет. Но это только подсказка.
h. Четыре главных тестовых случая нужно проверить : для операции ÷ (деление на 0), для √ (отрицательное число), “не достаточно операндов”, “переменная x не установлена”.
i. Если вы смотрите на проблему сообщений об ошибках в контексте шаблона

проектирования MVC, то хорошим аргументом будет представление ошибки “кодом” в Модели (**M** - часть в MVC) и передача “кода” Controller (**C** - часть в MVC). Controller будет реальным “ответственным” за представление ошибки на View (**V** - часть в MVC). Например, Controller может представлять ошибку на родном языке пользователя. Но для упрощения выполнения этого дополнительного пункта вы можете просто вернуть ошибку как **String** из Model и показывать пользователю эту **String** напрямую. Но между прочим, можно возразить, что поскольку **Strings** с ошибками - non-**private** в **CalculatorBrain**, то их вполне можно рассматривать как “коды ошибок” (которые могли бы быть транслированы Controller во что угодно).