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. `CalculatorBrain`, etc.). This is the last assignment for which you will have to replicate code from lecture by typing it in.

Now that we've added an MVC Model to our Calculator, we're going to push its capabilities a bit further. You will enhance your Calculator to allow the input of a "variable" into the Calculator's stack. In addition, you'll provide a way for the user to better see what has been entered so far.

Be sure to review the Hints section below!

Materials

- You will need to have successfully completed Assignment 1. This assignment builds on that.
- You will also need to watch video of lecture 3 and make the same changes to your Assignment 1 code. That video can be found in the same place you found this document.

Required Tasks

1. All of the changes to the Calculator made in lecture must be applied to your Assignment 1. 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 an `enum` as its primary internal data structure.
3. Your UI should always be in sync with your Model (the `CalculatorBrain`).
4. The extra credit item from last week to turn `displayValue` into a `Double?` (i.e, an Optional rather than a `Double`) is now required. `displayValue` should return `nil` whenever the contents of the `display` cannot be interpreted as a `Double`. Setting `displayValue` to `nil` should clear the display.
5. Add the capability to your `CalculatorBrain` to allow the pushing of *variables* onto its internal stack. Do so by implementing the following API in your `CalculatorBrain` …

   ```
   func pushOperand(symbol: String) -> Double?
   var variableValues: Dictionary<String,Double>
   ```

   These must do exactly what you would imagine they would: the first pushes a "variable" onto your brain's internal stack (e.g. `pushOperand("x")` would push a variable named `x`) and the second lets users of the `CalculatorBrain` set the value for any variable they wish (e.g. `brain.variableValues["x"] = 35.0`). `pushOperand` should return the result of `evaluate()` after having pushed the variable (just like the other `pushOperand` does).
6. The `evaluate()` function should use a variable's value (from the `variableValues` dictionary) whenever a variable is encountered or return `nil` if it encounters a variable with no corresponding value.
7. Implement a new read-only (`get` only, no `set`) `var` to `CalculatorBrain` to describe the contents of the brain as a `String` …

   ```
   var description: String
   ```

   a. Unary operations should be shown using "function" notation. For example, the input `10 cos` would display in the `description` as `cos(10)`.

   b. Binary operations should be shown using "infix" notation. For example, the input `3 ↵ 5 −` should display as `3−5`. Be sure to get the order correct!

   c. All other stack contents (e.g. operands, variables, constants like π, etc.) should be displayed unadorned. For example, `23.5` ⇒ `23.5`, `π` ⇒ `π` (not 3.1415!), the variable `x` ⇒ `x` (not its value!), etc.

   d. Any combination of stack elements should be properly displayed. Examples:

   `10 √ 3 +` ⇒ `√(10)+3`

`3 ↲ 5 + √` ⇒ `√(3+5)`

`3 ↲ 5 ↲ 4 + +` ⇒ `3+(5+4)` or (for Extra Credit) `3+5+4`

`3 ↲ 5 √ + √ 6 ÷` ⇒ `√(3+√(5))÷6`

e. If there are any missing operands, substitute a `?` for them, e.g. `3 ↲ +` ⇒ `?+3`.

f. If there are multiple complete expressions on the stack, separate them by commas: for example, `3 ↲ 5 + √ π cos` ⇒ `√(3+5),cos(π)`. The expressions should be in historical order with the oldest at the beginning of the string and the most recently pushed/performed at the end.

g. Your description must properly convey the mathematical expression. For example, `3 ↲ 5 ↲ 4 + ∗` must **not** output `3∗5+4`—it must be `3∗(5+4)`. In other words, you will need to sometimes add parentheses around binary operations. Having said that, try to minimize parentheses as much as you can (as long as the output is mathematically correct). See Extra Credit if you want to really do this well.

8. Modify the `UILabel` you added last week to show your `CalculatorBrain`'s `description` instead. It should put an `=` on the end of it (and be positioned strategically so that the `display` looks like it's the result of that `=`). This = was Extra Credit last week, but it is required this week.

9. Add two new buttons to your Calculator's keypad: `→M` and `M`. These 2 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` (if any)

b. `→M` should **not** perform an automatic ↲ (though it *should* reset "user is in the middle of typing a number")

c. Touching `M` should push an `M` *variable* (not the value of `M`) onto the `CalculatorBrain`

d. Touching either button should show the evaluation of the brain (i.e. the result of `evaluate()`) in the `display`

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

f. 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 …

`7 M + √` ⇒ `description` is `√(7+M)`, `display` is **blank** because `M` is not set

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

`14 +` ⇒ `display` now shows `18`, `description` is now `√(7+M)+14`

10. Make sure your C button from Assignment 1 works properly in this assignment.

11. When you touch the `C` button, the `M` variable should be **removed** from the `variableValues Dictionary` in the `CalculatorBrain` (not set to zero or any other value). This will allow you to test the case of an "unset" variable (because it will make `evaluate()` return `nil` and thus your Calculator's `display` will be empty if `M` is ever used without a →`M`).
12. Your UI should look good on any size iPhone in both portrait and landscape (don't worry about iPad until next week). This means setting up Autolayout properly, nothing more.

Hints

1. Consider using optional chaining in your implementation of `displayValue`.
2. Now that you have this better `displayValue`, make sure you use it properly everywhere in your `ViewController`.
3. If you implemented π by just pushing `M_PI` last week, you will probably have to enhance your `CalculatorBrain` to be able to push "constants" so that your `description` will show π instead of showing 3.1415926…
4. You probably do *not* want to implement π as a variable named π with a value of 3.1415926… Why not? Because programmers using your `CalculatorBrain` might clear all the variable values (for their own purposes) and might then be surprised to see π now not having a value.
5. If you're still a bit confused about what to do about π, don't overthink it. It is just another known `Op`. You can feel free to enhance `Op` as needed to support constants like π.
6. When clearing out your `display`, put a `" "` (space) in there, not `nil` or `""` (empty string), otherwise your `UILabel` will shrink down vertically, shifting your UI around.
7. Obviously `evaluate()` is very similar to the new `description` functionality you're being asked to implement (e.g., they both use recursive helper methods) so use it as a guide.
8. You'll probably be best off doing the "separating expressions with commas" part of the `description` task inside your `description` `var`'s code (rather than in its recursive helper method).
9. You can remove a lot of your history-collecting code from Assignment 1 since you've added new API in `CalculatorBrain` to report that directly. By the way, an elegant solution would probably find a single knothole in your `ViewController` through which to update that `UILabel` (but this is just a Hint, not Required).
10. You should be able to implement the `@IBAction` methods for `M` and `→M` with two or three lines of code each. This is not a Required Task, just sayin' is all.
11. 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's stack separately from the brain's variable values (even though your `C` button does clear both).
12. Your UI must always be in sync with your Model (the `CalculatorBrain`). There are sneaky ways to miss this (e.g. `C`, `M`, `→M`, etc.). Be careful.
13. If your Autolayout gets all messed up (you have conflicting constraints, etc.), consider starting over by removing all the constraints in the scene (using the button in the bottom right corner of the storyboard editor), moving the views to where they look

good in the square layout (using dashed blue lines), getting the constraints on your `UILabel`s set up (by using ctrl-drag to attach them to the sides and to the top and using the Size Inspector to edit any constraints if necessary), then selecting all the `UIButton`s and (using the Pin button in the bottom right of the storyboard editor) applying the constraints you want for all of the buttons at once (equal sizes, spaced evenly). Finally, open up the Document Outline and see if there are any warnings or errors in your constraints and click on the warning/error symbols there to fix them.

14. Autolayout is all about the dashed blue lines. Without them Xcode will struggle to understand what you intend.

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. Optionals
2. Closures
3. `enum`
4. `switch`
5. `Dictionary`
6. Tuples
7. Autolayout
8. Recursion (not really an iOS thing, but something you should know!)

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. Make your description have as few parentheses as possible for binary operations.
2. Add Undo to your Calculator. In Assignment 1's Extra Credit, you might have added "backspace". 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`.
3. Add a new method, `evaluateAndReportErrors()`. It should work like `evaluate()` except that if there is a problem of any kind evaluating the stack (not just unset variables or missing operands, but also divide by zero, square root of a negative number, etc.), instead of returning `nil`, it will return a `String` with *what the problem is* (if there are multiple problems, you can simply return any one of them you wish). Report any such errors in the `display` of your calculator (instead of just making it blank or showing some weird value). You must still implement `evaluate()` as specified in the Required Tasks above, but, if you want, you can have `evaluate()` return `nil` if there are *any* errors (not just in the "unset variable" or "not enough operands" case). The `push` and `perform` methods should still return `Double?` (which is kind of a wasted evaluation, but we want to be able to evaluate your Extra Credit separate from the Required Tasks).

Extra Credit Hints

Here are some thoughts about how to approach the Extra Credit items.

1. Parentheses Removal
   - 1.1. This will require adding the concept of "operator precedence" to `Op` in your `CalculatorBrain`.
   - 1.2. Just like `Op`'s `description` `var`, a "precedence" `var` could return a default value for most `Op`s, but then return a specific, associated value for binary operations. The precedence of a variable or constant or unary operation or operand is all the same, i.e., the highest precedence possible.
   - 1.3. It's probably fine to represent precedence as an `Int` (higher value meaning higher precedence). In this case, the highest precedence possible would be `Int.max`.
   - 1.4. Precedence only affects the *description* of the op stack (in other words, it is extra information that `description` needs in order to know how to optimize its description of the stack). It has no effect on how it is *evaluated*. Evaluation precedence is determined by the order of things on the stack.
2. Undo
   - 2.a. Check out the backspacing Extra Credit in Assignment 1 for some hints how to do the backspacing portion of this if you did not do it last week.
   - 2.b. Backspace/undo should **not** undo the setting of the `M` variable. This allows users to use the calculator to calculate a value of `M` they want, then undo to get back to the expression they're working on that uses `M`.
   - 2.c. You will likely need to add some new, non-`private` API in your `CalculatorBrain` (though the implementation of it is probably a one- or two-liner).
   - 2.d. If you've implemented your code cleanly up until now, this task can otherwise probably be done with a single method in your `ViewController` of a half a dozen lines of code or less. If it's requiring more changes than that, try to understand why and consider attacking this by reorganizing the rest of your code a bit first.
3. Reporting Errors
   - 3.a. What sort of data structure is good for "either-or" situations like this?
   - 3.b. Error reporting should have no effect on your `description`.
   - 3.c. It is likely that you'll need to have a way for `evaluateAndReportErrors()` to ask `BinaryOperation`s and `UnaryOperation`s what error (if any) would be generated if certain operand(s) were passed in.

3.d. One way to do this would be to have an associated value for `BinaryOperation`s and `UnaryOperation`s which is a *function* that analyzes potential arguments and returns an appropriate error `String` if performing that operation would generate an error (or `nil` otherwise). There are lots of other ways to attack this, but this is a "hints" section after all.

3.e. Most operations cannot report any errors, so you'll want to make it easy to create an operation that reports no errors. If you use the hints above, then passing in `nil` as the error testing function would be a simple way to do this. You can make an entire function type be an Optional by putting the function type in parentheses and then put the ? after. For example, `((Double, Double) -> String?)?` is an *Optional function* which takes two `Double`s and returns an Optional `String`.

3.f. Don't forget about Optional chaining syntax. For example, you could invoke an Optional function called `errorTest` like this …

```
if let failureDescription = errorTest?(argument) { }
```

… and the `if` would fail if `errorTest` itself was `nil` *or* if it is not `nil`, but the function `errorTest` refers to *returns* `nil`. Convenient.

3.g. In your `ViewController`, you might find that implementing a new `var` called `displayResult` (which handles both values and errors) and then re-implementing `displayValue` in terms of that new `var` will make your code much cleaner. Or maybe not. This is just a hint, after all, and you might decide to go in a different direction.

3.h. Four great test cases are ÷ (divide by zero), square root (of a negative number), "not enough operands" and "variable x not set".

3.i. If you think about the MVC ramifications of reporting an error, there's a good argument to report them as an error *code* from the Model (the "M" part of the MVC) to the Controller (the "C" part). The Controller would then be responsible for actually communicating the error to the user via the View (the "V" part). For example, the Controller might communicate the error in the user's native language. However, to make this Extra Credit simpler, you can simply return errors as `String`s from your Model and show those `String`s to the user directly. By the way, one could argue that as long as the error `String`s were not `private` in the `CalculatorBrain`, that those `String`s themselves could well be considered to be "error codes" (and could be translated or whatever by the Controller).