Developing iOS 5 Applications

Laura J. White & Janusz Chudzynski

ePress+

iBooks Author

Copyright

About the Authors

Laura J. White *is an Assistant Professor in the Computer Science Department, and Program Director for Software Engineering at the University of West Florida. She has been teaching at the University of West Florida since 1993. She earned a B.S. in Computer Engineering from the University of New Mexico in 1984 and a M.S. in Computer Science from the Naval Postgraduate School in 1989, and a Ph.D. in Instructional Design for Online Learning at Capella University in 2006. She is a retired Surface Warfare Officer from the U.S. Navy. Laura has coauthored a book chapter on assessment, and has written numerous conference and journal publications regarding teaching programming, online learning, and software oriented architecture. She attended the Worldwide Developers Conference (WWDC) in 2010 and has been enthusiastically developing apps and teaching iPhone programming at the University of West Florida ever since.*

Janusz Chudzynski *is a Software Engineer and iPhone Programmer for the Academic Technology Center at the University of West Florida. He has earned a M.S. in Computer Science at the University of West Florida, and a M.S. in Management and Marketing with a minor in Information Systems from Warsaw University Technology in 2006. Janusz started learning iOS programming on his own and developed a few apps while he was a student, then received an Apple Scholarship to attend the Worldwide Developers Conference (WWDC) in 2009. Since then he has been enthusiastically developing apps and has taught several iPhone Programming courses for the University of West Florida.*

Dedication

Many people touch our lives as we journey through life.
We dedicate this book to a few of those who have been most influential in our lives.

Leszek and Iwona Chudzynski, Janusz's parents who showed him how to move forward in life despite all adversities, and without whose support Janusz would not be in the place that he is now.

Jiyeon Song, a very special person in Janusz's life.

In memory of Leslie A. White, Laura's twin, who was an extremely kind, generous, and creative person, with an adventurous spirit, who loved music, cats, and games. And, in memory of Mona L. White, Laura's mother who is greatly missed.

Barbara S. White, Laura's aunt who courage and grace is unmatched, and who has always provided love and support to Laura in times of joy and sorrow.

Laura's close friends, Ruth Edwards and Marcia Holland, and many friends and colleagues at the University of West Florida--especially those who regularly attend Saturday afternoon clubhouse meetings.

Preface

Since the App Store was launched in 2008, millions of apps have been created and distributed by iOS developers for the iPod Touch, the iPhone, and the iPad. This book is intended to get new iOS developers started creating apps of their own for the App Store.

The basic concepts needed to get started developing apps are presented within the context of three App development projects. The authors have developed many apps that are available for download from the iTunes App Store, and it is their belief that learning to develop apps within the context of projects will provide a satisfying and rewarding learning experience. Readers are encouraged to review relevant resources provided at Apple's developer website to deepen their understanding of the concepts integrated into each project. A companion website is available at http://www.epressplus.com/2012/01/21/developing-ios5-applications/ for the download of content needed for these projects and for support related to this book.

Chapter 1

Introduction to iOS Programming

```
//
//  Developing iOS 5 Applications
//
//  Created by Laura J. White and Janusz Chudzynski
//  Copyright (c) 2012. All rights reserved.
//

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISwitch *iPhone;
@property (weak, nonatomic) IBOutlet UISlider *iPad;

- (IBAction)learniOS5Programming:(id)sender;

@end
```

Introduction to iOS Programming

Concepts emphasized in this chapter:

- *History of iPhone/iPad programming*
- *Features and characteristics of the iPhone and the iPad*
- *Developing apps for the iPhone and the iPad*
- *The Objective-C programming language*
- *Overview of projects in this book*
- *Deploying apps*

History of iPhone/iPad Programming

The first iPhone was introduced January 9, 2007, and then released June 29, 2007. This iPhone became known as the original iPhone. The iPhone was the first smart phone with a touch screen display and integrated message texting, email, music, camera, photos, video, and internet access, as well as third-party applications referred to as apps. The next version of the iPhone was the 3G, released in July 11, 2008. The iPhone 3GS was released in June 19, 2009. The fourth generation of the iPhone was the iPhone 4 that was released on June 24, 2010. The iPhone 4S is being released on October 14, 2011. The iPhone has become so popular that over 1.7 million iPhone 4 devices were sold over the first three days after its release.

In November 2009, Apple announced that there were 1 million apps created by developers available in the App store and at that time, over 2 billion apps had been downloaded from the App store. Only apps that have been reviewed and approved by Apple are made available in the App store. The Apple review and approval policy is somewhat controversial. The advantage is that Apple is able to maintain quality control over what is made available to users, the disadvantage is that developers must comply with standards that some consider constrain creative freedom. On January 27, 2010 the iPad was introduced. It was then released April 3, 2010. The iPad was not Apple's first tablet computer, however it was the first to use the same multitouch OS that was already hugely popular by users of iPods and iPhones. The iPad released with a 9.7 inch LCD display with fingerprint

and scratch resistant glass. There were 300,000 iPads sold on the first day of its release, and 80 days later on June 21, 2010 3,000,000 iPads had been sold.

By, April 2010 when Apple announced the release of the iPad, 50 million iPhones and 35 million iPod Touches had been sold to date. These numbers indicate that many programmers are successfully developing and deploying apps. This book is intended to help those with the desire to join the ranks of iPhone and iPad developers by learning the essential concepts and developing the skills necessary to successfully develop and deploy iPhone and iPad apps.

Features and Characteristics of the iPhone and the iPad

There are many essential features and characteristics of iPhone and iPad apps. Some characteristics are required and must be satisfied in every app that is made available in the App store. The iPhone Human Interface Guidelines provides fundamental human interface design principles that should be incorporated into all iPhone apps and describes the user interface components that can be used in the development of an app and provides guidance in using these components effectively.

The iPad Human Interface Guidelines generally supplement, but occasionally supercede the guidelines provided by the iPhone Human Interface Guidelines for apps that are specifically developed for iPads. These guidelines contain four chapters (a) Key iPad Features and Characteristics, (b) From iPhone Applications to iPad Applications, (c) iPad User Experience Guidelines, and (d) iPad UI Element Guidelines.

Device characteristics specific to the iPhone are that the iPhone has a compact screen size and has a default display orientation that is portrait orientation.

Device characteristics common to both the iPhone and the iPad that should be considered when developing apps are that (a) memory is limited, (b) people see one screen at a time, (c) people interact with one application at a time, (d) onscreen user help is minimal, (e) orientation can change, (f) applications respond to manual gestures rather than mouse clicks, (g) preferences are available in the Settings application, and (h) artwork has a standard bit depth and PNG format is recommended.

Device characteristics specific to the iPad are that (a) the iPad has a screen size of 1024 by 768 pixels, (b) there is no default orientation, (c) there are user interface elements available for the iPad that are not available for the iPhone such as split views and popup views, and (d) options are available to plug the iPad into an external keyboard.

Developing Apps for the iPhone and the iPad

The iPhone is driven by the iOS, Apple's mobile operating system that is also used on the iPod Touch and the iPad. The iOS is what provides the capability for users to interact with these devices through the multi-touch screen.

Third-party applications that can be installed and used on multi-touch screen devices running iOS are call apps. These apps are created by developers using Apple's iPhone software development kit (SDK). The iOS SDK includes Xcode, a collection of development tools. Within Xcode is the Xcode integrated development environment (IDE), Interface Builder, Objective-C, Simulators for iPhone and iPad devices, and Organizer.

The Xcode IDE provides a collection of tools that support the construction, editing, compiling, execution, and debugging of source code within a single window. The IDE also provides the capability to manage testing devices and packaging iPhone apps.

Interface Builder provides capabilities for designing and developing graphical user interfaces for applications that run on multi-touch screen devices. Xcode and Interface Builder are very tightly integrated; changes made in Interface Builder are automatically synchronized with the source code files in Xcode. Early development of apps required the use of Interface Builder and Xcode as two separate applications. Since the release of Xcode 4, Interface Builder has been integrated wholly within Xcode to the extent that it is virtually transparent to new developers.

Objective-C is an object-oriented programming language that is a superset of the standard C programming language. Objective-C is extremely fast and powerful with a flexible dynamic class system. The Cocoa and Cocoa Touch frameworks with high-level application programming interfaces (APIs).

The simulators run iPhone/iPad apps in the Mac OS environment so that developers can see how apps will appear and perform on a device from within the development environment without having to actually install apps on devices while they are being developed. The iPhone simulator can be launched by clicking on the Build and Run icon from within Xcode, the simulator will appear similar to that shown in Figure 1.

Figure 1 iPhone Simulator

The iPad simulator provides the developer with the capability to build and run iPad applications in the Mac OS environment. When building and running an iPad application within Xcode, the app will appear similar to Figure 2.

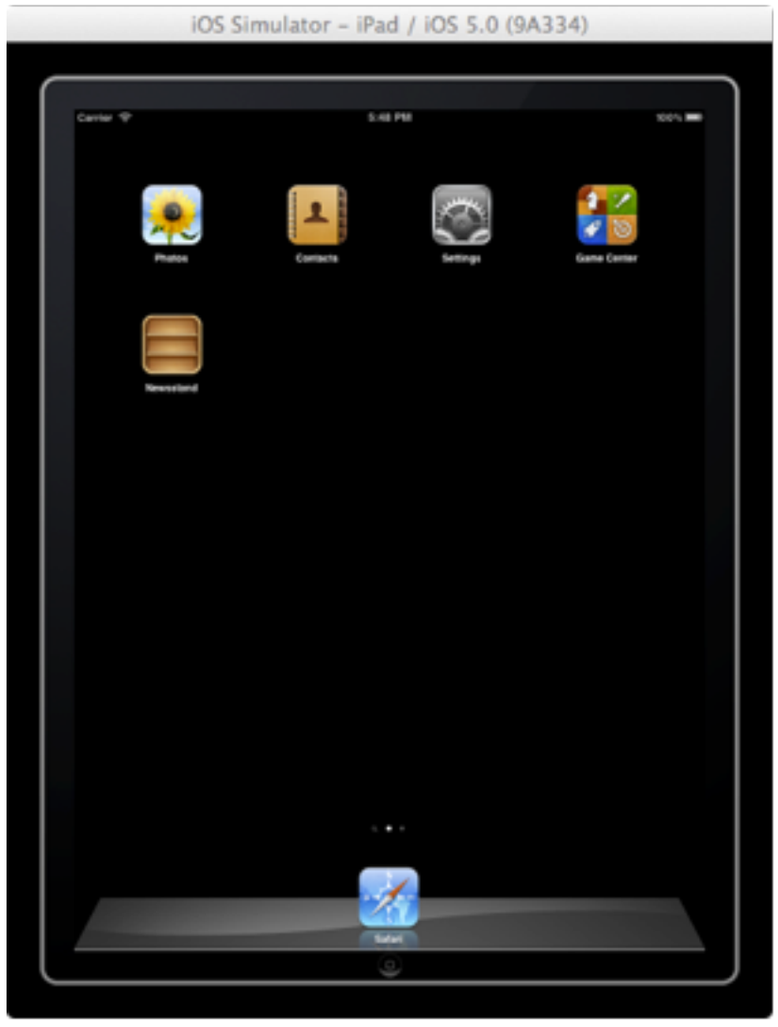

Figure 2 iPad Simulator

The Organizer provides the means to keep track of different iPhone devices and certificates. Organizer is also used when provisioning apps for the iTunes App store.

A wealth of resources and support for the development of iPhone apps is available from Apple in the iPhone Dev Center at http://developer.apple.com/devcenter/ios. The iOS Dev Center contains *Getting Started* documents and videos, Coding How-To's, SDK download links, and Featured Content based on current developer needs.

Apps for the iPhone and the iPad can be developed several ways. They can be developed entirely within Xcode—Apple's integrated development environment using the Objective-C programming language, or they can be developed using Xcode and Interface Builder in tandem where the user interface elements for an app are developed in Interface Builder and then defined inside Xcode. Generally, a new project is created within Xcode. When creating a new application the developer specifies the product as either an iPhone or an iPad application. Apps developed for the iPhone will run on the iPad in compatibility mode but will not provide the rich user experience that apps specifically designed for the iPad will generally provide. Apps developed for the iPhone will not fill the iPad screen since the size of the iPhone screen is 480 by 320 pixels, and the size of the iPad screen is 1024 by 768 pixels.

Apple has incorporated a feature on the iPad that provides the user with the option of doubling the size of the app if it detects that the resolution of the app is that of an iPhone.

It is also possible to run an iPad app in the iPhone simulator but the display of the app will be cut off since the iPhone screen is smaller than the iPad screen.

The Objective-C Programming Language

The Objective-C programming language is an objective-oriented programming language that underlies all iOS and Mac OS applications. The initial development of this language began in the early 1980s as a superset of the C programming language and has evolved over time. The Objective-C programming language has extensive libraries that support the development of high quality applications. To developed sophisticated applications, iOS developers must become familiar with the basics of the Objective-C programming language. Developers with a prior knowledge of C, C++, or Java will recognize many of the basic features of the Objective-C programming language.

Variables and Objects

Objective-C contains both primitive data types and object-oriented classes that can be used to create primitive variables and objects. Some of the most commonly used primitive types are:

int	Variables used to store/manipulate whole numbers
float	Variables used to store/manipulate real numbers
BOOL	Variables that only have to possible values such as true/false or YES/NO

A few examples of declarations for these primitive types are as follows:

```
int WholeNumber;
float realNumber;
BOOL isHappy;
```

Objects in Objective-C—as with other object oriented programming languages—are instances of classes that encapsulate attributes and behavior through methods defined in the classes. In Objective-C objects are referenced through pointers so when declared the asterisk is used to indicate that the variable is a pointer to an object of a particular class. The following declaration creates a pointer named myEmployee to an instance of the Employee class.

```
Employee *myEmployee;
```

Once the pointer is created, memory must be allocated for the object and its contents initialized, as in the following statement, which allocates and initializes memory for the instance of Employee that we just declared.

```
myEmployee=[[Employee alloc] init];
```

Operators

Objective-C contains the operators: assignment, arithmetic, modulo, combined assignment with arithmetic, increment, decrement, bitwise Boolean, and logical Boolean operators that are found in C, C++, and Java.

Decisions

Decision constructs in Objective-C include those common to C, C++, and Java. These constructs also use the use basically the same syntax as in the other languages.

Iteration

Objective-C contains four basic constructs for iteration. The while loop is intended for situations where iteration is needed zero or more times. The do while loop is intended for situations where iteration is needed one or more times. The for loop is intended for situations where iteration is intended zero or more times and the processing of an element is based on some increments of a variable. These iteration constructs are similar to those provided n C, C++, and Java.

Methods

This section will cover basic information about methods. Let's have a look at their syntax of a method declaration as shown in Figure 5. The + or – sign at the front of the definition indicates the type of the method. A + indicates that the method is a factory or class method and a – indicates that the method is an instance method. The type indicator is followed by a return type within parentheses. The return type in Figure 5 is void which means that the method does not return any value. The return type is followed by the name of the method. The name of the method in Figure 1.5 is insertObject. If the method does not require any arguments then the name of the method would be followed by a semicolon to denote the end of the method definition. If the method does require arguments as is shown in Figure 1.5 then the name of the method is followed by a colon and then an argument list of one or more arguments. Each argument in the list consists of an argument type in parenthesis and a name of the argument. A semicolon at the end of the list indicates the end of the method definition.

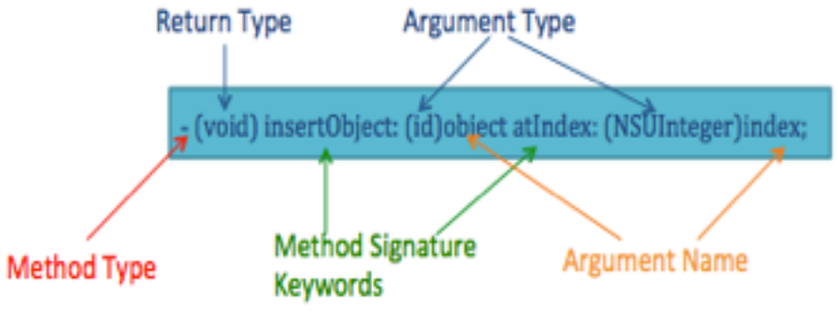

Figure 3 Method Declaration

Now let's look at the method type more closely. A class method is one that performs some operation on the class itself, such as creating a new instance of the class. The stringWithFormat method definition follows is an example of a class method.

```
+(NSString *) stringWithFormat;
```

An instance method is one that performs some operation on a particular instance of a class, such as setting its value, retrieving its value, displaying its value. The drive method definition that follows is an example of an instance method. Notice that the name of the class is not evident in the definition.

```
-(void) drive;
```

Next we look at invoking methods in Objective-C. Objective-C uses a message paradigm for invoking methods that stems from the Smalltalk programming language where a method is invoked by sending a message to it. The syntax for sending a message to a method is to enclose a class name or instance name followed by the name of the method within square brackets. For example, imagine that the class Ford has a class method named create that returns an object of class id. The method definition is +(id) create; and we send a message to it with [Ford create];

The Ford class can also have instance methods that are all named drive but with different argument lists.

```
-(void) drive;
-(void) drive: (float) speed;
-(BOOL) drive: (float) speed from:(NSString *) city1 to (NSString *) city2;
```

We can invoke these instance methods as follows.

```
[ford  drive];
[ford  drive:    50.0];
[ford  drive:    50.0 from: Destin to: Pensacola];
```

Creating Objects

Since we have two different types of methods – class and instance – there are two different ways to create objects which are instances of classes. Figure 6 shows how a new object or instance of the NSString class named string can be created. In both cases we start creating an object by providing the name of the class that the object will belong to, NSString and then the name of the object preceded with an asterisk that denotes that the name is actually a reference or pointer to the object. If we wish to create the object using a class method then we enclose the name of the class and the name of the object within square brackets and then allocation and initialization of the object is handled by the class. If we use instance

methods to create an object we enclose the class and alloc method within the square brackets, and then that message is enclosed within another message that invokes the init method for the class on the object just allocated.

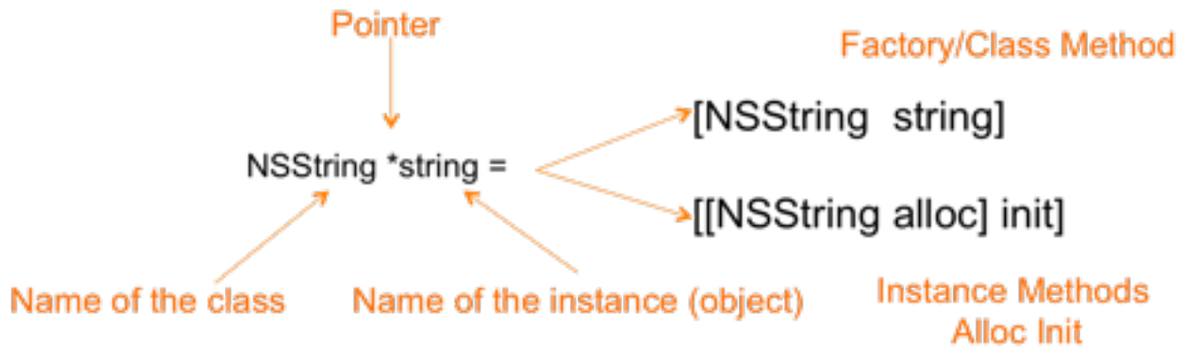

Figure 4 Using Instance and Class Methods to Create an Object

Creating an object using a class method is easier but the trade-off is that the programmer will have less control over the scope of the variable with regard to memory management than if the object is created using the alloc and init instance methods.

Objective-C Classes

In Objective-C classes are structured so that the interface for the class is defined within a .h file and the implementation for the class within a .m file. The interface contains declarations and definitions of variables and methods and links the class to its superclass. The implementation contains the method implementations--that is methods bodies with code to perform the functionality of the method. For example, the Rectangle class would be defined within the Rectangle.h file and implemented within the Rectangle.m file.

The interface file must be *included* or *imported* in any source module that depends on the class interface. For example, the first two lines in the code sample that follows are comments, and the third line imports the interface for Factory inside the Ford class. Interface files can be imported using the keywords include or import. The advantage of using import is that this ensures that the interface isn't included more than once during compilation. This is especially beneficial in large applications that consist of many classes.

The next line in the code sample presents the new class name and links it to its superclass, so in this case the class name is Ford and the superclass is NSObject. The superclass defines the position of the new class in the inheritance hierarchy. Following the first part of the interface

directive are braces that enclose declarations of instance variables for the class that can be accessed anywhere within the scope of the class. Three methods for the class are defined next, and the last element in the interface file is the end directive.

```
// Ford.h
// Interface file for the Ford class

#import "Factory.h"

@interface Ford : NSObject {
    NSString *color;
    NSString *dealer;
    int vinNumber;
}

//method definitions
-(void) repaintCar:(NSString *) newColor;
-(void) moveToDealer:(NSString *) newDealer;
-(id) initWithColor:(NSString *) initColor;
@end
```

The implementation for a class is structured very much like its interface. The next code sample contains the implementation for the Ford class. It begins with an import directive for the Ford.h file. Implementation files must import their own interface files, and can also import or include other classes as necessary that were not already imported or included in this class's interface file. The next line of code is the implementation directive. Since it imported its own interface file it does not need to include declarations and definitions already provided there. Then the method bodies for the three methods declared in the Ford.h file follow.

```
// Ford.m
// Implementation file for the Ford class

#import "Ford.h"
@implementation Ford

-(void) repaintCar: (NSString *) newColor{
    // Code to implement this behavior
}

-(void) moveToDealer: (NSString *) newDealer{
```

```
    // Code to implement this behavior
}

-(id) initWithColor: (NSString *) initColor{
    self=[super init];
    if(self) {
        // Code to implement this behavior
    }
    return self;
}

@end
```

Special Keywords

Objective-C contains some special keywords. The most common are self, super, and id. The keyword self searches for a method implementation in the usual manner, starting in the current class. The keyword super starts the search for a method implementation in the superclass of the class where super appears. The id keyword can be used as a generic type. For example if we have a method that returns a type of id, then any object can be returned. If an method argument has a type of id then any object can be passed to the method.

Format Specifiers

Objective-C contains the means to format strings and numbers through the combination of class methods and format specifiers. Consider the declaration for a string.

```
NSString *example = [NSString stringWithFormat:@"exampleText"];
```

In this example no special formatting of the string is needed, so even though we used the stringWithFormat method to create the string we simply put the text inside the double quotes. However, when we want to specify some formatting we can use various format specifiers—the three most common are:

%f for floats, and doubles

%d for integers

%@ for strings and other objects.

To use format specifiers we add them inside the double quotes, and then type a comma after the double quotes and then specify the variable to be formatted. To create a string that stores exampleText followed by the value of an int variable named value, we should use the following statement that includes the format specifier for an integer and a reference to the variable value.

```
NSString *example1 = [NSString stringWithFormat:@"exampleText %d", value];
```

The following statement includes a format specifier to display a float variable rather than an int variable named value:

```
NSString *example2 = [NSString stringWithFormat:@"exampleText %f", value]
```

To specify the minimum number of digits that should be used to display a floating point variable, a number can be added to the format specifier. The following statement displays the variable value using at least 5 places which in which the decimal point counts as one of the places.

```
NSString *example3 = [NSString stringWithFormat:@"exampleText %5f", value];
```

To specify the maximum number of digits that should be displayed after the decimal point for a float variable a decimal point followed by a number can be added to the format specifier. The following statement displays the variable value using at least 5 places with exactly 2 numbers after the decimal point:

```
NSString *example4 = [NSString stringWithFormat:@"exampleText %5.2f", value];
```

To display a string variable object we can use the @ symbol to print an object. The following example contains a string literal and a string variable in the format specifier.

```
NSString *text1 = @"More Text";
NSString *example5 = [NSString stringWithFormat:@"Some text and  %@", text1];
```

We can use multiple format specifiers in a single statement, just remember to list the variables that are associated with each format specifier separated by commas.

```
int num1=0;
float num2=10.0;
NSString *example6 = [NSString stringWithFormat:@"The numbers are  %d and %4.1f", num1, num2;
```

Memory Management

One of the new features contained in Objective-C for iOS 5 is automatic reference counting (ARC). We will use ARC in our projects. However, developers working with iOS 4 projects still need to manually manage their memory allocations. Fortunately the rules are not very complicated. First, we need to understand object ownership. Programmers are responsible for managing memory for objects they "own". So when does the programmer become the owner of an object? The programmer owns an object if it is:

- Created using an instance method (alloc-init)
- Created using the keyword new
- Called with the retain message
- Called using the copy message

Proper memory management requires that programmers relinquish ownership of objects they own when they are finished with them. Ownership of objects is relinquished by sending a release message or an autorelease message to it. You should not relinquish ownership of an object you do not own. A few simple examples of both proper and improper memory management are contained in the code sample below.

```
-(void) memoryTest {
    NSString* mem1=[[NSString alloc]initWithString:@"Some string"];
    NSMutableArray* array1=[NSMutableArray arrayWithCapacity:0];
    [array addObject:mem1];

    //Do something with the array here…
    [array1 release];   // Incorrect since it was created with a class method
    [mem1 release];  // Correct
}
```

Now let's look at another example to see how to manage memory for variables that are declared inside the interface part of a class. For this example we will use the Ford class again. As shown in the interface below, the class contains declarations for three instance variables: color, dealer, and vinNumber, along with three methods.

```
// Ford.h
// Interface file for the Ford class
```

```
#import "Factory.h"
@interface Ford : NSObject {
    NSString *color;
    NSString *dealer;
    int vinNumber;
}

//method declarations
-(void) repaintCar:(NSString *) newColor;
-(void) moveToDealer:(NSString *) newDealer;
-(id) init;
@end
```

Now let's look at the implementation file for this class. In the init method in the Ford.m file, the initial values of dealer, color and vinNumber are set. Since we need these objects outside the scope of the init method that allocates and initializes them we shouldn't release them inside the init method. Instead, we release them in the dealloc method. The role of the dealloc method is to free an object's own memory, and dispose of any resources it holds, and to relinquish ownership of instance variables. The dealloc method is invoked automatically in iOS applications – it should not be invoked or called in the code. Notice that the instance variable color is released in the dealloc method and not the other instance variables. We don't need to release vinNumber because it is a primitive variable, and we do not need to release dealer because it was created through the use of a class method and memory management is handled by the system for instances created by class methods.

```
// Ford.m
// Implementation file for the Ford class

#import "Ford.h"
@implementation Ford

-(void)repaintCar:(NSString*) newColor{
    // Code to implement this behavior
}

-(void)moveToDealer:(NSString*) newDealer{
    // Code to implement this behavior
}

-(id)init{
    self=[super init];
```

```
    if(self) {
        color=[[NSString alloc]initWithString:@"WHITE"];
        vinNumber=0;
        dealer=[NSString stringWithFormat:@"Toyota"];
    }
    return self;
}

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

@end
```

This section has provided a brief overview of essential aspects of Objective-C. For further details of the language refer to a separate text or to online tutorials and programming guides for the Objective-C programming language.

Overview of Projects in this Book

This book presents an introduction to useful resources for app developers, and then three iPhone development projects. Each project embodies new iPhone programming concepts and provides the opportunity for the developer to implement new features for iPhone apps.

The first project, in chapter 2, is the development of a Temperature Conversion App, in which readers will learn the basics of creating a simple user interface using the UIKit framework, and provided a few basic methods to implement actions for the user interface elements.

The second project is the development of a Photo Puzzle App in chapters 3 and 4. This project further reinforces concepts learned in chapter 2 and also introduces the reader to basic concepts of the Core Graphics framework, animation, modal view controllers, and more complex underlying action than was used in the Temperature Conversion App.

The third project in this book is the development of the Show Me App in chapter 5 that uses the Map Kit in chapter 4 demonstrate basics of using maps within an app.

The fourth project in this book is the development of the Treats! App in chapters 6 and 7. This project includes concepts regarding the use of touch events, timers, picker views, and displaying high scores for a game by saving and restoring data in the app.

The fifth project demonstrates the use of storyboarding, a new feature in iOS 5, to graphically build an app that consists of multiple screens or view controllers.

Deploying Apps

After an iPhone or iPad app performs as desired in the Xcode simulator, the next step is to deploy the app to an actual device. There are several steps involved in this process. In order to deploy apps to personal devices for testing, a developer must have signed up for an iOS standard, enterprise, or university developer program. In order to deploy apps to the iTunes App Store, a developer must have signed up for an iOS standard program. Once a developer has membership in one of these programs the developer can access resources in the iOS member center to obtain certificates and provisioning profiles necessary to deploy apps to the iTunes App Store. The member center How-To documents provide step-by-step instructions for submitting an app to the iTunes App Store.

Summary

Since the App Store was launched in 2008, millions of apps have been created and distributed by iOS developers for the iPod Touch, the iPhone, and the iPad. As the capability of iOS devices increases, the demand for those devices and apps for those devices also increases. Developing iOS apps is no longer a niche market for programmers. As the demand for apps dramatically increases so does the number of business that comprehend the value for establishing a mobile presence and interaction with customers via apps.

Features of the various devices—iPod Touch, iPhone, and iPad—that run iOS applications have been described and the distinctions between the devices has been emphasized. A brief introduction to the general process of developing an app using Apple products including Xcode, Interface Builder, and simulators has also been presented. The next part of this chapter presented basic principles of Objective C programming which underlies all iOS applications. This was followed by a brief description of the app development projects that are contained in the remaining chapters of this book. The last part of this chapter provided a brief overview of the process for distributing apps so that once readers have developed their skills and created their own apps they will be able to get started with the process of deploying apps to the iTunes App Store.

Review Questions

1. What app development components are provided by an iOS SDK.

2. Describe high-level characteristics of the Objective-C programming language which is used for the development of iOS apps.

3. What are the two types of methods that are indicated by a + or a - sign at the beginning of a method declaration?

4. Describe how the messaging paradigm relates to methods in Objective-C.

5. What keyword represents a generic type?

6. Explain how the keywords self and super are used.

7. What is the format specifier for an object?

8. List the conditions under which the programmer is responsible for manually managing memory.

9. What is required for a developer to deploy an app to an iOS device or to the iTunes App Store.

Chapter 2

Temperature Converter App: User Interface Controls

iBooks Author

Temperature Converter App: User Interface Controls

Concepts emphasized in this chapter:

- *Adding iOS user interface elements: buttons, labels, text Fields, and events*
- *Delegation patterns*
- *Making connections from the user interface to the code*
- *Adding custom code to implement an application*

Introduction

The first iPhone project will consist of the development of an application that converts Centigrade temperatures to Fahrenheit temperatures, and also converts Fahrenheit temperatures to Centigrade temperatures. Interface builder integrated within Xcode, basic user interface controls, and Xcode will be used in the development of this project. A plan should always be established before starting any programming project. The basic elements of the plan will be described in detail in the following sections of this chapter. The plan for this project will be to:

1. Create a new project in Xcode
2. Add an image to the project
3. Add the necessary user interface elements to the View that represents the screen of an iOS device
4. Create outlets and actions for the user interface elements
5. Add custom code to implement the functionality of the user interface elements
6. Run and test the application

Development

The development of the Temperature Converter application will be conducted in several phases. First, we will create a new project. Then we will build the user interface. Then we will write the custom code to provide the desired functionality for the user interface elements.

Create a New Project in Xcode

Launch Xcode. Xcode is often located at Macintosh HD > Developer > Applications > Xcode. When Xcode launches, it presents a window that displays, as shown in Figure 1, four options in the left side of the window and displays the names of projects previously created in Xcode in a panel in the right side of the window. Three of the options ***Connect to a repository, Learn about using Xcode, Go to Apple's developer portal*** may be explored before starting this project, or can be revisited later. After saving a new project and exiting Xcode, work can be continued on a project by selecting it in the *Recent Projects* panel. To get started with our temperature project, click on **Create a new Xcode project.**

Figure 1 Xcode Window

Choose the Single View Application template as shown in Figure 2, then click on the Next button.

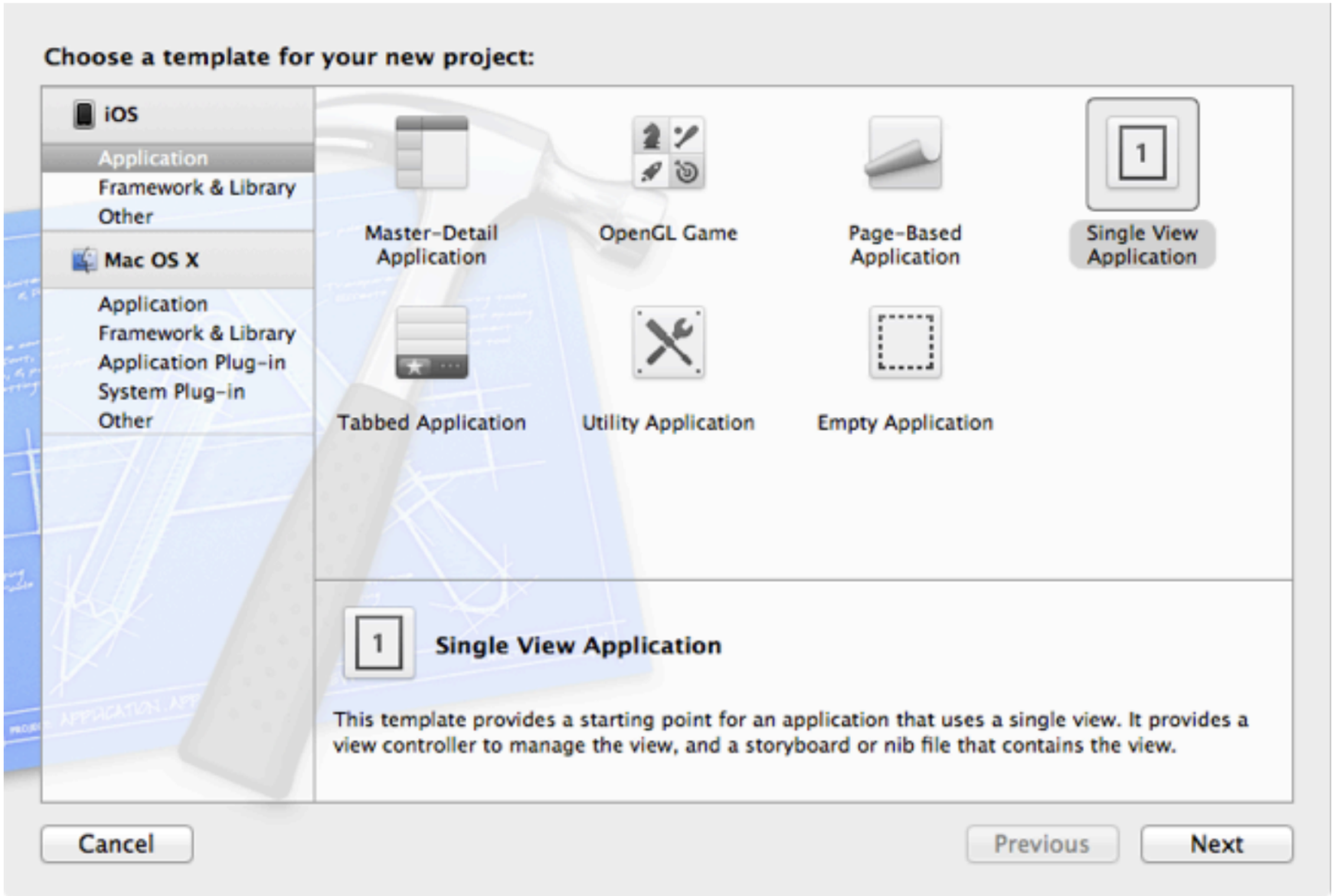

Figure 2 Creating a New Project – Selecting a Template

Enter *Converter* for the ***Product Name*** as shown in Figure 3. A different name can be used, but it will be easier to follow along with this project if you use the same name.

Enter *com* for the ***Company Identifier*** or leave it with the default value as shown in Figure 3.

Select Universal in the Device-Family menu.

Select the Use Automatic Reference Counting checkbox, but do not select the other checkboxes as shown in Figure 3. Click on the Next button.

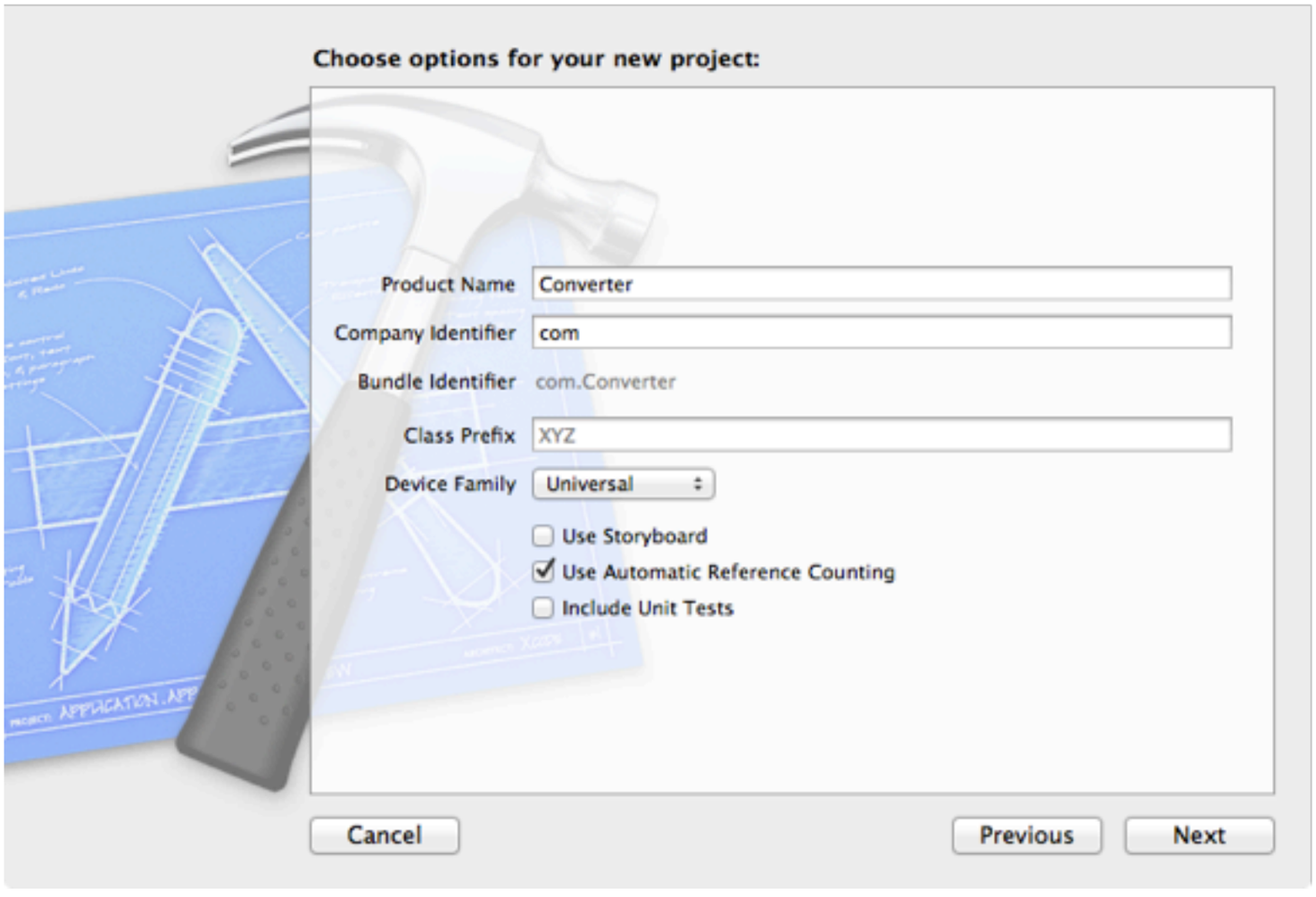

Figure 3 Creating a New Project – Naming the Project

Choose a location for your project as shown in Figure 4. Do not select the Source Control checkbox at the bottom of the window, then click on the ***Create*** button.

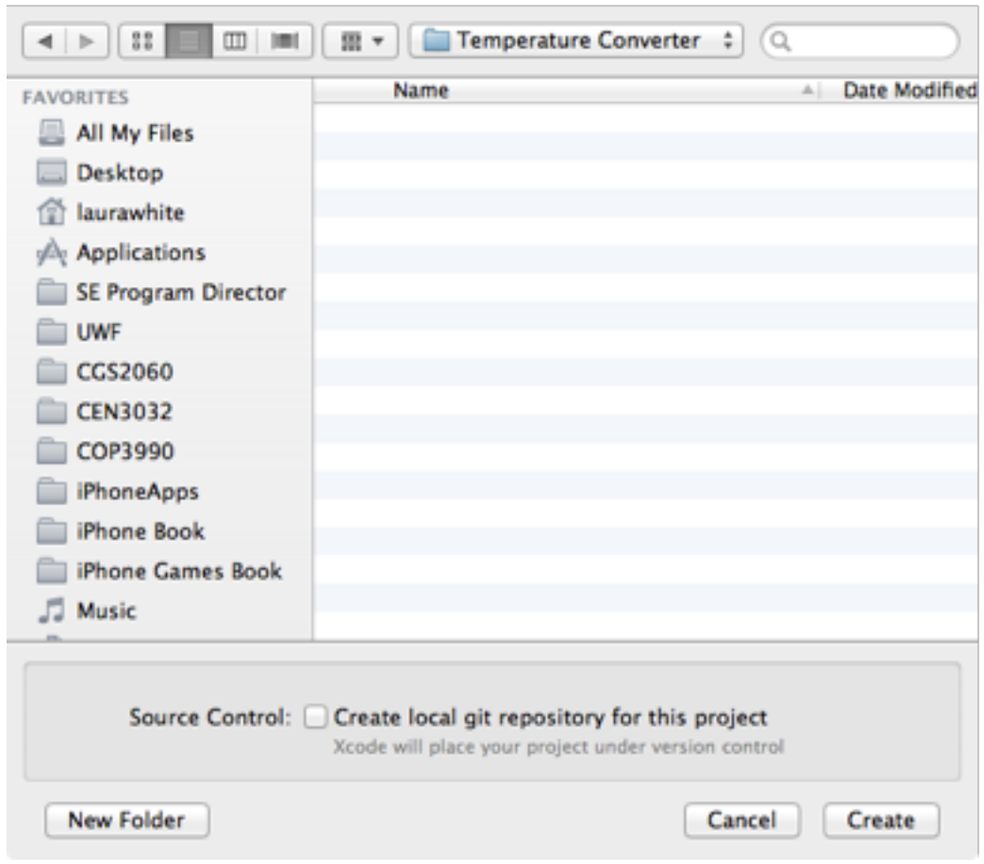

Figure 4 Creating a New Project – Selecting the Location for the Project

The Xcode Workspace Window

Once the project is created a single Xcode workspace window is launched as shown in Figure 5. Xcode fully integrates the Interface Builder design tool and the LLVM compiler into the Xcode Integrated Development Environment. It looks complex, but don't worry we will provide a short introduction of the main functions of Xcode.

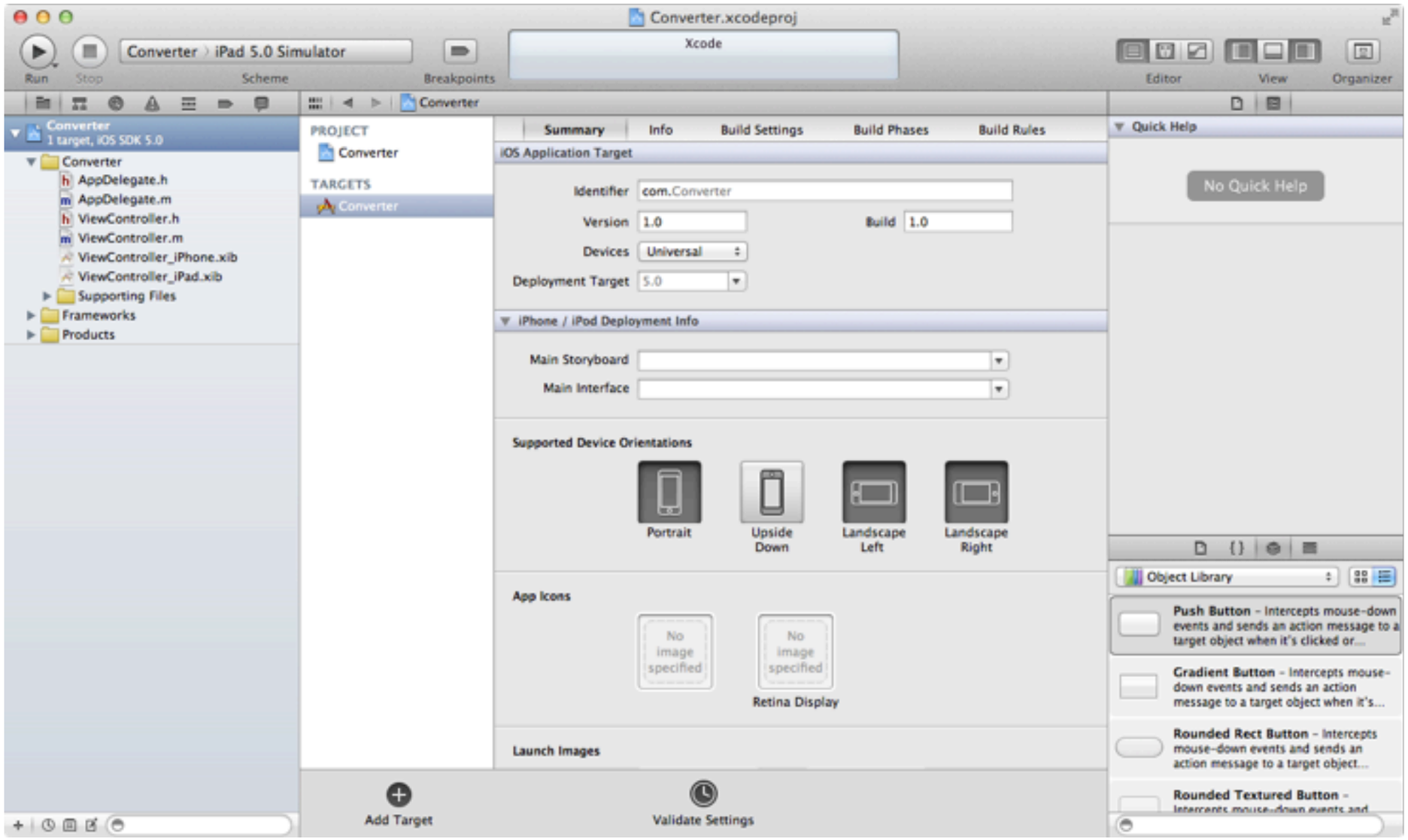

Figure 5 The Converter Project Inside the Xcode Workspace

Across the top of the Xcode workspace window is the Toolbar. Starting at the left of the Toolbar, the first things we see are the Run and Stop buttons that are used to run or stop our application inside the workspace. Notice the small down arrow associated with the Run button. There are five execute actions that can be performed in Xcode. These actions are Run, Test, Profile, Analyze, and Archive.

The next element in the Toolbar is the Scheme menu, as shown in Figure 6. In Xcode, the developer can select and create Schemes for their projects. A Scheme specifies what targets to build and what configurations to use when building a project.

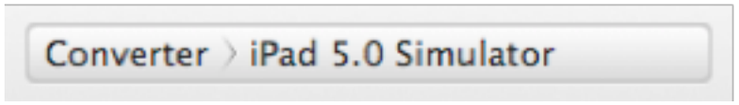

Figure 6 Xcode - Scheme Menu

The default schemes for an iOS project includes devices and simulators. When we click on the Scheme menu a pop-up window will appear with device options. Since we selected the Universal device family for our project, we will have options for an iOS Device, the iPad simulator, and the iPhone simulator as shown in Figure 7. A developer must be a member of one of Apple's iOS Developer Programs in order to run applications on an iOS device. Don't worry about the Update Simulators... option for now. In this chapter we will test our application in the iPhone 5.0 Simulator Environment so select iPhone 5.0 Simulator.

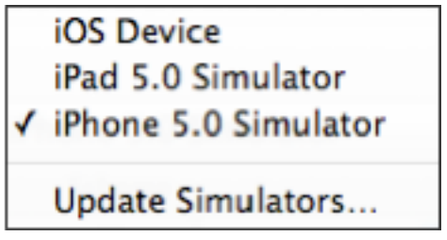

Figure 7 Choosing a Device to Build Inside Xcode

Notice the Breakpoints icon to the right of our scheme. It is used for debugging code. We are not going to use this option in this project. The center of the Toolbar contains a display error for messages related to actions performed in the workspace. Now look at the top right of the Toolbar. This is where the editor, view, and organizer options are located. It should appear similar to Figure 8.

Figure 8 View and Layout Options

The first group of Editor buttons in this part of the Toolbar are the editor selector buttons that are used to display different views of files within editor panes inside the Editor area of the Xcode workspace window. The first button selects the Standard editor which displays a file in a single editor pane. The second button selects the Assistant editor which is generally used to display a second file closely related to the file in the Standard editor within a second editor pane using the split editor pane. The third button selects Version editor which is generally used to display two versions of the same file within the split editor pane.

The next group of buttons are the View group used to select different areas of Xcode. The first button is used to select the Project's Navigator area on the left of the workspace window. The second button is used to display the Debug area at the bottom of the workspace window, and the third button is used to display the Utility area on the right side of the workspace window. Select the ViewController.h file in the Navigator area on the left. The Xcode workspace window should now appear similar to Figure 9.

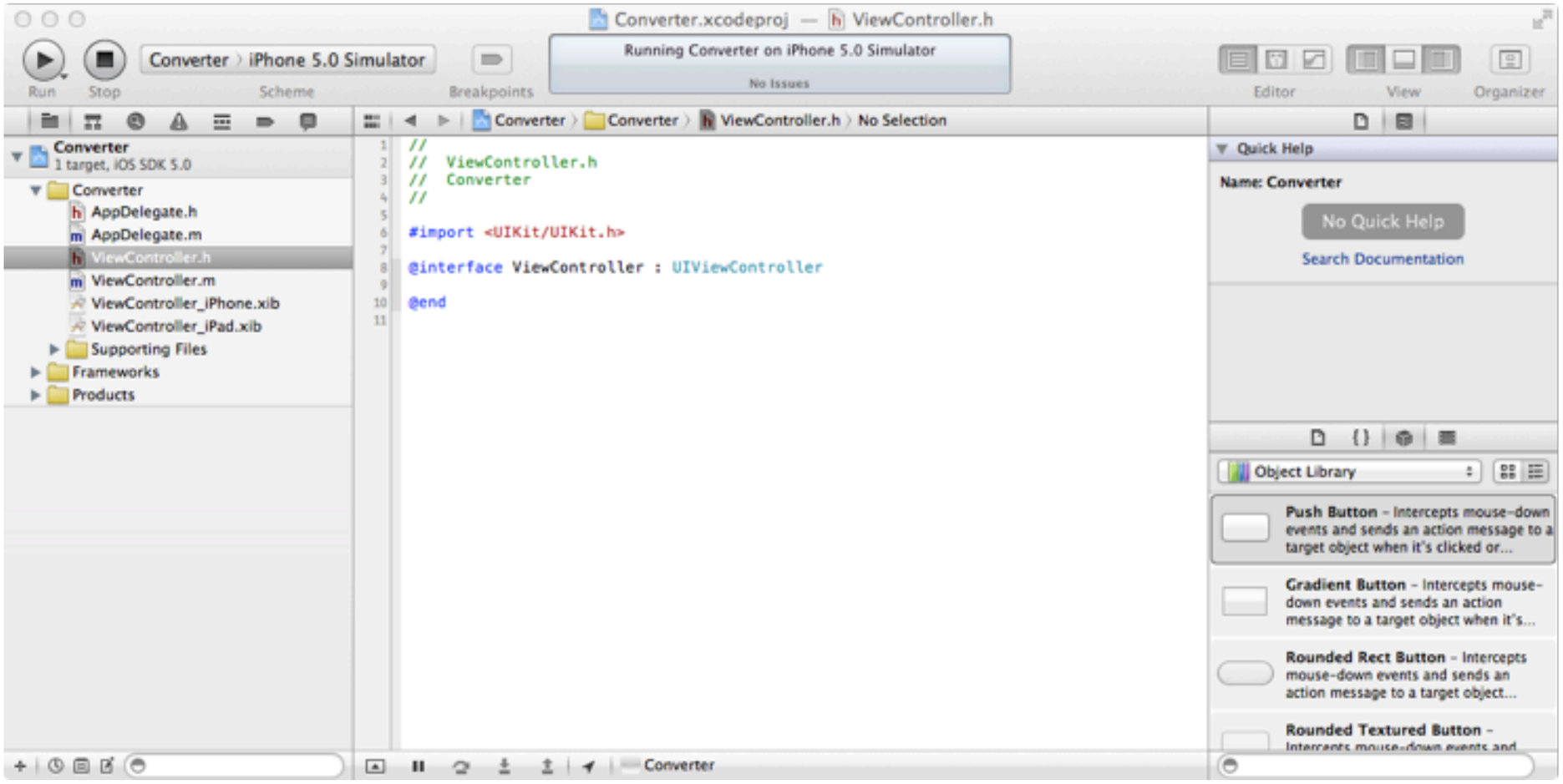

Figure 9 Xcode Workspace with a Header File in the Editor Area

It is important to understand how a project is structured inside Xcode so expand all of the folders in the project navigator area. The panel should appear similar to Figure 10.

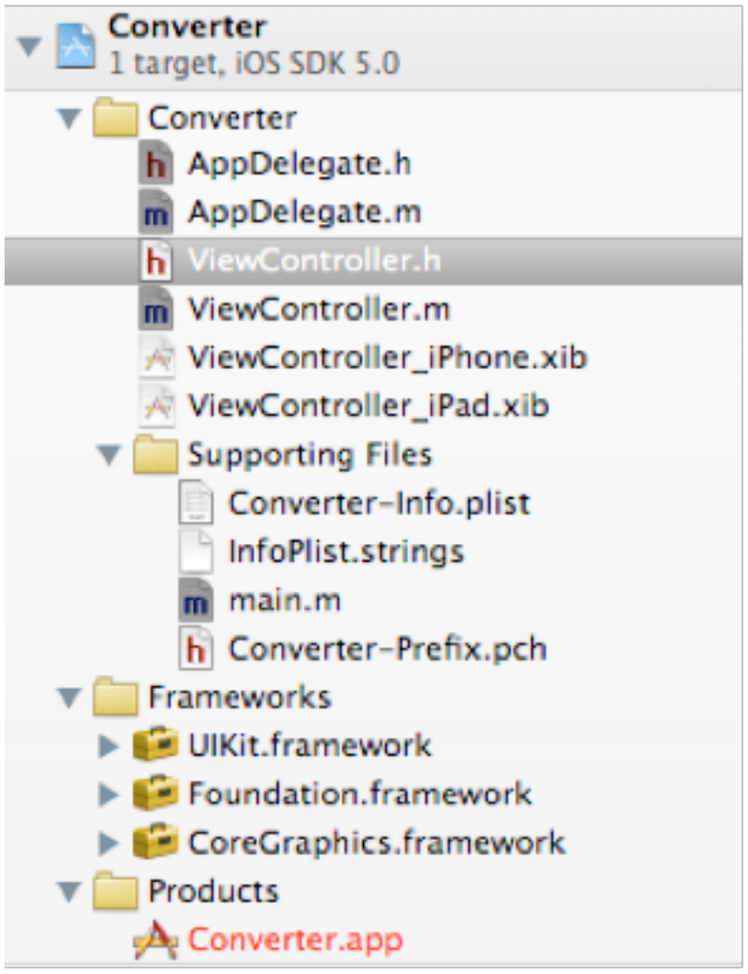

Figure 10 Project Navigator Area

All the files listed in the navigator area were created as a result of selecting the View Based Template when we created this project. The first two files listed in the Converter folder are AppDelegate.h and AppDelegate.m. All Objective-C classes are defined in two files. Just as in C, the .h files contain declarations and definitions of methods and variables, and the .m files contain the implementation of those methods.

Each iPhone project contains an AppDelegate class. Instead of subclassing and overriding methods for a complex object as is done in some object-oriented programming languages, in Objective-C we create objects from unmodified classes and then put our own custom code inside a delegate object that defines special characteristics for that object. As interesting events occur, the complex object sends messages to our delegate object. We use these messages to execute our custom code to implement the behavior we need. An AppDelegate class is generally responsible for handling critical system messages, such as moving to the background, suspending an application, and starting an application. The AppDelegate class sets the main view controller and the view that will be displayed on the screen. We will not modify the AppDelegate class for the project in this chapter.

The next four files in the navigator pane are the ViewController.h, ViewController.m, ViewController_iPhone.xib, and ViewController_iPad files. These files define a subclass of the UIViewController for our project. View controllers conform to an important concept in iOS programming, that uses the Model View Controller (MVC) pattern. This pattern provides a logical separation between that data that is used, and the view or user interface. The UIViewcontroller works as a mediator between the data and the user interface, and as such controls the operations used to process the data, and responds to changes and events that occur on the user interface. The MVC pattern is illustrated in the Figure 11.

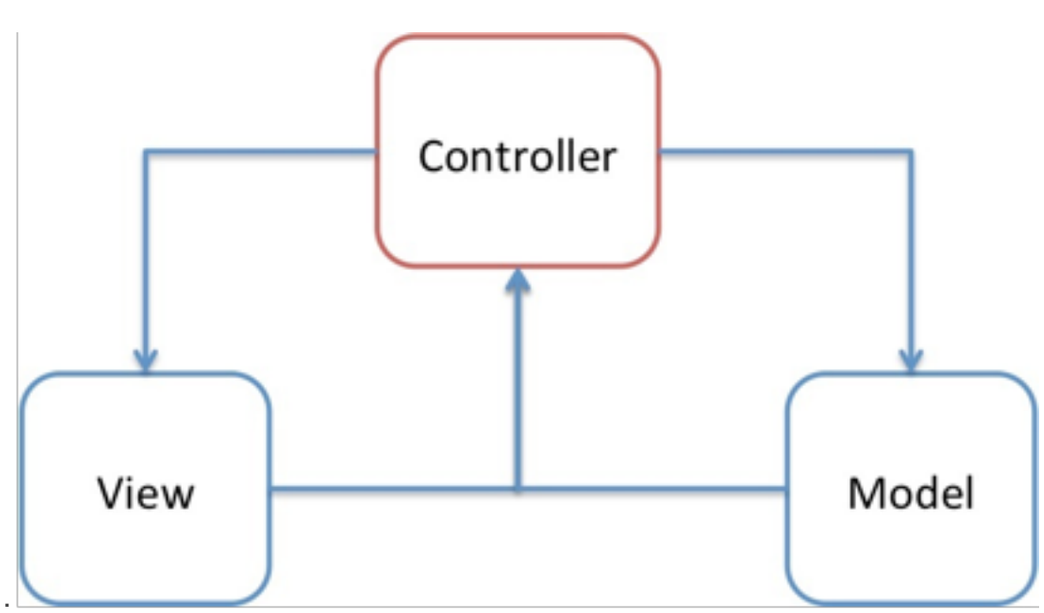

Figure 11 Model View Controller Pattern

There are numerous benefits of adopting the MVC pattern for iOS programming. These benefits include well-defined interfaces, reusability, and extensibility. How do we map the model in this figure to our application? The files ViewController.h and ViewController.m contain the code that defines the behavior for the application, and is where we will place our custom code later on in this chapter. These files are represented as the Controller in Figure 11. The ViewController.xib file stores the information about the user interface associated with the ViewController and is represented as the View in Figure 11. We will use this file to contain the user interface elements for our application. In applications that involve the use of data, that data represents the state of the application. This data is structured and stored within the application. This aspect of an application is represented by the model element in Figure 11. In simple applications this data can be embedded inside the view controller, which is what we will do in this chapter's project. In more complex applications the data might be managed within a database, plist files. xml files, or some other appropriate structure.

The next element in the Navigator pane is the Supporting Files folder which contains the following files:

- The Converter-Info.plist file contains the information about the application configuration. We will not make any changes to this file.
- The InfoPlist.strings file contains localized strings.
- The Converter–Prefix.pch file contains the header that is included in all source files for a project.
- The main.m file is responsible for starting the application. All iOS applications contain this file.

The next folder is the Frameworks folder which contains the iOS frameworks that are used within iPhone applications. By default all projects contain the (a) UIKit framework that is responsible for User Interface and Touch elements, (b) Foundation framework that contains basic data structures such as strings, and arrays, and (c) Core Graphics framework for drawing on the screen. Other frameworks may be added to a project by the developer as needed.

The last folder in our project is the Products folder that will contain the .app file that will contain the executable code for the application after it is built. We currently see the name we selected when we created this project with .app appended at the end.

Adding Resources to a Project

The developer may want to add additional resources to a project. In this section we will demonstrate how this is done, by adding an image to our project, using the following steps.

First, in the newly opened Xcode window, ctrl-click on the Project name at the top of the Navigator area in and then select ***New Group*** in the popup menu as shown in Figure 12. Then double click on the label for the New Group folder and change its name to Resources.

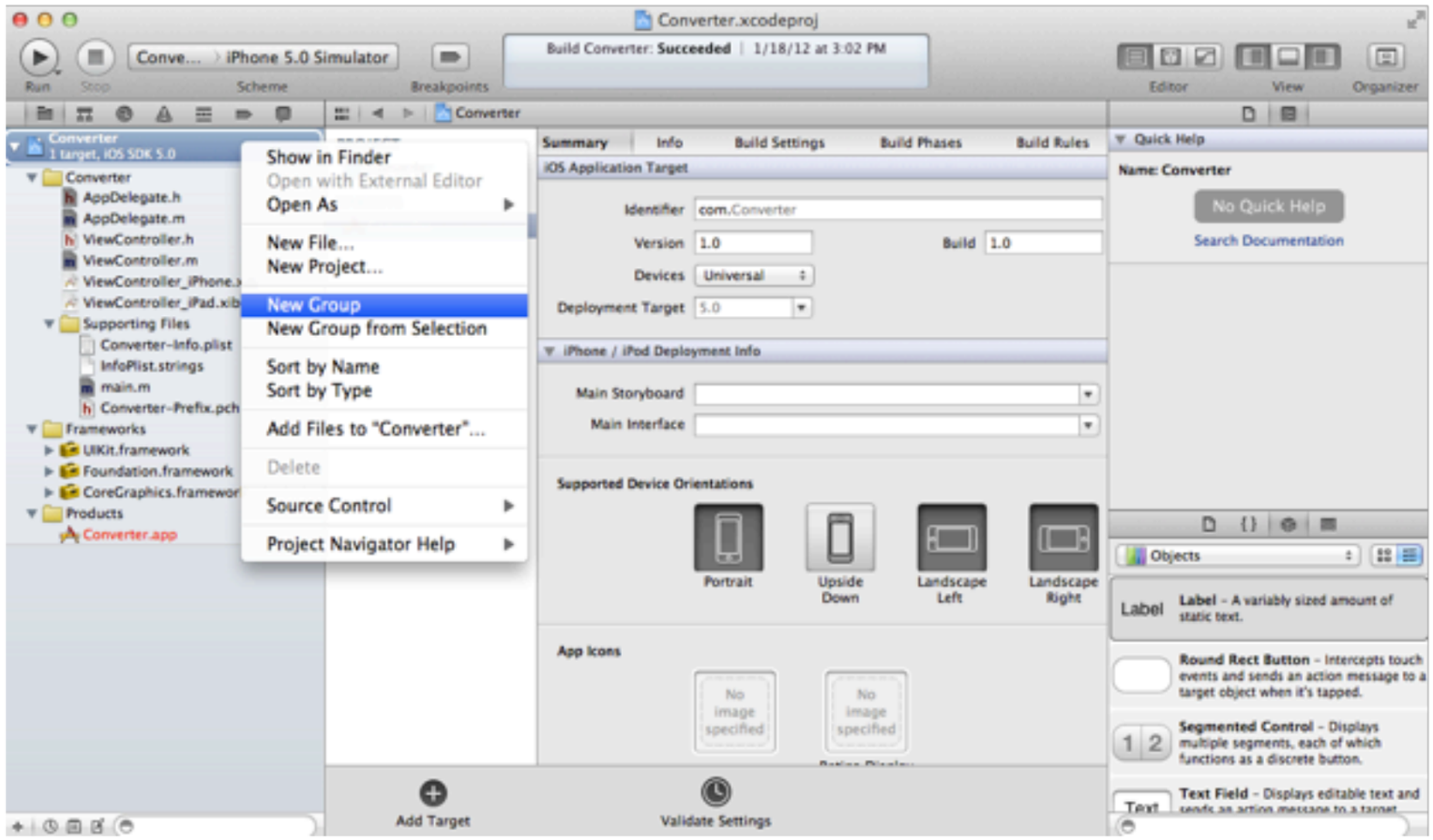

Figure 12 Creating a New Group in a Project

Now, download the image for the Converter app from the companion website for this book at http://www.epressplus.com/2012/01/21/developing-ios5-applications/. Select the logo.png image from the Chapter 2 folder and drag it to the newly created Resources group folder—be sure to check the *Copy items into destination group's folder (if needed)* as shown in Figure 13 and then click on the Finish button.

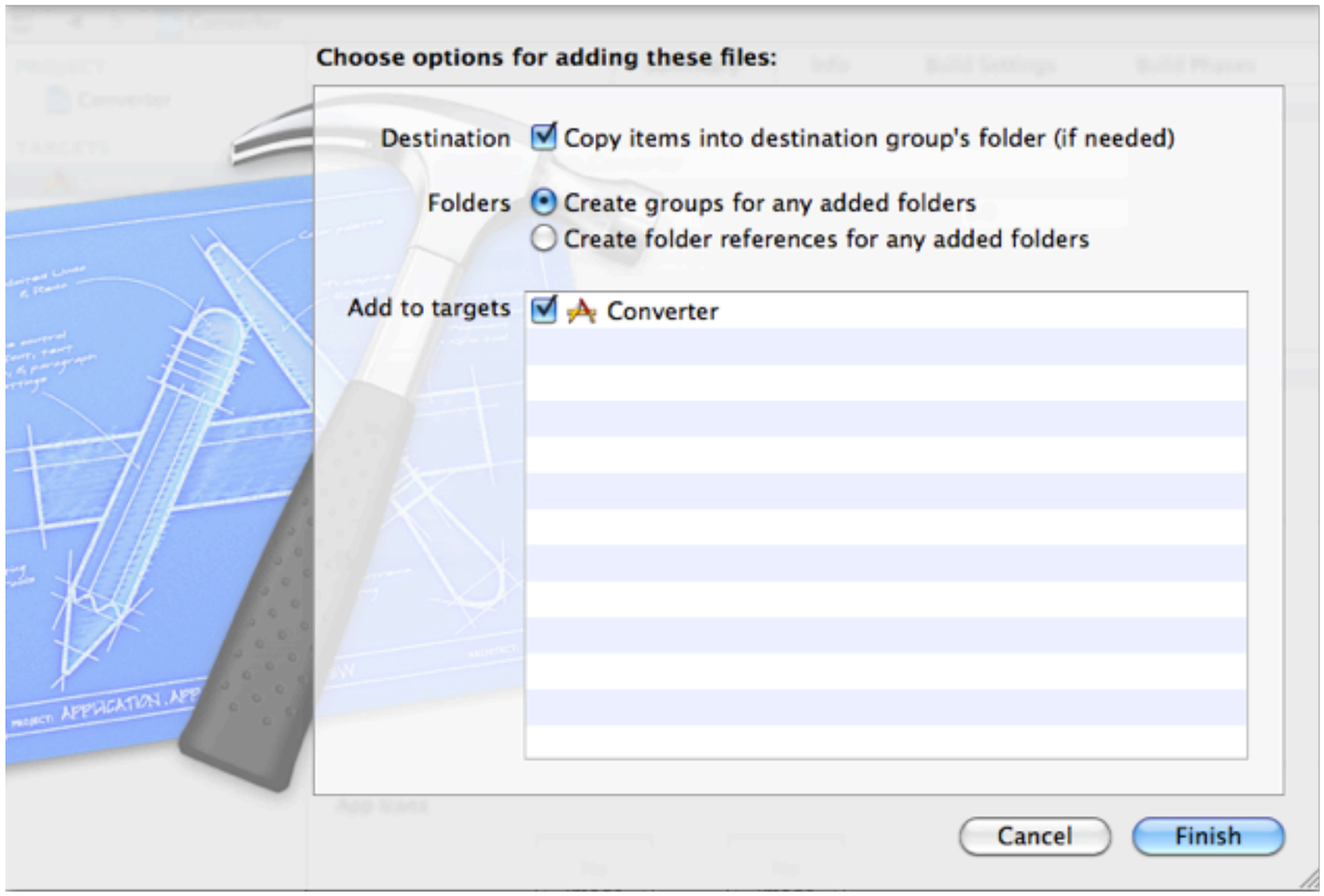

Figure 13 Adding Images to a Project

If everything went right the logo.png image will now be listed inside the Resources folder.

KEY POINT

Remember that images must be dragged into the Xcode Navigator panel in order to be added to a project. Copying files to the Applications folder will not suffice.

In this section, you created a new project in Xcode, and learned how to add resources to it. Good job! Now we can start adding elements to the user interface.

Adding the Necessary User Interface Elements

In this section we will create user interface elements for the application. Traditionally, this was done inside the Interface Builder application which was a separate application from Xcode, but these applications worked very well together to support the development of applications. Now, Interface Builder is fully integrated within Xcode. Information about the User Interface elements are contained in files with an .xib file extension. Expand the Converter folder if it isn't already. Then click on the ViewController_iPhone.xib to open the user interface editor panel inside the Xcode editing window. Before we start adding elements, double-check that the selected scheme is the iPhone 5.0 simulator as shown in the Figure 14.

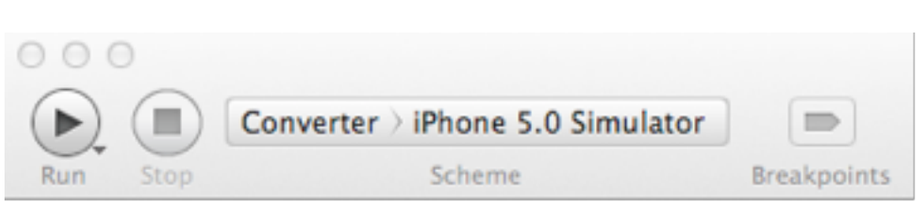

Figure 14 Simulator Running Mode

The gray View window shown in Figure 15 displays the same content that the application will display on the screen of an iPhone application. Currently it doesn't look too exciting -- nothing more than a grey rectangle – because we haven't added any elements to it yet. Now click on the ***Run*** button in Xcode to compile and run the application in the iPhone simulator. As shown in Figure 16, the application simply displays a grey rectangle in the simulator, just as it does in the View window.

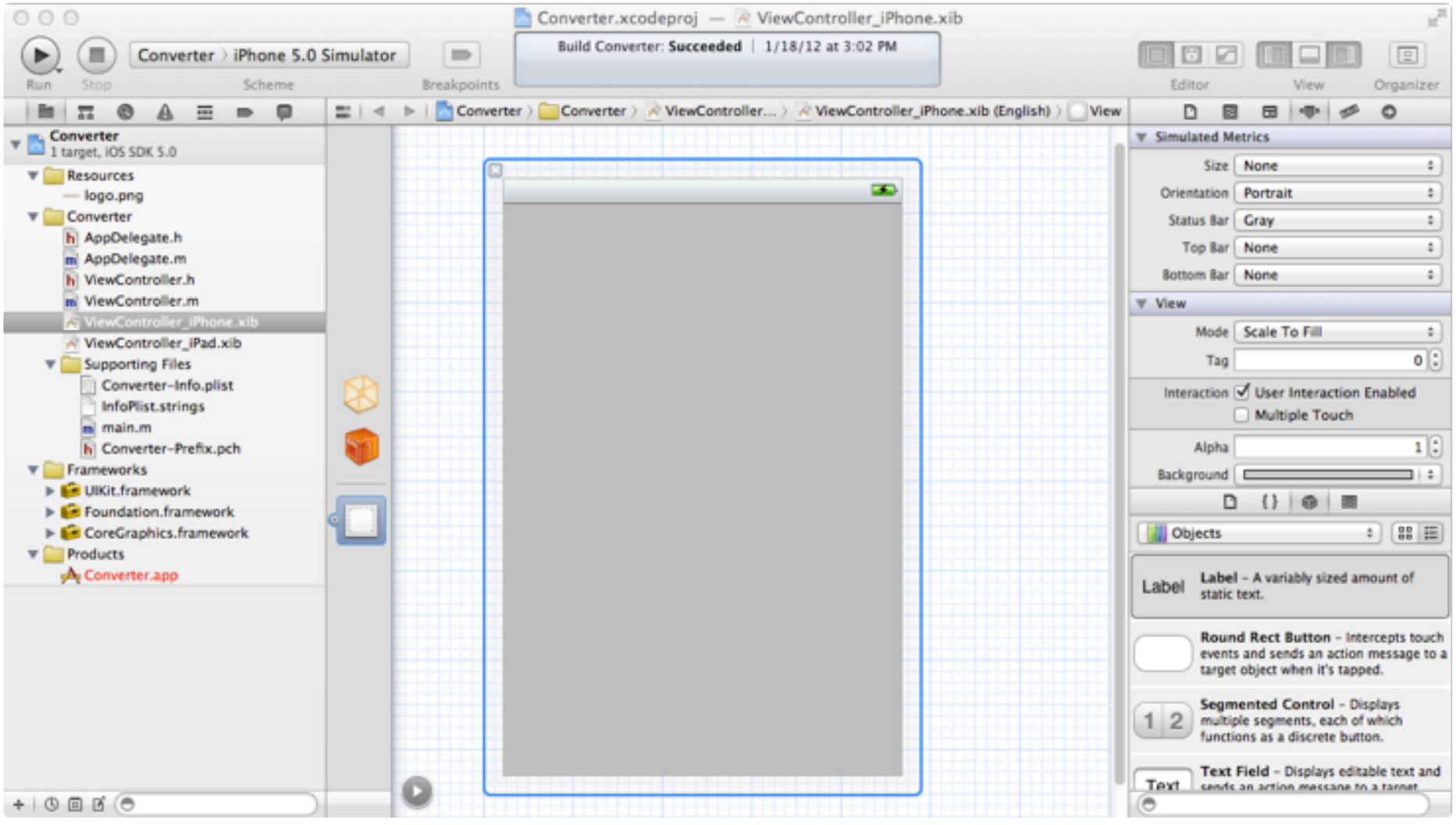

Figure 15 View Window in Editor Area

Figure 16 iPhone Simulator

We are now ready to start adding elements to our user interface. With the ViewController_iPhone.xib file selected in the Navigator area, navigate to View > Utilities > Show Object Library. The Utilities panel that contains the Object Library will open on the right side of the Xcode window in the Utilities area. Notice the buttons to the right of the drop down menu for the Object Library toward the bottom of the Utilities area. The Object Library can be displayed either as a list of elements as shown in Figure 17 or as icons as shown in Figure 18.

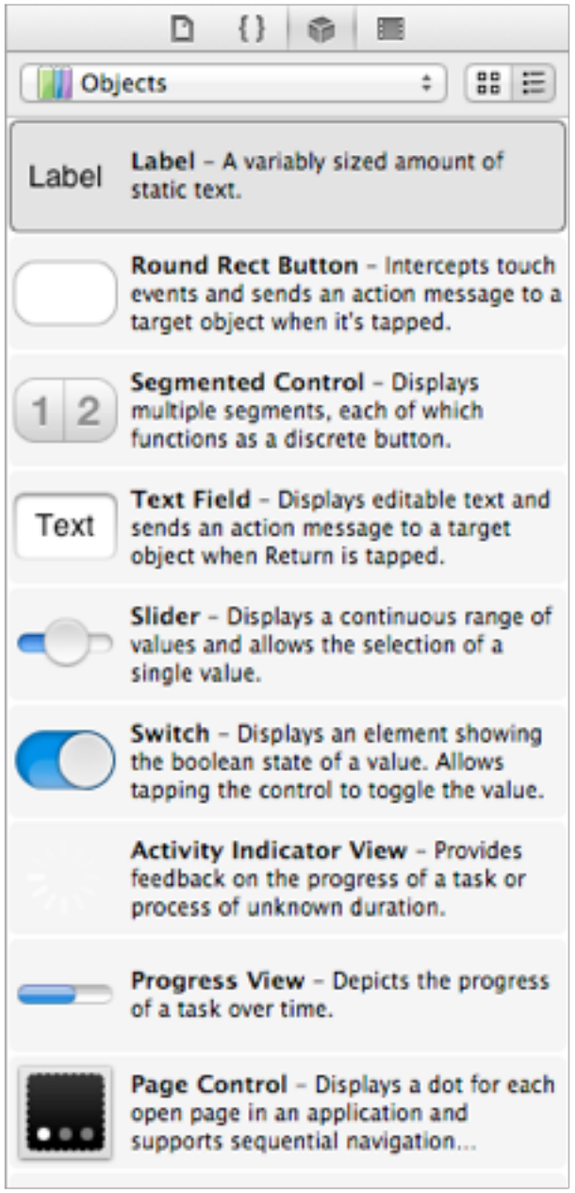

Figure 17 Objects Library - List Display

Figure 18 Objects Library icon Display

The Object Library contains the objects that can be used as elements in the user interface. We can categorize the objects in the library into several groups that can be viewed by clicking on the Object Library dropdown menu as shown in Figure 19.

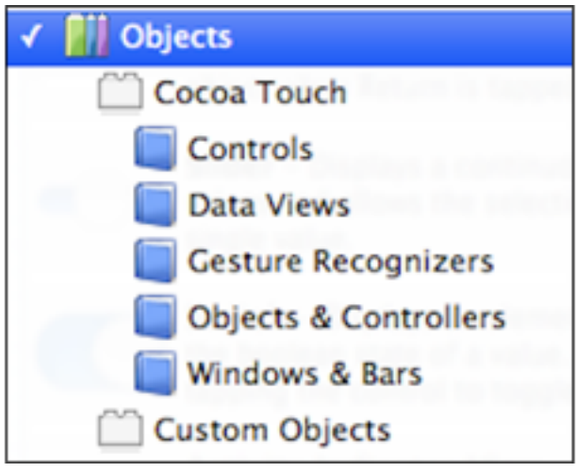

Figure 19 Objects Library Groups

Cocoa Touch – Objects & Controllers

Objects & Controllers are used to create view controllers and different types of user interface controllers such as tab bars and image pickers. View controllers are often associated with the screens of an application, tab bar controllers are used in an app to provide navigation to different screens.

Cocoa Touch – Data Views

Data Views are used to display data and information in different ways such as a Table View that displays data in the format of a table, a Web View for displaying HTML content, a Scroll View for displaying views that are bigger than the application window, an Image View for displaying images, and more. We will use an image view in our application, so click on the Image View icon in the Object Library panel and drag it into the View window. A UIImageView object is now displayed in the View window. This object's label indicates that it belongs to the UIImageView class. By default the Image View object will fill the entire View window. We change its size by clicking on any of the handles positioned around the border of the image view and then drag to the desired shape. For this project, resize the Image View object and position it so that so that it appears similar to Figure 20.

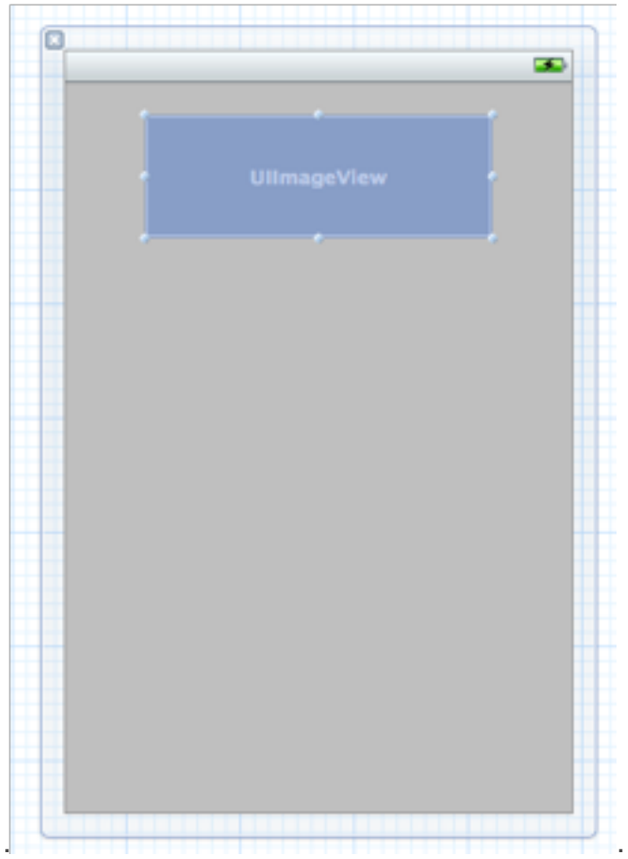

Figure 20 View Window

Cocoa Touch – Controls

Controls are used to create labels, buttons, sliders, segmented controls, text fields, and switches. We will use several of these objects in our Converter application. Click and drag the following four elements from the library to the View window: Segmented Control, Text Field, Label, and Round Rect Button. Click and drag the elements within the View window so that the objects appear in the View window similar to Figure 21. Once the objects have been placed in the View window they can then be connected to code that implements the desired behaviors. We will do this later in this chapter. This is a good time to save your work. Navigate to File > Save.

Figure 21 View with User Interface Elements.

There are two ways that information regarding File's Owner, First Responder, and View objects are accessed for a xib file. Notice the narrow gray panel between the Navigator area and the Editor area in Figure 22. It contains three icons, the gray box with the enclosed white rounded square selects the view, the orange box selects the First Responder interface in the Utilities panel, and the translucent yellow box selects the File's Owner interface. Alternatively, clicking on the gray circle with the white arrow in the lower left of the Editor area will reveal a detailed view for this information, as shown in Figure 23. These objects may be listed in a different order, which is fine, however, if any of these are missing it is necessary to retrace the previous steps in this section. If all of the objects are listed then we may now set the attributes for the objects that comprise our user interface elements.

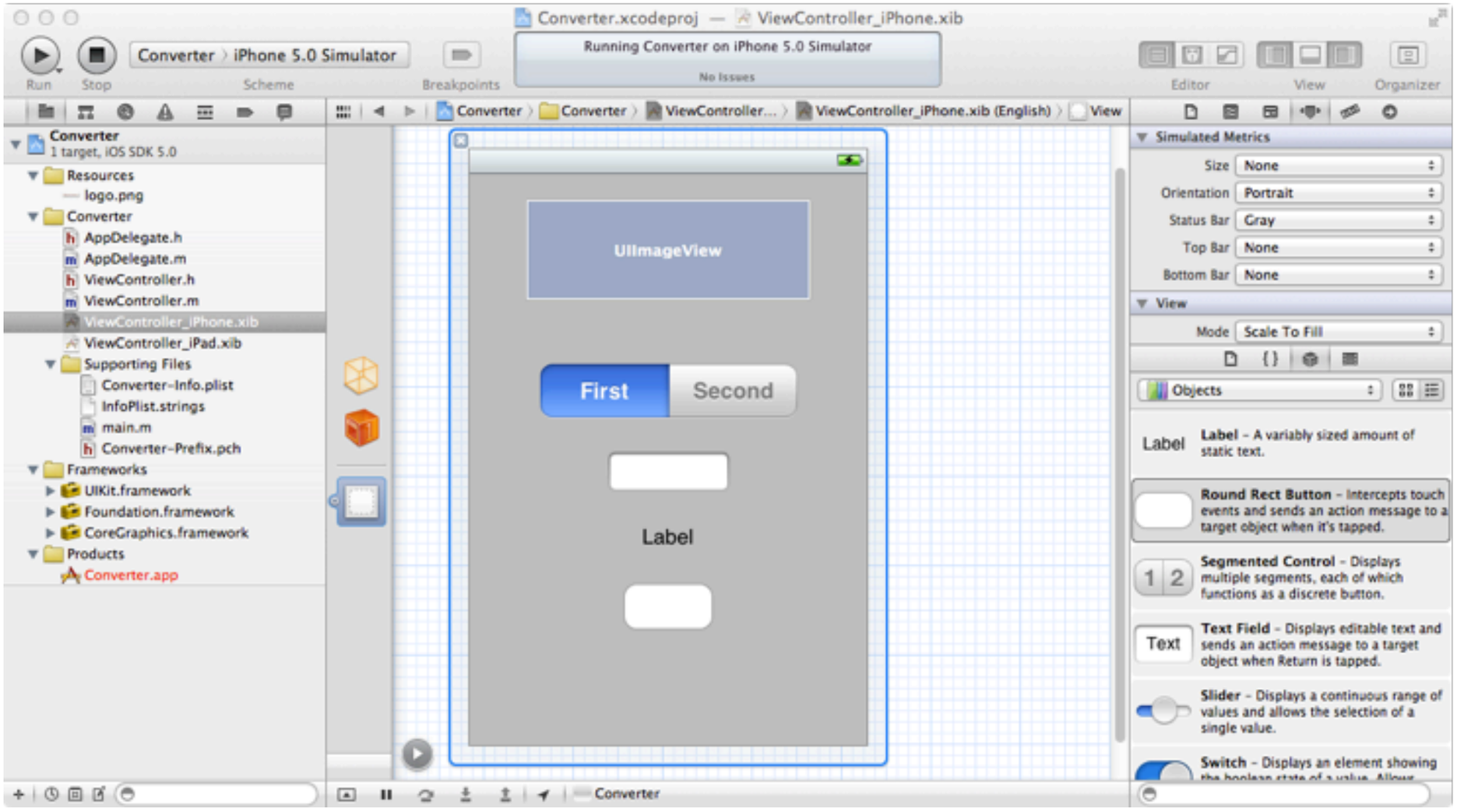

Figure 22 ViewController_iPhone.xib File

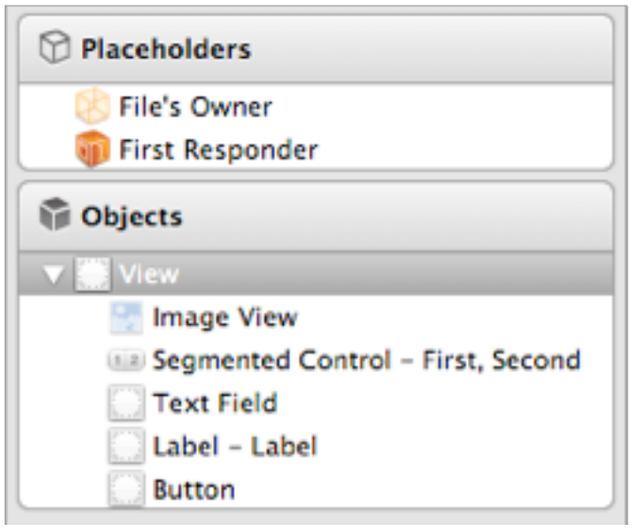

Figure 23 User Interface Elements for Temperature Converter Application

Setting Attributes for User Interface Elements

To set the attributes for the user interface elements, click on the View to select it and then navigate to View > Utilities > Show Attributes Inspector. to reveal the attributes, shown in Figure 24, for the View in the Utilities area. There are several attributes that can be defined, such as the orientation, navigation bars, background color, and more. Click on the Background menu in this pane and change the background to white.

Next click on the UIImageView object in the View window to reveal its attributes in the Utilities area.

Click on the Image menu and select logo.png in the list. The View window now has the Temperature Converter image from the companion website added to the project earlier in the location where the UIImageView object placeholder was previously.

Select the image in the View window then click on the Mode menu in the Image Attributes window and then select Aspect Fit – this will adjust the image to display proportionally within the Image View object. The attributes will now appear as shown in Figure 25 and the View window should now appear similar to Figure 26.

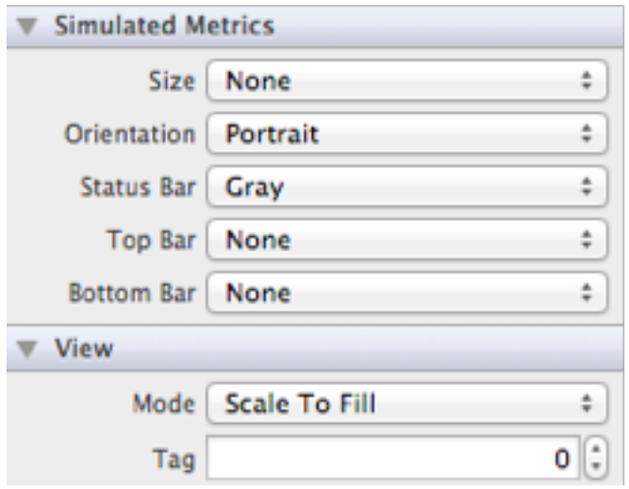

Figure 24 View Attributes Pane

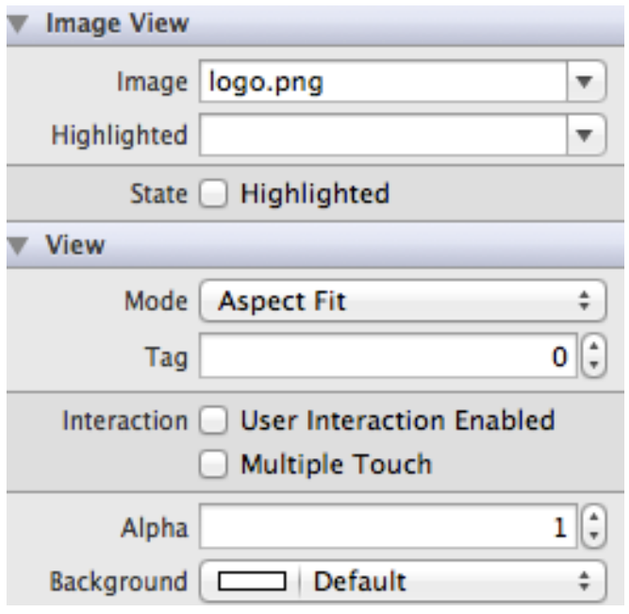

Figure 25 Image View Attributes Pane

Figure 26 User Interface Elements in Converter View Window

Save your work by navigating to File > Save. Then click on the Run button in the Xcode Toolbar to see how the application appears in the iPhone simulator. The iPhone simulator should appear similar to Figure 27. Good job! You have learned how to work with iOS image views and how to change the background of a project!

Figure 27 Partially Developed Temperature Converter App Displayed in the iPhone Simulator

The next user interface object in the View window is a segmented control. This element can be treated as a set of segments from which each segment functions as a button. Go ahead and click on the segmented control in the View window and then click on the View > Utilities > Show Attributes Inspector. A Segmented Control Attributes window similar to Figure 28 will appear. The Attributes Inspector is used to change the properties of the segmented control. Notice that the Segmented Control Attributes pane is divided vertically into three sections labeled Segmented Control, Control, and View. Let's have a look at the properties that can be changed for segmented control elements in the Segmented Control section.

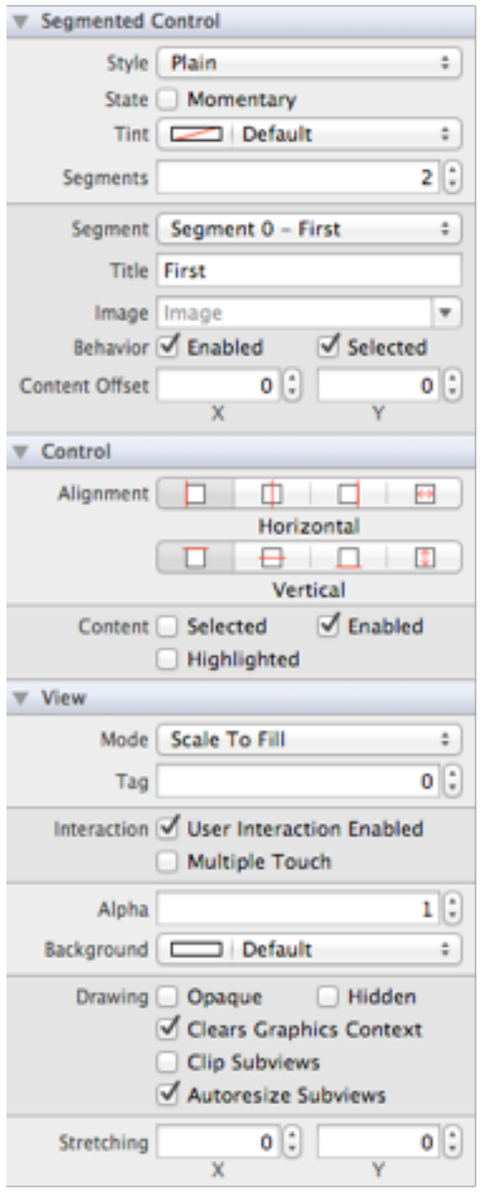

Figure 28 Segmented Control Attributes Pane

The first attribute, Style, provides the capability to apply different styles:

- Plain – the default view
- Bordered – plain segmented control with border
- Bar – a bit sleeker with grey coloring and white labels
- Bezeled – similar to bar but a bit wider with white and gray labels

Go ahead and play with the Style segments and choose the one that is most appealing. We chose bar for the value of this attribute.

The next attribute in the pane is Momentary which is selected or not with a checkbox. This attribute determines whether a momentary change of appearance or a permanent change in appearance is desired when the user taps on one of the segments in the segmented control. We do not check the box so that the change in appearance will be permanent. The next attribute in the Attributes Inspector window is Tint, which allows developers to change the color of the control. We will retain the default color scheme for our application. We also set the value of Segments to 2 since we will use two segments to switch between Celsius and Fahrenheit functionality.

The next section in the Segmented Control Attributes pane relates to attributes for each segment in the segmented control. The attributes that pertain to each segment are:

- Title
- Image
- Enabled Checkbox
- Selected Checkbox
- Content Offset

Each segmented control is enumerated from left to right as Segment 0 ... Segment n. By default Segment 0 is selected. Set the title for the first segment by typing Fahrenheit to Celsius into the Title text box. Set the title for the second segment by selecting Segment 1 and then typing Celsius

to Fahrenheit into the Title text box. An image could be applied to the segments but that is not desired for this application. The Enabled checkbox should be selected for each segment. The Selected checkbox should be checked for whichever segment should be selected by default. The offset attribute is not desired for this segmented control. At this point, the View window should appear similar to Figure 29. Now is a good time to Run the application in the simulator again. Click the Run button in the Xcode Toolbar. The screen in the iPhone simulator should appear similar to the View window in Figure 29.

Figure 29 View Window After Setting Segmented Control Attributes

The next two elements in the View window are a Text Field and a Label. We will look at the attributes for these elements now, but won't actually change any of these attributes at this point – we will change the values of the attributes for these elements programmatically later in this chapter.

Text Fields are used to process textual data provided by a user. Click on the Text Field in the View window and then navigate to View > Utilities > Show Attributes Inspector. The Text Field Attributes pane will appear similar to Figure 30. Notice that the Text Field Attributes pane contains three major sections: Text Field, Control, and View. The attributes within the Text Field section can be used to set the appearance, behavior, and input traits for the text fields.

To examine the attributes for the Label element that appears next in the View window, click on the Label element in the View window and then navigate to View > Utilities > Show Attributes Inspector. A Labels Attributes pane similar to Figure 31 will appear. Labels are one of the most commonly used user interface elements. A label is used to display text. Look at the attributes. The first and the most important one is Text where the desired text for the label may be entered. Notice that there is quite an assortment of attributes such as font, font size, and alignment that can be set to control how labels appear. For this project, we will set the attributes we need for this project programmatically later on.

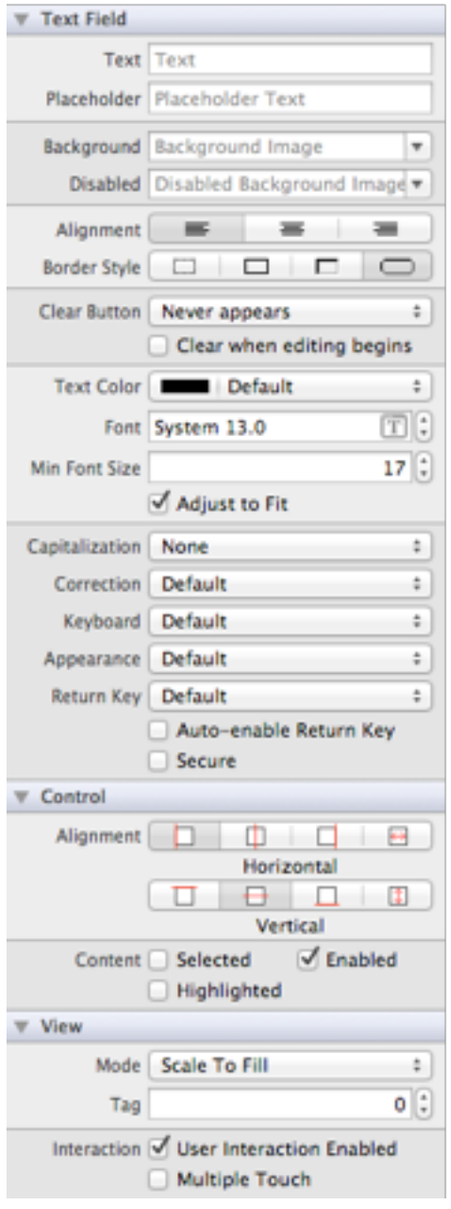

Figure 30 Text Field Attributes Pane

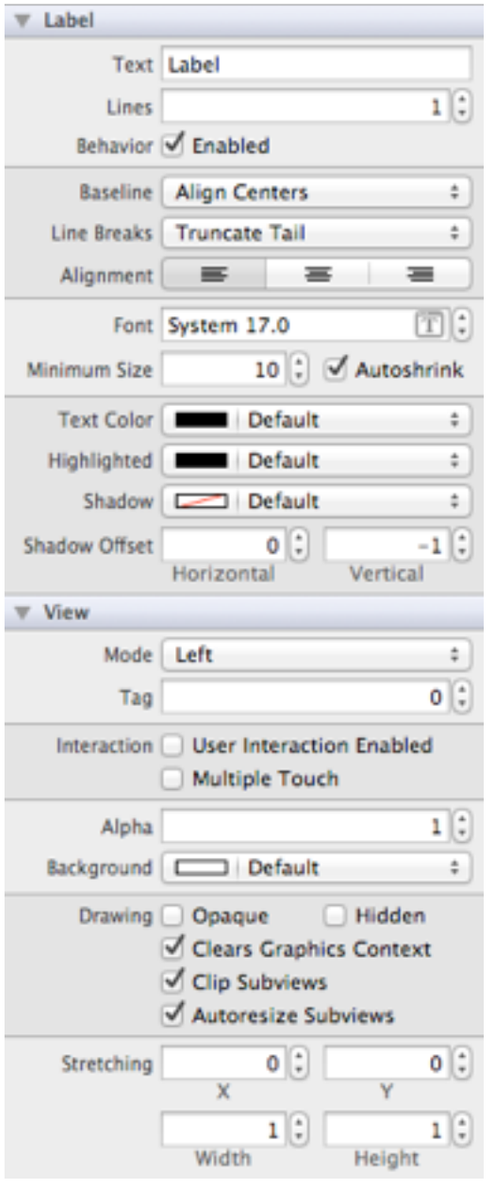

Figure 31 Label Attributes Pane

The next element in our View window is a Button. Buttons trigger methods when user interaction is detected. Click on the button in the View window and navigate to View > Utilities > Show Attributes Inspector to open the Button Attributes pane similar to Figure 32.

The Button Attributes pane is divided into three sections, Button, Control, and View. The first attribute in the Button section is Type, this allows the developer to select the type of button desired. The various types are:

- Custom: a transparent rectangle.
- Rounded Rectangle: a rectangle with rounded corners, the default form of the button.
- Detail Disclosure: a greater than symbol in a round button that is typically used to display additional details on a new screen.
- Info Light: displays the typical info icon inside a light colored button.
- Info Dark: displays the typical info icon inside a dark colored button.
- Add Contact: the button with the plus symbol typically used for adding an additional element, such as a new contact in the Apple Address Book.

Select Rounded Rect for the style of the button for the Converter project.

KEY POINT

There are several types of button options that have typical uses in native applications. It is important for developers to maintain use of buttons in a manner that is typical in native applications. This ensures that apps retain the behavior that is intuitive to users. This is an important consideration related to human interactions in high quality applications

The next attribute is the State Configuration to set the initial state of the button -- the default is fine for this project. The Title attribute contains the title of the button. For this application the word ***Convert*** should be typed into the Title text field. The button may need to be resized and repositioned to accommodate the title. There are also attributes to select an image for the button title rather than plain text, or to select a title background image for the button. Additional attributes are available to select colors for the text and shadows. At this point our View window should appear similar to Figure 33. This is a good time to confirm that everything we have created so far displays as expected inside the iPhone simulator. Navigate to File > Save to save your application, and then click on the Run button to view the Temperature Conversion application inside the iPhone simulator.

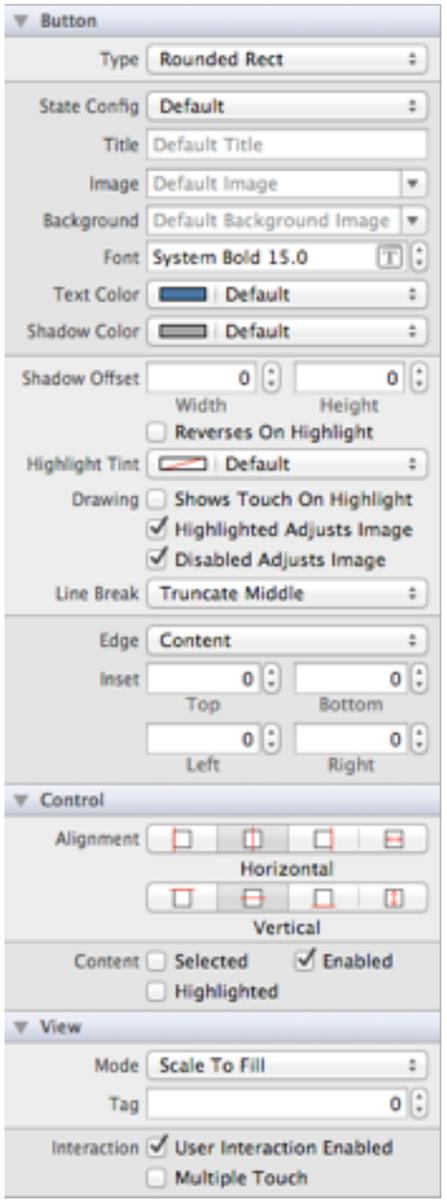

Figure 32 Button Attributes Pane

Figure 33 View Window

Creating Functionality for User Interface Objects in Xcode

So far we have created the user interface elements for the Temperature Converter application and defined attributes for some of those elements. We now need to create the functionality in the Objective-C code for the user interface elements in order to have a working Temperature Converter application.

The classes that implement the user interface elements for the Temperature Converter application are UIimageView for the image view element, UISegmentedControl for the segmented control, UITextField for the text field, UILabel for the label, and UIButton for the button. No functionality is needed for the image view element, and no functionality specific to the button itself is needed, however a method to compute a conversion to

Celsius and a conversion to Fahrenheit when the button is tapped is needed. We also need to create and customize instances of UISegmentedControl, UITextField, and UILabel in the code.

Connecting User Interface Elements to the Code

The first step in defining the behavior of the user interface elements is to connect them to the code. Connections link the user interface elements to the code so that they can be accessed and modified via the code when the application is running. Select the ViewController_iPhone.xib file inside the Navigator area then navigate to View > Assistant Editor > Show Assistant Editor, or click on the Assistant Editor button in the Xcode Toolbar. The editor area should appear similar to Figure 34 with the ViewController.h file shown in the Assistant Editor--note that we closed the Utility area using the button in the top right of the Toolbar. Recall, that the Assistant Editor will always open a file related to the one that is already displayed in the single editor area before the Assistant Editor is selected. For the ViewController_iPhone.xib file, there are two closely related files. ViewController.h and ViewController.m. An item in the jump bar above the editor, shown in Figure 35, provides the number of related files and arrows to select these files. In our case there is a 2 since there are two related files. If the ViewController.m file is displayed in the assistant editor rather than the ViewController.h file, then use one of the arrows to switch to the ViewController.h file which should appear as shown in Figure 34.

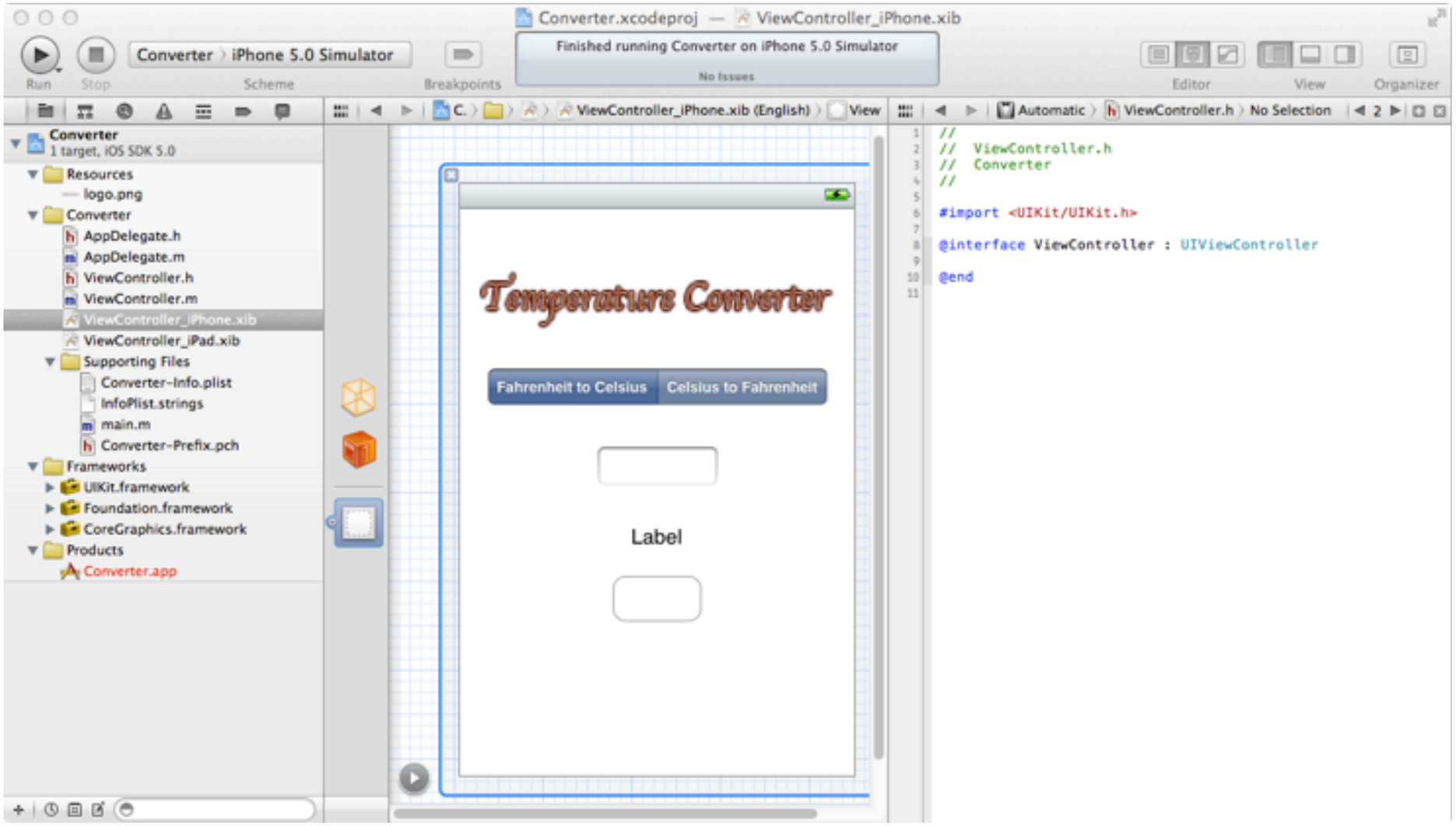

Figure 34 Editor Area Displaying the Assistant Editor

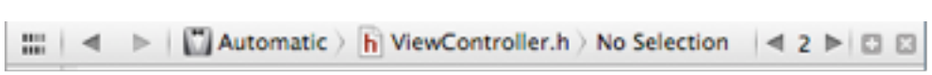

Figure 35 Editor Area Jump Bar

Now ctrl-click on the segmented control element in the view window, a new window will popup that should look similar to that shown in Figure 36.

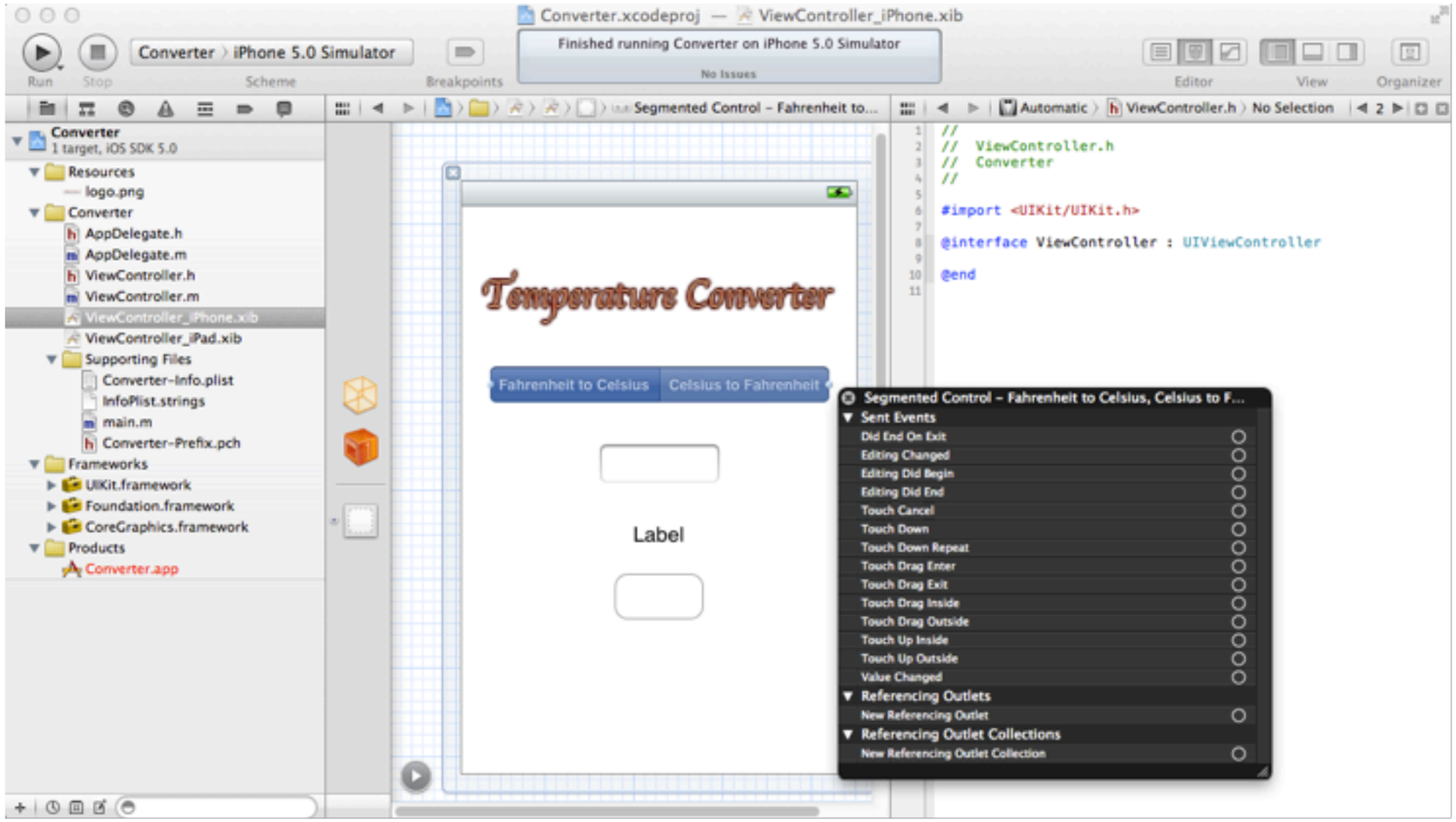

Figure 36 Segmented Control Events and Outlets

The Segmented Control window contains three sections. The Sent Events section contains events that can be used when working with a segmented control. The Referencing Outlets section provides the means to create a connection between an object in the current object to an outlet in another object. The third section allows us to create a outlet connections to a collection object. Sometimes you will need to create a complex User Interface for your applications and creating outlets for each of them might be a tedious process. You can use the outlet collections that will store User Interface elements in an array to reduce the need to define the behavior individually for each element.

Click on the circle to the far right of the New Referencing Outlet as shown in Figure 37 and then drag to the line above the end directive in the ViewController.h file. A popup window will appear as shown in Figure 37 when the mouse is released. Enter segmentedControl for the name of the Segmented Control outlet as shown in Figure 38.

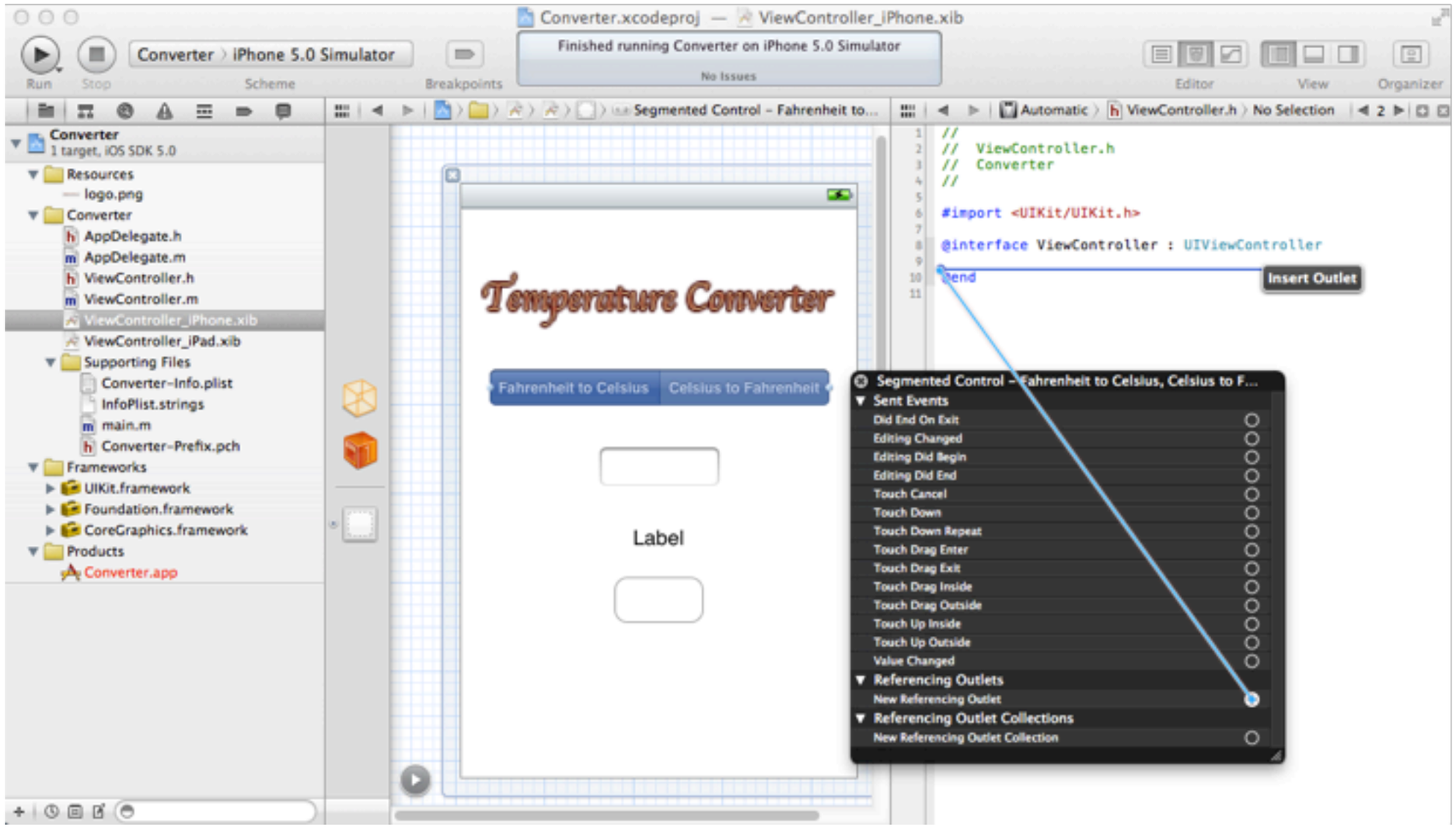

Figure 37 Connecting the Segmented Control to the Code

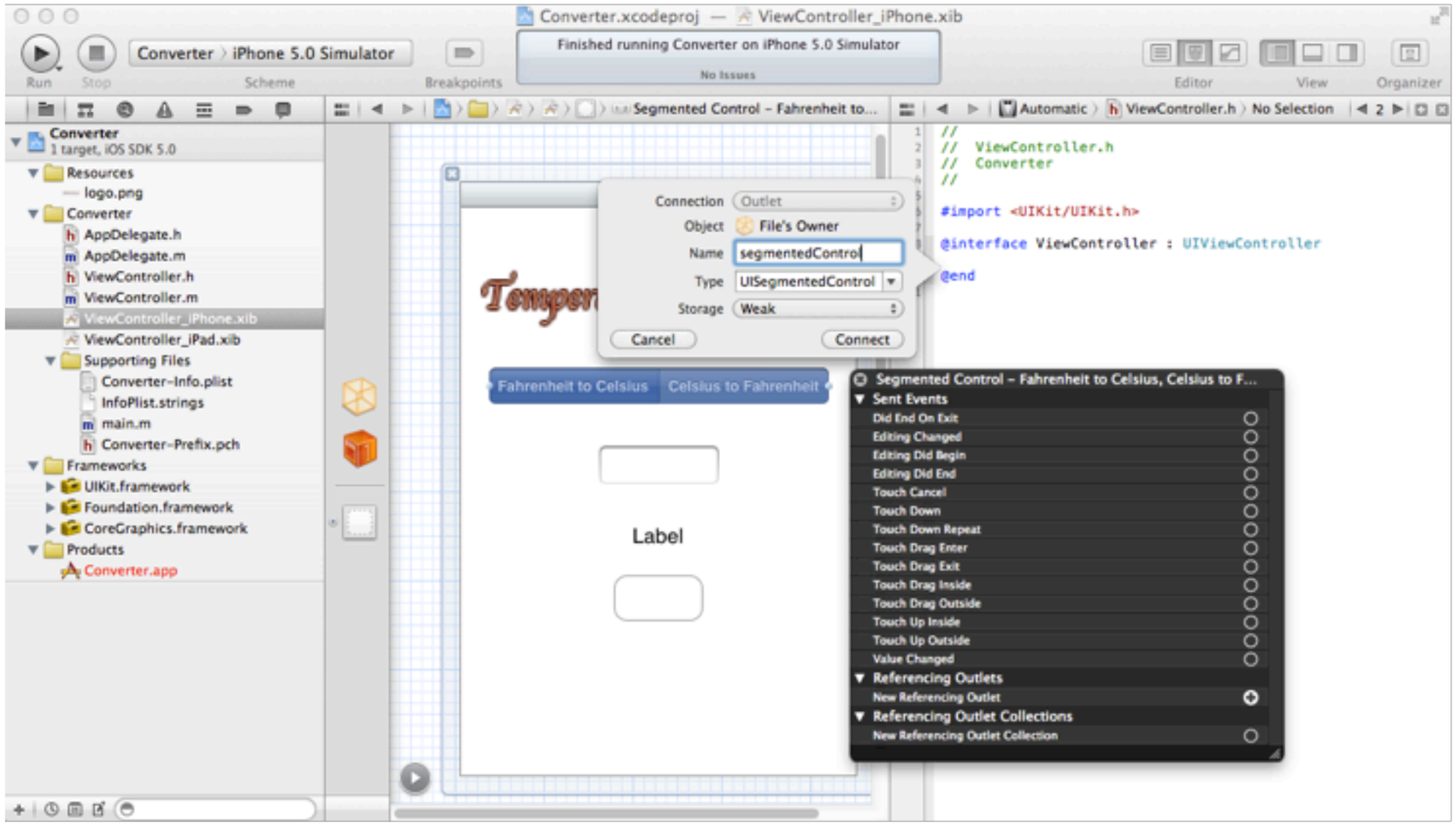

Figure 38 Creating the Outlet Connection for the Segmented Control

Notice the new declaration for the IBOutlet inserted in the following code.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@end
```

Repeat this same process of ctrl-clicking on an item in the View window then making a connection to the ViewController.h file for the text field and label. Name the outlet connection textField for the text field outlet and label the label label.
Now the ViewController.h file should appear similar to the code displayed below with IBOutlet properties for the segmented control, the text field and the label on the three lines above the end directive.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@property (weak, nonatomic) IBOutlet UITextField *textField;
@property (weak, nonatomic) IBOutlet UILabel *label;
@end
```

KEY POINT

Notice that the declarations for the UILabel, UITextField, and UISegmentedControl instance variables are preceded with the IBOutlet keyword. This keyword is used whenever a connection will be needed between the object in the code and the element in the user interface.

Synthesizing Properties and Memory Management

Navigate to the ViewController.m and notice the three synthesize directives near the top of the file as shown in the code below. Whenever properties are declared they must also be synthesized which creates accessor and mutator methods for these objects. In iOS 4.0 and later Xcode automatically adds the @property and the @synthesize directives for all reference outlets created as we have just done for the segmented control, text field, and label. Notice the synthesize directives inserted by Xcode into the ViewController.m file as shown in the following code.

```
//
//  ViewController.m
//  Converter
//

#import "ViewController.h"

@implementation ViewController
@synthesize segmentedControl;
@synthesize textField;
@synthesize label;
```

Xcode also inserted statements into the code that assigns a nil value to the references to the outlet connections as a result of the way that we created the outlet connections. Scroll down in the ViewController.m file and find the ViewDidUnload method. The first three statements in the ViewDidUnload method use an accessor method for each object to assign nil to those references in the following code.

```
- (void)viewDidUnload
{
    [self setSegmentedControl:nil];
    [self setTextField:nil];
    [self setLabel:nil];
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
```

Now that we have created connections between the button, text field, and label in the user interface to the ViewController class, when a user taps on the button we will be able to get the value of the text field and change the value of the label programmatically from within the code when running the application. Now we need to create a method to do that.

Creating User Interface Actions

In the previous section we created instances of the user interface elements, so that we can manipulate with their properties programmatically. Now it's time to create an action that will be triggered whenever a user taps on the Convert button on the screen. To do this, go back to the ViewController_iPhone.xib file and open the Assistant Editor for the ViewController.h file, and then ctrl-click on the Convert button. This time drag from the circle to the far right of the Touch Up Inside event to just above the end directive to insert an action. When you release the mouse, a Connection popup window will appear. Enter ***convertTemperature*** for the name of the action as shown in Figure 39, and then click on Connect. This will declare a new method for the class named convertTemperature that will execute whenever a user taps on the Convert Button. Notice the new declaration for the convertTemperature IBAction just above the end directive in the following code.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@property (weak, nonatomic) IBOutlet UITextField *textField;
@property (weak, nonatomic) IBOutlet UILabel *label;
- (IBAction)convertTemperature:(id)sender;
@end
```

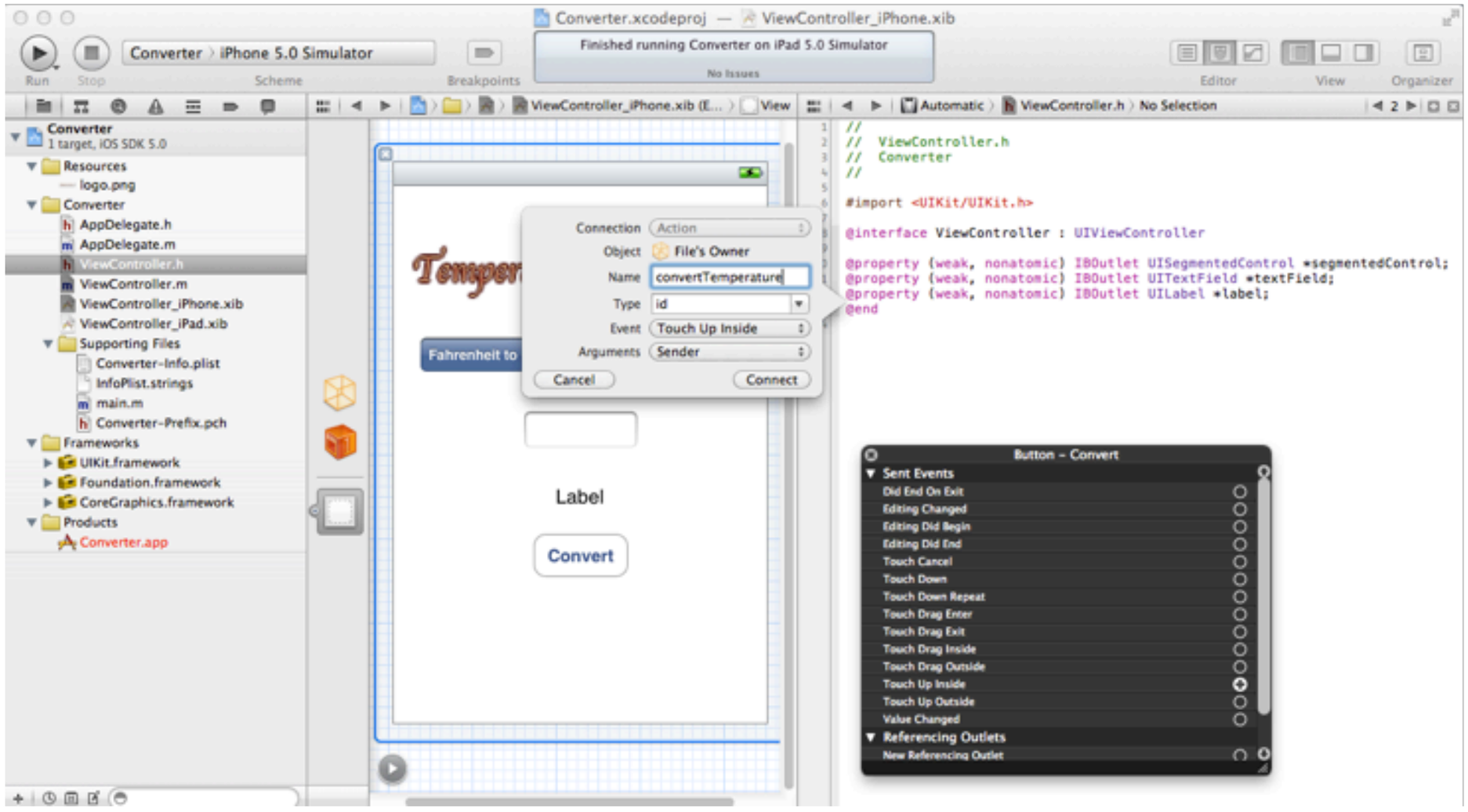

Figure 39 Creating the ConvertTemperature Action

Once the action is created, there will be a new method declaration for the action in the ViewController.h file and a new method definition in the ViewController.m file. The body of the convertTemperature method in the ViewController.m file is empty. It is up to us to add the custom code to perform the desired behavior for the button. Before we do that let's review the connections that we have made. Ctrl-click on the File's Owner icon in

the pane just to the left of the Editor area. A File's Owner window should popup similar to that shown in Figure 40. The Outlets part of the File's Owner window lists all the IBOutlets that have been created so far and the convertTemperature method listed under the Received Actions section of the File's Owner window.

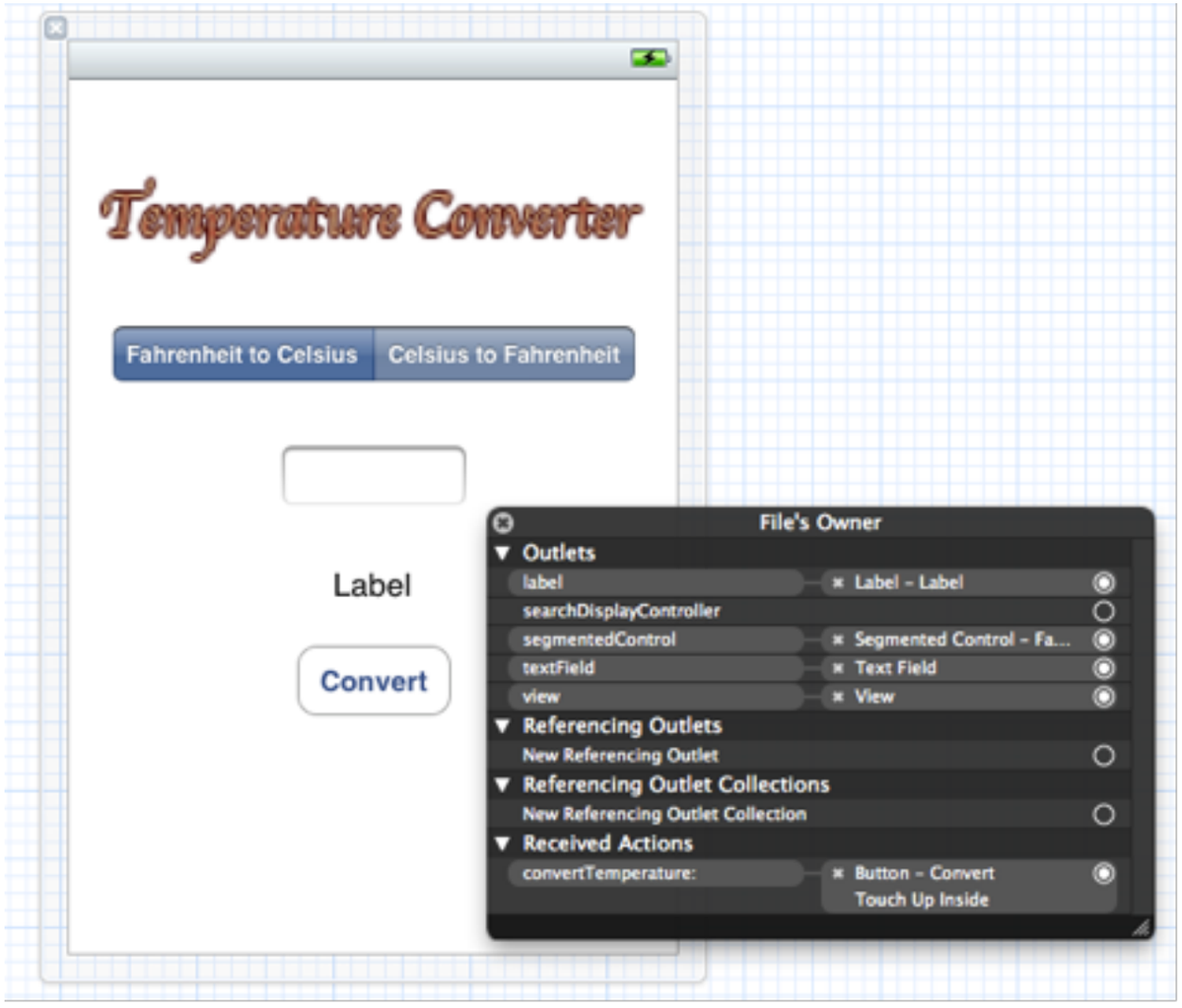

Figure 40 Outlets and Actions for the ConvertViewController

Adding Custom Code

So far Xcode has generated the code for us that is required for the outlet and action connections that we created for the elements in our user interface. Now we need to define the functionality we want the app to perform in order to perform temperature conversions inside the convertTemperature method in the ViewController.m file.

The first thing we will need is a variable to store the state of the segmented control so we know whether to convert the input the user enters into the text field from Celsius to Fahrenheit or from Fahrenheit to Celsius. Since there are only two segments in our segmented control we can define a boolean variable that be true if the first segment is selected so the value entered is a fahrenheit value that should be converted to celsius when the user taps the Convert button, or indicates will be false if the second segment is selected so the value entered into the text field is a celsius value that should be converted to fahrenheit. We can do this by adding the declaration for the fahrenheit variable shown in the code below. We also need a variable to store the value of the temperature the user enters into the text box, and the value of the converted temperature. Add the three declarations shown in the following code to the convertTemperature method in the ViewController.m file. Notice that once these declarations are added, yellow warning indicators popup in the left gutter of the Editor Area. Click on the triangles to see what the issues are. You will see that Xcode is informing you that these variables are not used in your program. This is a good type of warning because this situation often reveals a mistake by the programmer. However, in this case it is not a problem because we aren't done yet, and we will use these variables before we are done.

```
- (IBAction)convertTemperature:(id)sender
{
    BOOL fahrenheit;
    float userTemperature;
    float convertedTemperature;
}
@end
```

The next step is to add code that determines which segment is selected and then assign a boolean value to the fahrenheit variable as appropriate. Add the if-else construct that determines which as shown in the code segment below, just after the declarations. Notice that now that we have assigned values to the fahrenheit variable inside the if-else construct that there isn't a yellow warning symbol next to its declaration in Xcode anymore.

```
    if(segmentedControl.selectedSegmentIndex==0) {
        fahrenheit=FALSE;
```

```
}
else {
    fahrenheit=TRUE;
}
```

Now that we have code that determines what conversion the user desires, we can get the value from the text field and then perform the temperature conversion. The formulas for converting temperatures are as follows:

Centigrade = (5/9)(Fahrenheit-32) and Fahrenheit = (9/5)Centigrade + 32

First we get the value from the text field and assign it value to our conversion temperature variable. Notice that we get that value using the text property of our textField object and then convert it to a float value using the floatValue method so that we can assign it to our userTemperature variable. After this, we use an if-else construct to execute the proper formula for a temperature conversion depending on the value of the BOOL fahrenheit variable. Add the assignment and if-else construct in the following code segment after the code just previously added.

```
userTemperature=[textField.text floatValue];
if(fahrenheit) {
    convertedTemperature=((9.0/5.0)*userTemperature)+32;
}
else {
    convertedTemperature=(5.0/9.0)*(userTemperature-32);
}
```

The last two things we need to do in the convertTemperature method is to assign the value of the converted temperature to the text field on the screen, and then lower the keyboard after the user enters a temperature value and taps on the Convert button. In order to display the converted temperature in the text field we simply need to assign it to the text property of the label. But before we can do that we need to create a string that contains that value since the text property must be a string. We do this with the stringWithFormat method so that we can use a format specifier (%. 1f" to create a string that displays numbers with one place to the right of the decimal point, as in 32.5.

We also realize at this point that we have a minor problem. Once the user taps on the text field it becomes the first responder for the user interface and the keyboard animates from the bottom of the screen and covers the lower part of the screen and the keyboard covers the Convert button. Since we want to attach the method invocation that lowers the keyboard to the Convert button, we will need to rearrange the elements on our screen so that the Convert button is accessible while the keyboard is raised. We will do this once we are finished with the code for the convertTemperature method. Add the two statements in the following code segment to the end of the convertTemperature method.

```
label.text=[NSString stringWithFormat:@"%.1f", convertedTemperature];
```

```
    [textField resignFirstResponder];
}
@end
```

We have now added the custom code necessary to add the functionality for the temperature converter app. The complete method is as follows.

```
- (IBAction)convertTemperature:(id)sender
{
    BOOL fahrenheit;
    float userTemperature;
    float convertedTemperature;
    if(segmentedControl.selectedSegmentIndex==0) {
        fahrenheit=FALSE;
    }
    else {
        fahrenheit=TRUE;
    }
    userTemperature=[textField.text floatValue];
    if(fahrenheit) {
        convertedTemperature=((9.0/5.0)*userTemperature)+32;
    }
    else {
        convertedTemperature=(5.0/9.0)*(userTemperature-32);
    }
    label.text=[NSString stringWithFormat:@"%.1f", convertedTemperature];
    [textField resignFirstResponder];
}
@end
```

Now, go back to the View window for the user interface and rearrange the user interface elements so that the Convert button is positioned in the top part of the View window. You may also wish to adjust attributes in the Utilities area for the user interface elements to change colors, and fonts to suit your own preferences. We adjusted the attributes so that the final implementation of our app appears as shown in Figure 2.41.

Figure 41 Final Temperature Converter App in the iPhone Simulator

Save all the files and click on the Run button in Xcode to test your application. Now the application should behave as expected inside the iPhone simulator. Congratulations! There is one more thing that needs to be taken care of before we are finished which we will take care of now.

Universal Apps

Recall that we selected Universal for the device family when we first created this app. We have three choices for device family in Xcode 4. These choices are iPhone, iPad, and Universal. Apps that are developed specifically for the iPhone device family are generally optimized for the iPhone and iPod Touch platforms. These apps can be downloaded and run on iPads but don't generally provide the rich user experience that Apple intended for iPads. Developing Universal apps provides the greatest efficiency and maintainability for apps that incorporate best practices described in Apple's iOS Human Interface Guidelines. Apps developed specifically for the iPad don't run in the iPhone device family. Developers can create two completely different versions of the same app: one for the iPhone and one for the iPad. However, this can lead to a lot of redundancy and commonly result in maintainability difficulties.

The Universal device family allows developers to create one project for an app that maximizes the user experience uniquely for both the iPhone and the iPad. Since we chose the Universal device family when we created this Temperature Converter project. Xcode created two different View Controller xib files for our project. One view controller was named ViewController_iPhone.xib and one view controller was named ViewController_iPad.xib. So far we have only customized the ViewController_iPhone.xib file, next we need to customize the ViewController_iPad.xib file. But before we do, let's look at the mechanism within the app that facilitates the Universal app. Select the AppDelegate.m file in the Navigator area of Xcode and display this file in the Editor area and then find the didFinishLaunchingWithOptions method, as follows.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        self.viewController = [[ViewController alloc] initWithNibName:@"ViewController_iPhone" bundle:nil];
    } else {
        self.viewController = [[ViewController alloc] initWithNibName:@"ViewController_iPad" bundle:nil];
    }
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

The first thing this app does when it is launched is that it checks what kind of a device the app is running on. If the app is an iPhone then it sets the view controller to the ViewController_iPhone.xib file, otherwise it sets the view controller to the ViewController_iPad.xib file. We added simple elements to the ViewController.xib file and created IBOutlet and IBAction connections to the ViewController.h and ViewController.m files just as we had for the iPhone interface. We used the convention of using the same names for these elements as we had previously but with iPad appended to the first part of the name. Then we copied the convertTemperature method that we used for the iPhone interface and customized it for the iPad interface. After making similar changes, select the iPad 5.0 Simulator as the scheme and then click on the Run button to test your app in the iPad simulator. The screenshot for our app in the iPad simulator is shown in Figure 2.42. The complete code for this universal app is available on the companion website for this book.

Figure 42 Temperature Converter in the iPad Simulator

Summary

In this chapter we have learned about basic concepts of iOS programming and created a simple iPhone application. First we learned how to create and use Xcode templates, specifically we used the View Based Application template. In the latter part of the chapter we described the following user interface elements, a UIImageView element for displaying images, a UIButton element for facilitating execution of a given method when a button is touched by a user, a UILabel element for displaying text on the screen, a UISegmentedControl element for providing alternative options, and a UITextField element for receiving user input. The general process of incorporating user interface controls in an application is as follows:

1. Drag the UI controls to the iPhone View window for the .xib file.

2. Set the attributes for the elements in the Utilities panel in Xcode.

3. If access to the attributes of the user interface controls is necessary within the code then create Outlet connections and between the .xib and .h files

4. Provide the code inside the appropriate method in the .m file that changes the appearance or behavior of the UI control or user interface element.

Another important concept covered in this chapter was the delegation pattern. Delegation is a simple mechanism that allows the developer to assign responsibilities of one object to another object. For the Temperature Converter application the text field sends notifications to its delegate, the ViewController to resign control and lower the keyboard when the return button on the keyboard is tapped.

Review Questions

1. What is an IBOutlet? Why is it used? How is it used?

2. Which class is used to display text on an iPhone screen? Which property of the class is used to update the text?

3. What is an IBAction and why is it used?

4. Which event is used to implement the functionality of a button?

5. How are connections between user interface elements and the code created? Describe the steps.

6. Which property of the UISegmentedControl is used to check which segment is currently selected?

7. Which delegate method is used to lower a keyboard off the screen?

8. How do you relinquish ownership of an object? Which method should you use to perform that action?

9. What does the resignFirstResponder method do?

Exercises

1. Write a short program that displays an image on the screen.

2. Create a program that displays the following UI elements on the screen -- customize the appearance of all of the UI elements:

    a. UISegmentedControl

    b. UIButton

    c. UILabel

    d. UISlider

    e. UITextField

3. Modify the application developed in this chapter to convert pounds to kilograms, kilograms to pounds, meters to feet, and feet to meters.

4. Develop a simple calculator with the capability to add and subtract numbers.