Assignment I Walkthrough

Objective

Reproduce the demonstration (building a calculator) given in class.

Materials

By this point, you should have been sent an invitation to your sunet e-mail to join the iPhone University Developer Program. You must accept this invitation and download the iPhone SDK.

It is critical that you get the SDK downloaded and functioning as early as possible in the week so that if you have problems you will have a chance to talk to the TA's and get help. If you wait until the weekend (or later!) and you cannot get the SDK downloaded and installed, it is unlikely you'll finish this assignment on time.

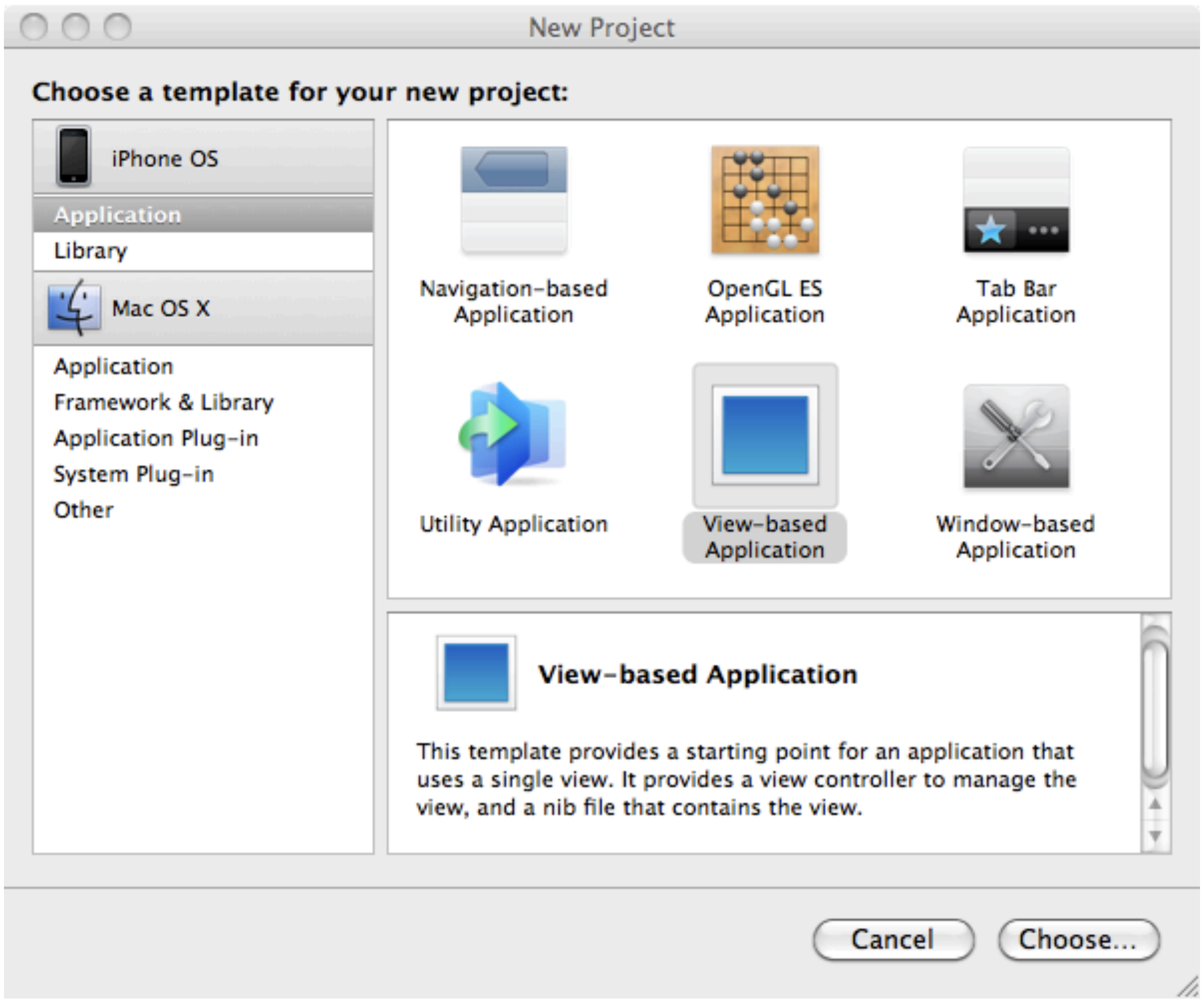
Brief

If you were in class on Wednesday and saw this walkthrough, you may feel like you can get by with a much briefer version included at the end of this document. You can always refer back to the detailed one if you get lost. The devil is often in the details, but sometimes you have to learn from the devil in order to be good.

Detailed Walkthrough

Part I: Create a new Project in Xcode.

1. Launch /Developer/Applications/Xcode .
2. From the Splash screen that appears, choose "Create a new Xcode project" or simply choose the "New Project ..." menu item from the File menu. The following window will appear:



This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a nib file that contains the view.

3. Click on View-based Application and then Choose... .

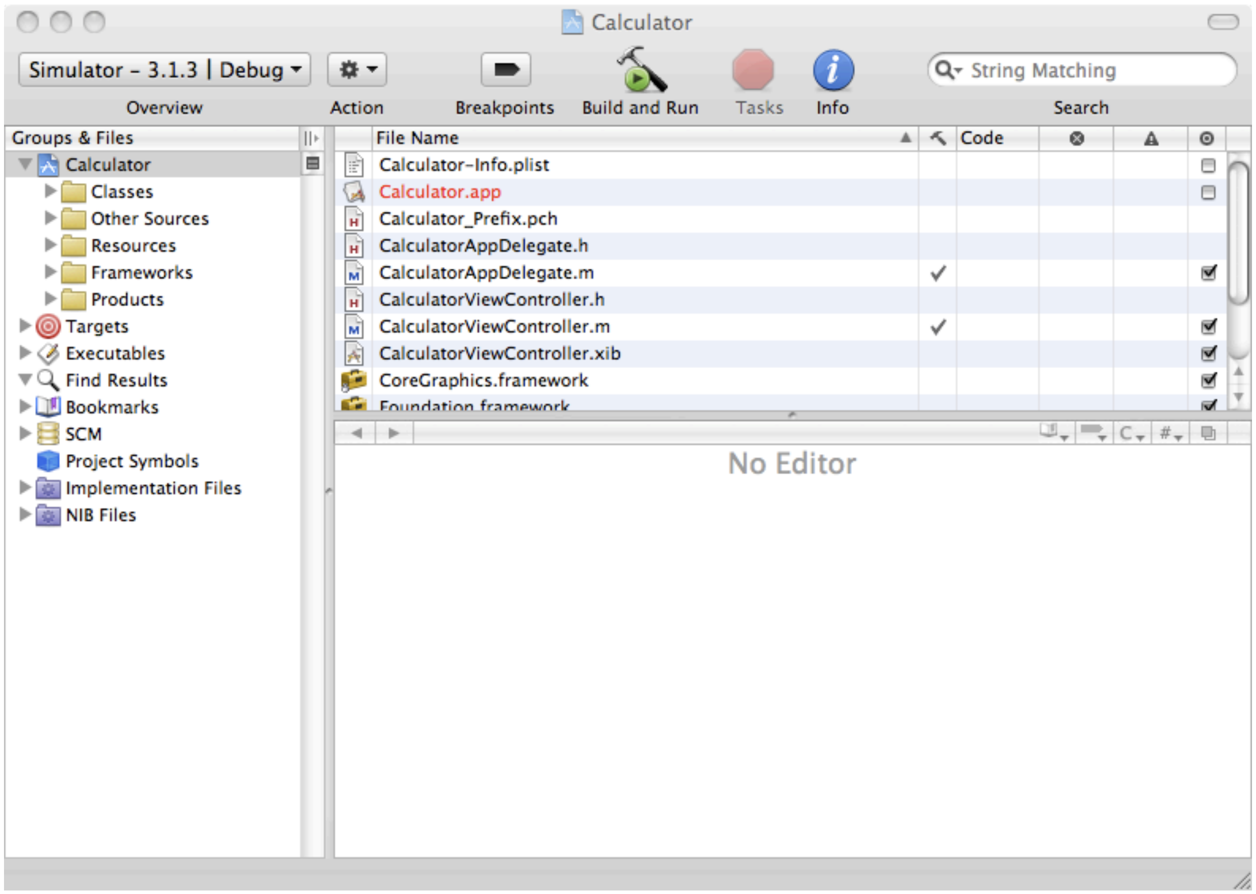
4. In the file chooser that is presented, do the following:

   a. navigate to a place where you want to keep all of your application projects for the quarter (a good place is ~/Developer/CS193p where ~ means "your home directory")

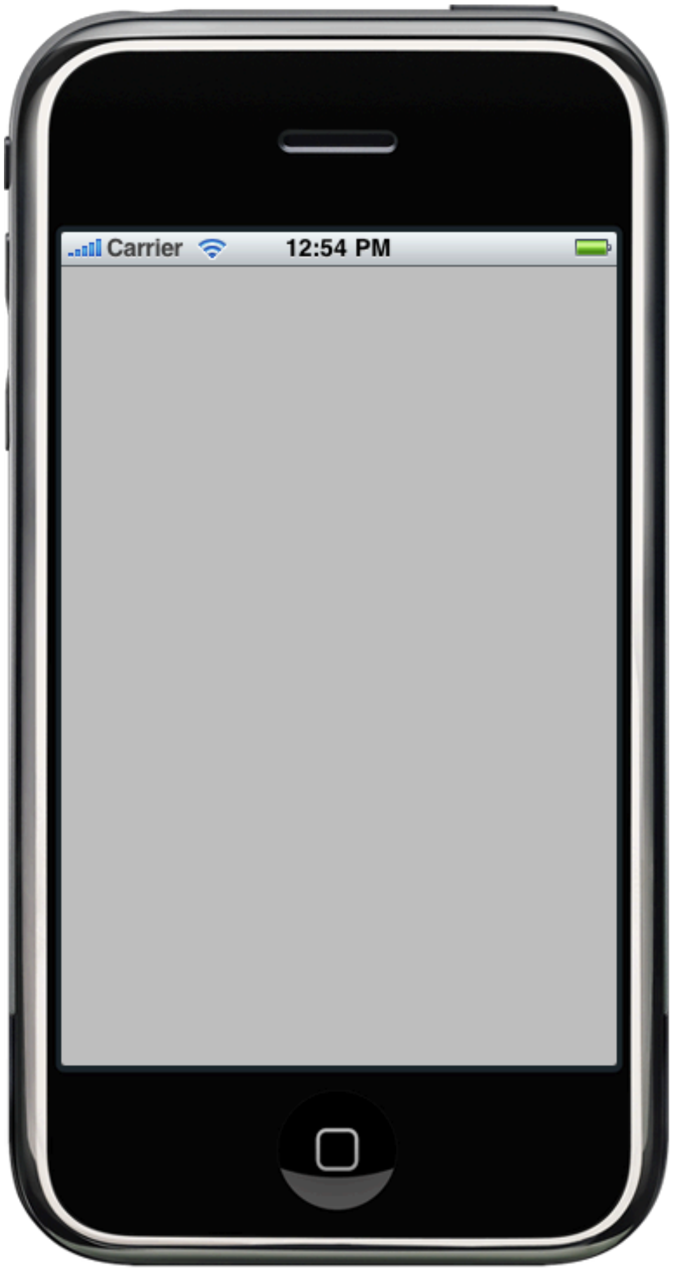
   b. type the name "Calculator" (for the rest of the walk-through to make sense to you, I highly recommend calling this project "Calculator")

   c. click Save

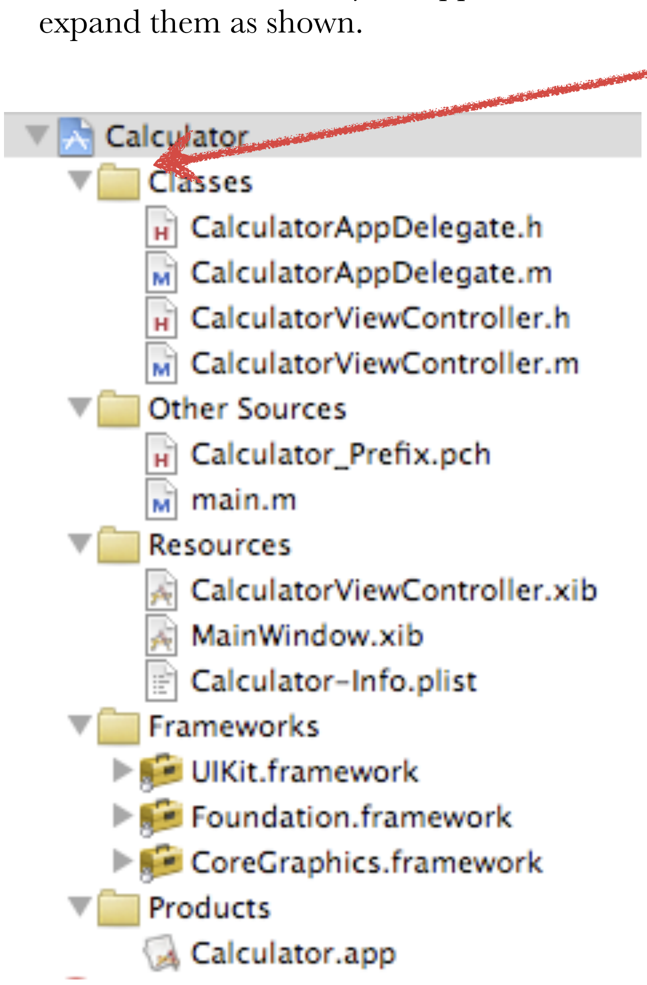
5. You have successfully created your first iPhone project! The following window will appear:



6. You can even run your application at this point by clicking **Build and Run**. Nothing will appear but a blank screen in the iPhone Simulator. If this works, you have likely successfully installed the SDK. If it does not, it might be time to check with a TA!



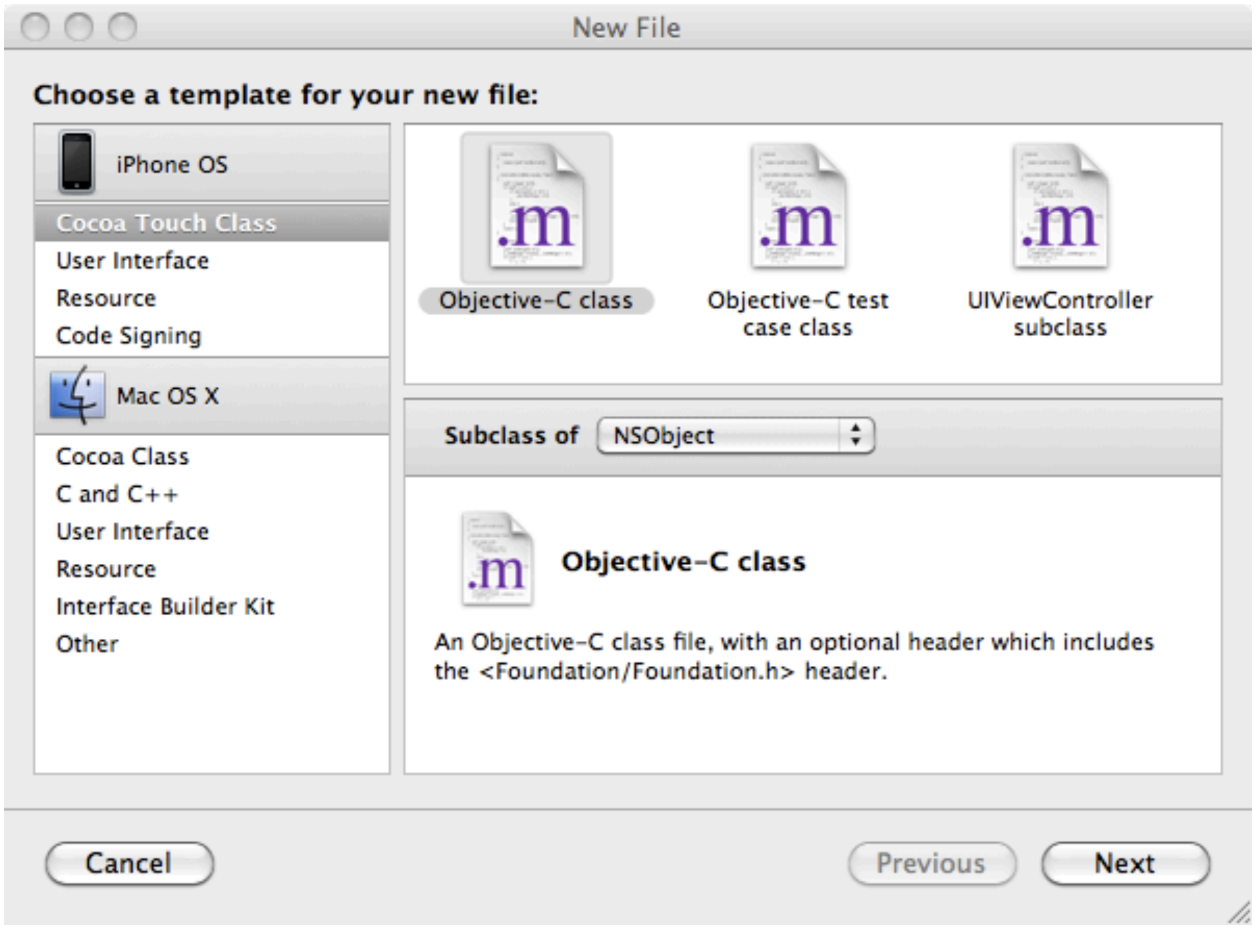
7. Go back to Xcode now. Notice in the upper left hand corner, there is a tree view. This is where all the files in your application are managed. Click on the little folders to expand them as shown.


```
Calculator
    Classes
        CalculatorAppDelegate.h
        CalculatorAppDelegate.m
        CalculatorViewController.h
        CalculatorViewController.m
    Other Sources
        Calculator_Prefix.pch
        main.m
    Resources
        CalculatorViewController.xib
        MainWindow.xib
        Calculator-Info.plist
    Frameworks
        UIKit.framework
        Foundation.framework
        CoreGraphics.framework
    Products
        Calculator.app
```

Note that in the Classes section there are `.h` and `.m` files for two different classes: `CalculatorAppDelegate` and `CalculatorViewController`. Don't worry about `CalculatorAppDelegate` for this assignment. The second one, `CalculatorViewController`, is our controller. We just need our Model. Let's create a new class called `CalculatorBrain` for that.

Part II: Create a class to be our Model

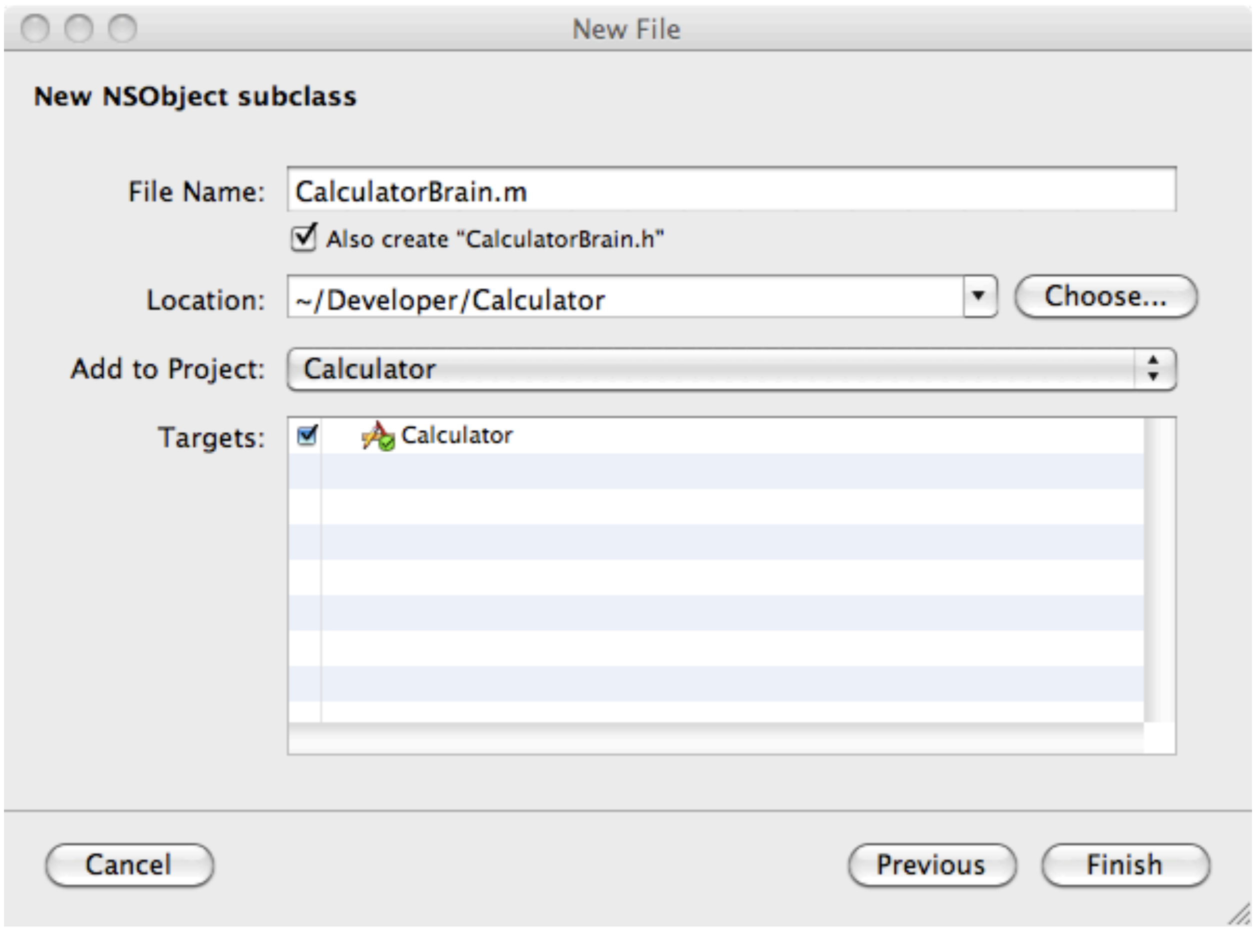
8. Here's how we create a new Objective-C class to be our Model. Choose New File ... from the File menu. The following dialog will appear:



An Objective-C class file, with an optional header which includes the <Foundation/Foundation.h> header.

9. Choose "Objective-C class" and click Next. Leave "Subclass of" on `NSObject`. Pretty much all objects in iPhone development subclass either from `NSObject` directly or from some object that inherits from `NSObject`.

10. Xcode will ask you for the name of this object. Type in `CalculatorBrain.m` and leave the "Also create `CalculatorBrain.h`" box checked because we want both a header file and an implementation file for our `CalculatorBrain`.



New NSObject subclass

File Name: CalculatorBrain.m

☑ Also create "CalculatorBrain.h"

Location: ~/Developer/Calculator

Add to Project: Calculator

Targets: ☑ Calculator

If Xcode dropped your `CalculatorBrain.h` and `CalculatorBrain.m` into somewhere in the Groups & Files area other than Classes, just drag them into the Classes area. You can move files around in the Groups & Files folders area freely at any time. Those little folders are just for your own organizational purposes, they have no semantic meaning to Xcode.

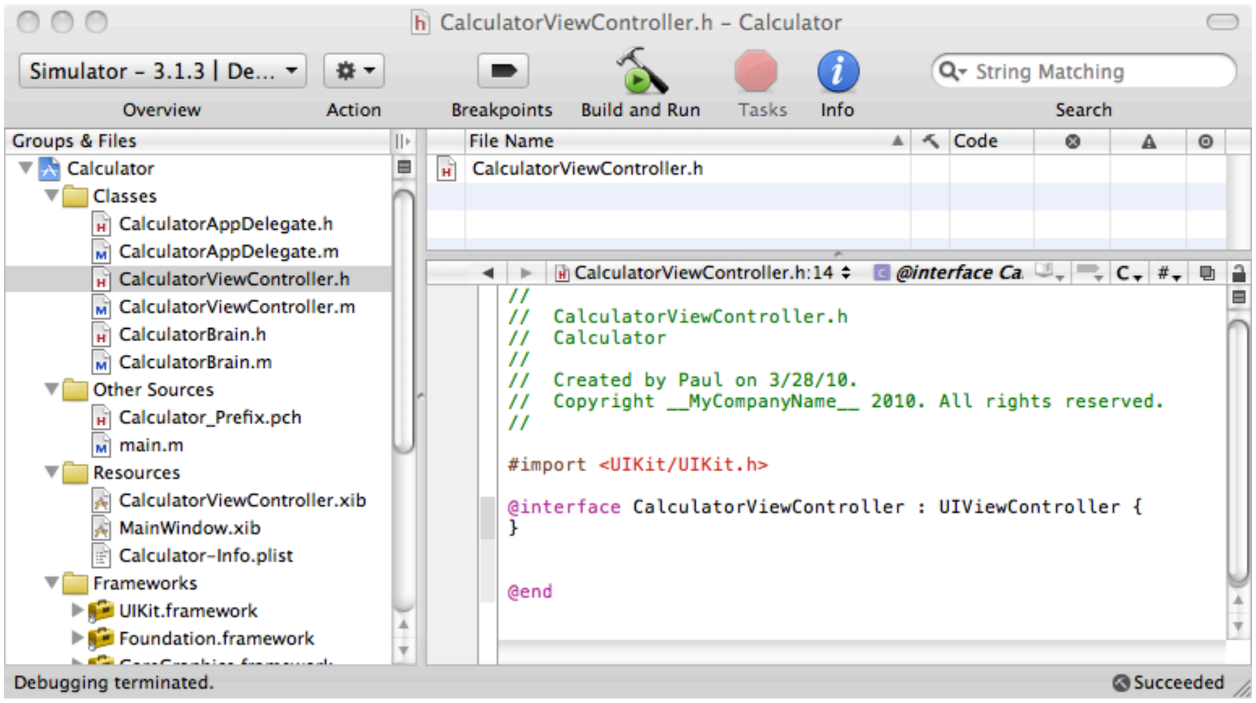
So now our Model is created (though obviously not implemented). Let's take a timeout from our Model and go back to our Controller.

Part III: Define the connections of our MVC's Controller

Now that both our Model and Controller classes exist, it's time to start defining and eventually implementing them. We'll start with defining our Controller.

11. In Xcode's Groups & Files area, find and click on `CalculatorViewController.h`. This is the header file of your calculator's Controller (we'll get to the implementation side of our Controller, `CalculatorViewController.m`, later).

You should see something like the following (for the purposes of this document, the windows have been resized to be as small as possible and still show the content):



```
//
//  CalculatorViewController.h
//  Calculator
//
//  Created by Paul on 3/28/10.
//  Copyright __MyCompanyName__ 2010. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController {
}


@end
```

Notice that Xcode has already put the `#import` we need and made our `CalculatorViewController` be a subclass of `UIViewController`. That's all good.

But our `CalculatorViewController`'s header file still needs the following: our outlets (instance variables that are going to point to things in our View), our actions (methods in our Controller that are going to be sent to us from our View), and an instance variable for our Model.

(In the interest of file size and space, we're going to focus now on the main part of the code itself and not show the entire window or the `#import` statements or comments at the top of each file, etc.)

12. Let's add the outlet which enables our `CalculatorViewController` to talk to the `UILabel` in the View of our MVC design which represents the calculator's display. We'll call that outlet `display`.

```
@interface CalculatorViewController : UIViewController
{
    IBOutlet UILabel *display;
}

@end
```

13. Now let's add an instance variable that points from our Controller to our `CalculatorBrain` (the Model of our MVC design). We need to add a `#import` at the top of the file as well so that `CalculatorViewController.h` knows where to find the declaration of `CalculatorBrain` (add it right after the already existing `#import <UIKit/UIKit.h>`).

```
#import "CalculatorBrain.h"


@interface CalculatorViewController : UIViewController
{
    IBOutlet UILabel *display;
    CalculatorBrain *brain;
}

@end
```

14. And finally (for now), let's add the two actions that our MVC design's View are going to send to us when buttons are pressed on the calculator.

```
@interface CalculatorViewController : UIViewController
{
    IBOutlet UILabel *display;
    CalculatorBrain *brain;
}

- (IBAction)digitPressed:(UIButton *)sender;
- (IBAction)operationPressed:(UIButton *)sender;

@end
```

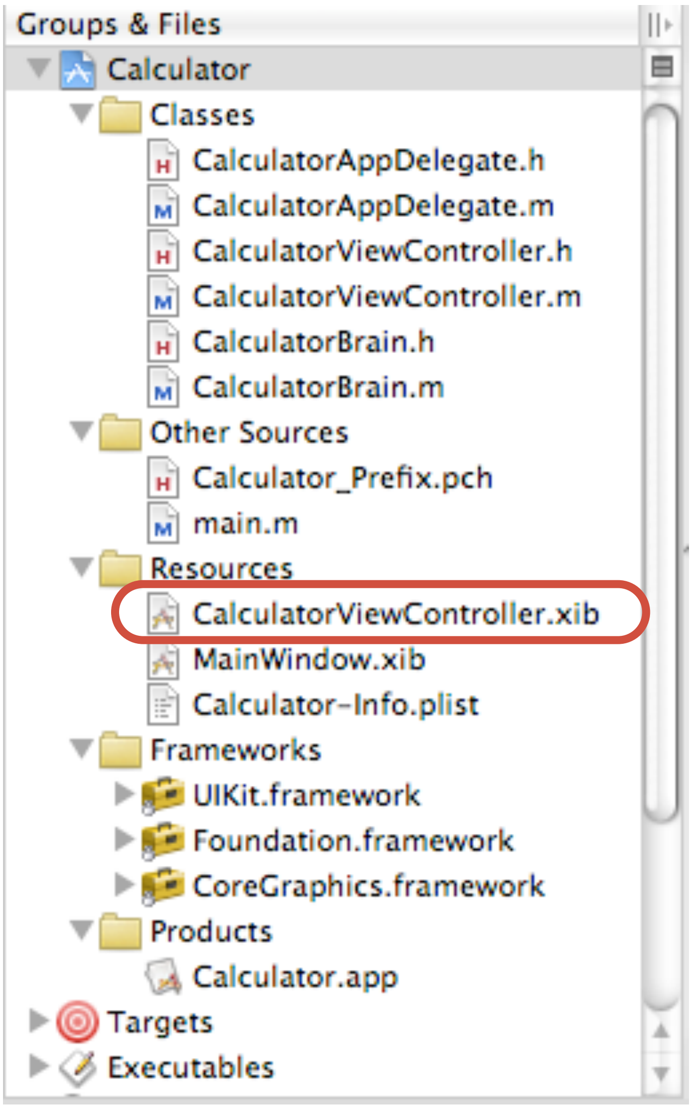
We may need more instance variables for our `CalculatorViewController` as we get into its implementation, but, for now, we've covered the connections of our MVC design for our Controller.

This would be a good time to Build & Run your application again. It'll still be blank, but you can make sure you haven't made any mistakes entering the code above.

Part IV: Wire up our MVC's View in Interface Builder.

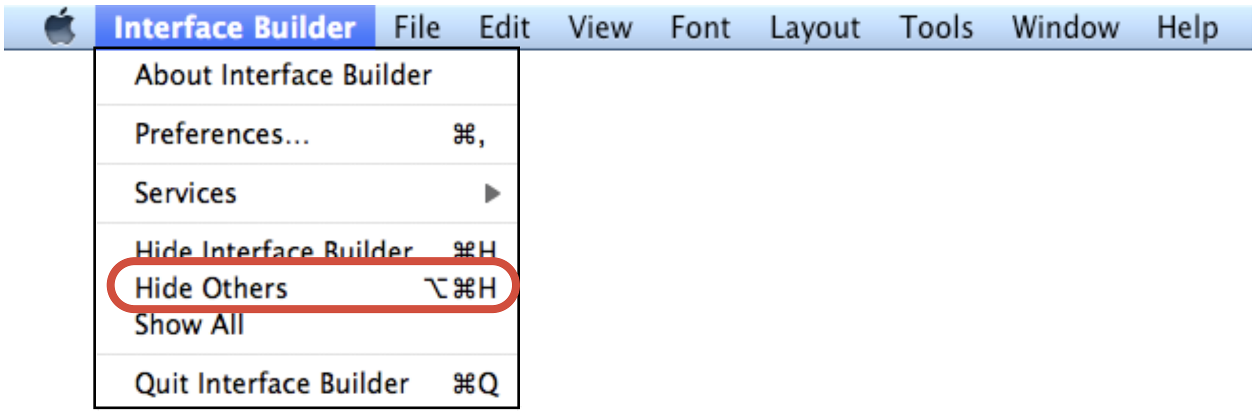
It's time to create the View part of our MVC design. We do not need to write any code whatsoever to do this, we use a tool called Interface Builder. When we created our project and told Xcode that we wanted a View-based project, it automatically created a template Controller (which we just worked on above) and also a template View (which is blank currently). The template View is in a file called `CalculatorViewController.xib`. We call this (for historical reasons) a "nib" file. Some people call it a "zib" file.

15. Open up `CalculatorViewController.xib` by double-clicking on it in the Groups & Files area of Xcode:

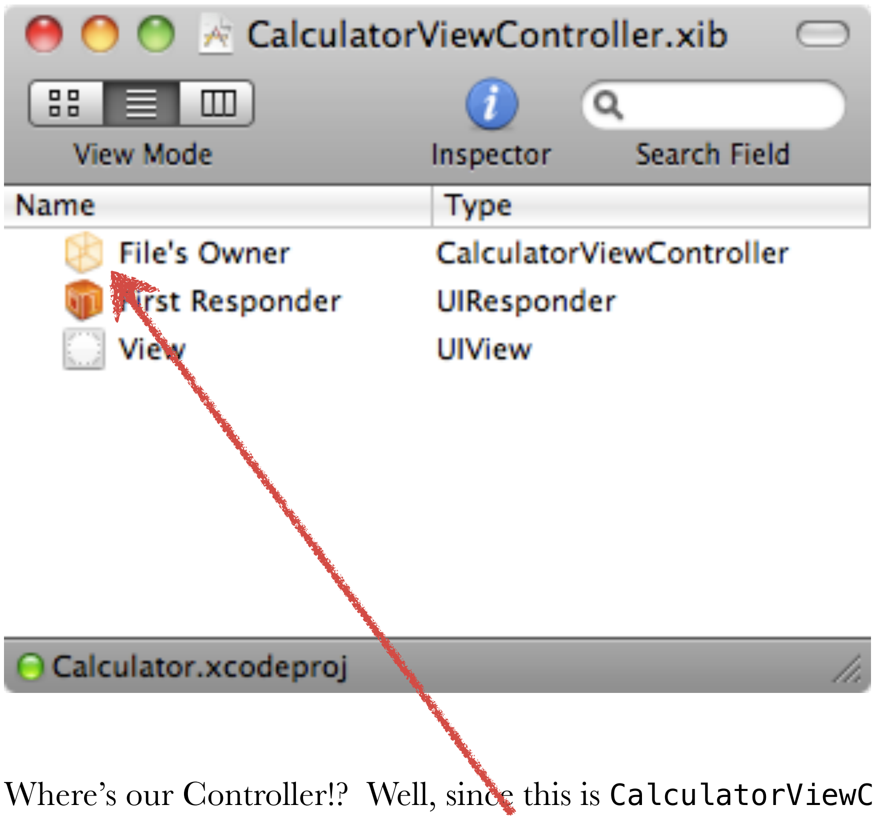


```
Groups & Files
Calculator
    Classes
        CalculatorAppDelegate.h
        CalculatorAppDelegate.m
        CalculatorViewController.h
        CalculatorViewController.m
        CalculatorBrain.h
        CalculatorBrain.m
    Other Sources
        Calculator_Prefix.pch
        main.m
    Resources
        CalculatorViewController.xib
        MainWindow.xib
        Calculator-Info.plist
    Frameworks
        UIKit.framework
        Foundation.framework
        CoreGraphics.framework
    Products
        Calculator.app
Targets
Executables
```

Interface Builder has three main windows plus the windows that contain any objects or groups of objects you're working on. It is strongly recommended that you choose "Hide Others" from Interface Builder's main menu so that all other applications are hidden. It makes it a lot easier to see what's going on in Interface Builder.



The "main window" in Interface Builder shows all of the objects in your `.xib` file:

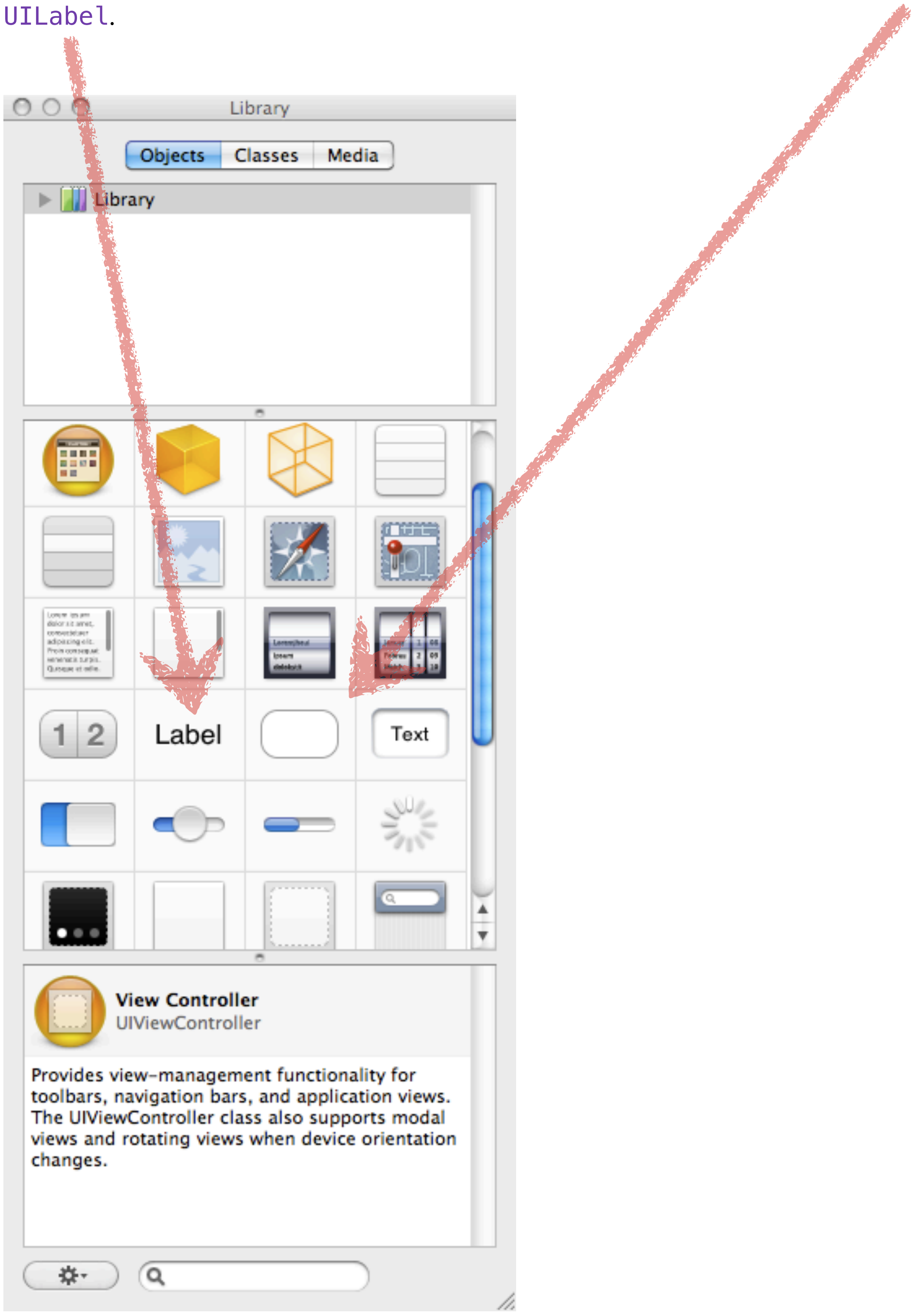


Name	Type
File's Owner	CalculatorViewController
First Responder	UIResponder
View	UIView

Where's our Controller!? Well, since this is `CalculatorViewController.xib`, our `CalculatorViewController` is the "File's Owner". So when we want to wire things up to our `CalculatorViewController`'s outlets and actions (instance variables and methods), this is where we'll drag to or from.

Note the "View Mode" choices in the upper left corner of this window. You can choose to view the objects in your Interface Builder file in a list mode or big icons or even in a hierarchical mode like the Finder. It's up to you.

The next window in Interface Builder is the Library window. It is called that because it contains a library of objects that you can select to help build your View from. If you explore it, you'll see that there are a LOT of objects you can use! We'll get into most of them as the quarter progresses, but today we're only going to use two: `UIButton` and `UILabel`.

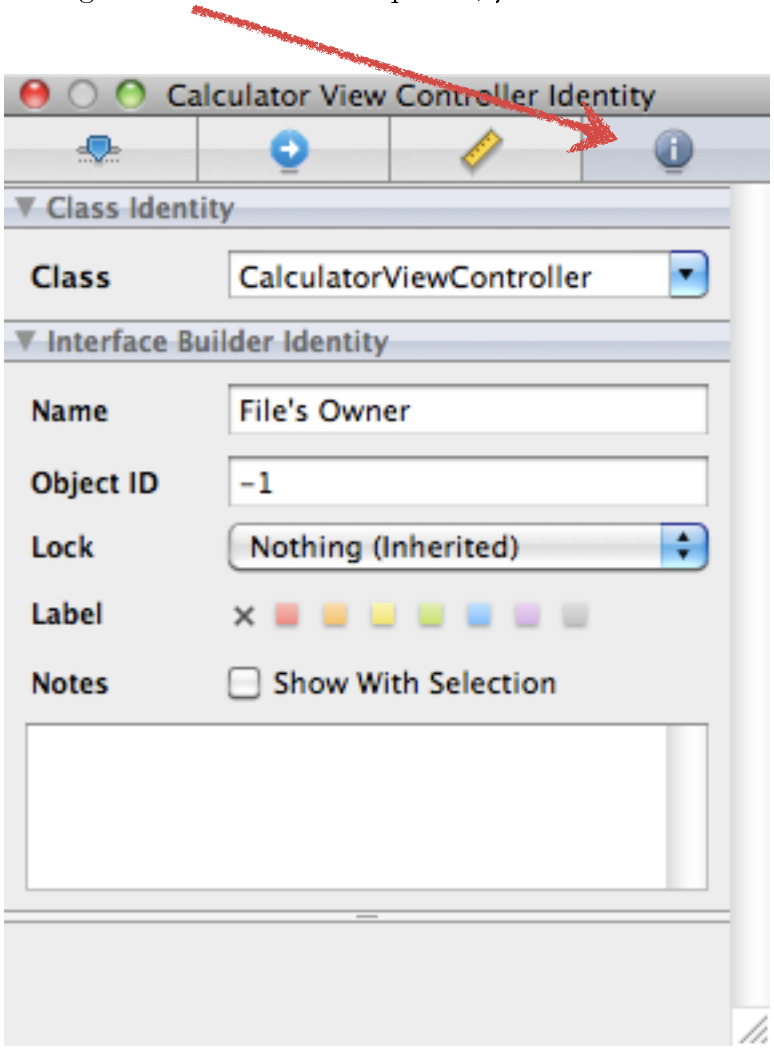


View Controller
UIViewController

Provides view-management functionality for toolbars, navigation bars, and application views. The UIViewController class also supports modal views and rotating views when device orientation changes.

The last window is the Inspector. The contents of this window change depending on which object you have selected (it "inspects" that object). Since we start out with nothing selected, it just says "Empty Selection." But if we click around on the objects in our main window (like File's Owner), we'll start seeing some properties and such that we can set on an object.

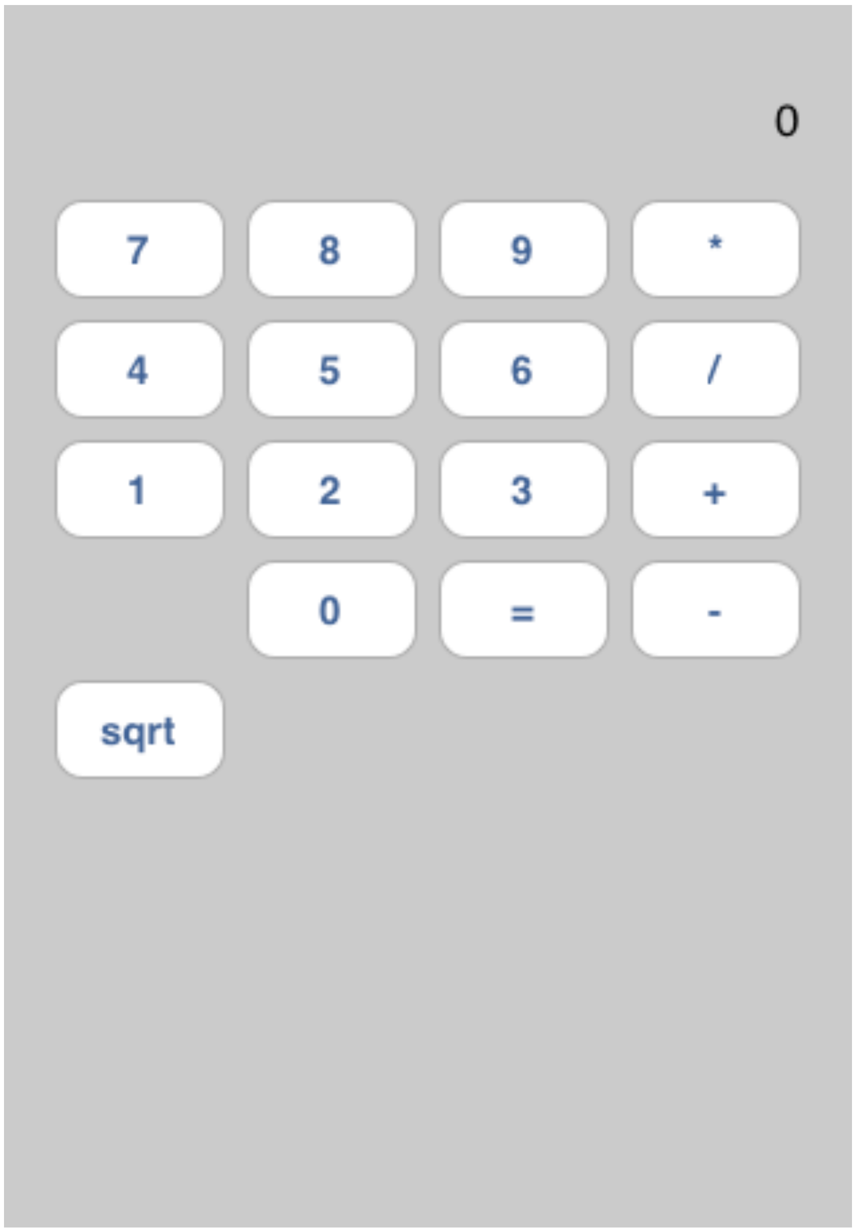
16. For example, if you click on File's Owner in the main window, and then click on the right most tab in the Inspector, you should see something like this:


Calculator View Controller Identity
Class Identity
Interface Builder Identity

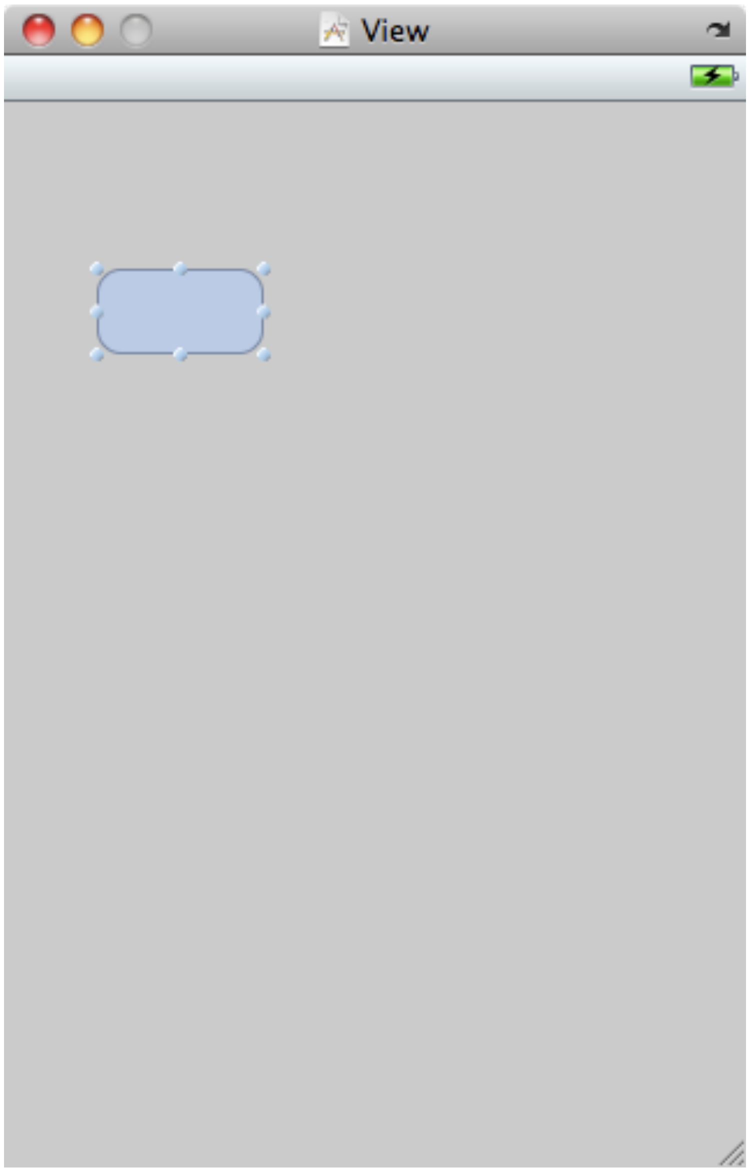
Class: CalculatorViewController
Name: File's Owner
Object ID: -1
Lock: Nothing (Inherited)
Label
Notes: Show With Selection

You can see that our File's Owner's class is `CalculatorViewController` as expected.

The only other window in Interface Builder is blank. That's all that there is so far of the View part of our MVC design. Our next step then, is to put some stuff in there. Here's what we want our user-interface to look like when we're done (it ain't pretty, but it's simple and, at this point, simplicity is more important):



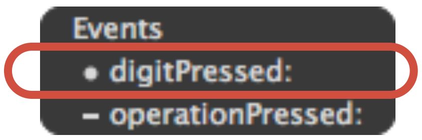
17. Let's start with the "7" key on our calculator. Locate a "Round Rect Button" (a `UIButton`) in the Library window in Interface Builder (there's an arrow pointing to it this document), then simply drag it out into our currently blank View

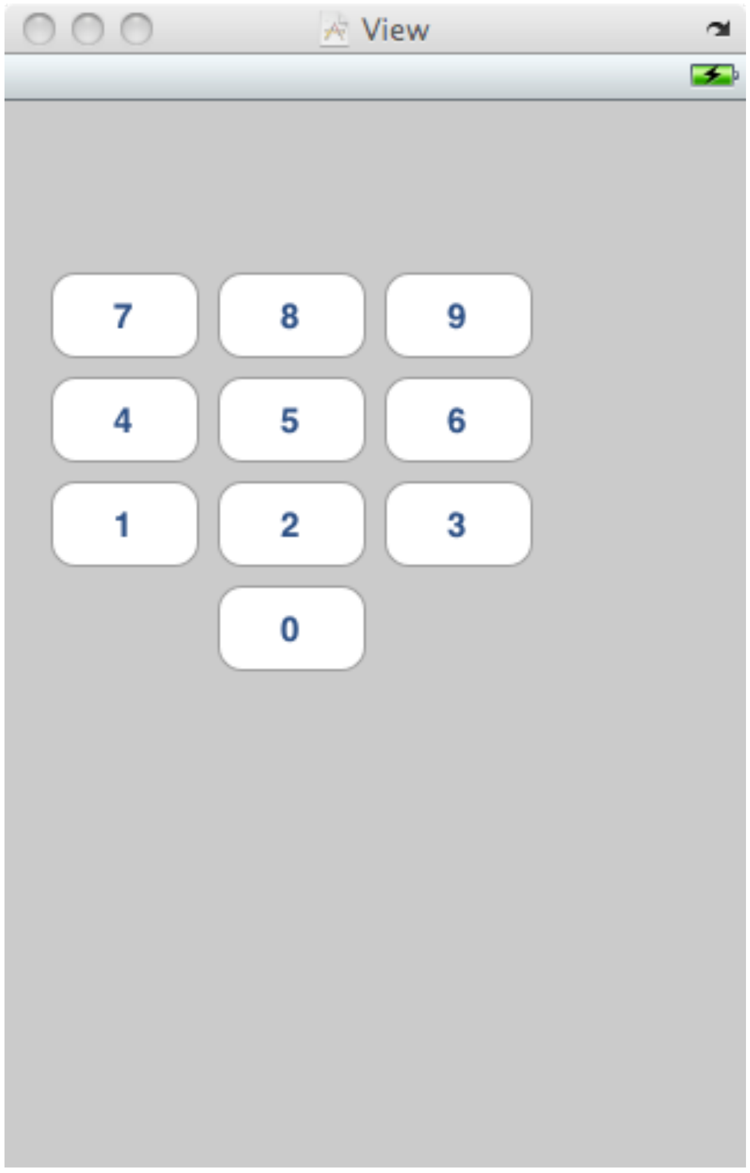


18. Now resize the `UIButton` to be 64 pixels wide (grab one of the little handles on the side of it), and then pick it up and move it toward the left edge. When you get close to the left edge, a vertical blue dotted line will appear letting you know that this is a pretty good left margin for the button in this view. Interface Builder will help you a lot like this with suggestions about lining things up, etc. You can place the button vertically at any position for now.

19. Okay, now for the most important part. Let's wire this button up to our `CalculatorViewController` (the "File's Owner" in the main window). Just hold down the **control** key and drag a line from the button to the File's Owner. There is a bold word in the previous sentence, don't miss it! If you don't hold down control while trying to drag this line, it's just going to pick the button up and start dragging it instead.

20. When you get close to the File's Owner, a blue box should appear around it. When you let go of the mouse button, the little window below and to the left should appear next to File's Owner. Pick `digitPressed:` and voila! Every time that button is touched by the user, it will send `digitPressed:` to your `CalculatorViewController`.





21. Now that you've made that connection, **copy and paste** that button 9 more times to make a total of 10 digit buttons. All of them will have this connection because copying and pasting an object in Interface Builder maintains connections. Lay out the 10 buttons approximately as shown.

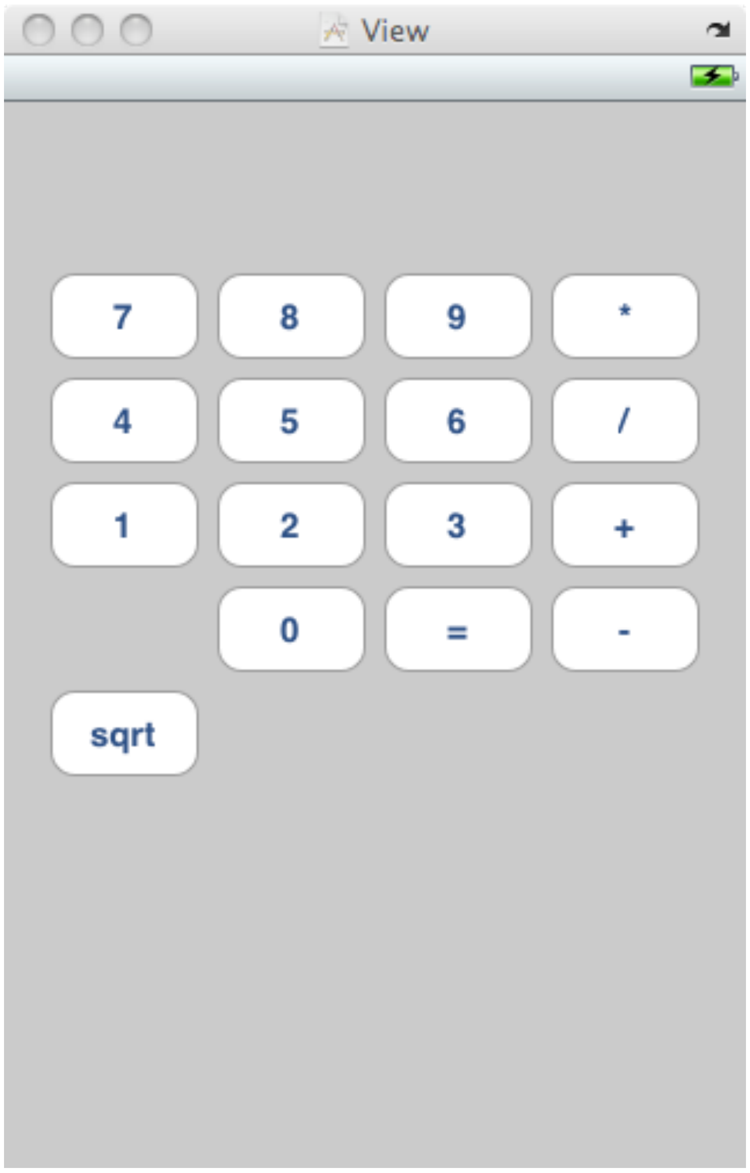
22. Double-click on each button to set it's title. Just use a single digit on each button. If you're having difficulty getting it to let you type the text on the button, try deselecting the button and double-clicking again (basically, just keep clicking on it until it lets you type in the number as the button's title). You can also enter button's titles (and change other properties), by selecting a button and clicking on the left-most tab in the inspector.

Now we're going to do the operation buttons.

23. **Drag out a new button from the Library window.** Do *not* copy and paste a digit button (because we don't want the `digitPressed:` action for these buttons). Resize the button to 64 pixels wide.

24. Hold down **control** and drag a line from this new button to File's Owner. Again, the little black window will appear. **This time select `operationPressed:`.**

25. Now you can copy and paste this 5 times (for / + - = and sqrt) and lay them out nicely. Double-click on each to set the title. **The titles must match the strings you use for your operations in CalculatorBrain.** This is probably not the best design choice, but, again, it's simple. Your UI should now look like this:



Almost there! We just need a display for our calculator.

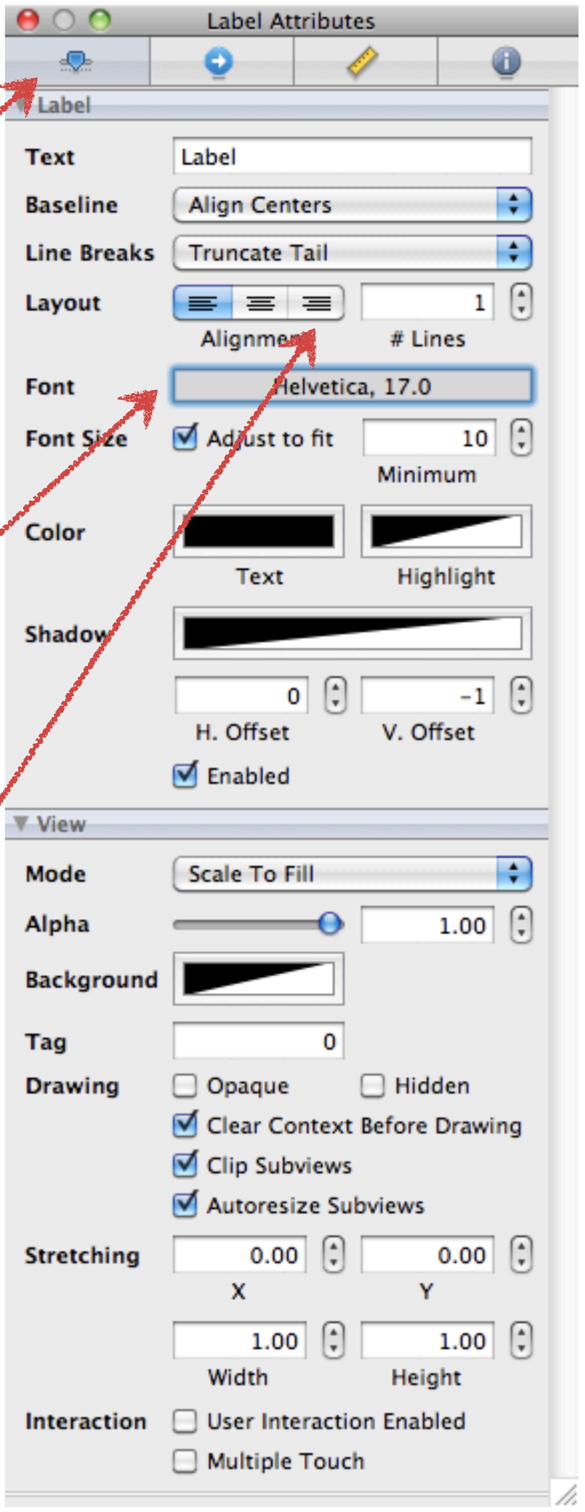
26. Drag out a label (`UILabel`) from the Library window and position and size it along the top of your UI. Double-click on it to change the text in there from "Label" to "0".

27. This time we're going to use the Inspector to change some things about the display. So make sure the `UILabel` is selected and then click on the left-most tab of the Inspector window. It should look the image on the right.

28. Let's start by making the font a little bigger by clicking where it says "Helvetica, 17.0". This will bring up a font choosing panel you can use to change the font. 24 point (at least) would be a good size.

29. Next, let's change the alignment. A calculator's display does not show the numbers coming from the left, it shows them right-aligned. So click on the little button that shows right alignment.

You can play with other properties of the `UILabel` (or a `UIButton`) if you want. Note that this window has a top section (Label) and a bottom section (View). Since `UILabel` inherits (in the object-oriented sense) from `UILabel`, its inspector also inherits the ability to set any properties `UILabel` has. Pretty neat!



Label Attributes

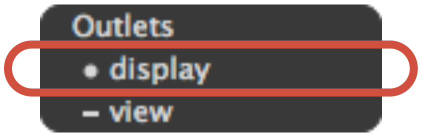
Label

Text: Label
Baseline: Align Centers
Line Breaks: Truncate Tail
Layout: Alignment; # Lines: 1
Font: Helvetica, 17.0
Font Size: ☑ Adjust to fit; Minimum: 10
Color: Text; Highlight
Shadow: H. Offset: 0; V. Offset: -1
☑ Enabled

View

Mode: Scale To Fill
Alpha: 1.00
Background
Tag: 0
Drawing: ☐ Opaque ☐ Hidden
☑ Clear Context Before Drawing
☑ Clip Subviews
☑ Autoresize Subviews
Stretching: X: 0.00; Y: 0.00; Width: 1.00; Height: 1.00
Interaction: ☐ User Interaction Enabled
☐ Multiple Touch

30. Okay, last thing in Interface Builder. Our `CalculatorViewController` needs to be able to send messages to the `display` to update it, so we need to hook it up to our instance variable (aka outlet) in our `CalculatorViewController`. Doing this is similar to setting up the messages sent by the buttons, but we control drag in the opposite direction this time. So hold down control and drag *from* File's Owner *to* the `UILabel`. When you release the mouse, the following little window should appear. Choose `display` (our `CalculatorViewController` also inherited an outlet called "view" from `UIViewController`, but that's going to be set automatically for us and we don't need to worry about that in this assignment).

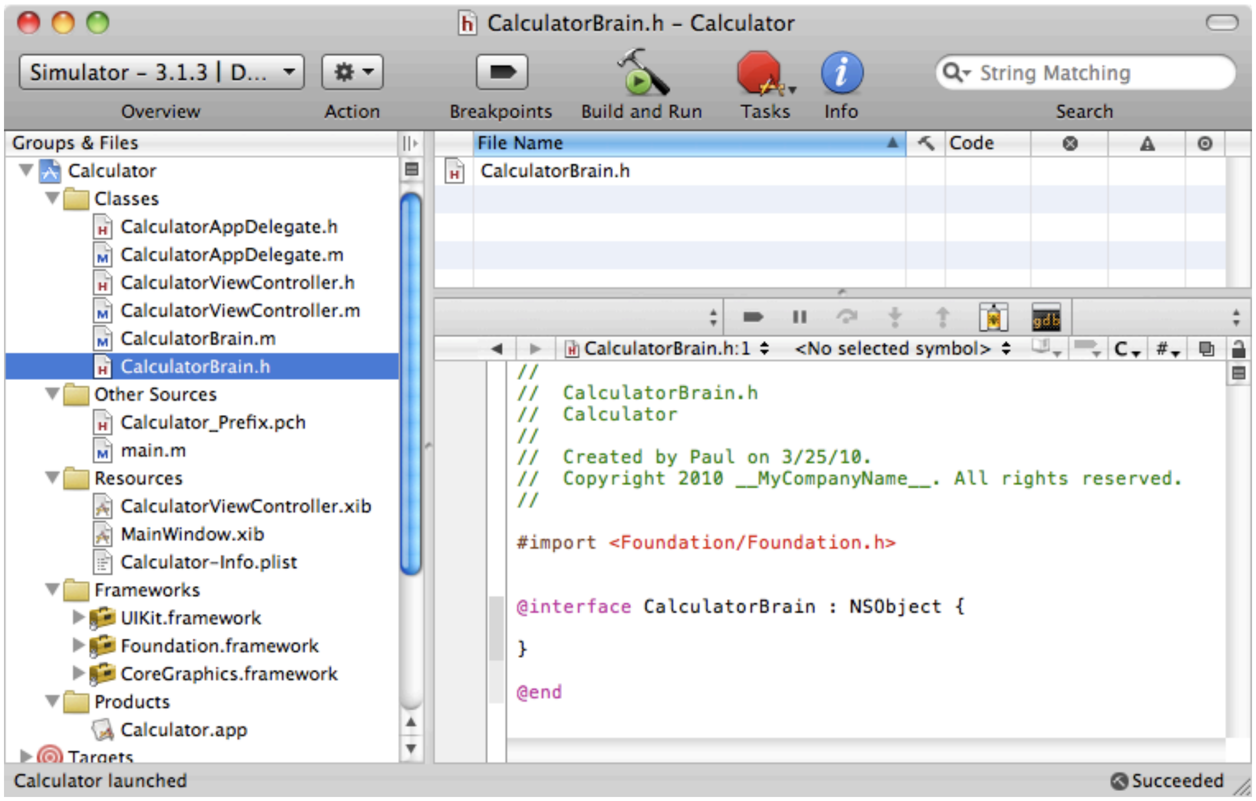


31. Save the file in Interface Builder and then you can quit and go back to Xcode. This would be another good time to Build and Run your program. At least now it won't be blank. Your application now has buttons, but it will **crash** if you touch them because there is no implementation (yet) for your Controller. We've still got some work to do before they will work. We have to implement our Model and our Controller.

Part V: Implement our Model

So far we have created a project, defined the API of our Controller (though we haven't implemented it yet), and wired up our View to our Controller in Interface Builder. The next step is to fill in the implementation of our model, `CalculatorBrain`.

32. Find and click on `CalculatorBrain.h` in the Groups & Files section.



```
//
//  CalculatorBrain.h
//  Calculator
//
//  Created by Paul on 3/25/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <Foundation/Foundation.h>


@interface CalculatorBrain : NSObject {

}

@end
```

You can see that some of the code for our `CalculatorBrain.h` is already there, like the fact that we inherit from `NSObject` and a `#import` for `NSObject` (via the Foundation framework's header file). But there are no instance variables or methods. We need to add those.

Our brain works like this: you set an operand in it, then you perform an operation on that operand (and the result becomes the brain's new operand so that the next operation you perform will operate on that). Things get a bit more complicated if the operation requires 2 operands (like addition or multiplication do, but square root does not). For now, let's add to `CalculatorBrain.h` a couple of things we know we're going to need.

33. Our brain needs an operand. It's going to be a floating point brain, so let's make that instance variable be a double.

```
@interface CalculatorBrain : NSObject
{
    double operand;
}

@end
```

34. Now let's add a method that lets us set that operand.

```
@interface CalculatorBrain : NSObject
{
    double operand;
}

- (void)setOperand:(double)aDouble;

@end
```

35. And finally let's add a method that lets us perform an operation.

```
@interface CalculatorBrain : NSObject
{
    double operand;
}

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

@end
```

Good enough for now. Let's proceed to our implementation.

36. Copy the two method declarations in CalculatorBrain.h, then switch over to CalculatorBrain.m and paste them in between the @implementation and the @end.

```
//
//  CalculatorBrain.m
//  Calculator
//
//  Created by Mr. Programmer on 12/31/10.
//  Copyright Mr. Programmer 2010. All rights reserved.
//

#import "CalculatorBrain.h"

@implementation CalculatorBrain

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

@end
```

The //'s at the beginning are comments. You can put a // in your code at any point, but the compiler will **ignore the rest of the line** after that.

Note that Xcode already put the #import of our class's header file in there for us.

Let's remove those pesky semicolons on the end of the method descriptions and replace them with open and close curly braces. In C, curly braces are what delineates a block of code.

```
@implementation CalculatorBrain

- (void)setOperand:(double)aDouble
{
}

- (double)performOperation:(NSString *)operation
{
}

@end
```

37. Don't forget to remove the semicolons!! This is a common error and will cause compiler warnings.

38. The implementation of `setOperand:` is easy. We just set our instance variable to the `aDouble` that was passed in. Later in the quarter we'll see that this sort of method (one which just sets an instance variable) is so common that the compiler can actually generate it for you.

```
- (void)setOperand:(double)aDouble
{
    operand = aDouble;
}
```

39. The implementation of `performOperation:` is also pretty simple for single operand operations like `sqrt`.

```
- (double)performOperation:(NSString *)operation
{
    if ([operation isEqual:@"sqrt"])
    {
          operand = sqrt(operand);
    }
    return operand;
}
```

The first line is important. It's the first time we've sent a message to an object using Objective-C code! That's what square brackets mean in Objective-C. The first thing after the open square bracket (`[`) is the object to send the message to. In this case, it's the `NSString` object that was passed in to us to describe the operation to perform. The next part is the name of the message. In this case, `isEqual:`. Then comes the argument for `isEqual:`. If the method had multiple arguments, the arguments would be interspersed with the components of the name (more on that later).

Notice that we return the current state of the `operand` when we perform an `operation` so that the caller can update its records on the matter.

Now let's think about operations with 2 operands. This is a bit more difficult. Imagine in your mind a user interacting with the calculator. He or she presses a digit, then an operation, then another digit, then when he or she presses another operation (or equals), *that's* when he or she expects the result to appear. Hmm. Not only that, but if he or she does 12 + 4 sqrt = he or she expects that to be 14, not 4. So single operand operations have to be performed immediately, but 2-operand operations have to be delayed until the next 2-operand operation is requested. Whew!

To do this, we're going to need a couple more instance variables to hold the operation that is waiting for it's second operand and the operand that is waiting along with it.

40. Go back to `CalculatorBrain.h` and add the two instance variables we need to support 2-operand operations: one variable for the operation that is waiting to be performed until it gets its second operand and one for the operand that is waiting along with it. We'll call them `waitingOperation` and `waitingOperand`.

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

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

41. Okay, back to `CalculatorBrain.m`. Here's an implementation for `performOperation:` that will support 2-operand operations too.

```
- (double)performOperation:(NSString *)operation
{
   if ([@"sqrt" isEqual:operation])
   {
      operand = sqrt(operand);
   }
   else
   {
      [self performWaitingOperation];
      waitingOperation = operation;
      waitingOperand = operand;
   }

    return operand;
}
```

Basically, if the `CalculatorBrain` is asked to perform an `operation` that is not a single-operand `operation` (look at the code that is invoked by the `else` that is on a line by itself) then the `CalculatorBrain` calls the method `performWaitingOperation` (which we haven't written yet) on *itself* (`self`) to perform that `waitingOperation`.

If we were truly trying to make this brain robust, we might do something like ignoring back-to-back 2-operand operations unless there is a `setOperand:` call made in-between. As it is, if a caller repeatedly performs a 2-operand operation it'll just perform that operation on its past result over and over. Calling a 2-operand operation over and over with no operand-setting in-between is a little bit undefined anyway as to what should happen, so we can wave our hands successfully in the name of simplicity on this one!

Careful readers will note also that the argument and the destination of the `isEqual:` message have been swapped from the version earlier on the page. Is this legal? Yes, quite. `@"sqrt"` is just as much of an `NSString` as `operation` is, even though `@"sqrt"` is a constant generated by the compiler for us and `operation` is not.

Finally, here's an example of adding another single-operand operation to our brain: `+/-`. It's been added to show you the importance of putting the `else`'s in the right place when you add new single-operand operations (**which you are going to be asked to do on your homework**).

```
- (double)performOperation:(NSString *)operation
{
   if ([@"sqrt" isEqual:operation])
   {
      operand = sqrt(operand);
   }
   else if ([@"+/-" isEqual:operation])
   {
      operand = - operand;
   }
   else
   {
      [self performWaitingOperation];
      waitingOperation = operation;
      waitingOperand = operand;
   }

     return operand;
}
```

We're not quite done here. We still need to implement `performWaitingOperation`. Note the message sent to `self`. This means to send this message to the object that is currently sending the message! Other OO languages sometimes call it "this." `performWaitingOperation` is going to be *private* to our `CalculatorBrain`, so we are *not* going to put it in `CalculatorBrain.h`, only in `CalculatorBrain.m`.

42. Here's the implementation of `performWaitingOperation`. **It's important that you put this code in your `CalculatorBrain.m` file somewhere *before* the implementation of `performOperation:`.** That's because `performWaitingOperation` is a *private* method. It is not in the public API. It must be declared or defined *before* it is used in a file. The best spot for it is probably between your implementation of `setOperand:` and your implementation of `performOperation:`.

```
- (void)performWaitingOperation
{
    if ([@"+" isEqual:waitingOperation])
    {
       operand = waitingOperand + operand;
    }
    else if ([@"*" isEqual:waitingOperation])
    {
       operand = waitingOperand * operand;
    }
    else if ([@"-" isEqual:waitingOperation])
    {
       operand = waitingOperand - operand;
    }
    else if ([@"/" isEqual:waitingOperation])
    {
        if (operand) {
           operand = waitingOperand / operand;
        }
    }
}
```

Pretty simple. We just use `if {} else` statements to match the `waitingOperation` up to any known operations we can perform, then we perform the `operation` using the current `operand` and the operand that has been waiting (`waitingOperand`).

Note that we fail silently on divide by zero (but at least we do not crash). We just do nothing. This is not very friendly to the user (an error message or something would be better), but we're trying to keep this simple, so for now, silent failure.

Note also that, as discussed above, we do nothing at all if the `waitingOperand` is an unknown operation (the `operand` is not modified in that case). It's a reasonable simplification, but an example of where you need to clearly understand your classes' semantics (and hopefully document them well to callers).

Okay, that's it for `CalculatorBrain`. Our Model is implemented. The only thing we have left to do is implement our Controller.

Part VI: Implement our Controller

Almost done. All that's left now is the code that gets invoked when a digit is pressed (`digitPressed:`) or an operation is pressed (`operationPressed:`). This code goes in our `CalculatorViewController`. We've already declared these methods in the header (`.h`) file, but now we have to put the implementation in the `.m` file.

43. Open `CalculatorViewController.m` and select and **delete all the "helpful" code Xcode has provided for you between the `@implementation` and the `@end`** (but leave those two lines there).

44. Now go back to `CalculatorViewController.h` and copy the two method declarations and paste them into `CalculatorViewController.m` somewhere between the `@implementation` and the `@end`. **Remove the semicolons** and replace them with `{ }` (empty curly braces). It should look something like this:

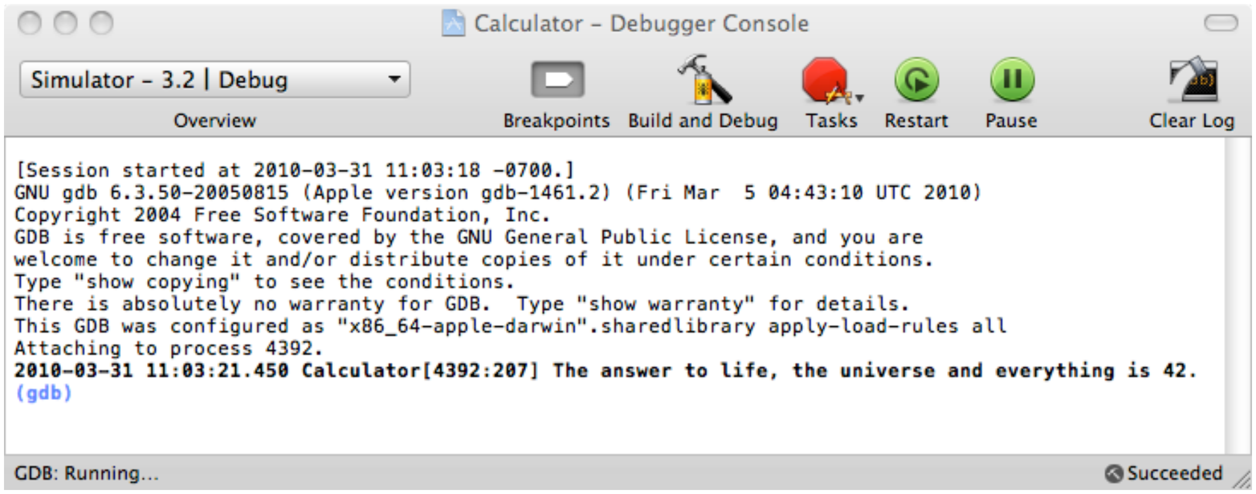
```
//
//  CalculatorViewController.m
//  Calculator
//
//  Copyright Mr. Programmer 2010. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

- (IBAction)digitPressed:(UIButton *)sender
{
}

- (IBAction)operationPressed:(UIButton *)sender
{
}

@end
```

Let's take a timeout here and look at a neat debugging trick we can use in our program. There are two primary debugging techniques that are valuable when developing your program. One is to use the debugger. It's super-powerful, but outside the scope of this document to describe. You'll be using it a lot later in the class. The other is to "`printf`" to the console. The development kit provides a simple function for doing that. It's called `NSLog()`.

We're going to put an `NSLog()` statement in our `operationPressed:` and then run our calculator and look at the console (where `NSLog()` outputs to) just so you have an example of how to do it. `NSLog()` looks almost exactly like `printf` (a common C function). The 1st argument is an `NSString` (*not* a `const char *`, so don't forget the `@`), and the rest of the arguments are the values for any `%` fields in the first argument. A new, non-`printf` `%` field is `%@` which means the corresponding argument is an `NSString`. Let's put the following silly example in `operationPressed:`

```
- (IBAction)operationPressed:(UIButton *)sender
{
    NSLog(@"The answer to %@, the universe and everything is %d.", @"life", 42);
}
```

When we click on an operation button in our running application, it will print out "`The answer to life, the universe and everything is 42.`" So where does this output go? Go to the Run menu in Xcode and choose Console. It will bring up a window. That's where the output goes. You can even click "Build and Run" (or "Build and Debug") in that window to run your application from there. Try it now. Click on an operation in your running application and hopefully you should see something like this:



```
[Session started at 2010-03-31 11:03:18 -0700.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1461.2) (Fri Mar  5 04:43:10 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".sharedlibrary apply-load-rules all
Attaching to process 4392.
2010-03-31 11:03:21.450 Calculator[4392:207] The answer to life, the universe and everything is 42.
(gdb)
```

Okay, back to our application. (You can delete the `NSLog()`.)

45. Let's do the real implementation of `operationPressed:` first. Note that the argument to `operationPressed:` is the `UIButton` that is sending the message to us. We will simply ask the `sender` for its `titleLabel` (`UIButton` objects happen to use a `UILabel` to draw the text on themselves), then ask that `UILabel` that is returned what it's `text` is. The result will be an `NSString` with a `+` or `*` or `=` or `sqrt` depending the title of the `UIButton`.

```
    NSString *operation = [[sender titleLabel] text];
```

Note the "nesting" of message sending. This is quite usual and encouraged.

46. Next we need ask our `brain` to perform that operation (we'll get to the setting of the operand in a minute). First we need our `brain`!. Where is it? We have an instance variable for it (called `brain`), but we never set it! So let's create a method (somewhere ***earlier*** **in `CalculatorViewController.m` than we're going to use it**, since it's private) that creates and returns our `brain`. Put it right after `@implementation`.

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

Note the `if (!brain)` part. Basically we only want to create one `brain`, so we only do the creation part if the `brain` does not exist. We create the `brain` by `alloc`ating memory for it, then `init`ializing it. We'll talk much more about memory management and the creation and destruction of objects next week. Don't worry about it for now.

47. Now that we have a method in our `CalculatorViewController.m` that returns a `CalculatorBrain` (our Model) for us to use, let's use it.

```
    double result = [[self brain] performOperation:operation];
```

Again, notice the nesting of `[self brain]` inside the other message send to `performOperation:`:.

48. We have the `result` of our `operation`, we just need to put it into our `display` now. That's easy too. We just send the `setText:` message to our `display` outlet (remember, it's wired up to the `UILabel` in our View). The argument we're going to pass is an `NSString` created using `stringWithFormat:`. It's just like the very common C function `printf()` but for `NSString` objects. If you don't know what `printf()` is, Google it! Note that we are sending a message directly to the NSString clas (i.e. not an instance of an NSString, but the class itself). That's how we create objects. We'll talk a lot more about that next week.

```
- (IBAction)operationPressed:(UIButton *)sender
{
    NSString *operation = [[sender titleLabel] text];
    double result = [[self brain] performOperation:operation];
    [display setText:[NSString stringWithFormat:@"%g", result]];
}
```

There's one other thing that happens when an operation button is pressed which is that if the user is in the middle of typing a number, that number gets "entered" as the operand for the next `operation`. We're going to need another instance variable to keep track whether a user is in the middle of typing a number. We'll call it `userIsInTheMiddleOfTypingANumber` (a good long, self-documenting name).

49. Switch back to `CalculatorViewController.h` and add the instance variable `userIsInTheMiddleOfTypingANumber`. It's type is going to be `BOOL` which is Objective-C's version of a boolean value (the original ANSI-C had no concept of a boolean, so this is what the inventors of Objective-C decided to call their boolean value). It can have two values, `YES` or `NO` and can be tested implicitly.

```
@interface CalculatorViewController : UIViewController
{
    IBOutlet UILabel *display;
    CalculatorBrain *brain;
    BOOL userIsInTheMiddleOfTypingANumber;
}
```

50. Now let's add some code to `operationPressed:` which simply checks to see if we are in the middle of typing a number and, if so, updates the operand of the `CalculatorBrain` to be what the user has typed (then we'll note that we are no longer in the middle of typing a number anymore).

```
- (IBAction)operationPressed:(UIButton *)sender
{
    if (userIsInTheMiddleOfTypingANumber) {
        [[self brain] setOperand:[[display text] doubleValue]];
        userIsInTheMiddleOfTypingANumber = NO;
    }
    NSString *operation = [[sender titleLabel] text];
    double result = [[self brain] performOperation:operation];
    [display setText:[NSString stringWithFormat:@"%g", result]];
}
```

But when does `userIsInTheMiddleOfTypingANumber` ever get set? Well, it gets set when the user starts typing digits. Great transition into `operationPressed:`, eh? So there are two different situations when a digit gets pressed. Either the user is in the middle of typing a number, in which case we just want to append the digit they typed onto what's been typed before, or they are not, in which case we want to set the display to be the digit they typed and note that they are now in the middle of typing a number.

51. Let's add our first line of code to `digitPressed:`. It retrieves the digit that was pressed from the `titleLabel` of the `UIButton` that sent the `digitPressed:` message (the `sender`).

```
- (IBAction)digitPressed:(UIButton *)sender
{
     NSString *digit = [[sender titleLabel] text];
}
```

52. Now that we have the digit, let's either append it to what's already been typed (using the `NSString` method `stringByAppendingString:`) or set it to be the new number we're typing and note that we have started typing.

```
- (IBAction)digitPressed:(UIButton *)sender
{
     NSString *digit = [[sender titleLabel] text];

     if (userIsInTheMiddleOfTypingANumber)
     {
         [display setText:[[display text] stringByAppendingString:digit]];
     }
     else
     {
         [display setText:digit];
         userIsInTheMiddleOfTypingANumber = YES;
     }
}
```

Whew! That's it, we're done. Now it's time to build and see what syntax errors (if any) we have.

Part VII: Build and Run

53. Click Build and Run in Xcode. You should have a functioning calculator!

If you have made any errors entering any of the code, Xcode will point them out to you in the Build Results window (first item in Xcode's Build menu). Hopefully you can interpret them and fix them. If your code compiles and runs but does not work, another common place to look for problems is with your connections in Interface Builder.

If it's still not working, feel free to e-mail us and we'll try to help. We'll also be having some office hours (see website for details on when and where they are).

Brief

Here's a brief outline of the walkthrough. If you saw and understood what went on in class, this may be sufficient for you. It is hyperlinked into the detailed walkthrough for easy reference.

If you choose this brief walkthrough and it does not work, please go back to the detailed walkthrough before sending e-mail to the class staff. Also, if you choose to use this brief walkthrough and don't really understand what you're doing, the rest of the first homework assignment might be difficult. The detailed walkthrough explains what is behind each step. This one does not.

1. Create a new View-based project in Xcode named Calculator.
2. Build and Run it.
3. Create the class which is going to be your MVC Model by choosing New File ... from the File menu and creating an Objective-C class (subclass of `NSObject`) called `CalculatorBrain`. It might be a good idea to drag the `.h` and `.m` file for this class into Classes section in the Groups & Files area if it didn't land there already. We'll implement this class later.
4. Build and Run.
5. Open up `CalculatorViewController.h` in Xcode and add a `UILabel` outlet called `display` for the calculator's display and two action methods, `digitPressed:` and `operationPressed:`. These will be used to hook your MVC Controller to your MVC View. Also add the `BOOL` instance variable `userIsInTheMiddleOfTypingANumber` since you'll need it later.
6. You'll also need an instance variable in your `CalculatorViewController` for your MVC Model. Name it `brain`. It is of type `CalculatorBrain *`.
7. Build and Run. You'll have few warnings because the compiler will have noticed that you have declared some methods in `CalculatorViewController.h` that you haven't yet implemented in `CalculatorViewController.m`. As long as they are only warnings and not errors in the code you've typed, your application will still run in the simulator. The UI is still blank of course.
8. Open `CalculatorViewController.xib` (which contains your MVC View), drag a `UIButton` out of the Library window, and hook it up via the `digitPressed:` message to File's Owner in Interface Builder's main window (File's Owner is your `CalculatorViewController`). Then copy and paste that button 9 times, double-click on the buttons to set their titles to be the digits, then arrange the buttons into a calculator keypad.

9. Drag out another `UIButton` from the Library window, hook it up to File's Owner via the `operationPressed:` message. Copy and paste it a few times and edit the button titles for all of your operations.

10. Drag out a `UILabel` from the Library window and position it at the top of your view. This will be your calculator's `display`. Drag a connection to it from File's Owner.

11. Save your `.xib` file and go back to Xcode. Build and Run. You should have a calculator with buttons now, but the application will crash when you touch the buttons because you haven't implemented `digitPressed:` or `operationPressed:`.

12. Time to implement the Model. In `CalculatorBrain.h`, add a `double` instance variable for the `operand` and two methods, one to set the `operand` called `setOperand:`, and one to perform an operation called `performOperation:`.

13. In `CalculatorBrain.m`, add the implementation for `setOperand:` to set the instance variable `operand`.

14. Also add the implementation for `performOperation:` and it's sister method `performWaitingOperation`. `performWaitingOperation` needs to appear *before* `performOperation:` in the `.m` file. Make sure you understand how these work or you will have difficulty with the rest of the homework.

15. Build and Run and fix any compile problems. Your calculator will still crash when you click on buttons because there is no implementation for the MVC Controller (`CalculatorViewController`) yet.

16. Type in the implementation of your `CalculatorViewController`. It has three methods ... `digitPressed:`, `operationPressed:` and the helper method `brain`. If you want, you can throw in an `NSLog()` to verify that your action methods are being called. Note that `brain` is private, so it needs to be defined earlier in the `.m` file than where it is used (in `performOperation:`). Again, make sure you understand how these work or the rest of the homework might be a problem for you. See the detailed walkthrough to get the complete story if need be.

17. Build and Run. Your calculator should work! If not, try fixing the compiler errors (you can see them in the Build Results window which is brought up by the first menu item in the Build menu). The most common problem at this point would be things not being wired up in Interface Builder correctly. Try using `NSLog()` to help find out if that is the case. If that looks okay, double-check all the code that you typed in. Or try the detailed walkthrough.