

Introduction To iOS

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 16 — 10/18/2012

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 46 times since then, with the most recent update released this past September with version 6.0
- iOS apps can be developed for iPhone, iPod Touch and iPad
 - iOS is used to run the Apple TV but apps are not currently supported for that platform

iOS 6.0

- I'll be covering iOS 6.0 which is the current “official version”
 - Version 6.0 was released on September 19, 2012
 - A major release with ~200 new features
- Most significant update was version 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 13
 - 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 third layer is the Media layer
 - Core Audio, OpenGL and OpenGL ES, Video and Image support, PDF, Quartz, Core Animation
- The fourth 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 will make use of services from all four layers

Introduction to Interface Builder (I)

- Interface Builder is a graphical editor inside of XCode for creating GUIs
 - It provides a drag and drop interface for constructing the graphical user interface of your mobile and desktop apps
 - It also lets you connect graphical widgets with properties defined in your source code
- Interface Builder stores the user interface widgets that it creates in .xib files; .xib stands for “XML Interface Builder”
 - A .xib file is a text file (but you will never edit it directly)
 - When a .xib file is copied over to a device, it is converted to a .nib file (NeXT Interface Builder) which is stored in binary format to save space

Introduction to Interface Builder (II)

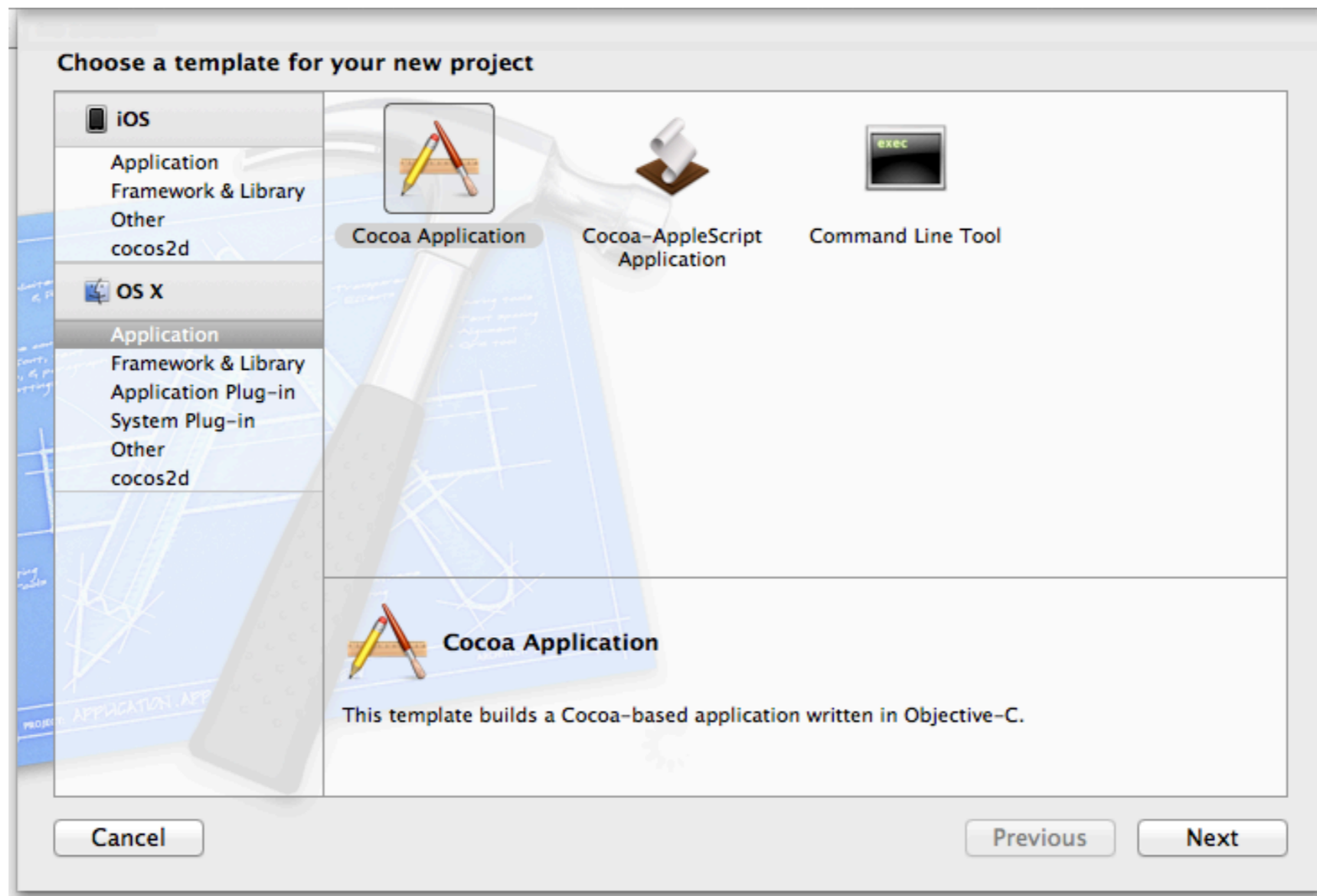
- The GUIs created by Interface Builder are
 - actual instances of the underlying UIKit classes
- When you save a .xib file, you “freeze dry” the objects and store them on the file system
 - When your app runs, the objects get reconstituted and linked to an object in your running app

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 (I)

- Launch XCode and create an OS X application (we'll get to iOS in a minute)
 - Select Cocoa Application rather than Command Line Tool



Step One: Create Project (II)

- Product Name: WebBrowser; Class Prefix: CU; Select “Use ARC”

Choose options for your new project:

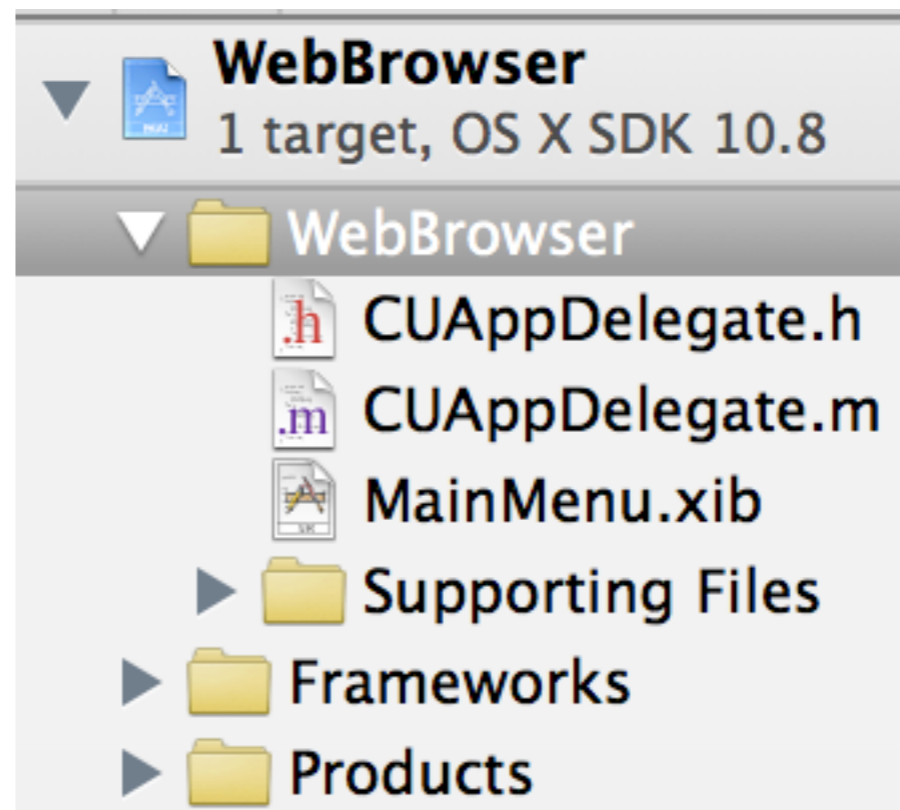
Product Name	WebBrowser
Organization Name	University of Colorado
Company Identifier	edu.colorado
Bundle Identifier	edu.colorado.WebBrowser
Class Prefix	CU
App Store Category	None
	<input type="checkbox"/> Create Document-Based Application
Document Extension	mydoc
	<input type="checkbox"/> Use Core Data
	<input checked="" type="checkbox"/> Use Automatic Reference Counting
	<input type="checkbox"/> Include Unit Tests
	<input type="checkbox"/> Include Spotlight Importer

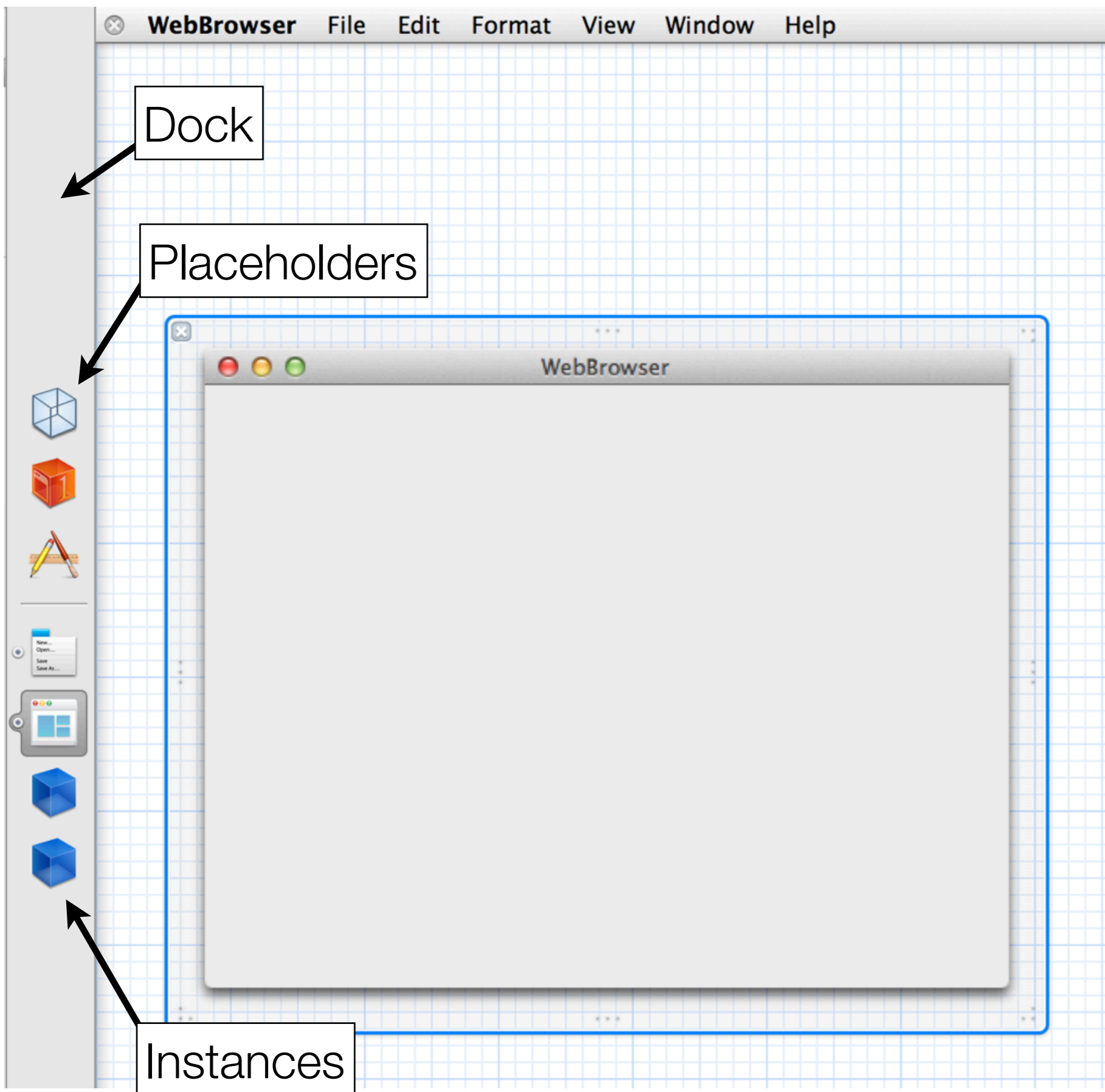
Cancel Previous Next

Save the resulting project wherever you want

Step Two: :Launch Interface Builder

- XCode creates a default set of files
 - We're going to ignore AppDelegate for now
 - Instead, we're going to select MainMenu.xib and launch Interface Builder

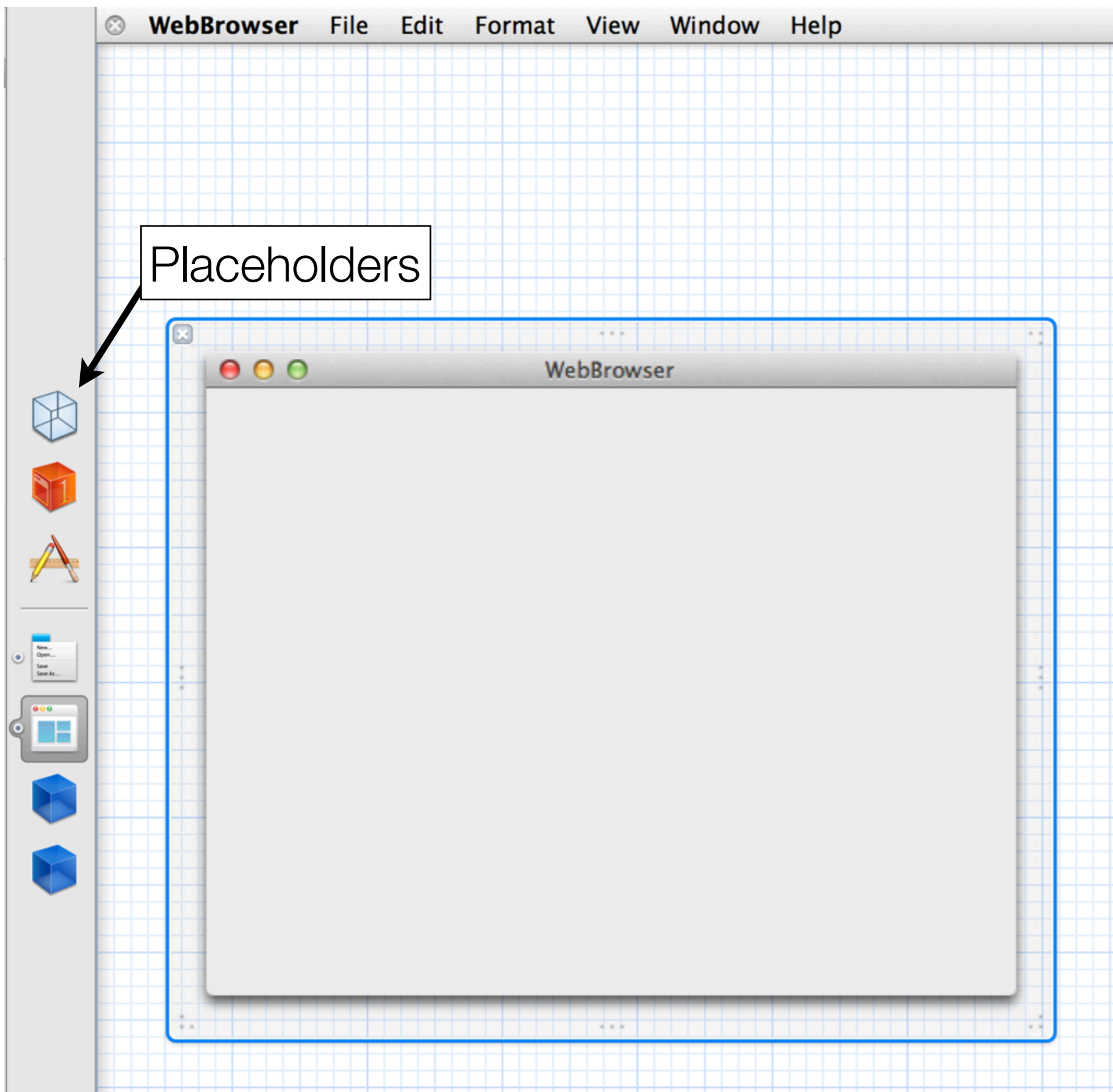




Here, we see a portion of Interface Builder's UI

On the left hand side is the Dock, showing **placeholders** and **instances**

On the right hand side is a canvas that can be used to layout the graphical user interface of your application

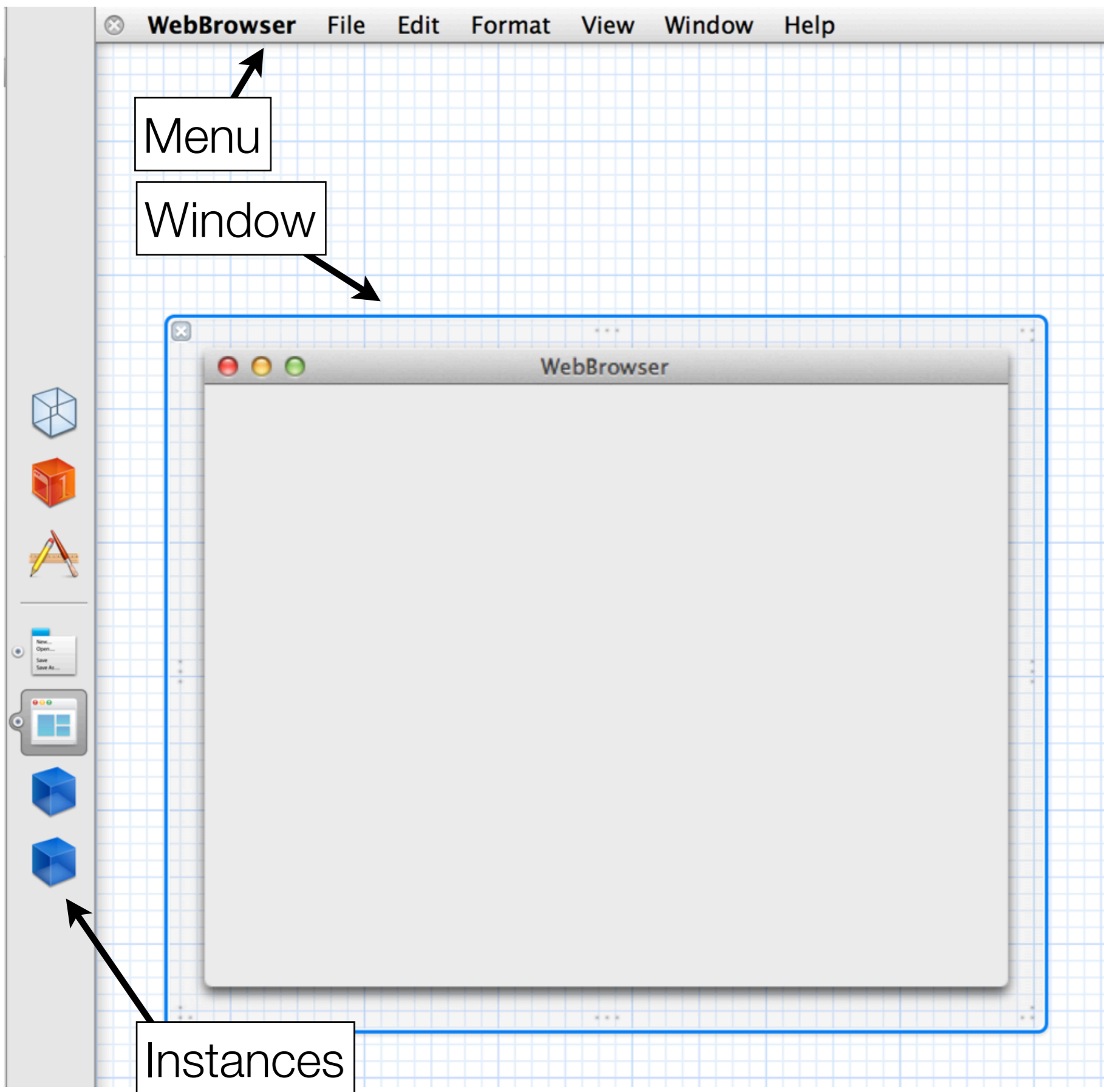


Placeholders are objects that exist “outside” of the .xib file

They exist before the .xib file is loaded and get connected at run-time

The icons for the placeholders represent the “File’s Owner”, the “First Responder”, and the “Application”

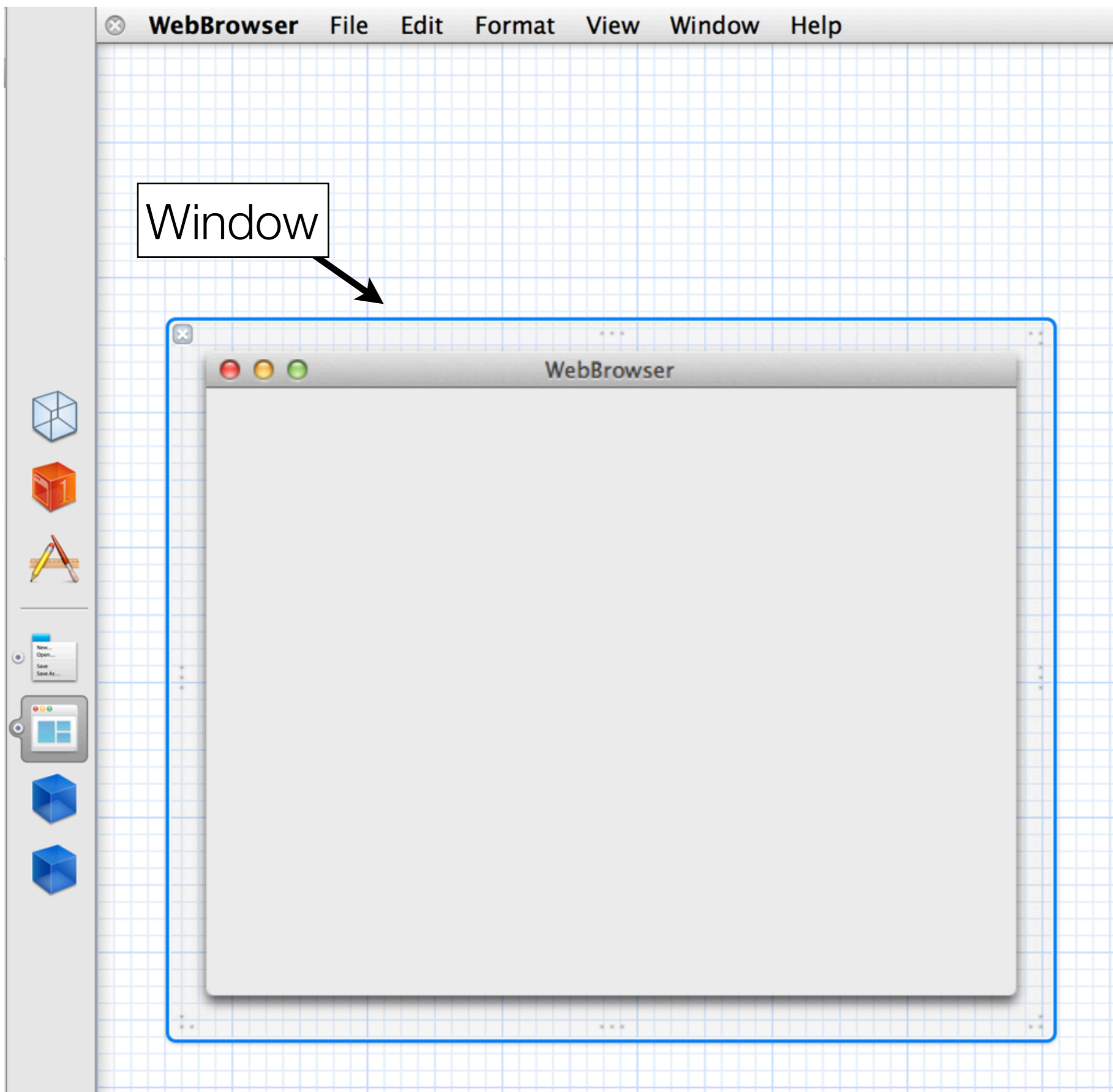
We’ll learn more about these roles later.



Instances are objects that exist inside of the .xib file

These objects typically are widgets but can also include controller objects and utility objects

By default, an OS application starts life with four objects—two of which are visible—the application's **menu** and the application's **window**; The other two objects are unimportant for now

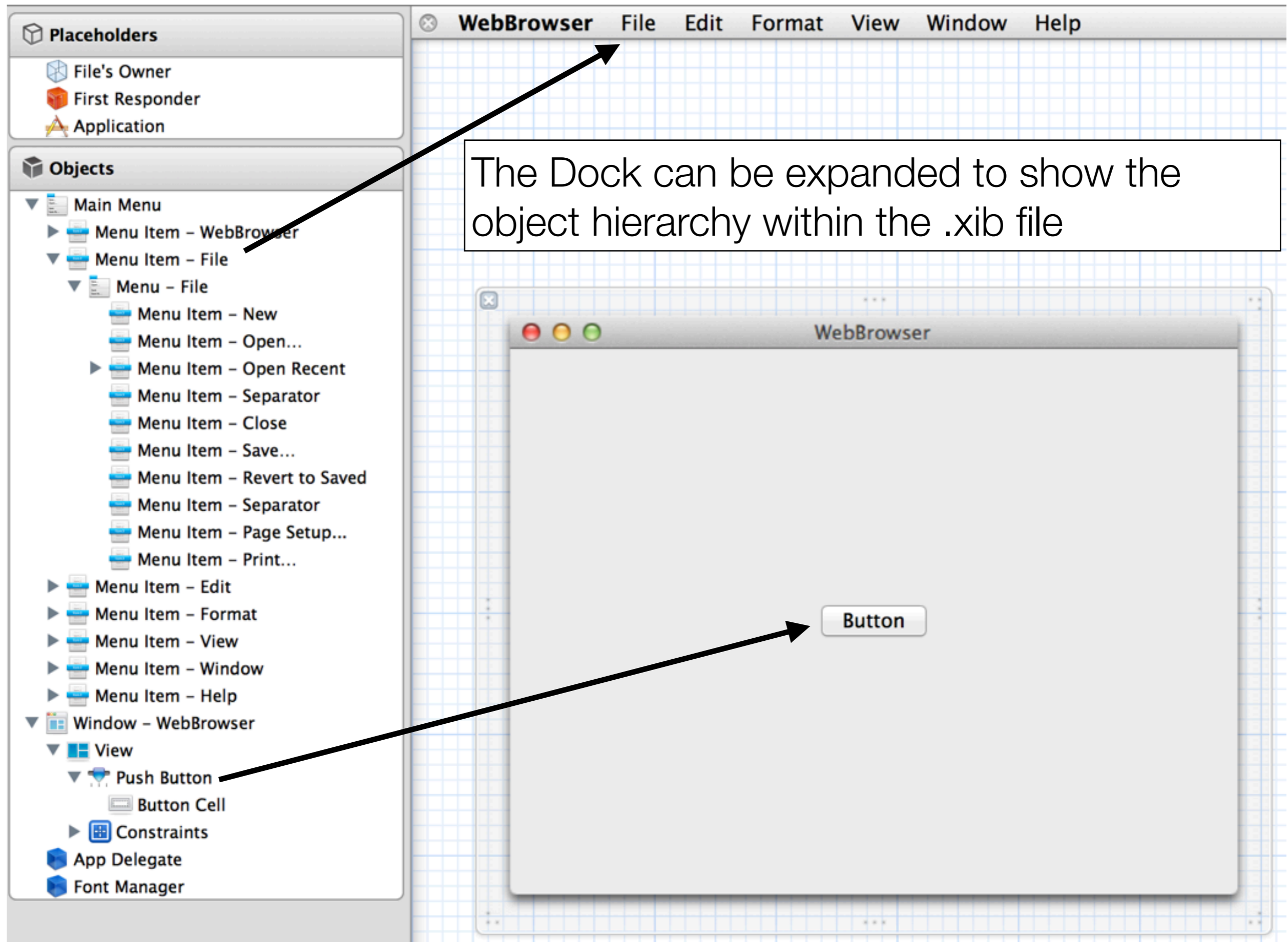


Just to emphasize, Interface Builder operates on “live” objects.

The window to the left is an actual instance of `NSWindow`

It is **not** a “simulation” of `NSWindow`.

You can manipulate this window via drag and drop; behind the scenes, Interface Builder is calling methods and setting properties to match your edits!



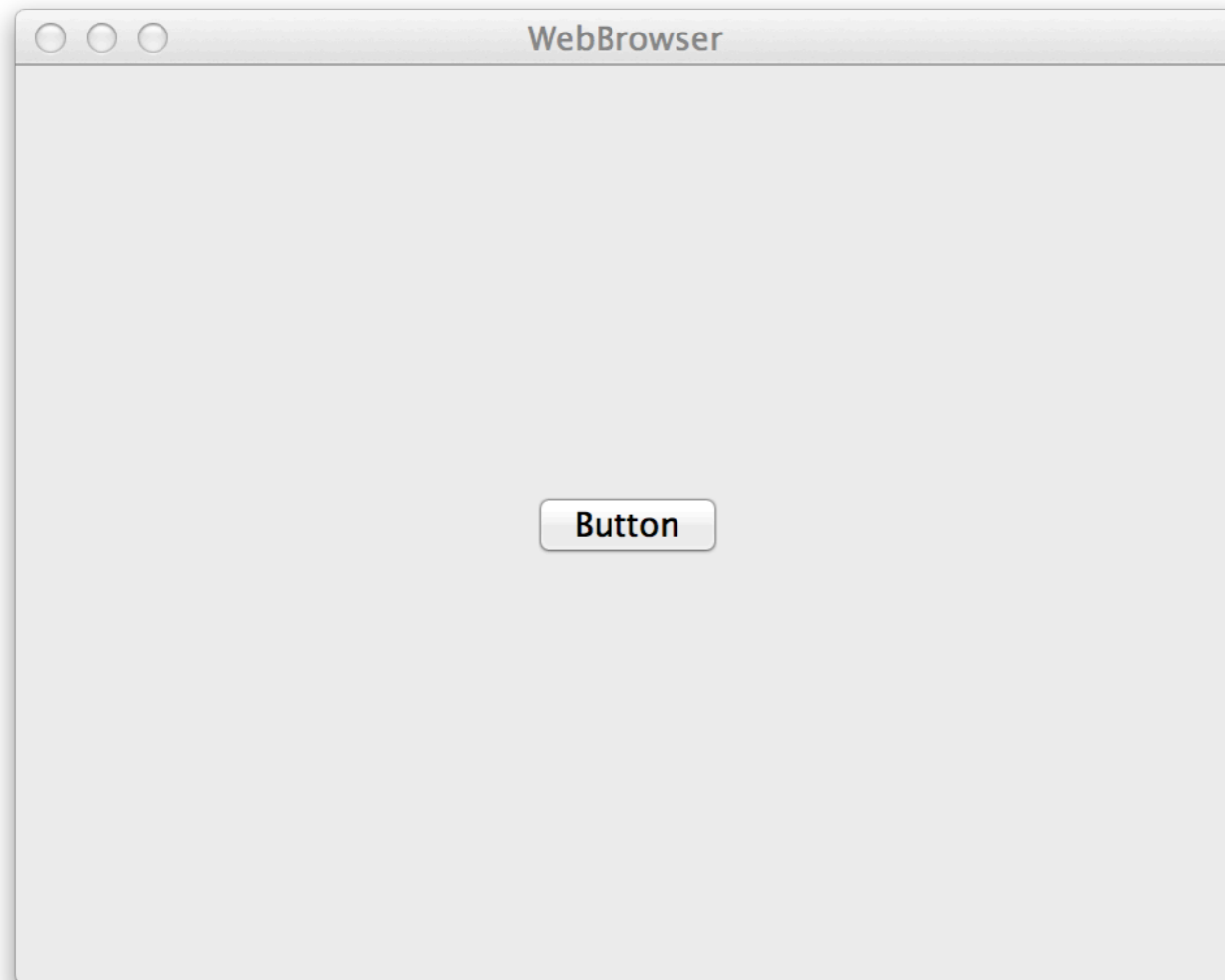
Object Connections (I)

- 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
- And by “connect”, I mean “establish a link between the two objects such that they can send messages to each other”
 - In practice, Interface Builder links are unidirectional
 - If you connect your CUAppDelegate object to your NSButton, then CUAppDelegate can send messages to your NSButton
 - If you want, the NSButton to talk to CUAppDelegate, you create a second connection going the opposite direction

Object Connections (II)

- Say a button should call a controller when clicked
 - You drag from the button
 - to the controller's icon in the dock
 - and then select the method the button should invoke
- These graphical actions are equivalent to the following code invocations
 - `[button setTarget: controller]`
 - `[button setAction: @selector(handleClick:)]`
- Note: `@selector()` is a way to reference a particular method at compile time
 - At run-time, `@selector()` will map "handleClick:" to the actual handleClick: method of the controller object

Step 3: Launch Application



When you run the application, you will see the UI you created in Interface Builder.

Exciting, isn't it?

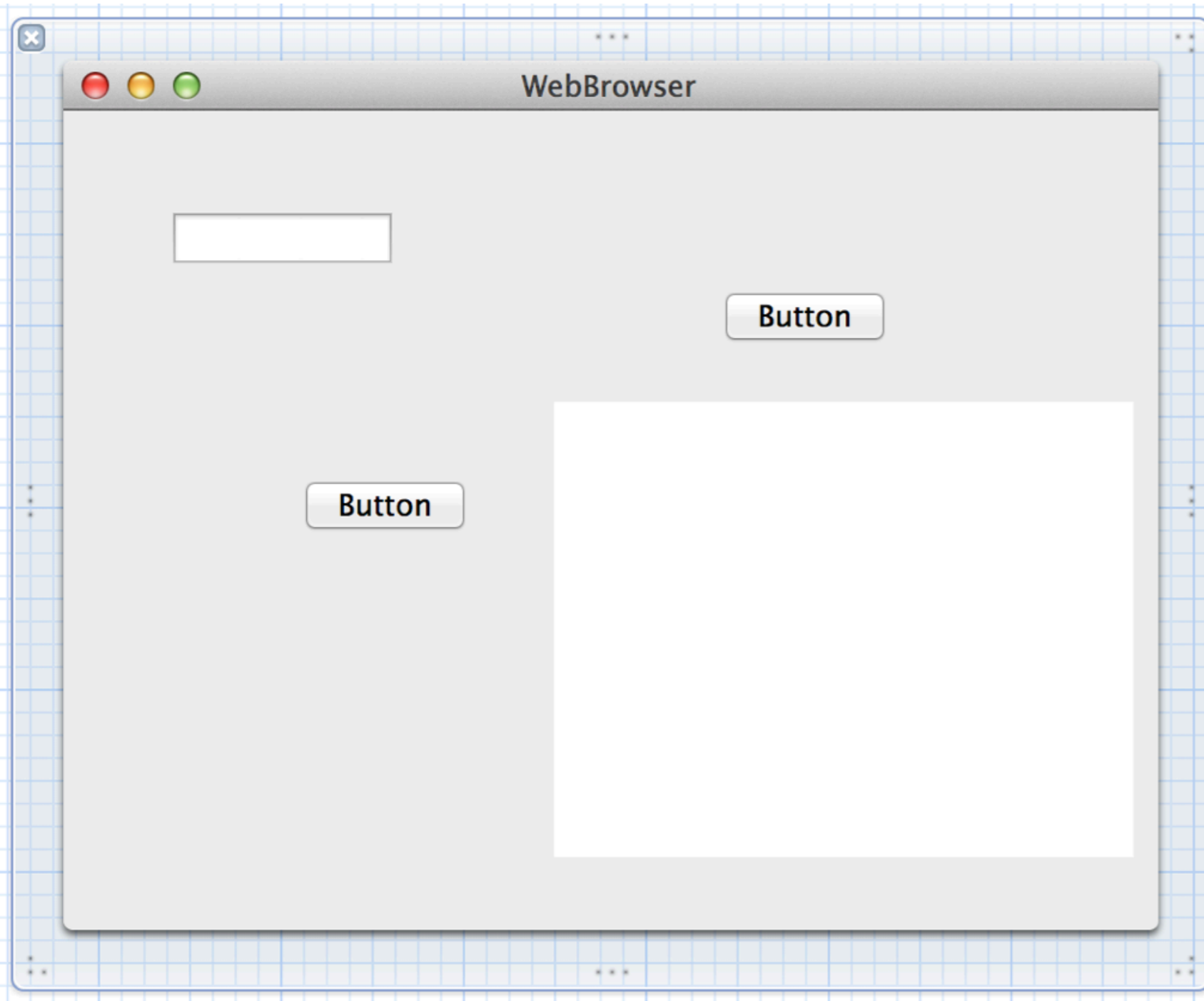
Quit the application and make changes to the window. Save the document, and run the program again.

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

Step 4: 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 4



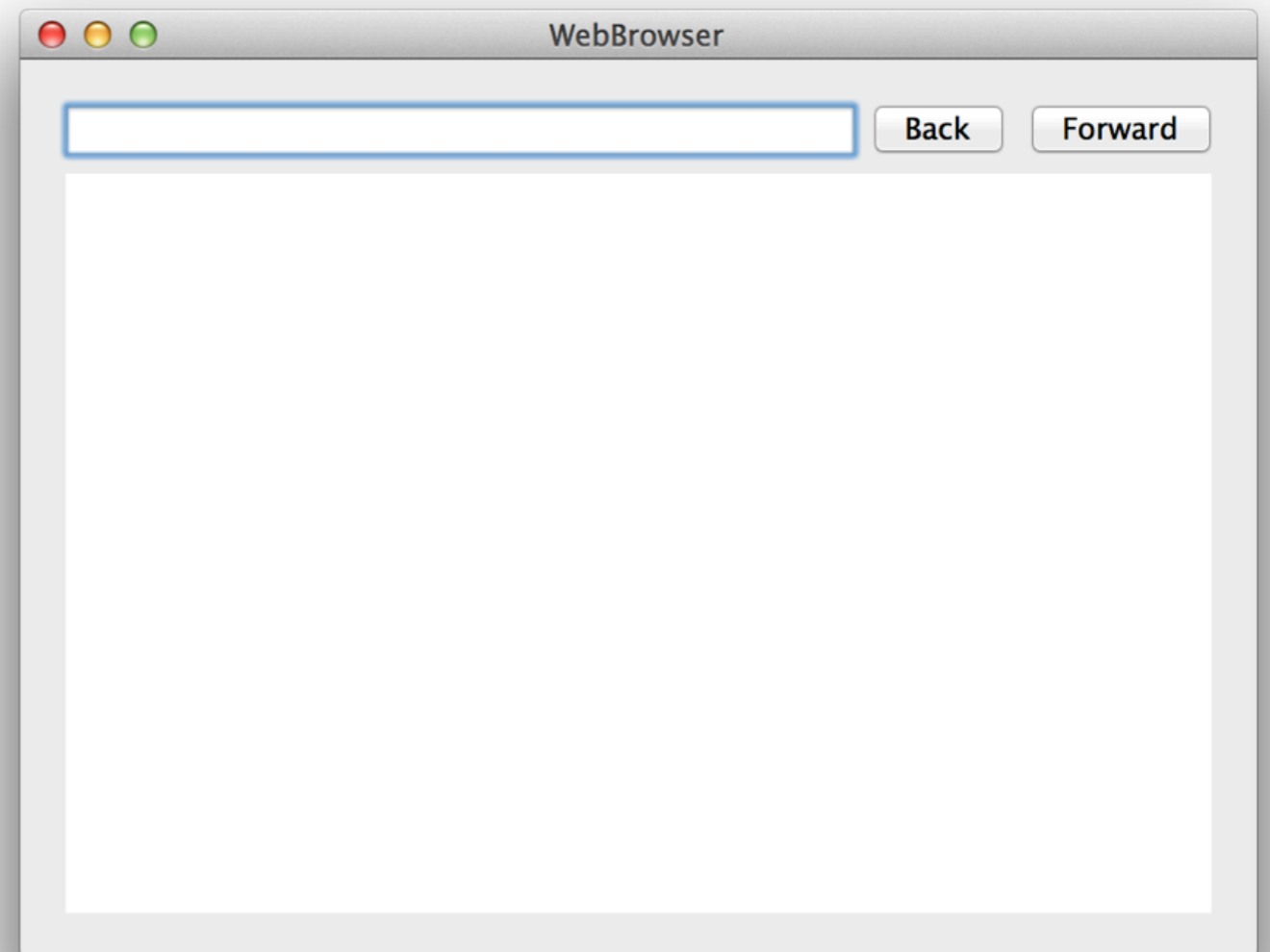
Our window now has four widgets but they are not yet placed where we want them

Step 5: 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
- Expand the Web view so that it now fills the rest of the window, following the guides to leave the appropriate amount of space

Step 5: Layout Widgets (II)

- Your window should look like the image to the right
- You can try your UI out by invoking the menu command
 - “Editor ⇒ Simulate Document”
- What’s cool is that the widgets resize correctly if you resize the window
 - All without writing a single line of code!
 - This is handled by Apple’s new constraint-based “auto-layout” system



Step 6: Make Connections (I)

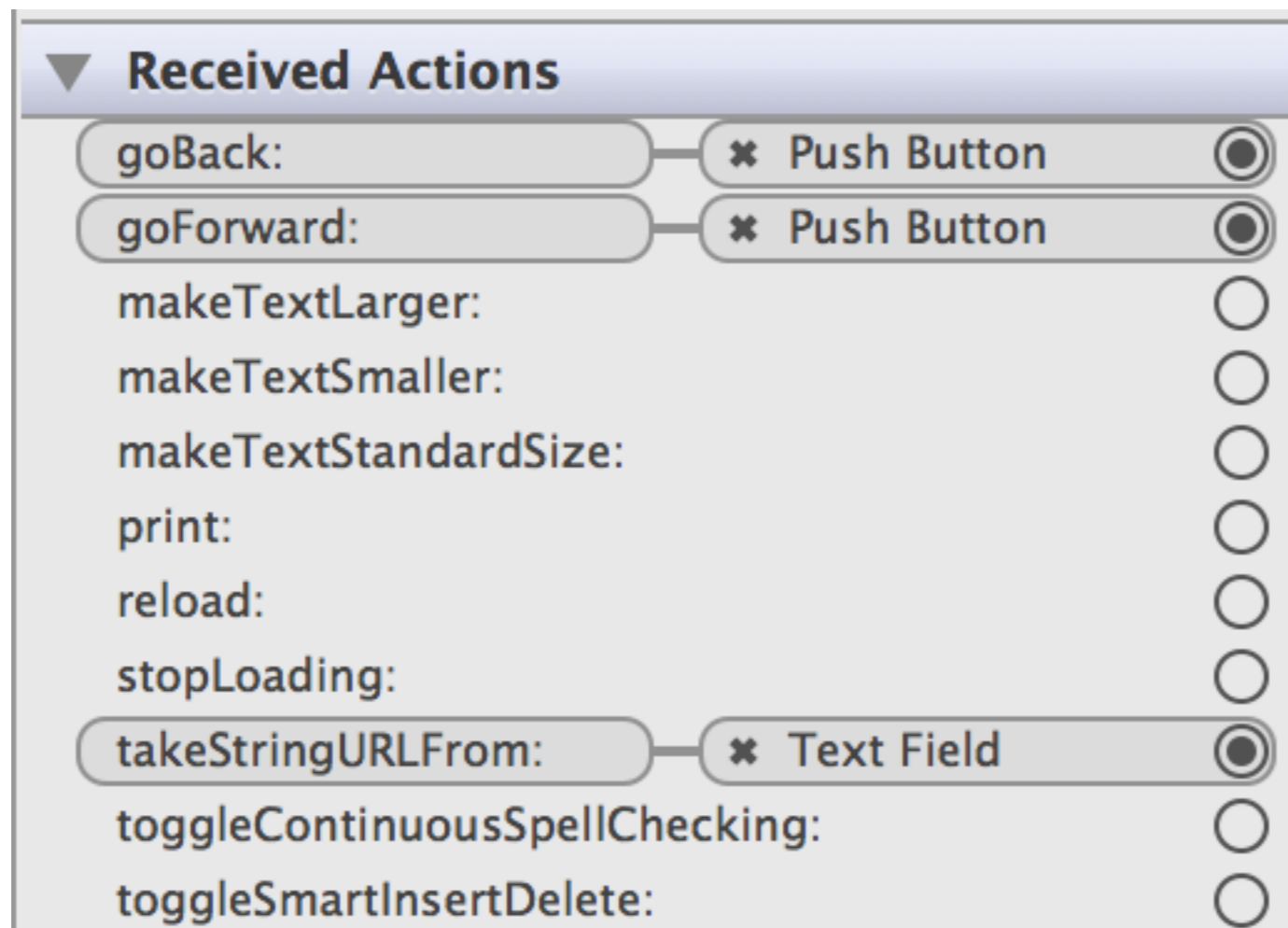
- We want to establish connections between the various widgets we've created
 - With Interface Builder, you do this via “Control Drag”
 - You hold down the control key, click on a widget and hold, and then finally drag to another widget
 - A menu will pop-up allowing you to specify a connection

Step 6: 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 the Web View class
 - they will be invoked when the source widget is triggered

Step 6: Make Connections (III)

- Check your connections by selecting the Web View and then bring up the Connections inspector (⌘⌘6)



This inspector will show that you have buttons wired up to the goBack: and goForward: methods and a text field wired up to the takeStringURLFrom: method

If you click on one of the buttons, you'll see the connection from the opposite end. There it appears under a section called Sent Actions

Step 7: 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
-
- This step ensures that the framework that implements the Web View object is available at run-time. If you skip this step, your app will compile but not run.

Step 8: Run the App; Browse the Web

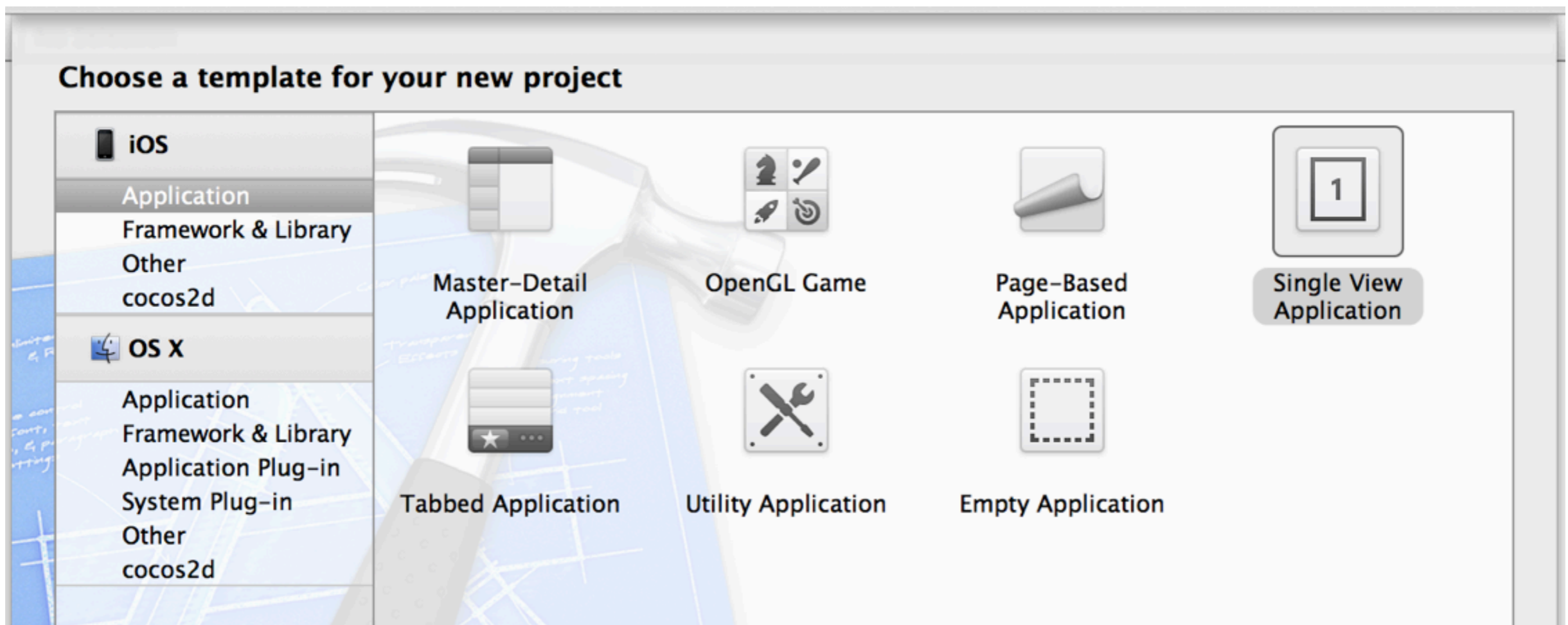
- Type a URL and click Return
 - Watch the page load
- Load another page
- Click the Back button
- Click the Forward button
- A simple web browser without writing a single line of code

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
- This 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 in Interface Builder and invoke methods on it by triggering other widgets
 - "Clicking" in the case of the buttons and "hitting enter" in the case of the text field

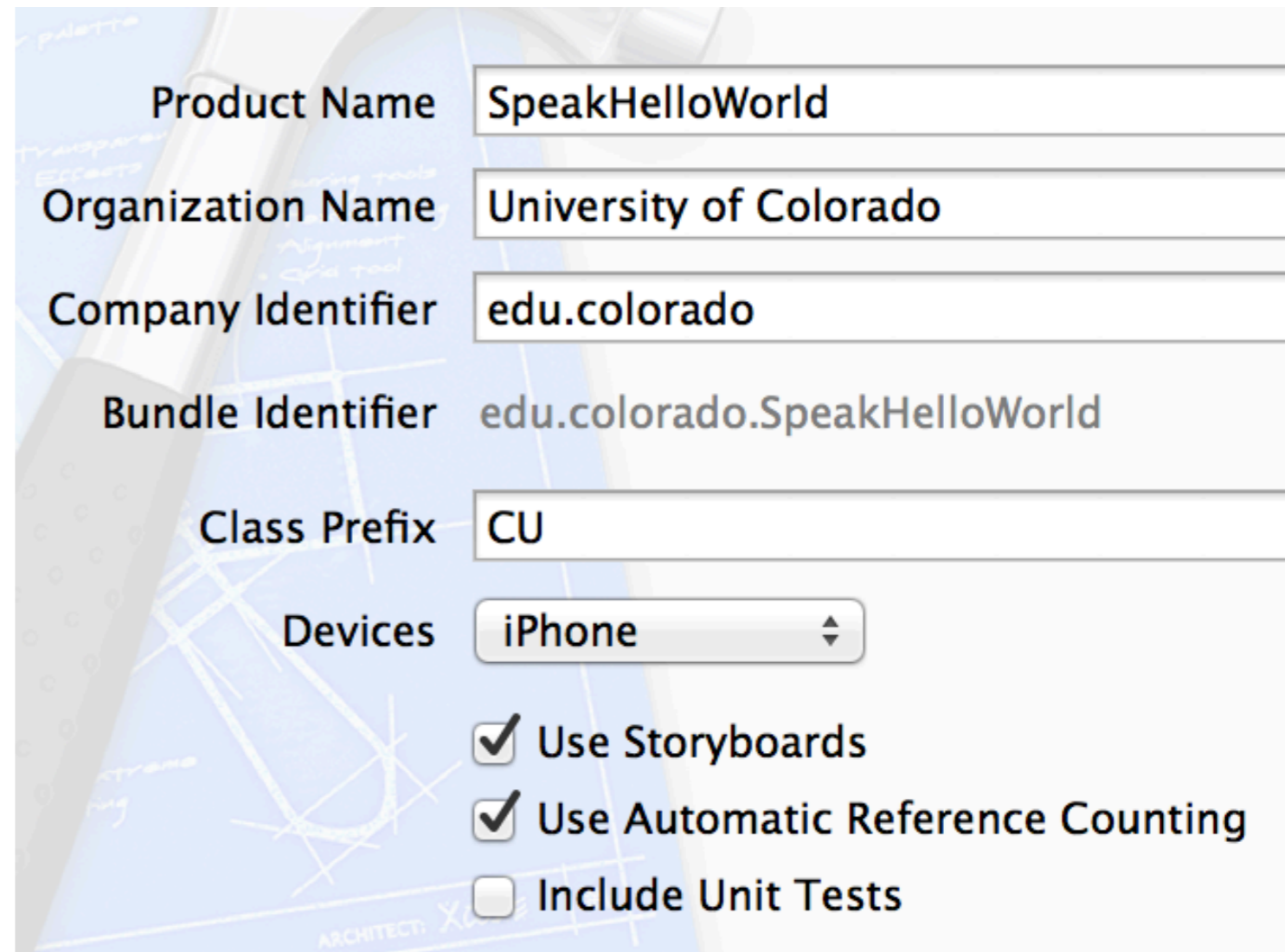
SpeakHelloWorld iOS App

- Let's create a “hello world” iOS app that can greet us (literally) in a variety of ways
 - Select New Project from the File Menu of XCode
 - Under the iOS section, select Single View Application



Configure Project

- Product Name: SpeakHelloWorld
- Class Prefix: CU
- Device: iPhone
- Click “Use Storyboards”
- Click “Use Automatic Reference Counting”
- Click Next and then save your project wherever you want

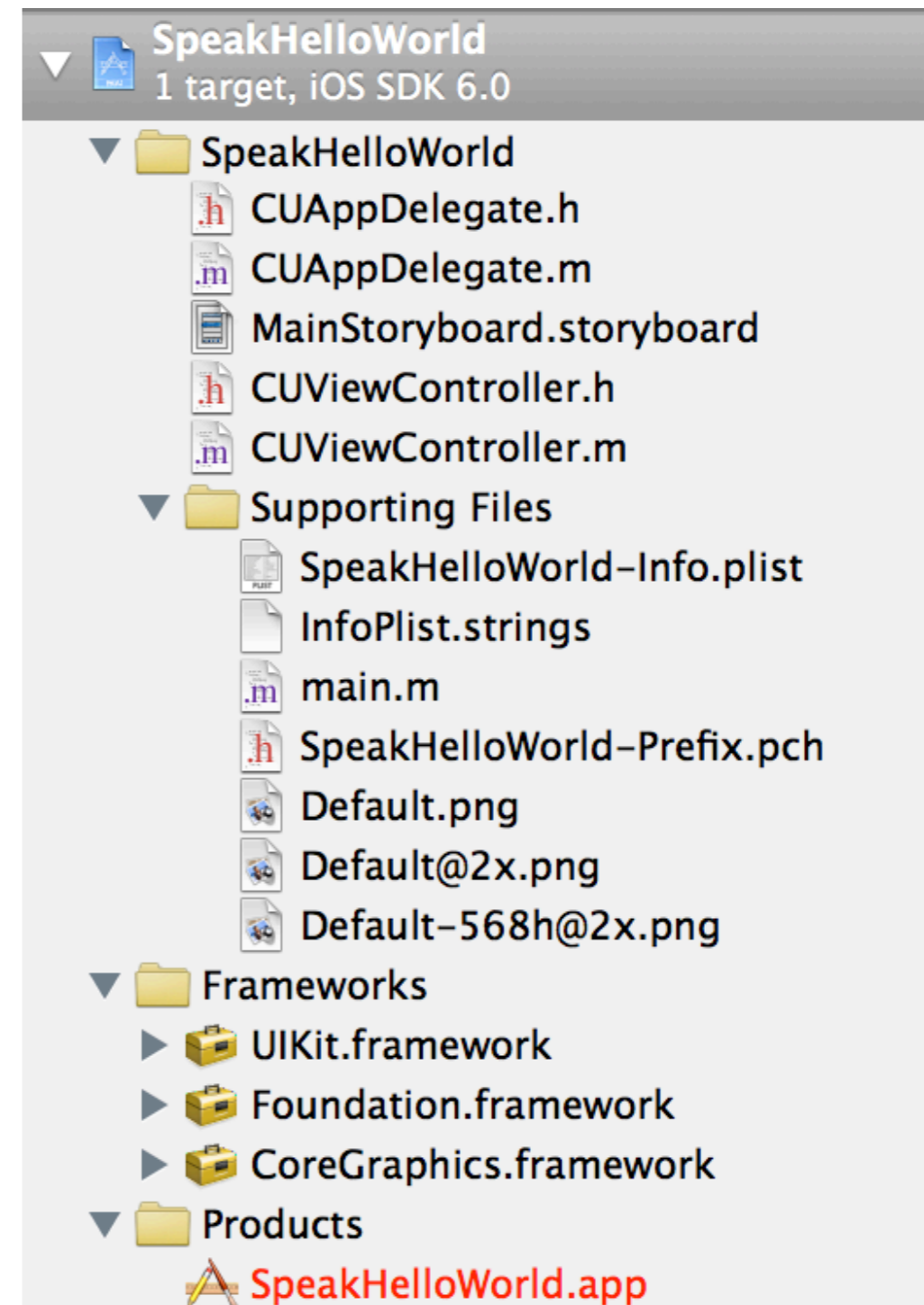


The screenshot shows the 'Configure Project' dialog in Xcode. The background is a faint architectural drawing. The configuration fields are as follows:

Product Name	SpeakHelloWorld
Organization Name	University of Colorado
Company Identifier	edu.colorado
Bundle Identifier	edu.colorado.SpeakHelloWorld
Class Prefix	CU
Devices	iPhone
	<input checked="" type="checkbox"/> Use Storyboards
	<input checked="" type="checkbox"/> Use Automatic Reference Counting
	<input type="checkbox"/> Include Unit Tests

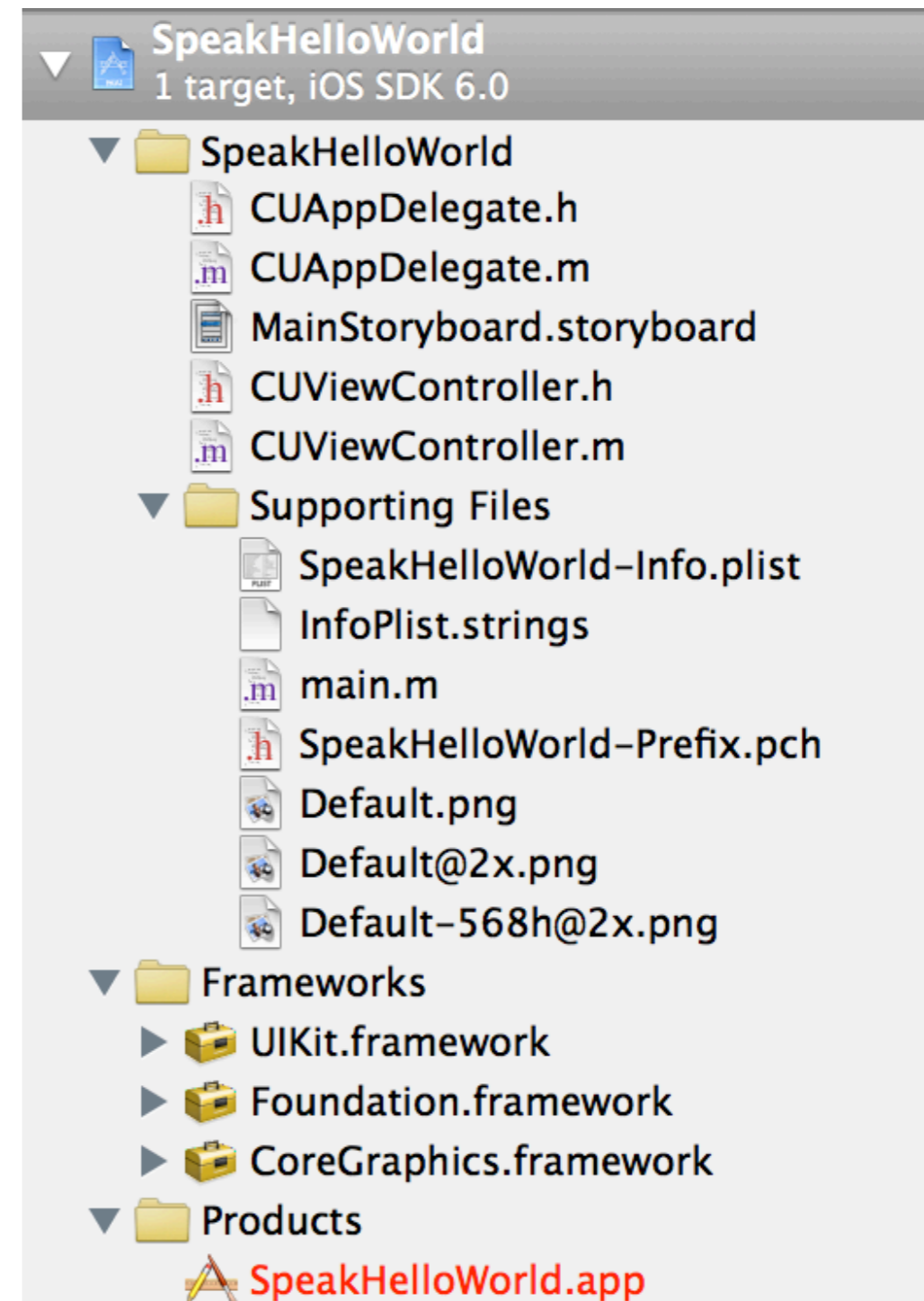
iOS Project Structure (I)

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



iOS Project Structure (II)

- Our project has the following set of components
 - UIApplication (created in main.m)
 - AppDelegate (discussed above)
 - MainStoryboard.storyboard
 - despite the new extension, you can think of this as a .xib file on steroids
 - UIViewController (the view controller is going to manage the single view of this application)
- How is this all connected?



iOS Project Structure (III)

- The main.m file is straightforward
 - It creates an autorelease pool (used by ARC to manage memory)
 - It invokes UIApplicationMain()
 - This method reads in the storyboard, creates the specified user interface, loads it into the app's window and notifies the AppDelegate that the application has launched. It then starts the event loop
 - It will remain in the event loop forever
 - The reason?
 - When a user “quits” an iOS app by pressing the Home button, the current app's process is simply moved to the background
 - it can then be brought back to the foreground at a later time or (eventually) the process gets destroyed while in a suspended state

iOS Project Structure (IV)

- The definition of our application delegate, CAppDelegate, looks like this

```
8
9  #import <UIKit/UIKit.h>
10
11 @interface CAppDelegate : UIResponder <UIApplicationDelegate>
12
13 @property (strong, nonatomic) UIWindow *window;
14
15 @end
16
```

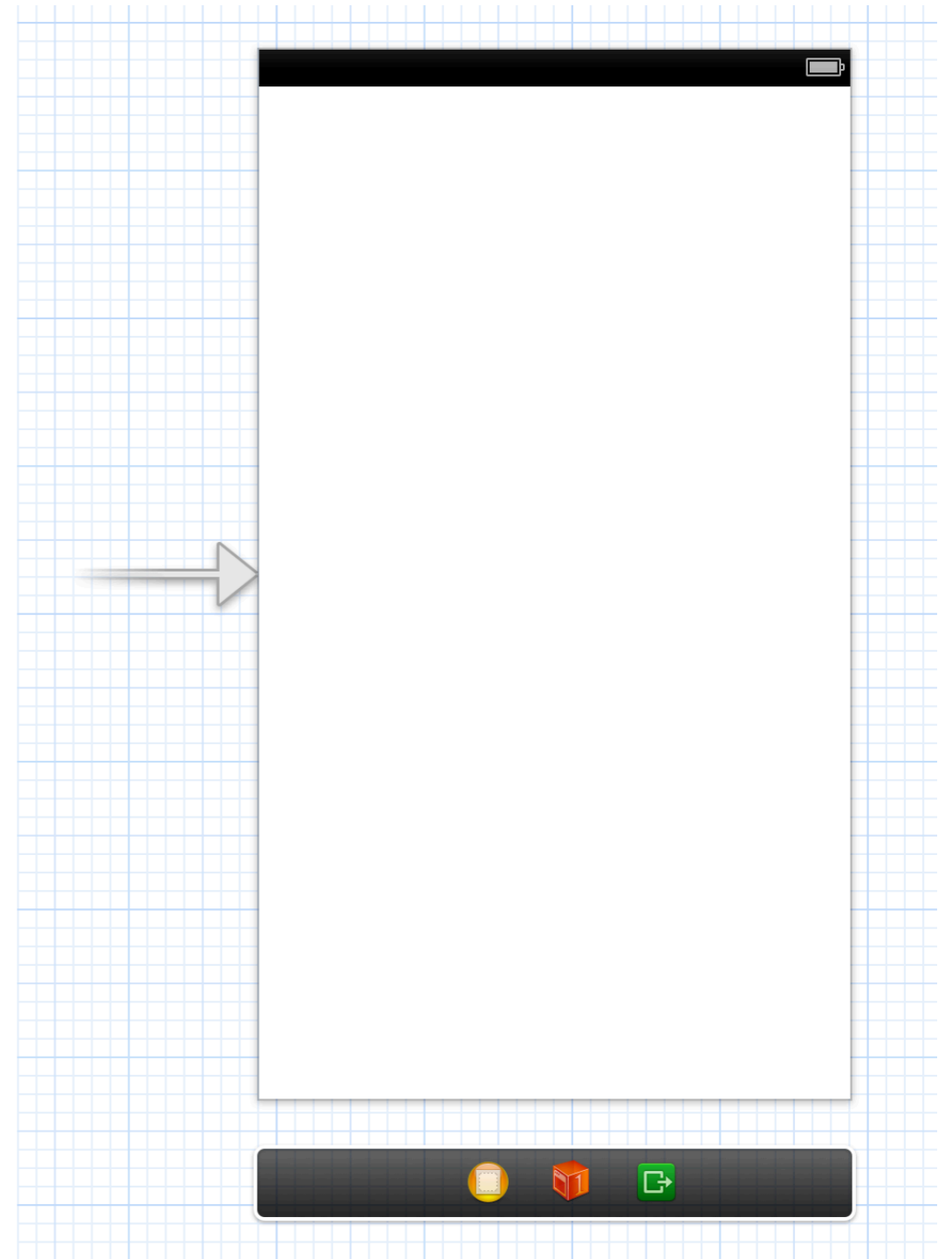
- It imports UIKit; it is a subclass of UIResponder; and it implements the UIApplicationDelegate protocol (think Java interface)
 - it also has a property that gives it access to the main window
 - An iOS app has only a single window that covers the entire display

iOS Project Structure (V)

- All of the methods for UIApplicationDelegate are optional
 - The most commonly implemented method is
 - - application:didFinishLaunchingWithOptions:
 - This method is called just before the UI appears on screen
 - You can perform various initialization tasks here
- For this simple app, all of our initialization will be handled by the storyboard and our view controller
 - As a result, you can ignore the code that appears in CUAppDelegate.m for this application; indeed, you can delete everything that appears between **@implementation CUAppDelegate** and **@end** in that file

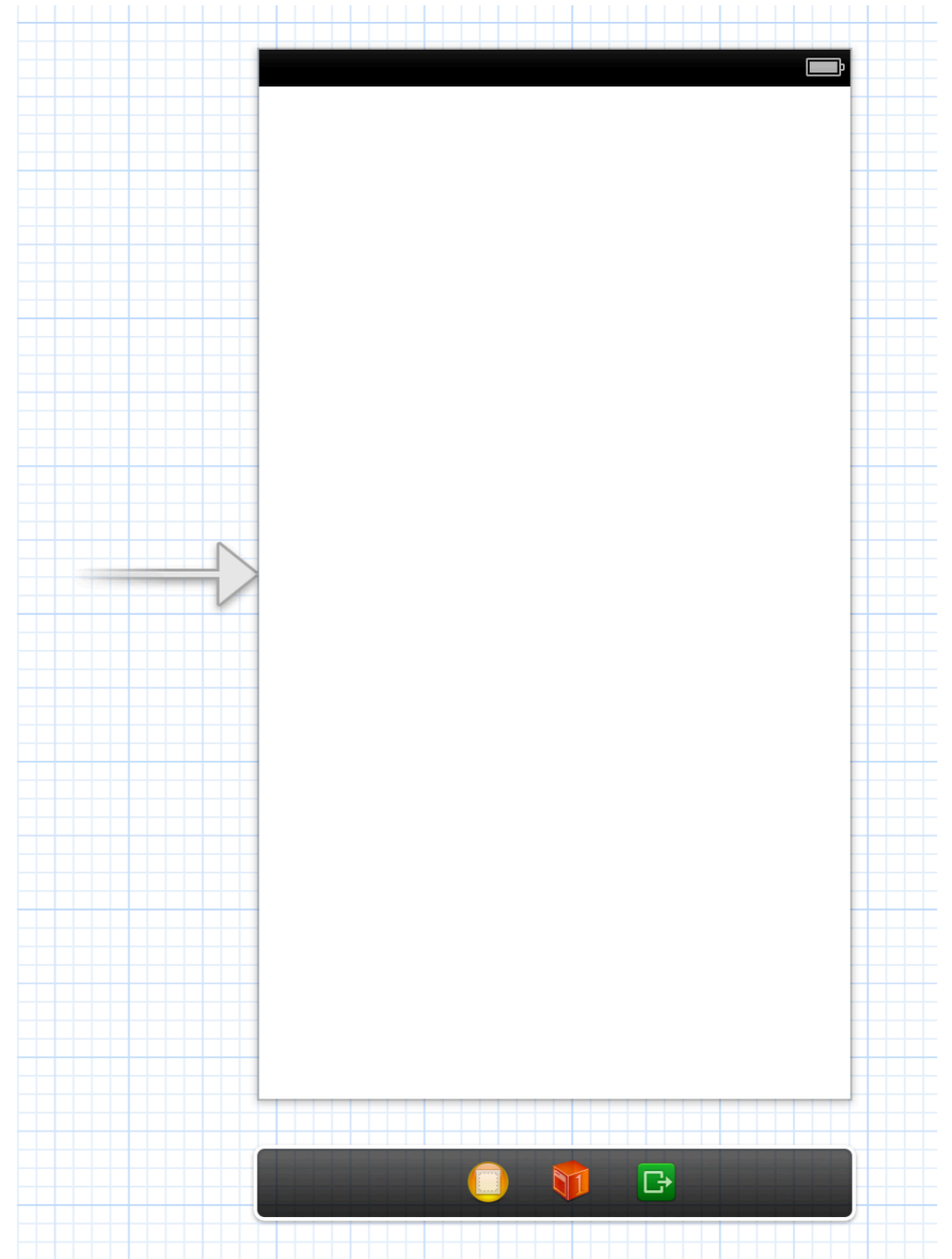
iOS Project Structure (VI)

- All that remains is the storyboard
 - It looks like this
- A storyboard consists of scenes and segues (transitions between scenes)
 - Our app has only a single scene that is controlled by CUIViewController
- The gray arrow indicates the first scene of the application



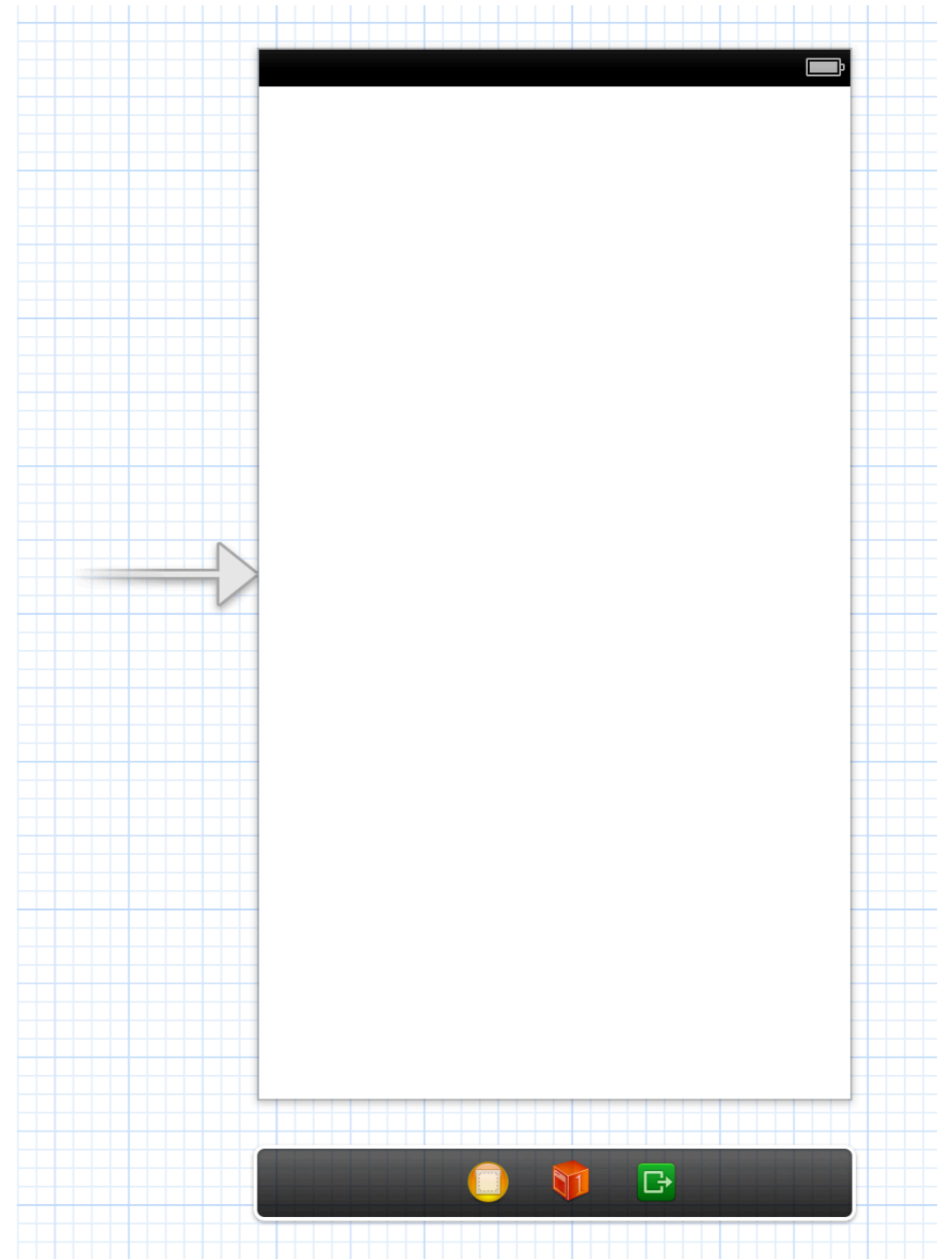
iOS Project Structure (VII)

- When a scene is loaded, the storyboard does the following
 - It instantiates the associated view controller
 - It sets the rootViewController property of the window to point at the newly instantiated view controller
 - It loads up the interface and calls various life cycle methods on the view controller to get the interface displayed and ready for events



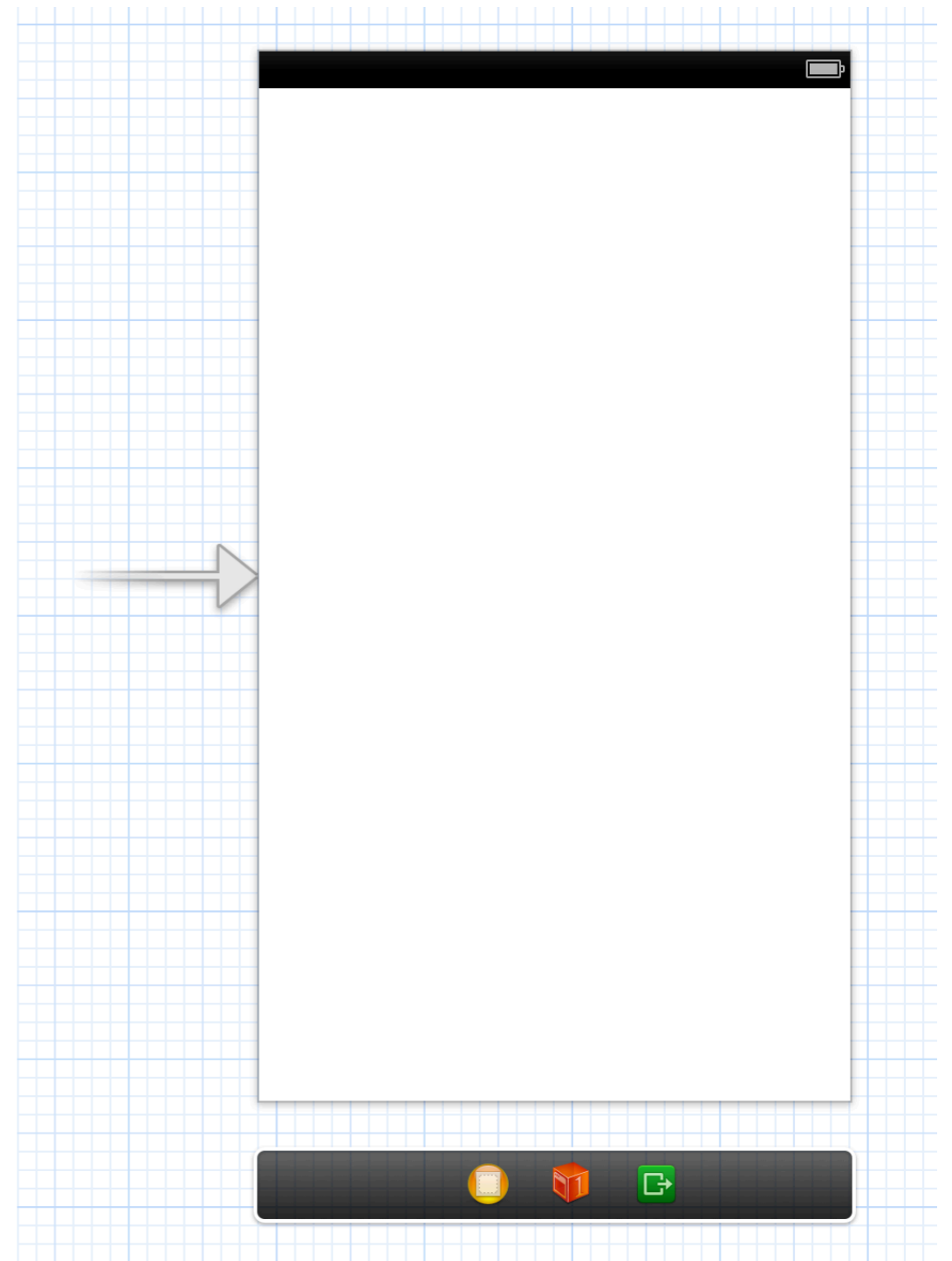
iOS Project Structure (VIII)

- We can add additional scenes with additional view controllers to the storyboard
- We can then specify how to transition between scenes
 - We'll see examples of this in our next lecture on iOS programming



iOS Project Structure (IX)

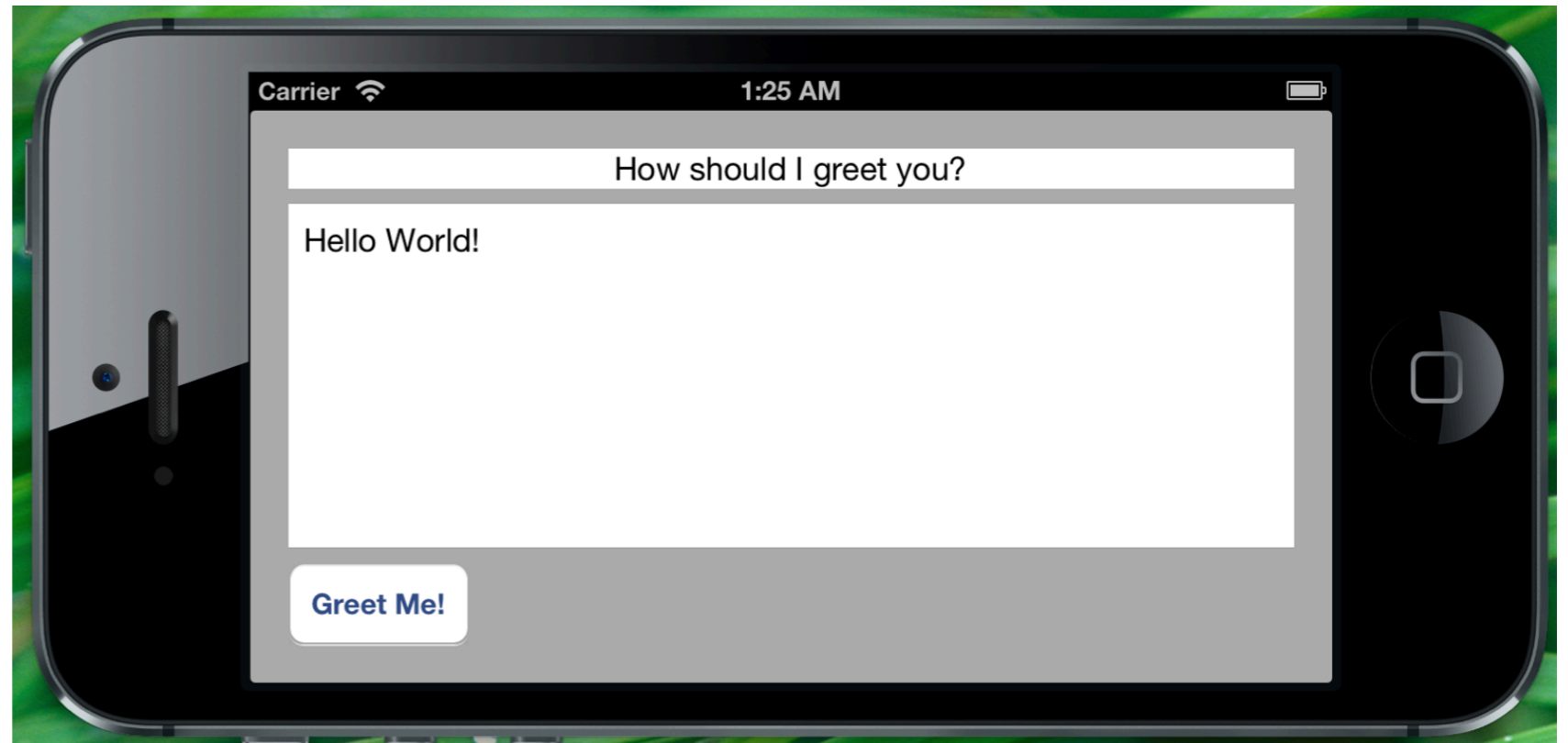
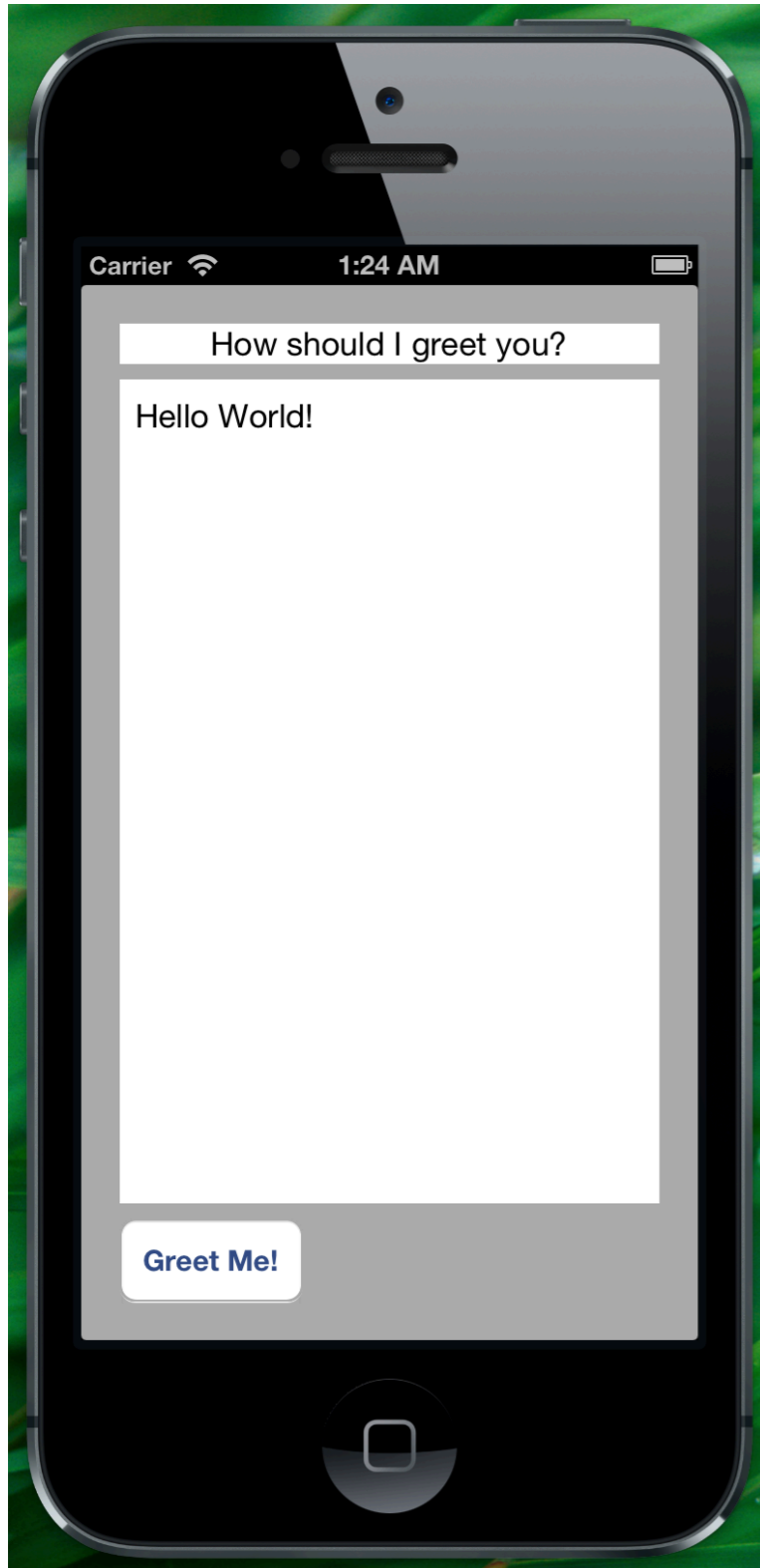
- Currently, our scene contains a single view object (the white background) and that object is connected to our view controller via a property defined by its parent class
 - UIViewController
- We can now drag other widgets onto this view and connect them to our view controller subclass via properties



Add an interface

- Back in Interface Builder
 - Change view's background color to light gray
 - Add a label that says "How should I greet you?"; give it a white background
 - Add a text view and enter "Hello World!" as its default text
 - Add one button that says "Greet Me!"
- Position them to present the UI on the next slide
 - Test out the UI by running the application and test switching the phone from portrait to landscape to confirm that auto-layout picked a reasonable set of defaults

Our UI



Only Skin Deep

- While this UI looks nice, it is currently only “skin deep”
 - It doesn’t actually do anything
 - besides letting you edit the text in the text view
 - But, if you try this, you’ll discover that you can’t make the keyboard go away!
 - We’ll fix this later
- To make progress, we’re going to connect these widgets with our view controller
 - We first need to have properties that point at the text view and the button

Making the connections (I)

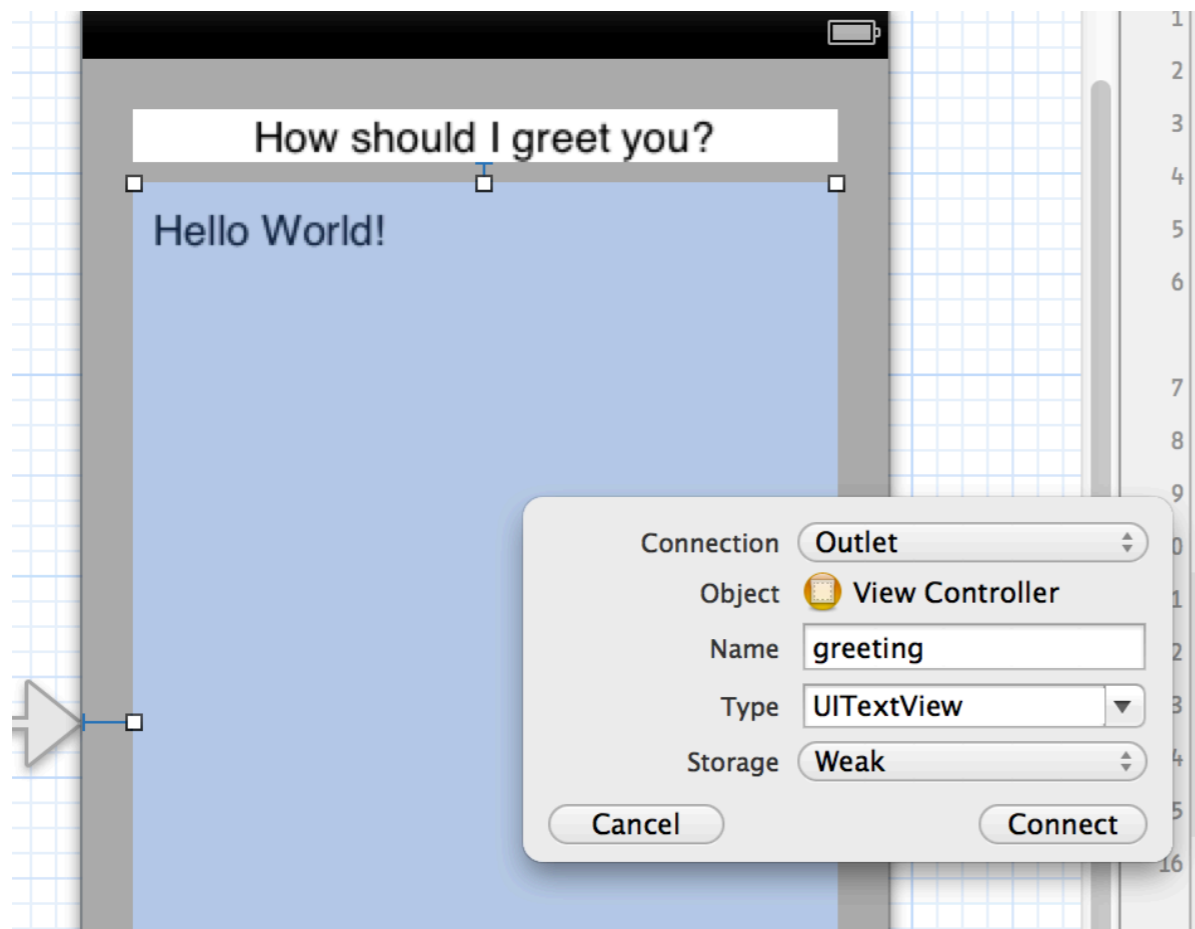
- We're going to use XCode's ability to generate properties for us
- To do this, make sure the storyboard file is selected and then click the assistant editor button in the toolbar; it looks like this:



- This brings up a split view in XCode in which you can see the storyboard and the text of an associated source code file; in this case, XCode selects automatically `CUViewController.h`
 - This file is where we want to create properties for our view controller

Making the connections (II)

- To make the connections, control click on the desired widget and drag over to the source code file until it indicates that it can create an “outlet”
 - An outlet is simply a property that points at a user interface widget
- Let go and the following dialog pops up



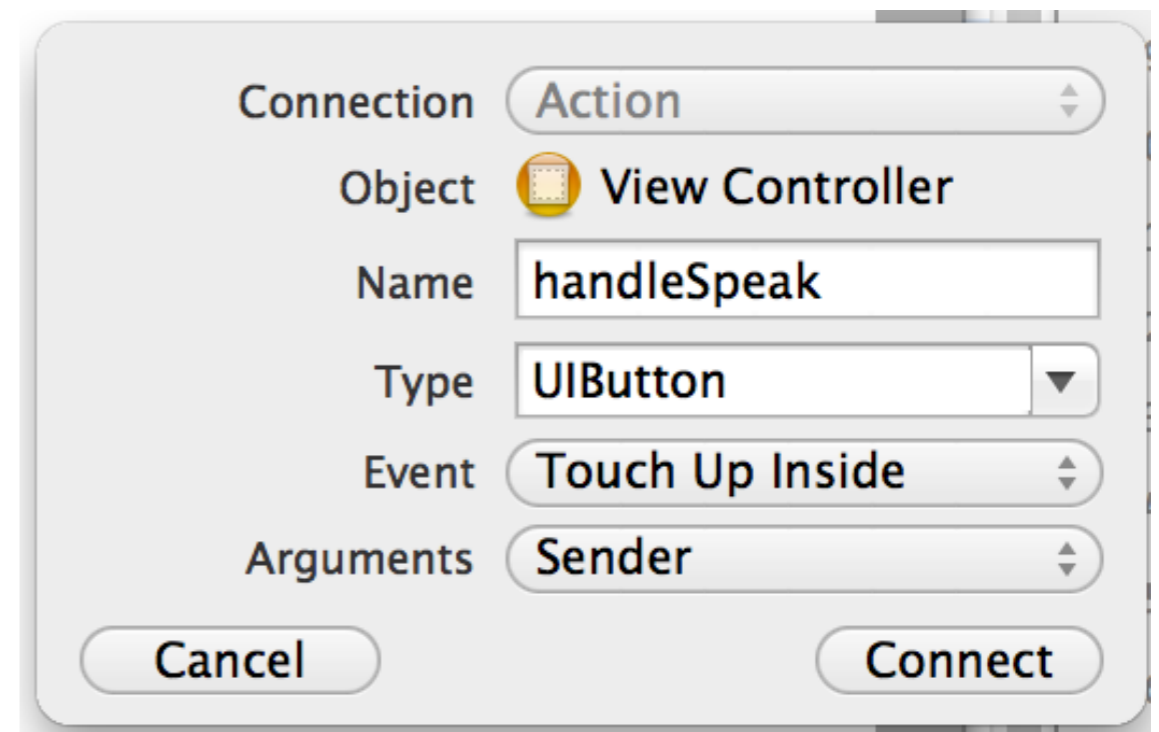
```
1 //  
2 // CUViewController.h  
3 // SpeakHelloWorld  
4 //  
5 // Created by Ken Anderson on 10/17/12.  
6 // Copyright (c) 2012 University of Colorado.  
7 // rights reserved.  
8 //  
9 #import <UIKit/UIKit.h>  
10  
11 @interface CUViewController : UIViewController  
12  
13  
14  
15  
16 @end
```

Making the connections (II)

- Let's call the outlet for the text view “greeting” and use Weak storage
 - We use “weak” here because we don't own the text view
 - It is owned by its enclosing view object
- Let's call the outlet for the Greet Me! button “speak” and use weak storage
- When we're done, the following properties have been defined, connected, and synthesized by XCode; IBOutlet is a special tag for Interface Builder
 - @property (weak, nonatomic) IBOutlet UITextView *greeting;
 - @property (weak, nonatomic) IBOutlet UIButton *speak;

Making the connections (III)

- Now we need to be able to handle events from the various widgets
- We'll start with the buttons
 - Switch the assistant editor to view the CUViewController.m file
 - Control drag from the Greet Me button over to the .m file until a pop-up indicates that you can insert an “action” (an action is simply a method)
 - Configure the dialog to look like this



Making the connections (IV)

- We now have a method that look like this in CUIViewController.m
 - - (IBAction)handleSpeak:(UIButton *)sender {}
- This method is connected to the button's Touch Up Inside event, which means this method will be called when we “touch” the Greet Me button
 - You can verify this by putting a simple NSLog() statement in the method body, running the app, and clicking the button

Getting Ready to Speak (I)

- Before we can make our button event handlers do more than just log that they have been called, we need to prepare our app to actually speak our greeting text
- To do this, we have to import a 3rd party library called OpenEars
 - It's available here: <<http://www.politepix.com/openears/>>
- We first add the following frameworks to our application
 - AudioToolbox and AVFoundation
- We do this the same way we added WebKit to our WebBrowser application on Slide 30
- We then drag the Framework folder from the OpenEars distribution into our Supporting Files group and have XCode copy the files into our project

Getting Ready to Speak (II)

- Now we add the following code to our View Controller
- import statements
 - `#import <Slt/Slt.h>`
 - `#import <OpenEars/FliteController.h>`
- properties (in the class extension; will discuss next lecture)
 - `@property (strong, nonatomic) FliteController *fliteController;`
 - `@property (strong, nonatomic) Slt *slt;`
- in the viewDidLoad method
 - `self.slt = [[Slt alloc] init];`
 - `self.fliteController = [[FliteController alloc] init];`

This creates the text to speech engine and gets it ready to be used.

Speaking

- We're now ready to tie everything together to make our application greet us
- When the handleSpeak method gets called we need to do the following
 - Get the text contained in the text view
 - Pass it to the say:withVoice: method of the fliteController
 - That's it! The code looks like this:

```
- (IBAction)handleSpeak:(UIButton *)sender {  
    [self.fliteController say:[self.greeting.text] withVoice:self.slt];  
}
```
- The text is retrieved using the text property of the text view. Since the text view is stored in our greeting property, we access the text with the phrase "self.greeting.text"

Let's polish the text view

- We need to make sure that the Greet Me button is enabled only 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 view to stop being the first responder to make the keyboard go away
 - Programmatically, we need to invoke the method `resignFirstResponder` on the text view
 - The trick is where do we make this call?

Handling the text view, part one (I)

- To make the keyboard go away when we click outside of the text view, 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 view to stop being the first responder
- 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)

Handling the text view, part one (II)

- Create a new action method called `dismissKeyboard:` and connect this new button to that action via its Touch Up Inside event.
- The `dismissKeyboard:` event will initially look like this:
 - - `(IBAction) dismissKeyboard: (UIButton*) sender {`
 - `[self.greeting resignFirstResponder];`
 - `}`
- You can now run the app, click in the text view, edit the text, and then click outside of the text view, and the keyboard will go away
- Done with the first task; now we want to make sure that the Greet Me button is only enabled when there is text in the text view

Handling the text view, part two (I)

- Now we need to handle enabling and disabling of the Greet Me button
 - We will update the `dismissKeyboard:` method to check the length of the string in the text value after it calls `resignFirstResponder`
 - If the length 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

- The new version of dismissKeyboard: looks like this
 - - (IBAction)dismissKeyboard:(UIButton *)sender {
 - [self.greeting resignFirstResponder];
 - self.speak.enabled = ([self.greeting.text length] > 0);
 - }
- Here we are setting the enabled property of the Greet Me button by passing either YES (true) or NO (false) after checking the length of the greeting
- We also have to configure our button to use light gray text when put in the disabled state. I'll show how to do that in Interface Builder during my demo.

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
 - dismissing keyboards when they are not needed
 - adding/using a third party framework

Coming Up Next

- Lecture 17: Intermediate iOS
- Lecture 18: Intermediate Android