Introduction To iOS

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 13 — 10/04/2011

Goals of the Lecture

- Present an introduction to the iOS Framework
- Coverage of the framework will be INCOMPLETE
    - We'll see the basics… there is a lot more to learn

History

- iOS is the name (as of version 4.0) of Apple's platform for mobile applications
  - The iPhone was released in the summer of 2007
  - According to Wikipedia, it has been updated 41 times since then, with the 42nd update slated to occur when version 5.0 is released (currently in private beta)
  - iOS apps can be developed for iPhone, iPod Touch and iPad; iOS is used to run Apple TV but apps are not currently supported for that platform

iOS 4.2

- I'll be covering iOS 4.3.5 which is the current "official version"
  - Version 4.3.5 was released on July 25, 2011; it was a security-related update
  - Most significant update was 4.2 in November 2010 when iOS was unified across all three hardware platforms (iPhone, iPad, Apple TV)

Acquiring the Software

- To get the software required to develop in iOS
    - Follow the instructions in Lecture 12
    - Installing XCode via the App Store installs all the software you need to develop for OS X and iOS

Tools

- Xcode: Integrated Development Environment
  - Provides multiple iOS application templates
  - Provides drag-and-drop creation of user interfaces
- iPhone Simulator
  - Provides ability to test your software on iPhone & iPad
- Instruments: Profile your application at runtime

iOS Platform (I)

- The iOS platform is made up of several layers
    - The bottom layer is the Core OS
        - OS X Kernel, Mach 3.0, BSD, Sockets, File System, …
    - The next layer up is Core Services
        - Collections, Address Book, SQLite, Networking, Core Location, Threading, Preferences, …

iOS Platform (II)

- The iOS platform is made up of several layers
    - The third layer is the Media layer
        - Core Audio, OpenGL and OpenGL ES, Video and Image support, PDF, Quartz, Core Animation
    - The final layer is Cocoa Touch (Foundation/UIKit)
        - Views, Controllers, Multi-Touch events and controls, accelerometer, alerts, web views, etc.
- An app can be written using only layer 4 but advanced apps can touch all four layers

Introduction to Interface Builder

- This slide used to say "Introduction to Interface Builder"
    - But Interface Builder is no more

Introduction to ~~Interface Builder~~

- This slide used to say "Introduction to Interface Builder"
  - But Interface Builder is no more
- So …

Welcome to XCode's XIB Editor!

- Doesn't quite have the same ring to it!

- So, I'm going to continue to call it "Interface Builder"
  - even though technically that's no longer its name

Introduction to Interface Builder (I)

- Interface Builder used to be a separate application
  - Now, it's functionality is integrated into XCode
  - Regardless, its functionality is extremely powerful
    - It provides a drag and drop interface for constructing the graphical user interface of your apps
    - The interface is stored in a .xib file: "XML Interface Builder"
      - When deployed, .xib is converted to .nib, a binary format

Introduction to Interface Builder (II)

- The GUIs created by Interface Builder are
  - **actual instances** of the **underlying UIKit classes**
    - When you save a .xib file, you "freeze dry" the objects and store them on the file system
    - When your app runs, the objects get reconstituted and linked to your application logic

Introduction to Interface Builder (III)

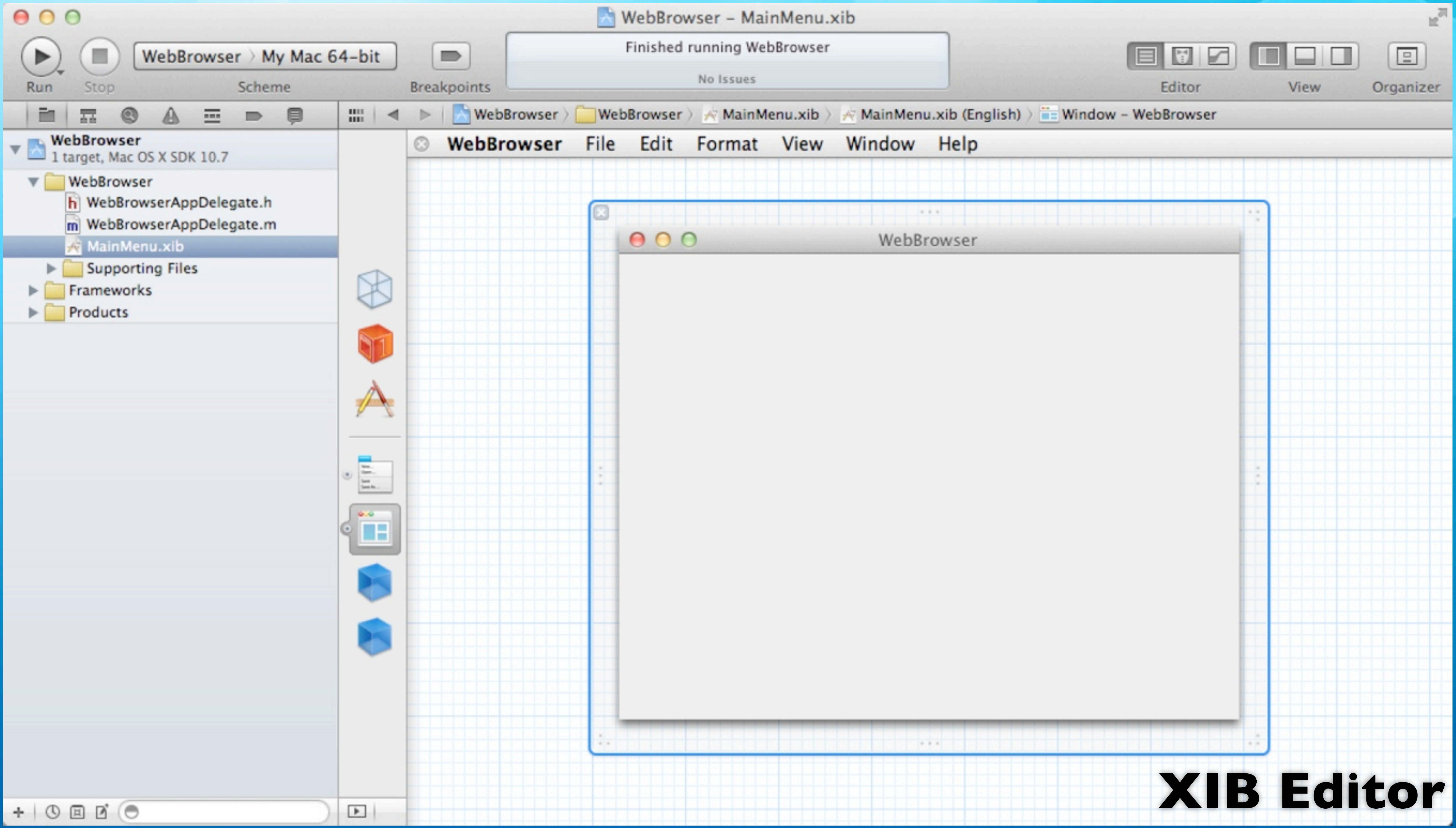
- This object-based approach to UI creation is what provides interface builder its power
    - To demonstrate the power of Interface Builder, let's create a simple web browser without writing a single line of code
        - The fact that we can do this is testament to the power of object-oriented techniques in general

Step One: Create Project

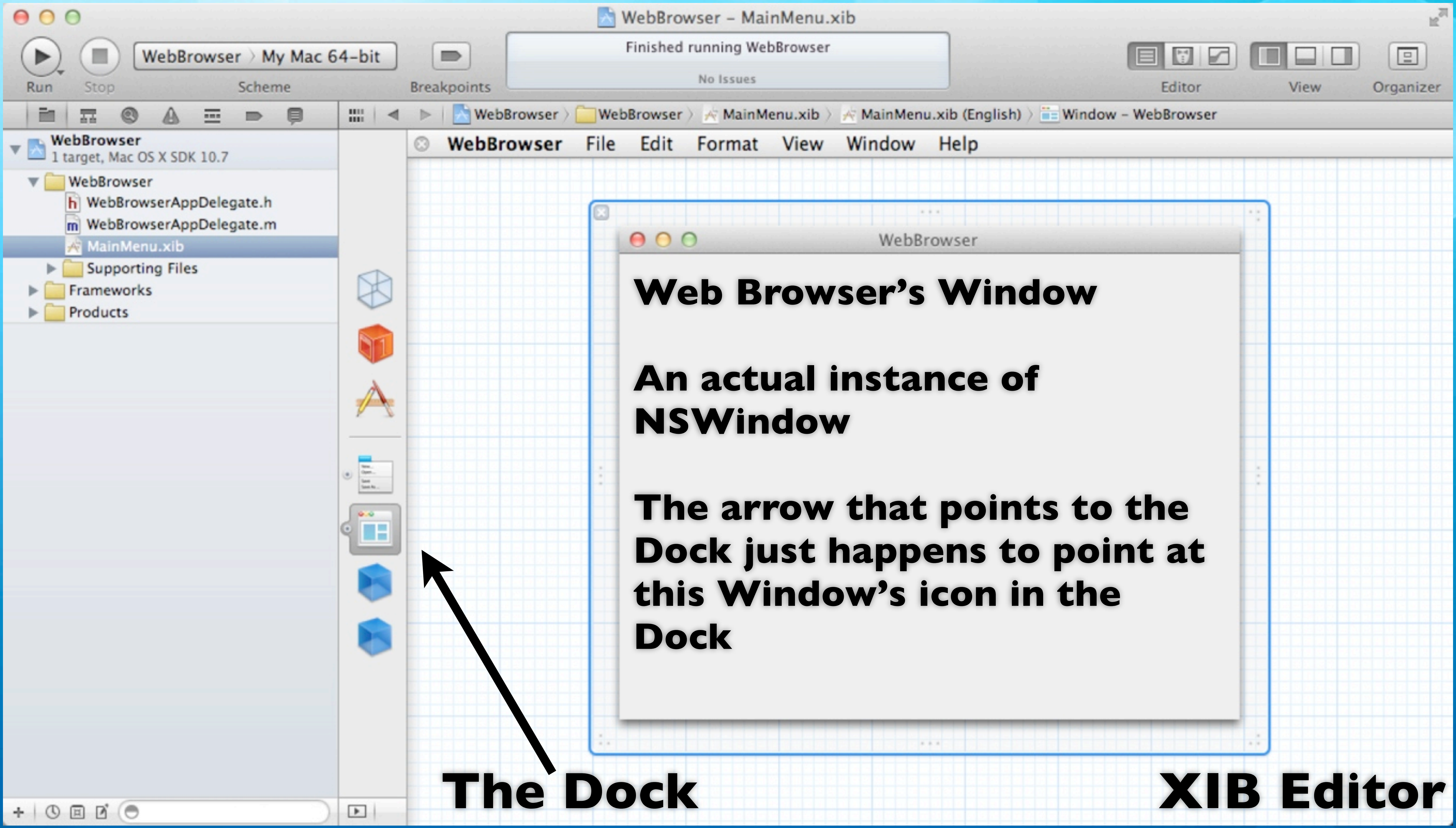
- Launch XCode and create a MacOS X application (we'll get to iOS in a minute)
  - Unlike last time, select Cocoa Application rather than Command Line Tool
  - Name the project WebBrowser and save it to disk
    - Click Build and Run to see that the default template produces a running application
      - It doesn't do anything but create a blank window

Step Two: Launch IB

- Click on MainMenu.xib
    - As mentioned earlier, a .xib is an XML file that stores the freeze dried objects that interface builder creates;
    - You will never edit it directly
- The .xib editor will open


WebBrowser – MainMenu.xib
Run
Stop
WebBrowser › My Mac 64-bit
Scheme
Breakpoints
Finished running WebBrowser
No Issues
Editor
View
Organizer
WebBrowser › WebBrowser › MainMenu.xib › MainMenu.xib (English) › Window – WebBrowser
WebBrowser
1 target, Mac OS X SDK 10.7
WebBrowser
WebBrowserAppDelegate.h
WebBrowserAppDelegate.m
MainMenu.xib
Supporting Files
Frameworks
Products
WebBrowser File Edit Format View Window Help
WebBrowser

XIB Editor


WebBrowser – MainMenu.xib
Run
Stop
WebBrowser › My Mac 64-bit
Scheme
Breakpoints
Finished running WebBrowser
No Issues
Editor
View
Organizer
WebBrowser › WebBrowser › MainMenu.xib › MainMenu.xib (English) › Window – WebBrowser
WebBrowser
1 target, Mac OS X SDK 10.7
WebBrowser
WebBrowserAppDelegate.h
WebBrowserAppDelegate.m
MainMenu.xib
Supporting Files
Frameworks
Products
WebBrowser File Edit Format View Window Help
WebBrowser

Web Browser's Window

An actual instance of NSWindow

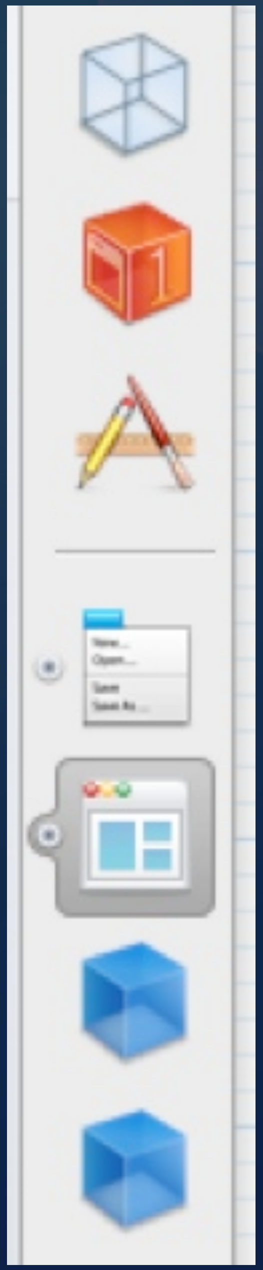
The arrow that points to the Dock just happens to point at this Window's icon in the Dock

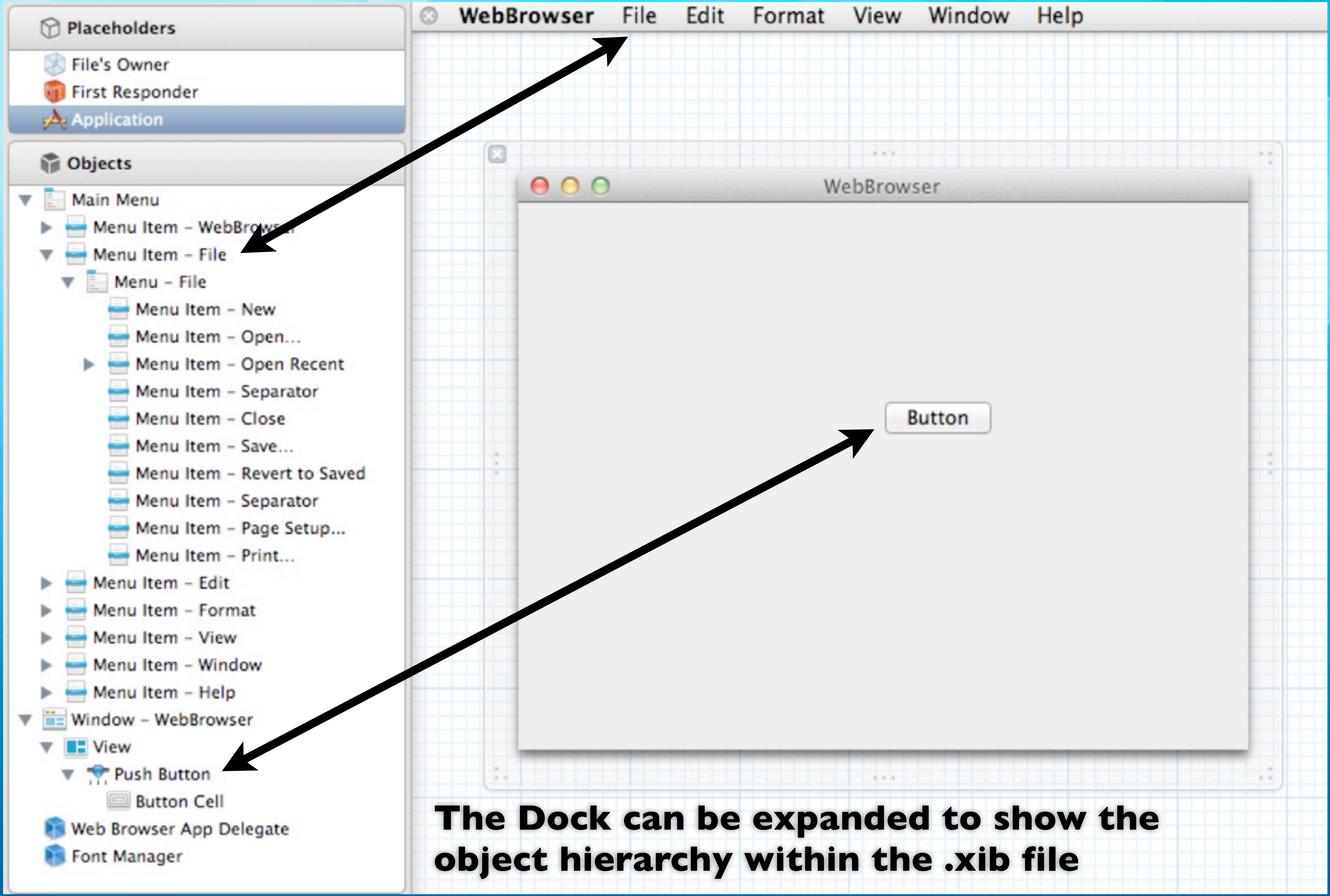
The Dock

XIB Editor

The Dock

- It holds "placeholders" and "instances"
- Placeholders represent objects "outside" of the .xib file
    - They exist before the .xib file is loaded and get connected at run-time
- Instances represent objects contained in this particular .xib file
    - Some instances, like windows, are containers and can contain many sub-instances





The Dock can be expanded to show the object hierarchy within the .xib file

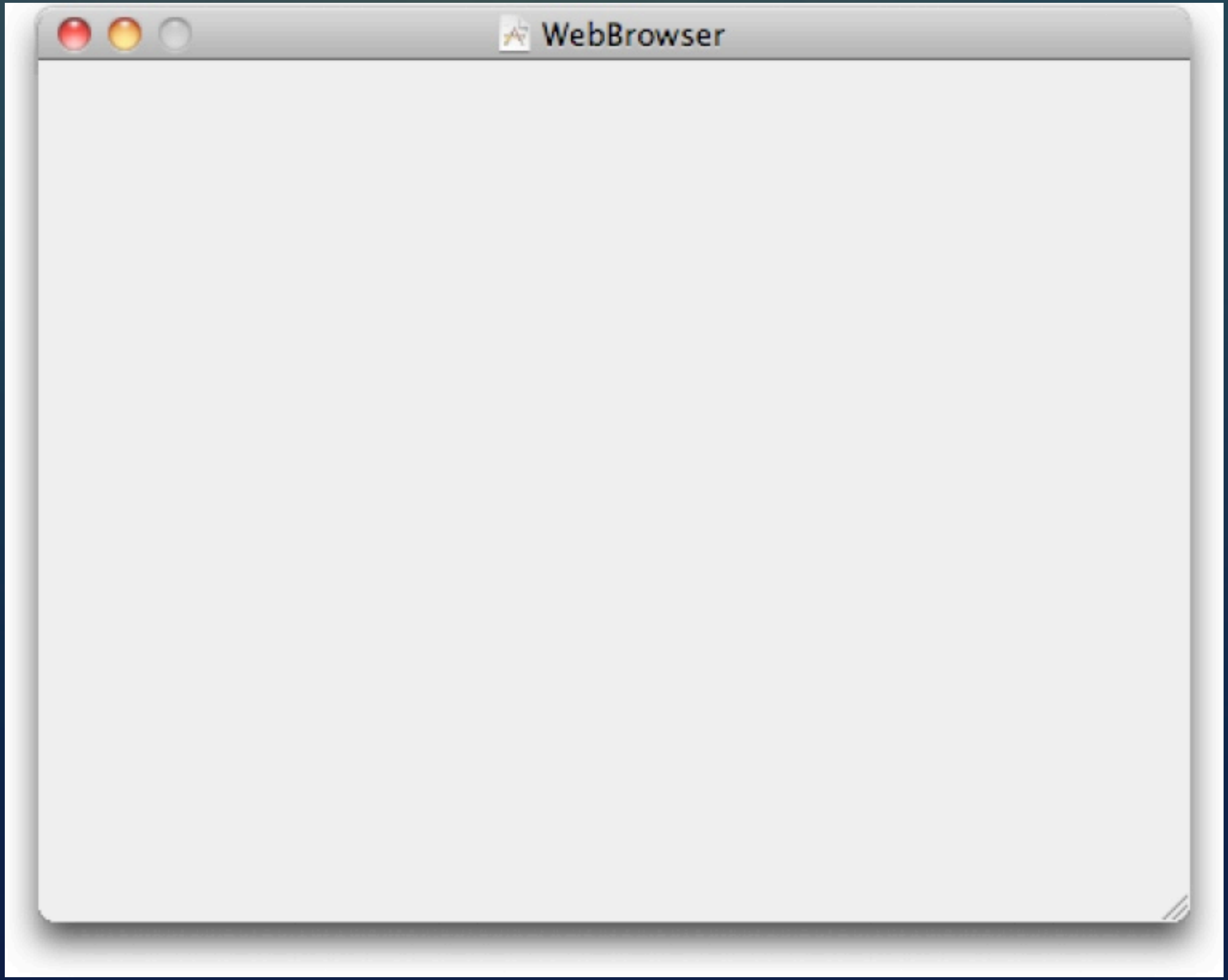
Object Connections (I)

- The cool thing about Interface Builder is that you can
  - instantiate instances of objects (widgets, controllers, …)
- and then
  - connect them together via drag and drop

Object Connections (II)

- Say a button should call a controller when clicked
  - You can drag from the button
  - to the controller's dock icon
  - and then select the method the button should invoke
- Equivalent to
  - [button setTarget: controller]
  - [button setAction: @selector(handleClick:)]

Our default Window



This is the window you saw when you first ran the application

Exciting, isn't it?

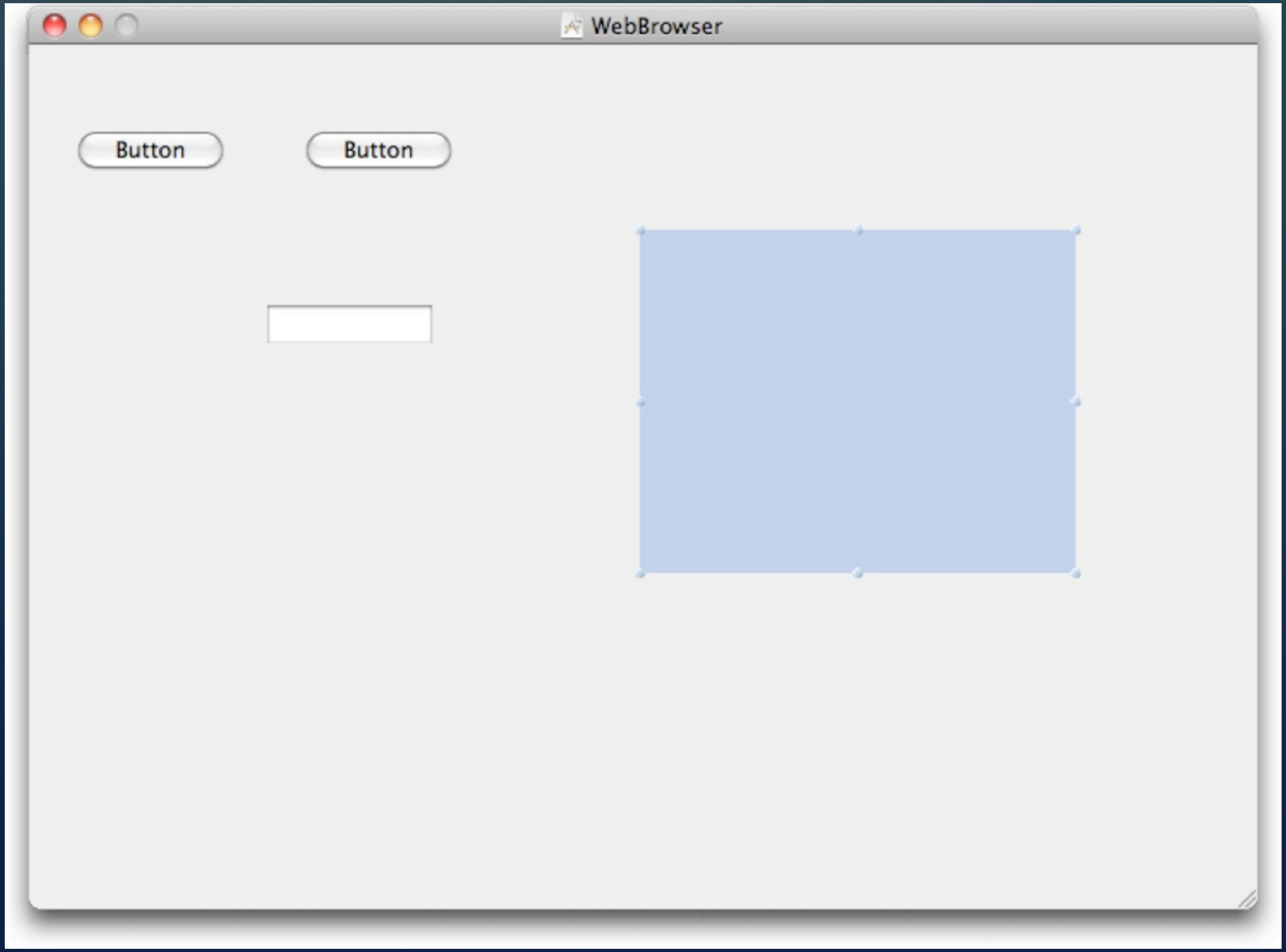
Try changing its size, save the document and run the program

You'll see your changes reflected; because the window in the editor and the window at run time are THE SAME WINDOW!

Step 3: Acquire Widgets

- Invoke View ⇒ Utilities ⇒ Show Object Library to bring up the widgets that can be dragged and dropped onto our window
- Type button in the search field and then drag two "push buttons" on to the window
    - It doesn't matter where you drag them just yet
- Type text field in the search field and then drag a "text field" on to the window (ignore "text field cell")
- Type "web" and drag a "web view" to the window

Results of Step 3



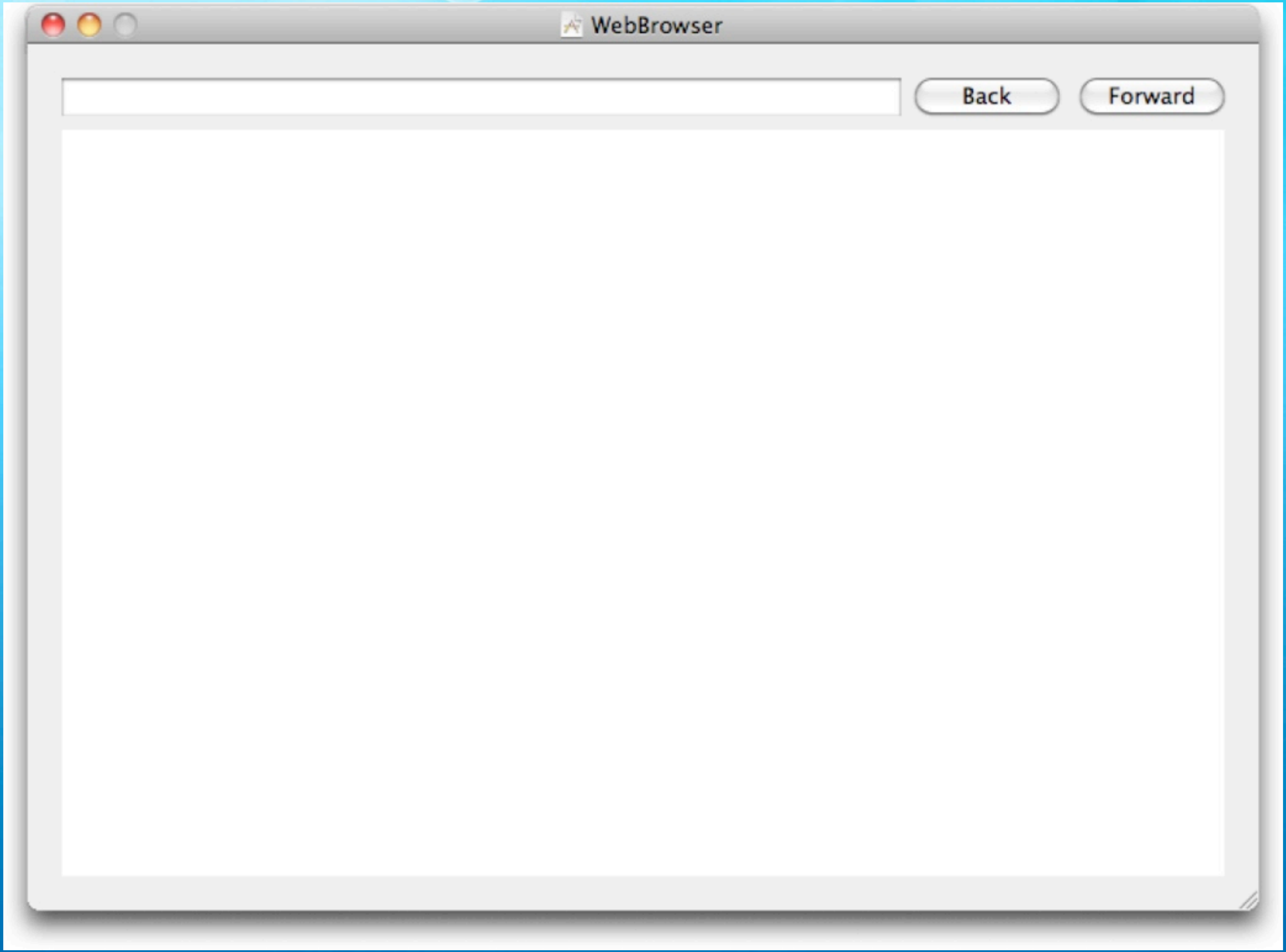
Window now has four widgets but they are not yet placed where we want them

Step 4: Layout Widgets (I)

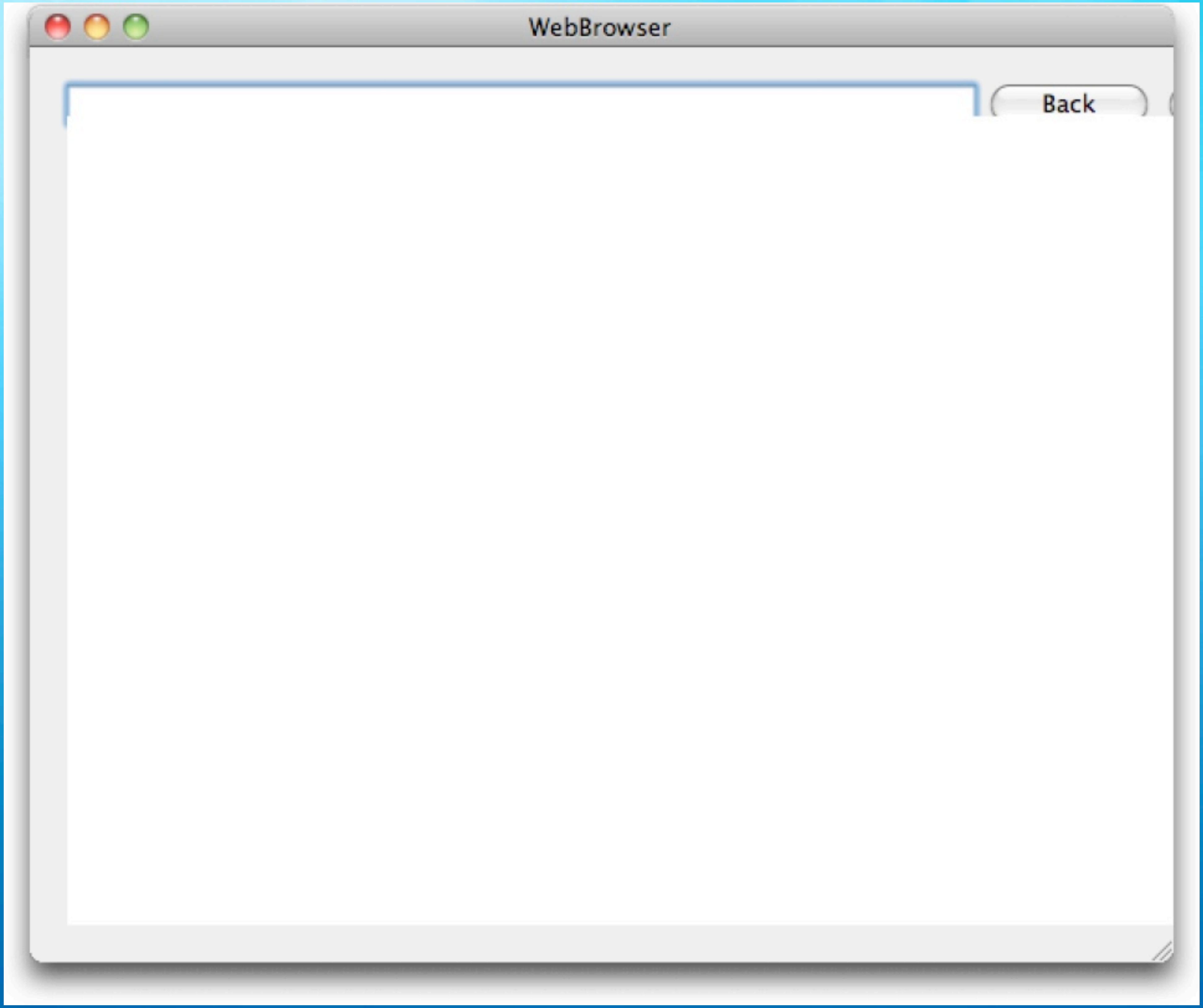
- Put the buttons in the upper right corner
  - Use the guides to space them correctly
  - Double click on them and name one "Back" and one "Forward"
- Put the text field in the upper left corner and stretch it out so it ends up next to the buttons
  - Again use the guides to get the spacing right
  - These guides help you follow Apple's human interface guidelines

Step 4: Layout Widgets (II)

- Expand the Web view so that it now fills the rest of the window, following the guides to leave the appropriate amount of space
    - Your window now looks like the image on the next slide

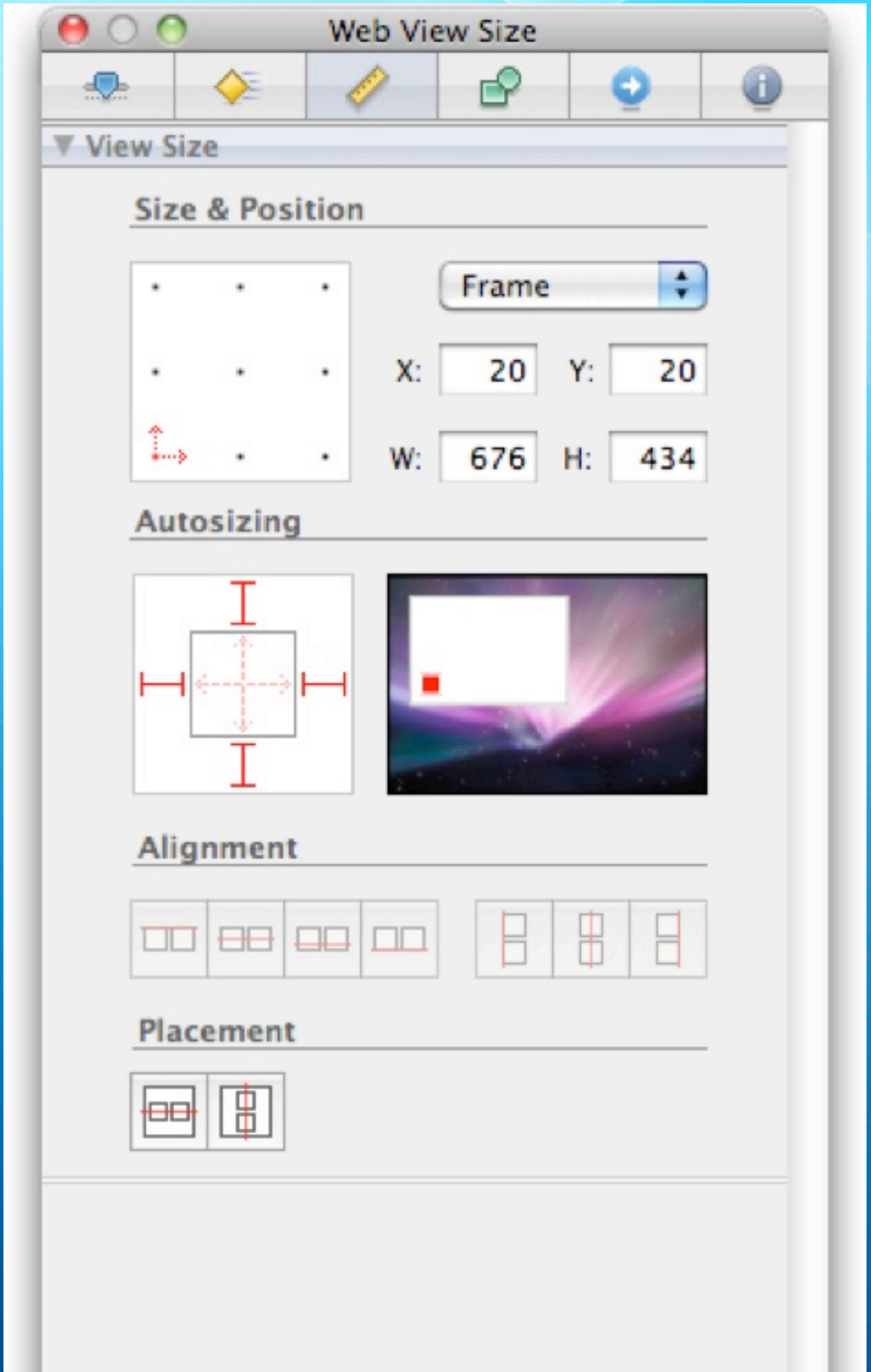


Everything is fine until you try to resize the window.
Invoke Editor ⇒ Simulate Document and try it out!



Whoops!

Fortunately, Interface Builder makes it easy to specify constraints on how widgets should behave when resize events occur



X: 20 Y: 20

W: 676 H: 434

With the Web View selected, select View ⇒ Utilities ⇒ Show Size Inspector

The Autosizing section provides the ability to specify resizing constraints

The outside brackets indicate whether a widget should try to remain relative to a particular side of the window during a resize event

The internal arrows (currently deselected) indicate whether a widget should grow horizontally or vertically during a resize event

For web view, we want all four brackets on (stay locked in place) and both arrows on (grow to fill all available space)

Select the window and be sure to set your window's minimum size too

Finish specifying autosizing behavior

- The two buttons need to be anchored on top and on the right hand side; they should not resize themselves during a window resize event
- The text field should be anchored on top, left and right; it should resize horizontally during a window resize event
- With these changes, your window should behave as expected during a resize event
  - Without writing a single line of code!

Step 5: Make Connections (I)

- We want to establish connections between the various widgets we've created
    - With Interface Builder, you do this via "Control Drags"
        - You hold down the control key, click on a widget and hold, and then finally drag to another widget
            - A menu will pop-up allowing you to specify a connection

Step 5: Make Connections (II)

- Establish the following connections
  - From Text Field to Web View: **takeStringURLFrom:**
  - From Back Button to Web View: **goBack:**
  - From Forward Button to Web View: **goForward:**
  - Note the colon symbol at the end of these names: ":"
    - these are Objective-C method names!
    - they are methods defined by Web View
    - they will be invoked when the source widget is triggered

Step 6: Link the Framework

- Save your .xib file
- Click on the project icon; select the WebBrowser target
- Select “Build Phases”
- Expand “Link Binary with Libraries”
- Click “+”
- Scroll down to WebKit.framework
- Select it and click Add

Step 7: Run the App; Browse the Web

- Type a URL and click Return
    - Watch the page load
- Load another page
- Click the Back button
- Click the Forward button

- A simple web browser without writing a single line of code

Discussion

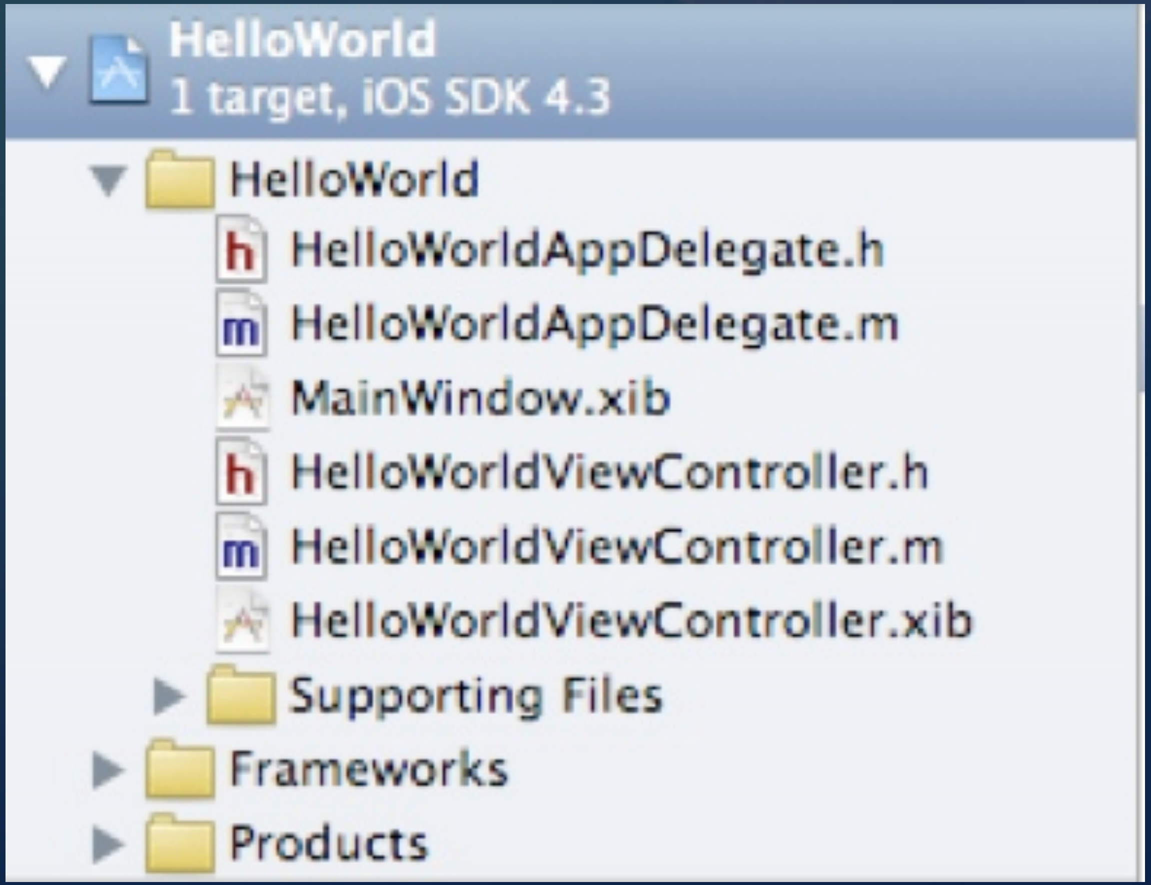
- This example is relevant to iOS programming because it shows all of the major mechanics of Interface Builder
  - We'll see a few more things Interface Builder can do when we link up code in XCode to widgets in Interface Builder
- Example demonstrates the power of objects; WebView is simply an instance of a very powerful object that makes use of Apple's open source WebKit framework
  - We can establish connections to it and invoke methods

Let's create a Hello World iOS App

- Select New Project from the File Menu
    - Click Application under iOS
    - Click View-based Application
    - Select iPhone
    - Click Choose and Name the App HelloWorld
    - Save the App
    - The project window opens

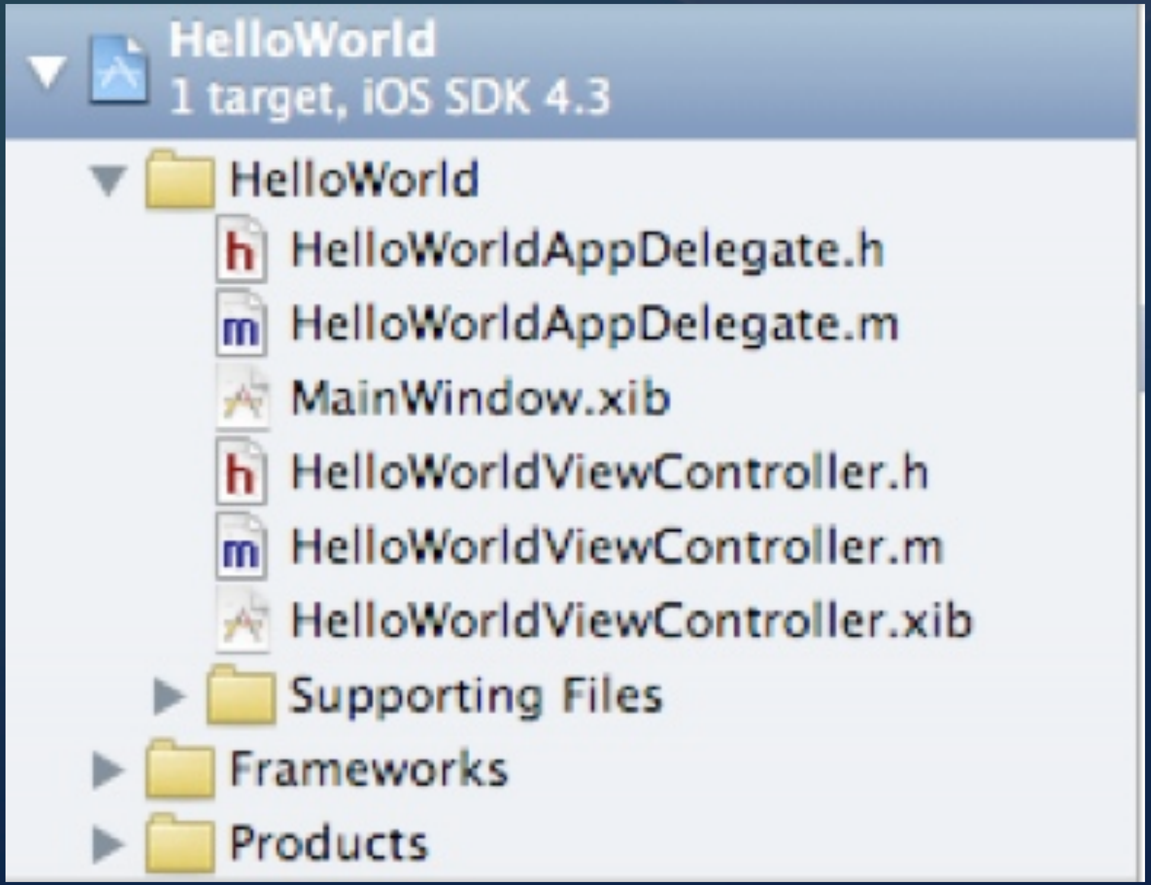
iOS Project Structure

- All iOS apps are instances of a class called UIApplication
  - You will never create an instance of that class
  - Instead, your application has an "AppDelegate" that is associated with UIApplication
    - The former will call your AppDelegate at various points in the application life cycle



iOS Project Structure

- Since this is a view-based application, an instance of a class called UIViewController was also created for you
- The application has a .xib file called MainWindow; the view controller has one too, it's called HelloWorldViewController.xib
- How is this all connected?



```
HelloWorld
1 target, iOS SDK 4.3
  HelloWorld
    HelloWorldAppDelegate.h
    HelloWorldAppDelegate.m
    MainWindow.xib
    HelloWorldViewController.h
    HelloWorldViewController.m
    HelloWorldViewController.xib
    Supporting Files
  Frameworks
  Products
```

iOS Project Structure

- The main.m file is straightforward
    - It creates an autorelease pool
    - Invokes UIApplicationMain()
        - This program reads in the .xib files, links up all the objects, loads the view controller, and starts the event loop, where it will remain until the application is told to shutdown
    - It then deallocates the pool and returns

iOS Project Structure

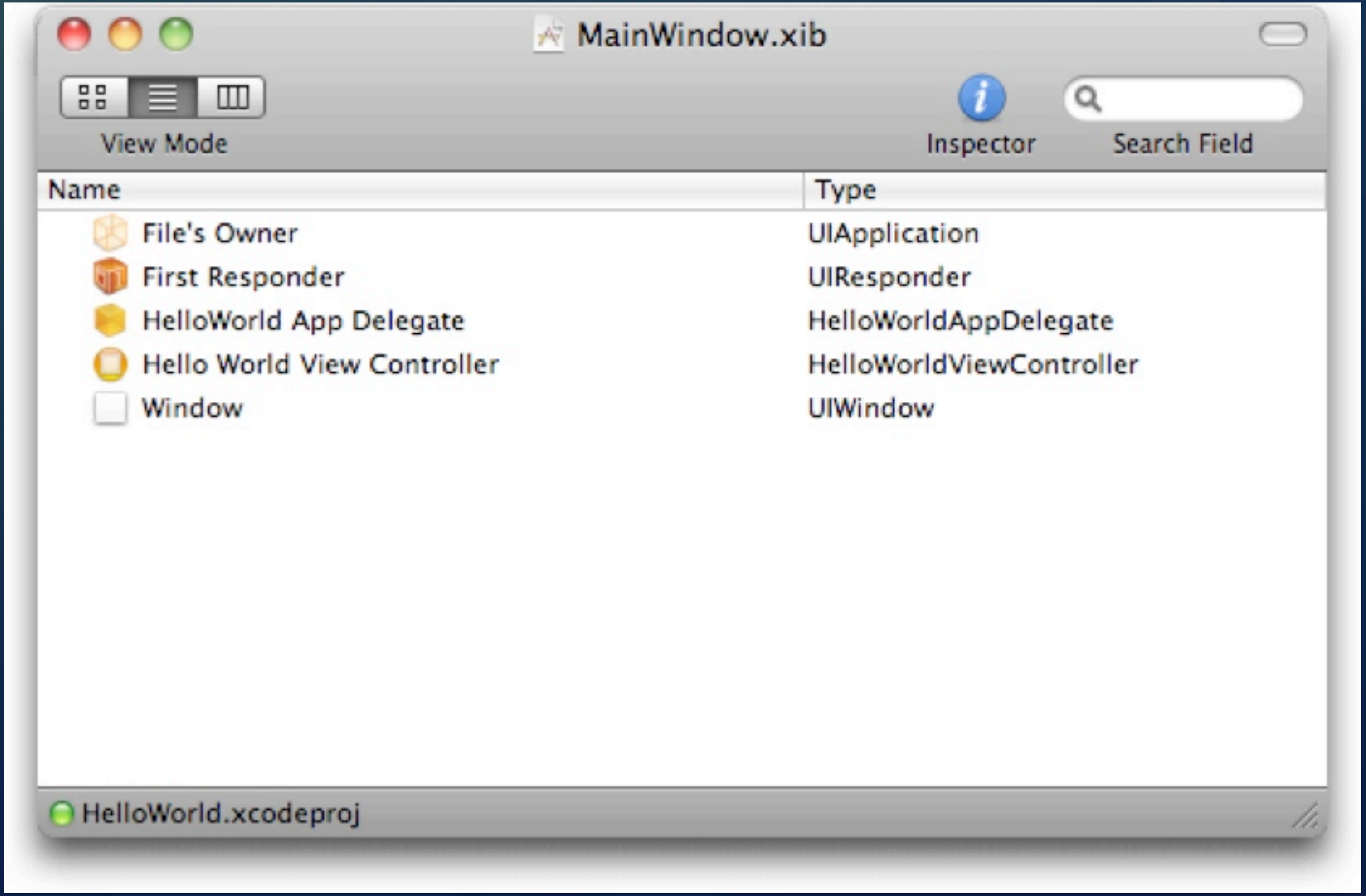
Note: Screenshot is from Interface Builder not from XCode 4; the same objects get created in both applications

- **Select MainWindow.xib**

Your app starts with three objects created by the MainWindow.xib file: Hello World App Delegate, Hello World View Controller and Window

No need to create them in code, they will be created automatically when this file is read by UIApplication

Cool!


MainWindow.xib

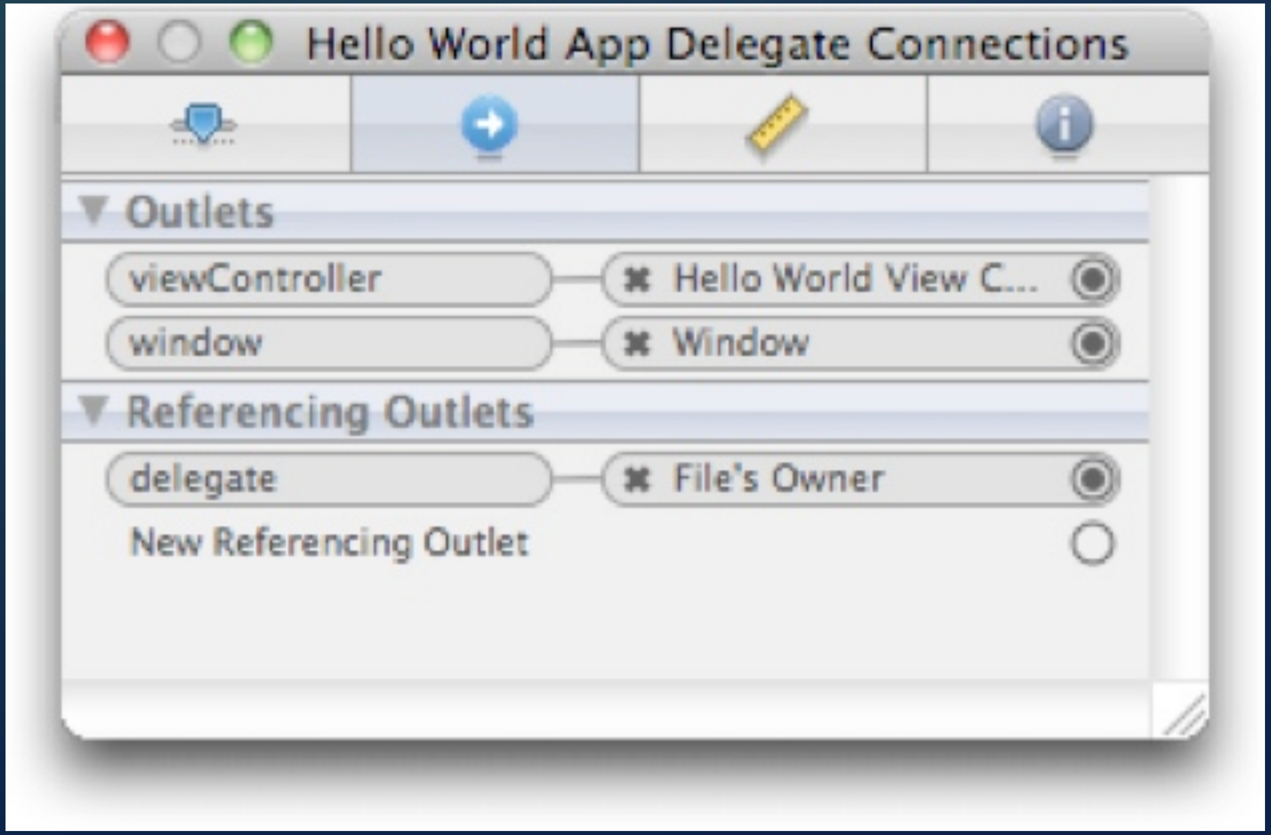
Name	Type
File's Owner	UIApplication
First Responder	UIResponder
HelloWorld App Delegate	HelloWorldAppDelegate
Hello World View Controller	HelloWorldViewController
Window	UIWindow

HelloWorld.xcodeproj

iOS Project Structure

- **Bring up the Connections Inspector**

This picture shows that not only does the .xib file create the Hello World App Delegate, it also connects it to the Hello World View Controller object via the "viewController" attribute and the Window via the "window" attribute


Hello World App Delegate Connections

Outlets

viewController	Hello World View C...
window	Window

Referencing Outlets

delegate	File's Owner
New Referencing Outlet	

Take a look at the code to see these attributes defined.

We also see that UIApplication (File's Owner) is wired to the App Delegate

iOS Project Structure

- Close the MainWindow.xib file without modifying it
- Now select the HelloWorldViewController.xib file
    - Here we see that this file creates an instance of UIView
- When MainWindow.xib creates the Hello World View Controller object, that object will, in turn, load its .xib file causing this view object to be created
    - The connection settings shows that this view will then be connected to the view controller (File's Owner)

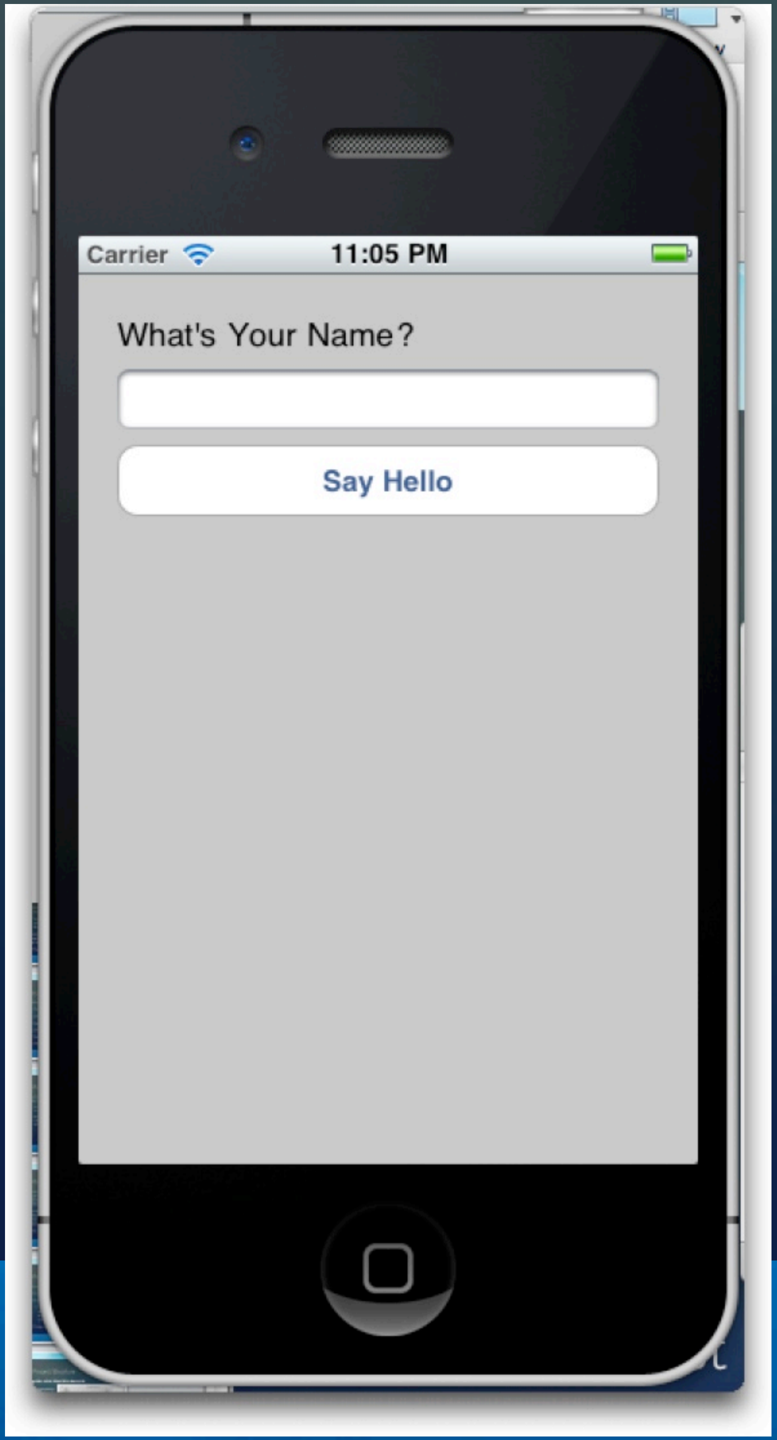
The Power of Interface Builder

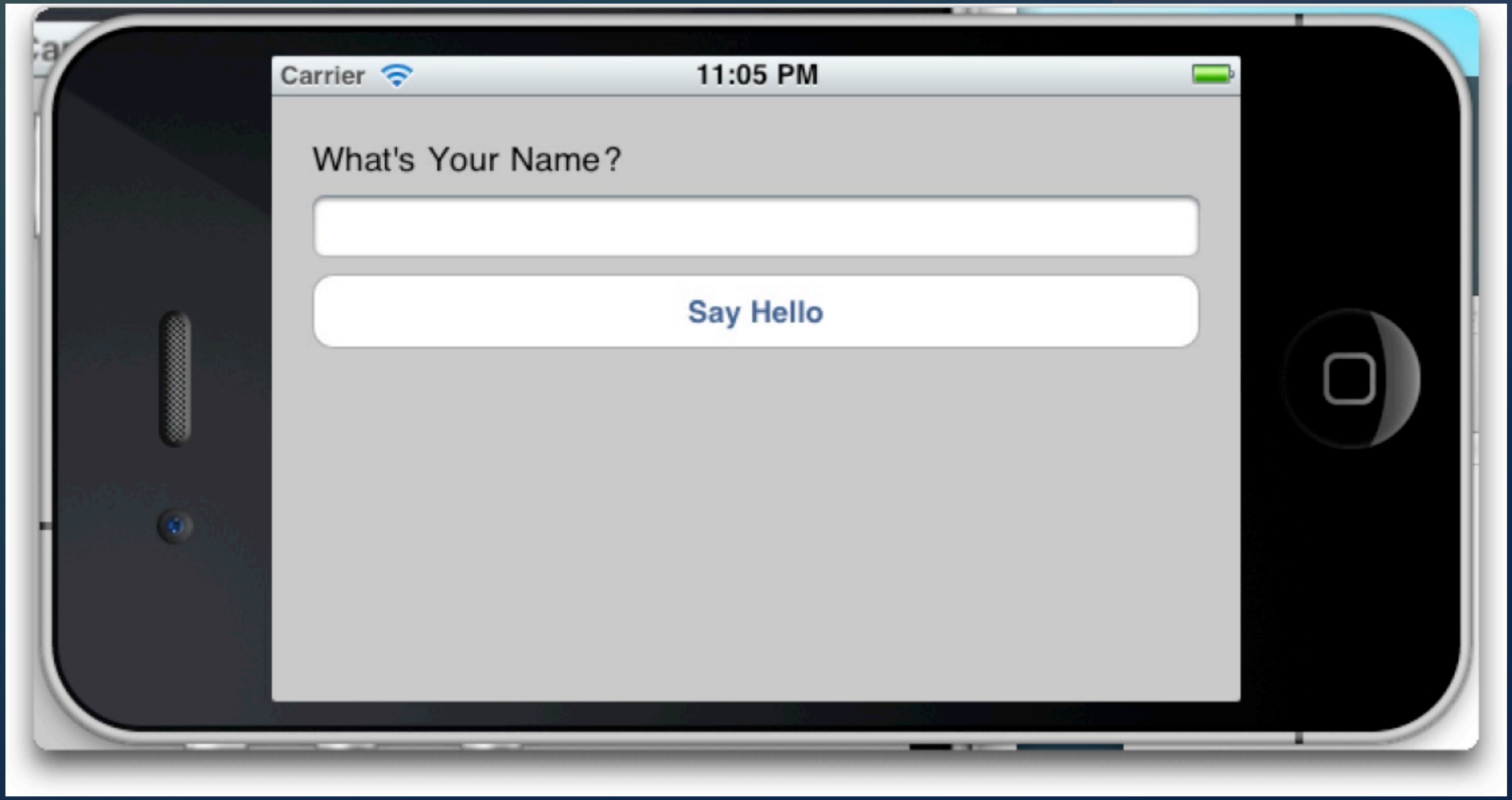
- All of this, once again, shows the power of Interface Builder
    - By creating objects in .xib files and specifying their connections, we completely eliminate the init code that would otherwise need to be written
    - UIApplication will load MainWindow.xib automatically creating the window, delegate, and view controller
        - The view controller will then load its .xib file, creating the view; the app will then display to the user
- Click Build and Run to confirm; Quit the iPhone Simulator

Add an interface

- Back in Interface Builder
  - Change view’s background color to white
  - Add a label that says “What’s your name?”
  - Add a text field
  - Add a button that says “Say Hello”
- Position them vertically and specify resize behavior
  - Test out the UI and test switching the phone from portrait to landscape until the UI does what you want

Our Simple UI





Also…

- Select the Button
    - Deselect the "Enabled" checkbox
        - We only want it enabled when the text field has some text
    - In State Config pop-up, select Disabled
        - then change Text Color to Light Gray Color

Configure the Code

- We now need to modify the code of our view controller
    - We need properties that point at the text field and button
        - because we are going to read the fields contents to get a name and we have to enable/disable the button
    - We also need a method that will get invoked when the button is pressed

Configuring the Code (I)

- In HelloWorldViewController.h add two properties
  - @property (nonatomic, retain) IBOutlet UITextField* name;
  - @property (nonatomic, retain) IBOutlet UIButton* hello;
- Add this method signature
  - - (IBAction) sayHello: (UIButton*) sender;
- IBOutlet and IBAction are clues to Interface Builder

Configuring the Code (II)

- **IBOutlet** tells Interface Builder that we will be linking a widget in the interface to this property
- **IBAction** tells Interface Builder that this method will be invoked by one of the widgets in the interface
    - In this case, it will be invoked when we click our button

Configuring the Code (III)

- In HelloWorldViewController.m, synthesize your properties
  - @synthesize name=_name;
  - @synthesize hello=_hello;
- These will be assigned automatically at run-time after we connect these properties to the appropriate widgets in Interface Builder (stay tuned)

Configuring the Code (IV)

- Add the following method body

```
- (IBAction) sayHello: (UIButton*) sender {
    NSString* greeting = [[NSString alloc]
                           initWithFormat:@"Hello, %@", self.name.text];
    UIAlertView*alert = [[UIAlertView alloc] initWithTitle:@"Hello
World!" message:greeting delegate:self cancelButtonTitle:@"Ok!"
otherButtonTitles:nil];
    [alert show];
    [greeting release];
    [alert release];
}
```

Configuring the Code (V)

- Set up auto-rotation: add this method

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
      return YES;
}
```

Connect Code and Interface

- In Interface Builder
  - Control drag from File's Owner to Text Field, select "name"
  - Control drag from File's Owner to Button, select "hello"
  - Control drag from button to File's Owner, select "sayHello:"
  - Save the file

Let’s deal with the text field

- We need to make sure that the button is enabled when the user has entered text
- We also need to make sure that the keyboard goes away when we are done editing
  - The keyboard shows up whenever there is a “first responder”: a widget that can respond to keystrokes
  - When we are done editing, we want the text field to stop being the first responder to make the text field go away

Handling the text field, part one

- Add to .h file
  - - (IBAction) doneEditing: (UITextField*) sender;
- Add to .m file
  - - (IBAction) doneEditing: (UITextField*) sender {
    - [self.name resignFirstResponder];
  - }
- Connect text field's Did End on Exit event to doneEditing:
  - Control-Click on the text field to see the events it can generate

Build and Test

- You're almost done
  - Build and Run the App
  - Test that you can enter text and make the keyboard go away by clicking return
    - Notice that keyboard won't go away if you click outside of text field (still have some work to do)
  - Quit the simulator

Handling the text field, part two

- To make the keyboard go away when we click outside of the text field, we need to create an invisible button that sits at the very bottom of the view hierarchy
    - If it gets clicked, it tells the text field to stop being the first responder

Handling the text field, part two

- Drag a push button out onto the view
    - make it as large as the view
    - Set it's type to custom
    - send it to the back of the view hierarchy
        - (Editor ⇒ Arrange ⇒ Send to Back)
- Create a new action method called dismissKeyboard: and connect this button to that action via its Touch Up Inside event. Give it same body as doneEditing:

Handling the text field, part three

- Now we need to handle enabling and disabling the text field
    - We will simply monitor the text field as it is being editing and check the length of the text field's string.
    - If it is greater than zero, then we'll enable the button
        - otherwise, we'll disable it
    - A really polished app would make sure that the entered name is not all spaces.

Add a new event handler

- Add a new event handler called checkLength:
- Give it the following code:
    - - (IBAction) checkLength: (UITextField*) sender {
        - [self.hello setEnabled:([self.name.text length] > 0)];
    - }
- Connect text field's Editing Changed event to checkLength:

Final Change

- Add
    - [self.name resignFirstResponder];
- to the first line of the sayHello: event handler
- We need to tell the keyboard to go away if it is showing when the button is pressed

Make it Universal

- A Universal app is one that runs on both iPhone and iPad
  - You essentially ask XCode to upgrade your iPhone program to a "universal" program
  - You add a new .xib file that specifies the UI for the iPad
  - You can then create a second view controller or hook up the existing view controller to the new .xib file
  - Regardless, the application will now autodetect which platform it is on and load the appropriate classes/ resources

Step One; There is no Step 2

- Select the HelloWorld Target
  - In the Devices pop-up, select Universal
- XCode creates a new .xib file that automatically hooks up to the existing HelloWorldViewController
  - Since we configured HelloWorldViewController.xib to autoresize its widgets, we are done!
- Select iPad Simulator 4.2 from XCode's pop-up and run the app

Wrapping Up

- Introduction to Interface Builder (XCode's XIB Editor)
  - Powerful, object-based GUI creation
- Basic introduction to iOS programming
  - iPhone application template
  - views and view controllers
  - hooking up code and widgets
  - making a universal application (Note: saw simplest case)

Coming Up Next

- Homework 4 Due Yesterday
- Lecture 14: Review for Midterm
- Lecture 15: Midterm
- Lecture 16: Review of Midterm