Assignment II: Calculator Brain

Objective

In this assignment, you're going to push your calculator's capabilities a bit further by allowing a "variable" as an input to the calculator and you will support undo. You'll also be preparing your calculator for next week's assignment as well.

This assignment must be submitted using the submit script described here by the start of lecture next Wednesday (i.e. before lecture 5). You may submit it multiple times if you wish. Only the last submission before the deadline will be counted.

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 have successfully completed Assignment 1. This assignment builds on that.

Required Tasks

1. Do not change, remove or add any public API (i.e. non-`private` `func`s and `var`s) from your `CalculatorBrain` from Assignment 1 except `undo()` and as specified in Required Task 3 and 4. Also, continue to use the `Dictionary<String,Operation>` as `CalculatorBrain`'s primary internal data structure for performing operations.
2. ~~Your UI should always be in sync with your Model (the `CalculatorBrain`).~~
3. Add the capability to your `CalculatorBrain` to allow the input of *variables*. Do so by implementing the following API in your `CalculatorBrain` …

   ```
   func setOperand(variable named: String)
   ```

   This must do exactly what you would imagine it would: it inputs a "variable" as the operand (e.g. set`Operand(variable: "x")` would input a variable named `x`). Setting the operand to `x` and then performing the operation `cos` would mean `cos(x)` is in your `CalculatorBrain`.
4. Now that you allow variables to be entered as operands, add a method to evaluate the `CalculatorBrain` (i.e. calculate its result) by substituting values for those variables found in a supplied `Dictionary` …

   ```
   func evaluate(using variables: Dictionary<String,Double>? = nil)
         -> (result: Double?, isPending: Bool, description: String)
   ```

   Note that this takes an Optional `Dictionary` (with `String`s as keys and `Double`s as values) as its argument and that that argument defaults to `nil` if not supplied when this method is called. Also note that it returns a tuple (the first element of which is an Optional `Double`). This method is not `mutating` and you are not allowed to make it so. If a variable that has been set as an operand is not found in the `Dictionary`, assume its value is zero.
5. ~~We made the `result`, `description` and `resultIsPending` `var`s non-`private` API in Assignment 1. That means we signed up to continue to support them even though we are now adding a new feature (variables) in this assignment which sort of makes them irrelevant. Really what we want to do is *deprecate* these (you'll see all sorts of deprecated iOS API in Xcode), but for now we will keep the old `result`, `description` and `resultIsPending` `var`s around and just implement each of them by calling `evaluate` with the argument `nil` (i.e. they will give their answer assuming the value of any variables is zero). However, do not use any of these `var`s anywhere in your code in this assignment. Use `evaluate` instead.~~
6. ~~You cannot use the Swift types `Any` or `AnyObject` anywhere in your solution.~~

7. Add two new buttons to your Calculator's UI: →M and M. Don't sacrifice any of the required operation buttons from Assignment 1 to add these (though you may add yet more operations buttons if you want). These two buttons will set and get (respectively) a variable in the CalculatorBrain called M.

   a. →M calls evaluate in your Model with a Dictionary which has a single entry whose key is M and whose value is the current value of the display, and then updates the display to show the result that comes back from evaluate. Until this button (or the clear button) is pressed again, this same Dictionary should be used every time evaluate is called.

   b. →M does **not** perform setOperand.

   c. Touching M should setOperand(variable: "M") in the brain and then show the result of calling evaluate in the display.

   d. →M and M are Controller mechanics, not Model mechanics (though they both use the Model mechanic of variables).

   e. This is not a very great "memory" button on our Calculator, but it can be used for testing whether our variable function implemented in our Model is working properly. Examples …

      9 + M = √ ⇒ description is √(9+M), display is 3 because M is not set (thus 0.0).

      7 →M ⇒ display now shows 4 (the square root of 16), description is still √(9+M)

      + 14 = ⇒ display now shows 18, description is now √(9+M)+14

8. Show the value of M (if set) in your UI somewhere.

9. Make sure your C button from Assignment 1 works properly in this assignment. In addition, it should discard the Dictionary it was using for the M variable (it should *not* set M to zero or any other value, just stop using that Dictionary until →M is pressed again). This will allow you to test the case of an "unset" variable.

10. Add an Undo button to your Calculator. In Assignment 1's Extra Credit, you might have added a "backspace" button. Here we're talking about combining both backspace and actual undo into a single button. If the user is in the middle of entering a number, this Undo button should be backspace. When the user is not in the middle of entering a number, it should undo the last thing that was done in the CalculatorBrain. Do not undo the *storing* of M's value (but DO undo the setting of a variable as an operand).

Hints

1. Except for the prohibition in Required Task 1, you can change `CalculatorBrain`'s `private` implementation (just not its public API) all you want. The main modification you must make is to start remembering the sequence of operands and operations that are input to the brain because you must be able reevaluate them with arbitrary variable values (and you must also be able to undo that sequence one step at a time).
2. Since `evaluate` is not `mutating` (and is not allowed to be), it will not (and cannot) be implemented by side-effecting `var`s in the `CalculatorBrain` (like `accumulator`). In fact, `accumulator` no longer needs to be a property in the `CalculatorBrain` `struct` at all.
3. Don't forget that methods can be nested inside other methods in Swift. This is just a Hint, not a Required Task.
4. Believe it or not, except for `evaluate`, you can probably implement every single other public method and `var` (including `clear` and `undo`) in a single line of code. And it makes sense that `evaluate` is a little more complicated than the others because that evaluation is at the heart of this `CalculatorBrain`'s function in life.
5. Some of you might be a little worried about the performance ramifications now that `result`, `description` and `resultIsPending` are all calling `evaluate` all the time. Before, the `result` of `evaluate` was essentially "cached" inside `CalculatorBrain`'s private `var`s (like `accumulator`). Of course, we're deprecating these anyway, but even if we weren't, unless these are being called in some loop somewhere (wait'll you see Assignment 3 by the way!), the performance characteristics of these `var`s is probably irrelevant since they're called so infrequently relative to much more expensive operations like drawing on the screen. Optimizing something that is not actually causing a performance problem at the expense of obfuscating your code is bad design. See Donald Knuth.
6. Required Tasks 3, 4 and 5 are Model tasks ONLY. They have nothing to do with the UI (i.e. your Controller and its View). You shouldn't change a single line of code in any file except the one containing your `CalculatorBrain` to implement these 3 tasks.
7. Look up the word deprecate in the dictionary if you have not heard that term when it comes to API. It means "to express disapproval of". It basically means you are telling people to stop using that API because you're probably going to get rid of it sometime in the future. As you look through the iOS documentation, you'll see many deprecated methods. Now you know what Apple is having to do when they want to get rid of some public API they introduced in the past.
8. Required Tasks 7 and 8 are Controller/View tasks ONLY. No additional changes in `CalculatorBrain` are required for these. For example, if `M` appears anywhere in your `CalculatorBrain` code, you've violated MVC. The `M` functionality is purely a UI

feature that just happens to use the `CalculatorBrain` Model's generic variable-processing functionality.

9. Required Tasks 9 and 10 will probably require some implementation in both your Model and your Controller. Be very clear on what goes where. Remember that your Model is UI-independent. Also remember that your Controller can only know about your Model's non-`private` API (i.e., it can't know anything about the Model's internal implementation).

10. Without undo, it's only possible to enter *numeric* values for `M`. If you try to store anything else as the value of `M` (for example, π, or `23÷2`), that expression becomes the new active expression in the calculator, replacing the one you wanted to evaluate in the first place. Once you have undo, though, you can back up to the previous expression after setting `M`. For example, enter `M` `cos`, then π, then →`M`, then undo (to get rid of the π) … now your calculator will show the value of `cos(M)` which should be `-1`.

11. Your Model is not just a `CalculatorBrain` anymore. Your Model is now made up of two different and completely separate `structs`: a `CalculatorBrain` and a `Dictionary`. (the one that contains `M`'s value). That's perfectly legal. There's no rule that says your Model has to be a single data structure.

12. The scope of this assignment is similar to last week's (i.e. if it is taking you more than 100 lines of code, you've probably gone down the wrong path somewhere).

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. `Array`
2. Value Semantics
3. Tuples
4. Default Parameter Values
5. Setting Dictionary Values
6. Other Assorted Swift Language Features
7. MVC

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.).
- Program can be made to crash (e.g. an `Optional` that's `nil` was unwrapped with `!`).
- 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.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice."
- Private API is not properly delineated.
- The boundaries of MVC are violated.
- The Swift API Design Guidelines were not properly adhered to.

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.

1. Have your calculator report errors. For example, the square root of a negative number or divide by zero. There are a number of ways to go about "detecting" these errors (maybe add an associated value to the `unary/binaryOperation` cases which is a function that detects an error or perhaps have the function that is associated with a `unary/binaryOperation` return something that is either an error or a result or ???). How you *report* any discovered errors back to users of the `CalculatorBrain` API will require some API design on your part, but don't force users of the `CalculatorBrain` API to deal with errors if they don't want to (i.e. allow Controllers that want to display errors to do so, but let those that don't just deal with `NaN` and `+∞` appearing in their UI). In other words, **don't break any callers of the API described above (who don't care about errors) to support this feature** (i.e., add methods/properties as needed instead). You are allowed to violate Required Task 11 to implement this Extra Credit item, but not Required Task 1 (you can enhance that data structure, but not switch to a new one).
2. Add app icons to your `Assets.xcassets` file (which we moved to Supporting Files in Lecture 1). The only trick here is to provide versions of your icon in all the right sizes.
3. Create a launch screen for your Calculator in `LaunchScreen.storyboard`. Remember that you will want to use some simple autolayout to make your launch screen look good on all platforms.