Assignment 1 Walkthrough – No Explicit Due Date

Objective

Build a working calculator as done in class.

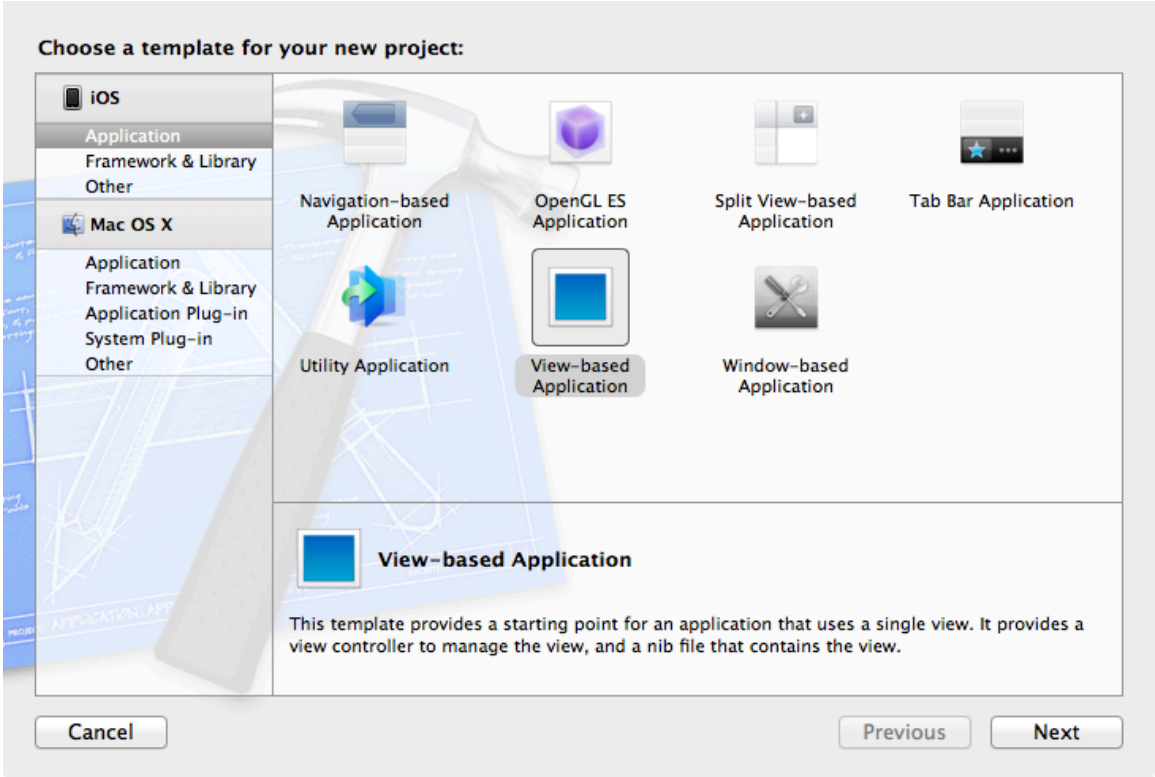
Goals

1. Get initial experience working with the iOS4 SDK.
2. Create a new project in XCode
3. Define a Model, View and Controller and connect them together.

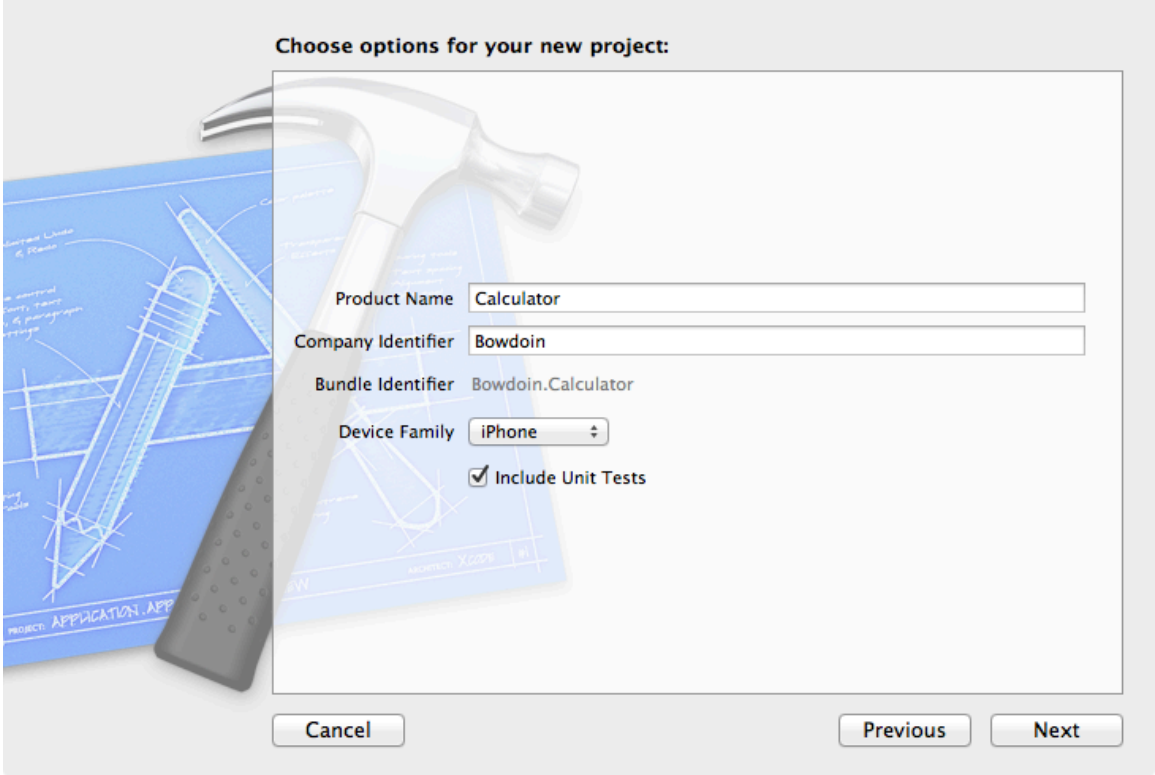
Detailed Walkthrough

Part I: Create a new project in XCode

1. Launch XCode.
2. From the screen that appears, choose Create a new XCode project. This can also be done by going to the File menu and selecting New->Project.
3. You will be asked to choose a template for your new project. Choose: **View-based application** and hit the **Next** button.

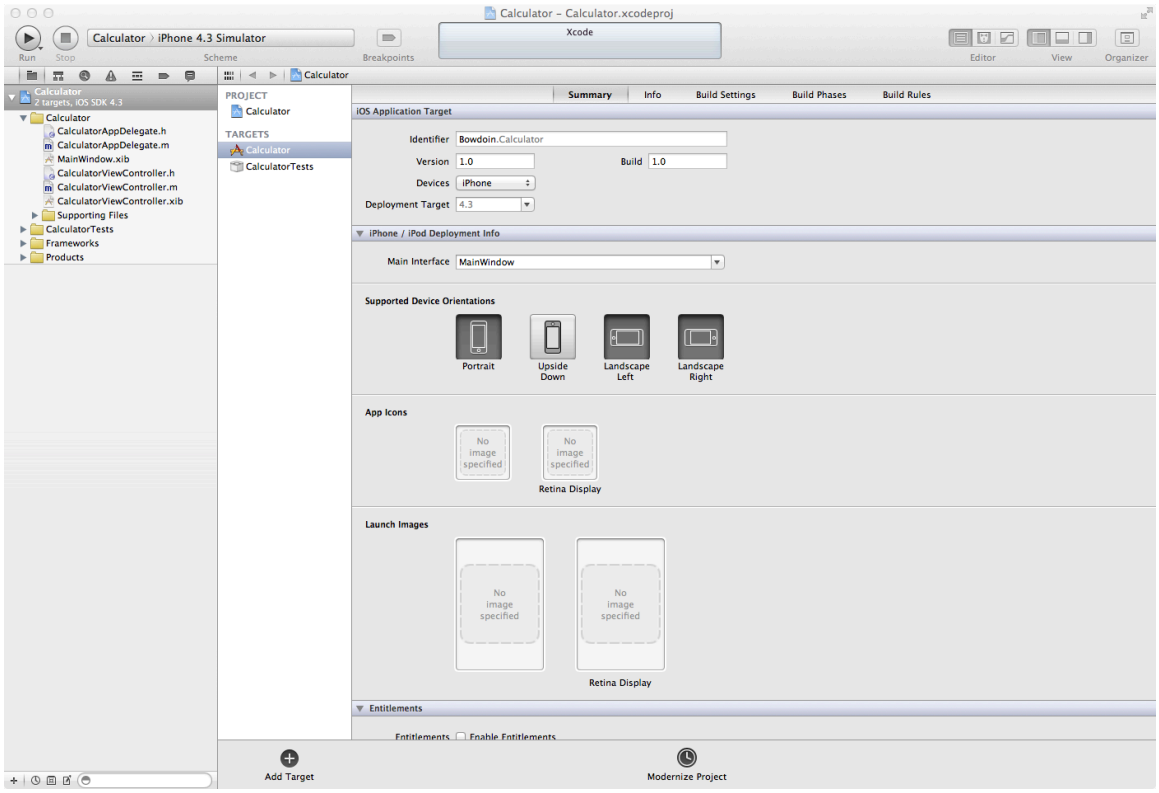


4. You will next have to pick a Product Name as well as a Device Family. For the Product Name use "Calculator." As for the Device Family you can either choose **iPhone** or **iPad**. We're going to start with **iPhone**.

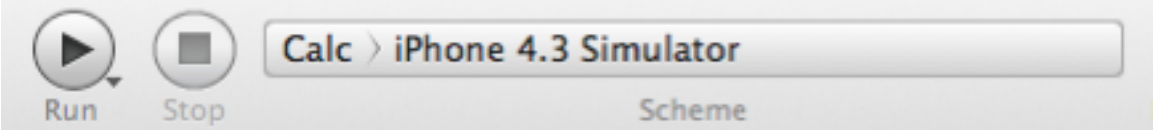


5. A navigation screen will come up and you can select where your project should be created. There is also a check box for whether you want to create a local **git** repository. Do this. We may or may not have time in the course to discuss **git**, but it is well worth your learning about it.

6. You have created an iOS app! The following window will appear.



In the upper left part of the window is a typical Play button that you can use to run one of the iOS simulators (iPhone or iPad). To the right of the button is a pull-down menu that can be set to iPhone or iPad (not for this app, but for other sorts). For now we'll stick to iPhone.



You can run the app right now. If everything has gone well you should just see what amounts to a blank iPhone.

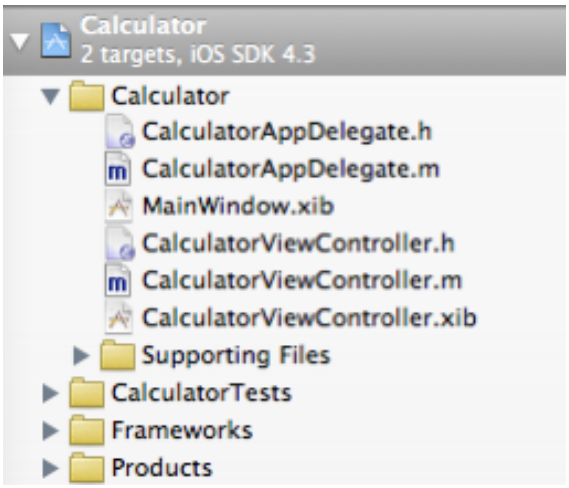
If that doesn't work you have trouble. At the left of your window you should see a bunch of file names.

7. You can quit the simulator now. Go back to XCode. At the top right there are a series of buttons. You can use these to control what it is you are working on and how you see it.

You'll want to play around with these and get to know what they do.

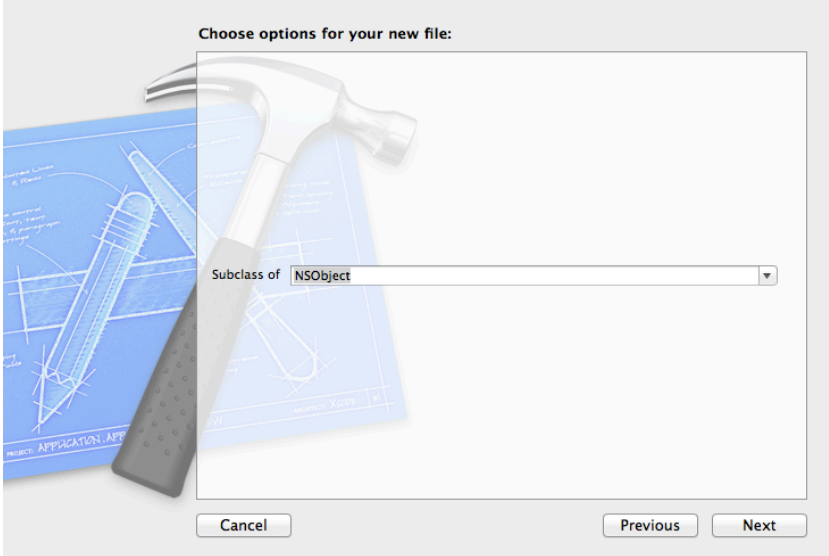
Meanwhile, on the left side of your window you will see a group of files and folders. For now most of these will look weird and hard to understand. As the term goes on we'll begin to learn about most of them. Some things to note and to look for. XCode has already created a bunch of files for you. These have names like CalculatorAppDelegate.h and CalculatorViewController.m. The view controller files

will be the key to the **Controller** part of our assignment. We'll create the **Model** in a minute. As for the **View**, we'll create that using XCodes's graphical tools.



Part II: Create a new class to be our Model

8. Next we will create a new Objective-C class to be our Model. Go to the **File** menu and select **New->New File**. You'll get another window. This window let's you select a template for your file. Choose **Objective-C class** and select **Next**. Yet another window will pop-up asking you what class you want your class to be a subclass of. You want to choose **NSObject**.



And yet another window will pop up. This one asking what your class should be called and where it should be saved. Type in **CalculatorBrain** here.

Your model has now been created. We'll work on writing it soon enough. In the meantime we will go back to our Controller to get started on wiring our app up.

Part III: Define the connections to/from the Controller

9. Find CalculatorViewController.h on your files on the left side of XCode. Click on it. This is the header file for your calculator's **Controller**. When you click on it the code should appear in the center pane of your window. A nice feature of XCode is that you can make both the .h and the .m files appear side by side by clicking on the center icon of the three **Editor** choices. Notice that XCode has already filled some code in for you. For example, it imports material from the **UIKit** library.

Our CalculatorViewController's header still needs a lot of things. The role of the **Controller** is to manage communication with the **View** and the **Model**. We do that in the following ways:

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**.

10. Let's add the *outlet* that enables our Controller to talk to a UILabel (an output-only text area). We'll call that *outlet* display.

Edit your .h file as follows (I'm ignoring the comments, import statement, etc. that come with it).

```
@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 to hook our **Controller** up to our **View**.

11. Now let's add an instance variable called brain that points from our **Controller** to our CalculatorBrain (our **Model**). 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
```

12. And finally, 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. no return value) except that XCode knows to pay attention to this method for the purposes of wiring your app together. What these function prototypes do is say that they are functions that will be called by UIButtons. In other words when you press the right button these are the functions that will be called. Notice that the UIButton itself will be sent to the method so the Controller can determine exactly what called it.

We'll need more stuff here later as we develop the app further, but for now we have covered what we will need to build our connections.

One thing to keep in mind as you are typing in all of this code is that XCode is constantly examining your code looking for mistakes. You'll often see little red triangles, or yellow triangles. These are signs of errors or warnings. If you look into CalculatorViewController.m at this point, for example, you would see a warning triangle. If you clicked on it it would tell you that you have not yet implemented the two functions you just prototyped. Note that it will still run (go ahead and try it).

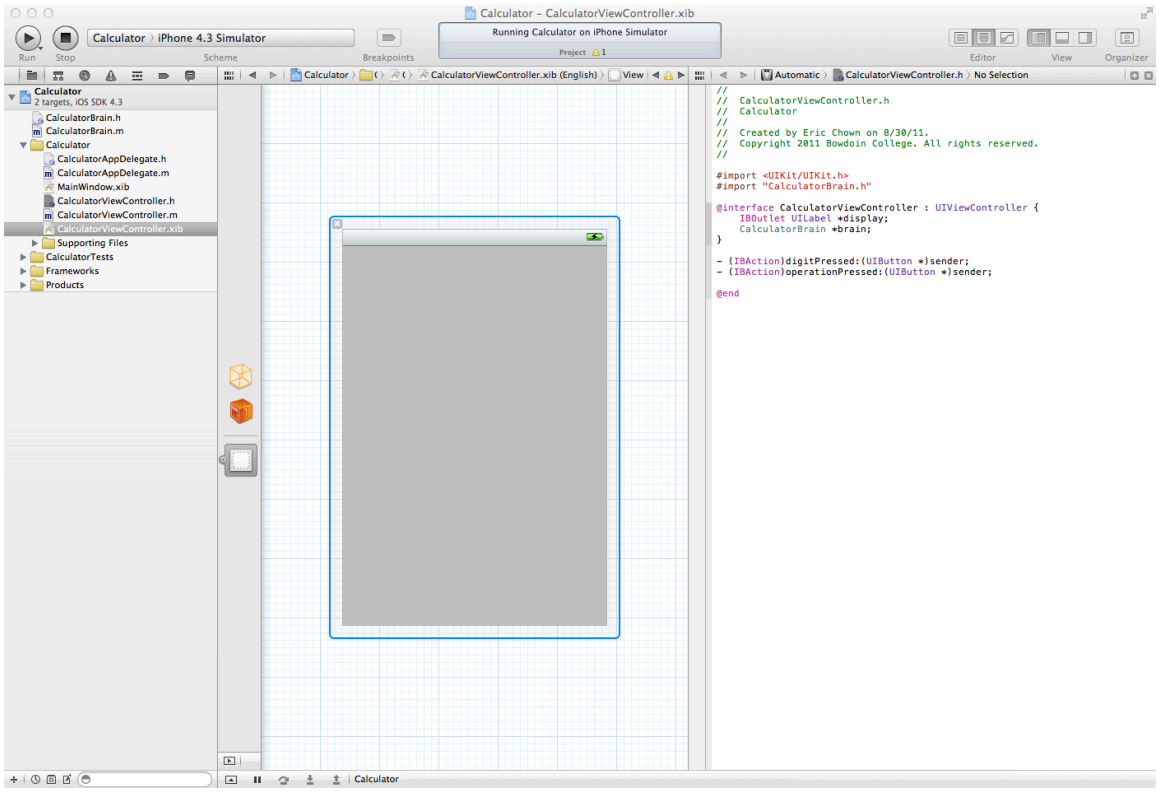
At this point we have declared what we need, now we can wire everything up.

Part IV: Create and wire up the View

It is time to create our view. The cool thing here is that we do not have to write any code at all to do this. We can do it all graphically within XCode. When we created our project we told XCode that it was a View-based project, so it automatically created a template **Controller** for us as well as a template **View**. This is in a file called CalculatorViewController.xib. Most people refer to this as a "nib" file because

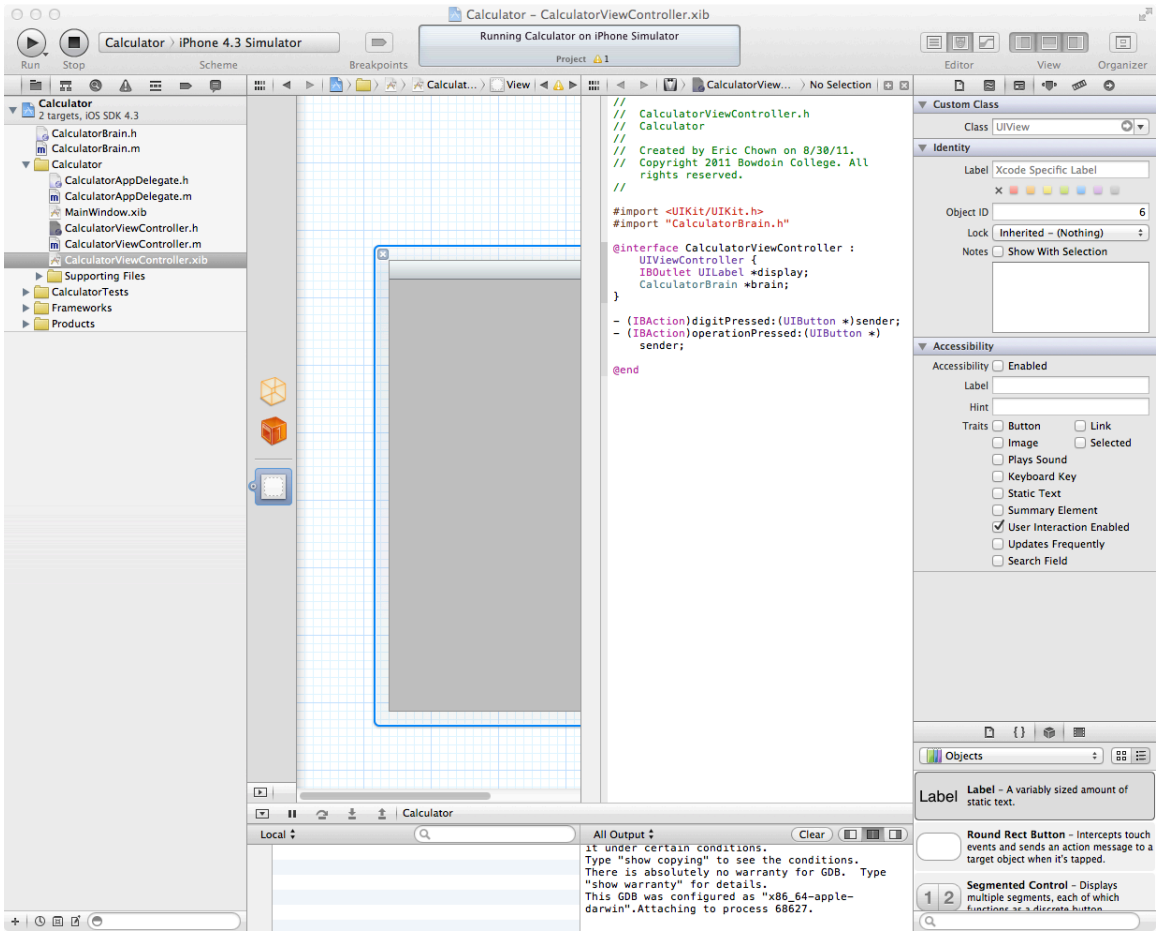
in earlier versions of XCode that is what it would have been. Some people call it a "zib" file.

13. You can open up the xib file simply by clicking on it. At this point you should see what looks like a blank iPhone.



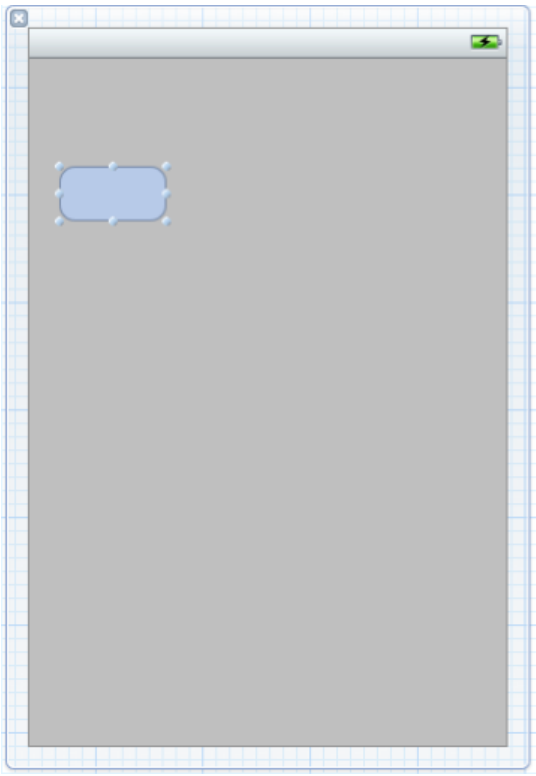
Note that between the blank iPhone and the file list are three little icons. If you mouse over them you'll see that the first is called the File Owner. So when we want to wire things up to our CalculatorViewController's outlets and actions, File Owner is the icon that we'll connect to. The second is the First Responder which we'll ignore for quite a while. The bottom is the View, the top-level UIView in our "view hierarchy." We'll get into the view hierarchy in much more detail throughout the term.

In the graphic above the right-hand pane contains the CalculatorViewController.h file. This is great because this is where we want to do the wiring. We also want to see some other things though. On the top right there are three icons above the word "View". I like to select them all. That gives me a screen like this:



They key here is what is in the lower right hand side. What you get are a bunch of generic view objects like buttons, labels and the like. We'll start dragging and dropping them into our view. That portion of the window is the **Library**. Above it is the **Inspector** (which has several subviews of its own).

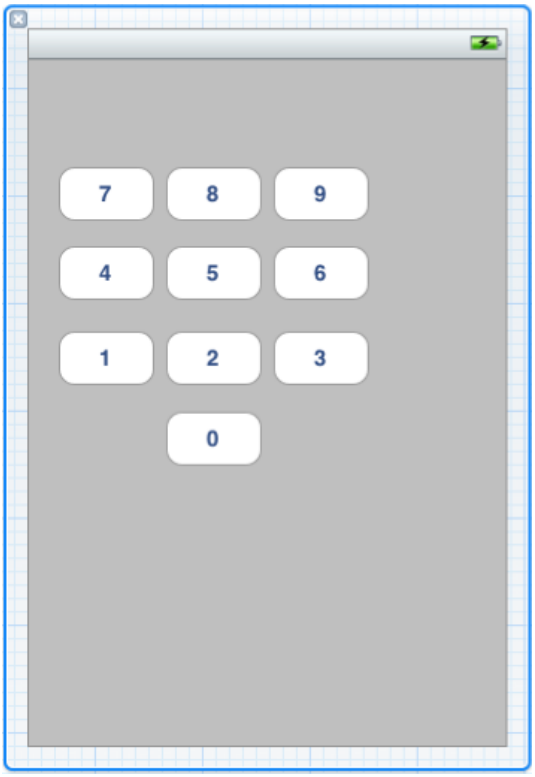
14. Let's build our interface. Start with the "7" key. Locate a Round Rect Button (a UIButton) in the Library. Now just drag it into our currently blank View.



15. Now resize the button to be 64 pixels wide (grab one of the little handles), and then pick it up and move it towards the left edge. When you get close a vertical blue line will appear letting you know that this is a good left margin for the button. You can put it at any vertical position for now.

16. Now for the good part. Hold down the control key and click on the button and drag it over to the digitPressed code. You should see a line as you drag, and as you get to the code it should highlight which method you are selecting. In this case you should make sure the digitPressed code is highlighted. As an alternative, you can control drag from the rectangle to the File Owner icon. When you do this you can choose between digitPressed and operationPressed. You would choose digitPressed for this button.

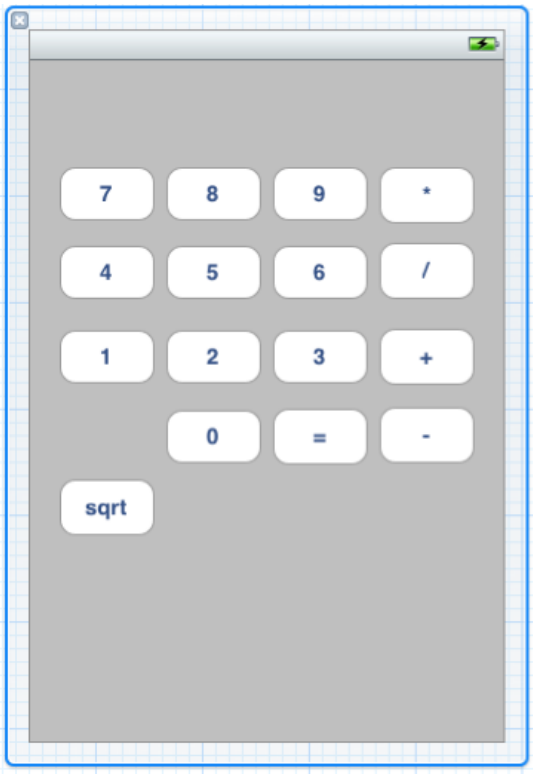
17. Now that you have done that you can copy and paste the button to make all of the digits. All of them will send digitPressed because the connections are copied too. You should end up with something that looks like this:



18. Double-click on the buttons to set their titles. When you have clicked on a button you'll also notice some settable options in the **Inspector** window.

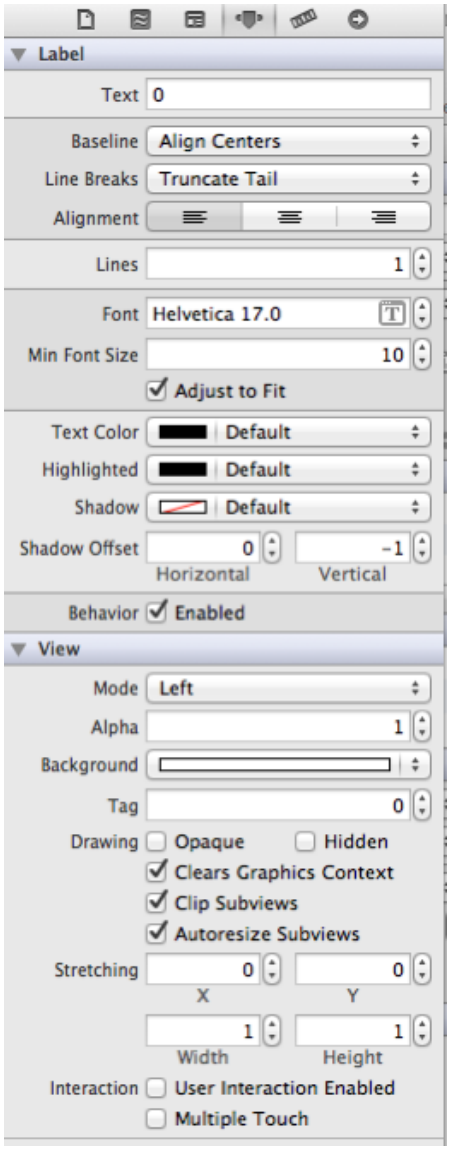
19. Now we're ready for some operations. Basically we are going to repeat the same process. However, do not copy and paste the digit buttons because you'll also get the digitPressed wiring which will be wrong for the operation buttons. Drag out a new round rect from the Library window. Again resize it to be 64 pixels wide.

20. Control-click and drag this time to the operationPressed prototype. Once you have done that you can again start copying (copy the new button you just made) and pasting. The titles are going to be important as you will check them in your CalculatorBrain. So if the title you pick doesn't match what you are looking for in the Brain then you will have trouble. At the end of this your UI should look something like this:



21. Now we need a display for our calculator. 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 from "Label" to "0".

22. This time we're going to use the Inspector to help us make changes. Make sure your new label is selected and click on the 4th icon above the inspector window. You should get a view like this:



You can make the font size bigger (e.g. 36 point) and change the alignment (use right alignment). Obviously you can change a bunch of other stuff here too. Note that we have a top section (Label) and a bottom section (View). This is because the UILabel inherits from UIView.

23. The last thing to change in our view is to make it possible for the CalculatorViewController to update our display. So we need to hook it up to that instance variable. In this case instead of control dragging from the button to the Owner (or .h file), we go in the opposite direction. So control-drag from the File's Owner to the UILabel. When you do that you will get a choice of display or view. Choose display.

24. This would be a good time to try running your program again. It should now have buttons. It will crash if you try them, but we'll get to that next.

Part V: Implement the Model

So far we have created a project, defined the API of our **Controller**, and wired up our **View** to our **Controller**. The next step is to fill in the implementation of our **Model**, CalculatorBrain. Remember, the number one design goal of this course is to do a clean separation of Model, View, and Controller. That will lead to better code, simpler modifications, and the better ability to re-use code.

25. Find and click on CalculatorBrain.h. You'll find that some of our code is already there. Notice that we import from the Foundation framework's header file and that we inherit NSObject. But that's it so far. We need instance variables and methods.

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 when the operation requires two operands (e.g. addition). For now, let's add a couple of things we're going to need.

25. 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
```

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

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

- (void) setOperand:(double)aDouble;
@end
```

27. 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. This is actually a pretty bad idea because it violates some of our guidelines for separating the **View** and the **Model**, but it really simplifies things and at this stage that is a good thing.

28. Copy the two method prototypes in CalculatorBrain.h, then switch to CalculatorBrain.m and paste them between the @implementation and the @end. For now I am ignoring the init method that XCode has automatically provided for you. Get rid of the ; at the end of each prototype and replace them with the paired curly brackets. You've probably noticed by now that XCode is eager to fill in things for you when it can. It takes a little getting used to, but it can really speed up typing and development.

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

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

}
```

29. Implementing setOperand is easy. It is basically a simple setter. We'll see in class very soon that this sort of thing is so common that XCode can actually generate the code for you automatically.

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

30. The implementation of performOperation is also pretty straightforward, at least for single operations like sqrt.

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

The first line is important. It is the first time we have sent a message to an object using Objective-C. The syntax looks very weird at first. The square bracket is how we tell Objective-C that we're sending a message to an object. We always use this syntax even when the object is ourself. The first thing after the square bracket is the object to send the message to. In this case it is 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: (note: I generally will leave the colon out of message names). Then comes the argument for isEqual. If the method had multiple arguments, they would be interspersed with the components of the name (more on that in class).

In Java this would look something like operation.isEqual("sqrt"). Dot notation means something different in Objective-C (as we'll see in class soon).

Sharp-eyed students will note that this code has problems with negative numbers. Feel free to fix that.

Now it is time to deal with operations that require two 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 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.

31. 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;
@end
```

32. Now 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!

33. How would you add another single-operand operation to our brain such as +/-? It is simple.

```
- (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;
}
```

You may note that I have swapped the places of the object (operation) and the NSString (@"+/-") here. It is perfectly legal because both are NSStrings even though one is a constant.

We still need to implement performWaitingOperation. Note that the message was sent to self. This means to send this message to the object that is currently sending the message! In Java you would use "this". We are going to use self a lot more than we ever used this in Java.

We want performWaitingOperation to be private to our Brain, so we are not going to put it in CalculatorBrain.h, only in CalculatorBrain.m. Do you see how this makes it private? When other classes want to use the CalculatorBrain class they normally #import "CalculatorBrain.h" which gives them anything defined in there. By not defining the method there we hide it from them. It turns out that this isn't truly private either as other classes could still call this method due to the dynamic nature of Objective-C. Putting the code only in the implementation file has one important side-effect – you must put the code for this method before any methods that will call it (namely performOperation).

34. Here is the implementation of performWaitingOperation. Remember to put it before 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;
        }
    }
}
```

We just match up the current waitingOperation to all of the operations that we can perform, then we perform the operation using the current operand and the waitingOperand.

Note that we silently fail on divide by zero (but do not crash!). Not user friendly, but simple.

Note also that we do nothing at all if the waitingOperation is something we haven't seen.

Part VI: Implement the Controller

All we have left is taking care of what happens when an actual button is pressed. Remember we wrote the prototype and wired the buttons to call it, but we haven't done the implementation yet.

35. 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 it in future assignments, but do not need it now.

36. Now copy the prototype code from the CalculatorViewController.h file back into the .m file. Again remove the ; and replace them with curly brackets. You should have something like this:

```
//
//  CalculatorViewController.m
//  Calculator
//
//  Created by Eric Chown on 8/30/11.
//  Copyright 2011 Bowdoin College. 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!)

37. 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);
}
```

Now if you click on an operation button while running the application it will print out "The answer to life the universe and everything is 42." So where does it print out. The answer is the **console** which you can find on the bottom of the window (depending on which editor buttons you have clicked).

38. OK, that was fun, but now let's get back to business. Let's replace the NSLog() statement with the actual implementation we need. 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 it's 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 standard procedure. You'll note at this stage that the code just typed will produce a warning message because all we have done is create a local variable that we don't use. We'll get to that.

39. Next we need ask our brain to perform that operation. First we need our brain! Where is it? We have an instance method for it (called brain), but we never set it. So let's create a method (somewhere earlier than we are going to use it in CalculatorViewController.m since it is private) that creates and returns out brain.

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

Note the `if (!brain)` part. We only want one brain and this code keeps us from creating more than one. We create the brain by alloc-ating memory for it, the init-ializing that memory. We'll talk about this next week and throughout the term.

40. Let's use that method. It returns a brain for us.

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

```
    NSString *operation = [[sender titleLabel] text];
    double result = [[self brain] performOperation:operation];
}
```

Obviously we're still not done since we have simply created another local variable.

41. Let's put it in our display. We just send the setText method to our display *outlet* (remember it was wired to the UILabel in the **View**). The argument we are going to pass is an NSString created using stringWithFormat. It is 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. Much more on this to come.

```
- (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.

We're almost there. We still need to worry about what happens when the user is in the middle of typing a number. We need to keep track of the digits as they are entered. We'll need another instance variable to do so. We'll call this one userIsInTheMiddleOfTypingANumber (an example of a self-documenting name).

42. Go back to CalculatorViewController.h and add the instance variable. Its type will be BOOL. A BOOL in Objective-C takes the values YES and NO.

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

43. Now let's go back and fix up operationPressed. Now we can check if the user is in the middle of typing, and if so, update the operand accordingly.

```
- (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]];
```

```
}
```

44. Let's move on to digitPressed. It can grab the label of the button that was pressed to figure out the digit.

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

45. Once we have that digit we either need to append it to the string of typed digits or set it to be the start of the number 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;
    }
}
```

If you recall normal C variables are not initialized. This might suggest that userIsInTheMiddleOfTyping needs to be initialized somewhere. It turns out that 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 the variable does not point to anything. That's how waitingOperation starts out back in CalculatorBrain's implementation. One thing that is very different in Objective-C and Java is that in Objective-C it is legal to send a message to nil! It does nothing. Even if the method would normally return a value it will return an appropriate zero value (e.g. if the method would return a double it will return 0.0). The one place you will run into trouble is when the message returns a C struct (the result in that case is undefined).

We're done! Let's try running.

Part VII: Build and Run

46. Click the Run button in XCode. You should have a working calculator!
You do not have to hand anything in. Your first assignment will build off this and will be out on Tuesday.