Assignment I: Calculator

Objective

This assignment has two parts. The goal of the first part of the assignment is to recreate the demonstration given in the second lecture. Not to worry, you will be given very detailed walk-through instructions in a separate document. It is important, however, that you understand what you are doing with each step of that walk-through because the second part of the assignment (this document) is to add a few extensions to your calculator which will require similar steps to those taken in the walk-through.

This assignment must be submitted using the submit script (see the class website for details) by the end of the day next Wednesday. You may submit it multiple times if you wish. Only the last submission will be counted. For example, it might be a good idea to go ahead and submit it after you have done the walk-through and gotten that part working.

Be sure to check out the Hints section below!

Materials

- By this point, you should have been sent an invitation to your Stanford e-mail to join the iPhone University Developer Program. You must accept this invitation and download the iPhone SDK.

  It is critical that you get the SDK downloaded and functioning as early as possible in the week so that if you have problems you will have a chance to talk to the TA's and get help. If you wait until the weekend (or later!) and you cannot get the SDK downloaded and installed, it is unlikely you'll finish this assignment on time.

- The walkthrough document for the first part of the assignment can be found on the class website (in the same place you found this document).

Required Tasks

1. Follow the walk-through instructions (separate document) to build and run the calculator in the iPhone Simulator. Do not proceed to the next steps unless your calculator builds without warnings or errors and functions as expected.
2. Your calculator already works with floating point numbers (e.g. if you press 3 / 4 = it will properly show the resulting value of 0.75), however, there is no way for the user to *enter* a floating point number. Remedy this. Allow only legal floating point numbers to be entered (e.g. "192.168.0.1" is not a legal floating point number).
3. Add the following four single-operand operators:
    - 1/x : inverts the number in the display (i.e. 4 becomes .25). Be sure to handle the case where the display currently contains a zero. You can fail silently, but don't crash.
    - +/-: changes the sign of the number in the display.
    - sin : calculates the sine of the number in the display.
    - cos : calculates the cosine of the number in the display.
4. Add the following three "memory" buttons:
    - Store : stores the current value of the display into a memory location. This button should *not* change the number that is in the display.
    - Recall : recalls the value in memory.
    - Mem + : adds the current value of the display to whatever's already in memory. This button should *not* change the number that is in the display.
5. Add a "C" button that clears everything (the display, any "waiting" operations, and the memory).
6. Avoiding the problems listed in the **Evaluation** section below is part of the required tasks of every assignment. This list grows as the quarter progresses, so be sure to check it again with each assignment.

Hints

These hints are not required tasks. They are completely optional. Following them may make the assignment a little easier (no guarantees though!).

1. Be careful when copying and pasting buttons in Interface Builder because the target/action of the button will be copied as well.
   - Sometimes this is what you want (e.g. if you're creating some more operation buttons from an existing operation button).
   - Sometimes it is not what you want (if you're erroneously copying and pasting a digit button to try and make an operation button).
   - Buttons can have multiple target/action pairs, so if you copy a digit button to make an operation button and then control-drag a connection from the new button to **File's Owner** (your `CalculatorViewController`) and hook it up to `operationPressed:`, then when you press that button *both* `digitPressed:` *and* `operationPressed:` will be sent. This is most definitely *not* what you want.
   - To disconnect an unwanted target/action, control-right-mouse on the sending object in **Interface Builder**. You will get a list of all the actions it sends (and on what event it sends it). Just scroll down to the unwanted one and click the little **X** to disconnect it.
2. There's an `NSString` method which you might find quite useful for doing the floating point part of this assignment. It's called `rangeOfString:` Check it out in the documentation. It returns an `NSRange` which is just a normal C struct which you can access using dot notation. For example, consider the following code:

```
NSString *greeting = @"Hello There Joe, how are ya?";
NSRange range = [greeting rangeOfString:@"Joe"];
if (range.location == NSNotFound) { … } // no Joe!
```

3. Non-object comparisons use `==` (double equals), not `=` (single equals). A single equals means "assignment." A double equals means "test for equality." See the last line of code above. *Object* comparisons for equality usually use the `isEqual:` method. Comparing *objects* using `==` is dangerous. `==` only checks to see if the two pointers are the same (i.e. they point to exactly the same instance of an object). It does not check to see if two different objects are semantically the same.
4. Don't forget that `NSString` constants start with `@`. See the `greeting` variable in the code fragment above. Constants without out the `@` (e.g. `"hello"`) are `const char *` and are rarely used on the iPhone.

5. Be careful of the case where the user *starts off* entering a new number by pressing the decimal point, e.g., they want to enter the number ".5" into their calculator. Handle this case properly.
6. `sin()` and `cos()` are functions in the normal BSD Unix C library. Feel free to use them to calculate sine and cosine.
7. When you add a single-operand operator, be careful of where you put your `if`'s, `else`'s and `{}`'s.
8. To clear an object pointer (instance variable, for example) to "nothing," set it to `nil`. Sending a message to `nil`, by the way, does nothing (e.g. it won't crash your application).
9. The three memory buttons and the clear button can all simply be treated like single-operand operations. Creating a bunch more methods to handle all those cases is a waste of code on your part and on the part of any object who wants to use the `CalculatorBrain`'s public API. These functions operate on or assign the current operand just like any other single-operand operation does, so there's no need to treat them differently.
10. This entire assignment can be done by adding one method (for the floating-point part), one instance variable (for the memory part), and less than 20 lines of code total (not including curly braces). Economy is valuable in coding: the easiest way to ensure a bug-free line of code is not to write the line of code at all.

Links

Most of the information you need is best found by searching in the documentation through Xcode (see the Help menu there), but here are a few links to Apple Conceptual Documentation that you might find helpful. Remember that we are going to go much more in-depth about Objective-C and the rest of the development environment next week, so don't feel the need to absorb these documents in their entirety.

- Objective-C Primer
- Introduction to Objective-C
- NSString Reference

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 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, etc.
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).

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

Here are a few ideas for some more things you could do to get some more experience with the SDK at this point in the game.

1. Make your calculator more user-friendly by putting a warning message somewhere when divide by zero or square root of a negative number happens. You could be very aggressive here and try to figure out how to use `UIAlertView`, or, more appropriate for this point in the class, just add another `UILabel` to your UI.
2. Add a `UILabel` somewhere in the user-interface which shows the contents of the `CalculatorBrain`'s memory. This will require the `CalculatorBrain` to "export" this information to your Controller via the `CalculatorBrain`'s API (i.e. a new method in its header file, `CalculatorBrain.h`).
3. Note that we have *not* asked you to implement a "clear memory (only)" button. This would be straightforward in your `CalculatorBrain` (it's just yet another single-operand operation), but it's a bit more of a challenge to your `CalculatorViewController` than the other memory operations are. Why is that? Hint: Clearing memory should have no effect on anything else the user is doing in the calculator at the time.
4. Implement a "backspace" button for the user to press if they hit the wrong digit button. This is not intended to be "undo," so if they hit the wrong operation button, they are out of luck!
5. Add a π button.
6. Implement a user-interface for choosing whether the operand to `sin()` or `cos()` is considered radians or degrees. When you call `sin(x)` in the C library, `x` is assumed to be in radians (i.e. 0 to 2π goes around the circle once), but users might want to enter 180 and press the sin button and get 0 instead of -0.8012 (which is the sine of 180 radians). You could use a `UIButton` for this and switch out the `titleLabel`'s `text` each time the `UIButton` is pressed, but a better way would be to see if you can figure out how to use a `UISwitch` by reading the documentation (if you dare!).
7. Add a `UILabel` somewhere in your user-interface which shows the state of a 2-operand operation in progress. For example, if the user hit the following buttons 3 + 4 * 5 = this extra field would show nothing until the +, at which point it would display 3 + then it would change to 7 * as soon as the * button was pressed. Like Extra Credit #2 above, you will need to augment your `CalculatorBrain`'s public API to share information about whether there is a 2-operand operation pending or not since that is currently private implementation in the `CalculatorBrain`. A cool place to put this `UILabel` is above the `display` and left-aligned.