Assignment 1: 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 couple of 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. 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 late days.

Be sure to check out the **Hints** section below!

Materials

- Before you start this assignment, you will need to download and install the iOS SDK and Xcode 4 from http://developer.apple.com or using the App Store on Mac OSX.

  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 CA'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 functions as expected and builds without warnings or errors.
2. Your calculator already works with floating point numbers (e.g. if you touch the buttons `3 Enter 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). Don't worry too much about precision in this assignment.
3. Add the following 4 operation buttons:
   - `sin` : calculates the sine of the top operand on the stack.
   - `cos` : calculates the cosine of the top operand on the stack.
   - `sqrt` : calculates the square root of the top operand on the stack.
   - π: calculates (well, conjures up) the value of π. Examples: `3 π *` should put three times the value of π into the `display` on your calculator, so should `3 Enter π *`, so should `π 3 *`. Perhaps unexpectedly, `π Enter 3 * +` would result in 4 times π being shown. **You should understand why this is the case.** NOTE: This required task is to add π as an operation (an operation which takes no arguments off of the operand stack), *not* a new way of entering an operand into the `display`.
4. Add a new text label (`UILabel`) to your user-interface which shows everything that has been sent to the `brain` (separated by spaces). For example, if the user has entered `6.3 Enter 5 + 2 *`, this new text label would show `6.3 5 + 2 *`. A good place to put this label is to make it a thin strip above the `display` text label. Don't forget to have the `C` button clear this too. All of the code for this task should be in your Controller (no changes to your Model are required for this one). You do not have to display an unlimited number of operations and operands, just a reasonable amount.
5. Add a "`C`" button that clears everything (for example, the `display` in your View, the operand stack in your Model, any state you maintain in your Controller, etc.). Make sure `3 7 C 5` results in `5` showing in the `display`. You will have to add API to your Model to support this feature.
6. If the user performs an operation for which he or she has not entered enough operands, use zero as the missing operand(s) (the code from the walkthrough does this already, so there is nothing to do for this task, it is just a clarification of what is required). Protect against invalid operands though (e.g. divide by zero).
7. 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. 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 normal C dot notation. For example, consider the following code:

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

2. You might also find the methods in `NSString` that start with the word "`substring`" or "`has`" to be valuable.
3. This is the C language, so *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 (e.g. two `NSString`s that contain the same characters). `isEqualToString:` is just like `isEqual:`, but it is implemented only by `NSString`.
3. 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 in iOS.
4. 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.
5. `sin()` and `cos()` are functions in the normal BSD Unix C library. Feel free to use them to calculate sine and cosine.
6. 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. 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.

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
- NSMutableArray 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. 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! It's up to you to decided how to handle the case where they backspace away the entire number they are in the middle of entering, but having the `display` go completely blank is probably not very user-friendly.
2. When the user hits an operation button, put an `=` on the end of the text label that is showing what was sent to the `brain` (required task #4). 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.
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 let them continue entering it, not force an `enterPressed` like other operations do. But if they are not in the middle of entering a number, then it would work just like any other single-operand operation (e.g. `sqrt`).

Screen Shots

This screen shot is for *example* purposes only. Note carefully that this section of the assignment writeup is **not** under the Required Tasks section. In fact, screen shots like this are included in assignment write-ups only at the request of past students and over objections by the teaching staff. Do not let screen shots like this stifle your creativity!