Your First iOS Application

General

2010-07-01

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures

Introduction

This tutorial shows how to create a simple iOS application. It is not intended to give complete coverage of all the features available, but rather to introduce some of the technologies and give you a grounding in the fundamentals of the development process.

You should read this document if you are just starting development for iOS using Cocoa Touch. Although the screenshots show the iPhone-sized simulator, this tutorial is appropriate for any device using iOS—you should use this tutorial to get started even if you intend to develop solely for the iPad.

You should already have some familiarity with the basics of computer programming in general and the Objective-C language in particular. If you haven't used Objective-C before, read through at least *Learning Objective-C: A Primer*.

The goal here is not to create a finely polished application, but to illustrate:

- How you create and manage a project using Xcode
- The fundamental design patterns and techniques that underlie all iOS development
- The basics of using Interface Builder
- How to make your application respond to user input using standard user interface controls

A secondary goal is to point out other documents that you must also read to fully understand the iOS development tools and techniques.

Important: To follow this tutorial, you must have installed the iPhone SDK and developer tools available from the iOS Dev Center.

This document describes the tools that are available for iPhone SDK v4.0—check that your version of Xcode is at least 3.2.2.

This tutorial does not discuss issues beyond basic application development; in particular, it does not describe how to submit applications to the App Store.

Organization of This Document

The document is split into the following chapters:

- "Tutorial Overview and Design Patterns" (page 9) provides an overview of the application you're going to create and the design patterns you'll use.
- "Creating Your Project" (page 13) shows you how to create the project using Xcode and describes how an application launches.
- "Adding a View Controller" (page 19) shows you how to customize a view controller class and create an instance of it.

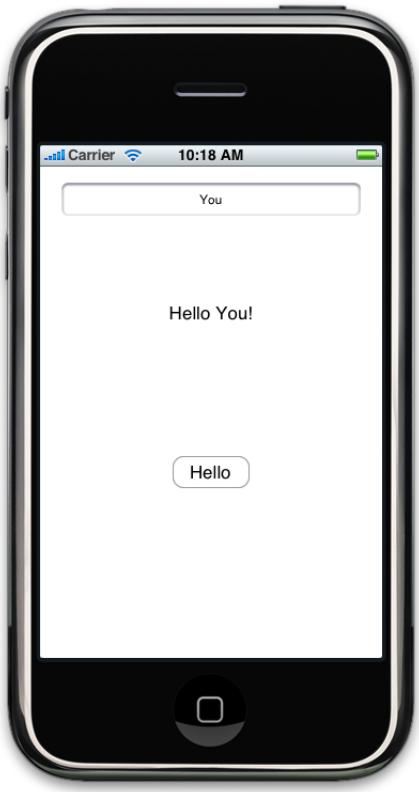
- "Inspecting the Nib File" (page 27) introduces you to two important concepts in Interface Builder: outlets, and the File's Owner proxy object
- "Configuring the View" (page 33) shows you how to add and configure the button and text fields.
- "Implementing the View Controller" (page 41) walks you through implementing the view controller, including memory management, the `changeGreeting:` method, and ensuring that the keyboard is dismissed when the user taps Done
- "Troubleshooting" (page 45) describes some approaches to solving common problems that you may encounter.
- "Where to Next?" (page 47) offers suggestions as to what directions you should take next in learning about iOS development.

Tutorial Overview and Design Patterns

This chapter provides an overview of the application you're going to create and the design patterns you'll use.

Tutorial Overview

In this tutorial, you're going to create a very simple application. It has a text field, a label, and a button. You can type your name into the text field and tap the button to say hello. The application updates the label's text to say "Hello, <Name>!":



Even though this is a very simple application, it introduces the fundamental design patterns, tools, and techniques that underlie all iOS development using Cocoa Touch. Cocoa Touch comprises the UIKit and Foundation frameworks which provide the basic tools and infrastructure you need to implement graphical, event-driven applications in iOS. It also includes several other frameworks that provide key services for accessing device features, such as the user's contacts. To learn more about Cocoa Touch and where it fits into iOS, read *iOS Technology Overview*. The main patterns you're going to use are described in "Design Patterns" (page 10).

In this tutorial, little regard is given to the user interface. Presentation is, however, a critical component of a successful iOS application. You should read the *iPhone Human Interface Guidelines* to understand how the user interface might be improved for a full-fledged application.

You'll also start to gain an understanding of how view controllers work and how they fit into the architecture of an iOS application.

Design Patterns

If you haven't already, you should make sure you read the design patterns chapter in *Cocoa Fundamentals Guide*, however the main patterns you're going to use are:

- Delegation
- Model View Controller
- Target-Action

Here's a quick summary of these patterns and an indication of where they'll be used in the application.

Delegation

Delegation is a pattern where one object sends messages to another object specified as its delegate to ask for input or to notify the delegate that an event is occurring. You typically use it as an alternative to class inheritance for extending the functionality of reusable objects.

In this application, the application object tells its delegate that the main start-up routines have finished and that the custom configuration can begin. For this application, you want the delegate to create an instance of a controller to set up and manage the view. In addition, the text field will tell its delegate (which in this case will be the same controller) when the user has tapped the Return key.

Delegate methods are typically grouped together into a protocol. A protocol is basically just a list of methods. If a class conforms to (or "adopts") a protocol, it guarantees that it implements the required methods of a protocol. (Protocols may also include optional methods.) The delegate protocol specifies all the messages an object might send to its delegate. To learn more about protocols and the role they play in Objective-C, see the "Protocols" chapter in *The Objective-C Programming Language*.

Model-View-Controller

The Model-View-Controller (or "MVC") design pattern sets out three roles for objects in an application.

Model objects represent data such as SpaceShips and Weapons in a game, ToDo items and Contacts in a productivity application, or Circles and Squares in a drawing application.

In this application, the model is very simple—just a string—and it's not actually used outside of a single method, so from a practical perspective in this application it's not even necessary. It's the principle that's important here, though. In other applications the model will be more complicated and accessed from a variety of locations.

View objects know how to display data (model objects) and may allow the user to edit the data.

In this application, you need a main view to contain several other views—a text field to capture information from the user, a second text field to display text based on the user's input, and a button to let the user tell us that the secondary text should be updated.

Controller objects mediate between models and views.

In this application, the controller object takes the data from the input text field, stores it in a string, and updates a second text field appropriately. The update is initiated as a result of an action sent by the button.

Target-Action

The target-action mechanism enables a view object that presents a control—that is, an object such as a button or slider—in response to a user event (such as a click or a tap) to send a message (the action) to another object (the target) that can interpret the message and handle it as an application-specific instruction.

In this application, when it's tapped, the button tells the controller to update its model and view based on the user's input.

Creating Your Project

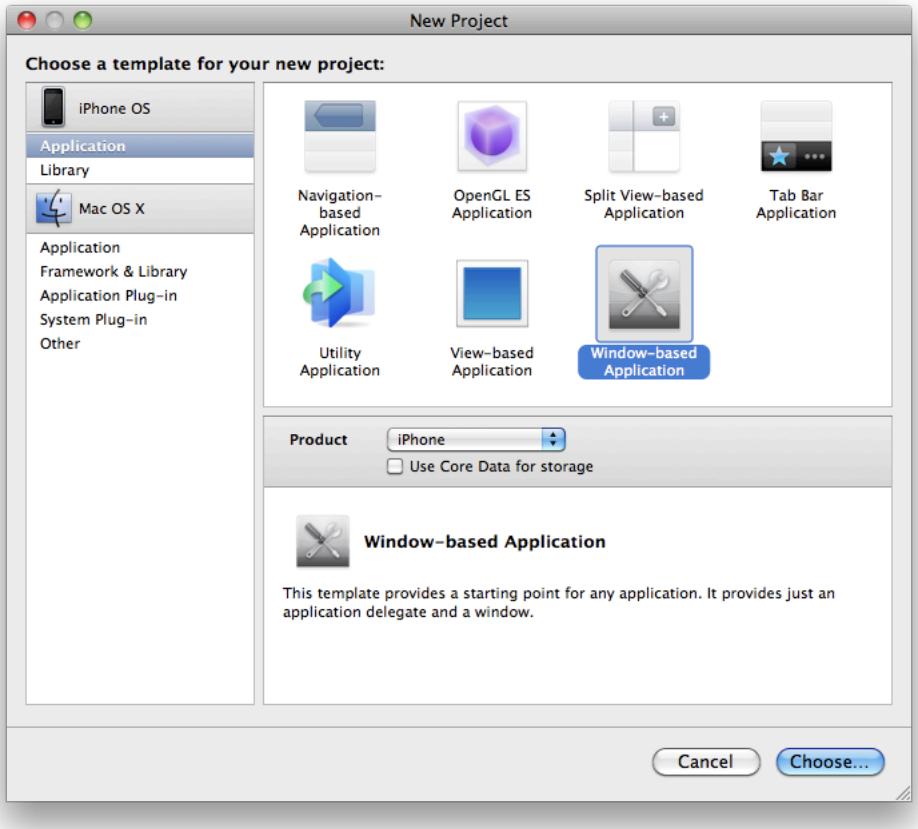
In this chapter, you create the project using Xcode and find out how an application launches.

Xcode

The main tool you use to create applications for iOS is Xcode—Apple's IDE (integrated development environment). You can also use it to create a variety of other project types, including Cocoa and command-line utilities.

> **Notes:** As a convention, >> denotes the beginning of a paragraph (sometimes including the following bulleted list) that contains steps that you must perform in the tutorial.
>
> In code listings, comments included in Xcode template files are not shown.

>> Launch Xcode (by default it's in `/Developer/Applications`), then create a new project by choosing File > New Project. You should see a new window similar to this:



Note: If you don't see the Product and "Use Core Data for storage" options, make sure you have installed version 4.0 of the iPhone OS SDK—you should have Xcode version 3.2.2 or later. You can still follow the tutorial even if you don't have 4.0 installed, just make sure you start with the Window-based Application.

>> Select the Window-Based Application.

>> In the Product popup menu, make sure iPhone is selected. (Do *not* select the option to use Core Data for storage. You don't use Core Data in this example.)

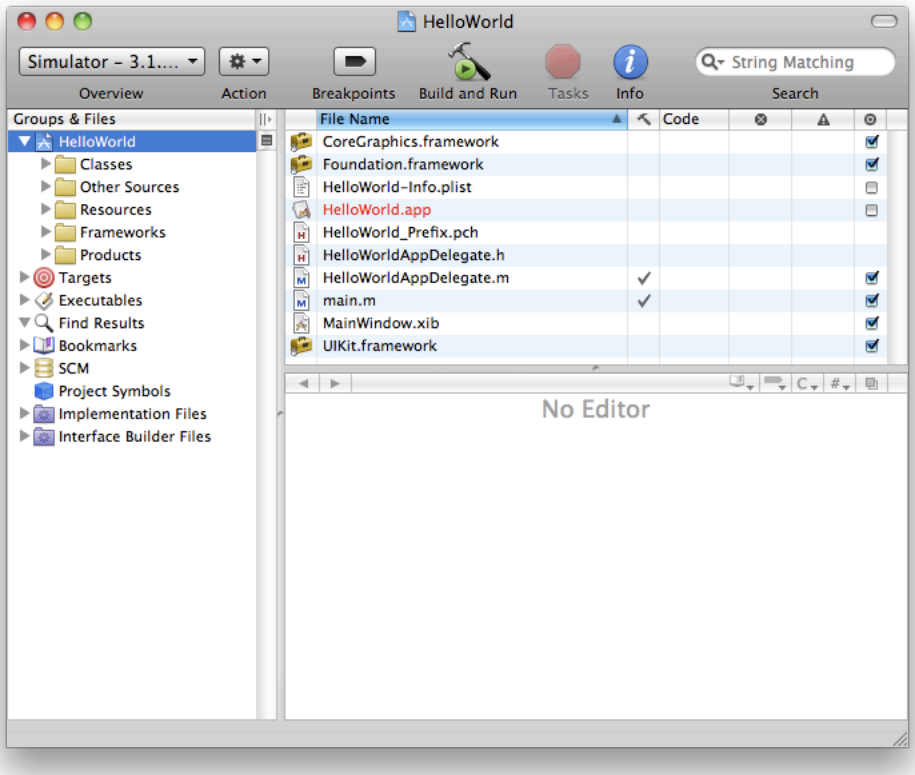
>> Click Choose.

A sheet appears to allow you to select where your project will be saved.

>> Select a suitable location for your project (such as the Desktop or a custom Projects directory), then give the project a name—`HelloWorld`—and click Save.

Note: The remainder of the tutorial assumes that you named the project `HelloWorld`, so the application delegate class is called `HelloWorldAppDelegate`. If you name your project something else, then the application delegate class will be called *YourProjectName*`AppDelegate`.

You should see a new project window like this:



If you haven't used Xcode before, take a moment to explore the application. You should read *Xcode Workspace Guide* to understand the organization of the project window and how to perform basic tasks like editing and saving files. You can now build and run the application to see what the Simulator looks like.

>> Choose Build > Build and Go (Run) or click the Build and Go button in the toolbar.

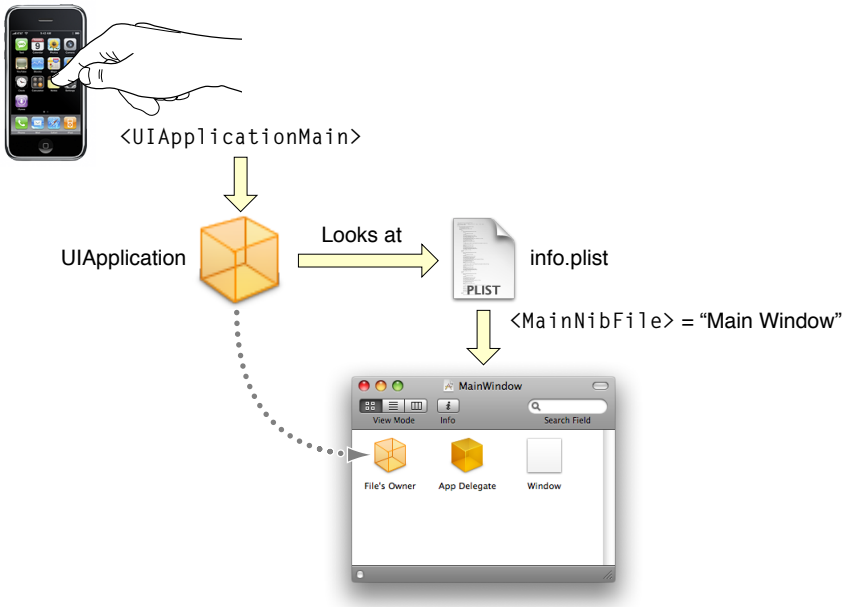
The iPhone Simulator application should launch automatically, and when your application starts up you should simply see a white screen. To understand where the white screen came from, you need to understand how the application starts up.

>> Quit the Simulator.

Application Bootstrapping

The template project you created already sets up the basic application environment. It creates an application object, connects to the window server, establishes the run loop, and so on. Most of the work is done by the `UIApplicationMain` function as illustrated in Figure 2-1.

Figure 2-1 Application bootstrapping



The `main` function in `main.m` calls the `UIApplicationMain` function:

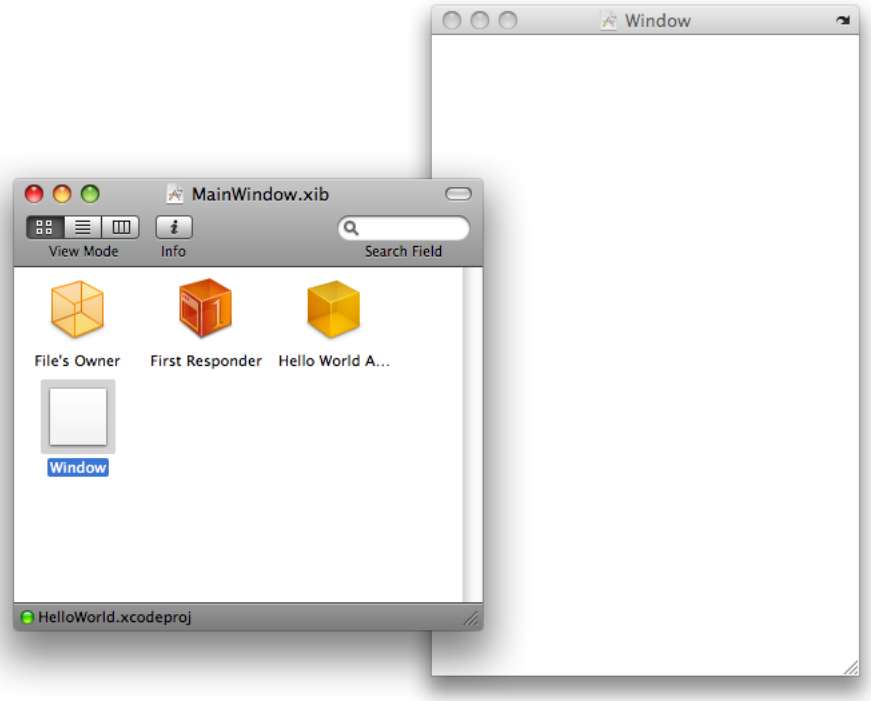
```
int retVal = UIApplicationMain(argc, argv, nil, nil);
```

This creates an instance of `UIApplication`. It also scans the application's `Info.plist` property list file. The `Info.plist` file is a dictionary that contains information about the application such as its name and icon. It may contain the name of the **nib file** the application object should load, specified by the `NSMainNibFile` key. Nib files contain an archive of user interface elements and other objects—you'll learn more about them later in the tutorial. In your project's `Info.plist` file you should see:

```
<key>NSMainNibFile</key>
<string>MainWindow</string>
```

This means that when the application launches, the MainWindow nib file is loaded.

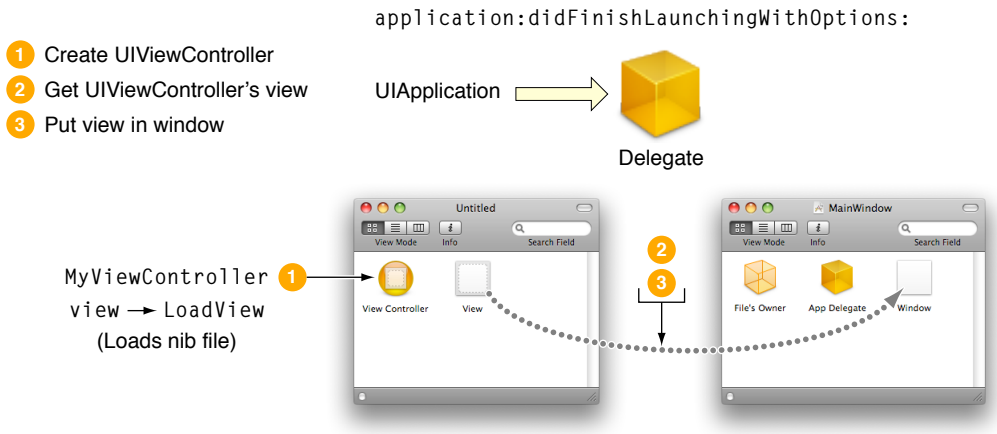
>> To look at the nib file, double-click `MainWindow.xib` in the Resources group in the project window (the file has the extension "xib" but by convention it is referred to as a "nib file"). Interface Builder launches and opens the file.



The Interface Builder document contains four items:

- A **File's Owner** proxy object. The File's Owner object is actually the `UIApplication` instance—File's Owner is discussed later, in "File's Owner" (page 28).
- A **First Responder** proxy object. The First Responder is not used in this tutorial but you can learn more about it by reading "Event Handling" in *iOS Application Programming Guide*.
- An instance of `HelloWorldAppDelegate` set to be the application's **delegate**. Delegates are discussed in the next section.
- A window. The window has its background set to white and is set to be visible at launch. It's this window that you see when the application launches.

After the application has finished launching, you can perform additional customization. A common pattern—and the one you'll follow in the next chapter—is illustrated in this diagram:



When the application object has completed its setup, it sends its delegate an `application:didFinishLaunchingWithOptions:` message. Rather than configuring the user interface itself, the delegate typically creates a **view controller** object (a special controller responsible for managing a view—this adheres to the model-view-controller design pattern as described in "Model-View-Controller" (page 10)). The delegate asks the view controller for its view (which the view controller creates on demand) and adds that as a subview of the window.

Recap

In this chapter you created a new project and learned about how the application launch process works. In the next chapter, you'll define and create an instance of a view controller.

Adding a View Controller

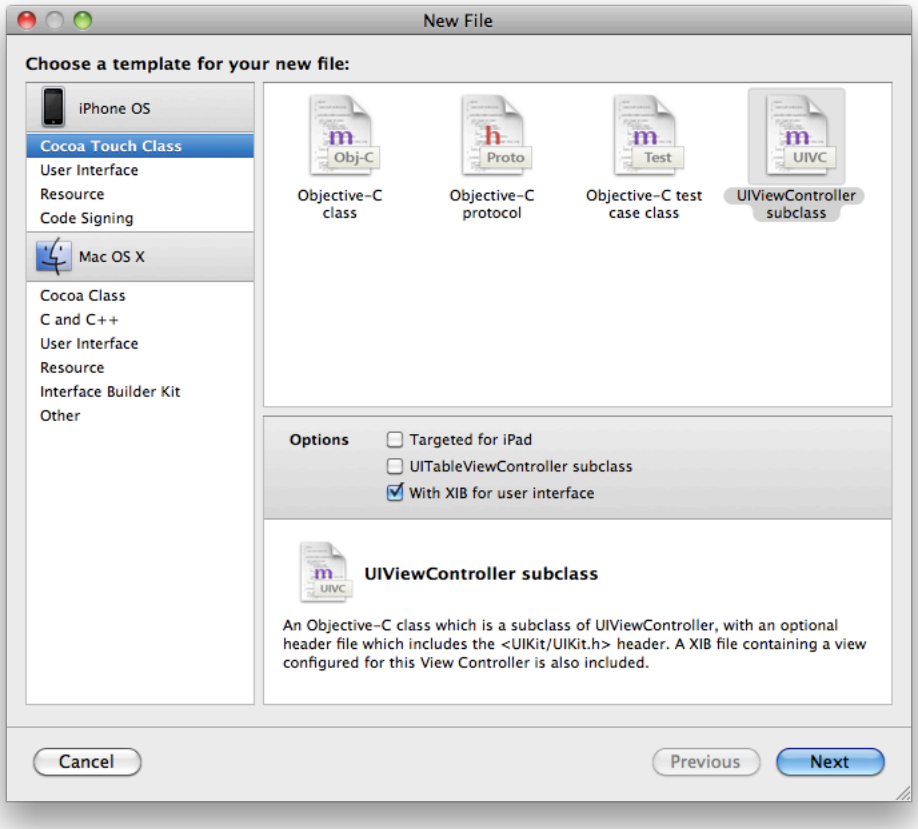
In this application you'll need two classes. Xcode's application template provided an application delegate class and an instance is created in the nib file. You need to implement a view controller class and create an instance of it.

Adding a View Controller Class

View controller objects play a central role in most iOS applications. As the name implies, they're responsible for managing a view, but on iOS they also help with navigation and memory management. You're not going to use the latter features here, but it's important to be aware of them for future development. UIKit provides a special class—`UIViewController`—that encapsulates most of the default behavior you want from a view controller. You have to create a subclass to customize the behavior for your application.

>> In Xcode, in the project organizer select either the project (`HelloWorld` at the top of the Groups and Files list) or the Classes group folder—the new files will be added to the current selection.

>> Choose File > New File and in the New File window. Select the Cocoa Touch Classes group, then select `UIViewController` subclass. In the Options section, choose only With XIB for user interface.

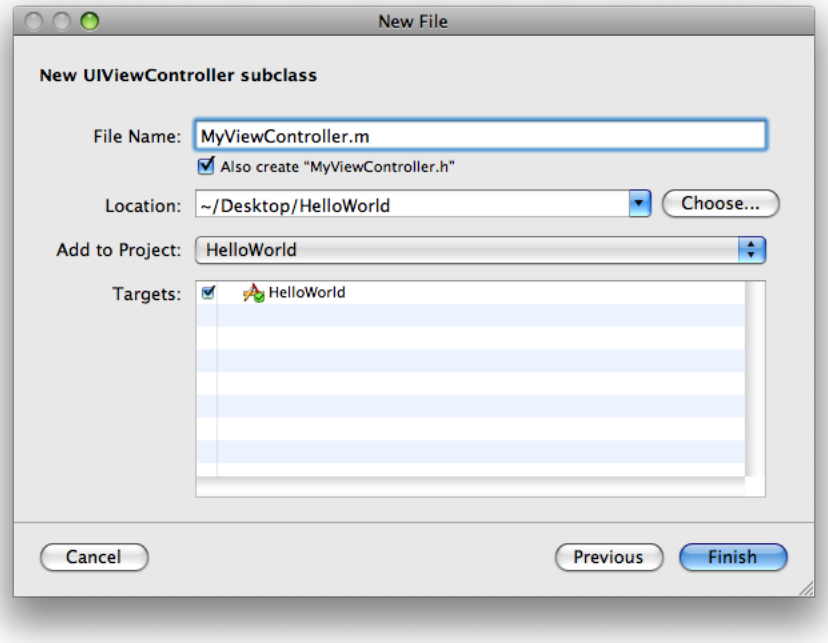


Note: If you don't see the "With XIB for user interface" option, make sure you have installed version 3.2 of the iPhone OS SDK—Xcode should show version 3.2.2 or later.

Selecting "With XIB for user interface" means that Xcode creates a nib file to accompany the view controller, and adds it to the project. (Nib files are discussed in detail in the next chapter.)

>> Click Next, and in the following screen give the file a new name such as `MyViewController` (by convention, class names begin with a capital letter). Make sure that both the `.m` and `.h` files are created and that the files marked as being added to the project's target, as shown here:

Figure 3-1 MyViewController



>> Press Finish and make sure that the files were added to your project.

If you look at the new source files, you'll see that stub implementations of various methods are already given to you. These are all you need for the moment; the next task is to create an instance of the class.

Adding a View Controller Property

You want to make sure that the view controller lasts for the lifetime of the application, so it makes sense to add it as an instance variable of the application delegate (which will also last for the lifetime of the application). (To understand why, consult *Memory Management Programming Guide*.)

The instance variable will be an instance of the `MyViewController` class. The compiler will generate an error, though, if you declare the variable but you don't tell it about the `MyViewController` class. You could import the header file, but typically in Cocoa you instead provide a **forward declaration**—a promise to the compiler that `MyViewController` will be defined somewhere else and that it needn't waste time checking for it now. (Doing this also avoids circularities if two classes need to refer to each other and would otherwise include each other's header files.) You then import the header file itself in the implementation file.

>> In the application delegate's header file (`HelloWorldAppDelegate.h`), add this forward declaration before the interface declaration for `HelloWorldAppDelegate`:

```
@class MyViewController;
```

>> Add the instance variable by adding the following line between the braces:

```
MyViewController *myViewController;
```

>> Add a declaration for this property after the closing brace but before `@end`:

```
@property (nonatomic, retain) MyViewController *myViewController;
```

Important: Like C, Objective-C is case-sensitive. `MyViewController` refers to a class; `myViewController` is a variable that here contains an instance of `MyViewController`. A common programming error in Cocoa is to misspell a symbol name, frequently by using a letter of an inappropriate case (for example, "tableview" instead of "tableView"). This tutorial deliberately uses `MyViewController` and `myViewController` to encourage you to look at the name carefully. When writing code, make sure you use the appropriate symbol.

Properties are described in the "Declared Properties" chapter in *The Objective-C Programming Language*. Basically, though, this declaration specifies that an instance of `HelloWorldAppDelegate` has a property that you can access using the getter and setter methods `myViewController` and `setMyViewController:` respectively, and that the instance retains the property (retaining is discussed in more detail later).

To make sure you're on track, confirm that your `HelloWorldAppDelegate` class interface file (`HelloWorldAppDelegate.h`) looks like this (comments are not shown):

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

@class MyViewController;

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    MyViewController *myViewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) MyViewController *myViewController;

@end
```

You can now create an instance of the view controller.

Creating the View Controller Instance

Now that you've added the view controller property to the application delegate, you need to actually create an instance of the view controller and set it as the value for the property.

>> In the implementation file for the application delegate class (`HelloWorldAppDelegate.m`), create an instance of `MyViewController` by adding the following code as the first statements in the implementation of the `application:didFinishLaunchingWithOptions:` method:

```
MyViewController *aViewController = [[MyViewController alloc]
            initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
[self setMyViewController:aViewController];
[aViewController release];
```

There's quite a lot in just these three lines. What they do is:

- Create and initialize an instance of the view controller class.
- Set the new view controller to be the `myViewController` instance variable using an accessor method.

Remember that you didn't separately declare `setMyViewController:`, it was implied as part of the property declaration—see "Adding a View Controller Property" (page 21).

- Adhere to memory management rules by releasing the view controller.

You create the view controller object using `alloc`, then initialize it using `initWithNibName:bundle:`. The `init` method specifies first the name of the nib file the controller should load and second the bundle in which it should find it. A bundle is an abstraction of a location in the file system that groups code and resources that can be used in an application. The advantages of using bundles over locating resources yourself in the file-system are that bundles provide a convenient and simple API—the bundle object can locate a resource just by name—and they take account of localization for you. To learn more about bundles, see *Resource Programming Guide*.

By convention, you own any objects you create using an `alloc` method (amongst others, see "Memory Management Rules"). You should also:

- Relinquish ownership of any objects you create.
- Typically use accessor methods to set instance variables anywhere other than in an initializer method.

The second line in the implementation uses an accessor method to set the instance variable, and then the third line uses `release` to relinquish ownership.

There are other ways to implement the above. You could, for example, replace the three lines with just two:

```
MyViewController *aViewController = [[[MyViewController alloc]
            initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]]
autorelease];
[self setMyViewController:aViewController];
```

In this version, you use `autorelease` as a way to relinquish ownership of the new view controller but at some point in the future. To understand this, read "Autorelease Pools" in the *Memory Management Programming Guide*. In general, however, you should try to avoid using `autorelease` wherever possible as it's a more resource intensive operation than `release`.

You could also replace the last line with:

```
self.myViewController = aViewController;
```

The dot notation invokes exactly the same accessor method (`setMyViewController:`) as in the original implementation. The dot notation simply provides a more compact syntax—especially when you use nested expressions. Which syntax you choose is largely personal preference, although using the dot syntax does have some additional benefits when used in conjunction with properties—see "Declared Properties" in *The Objective-C Programming Language*. For more about the dot syntax, see "Dot Syntax" in *The Objective-C Programming Language* in "Objects, Classes, and Messaging" in *The Objective-C Programming Language*.

Setting Up the View

The view controller is responsible for managing and configuring the view when asked. Rather than creating the window's content view directly, therefore, you ask the view controller for its view and add that as the subview for the window.

>> After releasing the view controller, add the following lines:

```
UIView *controllersView = [myViewController view];
[window addSubview:controllersView];
```

You could do this in one line:

```
[window addSubview:[myViewController view]];
```

But breaking it into two serves to highlight the side of memory management that is the converse of that which you saw earlier. Because you didn't create the controller view using any of the methods listed in "Memory Management Rules" in *Memory Management Programming Guide*, you don't own the returned object. Consequently you can simply pass it to the window and forget about it (you don't have to release it).

The final line from the template:

```
[window makeKeyAndVisible];
```

causes the window—now complete with your view—to be displayed on screen. You add your view before the window is displayed so that the user doesn't briefly see a blank screen before the real content appears.

Housekeeping

There are a few unfinished tasks to complete: You need to import the view controller's header file, synthesize the accessor methods, and—to conform to the rules of memory management—make sure the view controller is released in the `dealloc` method.

>> In the implementation file for the application delegate class (`HelloWorldAppDelegate.m`), do the following:

- At the top of the file, import the header file for `MyViewController`:

  ```
  #import "MyViewController.h"
  ```

- In the `@implementation` block of the class, tell the compiler to synthesize the accessor methods for the view controller:

  ```
  @synthesize myViewController;
  ```

- In the `dealloc` method, release the view controller in the first statement:

  ```
  [myViewController release];
  ```

Implementation Source Listing

To make sure you're still on track, confirm that your `HelloWorldAppDelegate` class implementation (`HelloWorldAppDelegate.m`) looks like this:

```
#import "MyViewController.h"
#import "HelloWorldAppDelegate.h"

@implementation HelloWorldAppDelegate
```

```
@synthesize window;
@synthesize myViewController;


- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    MyViewController *aViewController = [[MyViewController alloc]
              initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
    [self setMyViewController:aViewController];
    [aViewController release];

    UIView *controllersView = [myViewController view];
    [window addSubview:controllersView];
    [window makeKeyAndVisible];
}


- (void)dealloc {
    [myViewController release];
    [window release];
    [super dealloc];
}

@end
```

Test the Application

You can now test your application.

>> Compile and run the project (choose Build > Build and Run, or click the Build and Run button in Xcode's toolbar).

Your application should compile without errors and you should again see a white screen in Simulator.

Recap

In this section you added a new view controller class and its accompanying nib file. In the application delegate, you declared an instance variable and accessor methods for a view controller instance. You also synthesized the accessor methods and performed a few other housekeeping tasks. Most importantly, though, you created an instance of the view controller and passed its view to the window. In the next chapter you'll use Interface Builder to configure the nib file the controller uses to load its view.

Inspecting the Nib File

You use the Interface Builder application to create and configure nib files. There are two important concepts to introduce: outlets, and the File's Owner proxy object.

Interface Builder

Interface Builder is the application you use to create user interfaces. It doesn't generate source code, instead it allows you to manipulate objects directly and then save those objects in an archive called a nib file.

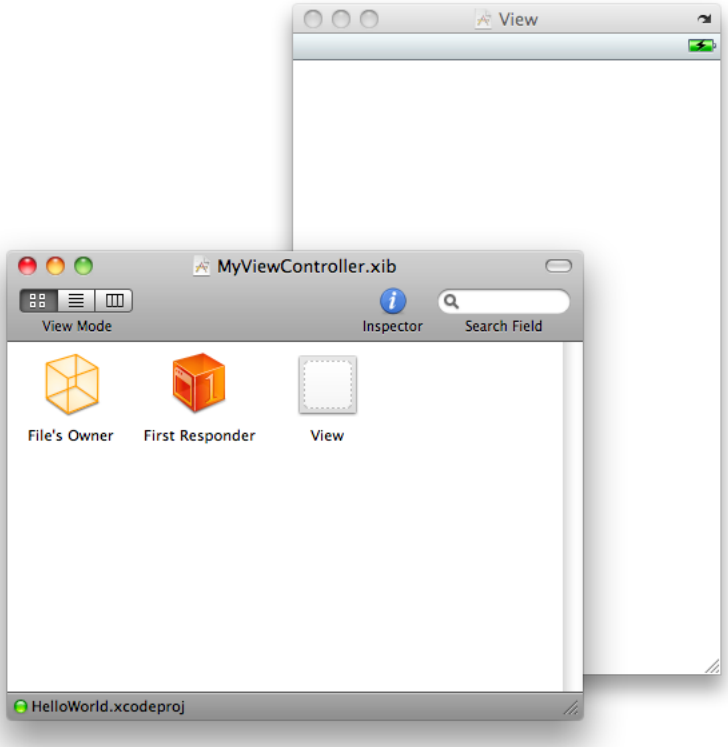
Terminology: Although an Interface Builder document may have a ".xib" extension, historically the extension was ".nib" (an acronym for "NextStep Interface Builder") so they are referred to colloquially as "Nib files."

At runtime, when a nib file is loaded the objects are unarchived and restored to the state they were in when you saved the file—including all the connections between them. To learn more about Interface Builder, read *Interface Builder User Guide*.

Inspect the Nib File

>> In Xcode, double-click the view controller's XIB file (`MyViewController.xib`) to open the file in Interface Builder.

The file contains three objects, the File's Owner proxy, the First Responder proxy, and a view. The view is displayed in a separate window to allow you to edit it.

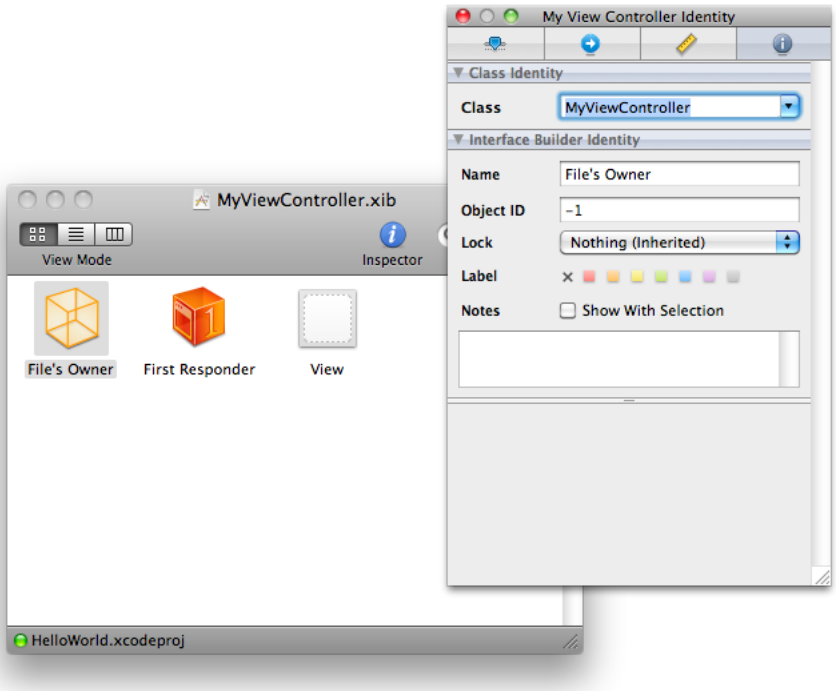


File's Owner

In an Interface Builder document, in contrast to the other objects you add to the interface, the File's Owner object is not created when the nib file is loaded. It represents the object set to be the owner of the user interface—typically the object that loads the interface. This is discussed in more detail in *Resource Programming Guide*. In your application, the File's Owner will be the instance of `MyViewController`.

To allow you to make appropriate connections to and from the File's Owner, Interface Builder has to know what sort of object File's Owner is. You tell Interface Builder the class of the object using the Identity Inspector. It was actually set when you created the nib file together with the view controller classes, but it's useful to look at the inspector now.

>> In the Interface Builder document window, select the File's Owner icon and then choose Tools > Identity Inspector to display the Identity inspector as shown here:

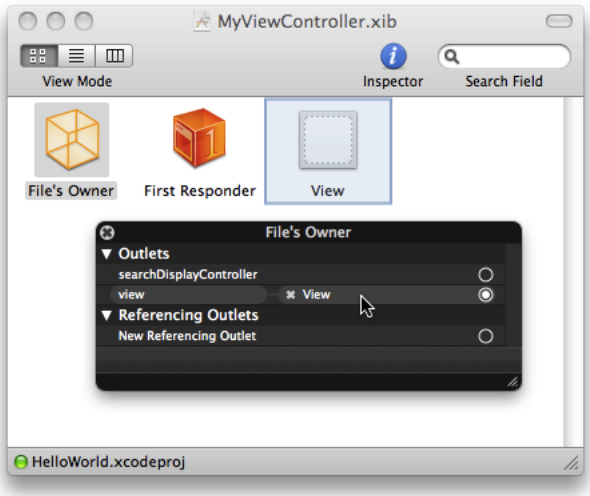


In the Class field of the Class Identity section, you should see `MyViewController`. It's important to understand that this is just a promise to Interface Builder that the File's Owner will be an instance of `MyViewController`. Setting the class doesn't ensure that the File's Owner will be an instance of that class. The File's Owner is simply whatever object is passed as the *`owner`* argument to the `loadNibNamed:owner:options:` method. (You don't use this method directly in the tutorial, but the view controller uses it to load the nib file.) If it's an instance of a different class, the connections you make in the nib file will not be properly established.

The View Outlet

You can look at—and make and break—an object's connections using an inspector panel.

>> In the Interface Builder document window, Control-click on File's Owner to display a translucent panel showing File's Owner's connections:



The only connection at the moment is the view controller's `view` outlet. An **outlet** is just an attribute (typically an instance variable) that happens to connect to an item in a nib file. The outlet connection means that when the nib file is loaded and the `UIView` instance is unarchived, the view controller's `view` instance variable is set to that view.

Loading the Nib File

The view controller loads the nib file automatically in its `loadView` method. Recall that you specified the name of the nib file to load as the first argument to `initWithNibName:bundle:` (see "Creating the View Controller Instance" (page 22)). The `loadView` method is typically called once during a view controller's lifetime and is used to create its view. When you invoke the view controller's `view` method, the controller automatically calls its own `loadView` method if the view hasn't been created. (If the view controller purges its view as a result of, for example, receiving a memory warning, then `loadView` will be invoked again to recreate the view if necessary.)

If you want to create the view controller's view programmatically, you can override `loadView` and create the view in your implementation.

If you initialize a view controller using `initWithNibName:bundle:` but you want to perform additional configuration after the view is loaded, you override the controller's `viewDidLoad` method.

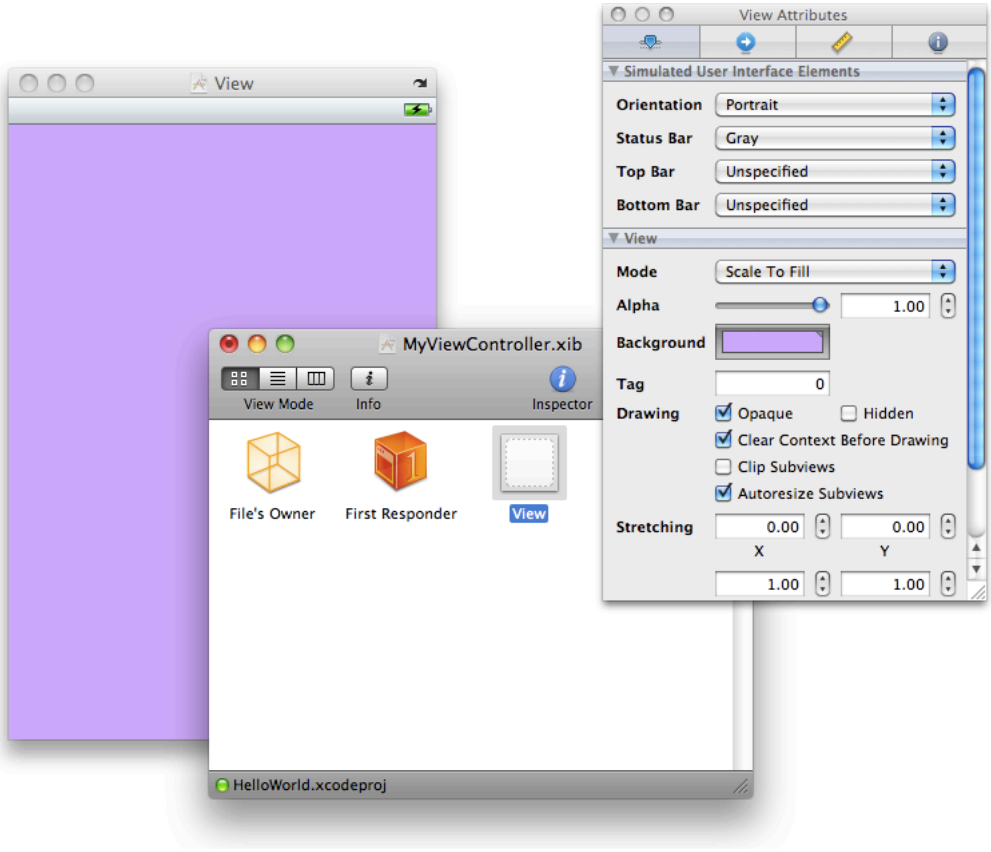
You can load nib files yourself using an instance of `NSBundle`. You can learn more about loading nib files in *Resource Programming Guide*.

Test the Application

To make sure your application's working correctly, you could set the background color of the view to something other than white and verify that the new color is displayed after the application launches.

>> In Interface Builder, select the view then choose Tools > Attributes Inspector to display the Attributes inspector.

>> Click the frame of the Background color well to display the Colors panel, then select a different color.



>> Save the nib file.

>> Compile and run the project (click the Build and Go button in the toolbar).

Your application should compile without errors and you should again see an appropriately-colored screen in Simulator.

>> Restore the view's background color to white and save the nib file.

Recap

In this section you inspected a nib file, learned about outlets, and set the background color of a view. You also learned more about resource loading and how the view controller loads the nib file.

In the next chapter you will add controls to the view.

Configuring the View

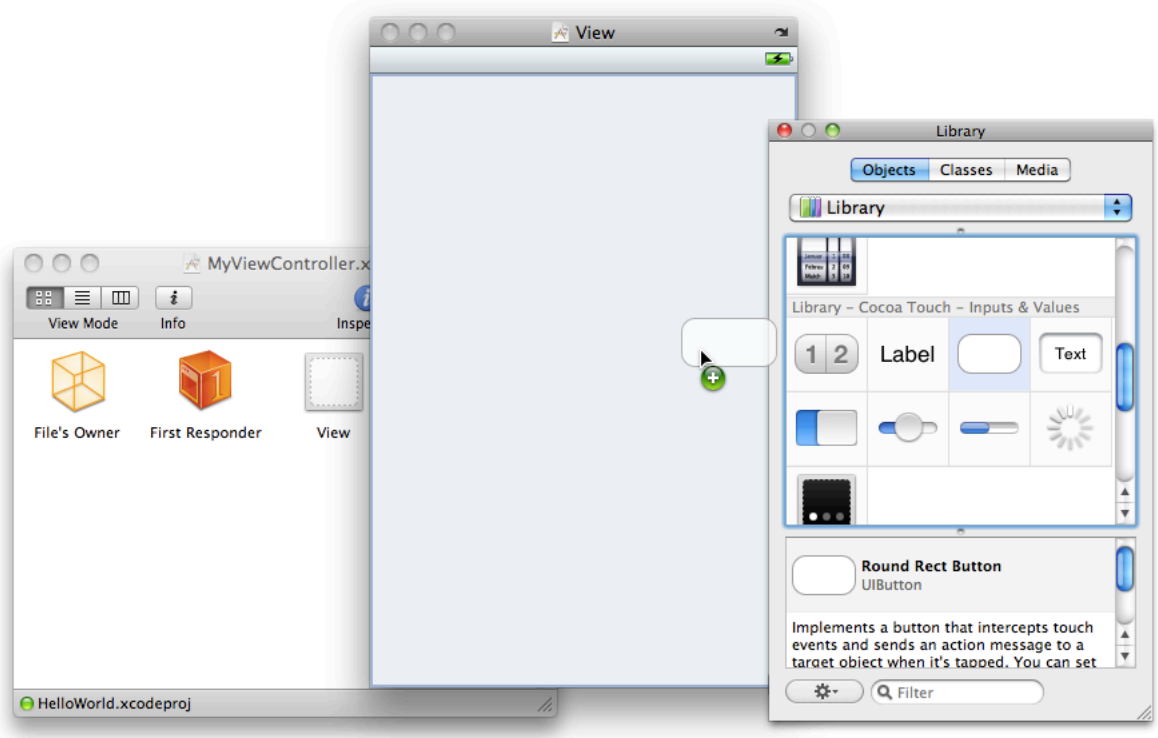
Interface Builder provides a library of objects that you can add to a nib file. Some of these are user interface elements such as buttons and text fields, others are controller objects such as view controllers. Your nib file already contains a view—now you just need to add the button and text fields.

Adding the User Interface Elements

You add user interface elements by dragging them from the Interface Builder library.

>> In Interface Builder choose Tools > Library to display the library window

You can drag view items from the library and drop them onto the view just as you might in a drawing application.

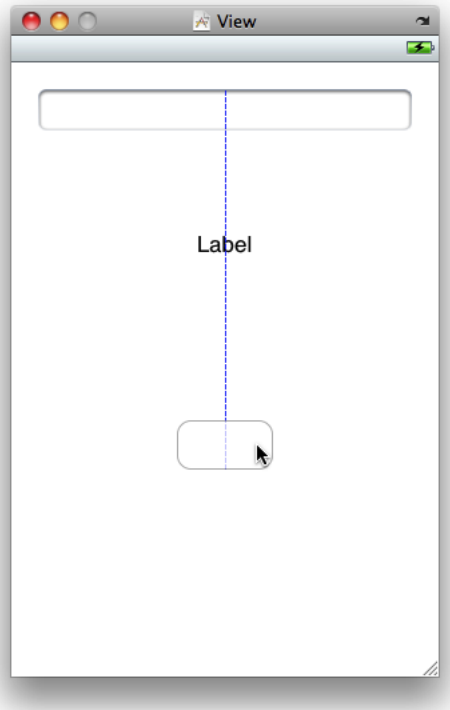


>> Add a text field (`UITextField`), a label (`UILabel`), and a button (`UIButton`) to the view.

You can then resize the items using resize handles where appropriate, and reposition them by dragging. As you move items within the view, alignment guides are displayed as dashed blue lines.

>> Lay out the elements so that they look like this:

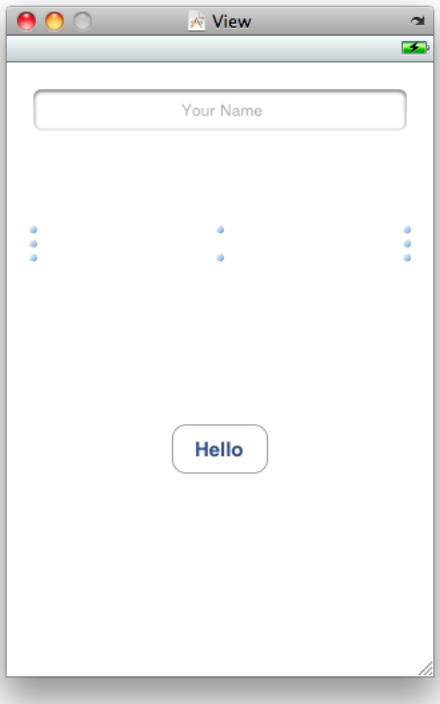
Figure 5-1 View containing user interface elements and showing a guide line



>> Make the following changes:

1. Add a placeholder string `Your Name` to the text field by entering it in the Text Field Attributes inspector.
2. Resize the label so that it extends for the width of the view.
3. Delete the text ("Label") from the label either using the Label Attributes inspector or by directly selecting the text in the label (double-click to make a selection) and press Delete.
4. Add a title to the button by double-clicking inside the button and typing `Hello`.
5. Use the inspector to set the text alignment for the text field and label to centered.

You should end up with a view that looks like this:



>> In the view section of the label's Label Attributes inspector, select Clear Context Before Drawing. This ensures that when you update the greeting the previous string is removed before the new one is drawn. If you don't do this, the strings are drawn on top of each other.

There are several other changes to make to the text field—it might be fairly obvious that the first change applies to the text field, but the others are less obvious. First, you might want to ensure that names are automatically capitalized. Second, you might want to make sure that the keyboard associated with the text field is configured for entering names, and that the keyboard displays a Done button.

The guiding principle here is that: you know when you're putting it on screen what a text field will contain. You therefore design the text field to make sure that at runtime the keyboard can configure itself to best suit the user's task. You make all of these settings using text input traits.

>> In Interface Builder, select the text field then display the Attributes inspector. In the Text Input Traits section:

- In the Capitalize popup menu, select Words
- In the Keyboard Type popup menu select, Default
- In the Keyboard Return Key popup menu, select Done

>> Save the file.

If you build and run your application in Xcode, when it launches you should see the user interface elements as you positioned them. If you press the button, it should highlight, and if you tap inside the text field, the keyboard should appear. At the moment, though, after the keyboard appears, there's no way to dismiss it. To remedy this, and add other functionality, you need to make appropriate connections to and from the view controller. These are described in the next section.

The View Controller Interface Declaration

To make connections to the user interface from the view controller, you need to specify outlets (recall that an outlet is just an instance variable). You also need a declaration for its very simple model object, the string.

>> In Xcode, in `MyViewController.h` add the following instance variables to the `MyViewController` class:

```
UITextField *textField;
UILabel *label;
NSString *string;
```

>> You then need to add property declarations for the instance variables and a declaration for the `changeGreeting:` action method:

```
@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, copy) NSString *string;
- (IBAction)changeGreeting:(id)sender;
```

`IBOutlet` is a special keyword that is used only to tell Interface Builder to treat an instance variable or property as an outlet. It's actually defined as nothing so it has no effect at compile time.

`IBAction` is a special keyword that is used only to tell Interface Builder to treat a method as an action for target/action connections. It's defined to `void`.

The view controller is also going to be the text field's delegate; as such, it must adopt the `UITextFieldDelegate` protocol. To specify that a class adopts a protocol, in the interface add the name of the protocol in angle brackets (<>) after the name of the class from which your class inherits.

>> Specify that the `UIViewController` object adopts the `UITextFieldDelegate` protocol by adding `<UITextFieldDelegate>` after `UIViewController`.

Your interface file should look like this:

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

@interface MyViewController : UIViewController <UITextFieldDelegate> {
    UITextField *textField;
    UILabel *label;
    NSString *string;
}
@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, copy) NSString *string;
- (IBAction)changeGreeting:(id)sender;
@end
```

>> Save the `MyViewController.h` file so that Interface Builder will notice the changes.

So that you can test the project at the end of this section, in the implementation file (`MyViewController.m`) implement a stub `changeGreeting:` method.

>> After the `@implementation MyViewController` line add:

```
- (IBAction)changeGreeting:(id)sender {
}
```

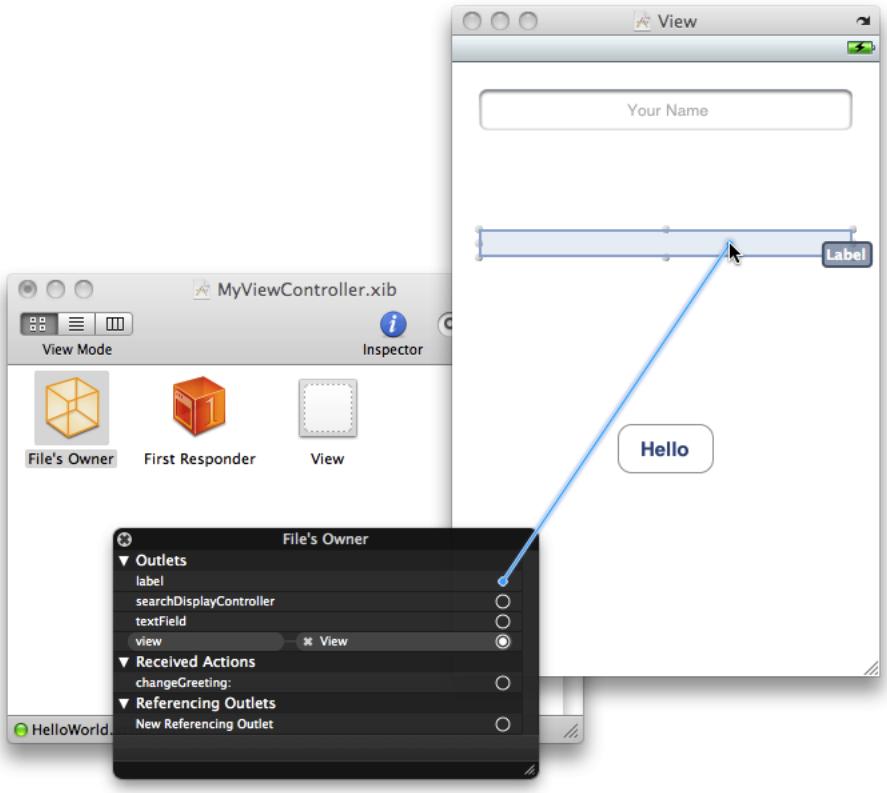
>> Save the file.

Making Connections

Now that you've defined the view controller's outlets and action, you can establish the connections in the nib file.

>> Connect the File's Owner's `label` outlet. Control click File's Owner to display a translucent panel that shows all the available outlets and actions, then drag from the circle to the right of the list to the destination to make the connection.

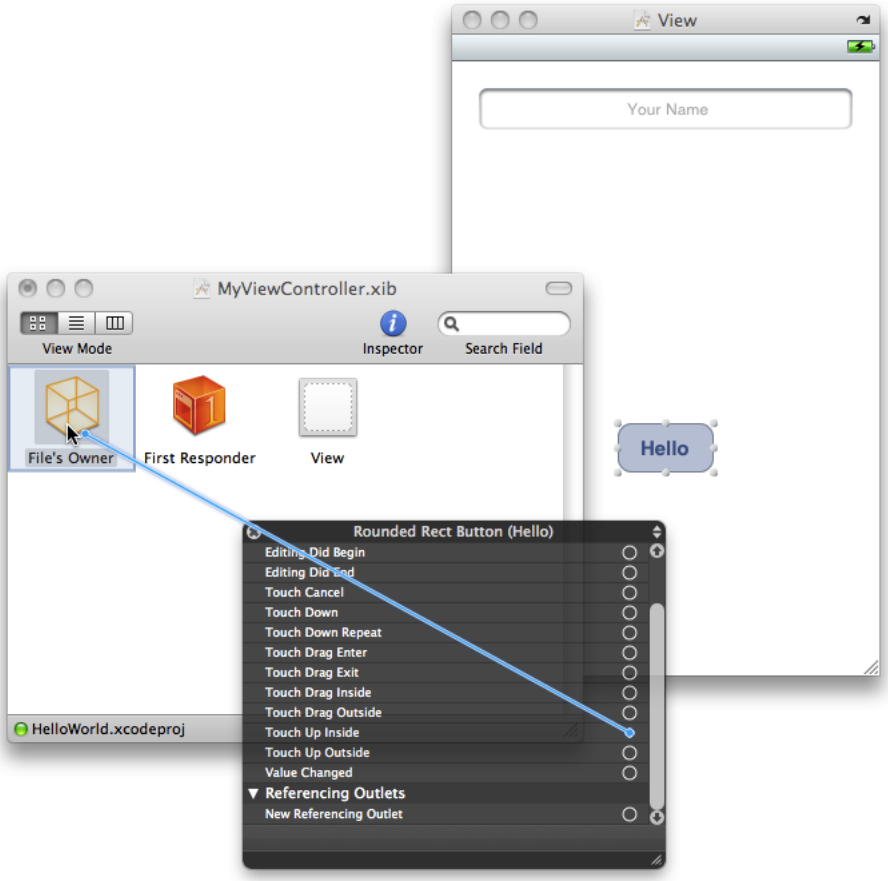
You can resize the panel to show more or fewer outlets and actions at a time by dragging the resize handle at the bottom right of the panel. If there is not sufficient space to show all the outlets and actions, the panel displays a scroller to allow you to navigate within the list view.

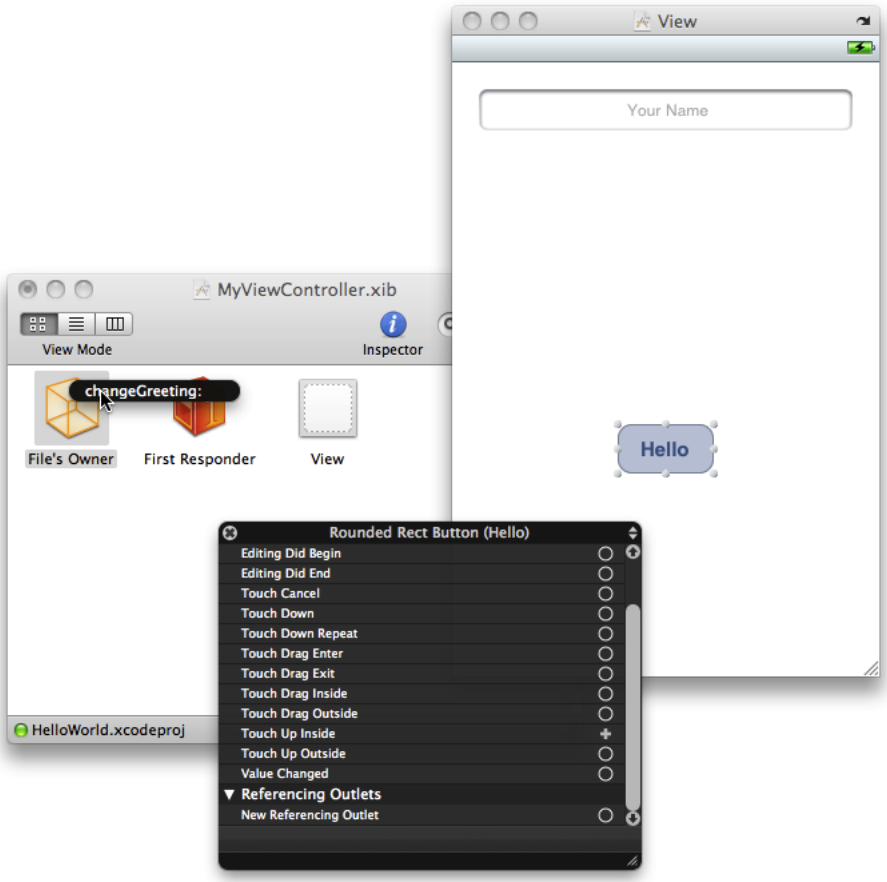


Interface Builder will not allow you to make connections to the wrong sort of element. For example, since you declared the `label` property to be an instance of `UILabel`, you cannot connect the `label` outlet to the text field.

>> Connect the File's Owner's `textField` outlet to the text field in the same way you connected the label.

>> Set the button's action method by Control-clicking the button to show the inspector, then dragging from the open circle in the Touch Up Inside Events list to the File's Owner icon and selecting `changeGreeting:` in the translucent panel that appears over File's Owner (you may have to scroll within the inspector to reveal the Touch Up Inside connection).





This means that when you run the application, when you lift your finger inside the button the button sends a `changeGreeting:` message to the File's Owner object. (For a definition of all the control events, see `UIControl`.)

The text field sends a message to its delegate when the user taps the Return button in the keyboard. You can use this callback to dismiss the keyboard (see "The Text Field's Delegate" (page 42)).

>> Set the text field's delegate to be the File's Owner (the view controller) by Control-dragging from the text field to the File's Owner and selecting `delegate` in the translucent panel that appears.

Testing

You can now test the application.

>> Building and running the project.

There will be several compiler warnings because you haven't yet implemented accessor methods for the properties—you'll fix these in the next chapter. You should find, though, that the button works (it highlights when you tap it). You should also find that if you touch in the text field, the keyboard appears and you enter text. There is, however, no way to dismiss the keyboard. To do that, you have to implement the relevant delegate method. You'll also do that in the next chapter.

Recap

You added instance variables and property declarations, and a declaration for the action method, to the view controller class interface. You added a stub implementation of the action method to the class implementation. You also configured the nib file.

Implementing the View Controller

There are several parts to implementing the view controller. You need to deal with the instance variables—including memory management—implement the `changeGreeting:` method, and ensure that the keyboard is dismissed when the user taps Done.

The Properties

You first need to tell the compiler to synthesize the accessor methods.

>> In the `MyViewController.m` file, add the following after the `@implementation MyViewController` line:

```
@synthesize textField;
@synthesize label;
@synthesize string;
```

This tells the compiler to synthesize accessor methods for these properties according to the specification you gave in the interface file. For example, the declaration of the `string` property is `@property (nonatomic, copy) NSString *string;`), so the compiler generates two accessor methods: `- (NSString *)string` and `- (void)setString:(NSString *)newString`. In the `setString:` method a copy is made of the string that's passed in. This is useful to ensure **encapsulation** (the passed-in string might be mutable—you want to make sure that the controller maintains its own copy). For more about encapsulation, see "Mechanisms Of Abstraction" in *Object-Oriented Programming with Objective-C*.

You must relinquish ownership in the `dealloc` method because all of the property declarations specify that the view controller owns the instance variables (`retain` and `copy` imply ownership, see "Memory Management Rules" in *Memory Management Programming Guide*).

>> In the `MyViewController.m` file, update the `dealloc` method to release the instance variables before invoking super's implementation:

```
- (void)dealloc {
    [textField release];
    [label release];
    [string release];
    [super dealloc];
}
```

The changeGreeting: Method

When it's tapped, the button sends a `changeGreeting:` message to the view controller. The view controller then should retrieve the string from the text field and update the label appropriately.

>> In the `MyViewController.m` file, complete the implementation of the `changeGreeting:` method as follows:

```
- (IBAction)changeGreeting:(id)sender {

    self.string = textField.text;

    NSString *nameString = string;
    if ([nameString length] == 0) {
        nameString = @"World";
    }
    NSString *greeting = [[NSString alloc] initWithFormat:@"Hello, %@!",
nameString];
    label.text = greeting;
    [greeting release];
}
```

There are several pieces to this method.

- `self.string = textField.text;`

  This retrieves the text from the text field and sets the controller's `string` instance variable to the result.

  In this case, you don't actually use the string instance variable anywhere else, but it's important to understand its role. It's the very simple model object that the view controller is managing. In general, the controller should maintain information about application data in its own model objects—application data shouldn't be *stored* in user interface elements.

- `@"World"` is a string constant represented by an instance of `NSString`.
- The `initWithFormat:` method creates a new string that follows the format specified by the format string, like the `printf` function. `%@` indicates that a string object should be substituted. To learn more about strings, see *String Programming Guide*.

The Text Field's Delegate

If you build and run the application, you should find that if you tap the button, the label shows "Hello, World!" If you select the text field and start typing, though, you should find that you have no way to indicate that you've finished entering text and dismiss the keyboard.

In an iOS application, the keyboard is shown automatically when an element that allows text entry becomes the first responder, and is dismissed automatically when the element loses first responder status. (You can learn more about first responder by reading "Event Handling" in *iOS Application Programming Guide*.) There's no way to directly message the keyboard; you can, however, make it appear or disappear as a side-effect of toggling the first responder status of a text-entry element.

In this application, the text field becomes first responder—and so the keyboard appears—when the user taps in the text field. You want the keyboard to disappear when the user taps the Done button on the keyboard.

The `UITextFieldDelegate` protocol includes a method, `textFieldShouldReturn:`, that the text field calls whenever the user taps the Return button (whatever text the button shows). Because you set the view controller as the text field's delegate ("Making Connections" (page 37)), you can implement this method to force the text field to lose first responder status by sending it the `resignFirstResponder` message—which has the side-effect of dismissing the keyboard.

>> In the `MyViewController.m` file, implement the `textFieldShouldReturn:` method as follows:

```
- (BOOL)textFieldShouldReturn:(UITextField *)theTextField {
    if (theTextField == textField) {
        [textField resignFirstResponder];
    }
    return YES;
}
```

In this application, it's not really necessary to include the `theTextField == textField` test since there's only one text field. It's worth pointing out the pattern, though, since there may be occasions when your object is the delegate of more than one object of the same type and you may need to differentiate between them.

>> Build and run the application; it should behave as you expect. (Tap Done to dismiss the keyboard when you have entered your name, then tap the Hello button to display "Hello, <Your Name>!" in the label.)

If the application doesn't behave as you expect, you need to troubleshoot (see "Troubleshooting" (page 45)).

Recap

You finished the implementation of the view controller and so completed your first iOS application. Congratulations.

Take a moment to think about how the view controller fits into the overall application architecture. You're likely to use view controllers in most iOS applications you write.

Then take a break, and start to think about what you should do next.

Troubleshooting

This section describes some approaches to solving common problems that you may encounter.

Code and Compiler Warnings

If things aren't working as they should, first compare your code with the complete listings as given in "Code Listings" (page 49).

Your code should compile without any warnings. Objective-C is a very flexible language, and so sometimes the most you get from the compiler is a warning. Typically you should treat warnings as very likely to be errors.

Check Connections in the Nib Files

As a developer, if things don't work correctly, your natural instinct is to check your source for bugs. Cocoa adds another dimension. Much of your application's configuration may be "encoded" in the nib files. And if you haven't made the correct connections, then your application won't behave as you expect.

If the text doesn't update when you tap the button, it might be that you didn't connect the button's action to the view controller, or connect the view controller's outlets to the text field or label.

If the keyboard does not disappear when you tap Done, you may not have connected the text field's delegate or connected the view controller's `textField` outlet to the text field (see "Making Connections" (page 37)). If you have connected the delegate, there may be a more subtle problem (see "Delegate Method Names" (page 45)).

Delegate Method Names

A common mistake with delegates is to misspell the delegate method name. Even if you've set the delegate object correctly, if the delegate doesn't implement the method with exactly the right name, it won't be invoked. It's usually best to copy and paste delegate method declarations from the documentation.

Where to Next?

This chapter offers suggestions as to what directions you should take next in learning about iOS development.

The User Interface

In this tutorial, you created a very simple iOS application. Cocoa Touch offers a rich development environment, though, and you've only scratched the surface. From here, you should explore further. Start with this application. As noted in the first chapter, the user interface is critical to a successful iOS application. Try to improve the user interface. Add images and color to the elements. Add a background image and an icon for the application. Look at the inspectors in Interface Builder to see how else you can configure elements.

Many iPhone applications support multiple orientations; applications for iPad should support all orientations. Make sure the view controller supports multiple orientations (see the `shouldAutorotateToInterfaceOrientation:` method), then (in Interface Builder) adjust your user interface to make sure that all the user interface elements are properly positioned if the view is rotated.

Creating User Interface Elements Programmatically

In the tutorial, you created the user interface using Interface Builder. Interface Builder allows you to assemble user interface components quickly and easily. Sometimes, however, you may want—or need—to create user interface elements in code (for example, if you create a custom table view cell you typically create and lay out the subviews programmatically).

First, open the `MyViewController` nib file and remove the text field from view.

If you want to create the entire view hierarchy for a view controller in code, you override `loadView`. In this case, however, you want to load the nib file then perform additional configuration (add another view). You therefore override `viewDidLoad` instead. (The `viewDidLoad` method gives you a common override point you can use whether you load the main view using a nib file or by overriding `loadView`.)

In `MyViewController.m`, add the following implementation of `viewDidLoad`:

```
- (void)viewDidLoad {

    CGRect frame = CGRectMake(20.0, 68.0, 280.0, 31.0);
    UITextField *aTextField = [[UITextField alloc] initWithFrame:frame];
    self.textField = aTextField;
    [aTextField release];

    textField.textAlignment = UITextAlignmentCenter;
    textField.borderStyle = UITextBorderStyleRoundedRect;

    textField.autocapitalizationType = UITextAutocapitalizationTypeWords;
```

```
    textField.keyboardType = UIKeyboardTypeDefault;
    textField.returnKeyType = UIReturnKeyDone;
    textField.delegate = self;
    [self.view addSubview:textField];
}
```

Notice that there's quite a lot of code compared with how easy it was to create and configure the text field in Interface Builder.

Build and run the application. Make sure it behaves as it did before.

Installing on a Device

If you have a suitable device connected to your computer via its 30-pin USB cable, and you have a valid certificate from the iOS Developer Program, set the active SDK for your project to "iPhone Device" (instead of "iPhone Simulator") and build and run the project. Assuming your code compiles successfully, Xcode then automatically uploads your application to your device. For more details, see *iOS Development Guide*.

Additional Functionality

Next you can try expanding on the functionality. There are many directions in which you can go:

- Rather than using a view as a canvas on which to drop prebuilt user interface controls, you might try writing a custom view that draws its own content or responds to touch events. For inspiration, look at examples such as *MoveMe* and *Metronome*.
- Although you used Interface Builder to layout the user interface for this application, many applications actually use table views to lay out the interface. This makes it easy to create an interface that extends beyond the bounds of the screen—allowing the user to easily scroll to reveal additional elements. You should first investigate how to create a simple list using a table view. There are several sample code projects that you can look at—including *TableViewSuite*—then you can create your own.
- Navigation controllers and tab bar controllers provide an architecture that allow you to create drill-down style interfaces and let the user select different views in your application. Navigation controllers often work in conjunction with table views, but both navigation controllers and tab bar controllers work together with view controllers. Have a look at some of the sample applications—such as *SimpleDrillDown*—that use navigation controllers and expand on them to create your own applications.
- You can often increase the size of your potential marketplace by localizing your application. Internationalization is the process of making your application localizable. To learn more about internationalization, read *Internationalization Programming Topics*.
- Performance is critical to good user experience on iOS. You should learn to use the various performance tools provided with Mac OS X—in particular Instruments—to tune your application so that it minimizes its resource requirements.

The most important thing is to try out new ideas and to experiment—there are many code samples you can look at for inspiration, and the documentation will help you to understand concepts and programming interfaces.

Code Listings

This appendix provides listings for the two classes you define. The listings don't show comments and other method implementations from the file templates.

HelloWorldAppDelegate

The header file: HelloWorldAppDelegate.h

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

@class MyViewController;

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    MyViewController *myViewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) MyViewController *myViewController;

@end
```

The implementation file: HelloWorldAppDelegate.m

```
#import "MyViewController.h"
#import "HelloWorldAppDelegate.h"

@implementation HelloWorldAppDelegate

@synthesize window;
@synthesize myViewController;


- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    MyViewController *aViewController = [[MyViewController alloc]
              initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
    [self setMyViewController:aViewController];
    [aViewController release];

    UIView *controllersView = [myViewController view];
    [window addSubview:controllersView];
    [window makeKeyAndVisible];
```

```
}


- (void)dealloc {
    [myViewController release];
    [window release];
    [super dealloc];
}

@end
```

MyViewController

The header file: MyViewController.h

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

@interface MyViewController : UIViewController <UITextFieldDelegate> {
    UITextField *textField;
    UILabel *label;
    NSString *string;
}

@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, copy) NSString *string;

- (IBAction)changeGreeting:(id)sender;

@end
```

The implementation file: MyViewController.m

```
#import "MyViewController.h"

@implementation MyViewController

@synthesize textField;
@synthesize label;
@synthesize string;

- (IBAction)changeGreeting:(id)sender {

    self.string = textField.text;

    NSString *nameString = string;
    if ([nameString length] == 0) {
        nameString = @"World";
    }
    NSString *greeting = [[NSString alloc] initWithFormat:@"Hello, %@!",
nameString];
    label.text = greeting;
```

```
    [greeting release];
}

- (BOOL)textFieldShouldReturn:(UITextField *)theTextField {
    if (theTextField == textField) {
        [textField resignFirstResponder];
    }
    return YES;
}

- (void)dealloc {
    [textField release];
    [label release];
    [string release];
    [super dealloc];
}

// Other methods from the template omitted
@end
```

Document Revision History

This table describes the changes to *Your First iOS Application*.

Date	Notes
2010-07-01	Updated for iOS 4.
2010-03-15	Updated for iPhone OS 3.2.
2009-10-08	Emphasized need to connect text field in Interface Builder.
2009-08-10	Removed a reference to dot syntax for an example that used an accessor method directly.
2009-06-15	Updated for iPhone OS 3.0.
2009-01-06	Corrected typographical errors.
2008-10-15	Clarified description of the target-action design pattern.
2008-09-09	Minor corrections and clarifications.
2008-06-09	New document that introduces application development for iPhone.

Document Revision History