Assignment II: Foundation Calculator

Objective

The goal of this assignment is to extend the `CalculatorBrain` from last week to allow inputting variables into the expression the user is typing into the calculator. This involves using Foundation classes, properties, a few class methods, and figuring out memory management: all things talked about in class this week.

You will not be "walked through" this assignment line by line, but there will be some significant guidance as to how to accomplish the goal.

Starting with this assignment, you will be evaluated on whether your code leaks memory, so be sure to understand the memory management ramifications of all of your code. We don't expect perfection on this front at this point, but ignore it at your peril because it gets more challenging as the weeks go by.

Next week's assignment will build on this week's so don't fall behind!

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.

Be sure to check out the Hints section below!

Also, check out the latest additions to the Evaluation section to make sure you understand what you are going to be evaluated on with this (and future) assignments.

Materials

- If you successfully accomplished last week's assignment, then you have all the materials you need for this weeks. It is recommended that you make a copy of last week's assignment before you start modifying it for this week's.

Required Tasks

1. Update your Calculator's code to use properties wherever possible, including (but not necessarily limited to) `UILabel`'s `text` property and `UIButton`'s `textLabel` property as well as using a private property for your `brain` in your Controller.
2. Fix the memory management problems we had in last week's version of the Calculator, including the Model not getting `released` in the Controller's `dealloc` and the leaks associated with `waitingOperation`.
3. Implement this API for your `CalculatorBrain` so that it functions as described in the following sections. You may need additional instance variables.

```
@interface CalculatorBrain : NSObject
{
    double operand;
    NSString *waitingOperation;
    double waitingOperand;
}

- (void)setOperand:(double)aDouble;
- (void)setVariableAsOperand:(NSString *)variableName;
- (double)performOperation:(NSString *)operation;

@property (readonly) id expression;

+ (double)evaluateExpression:(id)anExpression
        usingVariableValues:(NSDictionary *)variables;

+ (NSSet *)variablesInExpression:(id)anExpression;
+ (NSString *)descriptionOfExpression:(id)anExpression;

+ (id)propertyListForExpression:(id)anExpression;
+ (id)expressionForPropertyList:(id)propertyList;

@end
```

4. Modify your `CalculatorViewController` to add a target/action method which calls `setVariableAsOperand:` above with the title of the button as the argument. Add at least 3 different variable buttons (e.g. "x", "a" and "b") in Interface Builder and hook them up to this method.
5. Add a target/action method to `CalculatorViewController` which tests your `CalculatorBrain` class by calling `evaluateExpression:usingVariableValues:` with your Model `CalculatorBrain`'s current `expression` and an `NSDictionary` with a test set of variable values (e.g. the first variable set to 2, the second to 4, etc.). Create a button in your interface and wire it up to this method. The result should appear in the `display`.

Primary API Discussion

Your `CalculatorBrain` should do all the things it used to do (so the calculator you built in **Assignment 1 should not get broken by this assignment**).

This API works the same as before except that now the `CalculatorBrain` "remembers" each call to `setOperand:` and `performOperation:` (since the last clear all operation) in its instance variable `expression`. For example, consider the following code:

```
[brain performOperation:@"C"];
[brain setOperand:4];
[brain performOperation:@"+"];
[brain setOperand:3];
[brain performOperation:@"="];
```

This would still return 7 from the last `performOperation:`, but the `CalculatorBrain` would also be building an `expression` that represents `3 + 4 =` along the way. Now assume the caller of the API did the following:

```
[brain performOperation:@"+"];
[brain setVariableAsOperand:@"x"];
[brain performOperation:@"="];
```

The last `performOperation:`'s return value would now be undefined because the `CalculatorBrain` does not know the value of `x`. But it would still have built an `expression` which is `3 + 4 = + x =`. We'll talk about how this `expression` is stored inside the `CalculatorBrain` in a moment. But first let's consider the new API.

```
@property (readonly) id expression;
```

This property returns an object (**the caller has no idea what type of object it is and never will**) which represents the current `expression` so far (in our example, that is `3 + 4 = + x =`). The caller can take this object, hold on to it if it wants to by `retain`ing it, and eventually (any time it wants) evaluate it by calling this class method:

```
+ (double)evaluateExpression:(id)anExpression
         usingVariableValues:(NSDictionary *)variables;
```

Note that in order to evaluate `anExpression`, the caller has to (as you might expect) pass in an `NSDictionary` of the variables' values. The keys in this `variables` dictionary are `NSString` objects (e.g. `@"x"`) and the values are `NSNumber` objects (e.g. 43.7).

The return value of this method is `anExpression` evaluated with the values in `variables`.

Voila! Our `CalculatorBrain` API can now let a caller build an `expression`, give out an object which represents that `expression`, and then let the caller evaluate the `expression` given a dictionary which specifies the values of the variables!

We'll be testing it by touching some buttons on our existing calculator UI, including a few buttons that call `setVariableAsOperand:`, then grabbing the current `expression` from the `CalculatorBrain`'s `expression` property and calling `evaluateExpression:usingVariableValues:` on it with some test data in the `NSDictionary`.

But first let's go over how to implement this API ...

Implementation Discussion

Let's talk data structures and algorithms first (always a good place to start). How should the `expression` be represented internally to our `CalculatorBrain`? We'll need an instance variable for it. But what type of object should it be? And what should we call it?

As for the name of this instance variable, you *could* use "expression," but that might be kind of confusing because expression is the name of the public property we're going to return and so it might be better to call our instance variable `internalExpression` or something so we can keep it straight whether we're talking about the public thing we return or the internal thing we are using as our data structure.

And as for the type of this instance variable, we recommend `NSMutableArray`. Using this data structure will support the following algorithm: just throw an `NSNumber` (containing `operand`) in there each time `setOperand:` is called (i.e. the `operand` property is set) and an `NSString` (the operation) each time `performOperation:` is called.

But what about when `setVariableAsOperand:` is called? We suggest throwing an `NSString` in there (containing the name of the variable), but prepending a string to the variable name to identify it as a variable (e.g. `@"%"` or some such). We need to do this so we can tell the difference between variables and operations in our array (since both are `NSString` objects).

For example, if the caller sends `[brain setVariableAsOperand:@"x"]`, you would throw `@"%x"` into your `NSMutableArray`. Be careful, though, because `@"%"` by itself looks like it might be a valid operation someday! (Hint: A string in your array has to have a `length` of at least 1 + the `length` of your prepended string to be considered a variable.)

There are other ways to do it, but by doing it this way, you'll get more experience with `NSString` methods. You'll also keep your instance variable, `internalExpression`, as a pure property list, which will make implementing the class utility methods a lot easier (see below).

By the way, for code cleanliness, you can use the C keyword `#define` to define your prepended string at the top of your implementation file ...

```
#define VARIABLE_PREFIX @"%"
```

... and then use it like this example in your code: `NSString *vp = VARIABLE_PREFIX;`

So how do we implement our `@property (readonly) id expression`? The simplest way would be to simply write the getter to return our `internalExpression`. But this is a little bit dangerous. What if someone then used introspection to figure out that the thing we returned was an `NSMutableArray` and then added an object to it or something? That might corrupt our internal data structure! Much better for us to give out a `copy` of it. Make sure you get the memory management right though. The method `copy` starts with one of the three magic words `alloc/copy/new`, so you own the thing it returns and so you must be sure to `release` it at the right time. There's a mechanism specifically for that which was discussed in lecture. Use it here.

So then `evaluateExpression:usingVariableValues:` is simply a matter of enumerating `anExpression` using `for-in` and setting each `NSNumber` to be the `operand` and for each `NSString` either passing it to `performOperation:` or, if it has the special prepended string, substituting the value of the variable from the passed-in `NSDictionary` and setting that to be the `operand`. Then return the current `operand` when the enumeration is done.

But there is a catch: `evaluateExpression:usingVariableValues:` is a class method, not an instance method. So to do all this setting of the `operand` property and calling `performOperation:`, you'll need a "worker bee" instance of your `CalculatorBrain` behind the scenes inside `evaluateExpression:usingVariableValues:`'s implementation. That's perfectly fine. You can `alloc/init` one each time it's called (in this case, don't forget to `release` it each time too, but also to grab its `operand` before you release it so you can return that operand). Or you can create one once and keep it around in a C static variable (but in that case, don't forget to `performOperation:@"C"` before each use so that memory and `waitingOperation` and such is all cleared out).

Don't forget to get the memory management right. Remember that `NSMutableArray`, `NSMutableDictionary`, etc. all send `retain` to an object when you add it (or use it as a key or value) and then send `release` to any object when they themselves are `released` or when you remove the object.

Utility Class Methods Discussion

That's it for the basic operation and implementation of the variable-accepting `CalculatorBrain`. What about the other methods in the API? They are all **class methods** which are "utility" or "convenience" methods.

The first utility method just returns an `NSSet` containing all of the variables that appear anywhere in `anExpression`.

```
+ (NSSet *)variablesInExpression:(id)anExpression;
```

Remember that this `NSSet` should only contain one of each variable name so even if `@"x"` appears in `anExpression` more than once (e.g. `anExpression` like `x * x =`), it will only be in the returned `NSSet` once. That's what we want. You might find the `NSSet` method `member:` useful to help keep the set unique.

To implement this one, you just enumerate through `anExpression` (remember, it's an `NSArray`, even though the caller doesn't know that, `CalculatorBrain`'s internal implementation does) using `for-in` and just call `addObject:` on an `NSMutableSet` you create. It's fine to return the mutable set through a return type which is immutable because `NSMutableSet` inherits from `NSSet`. Be sure to get the memory management right!

Also, it is highly recommended to have this method return `nil` (not an empty `NSSet`) if there are no variables at all in `anExpression`. That way people can write code that reads like this: `if ([CalculatorBrain variablesInExpression:myExpression]) {}`. This is a smooth-reading way to ask if `myExpression` has any variables in it (this might be something your Controller wants to do, hint, hint).

The next one just returns an `NSString` which represents `anExpression` ...

```
+ (NSString *)descriptionOfExpression:(id)anExpression;
```

So for our above example, it would probably return `@"3 + 4 = + x ="`.

To implement this you will have to enumerate (using `for-in`) through `anExpression` and build a string (either a mutable one or a series of immutable ones) and return it. Just like in the rest of this assignment, the memory management must be right.

This method should be used in your Controller to update your display. Right now the `display` of your calculator shows the result of `performOperation:`, but as soon as the user presses a button that causes a variable to be introduced into the `CalculatorBrain`, you should switch to showing them this string in the display instead (since the return value from `performOperation:` is no longer valid).

It's okay, by the way, if, when the `userIsInTheMiddleOfTypingANumber`, you just show the number only until they hit an operation or variable key at which point you can go back to displaying the description of expression if there is a variable in the expression.

The final two "convert" `anExpression` to/from a property list:

```
+ (id)propertyListForExpression:(id)anExpression;
+ (id)expressionForPropertyList:(id)propertyList;
```

You'll remember from lecture that a property list is just any combination of `NSArray`, `NSDictionary`, `NSString`, `NSNumber`, etc., so why do we even need this method since `anExpression` is already a property list? (Since the expressions we build are `NSMutableArray`s that contain only `NSString` and `NSNumber` objects, they are, indeed, already property lists.) Well, because the **caller** of our API has no idea that `anExpression` is a property list. That's an internal implementation detail we have chosen not to expose to callers.

Even so, you may think, the implementation of these two methods is easy because `anExpression` is already a property list so we can just return the argument right back, right? Well, yes and no. The memory management on this one is a bit tricky. We'll leave it up to you to figure out. Give it your best shot.

Controller Discussion

Your controller only needs a few slight modifications. The purpose of these changes is solely to test the new `CalculatorBrain` API. Don't waste your time making it prettier than it has to be to effectively test.

First, you should add 3 or 4 new buttons which have a variable name on them (e.g. "x" or "var1") and, when pressed, call some method in your Controller which gets the button's title and calls `setVariableAsOperand:` in your new `CalculatorBrain`.

Second, you should change the place in your Controller where the `display` is updated so that any time there is a variable in the `CalculatorBrain`'s current `expression`, the `display` starts displaying the current `expression` using `descriptionOfExpression:` instead. You could do this based on setting a flag or something in your Controller, but an easier (and better) way is to just use the `variablesInExpression:` method to check whether the current `expression` has any variables in it. Again, it's okay to temporarily not do this if `userIsInTheMiddleOfTypingANumber`.

Lastly, you should implement a "Solve" button. This is just a testing button. It should ask the `CalculatorBrain` for it's current `expression` (`brain.expression`) and then create a variables `NSDictionary` with some test values for each of your variable buttons and then evaluate the `expression` with this dictionary of variable values using the class method `evaluateExpression:usingVariableValues:`.

Your UI will look something like the image above after the user has touched 3 + 4 + x =

Hints

1. If not all of the variables in `anExpression` are defined in the variables dictionary passed to `evaluateExpression:usingVariableValues:`, then the return value of this method is simply undefined. Again, if we were doing a real application, we would probably have it raise an exception or otherwise complain more bitterly if this happened. But for our simple homework case, we'll just fail silently (just as we do for divide by zero).
2. Your `@"C"` (clear all) operation will need some special handling now since your expression array has to be cleared too. Also, be careful of getting into an infinite loop if you decide to put `@"C"` in your expression-building array (you may or may not decide to do that in your implementation). Don't leak the memory of old objects when you clear all.
3. Things to think about with the "Solve" test button:
   a. You're might (or might not) want to `performOperation:@"="` at the beginning of this testing method because you'll do `3 + x` <Solve> and expect it to say `5` (if your test value for `x` is `2`, for example). So if you don't automatically `performOperation:@"="` first, you won't get what you're expecting.
   b. If you do (a) above, you might want to do it only in the case that the last character of `descriptionOfExpression:` is not `@"="`. This is for testing purposes only, so it's not critical either way, but you'll get a lot of `=====` in your `display` otherwise.
   c. Don't forget to check for `userIsInTheMiddleOfTyping` in your Solve method as well.
   d. Another good test in Solve might be to set the display to a string with the format string `@"%@ %g"` where the first argument is the `descriptionOfExpression:` and the second argument is the result of `evaluateExpression:usingVariableValues:`.
4. It might also be a good idea to make a private method in your `CalculatorBrain` which adds a term to your internal expression. You would then call that private method from `setOperand:`, `setVariableAsOperand:` and `performOperation:`. In that private method you could lazily instantiate your expression array at the beginning, add the term to your array in the middle, and do an `NSLog(@"expression = %@", internalExpression)` at the end, then watch your Console as you run your program. It's easy-mode debugging.
5. You've probably figured this out by now, but you must use introspection (`isKindOfClass:`) in your `evaluateExpression:usingVariableValues:` because `anExpression` is going to have `NSString` and `NSNumber` objects mixed together. You'll need it in some of your other class methods as well.

6. Don't let anything crash your program in `CalculatorBrain`. It should defend against any value being passed through its API.
7. Don't forget about implementing (and never calling, except to `super`) `dealloc`.
8. In C, && means "and, but stop evaluating as soon as a term is false." So, for example, `if ((string.length > 1) && ([string characterAtIndex:0] == '%'))` will not even send the `characterAtIndex:` method if the string `length` is not greater than 1. This is useful for writing readable code that is defended against crashing.

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 Documentation that you might find helpful. Get used to referring to this documentation to figure out how to do things. No one expects anyone to memorize every method in every object, so being facile with looking things up in the Apple Documentation is key to mastering the iPhone Development Environment.

- Objective-C Primer
- Introduction to Objective-C
- NSObject Reference
- NSString Reference
- NSNumber Reference
- NSArray Reference
- NSMutableArray Reference
- NSDictionary Reference
- NSSet Reference
- NSMutableSet Reference
- NSMutableDictionary Reference
- Property Lists

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).
- Code is too lightly or too heavily commented.
- Code crashes.
- **Code leaks memory!**

Extra Credit

1. Create `@property` statements (with corresponding `@synthesize` statements) for all of your `IBOutlet` instance variables in your Controller, then set them all to `nil` in your `viewDidUnload`. This will be required on all future assignments, so this is a good opportunity to practice it.
2. Instead of lazily instantiating your Model in your Controller, create it in a method in your Controller named `viewDidLoad` (it is called as soon as your Interface Builder file is finished loading). Then you can remove your `brain` method in your Controller and just use the instance variable `brain` directly.