Lab 3:
RPN Calculator App (2 of 3)
Task 1

Task: Add the following 4 operation buttons: sin (calculates the sine of the top operand on the stack), cos (calculates the cosine of the top operand on the stack), sqrt (calculates the square root of the top operand on the stack), π (calculates the value of π).

1. Launch Xcode and go to “File > Open” and select the Xcode project (.xcodeproj) inside the “Calculator(1of3)” folder.

2. Run the application in iOS Simulator and give it a look. Notice the clear button “C” and the floating point button “.” from assignments 1 and 2 in Lab 2. Type in 1.25 E 3.75 + (where E means Enter). The result should be 5.0. Clear and compute 24 * 26.

3. Stop running the application.

4. Look for the clearPressed and pointPressed methods inside CalculatorViewController.m to see how they are implemented.
Task 1

Task: Add the following 4 operation buttons: sin (calculates the sine of the top operand on the stack), cos (calculates the cosine of the top operand on the stack), sqrt (calculates the square root of the top operand on the stack), π (calculates the value of π).

5. Add four more buttons for the sin, cos, sqrt and π operations. Place them under the other buttons and align them to look nice. Note that you should copy and paste an operation button since we want our new operations to send the same operationPressed: action.

The next screenshot shows how your View should look like.

6. Switch to the CalculatorBrain.m tab.

We will modify the implementation of the performOperation: method to also compute these new operations. Note that sin(), cos() and sqrt() are functions in the normal BSD Unix C library. Feel free to use them to calculate sine, cosine and squared root.
self.display.text = [self.display.text stringByAppendingString:@"""]
else
    self.display.text = @"0."
    self.userIsInTheMiddleOfEnteringANumber = YES;
}

- (IBAction)clearPressed
{
    [self.brain emptyOperandStack];
    self.display.text = @"0";
    self.userIsInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)operationPressed:(UIButton *)sender
{
    if (self.userIsInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation]
    self.display.text = [NSString stringWithFormat:@"%.1f", result];
@end
Task 1

Task: Add the following 4 operation buttons: sin (calculates the sine of the top operand on the stack), cos (calculates the cosine of the top operand on the stack), sqrt (calculates the square root of the top operand on the stack), π (calculates the value of π).

7. Implement sin, cos and sqrt as unary operations.

8. Implement π as an operation which takes no arguments off of the operand stack.

Take a look at the next screenshot to check that your code is ok.

9. Run the application and test the new operations. Type in π E 2 / sin. The result should be 1.0 (that is sin(π/2)). Note that 3 π * should put three times the value of π into the display on your calculator, so should 3 E π *, so should π 3 *. Perhaps unexpectedly, π E 3 * + would result in 4 times π being shown. You should understand this is the case because π is an operation which takes no arguments off of the operand stack, not a new way of entering an operand into the display.
```swift
- (double)performOperation:(NSString *)operation {
    double result = 0;
    if ([operation isEqualToString:@"+"]) {
        result = [self.popOperand] + [self.popOperand];
    } else if ([operation isEqualToString:@"-"]) {
        result = [self.popOperand] - [self.popOperand];
    } else if ([operation isEqualToString:@"*"]) {
        double subtrahend = [self.popOperand];
        result = [self.popOperand] - subtrahend;
    } else if ([operation isEqualToString:@"/"]) {
        double divisor = [self.popOperand];
        if (divisor) result = [self.popOperand] / divisor;
    } else if ([operation isEqualToString:@"sin"]) {
        result = sin([self.popOperand]);
    } else if ([operation isEqualToString:@"cos"]) {
        result = cos([self.popOperand]);
    } else if ([operation isEqualToString:@"sqrt"]) {
        result = sqrt([self.popOperand]);
    } else if ([operation isEqualToString:@"^"]) {
        result = pow([self.popOperand], 3.14159268);
    } 
    [self pushOperand:result];
    return result;
}
@end
```
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

1. This task requires a major revision of the CalculatorBrain. In order to make our calculator run programs we have to add some public API in the CalculatorBrain. We also have to allow operations to be pushed and popped off the stack. Thus, we should rename the operandStack to programStack. (don't worry about errors in the CalculatorViewController at this time, we will solve them later).

2. Next, we need to add the public API needed to run programs. In order to run a program we need two things: a copy of the program stack that describes the program and a method that takes a program stack and runs the program.

Let us declare a read-only public @property that returns a copy of the programStack. Name this property program. The type of this property should be id, since we want our CalculatorBrain internal implementation of the “program” to be independent of the public API.

The next screenshot shows how to declare the program property.
The `readonly @property` will generate only the getter method. Thus, you will not be able to set this property through a setter method. In general, you can modify a `readonly` property by accessing it's instance variable from your implementation. By default properties are `readwrite`, so you don't have to add this keyword explicitly.
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

3. The value returned by the program @property is rather calculated than stored (it's a copy of the programStack). Instead of using @synthesize to generate the getter, we have to implement it ourself.

Make sure you return a copy of the programStack and not directly the programStack. Add this implementation after the implementation of the programStack getter.

4. Now let us add the runProgram: method declaration (inside CalculatorBrain.h) that runs programs. It should take an argument with general type id that is a program. Notice that this method is (and should be) independent of any CalculatorBrain instance. This is why we should declare it as a class method.

The following screenshot shows how your code should look by now.
It is a common convention to add class method declarations before instance methods.
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

5. Implement the runProgram: method (inside CalculatorBrain.m) so that it runs the program received as argument.

We should test if the program is an NSArray using introspection then make a mutableCopy of it so that we can operate on the stack.

To operate on the stack we can use another recursive function that will be very similar to the current performOperation: method, only that it will take the mutable copy of the program. Let us name this method popTermOffProgramStack: (we will implement it in a second as a class method). Note that we cannot call instance methods from inside class methods.

The following screenshot shows how your code should look by now. Note that sending the popTermOffProgramStack: message to the CalculatorBrain class generates an error since we haven't implemented it.
This method is not implemented yet. Note that `self` means the CalculatorBrain class here.
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

6. First we should declare the `popTermOffProgramStack:` method as a private class method (inside CalculatorBrain.m).

The argument of this method is an `NSMutableArray` that is a (or a part of a) program stack.

The next screenshot shows how `popTermOffProgramStack:` declaration should look like.

Note that you could directly implement this method (and any “private” method in general) without declaring it in the interface, but you would have to add the implementation before any other method that calls it. Otherwise you will get compiler errors for every call that precedes the method's implementation. To avoid this problem it is better to declare it in the Class Extension.
// CalculatorBrain.h

// Calculator

// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompany__. All rights reserved.

#import <Foundation/Foundation.h>

@interface CalculatorBrain : NSObject
@property (nonatomic, readonly) id program;
+(double)runProgram:(id)program;
-(void)emptyProgramStack;
-(void)pushOperand:(double)operand;
-(double)performOperation:(NSString *)operation;
@end

@end

// CalculatorBrain.m

// Calculator

// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompany__. All rights reserved.

#import "CalculatorBrain.h"

@interface CalculatorBrain
@property (nonatomic, strong) NSMutableArray *programStack;
@end

@implementation CalculatorBrain
@synthesize programStack = _programStack;
-(double)runProgram:(id)program {
  NSMutableArray *stack;
  if ([program isKindOfClass:[NSArray class]]) {
    stack = [program mutableCopy];
    return [self popTermOffProgramStack:stack];
  }

  NSMutableArray *programStack
  if (_programStack == nil) {
    _programStack = [[NSMutableArray alloc] init];
  }
  return _programStack;
}
-(id)program {
  return [self.programStack copy];
}
-(void)emptyProgramStack {
  [self.programStack removeAllObjects];
}
@end
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

7. We should implement the `popTermOffProgramStack:` method (inside CalculatorBrain.m) as a class method.

Because this method is very similar to the `performOperation:` instance method we are going to copy and paste the code from this method.

The method should work like this: we pop a term of the stack and we test whether it is an operand (`NSNumber`) or operation (`NSString`) using introspection. For operands we simply return the `doubleValue`, while for operations we do the appropriate action. Instead of calling `popOperand` instance method as in the `performOperation:` method, we should recursively call `popTermOffProgramStack:` to pop terms off the program stack.

The next screenshot shows how `popTermOffProgramStack:` should look like.
+ (double) popTermOffProgramStack:(NSMutableArray *)stack
{
    double result = 0;
    id topOfStack = [stack lastObject];
    if ((topOfStack) && [stack removeLastObject])
    {
        if ([topOfStack isKindOfClass:[NSNumber class]])
        {
            // We have on operand. Simply return its double value.
            result = ((topOfStack doubleValue);
        }
        else if ([topOfStack isKindOfClass:[NSString class]])
        {
            // We have an operation. Test the operation type and do the appropriate action.
            NSString *operation = topOfStack;
            if ([operation isEqualToString:@"+"])
            {
                result = [self popTermOffProgramStack:stack] +
                          [self popTermOffProgramStack:stack];
            }
            else if ([@"-" isEqualToString:operation])
            {
                result = [self popTermOffProgramStack:stack] +
                          [self popTermOffProgramStack:stack];
            }
            else if ([operation isEqualToString:@"-"])
            {
                double subtrahend = [self popTermOffProgramStack:stack];
                result = [self popTermOffProgramStack:stack] - subtrahend;
            }
            else if ([operation isEqualToString:@"/"])
            {
                double divisor = [self popTermOffProgramStack:stack];
                if (divisor) result = [self popTermOffProgramStack:stack] / divisor;
            }
            else if ([operation isEqualToString:@"-"])
            {
                result = [sin([self popTermOffProgramStack:stack])];
            }
            else if ([operation isEqualToString:@"-"])
            {
                result = [cos([self popTermOffProgramStack:stack])];
            }
            else if ([operation isEqualToNull])
            {
                result = sqrt([self popTermOffProgramStack:stack]);
            }
            else if ([operation isEqualToNull])
            {
                result = exp([self popTermOffProgramStack:stack]);
            }
            else if ([operation isEqualToNull])
            {
                result = 3.14159268;
            }
        }
    }
    return result;
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

8. We should change the `performOperation:` implementation to only push operations on the stack. Thus, we should rename it to `pushOperation:`. The method should return `void` since we don't need to return the result of the current operation anymore. Now we do this by “running” the Calculator's program using `runProgram:`.

To add the operation on the stack we simply call `addObject:` on the `programStack` (the same thing we do when we push operands). Note that operations are `NSString` objects and operands are `NSNumber` objects. We actually used this difference to determine whether we have an operand or an operation using introspection in the `popTermOffProgramStack:` method.

The next screenshot shows how the new `pushOperation:` implementation should look like.

9. We no longer need the `popOperand` method so we can delete it.
result = sin([self popTermOffProgramStack:stack]);
else if ([operation isEqualToString:@"cos")]
    result = cos([self popTermOffProgramStack:stack]);
else if ([operation isEqualToString:@"sqrt"])
    result = sqrt([self popTermOffProgramStack:stack]);
else if ([operation isEqualToString:@"m")]
    result = 3.14159268;
}
return result;
@end

-(NSMutableArray *)programStack
{
    if (_programStack == nil)
    {
        _programStack = [NSMutableArray alloc] init];
    }
    return _programStack;
}
-(id)program
{
    return [self.programStack copy];
}
-(void)emptyProgramStack
{
    [self.programStack removeAllObjects];
}
-(void)pushOperand:(double)operand
{
    NSNumber *operandObject = [NSNumber numberWithDouble:operand];
    [self.programStack addObject:operandObject];
}
-(void)pushOperation:(NSString *)operation
{
    [self.programStack addObject:operation];
}@end
Task 2

Task: Extend the CalculatorBrain with the capacity to run programs.

10. Update the CalculatotViewController to use the new methods defined by the CalculatorBrain Model.

First we need to update the clearPressed method that calls emptyOperandStack instead of emptyProgramStack.

11. Next we should update the operationPressed: method that calls performOperation: instead of the new pushOperation: method.

We also need to get the result of the current operation (since pushOperation: doesn't return it anymore). In order to do this we need to use the runProgram: class method. Note that we use the class method class to get the Class of the self instance object.

The next screenshot shows how your Controller should look like.

12. Run the application in iOS simulator and check that it works the same as before.
```c
- (IBAction)operationPressed:(UIButton *)sender
{  
    if (self.isUserInTheMiddleOfEnteringANumber) self.enterPressed;
    NSString *operation = [sender currentTitle];
    self.brain pushOperation:operation;
    double result = [self.brain runProgram:self.brain.program];
    self.display.text = [NSString stringWithFormat:@"%f", result];
}
@end
```
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

1. Go to the MainStoryboard.storyboard tab in Xcode.

2. Open the Utilities area.

3. You will need new public API in your CalculatorBrain to support this new feature, but let us focus on the variables first.

A variable will be specified as an NSString object. It will be pushed on the stack as other double operands. The value of the variable will be used when we “run” the program.

To simplify our implementation, we will define only three variables x, y and z. We have to define three buttons to set the value of these variables and three buttons that will let us introduce variables in the “program”.

Add an UIButton from the Object Library and place it under the other Calculator's buttons.
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

4. Resize it to 88 pixels wide.

5. Copy and paste this button 5 times and place them on two rows (3 buttons per row).


7. We should also use three displays that will show the current value of x, y and z variables, respectively.

   The easy way to do this is to copy the display label three times and resize the copies to 88 pixels wide.

   Your View should now be similar to the one presented in the next screenshot.
- (IBAction)operationPressed:(UIButton *)sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *operation = [sender currentTitle];
    self.brain.pushOperation:operation);
    double result = [self brain runProgram:self.brain.program];
    [self displayText:NSString stringWithFormat:@"%f", result];
}
@end
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

8. Open CalculatorViewController.h in Assistant Editor.

9. CTRL-drag from the first label in your View to your Controller header to add an outlet for that UILabel. Name this outlet “xDisplay”. Make sure you add it as a weak @property.

10. Do the same thing for the other two labels. Name them “yDisplay” and “zDisplay” respectively.

   The next screenshot shows how your Controller's header should look like.
// CalculatorViewController.h
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *display;
@property (weak, nonatomic) IBOutlet UILabel *xDisplay;
@property (weak, nonatomic) IBOutlet UILabel *yDisplay;
@property (weak, nonatomic) IBOutlet UILabel *zDisplay;

@end
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

11. Next we should add the actions for the “set” buttons. Open CalculatorViewController.m in Assistant Editor.

12. CTRL-drag from the “set x” button to your Controller's implementation to associate an action for this button.

13. Name the action “setX” and make sure you select None for Arguments.

14. In a similar way add two more actions for the “set y” and “set z” buttons. Name the actions “setY” and “setZ”, respectively.

The next screenshot gives you a hint of how your Controller's implementation should look like.
- (void)viewDidLoad
{
    [self setDisplay:nil];
    [self setXDisplay:nil];
    [self setYDisplay:nil];
    [self setZDisplay:nil];
    [super viewDidLoad];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

-(CalculatorBrain *)brain
{
    if (_brain == nil) _brain = [[CalculatorBrain alloc] init];
    return _brain;
}

-(IBAction)setX
{
}

-(IBAction)setY
{
}

-(IBAction)setZ
{
}

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.userIsInTheMiddleOfEnteringANumber)
    {
        self.display.text = [self.display.text stringByAppendingString:digit];
    }
    else
    {
        self.display.text = digit;
        self.userIsInTheMiddleOfEnteringANumber = YES;
    }
}

-(IBAction)pointPressed
{
    if (self.userIsInTheMiddleOfEnteringANumber)
    {
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

15. To set the x variable we simply copy the current display text and set it to the xDisplay text. Make sure you use dot notation for the xDisplay setter.

Also do NOT try to copy the display label itself and assign it to xDisplay. Doing so will replace the xDisplay outlet with the display outlet. Thus, we will have no way to reach the label for the x variable. The point is we only need the text inside the display UILabel.

16. Implement the setY and setZ methods in a similar way.

The next screenshot shows how the setX, setY and setZ methods should be implemented.
@interface ViewController
@property (nonatomic) BOOL userIsInTheMiddleOfEnteringANumber;
@property (nonatomic, strong) CalculatorBrain *brain;
@end

@implementation ViewController
@synthesize display;
@synthesize xDisplay;
@synthesize yDisplay;
@synthesize zDisplay;
@synthesize userIsInTheMiddleOfEnteringANumber;
@synthesize brain = _brain;

-(void)viewDidLoad
{
    [self setDisplay:nil];
    [self setXDisplay:nil];
    [self setYDisplay:nil];
    [self setZDisplay:nil];
    [super viewDidLoad];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

-(CalculatorBrain *)brain
{
    if (_brain == nil) _brain = [[CalculatorBrain alloc] init];
    return _brain;
}

-(IBAction)setX
{
    self.xDisplay.text = self.display.text;
}

-(IBAction)setY
{
    self.yDisplay.text = self.display.text;
}

-(IBAction)setZ
{
    self.zDisplay.text = self.display.text;
}

-(IBAction)digitPressed:(UI用户提供 this) sender
{
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

17. Next we should add the actions for the “x”, “y” and “z” buttons.

CTRL-drag from the “x” button to your Controller's implementation to associate an action for this button. Place this method before the operationPressed: method.

18. Name the action “variablePressed” and make sure you select Sender for Arguments.

19. Using CTRL-drag from the “y” and “z” buttons you can associate the same variablePressed: action. This works only if you save the Controller first (use CMD + S to do this).

20. Change the sender type to be UIButton * instead of id.

The next screenshot gives you a hint about how your Controller's implementation should look like by now.
- (IBAction)pointPressed
{
    if (self.isUserInTheMiddleOfEnteringANumber)
    {
        NSRange range = [self.display.text rangeOfString:@"."];
        if (range.location == NSNotFound)
        {
            self.display.text = [self.display.text stringByAppendingString:@"."];
        }
        else
        {
            self.display.text = @"0.";
            self.isUserInTheMiddleOfEnteringANumber = YES;
        }
    }
}

- (IBAction)enterPressed
{
    [self brain pushOperand:self.display.text doubleValue];
    self.isUserInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)clearPressed
{
    [self brain emptyProgramStack];
    self.display.text = @"8";
    self.isUserInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)variablePressed:(UUID) sender
{
}

- (IBAction)operationPressed:(UUID) sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *operation = [sender currentTitle];
    [self brain pushOperation:operation];
    double result = [CalculatorBrain runProgram:self brain program];
    self.display.text = [NSString stringWithFormat:@"%.2f", result];
}
@end
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

21. Next we should implement the `variablePressed:` action. It works pretty much the same as the `operationPressed:` method.

If the user is in the middle of entering a number we should press ENTER (by calling `enterPressed`) before adding the variable on the stack.

We need to get the variable name by looking at the `sender's currentTitle`. After this, we can push the variable on the stack using the `pushVariable:` method of the brain. We are going to add this method to our Model in a minute.

Then we have to put the variable name in the display.

See how to implement this method on the next screenshot.
- (IBAction)pointPressed
{
    if (self.isUserInTheMiddleOfEnteringANumber)
    {
        NSRange range = [self.display.text rangeOfString:@"."];
        if (range.location == NSRangeNotFound)
        {
            self.display.text = [self.display.text stringByAppendingString:@"." ];
        }
    }
    else
    {
        self.display.text = @"0.";
    self.isUserInTheMiddleOfEnteringANumber = YES;
    }
}

- (IBAction)enterPressed
{
    [self.brain pushOperand:[self.display.text doubleValue]];
    self.isUserInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)clearPressed
{
    [self.brain emptyProgramStack];
    self.display.text = @"0."
    self.isUserInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)variablePressed:(UIButton *)sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *variable = [sender currentTitle];
    [self.brain pushVariable:variable];
    self.display.text = variable;
}

- (IBAction)operationPressed:(UIButton *)sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *operation = [sender currentTitle];
    [self.brain pushOperation:operation];
    double result = [CalculatorBrain runProgram:self.brain.program];
    self.display.text = [NSString stringWithFormat:@"%.4f", result];
}
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

22. Open the CalculatorBrain.m tab in Xcode and let's add the `pushVariable:` method.

23. First we should declare it in the public API of the CalculatorBrain.

24. The implementation of the method is identical to the `pushOperation:` implementation. We simply add the variable to the `programStack` with the `addObject:` method.

See how to declare and implement this method on the next screenshot.
```swift
else if ([operation isEqualToString:@"sqrt"]) {
    result = sqrt([self popTermOffProgramStack:stack]);
} else if ([operation isEqualToString:@"^" && @"m"]) {
    result = pow([self popTermOffProgramStack:stack], [self popTermOffProgramStack:stack]);
}
return result;
}
-
-(NSMutableArray *)programStack
{
    if (_programStack == nil)
    {
        _programStack = ([NSMutableArray alloc] init);
    }
    return _programStack;
}
-
-(id)program
{
    return [self.programStack copy];
}
-
-(void)emptyProgramStack
{
    [self.programStack removeAllObjects];
}
-
-(void)pushOperand:(double)operand
{
    NSNumber *operandObject = [NSNumber numberWithDouble:operand];
    [self.programStack addObject:operandObject];
}
-
-(void)pushOperation:(NSString *)operation
{
    [self.programStack addObject:operation];
}
-
-(void)pushVariable:(NSString *)variable
{
    [self.programStack addObject:variable];
}
@end
```

// CalculatorBrain.h
// Calculator

// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import <Foundation/Foundation.h>
@interface CalculatorBrain : NSObject
@property (nonatomic, readonly) id program;
+(void)runProgram:(id)program;
-(void)emptyProgramStack;
-(void)pushOperand:(double)operand;
-(void)pushOperation:(NSString *)operation;
-(void)pushVariable:(NSString *)variable;
@end
```
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

25. Now we can add variable to the program, but we have to find a way to supply the values of the variables when the “program” is “run.” We must add a new version of the runProgram: class method with the following signature:

```c
+(double)runProgram:(id)program
    usingVariables:(NSDictionary *)variableValues;
```

The keys in the passed variableValues dictionary are NSString objects corresponding to the names of variables, and the values in the dictionary are NSNumber objects specifying the value of the corresponding variable.

See how to declare this method on the next screenshot.
// CalculatorBrain.h
//
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import <Foundation/Foundation.h>

@interface CalculatorBrain : NSObject
@property (nonatomic, readonly) id program;
@end

@implementation CalculatorBrain
@synthesize programStack = _programStack;
+(double)popTermOffProgramStack:(NSMutableArray *)stack;
@end

@end
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

26. It is time to add a private helper class method `isOperation:` that is useful in distinguishing between variables and operations in our implementation of `runProgram:usingVariableValues:`.

Declare this method in the private Class Extension.

27. To implement the `isOperation:` method we will use the `@"+,-,/,*,sin,cos,sqrt, "` constant string (that enumerates all operations) to search for the given operation. The method will return `NO` if the location of the operation in the constant string is `NSNotFound` and `YES` otherwise.

See how this method should look like on the next screenshot.
// CalculatorBrain.m
// Calculator
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 _MyCompanyName_. All rights reserved.

#import "CalculatorBrain.h"

@interface CalculatorBrain : NSObject
@property (nonatomic, strong) NSMutableArray *programStack;
+(BOOL)isOperation:(NSString *)operation;
+(double)popTermOffProgramStack:[NSMutableArray *]stack;
@end

@implementation CalculatorBrain
@synthesize programStack = _programStack;
+(double)runProgram:(id)program
{
    NSMutableArray *stock;
    if ([program isKindOfClass:[NSArray class]])
    {
        stock = [program mutableCopy];
    }
    return [self popTermOffProgramStack:stock];
}
+(BOOL)isOperation:(NSString *)operation
{ NSRange operationRange = [@"+,-,/,\*,sin,cos,\sqrt,m" rangeOfString:operation];
    if (operationRange.location == NSRangeNotFound) return NO;
    return YES;
}
+(double)popTermOffProgramStack:[NSMutableArray *]stack
{
    double result = 0;
    id topOfStack = [stack lastObject];
    if (topOfStack) [stack removeLastObject];
    if ([topOfStack isKindOfClass:[NSNumberOfClass]])
    {
        // We have an operand. Simply return its double value.
        result = [topOfStack doubleValue];
    }
    else if ([topOfStack isKindOfClass:[NSString class]])
    {
        // We have an operator. Test the operation type and do the appropriate action.
    }
    return result;
}
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

28. We can implement `runProgram:usingVariableValues:` without modifying the method `popOperandOffStack` at all.

We have to iterate through the `programStack` objects and detect the variables (that are `NSString` objects) using introspection. Because operations are also `NSString`s, we have to check that `isOperation:` method returns `NO` before we can replace the variable name in the `programStack` with the corresponding values from the `variableValues` dictionary.

In this case it is very useful to use the `NSMutableArray` method `replaceObjectAtIndex:withObject:`. Note that you cannot call this method inside a `for-in` type of enumeration (since you don’t have the index in that case): you would need a `for` loop that is iterating by index through the array.

See how to implement this method on the next screenshot.
@implementation CalculatorBrain
@synthesize programStack = _programStack;
+(double)runProgram:(id)program {
    if ([program isKindOfClass:[NSArray class]])
        stack = [program mutableCopy];
    return [self popTermOffProgramStack:stack];
}
+(double)runProgram:(id)program usingVariables:(NSDictionary *)variableValues {
    NSMutabelArray *stack;
    if ([program isKindOfClass:[NSArray class]])
        stack = [program mutableCopy];
    for (int i = 0; i < stack.count; i++) {
        id stackObject = [stack objectAtIndex:i];
        if ([stackObject isKindOfClass:[NSString class]])
            NSString *operationOrVariable = stackObject;
            if ([stackObject isKindOfClass:[NSString class]])
                NSNumber *variableValue = [variableValues objectForKey:operationOrVariable];
                if (variableValue == nil) variableValue = [NSNumber numberWithDouble:0];
                if (variableValue == nil) variableValue = [NSNumber numberWithDouble:0];
                [stack replaceObjectAtIndex:i withObject:variableValue];
        }
        return [self popTermOffProgramStack:stack];
    }
    +(BOOL)isOperation:(NSString *)operation
    { NSRange operationRange = @"+,-,/,\*,\sin,\cos,\sqrt,\m" rangeOfString:operation;  
        if (operationRange.location == NSNotFound) return NO;  
        return YES;  
    }
    +(double)popTermOffProgramStack:(NSMutableArray *)stack
    { double result = 0;
    ...}
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

29. We can now use the new method to run programs that takes an NSDictionary of variables with associated values.

Return to the MainStoryboard.storyboard tab in Xcode to change the operationPressed: method.

30. We have to build a dictionary with the variable names (as keys) and and associated values. The NSDictionary class method dictionaryWithObjectsAndKeys: is great for this task. It takes a nil-terminated list of objects and keys separated by comma.

31. We can send the runProgram:usingVariableValues: message to the CalculatorBrain class and pass the previously created dictionary for the second argument.

The next screenshot shows how the new operationPressed: implementation should look like.
```objc
    -(IBAction)enterPressed
    {
        [self.brain pushOperand:[self.display.text doubleValue]];
        self.userIsInTheMiddleOfEnteringANumber = YES;
    }

    -(IBAction)clearPressed
    {
        [self.brain emptyProgramStack];
        self.display.text = @"0";
        self.userIsInTheMiddleOfEnteringANumber = NO;
    }

    -(IBAction)variablePressed:(UIButton *)sender
    {
        if (self.userIsInTheMiddleOfEnteringANumber) [self enterPressed];
        NSString *variable = [sender currentTitle];
        self.brain pushVariable:variable;
        self.display.text = variable;
    }

    -(IBAction)operationPressed:(UIButton *)sender
    {
        if (self.userIsInTheMiddleOfEnteringANumber) [self enterPressed];
        NSString *operation = [sender currentTitle];
        [self.brain pushOperation:operation];
        NSDictionary *variableValues = [NSDictionary dictionaryWithObjectsAndKeys:
                                    [NSNumber numberWithDouble:xDisplay.text doubleValue], @"x",
                                    [NSNumber numberWithDouble:yDisplay.text doubleValue], @"y",
                                    [NSNumber numberWithDouble:zDisplay.text doubleValue], @"z",
                                    nil];
        double result = [self brain runProgram:self.brain program
                                           usingVariables:variableValues];
        self.display.text = [NSString stringWithFormat:@"%f", result];
    }
```
Task 3

Task: Add the capability to your CalculatorBrain to accept variables as operands (in addition to still accepting doubles as operands).

32. Run the application in iOS Simulator and use the new features. Press 36 then “set x”. Press C 25 then “set y”. Press C x y +. The calculator should display 61 (that is 36 + 25). Notice we don't have to press Enter after we add a variable.

It's great! We can now make programs that accept variables. But what if we want to run the “x y +” program again with different values for x and y. We cannot set both variables without changing the program. Of course, we could set x and y to the new values and then retype the “x y +” program. This would work, but it's not right!

We should be able to set variables and keep the program in the stack. And it would also be nice if we could see the program that we want to run in some auxiliary display.

We are going to address these issues in the next lab.
Assignment 1

Assignment: Adjust the CalculatorBrain `runProgram:` class method so that it runs programs with variables. Each variable should be replaced with zero for this method. Test this method by placing it in the `operationPressed:` method instead of the method that “runs” “programs” using the actual variable values.

Hints: There are two options for this assignment. You can either call the `runProgram:usingVariables:` and pass `nil` instead of a dictionary with variable values, or you can iterate through the stack objects and replace all variables with zero.

You would have to use `[NSNumber numberWithDouble:0]` in the latter case.
Assignment: Note that the `pushOperation:` and `pushVariable:` methods in CalculatorBrain do the same thing. You should replace them with a single method called `pushOperationOrVariable:`.

Hint: You would also have to change the `variablePressed:` and `operationPressed:` to push things on the stack using the new `pushOperationOrVariable:` method.
Congratulations!