iOS 7 App Development

Essentials

iOS 7 App Development Essentials

Developing iOS 7 Apps for the iPhone and iPad

Neil Smyth

This book is for sale at http://leanpub.com/ios7devessentials

This version was published on 2013-11-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Start Here

When the first iPhone was launched in 2007 it was largely dismissed as a potential threat by the incumbent handset manufacturers of the time. Similarly, the launch of the iPad three years later appeared to be of little concern to the giants of the Personal Computer industry. The reasoning at the time was that a tablet was simply a large smartphone that targeted an entirely different market to that of desktop PCs and laptops. The assumption was also made that if tablets became successful, Microsoft would simply find a way to dominate the market with a tablet-friendly version of Windows running on tablets from companies like Dell and HP.

Fast forward to today and consider just a handful of news headlines from the summer of 2013:

"BlackBerry goes up for sale after years of struggle in smartphone market". - The Guardian

"*BlackBerry to lay off up to 40% of its workforce.*" - Wall Street Journal

"Microsoft Takes $900 Million Write-off on Tablet". - Wall Street Journal

"Microsoft chief Steve Ballmer to retire within 12 months." - BBC News

"Microsoft to Buy Nokia's Device Unit." - Bloomberg

"Dell's Profit Declines 72% on Sluggish Sales of PCs." - New York Times

"HP posts revenue decline as PC sales weaken further." - VentureBeat

Not so long ago such headlines would have been unthinkable given the former success and market share held by the companies in question. What is particularly notable about these headlines, however, is that the events that prompted these news stories can all be traced back, at least in part, to one cause - the continued success of tablets and smartphones. The real problem for the companies mentioned in the headlines, however, is that the majority of these devices are not running operating systems from Microsoft, Nokia or Blackberry. They are, instead, predominantly running either iOS or Android.

As of June 2013, Apple had sold more than 400 million iOS based devices worldwide in the form of iPhones and iPads. The fact is, the technology landscape has changed dramatically in recent years and if you aren't writing apps for iOS yet, you probably should think about starting.

The goal of this book is to get you up to speed on both iOS development in general and the new features of the iOS 7 SDK.

How you make use of this book will depend to a large extent on whether you are new to iOS development, or have worked with iOS 6 and need to get up to speed on the features of iOS 7. Rest assured, however, that the book is intended to address both category of reader.

1.1 For New iOS Developers

If you are entirely new to iOS development then the entire contents of the book will be relevant to you.

Beginning with the basics, this book provides an outline of the steps necessary to set up an iOS development environment. An introduction to the architecture of iOS 7 and programming in Objective-C is provided, followed by an in-depth look at the design of iOS applications and user interfaces. More advanced topics such as file handling, database management, in-app purchases, graphics drawing and animation are also covered, as are touch screen handling, gesture recognition, multitasking, iAds integration, location management, local notifications, camera access and video and audio playback support. Other features are also covered including Auto Layout, Twitter and Facebook integration, event reminders, App Store hosted in-app purchase content, collection views and much more. New features of iOS 7 are also covered, including Sprite Kit-based game development, local map search and user interface animation using UIKit dynamics.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 7. Assuming you are ready to download the iOS 7 SDK and Xcode, have an Intel-based Mac and some ideas for some apps to develop, you are ready to get started.

1.2 For iOS 6 Developers

If you have already read the iPhone or iPad editions of iOS 6 Development Essentials, or have experience with the iOS 6 SDK then you might prefer to go directly to the new chapters in this iOS 7 edition of the book.

All chapters have been updated to reflect the changes and features introduced as part of iOS 7 and Xcode 5. Chapters included in this edition that were not contained in the previous edition, or have been significantly rewritten for iOS 7 and Xcode 5 are as follows:

· *Testing Apps on iOS 7 Devices with Xcode 5*

· *Working with iOS 7 Auto Layout Constraints in Interface Builder*

· *An iOS 7 Auto Layout Example*

· *iOS 7 UIKit Dynamics - An Overview*

· *An iOS 7 UIKit Dynamics Tutorial*

· *An Introduction to iOS 7 Sprite Kit Programming*

· *An iOS 7 Sprite Kit Game Tutorial*

· *An iOS 7 Sprite Kit Collision Handling Tutorial*

· *An iOS 7 Particle Emitter Tutorial*

· *Integrating iAds into an iOS 7 App*

· *iOS 7 Multitasking, Background Transfer Service and Fetching*

· *An iOS 7 Background Transfer Service Tutorial*

· *Working with MapKit Local Search in iOS 7*

· *Using MKDirections to get iOS 7 Map Directions and Routes*

· *Preparing and Submitting an iOS 7 Application to the App Store*

· *Promoting your iOS Apps using iAd Workbench*

In addition the following changes have also been made:

· All provisioning examples have been updated to use the new Capabilities settings panel in Xcode 5.

· All code examples have been modified were necessary for compatibility with both the 32-bit and 64-bit CPU architectures.

· *An iPad iOS 7 Split View and Popover Example* has been rewritten to use Storyboard instead of NIB files.

· *Creating a Simple iOS 7 App* has been updated to cover the new Preview Assistant and performance monitoring features of Xcode 5.

· *An iOS 7 Graphics Tutorial using Core Graphics and Core Image* chapter has been extended to include coverage of features such as shadowing and gradients.

· *Basic iOS 7 Animation using Core Animation* has also been updated to use the animation block methods approach to animating user interface objects.

1.3 Source Code Download

The source code and Xcode project files for the examples contained in this book are available for download at:

http://www.ebookfrenzy.com/retail/ios7/

1.4 Feedback

We want you to be satisfied with your purchase of this book. If you find any errors in the book, or have any comments, questions or concerns please contact us at *feedback@ebookfrenzy.com*.

1.5 Errata

Whilst we make every effort to ensure the accuracy of the content of this book, it is inevitable that a book covering a subject area of this size and complexity may include some errors and oversights. Any known issues with the book will be outlined, together with solutions at the following URL:

http://www.ebookfrenzy.com/errata/ios7.html

In the event that you find an error not listed in the errata, please let us know by emailing our technical support team at *feedback@ebookfrenzy.com*.

2. Joining the Apple iOS Developer Program

The first step in the process of learning to develop iOS 7 based applications involves gaining an understanding of the differences between *Registered Apple Developers* and *iOS Developer Program Members*. Having gained such an understanding, the next choice is to decide the point at which it makes sense for you to pay to join the iOS Developer Program. With these goals in mind, this chapter will cover the differences between the two categories of developer, outline the costs and benefits of joining the developer program and, finally, walk through the steps involved in obtaining each membership level.

2.1 Registered Apple Developer

There is no fee associated with becoming a registered Apple developer. Simply visit the following web page to begin the registration process:

http://developer.apple.com/programs/register/

An existing Apple ID (used for making iTunes or Apple Store purchases) is usually adequate to complete the registration process.

Once the registration process is complete, access is provided to developer resources such as online documentation and tutorials. Registered developers are also able to download older versions of the iOS SDK and Xcode development environment.

2.2 Downloading Xcode and the iOS 7 SDK

The latest versions of both the iOS SDK and Xcode can be downloaded free of charge from the Mac App Store. Since the tools are free, this raises the question of whether to upgrade to the iOS Developer Program, or to remain as a Registered Apple Developer. It is important, therefore, to understand the key benefits of the iOS Developer Program.

2.3 iOS Developer Program

Membership in the iOS Developer Program currently costs $99 per year. As previously mentioned, membership includes access to the latest versions of the iOS SDK and Xcode development environment. The benefits of membership, however, go far beyond those offered at the Registered Apple Developer level.

One of the key advantages of the developer program is that it permits the creation of certificates and provisioning profiles to test applications on physical devices. Although Xcode includes device simulators which allow for a significant amount of testing to be performed, there are certain areas of functionality, such as location tracking and device motion, which can only fully be tested on a physical device. Of particular significance is the fact that iCloud access, Reminders and In-App Purchasing can only be tested when applications are running on physical devices.

Of further significance is the fact that iOS Developer Program members have unrestricted access to the full range of guides and tutorials relating to the latest iOS SDK and, more importantly, have access to technical support from Apple's iOS technical support engineers (though the annual fee covers the submission of only two support incident reports).

By far the most important aspect of the iOS Developer Program is that membership is a mandatory requirement in order to publish an application for sale or download in the App Store.

Clearly, developer program membership is going to be required at some point before your application reaches the App Store. The only question remaining is when exactly to sign up.

2.4 When to Enroll in the iOS Developer Program?

Clearly, there are many benefits to iOS Developer Program membership and, eventually, membership will be necessary to begin selling applications. As to whether or not to pay the enrollment fee now or later will depend on individual circumstances. If you are still in the early stages of learning to develop iOS applications or have yet to come up with a compelling idea for an application to develop then much of what you need is provided in the Registered Apple Developer package. As your skill level increases and your ideas for applications to develop take shape you can, after all, always enroll in the developer program at a later date.

If, on the other hand, you are confident that you will reach the stage of having an application ready to publish or know that you will need to test the functionality of the application on a physical device as opposed to a simulator then it is worth joining the developer program sooner rather than later.

2.5 Enrolling in the iOS Developer Program

If your goal is to develop iOS applications for your employer then it is first worth checking whether the company already has membership. That being the case, contact the program administrator in your company and ask them to send you an invitation from within the iOS Developer Program Member Center to join the team. Once they have done so, Apple will send you an email entitled *You Have Been Invited to Join an Apple Developer Program* containing a link to activate your membership. If you or your company is not already a program member, you can enroll online at:

http://developer.apple.com/programs/ios/

Apple provides enrollment options for businesses and individuals. To enroll as an individual you will need to provide credit card information in order to verify your identity. To enroll as a company

you must have legal signature authority (or access to someone who does) and be able to provide documentation such as Articles of Incorporation and a Business License.

Acceptance into the developer program as an individual member typically takes less than 24 hours with notification arriving in the form of an activation email from Apple. Enrollment as a company can take considerably longer (sometimes weeks or even months) due to the burden of the additional verification requirements.

Whilst awaiting activation you may log into the Member Center with restricted access using your Apple ID and password at the following URL:

http://developer.apple.com/membercenter

Once logged in, clicking on the *Your Account* tab at the top of the page will display the prevailing status of your application to join the developer program as *Enrollment Pending*:

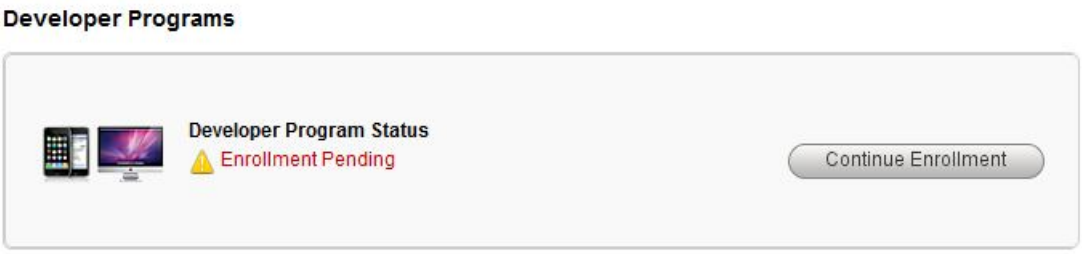

Figure 2-1

Once the activation email has arrived, log into the Member Center again and note that access is now available to a wide range of options and resources as illustrated in Figure 2-2:

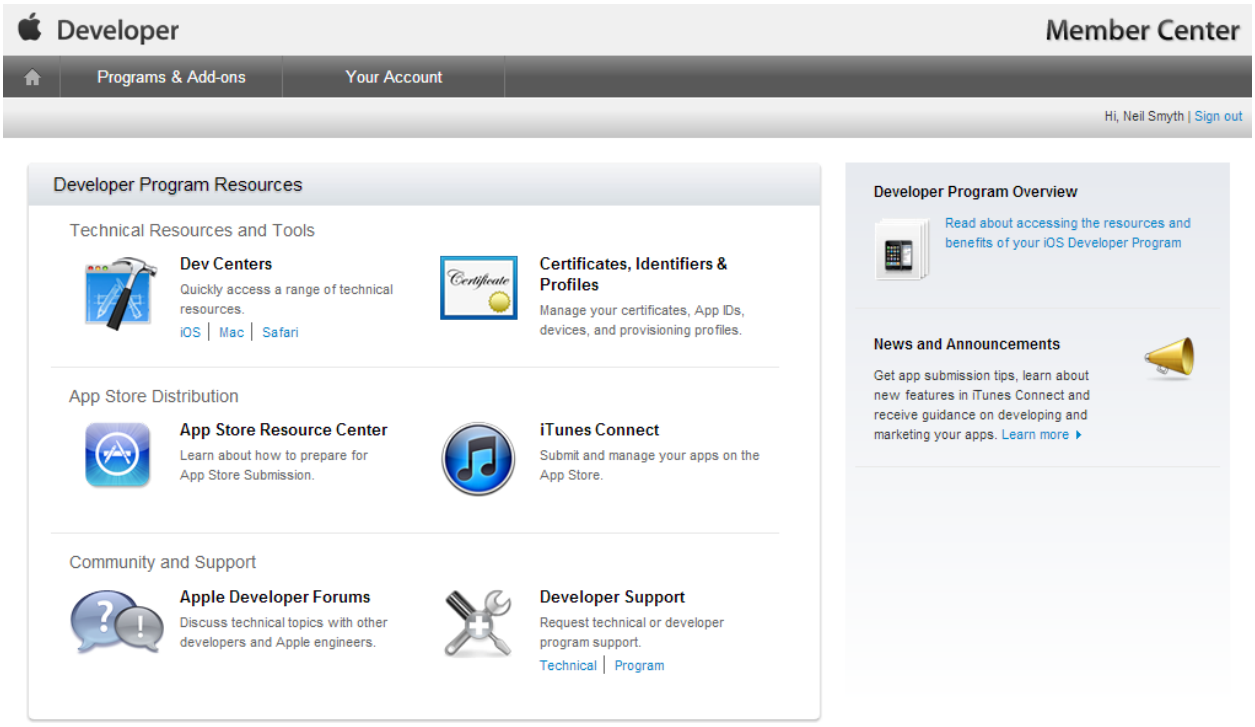

Figure 2-2

2.6 Summary

An important early step in the iOS 7 application development process involves registering as an Apple Developer and identifying the best time to upgrade to iOS Developer Program membership. This chapter has outlined the differences between the two programs, provided some guidance to keep in mind when considering developer program membership and walked briefly through the enrollment process. The next step is to download and install the iOS 7 SDK and Xcode 5 development environment.

3. Installing Xcode 5 and the iOS 7 SDK

iPhone apps are developed using the iOS SDK in conjunction with Apple's Xcode 5.x development environment. The iOS SDK contains the development frameworks that will be outlined in *iOS 7 Architecture and SDK Frameworks*. Xcode 5 is an integrated development environment (IDE) within which you will code, compile, test and debug your iOS applications. The Xcode 5 environment also includes a feature called Interface Builder which enables you to graphically design the user interface of your application using the components provided by the UIKit framework.

In this chapter we will cover the steps involved in installing both Xcode 5 and the iOS 7 SDK on Mac OS X.

3.1 Identifying if you have an Intel or PowerPC based Mac

Only Intel based Mac OS X systems can be used to develop applications for iOS. If you have an older, PowerPC based Mac then you will need to purchase a new system before you can begin your iOS app development project. If you are unsure of the processor type inside your Mac, you can find this information by clicking on the Apple menu in the top left hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog check the *Processor* line. Figure 3-1 illustrates the results obtained on an Intel based system.

If the dialog on your Mac does not reflect the presence of an Intel based processor then your current system is, sadly, unsuitable as a platform for iOS app development.

In addition, the iOS 7 SDK with Xcode 5 environment requires that the version of Mac OS X running on the system be version 10.8 or later. If the "About This Mac" dialog does not indicate that Mac OS X 10.7.4 or later is running, click on the *Software Update...* button to download and install the appropriate operating system upgrades.

Figure 3-1

3.2 Installing Xcode 5 and the iOS 7 SDK

The best way to obtain the latest versions of Xcode and the iOS SDK is to download them from the Apple Mac App Store. Launch the App Store on your Mac OS X system and, enter Xcode into the search box and click on the *Free* button to initiate the installation.

The download is over 1.6GB in size and may take a number of hours to complete depending on the speed of your internet connection.

3.3 Starting Xcode

Having successfully installed the SDK and Xcode, the next step is to launch it so that we can write and then create a sample iOS 7 application. To start up Xcode, open the Finder and search for *Xcode*. Since you will be making frequent use of this tool take this opportunity to drag and drop it into your dock for easier access in the future. Click on the Xcode icon in the dock to launch the tool. The first time Xcode runs you may be prompted to install additional components. Follow these steps, entering your username and password when prompted to do so.

Once Xcode has loaded, and assuming this is the first time you have used Xcode on this system, you will be presented with the *Welcome* screen from which you are ready to proceed:

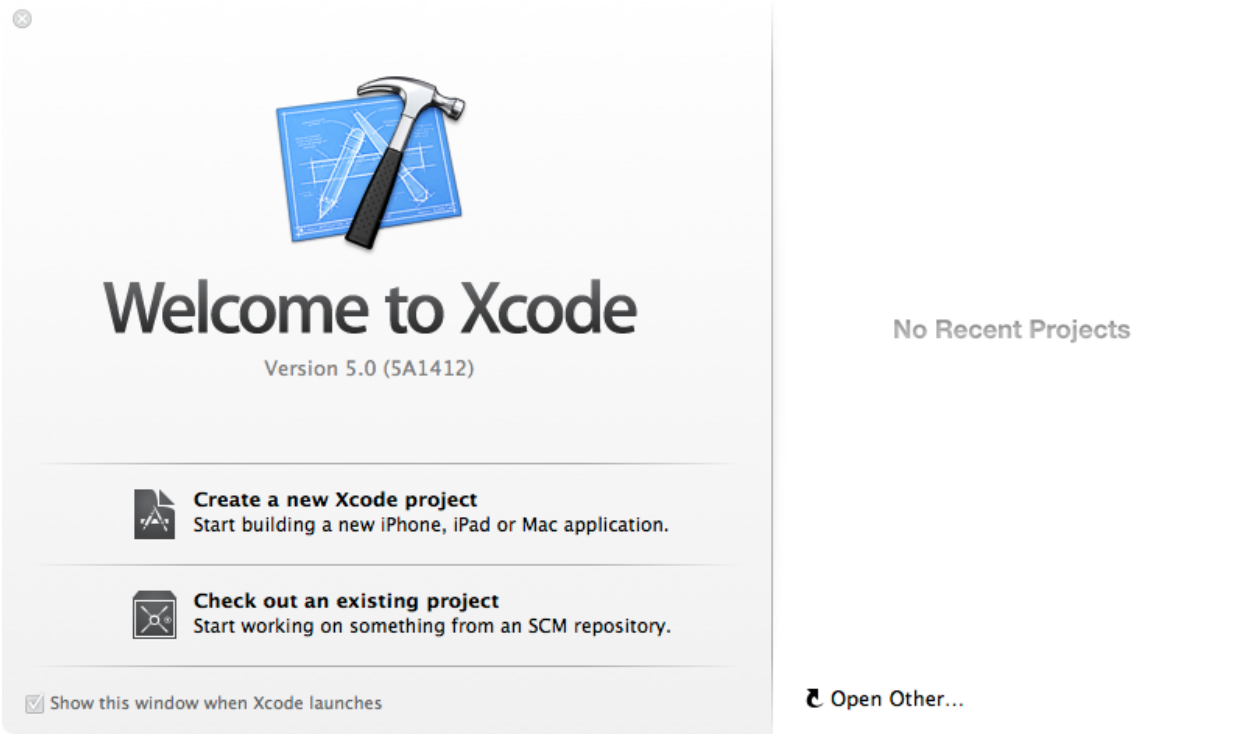

Figure 3-2

Having installed the iOS 7 SDK and successfully launched Xcode 5 we can now look at *Creating a Simple iOS 7 App*.

4. Creating a Simple iOS 7 App

It is traditional in books covering programming topics to provide a very simple example early on. This practice, though still common, has been maligned by some authors of recent books. Those authors, however, are missing the point of the simple example. One key purpose of such an exercise is to provide a very simple way to verify that your development environment is correctly installed and fully operational before moving on to more complex tasks. A secondary objective is to give the reader a quick success very early in the learning curve to inspire an initial level of confidence. There is very little to be gained by plunging into complex examples that confuse the reader before having taken time to explain the underlying concepts of the technology.

With this in mind, *iOS 7 App Development Essentials* will remain true to tradition and provide a very simple example with which to get started. In doing so, we will also be honoring another time honored tradition by providing this example in the form of a simple "Hello World" program. The "Hello World" example was first used in a book called the C Programming Language written by the creators of C, Brian Kernighan and Dennis Richie. Given that the origins of Objective-C can be traced back to the C programming language it is only fitting that we use this example for iOS 7.

4.1 Starting Xcode 5

As with all iOS examples in this book, the development of our example will take place within the Xcode 5 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 5 and the iOS 7 SDK* chapter of this book. Assuming that the installation is complete, launch Xcode either by clicking on the icon on the dock (assuming you created one) or use the Finder to locate the Xcode binary.

When launched for the first time, and until you turn off the *Show this window when Xcode launches* toggle, the screen illustrated in Figure 4-1 will appear by default:

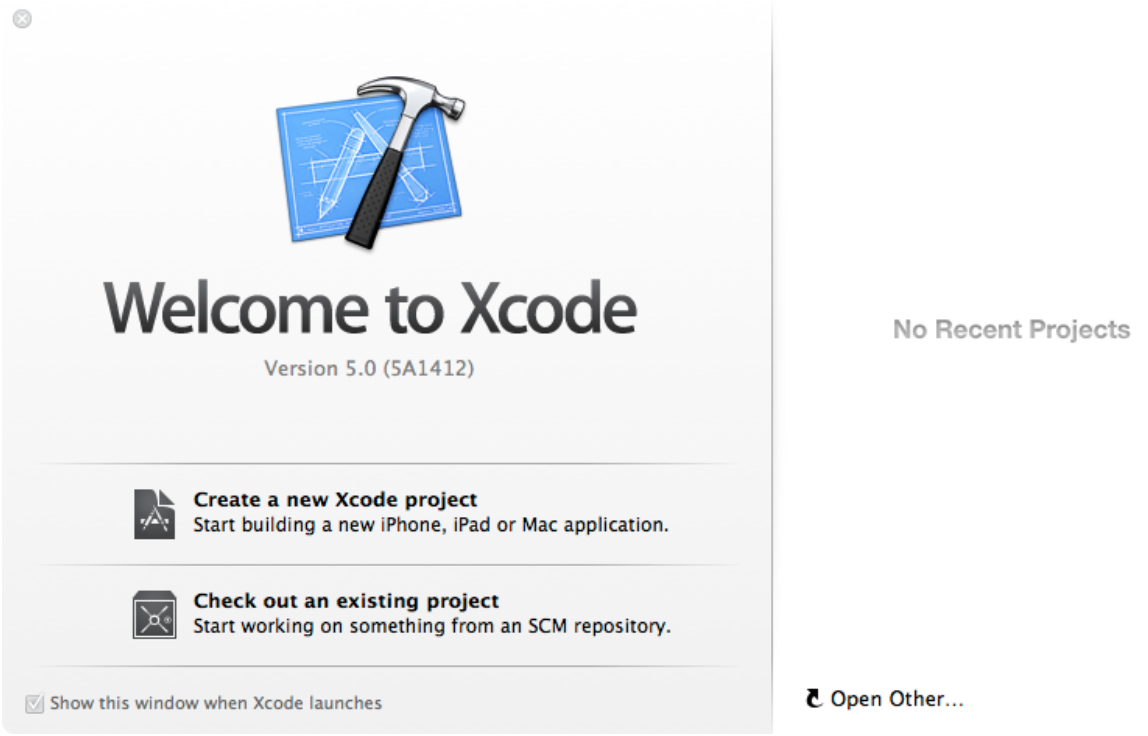

Figure 4-1

If you do not see this window, simply select the *Window -> Welcome to Xcode* menu option to display it. From within this window click on the option to *Create a new Xcode project*. This will display the main Xcode 5 project window together with the *New Project* panel where we are able to select a template matching the type of project we want to develop:

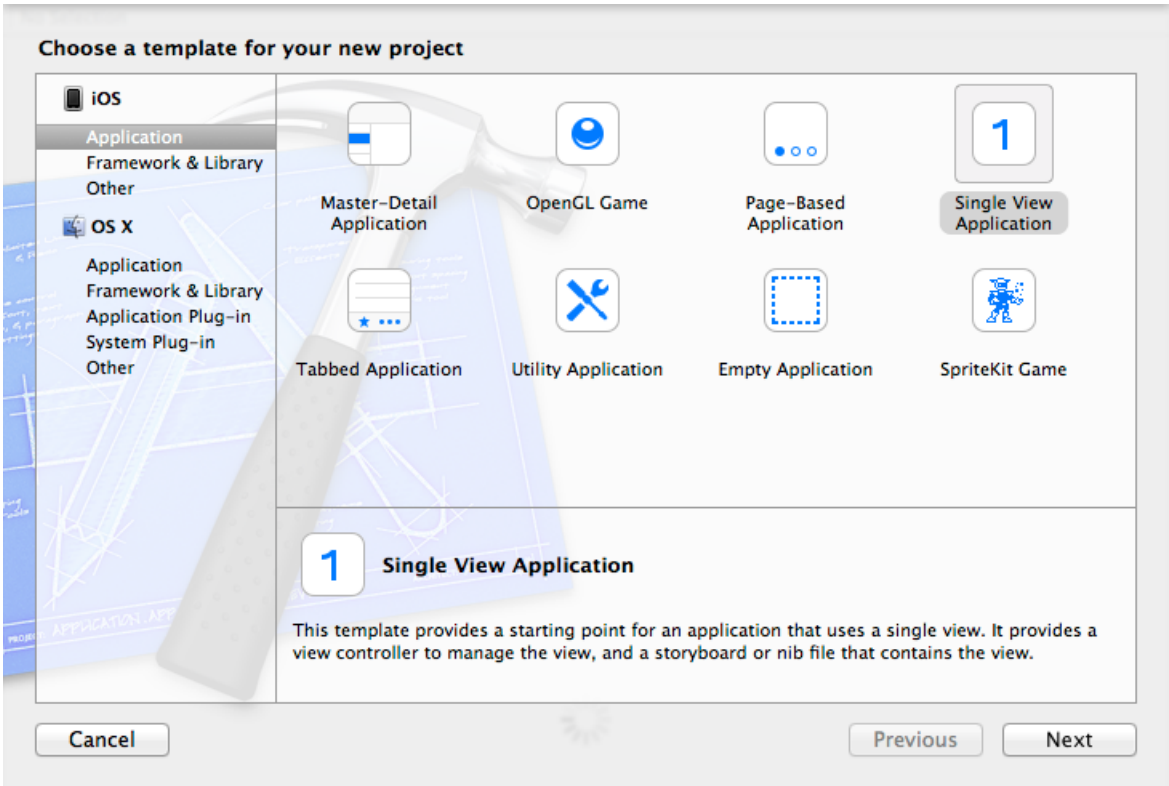

Figure 4-2

The panel located on the left hand side of the window allows for the selection of the target platform, providing options to develop an application either for iOS based devices or Mac OS X.

Begin by making sure that the *Application* option located beneath *iOS* is selected. The main panel contains a list of templates available to use as the basis for an application. The options available are as follows:

· **Master-Detail Application** - Used to create a list based application. Selecting an item from a master list displays a detail view corresponding to the selection. The template then provides a *Back* button to return to the list. You may have seen a similar technique used for news based applications, whereby selecting an item from a list of headlines displays the content of the corresponding news article. When used for an iPad based application this template implements a basic split-view configuration.

· **OpenGL Game** - The OpenGL ES framework provides an API for developing advanced graphics drawing and animation capabilities. The OpenGL ES Game template creates a basic application containing an OpenGL ES view upon which to draw and manipulate graphics, and a timer object.

· **Page-based Application** - Creates a template project using the page view controller designed to allow views to be transitioned by turning pages on the screen.

· **Tabbed Application** - Creates a template application with a tab bar. The tab bar typically appears across the bottom of the device display and can be programmed to contain items that, when selected, change the main display to different views. The iPhone's built-in *Phone* user interface, for example,

uses a tab bar to allow the user to move between favorites, contacts, keypad and voicemail.

· **Utility Application** - For iPhone projects, this option creates a template consisting of a two sided view. Pressing an Info button flips the view to the second view. For iPad based projects, an Info bar is created which, when selected, displays the second view in a popover.

· **Single View Application** - Creates a basic template for an application containing a single view and corresponding view controller.

· **Empty Application** - This most basic of templates creates only a window and a delegate. If none of the above templates match your requirements then this is the option to take.

· **SpriteKit Game** - Creates a project configured to take advantage of the Sprite Kit Framework for the development of 2D games.

For the purposes of our simple example, we are going to use the *Single View Application* template so select this option from the new project window and click *Next* to configure some project options:

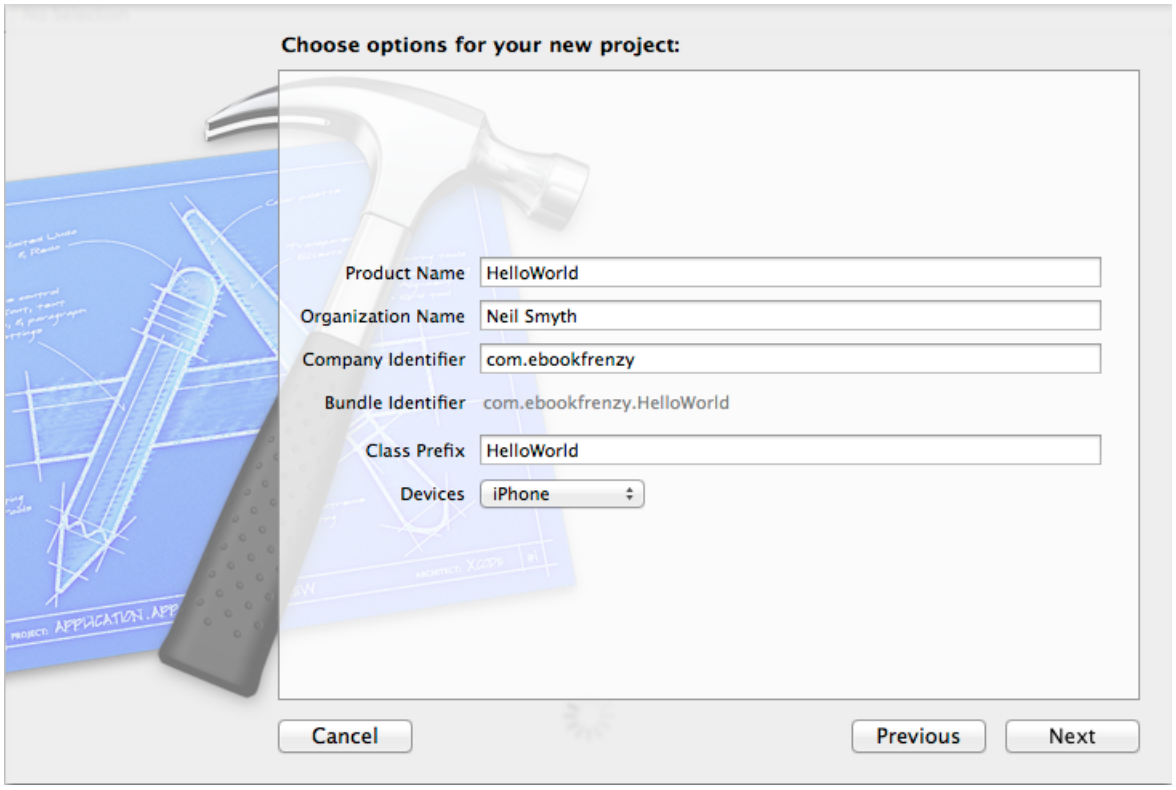

Figure 4-3

On this screen, enter a Product name for the application that is going to be created, in this case "HelloWorld". The company identifier is typically the reversed URL of your company's website, for example "com.mycompany". This will be used when creating provisioning profiles and certificates to enable applications to be tested on a physical iPhone or iPad device (covered in more detail in *Testing Apps on iOS 7 Devices with Xcode 5*). Enter the *Class Prefix* value of "HelloWorld" which will be used to prefix any classes created for us by Xcode when the template project is created.

Make sure that *iPhone* is currently selected from the *Devices* menu before clicking the *Next* button to proceed. On the final screen, choose a location on the file system for the new project to be created and click on *Create.*

Once the new project has been created, the main Xcode window will appear as illustrated in Figure 4-4:

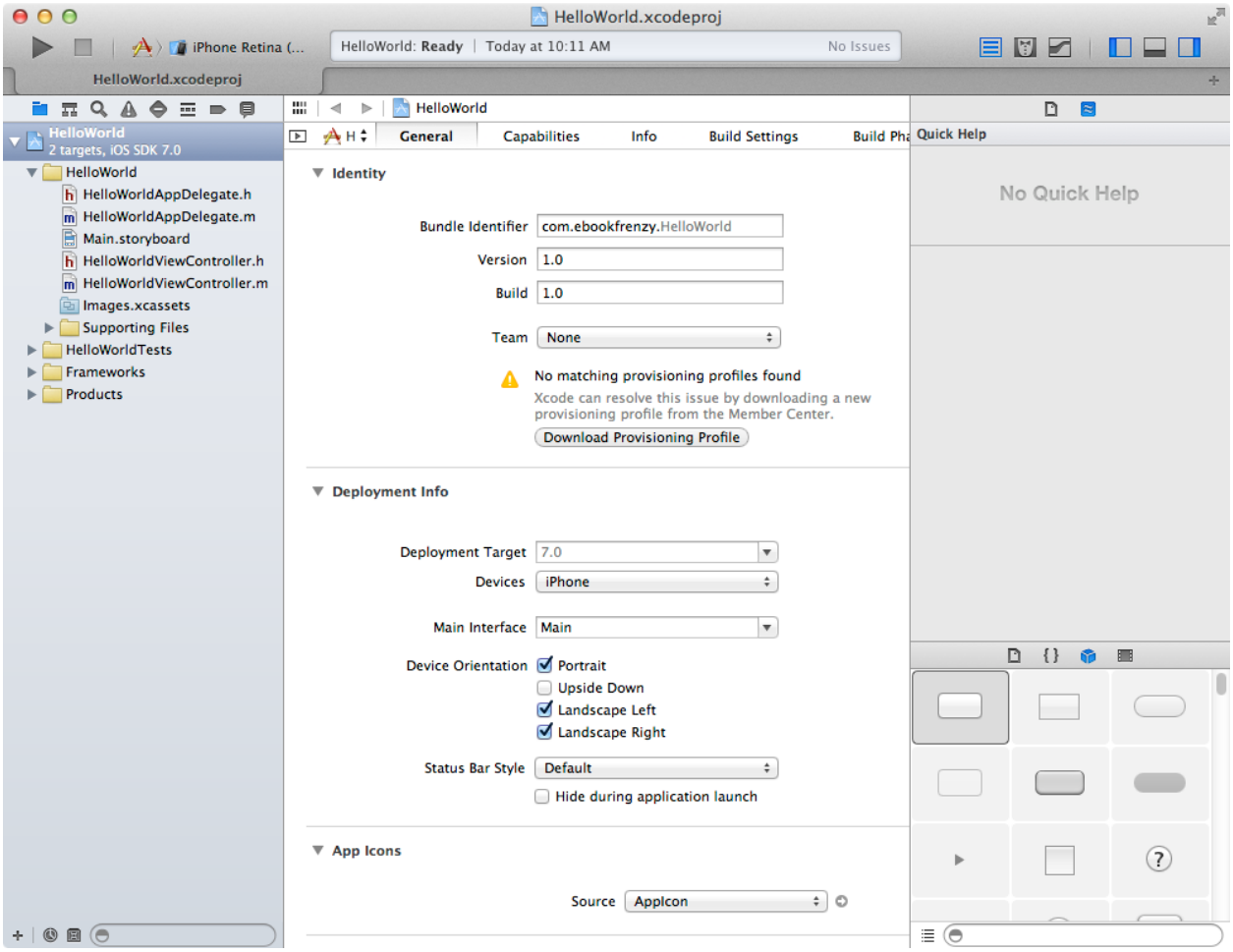

Figure 4-4

Before proceeding we should take some time to look at what Xcode has done for us. Firstly it has created a group of files that we will need to create our application. Some of these are Objective-C source code files (with a .m extension) where we will enter the code to make our application work, others are header or interface files (.h) that are included by the source files and are where we will also need to put our own declarations and definitions. In addition, the .storyboard file is the save file used by the Interface Builder tool to hold the user interface design we will create. Also present will be one or more files with a .plist file extension. These are *Property List* files which contain key/value pair information. For example, the *HelloWorld-info.plist* file contains resource settings relating to items such as the language, icon file, executable name and app identifier. The list of files is displayed in the *Project Navigator* located in the left hand panel of the main Xcode project window. A toolbar

at the top of this panel contains options to display other information such as build and run history, breakpoints and compilation errors.

By default, the center panel of the window shows a general summary of the settings for the application project. This includes the identifier specified during the project creation process and the target device. Options are also provided to configure the orientations of the device that are to be supported by the application together with options to upload icons (the small images the user selects on the device screen to launch the application) and launch screen images (displayed to the user while the application loads) for the application.

In addition to the General screen, tabs are provided to view and modify additional settings consisting of Capabilities, Info, Build Settings, Build Phases and Build Rules. As we progress through subsequent chapters of this book we will explore some of these other configuration options in greater detail. To return to the project settings panel at any future point in time, make sure the *Project Navigator* is selected in the left hand panel and select the top item (the application name) in the navigator list.

When a source file is selected from the list in the navigator panel, the contents of that file will appear in the center panel where it may then be edited. To open the file in a separate editing window, simply double click on the file in the list.

4.2 Creating the iOS App User Interface

Simply by the very nature of the environment in which they run, iOS apps are typically visually oriented. As such, a key component of just about any app involves a user interface through which the user will interact with the application and, in turn, receive feedback. Whilst it is possible to develop user interfaces by writing code to create and position items on the screen, this is a complex and error prone process. In recognition of this, Apple provides a tool called Interface Builder which allows a user interface to be visually constructed by dragging and dropping components onto a canvas and setting properties to configure the appearance and behavior of those components. Interface Builder was originally developed some time ago for creating Mac OS X applications, but has now been updated to allow for the design of iOS app user interfaces.

As mentioned in the preceding section, Xcode pre-created a number of files for our project, one of which has a .storyboard filename extension. This is an Interface Builder storyboard save file and the file we are interested in for our HelloWorld project is named *Main.storyboard.* To load this file into Interface Builder simply select the file name in the list in the left hand panel. Interface Builder will subsequently appear in the center panel as shown in Figure 4-5:

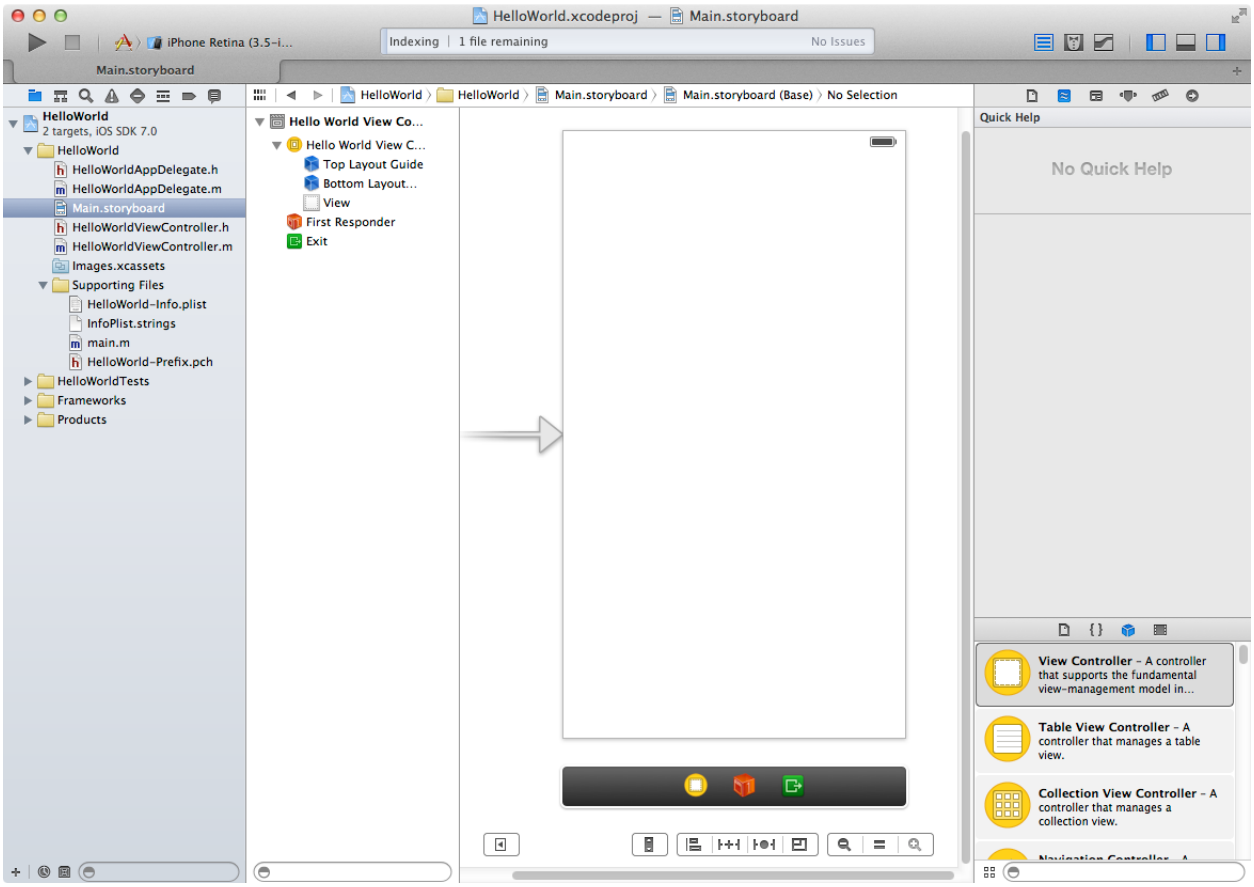

Figure 4-5

In the center panel a visual representation of the user interface of the application is displayed. Initially this consists solely of the UIView object. This *UIView* object was added to our design by Xcode when we selected the Single View Application option during the project creation phase. We will construct the user interface for our HelloWorld app by dragging and dropping user interface objects onto this UIView object. Designing a user interface consists primarily of dragging and dropping visual components onto the canvas and setting a range of properties and settings. In order to access objects and property settings it is necessary to display the Xcode right hand panel (if it is not already displayed). This is achieved by selecting the right hand button in the *View* section of the Xcode toolbar:

Figure 4-6

The right hand panel, once displayed, will appear as illustrated in Figure 4-7:

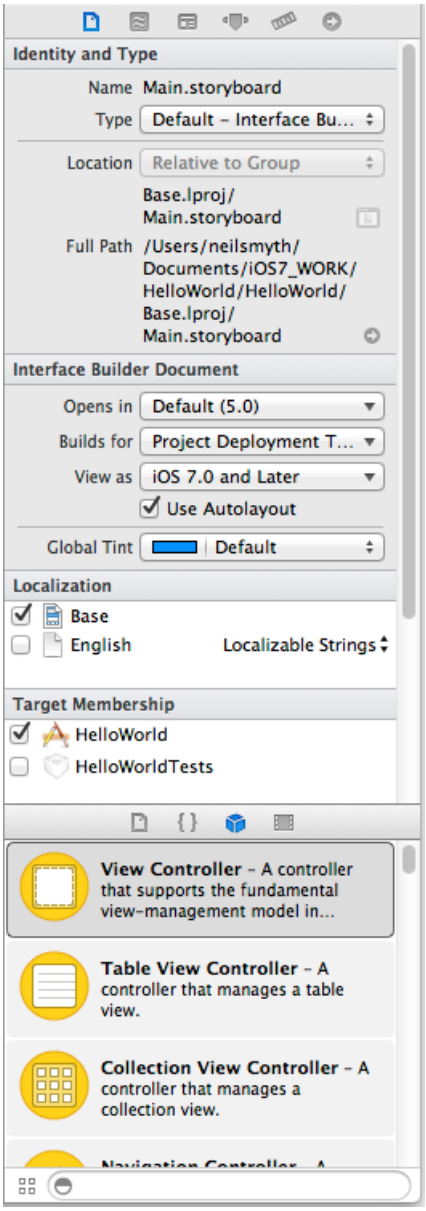

Figure 4-7

Along the top edge of the panel is a row of buttons which change the settings displayed in the upper half of the panel. By default the *File Inspector* is displayed. Options are also provided to display quick help, the *Identity Inspector*, *Attributes Inspector*, *Size Inspector* and *Connections Inspector*. Before proceeding, take some time to review each of these selections to gain some familiarity with the configuration options each provides. Throughout the remainder of this book extensive use of these inspectors will be made.

The lower section of the panel may default to displaying the file template library. Above this panel is another toolbar containing buttons to display other categories. Options include frequently used code snippets to save on typing when writing code, the Object Library and the Media Library. For

the purposes of this tutorial we need to display the Object Library so click on the appropriate toolbar button (represented by the three dimensional cube). This will display the UI components that can be used to construct our user interface. Move the cursor to the line above the lower toolbar and click and drag to increase the amount of space available for the library if required. The layout of the items in the library may also be switched from a single column of objects with descriptions to multiple columns without descriptions by clicking on the option located in the bottom left hand corner of the panel and to the left of the search box.

4.3 Changing Component Properties

With the property panel for the View selected in the main panel, we will begin our design work by changing the background color of this view. Start by making sure the View is selected and that the Attributes Inspector (*View -> Utilities -> Show Attributes Inspector*) is displayed in the right hand panel. Click on the white rectangle next to the *Background* label to invoke the *Colors* dialog. Using the color selection tool, choose a visually pleasing color and close the dialog. You will now notice that the view window has changed from white to the new color selection.

4.4 Adding Objects to the User Interface

The next step is to add a Label object to our view. To achieve this, either scroll down the list of objects in the library panel to locate the Label object or, as illustrated in Figure 4-8, enter *Label* into the search box beneath the panel:

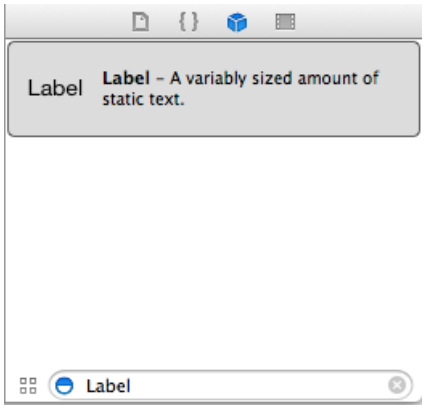

Figure 4-8

Having located the Label object, click on it and drag it to the center of the view so that the vertical and horizontal center guidelines appear. Once it is in position release the mouse button to drop it at that location:

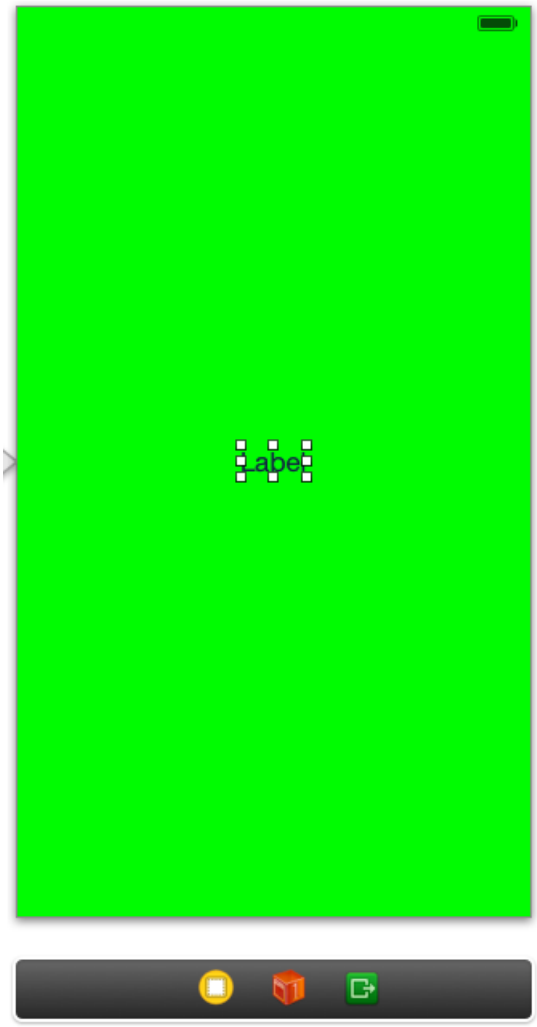

Figure 4-9

Using the resize markers surrounding the label border, stretch first the left and then right side of the label out to the edge of the view until the vertical blue dotted lines marking the recommended border of the view appear. With the Label still selected, click on the centered alignment button in the Attributes Inspector (*View -> Utilities -> Show Attributes Inspector*) to center the text in the middle of the screen. Click on the current font attribute setting to choose a larger *Custom* font setting, for example a Georgia bold typeface with a size of 24.

Finally, double click on the text in the label that currently reads "Label" and type in "Hello World". At this point, your View window will hopefully appear as outlined in Figure 4-10 (allowing, of course, for differences in your color and font choices).

Having created our simple user interface design we now need to save it. To achieve this, select *File -> Save* or use the Command+S keyboard shortcut.

Figure 4-10

4.5 Building and Running an iOS 7 App in Xcode 5

Before an app can be run it must first be compiled. Once successfully compiled it may be run either within a simulator or on a physical iPhone, iPad or iPod Touch device. The process for testing an app on a physical device requires some additional steps to be performed involving developer certificates and provisioning profiles and will be covered in detail in *Testing Apps on iOS 7 Devices with Xcode 5*. For the purposes of this chapter, however, it is sufficient to run the app in the simulator.

Within the main Xcode 5 project window, make sure that the menu located in the top left hand corner of the window (to the right of the square "stop" button) has the *iPhone Retina (4-inch)* simulator option selected and then click on the *Run* toolbar button (the triangular button located to the left of the stop button resembling a "play" button) to compile the code and run the app in the simulator. The small iTunes style window in the center of the Xcode toolbar will report the progress of the

build process together with any problems or errors that cause the build process to fail. Once the app is built, the simulator will start and the HelloWorld app will run:

Figure 4-11

4.6 Dealing with Build Errors

As we have not actually written or modified any code in this chapter it is unlikely that any errors will be detected during the build and run process. In the unlikely event that something did get inadvertently changed thereby causing the build to fail it is worth taking a few minutes to talk about build errors within the context of the Xcode environment.

If for any reason a build fails, the status window in the Xcode toolbar will report that an error has been detected by displaying "Build" together with the number of errors detected and any warnings. In addition, the left hand panel of the Xcode window will update with a list of the errors. Selecting an error from this list will take you to the location in the code where corrective action needs to be taken.

4.7 Testing Different Screen Sizes

With the introduction of the retina display, iPhone 5 series and iPad Mini, applications now potentially need to work on a variety of different screens sizes and resolutions.

In order to test the appearance of an application on these different displays, simply launch the application in the iOS Simulator and switch between the different displays using the *Hardware -> Device* menu options.

4.8 Testing User Interface Appearance in Different iOS Versions

In addition to testing the application on different devices, there is also the need to validate the appearance of an application on different versions of iOS. Whilst this was not an issue on versions of iOS prior to version 6, iOS 7 introduces a new look and feel for applications. Many user interface elements now have a significantly different appearance from those in earlier versions of iOS and a number also differ in size. In recognition of this fact, Xcode now includes a *Preview* assistant that enables the user interface of an application to be previewed as it will appear either on iOS 7 or earlier versions of iOS.

With the *Main.storyboard* file selected and the storyboard canvas displayed, select the Xcode *View -> Assistant Editor -> Show Assistant Editor*. A new panel will appear to the right of the storyboard canvas as shown in Figure 4-12:

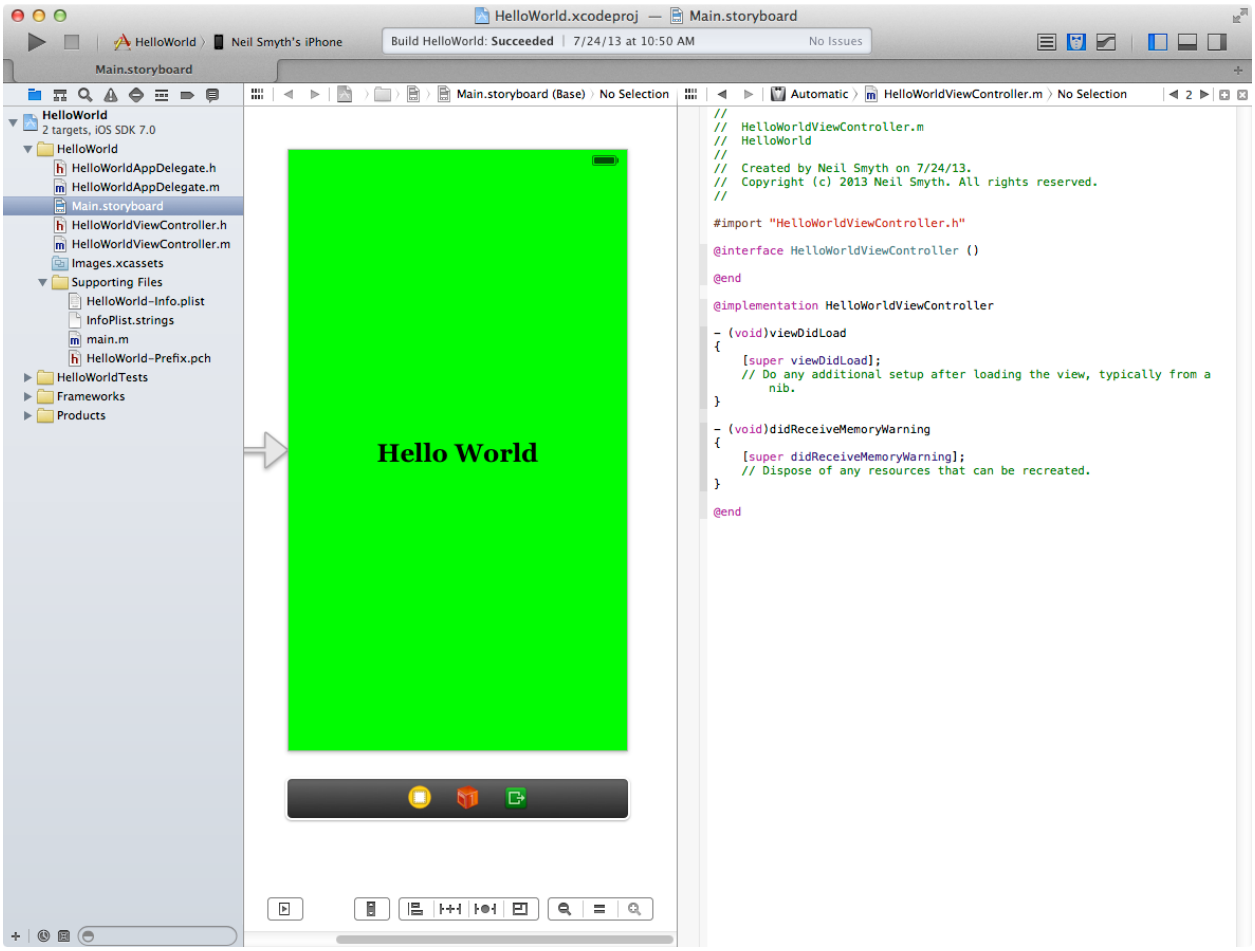

Figure 4-12

Within the assistant editor panel, the toolbar across the top of the panel will display an entry that reads either *Manual* or *Automatic* (or may be represented by a tuxedo icon). Click on this item to display a drop down menu and, from that menu, select the *Preview -> Main.storyboard (Preview)* menu option. The panel will change to show a representation of the user interface for the selected storyboard scene. Using the menu in the lower right hand corner, change the setting from *iOS 7.0 and Later* to *iOS 6.1 and Earlier*. Note that the status bar across the top of the user interface changes to reflect the non-transparent form of the bar present in pre-iOS 7 releases. In addition, the rotation button can be used to switch between landscape and portrait orientation and the size button used to switch between iOS device display sizes. Clearly, this example application does not handle rotation well, an issue that will be explored when the topic of Auto Layout is covered later in the book.

When working with applications that will be required to run on earlier versions of iOS, the preview tool provides a useful mechanism for testing the appearance of the application without the need to compile and run the code using different SDK configurations.

4.9 Monitoring Application Performance

Another useful feature of Xcode is the ability to monitor the performance of an application while it is running. This information is accessed by displaying the *Debug Navigator*.

When Xcode is launched, the project navigator is displayed in the left hand panel by default. Along the top of this panel is a bar with a range of other options. The sixth option from the left displays the debug navigator when selected as illustrated in Figure 4-13. When displayed, this panel shows a number of real-time statistics relating to the performance of the currently running application such as memory, CPU usage and iCloud storage access.

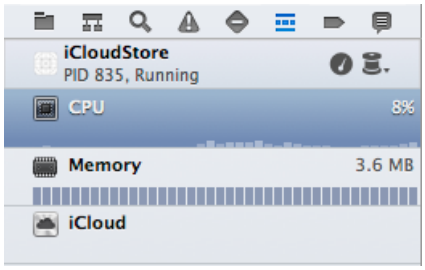

Figure 4-13

When one of these categories is selected, the main panel (Figure 4-14) updates to provide additional information about that particular aspect of the application's performance:

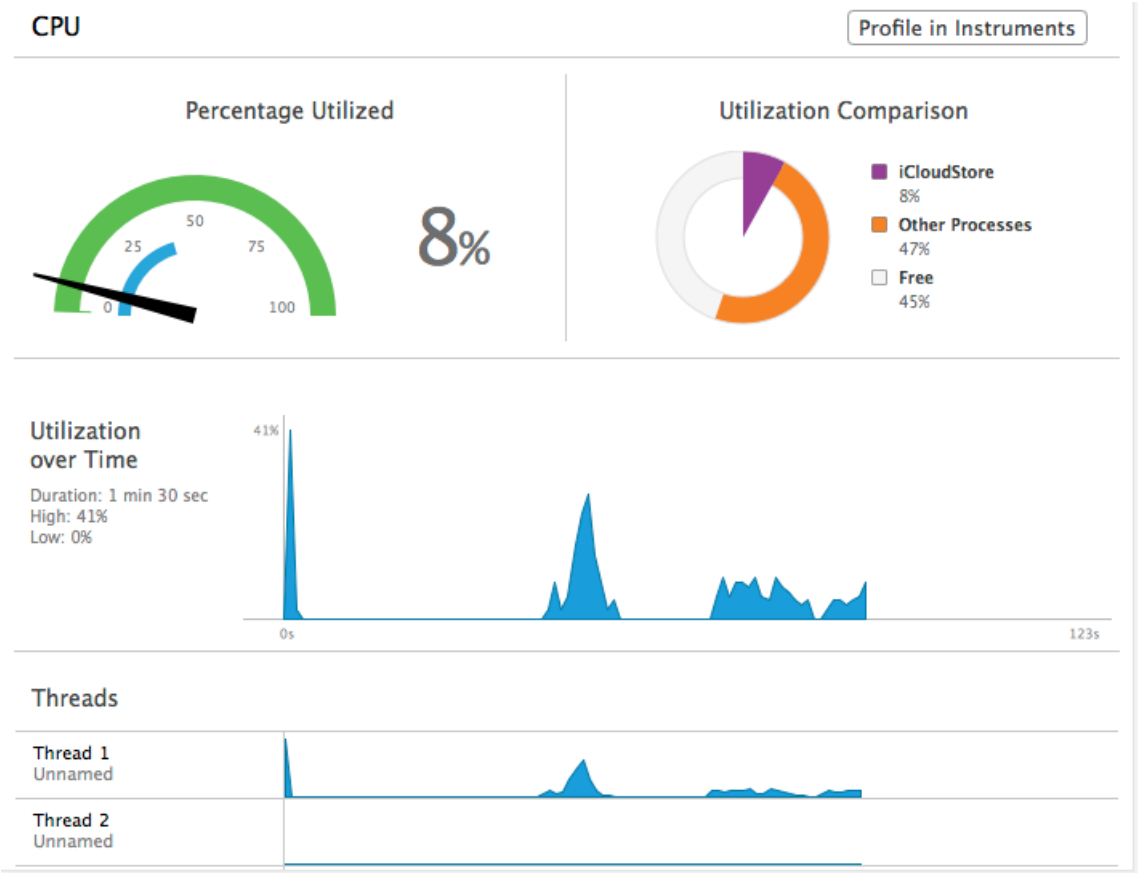

Figure 4-14

Yet more information can be obtained by clicking on the *Profile in Instruments* button in the top right hand corner of the panel.

4.10 Summary

Applications are primarily created within the Xcode development environment. This chapter has served to provide a basic overview of the Xcode environment and to work through the creation of a very simple example application. The chapter also explored the topic of testing the appearance of the user interface of an application on older iOS versions using the preview assistant. This is of particular importance given the dramatic changes in the appearance of application user interfaces between iOS 6 and iOS 7. Finally, a brief overview was provided of some of the performance monitoring features in Xcode 5.

5. iOS 7 Architecture and SDK Frameworks

By just about any measure, the iPhone and iPad represent an impressive achievement in the fields of industrial design and hardware engineering. When we develop apps for the iPhone and iPad, however, Apple does not allow us direct access to any of this hardware. In fact, all hardware interaction takes place exclusively through a number of different layers of software which act as intermediaries between the application code and device hardware. These layers make up what is known as an *operating system*. In the case of the iPhone and iPad, this operating system is known as iOS.

In order to gain a better understanding of the iOS 7 development environment, this chapter will look in detail at the different layers that comprise the iOS operating system and the frameworks that allow us, as developers, to write iPhone and iPad applications.

5.1 iPhone OS becomes iOS

Prior to the release of the iPad in 2010, the operating system running on the iPhone was generally referred to as *iPhone OS*. Given that the operating system used for the iPad is essentially the same as that on the iPhone it didn't make much sense to name it *iPad OS*. Instead, Apple decided to adopt a more generic and non-device specific name for the operating system. Given Apple's predilection for names prefixed with the letter 'i' (iTunes, iBookstore, iMac etc) the logical choice was, of course, *iOS*. Unfortunately, iOS is also the name used by Cisco for the operating system on its routers (Apple, it seems, also has a predilection for ignoring trademarks). When performing an internet search for iOS, therefore, be prepared to see large numbers of results for Cisco's iOS which have absolutely nothing to do with Apple's iOS.

5.2 An Overview of the iOS 7 Architecture

As previously mentioned, iOS consists of a number of different software layers, each of which provides programming frameworks for the development of applications that run on top of the underlying hardware.

These operating system layers can be presented diagrammatically as illustrated in Figure 5-1:

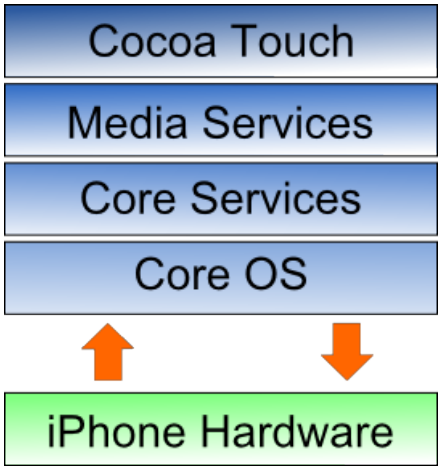

Figure 5-1

Some diagrams designed to graphically depict the iOS software stack show an additional box positioned above the Cocoa Touch layer to indicate the applications running on the device. In the above diagram we have not done so since this would suggest that the only interface available to the app is Cocoa Touch. In practice, an app can directly call down to any of the layers of the stack to perform tasks on the physical device.

That said, however, each operating system layer provides an increasing level of abstraction away from the complexity of working with the hardware. As an iOS developer you should, therefore, always look for solutions to your programming goals in the frameworks located in the higher level iOS layers before resorting to writing code that reaches down to the lower level layers. In general, the higher level of layer you program to, the less effort and fewer lines of code you will have to write to achieve your objective. And as any veteran programmer will tell you, the less code you have to write the less opportunity you have to introduce bugs.

Now that we have identified the various layers that comprise iOS 7 we can now look in more detail at the services provided by each layer and the corresponding frameworks that make those services available to us as application developers.

5.3 The Cocoa Touch Layer

The Cocoa Touch layer sits at the top of the iOS stack and contains the frameworks that are most commonly used by iOS application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API (as found on Apple desktop and laptop computers) and has been extended and modified to meet the needs of the iPhone and iPad hardware.

The Cocoa Touch layer provides the following frameworks for iOS app development:

5.3.1 UIKit Framework (UIKit.framework)

The UIKit framework is a vast and feature rich Objective-C based programming interface. It is, without question, the framework with which you will spend most of your time working. Entire books could, and probably will, be written about the UIKit framework alone. Some of the key features of UIKit are as follows:

· User interface creation and management (text fields, buttons, labels, colors, fonts etc)

· Application lifecycle management

· Application event handling (e.g. touch screen user interaction)

· Multitasking

· Wireless Printing

· Data protection via encryption

· Cut, copy, and paste functionality

· Web and text content presentation and management

· Data handling

· Inter-application integration

· Push notification in conjunction with Push Notification Service

· Local notifications (a mechanism whereby an application running in the background can gain the user's attention)

· Accessibility

· Accelerometer, battery, proximity sensor, camera and photo library interaction

· Touch screen gesture recognition

· File sharing (the ability to make application files stored on the device available via iTunes)

· Blue tooth based peer to peer connectivity between devices

· Connection to external displays

To get a feel for the richness of this framework it is worth spending some time browsing Apple's UIKit reference material which is available online at:

http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIKit_Framework/index.html

5.3.2 Map Kit Framework (MapKit.framework)

If you have spent any appreciable time with an iPhone or iPad then the chances are you have needed to use the Maps application more than once, either to get a map of a specific area or to generate driving directions to get you to your intended destination. The Map Kit framework

provides a programming interface which enables you to build map based capabilities into your own applications. This allows you to, amongst other things, display scrollable maps for any location, display the map corresponding to the current geographical location of the device and annotate the map in a variety of ways.

5.3.3 Push Notification Service

The Push Notification Service allows applications to notify users of an event even when the application is not currently running on the device. Since the introduction of this service it has most commonly been used by news based applications. Typically when there is breaking news the service will generate a message on the device with the news headline and provide the user the option to load the corresponding news app to read more details. This alert is typically accompanied by an audio alert and vibration of the device. This feature should be used sparingly to avoid annoying the user with frequent interruptions.

5.3.4 Message UI Framework (MessageUI.framework)

The Message UI framework provides everything you need to allow users to compose and send email messages from within your application. In fact, the framework even provides the user interface elements through which the user enters the email addressing information and message content. Alternatively, this information may be pre-defined within your application and then displayed for the user to edit and approve prior to sending.

5.3.5 Address Book UI Framework (AddressUI.framework)

Given that a key function of the iPhone and iPad is as communication devices and digital assistants, it should not come as too much of a surprise that an entire framework is dedicated to the integration of the address book data into your own applications. The primary purpose of the framework is to enable you to access, display, edit and enter contact information from the iOS address book from within your own application.

5.3.6 Game Kit Framework (GameKit.framework)

The Game Kit framework provides peer-to-peer connectivity and voice communication between multiple devices and users allowing those running the same app to interact. When this feature was first introduced it was anticipated by Apple that it would primarily be used in multi-player games (hence the choice of name) but the possible applications for this feature clearly extend far beyond games development.

5.3.7 iAd Framework (iAd.framework)

The purpose of the iAd Framework is to allow developers to include banner advertising within their applications. All advertisements are served by Apple's own ad service.

5.3.8 Event Kit UI Framework (EventKit.framework)

The Event Kit UI framework was introduced in iOS 4 and is provided to allow the calendar and reminder events to be accessed and edited from within an application.

5.3.9 Accounts Framework (Accounts.framework)

iOS 5 introduced the concept of system accounts. These essentially allow the account information for other services to be stored on the iOS device and accessed from within application code. Currently system accounts are limited to Twitter and Facebook accounts, though other services will likely appear in future iOS releases. The purpose of the Accounts Framework is to provide an API allowing applications to access and manage these system accounts.

5.3.10 Social Framework (Social.framework)

The Social Framework allows Twitter, Facebook and Sina Weibo integration to be added to applications. The framework operates in conjunction the Accounts Framework to gain access to the user's social network account information.

5.3.11 SpriteKit Framework (SpriteKit.framework)

The SpriteKit framework provides an environment for the rapid development of 2D based games including features such as sprite animation, collision detection, physics simulation and particle based special effects.

5.4 The iOS Media Layer

The role of the Media layer is to provide iOS with audio, video, animation and graphics capabilities. As with the other layers comprising the iOS stack, the Media layer comprises a number of frameworks which may be utilized when developing iOS apps. In this section we will look at each one in turn.

5.4.1 Core Video Framework (CoreVideo.framework)

The Core Video Framework provides buffering support for the Core Media framework. Whilst this may be utilized by application developers it is typically not necessary to use this framework.

5.4.2 Core Text Framework (CoreText.framework)

The iOS Core Text framework is a C-based API designed to ease the handling of advanced text layout and font rendering requirements.

5.4.3 Image I/O Framework (ImageIO.framework)

The Image I/O framework, the purpose of which is to facilitate the importing and exporting of image data and image metadata, was introduced in iOS 4. The framework supports a wide range of image formats including PNG, JPEG, TIFF and GIF.

5.4.4 Assets Library Framework (AssetsLibrary.framework)

The Assets Library provides a mechanism for locating and retrieving video and photo files located on the iOS device. In addition to accessing existing images and videos, this framework also allows new photos and videos to be saved to the standard device photo album.

5.4.5 Core Graphics Framework (CoreGraphics.framework)**

The iOS Core Graphics Framework (otherwise known as the Quartz 2D API) provides a lightweight two dimensional rendering engine. Features of this framework include PDF document creation and presentation, vector based drawing, transparent layers, path based drawing, anti-aliased rendering, color manipulation and management, image rendering and gradients. Those familiar with the Quartz 2D API running on MacOS X will be pleased to learn that the implementation of this API is the same on iOS.

5.4.6 Core Image Framework (CoreImage.framework)

A framework introduced with iOS 5 providing a set of video and image filtering and manipulation capabilities for application developers.

5.4.7 Quartz Core Framework (QuartzCore.framework)

The purpose of the Quartz Core framework is to provide animation capabilities on the iPhone and iPad. It provides the foundation for the majority of the visual effects and animation used by the UIKit framework and provides an Objective-C based programming interface for creation of specialized animation within iOS apps.

5.4.8 OpenGL ES framework (OpenGLES.framework)

For many years the industry standard for high performance 2D and 3D graphics drawing has been OpenGL. Originally developed by the now defunct Silicon Graphics, Inc (SGI) during the 1990s in the form of GL, the open version of this technology (OpenGL) is now under the care of a non-profit consortium comprising a number of major companies including Apple, Inc., Intel, Motorola and ARM Holdings.

OpenGL for Embedded Systems (ES) is a lightweight version of the full OpenGL specification designed specifically for smaller devices such as the iPhone and iPad. iOS 7 introduces support for OpenGL ES 3.0.

5.4.9 GLKit Framework (GLKit.framework)

The GLKit framework is an Objective-C based API designed to ease the task of creating OpenGL ES based applications.

5.4.10 NewsstandKit Framework (NewsstandKit.framework)

The Newsstand application is intended as a central location for users to gain access to digital editions of newspapers and magazines. The NewsstandKit framework allows for the development of applications that utilize this new service.

5.4.11 iOS Audio Support

iOS is capable of supporting audio in AAC, Apple Lossless (ALAC), A-law, IMA/ADPCM, Linear PCM, μ-law, DVI/Intel IMA ADPCM, Microsoft GSM 6.10 and AES3-2003 formats through the support provided by the following frameworks.

5.4.12 AV Foundation framework (AVFoundation.framework)

An Objective-C based framework designed to allow the playback, recording and management of audio content.

5.4.13 Core Audio Frameworks (CoreAudio.framework, AudioToolbox.framework and AudioUnit.framework)

The frameworks that comprise Core Audio for iOS define supported audio types, playback and recording of audio files and streams and also provide access to the device's built-in audio processing units.

5.4.14 Open Audio Library (OpenAL)

OpenAL is a cross platform technology used to provide high-quality, 3D audio effects (also referred to as positional audio). Positional audio may be used in a variety of applications though is typically used to provide sound effects in games.

5.4.15 Media Player Framework (MediaPlayer.framework)

The iOS Media Player framework is able to play video in .mov, .mp4, .m4v, and .3gp formats at a variety of compression standards, resolutions and frame rates.

5.4.16 Core Midi Framework (CoreMIDI.framework)

Introduced in iOS 4, the Core MIDI framework provides an API for applications to interact with MIDI compliant devices such as synthesizers and keyboards via the iPhone or iPad dock connector.

5.5 The iOS Core Services Layer

The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built and consists of the following frameworks.

5.5.1 Address Book Framework (AddressBook.framework)

The Address Book framework provides programmatic access to the iOS Address Book contact database allowing applications to retrieve and modify contact entries.

5.5.2 CFNetwork Framework (CFNetwork.framework)

The CFNetwork framework provides a C-based interface to the TCP/IP networking protocol stack and low level access to BSD sockets. This enables application code to be written that works with HTTP, FTP and Domain Name servers and to establish secure and encrypted connections using Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

5.5.3 Core Data Framework (CoreData.framework)

This framework is provided to ease the creation of data modeling and storage in Model-View-Controller (MVC) based applications. Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data within an application.

5.5.4 Core Foundation Framework (CoreFoundation.framework)

The Core Foundation framework is a C-based Framework which provides basic functionality such as data types, string manipulation, raw block data management, URL manipulation, threads and run loops, date and times, basic XML manipulation and port and socket communication. Additional XML capabilities beyond those included with this framework are provided via the libXML2 library. Though this is a C-based interface, most of the capabilities of the Core Foundation framework are also available with Objective-C wrappers via the Foundation Framework.

5.5.5 Core Media Framework (CoreMedia.framework)

The Core Media framework is the lower level foundation upon which the AV Foundation layer is built. Whilst most audio and video tasks can, and indeed should, be performed using the higher level AV Foundation framework, access is also provided for situations where lower level control is required by the iOS application developer.

5.5.6 Core Telephony Framework (CoreTelephony.framework)

The iOS Core Telephony framework is provided to allow applications to interrogate the device for information about the current cell phone service provider and to receive notification of telephony related events.

5.5.7 EventKit Framework (EventKit.framework)

An API designed to provide applications with access to the calendar, reminders and alarms on the device.

5.6 Foundation Framework (Foundation.framework)

The Foundation framework is the standard Objective-C framework that will be familiar to those who have programmed in Objective-C on other platforms (most likely Mac OS X). Essentially, this consists of Objective-C wrappers around much of the C-based Core Foundation Framework.

5.6.1 Core Location Framework (CoreLocation.framework)

The Core Location framework allows you to obtain the current geographical location of the device (latitude, longitude and altitude) and compass readings from within your own applications. The method used by the device to provide coordinates will depend on the data available at the time the information is requested and the hardware support provided by the particular iOS device model on which the app is running (GPS and compass were not featured on earlier models). This will either be based on GPS readings, Wi-Fi network data or cell tower triangulation (or some combination of the three).

5.6.2 Mobile Core Services Framework (MobileCoreServices.framework)

The iOS Mobile Core Services framework provides the foundation for Apple's Uniform Type Identifiers (UTI) mechanism, a system for specifying and identifying data types. A vast range of predefined identifiers have been defined by Apple including such diverse data types as text, RTF, HTML, JavaScript, PowerPoint .ppt files, PhotoShop images and MP3 files.

5.6.3 Store Kit Framework (StoreKit.framework)

The purpose of the Store Kit framework is to facilitate commerce transactions between your application and the Apple App Store. Prior to version 3.0 of iOS, it was only possible to charge a customer for an app at the point that they purchased it from the App Store. iOS 3.0 introduced the concept of the "in app purchase" whereby the user can be given the option to make additional payments from within the application. This might, for example, involve implementing a subscription model for an application, purchasing additional functionality or even buying a faster car for you to drive in a racing game. Since the introduction of iOS 6, content associated with an in-app purchase can now be hosted on, and downloaded from, Apple's servers.

5.6.4 SQLite library

Allows for a lightweight, SQL based database to be created and manipulated from within your iOS application.

5.6.5 System Configuration Framework (SystemConfiguration.framework)

The System Configuration framework allows applications to access the network configuration settings of the device to establish information about the "reachability" of the device (for example whether Wi-Fi or cell connectivity is active and whether and how traffic can be routed to a server).

5.6.6 Quick Look Framework (QuickLook.framework)

The Quick Look framework provides a useful mechanism for displaying previews of the contents of file types loaded onto the device (typically via an internet or network connection) for which the application does not already provide support. File format types supported by this framework include iWork, Microsoft Office document, Rich Text Format, Adobe PDF, Image files, public.text files and comma separated (CSV).

5.7 The iOS Core OS Layer

The Core OS Layer occupies the bottom position of the iOS stack and, as such, sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.

5.7.1 Accelerate Framework (Accelerate.framework)

The Accelerate Framework provides a hardware optimized C-based API for performing complex and large number math, vector, digital signal processing (DSP) and image processing tasks and calculations.

5.7.2 External Accessory Framework (ExternalAccessory.framework)

Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone or iPad via the dock connector or wirelessly via Bluetooth.

5.7.3 Security Framework (Security.framework)

The iOS Security framework provides all the security interfaces you would expect to find on a device that can connect to external networks including certificates, public and private keys, trust policies, keychains, encryption, digests and Hash-based Message Authentication Code (HMAC).

5.7.4 System (LibSystem)

As we have previously mentioned, iOS is built upon a UNIX-like foundation. The System component of the Core OS Layer provides much the same functionality as any other UNIX like operating system. This layer includes the operating system kernel (based on the Mach kernel developed by Carnegie Mellon University) and device drivers. The kernel is the foundation on which the entire iOS platform is built and provides the low level interface to the underlying hardware. Amongst other things, the kernel is responsible for memory allocation, process lifecycle management, input/output, inter-process communication, thread management, low level networking, file system access and thread management.

As an app developer your access to the System interfaces is restricted for security and stability reasons. Those interfaces that are available to you are contained in a C-based library called LibSystem. As with all other layers of the iOS stack, these interfaces should be used only when you are absolutely certain there is no way to achieve the same objective using a framework located in a higher iOS layer.

6. Testing Apps on iOS 7 Devices with Xcode 5

In the chapter entitled *Creating a Simple iOS 7 App* we were able to run an application in the iOS Simulator environment bundled with the iOS 7 SDK. Whilst this is fine for most cases, in practice there are a number of areas that cannot be comprehensively tested in the simulator. For example, no matter how hard you shake your computer (not something we actually recommend) or where in the world you move it to, neither the accelerometer nor GPS features will provide real world results within the simulator (though the simulator does have the option to perform a basic virtual shake gesture and to simulate location data). If we really want to test an iOS application thoroughly in the real world, therefore, we need to install the app onto a physical iOS device.

Many new features have been added Xcode 5 to make the task of the developer easier. One of these features makes it considerably easier to obtain the signing certificates and provisioning profiles that are necessary to perform testing of applications on physical iOS devices.

The previous edition of this book, which was based on Xcode 4, dedicated no less than 11 pages to the process of obtaining a developer certificate, App ID and provisioning profile to test an application on a physical iOS device. Much of this work is now performed automatically by Xcode resulting in a much simpler path to testing applications on iOS devices.

6.1 Configuring Xcode with Apple IDs

The first step in setting up a fully configured development environment involves entering the Apple ID associated with your Apple Developer Program membership.

To enter this information, start Xcode and select the *Xcode -> Preferences...* menu option. From within the preferences window, select the *Accounts* tab as illustrated in Figure 6-1:

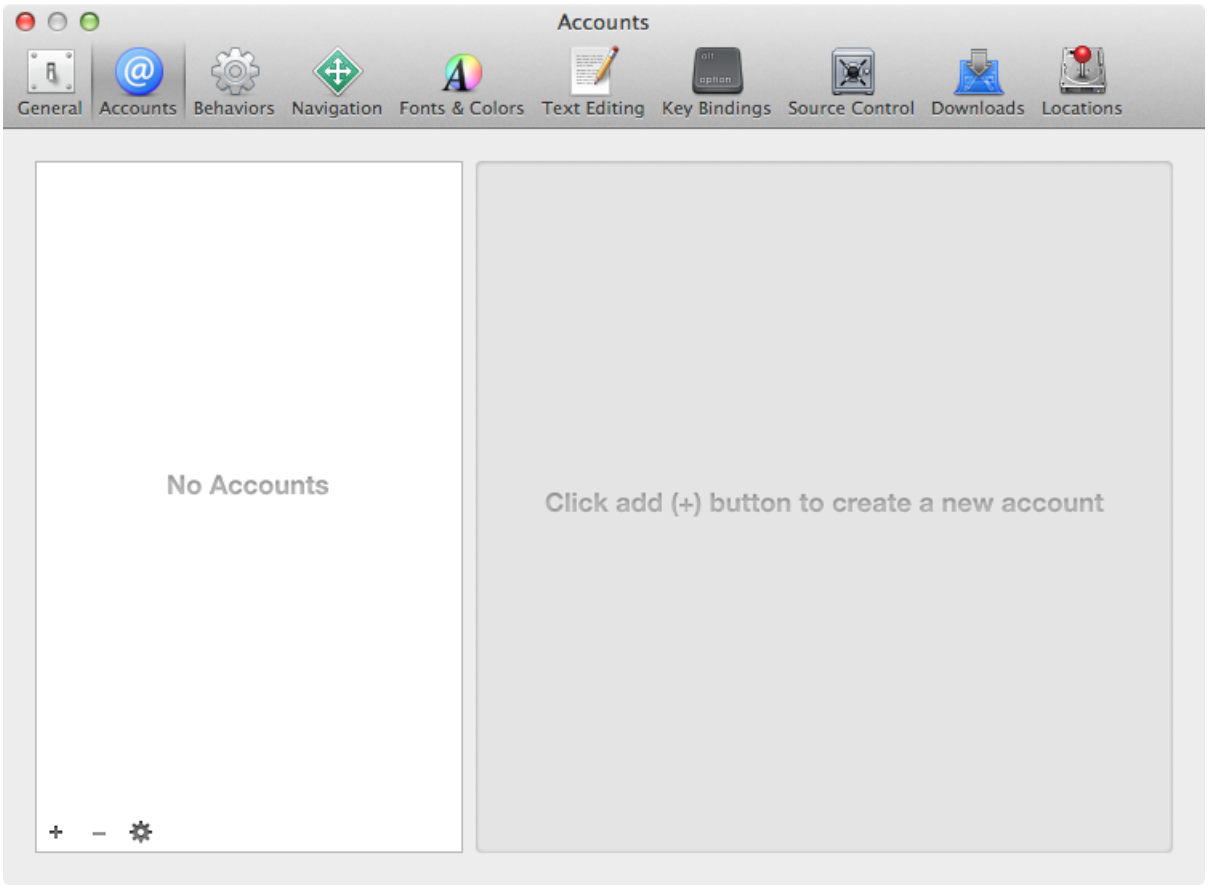

Figure 6-1

To add an Apple ID, click on the + button in the lower left hand corner and select *Add Apple ID...* from the drop down menu. When prompted to do so (Figure 6-2), either enter the Apple ID and password associated with your Apple Developer Program membership, or click on the *Join a Program...* button if you are not yet a member.

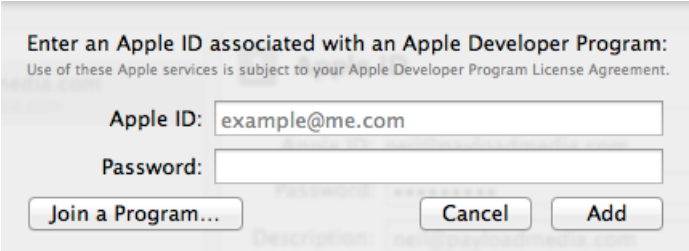

Figure 6-2

Repeat these steps to add additional Apple IDs if you are associated with more than one development team. Once the information has been entered, the accounts will be listed in the preferences window.

6.2 Generating Signing Identities

Before an application can be run on a physical iOS device for testing purposes it must first be signed with a *developer signing identity*. When the application is finished and ready to be placed on sale in the App Store it must first be signed with a *distribution signing identity*. Signing identities are comprised of a certificate and a private key.

Signing identities can be generated from within the Xcode account preferences panel. Begin by selecting the Apple ID for which the identities are to be generated before clicking on the *View Details...* button located in the lower right hand corner of the window. This will display a list of signing identities and any provisioning profiles associated with those identities. If no valid signing identities are listed (as is the cased in Figure 6-3), the next step is to generate them.

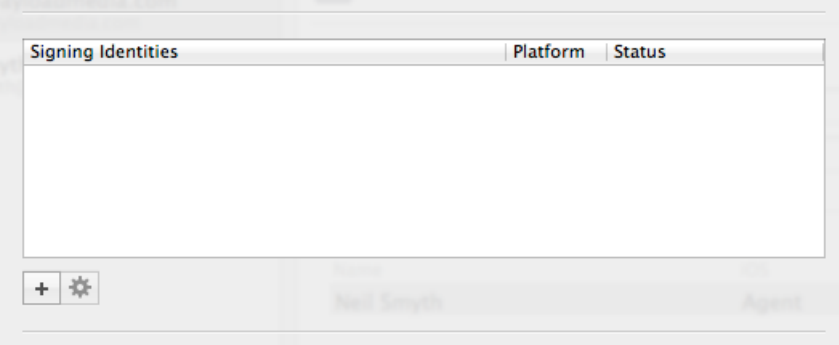

Figure 6-3

Begin by clicking on the + button and selecting the *iOS Development* option from the resulting menu. Xcode will then contact the Apple Developer member Center portal and request and download a developer signing identity. Repeat these steps, this time selecting *iOS Distribution* from the menu to create and download a distribution signing identity. Once completed, the two identities should now be listed as shown in Figure 6-4:

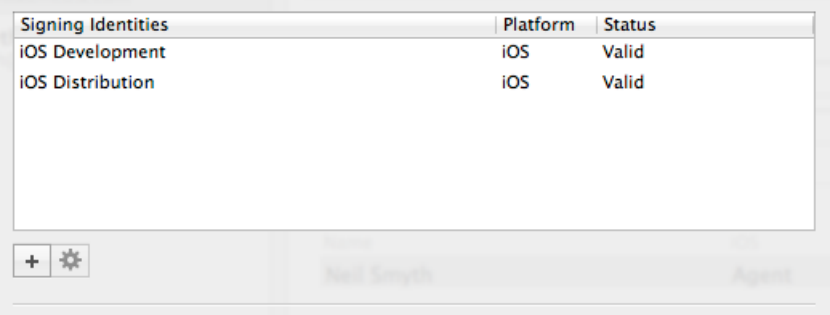

Figure 6-4

Once created, signing identities and account information can be migrated to other development computer systems by clicking on the button displaying a gear cog on the first account settings page and selecting the *Export Accounts...* menu option. On the destination system repeat these steps, this

time selecting the *Import Accounts...* option.__

It is worth noting that the certificates associated with the signing identities can also be viewed and created within the Apple Developer Member Center portal. Within a browser, navigate to the following URL and log in using your Apple ID credentials:

https://developer.apple.com/membercenter

Within the member center, click on the Certificates, Identifiers and Profiles option and choose *Certificates* from the list of options under the *iOS Apps* category. On the resulting page, the certificates for both signing identifies should be listed. Clicking on certificate will display details such as the expiration date as outlined in Figure 6-5:

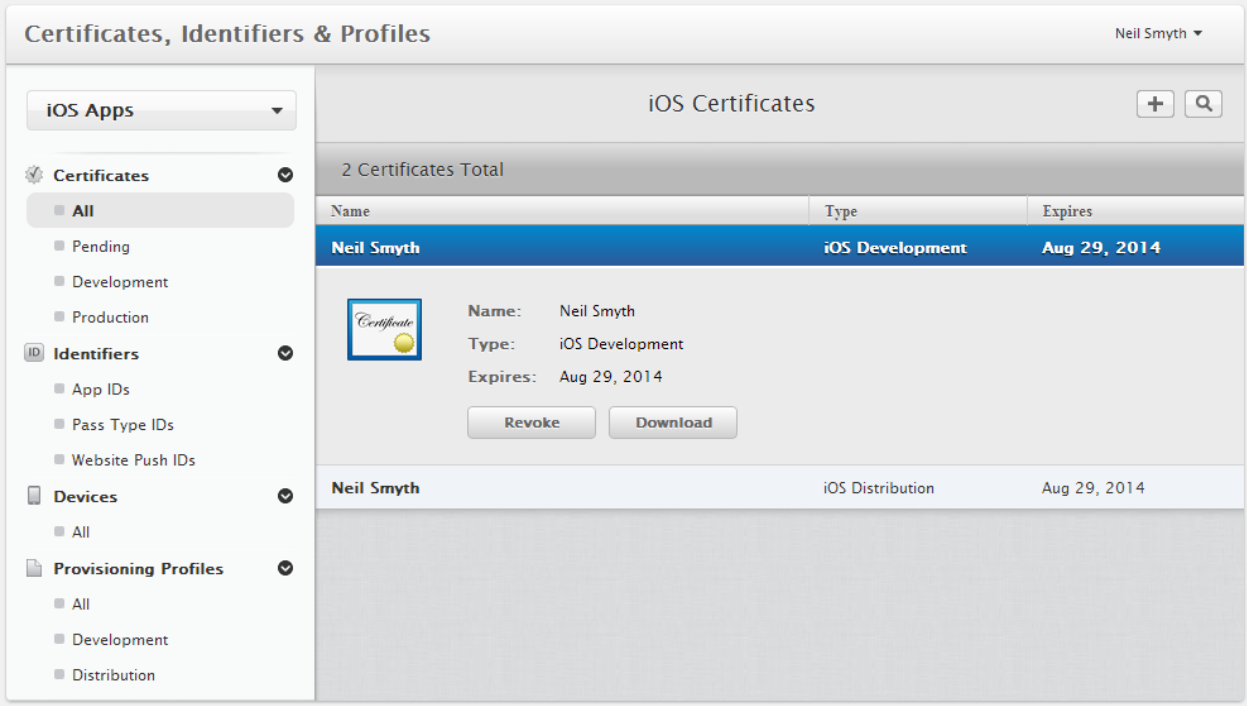

Figure 6-5

As can be seen in the left hand panel of Figure 6-5, the member center also provides options to manually create App IDs and Provisioning Profiles. With Xcode 5, however, these are typically created automatically.

6.3 Adding a Device to the Developer Portal

Having generated signing identities the next step is to register the iOS devices to be used for testing with the iOS member center. Note that Apple restricts developers to 100 provisioned devices within the membership year.

A new device may be added to the list of supported test devices from within the Xcode Organizer window. To add a device to the portal from within the Organizer, simply connect the device to the

development system, open the Organizer window in Xcode using the *Window -> Organizer* menu option, select the attached device from the left hand panel and click on either the *Add to Portal* or *Use for Development* button. The Organizer will connect to the developer portal and register the device for testing purposes.

6.4 Running an Application on a Registered Device

With a registered device connected to the development system, and an application ready for testing, refer to the device menu located in the Xcode toolbar. There is a reasonable chance that this will have defaulted to one of the iOS Simulator configurations (in the case of Figure 6-6, this is the iPhone Retina 4-inch simulator configuration).

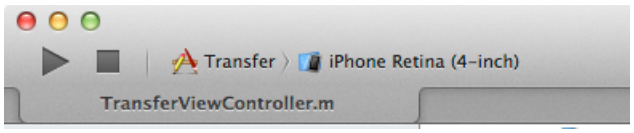

Figure 6-6

Switch to the physical device by selecting this menu and changing it to the device name as shown in Figure 6-7:

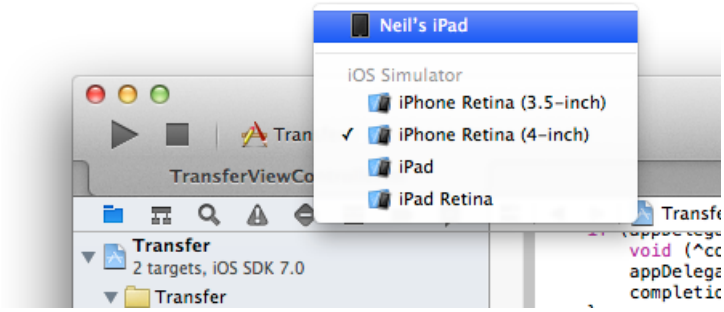

Figure 6-7

Xcode will request a provisioning profile that matches the App ID of the application and includes permission to run on the specified device, build the application using the developer signing identity before installing the application and provisioning profile on the device. Finally the application will be launched on the device.

6.5 Summary

Without question, the iOS Simulator included with the iOS 7 SDK is an invaluable tool for testing applications during the development process. There are, however, a number of situations where it is necessary to test an application on a physical iOS device. In this chapter we have covered the steps involved in provisioning applications for installation and testing on iPhone and iPad devices.

7. The Basics of Objective-C Programming

In order to develop apps for iOS it is necessary to use a programming language called Objective-C. A comprehensive guide to programming in Objective-C is beyond the scope of this book. In fact, if you are unfamiliar with Objective-C programming we strongly recommend that you read a copy of a book called *Objective-C 2.0 Essentials*. This is the companion book to *iOS 7 App Development Essentials* and will teach you everything you need to know about programming in Objective-C.

In the next two chapters we will take some time to go over the fundamentals of Objective-C programming with the goal of providing enough information to get you started.

7.1 Objective-C Data Types and Variables

One of the fundamentals of any program involves data, and programming languages such as Objective-C define a set of *data types* that allow us to work with data in a format we understand when writing a computer program. For example, if we want to store a number in an Objective-C program we could do so with syntax similar to the following:

```
int mynumber = 10;
```

In the above example, we have created a variable named *mynumber* of data type *integer* by using the keyword *int*. We then assigned the value of 10 to this variable.

Objective-C supports a variety of data types including int, char, float, double, boolean (BOOL) and a special general purpose data type named *id*.

Data type qualifiers are also supported in the form of long, long long, short, unsigned and signed. For example if we want to be able to store an extremely large number in our *mynumber* declaration we can qualify it as follows:

```
long long int mynumber =  345730489;
```

A variable may be declared as constant (i.e. the value assigned to the variable cannot be changed subsequent to the initial assignment) through the use of the *const* qualifier:

```
const char myconst = 'c';
```

7.2 Objective-C Expressions

Now that we have looked at variables and data types we need to look at how we work with this data in an application. The primary method for working with data is in the form of *expressions*.

The most basic expression consists of an *operator*, two *operands* and an *assignment*. The following is an example of an expression:

```
int myresult = 1 + 2;
```

In the above example the (+) operator is used to add two operands (1 and 2) together. The *assignment operator* (=) subsequently assigns the result of the addition to an integer variable named *myresult*. The operands could just have easily been variables (or a mixture of constants and variables) instead of the actual numerical values used in the example.

In the above example we looked at the addition operator. Objective-C also supports the following arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulo

Another useful type of operator is the compound assignment operator. This allows an operation and assignment to be performed with a single operator. For example one might write an expression as follows:

```
x = x + y;
```

The above expression adds the value contained in variable x to the value contained in variable y and stores the result in variable x. This can be simplified using the addition compound assignment operator:

```
x += y
```

Objective-C supports the following compound assignment operators:

Operator	Description
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x
x &= y	Assign to x the result of logical AND operation on x and y
x |= y	Assign to x the result of logical OR operation on x and y
x ^= y	Assign to x the result of logical Exclusive OR on x and y

Another useful shortcut can be achieved using the Objective-C increment and decrement operators (also referred to as *unary operators* because they operate on a single operand). As with the compound assignment operators described in the previous section, consider the following Objective-C code fragment:

```
x = x + 1; // Increase value of variable x by 1
x = x - 1; // Decrease value of variable x by 1
```

These expressions increment and decrement the value of x by 1. Instead of using this approach it is quicker to use the ++ and – operators. The following examples perform exactly the same tasks as the examples above:

```
x++; Increment x by 1
x--; Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the variable name the increment or decrement is performed before any other operations are performed on the variable.

In addition to mathematical and assignment operators, Objective-C also includes a set of logical operators useful for performing comparisons. These operators all return a Boolean (*BOOL*) *true* (1) or *false* (0) result depending on the result of the comparison. These operators are *binary operators* in that they work with two operands.

Comparison operators are most frequently used in constructing program flow control logic. For example an *if* statement may be constructed based on whether one value matches another:

```
if (x == y)
      // Perform task
```

The result of a comparison may also be stored in a *BOOL* variable. For example, the following code will result in a *true* (1) value being stored in the variable result:

```
BOOL result;
int x = 10;
int y = 20;
result = x < y;
```

Clearly 10 is less than 20, resulting in a *true* evaluation of the *x < y* expression. The following table lists the full set of Objective-C comparison operators:

Operator	Description
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y

Objective-C also provides a set of so called logical operators designed to return boolean *true* and *false*. In practice *true* equates to 1 and *false* equates to 0. These operators both return boolean results and take boolean values as operands. The key operators are NOT (!), AND (&&), OR (||) and XOR (^).

The NOT (!) operator simply inverts the current value of a boolean variable, or the result of an expression. For example, if a variable named *flag* is currently 1 (true), prefixing the variable with a '!' character will invert the value to 0 (false):

```
bool flag = true; //variable is true
bool secondFlag;
secondFlag = !flag; // secondFlag set to false
```

The OR (||) operator returns 1 if one of its two operands evaluates to *true*, otherwise it returns 0. For example, the following example evaluates to true because at least one of the expressions either side of the OR operator is true:

```
if ((10 < 20) || (20 < 10))
        NSLog (@"Expression is true");
```

The AND (&&) operator returns 1 only if both operands evaluate to be true. The following example will return 0 because only one of the two operand expressions evaluates to_ _*true*:

```
if ((10 < 20) && (20 < 10))
      NSLog (@"Expression is true");
```

The XOR (^) operator returns 1 if one and only one of the two operands evaluates to true. For example, the following example will return 1 since only one operator evaluates to be true:

```
if ((10 < 20) ^ (20 < 10))
      NSLog("Expression is true");
```

If both operands evaluated to true or both were false the expression would return false.

Objective-C uses something called a *ternary operator* to provide a shortcut way of making decisions. The syntax of the ternary operator (also known as the conditional operator) is as follows:

```
[condition] ? [true expression] : [false expression]
```

The way this works is that *[condition]* is replaced with an expression that will return either *true* (1) or *false* (0). If the result is true then the expression that replaces the *[true expression]* is evaluated. Conversely, if the result was *false* then the *[false expression]* is evaluated. Let's see this in action:

```
int x = 10;
int y = 20;

NSLog(@"Largest number is %i", x > y ? x : y );
```

The above code example will evaluate whether x is greater than y. Clearly this will evaluate to false resulting in y being returned to the NSLog call for display to the user:

```
2009-10-07 11:14:06.756 t[5724] Largest number is 20
```

7.3 Objective-C Flow Control with *if* and *else*

Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *flow control* since it controls the *flow* of program execution.

The *if* statement is perhaps the most basic of flow control options available to the Objective-C programmer.

The basic syntax of the Objective-C *if* statement is as follows:

```
if (boolean expression) {
// Objective-C code to be performed when expression evaluates to true
}
```

Note that the braces ({}) are only required if more than one line of code is executed after the *if* expression. If only one line of code is listed under the *if* the braces are optional. For example, the following is valid code:

```
int x = 10;
if (x > 10)
       x = 10;
```

The next variation of the *if* statement allows us to also specify some code to perform if the expression in the *if* statement evaluates to *false*. The syntax for this construct is as follows:

```
if (boolean expression) {
// Code to be executed if expression is true
} else {
// Code to be executed if expression is false
}
```

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
int x = 10;

if ( x > 9 )
{
         NSLog (@"x is greater than 9!");
} else {
         NSLog (@"x is less than 9!");
}
```

In this case, the second NSLog statement would execute if the value of x was less than 9.

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose we can use the *if ... else if ...* construct, the syntax for which is as follows:

```
int x = 9;

if (x == 10)
{
       NSLog (@"x is 10");
}
else if (x == 9)
{
       NSLog (@"x is 9");
}
else if (x == 8)
{
         NSLog (@"x is 8");
}
```

7.4 Looping with the *for* Statement

The syntax of an Objective-C *for loop* is as follows:

```
for ( ''initializer''; ''conditional expression''; ''loop expression'' )
{
      // statements to be executed
}
```

The *initializer* typically initializes a counter variable. Traditionally the variable name *i* is used for this purpose, though any valid variable name will do. For example:

```
i = 0;
```

This sets the counter to be the variable *i* and sets it to zero. Note that the current widely used Objective-C standard (c89) requires that this variable be declared prior to its use in the *for* loop. For example:

```
int i=0;

for (i = 0; i < 100; i++)
{
     // Statements here
}
```

The next standard (c99) allows the variable to be declared and initialized in the *for* loop as follows:

```
for (int i=0; i<100; i++)
{
    //Statements here
}
```

It is possible to break out of a *for* loop before the designated number of iterations have been completed using the *break;* statement.

7.5 Objective-C Looping with *do* and *while*

The Objective-C *for* loop described previously works well when you know in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Objective-C provides the *while* loop.

The *while* loop syntax is defined as follows:

```
while (''condition'')
{
      // Objective-C statements go here
}
```

7.6 Objective-C *do* ... while loops

It is often helpful to think of the *do ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *do ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once.

The syntax of the *do ... while* loop is as follows:

```
do
{
       // Objective-C statements here
} while (''conditional expression'')
```

8. The Basics of Object Oriented Programming in Objective-C

Objective-C provides extensive support for developing object-oriented iOS applications. The subject area of object oriented programming is, however, large. It is not an exaggeration to state that entire books have been dedicated to the subject. As such, a detailed overview of object oriented software development is beyond the scope of this book. Instead, we will introduce the basic concepts involved in object oriented programming and then move on to explaining the concept as it relates to Objective-C application development. Once again, whilst we strive to provide the basic information you need in this chapter, we recommend reading a copy of *Objective-C 2.0 Essentials* if you are unfamiliar with Objective-C programming.

8.1 What is an Object?

Objects are self-contained modules of functionality that can be easily used, and re-used as the building blocks for a software application.

Objects consist of data variables and functions (called *methods*) that can be accessed and called on the object to perform tasks. These are collectively referred to as *members*.

8.2 What is a Class?

Much as a blueprint or architect's drawing defines what an item or a building will look like once it has been constructed, a class defines what an object will look like when it is created. It defines, for example, what the *methods* will do and what the *member* variables will be.

8.3 Declaring an Objective-C Class Interface

Before an object can be instantiated we first need to define the class 'blueprint' for the object. In this chapter we will create a Bank Account class to demonstrate the basic concepts of Objective-C object oriented programming.

An Objective-C class is defined in terms of an *interface* and an *implementation*. In the interface section of the definition we specify the base class from which the new class is derived and also define the members and methods that the class will contain. The syntax for the interface section of a class is as follows:

```
@interface NewClassName: ParentClass {
    ClassMembers;
}
ClassMethods;
@end
```

The *ClassMembers* section of the interface defines the variables that are to be contained within the class (also referred to as *instance variables*). These variables are declared in the same way that any other variable would be declared in Objective-C.

The *ClassMethods* section defines the methods that are available to be called on the class. These are essentially functions specific to the class that perform a particular operation when called upon.

To create an example outline interface section for our BankAccount class, we would use the following:

```
@interface BankAccount: NSObject
{

}
@end
```

The parent class chosen above is the *NSObject* class. This is a standard base class provided with the Objective-C Foundation framework and is the class from which most new classes are derived. By deriving BankAccount from this parent class we inherit a range of additional methods used in creating, managing and destroying instances that we would otherwise have to write ourselves.

Now that we have the outline syntax for our class, the next step is to add some instance variables to it.

8.4 Adding Instance Variables to a Class

A key goal of object oriented programming is a concept referred to as *data encapsulation*. The idea behind data encapsulation is that data should be stored within classes and accessed only through methods defined in that class. Data encapsulated in a class are referred to as *instance variables*.

Instances of our BankAccount class will be required to store some data, specifically a bank account number and the balance currently held by the account. Instance variables are declared in the same way any other variables are declared in Objective-C. We can, therefore, add these variables as follows:

```
@interface BankAccount: NSObject
{
        double accountBalance;
        long accountNumber;
}
@end
```

Having defined our instance variables, we can now move on to defining the methods of the class that will allow us to work with our instance variables while staying true to the data encapsulation model.

8.5 Define Class Methods

The methods of a class are essentially code routines that can be called upon to perform specific tasks within the context of an instance of that class.

Methods come in two different forms, *class methods* and *instance methods*. Class methods operate at the level of the class, such as creating a new instance of a class. Instance methods, on the other hand, operate only on the instance of a class (for example performing an arithmetic operation on two instance variables and returning the result). *Class methods* are preceded by a plus (+) sign in the declaration and instance methods are preceded by a minus (-) sign. If the method returns a result, the name of the method must be preceded by the data type returned enclosed in parentheses. If a method does not return a result, then the method must be declared as *void*. If data needs to be passed through to the method (referred to as *arguments*), the method name is followed by a colon, the data type in parentheses and a name for the argument. For example, the declaration of a method to set the account number in our example might read as follows:

```
-(void) setAccountNumber: (long) y;
```

The method is an *instance method* so it is preceded by the minus sign. It does not return a result so it is declared as *(void)*. It takes an argument (the account number) of type *long* so we follow the *accountNumber* name with a colon (:) specify the argument type *(long)* and give the argument a name (in this case we simply use *y*).

The following method is intended to return the current value of the account number instance variable (which is of type *long*):

```
-(long) getAccountNumber;
```

Methods may also be defined to accept more than one argument. For example to define a method that accepts both the account number and account balance we could declare it as follows:

```
-(void) setAccount: (long) y andBalance: (double) x;
```

Now that we have an understanding of the structure of method declarations within the context of the class *interface* definition, we can extend our BankAccount class accordingly:

```
@interface BankAccount: NSObject
{
        double accountBalance;
        long accountNumber;
}

-(void) setAccount: (long) y andBalance: (double) x;
-(void) setAccountBalance: (double) x;
-(double) getAccountBalance;
-(void) setAccountNumber: (long) y;
-(long) getAccountNumber;
-(void) displayAccountInfo;
@end
```

Having defined the interface, we can now move on to defining the *implementation* of our class.

8.6 Declaring an Objective-C Class Implementation

The next step in creating a new class in Objective-C is to write the code for the methods we have already declared. This is performed in the *@implementation* section of the class definition. An outline implementation is structured as follows:

```
@implementation NewClassName
    ClassMethods
@end
```

In order to implement the methods we declared in the *@interface* section, therefore, we need to write the following code:

```
@implementation BankAccount

-(void) setAccount: (long) y andBalance: (double) x;
{
        accountBalance = x;
        accountNumber = y;
}

-(void) setAccountBalance: (double) x
{
        accountBalance = x;
}

-(double) getAccountBalance
{
        return accountBalance;
}

-(void) setAccountNumber: (long) y
{
        accountNumber = y;
}

-(long) getAccountNumber
{
        return accountNumber;
}

-(void) displayAccountInfo
{
        NSLog (@"Account Number %li has a balance of %f", accountNumber,
accountBalance);
}
@end
```

We are now at the point where we can write some code to work with our new BankAccount class.