iPhone iOS 5 Development Essentials

iPhone iOS 5 Development Essentials – First Edition

ISBN-13: 978-1466337275

Rev 2.3p

Table of Contents

Preface

This publication represents the third edition of the iPhone Application Development Essentials series of books. The first edition addressed iOS 4 development for the iPhone using Xcode 3 whilst the second revision was updated for the release of Xcode 4. This current edition has been fully updated to coincide with the public release of iOS 5.

This latest SDK release is by far the most significant upgrade to the operating system and development kit in the history of iOS and introduces new features such as iCloud support, Twitter integration, new document handling paradigms and automatic reference counting. The revision of this book was, similarly, no small undertaking involving over five months of work that evolved through the course of no fewer than seven iOS 5 SDK Beta releases prior to the final release. Much of the new content in this book would not have been possible without the timely responses from Apple's iOS development team to beta SDK bug reports and also without the persistence and support of other iOS 5 beta testers who were also working through the learning curve of the new features of iOS 5 with little in the way of documentation for guidance.

Without a doubt now is an exciting time to be an application developer. Prior to the iPhone and the App Store a developer was responsible for finding a way to bring a completed application to market. Success invariably went to the developer with the largest marketing budget or most web search traffic. The App Store, however, has brought a more level playing field to the application market with both large and small developers given equal status within the marketplace. Never before have hundreds of millions of users been required to go to a single place to locate and purchase software applications. Every one of those users is a potential customer for your applications.

Before you begin your journey into the world of iPhone application development I'd like to begin by imparting some things I have learned that don't necessarily relate to the actual coding of an application. Firstly, try to choose an application idea that fits in with something that interests you. Both the development and subsequent sales of an application can be a rollercoaster ride of ups and downs. Addressing a market about which you are passionate will provide the drive you need to overcome any obstacles.

Secondly, check Apple's terms and conditions for acceptance into the App Store before beginning development work. There is nothing worse than spending months of effort developing an application only to have it declined during the application review process. That said, however, do not be too discouraged by an initial rejection. Apple allows, and indeed encourages, developers to modify and resubmit applications for review, and minor adjustments based on Apple's feedback will often lead to subsequent acceptance into the App Store.

Thirdly, try to get the application as feature complete and polished as possible before submitting it to the App Store (but not the extent that you never actually finish it). If possible, get unbiased feedback from friends, colleagues and potential customers before going live. This will increase the chance that early reviews of the application will be positive thereby leading to more downloads.

Finally, remember to have fun. Opportunities as exciting as the smartphone application market come along in the technology business once every 10 to 20 years. You are right in the middle of a vast shift in the way that users acquire and use software applications. Once you've mastered the skills necessary to develop iPhone applications there really are no limits. Take this opportunity to learn the necessary skills and then, in the words of the late Steve Jobs, use this knowledge to build something "insanely great".

Chapter 1

1. About iPhone iOS 5 App Development Essentials

Although the technology marketplace appears to evolve at a rapid and continuous pace, much of this progress takes place in the form of incremental improvements. It is only once every 10 to 20 years that new technology truly results in sweeping changes to both the technology industry and consumer behavior. The late 1970s and early 1980s, for example, witnessed the introduction of the personal computer. The late 1990s, of course, saw the widespread adoption of the internet. A little over a decade later we are in the middle of yet another revolution in the form of smartphones and tablets.

In actual fact there are many similarities between the PC and smartphone revolutions. Both resulted in the widespread adoption of new technology by businesses and consumers alike. Both caused a massive surge of development activity resulting in large numbers of new applications being written. Perhaps most interestingly, however, both were triggered to a large extent by the actions of Apple, Inc. first with the introduction of the Apple II computer in 1977, then again 30 years later with the iPhone in 2007.

The iPhone and its peers in the smartphone market are remarkable technological achievements. In a device small enough to put in your pocket the iPhone can make phone calls, send and receive email, SMS and MMS messages, stream and play audio and video, detect movement and rotation, vibrate, adapt the display brightness based on the ambient lighting, surf the internet, run apps from a selection of hundreds of thousands, take high resolution photos, record video, tell you your exact location, provide directions to your chosen destination, play graphics intensive games and even detect when you put the device to your ear.

Perhaps the most amazing thing about the iPhone is that all of these capabilities and hardware features are available to you as an app developer. In fact, once you have an iPhone, an Intel-based Mac computer, the iOS SDK, a copy of the Xcode development environment and the necessary skills, the only limit to the types of apps you can create is your own imagination (and, of course, the restrictions placed on apps accepted into the Apple App Store).

The subject of this book is version 5 of the iOS operating system within the context of the iPhone. iOS 5 introduces a wide range of new opportunities for the iPhone application developer to utilize. Beginning with the basics, this book provides an overview of the iPhone hardware and the architecture of iOS 5. An introduction to programming in Objective-C is provided followed by an in-depth look at the design of iPhone applications and user interfaces. More advanced topics such as file handling, database management, 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. New iOS 5 specific features are also covered including page view controller implementation, iCloud based storage, Storyboard user interface design, image filtering with Core Image and Twitter integration.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for the iPhone. Assuming you have downloaded the iOS 5 SDK and Xcode, have an Intel-based Mac and some ideas for some apps to develop, you are ready to get started.

1.1 Example Source Code

The source code and Xcode project files for the examples contained in this book are available for download at *http://www.ebookfrenzy.com/code/iphoneios5.zip*.

1.2 Feedback

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

Chapter 2

2. The Anatomy of an iPhone 4S

Most books covering the development of apps for the iPhone tend to overlook the underlying hardware of the device and instead dive immediately into the software development environment. This is a shame because the iPhone is an incredible technical achievement that we are already starting to take for granted.

Take, for example, the iPhone 4S. This is a sleek device that is 115.2mm long, 58.6mm wide and 9.3 mm deep. Now, compare the size of your laptop or desktop computer to your iPhone. Then take a look at the specification for your computer and see if it has built in GPRS, EDGE and 3G wireless support, a digital compass, GPS, an accelerometer, a gyroscope, a proximity sensor, an ambient light sensor, Bluetooth capability, Wi-Fi, a multi-touch screen, a vibration generator and an 8 megapixel autofocus camera with built in flash and a second, 30 frame per second front facing camera. The chances are your much larger and heavier computer has only a small subset of these features. Next, check the expected battery life of your laptop and see if it will allow you to play music for 40 hours or video for 10 hours without needing a recharge. When you consider these capabilities you will hopefully begin to appreciate the engineering achievements behind the iPhone and other similar smartphone devices.

Now that we have set the scene, we can move on to discuss some of the hardware features built into the iPhone in a little more detail. Once again, we will do this within the context of the iPhone 4S.

2.1 iOS 5

Before we delve into the hardware of the iPhone we will start by talking about the operating system that sits on top of all the hardware. This operating system is called iOS 5 and is a variant of Apple's Mac OS X operating system which has been adapted to run on the iPhone. It is built upon a "UNIX-like" foundation called Darwin and consists of the Mach kernel, core services and media layers and the Cocoa Touch interface. iOS 5 is covered in greater detail in the chapter entitled *iOS 5 Architecture and SDK Frameworks*.

2.2 Display

The iPhone 4S has a 3.5 inch display with a resolution of 960 x 640 pixels capable of displaying 326 pixels per inch (ppi) with an 800:1 contrast ratio. The underlying technology is an In Plane Switching (IPS) LED, capacitive touch screen. The screen has a scratch, oil and fingerprint resistant oleophobic coated surface and includes a proximity sensor which automatically turns off the screen when you put the phone to your ear (presumably to extend the battery life during a phone call and to avoid making user interface selections with your ear). The device also has ambient light detection which adjusts the screen

brightness to ensure the optimal screen visibility in a variety of lighting conditions from bright sunlight to darkness.

2.3 Wireless Connectivity

The iPhone 4S supports a wide range of connectivity options. When within range of a Wi-Fi network, the device can connect at either 802.11b, 802.11g or 802.11n speeds.

For making phone calls or transferring data when not connected to Wi-Fi, the AT&T device supports GSM/EDGE connectivity (otherwise known as 2G). For faster speeds, support is also provided for connectivity via Universal Mobile Telecommunications System (UMTS), High-Speed Downlink Packet Access (HSDPA) and High Speed Uplink Packet Access (HSUPA). This is better known as 3G and provides data transfer speeds of up to 7.2 megabits per second.

The iPhone 4S also includes Bluetooth v4.0 support with Enhanced Data Rate (EDR) technology.

2.4 Wired Connectivity

Given the wide array of wireless options it is not surprising that the iPhone has little need for wired connections. In fact the iPhone only has two. One is a standard 3.5 mm headset jack for the attachment of headphones or other audio devices. The second is a proprietary, 30-pin dock connector which, by default, is used to provide a USB v2.0 connection for synching with a computer system and battery charging. In practice, however, this connection also provides audio and TV output via specialty cables.

2.5 Memory

The iPhone 4S comes in three editions, containing 16GB, 32GB and 64GB of memory respectively. The memory is in the form of a flash drive. Unlike some devices, the iPhone lacks the ability to supplement the installed memory by inserting additional flash memory cards.

2.6 Cameras

The iPhone 4S contains a 8 megapixel autofocus still camera which may also be used to record video at an HD resolution of 1080p included image stabilization and temporal noise reduction. In addition, the device also incorporates an LED flash and a VGA resolution, 30 fps front facing camera.

2.7 Sensors

The latest generation of iPhone has an array of sensors which would make even the most die-hard 1960s science fiction fan jealous. These consist of a proximity sensor which detects when the front of the phone is covered or otherwise obscured, an accelerometer which uses the pull of gravity to detect when the device is moved or rotated, a three-axis gyroscope and an ambient light sensor to detect current environmental light levels.

2.8 Location Detection

The iPhone 4S contains a digital compass and GPS support with Assisted GPS (A-GPS) support. Essentially this enables the iPhone to detect the direction the device is facing and to identify the current location by detecting radio signals from GPS satellites. In the event that GPS signals are unavailable or too weak to establish the current coordinates, the iPhone can also gain an approximate location using cellular and Wi-Fi information.

2.9 Central Processing Unit (CPU)

The central processing unit (CPU) of the iPhone 4S is the Apple A5, an Apple designed system-on-a-chip (SoC) consisting of a dual core ARM Cortex A9 chip combined with a dual core graphics processing unit (GPU). The Cortex A9 processor is designed by a British company named ARM Holdings which specializes in designing chips and then licensing those designs to third parties who then manufacture them. This differs considerably from the approach taken by companies such as Intel which both design and manufacture their own chips.

2.10 Speaker and Microphone

As with most other phones on the market, the iPhone includes both a built-in microphone and a speaker to enable the use of the device as a speakerphone. Both the speaker and microphone may be used by third party apps, though as is to be expected with a device the size of an iPhone, the sound quality of the speaker is widely considered to be poor.

2.11 Vibration

Though initially provided as a “silent ring” feature whereby the device vibrates to indicate an incoming call as an alternative to a ring tone (a feature common to most mobile phone devices), the vibration feature of the iPhone may also be used within applications to notify the user of a new event (such as a breaking news story) or to provide tactile feedback such as for an explosion in a game.

2.12 Summary

As we have seen in this chapter, the iPhone packs an impressive amount of technology into a very small amount of space. Perhaps the most exciting aspect of all this technology is that you can, almost without exception, access and utilize all this hardware within your own applications.

Chapter 3

3. iOS 5 Architecture and SDK Frameworks

In *The Anatomy of an iPhone 4S* we looked at the hardware contained within an iPhone 4S device. When we develop apps for the iPhone, 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, this operating system is known as iOS.

In order to gain a better understanding of the iPhone 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 applications.

3.1 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.

3.2 An Overview of the iOS 5 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 3-1:

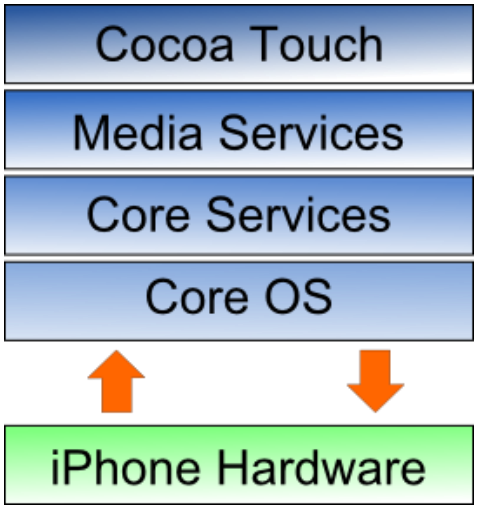

Figure 3-1

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 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 5 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.

3.3 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 iPhone 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 hardware.

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

3.3.1 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

3.3.2 Map Kit Framework (MapKit.framework)

If you have spent any appreciable time with an iPhone 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.

3.3.3 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.

3.3.4 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.

3.3.5 Address Book UI Framework (AddressUI.framework)

Given that a key function of the iPhone is as a communications device and digital assistant 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 iPhone address book from within your own application.

3.3.6 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.

3.3.7 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.

3.3.8 Event Kit UI Framework

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

3.3.9 Accounts Framework (Accounts.framework)

iOS 5 introduces 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 accounts, though other services such as Facebook 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.

3.3.10 Twitter Framework (Twitter.framework)

The Twitter Framework allows Twitter integration to be added to applications. The framework operates in conjunction the Accounts Framework to gain access to the user's Twitter account information.

3.4 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 iPhone apps. In this section we will look at each one in turn.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 Assets Library Framework (AssetsLibrary.framework)

The Assets Library provides a mechanism for locating and retrieving video and photo files located on the iPhone 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.

3.4.5 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.

3.4.6 Core Image Framework (CoreImage.framework)

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

3.4.7 Quartz Core Framework (QuartzCore.framework)

The purpose of the Quartz Core framework is to provide animation capabilities on the iPhone. 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 iPhone apps.

3.4.8 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.

iOS 3 or later supports both OpenGL ES 1.1 and 2.0 on certain iPhone models (such as the iPhone 3GS and iPhone 4). Earlier versions of iOS and older device models support only OpenGL ES version 1.1.

3.4.9 GLKit Framework (GLKit.framework)

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

3.4.10 NewsstandKit Framework (NewsstandKit.framework)

The Newsstand application is a new feature of iOS 5 and is intended as a central location for users to gain access to newspapers and magazines. The NewsstandKit framework allows for the development of applications that utilize this new service.

3.4.11 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.

3.4.12 AV Foundation framework (AVFoundation.framework)

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

3.4.13 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.

3.4.14 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.

3.4.15 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.

3.4.16 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's dock connector.

3.5 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.

3.5.1 Address Book Framework (AddressBook.framework)

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

3.5.2 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).

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 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.

3.5.7 EventKit Framework (EventKit.framework)

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

3.6 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.

3.6.1 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 with 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 iPhone model on which the app is running (GPS and compass are only featured on recent models). This will either be based on GPS readings, Wi-Fi network data or cell tower triangulation (or some combination of the three).

3.6.2 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.

3.6.3 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.

3.6.4 SQLite library

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

3.6.5 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).

3.6.6 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).

3.7 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.

3.7.1 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.

3.7.2 External Accessory Framework (ExternalAccessory.framework)

Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone via the 30-pin dock connector or wirelessly via Bluetooth.

3.7.3 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).

3.7.4 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.

Chapter 4

4. Joining the Apple iOS Developer Program

The first step in the process of learning to develop iOS 5 based iPhone 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.

4.1 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.

In order to obtain the latest versions of both the iOS SDK and Xcode, registered developers must either purchase them from the Mac App Store or enroll in the iOS Developer Program. The latest iOS SDK and Xcode package costs $4.99 to purchase from the Mac App Store and may be found at the following location:

http://itunes.apple.com/us/app/xcode/id422352214?mt=12&ls=1

This raises the question of whether to upgrade to the iOS Developer Program, or to remain as a Registered Apple Developer and simply purchase the latest iOS SDK and Xcode package. It is important, therefore, to understand the key benefits of the iOS Developer Program.

4.2 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 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.

4.3 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 by spending the nominal fee to purchase the latest iOS SDK and Xcode bundle. 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.

4.4 Enrolling in the iOS Developer Program

If your goal is to develop iPhone 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 4-1

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 4-2:

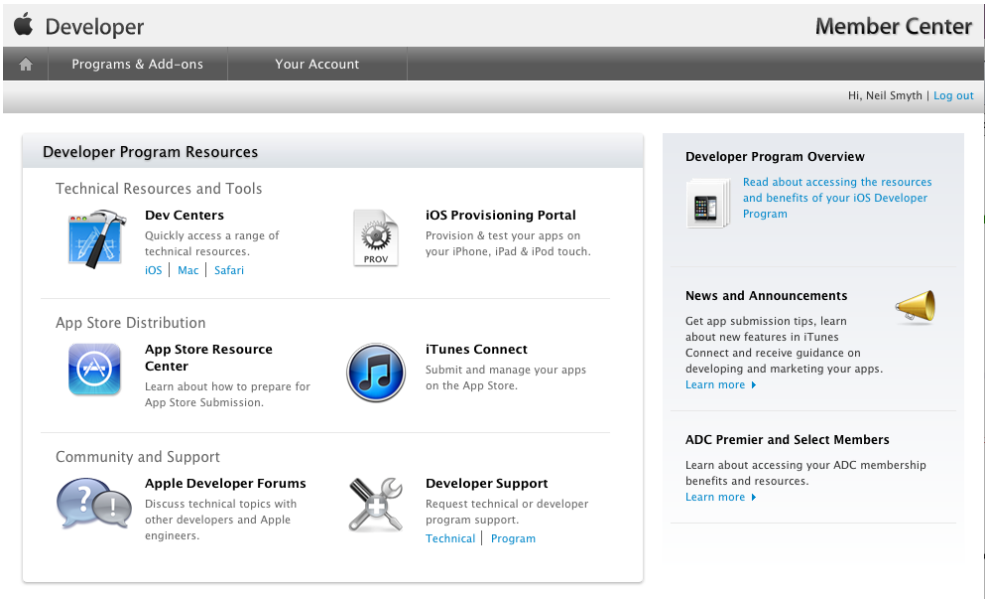

Figure 4-2

4.5 Summary

An important early step in iPhone iOS 5 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 5 SDK and Xcode development environment.

Chapter 5

5. Installing Xcode 4 and the iOS 5 SDK

iPhone apps are developed using the iOS SDK in conjunction with Apple's Xcode 4 development environment. The iOS SDK contains the development frameworks that were outlined in *iOS 5 Architecture and Frameworks*. Xcode 4 is an integrated development environment (IDE) within which you will code, compile, test and debug your iOS iPhone applications. The Xcode 4 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 4 and the iOS 5 SDK on Mac OS X.

5.1 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 iPhone app development project. If you are unsure of the processor type inside your Mac, you can find this information by opening the Finder and selecting the *About This Mac* option from the Apple menu. In the resulting dialog check the *Processor* line. Figure 5-1 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 iPhone iOS app development.

In addition, the iOS 5 SDK with Xcode 4.2 environment requires that the version of Mac OS X running on the system be version 10.6.6 or later. If the "About This Mac" dialog does not indicate that Mac OS X 10.6.6 or later is running, click on the *Software Update…* button to download and install the appropriate operating system upgrades.

Figure 5-1

5.2 Installing Xcode 4 and the iOS 5 SDK

The best way to obtain the latest versions of Xcode 4 and the iOS SDK is to download them from the Apple iOS Dev Center web site at:

http://developer.apple.com/devcenter/ios/index.action

In order to download Xcode 4 with the iOS 5 SDK, you will either need to be a member of the iOS Developer programs or purchase a copy from the Mac App Store at:

http://itunes.apple.com/us/app/xcode/id422352214?mt=12&ls=1

The download is over 3.5GB in size and may take a number of hours to complete depending on the speed of your internet connection. The package takes the form of a disk image (.dmg) file. Once the download has completed, a new window will open as follows displaying the contents of the .dmg file:

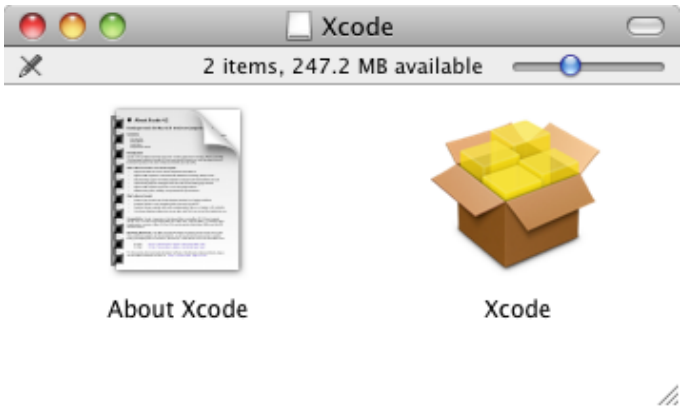

Figure 5-2

If this window does not open by default, it can be opened by clicking on the SDK disk drive icon on the desktop or by navigating to the Downloads directory of your home folder and double clicking on the corresponding dmg file.

Initiate the installation by double clicking on the package icon (the one that looks like an opening box) and follow the instructions until you reach the *Custom Install* screen:

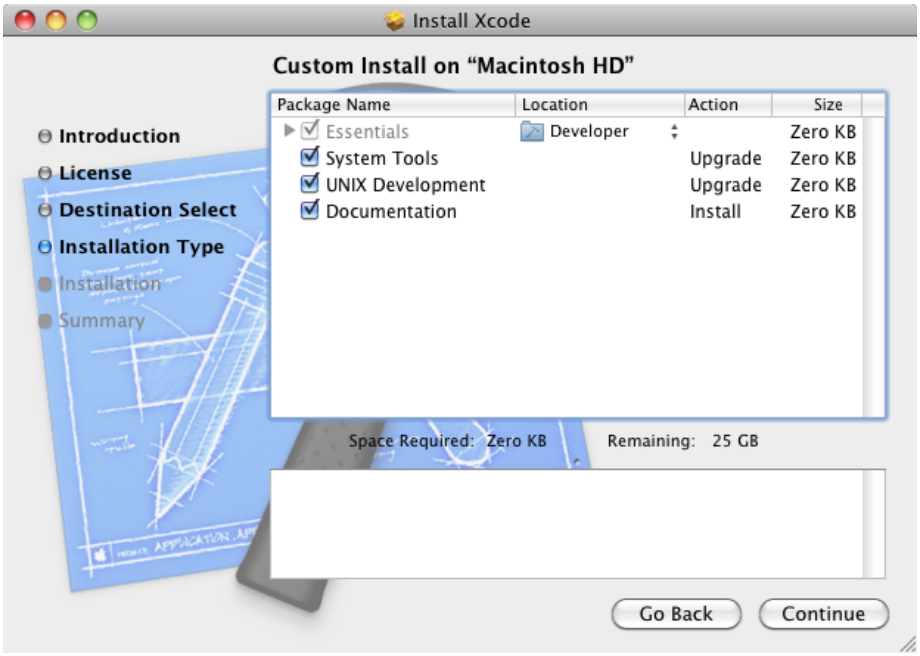

Figure 5-3

The default selections on this screen are adequate for most requirements so unless you have specific needs there is no necessity to alter these selections. Continue to the next screen, review the information and click *Install* to begin the installation. Note that you may first be prompted to enter your password as a security precaution. The duration of the installation process will vary depending on the speed and current load on the computer, but typically completes in 25 - 45 minutes.

5.3 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 iPhone 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.

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 5-4

Having installed the iOS 5 SDK and successfully launched Xcode 4 we can now look at *Creating a Simple iPhone iOS 5 App*.

Chapter 6

6. Creating a Simple iPhone iOS 5 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, *iPhone iOS 5 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 5 and the iPhone.

6.1 Starting Xcode 4

As with all iOS examples in this book, the development of our example will take place within the Xcode 4 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 4 and the iOS 5 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 6-1 will appear by default:

Figure 6-1

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 4 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:

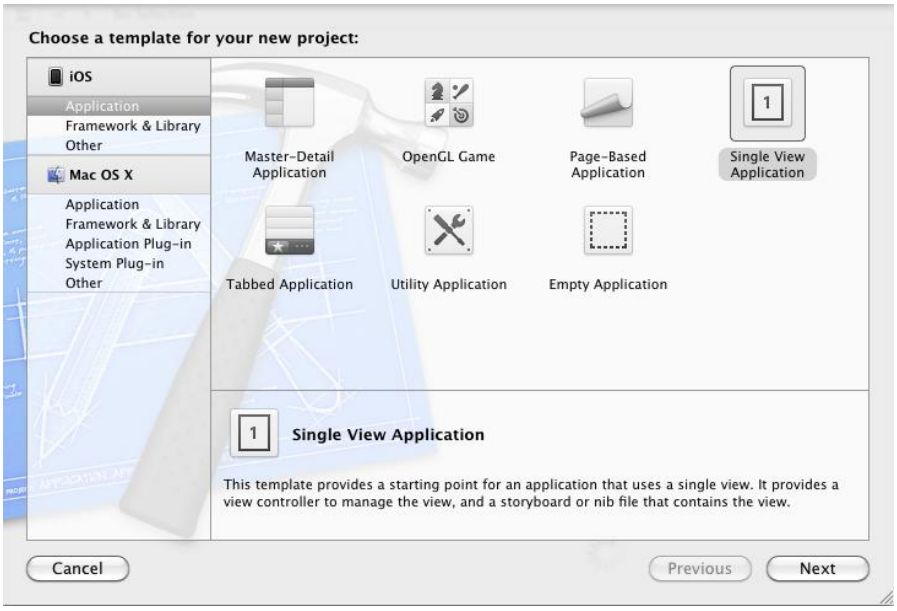

Figure 6-2

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 an iOS based device 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** – As discussed in *iOS 5 Architecture and SDK Frameworks*, 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** – Creates a template consisting of a two sided view. For an example of a utility application in action, load up the standard iPhone weather application. Pressing the blue info button flips the view to the configuration page. Selecting *Done* rotates the view back to the main screen.
- **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.

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:

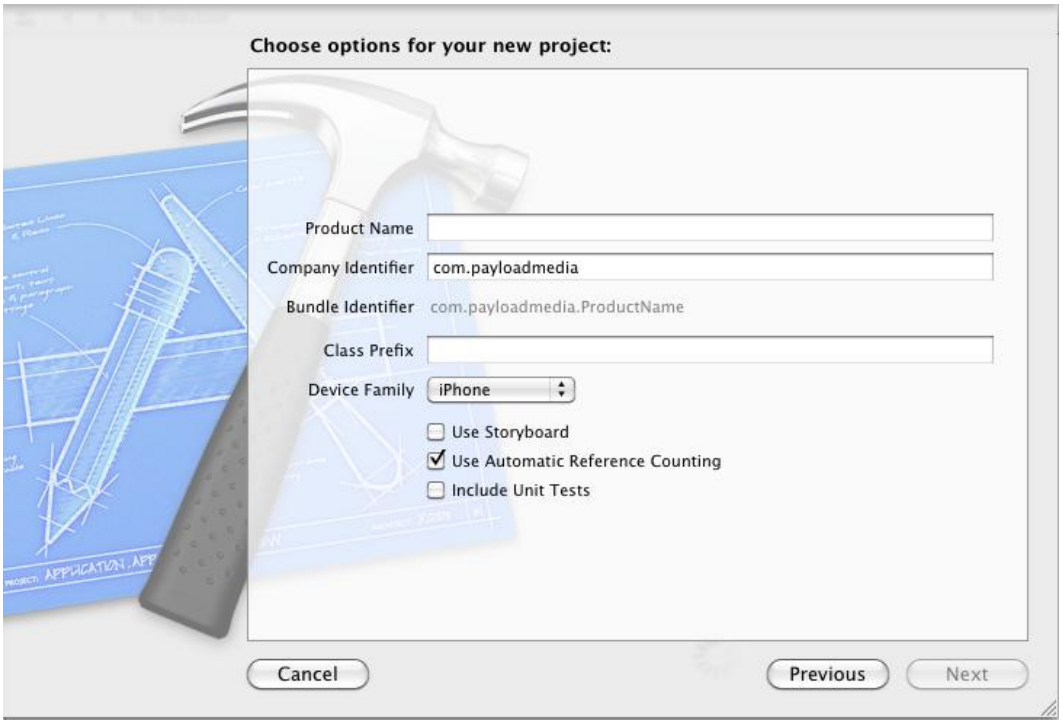

Figure 6-3

On this screen, enter a Product name for the application that is going to be created, in this case "HelloWorld" and make sure that the class prefix matches this name. 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 device (covered in more detail in *Testing iOS 5 Apps on the iPhone – Developer Certificates and Provisioning Profiles*). 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 *Device Family* menu and that neither the *Use Storyboard* nor the *Include Unit Tests* options are currently selected.

Automatic Reference Counting is a new feature included with the Objective-C compiler which removes much of the responsibility from the developer for releasing objects when they are no longer needed. This is an extremely useful new feature and, as such, the option should be selected 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 can click on *Create.*

Once the new project has been created the main Xcode window will appear as illustrated in Figure 6-4:

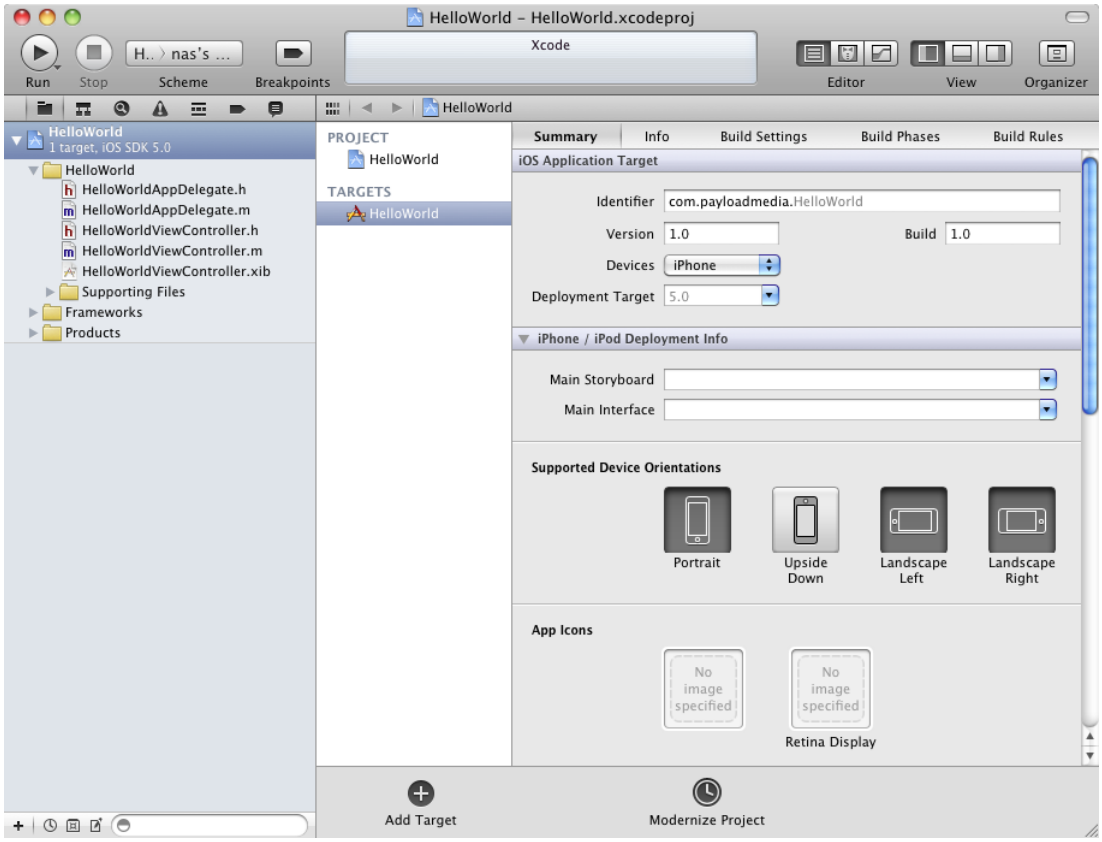

Figure 6-4

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 .xib file is the save file used by the Interface Builder tool to hold the user interface design we will create. Older versions of Interface Builder saved designs in files with a .nib extension so these files, even today, are called NIB files. 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 summary of the settings for the application. 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 an icon (the small image the user selects on the device screen to launch the application) and splash screen image (displayed to the user while the application loads) for the application.

In addition to the Summary screen, tabs are provided to view and modify additional settings consisting of 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 Summary 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.

6.2 Creating the iOS App User Interface

Simply by the very nature of the environment in which they run, iPhone 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 .xib filename extension. This is an Interface Builder save file (remember that they are called NIB files, not XIB files). The file we are interested in for our HelloWorld project is called *HelloWorldViewController.xib.* 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 6-5:

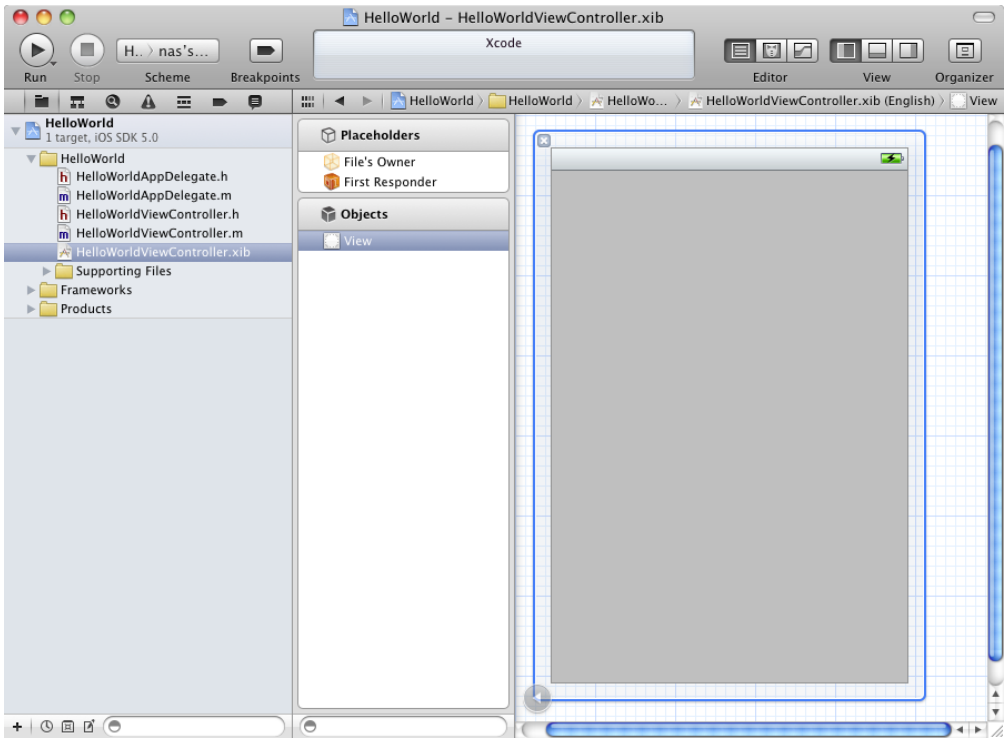

Figure 6-5

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. This is achieved by selecting the right hand button in the *View* section of the Xcode toolbar:

Figure 6-6

The right hand panel, once displayed, will appear as illustrated in Figure 6-7:

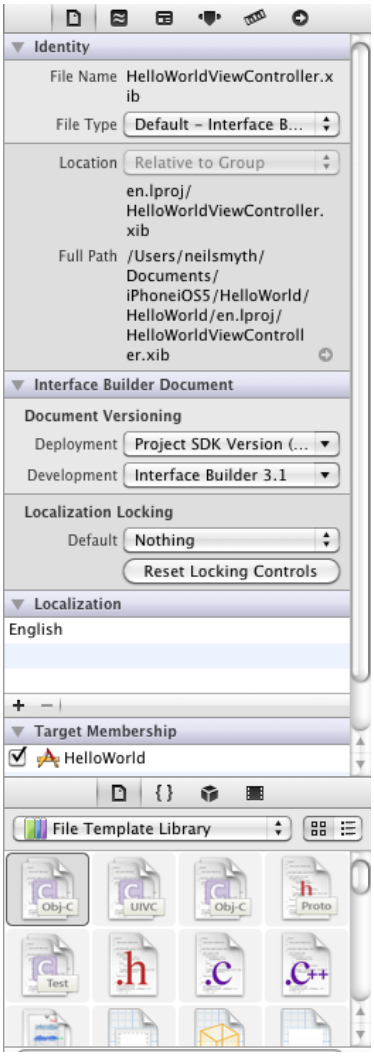

Figure 6-7

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 defaults 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 in the appropriate toolbar button (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. In addition, the objects are categorized into groups which may be selected using the menu beneath the toolbar. The layout buttons may also be used to switch from a single column of objects with descriptions to multiple columns without descriptions.

6.3 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. Begin by making sure the View is selected and that the Attribute Inspector (*View -> Utilities -> Show Attribute Inspector*) is displayed in the right hand panel. Click on the gray 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 gray to the new color selection.

6.4 Adding Objects to the User Interface

The next step is to add a Label object to our view. To achieve this, select *Cocoa Touch -> Controls* from the library panel menu, click on the *Label* object and drag it to the center of the view. Once it is in position release the mouse button to drop it at that location:

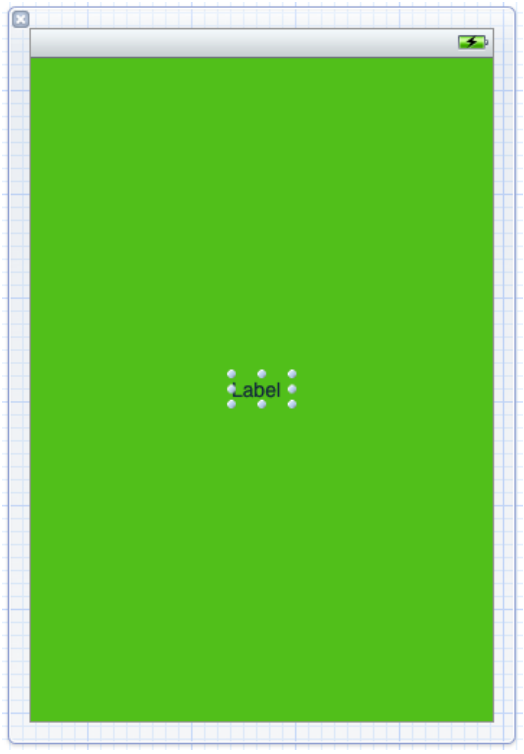

Figure 6-8

Using the blue 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 *Layout* attribute section of the Attribute Inspector (*View -> Utilities -> Show Attribute Inspector*) to center the text in the middle of the screen. Click on the current font attribute setting to choose a larger 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 6-9 (allowing, of course, for differences in your color and font choices):

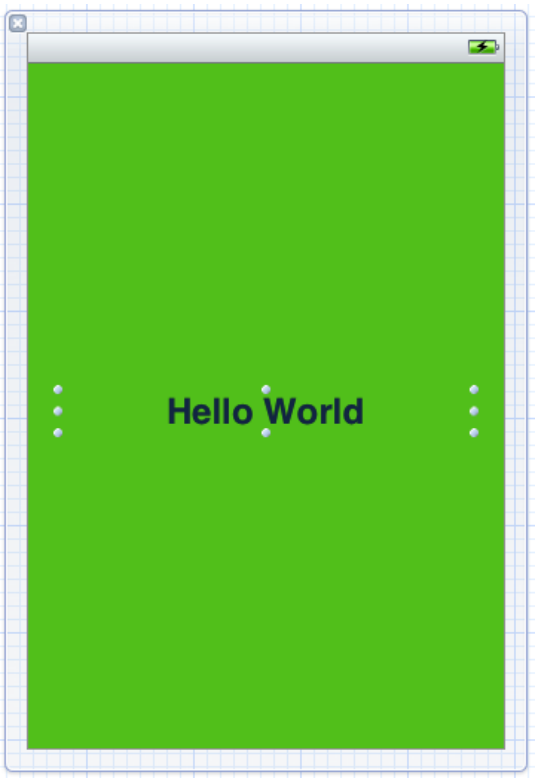

Figure 6-9

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.

6.5 Building and Running an iOS App in Xcode 4

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 iOS 5 Apps on the iPhone – Developer Certificates and Provisioning Profiles*. For the purposes of this chapter, however, it is sufficient to run the app in the simulator.

Within the main Xcode 4 project window make sure that the menu located in the top left hand corner of the window (to the right of the Stop button) has the *iPhone Simulator* option selected and then click on the *Run* toolbar 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 6-10

6.6 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 4 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.

Chapter 7

7. Testing iOS 5 Apps on the iPhone – Developer Certificates and Provisioning Profiles

In the chapter entitled *Creating a Simple iPhone iOS 5 App* we were able to run an application in the iOS Simulator environment bundled with the iOS 5 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 iPhone device.

In order to achieve this a number of steps are required. These include generating and installing a developer certificate, creating an App ID and provisioning profile for your application, and registering the devices onto which you wish to directly install your apps for testing purposes. In the remainder of this chapter we will cover these steps in detail.

Note that the provisioning of physical devices requires membership in the iOS Developer Program, a topic covered in some detail in the chapter entitled *Joining the Apple iOS Developer Program*.

7.1 Creating an iOS Development Certificate Signing Request

Any apps that are to be installed on a physical iPhone device must first be signed using an iOS Development Certificate. In order to generate a certificate the first step is to generate a Certificate Signing Request (CSR). Begin this process by opening the Keychain Access tool on your Mac system. This tool is located in the *Applications -> Utilities* folder. Once launched, the Keychain Access main window will appear as illustrated in Figure 7-1:

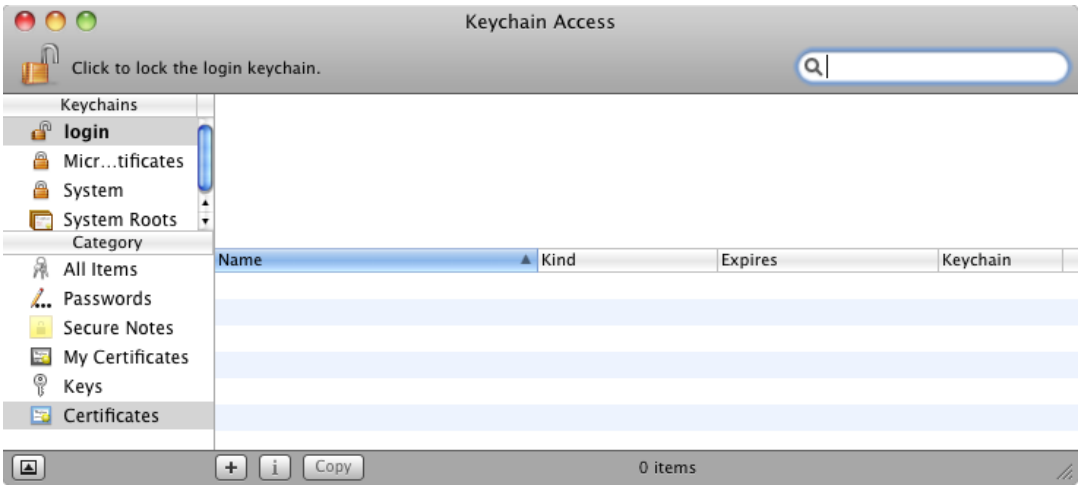

Figure 7-1

Within the Keychain Access utility, perform the following steps:

1. Select the *Keychain Access -> Preferences* menu and select *Certificates* in the resulting dialog:

Figure 7-2

2. Within the Preferences dialog make sure that the Online Certificate Status Protocol (OCPS) and Certificate Revocation List (CRL) settings are both set to *Off*, then close the dialog.

3. Select the *Keychain Access -> Certificate Assistant -> Request a Certificate from a Certificate Authority…* menu option and enter your email and name exactly as registered with the iOS Developer Program. Leave the *CA Email Address* field blank and select the *Saved to Disk* and *Let me specify key pair information* options:

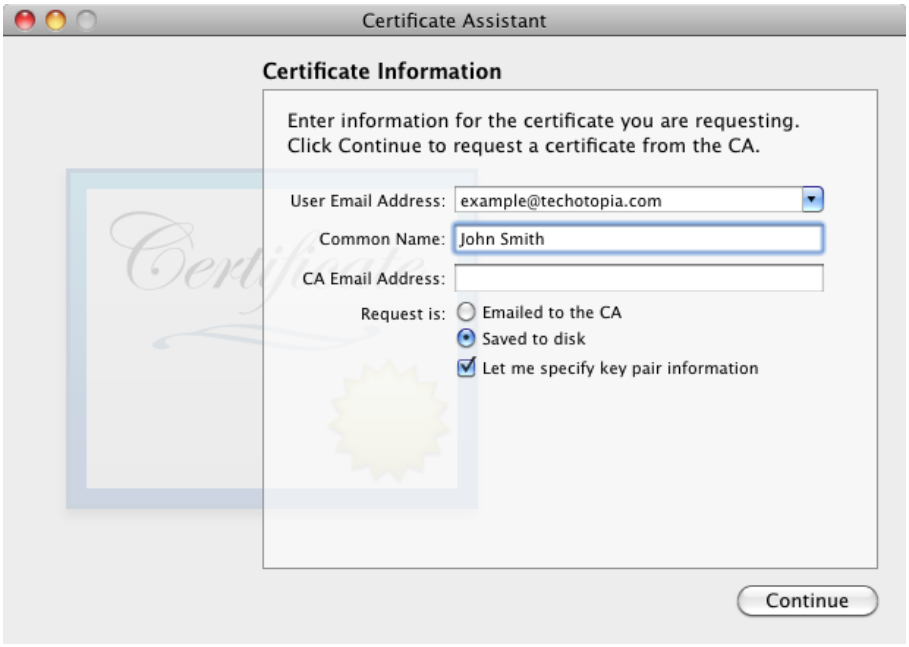

Figure 7-3

4. Clicking the *Continue* button will prompt for a file and location into which the CSR is to be saved. Either accept the default settings, or enter alternative information as desired at which point the *Key Pair Information* screen will appear as illustrated in Figure 7-4:

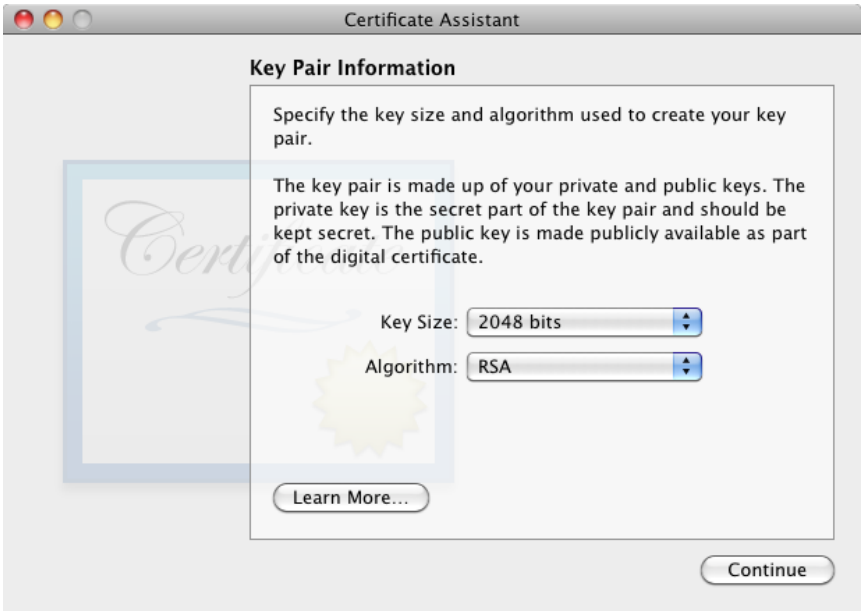

Figure 7-4

5. Verify that the 2048 bits key size and RSA algorithm options are selected before clicking on the *Continue* button. The certificate request will be created in the file previously specified and the *Conclusion* screen displayed. Click *Done* to dismiss the *Certificate Assistant* window.

7.2 Submitting the iOS Development Certificate Signing Request

Having created the Certificate Signing Request (CSR) the next step is to submit it for approval. This is performed within the iOS Provisioning Portal that is accessed from the Member Center of the Apple developer web site. Under *Developer Program Resources* on the main member center home page select *iOS Provisioning Portal.* Within the portal, select the *Certificates* link located in the left hand panel to display the Certificates page:

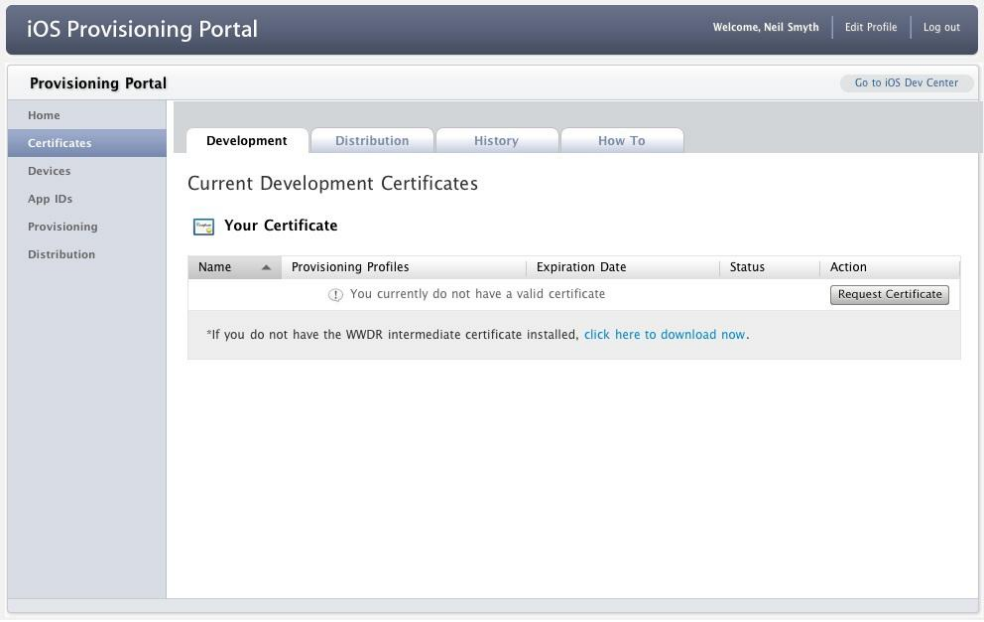

Figure 7-5

Click on the *Request Certificate* button, scroll down to the bottom of the text under the heading *Create an iOS Development Certificate* and click on the *Choose File* button. In the resulting file selection panel, navigate to the certificate signing request file created in the previous section and click on *Choose.* Once your file selection is displayed next to the *Choose File* button, click on the *Submit* button located in the bottom right hand corner of the web page. At this point you will be returned to the main Certificates page where your certificate will be listed as *Pending Issuance.*

Click on the link to download the *WWDR intermediate certificate* and, once downloaded, double click on it to install it into the keychain. This certificate is used by Xcode to verify that your certificates are both valid and issued by Apple.

If you are not the Team Administrator, you will need to wait until that person approves your request. If, on the other hand, you are the administrator for the iOS Developer Program membership you may approve your own certificate request by clicking on the *Approve* button located in the *Action* column of

the *Current Certificates* table. If no approval button is present simply refresh the web page and the certificate should automatically appear listed as *Issued.* Your certificate is now active and the table will have refreshed to include a button to *Download* the certificate:

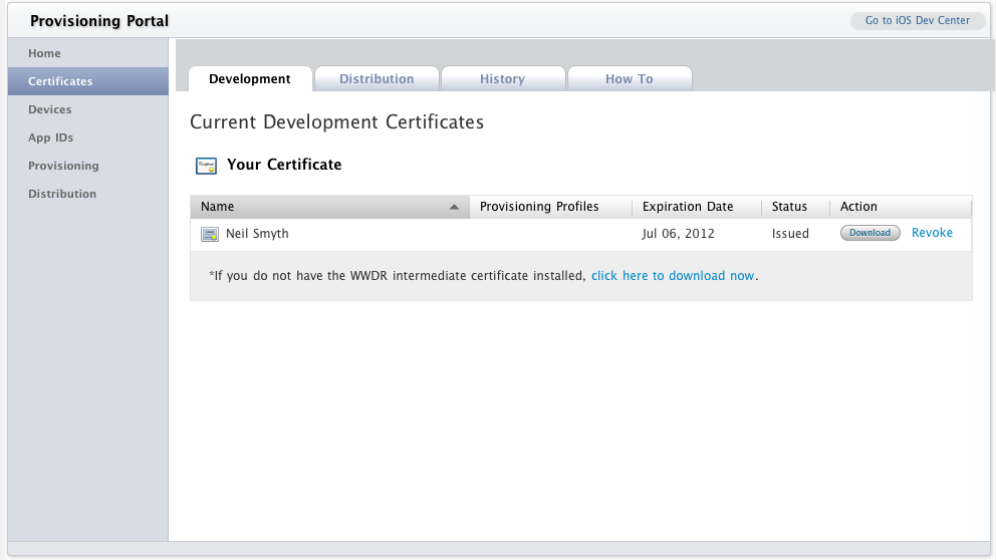

Figure 7-6

7.3 Installing an iOS Development Certificate

Once a certificate has been requested and issued it must be installed on the development system so that Xcode can access it and use it to sign any applications you develop. The first step in this process is to download the certificate from the iOS Provisioning Portal by clicking on the *Download* button located on the Certificates page outlined in the previous section. Once the file has downloaded, double click on it to load it into the Keychain Access tool. The certificate will then be listed together with a status (hopefully one that reads *This certificate is valid):*

Figure 7-7

Your certificate is now installed into your Keychain and you are ready to move on to the next step.

7.4 Assigning Devices

Once you have a development certificate installed, the next step is to specify which devices are to be used to test the iOS apps you are developing. This is achieved by entering the Unique Device Identifier (UDID) for each device into the Provisioning Portal. Note that Apple restricts developers to 100 provisioned devices per year.

A new device may be added to the list of supported test devices either from within the Xcode Organizer window, or by logging into the iOS Developer Portal and manually adding the device. To add a device to the portal from within Organizer, simply connect the device, open the Organizer window in Xcode using the *Organizer* toolbar button, select the attached device from the left hand panel and click on the *Add to Portal* button. The Organizer will prompt for the developer portal login and password before connecting and enabling the device for testing.

Manually adding a device, on the other hand, requires the use of the iPhone's UDID. This may be obtained either via Xcode or iTunes. Begin by connecting the device to your computer using the docking connector. Once Xcode has launched the Organizer window will appear displaying summary information about the device (or may be opened by selecting the *Organizer* button in the Xcode toolbar). The UDID is listed next to the *Identifier* label as illustrated in Figure 7-8:

Figure 7-8

Alternatively, launch iTunes, select the device in the left hand pane and review the Summary information page. One of the fields on this page will be labeled as *Serial Number.* Click with the mouse on this number and it will change to display the UDID.

Having identified the UDIDs of any devices you plan to use for app testing, select the *Devices* link located in the left hand panel of the iOS Provisioning Portal, and click on *Add Devices* in the resulting page. On the *Add Devices* page enter a descriptive name for the device and the 40 character UDID:

Figure 7-9

In order to add more than one device at a time simply click on the "+" button to create more input fields. Once you have finished adding devices click on the *Submit* button. The newly added devices will now appear on the main Devices page of the portal.

7.5 Creating an App ID

The next step in the process is to create an App ID for each app that you create. This ID allows your app to be uniquely identified within the context of the Apple iOS ecosystem. To create an App ID, select the *App IDs* link in the provisioning portal and click on the *New App ID* button to display the *Create App ID* screen as illustrated in Figure 7-10:

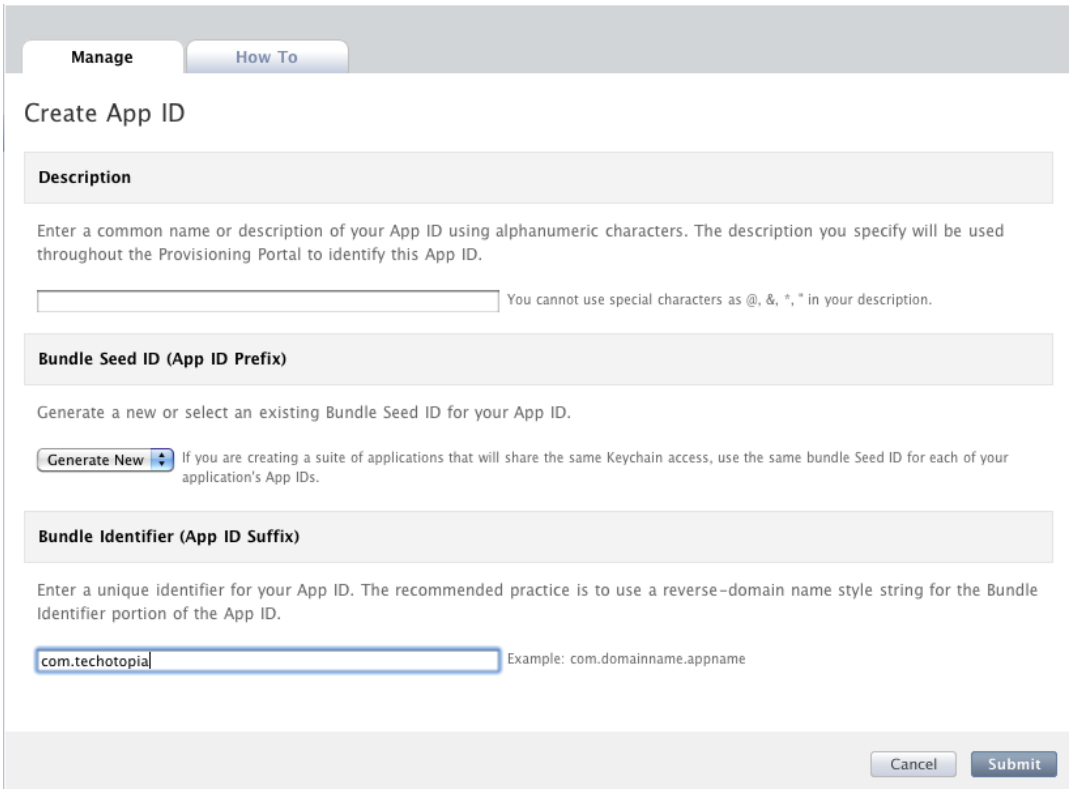

Figure 7-10

Enter a suitably descriptive name into the Description field and then make a Bundle Seed ID selection. If you have not created any previous Seed IDs then leave the default *Generate New* selection unchanged. If you have created a previous App ID and would like to use this for your new app, click on the menu and select the desired ID from the drop down list. Finally enter the Bundle Identifier. This is typically set to the reversed domain name of your company followed by the name of the app. For example, if you are developing an app called *MyApp,* and the URL for your company is *www.mycompany.com* then your Bundle identifier would be entered as:

```
com.mycompany.MyApp
```

If you would like to create an App ID that can be used for multiple apps then the wildcard character (*) can be substituted for the app name. For example:

```
com.mycompany.*
```

Having entered the required information, click on the *Submit* button to return to the main App ID page where the new ID will be listed.

7.6 Creating an iOS Development Provisioning Profile

The Provisioning Profile is where much of what we have created so far in the chapter is pulled together. The provisioning profile defines which developer certificates are allowed to install an application on a device, which devices can be used and which applications can be installed. Once created, the

provisioning profile must be installed on each device on which the designated application is to be installed.

To create a provisioning profile, select the *Provisioning* link in the Provisioning Portal and click on the *New profile* button. In the resulting *Create iPhone Provisioning Profile* screen, perform the following tasks:

1. In the *Profile Name* field enter a suitably descriptive name for the profile you are creating.
2. Set the check box next to each certificate to specify which developers are permitted to use this particular profile.
3. Select an App ID from the menu.
4. Select the devices onto which the app is permitted to be installed.
5. Click on the *Submit* button.

Initially the profile will be listed as *Pending.* Refresh the page to see the status change to *Active.*

Now that the provisioning profile has been created, the next step is to download and install it. To do so, click on the *Download* button next to your new profile and save it to your local system (note that the file will have a *.mobileprovision* file name extension). Once saved, either drag and drop the file onto the Xcode icon in the dock or onto the *Provisioning Profiles* item located under *Library* in the Xcode Organizer window. Once the provisioning profile is installed, it should appear in the Organizer window (Figure 7-11):

Figure 7-11

7.7 Enabling an iPhone Device for Development

With the provisioning profile installed select the target device in the left hand panel of the Organizer window and click on the *Use for Development* button. The Organizer will then prompt you for your Apple developer login and password.

Once a valid login and password have been entered, the Organizer will perform the steps necessary to install the provisioning profile on the device and enable it for application testing.

7.8 Associating an App ID with an App

Before we can install our own app directly onto a device, we must first embed the App ID created in the iOS Provisioning Portal and referenced in the provisioning profile into the app itself. To achieve this:

1. In the left hand panel of the main Xcode window, select the project navigator toolbar button and select the top item (the application name) from the resulting list.

2. Select the *Info* tab from in the center panel:

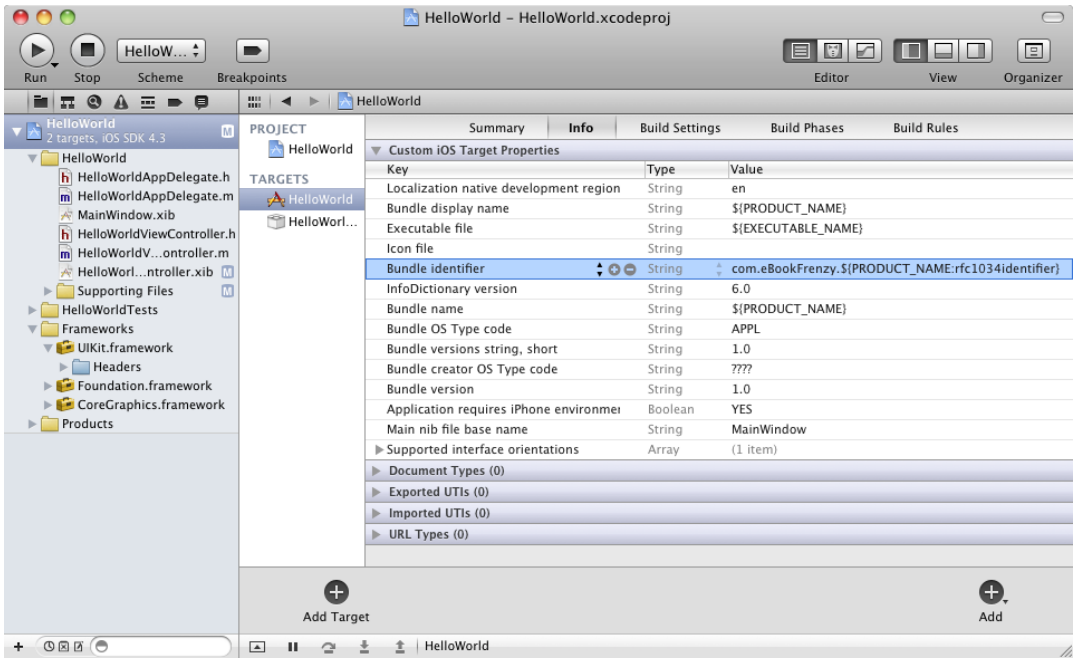

Figure 7-12

In the *Bundle Identifier* field enter the App ID you created in the iOS Provisioning Portal. This can either be in the form of your reverse URL and app name (for example *com.mycompany.HelloWorld*) or you can have the product name substituted for you by entering *com.mycompany.${PRODUCT_NAME:rfc1034indentifer}* as illustrated in Figure 7-12.

Once the App ID has been configured the next step is to build the application and install it onto the iPhone or iPod Touch device.

7.9 iOS and SDK Version Compatibility

Before attempting to install and run an application on a physical iPhone device it is important to be aware of issues relating to version compatibility between the SDK used for the development and the operating system running on the target device. For example, if the application was developed using version 4.3 of the iOS SDK then it is important that the iPhone on which the app is to be installed is running iOS version 4.3 or later. An attempt to run the app on an iPhone with an older version of iOS will result in an error reported by Xcode that reads "Xcode cannot run using the selected device. No Provisioned iOS devices are available. Connect an iOS device or choose an iOS simulator as the destination".

The absence in this message of any indication that the connected device simply has the wrong version of iOS installed on it may lead the developer to assume that a problem exists either with the connection or with the certification or provisioning profile. If you encounter this error message, therefore, it is worth checking version compatibility before investing what typically turns into many hours of effort trying to resolve non-existent connectivity and provisioning problems.

7.10 Installing an App onto a Device

Located in the top left hand corner of the main Xcode window is drop down menu which, when clicked, provides a menu of options to control the target run environment for the current app.

If either the iPhone or iPad simulator option is selected then the app will run within the corresponding simulated environment when it is built. To instruct Xcode to install and run the app on the device itself, simply change this menu to the *iOS Device* setting. Assuming the device is connected, click on the *Run* button and watch the status updates as Xcode compiles and links the source code. Once the code is built, Xcode will need to sign the application binary using your developer certificate. If prompted with a message that reads "codesign wants to sign using key "<key name>" in your keychain", select either *Allow* or *Always Allow* (if you do not wish to be prompted during future builds). Once signing is complete the status will change to "Installing <appname>.app on iPhone…". After a few seconds the app will be installed and will automatically start running on the device where it may be tested in a real world environment.

7.11 Summary

Without question, the iOS Simulator included with the iOS 5 SDK is an invaluable tool for application development. There are, however, a number of situations where it is necessary to test an application on a physical iPhone device. In this chapter we have covered the steps involved in provisioning applications for installation on an iPhone device.

Chapter 8

8. The Basics of Objective-C Programming

In order to develop iOS apps for the iPhone 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 *iPhone iOS 5 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.

8.1 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';
```

8.2 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))
      System.Console.WriteLine("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
```

8.3 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");
}
```

8.4 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.

8.5 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 follows:

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

8.6 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'')
```

Chapter 9

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

Objective-C provides extensive support for developing object-oriented iOS iPhone 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.

9.1 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*.

9.2 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.

9.3 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.

9.4 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.

9.5 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 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.

9.6 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 %i 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.

9.7 Declaring and Initializing a Class Instance

So far all we have done is define the blueprint for our class. In order to do anything with this class, we need to create instances of it. The first step in this process is to declare a variable to store a pointer to the instance when it is created. We do this as follows:

```
BankAccount *account1;
```

Having created a variable to store a reference to the class instance, we can now allocate memory in preparation for initializing the class:

```
account1 = [BankAccount alloc];
```

In the above statement we are calling the *alloc* method of the BankAccount class (note that *alloc* is a *class method* inherited from the parent *NSObject* class, as opposed to an *instance method* created by us in the BankAccount class).

Having allocated memory for the class instance, the next step is to initialize the instance by calling the *init* instance method:

```
account1 = [account1 init];
```

For the sake of economy of typing, the above three statements are frequently rolled into a single line of code as follows:

```
BankAccount *account1 = [[BankAccount alloc] init];
```

9.8 Automatic Reference Counting (ARC)

In the first step of the previous section we allocated memory for the creation of the class instance. In releases of the iOS SDK prior to iOS 5, good programming convention would have dictated that memory allocated to a class instance be released when the instance is no longer required. Failure to do so, in fact, would have resulted in memory leaks with the result that the application would continue to use up system memory until it was terminated by the operating system. Those familiar with Java will be used to relying on the *garbage collector* to free up unused memory automatically. Historically, Objective-C has provided similar functionality on other platforms but not for iOS. That has now changed with the introduction of automatic reference counting in the iOS 5 SDK and it is not necessary to call the release method of an object when it is no longer used in an application.

Whilst the ARC avoids the necessity to call the release method of an object it is still, however, recommended that any *strong* outlet references be assigned *nil* in the *viewDidUnload* methods of your view controllers to improve memory usage efficiency. As a result, examples in this book will follow this convention where appropriate.

When creating a new project, Xcode now provides the option to implement automatic reference counting in the application code. If this option is selected, the code should not make calls to *release*, *retain*, *autorelease* or *dealloc* methods. Management of objects at this level is now handled for you by ARC.

9.9 Calling Methods and Accessing Instance Data

Given the length of this chapter, now is probably a good time to recap what we have done so far. We have now created a new class called *BankAccoun*t. Within this new class we declared some instance variables to contain the bank account number and current balance together with some instance methods used to set, get and display these values. In the preceding section we covered the steps necessary to create and initialize an instance of our new class. The next step is to learn how to call the instance methods we built into our class.

The syntax for invoking methods is to place the object pointer variable name and method to be called in square brackets ([]). For example, to call the *displayAccountInfo* method on the instance of the class we created previously we would use the following syntax:

```
[account1 displayAccountInfo];
```

When the method accepts a single argument, the method name is followed by a colon (:) followed by the value to be passed to the method. For example, to set the account number:

```
[account1 setAccountNumber: 34543212];
```

In the case of methods taking multiple arguments (as is the case with our *setAccount* method) syntax similar to the following is employed:

```
[account1 setAccount: 4543455 andBalance: 3010.10];
```

9.10 Creating the Program Section

The last stage in this exercise is to bring together all the components we have created so that we can actually see the concept working. The last section we need to look at is called the *program section*. This is where we write the code to create the class instance and call the instance methods. Most Objective-C programs have a *main()* routine which is the start point for the application. The following sample main routine creates an instance of our class and calls the methods we created:

```
int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool =
    [[NSAutoreleasePool alloc] init];
   // Create a variable to point to our class instance
   BankAccount *account1;
   // Allocate memory for class instance
   account1 = [BankAccount alloc];
   // Initialize the instance
   account1 = [account1 init];
   // Set the account balance
   [account1 setAccountBalance: 1500.53];
   // Set the account number
   [account1 setAccountNumber: 34543212];
   // Call the method to display the values of
```

```
   // the instance variables
   [account1 displayAccountInfo];
   // Set both account number and balance
   [account1 setAccount: 4543455 andBalance: 3010.10];
   // Output values using the getter methods
   NSLog(@"Number = %i, Balance = %f",
     [account1 getAccountNumber],
     [account1 getAccountBalance]);
   [pool drain];
   return 0;
}
```

9.11 **Bringing it all Together**

Our example is now complete so let's bring all the components together:

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

// Interface Section Starts Here

@interface BankAccount: NSObject
{
        double accountBalance;
        long accountNumber;
}
-(void) setAccount: (long) y andBalance: (double) x;
-(double) getAccountBalance;
-(long) getAccountNumber;
-(void) setAccountBalance: (double) x;
-(void) setAccountNumber: (long) y;
-(void) displayAccountInfo;
@end

// Implementation Section Starts Here

@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 %i has a balance of %f",
          accountNumber, accountBalance);
}
@end

// Program Section Starts Here

int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   BankAccount *account1;
   account1 = [BankAccount alloc];
   account1 = [account1 init];
   [account1 setAccountBalance: 1500.53];
   [account1 setAccountNumber: 34543212];
   [account1 displayAccountInfo];
   [account1 setAccount: 4543455 andBalance: 3010.10];
   NSLog(@"Number = %i, Balance = %f",
   [account1 getAccountNumber], [account1 getAccountBalance]);
   [pool drain];
   return 0;
}
```

When the above code is saved, compiled and executed we should expect to see the following output:

```
2009-10-14 14:44:06.634 t[4287:10b] Account Number 34543212 has a balance of
1500.530000
2009-10-14 14:44:06.635 t[4287:10b] Number = 4543455, Balance = 3010.100000
```

9.12 Structuring Object-Oriented Objective-C Code

Our example is currently contained within a single source file. In practice, the convention is to place the interface and implementation in their own include files that are then *included* in the program source file. Generally the interface section is contained within a file called *ClassName.h* where *ClassName* is the name of the class. In our case, we would create a file called *BankAccount.h* containing the following:

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

```
@interface BankAccount: NSObject
{
        double accountBalance;
        long accountNumber;
}
-(void) setAccount: (long) y andBalance: (double) x;
-(double) getAccountBalance;
-(long) getAccountNumber;
-(void) setAccountBalance: (double) x;
-(void) setAccountNumber: (long) y;
-(void) displayAccountInfo;
@end
```

Next, the implementation section goes in a file traditionally named *ClassName.m* where ClassName once again is refers to the name of the class. For example, *BankAccount.m* will contain the following (note that it is necessary to import the *BankAccount.h* file into this file):

```
#import "BankAccount.h"

@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 %i has a balance of %f",
         accountNumber, accountBalance);
}
@end
```

Finally, we will create our program file and call it *bank.m* (though any suitable name will do as long as it has a *.m* filename extension). This file also needs to import our interface file (*BankAccount.h*):

```
#import "BankAccount.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool =
       [[NSAutoreleasePool alloc] init];

    BankAccount *account1;
    account1 = [BankAccount alloc];
    account1 = [account1 init];
    [account1 setAccountBalance: 1500.53];
    [account1 setAccountNumber: 34543212];
    [account1 displayAccountInfo];
    [account1 setAccount: 4543455 andBalance: 3010.10];
    NSLog(@"Number = %i, Balance = %f",
      [account1 getAccountNumber], [account1 getAccountBalance]);
    [pool drain];
    return 0;
}
```

Chapter 10

10. An Overview of the iPhone iOS 5 Application Development Architecture

So far we have covered a considerable amount of ground intended to provide a sound foundation of knowledge on which to begin building iPhone iOS 5 based apps. Before plunging into writing your first app, however, it is vital that you have a basic understanding of some key methodologies associated with the overall architecture of an iOS application.

These methodologies, also referred to as *design patterns,* clearly define how your applications should be designed and implemented in terms of code structure. The patterns we will explore in this chapter are *Model View Controller (MVC)*, *Subclassing*, *Delegation* and *Target-Action*.

If you are new to these concepts this can seem a little confusing to begin with. Much of this will become clearer, however, once we start working on some examples in subsequent chapters.

10.1 Model View Controller (MVC)

In the days before object-oriented programming (and even for a time after object-oriented programming became popular) there was a tendency to develop applications where the code for the user interface was tied tightly to the code containing the application logic and data handling. This coupling made application code difficult to maintain and locked the application to a single user interface. If, for example, an application written for Microsoft Windows needed to be migrated to Mac OS, all the code written specifically for the Windows UI toolkits had to be ripped out from amongst the data and logic code and replaced with the Mac OS equivalent. If the application then needed to be turned into a web based solution, the process would have to be repeated again. Attempts to achieve this feat were usually found to be prohibitively expensive and ultimately ended up with the applications being completely re-written each time a new platform needed to be targeted.

The goal of the MVC design pattern is to divorce the logic and data handling code of an application from the presentation code. In this concept, the Model encapsulates the data for the application, the View presents and manages the user interface and the Controller provides the basic logic for the application and acts as the go-between, providing instructions to the Model based on user interactions with the View and updating the View to reflect responses from the Model. The true value of this approach is that the Model knows absolutely nothing about the presentation of the application. It just knows how to store and handle data and perform certain tasks when called upon by the Controller. Similarly, the View knows nothing about the data and logic model of the application.

Within the context of an object-oriented programming environment such as the iOS 5 SDK and Objective-C, the Model, View and Controller components are objects. It is also worth pointing out that applications are not restricted to a single model, view and controller. In fact, an app can consist of multiple view objects, controller objects and model objects.

The way that a view controller object interacts with a Model is through the methods and properties exposed by that model object. This, in fact, is no different from the way one object interacts with another in any object-oriented programming environment.

In terms of the view controller's interactions with the view, however, things get a little more complicated. In practice, this is achieved using the *Target-Action pattern*, together with *Outlets* and *Actions.*

10.2 The Target-Action pattern, IBOutlets and IBActions

When you create an iOS 5 iPhone app you will typically design the user interface (the view) using the Interface Builder tool and write the view controller and model code in Objective-C using the Xcode code editor. The previous section looked briefly at how the view controller interacts with the model. In this section we will look at how the view created in Interface Builder and our view controller code interact with each other.

When a user interacts with objects in the view, for example touching and releasing a button control, an *event* is triggered (in this case the event is called a *Touch Up Inside* event). The purpose of the *Target-Action* pattern is to allow you to specify what happens when such events are triggered. In other words, this is how you connect the objects in the user interface you have designed in the Interface Builder tool to the back end Objective-C code you have written in the Xcode environment. Specifically, this allows you to define which method of which controller object gets called when a user interacts in a certain way with a view object.

The process of wiring up a view object to call a specific method on a view controller object is achieved using something called an *Action.* An action is a method defined within a view controller object that is designed to be called when an event is triggered in a view object. This allows us to connect a view object created within a nib file using Interface Builder to the code that we have written in the view controller class. This is one of the ways that we bridge the separation between the *View* and the *Controller* in our MVC design pattern. As we will see in *Creating an Interactive iOS 5 iPhone App*, action methods are declared using the *IBAction* keyword.

The opposite of an *Action* is the *Outlet*. As previously described, an Action allows a view object to call a method on a controller object. An Outlet, on the other hand, allows a view controller object method to directly access the properties of a view object. A view controller might, for example, need to set the text on a UILabel object. In order to do so an Outlet must first have been defined using the *IBOutlet* keyword. In programming terms, an *IBOutlet* is simply an instance variable that references the view object to which access is required.

10.3 Subclassing

Subclassing is an important feature of any object-oriented programming environment and the iOS SDK is no exception to this rule. Subclassing allows us to create a new class by deriving from an existing class and then extending the functionality. In so doing we get all the functionality of the parent class combined with the ability to extend the new class with additional methods and properties.

Subclassing is typically used where a pre-existing class does most, but not all, of what you need. By subclassing we get all that existing functionality without having to duplicate it and simply add on the functionality that was missing.

We will see an example of subclassing in the context of iOS 5 development when we start to work with view controllers. The UIKit framework contains a class called the UIViewController. This is a generic view controller from which we will create a subclass so that we can add our own methods and properties.

10.4 Delegation

Delegation allows an object to pass the responsibility for performing one or more tasks on to another object. This allows the behavior of an object to be modified without having to go through the process of subclassing it.

A prime example of delegation can be seen in the case of the UIApplication class. The UIApplication class, of which every iOS iPhone application must have one (and only one) instance, is responsible for the control and operation of the application within the iOS environment. Much of what the UIApplication object does happens in the background. There are, however, instances where it gives us the opportunity to include our own functionality into the mix. UIApplication allows us to do this by delegating some methods to us. As an example, UIApplication delegates the *didFinishLaunchingWithOptions*: method to us so that we can write code to perform specific tasks when the app first loads (for example taking the user back to the point they were at when they last exited). If you still have a copy of the Hello World project created earlier in this book you will see the template for this method in the *HelloWorldAppDelegate.m* file.

10.5 Summary

In this chapter we have provided an overview of a number of design patterns and discussed the importance of these patterns in terms of structuring iOS 5 applications to run on the iPhone. Whilst these patterns may seem unclear to some, the relevance and implementation of such concepts will become clearer as we progress through the examples included in subsequent chapters of this book.