Assignment I: 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.

Materials

• You will need to install the (free) program Xcode using the App Store on your Mac.

Required Tasks

Done 1. Get the Calculator working as demonstrated in lectures 1 and 2. The Autolayout portion at the end of the lecture is extra credit, but give it a try because getting good at autolayout requires experience. If you do not do autolayout, be sure to position everything so that it is visible on all iPhones (i.e. the upper left corner of the scene).

Done 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 have to add a new "`.`" button to your Calculator. Don't worry too much about precision or significant digits in this assignment.

Done 3. Add the following operations to your Calculator:

   a. `sin`: calculates the sine of the top operand on the stack

   b. `cos`: calculates the cosine of the top operand on the stack

   c. `π`: calculates (well, conjures up) the value of π. For example, `3 π ×` should put three times the value of π into the display on your calculator. Ditto `3 ↲ π x` and also `π 3 ×`.

Done 4. Add a `UILabel` to your UI which shows a history of every operand *and operation* input by the user. Place it at an appropriate location in your UI.

Done 5. Add a `C` button that clears everything (your `display`, the new `UILabel` you added above, etc.). The Calculator should be in the same state as it is at application startup after you touch this new button.

6. Avoid the problems listed in the Evaluation section below. This list grows as the quarter progresses, so be sure to check it again with each assignment.

Hints

Don't need 1. The String method `rangeOfString(substring: String)` might be of great use to you for the floating point part of this assignment. It returns an Optional. If the passed String argument cannot be found in the receiver, it returns `nil` (otherwise don't worry about what it returns for now).

Don't need 2. The floating point requirement can probably be implemented in a single line of code. Note that what you are reading right now is a Hint, not a Required Task. Still, see if you can figure out how to implement it in a single line (a curly brace on a line by itself is not considered a "line of code").

Done 3. Be careful of the case where the user starts off entering a new number by touching the decimal point, e.g., they want to enter the number `.5` into the calculator. It might well be that your solution "just works" but be sure to test this case.

Done 4. `sin()` and `cos()` are standard functions that are available in Swift to perform those operations for you.

Done 5. The value of π is available via the expression `M_PI`. E.g. `let x = M_PI`.

Done 6. You can think of π as an *operand* or you can think of it as an *operation* (i.e. a new kind of operation that takes no arguments off the stack but returns a value). Up to you. But, either way, it'd be nice to be able to add other constants to your Calculator with a minimum of code.

Done 7. You might want to place the C button somewhere other than in the grid that includes your keypad and operation buttons (since there's no room for it there). This may challenge your autolayout skills if you attack the autolayout credit item.

Done 8. 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 assignment requires very, very few lines of code, so if you find yourself writing dozens of lines of code, you are on the wrong track.

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. Classes
8. Functions and Properties (instance variables)
9. `let` versus `var`
10. Optionals
11. Computed vs. Stored properties
12. `String` and `Array`
13. Autolayout (extra credit)

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.
- Project does not build without warnings.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- 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.

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 SDK, but should not assume that they already know the (or a) solution to the problem.

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.

Done 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, but having the `display` go completely blank is probably not very user-friendly.

   You might find the global functions `countElements` and `dropLast` to be a great help with this. Both can take a `String` as their only argument. The first one is what you would generally think of as "length" and the other drops the last character from the `String` (and returns the result of doing so). You might be kind of weirded out if you alt-click on these functions. The types of the arguments and return values would require some significant explanation which there is not room for here but, in short, `String` is actually a collection of characters that can be indexed into and sliced into sub-collections of characters. That is why `countElements` (which takes a collection) and `dropLast` (which takes a sliceable thing) work on `String`.

Done 2. When the user hits an operation button, put an `=` on the end of the `UILabel` you added in the Required Task above. Thus the user will be able to tell whether the number in the Calculator's `display` is the result of a calculation or a number that the user has just entered. Don't end up with multiple occurrences of `=` in your `UILabel`.

Done 3. Add a `+/-` operation which changes the sign of the number in the `display`. Be careful with this one. If the user is in the middle of entering a number, you probably want to change the sign of that number and allow typing to continue, not force an enter like other operations do. On the other hand, if the user is not in the middle of typing a number, then this operation would work just like any other unary operation (e.g. `cos`).

Done 4. Change the computed instance variable `displayValue` to be an Optional `Double` rather than a `Double`. Its value should be `nil` if the contents of `display.text` cannot be interpreted as a `Double` (you'll need to use the documentation to understand the `NSNumberFormatter` code). *Setting* its value to `nil` should clear the `display` out.

Done 5. Use Autolayout to make your calculator look good on all different kinds of iPhones in both Portrait and Landscape orientations (don't worry about iPads for now). Just like we used ctrl-drag to the edges of our scene to position `display`, you can ctrl-drag between your `UILabel`s (and/or your C button) to fix their vertical/horizontal spacing relative to each other. Use the blue gridlines! It's probably a good idea to reset all of your autolayout (via the button in lower right corner), then use ctrl-drag to add constraints to things that are not part of the grid of keypad and operation buttons, then use the buttons in the lower right to lay out those (after you've moved them in to place relative to your UILabel(s), etc., using dashed blue lines, of course!).