

# Assignment I Walkthrough

---

## Objective

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

---

## Goals

1. Downloading and installing the iOS4 SDK.
2. Creating a new project in Xcode.
3. Defining a **Model**, **View** and **Controller** and connecting them together.
4. Using **Interface Builder** to create a user-interface.

---

## Materials

By this point, you should have been sent an invitation to your Stanford e-mail account to join the iPhone University Developer Program. You must accept this invitation and download the latest version of the iOS 4 SDK along with the latest version of Xcode.

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 past the weekend and you cannot get the SDK downloaded and installed, it is unlikely you'll finish this assignment on time.

---


## Brief

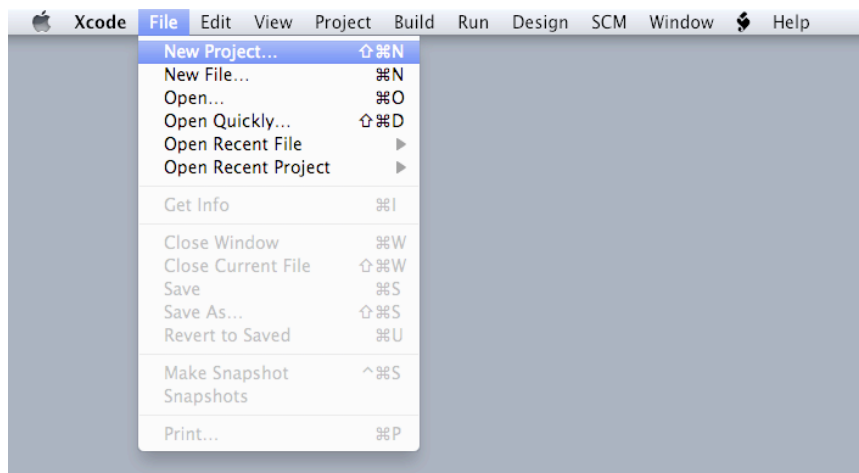
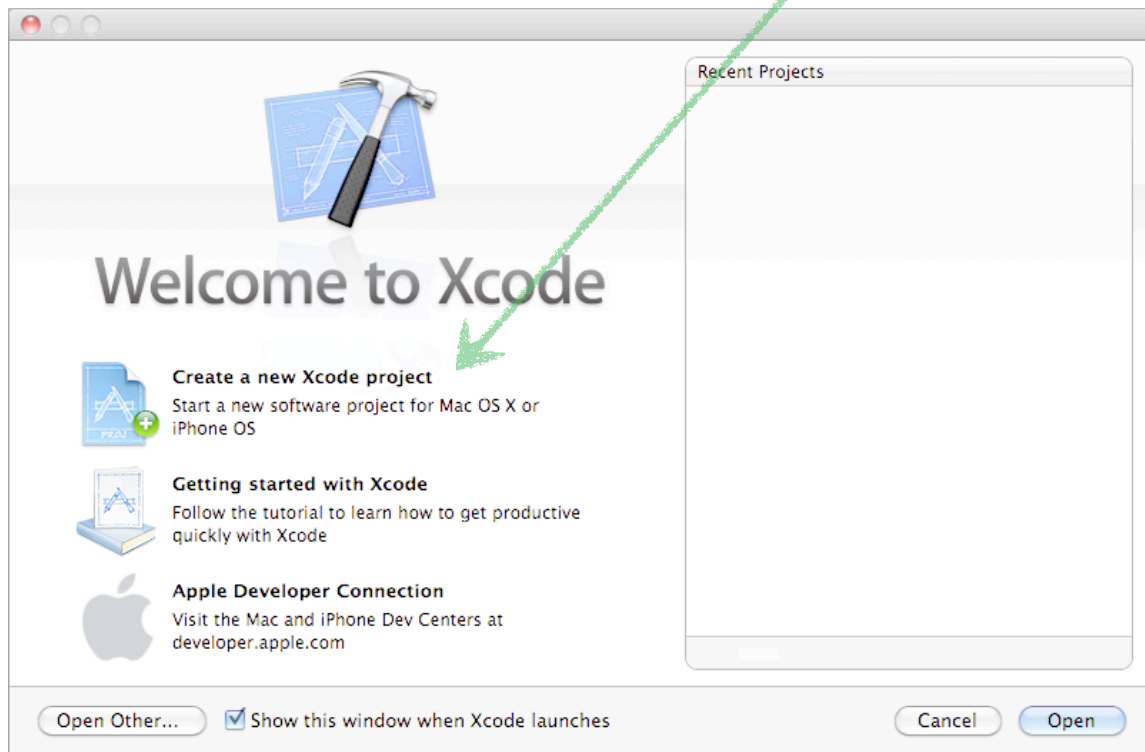
If you were in class on Thursday and saw this walkthrough, you may feel like you can get by with a [much briefer version](#) included at the end of this document. You can always refer back to the detailed one if you get lost. The devil is often in the details, but sometimes you have to learn from the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

---

## Detailed Walkthrough

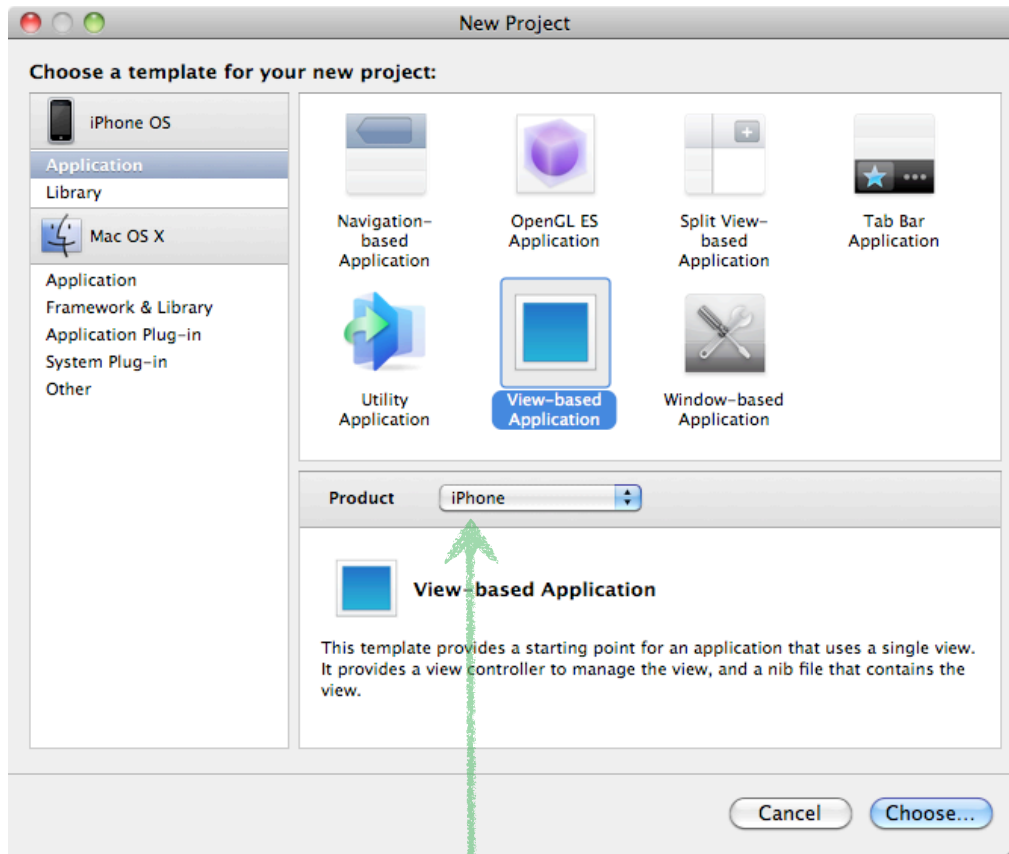
### Part I: Create a new project in Xcode

1. Launch /Developer/Applications/Xcode  .
2. From the splash screen that appears, choose **Create a new Xcode project**.



You can also create a new project by choosing **New Project ...** from the **File** menu at any time.

3. In the dialog that appears ...

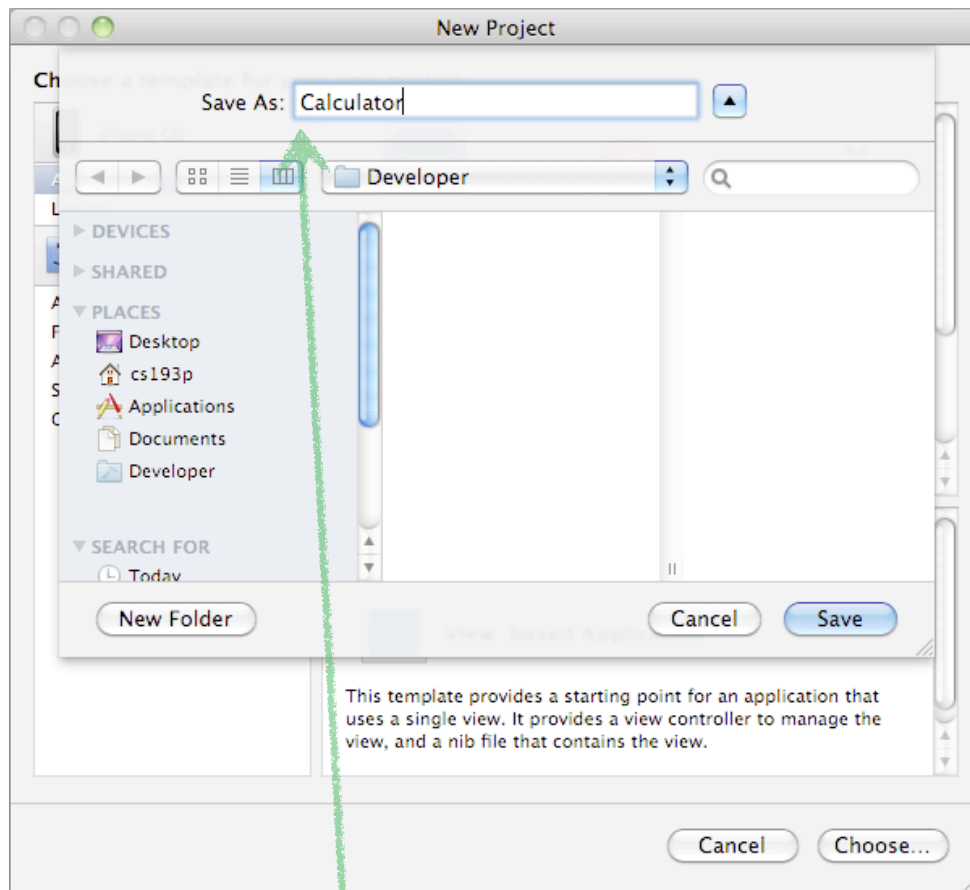


... click on  (Product should be set to **iPhone**), then .

The View-based Application template in Xcode creates an empty **View** as well as a subclass to be your **Controller** (with a little bit of template code in it).

The other templates in this dialog create applications with either no **View/Controller** (Window-based Application) or with a more sophisticated **Controller** and **View** (e.g. Navigation-based Application, Tab Bar Application or Split View-based Application). In this course, we will only be using the simple View-based Application template or the Window-based Application template (in which case we will build our more sophisticated **Controllers** in source code rather than using these templates).

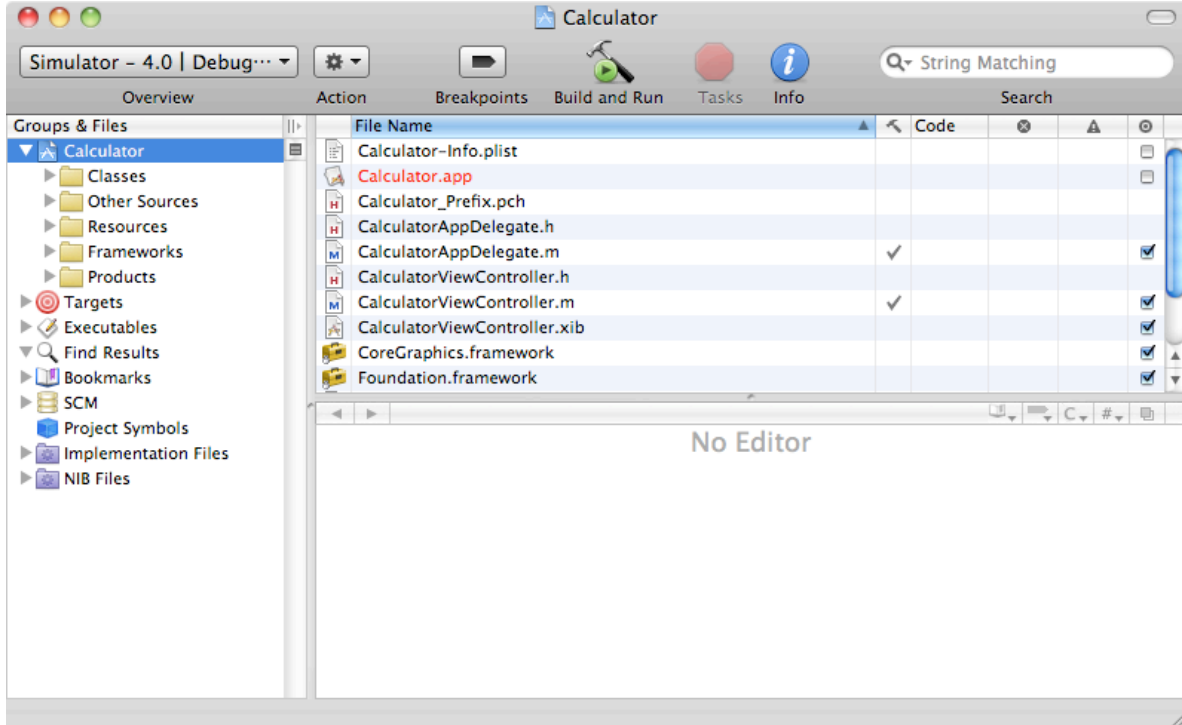
4. In the file chooser that is presented ...




... do the following:

- navigate to a place where you want to keep all of your application projects for this course (a good place is `~/Developer/cs193p` where `~` means “your home directory”)
- in the **Save As:** field, type the name **Calculator** (for the rest of the walk-through to make sense to you, it is highly recommend to call this project “Calculator”)
- click **Save** .

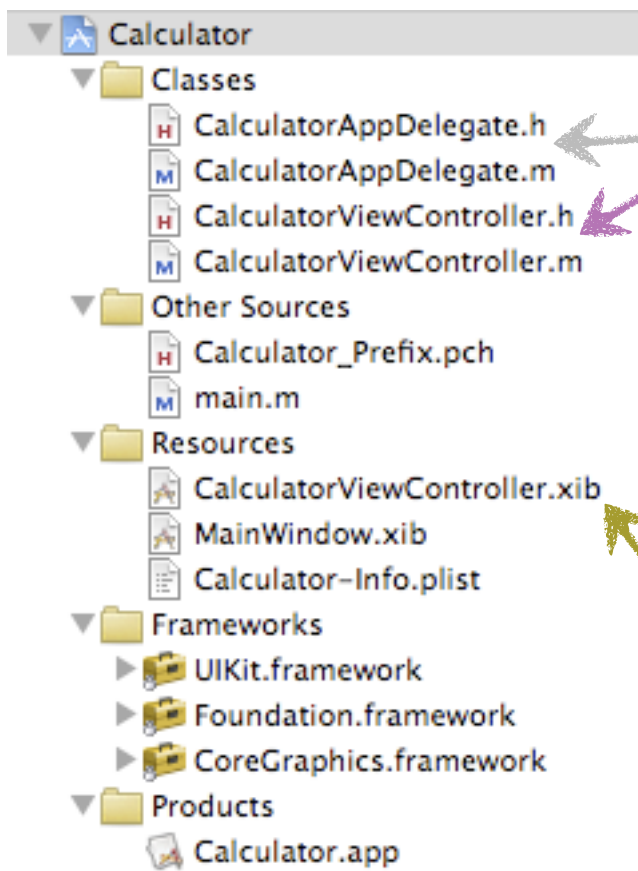
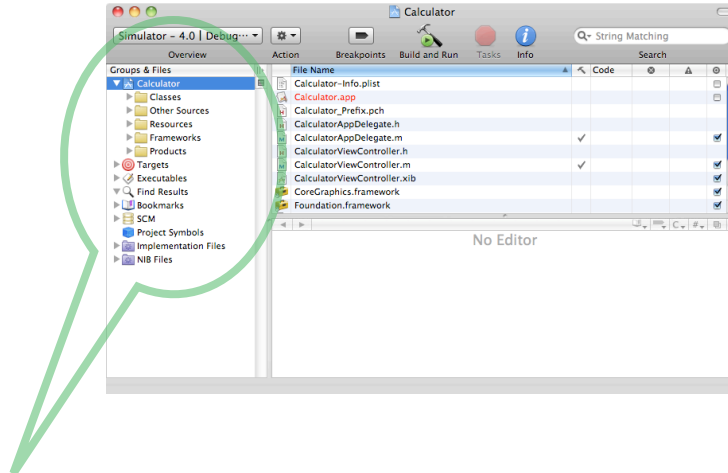
You have successfully created your first iOS project! The following window will appear.



5. 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 application. If this works, you have likely successfully installed the SDK. If it does not, check with a TA.



6. Go back to Xcode now (you can quit the iPhone Simulator application). Notice in the upper left hand corner, there is a tree of folders called **Groups & Files**. This is where all the files in your application are managed. Click on the little folders to expand them as shown.



Note that in the **Classes** section Xcode has automatically created `.h` and `.m` files for two different classes:

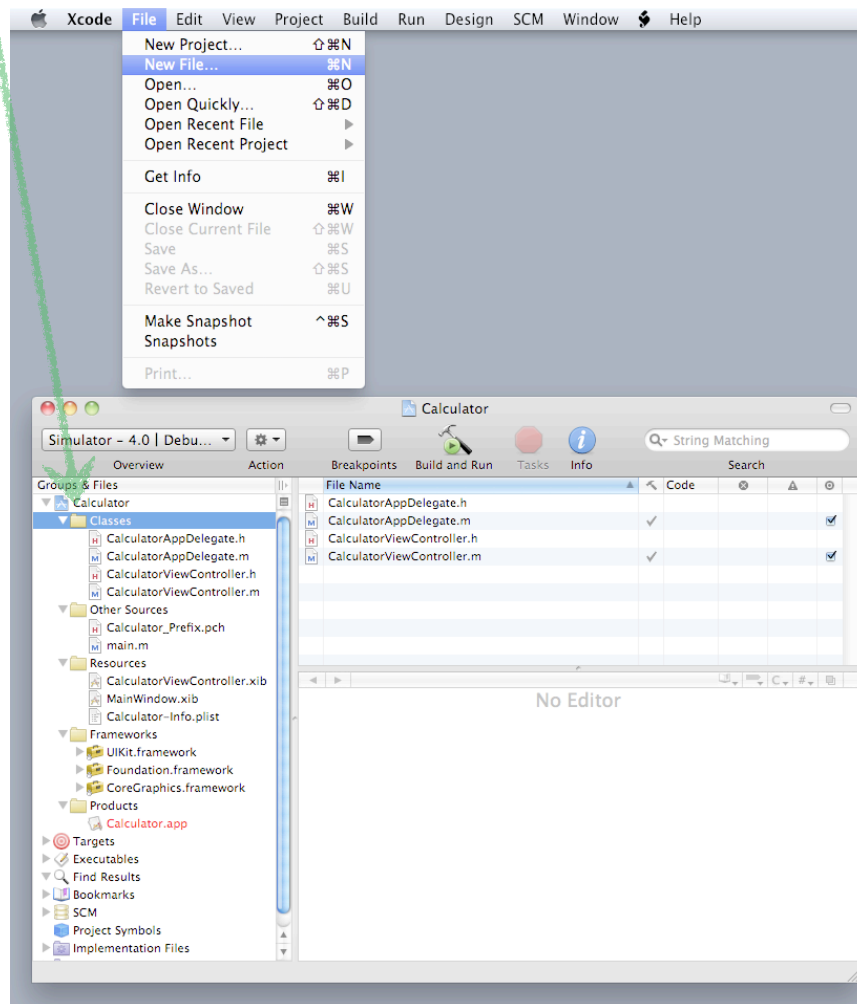
`CalculatorAppDelegate` and `CalculatorViewController`. Don't worry about `CalculatorAppDelegate` for this assignment. The second one, `CalculatorViewController`, is the source code for our `Controller`.

We'll create a new class in a moment called `CalculatorBrain` to be our `Model`.

So what about our `View`? We'll create that later using a graphical tool (which will store our work in the file `CalculatorViewController.xib`).

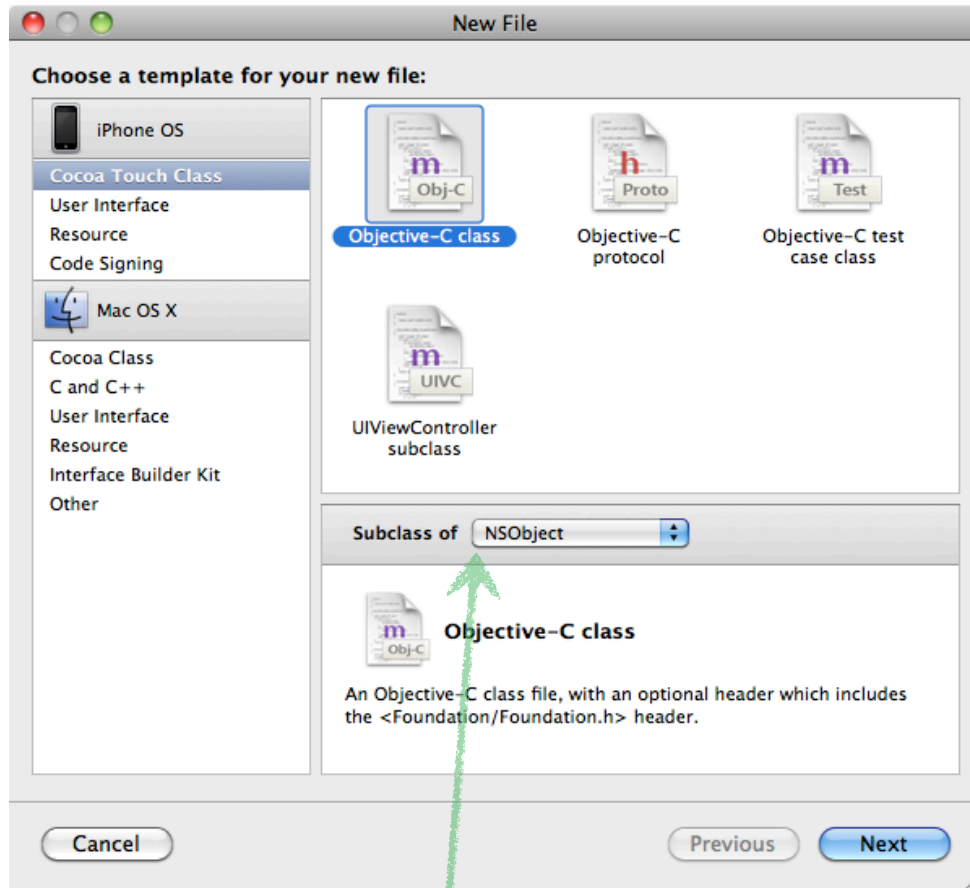
## Part II: Create a new class to be our **Model**

7. Here's how we create a new Objective-C class to be our **Model**. Click on the **Classes** folder in the **Groups & Files** area and then select **New File ...** from the **File** menu.



Selecting the **Classes** folder before choosing **New File ...** simply lets Xcode know that we want the newly created file to appear in the **Classes** folder. If you accidentally (or intentionally) create it in some other folder, you can drag it back to **Classes** at any time. Objective-C class files do not have to be in the **Classes** folder in order to be compiled and included in your program. The groupings in this **Groups & Files** area are up to you.

A dialog will appear to let you choose what sort of new file you want to add to your project. In this case, we want to create a new Objective-C class to be our [Model](#).

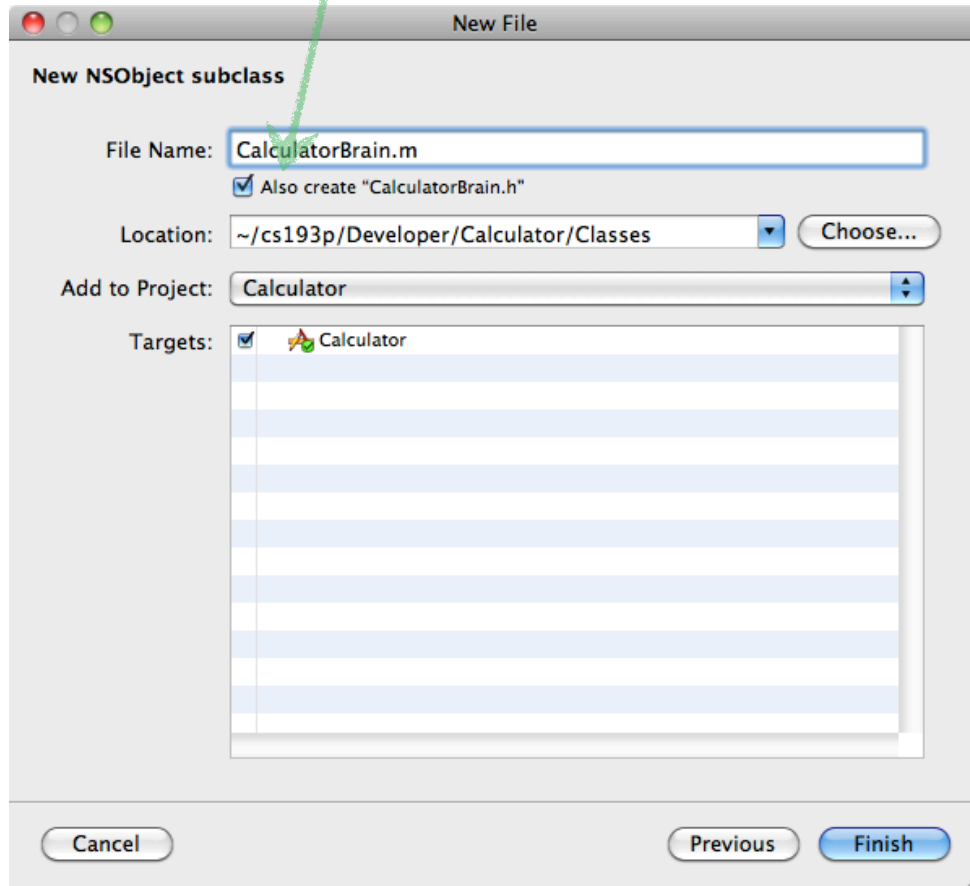


8. Click  **Objective-C class** (**Subclass of set to NSObject**), then .

Pretty much all objects in iOS development are subclasses (directly or indirectly) of NSObject.



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



Then click **Finish**.

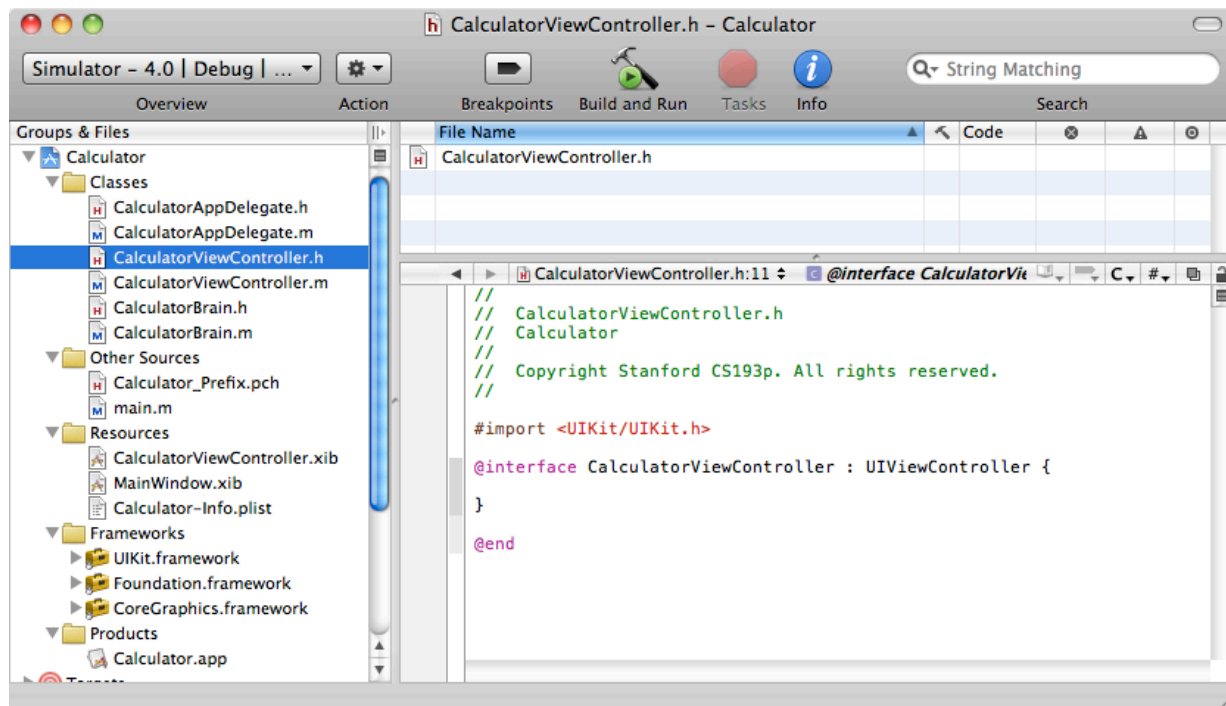
So now our **Model** is created (though obviously not implemented). Let's take a timeout from our **Model** and go back to our **Controller** to type in some declarations for the connections we need to make between our **Controller** and our **View**.

### Part III: Define the connections to/from the 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**.

10. 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):



Notice that Xcode has already put the `#import` of `UIKit` that we need and made our `CalculatorViewController` be a subclass of `UIViewController`. **Controller** objects are always a subclass (directly or indirectly) of `UIViewController`. That's all good.

But our `CalculatorViewController`'s header file still needs to define the following:

- a. *outlets* (instance variables in our **Controller** that point to objects in our **View**)
- b. *actions* (methods in our **Controller** that are going to be sent to us from our **View**)
- c. an instance variable in our **Controller** that points to 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.)

11. Let's add the *outlet* which enables our `CalculatorViewController` to talk to a `UILabel` (an output-only text area) representing our calculator's display in our `View`. We'll call that *outlet* `display`.

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

@end
```

Note the keyword `IBOutlet`. This keyword doesn't do anything except to identify the *outlets* to the graphical tool we will use (**Interface Builder**) to hook our `Controller` up to our `View`.

12. Now let's add an instance variable called `brain` 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`.

```
#import <UIKit/UIKit.h>
#import "CalculatorBrain.h"

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

@end
```

13. 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;
}

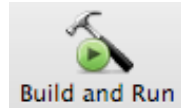
- (IBAction)digitPressed:(UIButton *)sender;
- (IBAction)operationPressed:(UIButton *)sender;

@end
```

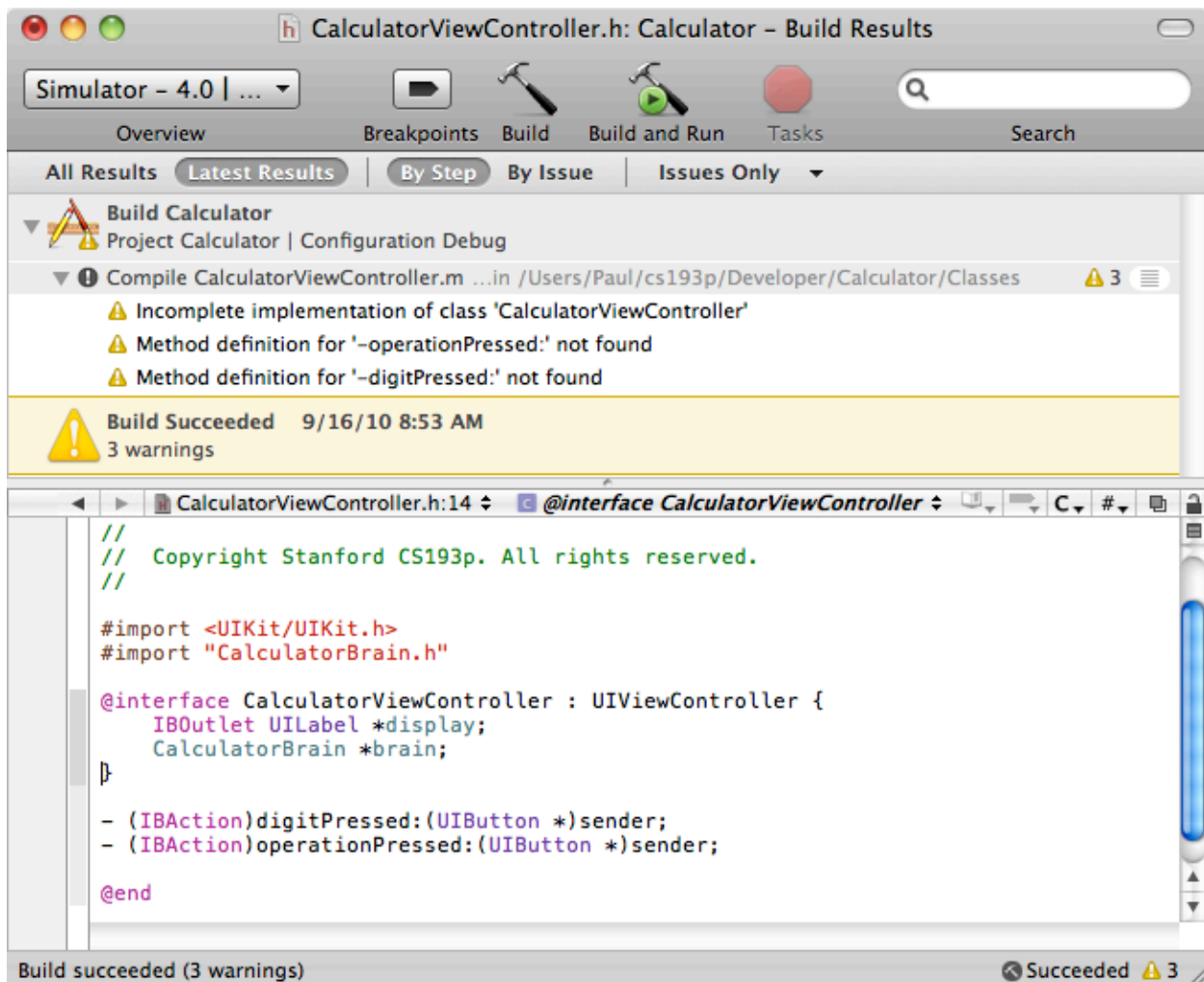
`IBAction` is the same as `void` (i.e. this method has no return value) except that the **Interface Builder** program knows to pay attention to this method.

Note also that the (only) argument to each method is a `UIButton` object (the object that is sending this message to our `Controller` when the user touches it in our `View`). Knowing which `UIButton` is sending the *action* is a must because otherwise we wouldn't know which digit is being pressed or which operation is being pressed.

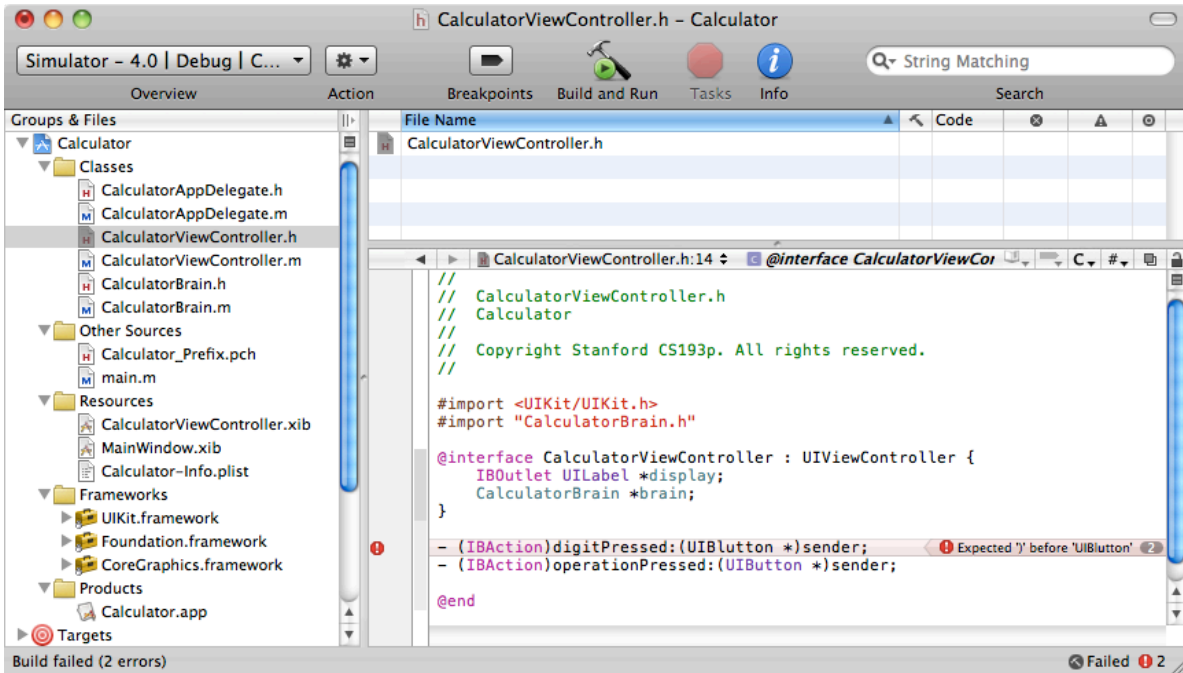
We may need more instance variables for our `CalculatorViewController` as we get into its implementation, but, for now, we've covered the connections of our MVC design from/to our `View` to/from our `Controller`.



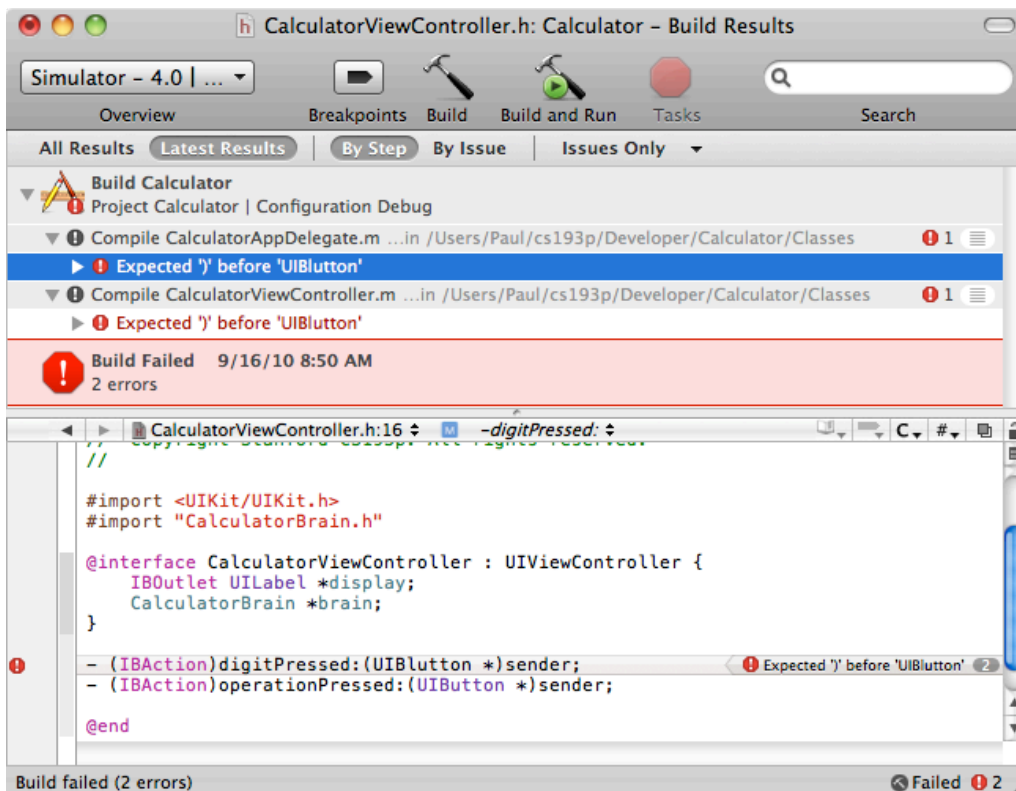
14. This would be a good time to **Build and Run** your application again. If you've typed the above all in correctly, you'll still get warnings because the compiler will notice that you've declared methods in your `Controller` that you have not yet implemented. You can check these warnings out in the **Build Results** window (from the **Build** menu). Note that it still says **Build Succeeded** (because these are only warnings, not errors).



If you have typed something in incorrectly, on the other hand, the little triangle in the bottom right corner will be a red circle instead (here `UIButton` is mistyped `UIBlutton`):



Clicking the red circle (or yellow triangle) will also bring up the **Build Results** window and show you what the error (or warning) is.

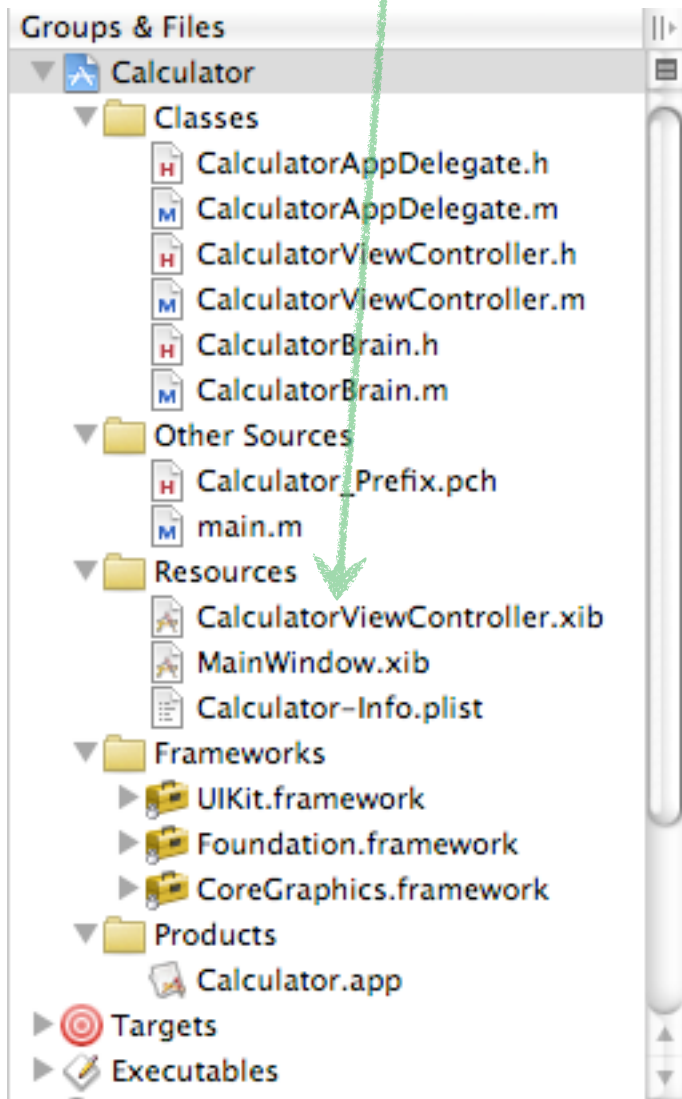


Now that we have declared our *outlets* and *actions* in our **Controller**, we can use the graphical tool **Interface Builder** to add some buttons and a display to our **View** and then “wire it up” to our **Controller**.

## Part IV: Create and wire up the **View** in Interface Builder

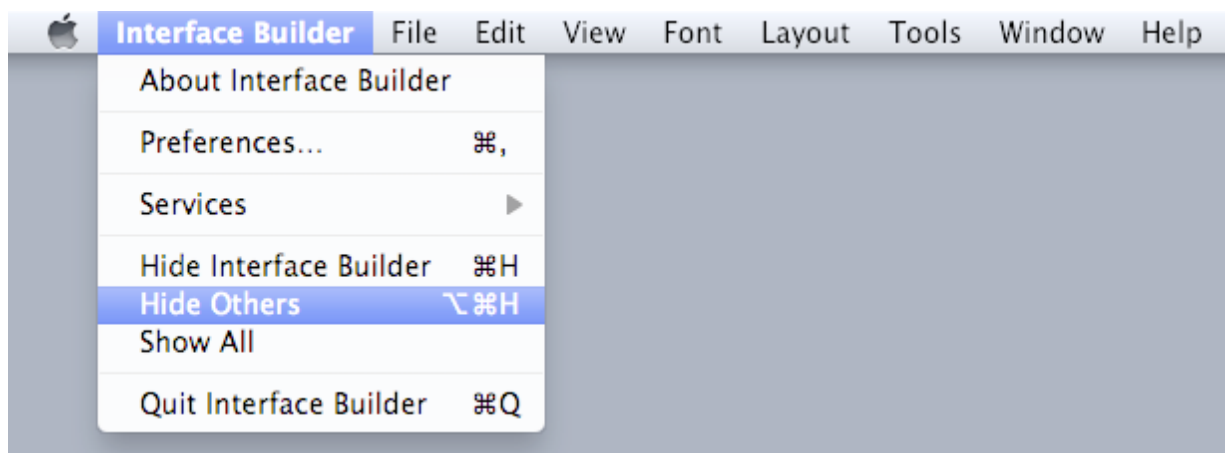
It's time to create the **View** part of our MVC design. We do not need to write any code whatsoever to do this, instead we use a tool called **Interface Builder**. When we created our project and told Xcode that we wanted a View-based project, it automatically created a template **Controller** (which 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 **Resources** section of the **Groups & Files** area of Xcode:



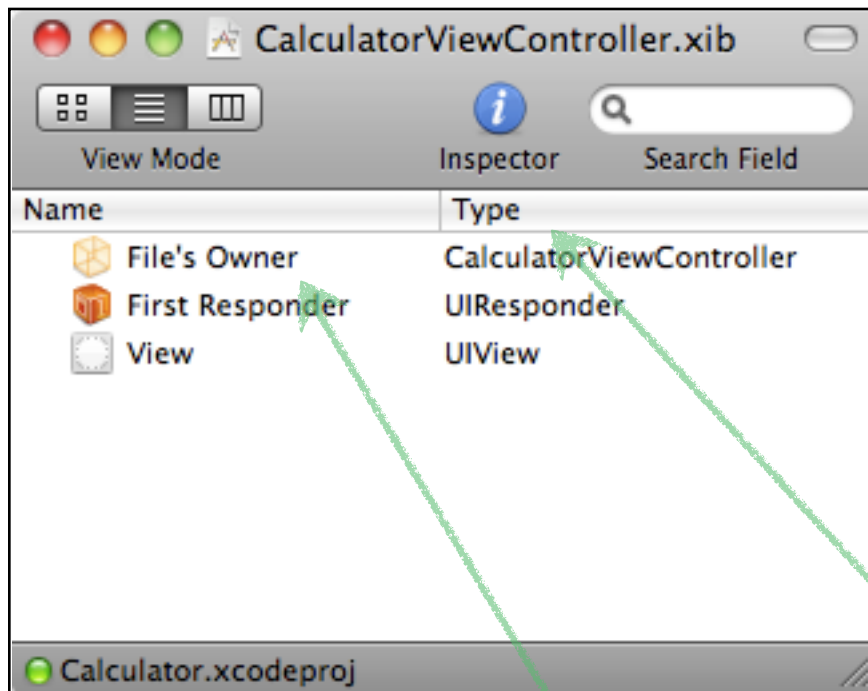
Clicking on the `.xib` file will open up another application: the graphical tool **Interface Builder**. We use it both to lay out our user-interface and to connect it up to our **Controller**.

16. 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**. You might also consider putting **Interface Builder** in its own space in **Spaces**.





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



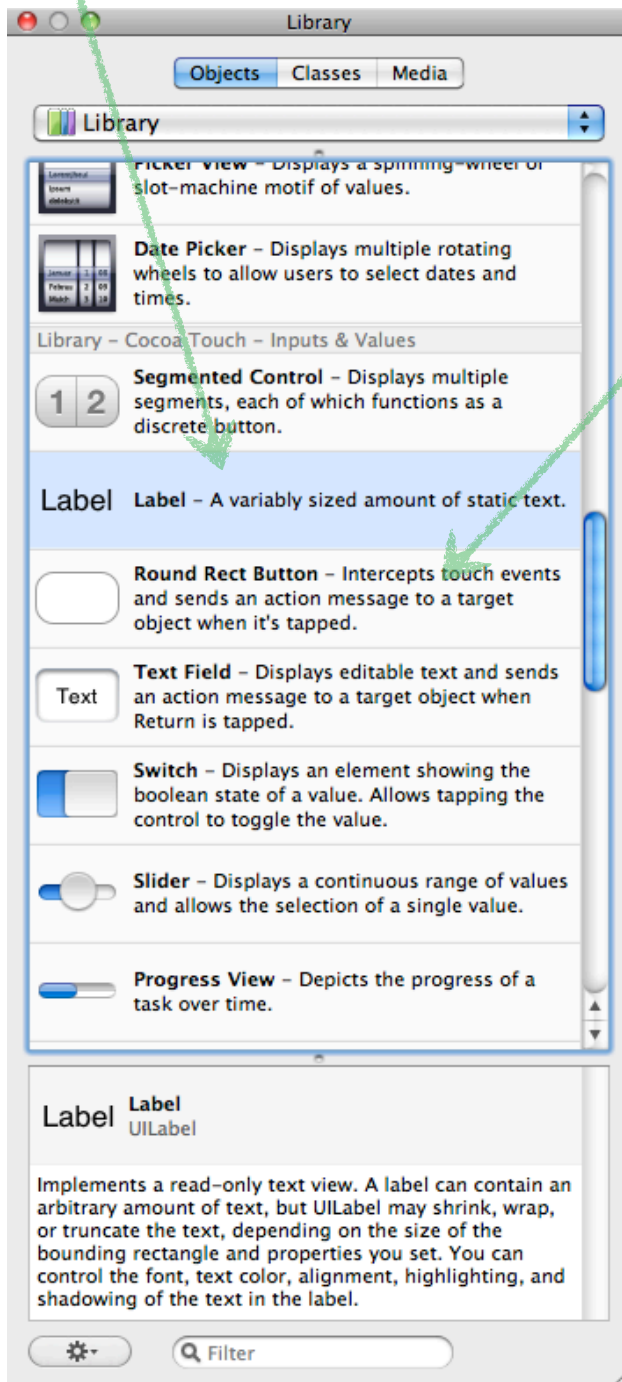
Where’s our **Controller**!? Well, since this is `CalculatorViewController.xib`, our `CalculatorViewController` is the **File’s Owner** (you can see that the **Type** of **File’s Owner** is shown as `CalculatorViewController`). So when we want to wire things up to our `CalculatorViewController`’s *outlets* and *actions* (instance variables and methods), **File’s Owner** is the icon that we’ll connect to.

Note the **View Mode** choices in the upper left corner of this window. You can choose to look at 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.

Ignore **First Responder** for now.

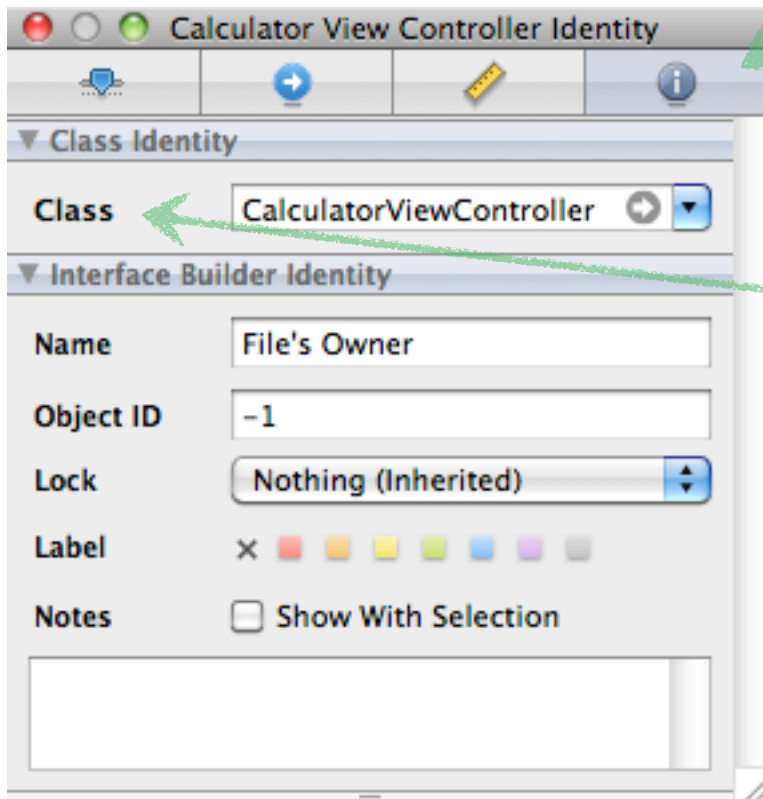
The other object in the list, **View**, is the top-level `UIView` in our “view hierarchy.” It is the top-level of our **View**. All `UIView` objects (`UIButton`, `UILabel`, etc.) are arranged in a hierarchy in which each has a *superview* and each may have any number of *subviews*. This **View** icon represents the *superview* of all the views we are going to arrange into our calculator’s interface (i.e., its display and its digit and operation buttons).

The next window in **Interface Builder** is the **Library** window. It is called that because it contains a library of objects from which you can select items to help build your **View**. 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**.




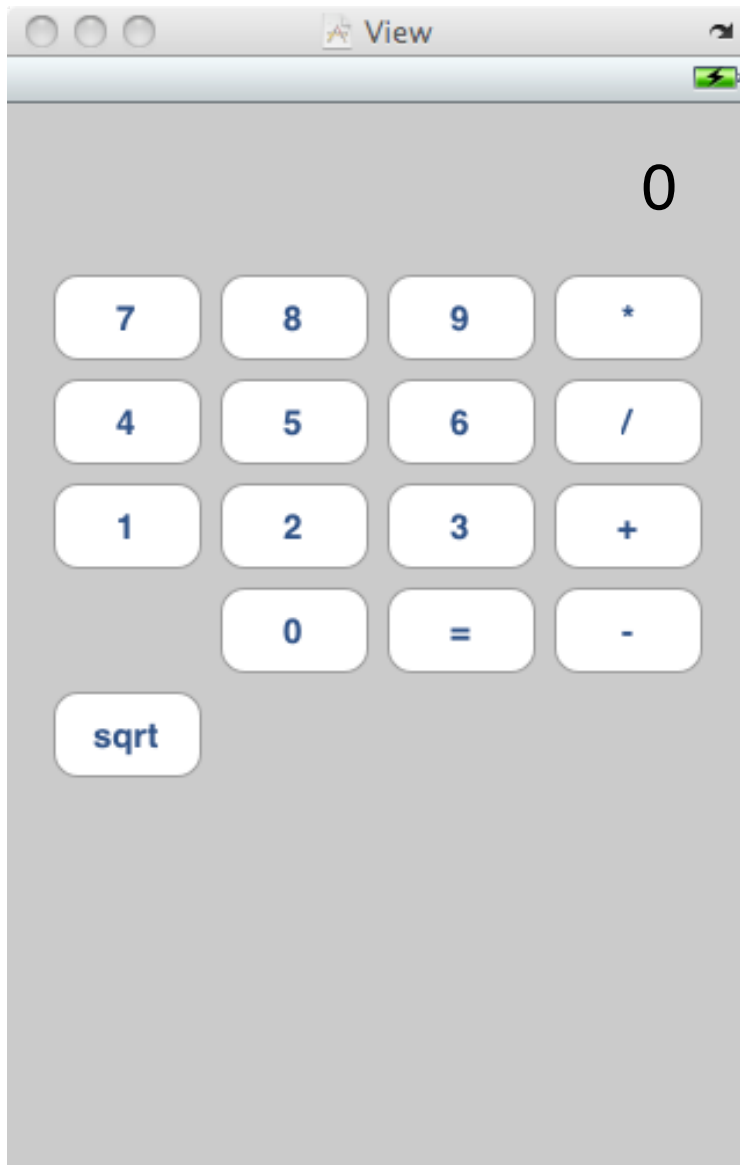
The last window is the **Inspector**. The contents of this window change depending on which object you have selected (it “inspects” the selected object). Since we start out with nothing selected, it just says **Empty Selection**. But if we click around on the objects in our main window, we’ll start seeing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

17. 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:




You can see that the **class** of our **File’s Owner** is **CalculatorViewController** as expected. Don’t change it!

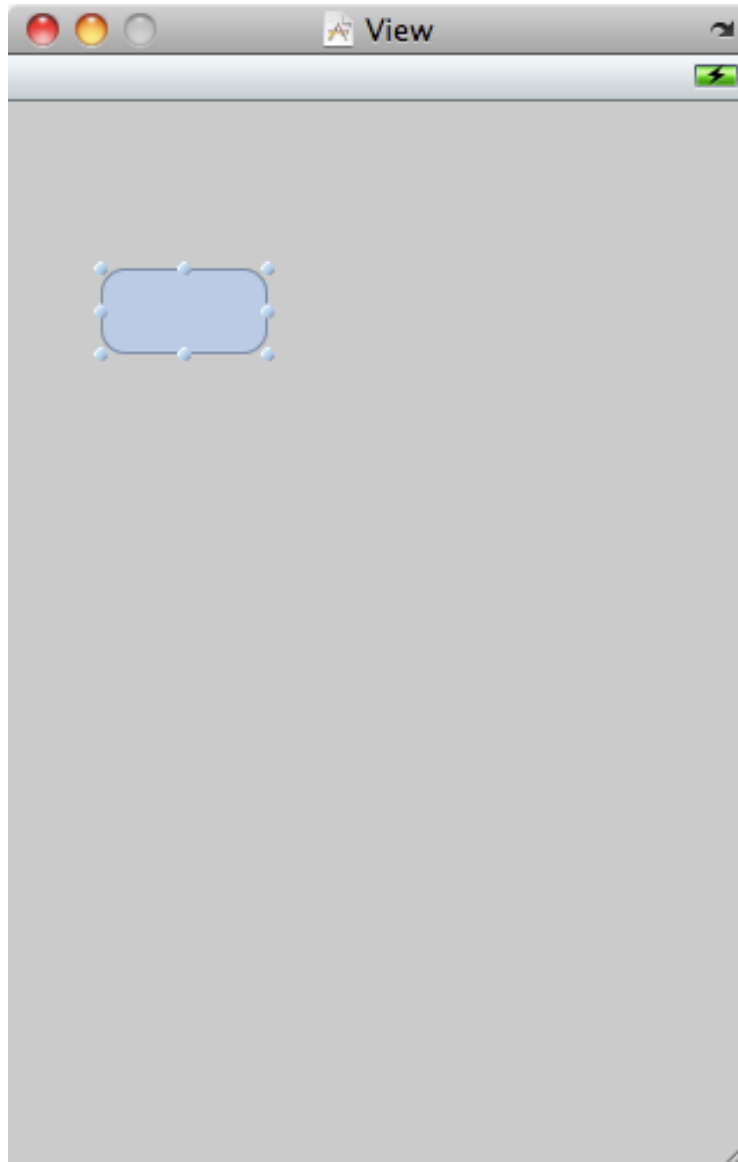
The only other window in **Interface Builder** is blank. It is showing the contents of the top-level of our **View** (i.e. the  **View** object that was listed in **Interface Builder**'s main window) and we haven't added any subviews to it yet. Our next step then, is to put some stuff in there. Here's 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):



The **0** at the top is our display (note that it is right-aligned). The other buttons are pretty self-explanatory.

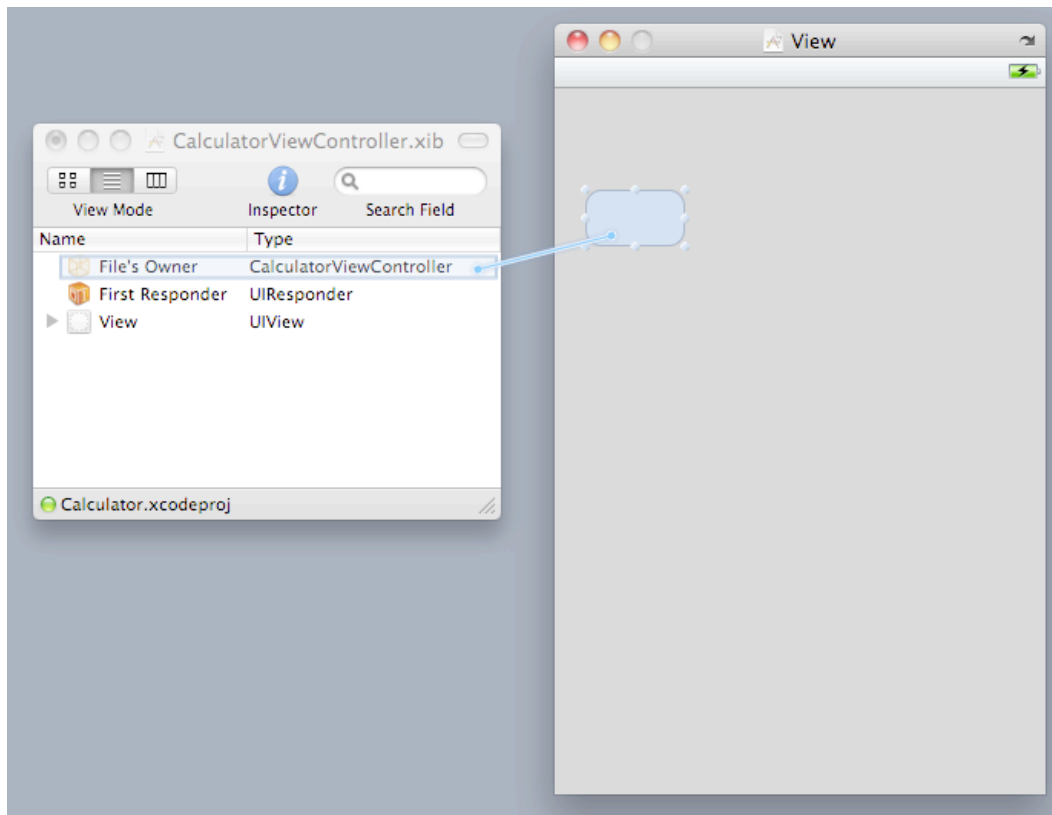
The gap below the **1** and above **sqrt** is left for you to fill in with part of your solution to your first homework assignment.

18. 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](#) in this document), then simply drag it out into our currently blank  **View**.

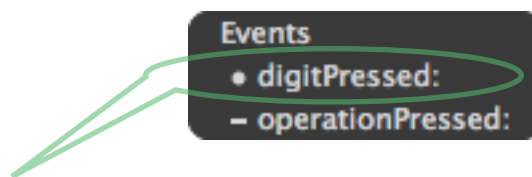


19. Now resize the `UIButton` to be 64 pixels wide (grab one of the little handles on the side of it), and then pick it up and move it toward the left edge. When you get close to the left edge, a vertical blue dotted line will appear letting you know that this is a pretty good left margin for the button. **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.

20. Okay, now for the most important part. Let's wire this button up to our `CalculatorViewController` (the 📁 **File's Owner** in the main window) so that it will send `digitPressed:` whenever the user touches it. Just hold down the control key and drag a line from the button to the 📁 **File's Owner**. 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.

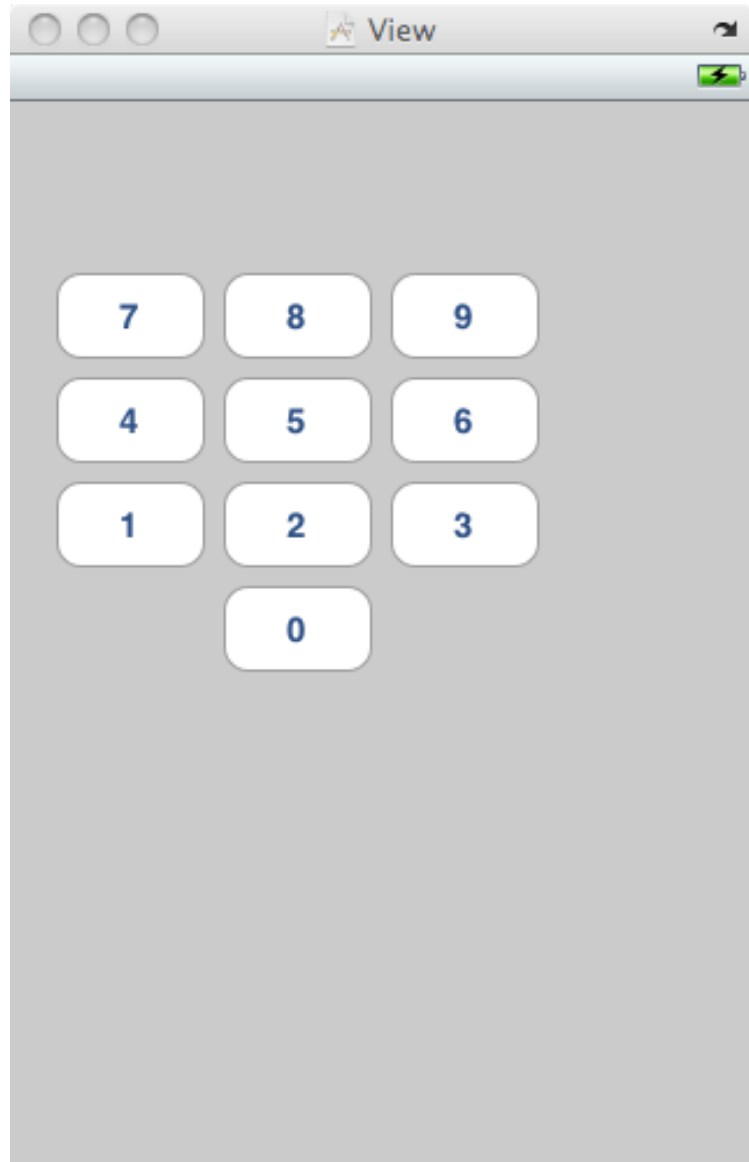


As you get close to the 📁 **File's Owner**, a blue box should appear around it. When you let go of the mouse button, the little window below should appear next to your mouse.




21. Pick `digitPressed:` and voila! Every time that button is touched by the user, it will send `digitPressed:` to your `CalculatorViewController`.

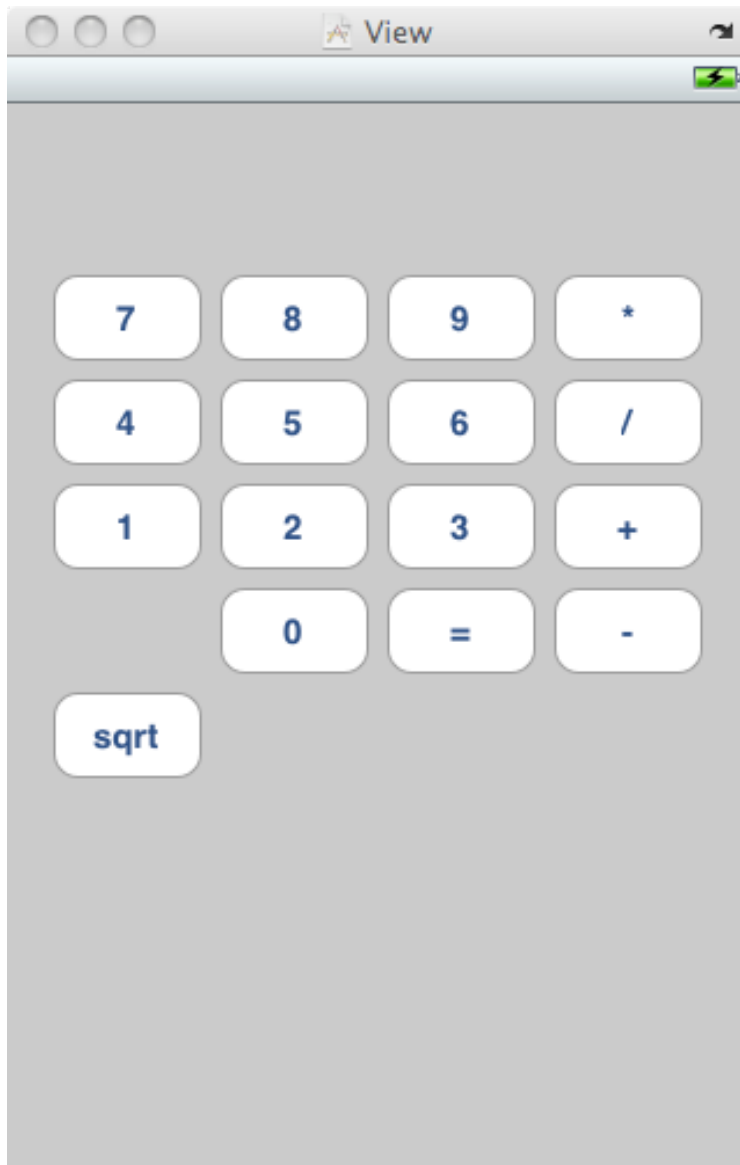
22. Now that you've made that connection, copy and paste that button 9 more times to make a total of 10 digit buttons. All of them will send **digitPressed:** because copying and pasting an object in **Interface Builder** maintains connections. Lay out the 10 buttons approximately as shown.
23. Double-click on the center of each button to set its title. Just use a single digit on each button. 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**.



Next we're going to do the operation buttons.

24. 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 the operation buttons). Resize the button to 64 pixels wide.
25. 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:**.
26. Now you can copy and paste this 5 times (you'll need \* / + - = and sqrt) and lay them out as shown on the next page. 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:

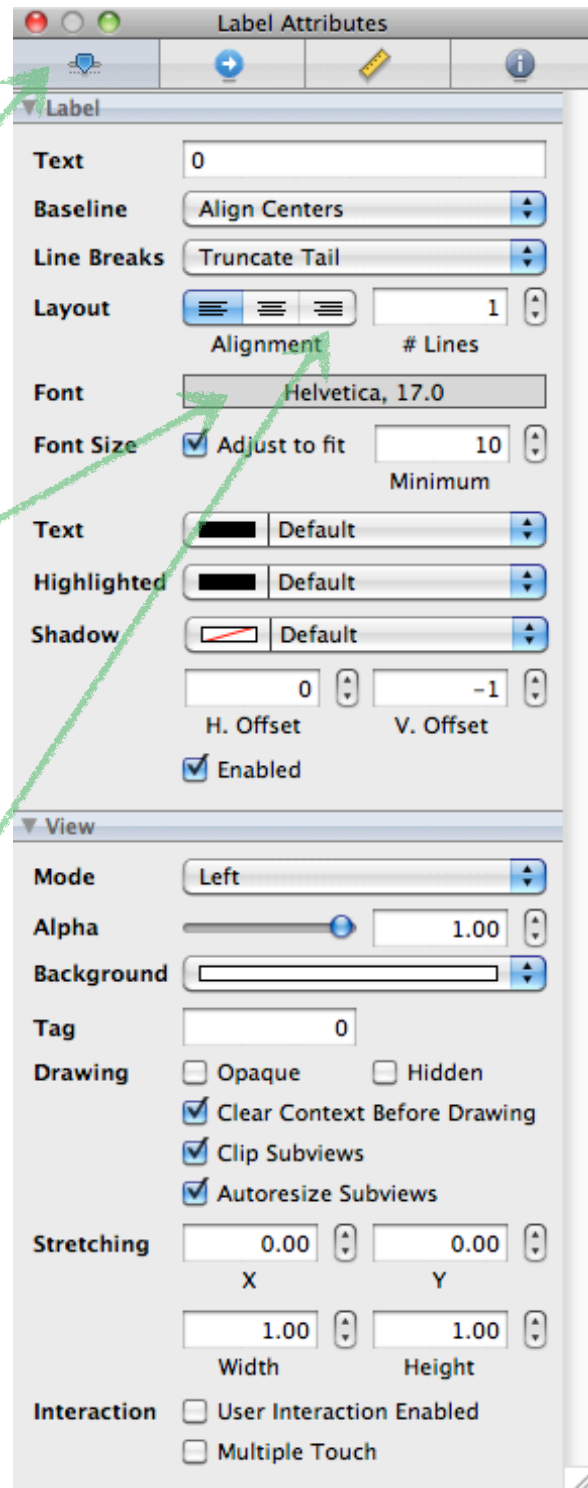




Almost there! We just need a display for our calculator.

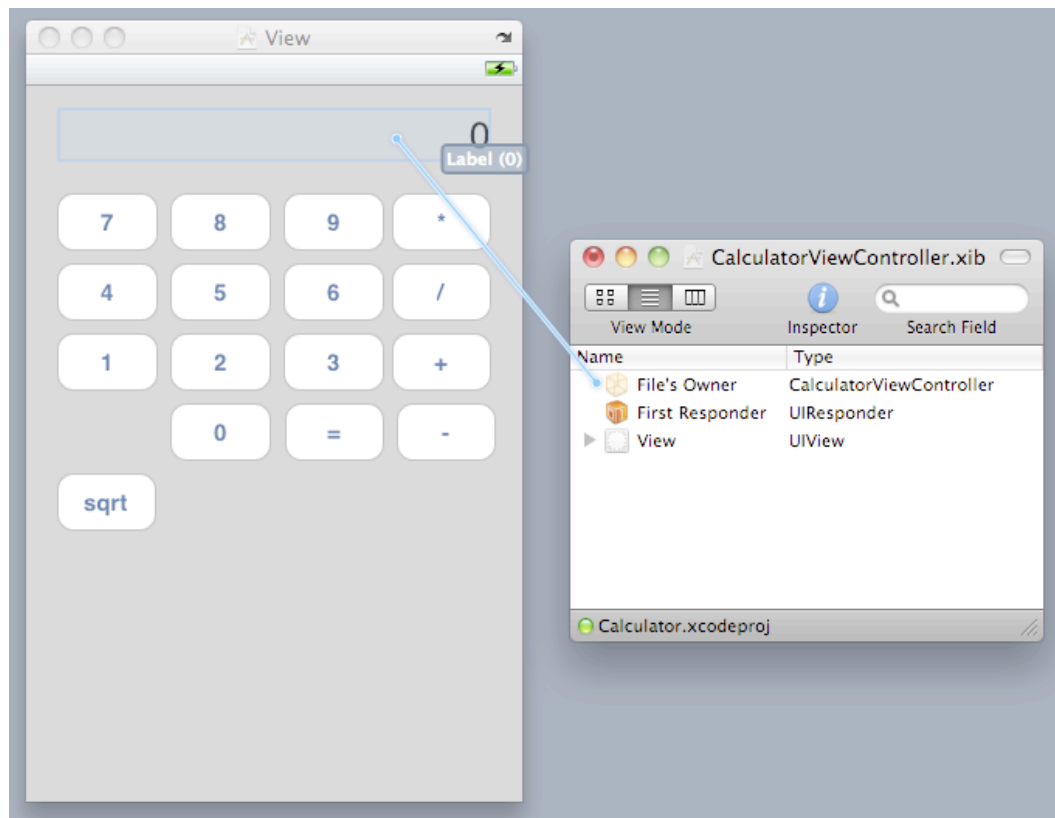
27. Drag out a label (**UILabel**) from the **Library** window and position and size it along the top of your UI. Double-click on it to change the text in there from “Label” to “0”.
28. This time we’re going to use the **Inspector** to change some things about the display. So make sure the UILabel is selected and then click on the left-most tab of the Inspector window. It should look like the image on the right.
29. Let’s start by making the font a little bigger by clicking the field next to where it says **Font**. This will bring up a font choosing panel you can use to change the font. 36 point would be a good size.
30. 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 **UIView**, its inspector also inherits the ability to set any properties **UIView** has.

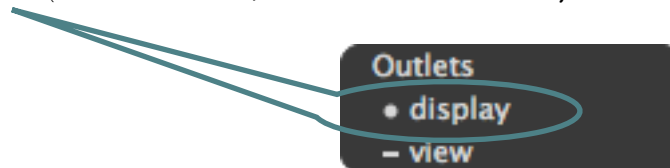


Okay, one last thing in **Interface Builder**. Our `CalculatorViewController` needs to be able to send messages to the `display` *outlet* to update it, so we need to hook it up to that instance variable in our `CalculatorViewController`. Doing this is similar to setting up the messages sent by the buttons, but we control-drag in the opposite direction this time.

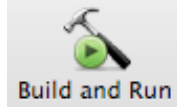
31. Control-drag from  **File's Owner** to the `UILabel`.



32. When you release the mouse, the following little window should appear. Choose `display` (the other *outlet*, `view`, is automatically set to be the top-level  **View**).



33. Save the file in **Interface Builder** and then you can quit and go back to Xcode.

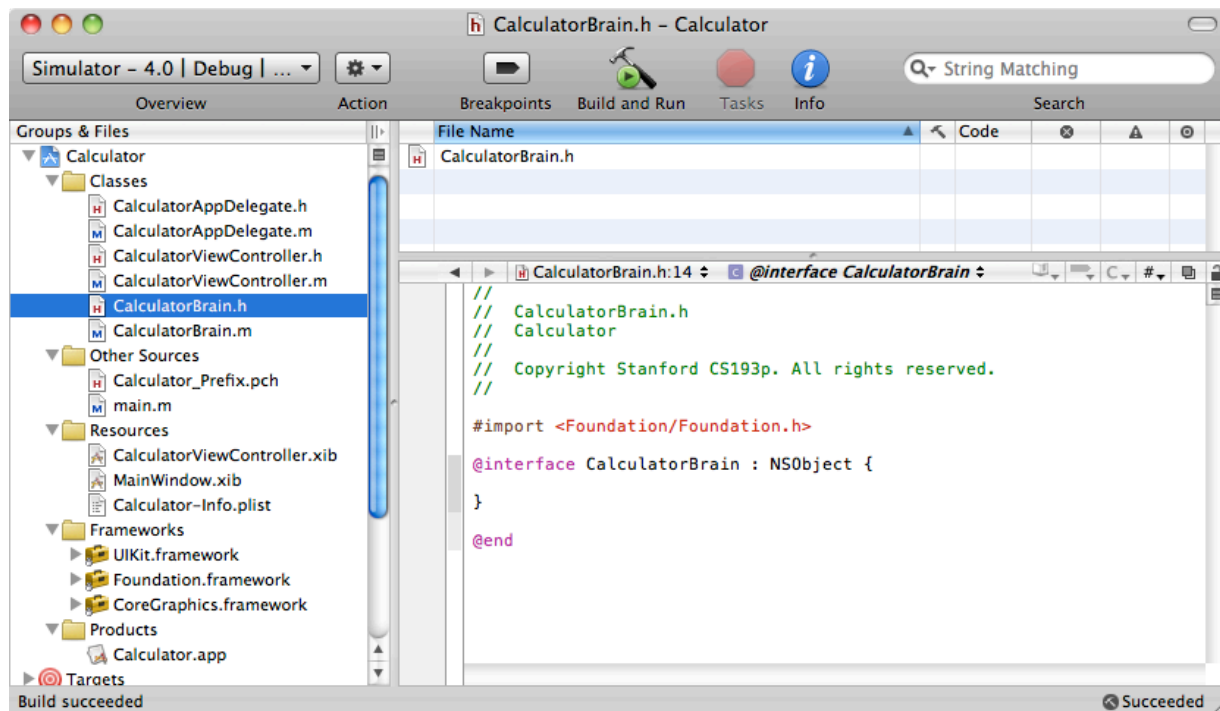


34. This would be another good time to **Build and Run** your program. At least now it won't be blank. Your application should have buttons, but it will crash if you touch them because there is no implementation (yet) for your **Controller**. You can see that the warnings the compiler gave us were meaningful (because indeed now our application is crashing due to the lack of implementation of those methods it is warning us about). We have now created our entire **View** and wired it up to our **Controller**, but we still have to implement the guts of both our **Model** and our **Controller**.

## Part V: Implement the 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`.

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



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.

36. First, 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
```

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

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

- (void)setOperand:(double)aDouble;

@end
```

38. 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
```

Note that the operation is specified using a string. That string is going to be the same as the one that's on an operation button. As mentioned above, using a string that has the same meaning both in your **Model** and your **View** is probably a bad idea, but in order to keep this implementation simple, we'll do it anyway.

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

39. 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  
//  
// Copyright Stanford CS193p. 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
```

40. The implementation of `setOperand:` is easy. We just set our instance variable to the `aDouble` that was passed in. Later in the course 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;
}
```

41. 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 (l) 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).

In other languages, this might look something like `operation.isEqual("sqrt")`. Dot notation means something a little different in Objective-C (we'll see this later as well).

By the way, don't worry about the fact that this code will not work for negative operands (the result will be NaN, not a number). If you want to add some code here to protect against that, feel free!

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 enters a number, then an operation, then another number, 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  $12 + 4 \text{ sqrt}$  = 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 (or equals) is requested.

42. Go back to `CalculatorBrain.h` and add 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;
```

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

```
- (double)performOperation:(NSString *)operation
{
    if ([operation isEqual:@"sqrt"])
    {
        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` (see that the code is invoked by the `else`) 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!



What would it look like to add another single-operand operation to our brain: +/-? It's simple, but be sure to place the `else`'s and curly braces in the right place.

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

    return operand;
}
```

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

We're not quite done here. We still need to implement `performWaitingOperation`. Note that this message is sent to `self`. This means to send this message to the object that is currently sending the message! Other object-oriented 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`.

44. 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 the 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.

45. Open `CalculatorViewController.m` and select and delete all the “helpful” code Xcode has provided for you between (but not including) the `@implementation` and the `@end`. We'll use some of this helpful code in future assignments, but not here.
46. 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 Stanford CS193p. 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 SDK 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

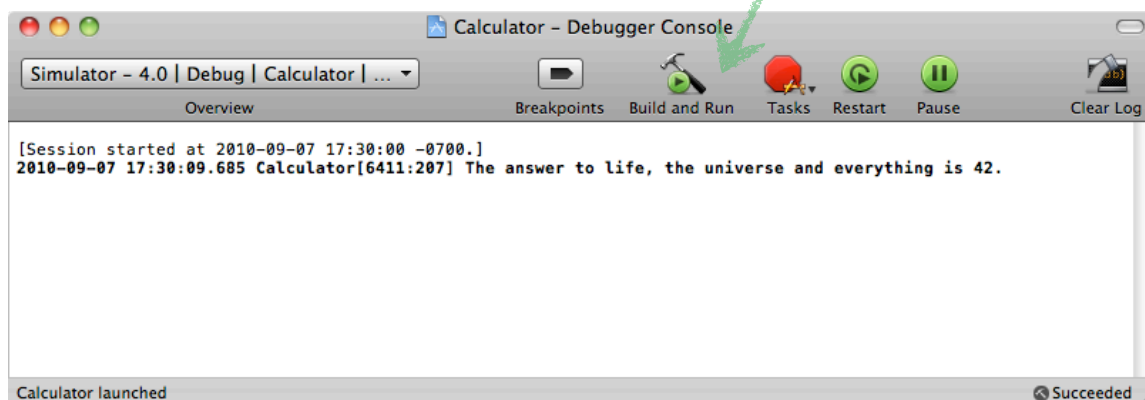
function). The 1st argument is an `NSString` (not a `const char *`, so don't forget the `@`), and the rest of the arguments are the values for any `%` fields in the first argument. A new kind of `%` field has been added, `%@`, which means the corresponding argument is an object. (The object is sent the message `description` to turn it into a string. The implementation of `description` is quite easy for the `NSString` class!)

47. 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);
}
```

Clicking on an operation button in our running application will print out "The answer to life, the universe and everything is 42." So where does this output go?

48. 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 you should see something like this:



49. Let's replace the `NSLog()` with the actual 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 the `UILabel` that is returned what its `text` is. The result will be an `NSString` with a `+` or `*` or `/` or `-` or `=` or `sqrt`.

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

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

50. 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.

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

```
- (IBAction)operationPressed:(UIButton *)sender
{
    NSString *operation = [[sender titleLabel] text];
    double result = [[self brain] performOperation:operation];
}
```

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

52. 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 `printf()` or `NSLog()` but for `NSString` objects. Note that we are sending a message directly to the `NSString` class (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 in our next few lectures.

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

The `printf` format `%g` means the corresponding argument is a `double`.

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).

53. Switch back to `CalculatorViewController.h` and add the instance variable `userIsInTheMiddleOfTypingANumber`. Its 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;
}
```

54. Now let's go back to `CalculatorViewController.m` and 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. We need to implement `digitPressed:` now anyway. Let's think about the logic of that method. 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.

55. 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];
}
```

56. Now that we have the digit, let's either append it to what's already been typed (using another `NSString` method called `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;
    }
}
```

You might wonder how (or even if) `userIsInTheMiddleOfTypingANumber` starts out as `NO`. It does because objects that inherit from `NSObject` get all of their instance variables set to zero. Zero for a `BOOL` means `NO`. Zero for an object pointer (also known as `nil`) means that that instance variable does not point to an object. That's how `waitingOperation` starts out back in `CalculatorBrain`'s implementation. It is perfectly legal to send a message to `nil`. It does nothing. If the method returns a value, it will return an appropriate zero value. Just be careful if the message returns a C struct (the result in that case is undefined).

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



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

If you have made any mistakes 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. There should be no more warnings either. If your code compiles and runs without warnings or errors 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 the lecture, 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. The UI will be blank, but if it does not run, you probably have a problem with your installation of the SDK.
3. Create the class which is going to be your **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 the [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](#) (your **Controller**) 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 **Controller** up to your **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** that points to your **Model**. Name it **brain**. It is of type **CalculatorBrain \***. Don't forget to `#import "CalculatorBrain.h"`.
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.

At this point you have created the header files for both your **Model** and your **Controller**. Next you need to create your **View**.

8. Open `CalculatorViewController.xib` (which contains your **View**), drag a `UIButton` out of the **Library window**, and control-drag from it to 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 above your calculator's keypad. 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:`.

Your **View** is complete. Next up is the implementation of your **Model**.

12. 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 its 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 **Controller** (`CalculatorViewController`) yet.

Your **Model** and **View** are complete. All that is needed now is the implementation of your **Controller**.

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 the `brain` method 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). If your code compiles without warnings or errors, 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](#).