Assignment II: Calculator Brain

Objective

You will start this assignment by enhancing your Assignment 1 Calculator to include the changes made in lecture (i.e., the `program` Property List and access control). This is the last assignment for which you will have to replicate code from lecture by typing it in.

You're then going to push its capabilities a bit further by allowing a "variable" as an input to the Calculator and support Undo.

This assignment must be submitted using the submit script described here by the start of lecture next Wednesday (i.e. before lecture 6). You may submit it multiple times if you wish. Only the last submission 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.
- You will also need to watch the recording of lecture 3 and make the same changes to your Assignment 1 code.

Required Tasks

1. All of the changes to the Calculator made in lecture must be applied to your Assignment 1 solution. This includes both the `program var` and properly setting access control on all methods and properties. Get this fully functioning before proceeding to the rest of the Required Tasks. And, as last week, *type the changes in*, do not copy/paste from anywhere.
2. Do not change any non-`private` API in `CalculatorBrain` and continue to use the `Dictionary<String,Operation>` as its primary internal data structure.
3. Your UI should always be in sync with your Model (the `CalculatorBrain`).
4. Add the capability to your `CalculatorBrain` to allow the input of *variables*. Do so by implementing the following API in your `CalculatorBrain` …

   ```
   func setOperand(variableName: String)
   var variableValues: Dictionary<String, Double>
   ```

   These must do exactly what you would imagine they would: the first inputs a "variable" as the operand (e.g. set`Operand("x")` would use a variable named `x`) and the second lets users of the `CalculatorBrain` API set the value for any variable they wish (e.g. `brain.variableValues["x"] = 35.0`). Your `CalculatorBrain` must support any number of variables. You can assume that no variable name will be the same as an operation symbol.
5. The `result var` must now properly reflect variable values (from the `variableValues` dictionary) whenever variables have been used (both in the past and in the future, see 8e below). If a variable has no value set in the dictionary, then use 0.0 as its value. If the `variableValues` dictionary is changed, then the `result` will have to change to reflect any new variable values.
6. Your `description var` must continue to work properly and should show the name of the variable (not its value) wherever it appeared as input.
7. The `program var` added in lecture will need to be upgraded to support variables too.
8. 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 more operations buttons if you want). These two buttons will set and get (respectively) a variable in the `CalculatorBrain` called `M`.
   a. `→M` sets the value of the variable `M` in the `brain` to the current value of the `display` and then shows the `brain`'s `result` in the `display`.
   b. `→M` does **not** perform `setOperand`.
   c. Touching `M` should `setOperand("M")` in the `brain` and then show the `brain`'s `result` 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's good for testing whether our variable function implemented above is working properly. Examples …

      `9 + M = √` ⇒ `description` is `√(9+M)`, `display` is 3 because `M` is not set (and so is 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`

9. Make sure your C button from Assignment 1 works properly in this assignment. In addition, it should **remove** any value for the `M` variable from the `variableValues Dictionary` in the `CalculatorBrain` (not set it to zero or any other value). 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 variable values (but DO undo the setting of a variable as an operand).

11. There are new Evaluation criteria this week, so be sure to check out that section below.

Hints

1. Even though users of the `CalculatorBrain`'s API input a variable with a method called `setOperand`, there's no reason that the `CalculatorBrain`'s internal implementation of variables can't use the operation mechanics to make it work (in fact, this is probably a great idea since you have so much infrastructure already for handling operations inside your `CalculatorBrain`).
2. The scope of this assignment is similar to last week's (i.e. it can be done in its entirety in a few dozen lines of code and if it is taking you more than 100 lines of code, you've probably gone down the wrong path somewhere).
3. Some things (like π or the result of another expression) are a bit tricky to store in `M` after you've already entered the expression you want evaluated (e.g. enter `cos(M)` and then try to set `M` to π). Once you implement Undo, you'll be able to do this (by "undoing" whatever expression, including just hitting π, that you used to calculate the value you wanted to store in `M`).
4. Don't forget to think about your `CalculatorBrain` as a reusable class: it would be more flexible (and there's no reason not) to allow programmers using its public API to clear the brain separately from the brain's variable values (even though your `C` button does clear both).
5. Consider using optional chaining in your implementation of `displayValue`.

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. Property Observer
4. Property List
5. String Manipulation
6. Setting Dictionary Values
7. Other Assorted Swift Language Features

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 unboxed 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.

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. There are some hints for you on the next page.

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 a tuple with both the result and an error (if any) 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 change any of the existing methods or properties in the non-`private` API of `CalculatorBrain` to support this feature** (add methods/properties as needed instead).