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

Task: Create a new application in Xcode called “Calculator”.

1. Launch Xcode and select the “Create a new Xcode project” option. If you don't see the splash window, you should go to “File > New > New Project...” in Xcode menu.

2. Select the Single View Application template and click Next.

3. Type in “Calculator” for the Product Name.

4. Enter “com.FMI.FirstName.LastName” for the Company Identifier. Notice how Bundle Identifier changes as you type. You should obtain something like “com.FMI.Radu.Ionescu.Calculator” as your bundle identifier.

5. Enter “Calculator” as the Class Prefix for the classes this template is going to generate for us.

6. Select “iPhone” for Device Family. We are going to make this application only for the iPhone (not iPad).
Task 1

Task: Create a new application in Xcode called “Calculator”.

7. Check “Use Storyboard”. Storyboards are a new (iOS 5) way to organize your MVC's Views that we are going to use.

8. Check “Use Automatic Reference Counting”.

9. We won't be creating Unit Tests for this application so we are going to leave the “Include Unit Tests” option unchecked.

10. Click Next.

11. Navigate to “~/Developer/Apps/” folder inside the home directory. If you want to keep your project for later use, please save it in a directory with your name like this: “~/Developer/Apps/<YourName>”.

12. Click Create to create your project directory inside the “~/Developer/Apps” folder.
Task 2

Task: Start building our Calculator’s View by adding a text label to be the Calculator's display.

1. Open up our MVC's View by clicking on MainStoryboard.storyboard in Project Navigator.

2. Hide the Document Outline if it's not already hidden.

3. Click on the “butler” icon on the Toolbar to show up the Assistant Editor. This will by default bring up the View’s Controller.

4. We don’t need the Project Navigator at the far left either, so let’s hide it by using the “Hide or show the Navigator” button available on the Toolbar.

5. Bring up the Utilities area by clicking on the “Hide or show the Utilities” button that is also available on the Toolbar.

6. In Utilities area, click on the Object Library (it might already be selected). Some objects (those appropriate to dragging into your View) should appear in the Object Library.
Task 2

Task: Start building our Calculator’s View by adding a text label to be the Calculator's display.

7. Find a UILabel object in Object Library.

8. Drag the UILabel from the Object Library to your View.

9. In Utilities area, click on the Attributes Inspector. You should see attributes of the Label you just created.

10. Grab the lower right "handle" on the label and resize it to 280 x 36 pixels. Use the dashed blue guidelines to pick a good size. Make sure you align and resize it as in the preview screenshot from the next slide.

11. The numbers in a Calculator's display are never (rarely?) left aligned, so let's change the alignment of the text in our display label to right by clicking on the appropriate button in the inspector.
// CalculatorViewController.h
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import <UIKit/UIKit.h>
@interface CalculatorViewController : UIViewController
@end
Task 2

Task: Start building our Calculator’s View by adding a text label to be the Calculator's display.

12. Let's also make the font bigger. Increase the font size to 24 point Helvetica.

13. We don't want our Calculator to appear with "Label" in its display! So double-click on the label to put it in an editing state then type 0.

14. Hold down CTRL while mousing down and dragging a line from the text label directly into the code of our Controller.

15. We are going to name this outlet display (since it is the display of our Calculator). So type "display" for the outlet name.

16. Select the "Weak" storage type and click Connect to create a property (called display) in our Controller which will point to this UILabel in our View.

The screenshot from the next slide gives you a hint of how everything should look like.
// CalculatorViewController.h
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *display;
@end
Task 3

Task: Add our Calculator’s keypad buttons.

1. Switch Assistant Editor to the Controller's implementation file named “CalculatorViewController.m”.

2. Delete the code that we don't need which was automatically added by Xcode. Make sure NOT to remove the @synthesize declaration and the implementation of the viewDidUnload method.

3. Drag a Round Rect Button from the Object Library to your View.

4. Grab the middle-right "handle" on the button and resize it. A width of 64 points works extremely well, so use that. The screenshot from the next slide gives you a hint of how everything should look like after this step.

5. Hold down CTRL while mousing down and dragging a line from the button directly to the text area where your code is.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"
@implementation CalculatorViewController
@synthesize display;
-
(void) viewDidUnload
{
    [self setDisplay:nil];
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
@end
Task 3

Task: Add our Calculator’s keypad buttons.

6. Enter “digitPressed” as the name of the action message (which makes sense since this button is going to be a digit button in our Calculator’s keypad).

7. You can leave the rest of the fields alone (the defaults are fine for this button). Then press Connect.

8. Similar to an outlet, you can mouse over the little icon on the left of the method implementation and see which object(s) in your View send(s) this message. Notice how the button highlights.

Check out the screenshot from the next slide for a hint of how everything should look like.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

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

-(IBAction)digitPressed:(id)sender
{
}
@end
Task 3

Task: Add our Calculator’s keypad buttons.

9. The type `id` is a very special type. There are some times when we want to use it because either we allow any class of object to be passed into a method (uncommon) or because the class of the object is opaque (it's like a cookie).

But neither of those cases applies here. In this case, we know that the `sender to digitPressed:` is going to be a `UIButton`. Therefore we are going to change this type to be “pointer to a `UIButton`” instead of “pointer to an object of any class”.

Select the type of the sender argument to this method and replace it with the type “`UIButton *`”.

Check out the screenshot from the next slide.
Using UIButton * rather than id is called "static typing". Static typing is purely a compiler thing. It has no effect on what happens at run time. The compiler will just generate better warnings if you try to write code that sends a message to sender which a UIButton does not recognize. If you send a message to sender that it does not recognize, your program will crash, regardless of whether you statically typed sender.
Task 3

Task: Add our Calculator’s keypad buttons.

10. Copy and paste our first button to make another button. The copied button will send the same action (`digitPressed:`) as the original.

11. Move the copied button to line up horizontally with the original (the dashed blue lines are awesome here).

12. Copy and paste again to obtain 3 buttons horizontally aligned.

13. Now copy and paste 3 buttons at a time.

14. Using copy and paste and the grid lines, create the entire keypad for the Calculator. It should look like in the screenshot from the next slide.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"
@implementation CalculatorViewController
@synthesize display;

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

- (IBAction)digitPressed:(UI_BUTTON *)sender
{
    @end
}
Task 3

Task: Add our Calculator’s keypad buttons.

15. Double-click on a button to make its text editable. Then type the number that goes in the appropriate spot.

16. Do this for all the buttons. Your keypad should now look like the one in the screenshot from the following slide.

17. Right-click on the nine button.

18. Mouse over the "Touch Up Inside" connection and you will see that the whole View will highlight (that's its way of showing you that this button sends its message to the Controller).

19. Right-click on the icon that represents the Controller (it's under the View).

20. Then mouse over this "Button - 4" entry. Notice how the button highlights.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

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

- (IBAction)digitPressed:(UIButton *)sender
{
    @end
}
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

1. We won't need the Utilities area for a while, so close it from the “Hide or show the Utilities” button from the Toolbar.

2. Start by declaring a local variable called `digit` inside the `digitPressed:` method implementation. This local variable will be of type "pointer to an `NSString` object".

3. Since all the buttons send the same action message to our Controller, we have to look at the action message’s argument (`sender`) to find out which one was touched.

   `UIButton` objects respond to the message `currentTitle` which returns an `NSString` containing the title of the button. We'll use that to figure out which button was touched.

Make sure your code looks like in the next screenshot.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@interface CalculatorViewController : UIViewController
@end

@implementation CalculatorViewController
-
(void) viewDidLoad
{
    [self setTitle:nil];
    [super viewDidLoad];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

-(IBAction)digitPressed:(UIمديرية a)sender
{
    NSString *digit = [sender currentTitle];
}
@end
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

4. Add an `NSLog()` to print the digit chosen by the user to the console. Your code should be something like the code in the next screenshot.

5. Run the application with this `NSLog()` in place.

6. Touch a few keys and check out the console for the messages printed by `NSLog()`.

7. Stop the simulator and delete the line of code with `NSLog()`.

8. Now that we have the `digit` from the button, we need to update our `display` by appending the `digit` onto the end of it. This actually only takes one line of code, but we’ll break it down into steps.

Let's make another local variable called `myDisplay` (of type "pointer to a UILabel") into which we'll just put the value of our `display` outlet. Make sure you express calling the getter of our `display` @property using dot notation.
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

9. Now that we have a pointer to our display, let's send it a message to find out what text is currently in it. The message to send is called (appropriately) `text`.

Add a line of code to get the `text` out of our display `UILabel` and store it in a local variable (a pointer to an `NSString` object) called `currentDisplayText`. Note that `text` is a property of `UILabel` and you can use dot notation to access it.

Look at the screenshot from the next slide to see how your code should look at this moment.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

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

- (IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    UILabel *myDisplay = self.display;
    NSString *currentDisplayText = myDisplay.text;

    @end
}
Task 4

Task: Write the code inside digitPressed: that will get executed whenever any of the keypad buttons gets touched.

10. There's really no need for the local variable myDisplay. So let's select and copy its value (self.display) and paste it where we use it.

11. Now you can delete the line of code that declares the local myDisplay variable.

12. Next we need to append the digit that the user just touched onto the end of what is currently in the display. Create a local variable called newDisplayText and assign it with the value returned by messaging the currentDisplayText local variable with stringByAppendingString: with the argument digit.

Look at the screenshot from the next slide to see how your code should look at this moment.
// CalculatorViewController.m
// Calculator
//
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompany__. All rights reserved.
//
#import "CalculatorViewController.h"

@implementation CalculatorViewController
@synthesize display;

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

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    NSString *currentDisplayText = self.display.text;
    NSString *newDisplayText = [currentDisplayText stringByAppendingString:digit];

@end
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

13. We need to set our `display UILabel`'s text to the new string with the digit appended to the end. You can use dot notation to set the `self.display.text` property to be the `newDisplayText`.

Note that dot notation for setters is exactly the same as dot notation for getters, it’s just that they appear on the left-hand side of equals signs rather than the right-hand side.

14. We don't need the `newDisplay` local variable really, so let's copy its value (the `stringByAppendingString:` message-sending construct) and paste it where it is used.

15. Then we can delete the previous line (that declares the `newDisplayText` variable).

Look at the screenshot from the next slide to see how your code should look at this moment.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController
@synthesize display;

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

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    NSString *currentDisplayText = self.display.text;
    [self.display.text = [currentDisplayText stringByAppendingString:digit]];
}
@end
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

16. The same thing happens for the `currentDisplayText` local variable. It is not really needed. Copy the value of `currentDisplayText` and paste it where it is used.

17. Okay, let's Run and see if this works!

18. Touch a few keypads. Notice the leading zero that doesn't look right on the display. We should try to fix the problem with the leading zero.

The problem with the leading zero is that we are appending new digits even if the user is not currently in the middle of entering a number. The display should get cleared when the user starts typing a new number instead of appending to whatever happens to be there (like the 0 at the beginning or some operation’s result later on).

Look at the screenshot from the next slide to see how things should look by now.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright © 2012 __MyCompany__. All rights reserved.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

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

- (IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    self.display.text = [self.display.text stringByAppendingString:digit];
}
@end
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

19. To fix this, we are going to need a `@property` to keep track of whether the user is in the middle of entering a number. But we don't want to add the `@property` to our header file because those properties are public.

So where do we add private properties? We need to add a private `@interface` to our implementation file. This is called a Class Extension.

The concept of “public versus private” in Objective-C is done via "header file versus implementation file". You declare public stuff in your header file’s `@interface-@end` block. You declare private stuff in your implementation file’s `@interface-@end` block.

20. Add a private boolean property to track whether the user is in the middle of entering a number. Check out the screenshot in the following slide to understand how to do this.
There's no strong or weak here because a BOOL is not a pointer.

This is called a Class Extension. Notice the ( ) at the end.

BOOL is the typedef used for boolean values in Objective-C. Its value is either YES or NO. NO is zero, YES is non-zero.

nonatomic means that the setter and getter for this property will not be thread-safe. You will always use this keyword unless you really know what you are doing. It's not really a problem because even though we will do lots of multi-threaded programming in iOS, virtually all methods in UIKit must be performed on the main thread of execution in your application (it is non-UI activity that we will put in other threads).

You might think `userIsInTheMiddleOfEnteringANumber` is sort of a silly name for a variable. But long variable names are encouraged in iOS development because Xcode completes them for you after only a few characters and self-documentation is very important to good coding style.
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

21. There are two warnings for the private `@property` declaration. The problem is that we declared the `@property`, but we have not implemented the getter (first warning) or the setter (second warning).

Let's use `@synthesize` (again) to implement both the getter and the setter for us! Note that `@synthesize` doesn't care whether your `@property` is public (declared in the header) or private (declared in the implementation file).
Task 4

Task: Write the code inside `digitPressed:` that will get executed whenever any of the keypad buttons gets touched.

22. Now we just need to only do the appending if the user is in the middle of entering a number.

But what value does `userIsInTheMiddleOfEnteringANumber` start out with? All properties start out with a value of zero. For a pointer to an object (like `display`) zero is called `nil`. Your program will not crash if you send a message to `nil`. It just does nothing in that case (any value the method returns will be zero).

If the user is not in the middle of typing, just start off a new number with the `digit` that was touched. In this case we must set `userIsInTheMiddleOfEnteringANumber` to `YES` because we are now in the middle of entering a number.

Check out the screenshot from the next slide to see the final implementation of `digitPressed:`.

23. That should do it. Let's run again.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright © 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@interface CalculatorViewController :
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

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

- (IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.isUserInTheMiddleOfEnteringANumber)
    {
        self.display.text = [self.display.text stringByAppendingString:digit];
    }
    else
    {
        self.display.text = digit;
        self.isUserInTheMiddleOfEnteringANumber = YES;
    }
}
@end
Task 5

Task: Add some more buttons (for operations *, /, +, - and Enter).

1. Bring back the Utilities area.

2. Drag a Round Rect Button from the Object Library to your View. Do NOT copy and paste a digit button to make this first operation button. Copying and pasting buttons brings the button's action message along with it and we want operation buttons to send a different message than digit buttons!

3. Resize the button to 64 x 37 pixels.

4. CTRL-drag to create this button's action.

5. Name the action `operationPressed`: then click Connect.

6. You should statically type the `sender` to `UIButton *`.

Check out the screenshot from the next slide to see how the button and the action should look like.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright © 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@interface CalculatorViewController :
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

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

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

- (IBAction)operationPressed:(UIButton *)sender
{
    self.userIsInTheMiddleOfEnteringANumber = YES;
}
Task 5

Task: Add some more buttons (for operations *, /, +, - and Enter).

7. Now use copy and paste to create 4 operation buttons. Again, do not copy and paste a digit button to make an operation button!

8. Set the title of each to these four operations to *, /, + and -, respectively. Check out the screenshot from the next slide to see how your view should look like by now.

9. We need an Enter button because an RPN calculator puts all of its operands on a stack and then operates on them. Enter is used to push a number onto the operand stack (e.g. 6 Enter 5 Enter + results in 11).

Drag a Round Rect Button from the Object Library to the View and place it between 0 and - buttons. Do NOT copy and paste either a digit button or an operation button to make an Enter button. The Enter button will have a different action than either digit buttons or operation buttons.
// CalculatorViewController.m
// Calculator
// Created by Radu-Tudor Ionescu on 2/27/12.
// Copyright © 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorViewController.h"

@interface CalculatorViewController
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

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

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

-(IBAction)operationPressed:(UIButton *)sender
{
}
@end
Task 5

Task: Add some more buttons (for operations *, /, +, - and Enter).

10. Resize it to 64 x 37 pixels and set its title to Enter.

11. CTRL-drag to create this button's action. Put this action before `operationPressed:` in the file. We are going to call the Enter action from `operationPressed:`, so it needs to be declared earlier in the file. This is not a compiler restriction, but only a good coding practice.

12. Name this action `enterPressed`.

13. There's something a little different about this action method. We don't need the `sender` argument because there's only one Enter key. Note that we can control whether an action message includes the `sender` as an argument.

Change the Arguments to None and click Connect.

Everything should look like in the following screenshot.
Notice the `enterPressed` action has no arguments.
Task 6

Task: Create the Model of our MVC for the calculator brain.

1. We cannot proceed any further with our implementation without the Model of our MVC. So we’re going to take a time-out from implementing our MVC’s Controller to go implement our MVC’s Model. Hide the Utilities area again.

2. Before we switch to our writing our Model, it would be nice to capture the setup on the screen (i.e. View and Controller) so that we can easily return to it later. We can do that using Tabs in Xcode (just like Tabs in a Browser).

In Xcode menu go to “File > New > New Tab” (or use the CMD + T shortcut) to open a new tab.

Notice the new tab starts out as a snapshot of the old tab. Once we start changing it, we can get back to the old arrangement by clicking the other tab.
Task 6

Task: Create the Model of our MVC for the calculator brain.

3. Time to create our Model. Do this by selecting “New File” from the File menu.

The File menu’s “New File…” item is the gateway to creating a wide variety of things in an application. Including not only new classes (as in this case), but also user-interface elements, database schema, and more.

4. Select the “Objective-C class” option and click Next.

5. We are going to call our Model's class “CalculatorBrain”, so type that in the Class field. Check out the following screenshot.

Our Model is going to be a direct subclass of NSObject, so this is correctly selected. NSObject is the root superclass of all objects in iOS. Classes inherit some nice generic functionality from NSObject including the method called description which returns an NSString representation of the object which is useful for debugging with NSLog().
Also notice the new tab.
Task 6

Task: Create the Model of our MVC for the calculator brain.

6. Then click Next to choose where to put your Model’s .m and .h files.

7. Make sure you put your Model’s .m and .h files in the same place as all of your other .m and .h files, that is in the "Calculator" subfolder.

8. Don’t put your Model’s .m and .h files in the top-level group in the Navigator either. Put them in this group one level down.

Check out the screenshot from the following slide to understand how to set things up.

9. Click Create. Notice the Navigator shows up after you create your Model. Also notice that Xcode has created stubs for both the header and implementation of our MVC’s Model and it has automatically renamed the current tab. You can name it yourself by double-clicking on it if you wish.

10. Close the Navigator to make space.
This chooser defines where your Model's .m and .h files will appear in the file system.

This pop-up defines where your Model's files will appear in the Navigator. (the area on the left that we hid at the start).

Other .m and .h files in your project.
Task 6

Task: Create the Model of our MVC for the calculator brain.

11. We’re going to start by defining the public API of our Model. All of our public API lives in the header file (that’s what makes it public). Public API are method and properties other objects (besides our Model itself) are allowed to invoke.

First add the `pushOperand:` which will provide a way to push operands onto our Calculator’s stack of operands. This method must return nothing (`void`). The argument must be a double-precision floating point number called `operand`.

Notice the warning that was introduced by declaring this method. It says "Incomplete implementation". This makes sense because we have not yet implemented `pushOperand:`.

The following screenshot will show you how to declare this method.
The `-` means this is an instance method (i.e. instances of this class respond to it). There is also such a thing as a class method (i.e. the class itself responds to it) which must be declared with a `+` at the beginning.
Task 6

Task: Create the Model of our MVC for the calculator brain.

12. Now add the `performOperation:` method that performs a given operation using the operands on the stack. We are going to use a string to describe the operation (the same string that is on the operation buttons in our UI). This is pretty bad design to have strings in the UI also have meaning in your Model, but it’s simple and so we’ll go with it for this demo.

This method returns a `double` (the result of performing the operation). The argument to this method is a pointer to an `NSString` object called `operation`.

13. Let’s stub out both of our methods in our implementation.

14. Let's have the `performOperation:` method return a default value of zero for now.

The `.m` and `.h` files of our model should like in the next screenshot.
// CalculatorBrain.m
// Calculator

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

#import "CalculatorBrain.h"

@interface CalculatorBrain : NSObject
-
(void)pushOperand:(double)operand
{
}
-
(double)performOperation:(NSString *)operation
{
    double result = 0;
    // perform operation here, store answer in result
    return result;
}
@end
Task 6

Task: Create the Model of our MVC for the calculator brain.

15. How are we going to store our stack of operands? We are going to use an array. “Pushing” onto our stack will just add an item to the end of the array. “Popping” will grab the last item in the array, then remove that item from the array.

Add a private `@interface` so that we can declare the array we need to store the operand stack.

16. Add a nonatomic and strong `@property` called `operandStack` of type “pointer to `NSMutableArray` object”.

We will cover much more later about arrays, strings, etc., but notice that this array’s class name is `NSMutableArray`. The base array class (`NSArray`) is not modifiable. Clearly that wouldn’t work for this class’s implementation.

Check out the next screenshot.
**strong** means keep this object (the array) around until I’m done using it. Most non-outlet @property **s** are **strong**. As we saw earlier, the alternative to **strong** is **weak**.

**weak** means “if no one else is interested in this object, then neither am I, so set this @property to nil (zero) if that becomes the case”. This time, our Model’s implementation is the only one interested in operandStack, so we must make it **strong**.
Task 6

Task: Create the Model of our MVC for the calculator brain.

17. Check the warnings for the previously declared property.

As we saw last time we added a @property, the compiler warns us that we need to create its getter (operandStack) and setter (setOperandStack:). It’s even suggesting that we use @synthesize to generate the getter and the setter.

Add an @synthesize for the operandStack in the implementation.

18. And then, as an exercise, let's type in exactly what @synthesize would generate. Check out the next screenshot to make sure you did this right.
// CalculatorBrain.m
// Calculator
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorBrain.h"

@interface CalculatorBrain : NSObject
@property (nonatomic, strong) NSMutableArray *operandStack;
@end

@implementation CalculatorBrain
@synthesize operandStack;

-(NSMutableArray *)&operandStack
{
    return operandStack;
}

-(void)setOperandStack:(NSMutableArray *)&anArray
{
    operandStack = anArray;
}

-(void)pushOperand:(double)operand
{
}

-(double)performOperation:(NSString *)operation
{
    double result = 0;
    // perform operation here, store answer in result
    return result;
}
@end
Task 6

Task: Create the Model of our MVC for the calculator brain.

19. Now that we have a stack, let's try to push an operand onto it. Note that `NSMutableArray` is an array of **objects** and a `double` is a primitive type, not an object. But there is a class called `NSNumber` which can be used to wrap primitive types into an object.

Wrap the operand with an `NSNumber` by using the `NSNumber`'s class method `numberWithDouble:` and store it into a “pointer to `NSNumber`” object called `operandObject`.

20. Then, add the `operandObject` to the stack with the `addObject:` method. Again, check out the next screenshot to make sure you did this right.
Recall that all @properties start out nil (zero). And recall that sending a message to nil does nothing. So this line of code will be doing nothing. Somewhere we need to initialize the operandStack @property.
Task 6

Task: Create the Model of our MVC for the calculator brain.

21. There is a perfect place to initialize `operandStack`. Its getter!

If someone tries to get `operandStack` and it is not initialized, initialize it before returning it. This sort of initialization is called “lazy instantiation” and is a common paradigm in iOS.

To create a mutable array (or any object in general) you should use the `[[NSMutableArray alloc] init]` construct. But first you must test if the operand stack is `nil`. Notice the implicit testing of a pointer (to see if it is `nil`) is `if (operandStack == nil)`. You could also say `if (!operandStack).

Check out the next screenshot to make sure you did everything right.
// CalculatorBrain.m
// Calculator
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorBrain.h"

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

@implementation CalculatorBrain
@synthesize operandStack;

-(NSMutableArray *)&operandStack
{
    if (operandStack == nil)
    {
        operandStack = [[NSMutableArray alloc] init];
    }
    return operandStack;
}

-(void)setOperandStack:(NSMutableArray *)&anArray
{
    operandStack = anArray;
}

-(void)pushOperand:(double)operand
{
    NSNumber *operandObject = [NSNumber numberWithDouble:operand];
    [self.operandStack addObject:operandObject];
}

-(double)performOperation:(NSString *)operation
{
    double result = 0;
    // perform operation here, store answer in result
    return result;
}
@end
Task 6

Task: Create the Model of our MVC for the calculator brain.

22. Recall the fact that @synthesize creates an instance variable with the same name as the property is dangerous. We can avoid this potential accident by having @synthesize use a different name for its instance variable than the name of the property.

Prefix the instance variable with an underscore (_operandStack).

23. Only setters and getters should access the instance variable directly. Fix the setter and getter to access the instance variable by its new name, _operandStack.

24. We are not going to do anything with the setter, so you can delete it. Remember that @synthesize will always create whichever setter and/or getter that you do not.

Check out the next screenshot to make sure you did everything right.
// CalculatorBrain.m
// Calculator
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "CalculatorBrain.h"

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

@implementation CalculatorBrain
@synthesize operandStack = _operandStack;

-(NSMutableArray *)operandStack
{
    if (_operandStack == nil)
    {
        _operandStack = [[NSMutableArray alloc] init];
    }
    return _operandStack;
}

-(void)pushOperand:(double)operand
{
    NSNumber *operandObject = [NSNumber numberWithDouble:operand];
    [self.operandStack addObject:operandObject];
}

-(double)performOperation:(NSString *)operation
{
    double result = 0;
    // perform operation here, store answer in result
    return result;
}
@end
Task 6

Task: Create the Model of our MVC for the calculator brain.

25. It is time to implement the `performOperation:` method. Let's start by implementing the + operation.

First, we should verify that the operation is @"+" with the `isEqualToString:` method. Then we must pop two operands and add them into the `result`. Let's write this.

Check out the next screenshot to make sure you did everything right.
// CalculatorBrain.h
// Calculator
// Created by Radu-Tudor Ionescu on 3/1/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.

#import "Foundation/Foundation.h"

@interface CalculatorBrain : NSObject
- (void)pushOperand:(double)operand;
- (double)performOperation:(NSString *)operation;
@end

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

#import "CalculatorBrain.h"

@property (nonatomic, strong) NSMutableArray *operandStack;

@end

@implementation CalculatorBrain
@synthesize operandStack = _operandStack;

- (NSMutableArray *)operandStack
{
    if (_operandStack == nil)
    {
        _operandStack = [[NSMutableArray alloc] init];
    }
    return _operandStack;
}

- (void)pushOperand:(double)operand
{
    NSNumber *operandObject = [NSNumber numberWithDouble:operand];
    [self.operandStack addObject:operandObject];
}

- (double)performOperation:(NSString *)operation
{
    double result = 0;
    if ([operation isEqualToString:@"+"])
    {
        result = [self popOperand] + [self popOperand];
    }
    return result;
}
@end
Task 6

Task: Create the Model of our MVC for the calculator brain.

26. Notice the error that tells us that `popOperand` is not implemented.

Implement `popOperand` method by getting the `lastObject` in our `operandStack` array, then returning that last object's `doubleValue`.

Note that `lastObject` is a method that `NSMutableArray` inherits from `NSArray` which returns the last object in the array. All the objects in our `operandStack` array are `NSNumber`s and `NSNumber` responds to the method `doubleValue` (which returns a `double`).

27. We got the value off the end of the array, but we also need to “pop” it off by removing it. We have to send the `removeLastObject` message to the `operandStack`, but not before testing if the `lastObject` is not nil.

The next slide shows you a screenshot with the `popOperand` method implementation.
Unlike `lastObject`, sending `removeLastObject` to an array that is empty will raise an exception (index out of bounds) and crash your program! That is why we check to see if we actually got a non-nil `operandObject` from the array before trying to call `removeLastObject`.

Sending `lastObject` to an array that is empty just returns `nil` (it does not raise an exception or do anything bad). And sending any message to `nil` returns `nil`.

Unlike `lastObject`, sending `removeLastObject` to an array that is empty will raise an exception (index out of bounds) and crash your program! That is why we check to see if we actually got a non-nil `operandObject` from the array before trying to call `removeLastObject`. 
Task 6

Task: Create the Model of our MVC for the calculator brain.

28. Implement the *, -, and / operations. Make sure to get the order of operands correct for the - and / operations! For example, the input “6 Enter 2 -” should be 4, not -4.

29. Finally, we must be sure to push the result back onto the stack so that the next operation we are asked to do will use it.

The screenshot from the next slide shows the final implementation of our CalculatorBrain Model.
We return zero on divide by zero instead of “not a number”.
We are sort of a “return zero on failure” calculator!

Notice that this time we send the isEqualToString: to the constant NSString the compiler creates for us when we use the @"*" notation. That NSString is every bit as much an NSString as operation is.

We return zero on divide by zero instead of “not a number”. We are sort of a “return zero on failure” calculator!
Task 7

Task: Finish the Controller implementation by connecting it to the Model.

1. Switch back to our Controller to finish it off. Click on the name of the file in the top bar and use it to navigate to your Controller’s implementation (CalculatorViewController.m).

2. Your Controller’s implementation (.m) file should appear on the left. Note that the Automatic Assistant switched the right-hand side to our Controller’s header file (instead of our Model’s). But actually, we want our Model’s header file on the right because we are going to use it in our Controller. But to have our Model’s header file on the right-side in Automatic mode, we need to create the relationship between our Controller and the Model in our code. We do that by #importing our Model into our Controller’s implementation.

#import the Model’s header file into our Controller’s implementation. Make sure you save the implementation at this moment.

The screenshot from the next slide shows the project configuration at this moment.
This is where you switch from the Model's implementation to the Controller's implementation (CalculatorViewController.m).
Task 7

Task: Finish the Controller implementation by connecting it to the Model.

3. Put the CalculatorBrain header file on the right in Assistant Editor. Check the next slide to understand how to do this.

Your Model’s header file should appear on the right. It is common to put the header file (public API) of a class on the right-hand side of the screen as you are working in your implementation on the left.

4. All we need to do next is to add a private @property in our Controller that points to our Model. And then use an instance of our Model to implement operationPressed: and enterPressed.

Add a @property (called brain) to hold a pointer from our Controller to our Model.
Click here and navigate through “Includes” to get CalculatorBrain.h to show up on the right. If you have not #imported CalculatorBrain.h into your CalculatorViewController.m or haven’t saved it, CalculatorBrain.h may not appear here.
Task 7

Task: Finish the Controller implementation by connecting it to the Model.

5. Add a `@synthesize` to create the brain setter & getter. Again, we will use the most common naming convention for the corresponding instance variable (underscore plus property name).

6. We need to implement the getter of the `brain @property` in order to lazily instantiate the `brain` (in its getter method).

See how to set things up in the next screenshot.
// CalculatorBrain.h

@interface CalculatorBrain : NSObject
-
- (void)pushOperand: (double)operand;
- (double)performOperation: (NSString *)operation;
@end

#import <Foundation/Foundation.h>
@interface CalculatorViewController : UIViewController
@property (nonatomic) BOOL isInTheMiddleOfEnteringANumber;
@property (nonatomic, strong) CalculatorBrain *brain;
@end

@implementation CalculatorViewController
@synthesize display; 
@synthesize isInTheMiddleOfEnteringANumber; 
@synthesize brain = _brain;
-
- (void)viewDidLoad
{
    [self setDisplay: nil];
    [super viewDidLoad];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
-
- (CalculatorBrain *)brain
{
    if (_brain == nil) _brain = [[CalculatorBrain alloc] init];
    return _brain;
}
-
- (IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.isInTheMiddleOfEnteringANumber)
    {
        self.display.text = [self.display.text stringByAppendingString:digit];
    }
    else
    {
        self.display.text = digit;
        self.isInTheMiddleOfEnteringANumber = YES;
    }
}
-
- (IBAction)enterPressed
{
}
-
- (IBAction)operationPressed:(UIButton *)sender
{
}
Task 7

Task: Finish the Controller implementation by connecting it to the Model.

7. Next let’s handle the Enter button being touched. All we need to do is push the double value of the display into our Model (self.brain).

Note that NSString responds to doubleValue as well. It tries to parse a double out of whatever is in the string. Luckily, there is no way to put anything but a number into our Calculator’s display.

8. Of course, touching Enter means we are no longer in the middle of typing a number.

We need to set userIsInTheMiddleOfEnteringANumber to NO.

See how to set things up in the next screenshot.
@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;
@synthesize brain = _brain;

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

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

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

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

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

// CalculatorBrain.h

@interface CalculatorBrain : NSObject

@end

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

#import <Foundation/Foundation.h>

@interface CalculatorViewController : UIViewController

@end

-(void)pushOperand:(double)operand;
-(double)performOperation:(NSString *)operation;
@end
Task 7

Task: Finish the Controller implementation by connecting it to the Model.

9. To implement `operationPressed:` we have to look at the button that sent us the action to determine which operation to perform. As for digits, we look at the `sender's currentTitle`.

10. Then we just perform the operation using our Model and store the result in local variable of `double` type.

11. We also need to update the display with the result of the operation. We can use the `NSString's stringWithFormat:` class method to convert the `double` result into a string object.

See how to implement the `operationPressed:` method on the next slide.
@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;
@synthesize brain = _brain;

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

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

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

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

-(IBAction)operationPressed:(UIButton *)sender
{
    NSString *operation = [sender currentTitle];
    double result = [self brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%.2f", result];
}
@end
Task: Finish the Controller implementation by connecting it to the Model.

12. By the way, when an operation is pressed and the user is in the middle of typing a number, let’s do an implicit Enter. For example, 6 Enter 4 - would be the same as 6 Enter 4 Enter -.

The screenshot from the next slide shows the final implementation of the operationPressed method.

13. All done! Hit Run again. Touch 360 Enter 9 /. The result should be 40.

14. Then touch 77 +. The result should be 360 / 9 + 77 = 117.
@implementation CalculatorViewController
@IBOutlet display;
@IBOutlet userIsInTheMiddleOfEnteringANumber;
@IBOutlet brain = _brain;
-(void)viewDidLoad
{
    [self setDisplay:nil];
    [super viewDidLoad];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
-(CalculatorBrain *)brain
{
    if (_brain == nil) _brain = [[CalculatorBrain alloc] init];
    return _brain;
}
-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.userIsInTheMiddleOfEnteringANumber)
    {
        self.display.text = [self.display.text stringByAppendingString:digit];
    }
    else
    {
        self.display.text = digit;
        self.userIsInTheMiddleOfEnteringANumber = YES;
    }
}
-(IBAction)enterPressed
{
    [self.brain pushOperand:self.display.text doubleValue:][
    userIsInTheMiddleOfEnteringANumber = NO;
    }
-(IBAction)operationPressed:(UIButton *)sender
{
    if (self.userIsInTheMiddleOfEnteringANumber) [self enterPressed];
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%.2f", result];
@end
Assignment 1

Assignment: Your calculator already works with floating point numbers (e.g. if you touch the buttons 3 Enter 4 / it will properly show the resulting value of 0.75), however, there is no way for the user to enter a floating point number. Remedy this. Allow only legal floating point numbers to be entered (e.g. “192.168.0.1” is not a legal floating point number).

Hint: Add another button for “.” and the action `pointPressed`. There’s an `NSString` method which you might find quite useful. It’s called `rangeOfString:`. Check it out in the documentation. It returns an `NSRange` which is just a normal C struct which you can access using normal C dot notation. For example, consider the following code:

```c
NSString *greeting = @"Hello There Joe, how are ya?";
NSRange range = [greeting rangeOfString:@"Bob"];
if (range.location == NSNotFound) { ... /* no Bob */ }
```

Be careful of the case where the user starts off entering a new number by pressing the decimal point, e.g., they want to enter the number “.5” into their calculator. Handle this case properly.
Assignment 2

Assignment: Add a “C” button that clears everything (for example, the display in your View, the operand stack in your Model, any state you maintain in your Controller, etc.). Make sure 3 7 C 5 results in 5 showing in the display.

Hint: Add another button for “C” and the action clearPressed. You will have to add API to your Model to support this feature. To be more precise you will have to declare and implement another method in CalculatorBrain to emptyOperandStack. Note that you can remove all objects from an NSMutableArray by calling a single method. Look into the NSMutableArray documentation for this.
Congratulations!