Assignment 1: Calculator

Objective

The goal of this assignment is to recreate the demonstration given in lecture and then make some small enhancements. It is important that you understand what you are doing with each step of recreating the demo from lecture so that you are prepared to do the enhancements.

Another goal is to get experience creating a project in Xcode and typing code in from scratch. Do not copy/paste any of the code from anywhere. Type it in and watch what Xcode does as you do so.

This assignment must be submitted using the submit script described here by the start of lecture next Wednesday (i.e. before lecture 3). You may submit it multiple times if you wish. Only the last submission before the deadline will be counted. For example, it might be a good idea to go ahead and submit it after you have reproduced what was shown in lecture and gotten that part working (even before you attempt the enhancements). If you wait until the last minute to try to submit and you have problems with the submission script, you'll likely have to use one of your valuable free late days.

Be sure to review the Hints section below!

Also, check out the latest in the Evaluation section to make sure you understand what you are going to be evaluated on with this assignment.

Materials

- You will need to install the (free) program Xcode 8 using the App Store on your Mac (previous versions of Xcode will NOT work). It is highly recommended that you do this immediately so that if you have any problems getting Xcode to work, you have time to get help from Piazza and/or the TAs in their office hours.
- A link to the video of the lectures can be found in the same place you found this document.

Required Tasks

1. Get the Calculator working as demonstrated in lectures 1 and 2.
2. Your Calculator already works with floating point numbers (e.g. if you touch `3 ÷ 4 =`, it will properly show 0.75), however, there is no way for the user to *enter* a floating point number directly. Fix this by allowing *legal* floating point numbers to be entered (e.g. "`192.168.0.1`" is **not** a legal floating point number!). You will need to have a "`.`" button in your calculator. Don't worry too much about precision or significant digits in this assignment (doing so is Extra Credit).
3. Add some more operations buttons to your calculator such that it has at least a dozen operations total (it can have even more if you like). You can choose whatever operations appeal to you. The buttons must arrange themselves nicely in portrait and landscape modes on all iPhone 6's and 7's.
4. Use color to make your UI look nice. At the very least, your operations buttons must be a different color than your keypad buttons, but otherwise you can use color in whatever way you think looks nice.
5. Add a `Bool` property to your `CalculatorBrain` called `resultIsPending` which returns whether there is a binary operation pending.
6. Add a `String` property to your `CalculatorBrain` called `description` which returns a description of the sequence of operands and operations that led to the value returned by `result` (or the result so far if `resultIsPending`). The character = (the equals sign) should never appear in this `description`, nor should … (ellipses).
7. Implement a `UILabel` in your UI which shows the sequence of operands and operations that led to (or is leading to if `resultIsPending`) what is (or "will be" if `resultIsPending`) showing in the `display`. If `resultIsPending` is `true`, put . . . on the end of the `UILabel`, else put =. If the `userIsInTheMiddleOfTyping`, you can leave the `UILabel` showing whatever was there before the user started typing the number. Examples …
   a. touching 7 + would show "7 + …" (with 7 still in the display)
   b. 7 + 9 would show "7 + …" (9 in the display)
   c. 7 + 9 = would show "7 + 9 =" (16 in the display)
   d. 7 + 9 = √ would show "√(7 + 9) =" (4 in the display)
   e. 7 + 9 = √ + 2 = would show "√(7 + 9) + 2 =" (6 in the display)
   f. 7 + 9 √ would show "7 + √(9) …" (3 in the display)
   g. 7 + 9 √ = would show "7 + √(9) =" (10 in the display)
   h. 7 + 9 = + 6 = + 3 = would show "7 + 9 + 6 + 3 =" (25 in the display)
   i. 7 + 9 = √ 6 + 3 = would show "6 + 3 =" (9 in the display)
   j. 5 + 6 = 7 3 would show "5 + 6 =" (73 in the display)
   k. 4 × π = would show "4 × π =" (12.5663706143592 in the display)

8. Add a `C` button that clears everything (your `display`, the new `UILabel` you added above, any pending binary operations, etc.). Ideally, this should leave your Calculator in the same state it was in when you launched it.

Hints

1. The `String` method `contains(String)` might be of great use to you for the floating point part of this assignment.
2. The floating point requirement can be implemented in a single line of code (a curly brace on a line by itself is not considered a "line of code"). Note that what you are reading right now is a Hint, not a Required Task.
3. Be careful to test (and nicely handle) the case where the user starts off entering a new number by touching the decimal point (i.e. the user touches the decimal point button while `userIsInTheMiddleOfTyping` is `false`).
4. Economy is valuable in coding. The easiest way to ensure a bug-free line of code is not to write that line of code at all. This entire assignment (not including Extra Credit) can be done in a few dozen lines of code (probably even two dozen), so if you find yourself writing more than 100 lines of code, you are almost certainly on the wrong track.
5. You can use any Unicode characters you want as your mathematical symbols for your operations. For example, x^2 and x^{-1} are perfectly fine mathematical symbols.
6. If you set a `UILabel`'s `text` to `nil` or an empty string, it will resize to have zero height (shifting the rest of your UI around accordingly). You may find this disconcerting for your users. If you want a `UILabel` to appear empty, but not be zero height, simply set its `text` to be `" "` (space).
7. You might want to add a call to `performPendingBinaryOperation()` in your `case binaryOperation` too (that way 6 x 5 x 4 x 3 = will work).
8. If you are worried that you might have broken something with your changes, keep a version of the Calculator that contains only the code from lecture as a reference and compare its results against the results of your updated Calculator.
9. There is no Required Task that requires you to deal with error-creating input (e.g. divide by zero or square root of a negative number). You can assume we won't try these while grading.
10. A simple way to think of the `description` of the `CalculatorBrain` is as a `String` representation of what led to what is in the `accumulator`. If you think of it this way, then all you have to do is to create that `String` representation every single time you set the value of the `accumulator` to something. The only "gotcha" is when you have a `pendingBinaryOperation` (so you'll have to handle that case specially).
11. If you use the approach in the Hint above to implement `description`, then you will find yourself with two variables (the `accumulator` and its `String` representation) whose values are always set at the same time and which never want to be out of sync with each other. Swift has a great data structure for variables that always go together like this: a tuple. We haven't covered this, but consider using it if you can figure it out.

12. Here's another thing we didn't cover that you can try to figure out: use the `??` operator (defaulting the value of an optional) at least once in your solution.

Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. Xcode
2. Swift
3. Target/Action
4. Outlets
5. `UILabel`
6. `UIViewController`
7. Functions and Properties (instance variables)
8. Computed vs. Stored properties
9. `let` versus `var`
10. `enum`, `struct` and `class`
11. Optionals
12. `String` and `Dictionary`
13. tuples (hopefully)
14. `??` operator (hopefully)

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Project does not build without warnings.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.
- The Swift API Design Guidelines were not properly adhered to.
- Failure to make private API private using the `private` keyword.

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the iOS API and knows how the Calculator code from lectures 1 and 2 works, but should not assume that they already know your (or any) solution to the assignment.

Extra Credit

We try to make Extra Credit be opportunities to expand on what you've learned this week. Attempting at least some of these each week is highly recommended to get the most out of this course. How much Extra Credit you earn depends on the scope of the item in question. In this first assignment, these are all pretty easy, so don't expect a ton of Extra Credit for them (they're warmups for future assignments).

1. Implement a "backspace" button for the user to touch if they hit the wrong digit button. This is not intended to be "undo," so if the user hits the wrong *operation* button, he or she is out of luck! It is up to you to decide how to handle the case where the user backspaces away the entire number they are in the middle of typing. You will probably find the Strings and Characters section of the Swift Reference Guide to be helpful here.
2. Figure out from the documentation how to use the iOS struct `NumberFormatter` to format your `display` so that it only shows 6 digits after the decimal point (instead of showing all digits that can be represented in a `Double`). This will eliminate (or at least reduce) the need for Autoshrink in your `display`. While you're at it, make it so that numbers that are integers don't have an unnecessary ".0" attached to them (e.g. show "4" rather than "4.0" as the result of the square root of sixteen). You can do all this for your `description` in the `CalculatorBrain` as well.
3. Make one of your operation buttons be "generate a random double-precision floating point number between 0 and 1". This operation button is not a constant (since it changes each time you invoke it). Nor is it a unary operation (since it does not operate on anything). Probably the easiest way to generate a random number in iOS is the global Swift function `arc4random()` which generates a random number between 0 and the largest possible 32-bit integer (`UInt32.max`). You'll have to get to double precision floating point number from there, of course.