Assignment 3

Foundations: Due September 29 at 10:00 a.m.

Objective

This week we are going to continue to build our calculator. Our major extension this week will be to allow users to input variables into the expressions they are typing. To do this you will need to use Foundation classes, properties, a few class methods, and some of your new memory management skills. All stuff we have recently covered in class.

Starting with this assignment I will be looking at your code for memory leaks. The code you have written for assignment 2 will have leaks. You'll need to plug them. I don't expect you to be perfect at this stage, but you really need to start learning how to do it.

I recommend that you make a copy of last week's assignment before you start modifying it for this week's assignment (of course if you are using git to manage your projects you don't need to do that!).

Requirements

1. Update your Calculator's code to use properties wherever possible, including (but not 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 inherent in your 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)experessionForPropertyList:(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") to your xib and hook them up to this method.
5. Add a target/action method to CalculatorViewController which tests your CalculatorBrain class by calling evaluateExpression:usingVariables: 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.

API Discussion

Your calculator should still do all the things it could do in assignment 2. Your changes should not break anything.

The biggest difference in your new calculator is that it will now start remembering calls to setOperand: and performOperation: (at least the calls 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 get to how this is stored later. First let's worry about the API.

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

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

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

So this method will take an expression and evaluate it using the NSDictionary of variables. The keys will be NSString objects (e.g. @"x") and the values will be NSNumber objects (e.g. 23.2). So the method will substitute the appropriate values into the expression for the variables and evaluate the whole shooting match (technical CS term) returning the result.

At this point a caller can build expressions, pass out an object that represents the expression, and let the caller evaluate the expression given a dictionary of values for the variables. Admittedly this may not seem all that worthwhile now, but in your next assignment we will turn this basic API into something a little more magical.

We'll test this out using the buttons on our calculator to build expressions and then later solve them.

Implementation

Since we are dealing with Foundation classes we are talking about data structures. The first, and most important, decision that we face is how to represent the expression in our CalculatorBrain. Naturally we'll need an instance variable for it, but what type should it be?

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, I 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? I 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

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)experessionForPropertyList:(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 NSMutableArrays 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. It will also turn out to be important on assignment 5.

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

Your Controller won't change much and the changes are mainly to help you test. This assignment is basically a transitional assignment to get us ready for the next one, so don't spend lots of time making things pretty, just focus on getting it to run properly.

First, you need to add some variable buttons. When these are pressed they should call methods in your Controller which get the button's title and calls setVariableAsOperand: in your updated CalculatorBrain.

Second, you need to change how the display is updated so that once we have variables in our expression the display starts showing the whole expression using descriptionOfExpression:. You could do this by creating a flag in your Controller, but a better way is to just use the variablesInexpression: method to check whether the current expression has any variables in it. Remember that it is ok not to do this if the user is in the middle of typing a number.

Finally, you should implement a "Solve" button. This is for testing purposes only and we will punt it in the next assignment. It should ask the CalculatorBrain for its current expression and then create an NSDictionary of variables with some test values for each of your variable buttons and then evaluate the expression using the dictionary and the class method evaluateExpression:usingVariableValues:.

Your UI will look something like this after the user has punched 3 + 4 + x =

Hints

1. This may be our toughest assignment. Start early. I cannot recommend strongly enough that you do this in small incremental pieces. Test each piece along the way before moving to the next. The order in the Task section is pretty good (the memory management can be put off to the end if you'd like).
2. 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).
3. 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.

4. Solve issues:
    a. You have a choice about whether to performOperation:@"=" at the beginning of this testing method because you'll do 3 + x <Solve> and expect it to say 5 (if x is 2 anyway). So if you don't automatically performOperation:@"=" first, you may not get what you expect.
    b. If you do as in (a) you might want to check if the equal is already at the end of the expression. You'll get lots of ==== if you don't. It isn't a big deal as we are just testing though.
    c. Don't forget to check userIsInTheMiddleOfTyping in Solve 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:.
5. You will need to 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. Your program shouldn't crash even when the user is entering crazy things.
7. Don't forget to implement (but not call) dealloc.
8. In C, && is done with "lazy evaluation." That means that as soon as one term is false it will stop. So, for example, if ((string.length > 1) && ([string characterAtIndex:0] == '%')) will not try the second part at all if the string length is not greater than 1. This is useful for defending against crashes.

Help

Most of what you need can be found in XCode help, but Apple Documentation is useful too. Even ace iPhone developers can't know every method in every object, so it pays to get good at using the documentation effectively.

Evaluation

All the same things from the last assignment still apply. Mainly I am looking for well written, well documented code that works as directed. Starting now I am also looking for memory leaks.