Welcome to CS193p!
This document is a walkthrough of the demonstration done in class. You will need this walkthrough to do your first homework assignment.

Yellow Bubbles mean “do something.”

Green Bubbles are just for “information.”

Red Bubbles mean “important!”

Green Bubbles with small text is for “minor notes.”

Welcome to CS193p!
This document is a walkthrough of the demonstration done in class. You will need this walkthrough to do your first homework assignment.

Yellow Bubbles mean “do something.”

Green Bubbles are just for “information.”

Red Bubbles mean “important!”

Green Bubbles with small text is for “minor notes.”
Launch Xcode 4 and click here to create a new project.

As you create projects, they will appear here.
Click on this iOS Application template. It creates a simple MVC application.

Xcode 4 can be used to develop both iOS and Mac OSX applications.

These buttons are used to select a template which Xcode 4 uses to generate some code to get you started.

Then click Next.
Our first application is going to be an RPN Calculator.

An RPN calculator pulls its operands off of a stack. Users type a number in, then hit Enter, then another number, then Enter again, then hit an operation button which works on those two “entered” numbers.

These fields describe your project. We’ll be filling them in during the next few slides.
The name of our project is going to be “Calculator” so type that in here.
Enter edu.stanford.cs193p.yourname

This field is used to uniquely identify your application.

Using an entity’s reverse DNS lookup string is a pretty good way to get a unique identifier.
We don’t want the names of the classes generated by the template to be too generic. That’s why we specify this prefix.

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

We don’t want the names of the classes generated by the template to be too generic. That’s why we specify this prefix.

Usually we use the name of the application for this prefix. In fact, older versions of Xcode would automatically do this whether we wanted it or not.
Our first application is going to be for the iPhone (not iPad). At least for starters.

A Universal application runs on both iPhone and iPad. In a Universal application, the iPad and the iPhone each has its own UI design (since they have different UI idioms). Xcode provides tools for designing two different UIs in the same application.
Storyboards are a new (iOS 5) way to organize your MVCs' Views. We're going to use them.
ARC is a fantastic upgrade to the compiler (in iOS 5) which causes it to generate all the code necessary to manage the memory allocation of objects. We definitely want that to be on!
We might have time near the end of the course to cover Unit Testing, but we won’t be creating any for our first application.

With the 3 switches in the shown positions, click Next.
Xcode wants to know where to store this project's directory.

Navigate to a directory called "Developer" in your home directory (create it if needed), then click Create to create your project directory inside ~/Developer.

If you don't have a Developer folder in your home directory, you can create it with this New Folder button.

We will definitely be covering source control in this course. But not for this first project, so leave this switch turned off.

Home directory.

"Developer" folder inside the home directory. There are no projects in it currently.

Navigate to a directory called "Developer" in your home directory (create it if needed), then click Create to create your project directory inside ~/Developer.
Congratulations, you've created your first iOS Application!

You'll probably want to make this window as big as possible. Xcode loves screen real-estate!

There's a lot of stuff in this window, but we won't be covering any of it in this first application.
Let's open up and look at our MVC's View by clicking on `MainStoryboard.storyboard`.

The Single View Application template we chose at the beginning has created a simple MVC for us.

Our MVC's View is inside `MainStoryboard.storyboard`.

`CalculatorViewController` is the code for our MVC's Controller.

`CalculatorViewController.m` is the code for our MVC's Controller.

Our MVC's View is inside `MainStoryboard.storyboard`.

We'll have to create our MVC's Model ourselves later.

Let's open up and look at our MVC's View by clicking on `MainStoryboard.storyboard`.

Don't worry about `CalculatorAppDelegate` for this project.
This is our MVC’s View.
It starts out blank, of course.

Click here to Run the application.

This should be selected.
If it’s not, that would explain why you’re not seeing your MVC’s View.
This is the “scheme chooser.” It lets you choose where to run your application. For example, the iPhone Simulator, iPad Simulator or on a device.

Xcode is building the application.

If you press and hold this Run button, other run options will be available, but we’re just using plain “Run” for now.
Hopefully you see this.

This bar that appeared at the bottom is part of the console and debugger. We'll cover that a bit later.
Congratulations again!
You’ve run your first iOS app in the iOS Simulator. You will use the simulator for most (but not all) of your assignments.

When you are done glorying in the wonder of it all, click the Stop button.

This button shows or hides the debugger/console. You can click it if you want to see.
This is the Document Outline. It contains a hierarchical, iconic view of all the objects in your MVC View. We're not going to use it during this first application. But you might want to check it out occasionally as you build this application.

This is the Navigator. It shows all the files in your project in a hierarchical arrangement of folders. The arrangement of folders is conceptual, it does not necessarily match what's in the file system. This area can also show symbols, search results, breakpoints, issues, etc. (see the icons at the top).

If you mouse over this button, you'll see this tooltip telling you that it will hide the Document Outline. Click the button to hide the Document Outline.
We need to see our MVC Controller now. But we still want our MVC View on screen at the same time. The way to have two things on the screen at once is to use the Assistant Editor. It is shown/hidden using this “butler” icon.

Click here to show the Assistant Editor. When an MVC View is showing, it will by default bring up the View’s Controller. That’s exactly what we want.
Here is the header (.h) file for our MVC Controller. It contains its public methods and properties and also defines its superclass (all Controllers in iOS inherit from UIViewController).

Notice the @interface - @end syntax.

UIKit.h imports all the iOS user-interface classes. #import is like #include, but better.

ALT-clicking on our Controller's header file (CalculatorViewController.h) would also have brought it up in the Assistant Editor, but it would have taken the Assistant Editor out of “Automatic mode.”

When the Assistant Editor is in Automatic mode, it will always be trying to put something sensible up in the right-hand side of the Editor.

We don’t need the Navigator at the far left either, so let’s hide it by turning this button off.
Let's make even more room for our code on the right by dragging this center bar to the left.
We're going to start building the user-interface in our MVC View. To do that, we'll need a text label and some buttons. We get those from the Object Library in the Utilities area (which is brought up via this button).

```objective-c
#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController
@end
```

Click here to bring up the Utilities area.
Utilities

The top part of this area shows information (identity, attributes, help, connections, dimensions) about the currently selected thing (file, object, etc.) at the left.

The bottom is a library of items (objects, code snippets, file templates).

// CalculatorViewController.h
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.
//
#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController
@end
Click on the Object Library (it might already be selected).

File Inspector
Shows information about the file containing the selected item.

The top bar will be darker gray if the selected item is in this half of the Assistant Editor.

Quick Help
If the selected item at the left has some documentation reference, this shows a "summary" version of it.

Media Library
Images, sounds, etc.

Object Library
Buttons, text fields, controllers, etc.

File Template Library
Templates for storyboards, classes, etc.

Code Snippet Library
Snippets of code for common tasks.
Occasionally, Xcode might download some documentation for you in the background.

Click on your View to select it.

Some objects (those appropriate to dragging into your View) should appear in the Object Library.

Attributes Inspector
See and set attributes of the selected item.

Size Inspector
Position and size the selected item.

Connections Inspector
Connections between your View and Controller.
Scroll down to find a Label object.
Let's start building our Calculator's View by dragging out a text label to be the Calculator's “display.”

Notice the dashed blue lines which Xcode displays as you drag which help you line things up nicely.

Labels are instances of the class `UILabel` in the iOS SDK.
Click on the Attributes Inspector. You should see attributes of the Label you just created.

Notice the little “handles” around the label. These can be used to resize it. That's exactly what we're going to do next.
Grab the lower right “handle” on the label and resize it. Use the dashed blue guidelines to pick a good size.

This little indicator will show you the exact size you’re resizing to.
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 by clicking on this button in the inspector.

Note that changes in the inspector are reflected immediately in the View.
Let's also make the font bigger. Click this tiny up arrow to increase the font size. 24 point Helvetica is nice.
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.

You can also edit the label's text in the Attributes Inspector.
It should look like this after you type 0.

Our Controller needs to be able to talk to its View. For example, in this case, we need to be able to update the display as digits are pressed (and with results from operations). We can make this connection between Controller and View directly with the mouse...
HOLD DOWN CTRL while mousing down and dragging a line from the text label directly into the code of our Controller.

If you do not hold down the CTRL key, this will not work.

You’ll notice that as you ctrl-drag over your code, an indicator will appear, making it easy to see exactly where in the file your outlet code will be.
Xcode now wants to know what kind of connection we want to make between the Controller and this object in the View. In this case, it has correctly guessed that we want an outlet. An outlet is just a property of our Controller through which we can talk to an element in our View.

The destination of this connection is our Controller since that's where we ctrl-dragged to.
We're going to name this outlet display (since it is the display of our Calculator).
An outlet is a pointer to an object (a UILabel in this case).

A strong pointer means the UILabel will stick around until we are done using the UILabel.

A weak pointer means the UILabel will only stick around as long as somebody else has a strong pointer to it. As soon as no one else has a strong pointer to an object that we have a weak pointer to, that object will go away and our pointer to it will be cleared and we won’t be able to talk to it (because it will be gone).

Since this window already has a strong pointer to this UILabel, weak is a good choice here.
Click Connect to create a property (called display) in our Controller which will point to this UILabel in our View.
Voilà!
Xcode has added a `@property` to our MVC Controller which is a pointer to a `UILabel` object. It has also hooked this `@property` up to the text label we dragged out into our MVC View.

So, whenever our Controller sends messages to the display `@property`, it will be talking to this `UILabel` instance.

@IBOutlet is just a word Xcode throws in here so that it can remember that this is an outlet `@property`. It doesn't actually mean anything to the compiler.

`UILabel` * is the type of this `@property` (that means "pointer to a UILabel object").

This property is a `weak` pointer.

nonatomic means "not thread-safe" -- more on that later.

Important!
Mouse over (i.e. hover the mouse over, do not click on) this little icon to see where this `@property` is connected. Notice that the label highlights.
If you click on this icon, it will show you a list of all the storyboards this property is hooked into. Remember that we said that a single application can support multiple UIs (e.g., iPhone & iPad). It does this with multiple storyboards. This is how a single Controller can support those different UIs.
Mousing over the only item in the list will select the label too.
Click here to switch to our Controller’s implementation so we can make some more connections to our View.

Our Controller’s interface (header).

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

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

Our Controller’s implementation.
This @synthesize is very important. We'll cover it in a moment.

Notice that the @implementation here does not specify the superclass. Only the @interface in the .h (header) file does that.

Welcome to the implementation (.m file) of your MVC Controller!

```objective-c
// CalculatorViewController.m
// Calculator

// Created by CS193p Instructor
// Copyright (c) 2011 Stanford University

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

#pragma mark - View lifecycle

-(void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

-(void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}
```
Xcode has added some extra code that we don't need for this application, so select everything from after the @synthesize display...
... all the way down to (but not including) the `@end`. Then hit delete to delete it.
Note the `@synthesize` that Xcode automatically added to our Controller’s implementation when it created the display `@property` (it did this when we ctrl-dragged to create the display outlet).

`@synthesize` creates two methods (display and `setDisplay:`).

The method `setDisplay:` is used by iOS to hook the `UILabel` up to the display `@property` at runtime (i.e. set the value of the pointer).

The method `display` is used by us to get this pointer to the `UILabel` so that we can send messages to the `UILabel`.

`@synthesize` also creates an instance variable to store this pointer.

We’re going to do more with `@property`s later in this document.
Okay, now it's time to add our Calculator's keypad buttons.

Drag a Round Rect Button from the Object Library to your View.
Grab the middle-right “handle” on the button and resize it. A width of 64 points works extremely well, so use that.

Remember that the term **outlet** refers to a @property through which we send messages to something in our View from our Controller (display is an outlet).

We use the term **action** to mean a method that is going to be sent from an object in our View to our Controller when something interesting happens in the user-interface.

So our next step is to specify the **action** that this UIButton is going to send to our Controller when the user touches it.

**Important!**
HOLD DOWN CTRL while mousing down and dragging a line from the button directly to the text area where your code is.

Xcode correctly guesses that you want to create an action with this ctrl-drag rather than an outlet.
When you release the mouse, this “action” dialog will appear.

This is the name of the action method.

This is the object to which the action message will be sent.

This is the kind of touch event that will cause this action to get sent.

This specifies the format of the message (more on that later).
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).

Then press Connect.

You can leave the rest of the fields alone (the defaults are fine for this button).
This is your first Objective-C method declaration!

Every argument (like `sender`) to an Objective-C method is preceded by a part of the method's name (like `digitPressed`) and a colon.

`id` is the type of the argument `sender`. `id` means “pointer to an object of any class”.

You might be surprised that this does not read “id *”. But that would make no sense because the type “id” is already a pointer so “id *” would be a pointer to a pointer. `id` does not mean “object of any class”, it means “pointer to an object of any class”.

`IBAction` is exactly the same as `void` (i.e. this method does not return any value). Xcode uses it instead of `void` just so it can tell an action method from other methods with a similar form.

Important!
Similar to an outlet, you can mouse over this little icon and see which object(s) in your View send(s) this message. Notice how the button highlights.

Okay, now every time this button is touched, `digitPressed:` is going to be sent to our Controller with the `UIButton` itself as the message’s sender argument.

Important!
If you click on this icon, you’ll see all the objects in all storyboards which send this action to your Controller.
And mousing over one in the list will highlight it (just like mousing over the icon does).
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 *”.

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`.
We need more buttons!

Copy and paste our first button to make another button.

The copied button will send the same action (digitPressed:) as the original.
Now move the copied button to line up with the original (the dashed blue lines are awesome here).
Copy and paste again.

```swift
// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender {
}
@end
```
And again.
Three at a time!
Using copy and paste and the grid lines, create the entire keypad for the Calculator. It should look like this.

```
// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender
{
}
@end
```
Double-click on a button to make its text editable.
Then type the number that goes in the appropriate spot.

```swift
// CalculateViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender
{
}
@end
Do this for all the buttons. Your keypad should now look like this.

// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController
@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender
{
}
@end
When you start building more complicated user-interfaces, it will be very important to be able to see your outlet and action connections. You can do this by right-clicking on any object in your MVC's View.

This shows that the 9 button sends `digitPressed:` to Calculator View Controller (your Controller) when a touch lifts up inside its borders.

There are no outlets which point to this button.

You can disconnect this action by clicking on this little x.

You can ctrl-drag from these little circles to make connections too.

The Connection Inspector will show all of this as well.
Mouse over this 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).
Right-click on the display.

This UILabel is connected via the display outlet to your Controller.
Again, mousing over this connection will highlight the whole view (that's its way of showing you that this outlet is connected to the Controller).

The whole View is highlighted again.
Right-click on this icon that represents your Controller ...

... then mouse over this “Button - 4” entry.

Notice that the 4 button above highlights.
... then mouse over this outlet.

Notice that the display label highlights.

Don't worry about this view outlet, we'll explain it later in this course.
If you have a warning here that `digitPressed:` is not implemented, it's probably because you need to save your `CalculatorViewController.m` file.
If it ever seems like your actions aren’t being sent or talking to your outlets from your code does nothing, this is a good place to go check to be sure everything’s wired up correctly.

Occasionally you might accidentally hook something up to the wrong action method name or hook something up to two different actions at the same time, so check here if things seem to be acting sort of messed up when it comes to outlets and actions.
It is time to write the code inside `digitPressed:` that will get executed whenever any of these buttons gets touched ...

We won’t need the Utilities area for a while, so close it.
Let's start by declaring a local variable called `digit` which will be of type “pointer to an `NSString` object.”

Remember that ALL Objective-C objects are allocated in the heap and we keep pointers to them. It would never be right to say “`NSString digit`” (i.e. without the *).

Yes, we could say `id digit =` here. But we should use static typing whenever possible.

Important!
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.

To send a message to an Objective-C object we use a syntax that starts with an open square bracket `[`, then a pointer to the object we want to send the message to (sender), then a space, then the name of the message to send (currentTitle).

Xcode tries diligently to help you as you type. It is smart about sensing what's going on in your code. Note that since we statically-typed `sender`, Xcode is only suggesting `UIButton` methods.
The message sending syntax ends with a `]` to match the `[` it started with.

Note: this method has no arguments.
We'll see a method with an argument later.

Uh oh, we have a problem. This little triangle is a warning that there's a problem with this line of code. A red dot here would mean an error in the code (which won't compile).
Click on the triangle to find out what the warning is.

This warning appears to be correct. We do not (yet) use the local variable digit in this method.

You should never submit code in this course that has any warnings or errors.
A very simple debugging technique is to log information to the console. This is very easy to do in Xcode. There is a printf-like function whose output goes to the console called NSLog().

Add an NSLog() to print the digit chosen by the user to the console.

The first argument to NSLog() is the printf-like formatting string. Note, however, that it is an NSString, not a const char*, so we need an @"" (a “constant”NSString), not just plain "". We'll cover constant NSString objects like this later in this document.

This printf-like formatting string only has one % element in it. And it's a very special one, just for Objective-C. %@ means that the corresponding argument to print is an object. Specifically it means “send the message description to the object and use the results as the string to print.” Our argument in this case is digit, an NSString. NSString returns itself from its description method. All objects in iOS respond to the message description because NSObject, the root of all classes in iOS, implements it.
Let's run with this `NSLog()` in place.

Notice that the console starts out hidden.
Debug Area (variables view and/or console)

Click here to show the debug area at the bottom.

This button will also show the debug area at the bottom.
We'll cover the debugger in a future lecture.

Click here to show the console and the variables view at the same time.

Clears the console output.
Click here to show only the console.

Click here to show only the console.

Scroll to the bottom of the console.

Scroll to the bottom of the console.
You should see output in the console from your NSLog().

All console logs are timestamped.
Stop the simulator.

Select this line of code and hit delete.

Let's remove the NSLog() and continue implementing digitPressed:

You can review old console output from the Navigator.

Notice that the debug area automatically disappears when you stop running.
Our warning is (correctly) back.

```c
// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController
@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
}
@end
```
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 (which is, itself, a “pointer to a UILabel”).

As you'll see, we don't really need a local variable here, but we're using it just to make it very clear how to get a pointer to our display.

Xcode knows that we (`self`) only respond to one method that starts with “disp” (our `display` @property getter).
The setter and getter methods were created (implemented) for us by this `@synthesize`.

(The `@synthesize` also created some storage for this pointer, but we'll talk about that later.)

A `@property` is nothing more than a setter method and a getter method. Here we are calling the getter method using normal method notation. The setter method will be called by the system at run-time to wire this outlet up.

Here's a reminder of what our Controller's header file looked like.

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

The setter and getter methods were created (implemented) for us by this `@synthesize`. (The `@synthesize` also created some storage for this pointer, but we'll talk about that later.)
It turns out that @property s are so important that there is a special Objective-C syntax just for @property setters and getters.

It's called “dot notation.”

Express calling the getter of our display @property using dot notation instead.

The old version is shown in this end-of-line comment.

These are two syntactically different expressions of EXACTLY the same thing (i.e., calling the display @property’s getter).
Now that we have a pointer to our display UILabel, let's send it a message to find out what text is currently in it. The message to send is called (appropriately) `text`.

Add this 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`.

```objective-c
NSString *currentDisplayText = [myDisplay text];
```
Here you can see that the text method in UILabel is actually a @property (it's the getter of the text property).

This is a good time to show how to link to the documentation. Xcode contains extensive reference documentation for all methods/classes. A quick lookup can be done simply by selecting a method or class name and choosing “Quick Help for Selected Item” from the Help menu.

Or you can hold down the option key and click on a term to get quick help. Hold down option and click on on the text method to get quick help for it.

Useful key-binding here. All key-bindings are settable in Xcode’s Preferences.

Quick Help window.
But often you want the full details on a method or class. You can find it by choosing “Search Documentation for Selected Text” in the Help menu. Or you can click on links in the Quick Help (as shown below).

@synthesize display;

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

Click on the word `text` here to open up the full documentation on that method.

Or you can hold down the option key and double-click on a term go to the documentation.
Each method’s arguments, return values, and what it does are explained in the documentation.

This is the Organizer window.

You can use this field to search the documentation ...

... and you can set bookmarks that you can see here.

text
The text displayed by the label.
@property nonatomic, copy) NSString *text

Discussion
This string is nil by default.

Availability
Available in iOS 2.0 and later.

Related Sample Code
iPhoneCoreDataRecipes
LocateMe
TopSongs
URLCache
XMLPerformance
Declared In
UILabel.h

textAlignment
The technique to use for aligning the text.
@property nonatomic) NSTextAlignment textAlignment

Discussion
This property applies to the entire text string. The default value of this property is NSTextAlignmentLeft.

Availability
Available in iOS 2.0 and later.

Related Sample Code
BatteryStatus
BubbleLevel
Available in iOS 2.0 and later.

Declared in
UILabel.h

textRectForBounds:limitedToNumberOfLines:

Returns the drawing rectangle for the label’s text.

- `(CGRect)textRectForBounds:(CGRect)bounds limitedToNumberOfLines:(NSInteger)numberOfLines

Parameters

bounds
The bounding rectangle of the receiver.

numberOfLines
The maximum number of lines to use for the label. The value 0 indicates there is no maximum number of lines and that the rectangle should encompass all of the text.

Return Value
The computed drawing rectangle for the label’s text.

Discussion
You should not call this method directly. This method should only be overridden by subclasses that want to change the receiver’s bounding rectangle before performing any other computations. Use the value in the numberOfLines parameter to limit the height of the returned rectangle to the specified number of lines of text. For this method to be called, there must be a prior call to the sizeToFit or sizeThatFits: method. Note that labels in UITableViewCell objects are sized based on the cell dimensions, and not a requested size.

The default implementation of this method returns the original bounds rectangle.

Availability
Available in iOS 2.0 and later.

Declared In
UILabel.h
UILabel Class Reference

Inherits from
 UIView : UIResponder : NSObject

Conforms to
 NSCoding
 NSCoding (UIView)
 NSAppearance (UIView)
 NSAppearanceContainer (UIView)
 NSObject (NSObject)

Framework
/System/Library/Developer/SDKs/iOS5.0.sdk/System/Library/Frameworks/UIKit.framework

Availability
Available in iOS 3.2 and later

Declared in
UILabel.h

Related sample code
 iPhoneCoreDataRecipes
 SimpleFTPSample
 SimpleNetworkScreams
 Testator
 URLCache

Overview

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with
This is an overview of UILabel.

The UILabel class implements a read-only text view. You can use this class to draw one or multiple lines of static text, such as those you might use to identify other parts of your user interface. The base UILabel class provides control over the appearance of your text, including whether it uses a shadow or draws with a highlight. If needed, you can customize the appearance of your text further.

The default content mode of the UILabel class is UIViewContentModeRedraw. This mode causes the view to redraw its contents every time its bounding rectangle changes. You can change this mode by modifying the inherited contentMode property of the class.

New label objects are configured to disregard user events by default. If you want to handle events in a custom subclass of UILabel, you must explicitly change the value of the userInteractionEnabled property to YES after initializing the object.

**Tasks**

Accessing the Text Attributes

- text property
- font property
- textColor property
- textAlignment property
- lineBreakMode property
- enabled property

Sizing the Label’s Text

- adjustFontToFitWidth property
- baselineAdjustment property
- minimumFontSize property
- numberOfLines property

Managing Highlight Values

- highlightedTextColor property
Here is a table of contents of all of `UILabel`'s @properties and methods.

Scroll down a little bit more.

There's the text @property.

Scroll down a little bit more.

Here is a table of contents of all of `UILabel`'s @properties and methods.

The names are links you can click.
Search for `UIButton`. And click in the results. The reason there are two of each topic here is that the documentation set for both iOS 5 and iOS 4.3 are loaded in this Xcode (you may have only one). You can be sure you have clicked on the right version (iOS 5's) here.
Scroll down a little bit.

Here's `currentTitle`. Turns out it's a `@property` too. We should be using dot notation for it as well!
Click here to browse the documentation.

Icons

Last Revision: Version 1.0, 2010-10-22
This sample demonstrates the proper use of application icons on iOS.

Build Requirements: iOS 4.1 or later
Runtime Requirements: iOS 3.2 or later

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

This sample demonstrates the proper use of application icons on iOS. This is a universal binary that supports iPhone/iPod touch/iPad and includes support for high resolution displays.

Each icon has one dimension of the pixel dimensions on it to display which icon is being used by various areas of iOS. The various icons display when using the Homescreen, Spotlight, the Settings app, different devices, and when creating an Ad Hoc build and adding it to iTunes.
Now that we know that text is actually a @property, let's use dot notation.
And there’s really no need for this local variable `myDisplay`.

So let’s select and copy its value (`self.display`) ...
... then select where we use it and paste.
Yes, it is perfectly legal to have multiple dots in an expression like this.

Do it!
Now we can delete the previous line.
Dot notation makes the code very simple and easy to follow.
Next we need to append the digit that the user just touched onto the end of what is currently in the display.

`stringByAppendingString:` is a method in the `NSString` class (obviously). It returns a new `NSString` which is a copy of the receiving `NSString` (currentDisplay) with the argument (digit) appended onto the end.
And finally we’ll use `UILabel`’s text `@property`’s setter (`setText:`) to set our display `UILabel`’s text to the new string with the digit appended to the end. And yes, we should be using dot notation here, but we’ll (briefly) look at it using method notation just to be clear what we are doing.
Note that the TAB key can be used to jump to the next argument to a method. This is what an argument looks like after having been tabbed to. Now you just type the argument you want (newDisplayText in this case) and it'll replace the (NSString *).
Again, we would probably never use this normal method syntax to set a `@property` like this. It's just for illustrative purposes.
Switch to using dot notation to set the UILabel’s text @property.

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.
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.
Then we can delete the previous line.
Ditto currentDisplay. Not really needed. Copy...
```swift
// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@implementation CalculatorViewController

@synthesize display;

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    NSString *currentDisplayText = self.display.text;
    self.display.text = [currentDisplayText stringByAppendingString:digit];
}
@end
```
... and delete previous line.
Again, see how dot notation has made this line of code very easy to read.
Okay, let's Run and see if this works!
Hmm, that leading 0 doesn’t look quite right, does it?
Well, at least it is appending digits!
Let's fix this problem of 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).

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.

Important!

Stop your application running in the simulator.
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.

Add a private @interface to your implementation file.

Note the ()

Important!

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.
Add a boolean property to track whether the user is in the middle of entering a number.

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. We will always use it in this course. 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).

There's no strong or weak here because a BOOL is not a pointer.
Let's see why there's a warning here.

Actually, there are 2 warnings on this line of code! Click on the 2 to see both of them.
The problem is that we've declared a `@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!

We almost always use `@synthesize` to implement our `@property` getters and setters. But even if we do, we can always implement the getter and/or the setter ourselves. Our implementation will trump `@synthesize`'s.

`@synthesize` also creates an instance variable to store our `@property` in (which is nice).

`@synthesize` doesn't care whether your `@property` is public (declared in the header) or private (declared in the implementation file).

No more warnings!

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

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.

Notice that we use dot notation to call the getter of our new `@property`.

What value does `userIsInTheMiddleOfEnteringANumber` start out with?

Good question. 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).

For a `BOOL` like `userIsInTheMiddleOfEnteringANumber`, zero means `NO`.
else if the user is not in the middle of typing, just start off a new number with the digit that was touched.

And, of course, in this case, we are now in the middle of entering a number.
Okay, that should do it.
Let's run again.
```swift
// Calculator

// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@interface CalculatorViewController :

@end

@interface CalculatorViewController :

@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

- (IBAction)digitPressed:(UI闩utton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.userIsInTheMiddleOfEnteringANumber)
    {
        self.display.text = [self.display.text stringByAppendingString:digit];
    } else{
        self.display.text = digit;
    self.userIsInTheMiddleOfEnteringANumber = YES;
}
@end
```
```c
// CalculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"

@interface CalculatorViewController :
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

-(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;
    }
}
@end
```
Okay, all working now!

Next we need to add some more buttons (for operations and Enter).
Bring back the Utilities area.
Press Stop
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!
// calculatorViewController.m
// Calculator
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.
//
#import "CalculatorViewController.h"

@interface CalculatorViewController :
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize isUserInTheMiddleOfEnteringANumber;

-(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
Ctrl-drag to create this button's action.

```swift
#import "CalculatorViewController.h"

@interface CalculatorViewController()
@property (nonatomic) BOOL userIsInTheMiddleOfEnteringANumber;
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (self.userIsInTheMiddleOfEnteringANumber) {
        self.display.text = [self.display.text stringByAppendingString:str];
    } else {
        self.display.text = digit;
        self.userIsInTheMiddleOfEnteringANumber = YES;
    }
}
@end
```
We'll call this action operationPressed:

Then click Connect.
Now use copy/paste to create 4 operation buttons. Again, do not copy/paste a digit button to make an operation button!

You should statically type this to UIButton *.
And set the title of each to these four operations.
Drag a Round Rect Button from the Object Library to the View.

Do NOT copy and paste either a digit button or an operation button to make an Enter button (drag a new one out). The Enter button will have a different action than either digit buttons or operation buttons.

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).
After you resize it and set its title to Enter, please 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.
We'll call this action `enterPressed`.

But there's something a little different about this action method. We don't need the `sender` argument because there's only one Enter key. We can control whether an action message includes the sender as an argument with this pull-down.

```
#import "CalculatorViewController.h"

@interface CalculatorViewController
@property (nonatomic) BOOL userIsInTheMiddleOfEnteringANumber;
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize userIsInTheMiddleOfEnteringANumber;

-(IBAction)digitPressed:(UIButton *)sender
{
    NSString *digit = [sender currentTitle];
    if (digit == [NSCharacterSet numberCharacterSet] characterRangeOfString:digit)
    {
        self.display.text = [self.display.text textByAppendingString:digit];
    }
    else
    {
        self.display.text = digit;
        self.userIsInTheMiddleOfEnteringANumber = YES;
    }
}
@end
```
... and click Connect.

Change the Arguments to None ...

... and click Connect.
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.
Before we switch to writing our Model, it'd 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).
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 here.
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.
We want to create a new Objective-C class, so click here.

Then click Next.
Our Model is going to be a direct subclass of NSObject.

NSObject is the root superclass of all objects in iOS. Classes inherit some nice generic functionality from NSObject including the method mentioned earlier called description which returns an NSString representation of the object which is useful for debugging with NSLog().

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

We’re going to call our Model’s class “CalculatorBrain,” so type that here.

Often you will override description to return something nice.
Make sure to click here so that you put your Model's .m and .h files in the same place as all of your other .m and .h files.

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.

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.
When you have the locations above set properly, click Create.
Xcode has created stubs for both the header and implementation of our MVC's Model.

Close the Navigator to make space.

Hopefully your Model's files (CalculatorBrain.m and .h) are here with the rest of your source. If not, you can drag them to where you want them.

Of course, you can also make more groups here by right clicking anywhere in the Navigator to bring up a context-sensitive menu.
We can always get back to our View + Controller via this tab.

Notice that Xcode has automatically renamed this tab. You can name it yourself by double-clicking on it if you wish.

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 this method which will provide a way to push operands onto our Calculator’s stack of operands.

Hmm, adding this API to our header file has created a warning in our implementation!

The argument to this method is a double-precision floating point number.

This method returns nothing.

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). We’ll discuss that later in the course.
Let's take a look at that warning.

"Incomplete implementation."
Makes sense.
We have not yet implemented `pushOperand`.
Now add the method that performs a given operation using the operands on the stack. We're 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.

The argument to this method is a pointer to an object (an `NSString`).

This method returns a `double` (the result of performing the operation).
Let's stub out both of our methods in our implementation.
Let's have this return a default value of zero for now. We'll put the actual guts of this in here in a moment.
How are we going to store our stack of operands? We're 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.

We need a private `@interface` again so that we can declare the array we need to store the operand stack.
We'll 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.

Add a `@property` for our operand stack.

Recall that `nonatomic` means not thread-safe. `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`.
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`!
You can see that it's quite simple. It just stores and retrieves from an instance variable it creates.

Add an `@synthesize` for the `operandStack`.

And then, as an exercise, let's type in exactly what `@synthesize` would generate.

The fact that `@synthesize` creates an instance variable with the same name as the property is dangerous. More on this in a moment.
Now that we have a stack, let's try to push an operand onto it.

Uh oh.

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

This is a class method of `NSNumber`. Don’t worry about the syntax of this for now.

Wrap the operand with an `NSNumber`.

Looks nice!

But there’s a problem with this line of code.

This is a class method of `NSNumber`. Don’t worry about the syntax of this for now.
Recall that all `@property`s 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`.

There's 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 is how you create an object. We'll talk about `alloc` and initialization later.

Notice the implicit testing of a pointer to see if it is `nil`. You could also say `if (operandStack == nil)`. Either is fine.

This sort of initialization is called “lazy instantiation” and is a common paradigm in iOS.

Important!
There's danger here!
What if we accidentally left out this "self."?

@implementation CalculatorBrain
@synthesize operandStack;

- (NSMutableArray *)operandStack
{
    if (!operandStack) {
        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 the operation here, store answer in result
    return result;
}
This would be bad because we would be accessing the synthesized instance variable directly and thus not calling the getter. As a result, we would not be getting lazy instantiation!

And yet there is no compiler warning to help us notice that.

You can delete this and see that no error will appear!
Changing the name used by `@synthesize` to create its instance variable will make it very clear when we accidentally forget `self`. Notice that there are errors now when we access the instance variable directly.

We can avoid this potential accident by having `@synthesize` use a different name for its instance variable than the name of the property. We do that using this equals-sign syntax.

Prefixing the property name with an underbar is the most common naming convention for an instance variable created by `@synthesize`.

Changing the name used by `@synthesize` to create its instance variable will make it very clear when we accidentally forget `self`. Notice that there are errors now when we access the instance variable directly.
Fix the setter and getter to access the instance variable by its new name, `_operandStack`.

Then fix this error by putting the `self` back in.

ONLY setters and getters should access the instance variable directly!! There are rare exceptions, but for now, stick to this rule.
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.

The only time you couldn't implement only one of the setter or the getter is if the `@property` is not nonatomic (will not happen in this course). Because, in that case, you'd have to match `@synthesize`'s locking code.
Scroll down to the bottom so we can make room to type in our implementation of `performOperation:`.
Let's try to implement the + operation.

We'll check this error out in a moment.

"+" is a constant NSString. The compiler creates an NSString object for you. Notice the @! Without the @, "" means const char *. You almost never want a const char * in iOS. You wantNSString objects. Forgetting the @ is a common coding mistake.

Important!
The problem is that we need to implement `popOperand`.

Notice that if a warning or error is too verbose to fit in the line, you can mouse over it to get a tooltip with the full text of the warning or error.
Implement `popOperand` by getting the `lastObject` in our `operandStack` array, then returning that last object's `doubleValue`.

All the objects in our `operandStack` array are `NSNumber`s and `NSNumber` responds to the method `doubleValue` (which returns a `double`).

`lastObject` is a method that `NSMutableArray` inherits from `NSArray` which returns the last object in the array.

But this is not quite right yet ...
We got the value off the end of the array, but we also need to "pop" it off by removing it.

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.

Important!
Implement the * operation.

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 must be sure to get the order of operands correct!
The input “6 Enter 2 -” should be 4, not -4.

Implement the - operation.

We must be sure to get the order of operands correct! The input “6 Enter 2 -” should be 4, not -4.
Implement the \(/\) operation.

Again, we're getting the order of operands right.

We return zero on divide by zero instead of “not a number”. We're sort of a “return zero on failure” calculator!
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.
That's it for our Model!

Now we're going to switch back to our Controller to finish it off...
Click on the name of the file in the bar and use it to navigate to your Controller's implementation (CalculatorViewController.m).
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're 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 
\texttt{#import}\n
our Model into our Controller's implementation ...

Your Controller's implementation (.\texttt{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're 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 \texttt{#import}\n
our Model into our Controller's implementation ...

Your Controller's implementation (.\texttt{m}) file should appear on the left.
#import the Model's header file into our Controller's implementation.
# Make sure this file is saved.
//(This icon should not be grayed out.)

// CalculatorViewController.m
// Calculator
//
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import "CalculatorViewController.h"
#import "CalculatorBrain.h"

@interface CalculatorViewController : UIViewController

@property (nonatomic) BOOL isInTheMiddleOfEnteringANumber;
@end

@implementation CalculatorViewController

@synthesize display;
@synthesize isInTheMiddleOfEnteringANumber;

-(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;
    }
}
@end

// CalculatorViewController.h
// Calculator
//
// Created by CS193p Instructor.
// Copyright (c) 2011 Stanford University.

#import <UIKit/UIKit.h>

@interface CalculatorViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *
@end
If you have not #imported CalculatorBrain.h into your CalculatorViewController.m or haven't saved it, CalculatorBrain.h may not appear here.

Click here and navigate through “Includes” to get CalculatorBrain.h to show up on the right.
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.
Okay, we are in the home stretch. All we need to do is to add a pointer (@property) in our Controller that points to our Model. And then use an instance of our Model to implement operationPressed: and enterPressed.
Need getter and setter, of course.
Add an @synthesize to create the brain setter & getter. We’ll use the most common naming convention for the corresponding instance variable (underbar plus property name).
While we're at it, let's lazily instantiate the brain in its getter method.

Yes, we should have gone back and added $= \_\text{display}$ and $= \_\text{userIsInTheMiddleOfEnteringANumber}$ to these other @synthesize s.
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).

Yes, 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!

We're using our Model here.

Notice the nesting of method calls.

Stanford CS193p Fall 2011
Of course, touching Enter means we are no longer in the middle of typing a number.
To implement operationPressed: we have to look at the button that sent us the action to determine which operation to perform.

If you statically typed sender, you could use dot notation here to get currentTitle, e.g., sender.currentTitle.
Now we just perform the operation using our Model.
And then update the display with the result.

```objc
NSString *operation = [sender currentTitle];
double result = [self.brain performOperation:operation];
self.display.text = [NSString stringWithFormat:@"%g", result];
@end
```

stringWithFormat: is a class (not instance) method. Don't worry about that for now.

stringWithFormat: takes a printf-like format string. %g means “floating point number”.
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 -.
All done!
Hit Run again.

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

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

- (IBAction)operationPressed:(id)sender
{
    if (self.userIsInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%g", result];
}
@end
- (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]];
    self.userIsInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)operationPressed:(id)sender
{
    if (self.userIsInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%.2f", result];
}
@end
- (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]];
    self.userIsInTheMiddleOfEnteringANumber = NO;
}

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

@end
-(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;
    }
}

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

-(IBAction)operationPressed:(id)sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%g", result];
}
@end
- (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]];
    self.userIsInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)operationPressed:(id)sender
{
    if (self.userIsInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }

    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%.2f", result];
}
@end
Hopefully this is 521 divided by 6!
-(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;
    }
}

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

-(IBAction)operationPressed:(id)sender
{
    if (self.isUserInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }
    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%.g", result];
}
@end
- (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]];
    self.userIsInTheMiddleOfEnteringANumber = NO;
}

- (IBAction)operationPressed:(id)sender

    if (self.userIsInTheMiddleOfEnteringANumber) {
        [self enterPressed];
    }

    NSString *operation = [sender currentTitle];
    double result = [self.brain performOperation:operation];
    self.display.text = [NSString stringWithFormat:@"%g", result];

@end
Congratulations, you've built your first iOS application!

That's all there is!

Hopefully this is 521 divided by 6 plus 23!