



Learn by doing: less theory, more results

Xcode 4

iOS Development

Use the powerful Xcode 4 suite of tools to build applications for the iPhone and iPad from scratch

Beginner's Guide

Steven F. Daniel

[PACKT]
PUBLISHING

Xcode 4 iOS Development

Beginner's Guide

Use the powerful Xcode 4 suite of tools to build applications for the iPhone and iPad from scratch

Steven F. Daniel



BIRMINGHAM - MUMBAI

Xcode 4 iOS Development

Beginner's Guide

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2011

Production Reference: 1160811

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-849691-30-7

www.packtpub.com

Cover Image by Tom Glasspool (t.glasspool@gmail.com)

Credits

Author

Steven F. Daniel

Project Coordinator

Leena Purkait

Reviewers

Cory Bohon

Mark Hazlett

Proofreader

Mario Cecere

Acquisition Editor

Steven Wilding

Graphics

Valentina D'silva

Geetanjali Sawant

Development Editor

Chris Rodrigues

Production Coordinator

Shantanu Zagade

Technical Editor

Dayan Hyames

Cover Work

Shantanu Zagade

Indexer

Monica Ajmera Mehta

About the Author

Steven F. Daniel is originally from London, England, but lives in Australia. He is an experienced software developer with more than 13 years of experience in developing desktop and web-based applications for a number of companies, in sectors including insurance, banking and finance, oil and gas, and local government. *Xcode 4 iOS Development Beginner's Guide* is his first book.

Steven is always interested in emerging technologies, and is a member of the SQL Server Special Interest Group (SQLSIG) and the Java Community. He is the owner and founder of **GenieSoft Studios** (<http://www.geniesoftstudios.com/>), a software development company based in Melbourne, Victoria, that currently develops games and business applications for the iOS, Android, and Windows platforms.

Steven was the co-founder and Chief Technology Officer (CTO) of SoftMpire Pty Ltd., a company that focused primarily on developing business applications for the iOS and Android platforms. You can check out his blog at <http://geniesoftstudios.com/blog/>, or follow him on Twitter at <http://twitter.com/GenieSoftStudio>.

This book is dedicated to:

Chan Ban Guan, for the patience, support, encouragement, and understanding all of those times when I couldn't go out as I needed to write in order to meet the deadlines.

My family for their continued love and support, and for always believing in me.

Chan Jie Hou, may God watch over you and keep you safe.

This book would not have been possible without your love and understanding.

Thank you from the bottom of my heart.

Acknowledgement

No book is the product of just the author—he just happens to be the one with his name on the cover.

A number of people contributed to the success of this book, and it would take more space than I have to thank each one individually.

A special shout out goes to Steven Wilding, my Acquisition Editor, who is the reason that this book exists. Thank you, Steven, for believing in me, and for being a wonderful guide through this process. I would also like to thank Leena Purkait for ensuring that I stayed on track and got my chapters in on time.

Thank you also to the entire Packt Publishing team for working so diligently to help bring out a high quality product.

To the engineers at Apple for creating the iPhone, and providing developers with the tools to create fun and sophisticated applications, you guys rock.

Finally, I'd like to thank all of my friends for their support, understanding, and encouragement during the writing process. It is a privilege to know each one of you.

About the Reviewers

Cory Bohon is a professional blogger and contributor to *MacLife* magazine, and a Mac and iPhone developer, experienced in Java, C/C++, Objective-C, and PHP. He is currently attending the University of South Carolina Upstate, where his current research interests include accessible user interface design and mobile application development.

Mark Hazlett is a mobile and web applications developer located in Calgary, Alberta, Canada. He has a true passion for mobile application development, especially on developing for the iPhone. In his spare time, Mark likes to read about new technologies, learn new languages, and start new projects.

He is constantly learning new technologies and applying them to as many personal projects as he can find time for. All in all, he is extremely passionate about usability and user interaction and tries to apply best practices to all of his projects.

I would like to thank my family: Tom, Jan, and Ryan for always being extremely supportive in my many endeavors. I would also like to thank Packt Publishing for giving me the opportunity to review this book.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ◆ Fully searchable across every book published by Packt
- ◆ Copy and paste, print and bookmark content
- ◆ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Introducing Xcode 4 Tools for iOS Development	7
Development using the Xcode tools	8
iPhone SDK core components	8
Inside Xcode, Cocoa, and Objective-C	9
The iPhone Simulator	10
Layers of the iOS architecture	11
The Core OS layer	11
The Core Services layer	13
The Media layer	14
The Cocoa-Touch layer	14
Understanding Cocoa, the language of the Mac	16
What are Design Patterns?	16
What is the difference between Cocoa and Cocoa-Touch?	16
The Model-View-Controller	16
What is Object-Oriented Programming?	17
What is Data Hiding?	17
What is Objective-C?	20
Directives	21
Objective-C classes	21
The @interface directive	21
The @implementation directive	22
Class instantiation	22
Class access privileges	22
Introducing the Xcode Developer set of tools	23
Introducing the core tools	23
The Welcome to Xcode screen	23
The Xcode Integrated Development Environment	24

Features of the iPhone Simulator	25
Companion tools and features	26
Instruments	26
iPhone OS4 SDK new features	28
Summary	29
Chapter 2: Introducing the Xcode 4 Workspace	31
Downloading and installing the iOS SDK	32
Removing the Xcode Developer Tools	35
Getting to know the Xcode Development Environment	35
One environment to bind them all	35
Working within a single-development environment	36
Creating a new project	37
Migrating older projects into the new environment	37
Writing a simple iPhone application	38
Time for action – creating your first iPhone application	38
Working with the new Xcode Assistant	46
Introducing the Xcode 4 Workspace Environment	48
Application ToolBar	48
Application Status Bar/Activity Window	49
WorkSpace Settings	49
Introducing the Unified Navigation UI	50
Listing files in a project	50
Sorted Symbols	51
Central Search Interface	52
Issues Tracking	53
Using Static Analysis to find potential problems	53
Debugging data with Compressionable Stack Traces	54
Active/inactive breakpoints	54
Collection of Logs	55
Jump Bar	55
Using Code Assistants	55
Introducing the new and improved LLVM Compiler 2.0	55
Version Editor	56
File Templates Library	56
Code Snippets Library	57
Object Library	58
Media Library	59
Resetting Xcode's Development Environment Settings	60
Xcode Workspace Preferences	60
General	61
Behaviors	61

Fonts & Colors	61
Text Editing	62
Key Bindings	62
Documentation	62
Locations	62
Source Trees	63
Distributed Builds	63
Summary	64
Chapter 3: Working with the Interface Builder	65
Getting to know the Interface Builder environment	66
Adding Controls to your user interface	67
Time for action – creating the HelloXcode4_GUI application	67
Application structure of our HelloXcode4 example application	70
The MainWindow.xib file	72
The Core Application Architecture layer	72
The application life cycle	73
Time for action – adding object controls to our View	74
Understanding Rotatable Interfaces	79
Time for Action – enabling Interface Rotation	79
Relocating controls within the view on Rotation	80
Making our Components work together	81
Time for action – binding Control Objects	82
Time for action – repositioning the Controls	84
Enhancing our iPhone application	87
Time for action – hiding the keyboard	87
Introducing Document-based applications	89
Time for action – creating a Document-based application	90
File saving and loading	94
Time for action – implementing file saving and loading	95
Summary	98
Chapter 4: Working with the Xcode Frameworks	99
Introducing the Frameworks	100
Using Frameworks and APIs in iPhone development	102
Core Data Frameworks	103
Building a simple database application	104
Time for action – creating the Core Data application	104
AV Foundation Frameworks	117
Playing an audio File	118
Creating an application to play an audio file	119

Time for action – creating the MediaPlayer application	119
Playing a movie using Media Player	125
Time for action – creating the MediaPlayer application	125
Core Location Framework	131
Time for action – making your application location aware	131
Map Kit Framework—new and improved	135
Time for action – creating a simple geographical application	136
New Framework APIs	140
Summary	142
Chapter 5: Designing Application Interfaces using MVC	143
Developing iOS applications using MVC design	144
Reusing tested (or standard) solutions: Design patterns	144
Understanding the Model-View-Controller design pattern	144
Implementing MVC using Xcode and Interface Builder	145
Time for action – building a Pizza order application	145
Time for action – binding our Controls using Outlets and Actions	147
Implementing views	152
Implementing view controllers	152
Time for action – declaring input field as a property of View Controller	154
Creating a view-based application template	154
Time for Action – creating the FavoriteColor application	155
Time for action – binding our Controls using Outlets and Actions	156
Implementing Table Views	159
Time for action – creating a Table view application	159
Grouping row items into sections	163
Time for action – grouping row items in our TableViewExample application	163
Understanding Navigation-based applications	168
Using Switches, Sliders, Segmented Controls, and Web Views	169
Time for action – creating the SwitchesSlidersSegments project	170
Time for action – binding our Controls using Outlets and Actions	173
Creating an application to scroll through large content	177
Time for action – creating the ScrollingViews project	177
Time for action – binding our Controls using Outlets and Actions	179
Understanding Pickers	181
Date Pickers	181
Time for action – creating the Date Picker project	182
Time for action – binding our Controls using Outlets and Actions	183
Custom Pickers	186
Time for Action – creating the Custom Picker project	186
Time for action – binding our Controls using Outlets and Actions	188

Handling basic user input and output	192
Button Controls	192
Text Fields	192
Text Views	192
Labels	193
Using Text Fields, Text Views, and Buttons	193
Time for action – creating application with Text fields, Text Views, and Buttons	193
Time for action – binding our Controls using Outlets and Actions	195
Summary	199
Chapter 6: Displaying Notification Messages	201
Exploring the notification methods	201
Generating alerts	202
Time for action – creating the GetUsersAttention application	202
Time for action – adding the AudioToolbox Framework to our application	203
Building our user interface	205
Time for action – adding controls to our View	205
Creating events	207
Time for action – implementing the Show Activity Indicator method	207
Time for action – implementing the Display Alert Dialog method	210
Responding to Alert Dialog Button presses	211
Using Action Sheets to associate with a view	214
Time for action – implementing the Display Action Sheet method	214
Responding to Action Sheet Button presses	215
Customizing an Action Sheet	217
Time for action – handling alerts via sounds and vibrations	217
Summary	221
Chapter 7: Exploring the MultiTouch Interface	223
Introducing the MultiTouch architecture	224
Detecting taps	226
Time for action – creating the TapExample project	226
Time for action – binding our Controls	228
Detecting swipes	231
Time for action – creating the SwipeExample project	232
Detecting pinches	236
Time for action – creating the PinchExample project	236
Detecting shakes	242
Time for action – creating the ShakeExample project	243
Time for action – implementing the motionBegan, motionEnded, and motionCancelled methods	245

Exploring the Accelerometer/Gyroscope	249
Understanding the Core Motion Framework	249
Sensing orientation	250
Time for action – creating the OrientationExample project	250
Detecting device tilting	254
Time for action – creating the AccelGyroExample project	254
Summary	260
Chapter 8: Debugging Xcode Projects	261
Introducing the new and improved Debugger	261
Debugger toolbar	262
Stack trace panel	263
Disassembly view	263
Code Editor window	264
Console output window	265
Creating a new debugging project	266
Time for action – creating the DebuggingExample project	266
Running and debugging the project	268
Handling errors	268
Runtime errors	269
Syntax errors	269
Logic errors	269
Using Fix-it to correct code as you type	270
Time for action – setting up the LLVM compiler	270
Debugging with breakpoints	272
Using NSLog to track changing properties	273
Exploring the new Debugger	275
Debugging features in the Code Editor	275
The Activity Viewer/Progress window	276
Defining a scheme for project builds using the Scheme Editor	276
Time for action – using the Scheme Editor to define a Scheme	277
Viewing the Static Analysis results	278
Time for action – running the Static Analyzer	279
Time for action – configuring your project to perform automatic Static Analysis	280
Time for action – Detecting a memory leak	281
Time for action – detecting an instance of an uninitialized variable	282
Viewing the Issues Navigator	284
Viewing the Program Build log	284
Understanding and using code completion	286
Time for action – working with code completion	286

Time for action – stopping Xcode from alerting you to problems	288
Navigating through threads and stacks in the Debugger	289
Summary	292
Chapter 9: Source Code Management with the Version Editor	293
Introducing the new Version Editor	294
Introducing Subversion	296
Installing a local Subversion server	296
Creating a repository	297
Time for action – setting up a local Subversion repository	298
Configuring the repository in Xcode	300
Time for action – configuring the Subversion repository	300
Adding items to an existing repository	303
Time for action – adding our TapExample project to the repository	303
Getting a working copy of the project out of the repository	305
Time for action – checking out the project from the repository	305
Xcode source-control features and file statuses	308
Comparing different versions of a file side-by-side	311
Using Timeline to select and compare revisions	312
Using Track Blame to check past check-ins	313
Using Log Mode to list all revisions chronologically	314
Using the Repository Organizer to keep track of your files	315
Using Git to manage multiple projects	317
Time for action – creating a new Xcode project using Git	318
Time for action – assigning address book identities within the organizer	319
Summary	323
Chapter 10: Making your Applications Run Smoothly	325
Introducing Instruments	326
Tracking down and fixing memory leaks	328
Time for action – creating the InstrumentsExample project	329
Time for action – running and Profiling the project	330
Adding and configuring Instruments	335
Using the Instruments Library	335
Locating an Instrument within the Library	336
Adding and removing Instruments	339
Configuring an Instrument	340
Other components of the Instruments family explained	342
New Instruments in Xcode 4	343
Automated Testing	343
Performance and Power Analysis	343
Time Profiler	344

Energy Diagnosis	344
Tracking iPhone graphics performance using OpenGL ES Driver	344
Summary	346
Chapter 11: Distributing your Application	347
Build configurations – debug to release	348
The iPhone Developer Program	348
Setting up your iPhone development team	349
Time for action – setting up the team	350
Getting an iOS development certificate	354
Time for action – generating a Certificate Request	354
Time for action – getting the certificate	357
Registering devices for testing	360
Time for action – registering devices	360
Creating application IDs	362
Time for action – creating the application ID	362
Creating a Provisioning Profile	365
Time for action – creating the profile	365
Using the Provisional Profile to install an App on an iOS device	368
Time for action – creating and deploying the app to an iOS device	368
Getting a Distribution Certificate for your app	373
Time for action – getting the Distribution Certificate	373
Archiving and submitting Apps using Xcode 4	375
iOS Human Interface Guidelines	377
Testing your application	377
Preparing your App for submission through iTunes Connect	378
Avoiding rejection of your App	381
Pricing your app	382
Adding your App to iTunes Connect	382
Time for action – uploading the application icon and screenshot images	383
Using iTunes Connect to manage your Apps	384
Marketing and promoting your app	386
iOS Developer Documentation	387
Summary	389

Appendix: Pop Quiz Answers	391
Chapter 3	391
Chapter 4	391
Chapter 5	391
Chapter 6	392
Chapter 7	392
Chapter 8	393
Chapter 9	393
Chapter 10	393
Chapter 11	393
Index	395

Preface

The iPhone is one of the hottest mobile devices on the planet. Whether you are just starting out with iPhone Development or already have some knowledge in this area, you will benefit from what this book covers. Using this book's straightforward, step-by-step approach, you will go from Xcode 4 apprentice to Xcode 4 Jedi master in no time.

Xcode 4 iOS Development Beginner's Guide will help you learn to build simple, yet powerful applications for the iPhone from the ground up. You will master the Xcode 4 tools and skills needed to create applications that are simple yet, like Yoda, punch far above their weight.

In this book, I have tried my level best to keep the code simple and easy to understand. I have provided step-by-step instructions with screenshots at each step to make it easier. You will soon be mastering the technology and skills needed to create some stunning applications. Feel free to contact me at geniesoftstudios@gmail.com for any queries. Any suggestions for improving this book will be highly appreciated.

What this book covers

Chapter 1, Introducing Xcode 4 Tools for iOS Development, introduces the developer to the Xcode developer set of tools, the new features of the iOS 4 SDK and the iOS Architecture Layers and their components. It also includes a discussion of Cocoa, Cocoa-Touch, and the basics of object-oriented programming using Objective-C.

Chapter 2, Introducing the Xcode 4 Workspace, discusses how to download and install the Xcode 4 and iOS4 SDK and introduces you to the Xcode 4 development environment and the different types of libraries that are part of the workspace to create a simple iPhone application.

Chapter 3, Working with the Interface Builder, introduces the developer to the Interface Builder application and explains the iOS application life cycle when an application is run. It also covers how to implement file saving and loading of Document-based applications, as well as how to reposition the controls within the view when the device is rotated.

Chapter 4, Working with the Xcode Frameworks, introduces the developer to the different types of Xcode frameworks for audio and video playback, and Core Location services for determining geographical locations. It also covers how to build a simple database application using the Core Data Framework.

Chapter 5, Designing Application Interfaces using MVC, introduces the developer to the various layers of MVC and design patterns and the importance of implementing these in iOS applications. It also covers how to interact with the user, with lots of code examples.

Chapter 6, Displaying Notification Messages, explores the different notification methods through which we can communicate with the user to grab their attention, by using alerts, activity indicators, sounds, and vibrations, with lots of code examples.

Chapter 7, Exploring the MultiTouch Interface, shows you how easy it is to incorporate both single-touch and multi-touch support into your applications and include support for tapping, pinching, and swipes. You will also learn about the built-in shake gesture and how to go about responding to the shake motions, before finally learning about the accelerometer and the new gyroscope features, as well as how to control your application UI when the orientation changes.

Chapter 8, Debugging Xcode Projects, shows us how to go about debugging our projects, through the use of the various debugging tools that Xcode provides. We are also introduced to the new debugging features of the editor, and how to use the Static Analyzer tool to determine potential memory leaks, dead code, and unreachable code, as well as using the new Fix-it! feature to correct syntax errors as we type.

Chapter 9, Source Code Management with the Version Editor, focuses on the new features of the Xcode Version Editor that has been integrated directly within the Xcode 4 IDE and provides you with an easy way to manage your source code. By using this tool, you are able to travel back through your revisions to compare previous changes made throughout the life cycle of the file.

Chapter 10, Making your Applications Run Smoothly, focuses on how we can effectively use Instruments within our applications to track down memory leaks and bottlenecks within our applications that could potentially cause our application to crash on the user's iOS device. We take a look into each of the different types of built-in instruments, which come as part of the Instruments application and how we can use the Leaks instrument to help track down and determine where memory leaks are happening within our code. We also look at how we can configure instruments to display data differently within the trace document that is being reported.

Chapter 11, Distributing your Application, provides you with the necessary steps that are required to submit your applications to the App Store. It explains how to register devices for testing and how to create and obtain provisioning profiles for development and distribution.

What you need for this book

This book assumes that you have an Intel-based Macintosh running Snow Leopard (Mac OS X 10.6.2 or later). You can use Leopard, but I would highly recommend upgrading to Snow Leopard, as there are many new features in Xcode that are available only on Snow Leopard. We will be using Xcode, an integrated development environment used for creating applications for the iPad, iPhone, and other Mac applications. You can download the latest version of Xcode at the following link: <http://developer.apple.com/xcode/>.

Who this book is for

If you ever wanted to learn how to build iOS applications and make your mark within the iOS industry and have your applications compete with the rest, this book is for you. You should have some basic programming experience with Objective-C, and a good understanding of OOP, as well as some knowledge of database design.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " If you observe the content of the `MyClass.h` file, you will notice that at the top of the file is a `#import` statement."

A block of code is set as follows:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    [window makeKeyAndVisible];
    return YES;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

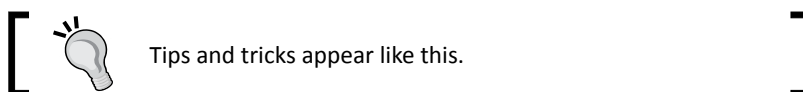
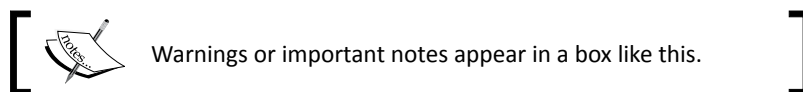
```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Override point for customization after application launch.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
    return YES;
}
```

Any command-line input or output is written as follows:

```
defaults delete com.apple.Xcode
rm -rf ~/Library/Application\ Support/Xcode
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Next** button to proceed to the next step of the wizard."



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing Xcode 4 Tools for iOS Development

Welcome to the exciting world of iPhone Programming using Xcode 4. Since the release of the original iPhone back in 2007, it has taken the world by storm and opened up a whole new world to developers. This unique device comprises a multi-touch interface, video and audio playback capabilities, stunning graphics and sound, map and localization services, always-on internet and Wi-Fi services, and a whole range of built-in sensors which can be used to create everything from stunning games to business applications.

You are probably eager to get stuck right in and start creating that next big thing to hit the AppStore, and start to join those other tens of thousands of developers. The goal of this chapter is to give you enough insight into Xcode and its components, the framework layers of the iOS architecture, and some basics of Objective-C.

By the end of this book, you will have a firm grasp of Cocoa-Touch, and understand most of the design patterns used by the Objective-C frameworks in order to go on and build some fantastic applications. These could be along the lines of a real estate application that shows a listing of all properties within the surrounding suburbs using an SQLite database to store its data, and core location services to display the property on the map, with directions on how to get there. You could even create an educational game using flash cards and sounds. The possibilities are endless; all it takes is an idea and you.

In this chapter, we will:

- ◆ Learn about the features and components of the Xcode development tools
- ◆ Learn about Xcode, Cocoa, Cocoa-Touch, and Objective-C
- ◆ Take a look into each of the iOS technology layers and their components
- ◆ Take a look into what comprises the Xcode developer set of tools
- ◆ Take a look at the new features in the iOS4 SDK

There is a lot of fun stuff to cover, so let's get started.



Throughout this chapter and the rest of this book, I will be making references to iOS. Apple is calling its new Operating System "iOS 4", and it is the next generation of the world's most innovative operating system and works for the **iPhone**, **iPad**, and **iPod Touch**.

Development using the Xcode tools

In order to develop for the iPhone or any other iOS device, you will need to download and install the Xcode developer tools. You can find these development tools on the CDs that come with your Mac. However, it is always best to download the latest version of the developer tools from the Apple website at <http://developer.apple.com/>.

Before you proceed to download these tools, Apple requires you to register as an iOS developer at <http://developer.apple.com/programs/register/>. The registration is free and provides you with access to the iOS SDK (software development kit) and other resources that are useful for getting started.

iPhone SDK core components

The iPhone SDK includes a suite of development tools to assist you with development of your iPhone, and other iOS device applications. We describe these in the table below:

COMPONENT	DESCRIPTION
Xcode	This is the main Integrated Development Environment (IDE) that enables you to manage, edit, and debug your projects.
DashCode	This enables you to develop web-based iPhone and iPad applications, and Dashboard widgets.
iPhone Simulator	The iPhone Simulator is a Cocoa-based application that provides a software simulator to simulate an iPhone or iPad on your Mac OS X.

COMPONENT	DESCRIPTION
Interface Builder	This is the graphical visual editor for designing your user interfaces for your iPhone and iPad applications. In previous releases of Xcode, this was a separate standalone application. In Xcode 4, this has now been integrated as part of the development IDE.
Instruments	These are the Analysis tools that help you optimize your applications and monitor for memory leaks in real-time.

The Xcode tools require an Intel-based Mac running Mac OS X version 10.6 or later in order to function correctly.

Inside Xcode, Cocoa, and Objective-C

Xcode 4 is a complete toolset for building Mac OS X (Cocoa-Based) and iOS applications. The new single-windowed development interface has been redesigned to be a lot easier and even more helpful to use than it has been in previous releases. It can now also identify mistakes in both syntax and logical errors, and will even fix your code for you.

It provides you with the tools to enable you to speed up your development process, therefore becoming more productive. It also automates deployment of both your Mac OS X and iOS applications.

The Integrated Development Environment (IDE) allows you to do the following:

- ◆ Create and manage projects, including specifying platforms, target requirements, dependencies, and build configurations
- ◆ Supports syntax colouring and automatic indenting of code
- ◆ Enables you to navigate and search through the components of a project, including header files and documentation
- ◆ Enables you to Build and Run your project
- ◆ Enables you to debug your project locally, run within the iOS simulator, or remotely, within a graphical source-level debugger

Xcode incorporates many new features and improvements, apart from the redesigned user interface; it features a new and improved **LLVM (Low Level Virtual Machine)** compiler that has been supercharged to run three times faster and 2.5 times more efficiently.

This new compiler is the next generation compiler technology designed for high-performance projects and completely supports C, Objective-C, and now C++. It is also incorporated into the Xcode IDE and compiles twice as fast and quickly as GCC and your applications will run faster.

The list below includes the many improvements made to this release:

- ◆ The interface has been completely redesigned and features a single-window integrated development interface.
- ◆ Interface Builder has now been fully integrated within the Xcode development IDE.
- ◆ Code Assistant opens in a second window that shows you the file that you are working on, and can automatically find and open the corresponding header file(s).
- ◆ Fix-it checks the syntax of your code and validates symbol names as you type. It will even highlight any errors that it finds and will fix them for you.
- ◆ The new Version Editor works with GIT (*Open-Source*) version control software or Subversion. This will show you the file's entire SCM (software configuration management) history and will even compare any two versions of the file.
- ◆ The LLDB debugger has now been improved to be even faster and it uses less memory than the GDB debugging engine.
- ◆ The new Xcode 4 development IDE now lets you work on several interdependent projects within the same window. It automatically determines its dependencies so that it builds the projects in the right order.

Xcode allows you to create a number of build configurations to test your iOS applications, for debugging using the Static Analyzer, or profiling using the Instruments application. Xcode also allows you to archive your application in order to submit to the Apple App Store for review.

It supports several source-code management tools, namely, CVS "*Version control software which is an important component of the Source Configuration Management (SCM)*" and Subversion that allows you to add files to a repository, commit changes, get updated versions, and compare versions using the Version Editor tool.

The iPhone Simulator

The iPhone Simulator is a very useful tool that enables you to test your applications without using your actual device, whether this is your iPhone or any other iOS device. You do not need to launch this application manually, as this is done when you Build and Run your application within the Xcode Integrated Development Environment (IDE). Xcode installs your application on the iPhone Simulator for you automatically.

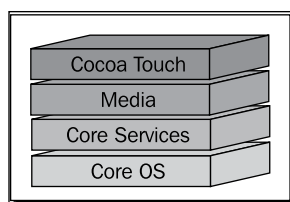
The iPhone Simulator also has the capability of simulating different versions of the iPhone OS, and this can become extremely useful if your application needs to be installed on different iOS platforms, as well as testing and debugging errors reported in your application when run under different versions of the iOS.



While the iPhone Simulator acts as a good test bed for your applications, it is recommended to test your application on the actual device, rather than relying on the iPhone Simulator for testing. This is because the speed of the iPhone Simulator relies on the performance of your Mac instead of the actual device. The iPhone Simulator can be found at the following location: `/Developer/Platforms/iPhoneSimulator.Platform/Developer/Applications`.

Layers of the iOS architecture

Apple describes the set of frameworks and technologies that are currently implemented within the iOS operating system as a series of layers. Each of these layers is made up of a variety of different frameworks that can be used and incorporated into your applications:



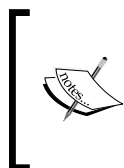
We will now go into detail and explain each of the different layers of the iOS Architecture; this will give you a better understanding of what is covered within each of the Core layers.

The Core OS layer

This is the bottom layer of the hierarchy and is responsible for the foundation of the operating system which the other layers sit on top of. This important layer is in charge of managing memory—allocating and releasing memory once the application has finished with it, taking care of file system tasks, handling networking, and other operating system tasks. It also interacts directly with the hardware.

The Core OS layer consists of the following components:

COMPONENT NAME	DESCRIPTION
OS X Kernel	Based on Mach 3.0, it is responsible for every aspect of the operating system.
Mach 3.0	A subset of the OS X Kernel responsible for running applications within a separate process.
BSD (Berkeley Standard Distribution)	Based on the kernel environment within the Mac OS X it is responsible for the drivers, and low-level UNIX interfaces of the operating system.
Sockets	Part of the CFNetwork Framework for providing access to BSD sockets, HTTP and FTP protocol requests.
Security	The Security Framework provides functions for performing cryptographic functions (encrypting/decrypting data). This includes interacting with the iPhone keychain to add, delete, and modify items.
Power Management	Conserves power by shutting down any hardware features that are not currently being used.
Keychain	Part of the Security Framework for handling and securing data.
Certificates	Part of the Security Framework for handling and securing data.
File System	The System Framework gives developers access to a subset of the typical tools they would find in an unrestricted UNIX development environment.
Bonjour	Part of the CFNetwork Framework for providing access to BSD sockets, HTTP and FTP protocol requests, and Bonjour discovery over a local-area-network.



For more information on the iOS Core OS layer, please refer to the Apple Developer Connection website at http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40007898-CH11-SW1

The Core Services layer

The Core Services layer provides an abstraction over the services provided in the Core OS layer. It provides fundamental access to the iPhone OS services. The Core Services Layer consists of the following components:

COMPONENT NAME	DESCRIPTION
Collections	Part of the Core Foundation Framework which provides basic data management and service features for iOS applications.
Address Book	Provides access to the user's Address Book contacts on the iOS device.
Networking	This is part of the System Configuration Framework, which determines network availability and state on an iOS device.
File Access	Provides access to lower-level operating system services.
SQLite	This lets you embed a lightweight SQL database into your application without running a separate remote database server process.
Core Location	Used for determining the location and orientation of an iOS device.
Net Services	Part of the System Configuration to determine whether a Wi-Fi or cellular connection is in use and whether a particular host server can be accessed.
Threading	Part of the Core Foundation Framework which provides basic data management and service features for iOS applications.
Preferences	Part of the Core Foundation Framework which provides basic data management and service features for iOS applications.
URL Utilities	Part of the Core Foundation Framework which provides basic data management and service features for iOS applications.

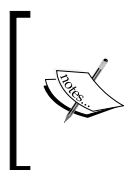


For more information on the iOS Core Services layer, please refer to the Apple Developer Connection website at http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40007898-CH10-SW5

The Media layer

The Media layer provides multimedia services that you can use within your iPhone, and other iOS devices. The Media layer is made up of the following components:

COMPONENT NAME	DESCRIPTION
Core Audio	Handles playback and recording of audio files and streams and also provides access to the device's built-in audio processing units.
OpenGL	Used for creating 2D and 3D animations
Audio Mixing	Part of the Core Audio Framework, provides the possibility to mix system announcements with background audio. For example, iOS would announce callerID while fading in/out the background media.
Audio Recording	Provides the ability to record sound on the iPhone using the AVAudioRecorder class.
Video Playback	Provides the ability to playback Video using the MPMoviePlayerController class.
Image Formats: JPG, PNG, and TIFF	Provides interfaces for reading and writing most image formats – part of the Image I/O Framework.
PDF	Provides a sophisticated text layout and rendering engine.
Quartz	Framework for image and video processing, and animation using the Core Animation technology.
Core Animations	Provides advanced support for animating views and other content. This is part of the Quartz Framework.
OpenGL ES	This is a subset of the OpenGL Framework for creating 2D and 3D animations.



For more information on the iOS Media layer, please refer to the Apple Developer Connection website at http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40007898-CH9-SW4

The Cocoa-Touch layer

The Cocoa-Touch layer provides an abstraction layer to expose the various libraries for programming the iPhone, and other iOS devices. **You probably can understand why** Cocoa-Touch is located at the top of the hierarchy due to its support for Multi-Touch capabilities. The Cocoa-Touch layer is made up of the following components:

COMPONENT NAME	DESCRIPTION
Multi-Touch Events	These are the events which are used to determine when a Tap, Swipe, Pinch, double-tap has happened. That is, TouchesMoved, TouchesBegan, TouchesEnded.
Multi-Touch Controls	Based on the Multi-Touch model, this determines when a user has placed one or more fingers touching the screen before responding to the action accordingly.
View Hierarchy	Deals with the Model-View-Controller and the objects within the view.
Alerts	Using the UIAlertView class, these are used to communicate to the user when an error happens, or to request further input.
People Picker	Based on the AddressBook Framework, which displays the person's contact details.
Controllers	Based on the Model-View-Controller for presenting standard system interfaces and provides much of the logic needed to manage basic application behaviors. For example, managing the reorientation of views in response to device orientation changes.
Accelerometer/Gyroscope	Responds to motion and measures the degree of acceleration, and rate of rotation around a particular axis.
Localization/Geographical	Adds maps and satellite images to location-based apps, similar to the one provided by the Maps application.
Web Views	Provides a view to embed web content and display rich HTML.
Image Picker	Provides a potentially multi-dimensional user-interface element consisting of rows and components.



For more information on the iOS Cocoa-Touch Layer, please refer to the Apple Developer Connection website at http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1

Understanding Cocoa, the language of the Mac

Cocoa is defined as the development framework used for the development of most native Mac OS X applications. A good example of a Cocoa-related application is Mail or Text Edit.

This framework consists of a collection of shared object code libraries known as the Cocoa frameworks. It consists of a runtime system and a development environment. This set of frameworks provides you with a consistent and optimized set of pre-built code modules that will speed up your development process.

Cocoa provides you with a rich layer of functionality, as well as a comprehensive object-oriented like structure and APIs on which you can build your applications. Cocoa uses the **Model-View-Controller (MVC)** design pattern.

What are Design Patterns?

Design Patterns represent and handle specific solutions to problems that arise when developing software within a particular context. These can be either a description or a template, on how to solve a problem in a variety of different situations.

What is the difference between Cocoa and Cocoa-Touch?

Cocoa-Touch is the programming language framework that drives user interaction on iOS. It consists of and uses technology derived from the Cocoa framework and was redesigned to handle multi-touch capabilities. The power of the iPhone and its user interface are available to developers throughout the Cocoa-Touch frameworks.

Cocoa-Touch is built upon the Model-View-Controller structure; it provides a solid stable foundation for creating mind blowing applications. Using the Interface Builder developer tool, developers will find it both very easy and fun to use the new drag-and-drop method when designing their next great masterpiece application on iOS.

The Model-View-Controller

We will just touch on this subject briefly as this will be covered in *Chapter 5, Designing Application Interfaces using MVC*. Basically, the Model-View-Controller (MVC) comprises of a logical way of dividing up the code that makes up the **GUI (Graphical User Interface)** of an application. Object-Oriented applications like Java and .Net have adopted the MVC design pattern.

The MVC model comprises of three distinctive categories:

- ◆ **Model:** This part defines your application's underlying data engine. It is responsible for maintaining the integrity of that data.
- ◆ **View:** This part defines the user interface for your application and has no explicit knowledge of the origin of data displayed in that interface. It is **made up of** Windows, controls, and other elements that the user can see and interact with.
- ◆ **Controller:** This part acts as a bridge between the model and view and facilitates updates between them. It **binds the Model and View together and the application** logic decides how to handle the user's inputs.

What is Object-Oriented Programming?

Object-Oriented programming (OOP), provides an abstraction layer of the data on which you operate. It provides a concrete foundation between the data and the operations that you perform with the data, in effect, giving the data behavior.

By using the power of Object-Oriented programming, we can create classes and later extend its characteristics to incorporate additional functionality. Objects within a class can be protected to prevent those elements from being exposed; this is called "Data Hiding".

What is Data Hiding?

Data Hiding is an aspect of Object-Oriented Programming (OOP) that allows developers to protect private data and hide implementation details by encapsulating this within a class.

In the following Java code example, we take a look at how we implement hiding of variables, and only exposing those methods to extract this information:

```
public class Dog {  
    // public - a violation of data hiding 'rules'  
    public string name;  
  
    // private - not visible outside the class  
    Private float age;  
    .  
    .  
    .  
}
```




Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

In the partial `Dog` class snippet above, we show how we are able to expose and hide variables within a class. In the following example below, we look at how we go about using the Getters and Setters to store and retrieve information, without directly assigning values to the variables:

```
public class Dog {
    private String name;
    private String gender;
    private String breed;
    private float weight;
    private float height;
    private float age;

    // Here are 'get' and 'set' methods for the private variable
    declared above.
    public void setName(String name){
        this.name = name;
    }
    public String getName() { return name;}
    public void setGender (String gender){
        this.gender = gender;
    }
    public String getGender() { return gender;}
    public void setBreed (String breed){
        this.breed = breed;
    }
    public String getBreed() { return breed;}
    public void setHeight(float height ) {
        If (height >= 0) { this.height = height;}
    }
    public float getHeight() { return height;}
    public void setWeight(float weight){
        if (weight >= 0) { this.weight = weight;}
    }
    public float getWeight() { return weight;}
    public float getAge() { return age;}
    public void setAge(float age){
        if (age >= 0) { this.age = age;}
    }
}
```

In the above code snippet, we have declared a number of different variable data types as well as the getters and setters to retrieve and assign values to these variables. Generally speaking, all variables declared within a class should be made private; since these types are not made visible outside the class. The only methods which should be made visible to a class must be made public and therefore supplemented with the "get" and "set" methods. In the following code snippet, we look at how we use the getters and setters to store and retrieve the data:

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Dog myDog = new Dog();

        // Use the 'set' method to assign each property.
        myDog.setName("Frodo");
        myDog.setGender("Male");
        myDog.setBreed("Pug");
        myDog.setWeight((float)9.0);
        myDog.setHeight((float)36.0);
        myDog.setAge((float)10.0);

        // Output each of the values using the Getters.
        System.out.println("Name    = " + myDog.getName());
        System.out.println("Gender = " + myDog.getGender());
        System.out.println("Breed  = " + myDog.getBreed());
        System.out.println("Weight = " + myDog.getWeight());
        System.out.println("Height = " + myDog.getHeight());
        System.out.println("Age    = " + myDog.getAge());
    }
}
```

In the above code snippet, we declare a new instance of our `Dog` class and then use the `Setters` method to add values to each of our private method variables, before finally displaying this information out to the screen via the `Getters` methods.



If you are interested in learning more about Object-Oriented Programming, please consult the Apple Developer documentation or via the following link: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooOOP.html

What is Objective-C?

When I first started to develop for the iPhone, I realised that I needed to learn Objective-C, as this is the development language for Mac and iOS. I found it to be one of the strangest looking languages I had ever come across. Today, I really enjoy developing and working with it, so will you too.

Objective-C is an object-oriented programming language used by Apple primarily for programming Mac OS X, iPhone, and other iOS applications. It is an extension of the C Programming Language. If you have not done any OOP programming before, I would seriously recommend that you read the OOP document from the Apple Developer website.

On the other hand, if you have used and are familiar with C++, .Net or Java, learning Objective-C should be relatively easy for you to understand.

Objective-C consists of two types of files:

- ◆ .h: These types of files are called 'Header' or 'Interface files'
- ◆ .m: These types of files are those which contain your program code logic and make use of the 'Header' files. These are also referred to as 'implementation' files.

Most Object-Oriented development environments consist of the following parts:

- ◆ An Object-Oriented programming language
- ◆ An extensive library consisting of objects
- ◆ A development suite of developer tools
- ◆ A runtime environment

For example, here is a piece of code written in Objective-C:

```
-(int)method:(int)i {  
    return [self square_root: i];  
}
```

Now, let's examine the code line by line to understand what is happening.

We declare a function called `method` and a variable `i`, which is passed in as a parameter. We then pass the value to a function called `square_root` to calculate the value and the calculated result is returned.

If we were to compare this same code to how it would be written within C, it would look like this:

```
int function(int i) {  
    return square_root(i);  
}
```

Directives

In C/C++, we use directives to include any other header files that our application will need to access. This is done by using **#include**. In Objective-C, we use the **#import** directive. If you observe the content of the `MyClass.h` file, you will notice that at the top of the file is a `#import` statement:

```
#import <UIKit/UIKit.h>
@interface MyClass : NSObject{
}
@end
```

The `#import` statement is known as a "*pre-processor directive*." As I mentioned previously, in C/C++, you use the `#include` pre-processor directive to include a file's content with the current source file. In Objective-C, you use the `#import` statement to do the same, with the exception that the compiler ensures that the file is only included once.

To import a header file from one of the framework libraries, you would specify the header filename using the angle brackets (< >), within the `#import` statement. If you wanted to import one of your own header files to be used within your project, you would specify and make use of the double quote marks (" "), as you can see from our code file, `MyClass.m`:

```
#import "MyClass.h"
@implementation MyClass
@end
```

Objective-C classes

A **Class** can simply be defined as a representation of a type of object; think of it as a blueprint that describes the object. Just as a single blueprint can be used to build multiple versions of a car engine, a class can be used to create multiple copies of an object. In Objective-C, you will spend most of your time dealing with classes and class objects. An example of a class object is the `NSObject` class. `NSObject` is the root class of most of the Objective-C classes. It defines the basic interface of a class and contains methods that are common to all classes that inherit from it.

The @interface directive

To declare a class, you use the **@interface** compiler directive that is declared within `MyClass.h`, as follows:

```
@interface MyClass : NSObject {
}
```

The @implementation directive

To implement a class declared within a header file, you use the **@implementation** compiler directive, as follows:

```
#import "MyClass.h"
@implementation MyClass
@end
```

Class instantiation

In Objective-C, in order for us to create an instance of a class, you would typically use the **alloc** keyword to allocate memory for the object and then return the variable in a class type. This is shown in the following example:

```
MyClass *myClass = [MyClass alloc];
```

If you are familiar with other languages such as Java or C#.Net, instantiating a class can be done as follows:

```
MyClass myClass = new MyClass();
```

Class access privileges

In OOP, when you are defining your classes, bear in mind that by default, the access privilege of all fields within a class are **@protected**. These fields can also be defined as **@public**, or **@private**.

The following table below shows the various access privileges that a class can contain:

ACCESS PRIVILEGE	DESCRIPTION
@private	A class member is only visible to the class that declares it.
@public	A class member is made visible to all classes that instantiate this class.
@protected	Class members are made visible to the class that declares it as well as other classes which inherit from the base class.



We have only covered a small part of the Objective-C programming concepts. If you are interested in reading a bit more about this area, please refer to the following website: <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>

Introducing the Xcode Developer set of tools

The Xcode developer set of tools comprise of the Xcode Integrated Development Environment (IDE), Interface Builder, iPhone Simulator, and Instruments for Performance Analysis. These tools have been designed to integrate and work harmoniously together.

Introducing the core tools

The **Xcode IDE** is a complete full-featured development environment, which has been redesigned and built around to allow for a better smoother workflow development environment. With the integration of the GUI designer (Interface Builder), it allows a better way to integrate the editing of source code, building, compiling, and debugging.

The **Interface Builder** is an easy to use GUI designer which enables you to design every aspect of your applications UI, for Mac OS X and iOS applications.

All of your form objects are stored within one or more resource files, these files contain the associated relationships to each of the objects. Any changes that you make to the form design are automatically synchronized back to your code.

The **iPhone Simulator** provides you with a means of testing your application out, and to see how it will appear on the actual device. The Simulator makes it a perfect choice to ensure that your user interface works and behaves the way you intended it to and makes it easier for you to debug your application. The iPhone Simulator does contain some limitations, which cannot be used to test certain features, so it is always better to deploy your app to your iOS device.

The Welcome to Xcode screen

To launch Xcode, double-click on the Xcode icon located in the `/Developer/Applications` folder. Alternatively, you can use **Spotlight** to search for this: simply type **Xcode** into the search box and Xcode should be displayed in the list at the top.

When Xcode is launched, you should see the **Welcome to Xcode** screen as shown in the screenshot below. From this screen, you are able to create new projects, check out existing projects from the SCM, and modify those files within the Xcode integrated development environment. It also contains some information about learning Xcode as well as Apple Developer resources.

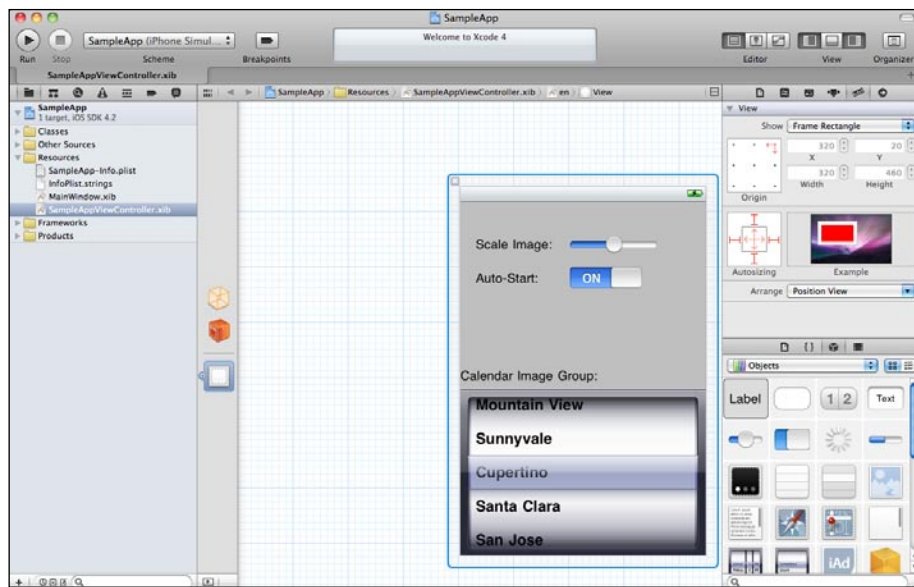
The panel to the right-hand side of the screen will display any recent projects which you have opened. These can be opened and loaded into the IDE by clicking on them:



The Xcode Integrated Development Environment

The Xcode Integrated Development Environment is what you will be using to start to code your iPhone applications.

This consists of a single-window user interface, consisting of the Project Window, Jump and Navigation Bars, and the newly integrated Interface Builder designer. We will talk more about the Xcode 4 workspace environment in *Chapter 2, Introducing the Xcode 4 Workspace*:



Features of the iPhone Simulator

The "*iPhone Simulator*" simulates various features of a real iOS device.

The screenshot below displays the iPhone 4 simulator:



Below we list some of the features which you are able to test using the iPhone Simulator:

TYPE	DESCRIPTION
Screen Rotation:	Left, Top, and Right
Support for gestures:	Tap
	Touch and Hold
	Double Tap
	Swipe

TYPE	DESCRIPTION
Low-Memory warning simulations	Flick
	Drag
	Pinch
	Notifies all running applications whenever the amount of free memory falls below a safe threshold. This can happen when memory is allocated to objects, but never released.

Although the iPhone simulator is just a simulator to simulate certain tasks, it does come with the following limitations:

- ◆ Making phone calls
- ◆ Accessing the Accelerometer/Gyroscope
- ◆ Sending and receiving SMS messages.
- ◆ Installing applications from the App Store
- ◆ Accessibility to the camera
- ◆ Use of the microphone
- ◆ Several core OpenGL ES features

Companion tools and features

These tools are classified as profiling tools which are the instruments that handle the following:

- ◆ Performance and Power Analysis tools
- ◆ Unit testing tools
- ◆ Source Code Management (SCM)/Subversion
- ◆ Version Comparison tool

Instruments

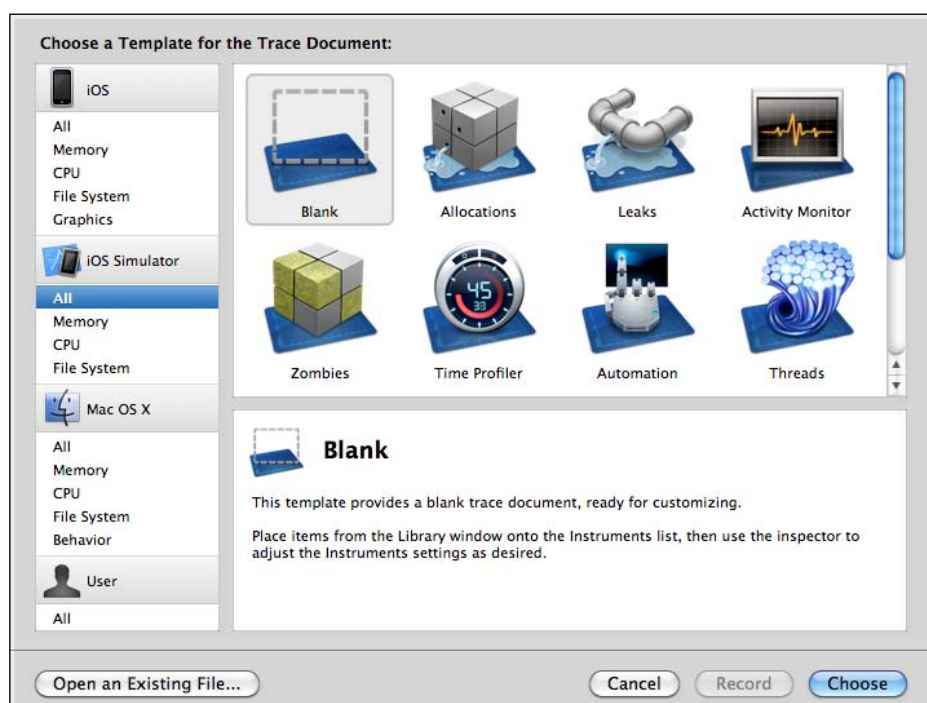
The Xcode instruments allow you to dynamically trace and profile the performance of your Mac OS X, iPhone, and iPad applications. You can also create your own Instruments using **DTrace** and the Instruments custom builder.



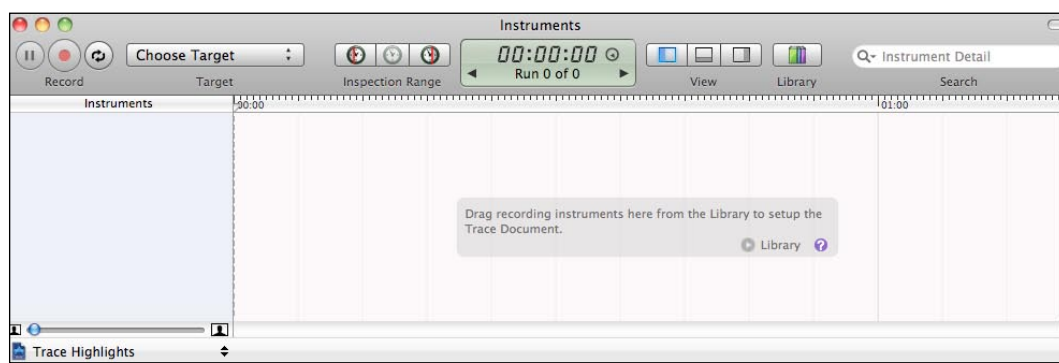
We do not cover DTrace in this book, if you are interested in reading a bit more about this area; please consult the Apple Developer Documentation at the following: http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/dtrace.1.html#//apple_ref/doc/man/1/dtrace. DTrace has not been ported to iOS, so it is not possible to create a custom instrument for devices running iOS.

Through the use of instruments, you can achieve the following:

- ◆ Ability to perform stress-tests on your applications
- ◆ Monitor your applications for memory leaks, which can cause unexpected results
- ◆ Gain a deeper understanding of the execution behaviour of your applications
- ◆ Track down difficult-to-reproduce problems in your applications
- ◆ In the screenshot displayed below, it shows you the current list of available instrument templates which you can choose from, to perform a variety of different traces on your iOS applications. We will be discussing and using these in greater detail, when we come to *Chapter 10, Making your Applications run smoothly*.



- ◆ In the following screenshot, we display the Instruments environment where you can start to create your robust test harness for your application to ensure that any memory leaks and resource-intensive tasks are rectified to avoid problems later when your users download your app and experience issues:



If you are interested in learning more about the other applications that are included with Xcode and the iOS 4 SDK, please consult the Apple Developer documentation at the following: <http://developer.apple.com/>

iPhone OS4 SDK new features

The iOS4 SDK comes with over 1,500 APIs that contain some high quality enhancements and improvements which allow endless possibilities for developers to create some stunning applications. The table below provides an overview of these features:

FEATURE	DESCRIPTION
Multi-Tasking	This is perhaps the most awaited feature. It is an assortment of seven different services: Audio, VoIP, location, local and push notifications, task completions, and fast app switching that will make it possible and simple enough to use many applications at the same time. This will allow you to play the audio continuously, receive calls while your device is locked or other apps are being used. Location-based applications will continue to guide you, and receiving alerts will be possible without the app running and the app will finish even when the customer leaves in the middle of it.

FEATURE	DESCRIPTION
Apps Folder	Another important feature in iOS4 is the Apps Folder. This feature allows you to drag an icon on top of another one and a new folder will be automatically created. It will be named according to the name of the category the particular icon or application comes from.
Game Center	Game Center provides social networking services, where you can take part in leader boards and participate in other online activities with other players.
iAd	iAd is a mobile advertising platform to allow developers to incorporate advertisements into their application. It is currently supported on iPhone, iPod Touch, and iPad.



The list above contains some of the important new features of iOS 4. If you are interested in a more detailed listing of all of the features in each of the releases; check out the following: http://en.wikipedia.org/wiki/iPhone_OS_Version_History.

Summary

In this chapter, hopefully you have gained a good understanding of Xcode and the development tools and the new and improved single-windowed development IDE. We have also covered some of the basics relating to Object-Oriented Programming and Objective-C. It will soon become apparent why Objective-C was chosen as the language of choice for developing Mac OS X and iOS applications.

Now that we've learned about what comprises the Xcode developer tools, we are now ready to get stuck into and learn about the new Xcode 4 workspace and development environment.

In the next chapter, we will dive right in and take a closer look at the Xcode 4 workspace environment as well as using some of the tools we have explained in this chapter. We will also start to develop a simple application using Xcode and Interface Builder.

2

Introducing the Xcode 4 Workspace

In the previous chapter, we covered the Xcode development tools. We also took a look into Xcode, the Cocoa Framework, and Cocoa-Touch layers and delved into a crash-course on the features of Objective-C. Finally, we looked into the new features of iOS 4.

*In this chapter, we will learn how to download and install the Xcode developer tools and **Software Development Kit (SDK)**. We will familiarise ourselves with the newly re-designed development environment, and introduce you to the Xcode 4 workspace and preferences as well as working with the new Xcode Assistant and Code Assistants.*

We will finally create a simple iPhone application, that will incorporate the use of Views and View Controllers.

In this chapter, we will:

- ◆ Learn how to download and install the Xcode Development Tools.
- ◆ Introduce you to the new Xcode Development Environment.
- ◆ Introduce you to the Xcode 4 Workspace and the Unified Navigator UI
- ◆ Learn about the Xcode Workspace Preferences
- ◆ Create a simple "Hello World" iPhone application using Views and View Controllers
- ◆ Learn how to reset the Xcode development environment

We have got quite a bit to cover, so let's get started.

Downloading and installing the iOS SDK

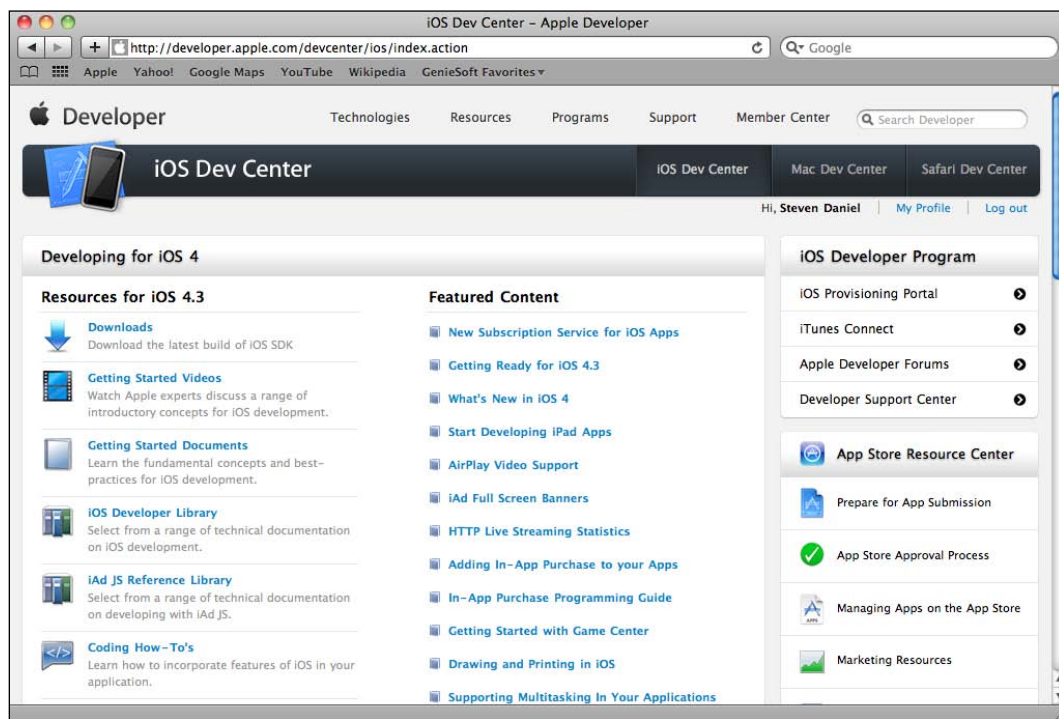
Before we can start to build our iOS applications, you must first sign up as a registered iOS Developer at <http://developer.apple.com/programs/ios/>. The registration process is free and provides you with access to the iOS SDK (Software Development Kit) and other developer resources that are really useful for getting you started.

Once you have signed up, you can then download the iPhone SDK as shown in the screenshot below. It is worthwhile making sure that your machine satisfies the following system requirements prior to your downloading the iPhone SDK:

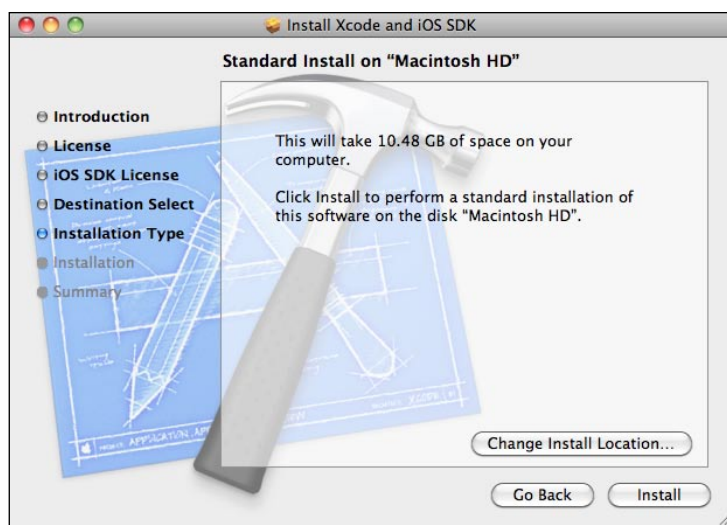
- ◆ Only Intel Macs are supported, so if you have **another processor type** (such as the older G4 or G5 Macs), you're out of luck.
- ◆ You have updated your system with the latest Mac OS X release



If you want to develop applications for the iPad and iPod Touch, this uses the same operating system (OS) as the iPhone, so you can still use the iPhone SDK. This SDK allows you to create universal applications that will work with both the iPhone and iPad running on iOS 4.2 and above.

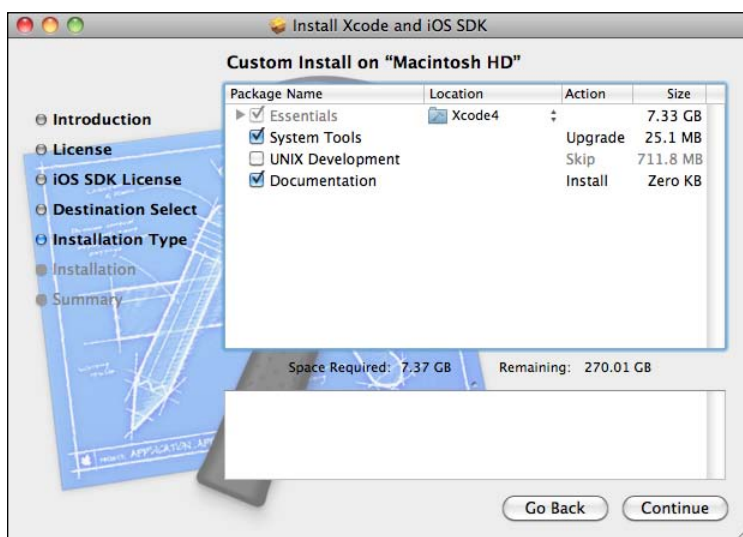


Once you have downloaded the SDK, you can proceed with installing it. You will be required to accept a few licensing agreements. You will then be presented with a screen to specify the destination folder in which to install the SDK:



If you select the default settings during the installation phase, the various tools will be installed in the `/Xcode4/Applications` folder.

The installation process takes you through the custom installation option screens. You probably would have seen similar screens to this if you have installed other Mac software. The screenshot below shows what you will see here:



These options give you a little more control over the installation process. For example, you are able to specify the folder location to install Xcode as well as setting a variety of other options.

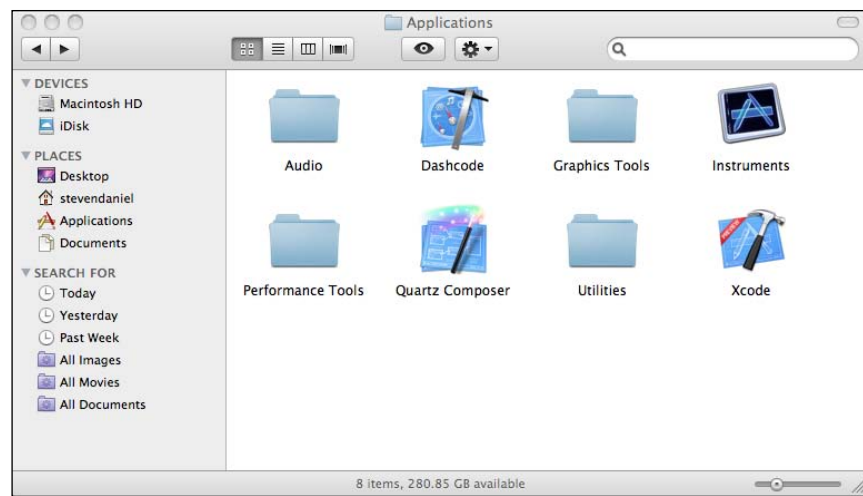


By default, Xcode 4 will be installed on the root directory of your Hard Drive in the following folder location: /Xcode4/Applications.

The iOS4 SDK comes as part of the "**xcode_4.0.1_and_ios_sdk_4.3.dmg**" download which we installed in the previous section. You are also able to download this separately from the download page. The SDK consists of the following components which are mentioned in the list below and provides a reference to the relevant area to find out more information:

- ◆ Xcode: Please refer to *Chapter 1, iPhone SDK Core Components* for more information on this component.
- ◆ DashCode: Please refer to *Chapter 1, iPhone SDK Core Components* for more information on this component
- ◆ iPhone Simulator: Please refer to *Chapter 1, iPhone SDK Core Components* for more information on this component
- ◆ Interface Builder: Please refer to *Chapter 1, iPhone SDK Core Components* for more information on this component
- ◆ Instruments: Please refer to *Chapter 1, iPhone SDK Core Components* for more information on this component

The following image displays a list of the various tools that are installed as part of the default settings during the installation phase. These are installed in the /Xcode4/Applications folder:



Removing the Xcode Developer Tools

Should you ever wish to uninstall **Xcode** (in the event that something went disastrously wrong) it is a very straightforward process. Open the terminal window and run the **uninstall-devtools** script:

```
sudo <Xcode>/Library/uninstall-devtools --mode=all
```

<Xcode> is the directory where the tools are installed. For typical installations, the full path is `/Xcode4/Library/uninstall-devtools`



Before you proceed to do this, make sure that this is what you really intend to do as once it's gone, it's permanently deleted. In any event, you can always choose to reinstall the Xcode developer tools. It is also worth checking that the `/Xcode4/Library/` folder has also been removed. If not, just move it to the Trash.

Getting to know the Xcode Development Environment

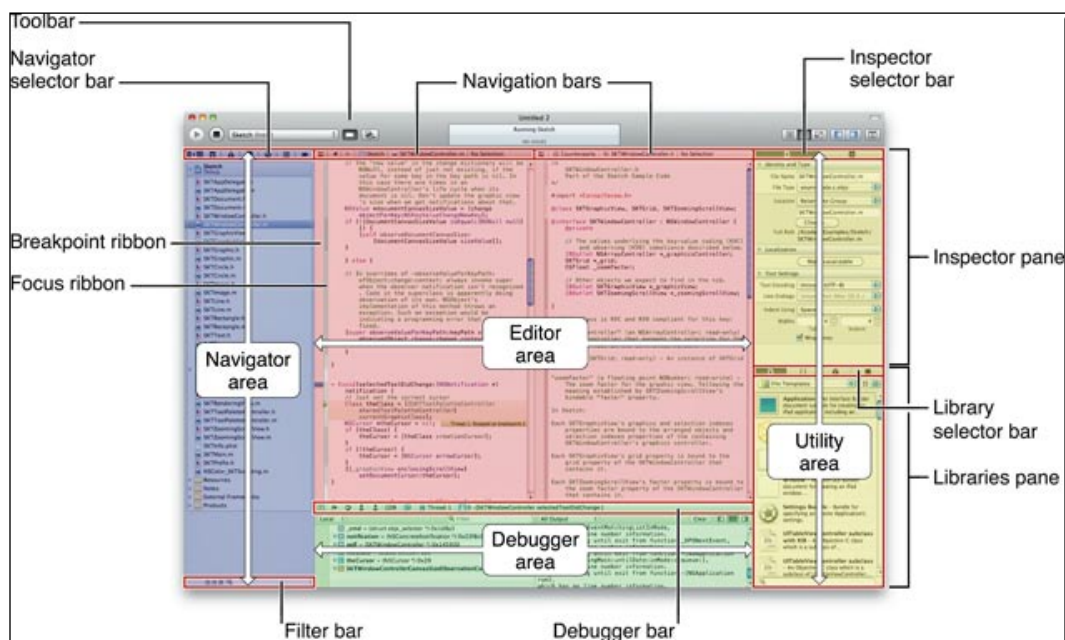
I am hoping that up to this point, you would have successfully downloaded and installed the Xcode 4 developer tools and iOS SDK. From this point on, we will start to familiarise ourselves with the newly redesigned Xcode 4 development workspace and what is involved in migrating older projects, as well as creating a new project from scratch.

One environment to bind them all

The first thing that you will notice when you fire up the Xcode 4 IDE is that the interface has gone through some significant changes that provide better workflow to enable developers to be more productive. The new interface features a LCD-like display, similar to that found in iTunes. This is called the Activity Window and is used for displaying warnings and compiler error messages during the build process of a project.

Working within a single-development environment

The main area of Xcode (called the **workspace**) is dedicated to the document that is currently being viewed, whether it is code, a data model, or the project's graphical interface. This area can also be arranged to view multiple documents, to allow a comparison of their differences (such as comparing two versions of the same code file). The content area also has support for viewing of PDFs and other file types supported by the extendable Quick Look feature:



One other new thing that you will notice is that, above the content area is what is called the 'Jump Bar' that presents a hierarchical "breadcrumb-like" listing, which was introduced in iTunes and Finder; with its only difference being that it is fully interactive; users can click on any path along the bar and select a popup that allows them to navigate at that level via a dynamic popup window that is displayed.

It also contains a Project Navigator control. This contains icon tabs that present a variety of different types of development-related information and are contained within the same window, and these are as follows:

- ◆ Provides a means of listing all of your projects and files, that are contained within your workspace
- ◆ A Project Symbol listing which lists all classes and methods used by your project

- ◆ A new and improved search feature, which can perform project-wide searching
- ◆ Issues listing containing all of the build errors that your project contains
- ◆ Helpful debugging information
- ◆ An area, that shows you all **breakpoints, which are flagged in your project**
- ◆ Contains a section, which lists all of the Build logs

Creating a new project

To create a new project in Xcode 4, launch the Xcode 4 development IDE and then click on **Create a new Xcode project** in the **Welcome to Xcode** startup screen. This is shown in the screenshot below. If you already have Xcode 4 opened, choose **File | New | New Project**:



Migrating older projects into the new environment

Xcode 4 provides you with a number of quick and easy ways to migrate your older Xcode projects into the new development environment. You can choose to right-click on your project and select to open the project in Xcode 4, or you can similarly drag your project onto the Xcode 4 icon.

You can also use the **File | Open** command within the Xcode 4 IDE. Xcode 4 can read and build projects built in Xcode 3.2 through to 3.2.6. Opening your project(s) in Xcode 4 does not upgrade or alter it in any way and any changes that you make to the project will remain compatible with earlier versions of Xcode.

Writing a simple iPhone application

This is where the real fun starts, and I know that you are eager to get stuck in and start creating a simple iPhone application. We will be creating a simple *Hello to Xcode 4* application making use of View Controllers. We won't be using Xcode's Interface builder in this chapter, as it will be covered in *Chapter 3, Working with the Interface Builder*. So let's get started.

Time for action – creating your first iPhone application

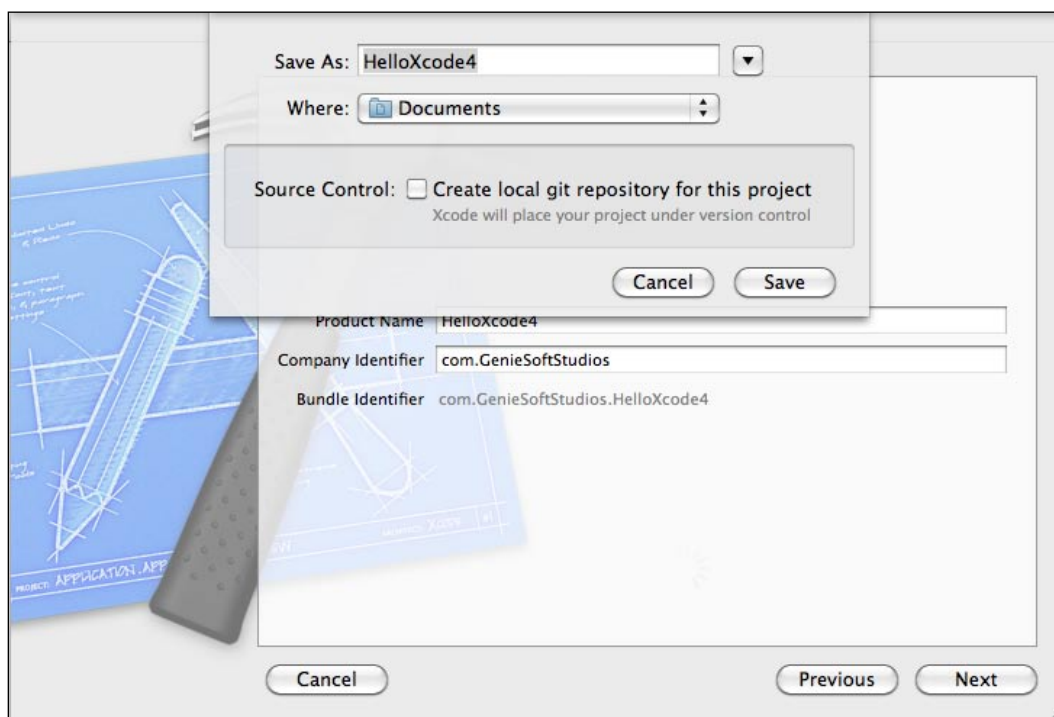
In this section, we will look at building a simple iPhone application to display a welcome message to our View.

Before we can proceed with creating our "HelloXcode4" application, we must first launch the **Xcode** development environment. If you haven't already got this open, it can be located within the `/Xcode4/Applications` folder, or you can use the spotlight feature to help you search for this. What we now need to do is to create our project by following these simple steps below:

1. Click on the **Create a new Xcode project**. This will bring up the project template dialog as shown in the screenshot below.
2. Select the **Window-based application**. What this template is going to give you is a Window and an application delegate. An application delegate (**UIApplicationDelegate**) is an object that responds to messages from a **UIApplication** object. It is worth mentioning here that there can only be one **UIApplication** object, and the project template takes care of creating this for us.
3. Click on the **Next** button and you will be prompted to enter a name for your project.
4. Enter **HelloXcode4** for the Product Name. Don't worry about the Company Identifier as this will be populated for you by default. This is used for submission of your App to the AppStore.
5. Click on the **Next** button to proceed to the next step of the wizard.
6. You will then be asked to choose a location where you would like to save your project. This can be any location that you desire.

7. You will also notice that there is an option to automatically save your project to Source Control, which will create a local repository on disk so you can check-in your work. This option is not checked by default. It is advisable to have this checked, if you are working in a team environment where you have multiple people working on the same project.

Xcode 4 provides support for both Git and SVN for Version Control. We will cover this in greater detail when we come to *Chapter 9, Source Code Management with the Version Editor*.



Once your project has been created, you will be presented with the Xcode interface. All files that the project template created for you will be displayed within the Project Navigator window section:

The important files to take note of are the following: `main.m`, `HelloXcode4AppDelegate.h`, and `HelloXcode4AppDelegate.m`.

The main function is where the single `UIApplication` object is created and the function called `UIApplicationMain()` takes care of that. You may be asking yourself, how does the `HelloXcode4AppDelegate` get hooked up to the `UIApplication` object? Remember when we created our project earlier on, the creation process created an Interface Builder (`.xib`) file for us called `MainWindow.xib`; this is the file that takes care of forming this relationship for us.

Now, we will check out the implementation of the delegate, `HelloXcode4Delegate.m`.

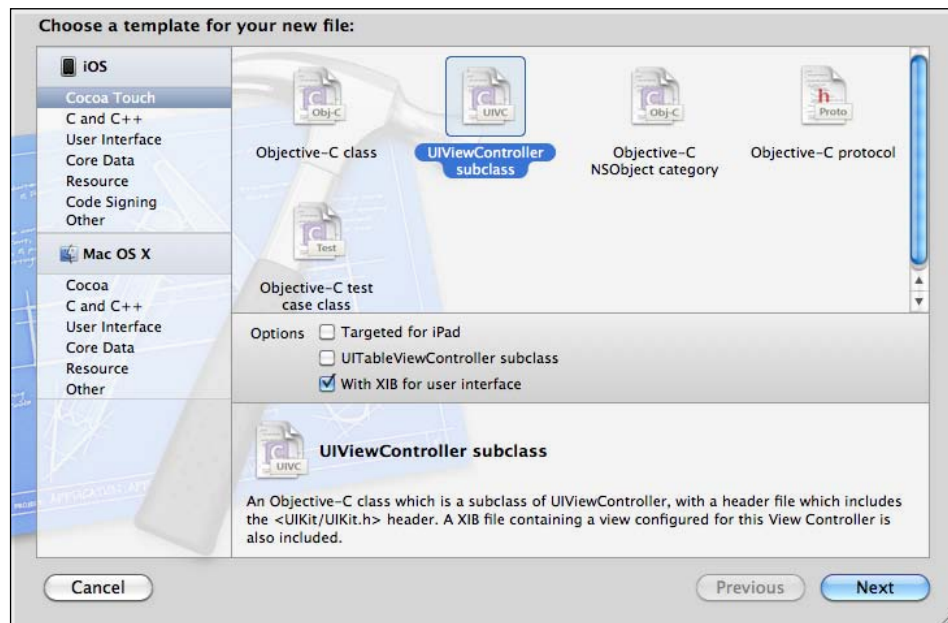
There are several messages that we can get from the `UIApplication` object.

However, the template has already created the one which we care about: — `applicationDidFinishLaunchingWithOptions:`

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    [window makeKeyAndVisible];
    return YES;
}
```

This function is where we will be creating our view controller, which will eventually hold our label object for our text output. In order to create a view controller, we first need to add another class to our project that subclasses `UIViewController`:

1. Select the **Classes** folder, located within your project and then choose **File | New | New File...**, or **Command + N**:



2. Next, select the **Cocoa Touch Class** in the left column under **iOS** and select the **UIViewController subclass** template from the list of available templates.
3. Click on the **Next** button to proceed to the next step of the wizard.
4. Enter **HelloXcode4ViewController** as the name of the file to create, and then click on the **Save** button.

You will notice when you look at the new implementation file `HelloXcode4ViewController.m` that it contains some commented out functions as shown in the following code snippet:

```
@implementation HelloXcode4ViewController

/*
// The designated initializer. Override if you create the
controller programmatically and want to perform customization that
is not appropriate for viewDidLoad.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle
*)nibBundleOrNil {
    if ((self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil])) {
        // Custom initialization
    }
    return self;
}
*/

/*
// Implement viewDidLoad to do additional setup after loading the
view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/

/*
// Override to allow orientations other than the default portrait
orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/
```


View controllers are meant to be used by overriding the base implementation of various methods. In our case, we want to override the `loadView` method, which is used to manually populate a view controller:

```
// Implement loadView to create a view hierarchy programmatically,  
// without using a nib file.  
- (void) loadView
```

We don't need to add controls directly to the view controller as these are added to a `UIView`, that is a property of the view controller. We have not allocated the view yet, so we are going to start to do this here.

First, we need to create a `UIView` object that is the size of our display and link this up to the view controllers `view` property:

```
// Create a frame that sets the bounds of the view  
CGRect frame = CGRectMake(0, 0, 960, 640);  
  
// Allocate our view  
self.view = [[UIView alloc] initWithFrame:frame];  
  
// Set the view's background color  
self.view.backgroundColor=[UIColor greenColor];
```

View Controllers have a `view` property that needs to be set to our new `UIView` object. Next, we will be creating a label to store our text output:

```
// set the position of our text label  
frame = CGRectMake(10, 170, 350, 50);  
  
// allocate memory for our label  
UILabel *label = [[UILabel alloc] initWithFrame:frame];  
  
// Assign some text to our label control  
label.text = @"iPhone Programming using Xcode 4";  
label.textColor = [UIColor redColor];  
  
// now, add the label to the view  
[self.view addSubview:label];  
  
// it is a good idea to release the memory allocated by our label  
[label release];  
}
```

Now that we have created our view controller, which contains a label that will display the words **iPhone Programming using Xcode 4**, we need to create an instance of our view controller and add it to our application. If you recall when we were discussing the function `applicationDidFinishLaunching`, this is where we will be adding our view controller.

Before we can do this, we first need to add a `#import` statement at the top of our `HelloXcode4AppDelegate.m` implementation file, in order for the compiler to find the declaration of this object. This is shown below:

```
#import "HelloXcode4AppDelegate.h"
#import "HelloXcode4ViewController.h"
```

Now that you have done this, we are ready to start creating our View Controller code within this file (`HelloXcode4AppDelegate.m`):

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

    // allocate the memory for our view controller
    self.viewController = [HelloXcode4ViewController alloc];

    // now, we need to add our view controller to the window
    [window addSubview:self.viewController.view];

    // Override point for customization after application launch.
    [window makeKeyAndVisible];
    return YES;
}
```

Just as we have done in the similar examples above, we needed to allocate a new `HelloXcode4ViewController` and then add that to our window. We could just create our view controller locally, but this is not a good idea as it can cause memory leaks.

A much better approach is to create a reference, which can be released once we have finished using it. I have created a property called `viewController`, which will be used to store our view controller.

This has been defined within the `HelloXcode4AppDelegate.h` file and is shown in the following code snippet:

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

@class HelloXcode4ViewController;

@interface HelloXcode4AppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;

    HelloXcode4ViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) HelloXcode4ViewController
*viewController;

@end
```

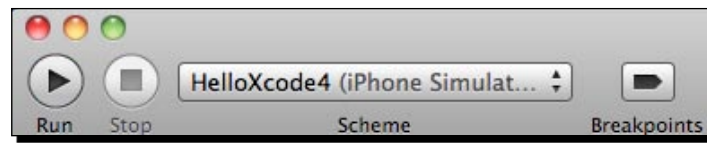
We now need to tell the compiler to actually create the getter and setter functions for our new property. We need to do this back in our `HelloXcode4AppDelegate.m` implementation file by adding the **@synthesize** property. This is shown below:

```
@synthesize window;
@synthesize viewController;
```

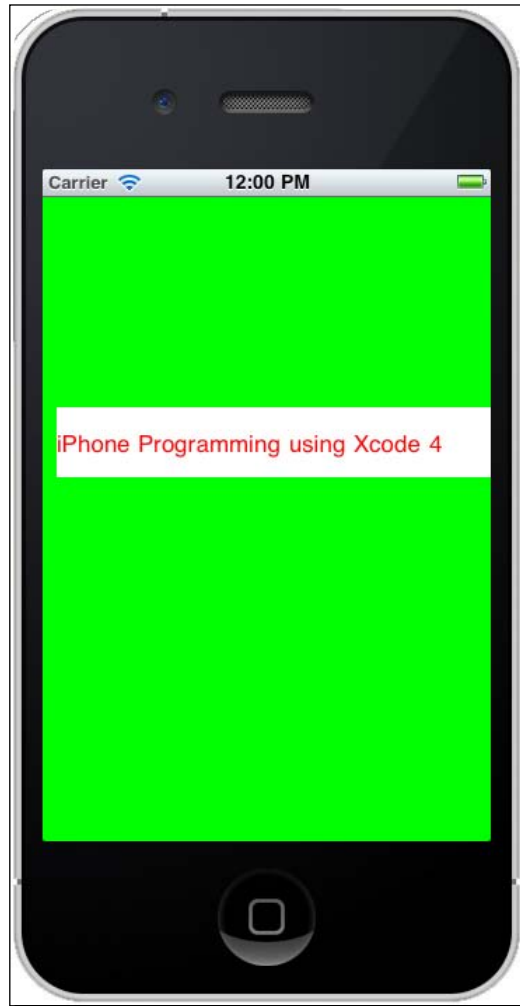
You will see that one already exists for our window object, so just place this underneath. Finally we need to release our reference to our view controller property when the **dealloc** method is called. You should notice that a `dealloc` function already exists within the `HelloXcode4AppDelegate.m` file; we just need to modify it to include removal of our view controller object:

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

So there you have it, we have added our view controller to our view and have released the memory when the device is stopped. You are now ready to Build and Run your application. You can do this in one of two ways, either click on the play button within the Xcode 4 IDE, or click on **Product | Run "HelloXcode4"**:



When you run the application, you should see an image similar to the screenshot below:



What just happened

We first created a `CGRect` object, which sets the bounds for our view. We then positioned our view to start from top (0, 0) and set it to the size of the iPhone display of (960, 640) as well as setting our view background to green.

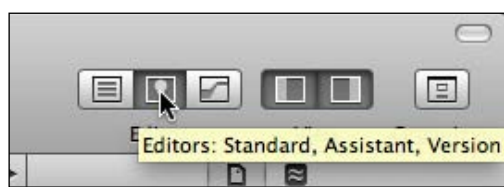
Next, we needed to specify where we would like to position our label text. In order to do this, we needed to create another `CGRect` object and then allocate memory for our label object and then add the label to our view controller.

Finally, we released the memory used by our label object (this is good programming practice and will prevent your application from having memory leaks).

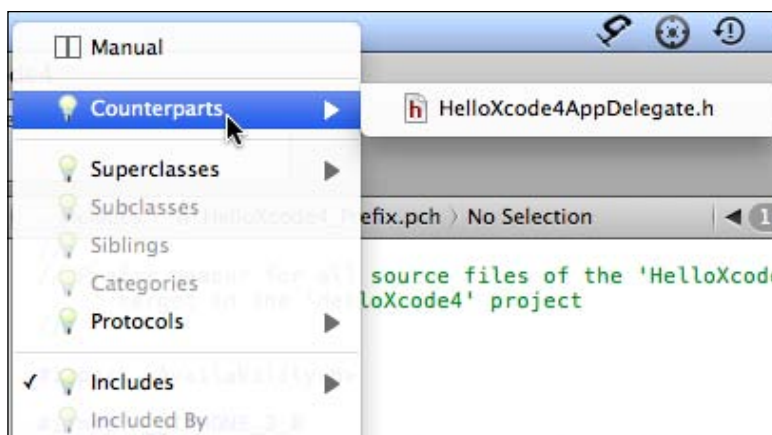
In the `HelloXcode4AppDelegate.h` file, you will notice that we have declared an **@class** declaration at the top of the interface (`HelloXcode4AppDelegate.h`) file. This `HelloXcode4ViewController` class is a forward declaration and it tells the compiler that a class name with this exists within our project. Without this declaration, the compiler would report an error at the first reference to `HelloXcode4ViewController`. In the **@interface** section of this file, we declare a member variable to hold our view controller. We then use the **@property** method to wrap our member variable with implicit getters and setters.

Working with the new Xcode Assistant

A new improved feature, included within Xcode 4, is the Xcode Assistant. This assistant comes with two modes of tracking: Automatic and Manual mode. Automatic tracking mode comes with several criteria, which you can choose from. These are: **Counterparts**, **Superclasses**, **Subclasses**, or **Siblings**. The Assistant selects the file/s that best meet the selected criteria and opens those files in the Assistant Pane within the source editor. In Manual Mode, you select the file to display in the Assistant Pane. You can also use the Version Editor to compare any two versions of the file. To enable the Assistant, click on the **Assistant** button in the workspace editor:

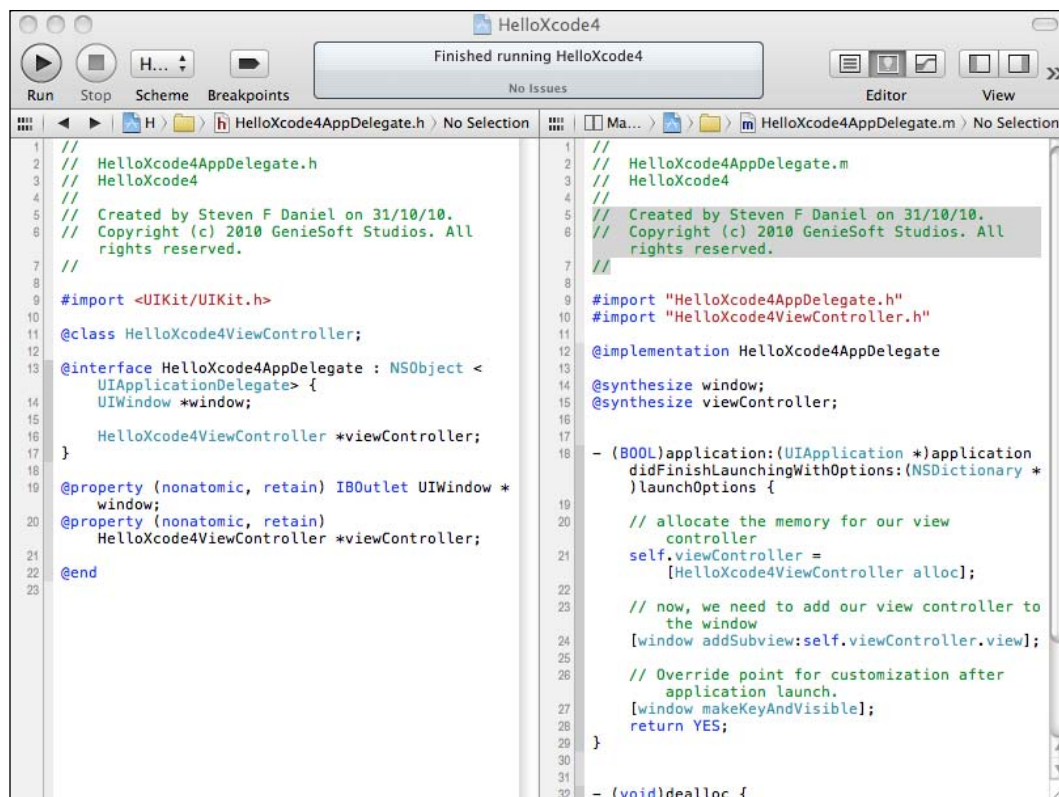


When this button is selected, the Assistant opens a second editor pane in the main editor area of the workspace window. For source files, the assistant displays the counterpart of the file, which is displayed in the standard editor pane. For instance, if you have opened an implementation file, the assistant will display the corresponding header interface file and vice-versa. You also have the ability to choose from several other display criteria to be used by the assistant. To do this, click on the Counterparts item in the navigation bar for the assistant editor pane. This is shown in the following screenshot, and the choices, which are offered to you, are dependent on the type of file that you are viewing:



Selecting a file in the project navigator will cause that file to be shown in the normal editor pane and automatically track and show the associated file counterpart in the other pane. If the assistant is set to use Manual mode, it will allow you to open any file in the assistant editor pane, including the same file as the one you're working on in the other editor pane. If you hold down the *Option* key and click or just hold down the *Option* key when selecting the file in the project manager, it will display the chosen file in the Assistant pane, rather than the standard pane and therefore switches the Assistant into manual mode.

The Assistant editor pane can be displayed to the right of the standard editor or below it. If you wanted to do this, you would Choose **View | Editor | Change Split Orientation** to switch between a vertical and horizontal split for the two editor panes:



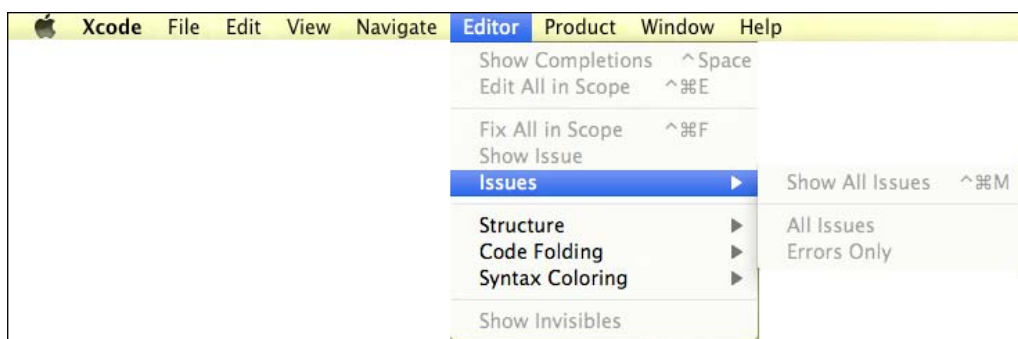
Introducing the Xcode 4 Workspace Environment

Xcode 4 uses one type of main window, called the workspace window, to hold most of the data you need. You can have as many workspace windows open as you need. A second window, called the Organizer window, is used for organizing your projects and reading documentation. For iOS projects, the Organizer window is also used for managing devices.

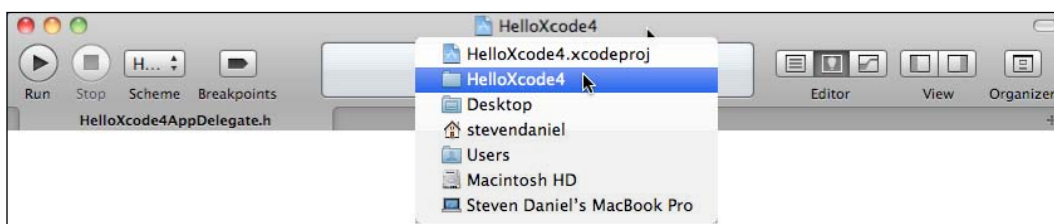
Application ToolBar

The Xcode toolbar provides you with quick access shortcuts to various common functions, that are used within the IDE environment. Some examples are code folding, version editor, and code assistants.

The following screenshot shows you the Xcode application toolbar which contains the types of options that you have made available to you within the editor for Code Folding, Syntax Coloring, and any issues your project contains:

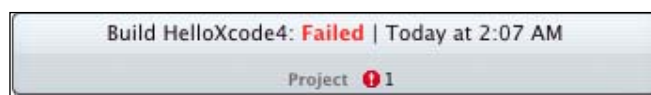


One other neat feature that has been integrated into Xcode 4 is the ability to open other files outside your project. This is done by right-clicking and selecting the project name located above the Activity window as shown in the following screenshot:



Application Status Bar/Activity Window

This provides you with system messages about the progress of a range of activities including compiler and syntax errors and information relating to project builds. In previous versions of Xcode, this was known as the status bar:



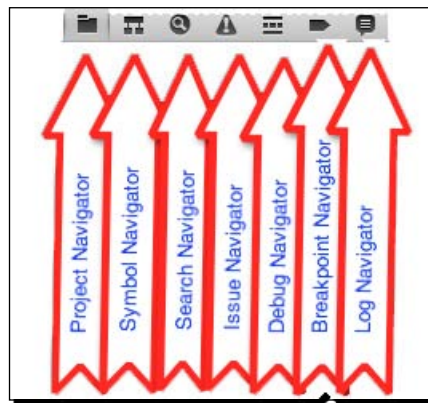
Workspace Settings

If you have used previous versions of Xcode, you will notice that the many windows used to perform the development tasks you work with on a daily basis, have now been combined into a single window. The work area has had several unique UI elements applied that make it much easier to work on many different tasks, even multiple projects, and best of all without cluttering your work area.

Introducing the Unified Navigation UI

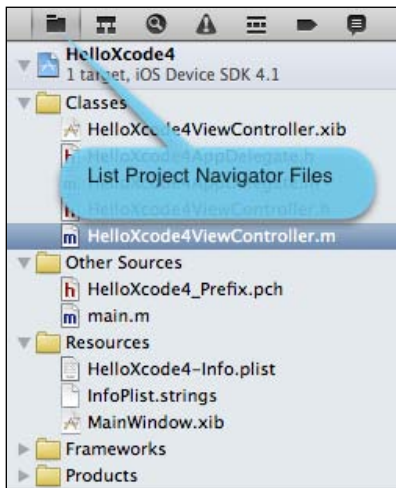
The newly introduced Navigation UI contains a list of useful navigators that provide you with an easy way to filter your project. This is located on the left-hand side within the Xcode workspace and includes a List of files in your project, Sorted Symbols, a Central Search Interface, Issues Tracking, Debugging data with Compressionable Stack traces, Active and Inactive Breakpoints, and a persistent Collection of logs.

The Unified Navigator provides live filtering of content and search results so that you can spend more time focusing on your current task:



Listing files in a project

The Project Navigator shows you a list of all of your projects, folders, and files that are included within your project workspace:



The Project navigator also contains badges, which are for SCM and pertain to subversion and code repositories. We will be discussing these in *Chapter 9, Source Code Management with the Version Editor*.

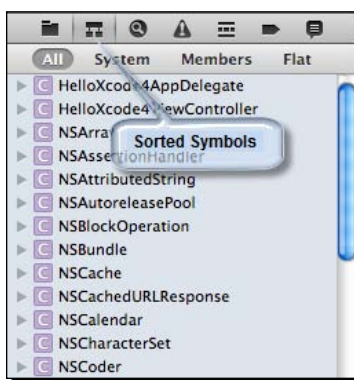
Status	Description
M	Specifies that the file(s) have been modified locally.
U	Specifies that the file(s) have been updated in the repository.
A	Specifies that the file(s) have been added locally.
D	Specifies that the file(s) have been deleted locally.
I	Specifies that the file(s) have been ignored.
R	Specifies that the file(s) have been replaced within the repository.
*	Specifies that the folder contents contain mixed statuses.
?	Not yet added to the source control repository.



It is worth mentioning that these badges propagate up to the highest container, so that you can see the source control status of the whole workspace, regardless of the level disclosed. These are discussed in greater detail in *Chapter 9, Source Code Management with the Version Editor*.

Sorted Symbols

The **Symbol Navigator** allows you to browse through the symbols within your project. It is a good idea to wait until Xcode has finished indexing your project before you use this feature:



To specify exactly what you would like to have listed within this view, use the Search text box which is located at the bottom of the navigator as well as the scope buttons located at the top:

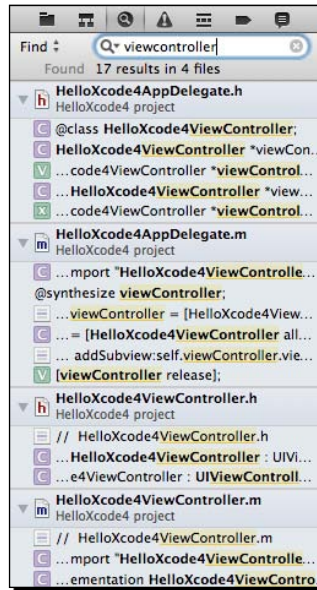


An explanation of each of these scope buttons is given below:

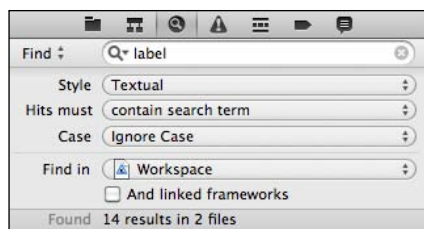
SCOPE BUTTON	DESCRIPTION
All	When this button is selected, the symbol navigator displays all types of symbols. Alternatively, when this is not selected, it displays all the classes and protocols.
System	When this option is selected, the navigator displays all the symbols that are contained within the system frameworks as well as those that are contained within your project. To see only the symbols in your project, deselect this button.
Members	When this option is selected, the navigator displays the members of classes.
Flat	When this option is selected, the navigator displays all of the classes arranged alphabetically. When this option is deselected, the navigator displays the class hierarchy.

Central Search Interface

The Central Search Interface allows you to search for a specific term throughout your entire project, or projects that are contained within the Xcode workspace. The results found will be displayed within this pane:



Searches can be customized by clicking on the magnifying glass within the search field and choosing **Show Find Options** to display the **Find Options** dialog as shown below. You will also notice that the **Find Options** dialog allows you to search on Regular Expressions as well:



You are also able to filter the results returned to display what you would like to have listed in this view at any given time. You can do this by using the Search text box, which is located at the bottom of the navigator. An explanation of each of these symbols is given below:

TYPE	DESCRIPTION
=	When this is displayed, these contain any comment fields, which contain the matching search term.
C	When these types are displayed, they denote the classes used by the system frameworks. In our HelloXcode4 example project above, our variable is making use of the system framework <code>UIViewController</code> class.
V	When these types are displayed, these relate to any variables, which contain the search term. In our HelloXcode4 example project, we have declared a pointer variable <code>*viewController</code> which has been declared of type <code>HelloXcode4ViewController</code> that inherits the <code>UIViewController</code> class.

Issues Tracking

During the building process of your application, the compiler analyzes your program code for potential problems. If the compiler finds any problems while building your code, the issues navigator is displayed. Selecting any of the errors or warnings in the list will display the line at which the problem occurred in the source editor. The issues navigator allows you to display problems by file or by type.

Using Static Analysis to find potential problems

Another neat new feature, which has been integrated into the Xcode 4 IDE, is the Static Analyzer. This new addition allows you to examine the syntax of your code for bugs prior to building. The analyzer displays the program logic-flow of your application, which can help you track down potential bugs found within your application. Don't worry too much about this, as we will be covering this in greater detail when we come to *Chapter 8, Debugging Xcode Projects*.

Debugging data with Compressionable Stack Traces

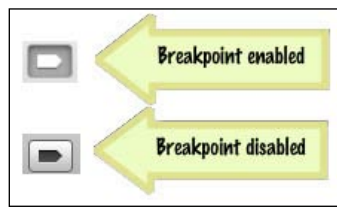
Whenever you pause execution of your program code, or run your code up to the point where your breakpoint exists, Xcode will open up the debug navigator and display the threads that were running when the execution was paused. Under each of these threads listed, is the thread stack at the point up to which your program executed. If you select a stack frame, you will see the corresponding source file or disassembled object code, that will be displayed within the source editor window.

There is also a feature to allow you to display how much stack information you would like the threads and stacks navigator to display and this can be done by using the slider which is located at the bottom of the debug navigator window. We will be discussing this in greater detail when we come to *Chapter 8, Debugging Xcode Projects*.

Active/inactive breakpoints

Setting breakpoints within your code can be useful when you want to stop at known points within your code where you might know the problem exists, or you may just want to check the values of variables within your source code.

In any event, these can be very useful. To set breakpoints within your code, you need to have a source code file open and then click in the gutter area next to the spot where you would like to stop execution. When you add a breakpoint within your code, Xcode automatically enables breakpoints, as indicated by the breakpoint state button in the toolbar:



You can toggle between enabled and disabled states of breakpoints by clicking on the breakpoint-state button. You can also disable an individual breakpoint by clicking on its icon. To remove a breakpoint completely, drag it out of the gutter area. We will be discussing this in greater detail when we come to *Chapter 8, Debugging Xcode Projects*.

Collection of Logs

The log navigator in Xcode 4 replaces what was known as the build log window within Xcode 3. When you select one of the builds in the build log, the results are displayed within the editor area. To proceed to see the error or warning listed within the build, double-click on it to have it opened within the source editor pane. This view can also display a more verbose listing of the error or warning, by clicking on the list icon at the end of the command line. We will be discussing this in greater detail when we come to *Chapter 8, Debugging Xcode Projects*.

Jump Bar

This is a new addition to Xcode 4, which I absolutely love. This neat feature, that looks like more of a "Breadcrumb" Bar, provides you with a quick way of navigating from one folder to another from within the Xcode IDE. It is displayed at the top of every editor pane and shows the relative location of your current file. If you click on any location, it will jump to any other folder at the same level:



Using Code Assistants

A new feature that has been introduced within Xcode 4 is the use of Code Assistants. This includes a new Fixit feature that provides advanced code completion and flags common bugs or typos. The **Fix-it** feature can suggest appropriate symbol spellings and supply correct punctuation, assisting developers to write code faster, making fewer mistakes. Through the use of Static analysis you can find and flag common bugs and errors such as a failure to properly release memory that is no longer needed. We will be covering this in greater detail when we come to *Chapter 8, Debugging Xcode Projects*.

Introducing the new and improved LLVM Compiler 2.0

This technology is an open source compiler technology, which is currently being led by Apple's compiler team to be used in several high-end performance projects around the globe. The LLVM 2.0 compiler has also been substantially updated and now compiles twice as fast as the GCC compiler, producing applications which run faster. It has been rewritten as a set of optimized code libraries, which have been designed around today's modern chip architectures.

It has been fully integrated into the Xcode 4 development IDE and provides complete support for the following languages: C, Objective-C, and C++.

Version Editor

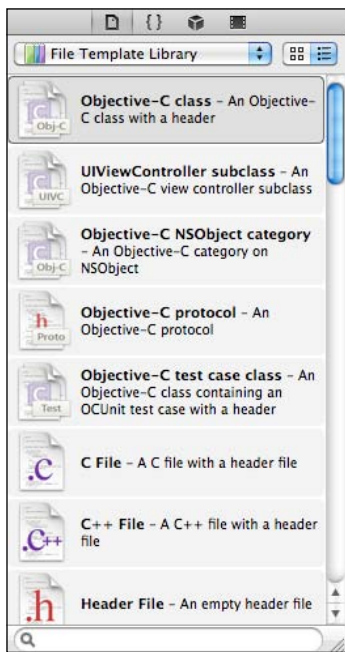
This new improved version makes your life a lot easier, compared to previous versions of Xcode. This is now part of the Xcode 4 IDE, and makes it easy for you to see any two versions of your source code side-by-side which you want to compare, in real time, all within your Xcode 4 development workspace. You are also able to view this comparison view much like Apple's Time Machine. With its timeline look and feel, you are able to drag the slider in the middle that allows you to travel back in time through your project, allowing you to compare any two versions of your file.

This slick editor interface can also show you a detailed log of post events and track blame for post check-ins. It is even possible to manage multiple projects within a single Xcode 4 workspace. We will be covering this in greater detail when we come to *Chapter 9, Source Code Management with the Version Editor*.

File Templates Library

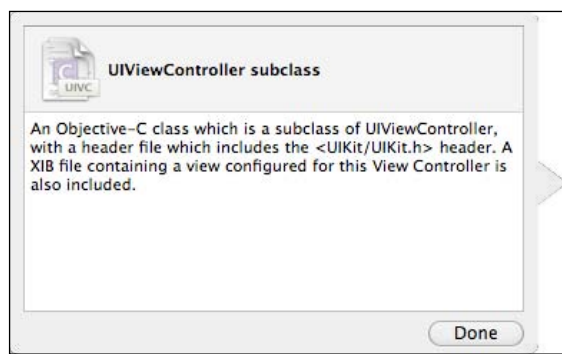
The File Templates library ranges from a variety of templates for applications to subclasses of commonly used Cocoa classes. To use a template, using your mouse, drag it from the library to the folder in the Project navigator where you want to keep it.

File Templates have sub-categories, from which you can choose from the popup menu below the line of buttons. You can also display the templates as icons, or icons with their associated text as shown in the following screenshot:



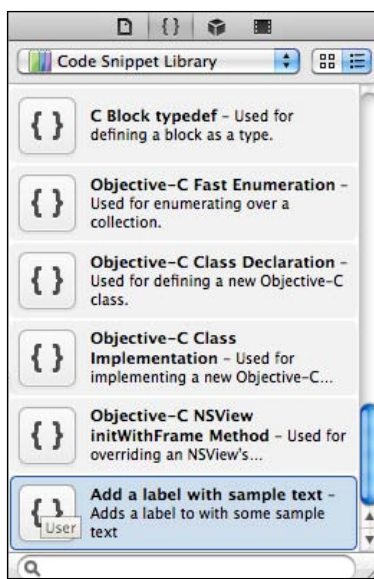


When you click on a template, an information window will open to the left and contain a description about how to go about using the template. Refer to the screenshot below, which explains this.



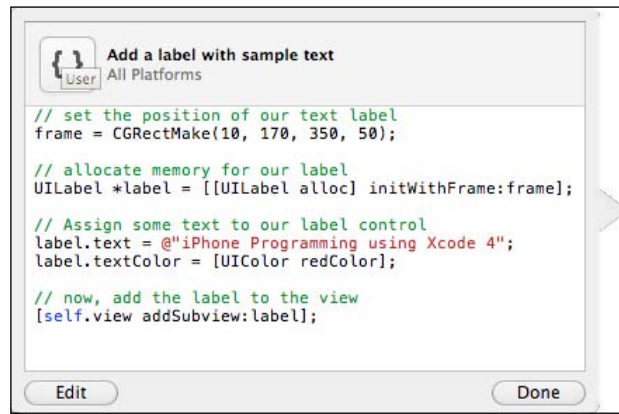
Code Snippets Library

The Code Snippets library contains fragments of reusable code that you can use within your applications. In order to use one of these predefined code snippets, you need to select it with your mouse and drag it into the code editor to the source code file you want to apply this to:



You also have the ability to create your own reusable code snippets to be used within the other interesting projects that you create. To do this, first highlight the piece of code within your source code and drag it into the code snippet library window panel.

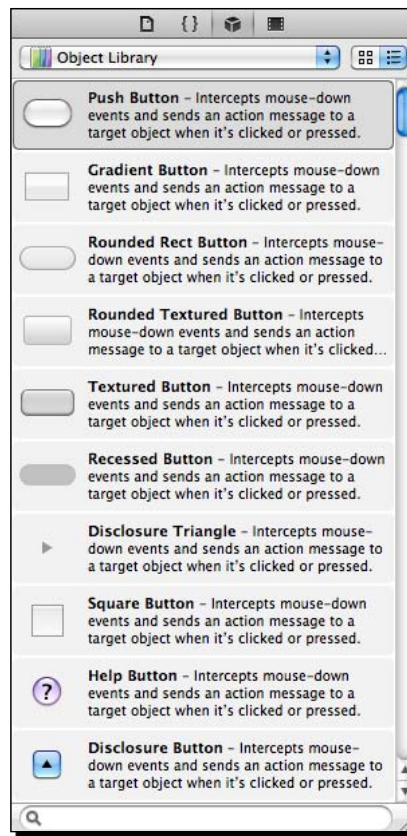
A popup window will be displayed alongside your code snippet. Provide your code snippet with a meaningful name as well as a completion shortcut (optional). Using a completion shortcut, allows you to add it to your source code without having to grab it from the Code snippet library. You just type in the name and your code will appear:



If you are adding any new code snippets or editing code snippets that are added to the Code Snippet Library, these snippets will be flagged with the word "User" to differentiate between what are user-defined and System code snippets.

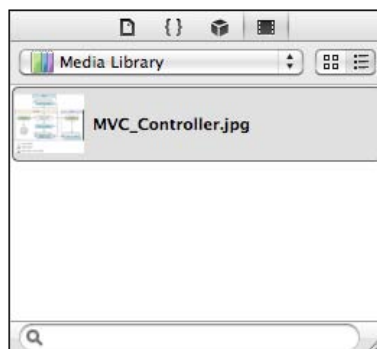
Object Library

The Object Library contains various objects, that are organized with subcategories and provide information in popup windows as we have seen with File Templates and Code Templates. These objects are for use with Interface Builder (We will be talking more about Interface Builder in the next chapter). The Screenshot below shows a sample list of the different types of objects that the Object Library contains:



Media Library

The Media Library includes graphics and icons that are located within the `Resources` folder of your project or workspace. There is a Search field located at the bottom of the libraries pane, which you can use to filter the library items that are displayed from within the selected library:



Resetting Xcode's Development Environment Settings

In the event that something goes horribly wrong by which you have made some changes to your Xcode development environment settings and have caused your IDE to become unstable, don't panic! There is a very easy way to reset your environment settings back to their default settings. To do this, we first need to open up the Terminal utility application using **Shift | Command | U**. At the command prompt, type in the following commands:

```
defaults delete com.apple.Xcode
rm -rf ~/Library/Application\ Support/Xcode
```

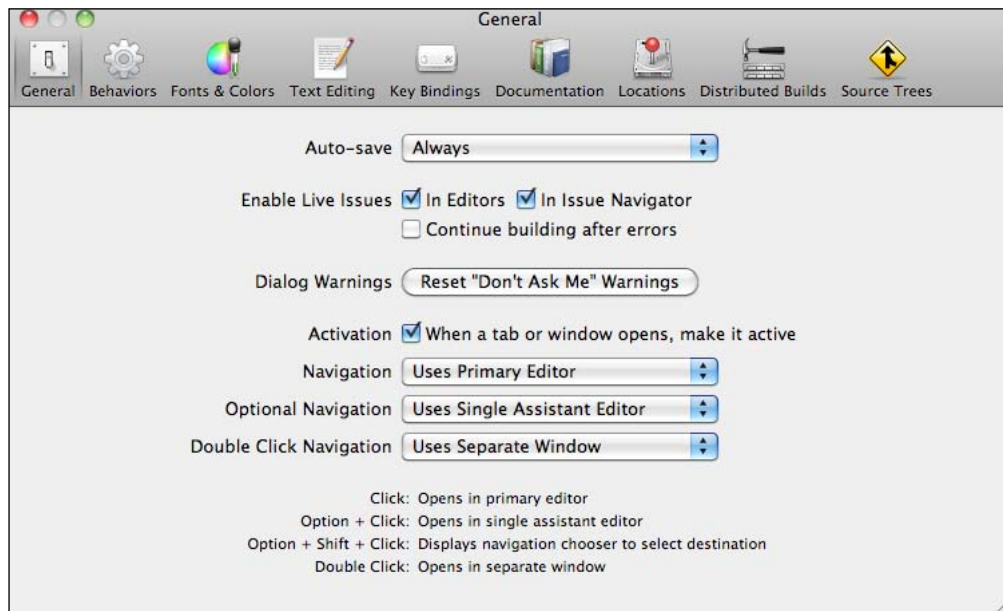
These commands will delete the `plist` file for Xcode and put everything back to their default settings. Before you do this, take note of the following:



This will remove all of your Xcode preferences, which includes: layout choices, file history, and toolbar settings.

Xcode Workspace Preferences

Xcode gives you the ability to change and customize the default installation settings to be more to your liking. We will be focusing on what each of these buttons comprise of in the sections below:



General

This button allows you to control general environment settings such as your auto-save and navigation preferences:

- ◆ **Auto-save:** Xcode has the ability to automatically auto-save files that you have changed before building, quitting Xcode, committing files to a repository, or closing a workspace window.
- ◆ **Issues:** Xcode can display runtime warnings and errors in place in the editors, and also in the issue navigator. You can select one or both of these options. Select the **Continue building after errors** checkbox if you want Xcode to continue building a target when an error occurs.
- ◆ **Dialog Warnings:** Xcode has several warning dialogs that you can disable by clicking a **Do not show this message again** checkbox in the dialog. To re-enable all of these warning dialogs, click on the **Reset "Don't Ask Me" Warnings** button.
- ◆ **Activation:** To shift the focus to a new tab or window when it opens, select the **When a tab or window opens, make it active** checkbox.
- ◆ **Navigation:** You can use this to configure what happens when you navigate to and select a file in the project navigator or jump bar.
- ◆ **Optional navigation:** You can use the navigation chooser to set how you would like the Xcode development environment to look. You can specify what you want to do when you open a file. You can specify to have it open in any editor pane in any window and tab, or to open the file in a new editor pane, window, or tab.

Behaviors

The Behaviors preference pane allows you to specify actions that occur when certain operations are initiated or completed. You use this to tailor your workflow, for example, to always show the latest Build Log when you start a build. Triggers include starting and stopping building, testing, launching, searching, or restoring a device. Actions include playing sounds, bouncing the dock icon, or executing a script.

Another example where this is useful is when you want Xcode to display the debug area when your code pauses at a breakpoint or you can choose to display the Issues Navigator when the build fails. You can even specify Xcode to go to a specific tab within your workspace, all handled right within the Alerts pane.

Fonts & Colors

These preferences determine how you would like to configure the Xcode Integrated Development Environment editor setup to suit your needs. From here, you are able to choose from a variety of color schemes or customize one to your liking.

Text Editing

These preferences allow you to configure the Xcode Integrated Development Environment editor setup to suit your needs. You can specify to turn on/off line numbering, code folding, code completion, as well as many other options relating to indentation of your code.

Key Bindings

Key binding refers to the shortcuts that you use to access operations within the Xcode development IDE. These generally follow the norms for Mac OS X software, but the key bindings specified within this section are specific to Xcode.

Documentation

This preference panel lets you define how to manage the Apple Developer documentation. You may recall that this was an option during the installation of Xcode. Selecting the checkbox ensures that you keep the documentation up-to-date; similarly you can use the button to do this on demand. By default, you will have access to the Mac OS X Snow Leopard core library and the Xcode 4 iOS developer libraries. You can also specify and choose from a range of other libraries.

Locations

When you build your project within Xcode 4, the progress is shown in the Activity View located on the toolbar. The build steps are recorded within the Build log of the log navigator. This keeps a catalog of the build logs so you can see results from previous builds if required.

You can specify the location of where you would like to store these **project-generated files** as necessary to suit your workflow. By default, Xcode stores project-generated files in standard folders inside `~/Library/Developer/Xcode`. **The following form options** are explained below:

- ◆ Derived Data (index, logs, build) location:

Xcode 4 provides a new way for indexing files by creating an index for the entire workspace. This resolves any issues that occurred in previous versions when referencing across projects. The indexer uses the new LLVM compiler 2.0 to parse its source files and this improves performance dramatically.

- ◆ **Snapshots Location:**

The Snapshots feature has been modified to be much faster and more reliable than it has been in previous versions of Xcode. In order to be able to use Snapshots, you need to ensure that you install the System Tools as part of your Xcode 4 installation.

- ◆ **Archives Location:**

This location specifies where you would like to create distribution builds of your product. These could be used to provide development milestones, or posting nightly builds of your product, or for distributing the final release of your product.

- ◆ **Build Location:**

Xcode 4 allocates a common directory for each project or workspace with each project being placed within a separate folder under its project name. If the same project is located within two separate workspaces, the same project is created in two different locations so that its precompiled headers, indexes, and build products do not conflict with one another. If you don't want to accept the default folder location, you can change this to something more appropriate.

Source Trees

This preference is aimed at software development teams for which a particular source file/s is required to be located in a different location other than the project folder. An example of this might be where you want to share a custom class among other developers or team of developers.

Distributed Builds

The Distributed Builds preferences pane is used to set up workgroup builds and to make your computer available on the local area network to teammates for their workgroup builds. By using this set up, you can reduce your build times by adding the processing power of your teammates' computers to your own by converting your local builds to workgroup builds.

Workgroup builds are network-based builds that can use more than one computer to build a product. Workgroup builds involve a build client and one or more build servers. The build client is the computer that initiates the build and uses a set of build servers to complete it. **Build servers** are computers that have been made available to assist build clients in completing their builds. One computer can be both a client and a server.

Summary

In this chapter, we covered a substantial amount of topics relating to the Xcode 4 workspace environment. We saw how to go about downloading and installing Xcode 4 and the iOS4 SDK as well as what comprises the Xcode 4 IDE and the different types of libraries that are part of the Xcode 4 workspace (**File Templates**, **Code Snippets**, **Object Library**, and so on). We also created a very simple iPhone application using View Controllers and using the system frameworks to display a label to the view.

Now that we've learned about the main components of the Xcode 4 workspace environment and how to go about building a simple iPhone application, we are ready to start focusing on the GUI side of things with Interface Builder.

In the next chapter, we will dive right in and work with the Interface Builder (**GUI**) application. We will also improve on our very simple iPhone application and start to add controls from our Object library as well as learning how to connect these components up to the program logic so that we end up with a fully functional iPhone application.

3

Working with the Interface Builder

In this chapter, we will learn about the interface builder application and how we can use this visual tool to design our user interfaces for our iOS applications. We will also look at how we can use objects from the Object library to create the necessary outlets and actions so that we can programmatically use these within our code, as well as learning how to reposition and bind objects so that they work together.

In this chapter, we will:

- ◆ Familiarise ourselves with the Interface Builder tool
- ◆ Learn how to add controls to our view-based controller using MVC
- ◆ Learn how to Position and Align controls to the User Interface
- ◆ Understand what are Rotatable and Resizable Interfaces
- ◆ Learn how to reposition controls within the view when the device is rotated
- ◆ Learn how to bind UI control objects to code
- ◆ Enhance our "Hello World" iPhone application, created in the previous chapter
- ◆ Learn how to implement File Saving and Loading

We have got quite a bit to cover, so let's get started.

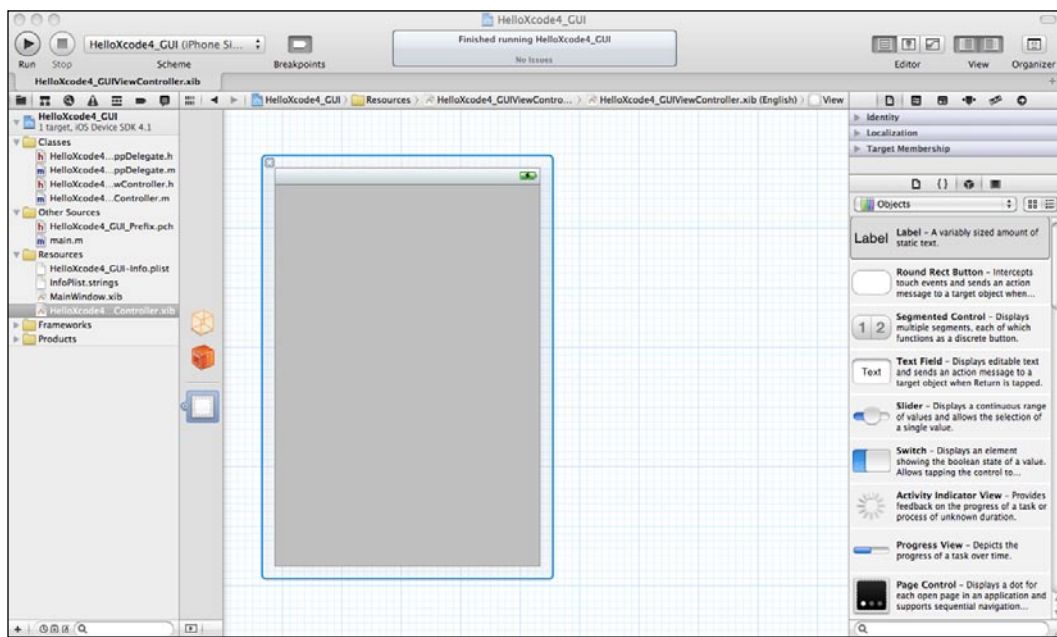
Getting to know the Interface Builder environment

Interface Builder (IB) is a visual tool that enables you to design the user interface for your iPhone or iPad applications. By using Interface Builder, you are able to drag and drop views onto your window and then connect the various views with **outlets** and **actions** so that they can programmatically interact with your code.

In Xcode 4, Interface Builder (IB) appears in the editor area of the workspace window when you select an **xib** file from the project navigator window. When you open an **xib** file, the Interface Builder file inspectors appear in the utility area, and you are able to select Interface Builder Objects from within the libraries' pane and drag them into your interface builder canvas area.



An **xib** file is what gets created by Interface Builder, and contains the user interface design and objects. In previous versions of Xcode, **nib** files were used and were created by NeXTStep computers back in the mid-late 80s. More information on the nib file format can be found at the following location: http://en.wikipedia.org/wiki/Interface_Builder.



Adding Controls to your user interface

Now, this is where the real fun starts. We will start by creating our iPhone application. Instead of creating a typical "Hello World" application, we will be making this one a bit more flexible.

The program will present the user with a (`UITextField`) field for typing in their name, as well as a (`UIButton`) button which will display the message "Welcome to iOS Programming" followed by the name of the user. This will be outputted to our Output (`UILabel`) label control.

We have some fun work ahead, so grab yourself a cup of your favorite beverage and let's get started.

Time for action – creating the HelloXcode4_GUI application

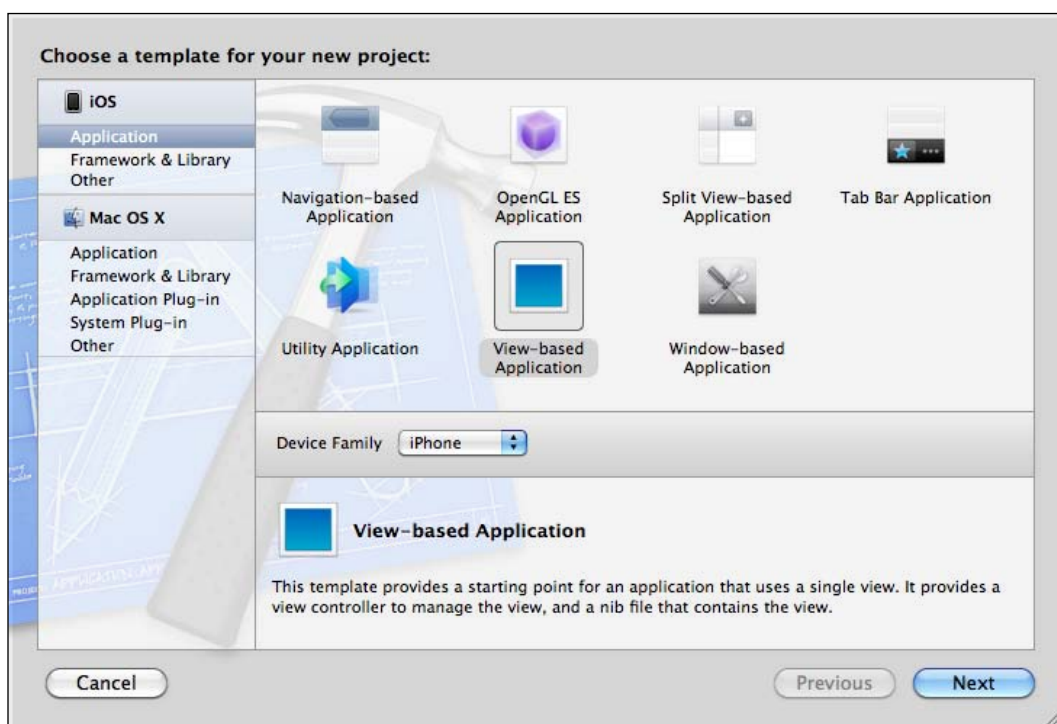
Before we can proceed with creating our "HelloXcode4_GUI" application, we must first launch the **Xcode** development environment. This can be located in the `/Xcode4/Applications` folder. Alternatively, you can use the spotlight to search for Xcode by typing xcode into the search box window.

When you launch Xcode, you will be presented with the **Welcome to Xcode** screen, as shown in the following screenshot:



It is very simple to create this in Xcode. Just follow the steps listed below:

1. Choose **Create a new Xcode project**, or **File | New Project**. This will bring up the project template dialog, which is shown in the screenshot below.
2. We need to select the **View-based application** template to use. What this template provides us with is a starting point for an application that uses a single view. It provides a view controller to manage the view, and an xib file that contains the view.
3. Ensure that you have selected **iPhone** from under the **Device Family** dropdown, as the type of view to create:

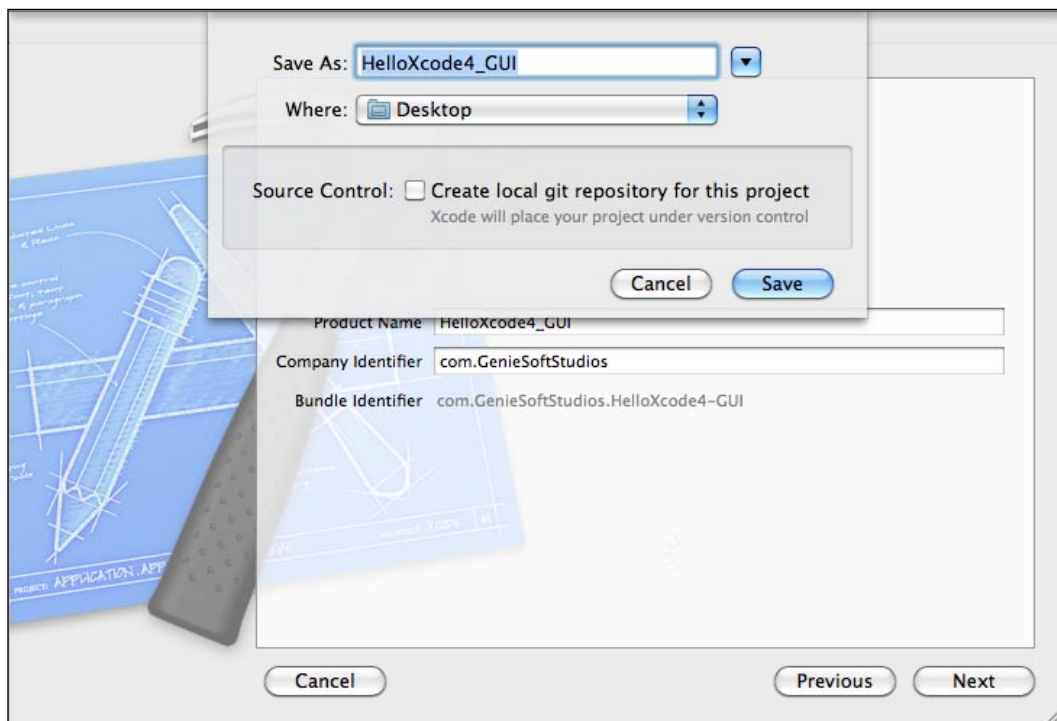


4. Click on the **Next** button. You will be prompted to enter in a name for your project.
5. Enter in **HelloXcode4_GUI** and then click on the **Next** button to proceed to the next step of the wizard. You will then be asked to choose the location where you would like to save your project.

You will also notice that there is an option to automatically save your project to Source Control. If this option gets checked, this will create a local repository on disk so you can check-in your work. This option is not checked by default.



Source Control allows you to keep track of changes made to a file and who applied those changes. If you have worked with Microsoft Visual Source-Safe or Tortoise SVN, you will already be familiar with Source Control and how it works. We will be covering this in *Chapter 9, Source Code Management with the Version Editor*.



What just happened?

In the above section, we looked at the steps involved in creating a View-based Application for our HelloXcode4_GUI application. In the next section, we will take a look at the main components of our application and the core application architecture, before taking a look at the Application Life-Cycle for all iOS applications.

Application structure of our HelloXcode4 example application

Once our project has been created, you will be presented with the Xcode workspace interface. All files that the project template created for you will be displayed within the Project Navigator window section. The important files to take note of are the following:

- ◆ Main.m
- ◆ HelloXcode4_GUIAppDelegate.h, and
- ◆ HelloXcode4_GUIAppDelegate.m

Main.m

The **main** function is where the single `UIApplication` object is created and the function called **UIApplicationMain()** takes care of that. This function takes four parameters, and uses them to initialize the application. There is no reason to edit this main function as it is just there to simply kick-start your application xibs and view controllers into running:

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

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Although you should never have to change the default values passed into this function, it is worth explaining their purpose in terms of starting the application. In addition to the `argc` and `argv` parameters passed into `main`, this function takes two string parameters that identify the principal class (that is, the class of the application object) and the class of the application delegate.

If the value of the principal class string is `nil`, `UIKit` uses the `UIApplication` class by default. If the value of the application delegate's class is `nil`, `UIKit` assumes that the application delegate is one of the objects loaded from your application's main nib file (which is the case for applications built using the Xcode templates). Setting either of these parameters to a non-`nil` value causes the `UIApplicationMain` function to create an instance of the corresponding class during application startup and use it for the indicated purpose. If your application uses a custom subclass of `UIApplication` (which is not recommended, but certainly possible), you would specify your custom class name in the third parameter.

HelloXcode4_GUIAppDelegate.h

In the `HelloXcode4_GUIAppDelegate.h` interface file, we need to create a forward @class declaration `HelloXcode4_GUIViewController`. This basically tells the compiler a class with the name exists. In the @interface section, we declare a member variable to hold our view controller. Lastly, we use the @property directive to wrap our member variables with implicit get and set functions:

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

@class HelloXcode4_GUIViewController;

@interface HelloXcode4_GUIAppDelegate : NSObject
    <UIApplicationDelegate> {
        UIWindow *window;
        HelloXcode4_GUIViewController *viewController;
    }

    @property (nonatomic, retain) IBOutlet UIWindow *window;
    @property (nonatomic, retain) IBOutlet HelloXcode4_GUIViewController
        *viewController;

@end
```

The `HelloXcode4_GUIAppDelegate` gets hooked up to the `UIApplication` object via the Interface Builder (`.xib`) file called `MainWindow.xib`. This file contains a `Window` and a `View`.

HelloXcode4_GUIAppDelegate.m

The associated implementation file of the delegate, `HelloXcode4_GUIAppDelegate.m`, contains several messages that we can get from the `UIApplication` object and the template has already created one for us—`applicationDidFinishLaunchingWithOptions`:

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

    // Override point for customization after application launch.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
    return YES;
}
```

You will also notice that when the project was created, it added our view controller. This will eventually hold our label object for our text output. You will notice when you look at the new implementation file, `HelloXcode4_GUIViewController.m` that it contains some commented out functions.

The MainWindow.xib file

Another task that occurs at initialization time is the loading of the application's main nib file. If the application's information property list (HelloXcode4_GUI-info.plist) file contains the (NSMainNibFile)Main nib file base name key, the UIApplication object loads the nib file specified by that key as part of its initialization process.

The main nib file is the only file that is loaded for you automatically. However, you can load additional nib files later as needed.

Nib files are disk-based resource files that store a snapshot of one or more objects. The main nib file of an iOS application typically contains a window object, the application delegate object, and perhaps one or more other key objects for managing the window.

Loading a nib file reconstitutes the objects in the nib file, converting each object from its on-disk representation to an actual in-memory version that can be manipulated by your application.

Objects loaded from nib files are no different than the objects you create programmatically. For user interfaces, however, it is often more convenient to create the objects associated with your user interface graphically (using the Interface Builder application) and store them in xib files rather than create them programmatically. It is worth mentioning at this point, anything that can be done within the Interface Builder application can also be created dynamically through code. As you can see from the code snippet below, we create a button control and set its size and caption, before adding it to our view:

```
- (void) viewDidLoad
{
    UIButton * btn = [UIButton
        buttonWithType:UIButtonTypeRoundedRect];
    btn.frame = CGRectMake(0, 0, 100, 50);
    [btn setTitle:@"Hello, world!" forState:UIControlStateNormal];
    [self.view addSubview:btn];
}
```

The Core Application Architecture layer

Every iOS application that you develop is built using the UIKit framework. Games on the other hand use the Core Graphics and OpenGL/ES frameworks. The UIKit framework provides you with the key objects needed to run the application and to coordinate the handling of user input and the display of content on the screen.

From the time your application is launched, to the time that it exits, the UIKit framework manages the majority of the application's key infrastructure. An iPhone application receives events continuously from the system and must respond to those events. Receiving the

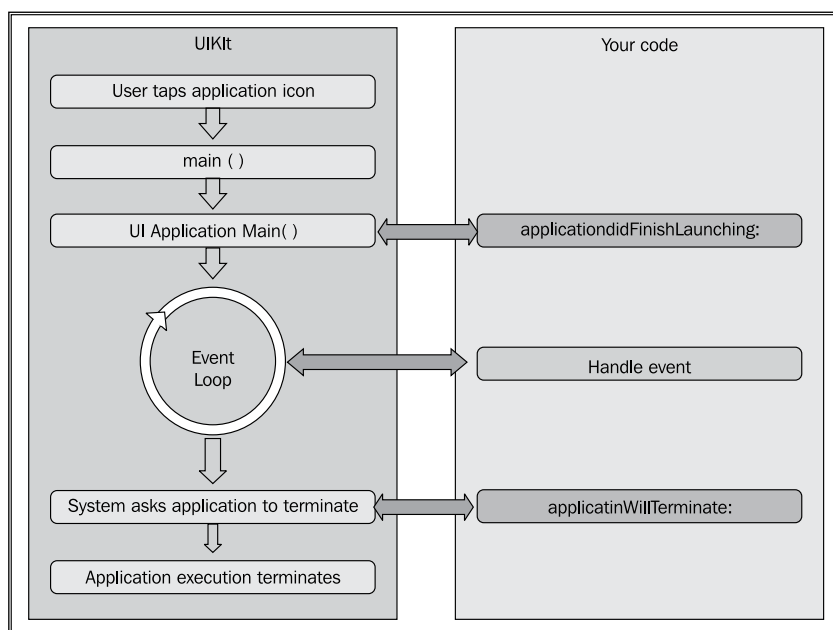
events is the job of the `UIApplication` object, but responding to the events is the responsibility of your custom code. In order to understand where you need to respond to events, it helps to understand a little bit about the overall life cycle and event cycles of an iPhone application. We describe these cycles below.

The application life cycle

The application life cycle comprises the sequence of events that occur between the launch and termination of your application. When a user launches your application by tapping the icon on the Home screen, the system displays some transitional graphics and proceeds to launch your application by calling its **main** function.

This function handles the bulk of the initialization work before being handed over to the **UIKit**, that loads the application's user interface and enters its event loop. UIKit coordinates the delivery of events to your custom objects and responds to commands issued by your application. Whenever a user performs an action that would cause your application to quit, UIKit notifies your application and begins the termination process.

In the screenshot below, it shows the simplified startup life cycle for a newly launched iOS application. This diagram shows the sequence of events that occur between the time the application starts up and the point at which another application is launched. At key points in the application's life, UIKit sends messages to the application delegate object to let it know what is happening. During the event loop, UIKit also dispatches events to your application's custom event handlers, which are your views and view controllers:



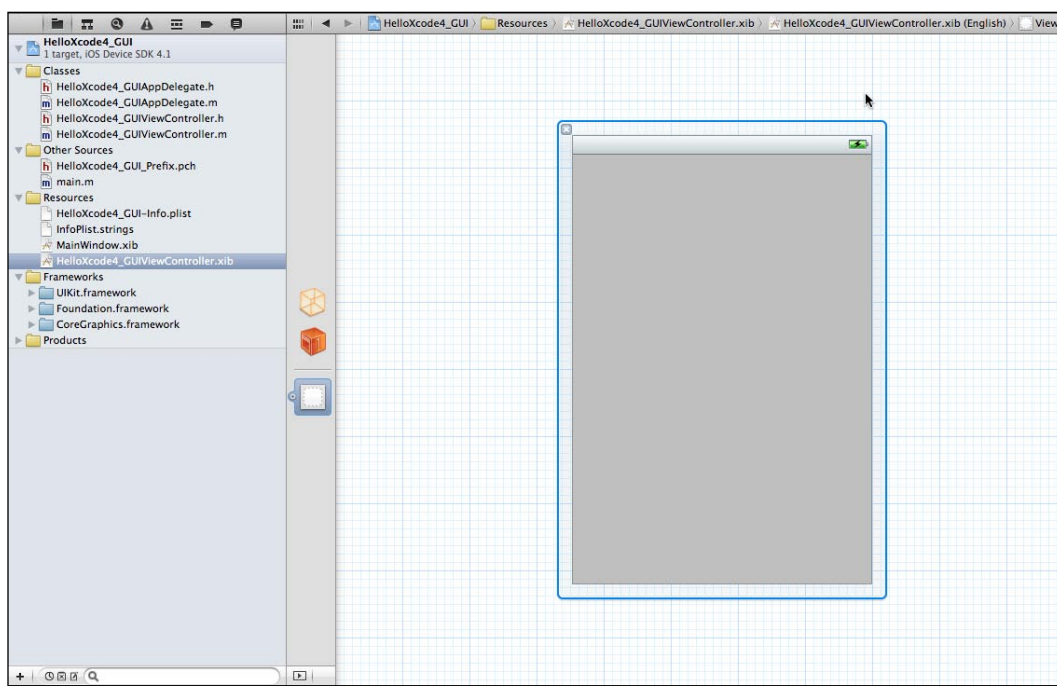
At initialization and termination, UIKit sends specific messages to the application delegate object to let it know what is happening. During the event loop, UIKit dispatches events to your application's custom event handlers.

The `applicationDidFinishLaunching:` method is created automatically and is used to kick off your application. This event fires automatically on your delegate whenever your application launches.

The `applicationWillTerminate:` method is called at termination time when your application is running in the foreground or background to perform any required cleanup. You can use this method to save user data or application-state information that you would use to restore your application to its current state on a subsequent launch. This method is not called if your application is currently suspended.

Time for action – adding object controls to our View

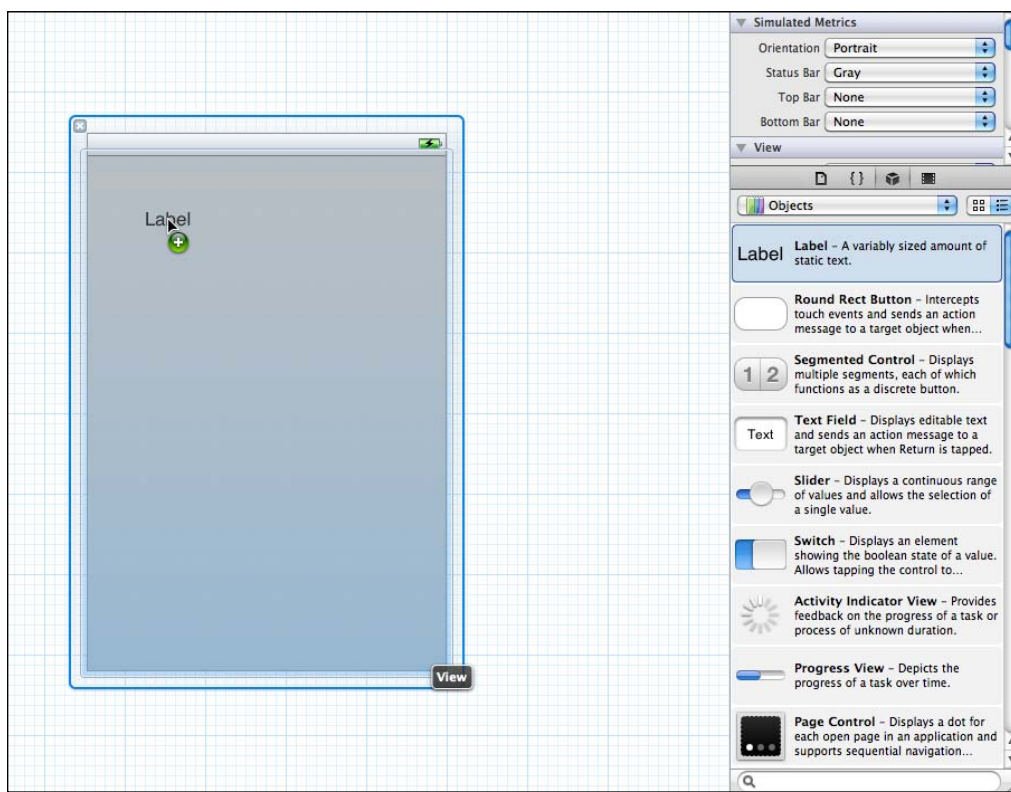
We will now start to design our user interface, using the controls from our Xcode Object Library. Firstly, from the Project Navigator window, and under the Resources folder, select the `HelloXcode4_GUIViewController.xib` file. A blank View canvas will be displayed to which we will start to add our components:



In the coming pages, we will be adding the following control items to our View Controller:

- ◆ (UILabel)
- ◆ (UITextField)
- ◆ (UILabel)
- ◆ (UIButton)

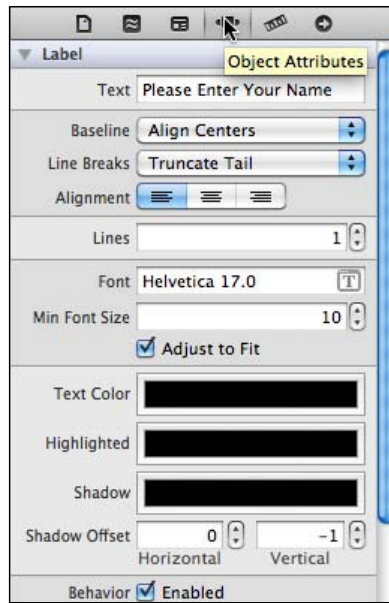
In the next section, we will start to build the user interface for our example application by dragging some of the UI components to our canvas, and changing some of the components' properties:



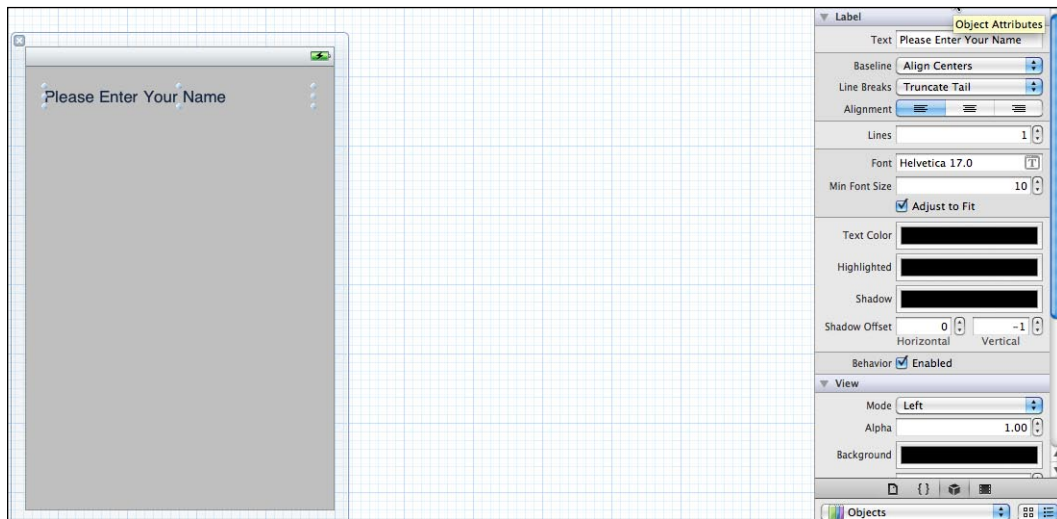
Next, to add an item to our canvas, follow these steps:

1. Simply drag the (**UILabel**) Label item from the Object Library to the view, as shown in the screenshot above. You can use the search field located at the bottom of the Object Library to locate any of the UI elements.
2. Once the label control has been added, select this control and click on the **Object Attributes** button and enter the text **Please Enter Your Name:**

You will notice that the **Object Attributes** properties pane is displayed and contains various properties associated with this particular control. You are able to apply the **Text Color** and **Background Color** for your label, as well as alignment:



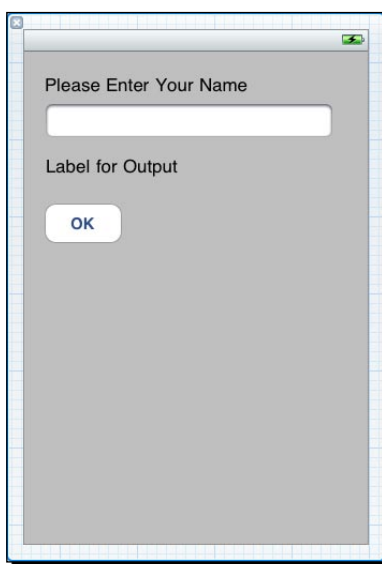
The screenshot below shows our updated label control with the text which we added previously. Next, we will start to add our remaining controls to our **View Controller**:



What we now need to do is to add the remaining control objects to our View. We still need to add our UITextField, UIButton, and UILabel objects. To achieve this, follow these steps:

1. From the Object Library, select and drag a (UITextField) TextBox control to the view and place this control directly under our label control **Please Enter Your Name**. Resize the control accordingly.
2. Next, select and drag a (UILabel) Label control and add this to our view. Resize this accordingly as this will be used to display the output. Click on the **Object Attributes** tab and enter the text **Label for Output**.
3. We are nearly there, so just hang in for a couple of minutes. We need to add a (UIButton) Round Rect Button control to our view. Modify the Object Attributes of the Round Rect Button control and set its Title to **OK**.

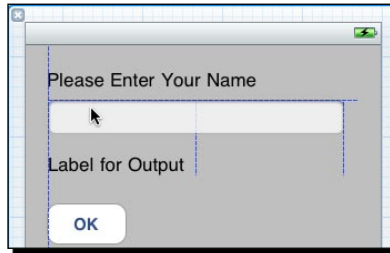
If you have followed the steps correctly, your view should look like something shown in the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



If you were to run this application, all you would see would be the controls as laid out on your screen. The only interaction that will happen up to this point is that the keyboard will be displayed if you clicked into the Text Field.

Clicking on any of the other controls will not do anything as we will need to hook these up. This is done in the section *Binding Control Objects* later in this chapter where we will be creating and making use of Outlets and Actions.

Interface Builder provides some nice features for aligning and setting the properties of visual elements within your View. These are called guidelines or crosshairs which enable you to ensure that your controls align up correctly. In the screenshot below, we will look at how we can use these guidelines to position our controls within the view:



when you change the size and position of a Label, Button, or Text Fields. These are referred to as automatic guides. They provide you with a good way to optimally position controls within your user interface and ensure that you conform to the Apple Human Interface Guidelines to ensure a consistent visual and behavioral experience throughout your application, by providing a professional look and feel, as well as ensuring that your application looks and behaves the same way as existing iOS applications on the iPhone.

For more information on the Apple Human Interface Guidelines, please refer to the following: http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines/XHIGIntro/XHIGIntro.html%23//apple_ref/doc/uid/TP30000894-TP6.



Interface Builder will also show dynamically updated guides and will even show you the pixel distances between various points within the window. You can also add your own custom guides via the **Editor | Add Vertical Guide** (or **Horizontal Guide**) menu. These will remain visible while you are designing your user information, but don't display when your application is running.

What just happened?

In the above section, we looked at the steps involved in adding a number of controls to our view from the Xcode Object Library as well as setting their control properties using the Object Attributes button. Finally, we look at how we can use the guidelines and crosshairs to position and align visual elements within our view to ensure that they line up correctly.

Understanding Rotatable Interfaces

Rotatable and Resizable interfaces allow you to look at your applications or web content in various views. For instance, say you wanted to view a website or play a game in landscape mode; the iPhone introduces a way and provides on-the-fly rotation which is fast, and provides a natural feel.

When designing your iOS applications, think about how the user will be interacting with your application. Will you be designing an app that will force portrait mode only, or will it be flexible enough to allow for multiple views. The best part of all this is that the process to enable rotation is a painless process.

Time for Action – enabling Interface Rotation

To allow your application's interface to rotate and resize, all that is required is a single method. When the iPhone wants to check to see whether it should rotate your interface, it sends the `shouldAutorotateToInterfaceOrientation:` message to your view controller, along with a parameter that indicates which orientation it wants to check.

Your implementation of `shouldAutorotateToInterfaceOrientation:` should compare the incoming parameter against the different orientation constants in the iOS, by either returning `TRUE` (or `YES`) if you want to support that orientation.

The four basic screen orientation constants are described below:

ORIENTATION METHOD	iOS ORIENTATION CONSTANT
Portrait	<code>UIInterfaceOrientationPortrait</code>
Portrait upside-down	<code>UIInterfaceOrientationPortraitUpsideDown</code> (This is rarely used on the iPhone as this is mainly used and implemented on the iPad.)
Landscape Left	<code>UIInterfaceOrientationLandscapeLeft</code>
Landscape Right	<code>UIInterfaceOrientationLandscapeRight</code>

For example, to allow your iOS interface to rotate to either the portrait or landscape left orientations, you would implement `shouldAutorotateToInterfaceOrientation:` in your view controller as the following:

```
- (BOOL) shouldAutorotateToInterfaceOrientation: (UIInterfaceOrientation)
    interfaceOrientation {
    Return (interfaceOrientation == UIInterfaceOrientationPortrait ||
        interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
}
```

What just happened?

The `Return` statement handles everything and it returns the result of an expression comparing the incoming orientation parameter (`interfaceOrientation`) to the `UIInterfaceOrientationPortrait` and `UIInterfaceOrientationLandscapeLeft`. If either of the comparisons is true, the function returns `TRUE`. Alternatively, if either one of the possible orientations are checked, the function returns `FALSE`.

By adding this simple method to your view controller, your application will automatically sense and rotate the view for portrait or landscape left orientations.

Relocating controls within the view on Rotation

When we rotate our iOS application, the screen dimensions shift. The only problem is that we still end up with the same amount of free usable space, but our view is laid out differently. In order to ensure that controls fully utilise and resize automatically for the new orientation, we use the combination of rotatable and resizable screen rotation.

The Simulator supports changes in view orientation. To change the view to landscape mode, press the *Command + Right Arrow* key combinations. The screenshot below shows how your application looks in landscape mode. Press the *Command + Left arrow* key to change it back to portrait mode:



You will notice that your application did not respond to the changes in view orientation. This is because you need to modify your code so that when the view orientation changes, the event that handles this fires.

In Xcode, open the `HelloXcode4_GUIViewController.m` file and look for the following code snippet:

```
-  
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)  
interfaceOrientation {  
    // Return YES for supported orientations  
    return (interfaceOrientation == UIInterfaceOrientationPortrait);  
}
```

Modify the code above to return YES. This is shown below:

```
-  
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)  
interfaceOrientation {  
    // Return YES for supported orientations  
    return YES;  
}
```

When you run the application again, you will notice that your application rotates as the view orientation changes:



Making our Components work together

In this section, we will be discussing how to go about making our `HelloXcode4_GUI` application and its components interact with each other. We want to ensure that when the user enters in their name, and clicks on the **OK** button, the Output will display Welcome to iOS Programming along with the user name.

We will take a look at how to go about connecting our controls via the use of **Outlets** and **Actions**. The controls that we will create are shown in the table below:

DATA TYPE	OBJECT NAME
(UILabel)	lblName
(UITextField)	txtUsername
(UILabel)	lblOutput
(UIButton)	btnOK

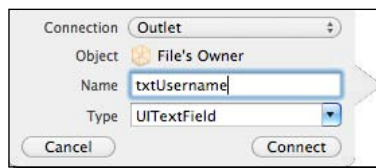
Time for action – binding Control Objects

The way in which we achieve this is to connect our controls via the use of outlets and actions which we will be discussing in this section:

1. Open the `HelloXcode4_GUIViewController.xib` file.
2. Select the **Username** textbox control and hold down the *Ctrl* key while using the mouse to drag this into the `HelloXcode4_GUIViewController.h` interface file and release the mouse button:



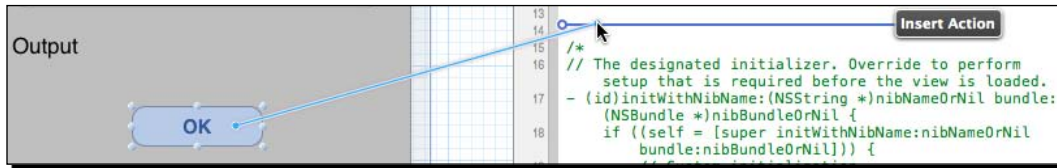
3. Next, once we have dragged the object to which we want to bind within our code, we need to specify its connection type, and provide a name and type as shown in the screenshot below:



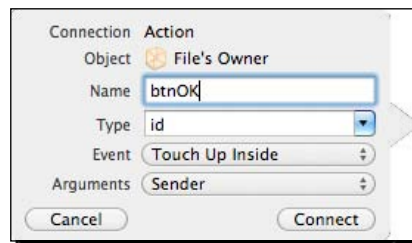
An outlet (`IBOutlet`) is basically a variable by which an object can be referenced. An example of this could be that you have created a field in Interface Builder used to collect the user's name or e-mail address and you have created an outlet for this in your code called `username`. Using this outlet, you would be able to access or change the contents of this field:

```
IBOutlet UILabel *Username;
```

4. Next, we need to create an Action for our **OK** button which will be used to display the greeting message on the screen when it is pressed. Open the `HelloXcode4_GUIViewController.m` implementation file.
5. Select the **OK** button, hold down the *Ctrl* button while using the mouse to drag this into our implementation file and release the mouse button. This is shown in the screenshot below:



6. Next, once we have dragged the object to which we want to bind within our code, we need to specify its connection type, and provide a name and type as shown in the screenshot below. The **Type**, **Event**, and **Arguments** need not be filled in, as these are the default values:



An Action (`IBAction`) on the other hand, is basically a method defined within your code that is called when an event takes place. Objects such as Buttons and Switches can trigger actions when a user performs a task such as touching objects on the screen.

```
-(IBAction)displayName:(id) sender;
```

7. Next, we need to enter in the following code snippet, which will be responsible for displaying the greeting message on the screen when the user enters in their name and the **OK** button is pressed:

```
-(IBAction)btnOK:(id) sender {
    NSString *WelcomeMsg=[NSString alloc] initWithFormat:@"Welcome
    to iOS Programming %@",txtUsername.text];

    lblOutput.text=WelcomeMsg;
    lblOutput.textColor=[UIColor blueColor];
}
```

What just happened?

We created an action button, **btnOK** which when clicked, displays the contents of the text field entered by the user. It formats the output as a string output, sets the color to blue and displays the text. As you can see, by adding this simple method to your view controller, your application will automatically sense and rotate the view for portrait or landscape left orientations.

Format specifiers allow you to manipulate how you would like to represent your data to the screen. These types of specifiers use the standard C format specifiers. In the example above where we are initializing our output message, we use the `%@` specifier which tells the `initWithFormat` function that we are expecting to format the text entered within our `txtUsername` control as `String`.

In Standard C, this could be written as:

```
printf("%s",txtusername.text);
```



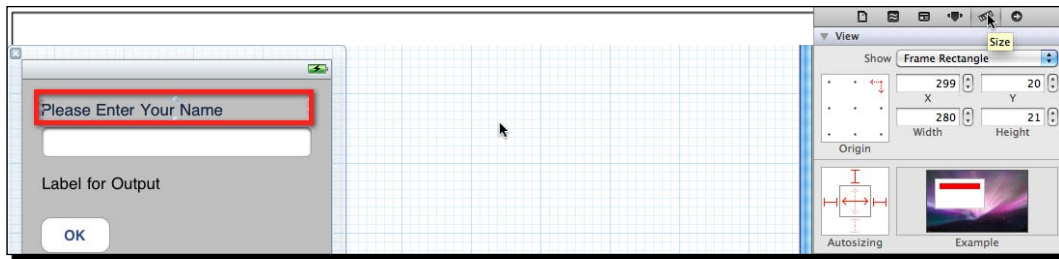
For more information on the format specifiers, please consult Apple's String Programming Guide for Cocoa at the following location: <http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html>.

Time for action – repositioning the Controls

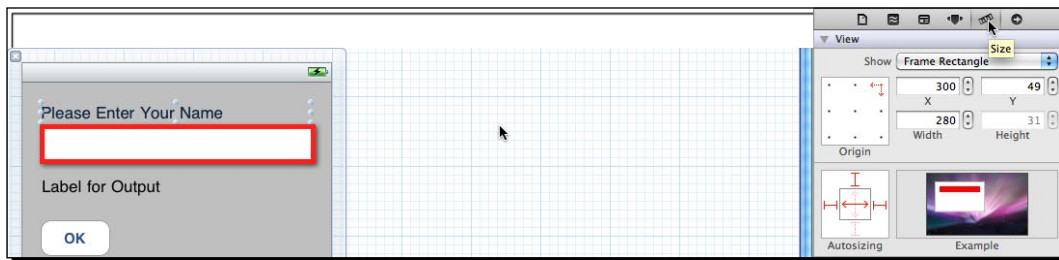
In the previous section, we looked at how to change the orientation of our view using the iPhone Simulator and also observed that the size and positioning of the controls remained. This is not a desirable way to do things in the real-world, as this does not give the user a good experience while using your application. Ideally, you should reposition your controls on the screen so that they change according to the view orientation that the phone is currently in.

In order to reposition the controls within our view, let's go back to our previous example in Interface Builder and follow these steps:

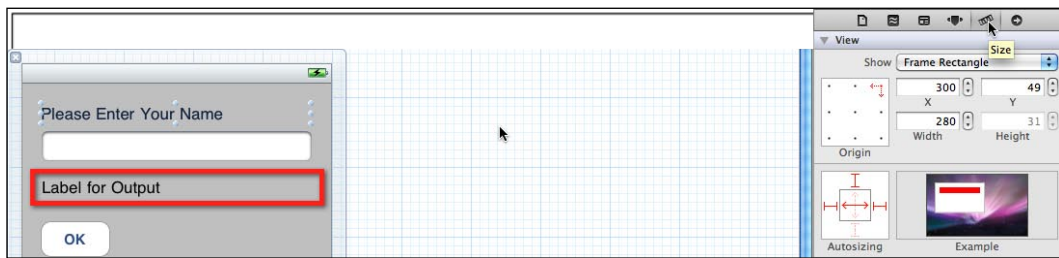
1. Select the **Label** control; select **View | Utilities | Size**.
2. Modify the **Autosizing** attribute of the control as shown in the screenshot below. This will cause the Label control to expand/contract as the view orientation changes. At the same time, the control will anchor to the left, top, or right of the screen:



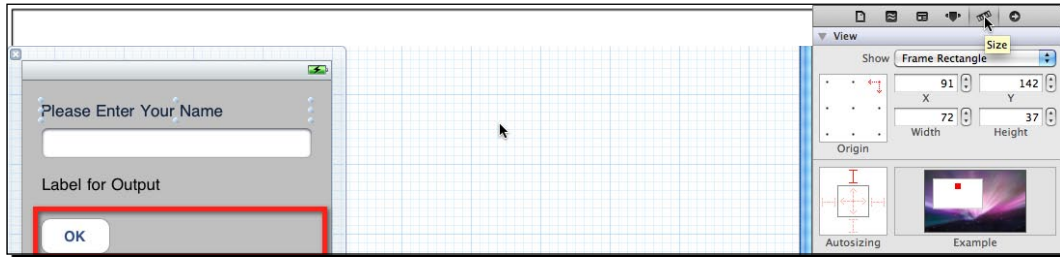
3. Modify the **AutoSizing** attribute for the Text Field control as shown in the screenshot below:



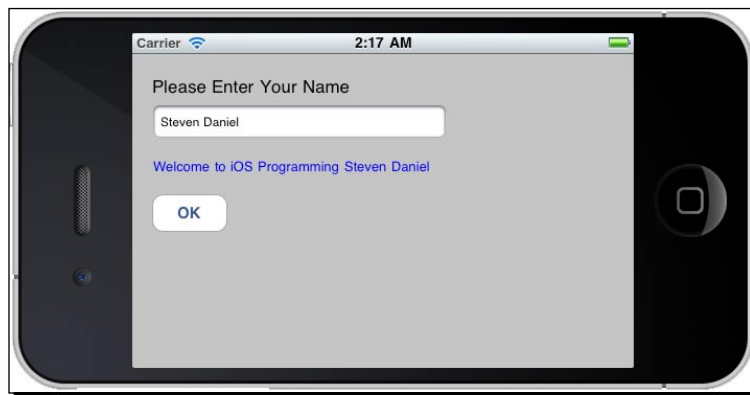
4. Modify the **AutoSizing** attribute for the Label Field control which will be used to display our output, as shown in the screenshot below:



5. Modify the Autosizing attribute for the **Round Rect Button** control which is shown in the screenshot below. You will notice that we are not going to resize the control when the view orientation changes. We just need to anchor this control to the top of the screen:



6. Run the application, and rotate your screen so that the view changes, and you will see that your changes have been applied. This is shown in the screenshot below:



What just happened?

In this section, we took a look at the steps involved in making our controls reposition themselves within the view when the device has been rotated. We also looked at how we are able to use the Autosizing Attribute feature of the object to allow our controls to expand and contract as the view changes.

Enhancing our iPhone application

Now that we have learned how to position controls within our view, we are going to apply some enhancements to our `HelloXcode4_GUI` application. You will notice that when you type into the name field, the keyboard appears, but when you click on the **Done** or **Return** button(s), nothing happens and the keyboard still stays visible making other controls on your view impossible to get to.

In this section, you will be learning about how to address this problem, by hiding the keyboard when the Done or Return buttons are pressed.

When an object processes input, these are called *responders*. In the case of a text field or text view, when it gains first responder status, the keyboard is displayed and will remain on screen until the field gives up, or *resigns* its responder status.

Time for action – hiding the keyboard

In the case of our Username text field, we could resign its first responder status and get rid of the keyboard by adding the following line of code to our `HelloXcode4_GUIViewController.m` file:

```
[txtUsername resignFirstResponder];
```

Calling the **resignFirstResponder** method tells the input object to give up its claim to the input control, hence the keyboard disappears.

There is a second common method for hiding the keyboard in iOS applications through the `Did End on Exit` event of the field. This event occurs when the **return** or **done** keyboard button is pressed.

To add keyboard hiding to our `HelloXcode4_GUI` application, follow these steps:

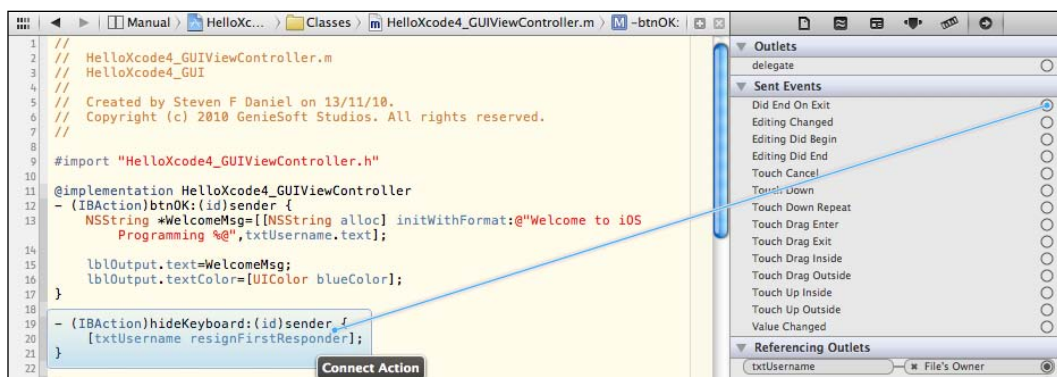
1. Switch back to our application in Xcode and create the action declaration for a method `hideKeyboard` within the `HelloXcode4_GUIViewController.h` file.

```
-(IBAction)hideKeyboard:(id)sender;
```
2. Next, we need to implement the `hideKeyboard` method within the `HelloXcode4_GUIViewController.m` file by adding the following code after the **@implementation** directive:

```
-(IBAction)hideKeyboard:(id)sender {
    [txtUsername resignFirstResponder];
}
```

To connect fields to the `hideKeyboard` method manually, follow these steps:

1. Open the `HelloXcode4_GUIViewController.xib` file which should display within Interface Builder.
2. Select the field, and drag its connector into the `HelloXcode4_GUIViewController.m` file.
3. Create an **IBAction** for the **Did End On Exit** event as shown in the screenshot below:



What just happened?

In this section, we looked at the two different ways in which we can hide the keyboard when the **Done** or **Return** buttons are pressed. We saw that in order to hide the keyboard onscreen; we must send the **resignFirstResponder** message to the object that currently controls the keyboard (such as a text field). If we don't do this, the keyboard is displayed and will remain on screen until the field gives up, or *resigns* its responder status. We also looked at how we can connect up to our text field an event **Did End On Exit** which points to our **hideKeyboard** method.

Have a go hero – enhancing the HelloXcode4 example

Now that you have the basic example working, try improving the user experience a bit. The application needs to be enhanced to allow the user to enter their age, gender, occupation, and location. When the OK button is pressed; it should check the contents of each of these fields to ensure that they have been filled in and display an error message in red text if any of these have not been filled in, and update the background of the control that has not been filled in:

1. Modify the **MainWindow.xib** file to include the additional field objects. Refer to the section *Adding object controls to our View*.
2. Create the necessary outlets. Refer to the section *Making our Components work together* on how to do this.
3. Connect the method to each control to hide the keyboard when the **Done** or **Return** buttons are pressed. Refer to the section *Hiding the Keyboard*.
4. Update the method call to create each of the conditions that performs the check and updates the label.
5. Change the background color of the control that has not been filled in to be green. You will need to make use of the `UIColor` class. Refer to the section *Binding Control Objects*, or the Apple Developer Documentation located at the following: http://developer.apple.com/library/ios/#documentation/uikit/reference/UIColor_Class/Reference/Reference.html.

For more of a challenge, try using the `||` notation instead of separate **if** statements. This will enable you to check each form field in one hit. Once you have that working, you will have a more user-friendly application that provides more information to be entered along with validation on those fields and that provides error trapping.

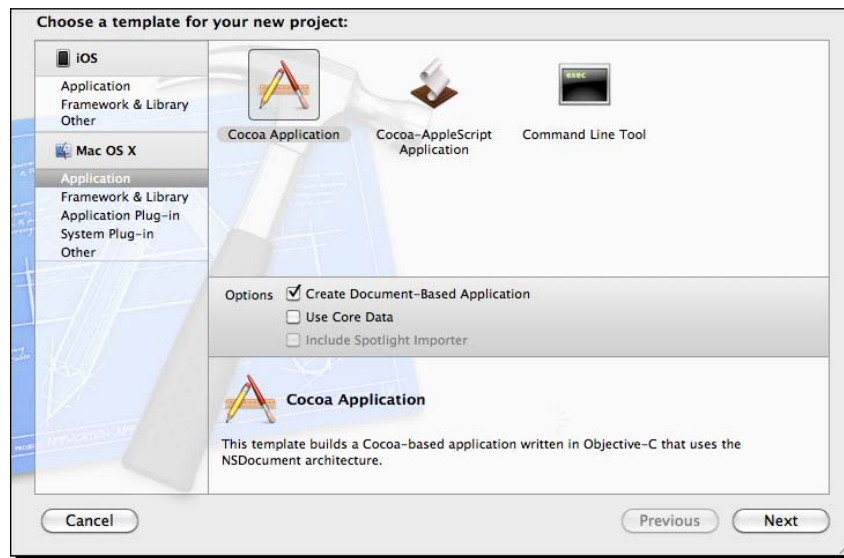
Introducing Document-based applications

Document-Based application applies to the Cocoa environment and can be thought of as a mini-application. Creating these types of applications provides you with the ability to have multiple document windows opened at the same time, allowing you to switch focus between each of them. It provides a framework for generating identically contained, but uniquely composed sets of data that can be stored in files. An example of a document-based application is `TextEdit`.

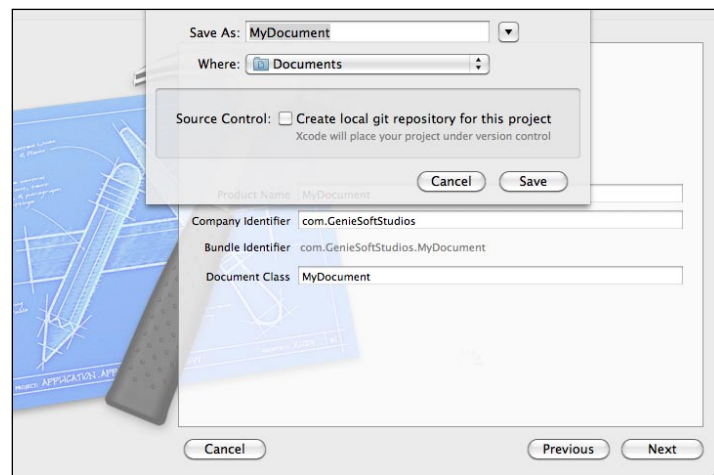
Time for action – creating a Document-based application

It is very simple to create this in Xcode; just follow the steps listed below:

1. Under the Mac OS X section header, select **Application** and then select the Cocoa Application icon.
2. Under the options pane, ensure that you have checked **Create Document-Based Application**; if this is not selected, Cocoa will create a single-windowed application:

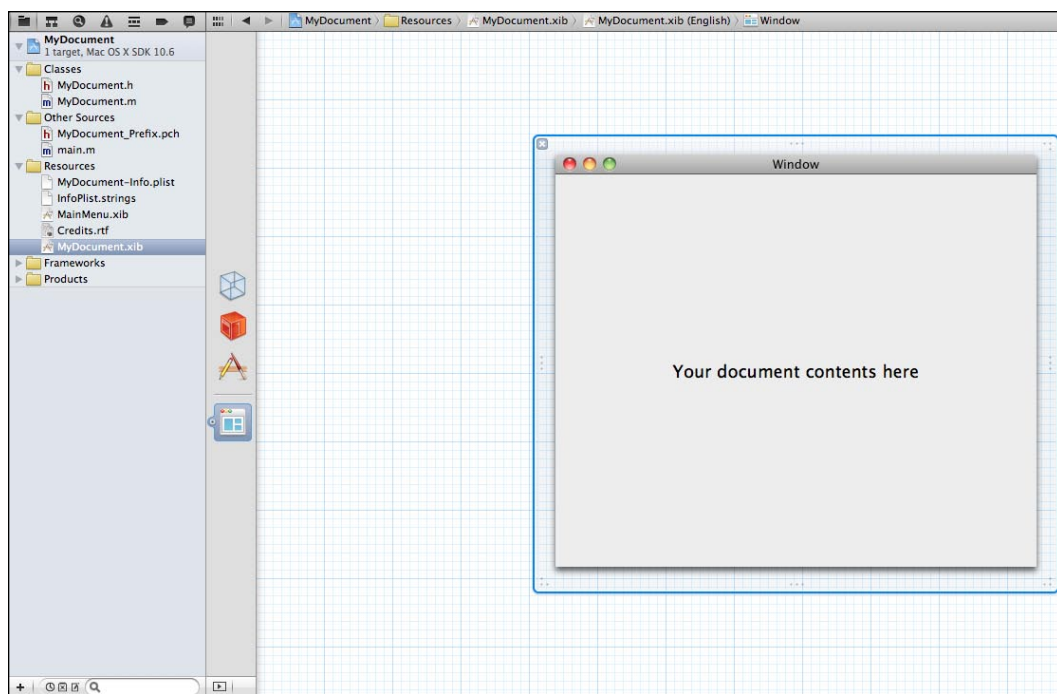


3. Click the **Next** button to proceed to the next step, and enter a name for your project:

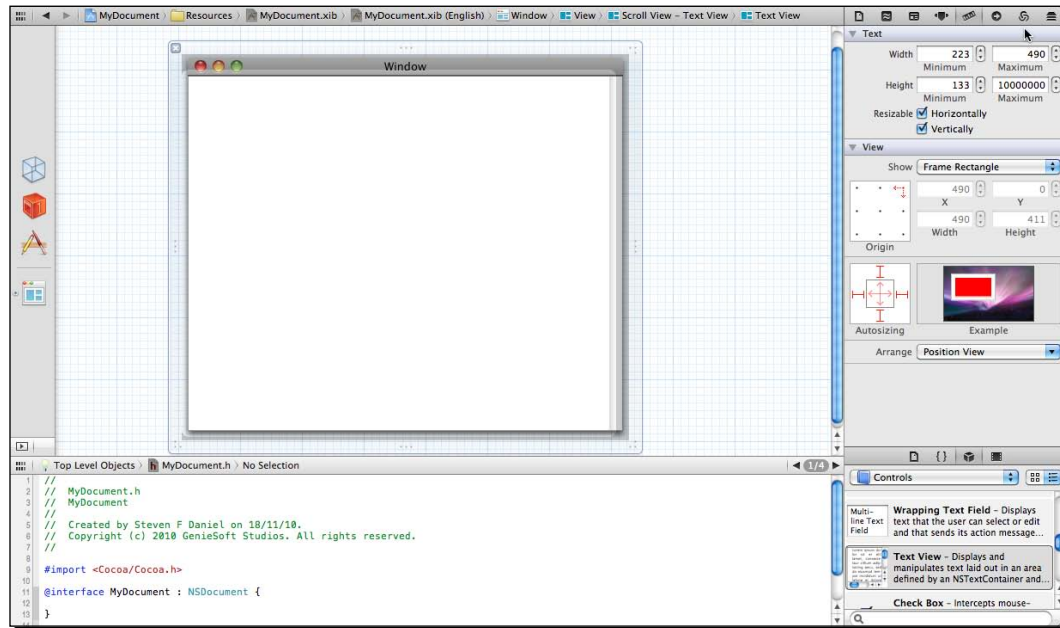


4. When you have entered a name for your project, click on the **Save** button. Your project will be saved in the folder specified and the Xcode workspace will then be displayed.

Let's start to see what the project wizard has created for us; the important files to take note of are the `MyDocument.m`, `MyDocument.h`, and `MyDocument.xib`. Start by double-clicking on the `MyDocument.xib` file; you will notice that there is already a Text Field that was created containing the text *"Your document contents here"*. Just delete that, as we will be creating a new one for us to use:

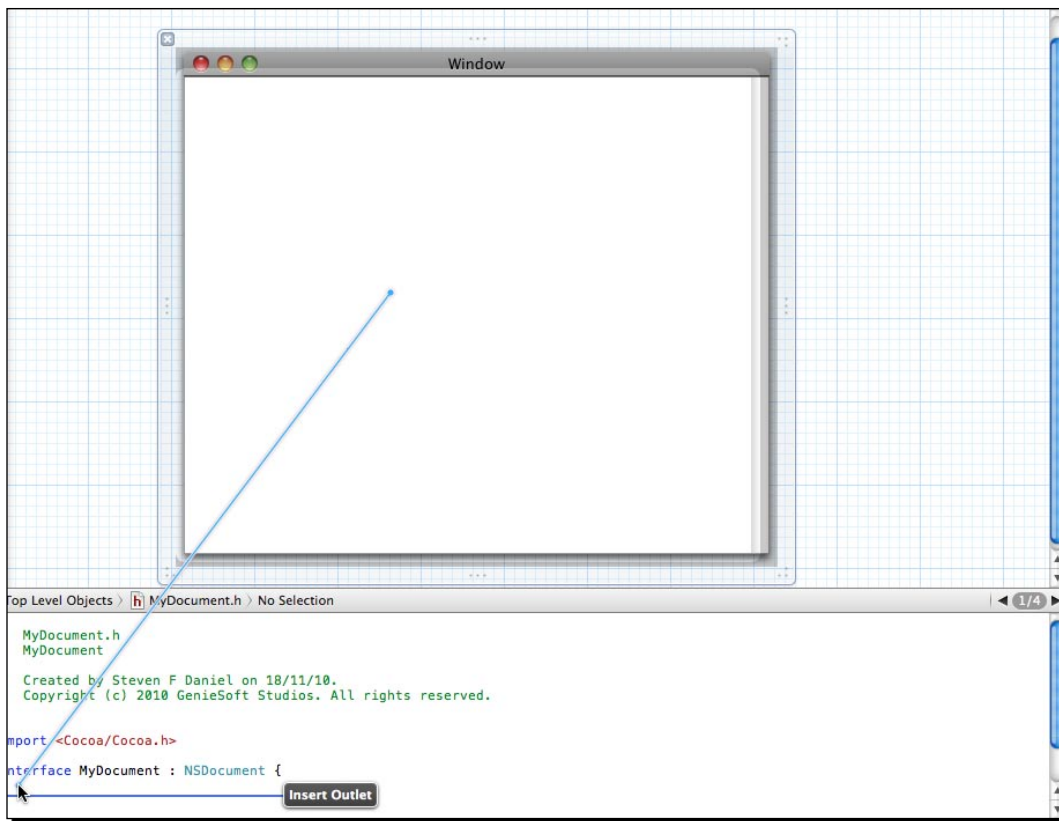


From the Object Library pane, drag and drop a Text View control and use the automatic guidelines to resize the control to fill most of the window by using the Autosizing settings, and enabling all four of the I-Bars and both the horizontal and vertical arrows. This is shown in the screenshot below:

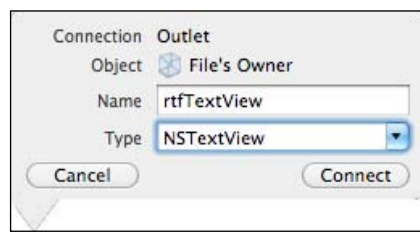


Before we can use our Text View control, we need to create an Outlet for this, in order for us to Load and Save documents. In the following section, we will look at how we go about creating Outlets, as well as adding properties, variables, and methods.

In the following screenshot, we need to create a connection within our `MyDocument.h` interface file, so that it points to our document Text View control. This is shown in the following steps:



1. Drag the connection of the Text View control into the interface header file of `MyDocument.h`. A Connection dialog will be displayed as shown in the screenshot below.
2. Next, provide a name for our Text View control as we will be referencing this control when we come to load and save our files:



What just happened?

In this section, we learned about Document-based applications and how we are able to bind controls and create Outlets for them so that they can be referenced within the code.

Creating Document-based applications enables you to do the following:

- ◆ They allow you to create new and open existing documents that are stored in files
- ◆ Save documents under user-designated names and locations
- ◆ Revert to saved documents
- ◆ Close documents (usually after prompting the user to save edited documents)
- ◆ Print documents and allow the page layout to be modified
- ◆ Represent data of different types internally
- ◆ Monitor and set the document's edited status and validate menu items
- ◆ Manage document windows, including setting the window titles
- ◆ Handle application and window delegation methods (such as when the application terminates)

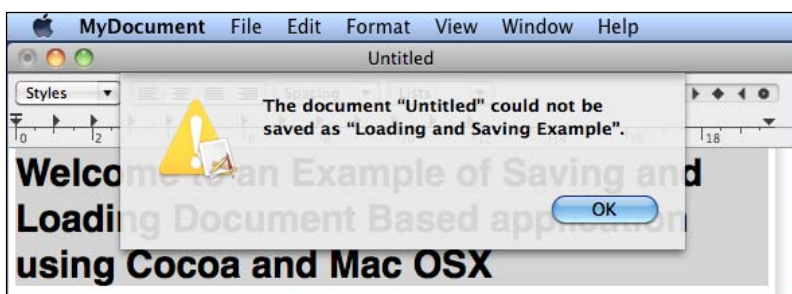


If you would like to read up a bit more about Document-Based applications, check out the *Apple Developer Connection* website at the following website location:
<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Documents/Documents.html>.

File saving and loading

When you try to run your application, you will see that you have the capability of a mini-word processor and you have the ability to type, copy, paste, apply formatting to fonts and colors, as well as showing the ruler and setting tabs.

When you try to save your document, it appears that your application doesn't know how to save your document. You receive the following error message which is shown in the screenshot below:



Time for action – implementing file saving and loading

In order to make our application have the capability to Save and Load, follow these steps:

1. Modify the `MyDocument.h` header file to include the following piece of highlighted code:

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

@interface MyDocument : NSDocument {

    IBOutlet NSTextView *rtfTextView;
    NSAttributedString *docString;
}

@property(retain) NSTextView *rtfTextView;
@property(retain) NSAttributedString *docString;

@end
```

2. The next stage is to implement the various methods in the `MyDocument.m` implementation file. You will find that we already have the method stubs defined within this file, so we just need to fill these in. The method called `windowControllerDidLoadNib` is the method which gets run when the user interface is loaded, and its purpose is to populate the Text View control. Add the following highlighted code into this method:

```
- (void)windowControllerDidLoadNib:(NSWindowController
*)aController {
    [super windowControllerDidLoadNib:aController];

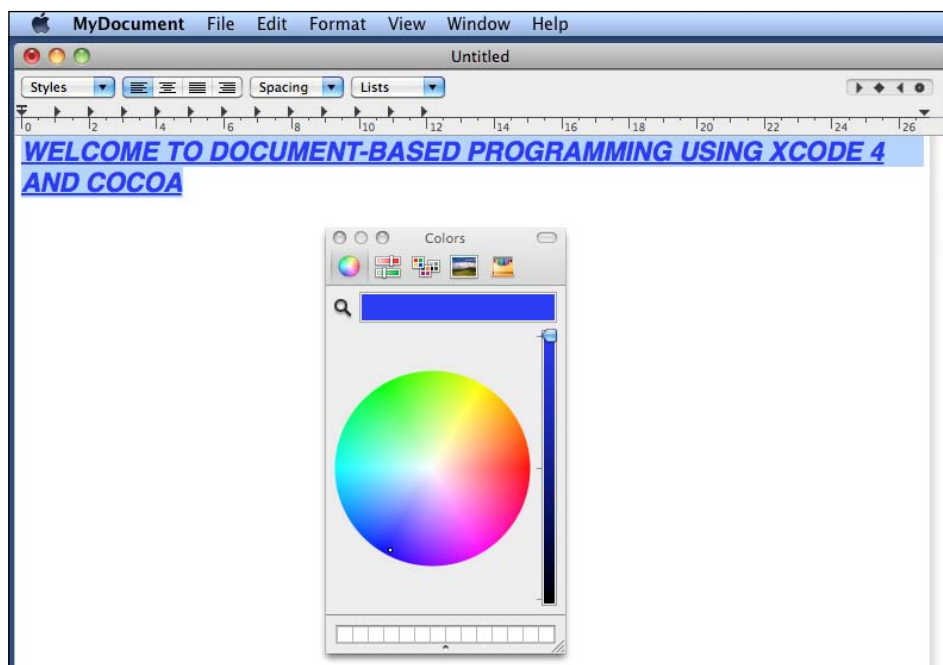
    // Add any code here that needs to be executed once the
    // windowController has loaded the document's window.
    if (self.docString != nil){
        [[rtfTextView textStorage] setAttributedString:self.docString];
    }
}
```

3. We are now going to add the code for the Reading and Writing. Locate the `dataOfType` method and add the following highlighted code:

```
- (NSData *)dataOfType:(NSString *)typeName error:(NSError
**)outError {
    NSData *rtfData;
    self.docString=rtfTextView.textStorage;
    rtfData=[NSArchiver archivedDataWithRootObject:self.docString];
    return rtfData;
}
```

4. Next, we are now going to add the code for Reading the contents back into the document. Locate the `readFromData` method and add the following highlighted code:

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName
error:(NSError **)outError {
    NSAttributedString *tempString=[NSUnarchiver
        unarchiveObjectWithData:data];
    self.docString=tempString;
    return YES;
}
```
5. Now that you have added all of the relevant code to the `MyDocument.m` implementation file, you are ready to run your application. A screenshot of the final output is displayed below:



What just happened?

In this section we declared outlets, variables, and methods that we implemented into our `MyDocument.m` file. We then used the `NSAttributedString` class to help us manage the string which we needed to get and set the string contents from the `TextView` Control. So that is why we needed to create an `Outlet` for this.

We then set the value of the `textStorage` property of the Text View control to be the value of the attributed string property of the current window object. We also defined two datatypes `rtfData` and `docString`, that will be used to store the document contents using the `NSArchiver` class. We finally added code to our `readFromData` method to read the contents of the file back into the document window by using the `NSUnarchiver` class.

Pop quiz – Actions and Rotatable Interfaces

1. Which iOS Orientation constant is used to set the device to Portrait?
 - a. `UIInterfaceOrientationPortraitUpsideDown`.
 - b. `UIInterfaceOrientationLandscapeLeft`.
 - c. `UIInterfaceOrientationLandscapeRight`.
 - d. `UIInterfaceOrientationPortrait`.
2. Which of the following statements is true about Document-based applications?
 - a. Saves documents under user-designated names and locations.
 - b. Prints documents and allow the page layout to be modified.
 - c. Represents data of different types internally.
 - d. Manages document windows, including setting window titles.
 - e. Represents data of different types internally.
 - f. Allows you to perform Hexadecimal calculations.
3. When creating an Action to a button, what Event do we use?
 - a. Touch Up Inside.
 - b. Touch Down.
 - c. Touch Cancel.
 - d. Touch Drag Enter.

Summary

In this chapter, we covered the Interface Builder (IB) application, and how to go about creating a very simple application which interacted with the user to display some text to the screen using outlets and actions. We also got an insight into the iOS Application-Life-Cycle, and what happens when an application is loaded by the user.

We also enhanced our iPhone application and added some rotation and resized the controls. We also looked into the various ways of hiding the keyboard by resigning responders.

Now that we've learned about how to go about creating an application using Interface Builder, we are ready to start to focus on the iOS 4 Xcode Frameworks.

In the next chapter, we will take a look at the iOS frameworks and where these are located. We will create a simple Pop Quiz database application which will be making use of the Core Data Frameworks, as well as creating an application to play video and audio. We will also get acquainted with the Core Location and Map Kit Framework improvements, as well as the new framework APIs.

4

Working with the Xcode Frameworks

In this chapter, we will take a look at the different Xcode (Cocoa) development frameworks and where these are located. Frameworks provide you with a hierarchical directory that encapsulates shared resources, such as a dynamic shared library, xib files, image files, header files, and reference documentation in a single package. Multiple applications are then able to use all of these resources simultaneously. The system loads these into memory as needed and shares one copy of the resource among all applications whenever possible.

In this chapter, we will:

- ◆ Introduce the different sets of Xcode Frameworks and their locations
- ◆ Take an insight into the Core Data Frameworks and build a simple database application to save and retrieve data
- ◆ Learn how to play an audio file using the AV Foundation Frameworks
- ◆ Learn how to play a movie using the Media Player Frameworks
- ◆ Understand what comprises the Core Location Framework
- ◆ Learn about the Map Kit Framework and the improvements
- ◆ Learn how to go about building a simple Geographical Application

We have got quite a bit to cover, so let's get started.

Introducing the Frameworks

You may be wondering, what exactly are frameworks? Well, to sum it up, frameworks are basically a group of code libraries that provide specific functionalities that save you time in building common features yourself.

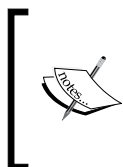
In previous chapters, you have already been using Apple's frameworks; in fact, every time you use your Mac you make use of these frameworks. Most software that you use on your Mac uses these frameworks, including Apple's Mac OS X operating system.

Frameworks are an integral part of Cocoa, which is an object-oriented environment which you use to build Mac OS X software—including Mac and iOS applications. Cocoa consists of the following main components:

- ◆ The Xcode Tools
- ◆ The Core Frameworks: This includes the Foundation and Application Kit
- ◆ Additional Frameworks: WebKit, Core Data, and Core Animation, and so on

The above frameworks are highly optimized code libraries which provide you with a straightforward and consistent interface to the features within the Mac OS X operating system. If you wanted to implement a list box, command button, or web browser window; these would operate the same way as those used in such programs as Safari, iTunes, or Mail.

The Core Foundation frameworks are the fundamental Cocoa frameworks that are used in all Cocoa applications and inherit all features from the **NSObject**.



NSObject was covered in *Chapter 1, Introducing Xcode 4 Tools for iPhone Development* under the section *Objective-C Classes* or refer to the Apple Developer Documentation at: http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/Reference/Reference.html

The main core frameworks which are supplied by Apple are located within the `/System/Library/Frameworks` folder. If you take a look within this folder, you will notice that it contains folders for every framework.

The table below describes each of the frameworks which are available to iOS devices. If you are interested in taking a look at these frameworks, you can find these located in the `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/<iOS_SDK>/System/Library/Frameworks` directory.



<Xcode> : The path to where your Xcode installation directory is located.
<iOS_SDK> : The specific SDK version that your project is targeting.

FRAMEWORK NAME	DESCRIPTION
<code>AddressBook.Framework</code>	This framework provides access to the centralized database for storing user contact information.
<code>AddressBookUI.Framework</code>	This framework provides the User Interface to display the contacts stored within the Address Book database.
<code>AudioToolBox.Framework</code>	Provides low-level C APIs for audio recording and playback as well as managing the audio hardware.
<code>AudioUnit.Framework</code>	Provides the interface for iOS supplied audio processing and plug-ins within your application.
<code>AppKit.Framework</code>	Provides you with all the objects you need to implement your graphical, event-driven user interface: windows, panels, buttons, menus, scrolling views, and text fields. This framework handles all the details for you as it efficiently draws on the screen, communicates with hardware devices and screen buffers, clears areas of the screen before drawing, and clips views.
<code>AVFoundation.Framework</code>	Provides low-level C APIs for audio recording and playback as well as managing the audio hardware.
<code>CFNetwork.Framework</code>	Provides access to the network features (services and configurations) such as HTTP, FTP, and Bonjour services.
<code>CoreAudio.Framework</code>	Declares data types and constants used by other Core Audio Interfaces.
<code>CoreData.Framework</code>	Provides a generalized solution for object graph management from within your application.
<code>CoreFoundation.Framework</code>	Provides abstraction for common data types, Unicode Strings, XML, URL Resources, and so on.
<code>CoreGraphics.Framework</code>	Provides C-based APIs for 2D rendering. This is based on the Quartz drawing engine.
<code>CoreLocation.Framework</code>	Provides location-based information using a combination of GPS, Cell ID, and Wi-Fi networks.
<code>ExternalAccessory.Framework</code>	Provides a way to communicate with accessories.
<code>Foundation.Framework</code>	Provides the foundation classes for Objective-C, such as the NSObject, basic data types, and operating system services, and so on.

FRAMEWORK NAME	DESCRIPTION
<code>GameKit.Framework</code>	Provides networking capabilities for games, and is used for peer-to-peer connectivity and in-game voice features.
<code>IOKit.Framework</code>	Provides capabilities for driver development.
<code>MapKit.Framework</code>	Provides an embedded map interface for your application.
<code>MediaPlayer.Framework</code>	Provides facilities for playing movies and audio files.
<code>MessageUI.Framework</code>	Provides a view-controller-based interface for composing e-mail messages.
<code>MobileCoreServices.Framework</code>	Provides access to standard types and constants.
<code>OpenAL.Framework</code>	Provides an implementation of the OpenAL specification.
<code>OpenGLES.Framework</code>	Provides a compact and efficient subset of the OpenGL API for 2D and 3D drawing.
<code>QuartzCore.Framework</code>	Provides ability to configure animations and effects and then renders those effects via hardware.
<code>Security.Framework</code>	Provides the ability to secure your data and control access to software.
<code>StoreKit.Framework</code>	Provides support for your applications to handle in-app purchases.
<code>SystemConfiguration.Framework</code>	Provides the ability to determine network availability and state on the device.
<code>UIKit.Framework</code>	Provides the fundamental objects for managing an application's UI.

Using Frameworks and APIs in iPhone development

For the rest of this chapter, we will be creating a variety of applications that will be using the Cocoa Frameworks, to create very capable, rich-media applications, to handle playing movie and audio files, and navigational applications.

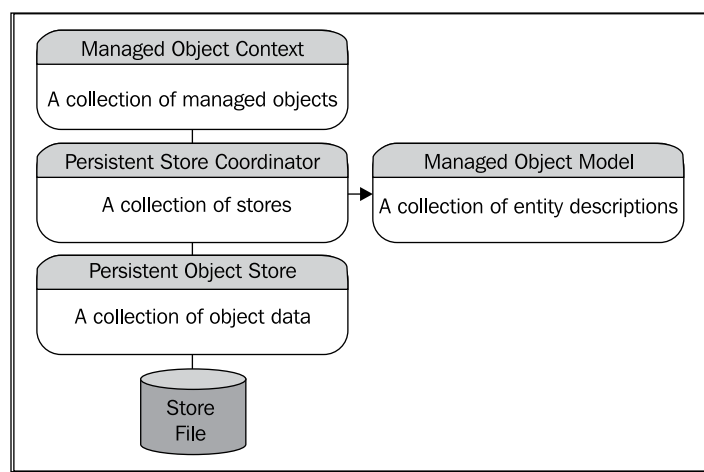
Core Data Frameworks

The Core Data framework is a framework which manages where data is stored, how this data is stored, how it is cached, and how it handles memory management. This framework can be described as a *"Schema-driven object graph management and persistence framework"* and was first ported to the iPhone from Mac OS X, and came as part of the iPhone 3.0 SDK release.

So what exactly is the Core Data framework? If you are familiar with the Entity-Framework which is available in Microsoft .NET, this is of a similar nature. The Core Data framework is an abstraction layer which sits on top of an **SQLite database**, and **enables developers to easily** implement data-centric applications by modelling your data storage around entities (**classes**) that contain the relationships between them.

We are only going to scratch the surface on what is available with Core Data. Since this is such a large topic, there are many books as well as online resources made available which go into more depth than we will here. I hope to give you a general overview of what Core Data is and how to implement it.

The following image shows the simplest and most common configuration of the stack. Objects that you usually work directly with are located at the top of the stack, as well as the managed object context and the managed objects that it contains:



Just to give you an insight of the three main management object models that the Core Data Framework contains, I have described these below:

OBJECT MODELS	DESCRIPTION
Managed Object Context	This associates the in-memory objects with their associated in-storage counterparts.
Managed Object	This is the in-memory representation of a data-model object and is saved to storage in a persistent store (table).
Managed Object Model	This is the object-relational schema, which contains the entity descriptions that are required to build the managed objects.

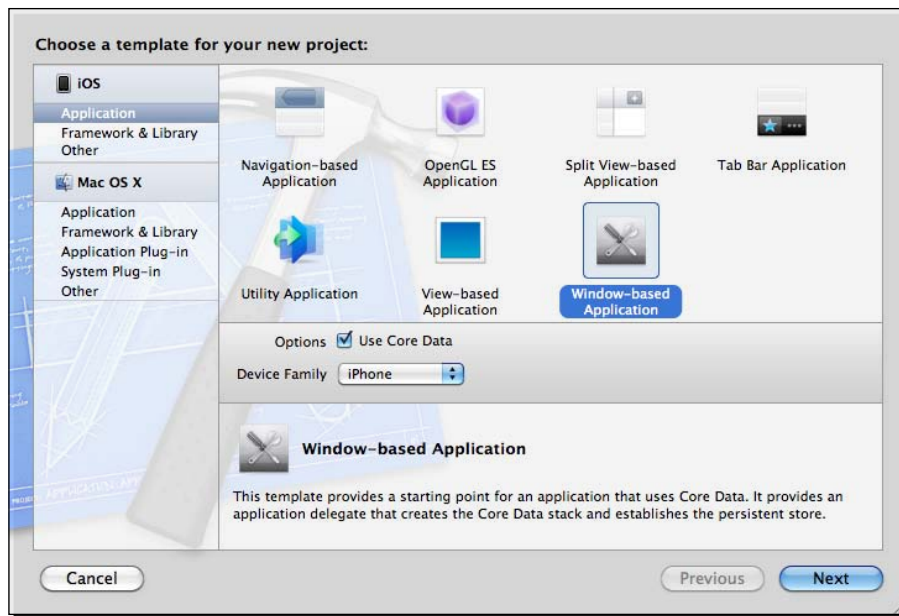
Building a simple database application

In this section, we will look at building a simple iPhone application using Core Data to allow the user to enter their name, date of birth and gender, and save and retrieve this information from the database.

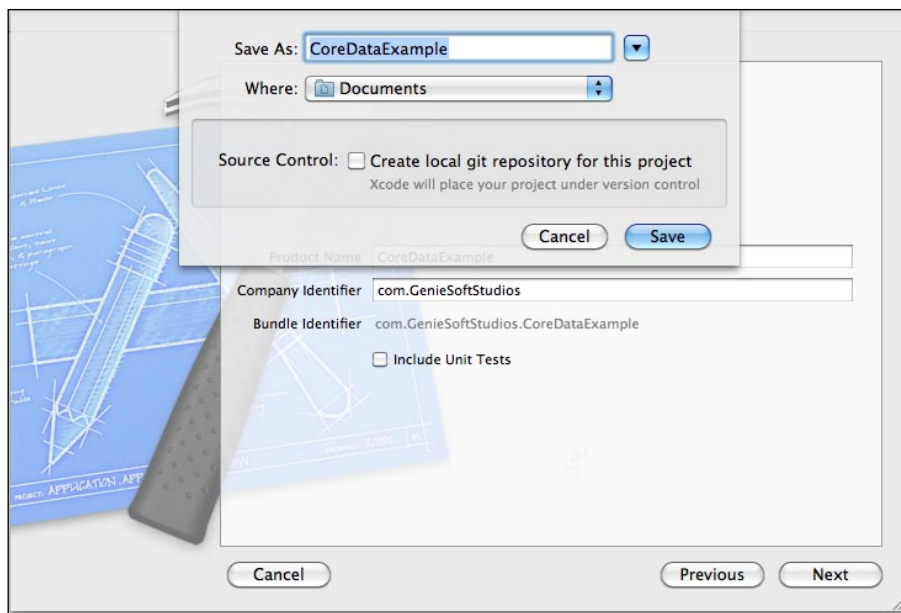
Time for action – creating the Core Data application

Before we can proceed, we first need to create our "*CoreDataExample*" project. To refresh your memory, you can refer to the section that we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section, *Creating your first iPhone application*.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Window-based Application** template from the list of available templates.
4. Select **iPhone** from under the **Device Family** dropdown.
5. Ensure that you have checked **Use Core Data** from under the **Options** section:



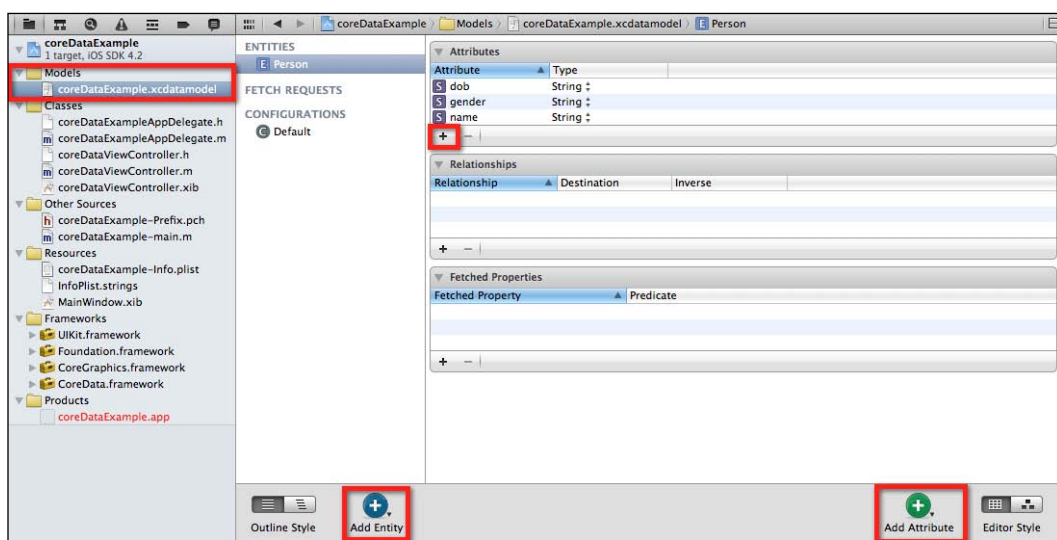
6. Click on the **Next** button to proceed to the next step in the wizard.
7. Enter in **CoreDataExample** and then click on the **Next** button to proceed to the next step in the wizard:



8. Specify the location where you would like to save your project.
9. Click on the **Save** button to continue and display the Xcode workspace environment.

There is not a lot of setting up to do as the **Use Core Data** option which we checked when we created our project, has automatically set up some important variables and has also created the files for us in our project. We don't need to include the `CoreData.Framework`, as this has been automatically added for us.

The **CoreDataSample.xcdatamodel** file is where we will be defining the database schema for our SQLite database and this file is located under the **Models** section within our project workspace:



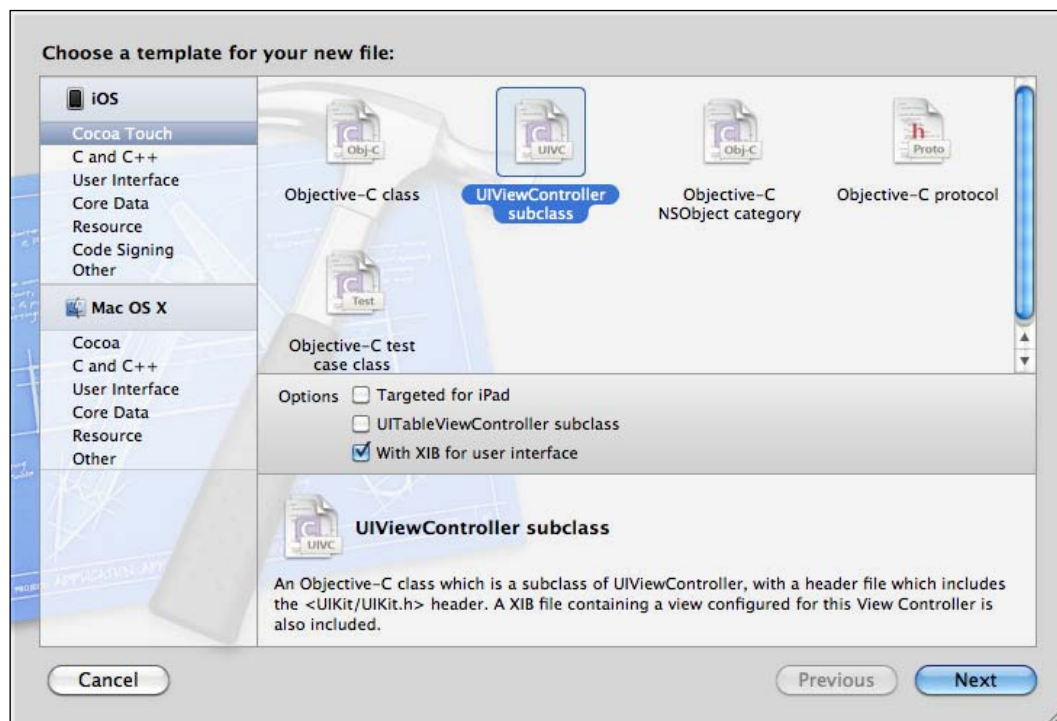
Follow these steps to create a new entity and then to add attributes to it, which will be used to represent the data that is going to be stored:

1. Click on the **+ Add Entity** button, located in the bottom left-hand corner of the entity panel and name this entity **Person**.
2. Click on the **+ Add Attribute** button located in the bottom right of the entity panel, or similarly from the **Attributes** pane and enter **name** for the attribute.
3. Change the attribute type to **String** from the type selection box.
4. Repeat steps 2 and 3 to add the remaining attributes for **dob** and **gender**.
5. Save your project using **File | Save**, as we are done defining our database table schema.

We are now ready to start writing the code to save and retrieve our data.

What we now need to do is to create our own view controller, that will be used to handle the saving and loading of the data as well as containing our user interface:

1. From the `Classes` folder in your project, *ctrl + click* on the folder, and select *New File....* The following screen is displayed as shown below:

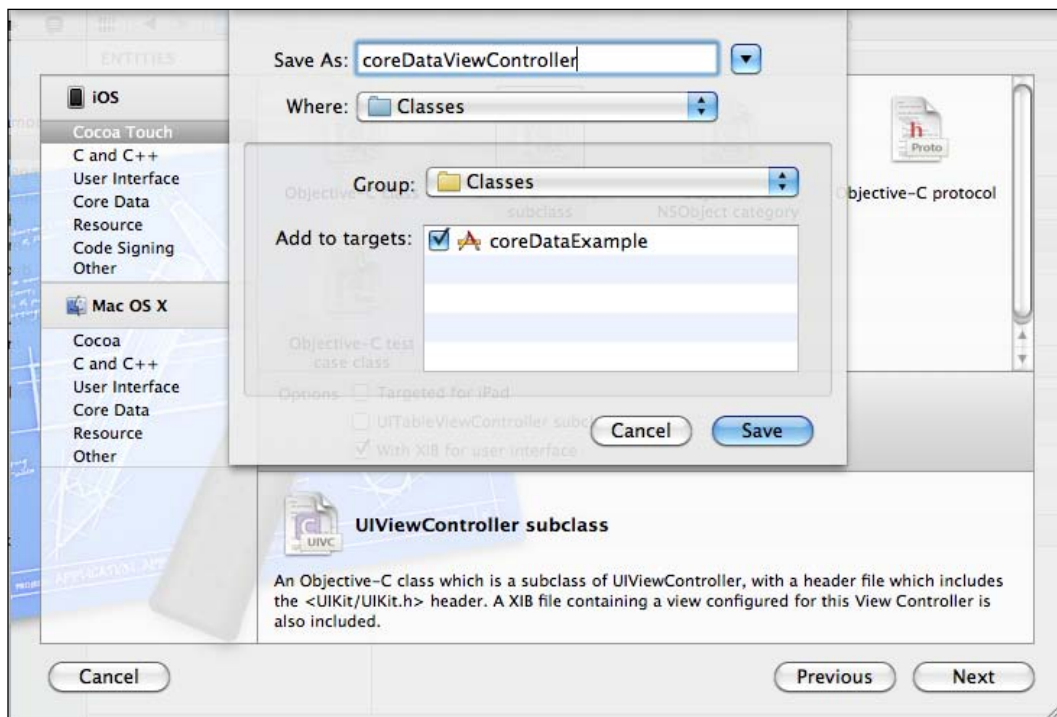


2. Select the **UIViewController subclass** template from the list of available templates.
3. Ensure that the **With XIB for user interface** option is selected from under the **Options** section.
4. Click on the **Next** button to proceed to the next step of the wizard.

5. Enter **coreDataViewController** as the name of the file to create, and then click on the **Save** button.



The **Source Control: Create local git repository for this project** is not checked by default.



Now that we have added our view controller class to our application, our next task is to modify our application delegate and to make this the main root view controller:

1. Open the `CoreDataExampleAppDelegate.h` interface file, located within the `Classes` folder of your project and add the following highlighted code as shown in the code snippet below:

```
@class coreDataViewController;  
  
@interface CoreDataExampleAppDelegate : NSObject  
    <UIApplicationDelegate> {  
    coreDataViewController *viewController;  
    }  
}
```

```
@property (nonatomic, retain) IBOutlet CoreDataViewController
    *viewController;

@end
```

What we have done in the code above is that we have declared an instance to our view controller so that we can reference this object. What we need to do now is to modify the call to the method `applicationDidFinishLaunching`.

2. Open the `CoreDataExampleAppDelegate.m` interface file, located within the `Classes` folder, then modify your code module to include the additional highlighted references as shown in the code snippet below:

```
#import "coreDataExampleAppDelegate.h"
#import "CoreDataViewController.h"

@implementation CoreDataExampleAppDelegate

@synthesize window, viewController;

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

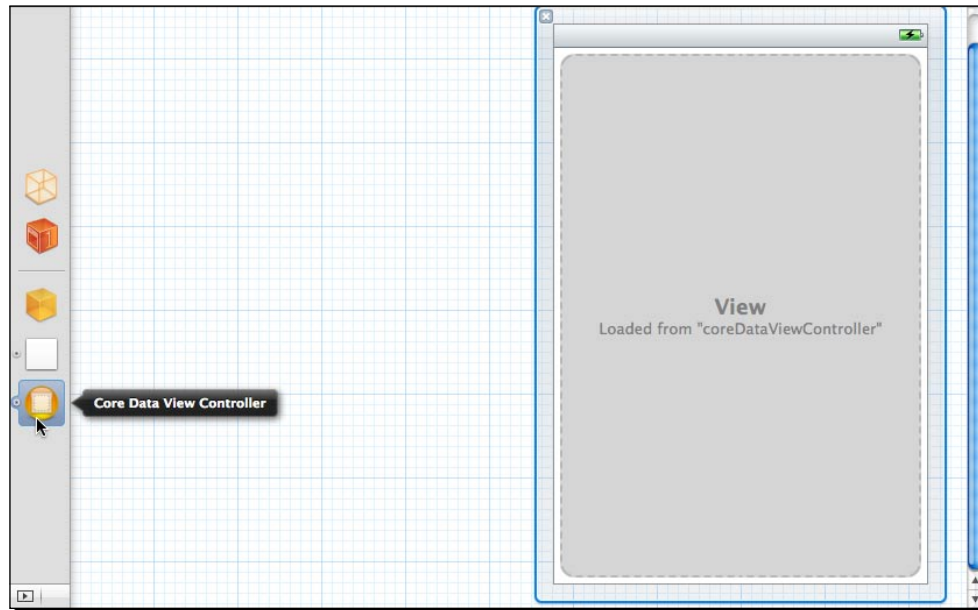
    // Override point for customization after application launch.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
    return YES;
}
```

What we have added into our `coreDataExampleAppDelegate.m` implementation file is a reference to our view controller so that it can be added to our main window when it has finished launching. We needed to include a reference to our `coreDataViewController.h` interface so that we can synthesize our accessibility objects for the `viewController` object.

Our next step is to add and associate a new view controller object with the `MainWindow.xib` file and then connect it to our application window delegate class:

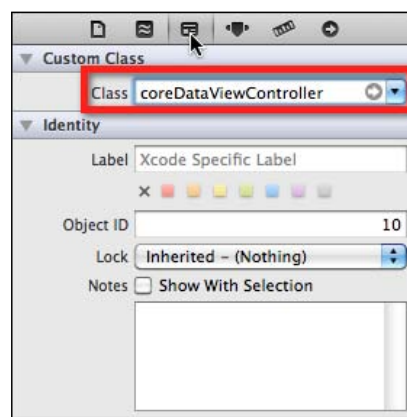
1. Select the `MainWindow.xib` file which is located under the `Resource` node.
2. Drag and drop a View Controller object from the Xcode Object Library onto the `MainWindow.xib` window.

If you have done this correctly, you should see a new View Controller appear under the existing objects as shown in the screenshot below:

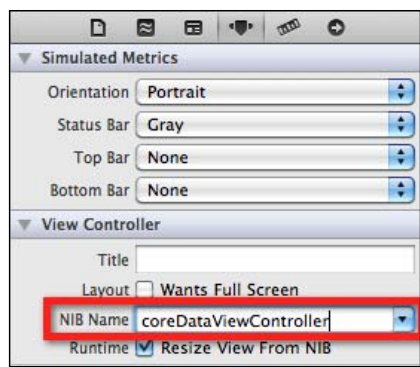


What we now need to do is update the class value for our ViewController to point to our `coreDataViewController` class which we created previously. This will contain our form objects which we will be creating later on:

1. With the View Controller currently selected, navigate to **View | Utilities | Identity Inspector**, and then change the **Custom Class** value to read `coreDataViewController`:

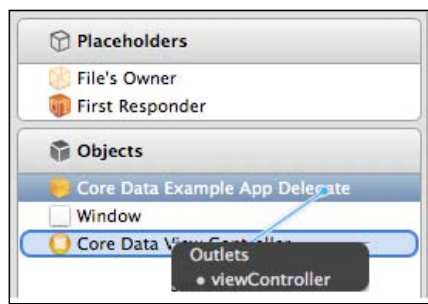


- Next we need to update the NIB Name for our view controller. This can be achieved by selecting the Attributes page in the inspector window, or **View | Utilities | Attributes Inspector** and then assign **coreDataViewController** as the **NIB** file:



Our final task is to establish the connection between the application delegate and the view controller.

- Click on the **coreData Example App Delegate** item in the `MainWindow.xib` window and drag this onto the **core Data View Controller** which will result in a blue line appearing.
- Upon releasing the mouse button, select the **ViewController** item from the menu. This is shown in the screenshot below:



Now that we have associated and bound our View Controller to our application, we are ready to add some actions and outlets to the class.

For this example, we will be building a Graphical User Interface to demonstrate the use of the Core Data Framework. This will accept a Name, Date of birth, and Gender. We will need to create events which will handle the saving and loading:

1. Open the `coreDataViewController.h` interface file located within the `Classes` folder, then modify your code module to include the additional highlighted references as shown in the code snippet below:

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

@interface coreDataViewController : UIViewController {
    IBOutlet UITextField *Name;
    IBOutlet UITextField *DOB;
    IBOutlet UITextField *Gender;
    IBOutlet UILabel *recordsFound;
}

@property (nonatomic, retain) IBOutlet UITextField *Name;
@property (nonatomic, retain) IBOutlet UITextField *DOB;
@property (nonatomic, retain) IBOutlet UITextField *Gender;
@property (nonatomic, retain) IBOutlet UILabel *recordsFound;

- (IBAction)saveData:(id)sender;
- (IBAction)searchData:(id)sender;
- (IBAction)clearData:(id)sender;

@end
```

What we have added into our `coreDataViewController.h` interface file, is that we have set up what fields our form will contain, as well as setting up properties to these objects so we can reference these when we come to Save and Load our data. We have also declared the methods within our header file to avoid warnings at compile time.

2. Next, open the `coreDataViewController.m` implementation file located within the `Classes` folder, then modify your code module to include the additional highlighted references as shown below:

```
#import "coreDataViewController.h"
#import "coreDataExampleAppDelegate.h"

@implementation coreDataViewController
@synthesize recordsFound, Name, DOB, Gender;

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

```
- (IBAction)searchData:(id)sender {  
}  
  
- (IBAction)clearData:(id)sender {  
}
```

As well as declaring our **IBOutlet**s which we will be using on our form, we need to add an **@synthesize** directive which will allow us to use them in our `coreDataViewController.m` implementation file.

Our final part will be to create the user interface design, bind those objects to the outlets which we created within our `CoreDataViewController.h` interface file, and then create the necessary code for our events. We are now going to add our `UITextField`, `UIButton`, and `UILabel` objects. To achieve this follow these simple steps:

1. Open the `coreDataViewController.xib` from within our `CoreDataExample` project.
2. From the Object Library, select and drag a (`UIToolBar`) Toolbar control to the view and place it to the top of our View Controller.
3. From the Object Library, select and drag a (`UIButton`) Button control and drag it to the top left-hand position of our toolbar. From the Attributes Inspector window, set the caption property of the control to **Save** and the Style of the button to be **Bordered**, and set its identifier to be **Refresh**.
4. Repeat the same process as outlined in the previous step to add the Search and Save buttons, and assign their styles accordingly as shown below:

Name: Search	Style: Bordered	Identifier: Search
Name: Save	Style: Bordered	Identifier: Save
5. Now, we need to select and drag a (`UILabel`) Label control and drag it to your form and set its Title to display the text Name.
6. Select and drag a (`UITextField`) Textbox control and drag it to your form, and position it next to your Name Label.
7. Repeat steps 5-6 to add the remaining two fields for the Date of Birth and Gender.

8. Finally, select a (**UILabel**) label control and drag this to your view, position it somewhere underneath the Gender field. This is just for display purposes:



When the user clicks on the **Save** button, we call our `saveData` method. In order for this to work, we must implement the code to obtain the managed object context and create and store the managed objects containing the data that has been entered by the user.

In our `CoreDataViewController.m` implementation file, locate the `saveData` method and add the following code as shown below:

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

    CoreDataExampleAppDelegate *appDelegate = [[UIApplication
        sharedApplication] delegate];
    NSManagedObjectContext *context = [appDelegate
        managedObjectContext];
    NSManagedObject *newPerson;

    newPerson = [NSEntityDescription
        insertNewObjectForEntityForName:@"Person"
        inManagedObjectContext:context];

    [newPerson setValue:Name.text forKey:@"name"];
    [newPerson setValue:DOB.text forKey:@"dob"];
    [newPerson setValue:Gender.text forKey:@"gender"];

    Name.text = @"";
```

```

        DOB.text = @"";
        Gender.text = @"";

        NSError *error;
        [context save:&error];

        recordsFound.text = @"Details have been saved to the database.";
    }

```

What we are doing in the above code is using our application delegate instance to identify the managed object context. We then use this context object to create a new managed object using the **Person** entity item, and then use the `setValue` method of the managed object to store the name, date of birth, and gender fields. Finally, we call the context's `save` method to save this data.

In order to give the user the ability to search for a person's name, we need to implement the code for `searchData()` method. We will need to do the same as we did in the `saveData()` method, but with one exception, that being to ensure that only objects with the name entered by the user are retrieved by the user.

In our `CoreDataViewController.m` implementation file, locate the `searchData` method and add the following code as shown in the code snippet:

```

- (IBAction)searchData:(id)sender {

    CoreDataExampleAppDelegate *appDelegate = [[UIApplication
        sharedApplication] delegate];
    NSManagedObjectContext *context = [appDelegate
        managedObjectContext];
    NSEntityDescription *entityDesc = [NSEntityDescription
        entityWithName:@"Person" inManagedObjectContext:context];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];

    [request setEntity:entityDesc];
    NSPredicate *pred = [NSPredicate predicateWithFormat:@"(name =
        %@)", Name.text];
    [request setPredicate:pred];

    NSManagedObject *matches = nil;
    NSError *error;
    NSArray *objects = [context executeFetchRequest:request
        error:&error];

    if ([objects count] == 0) {
        recordsFound.text = @"No matches were found matching your
            criteria";
    }
}

```

```
    } else {
        matches = [objects objectAtIndex:0];
        DOB.text = [matches valueForKey:@"dob"];
        Gender.text = [matches valueForKey:@"gender"];
        recordsFound.text = [NSString stringWithFormat:@"%d Matches Found", [objects count]];
    }
    [request release];
}
```

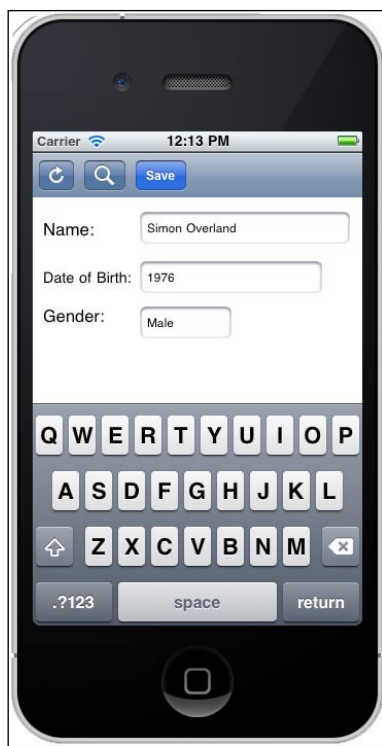
What we are doing in the above code is using our application delegate instance to identify the managed object context as we did for our `saveData` method. We then use this context object to create a new managed object using the **Person** entity item, and then create a predicate object to only return objects matching the name specified by the user. If a match is found, those objects are placed in an array, and then the objects are displayed to the form fields using the **valueForKey** method.

It is good programming practice to always release the memory used by your objects. This will save you time investigating programming errors down the track. We will need to add these form object outlet controls to our `viewDidLoad()` and `dealloc()` methods that are contained within the `coreDataViewController.m` file:

```
- (void)viewDidUnload {
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.Name = nil;
    self.DOB = nil;
    self.Gender = nil;
    self.recordsFound = nil;
}

- (void)dealloc {
    [Name release];
    [DOB release];
    [Gender release];
    [recordsFound release];
    [super dealloc];
}
```

The `viewDidUnload` method is called when our `ViewController` has been unloaded. What we are doing in the above code snippet is freeing up the memory which we allocated during program execution. The `dealloc` method is called as a final step when our application has ended. In this code module, we are releasing each of our objects. To end this tutorial, I have included a screenshot which shows the final Core Data Example application with all visual components being populated with data from our Core Data database:



What just happened?

In this section, we looked at how we can build a simple database application using Core Data and SQLite. We looked at what are the differences between Entities and Attributes, and how to go about creating these, and then finally looked at how we use the Managed Object Context and Managed Object Model to save and retrieve data from our SQLite database.

AV Foundation Frameworks

The AV Foundation framework provides an Objective-C interface which handles the recording and playing of audio and video content within your iOS application. This framework also provides the `AVAudioSession` class which handles the configuring and managing of your application's audio session.

The `AVAudioSession` class handles the following:

- ◆ Playing an audio File
- ◆ Playing a movie using Media Player

Whilst the Media Player Framework proves to be great with handling your entire general media playback, the AV Foundation Framework offers audio recording features making it possible to record new sound files directly within your application.

In order to make it possible for your application to handle playing of audio and recording within your application, you will need to include two new classes:

FRAMEWORK NAME	DESCRIPTION
AVAudioRecorder	Records Audio in a variety of different formats to memory or to a local file on the iPhone. The recording process is even clever enough to continue while other functions are running in your application.
AVAudioPlayer	Plays back audio files of any length. By using this class, you can implement a backing game soundtrack or other complex audio applications and you have complete control over the playback, including the ability to layer multiple sounds on top of one another.

As you can see, the Media Player Framework and the AV Foundation Framework work hand in hand in order to handle playing of audio files.

Playing an audio File

Playing an audio file is a simple process. Before we can do this, we need to first include the AV Foundation Framework into our project. In the next section, we will be creating a simple application which will play an audio file using the MediaPlayer Framework.

The Media Player Framework is used for playing back video and audio from either local or remote resources and can be used to call up the iPod interface from your application from which you are able to select songs to play back. The Framework integrates well with all the built-in media features that your phone has to offer and we will be making use of the following five classes in our sample application:

FRAMEWORK NAME	DESCRIPTION
MPMoviePlayerController	Allows playback of a piece of media, which is either located on the iPhone file system or through a remote URL. The player controller can provide a GUI for scrubbing through video, pausing, fast forwarding, or rewinding.
MPMediaPickerController	Presents the user with an interface for choosing media to play. You can filter the files which are displayed by using the media picker, or allow the selection of any file from the media library.
MPMediaItem	A single piece of media, such as a song.

FRAMEWORK NAME	DESCRIPTION
<code>MPMediaItemCollection</code>	Represents a collection of media items that will be used for playback. An instance of <code>MPMediaPickerController</code> returns an instance of <code>MPMediaItemCollection</code> that can be used directly within the next class—the music player controller.
<code>MPMusicPlayerController</code>	Handles the playback of media items and media item collections. Unlike the movie player controller, the music player allows playback from anywhere in your application.

The MediaPlayer Framework supports playback of the following file formats:

- ◆ AAC (16 to 320Kpbs)
- ◆ AIFF
- ◆ AAC Protected (MP4 from iTunes Store)
- ◆ MP3 (16 to 320Kbps)
- ◆ MP3 VBR
- ◆ Audible (formats 2-4)
- ◆ Apple Lossless
- ◆ WAV
- ◆ MOV, M4V, MPV, or MP4 video codec formats

Creating an application to play an audio file

Playing audio files is one of the common tasks on the iPhone, apart from playing video content. What we will be achieving in this section is to build a simple application which will contain two buttons—Play Audio and Stop Audio.

Time for action – creating the MediaPlayer application

Before we can proceed, we first need to create the `MediaPlayer` project. To refresh your memory, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section, *Creating your first iPhone application*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family dropdown.

5. Click on the **Next** button to proceed to the next step in the wizard.
6. Enter **MusicPlayer** as the name of the project, and then click on the **Next** button to proceed to the next step of the wizard.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

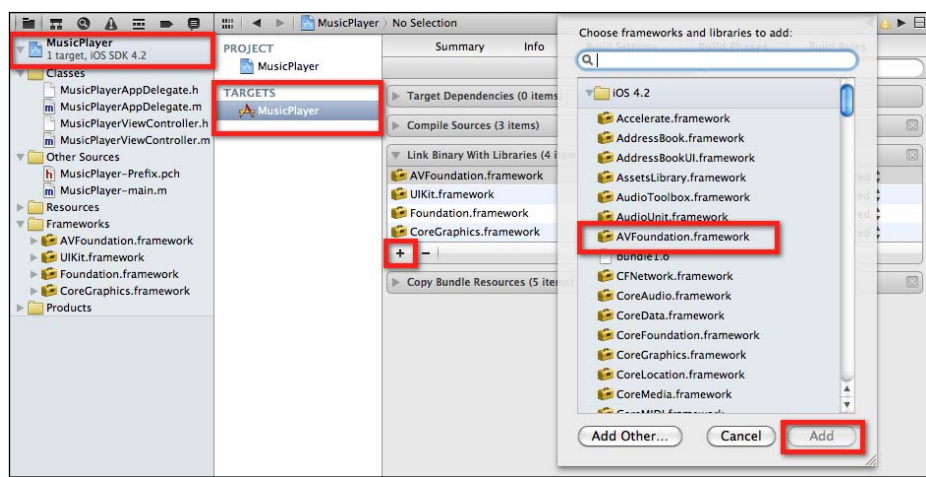
 The **Source Control: Create local git repository for this project** is not checked by default.

Now that we have created our `MusicPlayer` project, we need to add an important framework to our project to enable our application to have the ability to play audio files.

To add the AV Foundation Framework to your project, select the Project Navigator Group, and then follow these simple steps as outlined below:

1. Select your Project.
2. Then select your project target from under the TARGETS group.
3. Select the 'Build Phases' tab.
4. Expand the 'Link binary with Libraries' disclosure triangle.
5. Finally, use the + to add the library you want. You can also search if you can't find the framework you are after from within the list.

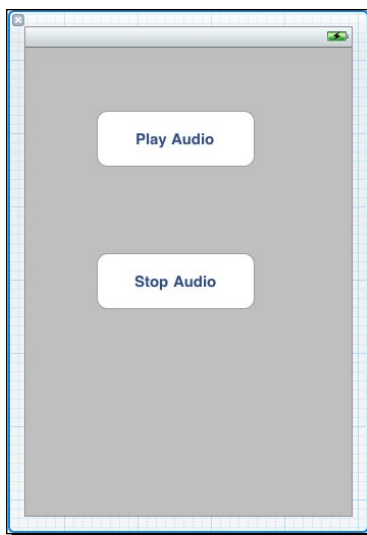
If you are still confused how to go about adding the frameworks, follow the screenshot below which highlights the areas that you need to select (*surrounded by a red rectangle*):



Now that you have added the **AVFoundation.framework** into your project, you need to start building your user interface which will be responsible for playing the audio:

1. From the Object Library, select and drag a (UIButton) Round Rect Button control and add this to our view, resize accordingly and then modify the Object Attributes section of the Round Rect Button and set its title to **Play Audio**.
2. From the Object Library, select and drag a second (UIButton) Round Rect Button control and add this to our view, resize accordingly and then modify the Object Attributes section of the Round Rect Button and set its title to **Stop Audio**.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same as mine, feel free to adjust yours:



As you can see, our form doesn't do much at this stage and if you were to run this application in the simulator, you would see the controls as laid out on your screen. Our next step is to start to add some code into our `MusicPlayerViewController.h` interface file so that we can utilise the methods that the `AVAudioPlayer` class has to offer:

1. Open the `MusicPlayerViewController.h` interface file, located within the **Classes** folder and insert the following lines of code below the `UIKit` `#import` statement:

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

@interface MusicPlayerViewController : UIViewController {
    AVAudioPlayer *player;
```



```
}

@property(n nonatomic, retain) AVAudioPlayer *player;

@end
```

What we have just declared in the code snippet above, is to make our View Controller aware of what the `AVAudioPlayer` class header has to offer by exposing all of its class methods.

2. Next, open the `MusicPlayerViewController.m` implementation file, located within the `Classes` folder, then add the following lines of code:

```
#import "MusicPlayerViewController.h"
@implementation MusicPlayerViewController
@synthesize player;
```

We then need to declare an instance variable (**player**) to our `AVAudioPlayer` class, and create a property which will enable us to reference the instance variable (**player**) from within our `MusicPlayerViewController.m` implementation file; this object provides us with a simplified approach to interact with this variable and use it throughout our implementation file. You can probably think of this as a pointer variable to our `AVAudioPlayer` class which we defined in our associated interface file.

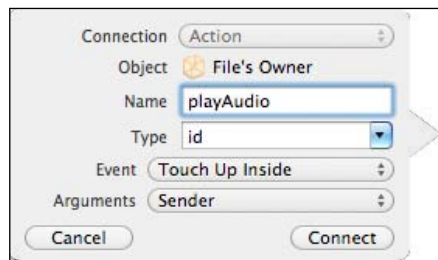
In order for our buttons to work, we need to create the associated actions for each of these, as well as writing the necessary code to perform those tasks. This section will show you how to go about connecting your buttons to action events, with each performing the task of playing or stopping audio playback. So let's get started:

1. We need to create an action event, select the **Play Audio** button, and hold down the **Ctrl** key while you drag this into the `MusicPlayerViewController.m` implementation file class as shown below:



2. Specify a name for the action that you want to create. Enter **playAudio** as the name of the action.

3. Set the type of event to be **Touch Up Inside**:



4. Click on the **Connect** button to have Xcode create the event.
5. Repeat steps 3 – 4 to create the action event for the **stopAudio** button.
6. Add the following code snippet to our `playAudio` function which will handle playing our sample audio file:

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

    // Get the file path to the song
    NSString *filePath = [NSBundle mainBundle] pathForResource:@"music"
ofType:@"mp3"];

    // Convert the file path to a URL
    NSURL *fileUrl = [NSURL alloc] initWithFilePath:filePath];

    //Initialize the AVAudioPlayer
    self.player = [[AVAudioPlayer alloc] initWithContentsOfURL:
fileUrl
error:nil];

    // Release the memory allocated to our objects
    [filePath release];
    [fileUrl release];

    // Play the audio file
    [self.player play];
}
```

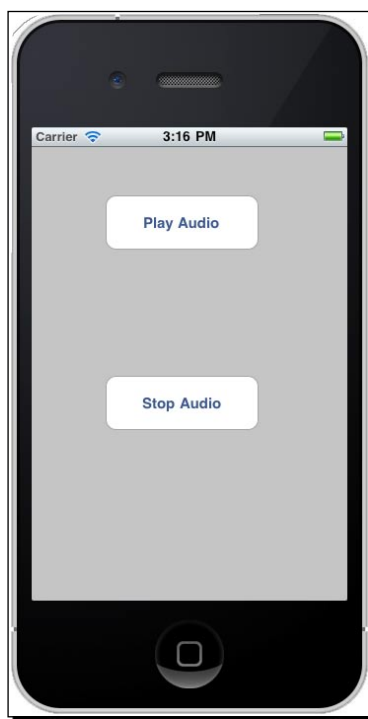
What we have just added to our `playAudio` function is that we have declared a variable (NSString) `filePath` which will contain the file path to our audio file. Next we create a (NSURL) `fileUrl`, which converts our file path to an object which is what the `AVAudioPlayer` needs when it is being initialized. Next, we set up and initialise the `AVAudioPlayer`, and assign this to our player object. Finally, we release the memory allocated to our `filePath` and `fileUrl` objects and then play the audio.

7. Add the following code snippet to our `stopAudio` function which will handle stopping our audio file playback:

```
- (IBAction)stopAudio:(id)sender {  
    [self.player stop];  
}
```

In the above code snippet, we pass the `stop` method to our player object. By doing this, it tells the AV Foundation framework and the `AVAudioPlayer` class to cease all playback of audio.

The screenshot below shows our `MusicPlayer` application running on the iPhone Simulator:



What just happened?

As you can see, by using the AV Foundation Framework and the `AVAudioPlayer` class, you can incorporate audio and sounds within your iPhone applications. Feel free to experiment further with the AV Foundation Framework. There is a wealth of documentation out there which can help you understand this framework and many others in greater depth than we have covered within this section.

In the next section, we will be looking at how we go about playing a movie file using the Media Player Framework.

Playing a movie using Media Player

Playing videos is one of the most common tasks on the iPhone. On the iPhone, all videos must be played full-screen. Before we can play any videos, we need to include the `MediaPlayer` Framework in our application; you will notice that this is the same framework that we used when we were playing our audio file. In the next section, we will be creating a simple application to play a movie file, so let's get started.

Time for action – creating the MediaPlayer application

Before we can proceed, we first need to create our `MediaPlayer` project. To refresh your memory, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section *Creating your first iPhone application*.

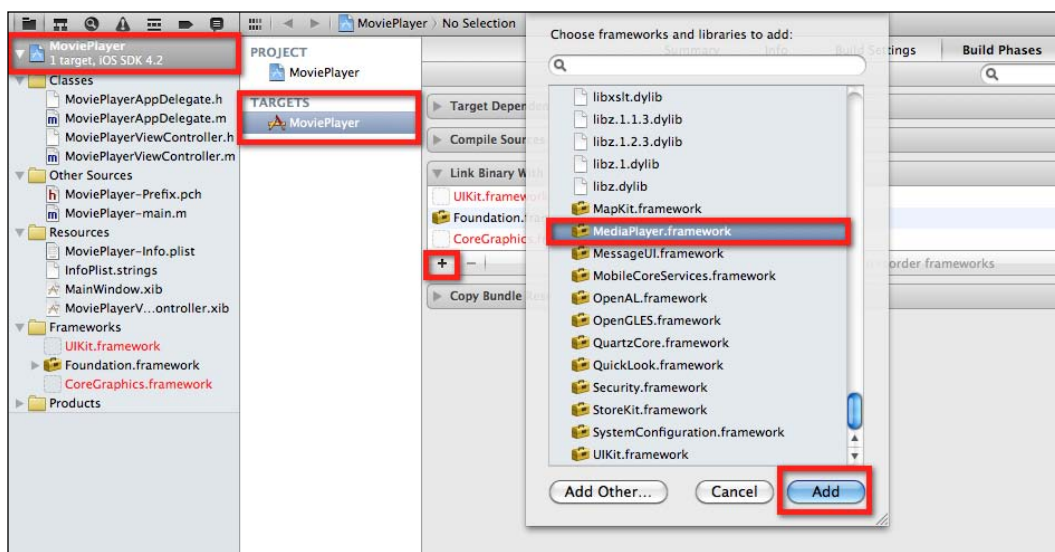
- 1.** Launch Xcode from the `/Xcode4/Applications` folder.
- 2.** Choose **Create a new Xcode project**, or **File | New Project**.
- 3.** Select the View-based Application template from the list of available templates.
- 4.** Select **iPhone** from under the Device Family dropdown.
- 5.** Click on the **Next** button to proceed to the next step in the wizard.
- 6.** Enter in **MediaPlayer** as the name for your project, and then click on the **Next** button to proceed to the next step of the wizard.
- 7.** Specify the location where you would like to save your project.
- 8.** Click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our `MediaPlayer` project, we need to add an important framework to our project to enable our application with the ability to play movie files.

To add the Media Player Framework to your project, select the Project Navigator Group, and then follow these simple steps as outlined below:

1. Select your Project.
2. Then select your project target from under the TARGETS group.
3. Select the 'Build Phases' tab.
4. Expand the 'Link binary with Libraries' disclosure triangle.
5. Finally, use the + to add the library you want. You can also search if you can't find the framework you are after from within the list.

If you are still confused how to go about adding the frameworks, follow the screenshot below which highlights the areas that you need to select (*surrounded by a red rectangle*):

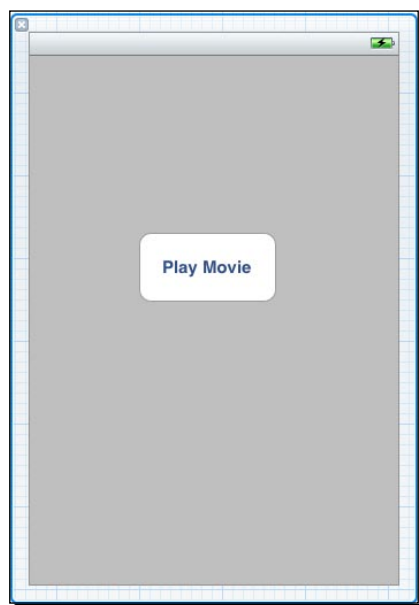


Now that you have added the **MediaPlayer.framework** into your project, we need to start building our user interface which will be responsible for playing the movie:

1. From the Object Library, select and drag a (UIButton) Round Rect Button control and add this to our view.
2. Resize accordingly and then modify the Object Attributes section of the Round Rect Button and set its title to **Play Movie**.

We don't need to add a stop button, as we will be adding an event which will handle this for us when the movie has finished playing.

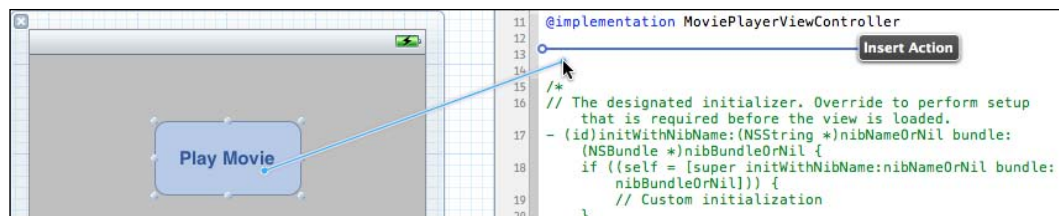
If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same as mine, feel free to adjust yours:



As you can see, our form doesn't do much at this stage and if you were to run this application in the simulator, you would see the controls as laid out on your screen

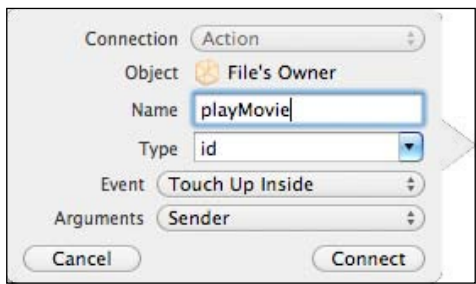
The following text will show you how to connect your buttons up to action events which will each perform the task of playing the video. So let's get started:

1. We need to create an action event, select the **Play Movie** button, and hold down the *Ctrl* key while you drag this into the `MoviePlayerViewController.m` implementation file class as shown below:



2. Specify a name for the action that you want to create. Enter in `playMovie` as the name of the action.

3. Set the type of event to be **Touch Up Inside**:



4. Click on the **Connect** button to have Xcode create the event.

We now need to add the code to our `playMovie` function which will handle playing our sample movie file. Enter in the following code snippet to this function:

```
- (IBAction)playMovie:(id)sender {
    NSString *filepath = [[NSBundle mainBundle]
        pathForResource:@"sample-movie" ofType:@"mp4"];
    NSURL *fileURL = [NSURL fileURLWithPath:filepath];
    MPMoviePlayerController *moviePlayerController =
        [[MPMoviePlayerController alloc] initWithContentURL:fileURL];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(moviePlaybackComplete:)
        name:MPMoviePlayerPlaybackDidFinishNotification
        object:moviePlayerController];

    [self.view addSubview:moviePlayerController.view];
    moviePlayerController.fullscreen=YES;
    [moviePlayerController play];
}
```

What we have just added to our `playMovie` function is that we have declared a variable (NSString) `filePath` which will contain the file path to our movie file. Next we create a (NSURL) `fileUrl` which converts our file path to an object, which is what the `MPMoviePlayerController` needs when it is being initialized.

We then add the `MPMoviePlayerController` view to our custom view controller so that it will appear on the screen. We specify that we want to display this fullscreen, and finally we tell the `moviePlayerController` to commence playback.

Since we have allocated memory to our `moviePlayerController` object, at this stage we haven't released it yet, this being due to not knowing when the movie playback will actually finish. Fortunately, the `MPMoviePlayerController` object comes prebuilt with methods to handle this scenario and will dispatch a notification method called `MPMoviePlayerPlaybackDidFinishNotification` to the `NSNotificationCenter` when the movie playback completes. This is shown in the highlighted code in the above snippet.

When we play back video content within our iPhone applications, you will sometimes need to modify the `scalingMode` property of the `MPMoviePlayerController` object. By setting this property it will determine how the movie image adapts to fill the playback size that you have defined. The following scaling modes currently exist and are displayed below:

- ◆ `MPMovieScalingModeNone`
- ◆ `MPMovieScalingModeAspectFit`
- ◆ `MPMovieScalingModeAspectFill`
- ◆ `MPMovieScalingModeFill`

The two main common scaling modes used are the `MPMovieScalingModeAspectFill` and `MPMovieScalingModeFill`.

In order to implement this property in your application, insert the following line of code just before the `[moviePlayerController play]` statement:

```
moviePlayerController.scalingMode = MPMovieScalingModeFill;
```

When you run your application, you will notice that the video fills the entire available space.

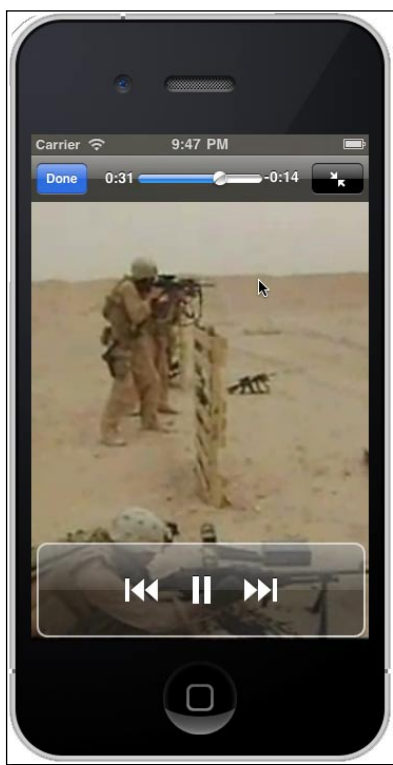
Next, we need to create the `moviePlaybackComplete:` method which will be responsible for releasing our `moviePlayerController` object. This is shown in the code snippet below:

```
- (void)moviePlaybackComplete:(NSNotification *)notification
{
    MPMoviePlayerController *moviePlayerController = [notification
    object];
    [[NSNotificationCenter defaultCenter] removeObserver:self
    name:MPMoviePlayerPlaybackDidFinishNotification
    object:moviePlayerController];
    [moviePlayerController.view removeFromSuperview];
    [moviePlayerController release];
}
```


In the above code snippet, we pass the object to the notification method. This is whatever we have passed in the previous code snippet, this being the `moviePlayerController` object. We start by retrieving the object using the `[notification object]` statement and then reference it with the new `MPMoviePlayerController` pointer.

We then send a message back to the `NSNotificationCenter` method which removes the observer we previously registered within our `playMovie` function. We finally proceed with cleaning up our custom view controller from our display, and then release the memory we previously allocated to our `moviePlayerController` object.

The screenshot below shows our `MoviePlayer` application running on the iPhone Simulator with movie playback in portrait mode; support is available to display this in landscape mode:



What just happened?

In this section, we learned about the `MediaPlayer` framework, and how we can use this within our applications to give us the ability to play audio and video. We learned about the various scaling modes for video playback and how to implement these.

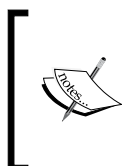
So there you have it. As you can see, by using the Media Player Framework and the `MPMoviePlayerController` class, you can incorporate movie playback within your iPhone applications.

Core Location Framework

Your iPhone comes built with the GPS (Global Positioning System) hardware as well as some location and mapping software which shows you where you are at any given time by using the built-in GPS receiver as well as information from cellular towers to derive this information.

So, what is the Core Location Framework? It can be defined as a framework which adds location awareness to your iPhone applications and makes use of the following events:

METHOD NAME	DESCRIPTION
<code>CLLocationManager</code>	A class which provides you with the mechanisms by which location information gets delivered to your application. A single instance of this class needs to be created, then optionally set some accuracy properties, and then call the <code>startUpdatingLocation</code> method.
<code>CLLocation</code>	Handles the events which are generated by the delegate methods located within your <code>CLLocationManager</code> instance. The <code>CLLocation</code> object not only encapsulates the geographical coordinates, but also captures the underlying information, such as speed, altitude and direction, as well as various other properties.



The Core Location Framework can use any of the following technologies: GPS, Cellular network, or Wi-Fi. If GPS is present, it is used first by the framework. However, if the device does not support GPS, or if an error occurred while obtaining the current location via GPS, the framework will use your Cell phone's network, and then use Wi-Fi.

Time for action – making your application location aware

Before we can proceed, we first need to create our `CoreLocation` project. To refresh your memory, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section *Creating your first iPhone application*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.

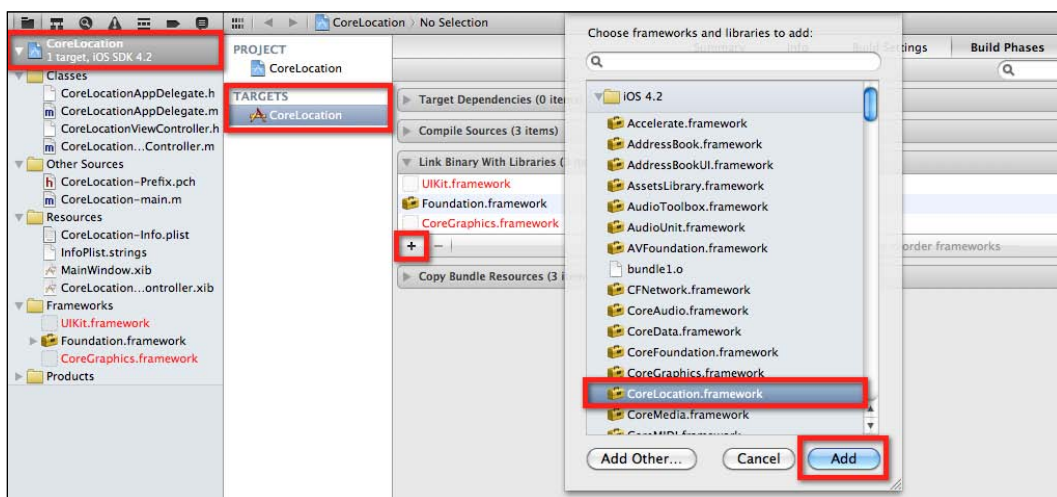
4. Select **iPhone** from under the Device Family dropdown, as the type of view to create.
5. Click on the **Next** button to proceed to the next step in the wizard.
6. Enter in **CoreLocation** and then click on the **Next** button to proceed to the next step of the wizard.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our `CoreLocation` project, we need to add an important framework to our project to give our application the ability to provide location information.

To add the CoreLocation Framework to your project, select the Project Navigator Group and then follow these simple steps as outlined below:

1. Select your Project.
2. Then select your project target from under the TARGETS group.
3. Select the 'Build Phases' tab.
4. Expand the 'Link Library with Libraries' disclosure triangle.
5. Use the + to add the library that you want. You can also search if you can't find the framework you are after from within the list.

If you are still confused about how to add the frameworks, follow the screenshot below which highlights the areas that you need to select (*surrounded by a red rectangle*):



Now that we have added the **CoreLocation.framework** into your project, we need to add the code which will be responsible for displaying our location information.

In order to make our application location-aware, we need to import the `<CoreLocation/CoreLocation.h>` interface file and implement the `CLLocationManagerDelegate` protocol declaration and create an instance variable to point to our location manager object:

1. Open the `CoreLocationViewController.h` interface file, located within the `Classes` folder and then add the following code as shown in the snippet below:

```
#import <CoreLocation/CoreLocation.h>
@interface CoreLocationViewController : UIViewController
    <CLLocationManagerDelegate> {
    CLLocationManager *locationManager;
}
```

In the above code snippet, we included a reference to the `CoreLocation.h` interface library; we then created a delegate method of the `CLLocationManager` method and then declared a variable which will hold the location to our `CLLocationManager` object.

We haven't quite finished yet. What we now need to do is modify our `ViewDidLoad` method located within our `CoreLocationViewController.m` implementation file.

2. Open the `CoreLocationViewController.m` implementation file located within the `Classes` folder, then locate and uncomment the `ViewDidLoad` method, and then add the following code as shown in the snippet below:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    locationManager = [[CLLocationManager alloc] init];
    locationManager.delegate = self;
    [locationManager startUpdatingLocation];
}
```

In the above code snippet, what we have done is initialize and allocate memory to our location manager object which we declared within our `CoreLocationViewController.h` file. We then set up the delegate method to be the current View Controller object, and then we tell the location manager to start sending location event information.

What we have been doing so far is setting up our application to be location-aware, but currently we still don't have any way of dealing with the information that the location manager is sending back. Our next step is to implement two delegate methods of the `CLLocationManager` object. The first one we will be implementing is the `locationManager:didUpdateToLocation` method which is called whenever the location manager updates to a new location. The location manager passes the previous location and the new location, unless this is the first time, hence resulting in the `fromLocation` being `nil`.

The second method which we will implement will be the `didFailWithError`. This method captures any errors which have occurred within the location manager when trying to retrieve a location value, in which case we will want to stop our location manager and make a call to the `stopUpdatingLocation` method in order to conserve battery power on the device.

In order to determine the current location, we need to write the code for it. Follow these steps to implement the `didUpdateToLocation` method:

1. Open the `CoreLocationViewController.m` implementation file located within the `Classes` folder.
2. Add the following code as shown in the code snippet directly under the `viewDidLoad` method:

```
- (void) locationManager: (CLLocationManager *) manager
    didUpdateToLocation: (CLLocation *) newLocation
    fromLocation: (CLLocation *) oldLocation
{
    NSLog(@"Location found at the following coordinates:
        %@", newLocation.description);
    [locationManager stopUpdatingLocation];
}
```

In the above code snippet: what this is doing is obtaining the current location from the `CLLocationManager` delegate and updating the `newLocation` and `oldLocation` values. We then use the `description` method of the `newLocation` object to extract and display the coordinates to the debug window using the `NSLog` function.

In this final section, we will implement the code to handle errors when the `CLLocationManager` is unable to retrieve location information. We use the `error` property which contains the information sent back from the `CLLocationManager` object and then we call the `stopUpdatingLocation` method of our location manager object:

1. Open the `CoreLocationViewController.m` file, located within the `Classes` folder.

2. Add the following code snippet directly underneath the `didUpdateToLocation` function:

```
- (void) locationManager: (CLLocationManager *) manager
    didFailWithError: (NSError *) error
{
    NSLog(@"An error has occurred, error details are: %@", error);
    [locationManager stopUpdatingLocation];
}
```

In the above code snippet, when an error occurs and the `CLLocationManager` fails to bring back location information for whatever reason, we need to stop this event by calling the `stopUpdatingLocation` method which ceases further updates.

What just happened?

In this section, we learned how we can use the Core Location Framework to incorporate GPS-like functionality within our applications. We looked at how we can use the `CLLocationManager` method to handle and provide updates to our current whereabouts, as well as the `didUpdateToLocation` and `didFailWithError` method to determine whether the location manager updated to a new location and what to do when we experienced an error, such as connection lost or unable to retrieve the current location.

Map Kit Framework—new and improved

Mapping applications can now include overlays that can identify regions on a map and also allow you to draw routes with annotations for customized directions and other functionalities.

Before we proceed with the next section, it is worth giving you an understanding of what the MapKit mapping framework is; this framework is based on the Google Maps engine and gives you the ability to add interactive maps to your applications. By adding this functionality, it gives you the flexibility of scrolling and zooming Maps to any region within the world. Place Holders (or annotations) can be added to the map to display additional information.

In the next section, we will be creating a map-based application making use of the Core Location and Map Kit Frameworks, so let's get started.

Time for action – creating a simple geographical application

Before we can proceed, we first need to create our `MapKit` project. To refresh your memory, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section *Creating your first iPhone application*:

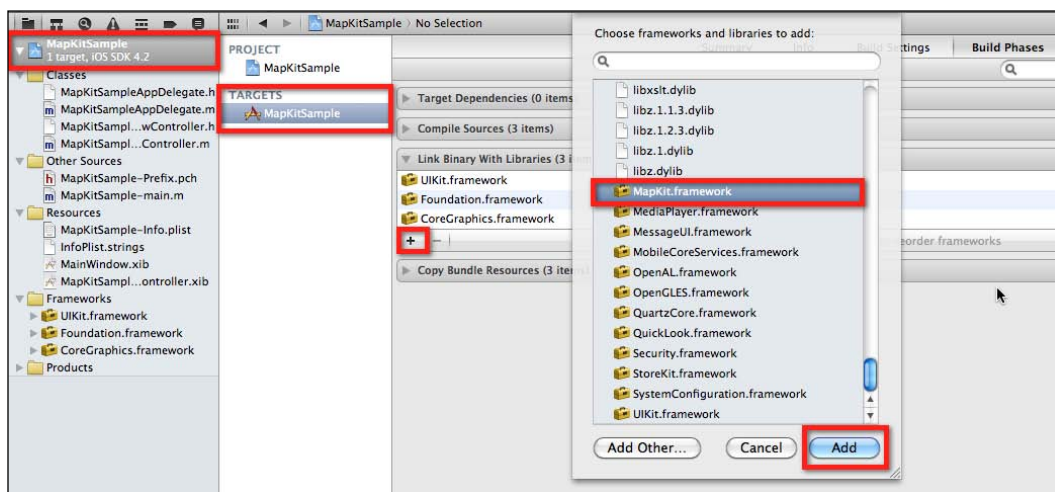
1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family dropdown.
5. Click on the **Next** button to proceed to the next step in the wizard.
6. Enter in **MapKitSample** and then click on the **Next** button to proceed to the next step of the wizard.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our `MapKitSample` project, we need to add an important framework to our project to enable our application to have the ability to view map information.

To add the MapKit Framework to your project, select the Project Navigator Group and then follow these simple steps as outlined below:

1. Select your Project.
2. Then select your project target from under the TARGETS group.
3. Select the 'Build Phases' tab.
4. Expand the 'Link Library with Libraries' disclosure triangle.
5. Use the + to add the library that you want. You can also search if you can't find the framework you are after from within the list.

If you are still confused about how to add the frameworks, follow the screenshot below which highlights the areas that you need to select (*surrounded by a red rectangle*):



Now that you have added the **MapKit.framework** into your project, we need to import the code into the View Controller which will be responsible for displaying our map location information.

In order to make our application display the map to our view, we will need to import the `<MapKit/MapKit.h>` interface header file, so that we can utilise its methods:

1. Open the `MapKitSampleViewController.h` interface file, located within the `Classes` folder, then add the following code as shown in the snippet below:

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

@interface MapKitSampleViewController : UIViewController {
    MKMapView *mapView;
}
```

In the above code snippet, we have included a reference to the Cocoa `MapKit.h` header file which will expose its methods so that we can use these within our `MapKitSample` implementation file. Then we have created an instance variable (**mapView**) which is a pointer to our `MKMapView` object which is responsible for holding our Map location information.

2. We haven't quite finished yet. What we now need to do is modify our `ViewDidLoad` method located within our `MapKitSampleViewController.m` implementation file. Open the `MapKitSampleViewController.m` implementation file.

Locate and uncomment the `ViewDidLoad` method. Add the following code as shown in the snippet below:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    mapView = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    [self.view addSubview:mapView];
}
```

In the above code snippet, what we have actually done is initialize and allocate memory for our `mapView` object which we declared within our `MapKitSampleViewController.h` file, and then we add our `mapView` object to our current view so that we can display on the screen.

If you were to Build and Run your application now, you will see a map displayed within the iOS Simulator. You have the ability to navigate around the map and zoom in and out.

The `mapKit` framework has the ability to show you your current location within the map. It also allows you to set a variety of `mapTypes`. We will be adding some additional code to our `ViewDidLoad` method which is located within our `MapKitSampleViewController.m` implementation file:

1. Open the `MapKitSampleViewController.m` implementation file, located within the `Classes` folder.
2. Locate the `ViewDidLoad` method and then add the following highlighted code as shown in the snippet below:

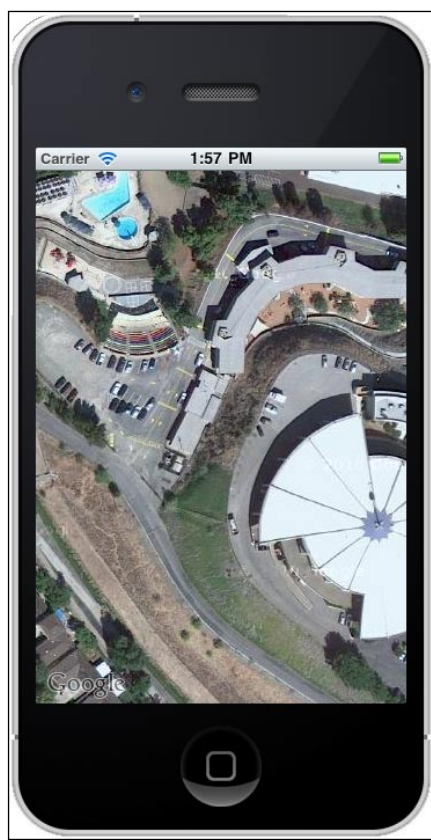
```
- (void)viewDidLoad {
    [super viewDidLoad];
    mapView = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    mapView.mapType = MKMapTypeHybrid;
    mapView.showsUserLocation = YES;
    [self.view addSubview:mapView];
}
```

In the above code snippet, what we have added is the ability to display our map in Hybrid view (combination of satellite view and road information) as well as telling our map to display our current location which will be indicated by an animated blue marker.

The iOS native Maps application allows you to choose from these three possible Map Types which are explained below:

MAP TYPE CONSTANT	DESCRIPTION
<code>MKMapTypeStandard</code>	This is the default type of map to display if none is specified and this type will show a normal map containing street and road names.
<code>MKMapTypeSatellite</code>	Setting this type of map will display Satellite view information.
<code>MKMapTypeHybrid</code>	This type of map will show a combination of a Satellite view with road and street information overlaid.

We have finally made it. If you Build and Run your application; you should now see a map displayed with the animated blue marker flashing. I have zoomed in at a random location to show the capabilities of the MapKit Framework which are shown in the screenshot below:





When running MapKit applications using the iPhone Simulator, it will always default to Apple's Headquarters which is located at *1, Infinite Loop based out at California*. In order to get a better location, it is better to use your iOS device because the iOS simulator on the Mac OS X Snow Leopard can geocode your IP address and get the approximate location.

What just happened?

In this section, we looked at the two powerful frameworks, Core Location and MapKit. As we saw in our example application, these two frameworks are often used together to deliver the functionality of a location-aware iPhone application.

We use the location manager to receive updates on the user's location, and showed a map using the satellite view. So as you can see, you can do some pretty neat stuff with Core Location.

New Framework APIs

The iOS 4 SDK comes equipped with several new framework APIs. We will be describing what each of these fantastic new additions to the iOS SDK consist of below:

- ◆ **Quick Look:** Just like Mail can preview documents, Quick Look will allow developers to present the same functionality in their apps.
- ◆ **Calendar Access:** The new Calendar Access API includes a new way for your application to create and edit events directly from within the calendar application, by using the Event Kit Framework. This allows you to create recurring events, set up start and end times, and then assign them to any calendar on the device.
- ◆ **In-App SMS:** The in-App SMS API allows a way for developers to include the functionality of sending SMS messages, directly from within their application. Although this feature supports sending of messages to multiple numbers, it does not include the ability to handle replying to SMS messages and does not include support for sending MMS messages.
- ◆ **Photo Library Access:** With the Photo Library Access API, your iOS applications now have the ability to directly access user photos and videos by using the Media Library APIs.

Have a go hero – modifying the Core Data example

Now you have a good working knowledge of Core Data and how to go about adding entities to a database application. The task will be to modify the database model, add some form fields and create some entities to allow data to be written and retrieved from a SQLite database:

1. Create two additional entities—**Job Title** and **Country** and set their type to **String**. You can refer to the section *Building a simple database application* located in this chapter. Remember to save your project after you have added these attributes.
2. Next, create two labels for the two entities above and modify their captions appropriately.
3. Next, create the text fields and outlets for the two entities that you created in the first step prior to creating the synthesize methods for each of them.
4. Next, modify the `saveData` method to save the data entered in the two fields.
5. Modify the `searchData` method to retrieve the data and display the contents to those two new entities.
6. You will need to modify the `viewDidLoad` method to release the memory used by those two entities and then release the memory to those objects within the `dealloc` method.
7. Once you are satisfied that everything has been completed, you can Compile and then Build and Run the application.

Once you have implemented the above, you will have a fully customized application which will be able to save and retrieve data for the newly created fields.

Pop quiz – Core Data / Media Playback and Core Location

1. What option needs to be selected when creating a Core Data application?
 - a. iPhone
 - b. Window-based application
 - c. Use Core Data
2. When writing to an SQLite database field, what two methods need to be set?
 - a. `getValue`
 - b. `setValue`
 - c. `getKey`
 - d. `forKey`

3. What Framework allows you to play the following file formats: AIFF, AAC, MP3, MOV and M4V?
 - a. AVFoundation.Framework
 - b. MediaPlayer.Framework
4. What method of the MPMoviePlayerController allows for scaling?
 - a. scaleMode
 - b. fullScreen
 - c. scalingMode
5. What method of the CLLocationManager object gets called when the location changes?
 - a. startUpdatingLocation
 - b. didFailWithError
 - c. didUpdateToLocation
6. What method of the CLLocationManager class would you use to capture and handle errors?
 - a. didUpdateToLocation
 - b. didFailWithError
 - c. didGetLocationFailed
 - d. didUnable...ToGetLocation

Summary

In this chapter, we covered the Xcode Frameworks, and how to go about creating a variety of applications to play audio and video, creating a Map-based application using the Core Location and MapKit frameworks, as well as taking a look into the Cocoa Core-Data Framework to build a database application. We then took a look into each of the new APIs and improvements that have been added to the iOS 4 SDK.

Now that we've learned about each of the frameworks that come as part of the iOS4 SDK, and having used some of these frameworks to create sample projects to play audio and movies and GPS location services, we are ready to start focussing on what comprises the Model-View-Controller Application design pattern and its relation to Xcode and Interface Builder.

In the next chapter, we will take a look into the MVC Frameworks and gain a better understanding of the MVC design pattern structure, and how both Xcode and Interface Builder are used to implement MVC when creating views.

5

Designing Application Interfaces using MVC

In this chapter, we will learn about the Model-View-Controller (MVC) and how we can use this technology to build applications using Interface Builder. We will look at each of the components involved in MVC, and why this technology is a better alternative to the way you program, by keeping your program logic separate from your user interface.

In this chapter, we will:

- ◆ Understand the Model-View-Controller (MVC) design **pattern structure and how** Xcode and Interface Builder implement MVC. Learn how to use the View-Based Application Template.
- ◆ Understand what Table-Views are and how to go about creating a simple application.
- ◆ Understand what Navigation-based **Applications and Rotatable and Resizable** Interfaces are and how to go about Repositioning controls within the View on Rotation.
- ◆ Learn how to use Switches, Sliders, Segmented Controls, **Scrolling Views, Web Views**, Pickers, Date Pickers, and how to implement Custom Pickers.
- ◆ Learn how to handle basic user Input/Output using TextFields, TextViews, and Buttons.

We have got quite a bit to cover, so let's get started.

Developing iOS applications using MVC design

To begin this chapter, we will start by explaining "what a Design Pattern" is. This will help you understand how this interacts with MVC and then go into the various layers of what makes up the **Model-View-Controller** and why this pattern was chosen when developing iOS applications.

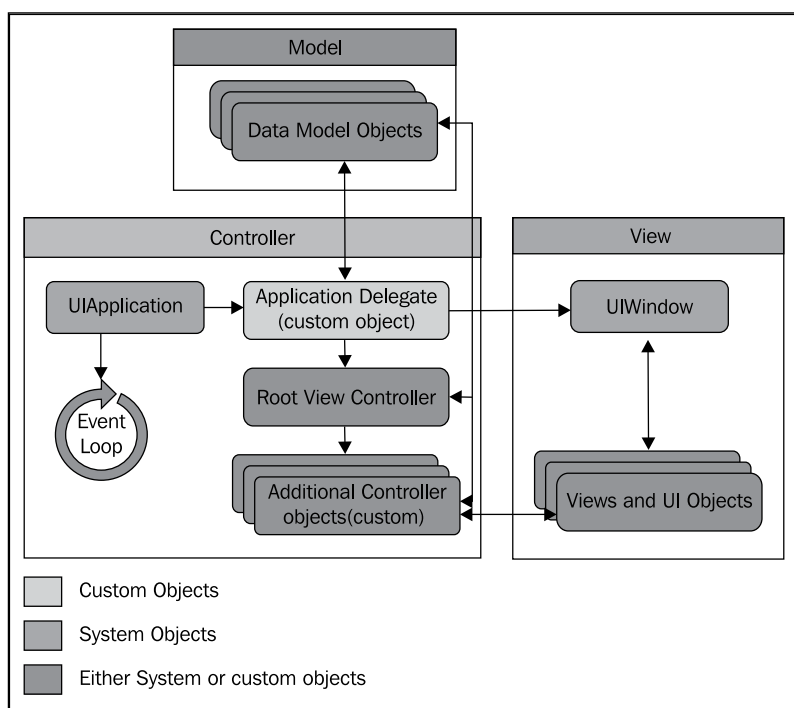
Reusing tested (or standard) solutions: Design patterns

So what is a Design Pattern? A **Design Pattern** is basically something that offers a solution to a problem by providing a set of general, reusable, and tested solutions pertaining to common programming scenarios. Some examples of design patterns are the MVC Controller, and Delegates which you have no doubt used in previous chapters when creating View-Based or Window-Based applications.

Design patterns are used extensively throughout the iOS Frameworks, so if you create an iPhone application, there is no doubt you will be using these in some way.

Understanding the Model-View-Controller design pattern

Model-View-Controller (MVC) Pattern separates an applications data structure into three separate parts, the Model, the View, and the Controller as shown in the following figure:



The table below describes each of the components that make up and are part of the MVC model:

MVC TYPE	DESCRIPTION
Model	The model provides the underlying data and methods that provide information to the rest of the application.
View	This view can consist of and is made up of one or more views, which contain different types of objects, Buttons, Switches, Text Fields, and so on that a user can interact with. This is basically defined as your user interface which the user will see.
Controller	This part is the most important part which handles the interaction between different types of objects. It handles receiving of user input and then determines how to handle this accordingly. A Controller has the ability to access and update a view using information from the model, that is, updating some text of a field.

So as you can see, by using the MVC design approach, you can break your application into these three parts: the **Model**, the **View**, and the **Controller**. By using MVC, it forces you to program in a more structured approach and allows you to even reuse code and the same model design across multiple applications which use different views and controllers.

In the next section, we will take a look at how Xcode and Interface Builder (IB) implement and use the Model-View-Controller.

Implementing MVC using Xcode and Interface Builder

In the previous chapters, you learned about Xcode and how to use Interface Builder. You have also learned how to connect the objects in the XIB file to the code in an application via the use of **Outlets** (`IBOutlets`) and **Actions** (`IBActions`) and what we were actually doing was binding a view to a controller.

In the following section, we will take a look at what a View is and what a Controller is by building a simple Pizza Ordering application that will allow the user to make a series of choices from the menu and then calculate the total cost.

Time for action – building a Pizza order application

Before we can proceed and create our "PizzaOrders" project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.

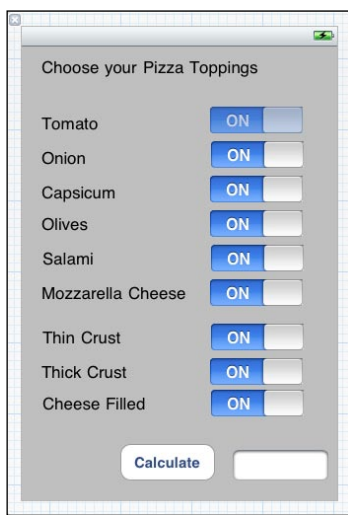
4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Once your project has been created, you will be presented with the Xcode interface along with all files that the project template created for you displayed within the Project Navigator window pane.

In this section, we will start to add the components required to build our application user interface. To refresh your memory, you can refer to the section *Adding Controls to your User Interface* which we covered in *Chapter 3, Working with the Interface Builder*:

1. From the Object Library, select and drag a (UILabel) control to the view and position the control at the top of our view. We will need to click on the Object Attributes tab and label our control **Choose your Pizza Toppings** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
2. From the Object Library, select and drag a (UILabel) control to the view and position the control underneath the heading which we just created and added. We will need to click on the Object Attributes tab and label our control **Tomato** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
3. Next, we will need to add a (UISwitch) Switch control which will be used to determine whether we want to add Tomatoes to our Pizza or not. From the Object Library, select and drag a UISwitch control and place it directly beside the label control which we created in the previous step.
4. Repeat steps 10 through 11 to add the remaining UILabel and UISwitch controls for the following:
Onion, Capsicum, Olives, Salami, Mozzarella Cheese, Thin Crust, Thick Crust, Cheese Filled.
5. Next, add a (UIButton) Round Rect Button control to our view. Modify the Object Attributes of the Round Rect Button control and set its title to read Calculate.
6. Our final step is to add a (UITextField) control which will be used to display the total price of your Pizza. From the Object Library, select and drag a UITextField control and place it to the right of the **Calculate** button which we created in the previous step.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In this section, we created a simple Pizza Ordering application to handle pizza topping selections provided by the user and then calculate the purchase price. We looked at how to go about adding a number of `UILabel` controls which will be used to display each of the available pizza toppings to choose from, and a `UISwitch` control to show which toppings have been selected. We then added a `UITextField` control which will be used to display the total cost of your pizza selections. Finally we added a `UIButton` control which will handle the processing to display the total purchase price for your pizza, based on the toppings selected.

Time for action – binding our Controls using Outlets and Actions

Our final step will be to bind our control objects to our View Controller and connecting these via outlets and actions. We covered this in the section *Making our Components work together* in *Chapter 3, Introducing Interface Builder*.

1. Once you have created your outlets within the `PizzaOrdersViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller know what controls we will be dealing with:

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

```
@interface PizzaOrdersViewController : UIViewController {
```

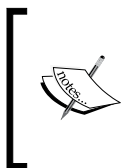
```
IBOutlet UISwitch *toppingTomato;
IBOutlet UISwitch *toppingOnion;
IBOutlet UISwitch *toppingCapsicum;
IBOutlet UISwitch *toppingOlives;
IBOutlet UISwitch *toppingSalami;
IBOutlet UISwitch *toppingMozarella;
IBOutlet UISwitch *crustThin;
IBOutlet UISwitch *crustThick;
IBOutlet UISwitch *crustCheeseFilled;
IBOutlet UITextField *TotalPayment;
IBOutlet UIButton *Calculate;
}

@property (nonatomic, retain) IBOutlet UISwitch *toppingTomato;
@property (nonatomic, retain) IBOutlet UISwitch *toppingOnion;
@property (nonatomic, retain) IBOutlet UISwitch *toppingCapsicum;
@property (nonatomic, retain) IBOutlet UISwitch *toppingOlives;
@property (nonatomic, retain) IBOutlet UISwitch *toppingSalami;
@property (nonatomic, retain) IBOutlet UISwitch *toppingMozarella;
@property (nonatomic, retain) IBOutlet UISwitch *crustThin;
@property (nonatomic, retain) IBOutlet UISwitch *crustThick;
@property (nonatomic, retain) IBOutlet UISwitch
*crustCheeseFilled;

@property (nonatomic, retain) IBOutlet UITextField *TotalPayment;
@property (nonatomic, retain) IBOutlet UIButton *Calculate;

@end
```

2. Our next step is to create the action event in our `PizzaOrdersViewController.m` implementation file. But we first need to synthesize our properties so that we can use them within our view controller.



If we don't declare these, we will receive warning messages which could result in unexpected results occurring in your application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "PizzaOrdersViewController.h"

@implementation PizzaOrdersViewController
```

```

@synthesize Calculate;
@synthesize TotalPayment;
@synthesize toppingTomato, toppingOnion, toppingCapsicum, toppingOlives,
    toppingSalami, toppingMozarella, crustThin, crustThick,
    crustCheeseFilled;

```

- 3.** Next, we declare our (IBAction) calculatePurchase method which will be responsible for displaying the total amount that the user needs to pay for their customized Pizza selection:

```

- (IBAction)calculatePurchase:(id)sender {
// Declare and initialise each of our pizza toppings and base
types.
    float totalAmount = 0.00;
    float tomatoAmount = 0.00;
    float onionAmount = 0.00;
    float capsicumAmount = 0.00;
    float oliveAmount = 0.00;
    float salamiAmount = 0.00;
    float mozzarellaAmount = 0.00;
    float thinCrustAmount = 0.00;
    float thickCrustAmount = 0.00;
    float cheeseCrustAmount = 0.00;

    // Handle each of our topping selections

    // Check to see if we have chosen to include Tomatoes on our
Pizza
    if ([toppingTomato isOn]){
        tomatoAmount = 0.50;
    }
    else { tomatoAmount = 0; }

    // Check to see if we have chosen to include Onions on our Pizza
    if ([toppingOnion isOn]){
        onionAmount = 0.80;
    }
    else { onionAmount = 0; }

    // Check to see if we have chosen to include Capsicum on our
Pizza
    if ([toppingCapsicum isOn]){
        capsicumAmount = 0.80;
    }
    else { capsicumAmount = 0; }
    // Check to see if we have chosen to include Olives on our Pizza
    if ([toppingOlives isOn]){

```

```
        oliveAmount = 0.80;
    }
    else { oliveAmount = 0; }

    // Check to see if we have chosen to include Salami on our Pizza
    if ([toppingSalami isOn]){
        salamiAmount = 0.80;
    }
    else { salamiAmount = 0; }

    // Check to see if we have chosen to include Mozzarella on our
    Pizza
    if ([toppingMozarella isOn]){
        mozzarellaAmount = 0.80;
    }
    else { mozzarellaAmount = 0; }

    // Check to see if we have specified to have a Thin Crust Pizza.
    if ([crustThin isOn]){
        thinCrustAmount = 2.00;
    }
    else { thinCrustAmount = 0; }

    // Check to see if we have specified to have a Thick Crust
    Pizza.
    if ([crustThick isOn]){
        thickCrustAmount = 2.50;
    }
    else { thickCrustAmount = 0; }

    // Check to see if we have specified to have a Cheese Filled
    Crust
    Pizza.
    if ([crustCheeseFilled isOn]){
        cheeseCrustAmount = 3.00;
    }
    else { cheeseCrustAmount = 0; }

    // Calculate our total amount based on what has been chosen
    totalAmount = (tomatoAmount + onionAmount +
        capsicumAmount+oliveAmount +
        salamiAmount + mozzarellaAmount + thinCrustAmount +
        thickCrustAmount + cheeseCrustAmount);

    // Output the total amount to the screen.
    TotalPayment.text = [[NSString alloc]
        initWithFormat:@"%5.2f",totalAmount];
}
```

4. Our final step is to release the memory used by our view control objects which we have declared. Xcode 4 creates these for you automatically when you declare the outlets in your `PizzaOrdersViewController.h` interface file:

```
- (void)dealloc {  
    [toppingTomato release];  
    [toppingOnion release];  
    [toppingCapsicum release];  
    [toppingOlives release];  
    [toppingSalami release];  
    [toppingMozarella release];  
    [crustThin release];  
    [crustThick release];  
    [crustCheeseFilled release];  
    [TotalPayment release];  
    [Calculate release];  
    [super dealloc];  
}
```

The screenshot below shows our Pizza Orders application running within the iOS simulator with the output displaying the user's chosen Pizza selections:



What just happened?

In this section, we added the program logic to our Pizza Ordering application to calculate and display the total cost of the pizza based on the selections made by the user. When the **Calculate** button is pressed, a call is made to the `calculatePurchase` method which we created within our `PizzaOrdersViewController.m` implementation file. This method determines what options have been chosen, and then calculates the cost for each chosen topping and crust size before finally displaying the total cost of your customized pizza out to the `UITextField` control.

Implementing views

What are views comprised of? Views can be made up of one or more views, and these can also be created programmatically to be added as a sub-view. They can also contain different types of user interaction objects like Buttons, Switches, Text Fields, and so on. When views are loaded at runtime, they can create any number of objects, and even implement a basic level of interactivity on their own, for example, displaying a virtual keyboard when the user clicks on a text area/field control. Views are entirely independent of any application logic. By using this separation, it conforms to the core principles of the MVC design approach.

Implementing view controllers

View Controllers handle the interactions within a view and also establish the connection points for outlets and actions by using the `IBOutlet` and `IBAction` directives. These directives get added to your project's implementation file code. These directives will also need to be added to the headers of your view controller.

An outlet (`IBOutlet`) is basically a variable by which an object can be referenced. An example of this could be that you have created a field in your view to collect the user's name or e-mail address and you have created an outlet for this in your code called **userName**.

By using this outlet, you are able to access or change the contents of this field. Before you are able to use this field, a reference to this would need to be declared in the View Controller header file as follows:

```
IBOutlet UITextField *userName;
```

By declaring this reference, it allows you to use Interface Builder to visually connect the Views Text Field objects to the **userName** variable, so that your code can fully interact with the `TextField` object by changing its properties, calling its methods, and so on.

Before you can access the property of the username field, you will need to use the two important Objective-C **@Property** and **@Synthesize** directives which we need to declare in our code:

DIRECTIVE	DESCRIPTION
@Property	Declares elements in a class that should be exposed via the <i>getters</i> and <i>setters</i> . Properties are defined with a series of attributes, these being the more frequent nonatomic and retain methods.
@Synthesize	Creates simplified <i>getters</i> and <i>setters</i> , making retrieving and setting values of an object very simple.

Getters and Setters use a method called an *accessor*. An accessor is a method for getting or setting the value of an instance variable. An accessor that gets the instance variable's value is called a **getter**; an accessor that sets the instance variable's value is called a **setter**. A setter's name should start with set and be followed by a capitalized version of the instance variable's name. If the instance variable is named `myVar`, the setter should be named `setMyVar`. The setter should take one parameter: the new value to be assigned to the instance variable.

A getter should have the same name as the instance variable. If the instance variable is named `myVar`, the getter should be named `myVar`. (This will not cause you or the compiler any confusion, because variable names and method names are used in completely different contexts.)

An action (`IBAction`) on the other hand is basically a method defined within your code that is called when an event takes place. Objects such as Buttons and Switches can trigger actions when a user interacts with an event, such as touching objects on the screen:

```
- (IBAction)displayName:(id)sender {
    NSString *Welcome = [NSString alloc] initWithFormat:@"Hello
    %@", userName.text];
    lblOutput.text = Welcome;
    lblOutput.textColor = [UIColor blueColor];
}
```

You will notice that our function declaration contains a `sender` parameter with the type declared as `id`. We describe these in the table below:

DIRECTIVE	DESCRIPTION
Sender	This is basically a generic type of object that is used when we don't know the type of object that we will be working with.
id	This enables you to write code that does not tie itself to a specific type of class.

Time for action – declaring input field as a property of View Controller

If we wanted to declare the **userName** instance of the **UITextField**, we would do it as follows:

1. Declare it as a property in the header file of our View Controller as shown below:

```
@property (retain, nonatomic) NSString *userName;
```
2. We will then need to use the **@synthesize** directive in the implementation file of our View Controller class, to create the getters and setters as shown below:

```
@synthesize userName;
```
3. Once we have added these lines, we are then able to retrieve the value from our **userName** text property by using:

```
theUserName = userName.text;
```
4. If we wanted to assign something to this object, we can do so as follows:

```
userName.text = @"Joe Bloggs";
```

What just happened?

In this section, we looked at the steps required in declaring and using outlets to access or change the contents of form fields within the view. We then looked at the two important Objective-C directives: `@property` and `@synthesize` that need to be declared before we are able to communicate with the form controls to retrieve or set their values.

In the next section, we will start to create a view-based application to ask the user for their favourite color and then display this back to the user.

Creating a view-based application template

Whenever you create a View-based application project using Xcode, you will notice that it contains and automatically creates a single view-controller for you.

In previous chapters, we have made use of this template many times, but you may not have a clear understanding of the full inner workings under the hood. In the following section, we will be diving into the details and take a look at how everything hangs together to make your application work.

The project we will be creating will be a simple application that will present the user with a (**UITextField**) textbox for typing in some text and a (**UIButton**) button which will handle the processing. We will also be including a (**UILabel**) label control which will be updated based on the information the user enters into the textbox control.

Time for Action – creating the FavoriteColor application

Before we can proceed and create our `FavoriteColor` project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the **Device Family** drop-down.
5. Click on the **Next** button to proceed to the next step in the wizard.
6. Enter in **FavoriteColor** and then click on the **Next** button to proceed to the next step in the wizard.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

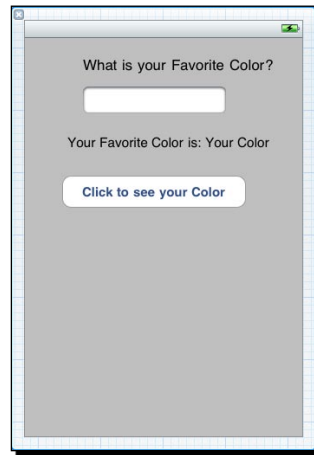
Once our project has been created, you will be presented with the Xcode interface along with all files that the project template created for you displayed within the Project Navigator window pane.

In this section, we will start to add the components required to build our application user interface that will ask the user for their favourite color. To refresh your memory, you can refer to the section *Adding Controls to your User Interface* which we covered in *Chapter 3, Working with the Interface Builder*:

1. From the Object Library, select and drag a (`UILabel`) control to the view and position the control at the top within our view. We will need to click on the **Object Attributes** tab and label our control **What is your Favorite Color?** by setting the `text` property under the `Label` section. Resize the control accordingly so that all of the text is displayed.
2. Next, we will need to add a (`UITextField`) control which will accept the information entered by the user. From the Object Library, select and drag a `UITextField` control and place it directly under our label control which we created in the previous step.

3. We now need to add another (UILabel) Label control which will be used to display the user's favourite color when the button is pressed. From the Object Library, select and drag a (UILabel) Label control and place this under the (UITextField) control which you created earlier on. We will need to click on the Object Attributes tab and label our control **Your Favorite Color is: Your Color** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
4. Our final step is to add a (UIButton) Round Rect Button control to our view. Modify the Object Attributes of the Round Rect Button control and set its title to read **Click to see your Color**.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In this section, we looked at how to create a view-based application and add the controls to our View Controller. We looked at how to go about adding a UILabel Control which will be used to display our question, and a UITextField control which will accept the input. We added another UILabel control which will display what the user entered, and finally a UIButton control which will handle the processing to display a message back to the user, **Your Favorite Color is;** and their entered color.

Time for action – binding our Controls using Outlets and Actions

Our final step will be to bind our control objects to our View Controller and connect these via outlets and actions. We covered this in the section *Making our Components work together in Chapter 3, Working with the Interface Builder*:

1. Once you have created your outlets within your `FavoriteColorViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller be aware of what controls we will be dealing with:

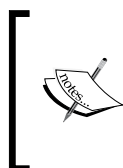
```
#import <UIKit/UIKit.h>

@interface FavoriteColorViewController : UIViewController {
    UITextField *txtColor;
    IBOutlet UILabel *lblColor;
}

@property (nonatomic, retain) IBOutlet UITextField *txtColor;

@end
```

2. Our next step is to create the action event in our `FavoriteColorViewController.m` implementation file. But we first need to synthesize our properties so that we can use them within our view controller.



If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "FavoriteColorViewController.h"

@implementation FavoriteColorViewController
@synthesize txtColor, lblColor;
```

3. Next, we declare our (`IBAction`) `ChosenColor` method which will be responsible for displaying the user's chosen color. What we are doing here is declaring an `NSString` object `ChosenColor` which contains a friendly message and the color that was entered.

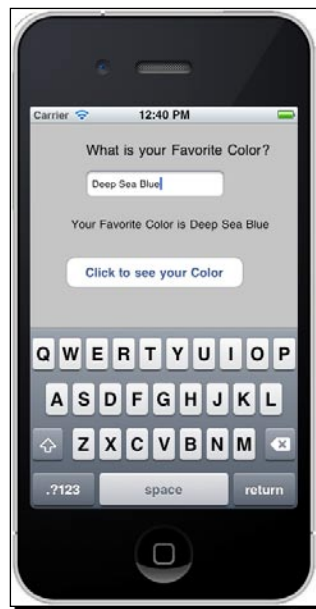
We then assign this to the `text` property of our (`UILabel`) `lblColor` control. The code snippet below shows the `ChosenColor` action event which will do just that:

```
- (IBAction)ChosenColor:(id) sender {
    NSString *ChosenColor = [[NSString alloc] initWithFormat:@"Your
        Favorite Color is %@",txtColor.text];
    lblColor.text = ChosenColor;
}
```

4. Our final step is to release the memory used by the view control objects which we declared. Xcode 4 creates these for you automatically when you declare the outlets in your `FavoriteColorViewController.h` interface file:

```
- (void)dealloc {  
    [txtColor release];  
    [lblColor release];  
    [super dealloc];  
}
```

The screenshot below shows our Favorite color application running within the iOS simulator with the output displaying the user's favorite color:



As you can see from the output shown above, we can create applications to communicate with the user and respond to what has been entered.

What just happened?

In this section, we looked at how we are able to bind our control objects to our view controller using outlets and actions. We also added some code to our interface and implementation files to build the application, and then declared an action event `ChosenColor`. This method is responsible for displaying the user's chosen color. In our final steps, we released the memory used by our view-controller objects via our `dealloc` method. In the next section, we will look at how we can implement Table views when space is a restriction within the view.

Implementing Table Views

Table Views make use of the `UITableView` class and are the main interface elements that are used to display lists and hierarchical data on the iPhone. Some examples of where Table Views are used on the iPhone are: *the Settings, Clock, Contact, and Notes*. These applications all use Tables Views as their main interface element.

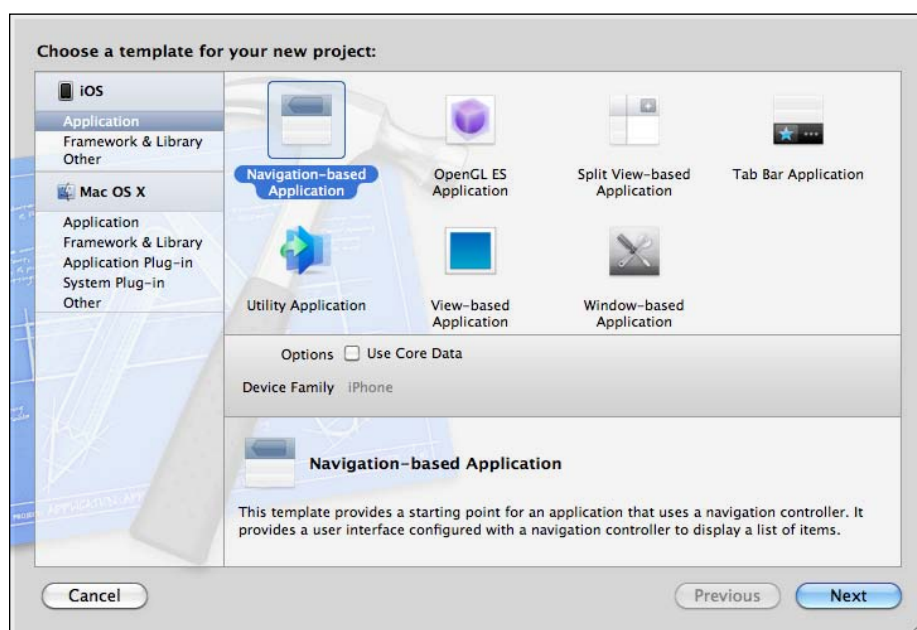
Table Views represent their data in rows and sections. A row is defined as an individual item which is stored in the table view. Rows can be grouped into sections and each section can contain both a header and footer.

Table view Controllers can be used to display data from a data source and can also be configured to update the data when editing the table view.

Time for action – creating a Table view application

Before we can proceed and create our `TableViewExample` project, to refresh your memory, you can refer to the section *Creating your first iPhone Application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the Navigation-based Application template from the list of available templates:



4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Once our project has been created, you will be presented with the Xcode interface along with all files that the project template created for you displayed within the Project Navigator window pane. You will notice that the Wizard created the following files which are described below:

COMPONENT NAME	DESCRIPTION
RootViewController.h	Interface header file which inherits from the UITableViewController object class and where declaration of the variables is done.
RootViewController.m	The implementation file which contains the methods pertaining to how the Table View Control operates and functions.
RootViewController.xib	This is the user interface file which contains the Table-View Control added by default.

The screenshot below displays the default table view control that Xcode creates for you when you create a Navigation-based application. The data is represented in rows, much like a spreadsheet and they can be grouped under section headings. They also can contain footers for each section:



We are now ready to start creating our sample application to add elements into our Table View Controller. We need to create an instance variable which will be used to store our table contents:

1. Open the `RootViewController.h` interface file, located within the `Classes` folder of your project and add the following code declaration:

```
#import <UIKit/UIKit.h>
@interface RootViewController : UITableViewController {
    NSArray *arrVegetables;
}
@end
```

2. Now that we have declared our **arrVegetables** array, we can start to add the elements to it. Open the `RootViewController.m` implementation file, located within the `Classes` folder of your project and locate the `viewDidLoad` method and add the following code:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.title = @"Vegetables";
    arrVegetables= [[NSArray
        alloc] initWithObjects:@"Celery",@"Capsicum",@"Potato",
        @"Peas",@"Broccoli",@"Carrots",@"Cabbage",nil];
}
```

3. We now need to edit the method `tableView:numberOfRowsInSection` to inform the Table View, how many rows of data it will need to display. We do this by returning the size of our vegetables array object, as shown in the code snippet below:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger) section {
    return [arrVegetables count];
}
```

4. Our next step is to update the `tableView:cellForRowAtIndexPath` method. We only need to implement the writing of the cell contents as most of the code is already done for us. All that we need to do is set the text for each cell using the data from our array. We do this by looking at the current row within our Table View and retrieve the array index location from our **arrVegetables** array by using the row index of the cell:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

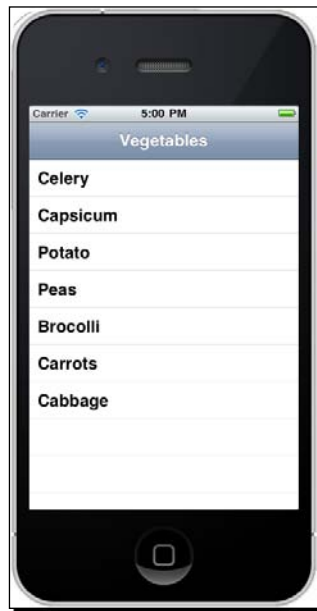
    if (cell == nil){
        cell= [[UITableViewCell
```



```
        alloc] initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier] autorelease];
    }
    // Configure our Table View Cell
    cell.textLabel.text=[arrVegetables objectAtIndex:indexPath.row];
    cell.textLabel.text = vegetables;
    return cell;
}
```

5. We are now ready to build and compile our TableViewExample application.

The screenshot below shows our TableViewExample displaying a list of Vegetables populated from our array to our Table View running within the iOS simulator:



What just happened?

In this section, we created a table view navigation-based application to allow us to represent our data in a list. We then created and initialized an array that was used to store the list of vegetables that will be used as the datasource to populate the table view. Next, we used the `numberOfRowsInSection` method of the `tableView` class to determine how many rows would need to be displayed within the table by using the `count` property of the array. In our last part, we called the `cellForRowAtIndexPath` method to update the table header and then cycled through every item within our `arrVegetables` array and outputted this at each row.

Grouping row items into sections

Table Views also give us the ability to group rows into sections. So for our next example, we will learn how this can be done. Grouping your rows into sections can be very useful when displaying sets of information relating to a particular type. For example, if you want to group country regions or breed of dogs, this can be very useful.

Time for action – grouping row items in our TableViewExample application

In our example, we will be grouping our previous example of vegetables into two groups, Organic and Non Organic:

1. Open our previously created TableViewExample application.
2. Open and modify the RootViewController.h interface file to include two additional arrays which will be used to hold our Organic and Non Organic items. Modify your RootViewController.h file as shown in the code snippet below:

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

@interface RootViewController : UITableViewController {
    NSArray *arrOrganic;
    NSArray *arrNonOrganic;
}
@end
```

3. Next, modify the RootViewController.m implementation file. We need to update the viewDidLoad method to include the following code snippet as shown below:

```
- (void)viewDidLoad {

    [super viewDidLoad];

    self.title = @"Vegetables";
    arrOrganic= [[NSArray
        alloc] initWithObjects:@"Celery",@"Capsicum",@"Brocolli",
        @"Cabbage",nil];
    arrNonOrganic= [[NSArray
        alloc] initWithObjects:@"Potato",@"Peas",@"Carrots",nil];
}
```

4. Next, update the method `tableView:numberOfRowsInSection` to automatically determine how many rows will be displayed within each section. We use the `switch` statement and return the number of items in our array for each:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section {
    switch (section)
    {
        case 0:
            return [arrOrganic count];
            break;
        case 1:
            return [arrNonOrganic count];
            break;
        default:
            return 0;
            break;
    }
}
```

5. We then need to update our `cellForRowAtIndexPath` method to use the correct array when you are populating the table view cells. This is shown in the code snippet below:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }
    NSString *vegetables;

    switch (indexPath.section)
    {
        case 0:
            vegetables = [arrOrganic objectAtIndex:indexPath.row];
            break;
        case 1:
            vegetables = [arrNonOrganic objectAtIndex:indexPath.row];
```

```
        break;
    }
    // Configure the cell.
    cell.textLabel.text=vegetables;
    return cell;
}
```

- 6.** In order to ensure that all of our sections are displayed in our Table View, we need to update our `numberOfSectionsInTableView` method to ensure that it returns the correct number of sections. Modify this method as shown in the code snippet below:

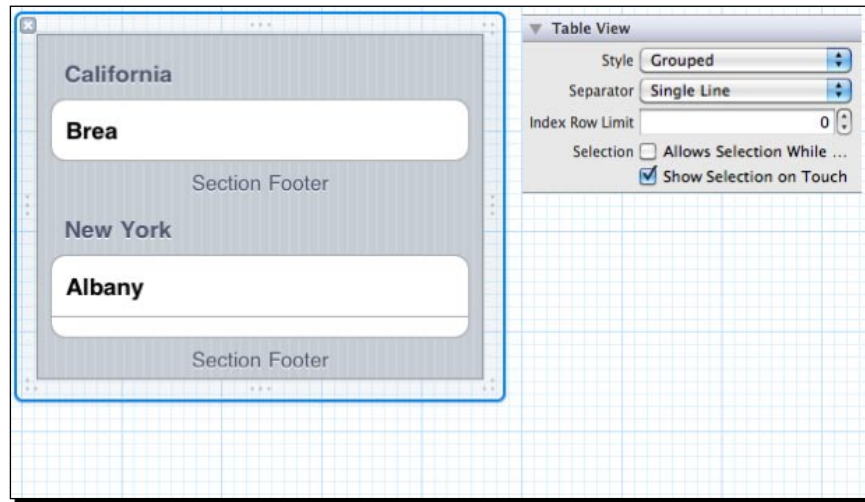
```
// Customize the number of sections in the table view.
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{

    return 2; // the number of sections our view contains
}
```

- 7.** We now need to implement the `titleForHeaderInSection` method which will be used to display the title text above each section:

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    switch (section)
    {
        case 0:
            return @"Organic";
            break;
        case 1:
            return @"Non Organic";
            break;
        default:
            return nil;
            break;
    }
}
```

8. We are nearly there; we just need to update our style for our Table View control. We can't modify this through code as this is a read-only property and prevents us from doing so, so we need to modify this using Interface Builder. The screenshot below shows how this can be done:



9. The above screenshot shows the **Object Attributes** screen. In order to set our style so that it displays our sections as groups, we need to set the **Style** property under the Table View section to show **Grouped** and change the **Separator** to display as **Single Line**.
10. We are now ready to Compile, Build and Run our modified TableViewExample application.

The screenshot below shows our modified TableViewExample application running within the iOS simulator and displaying a list of Vegetables populated from our array to our Table View and broken down into the two separate sections of Organic and Non Organic using our style which we applied to our Table View Control:



What just happened?

In the above section, we created two simple projects to write details to our Table View control, and break the items up into groups. We first created two arrays which will be used to hold our list of Organic and Non Organic vegetables. We then created an array for each of our sections which will be displayed to our Table View control. Next, we needed to check to see which section is being requested and then return the total number of rows to be displayed within the section. We use the **break** statement to prevent the section from falling through to the other parts within our `switch` statement. If we had excluded our `break` statement from the first part, we would have returned the Non Organic array count.

As you can see, by adding some minimal code to the `RootViewController.m` implementation file, we can create sophisticated applications that look professional and are easy to use.

Understanding Navigation-based applications

Navigation-based applications (as you just saw in the previous section) are a great tool for displaying information within lists and allowing the user to choose from the list. However, you will find as you start creating your own applications that Tables are rarely used on their own and are used in conjunction with a navigation controller to allow a user to *drill down* through multiple views of data, with the ability to navigate back and forth.

You may have seen, or used some navigation applications on the iPhone. A good example of where this is used is the Contacts application, where you can select from a group of individuals, then drill down to a specific person, and then view personal information about the person. There are also controllers which allow the user to navigate back to the previous level:



The implementation of the Navigation controller is pretty simple as you saw when you created the two `TableViewExample` applications. You are probably wondering *What happens when a new view is displayed?* Well, the navigation controller does what is known as "**push**" its view controller onto the stack. What this means is that a new instance of the controller is instantiated and added to the stack, and then the previous controller gets pushed further down the stack. A better visual representation which would help you understand this could be to think of a stack as a set of plates being placed on top of each other.

When the user decides it's time to return back to the previous screen, the navigation controller "**pops**" the current view off the stack which results in this being unloaded. The previous view controller then moves to the top of the stack and then this becomes active again, hence allowing the user to navigate onto another item.

Using Switches, Sliders, Segmented Controls, and Web Views

In the next section, we will be making use of the different types of controls available on the iPhone. We will be creating a simple application which will utilise each of these types and hopefully give you an insight into the power of each of these controls. But, first let us learn about them:

- ◆ **Switches:** Switches are used in most desktop applications and are often displayed as something being "active" or "inactive" either by checking or unchecking a check box, or by choosing from a list of radio buttons. On the iPhone, Apple provides you with similar controls, the Switches and Segmented controls Switches (`UISwitch`) are represented as a simple ON/OFF UI element, and return a Boolean value of `TRUE` or `FALSE`.
- ◆ **Sliders:** Sliders (`UISlider`) are a convenient control used to visually display a starting point within a range of values. You may have seen this control used within the iPod player on the iPhone, where you can increase or decrease the audible volume or move to a point within a song.
- ◆ **Segmented Controls:** Segmented (`UISegmentedControl`) controls present a linear line of buttons and are sometimes referred to as button bars, with a single button being active within the bar. These types of controls are frequently used to switch between different views within an application; these could be anything from configuration to a page of results.

If you wanted to determine the currently selected button within a segmented control, this is done via the `selectedSegmentIndex`, which returns the number of the button chosen (starting at 0 up to X).

- ◆ **Web Views:** Web Views (`UIWebView`) provide you with advanced features that allow you to present HTML, load web pages, and also incorporate pinching and zooming capabilities. Web Views offer support to display a wide range of file formats, as listed below:
 - HTML and CSS Files
 - Microsoft Word Documents
 - Microsoft Excel Spreadsheets
 - Apple's Keynote Presentations
 - Apple's Numbers Spreadsheet

- ❑ Apple's Pages Documents
- ❑ PDF Files
- ❑ PowerPoint Presentations

These files can be added as a resource file to your project and then displayed within the web view.

Time for action – creating the SwitchesSlidersSegments project

Before you can proceed and create your `SwitchesSlidersSegments` project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

- 1.** Launch Xcode from the `/Xcode4/Applications` folder.
- 2.** Choose **Create a new Xcode project**, or **File | New Project**.
- 3.** Select the View-based Application template from the list of available templates.
- 4.** Select **iPhone** from under the Device Family drop-down.
- 5.** Specify the location where you would like to save your project.
- 6.** Click on the **Save** button to continue and display the Xcode workspace environment.
- 7.** Specify the location where you would like to save your project.
- 8.** Click on the **Save** button to continue and display the Xcode workspace environment.

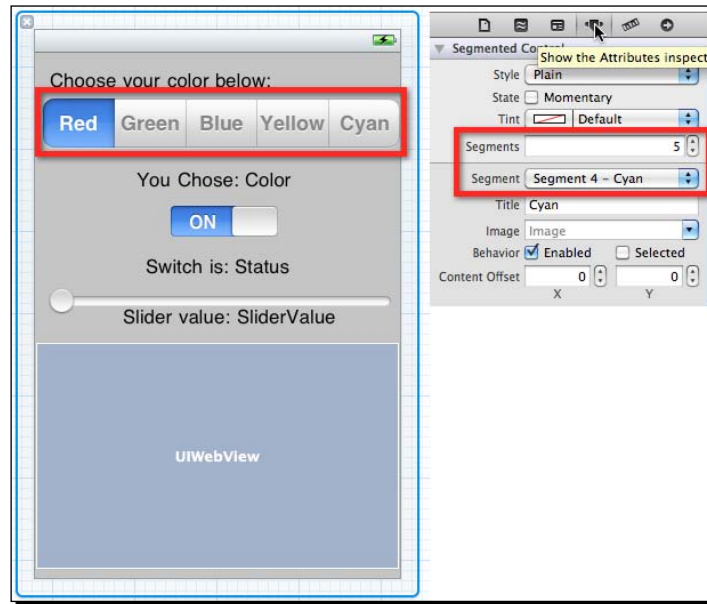
Once our project has been created, you will be presented with the Xcode interface along with all the files that the project template created for you displayed within the Project Navigator window pane.

In this section, we will start to add the components required to build our application user interface that will show how we can use switches, slider, and web view controls to display a web page within our view. To refresh your memory, you can refer to the section *Adding Controls to your User Interface* which we covered in *Chapter 3, Working with the Interface Builder*:

- 1.** From the Object Library, select and drag a (`UILabel`) label control to the view and position it at the top within our view. You will need to click on the **Object Attributes** tab and label the control **Choose your color below:** by setting the text property under the `Label` section. Resize the control accordingly so that all of the text is displayed.

2. Next, from the Object Library, select and drag a (UISegmentedControl) control and drag it to the view. From the Object Attributes tab, ensure that under the view section, the **Mode** has been set to read **Scale to Fill**.
3. Now we need to add another (UILabel) Label Control which will be used to display the color chosen by our segmented control. From the Object Library, select and drag a (UILabel) control and place this under our segmented control object. Click on the Object Attributes tab and label the control **You Chose: Color** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
4. We now need to add our (UISwitch) Switch component to our view. From the Object Library, select and drag a (UISwitch) control and place this under our **You Chose: Color** label control object.
5. Now we need to add another (UILabel) Label Control which will be used to display the status of our switch object. From the Object Library, select and drag a (UILabel) control and place this under our UISwitch control object. Click on the Object Attributes tab and label our control **Switch is: Status** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
6. Our next step is to add a (UISlider) slider control which will be responsible for displaying the current value of our slider out to our label control. From the Object Library, select and drag a (UISlider) control and place this directly under our **Switch is: Status** label control object.
7. We now need to add a (UILabel) Label control which will be used to display the value of where our slider object is up to. From the Object Library, select and drag a (UILabel) control and place this under our UISlider control object. Click on the Object Attributes tab and label our control **Slider value: SliderValue** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
8. Our final part will be to add our Web View (UIWebView) control which will be used to display our web page. From the Object Library, select and drag a (UIWebView) control and place this under our (UILabel) control object. Resize the control accordingly so that all of the text is displayed.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



The screenshot above shows you how you are able to increase and/or decrease the number of segments that the control can have by using the **Segments** increment, and the segment names can be changed by using the **Segment** drop-down from within the Objects Inspector pane.

What just happened?

In this section, we looked at how to add and make use of each of the controls available on the iPhone. We added a `UISegmentedControl` which allowed the user to make their selection from a list of colors. We then added a `UILabel` control to display the color chosen by the user. We then added a `UISwitch` control to determine if the switch is ON or OFF. Next, we added a `UISlider` control to be used to update the label to show its current slider value. Finally, we added a Web View control which was used to display the contents of a web page.

Time for action – binding our Controls using Outlets and Actions

Our final step will be to bind our control objects to our View Controller and connecting these via outlets and actions. We covered this in the section *Making our Components work together* in Chapter 3, *Working with the Interface Builder*:

1. Once you have created your outlets within your `SwitchesSlidersSegmentsViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller know what controls we will be dealing with:

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

@interface SwitchesSlidersSegmentsViewController :
UIViewController {

    IBOutlet UISegmentedControl *colorChoice;
    IBOutlet UISwitch *toggleSwitch;
    IBOutlet UIWebView *ourWebView;
    IBOutlet UILabel *toggleValue;
    IBOutlet UILabel *chosenColor;
    IBOutlet UILabel *sliderValue;
    IBOutlet UISlider *ourSlider;
}

@property (nonatomic, retain) IBOutlet UILabel *sliderValue;
@property (nonatomic, retain) IBOutlet UISlider *ourSlider;
@property (nonatomic, retain) IBOutlet UISegmentedControl
*colorChoice;
@property (nonatomic, retain) IBOutlet UISwitch *toggleSwitch;
@property (nonatomic, retain) IBOutlet UILabel *toggleValue;
@property (nonatomic, retain) IBOutlet UILabel *chosenColor;

@end
```

2. Our next step is to create the action event in our `SwitchesSlidersSegmentsViewController.m` implementation file, but we first need to synthesize our properties so we can use them within our view controller.



If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "SwitchesSlidersSegmentsViewController.h"

@implementation SwitchesSlidersSegmentsViewController
@synthesize ourSlider,colorChoice,toggleSwitch,sliderValue,
toggleValue,chosenColor;
- (IBAction)getSwitchValue:(id)sender {
// Determines the status of our switch
if ([toggleSwitch isOn]){
toggleValue.text=@"Switch is: ON";
} else {
toggleValue.text=@"Switch is: OFF";
}
}
- (IBAction)getSliderValue:(id)sender {
// Gets the current value from our slider control and
// displays it to our label.
sliderValue.text=[[NSString alloc]initWithFormat:@"Slider value:
%1.2f",ourSlider.value];
}
// Determines what colour has been selected, then changes the
background colour of our label.
- (IBAction)getColor:(id)sender {
// Get the currently selected item from our Segmented Control
switch (colorChoice.selectedSegmentIndex )
{
case 0:
chosenColor.backgroundColor=[UIColor redColor];
break;
case 1:
chosenColor.backgroundColor=[UIColor greenColor];
break;
case 2:
chosenColor.backgroundColor=[UIColor blueColor];
break;
case 3:
chosenColor.backgroundColor=[UIColor yellowColor];
}
```

```

        break;
    case 4:
        chosenColor.backgroundColor=[UIColor cyanColor];
        break;
    default:
        chosenColor.backgroundColor=[UIColor redColor];
        break;
}
chosenColor.text=[[NSString alloc]initWithFormat:@"You Chose:
%@", [colorChoice
    titleForSegmentAtIndex:colorChoice.selectedSegmentIndex]];
}

```

3. Next, add the necessary code to our `viewDidLoad` method which will be responsible for making our view work. We need to add the highlighted code into our `viewDidLoad` event which will do just that:

```

- (void)viewDidLoad {
    NSURL *appleUrl;
    appleUrl=[[NSURL alloc]initWithString:@"http://www.apple.com/"];
    [ourWebView loadRequest:[NSURLRequest requestWithURL:
appleUrl]];
    [super viewDidLoad];
}

```

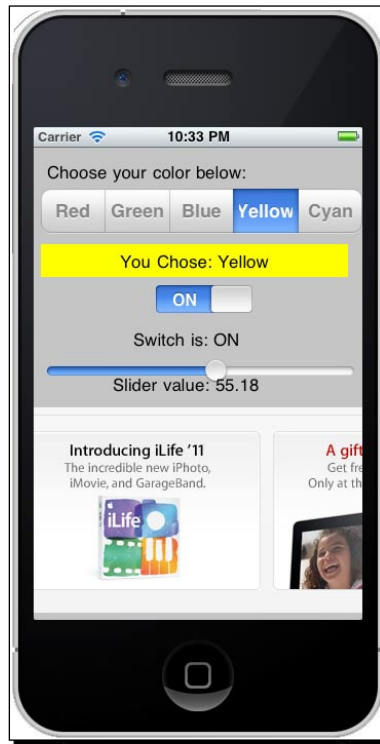
4. Our final step is to release the memory used by the view control objects which we declared. Xcode 4 creates these for you automatically when you declare the outlets in your `SwitchesSlidersSegmentsViewController.h` interface file:

```

- (void)dealloc {
    [toggleSwitch release];
    [colorChoice release];
    [ourWebView release];
    [sliderValue release];
    [toggleValue release];
    [chosenColor release];
    [ourSlider release];
    [super dealloc];
}

```

The screenshot below displays our application running within the iOS simulator, with each of the controls showing their output:



What just happened?

In this section, we added the program logic into our `SwitchesSlidersSegmentsViewController.m` implementation file to determine the current state of our switch object. We use the `isOn` property to determine the state our switch is in. If it is ON, we return the text *Switch is: ON*, otherwise, if it is OFF, we return the text *Switch is: OFF*. We then determine the current value of our slider control object. This is defined as using the `value` method of our `Slider` control. Once we have obtained this, we output the text to our **SliderValue** label.

In our next step, we need to find out the index of our **colorChoice** segmented control. If you remember, we mentioned that this control starts from **0** to **Total_No**. We use the `colorChoice.selectedSegmentIndex` method to derive this. Once we have got this information, we can set the background color of our `chosenColor` label control.

In our final step, we create an instance of the `NSURL` class which will be used to store the URL address for the Apple website. We then create an `NSURLRequest` object that we pass to the web view control when the view is loaded. We finally pass our `appleURL` to the `requestWithURL` method of our Web View control, which handles the displaying of the web page.

Creating an application to scroll through large content

Scrolling views can become very handy when space is limited within the view which you are working on. By using an instance of the `UIScrollView` class, you can add controls and interface elements to allow these to fit within the physical boundaries of the iPhone.

We will create a very basic scrolling view application to show how to implement this control. To enable scrolling within the view, we need to define a property called `contentSize`, which will be used to set the area of the content that needs to be scrolled both horizontally and vertically..

Time for action – creating the ScrollingViews project

Before we can proceed and create our `ScrollingViews` project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

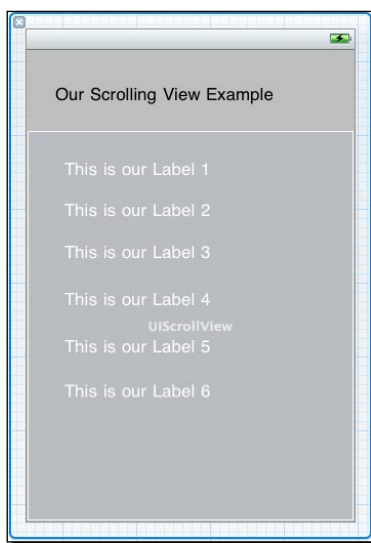
1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Once our project has been created, you will be presented with the Xcode interface along with all files that the project template created for you displayed within the Project Navigator window pane.

Now we need to add the controls to our View Controller. We will be adding a (UIScrollView) Scroll view Control which will be used as the container to handle the scrolling. We will also be creating six (UILabel) label controls which will be placed within the UIScrollView:

- 1.** From the Object Library, select and drag a (UILabel) label control to the view and position it at the top within our view. You will need to click on the Object Attributes tab and label our control **Our Scrolling View Example** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
- 2.** Next, from the Object Library, select and drag a (UIScrollView) control and drag it to the view. From the Object Attributes tab, ensure that from under the view section, the Mode has been set to read Scale to Fill.
- 3.** Now we need to add the (UILabel) Label Controls to our UIScrollView. From the Object Library, select and drag a (UILabel) control to the UIScrollView and position the control at the top within our Scroll view. Click on the Object Attributes tab and label our control **This is our Label 1** by setting the text property under the Label section. Resize the control accordingly so that all of the text is displayed.
- 4.** Repeat the above step, to add the remaining five (UILabel) label controls, positioning them apart from each other and setting each of their text properties to read **This is our Label No** where **No** is the label number.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In this section, we looked at how we can use scrolling views to handle our view when space becomes very limited. We used the `UIScrollView` class and added some label controls to show some of the basic features of scrolling within the view. In the next section, we will look at how we can bind the control to enable scrolling within the view.

Time for action – binding our Controls using Outlets and Actions

Our final steps will be to bind our control objects to our View Controller and connecting these via outlets and actions. We covered this in the section *Making our Components work together* in *Chapter 3, Introducing Interface Builder*:

1. Once you have created your outlets within your `ScrollingViewsViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller know what controls we will be dealing with:

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

@interface ScrollingViewsViewController : UIViewController {
    IBOutlet UIScrollView *ourScroller;
}
@property (nonatomic, retain) IBOutlet UIScrollView *ourScroller;

@end
```

2. Our next step is to create the action event in our `ScrollingViewsViewController.m` implementation file, but we first need to synthesize our properties so we can use them within our view controller.

[



If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

]

```
#import "ScrollingViewsViewController.h"

@implementation ScrollingViewsViewController
@synthesize ourScroller;
```

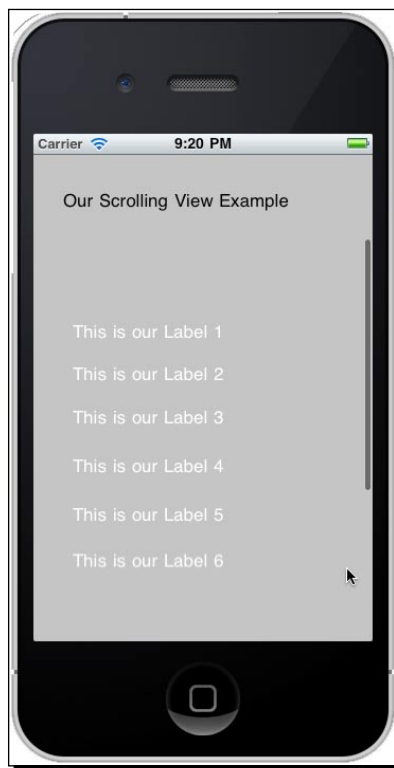
- 3.** Our next step is to add the necessary code to our `viewDidLoad` method which will be responsible for making our view work. We need to add the highlighted code into our `viewDidLoad` event which will do just that:

```
- (void)viewDidLoad {  
    ourScroller.contentSize=CGSizeMake(200,500);  
    [super viewDidLoad];  
}
```

- 4.** Our final step is to release the memory used by our view control objects which we have declared. Xcode 4 creates these for you automatically when you declare the outlets in your `ScrollingViewsViewController.h` interface file:

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

The screenshot below shows our `ScrollingViews` application running within the iOS simulator and displaying the scrollable content:



What just happened?

In this section, we declared some code within our `viewDidLoad` method to set the size of the content that will be scrolled up and down. Next, we set the region size to scroll horizontally and vertically. The `CGSizeMake(<width>, <height>)` function contains a `width` and a `height` property. We have told our scroll view (`ourScroller`) to scroll up **200** pixels horizontally and **500** pixels vertically.

Understanding Pickers

These types of controls implement the (`UIPickerView`) classes and are a unique feature on the iPhone and they present a series of multi-value options using a spinning interface—more like a rotating slot machine. The control comprises of segments which are referred to as components, and displays rows of values that the user can choose from. This control is frequently used on the iPhone when you need to set the date and/or time.

Date Pickers

The Date Picker (`UIDatePicker`) control implements the `UIDatePicker` class, and allows the user to specify a date. The control returns a date object of type `NSDate`.

In the example which will follow, we will get the user to select a date and then display this chosen value to a (`UILabel`) control. The `NSDateFormatter` object allows you to customize the output of how you would like your date to look. The table below shows you the different types of format which are available to you:

DATE FORMAT	DESCRIPTION
MMMM	Displays the full name of the month.
d	Displays the day of the month, with no leading zero.
YYYY	Displays the full four-digit year.
hh	Displays a two-digit hour (with leading zero if required).
mm	Displays the minutes with two digits.
ss	Displays the seconds with two digits.
a	Displays a.m. or p.m.

There may be times when you want to display your date picker in a mode other than default. Fortunately, the Date Picker can be customized to work in four different modes which are explained below:

MODE	DESCRIPTION
Date and Time	This option provides the ability to choose both a date and a time.
Time	Displays only time values.
Date	Displays only date Values.
Timer	This option displays a clock-like interface for choosing duration period.

Time for action – creating the Date Picker project

Before we can proceed and create our `DatePickersExample` project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

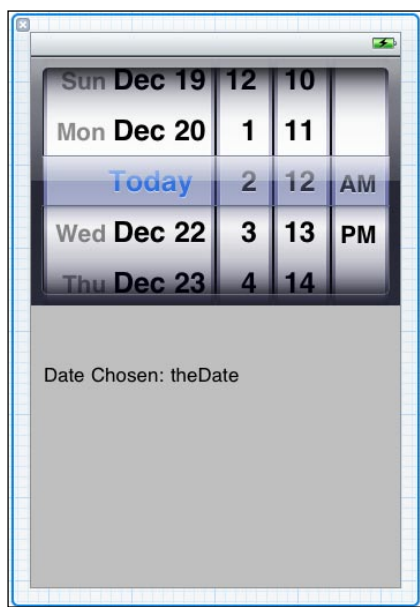
1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

Once our project has been created, you will be presented with the Xcode interface along with all the files that the project template created for you displayed within the Project Navigator window pane.

Now we need to add the controls to our View Controller. We will be adding a (`UIDatePicker`) control which the user can select the date from, as well as creating a (`UILabel`) label control which will be used to display the date selected:

1. From the Object Library, select and drag a (`UIDatePicker`) control to the view and position the control at the top within our view.
2. We now need to add a (`UILabel`) Label control. From the Object Library, select and drag a (`UILabel`) Label control and place this under the (`UIDatePicker`) control. We will need to click on the **Object Attributes** tab and label our control **Date Chosen: theDate** by setting the `text` property under the `Label` section. Resize the control accordingly so that all of the text is displayed.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In this section, we created a simple application that used the `UIDatePicker` control to allow the user to select a date and then display the chosen date to our label control `Date Chosen: theDate`. In the next section, we will look at how to bind the `UIDatePicker` control to handle the displaying of the chosen date.

Time for action – binding our Controls using Outlets and Actions

Our final steps will be to bind our control objects to our View Controller and connecting these via outlets and actions. We covered this in the section *Making our Components work together* in *Chapter 3, Introducing Interface Builder*:

1. Once you have created your outlets within your `DatePickersExampleViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller know what controls we will be dealing with:

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

@interface DatePickersExampleViewController : UIViewController
{
```

```
        UIDatePicker *theDate;
        UILabel *ourLabel;
    }
    @property (nonatomic, retain) IBOutlet UIDatePicker *theDate;
    @property (nonatomic, retain) IBOutlet UILabel *ourLabel;

@end
```

2. Our next step is to create the action event in our `DatePickersExampleViewController.m` implementation file, but we first need to synthesize our properties so we can use them within our view controller.

 If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "DatePickersExampleViewController.h"

@implementation DatePickersExampleViewController
    @synthesize theDate, ourLabel;
```

3. Next, we declare our (`IBAction`) `getDate` method which will be responsible for displaying the selected date to our Label control object. The code snippet below shows the `getDate` action event which will do just that:

```
- (IBAction)getDate:(id)sender {
    NSString *dateChosen;
    NSDateFormatter *dateFormat;

    dateFormat=[[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"MMMM d, yyyy hh:mm:ssa"];

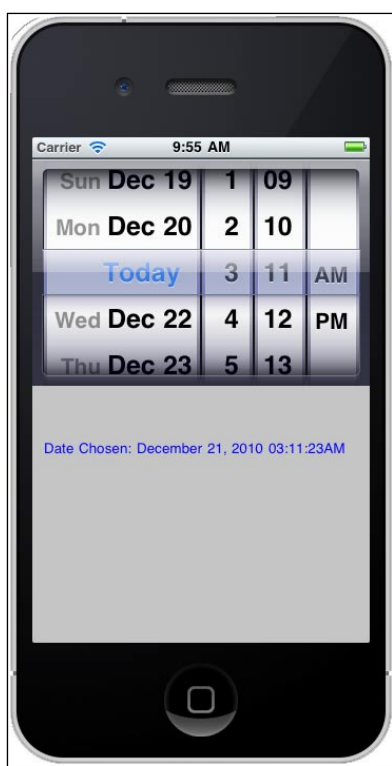
    dateChosen=[[NSString alloc] initWithFormat:@"Date Chosen:
%@", [dateFormat stringFromDate:[sender date]]];

    ourLabel.text=dateChosen;
    ourLabel.textColor=[UIColor blueColor];
}
```

4. Our final step is to release the memory used by our view control objects which we declared. Xcode 4 creates these for you automatically when you declare the outlets in your `DatePickersExampleViewController.h` interface file:

```
- (void)dealloc {  
    [theDate release];  
    [ourLabel release];  
    [super dealloc];  
}
```

The screenshot below shows our `DatePickersExample` application running within the iOS simulator and displaying the `UIDatePicker` control with a date selected and displayed:



What just happened?

In this section, we created an action method `getDate` that will be responsible for displaying the selected date from our `UIDatePicker` control. Within this function, we needed to specify the date format to use by using the `NSDateFormatter` object and apply this to the date returned by the sender object. We then finally output the date to our label control and set the foreground color to be Blue using the `UIColor` object.

Custom Pickers

As mentioned previously, we will be implementing a custom picker example later on in this chapter and making use of the (UIPickerView) object. You will notice that when you initially use this object, it starts off with a basic spinning view and all that we need to do is provide it with the data to display and describe how it should be displayed by setting its DataSource and delegate properties to point to our View Controller.

Time for Action – creating the Custom Picker project

Before we can proceed and create our CustomPickers project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the /Xcode4/Applications folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

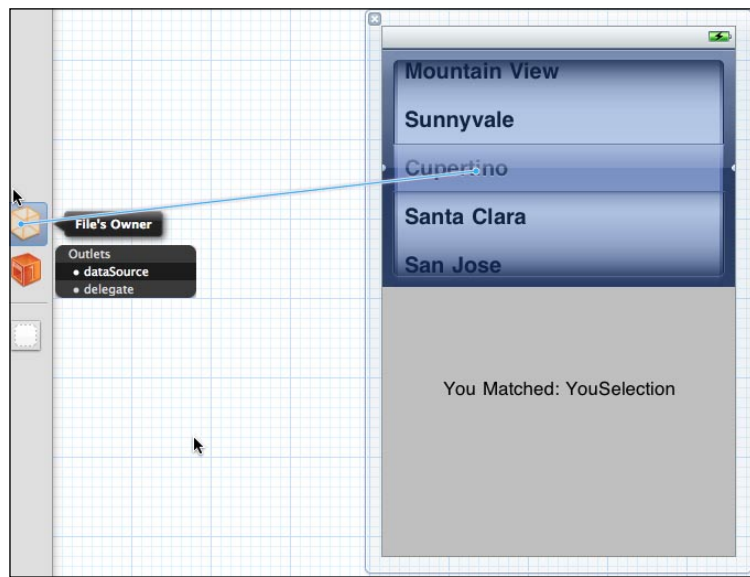
Once our project has been created, you will be presented with the Xcode interface along with all the files that the project template created for you displayed within the Project Navigator window pane.

Now we need to add the controls to our View Controller. We will be adding a (UIPickerView) control which the user can use to select the animal and associated sound, as well as create a (UILabel) label control which will be used to display the date selected:

1. From the Object Library, select and drag a (UIPickerView) control to the view and position the control at the top within our view.
2. We now need to add a (UILabel) Label control which will be used to display the animal selected from our Picker View. From the Object Library, select and drag a (UILabel) Label control and place this under the (UIPickerView) control.

3. Next, click on the **Object Attributes** tab and label the control **You Matched: YouSelection** by setting the `text` property under the **Label** section. Resize the control accordingly so that all of the text is displayed.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



4. Next, we need to connect our UIPickerView object control to the `dataSource` and `delegate` Objects. This is to conform to the UIPickerViewDataSource and UIPickerViewDelegate protocols and we will need to modify our `CustomPickersViewController.h` interface class so that our picker can use the methods of the UIPickerView class.

What just happened?

In this section, we created a simple application to use the UIPickerView control which will be used to display a list of animals and the noises that they make. We also added a UILabel control to our view which will be used to show the correct association between the animal and the noise. Next, we looked at how to connect our UIPickerView control to the datasource that will be used to display our information. In the next section, we will look at how to go about populating our control.

Time for action – binding our Controls using Outlets and Actions

Our final steps will be to bind our control objects to our View Controller and connecting these via outlets and actions. We covered this in the section *Making our Components work together* in *Chapter 3, Working with the Interface Builder*:

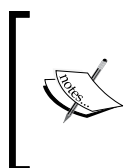
1. Create the necessary outlets within your `CustomPickersViewController.h` interface file. Once you have done this, it should look something like this. You will notice that we have additional classes of our `UIViewController` class; this is why we had to link our `UIPickerView` up to our `Datasource` and `Delegate` objects. We are also creating two array objects which will hold our animal types and the noises that they make:

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

@interface CustomPickersViewController : UIViewController
<UIPickerViewDataSource, UIPickerViewDelegate> {
    NSArray *animalType;
    NSArray *animalNoise;
    UILabel *matchResult;
}
@property (nonatomic, retain) IBOutlet UILabel *matchResult;

@end
```

2. Next, create the action event in our `CustomPickersViewController.m` implementation file, but we first need to synthesize our properties so that we can use them within our view controller.



If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "CustomPickersViewController.h"

@implementation CustomPickersViewController
    @synthesize matchResult;
```

3. Our next step is to add the necessary code to our `viewDidLoad` method which will be responsible for making our custom picker control work. We need to add the highlighted code to our `viewDidLoad` event which will do just that:

```
- (void)viewDidLoad {
    animalType= [ [NSArray
```

```

        alloc] initWithObjects:@"Dog",@"Cat",@"Pig",@"Mouse",
        @"Snake",nil];

        animalNoise= [NSArray
alloc] initWithObjects:@"Sssss",@"Squeak",@"Oink",@"Meow",
@"Woof",nil];
        [super viewDidLoad];
    }

```

- 4.** Our next step is to declare the following method calls to our (UIPickerView) control object:

```

- (NSInteger)numberOfComponentsInPickerView:(
    UIPickerView *)pickerView {
    return 2; // determines how many sections our picker will
    be using.
}
- (NSInteger)pickerView:(UIPickerView *)pickerView numberOfRows
    InComponent:(NSInteger)component {
    if (component == 0) {
        return [animalType count];
    } else {
        return [animalNoise count];
    }
}
- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:
    (NSInteger)row forComponent:(NSInteger)component
{
    if (component == 0) {
        return [animalType objectAtIndex:index:row];
    } else {
        return [animalNoise objectAtIndex:index:row];
    }
}

```

- 5.** Our next step is to declare our (void) pickerView method which will be responsible for displaying the selected animal type and the noise that each makes to our Label control object. The code snippet below shows the pickerView method which will do just that:

```

- (void)pickerView:(UIPickerView *)pickerView didSelectRow:(
    NSInteger)row inComponent:(NSInteger)component {

    NSString *matchedType;

    int selectedType;
    int selectedNoise;
    int matchedNoise;

```

```
        selectedType = [pickerView selectedRowInComponent:0];
        selectedNoise = [pickerView selectedRowInComponent:1];

        matchedNoise = ([animalNoise count]-1)-[pickerView
selectedRowInComponent:1];

        if (selectedType == matchedNoise) {
            matchedType=[NSString alloc] initWithFormat:@"You are
correct, a
%@ does go '%@'!",
                [animalType objectAtIndex:selectedType],
                [animalNoise objectAtIndex:selectedNoise]];
        } else {
            matchType=[NSString alloc] initWithFormat:@"You are
incorrect, a
%@ does not go '%@'!",
                [animalType objectAtIndex:selectedType],
                [animalNoise objectAtIndex:selectedNoise]];
        }

        matchResult.text = matchType;
        matchResult.textColor=[UIColor redColor];

        [matchType release];
    }
}
```

- 6.** Our final step is to release the memory used by our view control objects which we have declared. Xcode 4 creates these for you automatically when you declare the outlets in your CustomPickersViewController.h interface file:

```
- (void)dealloc {
    [animalType release];
    [animalNoise release];
    [super dealloc];
}
```

The screenshot below shows our `CustomPickers` example application running within the iOS simulator and displaying the selected animal and the associated noise that it makes:



What just happened?

In this section, we set up two array objects that will be used to hold the animal types and the sounds that they make. Next, we set up the number of sections that our custom picker control will have, as well as the number of rows for each section based on the number of items within each of the arrays. This information is then populated into each section.

Next, we defined three integers (**`selectedType`**, **`selectedNoise`**, and **`matchedNoise`**). These will be used to hold what has been currently selected by the user, using the formula below:

```
matchedNoise = ([animalNoise count] - 1) - [pickerView  
selectedRowInComponent:1];
```

What this formula does is determine how many **animalNoise** items we have in our array, and then minuses one from it (*as arrays are zero based*). We then determine the element position which the user has selected within the array. Next, we do a comparison and if the **animalType** which has been selected matches the position in our **animalNoise** array, we display the message **You are correct**; otherwise, we display **You are incorrect** along with the chosen animal type and the associated sound that they make. We then write this out to our (UILabel) control and set the foreground text color to Red. We then release the object when we are done.

Handling basic user input and output

The iPhone provides us with many different ways in which we can display information to the user, as well as the different ways in which we can collect this information. In the next section, we will build a simple application, making use of the following controls. First, let us learn about the basic controls available for collecting information and displaying this information to the user.

Button Controls

One of the most common interactions you will have when developing your **iOS** applications, is responding to a user request when they touch on the (UIButton) button control. You have used buttons in previous chapter exercises, these are elements of a view that respond to an event that the user triggers from within the user interface by usually using the "Touch Up Inside" event which indicates that the user has pressed and released this button. This then triggers action (IBAction) event which fires off the associated code attached to the action.

Text Fields

Text Fields (UITextField) provide the ability to accept input from the user. This can be information requesting their name or address information. Using the various different keyboard layouts that the iOS SDK provides you enables you to constrain the type of input allowable.

Text Fields are very similar to Buttons in the sense that they also have the ability to allow you to respond to events while the user is entering in information, but are mainly used to read information from these fields by using the controls `text` property.

Text Views

The Text View (UITextView) control is very similar to the Text Field, except that the Text View control presents a scrollable and editable block of text and gives the user the ability to either read or modify the contents. These types of controls are mainly used when more than a few words of input are required.

Text Views don't provide any support for automatically reducing the size of the font like you can do with Text Fields. They also don't provide support for clearing the text, other than through programmatically setting the `text` property. Also, if you try to apply a style to this control, all text will contain the same style. Apple recommends that if you need to handle your text differently, you should use the `UIWebView` control as an alternative.

Labels

Labels (`UILabel`) are very useful elements and are used to display strings within the view by setting their `text` property. Label controls contain a wide range of additional properties which control how the contents of the label will look, such as the Font and size of the text, the alignment, and setting the background and foreground color.

Using Text Fields, Text Views, and Buttons

You have already made use of the Text Fields and Button controls in previous chapter examples, but we have never touched upon the use of `TextViews` up until now. We will start by creating a very simple application that will show how to use each of these controls.

Our application will contain three (`UITextField`) `TextField` input boxes, a (`UIButton`) button control which will process the input fields, and a (`UITextView`) control which will be used to output the contents.

Time for action – creating application with Text fields, Text Views, and Buttons

Before we can proceed and create our `TextViewsandButtons` project, to refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Select **iPhone** from under the Device Family drop-down.
5. Specify the location where you would like to save your project.
6. Click on the **Save** button to continue and display the Xcode workspace environment.
7. Specify the location where you would like to save your project.
8. Click on the **Save** button to continue and display the Xcode workspace environment.

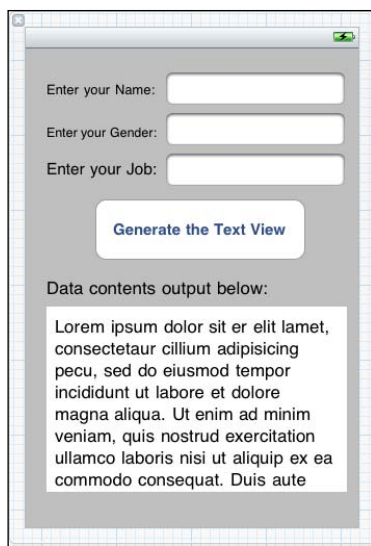
Once our project has been created, you will be presented with the Xcode interface along with all files that the project template created for you displayed within the Project Navigator window pane.

Now we need to add the controls to our View Controller. We will be adding a series of (UILabel) label controls which will be used display each of the labels as well as the associated (UITextField) textboxes which will be used to gather the information.

We will also be adding a (UIButton) control which will handle the processing, and finally, we will be adding a UITextView control which will be used to display the output of the fields entered by the user:

1. From the Object Library, select and drag a (UILabel) control to the view and position the control at the top within our view. We will need to click on the `Object Attributes` tab and label our control **Enter your Name:** by setting the `text` property under the `Label` section. Resize the control accordingly so that all of the text is displayed.
2. Next, we will need to add a (UITextField) control which will be accepting the information entered by the user. From the Object Library, select and drag a UITextField control and place it directly under the label control which we created in the previous step.
3. Repeat the steps above and add the (UILabel) labels and (UITextFields) Text Fields for adding the following: **Enter Your Gender** and **Enter Your Job**. Ensure that you leave enough space between each object.
4. Our next step is to add a (UIButton) Round Rect Button control to our view. Modify the Object Attributes of the Round Rect Button control and set its title to read **Generate the Text View**. Don't forget to resize the control so that all of the text can be read easily.
5. We now need to add another (UILabel) Label control which will be used as a heading placeholder for the UITextView control. From the Object Library, select and drag a (UILabel) Label control and place this under the (UIButton) control which you created earlier on. We will need to click on the `Object Attributes` tab and label our control **Data contents output below:** by setting the `text` property under the `Label` section. Resize the control accordingly so that all of the text is displayed.
6. Our final step is to add a (UITextView) control to our view. Adjust the size to how large or small you would like the control to be.

If you have followed the steps correctly, your view should look something like the screenshot below. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In this section, we looked at how to create a simple application to use a number of iPhone controls to handle input and display this information into a `TextView` control. The `TextView` control is a fantastic control as it supports text coloring and multiple lines, which are great for displaying information about people. In the next section, we look at how to add the program logic to make our application work.

Time for action – binding our Controls using Outlets and Actions

Our final steps will be to bind our control objects to our View Controller and connect these via Outlets and Actions. We covered this in the section *Making our Components work together* in *Chapter 3, Working with the Interface Builder*:

1. Once you have created your outlets within your `TextViewsandButtonsViewController.h` interface file, it should look something like this. What we are doing here is basically letting our View Controller know what controls we will be dealing with:

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

@interface TextViewsandButtonsViewController : UIViewController
{
    IBOutlet UITextField *txtName;
    IBOutlet UITextField *txtGender;
    IBOutlet UITextField *txtJob;
}
```

```
IBOutlet UITextView *txtTextView;
}
@property (nonatomic, retain) IBOutlet UITextField *txtGender;
@property (nonatomic, retain) IBOutlet UITextField *txtName;
@property (nonatomic, retain) IBOutlet UITextField *txtJob;
@property (nonatomic, retain) IBOutlet UITextView *txtTextView;

@end
```

2. Our next step is to create the action event in our `TextViewsandButtonsViewController.m` implementation file; but we first need to synthesize our properties so that we can use them within our view controller.

 If we don't declare these, we will receive warning messages which could result in unexpected results occurring in our application. It is also good to declare these as we are able to deallocate the memory used by these objects in our **dealloc** method.

```
#import "TextViewsandButtonsViewController.h"

@implementation TextViewsandButtonsViewController
@synthesize txtGender, txtName, txtJob, txtTextView;
```

3. Next, we declare our (IBAction) `generateTextView` method which will be responsible for displaying the contents of each of our text fields to the Text View Control. The text view control is a great control as it supports the MultiLine feature which allows text to be formatted nicely. The code snippet below shows the **generateTextView** action event which will do just that:

```
- (IBAction)generateTextView:(id)sender {
    NSString *TextView = [[NSString alloc] initWithFormat:@"Name:
    %@\nGender: %@\nJob: %@\n",txtName.text,txtGender.text,
    txtJob.text ];

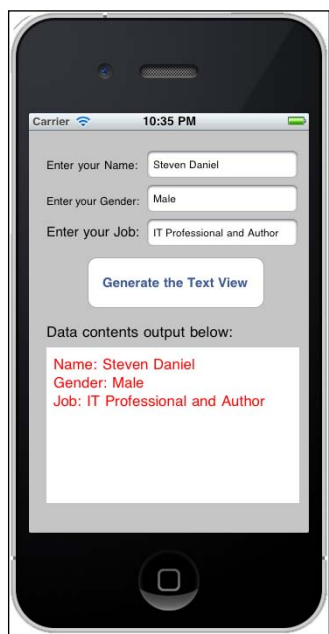
    txtTextView.text=TextView;
    txtTextView.textColor=[UIColor redColor];
}
```

4. Our final step is to release the memory used by the view control objects which we declared. Xcode 4 creates these for you automatically when you declare the outlets in your `TextViewsandButtonsViewController.h` interface file:

```
- (void)dealloc {
    [txtName release];
    [txtGender release];
```

```
[txtJob release];  
[txtTextView release];  
[super dealloc];  
}
```

The following screenshot shows our `TextViewandButtons` example application running within the iOS simulator and displaying the entered information from each of the form controls within the `Textview` control:



What just happened?

In this section, we looked at how to create the necessary outlets to our controls within our `TextViewsandButtonsViewController.h` interface file to make our View Controller aware of what controls we will be dealing with.

Next, we declared an action `generateTextView` method that is responsible for displaying the contents of each of our text fields to the `Text View` Control. This control is a great control as it supports the `MultiLine` feature which allows text to be formatted nicely. Finally, we declared an `NSString` object `TextView` which contains each of our form fields nicely separated by a new line and the data associated with each control. We then assign this to the `text` property of our (`UITextView`) `txtTextView` control and then set the foreground text color to be red.

Have a go hero – modifying the Table View example

Now that you have a good working knowledge of Table Views and know how to go about adding entries and sections, the task will be to modify our Table View example sections project to include a list of Organic and Non-Organic fruits. These will also need to be grouped into their appropriate sections:

1. Create two new objects that will be used to store each of our different types of fruits. You can refer to the section *Implementing Table Views* located in this chapter.
2. Next, implement two new arrays that will be used to hold our Organic and Non-Organic fruits. You can refer to the section *Implementing Table Views* located in this chapter.
3. Next, locate the **numberOfRowsInSection** method and create two new `case` statements that will be used to handle our new array objects. You can refer to the section *Implementing Table Views* located in this chapter.
4. Next, locate the **cellForRowAtIndexPath** method and create two new `case` statements for our two new array objects. You can refer to the section *Grouping our Row items into Sections* located in this chapter.
5. Next, locate the **numberOfSectionsInTableView** method and update the number of sections to handle the additional sections. You can refer to the section *Grouping our Row items into Sections* located in this chapter.
6. Next, we need to locate the **titleForHeaderInSection** method and add the headers for Organic and Non-Organic fruits. You can refer to the section *Grouping our Row items into Sections* located in this chapter.
7. Once you are satisfied that everything has been completed, you can Compile, Build and Run the application.

Once you have implemented the above, you will have a fully customized application that will display Organic and Non-Organic Vegetables and Fruits.

Pop quiz – Table Views / repositioning Controls

1. What method of the `tableView` should you update to initialise how many rows a section would contain?
 - a. `cellForRowAtIndexPath`
 - b. `numberOfRowsInSection`

2. Which set of Xcode templates contains a default `RootViewController.xib` file?
 - a. Window-based Application
 - b. View-based Application
 - c. Navigation-based Application
 - d. All of the above
3. If you wanted to reposition each of the controls within the view, what section of the Object Properties would you need to change?
 - a. Identity Inspector
 - b. Object Attributes Inspector
 - c. Size Inspector
4. What option would you need to set to make your table view appear as grouped?
 - a. Single Line
 - b. Grouped
 - c. All of the above

Summary

In this chapter, we had some insight into what design patterns are and their importance when developing iPhone applications that use the Model-View-Controller (MVC) application design, and how Xcode and Interface Builder implement MVC.

We also took a look at the different types of templates, the View-Based template and the Navigation-based template and created some simple applications to highlight each of these. We also used some of the different controls which we hadn't got round to using in previous chapters. We looked at how to use Switches, Sliders, Segmented Controls, Scrolling View, and Web Views. We also learned how to use the various Pickers, as well as creating our own custom-built picker control using the `UIPickerView` control to handle multiple selections.

We finished up the chapter by looking into how to handle basic user input and output as well as learning how to use the Text Field, Text View, and Button control objects.

We have learned about the various layers of MVC, the design patterns, and the importance of using the Model-View-Controller when creating iPhone applications, through using Table Views, Switches, Segmented Controls, and custom pickers. We are now ready to start focusing on how to get the user's attention through the use of the different notification methods.

In the next chapter, we will be taking a look into interacting with the user through the various notification methods of generating Alerts, and using Action Sheets. We will also be looking into how to handle alerts using sound and vibrations.

6

Displaying Notification Messages

In this chapter, we will be focusing on the different methods in which we can make our applications communicate and grab the user's attention. You may, for instance, want to notify the user that an error has occurred, or that the user will need to wait while information is being retrieved or saved.

The iPhone provides developers with many ways in which they can add informative messages to their applications to alert the user. We will be looking at the various types of notification methods, ranging from alerts, activity indicators, audio sounds, and vibrations.

We will be taking a look at these in more detail through each of the examples, which we will be building throughout this chapter.

In this chapter, we will:

- ◆ Explore and use the different notification methods
- ◆ Learn how to generate alerts to notify the user
- ◆ Learn how to go about using action sheets to associate with views
- ◆ Handle alerts via sounds and vibrations

We have got quite a bit to cover, so let's get started.

Exploring the notification methods

You will have noticed by now, that applications on the iPhone are user-centric, meaning that they don't operate without a user interface and don't perform any background operations.

These types of applications enable users to work with data, play games, or communicate with other users. Despite these, at some point an application will need to communicate with the user. This can be as simple as a warning message, or providing feedback or even asking the user to provide some information.

The iPhone and Cocoa-Touch use three special methods to gain your attention and are explained below:

CLASS	DESCRIPTION
UIAlertView	This class creates a simple modal alert window that presents the user with a message and a few options. Modal elements require the user to interact with them before they can proceed. These types of elements are displayed (layered) on top of other windows and block the underlying objects until the user responds to one of the actions presented.
UIActionSheet	These types of classes are similar to the UIAlertView class, except that they can be associated with a given view, tab bar, or toolbar and become animated when it appears on the screen. Action Sheets do not have an associated message property; they contain a single title property.
System Sound Services	This enables playback and vibration and supports various file formats (CAF, AIF, and WAV Files) and makes use of the AudioToolBox framework.

Generating alerts

There is no doubt that you will need to incorporate alerts into your applications. These can be very useful to inform the user of when the application is running, and can be a simple message such as memory running low, or that an application or internal error has occurred. We can notify the user in a number of ways using the **UIAlertView** class, and it can be used to display a simple modal message or gather information from the user.

Time for action – creating the *GetUsersAttention* application

Before we can proceed with creating our *GetUsersAttention* application, we must first launch the Xcode development environment. If you need to refresh your memory on how to go about creating a new Xcode project, you can refer to the section *Creating the Project* in *Chapter 3, Introducing Interface Builder*:

1. Select the View-based application template from the project template dialog box.
2. Ensure that you have selected **iPhone** from under the Device Family dropdown, as the type of view to create.
3. Next, you will need to provide a name for your project.
4. Enter **GetUsersAttention** and then choose a location where you would like to save the project.

Once your project has been created, you will be presented with the Xcode interface, along with the project files that the template created for you within the **Project Navigator** Window.

What just happened?

In this section, we looked at the steps involved in creating a View-based application for our `GetUsersAttention` application. In the next section, we will take a look at how we can add the `AudioToolbox` Framework into our project to incorporate sound.

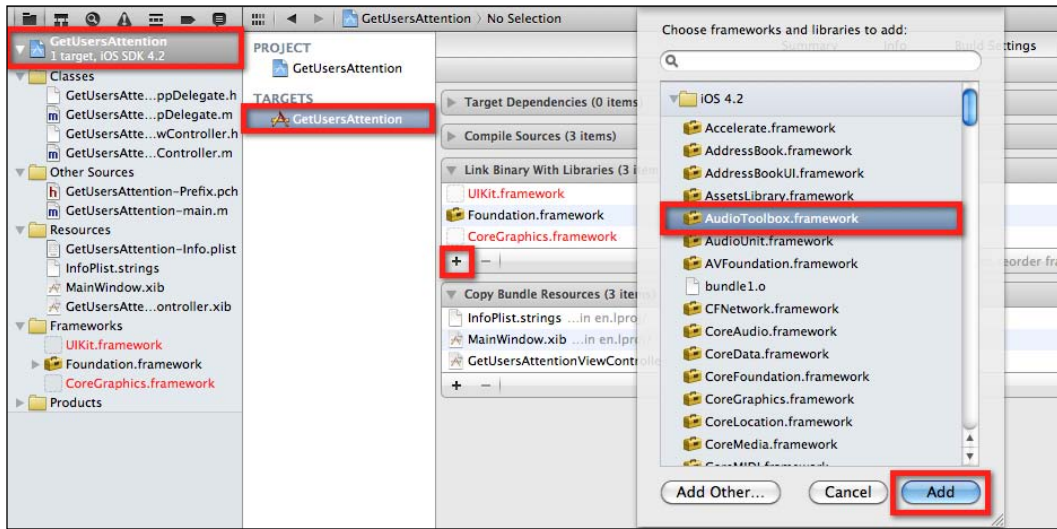
Time for action – adding the AudioToolbox Framework to our application

Now that we have created our project, we need to add the `AudioToolbox` Framework to our project. This is an important framework which will provide us the ability to play sound and vibrate the phone. It is similar to the `MediaPlayer` Framework that we used in previous chapters, but only supports a limited number of audio file formats.

To add the new frameworks or additional frameworks into your project, select the Project Navigator Group, and then follow these simple steps as outlined below:

1. Select your Project within the **Project Navigator** Window.
2. Then select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Library with Libraries** disclosure triangle.
5. Then finally, use the **+** button to add the library that you want to add; if you want to remove a framework, highlight it from the group and click on the **-** button. You can also search for the framework if you can't find it in the list shown.

If you are still confused on how to go about adding these frameworks, refer to the following image, which highlights what parts you need to select (*highlighted by a red rectangle*):



What just happened?

In the above section, we looked at how we are able to add frameworks to our application. We looked at the differences between the MediaPlayer and AudioToolbox frameworks, and the limitations of the two.

Adding frameworks to your application allows you to extend your application and utilise those features in your application to avoid reinventing the wheel. When you add frameworks to your application, the system loads them into memory as needed and shares the one copy of the resource among all applications whenever possible.

Now that we have added the `AudioToolbox.framework` to our project, our next step is to start creating our user interface. In the next section, we will be taking a look at how we start to build our user interface and create events.

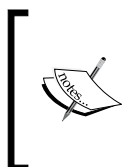
Pop quiz – Frameworks

1. Which framework allows you to vibrate the phone and play sound?
 - a. MediaPlayer
 - b. AudioToolbox
 - c. CoreAudio

2. Which framework allows for playing a limited number of file formats?
 - a. AVFoundation
 - b. CoreAudio
 - c. MediaPlayer
 - d. AudioToolbox
3. Under which tab is the **Link Binary with Libraries** section located?
 - a. Build Settings
 - b. Build Rules
 - c. Summary
 - d. Build Phases

Building our user interface

User interfaces provide a great way to communicate with the user in order to either obtain information or to display notifications. A good interface is one that provides a good consistent flow throughout your application as you navigate from screen to screen. This involves considering the screen size of your view. In the next section, we look at how to add some controls to our view to build our interface.



To obtain further information about what constitutes a good user interface, Apple provides these iOS Human Interface Guidelines which can be obtained at the following location: <http://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/MobileHIG.pdf>.

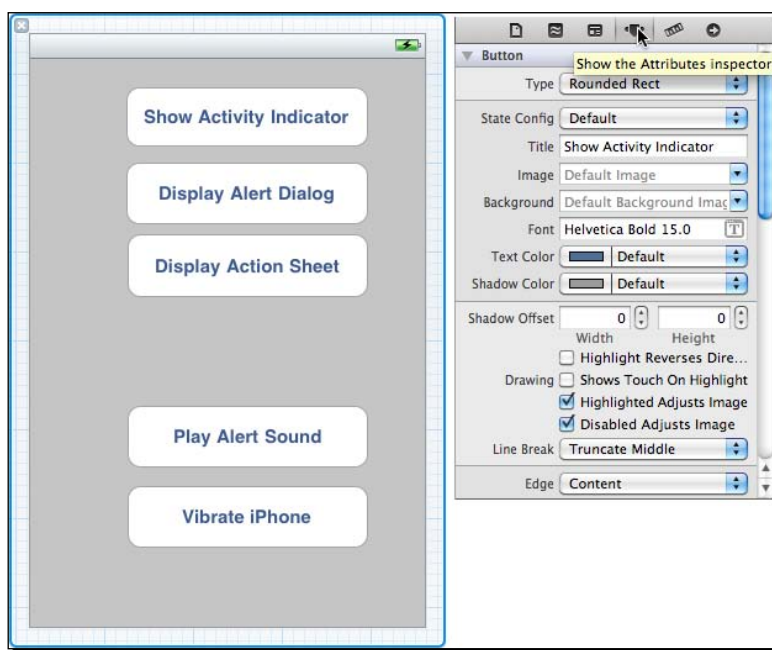
Time for action – adding controls to our View

We will be adding five button (`UIButton`) controls which will be handling our actions to display alerts and Action Sheets, playing sounds, and vibrating the iPhone.

1. From the Object Library, select and drag a (`UIButton`) Round Rect Button control onto our view. Modify the Object Attributes of the Round Rect Button control and set its title to read "Show Activity Indicator".
2. From the Object Library, select and drag a (`UIButton`) Round Rect Button control onto our view. Modify the Object Attributes of the Round Rect Button control and set its title to read "Display Alert Dialog".

3. From the Object Library, select and drag a (UIButton) Round Rect Button control onto our view. Modify the Object Attributes of the Round Rect Button control and set its title to read "Display Action Sheet".
4. From the Object Library, select and drag a (UIButton) Round Rect Button control onto our view. Modify the Object Attributes of the Round Rect Button control and set its title to read "Play Alert Sound".
5. From the Object Library, select and drag a (UIButton) Round Rect Button control onto our view. Modify the Object Attributes of the Round Rect Button control and set its title to read "Vibrate iPhone".

If you have followed everything correctly, your view should look something like the following screenshot. If it doesn't look quite the same, feel free to adjust yours:



What just happened?

In the above section, we looked at how we are able to use the Object Library to add controls to our view and customize their properties in order to build our user interface. In the next section, we will take a look at how to create events to respond to button events.

Creating events

Now that we have created our user interface, we need to create the events that will respond when we click on each of the buttons. If you need to refresh your memory on how to go about this, you can refer to the section *Making our Components work together* in Chapter 3, *Introducing Interface Builder*.

We first need to create an instance of our `UIAlertView` class, called **baseAlert**, which will be used by our **Show Activity indicator** event and will be used to dismiss the activity after a period of time has lapsed.

Open the `GetUsersAttentionViewController.h` interface file and add the following highlighted code as shown in the code snippet below:

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

@interface GetUsersAttentionViewController : UIViewController
    <UIAlertViewDelegate, UIActionSheetDelegate>{
        UIAlertView *baseAlert;
    }
@end
```

We could have declared this within our `GetUsersAttentionViewController.m` implementation file, but I prefer to declare it in this class as it can be referenced throughout your application.

You will notice from the code snippet above that we have made reference to two delegate protocols within our `GetUsersAttentionViewController.h` interface file; this enables us to capture and respond to the button event presses used by our Action Sheet and Alert Views. This will become apparent when we start adding the code events for our Alert Views and Action Sheets.

Time for action – implementing the Show Activity Indicator method

When you are performing tasks which are taking a period of time, you will want to provide the user with some form of notification. For tasks for which we are not sure how long it will take, we can use the `UIActivityIndicatorView` class, which provides us with and is represented as an animated spinner graphic.

The default size of the Activity indicator is a 21-pixel square, but can be changed to 36-pixels by using the `UIActivityIndicatorViewStyleWhiteLarge` style. In this section, we will look at how to go about implementing this class, combined with the `UIAlertView` control:

1. Open the `GetUsersAttentionViewController.m` implementation file.
2. In the action event which you created for the **Show Activity Indicator** button, add the following code:

```
// Displays our progress indicator with a message
- (IBAction)showProgress:(id)sender {

    // initialize our Alert View window without any buttons
    baseAlert=[[UIAlertView alloc]initWithTitle:@"Please
    wait,\ndownloading updates...." message:nil delegate:self
    cancelButtonTitle:nil otherButtonTitles:nil] autorelease];

    // Display our Progress Activity view
    [baseAlert show];

    // create and add the UIActivity Indicator
    UIActivityIndicatorView
    *activityIndicator=[[UIActivityIndicatorView
    alloc]initWithActivityIndicatorStyle:UIActivityIndicatorViewStyl
    eWhiteLarge];
    activityIndicator.center=CGPointMake(baseAlert.bounds.size.width
    / 2.0f,baseAlert.bounds.size.height-40.0f);

    // initialize to tell our activity to start animating.
    [activityIndicator startAnimating];
    [baseAlert addSubview:activityIndicator];
    [activityIndicator release];

    // automatically close our window after 3 seconds has passed.
    [self performSelector:@selector(showProgressDismiss)
    withObject:nil afterDelay:3.0f];
}
```

3. Next, we need to create an event to dismiss the progress indicator. Create the following **showProgressDismiss** event and add the following code:

```
// Delegate to dismiss our Activity indicator after the number of
seconds has passed.
- (void) showProgressDismiss
{
    [baseAlert dismissWithClickedButtonIndex:0 animated:NO];
}
```

What just happened?

In the above section, we looked at how we can use the `UIAlertView` alert class and the `UIActivityIndicatorView` classes to provide the ability to perform animation within our view. We also took a look at using the `startAnimating` method to start animating our activity indicator.

Next, we declared and instantiated an instance of our `UIAlertView` dialog and then declared and instantiated an instance of the `UIActivityIndicatorView` class and positioned this to be centred within our alert dialog.

We then made a call to the **`startAnimating`** method, which will display the activity indicator and cause the activity indicator graphic to start animating within our alert window.

After we have added this to our alert window view, we then need to release our **`activityindicator`** object once it has been added to the alert view, and then set up a delay to dismiss our activity view after a period of three seconds has lapsed, by calling the **`showProgressDismiss`** method.

 You can call the **`stopAnimating`** method to stop the activity view from animating, but you will need to remember to set the **`hideWhenStopped`** property if you want to permanently hide the activity view.

Finally we created our **`showProgressDismiss`** method to dismiss the activity indicator after a number of specified seconds lapsed by our **`showProgress`** event when the number of specified seconds has lapsed within the **`afterDelay`** property.

When using alert dialogs that don't contain any buttons, these don't properly call back to the delegate and therefore don't auto-dismiss correctly, as we have seen in our example, so we need to manually call the `dismissWithClickedButtonIndex:animated:` method which will close our alert dialog and stop our activity view from animating.

Have a go hero – adding a second activity indicator

Now that you have a good working knowledge of how to go about creating activity indicators and using these within your application, the task will be to add a secondary activity indicator. This will need to display a message after the downloading updates message stating that the updates are being finalized:

1. You will need to create another instance of the activity indicator You can refer to the section *Implementing the Show Activity Indicator* located in this chapter.
2. Next, you will need to add the activity indicator to the base class view. You can refer to the section *Implementing the Show Activity Indicator* located in this chapter.

3. Finally, set up a delay and create a new method to dismiss the activity indicator after a delay of five seconds has passed. You can refer to the section *Implementing the Show Activity Indicator* located in this chapter.

Once you have that working, you will have mastered how to create an application that contains more than one form of notification to grab the user's attention. This lets the user know that updates are being finalized.

Pop quiz – Activity Indicators

1. What method of the activityIndicator allows you to cease animation permanently?
 - a. `hideWhenStopped`
 - b. `startAnimating`
 - c. `stopAnimating`
2. What method starts animation of the activityIndicator?
 - a. `stopAnimating`
 - b. `startAnimating`
3. When using the `stopAnimating` method of the activityIndicator, what method should you use to permanently hide the activityview?
 - a. `release`
 - b. `stopAnimating`
 - c. `hideWhenStopped`

Time for action – implementing the Display Alert Dialog method

Our next step is to implement our `displayAlertDialog` method. This method will be responsible for displaying an alert message to the user when the **Display Alert Dialog** is pressed. The user will be able to respond to the buttons displayed, which will dismiss the dialog:

1. Open the `GetUsersAttentionViewController.m` implementation file.
2. In the action event which you created for the **Display Alert Dialog** button, add the following code:

```
// Handles of the setting up and displaying of our Alert View
Dialog
- (IBAction)displayAlertDialog:(id)sender {
    // Declare an instance of our Alert View dialog
```

```

UIAlertView *dialog;

// Initialise our Alert View Window with options
dialog = [[UIAlertView alloc] initWithTitle:@"Alert Message"
    message:@"Have I got your attention" delegate:self
    cancelButtonTitle:@"Cancel" otherButtonTitles:@"OK",nil];

// display our dialog and free the memory allocated by our
    dialog box
[dialog show];
[dialog release];
}

```

What just happened?

In the above section, we looked at how we can use the `UIAlertView` alert class to display a series of buttons and display a message based on the button pressed. We started by declaring and instantiating an instance of the `UIAlertView` class with a variable **dialog**. We then initialise our alert view to display the required buttons which we would like to have displayed and then display the dialog and release the memory used.

You will notice when we declared our `UIAlertView` class, it comprised of a number of parameters that are associated with this control. These are explained below:

ALERT PARAMETERS	DESCRIPTION
<code>initWithTitle</code>	Initializes the view and sets the title that will be displayed at the top of the alert dialog box.
<code>message</code>	This property sets the string that will appear within the content area of the alert dialog box.
<code>delegate</code>	Contains the object that will serve as the delegate to the alert. If this is set to nil , then no actions will be performed when the user dismisses the alert.
<code>cancellationToken</code>	This sets the string shown in the default button for the alert.
<code>OtherButtonTitles</code>	Adds additional buttons to the sheet that are delimited by commas as shown: <code>otherButtonTitles:@"Out to Lunch",@"Back in 5 Minutes",@"Gone Fishing"</code> .

Responding to Alert Dialog Button presses

In order for us to be able to capture the button that the user has pressed, we use the `clickedButtonIndex` method of the `alertView` property. This provides the button index of the pressed button and starts from 0. In the following code snippet, we look at how we are able to capture and respond to the actions when the user presses each of the buttons.

To get started, open the `GetUsersAttentionViewController.m` implementation file and create the following delegate function underneath the `displayAlertDialog` method:

```
// Responds to the options within our Alert View Dialog
- (void)alertView:(UIAlertView *)alertView
  clickedButtonAtIndex:(NSInteger)buttonIndex
{
    // String will be used to hold the text chosen for the button
    pressed.
    NSString *buttonText;

    // Determine what button has been selected.
    switch (buttonIndex)
    {
        case 0: // User clicked on Cancel button
            buttonText=@"You clicked on the 'Cancel' button";
            break;
        case 1: // User clicked on the OK button
            buttonText=@"You clicked on the 'OK' button";
            break;
        default: // Handle invalid button presses.
            buttonText=@"Invalid button pressed.";
    }

    // Initialise our Alert Window
    UIAlertView *dialog=[[UIAlertView alloc] initWithTitle:@"Alert
    Message" message:buttonText delegate:nil cancelButtonTitle:@"OK"
    otherButtonTitles:nil,nil];

    // display our dialog and free the memory allocated by our dialog
    box
    [dialog show];
    [dialog release];
}
```

In the above code snippet, we declared a delegate method which handles the button presses and retrieves the index of the button which was pressed. We declare an `NSString` variable **buttonText**, which will be used to store the title text to be displayed by our `UIAlertView`.

In order to determine the index of the button which was pressed, we perform a switch statement and then set up the **buttonText** variable with the associated text. Finally, we declare and instantiate a `UIAlertView` object, which will be used to display the greeting for the button which was pressed. We then display the dialog and then finally release the memory used.

If we wanted to retrieve the selected button using its text property, we would do so as shown in the following code snippet:

```
NSString *buttonTitle=[actionSheet buttonTitleAtIndex:buttonIndex];
if ([buttonTitle isEqualToString:@"OK"])
{
    buttonText=@"You clicked on the 'OK' button";
}
```

Finally, we looked at another way in which we can derive what button has been pressed. We declare an object `NSString` **buttonTitle**, which retrieves the text label for the button pressed on the action sheet. We then use the `isEqualToString` method to perform the comparison.

Have a go hero – adding additional buttons and creating the events

I will let you put into practice what you have just learnt.

Our application needs some additional buttons to be added to our Alert Dialog, and also needs to have the necessary code created to display the associated messages based on the button pressed. One way to do this would be as follows:

1. Locate the `displayAlertDialog` method
2. Add the necessary buttons to create within the `otherButtonTitles` property, separated by commas
3. Locate the `alertView: (UIAlertView *)` method
4. Add the necessary `buttonText` messages within the `switch` statement based on the `buttonIndex` of the button

Once you have that working, you will have extended your application to handle multiple buttons.

Pop quiz – Alert Dialogs and Button Indexes

1. What property would you modify to allow for additional buttons?
 - a. `initWithTitle`
 - b. `cancelButtonTitles`
 - c. `otherButtonTitles`

2. What is the purpose of the destructive button?
 - a. Sets the string to be shown in the default button for the alert.
 - b. Initializes the sheet with the specified title string.
 - c. The title of the option that will result in the information being lost.

Using Action Sheets to associate with a view

Action sheets are very similar to alerts in how they are initialized, and how the user responds to decisions. However, action sheets can be associated within a given view, tab bar, or toolbar and are animated when they become associated with the view onscreen. Action sheets also provide a separate button which is displayed as bright-red to alert the user to potential deletion of information.

Time for action – implementing the Display Action Sheet method

Action sheets provide the user with a variety of options to choose from. For instance, if a user was sending an SMS and there was a problem with it being sent, an action sheet will pop up asking the user if they want to Try again or Dismiss:

1. Open the `GetUsersAttentionViewController.m` implementation file.
2. In the action event which you created for the **Display Action Sheet** button, add the following code:

```
// Displays our Action Sheet
- (IBAction)displayActionSheet:(id)sender {

    // Define an instance of our Action Sheet
    UIActionSheet *actionSheet;

    // Initialise our Action Sheet with options
    actionSheet=[[UIActionSheet alloc]initWithTitle:@"Available
    Actions" delegate:self cancelButtonTitle:@"Cancel"
    destructiveButtonTitle:@"Close" otherButtonTitles:@"Open
    File",@"Print",@"Email", nil];

    // Set our Action Sheet Style and then display it to our view
    actionSheet.actionSheetStyle=UIBarStyleBlackTranslucent;
    [actionSheet showInView:self.view];
}
```

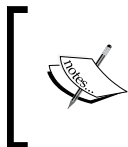
What just happened?

In this section, we added some code that will be called when the button **Display Action Sheet** is pressed. What this code is doing is declaring and instantiating an object **actionSheet** based on the `UIAlertSheet` class.

We then initialise our action sheet to display the required buttons which we would like to have displayed and then apply the action sheet style, then display the action sheet into the current view controllers view by using the `showInView:self.view` method.

You will notice when we declare our action sheet, that it comprises of a number of parameters associated with this control, which are explained below:

ACTIONSHEET PARAMETERS	DESCRIPTION
<code>initWithTitle</code>	Initializes the sheet with the specified title string.
<code>delegate</code>	Contains the object that will serve as the delegate to the sheet. If this is set to nil , the sheet will be displayed, but pressing a button will have no effect except dismissing the sheet.
<code>cancelButtonTitle</code>	This sets the string shown in the default button for the alert.
<code>destructiveButtonTitle</code>	The title of the option that will result in information being lost. This button is represented in bright red. However, if this is set to nil , then no destructive button will be displayed.
<code>OtherButtonTitles</code>	Adds additional buttons to the sheet that are delimited by commas as shown: otherButtonTitles:@"Item 1",@"Item 2",@"Item 3".



As of iPhone OS 3.0, action sheets can include up to seven buttons while maintaining the standard layout. If you happen to exceed this, the display will automatically change into a scrolling table view control, with the ability to add as many options as you need.

Responding to Action Sheet Button presses

In order for us to be able to capture the button that the user has pressed we use the `clickedButtonIndex` method of the `actionSheet` property. This provides the button index of the pressed button and starts from 0. In the following code snippet, we look at how we are able to capture and respond to the actions when the user presses each of the buttons.

Create the following delegate function located under the `displayActionSheet` method:

```
// Delegate which handles the processing of the option buttons selected
- (void)actionSheet:(UIActionSheet *)actionSheet
  clickedButtonAtIndex:(NSInteger)buttonIndex
{
    // String will be used to hold the text chosen for the button
    pressed.
    NSString *buttonText;

    // Determine what button has been selected.
    switch (buttonIndex)
    {
        case 0: // We selected the Close button
            buttonText=@"You clicked on the 'Close' button";
            break;
        case 1: // We selected the Open File button
            buttonText=@"You clicked on the 'Open File' button";
            break;
        case 2: // We selected the Print button
            buttonText=@"You clicked on the 'Print' button";
            break;
        case 3: // We selected the Email button
            buttonText=@"You clicked on the 'Email' button";
            break;
        case 4: // We selected the Cancel button
            buttonText=@"You clicked on the 'Cancel' button";
            break;
        default: // Handle invalid button presses.
            buttonText=@"Invalid button pressed.";
    }
    // Initialise our Alert Window
    UIAlertView *dialog=[[UIAlertView alloc] initWithTitle:@"Alert
    Message" message:buttonText delegate:nil cancelButtonTitle:@"OK"
    otherButtonTitles:nil,nil];

    // display our dialog and free the memory allocated by our dialog
    box
    [dialog show];
    [dialog release];
}
```

In the above code snippet, we declare a delegate method which handles the button presses and retrieves the index of the button which was pressed based on the order in which they were added. We declare an `NSString` variable **buttonText** which will be used to store the title text to display by our `UIAlertView`. In order to determine the index of the button which was pressed, we perform a switch statement and then set up the **buttonText** variable with the associated text. Finally, we declare and instantiate a `UIAlertView` object which will be used to display the greeting for the button which was pressed. We then display the dialog and then finally release the memory used.

Just in the same way as we did for alerts, if you want to retrieve the selected button using its **text** property, we would do so as shown in the following code snippet:

```
NSString *buttonTitle=[actionSheet buttonTitleAtIndex:buttonIndex];
if ([buttonTitle isEqualToString:@"Close"])
{
    buttonText=@"You clicked on the 'Close' button";
}
```

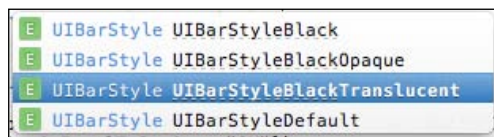
In the above code snippet, we looked at an alternative way by which we can derive what button has been pressed. We declared an object `NSString` **buttonTitle**, which retrieves the text label for the button pressed on the action sheet. We then used the `isEqualToString` method to perform the comparison.

Customizing an Action Sheet

Action sheets can take on numerous different user interface (UI) styles which are derived from the `UIBarStyle` class and can be applied to the **actionSheetStyle** property as demonstrated below:

```
actionSheet.actionSheetStyle=UIBarStyleBlackTranslucent;
```

This code renders the action sheet in a translucent black style. If you wanted to inherit the style of the views toolbar provided to which you have applied a style, you could use the `UIActionSheetStyleAutomatic` or if you preferred to go for a more solid-black classy finish, you could use and apply the `UIActionSheetStyleBlackOpaque` style:



Time for action – handling alerts via sounds and vibrations

Before we can play any sound or perform vibrations on our iPhone, we must first import the `AudioToolBox` library so that we can make use of its properties and methods:

1. Open the `GetUsersAttentionViewController.m` implementation file and add the following highlighted code as shown in the code snippet below:

```
#import "GetUsersAttentionViewController.h"
#import "AudioToolBox/AudioToolBox.h"
@implementation GetUsersAttentionViewController
```


2. Our next step is to implement our `playAlertSound` method. This method will be responsible for playing a short 30 second sound when the **Play Alert Sound** button is pressed. In the action event which you created for the **Play Alert Sound** button, add the following code:

```
// Plays an Alert Sound
- (IBAction)playAlertSound:(id)sender {
    SystemSoundID soundID;
    NSString *soundFile = [[NSBundle
        mainBundle]pathForResource:@"Teleport" ofType:@"wav"];
    AudioServicesCreateSystemSoundID((CFURLRef)[NSURL
        fileURLWithPath:soundFile], &soundID);
    AudioServicesPlaySystemSound(soundID);
}
```

3. Our next step is to implement our `vibratePhone` method. This method will be responsible for making our phone vibrate when the **Vibrate iPhone** button is pressed. Enabling the ability to provide feedback via vibration is a very simple and painless technique and all that is required is to pass a **System Sound ID** value to the `AudioServicesPlaySystemSound` method as we will see in a minute.
4. In the action event which you created for the **Vibrate iPhone** button, add the following highlighted code:

```
- (IBAction)vibratePhone:(id)sender {
    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
}
```

Now that we have finally created the necessary methods within our application to enable us to play sounds and vibrations, our next step is to build our application, as shown below.

5. We are now ready to build and compile our **GetUsersAttention** application. The screenshot below shows the output from each of the buttons when they are pressed:



So there you have it. We have explored five different ways in which we can communicate effectively with the user. By implementing these methods into your own applications, you will not only make your application more user-friendly, but it will also look more professional.

What just happened?

In the above section, we looked at how to implement sounds and vibrations in our application to grab the user's attention. In order for that to happen, we declared a variable called `SoundID` which will be used to refer to the sound file. Next, we declared an `NSString` **soundFile** variable which will contain the path to the sound file location by using the `NSBundle` class method `mainBundle` which corresponds to the directory location where the sound file **Teleport.wav** is located and use the `ofType` method to identify the type of sound file we want to play.

Once we have defined our path, we use the `AudioServicesCreateSystemSoundID` function to create a `SystemSoundID` that will be used to actually play the file.

This function takes the following two parameters: `CFURLRef` and `fileURLWithPath`. The `CFURLRef` parameter points to the location where the file is kept. The second parameter is a pointer to the `SystemSoundID` class that will be used to store the memory address of the file. The `fileURLWithPath` method returns an `NSURL` object which is what our `CFURLRef` is expecting.

Once we have set up our **SoundID** properly, all that is required is to play the sound which is achieved by passing the `SoundID` variable to the `AudioServicesPlaySystemSound` method.

We finally passed the `kSystemSoundID_Vibrate` constant variable to our `AudioServicesPlaySystemSound` method to allow our device to handle vibrations which have been defined within the `AudioToolBox.h` header file.

Have a go hero – adding Action Sheet items / changing appearance

I will let you practice what you have just learnt in this chapter.

Our `GetUsersAttention` application needs to be enhanced. We need to add some additional buttons and change the appearance of our action sheet. One way to do this would be as follows:

1. Within our `displayActionSheet` method, modify the `otherButtonTitles` property to include the additional button titles
2. Modify the `actionSheet:(UIActionSheet *)actionSheet` method, and add the additional button indexes to the `switch` statement
3. Compile and execute the application and check to ensure that the relevant button text appears when the buttons are pressed

Once you have made the relevant changes, you will experience how easy it is to modify action sheets to change their appearance and to allow for additional items to be displayed.

Pop quiz – sounds and vibrations

1. When changing the appearance of an action sheet, which method do you use?
 - a. `actionsheet`
 - b. `actionSheetRibbon`
 - c. `actionSheetStyle`
2. When changing the appearance of an action sheet, what class does it derive from?
 - a. `UIFont`
 - b. `UIAlertView`
 - c. `UIBarStyle`
3. What method do you use to vibrate the iPhone?
 - a. `AudioPlaySystemSound`
 - b. `PlaySystemSound`
 - c. `AudioServicesPlaySystemSound`

Summary

In this chapter, we learned about the different types of notification methods and modal dialogs which we can use to communicate effectively with the user.

We also learned how to go about using alerts and Action sheets and how to set the UI appearance of the Action Sheet using its various styles.

We finally looked at the two non-visual means by which we can communicate with the user by using sounds and vibrations using the `AudioToolBox` framework. Simply by using this framework, you make your applications more exciting and easily add short playing sounds and vibrate the iPhone.

Now that we have learned about the various ways in which you can communicate with the user, through alerts and action sheets, and sounds and vibration; we are ready to start focusing on how to handle the iPhone MultiTouch Architecture and learn how we can detect swipes, taps, pinches, and shaking. We will also be looking into the new iPhone 4 orientations by using the gyroscope feature. All of this will be covered in the next chapter.

7

Exploring the MultiTouch Interface

The Apple iOS device's primary interface with which you communicate is its large Multi-Touch display. Since there is no physical keyboard attached, everything is done via the screen to allow you to interact with your applications in a more natural way. Any object can be moved around the screen, zoomed in and out, and scrolled up and down using simple gestures.

In this chapter, we will see how easy it is to incorporate both single-touch and multi-touch support into our applications, to handle device taps, swipes, pinches, as well as responding to shake motions, before finally learning about the accelerometer and the gyroscope, and how to handle the situation when your iOS device orientation has changed.

In this chapter, we will be covering the following topics:

- ◆ Explore the iOS MultiTouch Architecture
- ◆ Learn how to detect taps, swipes, and pinches
- ◆ Learn how to use the built-in shake gesture
- ◆ Explore the new features of the Accelerometer and Gyroscope
- ◆ Learn how to handle and sense device orientation
- ◆ Learn how to detect when a device has been tilted

We will be taking a look at these in more detail through each of the examples which we will be building throughout this chapter.

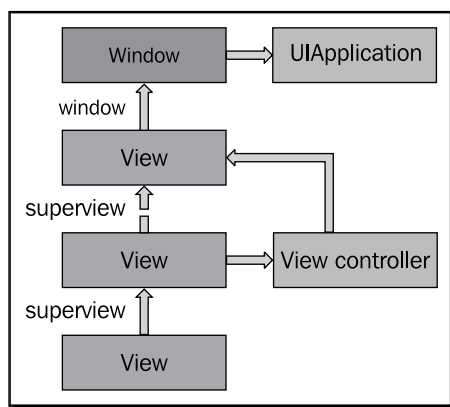
We have got quite a bit to cover, so let's get started.

Introducing the MultiTouch architecture

The **MultiTouch** gesture architecture is based on the concept of a responder chain. When the user performs an action, for instance, tapping or swiping the screen, the system will proceed to generate instances of the `UIEvent` class to indicate that some user interaction has occurred and then the chain responder objects are given a chance to respond.

In order for you to gain a better understanding of how the responder chain events are handled, the screenshot below will explain what happens when each of the events are fired.

Each of the **UIEvent** events which get created represents a distinctive gesture which is currently being processed with each of the `UIEvents` received containing the parts that make up the gesture. Gestures such as a simple finger movement across the screen, a swipe, or a tap are all instances of the `UITouch` class:



As you will see from the image above, each of the responder chains are made up of a series of linked responder objects, each of these being an implementation of the `UIResponder` class.

By looking at the diagram, you will notice that it sort of resembles the view hierarchy, which means that it starts at the lowest subview before making its way all the way up through the hierarchy.

The first link in the chain is known as the first responder and is executed first. If for some reason, this cannot respond to the event, it gets bubbled up to the next responder in the chain and then tries to process the event. This process continues until there are no more responders to process and then proceeds to the applications `UIWindow` instance before finally finishing up at the applications `UIApplication` instance which then gets a chance to handle the event.



It is not unusual for an event to pass all the way up through the chain without getting any responses. This does not cause any problems and ensures that the process starts and ends correctly.

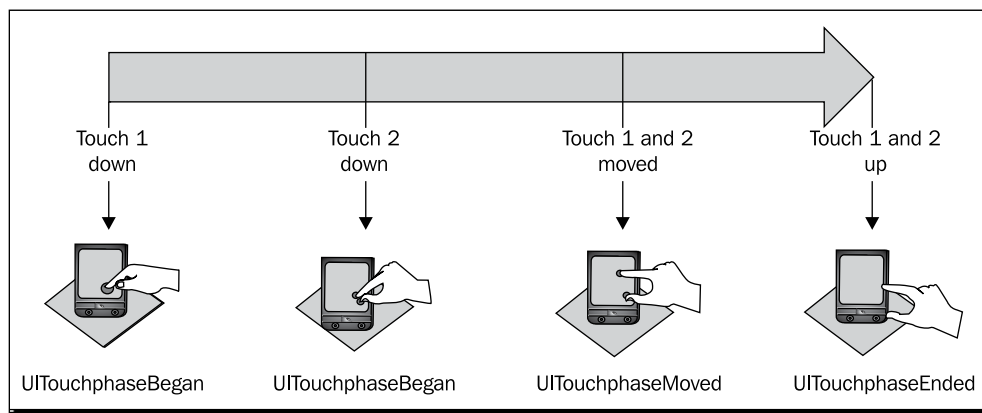
Before you can start to use **MultiTouch** events within your application, you need to insert the code into the responder chain by inserting and implementing any of the four `UITouch` events which are part of the `UIResponder` class.

The `UIView` and `UIViewController` classes are all part of the `UIResponder` class as they can respond to and handle events within the view. By implementing any of the methods below, you are then able to override the methods used by your view or view controller:

MOTION EVENT METHODS

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event

The screenshot below shows you each of the events that get executed when your view or view controller receives a touch:



Each of the **UITouch** touches that get received has a phase, location, the view in which the touch occurred, a timestamp and a count of the number of taps that occurred. In the table below, we explain what happens when each of the touch phases are fired:

UITOUCH PHASE EVENTS	DESCRIPTION
UITouchPhaseBegan	Occurs at the beginning of the touch life cycle when the user has touched an area of the iPhone screen.
UITouchPhaseMoved	Occurs when the user has moves their finger or fingers around the screen of the iPhone.
UITouchPhaseStationary	Occurs when the user has paused on an area of the screen.
UITouchPhaseEnd	Occurs when the user has removed their fingers from the screen of the iPhone.
UITouchPhaseCancelled	Occurs when the iOS device determines that something has happened and needs to abort the gesture. An example of this can be due to a system interruption caused when you are receiving an incoming phone call, or when an application or window view is no longer active.

Detecting taps

The iPhone is equipped to keep track of taps that occur when the user taps on an area of the screen and delivers this as a single tap event to the **UIResponder** event chain with the count of the number of taps that have occurred. Just like with mouse clicks, the iPhone can receive and handle both single or double taps.

Time for action – creating the TapExample project

Before we can proceed, we first need to create the `TapExample` project. To refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.

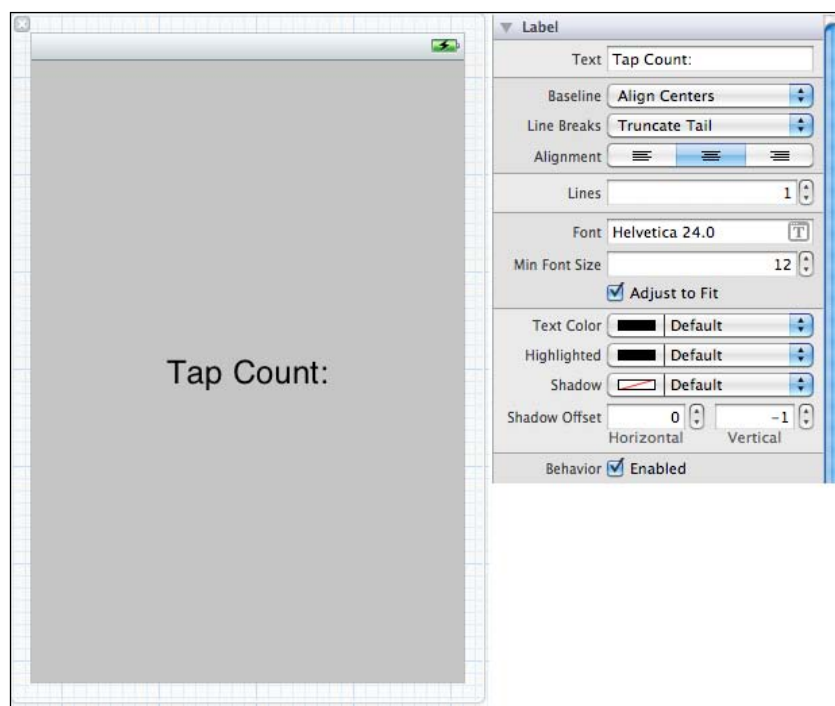
7. Specify a name for the project which you want to create.
8. Enter **TapExample** and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once our project has been created, you will be presented with the Xcode interface, along with the project files that the template has created for you within the Project Navigator Window.

Our next step is to concentrate and start to build our user interface. We will be keeping this example simple and will be adding just one control which will display how many taps have occurred when the user taps on the view:

1. From the Object Library, select and drag a (UILabel) label control and drag it onto the view.
2. Modify the Object Attributes of the label control, and set the title to read **Tap Count:**
3. Adjust the **Tap Count:** label font size to be **Helvetica 24.0**

If you have followed everything correctly, your view should look something like the screenshot below. Feel free to adjust yours accordingly:



What just happened?

In the above section, we looked at the steps involved in creating a View-based application for our `TapExample` application. We then looked at the steps involved in building our user interface by using the Label control from the Object Library and modifying some of the properties associated with the control to set the size of the label. In the next section, we will take a look at how we can bind the label control to an event, and set the background color of the view depending on the total number of taps made.

Time for action – binding our Controls

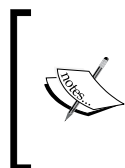
Now that we have created a user interface, we need to create the events so that we can update this label to show how many taps have occurred. If you need to refresh your memory on how to go about this, you can refer to the section *Making our components work together* which we covered in *Chapter 3, Working with the Interface Builder*:

1. Open the `TapExampleViewController.h` interface file located within the `Classes` folder of your project and add the following code:

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

@interface TapExampleViewController : UIViewController {
    UILabel *tapCountLabel;
}
@property (nonatomic, retain) IBOutlet UILabel *tapCountLabel;

@end
```
2. We now need to open our `TapExampleViewController.m` implementation file, located within the `Classes` folder of our project so that we can synthesize our properties to be able to use them.



If we don't declare these, we will receive compiler warning messages which can result in unexpected results occurring in our application and can even make our application terminate unexpectedly which will not be too pleasing to our users.

```
#import "TapExampleViewController.h"

@implementation TapExampleViewController
@synthesize tapCountLabel;
```

3. Our next step is to implement the `touchesBegan:touches` method as we need to declare a `UITouch` variable ***touch** that will be used to retrieve the touch events. We then need to construct our `labelOutput` string to show how many times the user tapped, and then based on the number of taps made, we set the background color of our view.

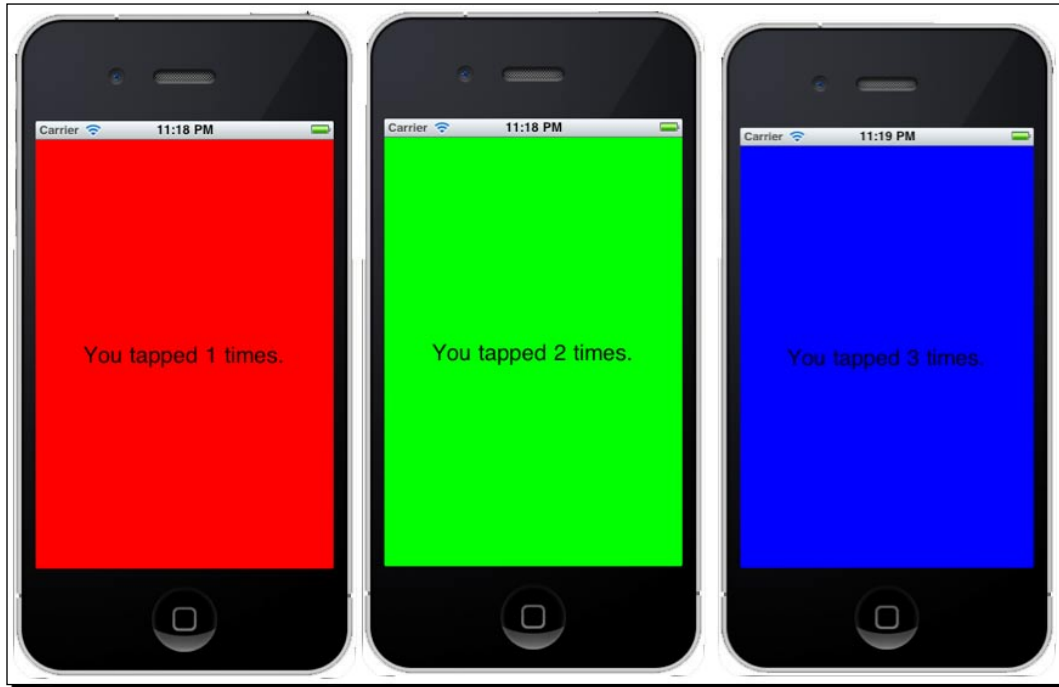
With the `TapExampleViewController.m` implementation file open, create the following event as shown in the code snippet below:

```
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    NSString *labelOutput;
    UITouch *touch = [[event allTouches] anyObject];
    labelOutput = [NSString stringWithFormat:@"You tapped %i
        times.", [touch tapCount]];
    tapCountLabel.text=labelOutput;
    switch ([touch tapCount])
    {
        case 1:
            self.view.backgroundColor=[UIColor redColor];
            break;
        case 2:
            self.view.backgroundColor=[UIColor greenColor];
            break;
        case 3:
            self.view.backgroundColor=[UIColor blueColor];
            break;
        case 4:
            self.view.backgroundColor=[UIColor yellowColor];
            break;
        case 5:
            self.view.backgroundColor=[UIColor orangeColor];
            break;
        default:
            self.view.backgroundColor=[UIColor redColor];
            break;
    }
}
```

4. Our final step is to release the memory used by the view controller objects which we have declared. Xcode creates these for you automatically when you declare the outlets in your `TapExampleViewController.h` interface file, located within the Classes folder of your project:

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

5. We are now ready to build and compile our **TapExample** application. The screenshot below shows the output for the number of taps that have been pressed:



So there you have it; we have successfully created a simple, yet effective application which can respond to user taps. By implementing any of the four touch events into your application, you can respond to each accordingly.

What just happened?

What we just covered in the section above were the steps involved in hooking up our **tapCountLabel** control to the touches event method of the view controller. This enabled us to determine the number of times the view was tapped. This was handled by the implementation of the `touchesBegan:touches` method. Based on the total number of taps made, the application then updated the **labelOutput** control to display how many times the user tapped the view and also changed the background color.

Have a go hero – modify the program to change background

I will let you put into practice what you have just learnt.

Our application needs an addition made to the code to change the background color to purple whenever the number of taps made exceeds 10. One way to do this would be as follows:

1. Create another `case` item inside the `switch` statement.
2. Initialise the `backgroundColor` method of the view to change its background color to **Purple** which is derived from the `UIColor` class.
3. Include a `break` statement after the added code to avoid this from falling through into any other `case` statements you have.
4. Compile, Build and Run the application.

Once you have that working, you will have customized the application to change the background color based on the number of taps exceeding 10.

Pop quiz – tap counts

1. Which method allows you to determine the number of taps made?
 - a. `tapCount`
 - b. `touchCount`
 - c. `getCount`
2. What method determines when a touch has been performed?
 - a. `touchesMoved`
 - b. `touchesCancelled`
 - c. `touchesBegan`

Detecting swipes

Up to now, you have looked at the tap gesture of the iPhone and have learned how to handle and respond to the events. You may have noticed while working with the tap gesture that it only requires one `UIResponder` method. When dealing with swipes and the swipe gesture, this is more involved and the responder lasts longer.

The swipe gesture starts with the `touchesBegan:touches:withEvent` method when it detects that a finger or fingers have first touched the screen. The responder will then call a series of `touchesMoved:touches:withEvent` method calls when the user proceeds to move their fingers across the screen. The method `touchesEnded:touches:withEvent` is called when the responder realizes that the user has removed their finger or fingers from the screen.

Time for action – creating the SwipeExample project

To learn a bit more about how we go about handling and using the Swipe gesture, let's proceed and create a View-based application:

1. Launch Xcode from the /Xcode4/Applications folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project which you want to create.
8. Enter **SwipeExample** and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once your project has been created, you will be presented with the Xcode interface, along with the project files that the template created for you within the Project Navigator Window. We are now ready to start implementing the code to detect and handle when a swipe has occurred, by following these simple steps:

1. Open the `SwipeExampleViewController.h` implementation file, located within the `Classes` folder of your project and add the following code:

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

#define minGestureLength 25 // The Minimum length to denote a
swipe.
#define allowableVariance 5 // The variance of 5 pixels in
length.
#define delayFactor      3 // Define our delay factor

@interface SwipeExampleViewController : UIViewController {
    CGPoint currentStartingPoint;
}

@end
```

2. Next, open the `SwipeExampleViewController.m` implementation file located within the `Classes` folder of your project so that we can start to use these within our application. We need to locate the `viewDidLoad` method and add the following highlighted code:

```
// Implement viewDidLoad to do additional setup after loading the
view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor blackColor];
}
```

3. Our next step is to create the `touchesBegan:withEvent` method and enter in the following code:

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    currentStartingPoint = [touch locationInView:self.view];
}
```

4. Next, we create the `touchesMoved:withEvent` method, which will be used to handle the swipe and work out how far the user has travelled within the view to make it validate that a swipe has taken place. Enter in the following code:

```
-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
    UITouch *touch = [touches anyObject];
    CGPoint currentPosition = [touch locationInView:self.view];

    // Calculate how far the user's finger has moved both
    horizontally
    and vertically from its starting position.
    CGFloat deltaX = fabsf(currentStartingPoint.x - currentPosition.
x);
    CGFloat deltaY = fabsf(currentStartingPoint.y - currentPosition.
y);

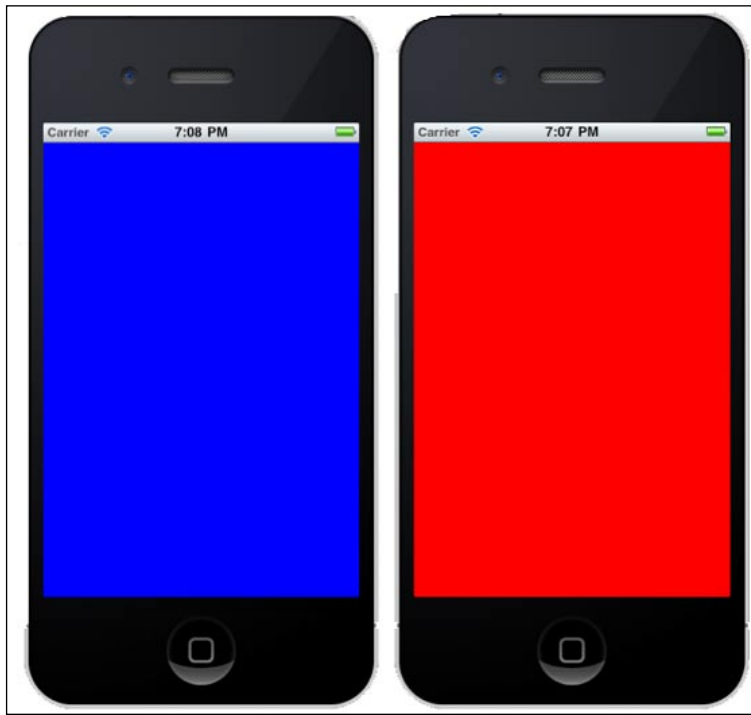
    // Check to see if we are currently doing a Horizontal Swipe
    if(deltaX >= minGestureLength && deltaY <= allowableVariance){
        // Horizontal Swipe detected, so set our background color to
Red
        // reset the background color to black after our delay of 3
seconds have passed.
        self.view.backgroundColor = [UIColor redColor];
        [self performSelector:@selector(resetBackground) withObject:
nil
        afterDelay:delayFactor];
    }
}
```



```
// Check to see if we are currently doing a Vertical Swipe
else if(deltaY >= minGestureLength && deltaX <=
allowableVariance){
    // Vertical Swipe Detected.
    self.view.backgroundColor=[UIColor blueColor];
    // [self performSelector:@selector(resetBackground)
    withObject:nil afterDelay:10];
    [self performSelector:@selector(resetBackground) withObject:
nil
    afterDelay:delayFactor];
}
}

// Handles resetting the background
- (void)resetBackground
{
    self.view.backgroundColor=[UIColor blackColor];
}
```

We have finally made it and now we are ready to Compile, Build and Run our **SwipeExample** application. The screenshot below shows the output when the application is run, and when the user swipes in the horizontal direction and vertical direction:



What just happened?

What we covered in this section were the steps involved to create our `SwipeExample`. We define a number of variables that will be used to determine how far the user moved within the view in order to be counted as a swipe. We declare the minimum gesture length and an allowable variance variable range to be either five pixels above or below. We then located and modified the `viewDidLoad` event method to set and initialize the background color of the view to black when the application is loaded. Next, we declared a call to the `touchesBegan` method to obtain the current starting point when the user first places their finger or fingers onto the iOS screen and then declared an instance variable **`touch`** which is used to grab the position of the location within the view and then calculate how far the user's finger has moved since its last starting position.

We then determine if the swipe was a horizontal action or a vertical action and then set the view background colors accordingly and implement a timer to reset the background back to black by calling the `resetBackground` method after a number of seconds have passed, which is handled by the `afterDelay` property.



The function `fabsf()` is from the standard C math library that returns the absolute value of a float.

Have a go hero – adjust the `delayFactor` and change the background

I will let you put into practice what you have just learnt.

What we need to do here is modify our `SwipeExample` application to change the delay factor before resetting the background. Instead of setting this back to black, we need to allow it to cycle through a number of colors. One way to do this would be as follows:

1. In the `SwipeExampleViewController.h` interface file, change the **`delayFactor`** variable to another value. This can be anything that you like.
2. Locate the `resetBackground` method inside the `SwipeExampleViewController.m` implementation file, and create a series of `self.view.backgroundColor` statements, and set the background color for each.

Once you have completed this task, Compile, Build and Run the application.

Pop quiz – tracking and identifying swipes

1. In order to receive events of the `UIEventTypeMotion`, what event method must the view or view-controller override?
 - a. `motionEnded`.
 - b. `motionCancelled`.
 - c. `motionBegan`.
 - d. All of the above
2. What is always the last responder in an iOS applications responder chain?
 - a. `UIResponder`.
 - b. `UIKit`.
 - c. `UIApplication`.

Detecting pinches

So far, we have looked at how we can detect when a user taps and swipes on the iPhone device. There is still one more **MultiTouch** gesture that we need to cover, that being the two-finger pinch. A pinch is basically when two fingers are placed on the iOS device and then moved apart or brought closer together to simulate a zoom-in or zoom-out feature. Apple uses this feature in many of its applications: Safari web browser, Maps, and resizing of various images which you take with your iPhone camera or download from the internet.

Time for action – creating the PinchExample project

To learn a bit more about how we go about handling and setting up the pinch gesture, let's proceed and create a View-based application:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project that you want to create.

8. Enter **PinchExample** and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once our project has been created, you will be presented with the Xcode interface, along with the project files that the template has created for you within the Project Navigator Window. We are now ready to start implementing the code that will be used to detect and handle when a pinch has occurred, by following these simple steps:

1. Open the `PinchExampleViewController.h` interface file, located within the **Classes** folder of your project and add the following code:

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

@interface PinchExampleViewController : UIViewController {
    UIView *ourBox;
}

@end
```

2. Next, open the `PinchExampleViewController.m` implementation file, located within the **Classes** folder of your project and add the following code:

```
// Implement viewDidLoad to do additional setup after loading
// the view, typically from a nib.
- (void)viewDidLoad {

    [super viewDidLoad];

    // initialise and create our Box
    float boxSize = 100.0;
    CGRect ourBoxRect = CGRectMake(100,150,boxSize,boxSize);
    ourBox = [[UIView alloc] initWithFrame:ourBoxRect];
    ourBox.backgroundColor = [UIColor greenColor];

    // we need to tell our object that we want to be able to
    // handle multiple touches
    ourBox.userInteractionEnabled = YES;

    // initialise our view background color and then add the box
    // to the view.
    self.view.backgroundColor = [UIColor blackColor];
    [self.view addSubview:ourBox];
}
```

- 3.** Next, we need to create the `distanceBetweenPoints` method within the `PinchExampleViewController.m` implementation file and add the following code:

```
// Calculate the distance between the two points
CGFloat distanceBetweenPoints(CGPoint pt1, CGPoint pt2) {
    CGFloat distance;
    CGFloat xDifferenceSquared = pow(pt1.x - pt2.x, 2);
    CGFloat yDifferenceSquared = pow(pt1.y - pt2.y, 2);
    distance = sqrt(xDifferenceSquared + yDifferenceSquared);
    return distance;
}
```

- 4.** Next, create the `transformWithScale` method within the `PinchExampleViewController.m` implementation file and add the following code:

```
CGAffineTransform transformWithScale(CGAffineTransform
    oldTransform, UITouch *touch1, UITouch *touch2) {
    CGPoint touch1Location = [touch1 locationInView:nil];
    CGPoint touch1PreviousLocation = [touch1
previousLocationInView:nil];
    CGPoint touch2Location = [touch2 locationInView:nil];
    CGPoint touch2PreviousLocation = [touch2
previousLocationInView:nil];

    // Get distance between points
    CGFloat distance =
distanceBetweenPoints(touch1Location, touch2Location);
    CGFloat prevDistance =
distanceBetweenPoints(touch1PreviousLocation,
touch2PreviousLocation);

    // Figure out the new scale ratio
    CGFloat scaleRatio = distance / prevDistance;
    CGAffineTransform newTransform =
CGAffineTransformScale(oldTransform, scaleRatio, scaleRatio);

    // Return result
    return newTransform;
}
```

- 5.** Next, create the `transformWithRotation` method within the `PinchExampleViewController.m` implementation file and add the following code:

```
CGAffineTransform transformWithRotation(CGAffineTransform
oldTransform, UITouch *touch, UIView *view, id superview) {
    CGPoint pt1 = [touch locationInView:superview];
    CGPoint pt2 = [touch previousLocationInView:superview];
    CGPoint center = view.center;
    CGFloat angle1 = atan2( center.y - pt2.y, center.x - pt2.x );
    CGFloat angle2 = atan2( center.y - pt1.y, center.x - pt1.x );

    CGAffineTransform newTransform =
CGAffineTransformRotate(oldTransform, angle2-angle1);

    // Return result
    return newTransform;
}
```

- 6.** Next, create the `touchesMoved` method within the `PinchExampleViewController.m` implementation file and add the following highlighted code:

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent
*)event {

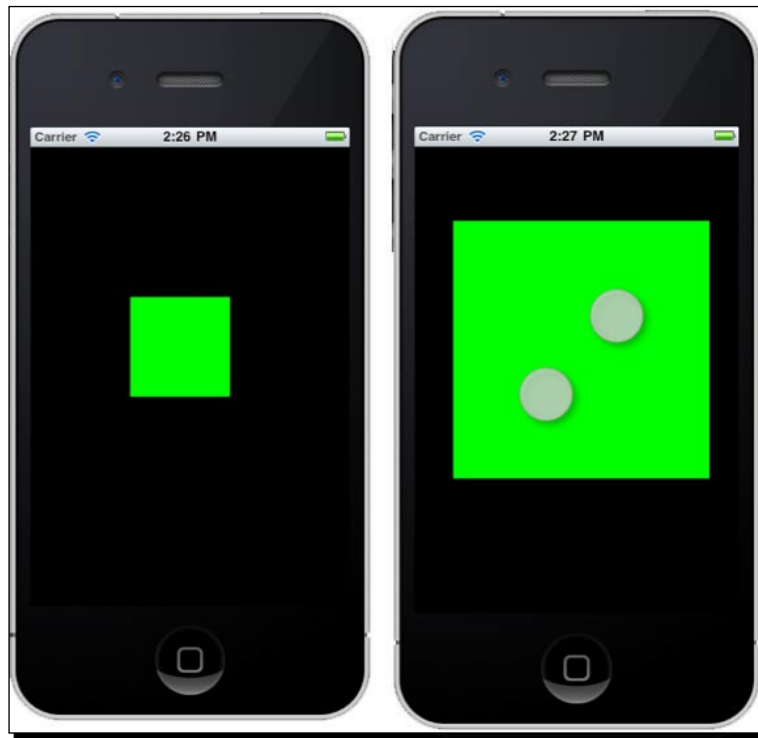
    if ([[event touchesForView:ourBox] count] == 1)
    {
        UITouch *touch = [[[event touchesForView:ourBox]
allObjects] objectAtIndex:0];
        ourBox.transform =
transformWithRotation(ourBox.transform, touch, ourBox, self.view);
    }

    if ([[event touchesForView:ourBox] count] == 2)
    {
        UITouch *touch1 = [[[event touchesForView:ourBox]
allObjects] objectAtIndex:0];
        UITouch *touch2 = [[[event touchesForView:ourBox]
allObjects] objectAtIndex:1];
        ourBox.transform = transformWithScale(ourBox.transform,
touch1, touch2);
    }
}
```

7. Our final step is to release the memory used by our view controller objects which we declared in our `PinchExampleViewController.h` interface file. Add the following highlighted code as shown below to your **dealloc** method:

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

We have finally made it and now we are ready to Compile, Build and Run our **PinchExample** application. The screenshot below shows the output when the application is run, and when the user resizes the image.



So there you have it; by using the Core Graphics library and implementing some methods, you can make your application incorporate the ability to support MultiTouch features.



To resize the image from within the iPhone Simulator, hold down the *Control* + *Option* buttons and then use your mouse to stretch the image.

What just happened?

What we covered in this section were the steps involved in creating our `SwipeExample`. We declared an instance `UIView` variable **ourBox** which will be eventually placed in our view. Next, we declared an instance `CGRect` variable **ourBoxRect** which is used to define our rectangle. We then allocated the memory for our box, set the background color and then set this up to handle and use multitouch events. We then proceeded to set the background color of our view and then added our box as a subview to the current view controller.



The algorithm to calculate distance between two points is made freely available on the Wikipedia website at the following location: <http://en.wikipedia.org/wiki/Distance>.

After adding the box to the view, we need to determine the touch locations of the two points. We make use of our `distanceBetweenPoints` function to calculate the distance of the previous location, and then work out the scale ratio portion of our box image that we need to redraw back to the view. Next, we work out the touch location positions which are derived from the `UITouch` class as well as determining the angle to which the object needs to be rotated within the view. We use the `CGAffineTransformRotate` function which is part of the Xcode `CoreGraphics` library to transform the object within the view. Next, we need to determine if we are handling a single touch and if this is the case we just want to call and rotate our box object within the view. However, if we have determined that we are handling more than a single touch, we need to call our function to scale the image and redraw it to the view, before finally releasing the memory used by the `dealloc` method.

Have a go hero – handling more than two fingers

I will let you put into practice what you have just learnt.

Our application needs to be modified to cater for more than two fingers being placed down on the view at any given time. We need to incorporate into our application the ability for the box to be moved around the screen while it is being rotated. One way to do this would be as follows:

1. In the `touchesMoved` method, add another event type to check if the number of touches are 3.
2. Declare another `touch3` instance of the `UITouch` class.
3. Use the `CGPoint` class to determine the location of the two points and then reposition the box object within the view.

Once you have that working, you will have a more user-friendly application which will allow the user to resize, rotate, and move the object around the view.

Pop quiz – pinches and transformations

1. When identifying pinches, how many instances of the `UITouch` method are allowed?
 - a. one
 - b. three
 - c. two
2. What method allows you to Rotate an object?
 - a. `CGAffineTransform`.
 - b. `transformWithRotation`.
3. What method allows you to Scale your object?
 - a. `CGAffineTransformScale`.
 - b. `transformWithScale`.

Detecting shakes

When the iPhone device is shaken, the system makes use of the accelerometer and then interprets the accelerometer data to see if it is a shake instruction.

If this has been determined to be a shake gesture, the system creates a `UIEvent` object which represents this gesture and then sends the object to the currently active application for processing.

Using the shake gesture on the iPhone is a lot simpler to use than touch events. Events are still generated when a motion starts or stops and it is even possible for you to track individual motions as you would do with touch events.

In order to make your applications incorporate and handle the iOS shake gesture, this can be easily accomplished by implementing the following three methods as shown in the table below:

METHOD	DESCRIPTION
<code>motionBegan:motion:withEvent:</code>	This method is called when a motion event begins.
<code>motionEnded:motion:withEvent:</code>	This method is called when a motion event has ended.
<code>motionCancelled:motion:withEvent:</code>	This method is called if the system thinks that the motion is not a shake. Shakes are determined to be approximately a second or so in length.



Motion events were first introduced in the iOS 3.0 SDK, with shaking motions currently being interpreted as gestures which then move on to become motion events.

Time for action – creating the ShakeExample project

To learn a bit more about how we go about handling and setting up the shake gesture, let's proceed and create a View-based application:

1. Launch Xcode from the /Xcode4/Applications folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project which you want to create.
8. Enter `ShakeExample` and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once our project has been created, we will be presented with the Xcode interface, along with the project files that the template has created for you within the Project Navigator Window.

We are now ready to start implementing the code that will be used to detect when a shake has occurred on the iOS device, by following these simple steps:

1. Next, open the `ShakeExampleViewController.m` implementation file, located within the `Classes` folder of your project and add the following code as shown below:

```
// Implement viewDidLoad to do additional setup after loading
the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor greenColor];
}
```

2. Next, in order for our view or view controller to start receiving motion events of the `UIEventTypeMotion` type, we need to make our view or view controller the first responder in the `UIResponder` responder chain and this must override one or more of the three `UIResponder` motion event methods which are shown below:

- a. `-(void) motionBegan: (UIEventSubtype) motion
withEvent: (UIEvent *) event`
- b. `-(void) motionEnded: (UIEventSubtype) motion
withEvent: (UIEvent *) event`
- c. `-(void) motionCancelled: (UIEventSubtype) motion
withEvent: (UIEvent *) event`

For the purpose of this example, we will be just overriding the `motionEnded:motion:withEvent` which will show an alert message when the shake gesture ends.

3. Next, we need to allow our view controller to support and start receiving motion events. With the **ShakeExampleViewController.m** implementation file still open, add the following code snippet above the `viewDidAppear` method:

```
- (BOOL) canBecomeFirstResponder {  
    return YES;  
}  
  
- (void) viewDidAppear: (BOOL) animated {  
  
    [self becomeFirstResponder];  
    [super viewDidAppear:animated];  
}
```

What just happened

What we covered in this section were the steps involved in creating our `ShakeExample`. We initialized our view background color to green in the `viewDidLoad` event to indicate that no shake has occurred yet. Next, we needed to make our view controller the first responder in the `UIResponder` responder chain by overriding the **`motionEnded:motion:withEvent`** which will show an alert message when the shake gesture ends. Finally, we needed to add a method inside the `viewDidAppear` method in the `ShakeExampleViewController.m` implementation file to allow our view or view controller to be able to support the motion events. If we don't include this, none of the motion events will fire and our application will not behave as we have designed it to.

Time for action – implementing the `motionBegan`, `motionEnded`, and `motionCancelled` methods

We are now ready to start implementing the code that will be used to detect when a shake has occurred on the iOS device. We will learn about the various motion methods and how to implement these by following these simple steps:

1. To begin, open the `ShakeExampleViewController.m` implementation file, located within the `Classes` folder of your project and add the following code:

```
- (void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent
*)event {
    if (event.type == UIEventTypeMotion && event.subtype ==
UIEventSubtypeMotionShake)
    {
        self.view.backgroundColor=[UIColor yellowColor];
        NSLog(@"Device has been shaken");
    }
}
```

Next, we need to implement the code to handle when the motion has ended. Unfortunately, it is not possible to track individual motions as you can do with touch events.

2. Add the following code to our `ShakeExampleViewController.m` implementation file:

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent
*)event {
    if (event.type == UIEventTypeMotion && event.subtype ==
UIEventSubtypeMotionShake)
    {
        // Declare an instance of our Alert View dialog
        UIAlertView *dialog;

        // Initialise our Alert View Window with options
        dialog =[[UIAlertView alloc] initWithTitle:@"Device has
been
shaken" message:@"I was asleep, now i'm awake.
Press OK to reset" delegate:self cancelButtonTitle:nil
otherButtonTitles:@"OK",nil];

        // display our dialog and free the memory allocated by our
dialog
        box
        [dialog show];
    }
}
```

```
        [dialog release];
    }
}
// Responds to the options within our Alert View Dialog
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex
{
    // String will be used to hold the text chosen for the button
    pressed.
    NSString *buttonTitle=[alertView buttonTitleAtIndex:
buttonIndex];
    if ([buttonTitle isEqualToString:@"OK"])
    {
        self.view.backgroundColor=[UIColor greenColor];
        NSLog(@"Device has stopped shaking");
    }
}
```

Next we need to implement the `motionCancelled` event. This event is called when the system thinks that the type of motion is not a shake.

3. Add the following code to our `ShakeExampleViewController.m` implementation file:

```
- (void)motionCancelled:(UIEventSubtype)motion
withEvent:(UIEvent *)event
{
    self.view.backgroundColor=[UIColor blackColor];
    NSLog(@"Device shake has been cancelled");
}
```

We have finally made it and now we are ready to Compile, Build and Run our **ShakeExample** application. Given that you have typed in everything correctly, your code should compile without any issues.

If all compiles well, your application should resemble something like what is shown in the screenshot below showing the output when the device has been shaken and when it has not. You will notice that where we have told our application to log those events, these appear in the output window:



So there you have it. In just a few simple steps, you can make your application shake-aware and you can create some fantastic applications using this feature.

What just happened?

In this section, we looked into the various methods that we can use to detect when a motion happens on the iOS device. We looked at how to implement the `motionBegan` method and set the background color to yellow when the iOS device has detected that a shake has occurred.

When the device determines that the motion has stopped, the `motionEnded` method is called and that is where we can detect what type of event happened. In this case, we then declare and instantiate our instance of the `UIAlertView` class and display a message to the user, alerting them that the shake has ended.

The method `motionCancelled` is called if the system thinks that the motion is not a shake. Shakes are determined to be approximately a second or so in length. It then calls the `motionEnded` method and sets the background color of our view controller to black.

Have a go hero – modifying the ShakeExample application

I will let you put into practice what you have just learnt.

What we need to do for this example is to expand and modify the message that comes up when the device has been shaken to allow the user to choose from a number of options. The first button will say **Sleep**, and the second one **Awake**. We want to be able to store the chosen response value selected by the user and when Awake is pressed, we have a message which says **I was awake, now I am asleep**, and vice-versa. One way to do this would be as follows:

1. Modify the `otherButtonTitles` property in the `UIAlertView` in the `motionEnded` method.
2. Add the `Awake` and `Sleep` button titles.
3. In the `alertView: (UIAlertView)` method, create separate `if` statements for both scenarios.
4. Create another `UIAlertView` class variable in the above method with the associated text.
5. Once you have done this, Compile, Build and Run your application.

Once you have that working, you will have a more user-friendly application which lets the user choose between two different responses and have the associated text display.

Pop quiz – motion events

1. What are the three methods that need to be implemented for motion?
 - a. `motionBegan`.
 - b. `motionDead`.
 - c. `motionInitialise`.
 - d. `motionCancelled`.
 - e. `motionEnded`.
2. You have implemented the motion methods, but when you run your application, nothing happens. Why?
 - a. `motionBegan` method has not been created.
 - b. `[self becomeFirstResponder]` has not been added to the `viewDidLoad` method.
 - c. `-(BOOL) canBecomeFirstResponder` method has not been implemented to return YES.

Exploring the Accelerometer/Gyroscope

So far you have been focusing on how to detect when a user performs taps, swipes, pinches and how to detect device shakes. We now move on to the really exciting stuff, not that what you have already covered is not exciting, but the iPhone's accelerometer is much more powerful than you think and is capable of giving you live data for all three dimensions of the (x, y, and z) axes when the phone is tilted.

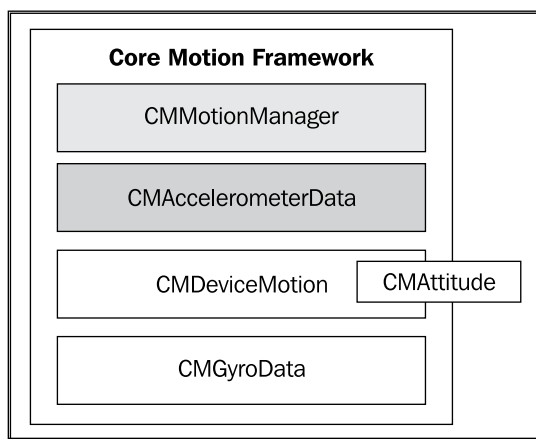
The iPhone's accelerometer data is delivered via the `UIAccelerometer` class and the delegate `accelerometer:didAccelerate` method which provides you with the data for each of the three axes, each being of `UIAcceleration` class. Each of the values returned has a range between **-1** and **+1** with **0** being the middle centre point. When the device is moved or tilted, these values increase or decrease.



The iPhone 4 adds another sensor: a three-axis gyroscope, and when combining the gyroscope with the accelerometer, this gives the iPhone 4 six axes on which it can operate and was designed to make the iPhone 4 more sensitive, responsive, and powerful for gaming.

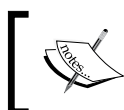
Understanding the Core Motion Framework

The Core Motion Framework is a system framework which obtains motion data from sensors on the iPhone device. The application can then use these values. Handling of the sensor data is handled within the Core Motion's own thread and it detects the motion events for the accelerometer and the gyroscope (*which is currently only available on the iPhone 4*):



The table below describes each of the components which make up the Core Motion framework:

CORE MOTION CLASSES	DESCRIPTION
<code>CMMotionManager</code>	This class defines a manager class which encapsulates measurements of motion data.
<code>CMAccelerometerData</code>	This class records measurement of device acceleration and gathers data from the accelerometer for each of its three axes.
<code>CMDeviceMotion</code>	This captures device-motion data from both the Accelerometer and Gyroscope.
<code>CMAttitude</code>	This is contained as part of the <code>CMDeviceMotion</code> class and contains properties which give different measurements of attitude, including the following: roll, pitch, and yaw.
<code>CMGyroData</code>	This class records the devices rate of rotation along its three spatial axes from the gyroscope.



The iPhone Simulator does not support the Accelerometer and Gyroscope features, so in the event that you want to run the examples shown in this chapter, you will need to deploy them to your iPhone device.

Sensing orientation

In order to determine which way the iPhone device is facing, you can get this information by using the `UIDevice` class and then using its orientation property.

By registering the `UIDeviceOrientationDidChangeNotification` notification method of the `UIDevice` class, you are not only told when the iPhone has been rotated between the **Portrait** and **Landscape** views, but also if the phone is in the facing up or facing down view. We will be taking a look at these in more detail when we create our sample application for this section.

Time for action – creating the OrientationExample project

We will now proceed with creating our `OrientationExample` project which will show how to determine when the device orientation changes:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.

3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family dropdown as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project which you want to create.
8. Enter **OrientationExample** and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once our project has been created, you will be presented with the Xcode interface, along with the project files that the template created for you within the Project Navigator Window:

9. Open the `OrientationExampleViewController.m` implementation file, located within the `Classes` folder of your project and locate the **viewDidLoad** method and add the following code:

```
// Implement viewDidLoad to do additional setup after loading
the view, typically from a nib.
- (void)viewDidLoad {
    [[UIDevice
currentDevice]beginGeneratingDeviceOrientationNotifications];

    [[NSNotificationCenter defaultCenter]
addObserver:self selector:@selector(hasOrientationChanged:)
name:@"UIDeviceOrientationDidChangeNotification"
object:nil];
    [super viewDidLoad];
}
```

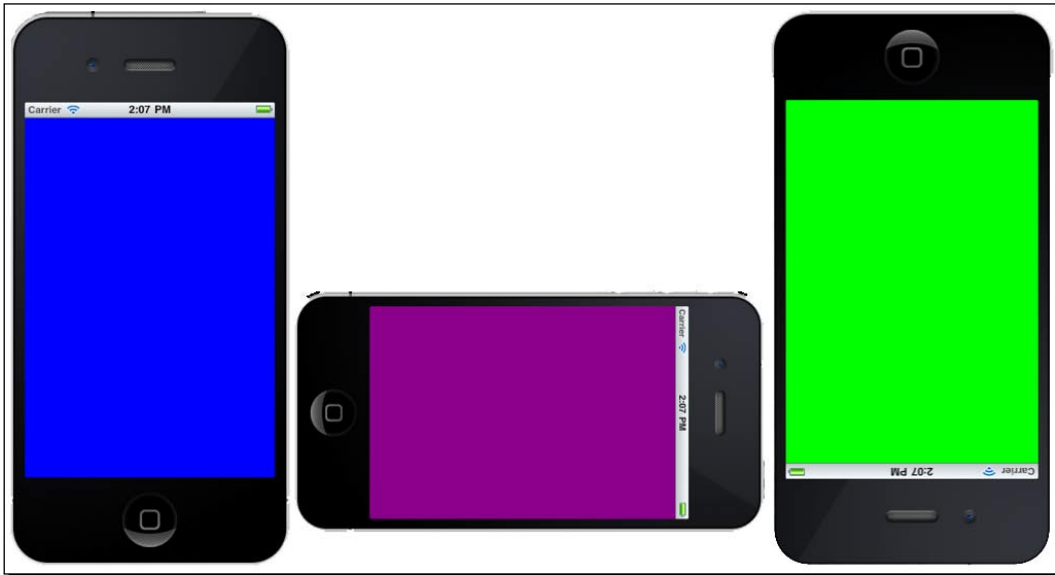
10. Next, we need to implement the method which will be responsible for handling when changes in orientation have been detected. Add the following code:

```
- (void)hasOrientationChanged:(NSNotification *)notification {
    UIDeviceOrientation currentOrientation;
    currentOrientation = [[UIDevice currentDevice] orientation];

    switch (currentOrientation) {
        case UIDeviceOrientationFaceUp:
            self.view.backgroundColor = [UIColor brownColor];
            break;
        case UIDeviceOrientationFaceDown:
```

```
        self.view.backgroundColor = [UIColor magentaColor];
        break;
    case UIDeviceOrientationPortrait:
        self.view.backgroundColor = [UIColor blueColor];
        break;
    case UIDeviceOrientationPortraitUpsideDown:
        self.view.backgroundColor = [UIColor greenColor];
        break;
    case UIDeviceOrientationLandscapeLeft:
        self.view.backgroundColor = [UIColor redColor];
        break;
    case UIDeviceOrientationLandscapeRight:
        self.view.backgroundColor = [UIColor purpleColor];
        break;
    default:
        // Handle cases where orientation fails
        self.view.backgroundColor = [UIColor blackColor];
        break;
    }
}
```

We have finally made it and now we are ready to Compile, Build and Run our **OrientationExample** application. Given that you have typed in everything correctly, your code should compile without any issues. Try changing the different views of orientation by pressing the *Command + Left Arrow* and *Command + Right arrow* if you are running this within the iPhone simulator:



So there you have it; in just a few simple steps, you can make your applications determine and respond to a device when its view orientation has changed.

What just happened?

What we covered in this section were the steps involved in to creating our `OrientationExample` project. We began by telling our iPhone device to start generating notifications for each of the changes in orientation and then set up an observer to the `UIDeviceOrientationDidChangeNotification` notification class which is fired each time the device changes its orientation. Next, we determine the current orientation that our phone is in by using deriving this from the `UIDeviceOrientation` class. We then proceed and determine what the current orientation is by using a `case` statement and then change the background color of our view.

Have a go hero – modify the OrientationExample application

I will let you put into practice what you have just learnt.

Our application needs be modified slightly to allow it to only change the color when the device is in Portrait and Landscape Left. One way to do this would be as follows:

1. Modify the `hasOrientationChanged` method in the `OrientationExampleViewController.m` implementation file.
2. Comment out cases where they do not equal `UIDeviceOrientationPortrait` and `UIDeviceOrientationLandscape`.
3. Once you have done this, Compile, Build and Run your application.

Once you have that working, you will have an application that provides interaction with the user when the device is in Portrait and Landscape.

Pop quiz – sensing orientation

1. What class determines the current orientation of the iOS device?
 - a. `UIDevice`
 - b. `UIOrientation`
 - c. `UIDeviceOrientation`
2. What method needs to be initialized to allow for device orientation notifications?
 - a. `UIDeviceCurrentDevice`
 - b. `beginGeneratingDeviceOrientationNotifications`

Detecting device tilting

In this example, we will be looking at how to derive the values from our Accelerometer and Gyroscope and set the background color of our view as well as setting the background to fade in and out from transparent to opaque when the device is tilted.

By using the Accelerometer and Gyroscope, you will be able to create some great applications; from car games to flight simulators. You will have a reasonable amount of understanding on how to go about implementing both of these in your own applications after having gone through the next section.

Time for action – creating the AccelGyroExample project

We will now proceed with creating our AccelGyroExample project which will show how to determine and handle the accelerometer and gyroscope features:

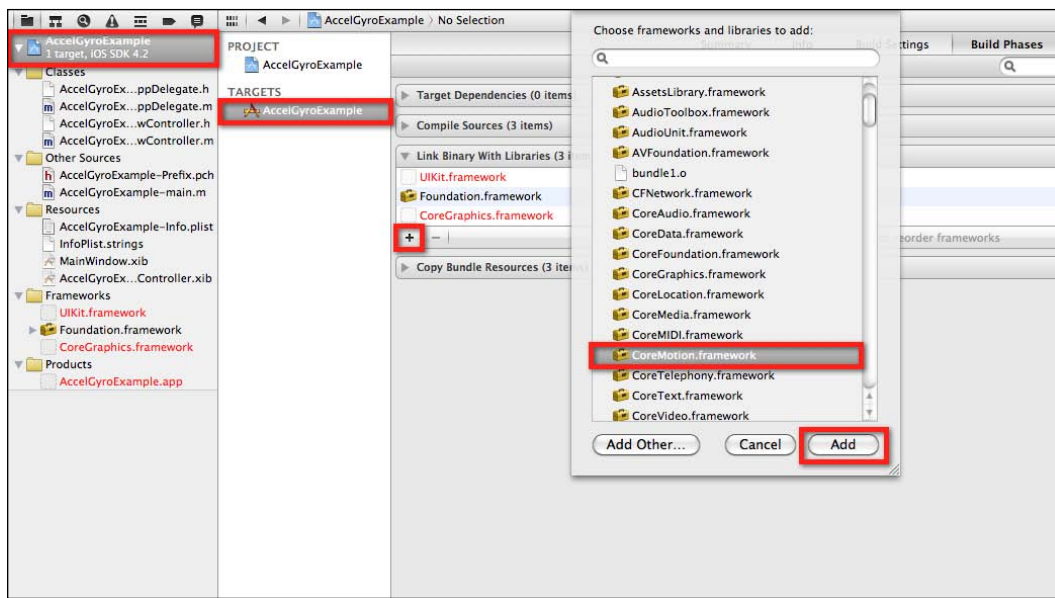
1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project which you want to create.
8. Enter `AccelGyroExample` and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once your project has been created, you will be presented with the Xcode interface, along with the project files that the template created for you within the Project Navigator Window.

Before we can start using the Accelerometer and Gyroscope features of the iPhone, we need to add an important framework to our project to enable us to use the accelerometer, so let's do that now by following these steps:

1. In the project navigator, select and highlight your project.
2. Select and click on your Target.
3. Select the **Build Phases** tab.

4. Open the **Link Binaries With Libraries** expander.
5. Click the **+** button.
6. Select the **CoreMotion.Framework** framework from the list.
7. Click **Add**:



Now that you have added the **CoreMotion.Framework** into your project, you need to import the code into the View Controller which will be responsible for handling the motion:

1. Open the `AccelGyroExampleViewController.h` implementation file, located within the `Classes` folder of your project and add the following code. You will notice that we have also included the `CoreMotion/CoreMotion.h` interface file so that we can make use of the methods that this class contains:

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

@interface AccelGyroExampleViewController : UIViewController
<UIAccelerometerDelegate>{
    CMMotionManager *motionManager;
}

@property (nonatomic, retain) CMMotionManager *motionManager;

@end
```

2. Next, we need to create a property object which will enable us to reference the instance variable **motionManager** from within our `AccelGyroExampleViewController.m` implementation file:

```
#import "AccelGyroExampleViewController.h"
```

```
@implementation AccelGyroExampleViewController
```

```
@synthesize motionManager;
```

3. Next, open the `AccelGyroExampleViewController.m` implementation file, located within the `Classes` folder of your project and add the following code as shown in the code snippet below:

```
// Handle processing of the Accelerometer
-(void)handleAcceleration:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    UIAccelerationValue xAxes;
    UIAccelerationValue yAxes;
    UIAccelerationValue zAxes;

    xAxes = acceleration.x;
    yAxes = acceleration.y;
    zAxes = acceleration.z;

    if (xAxes > 0.5) { // Check to see if we are Moving Right
        self.view.backgroundColor = [UIColor purpleColor];
    } else if (xAxes < -0.5) { // Check to see if we are Moving Left
        self.view.backgroundColor = [UIColor redColor];
    } else if (yAxes > 0.5) { // Check to see if we are Upside
        Down.
        self.view.backgroundColor = [UIColor yellowColor];
    } else if (yAxes < -0.5) { // Check to see if we are Standing
        Up.
        self.view.backgroundColor = [UIColor blueColor];
    } else if (zAxes > 0.5) { // Check to see if we are Facing Up.
        self.view.backgroundColor = [UIColor magentaColor];
    } else if (zAxes < -0.5) { // Check to see if we are Facing
        Down.
        self.view.backgroundColor = [UIColor greenColor];
    }
    double value = fabs(xAxes);
    if (value > 1.0) { value = 1.0;}
    self.view.alpha = value;
}
// Handles rotation of the Gyroscope
-(void)doGyroRotation:(CMRotationRate)rotation {
    double value =
```

```

        (fabs(rotation.x)+fabs(rotation.y)+fabs(rotation.z))/8.0;
    if (value > 1.0) { value = 1.0;}
    self.view.alpha = value;
}

```

- 4.** Next, open the `AccelGyroExampleViewController.m` implementation file, located within the `Classes` folder of your project and locate the **`viewDidLoad`** method and add the following code:

```

// Checks to see if Gyroscope is available on the device
- (BOOL) isGyroscopeAvailable
{
#ifdef __IPHONE_4_0
    CMMotionManager *gyroManager = [[CMMotionManager alloc] init];
    gyroManager.gyroUpdateInterval = 1.0/60.0;
    BOOL gyroAvailable = gyroManager.gyroAvailable;
    [gyroManager release];
    return gyroAvailable;
#else
    return NO;
#endif
}

```

- 5.** We now need to add some code to our **`viewDidLoad`** method. With the `AccelGyroExampleViewController.m` still open, locate the **`viewDidLoad`** and add the following code:

```

- (void)viewDidLoad {
// Set up our accelerometer interval and delegate
UIAccelerometer *accelerometer = [UIAccelerometer
    sharedAccelerometer];
accelerometer.updateInterval = 0.5;
accelerometer.delegate = self;

// Check to see if the device supports the Gyroscope feature
if ([self isGyroscopeAvailable] == YES) {
    motionManager = [[CMMotionManager alloc] init];
    [motionManager
    startGyroUpdatesToQueue:[NSOperationQueue currentQueue]
    withHandler:^(CMGyroData *gyroData, NSError *error)
    {
        [self doGyroRotation:gyroData.rotationRate];
    }];
}
else { // Device does not support the gyroscope feature

```



```
        NSLog(@"No Gyroscope detected. Upgrade to an iPhone 4.");  
        [motionManager release];  
    }  
    self.view.backgroundColor = [UIColor magentaColor];  
    [super viewDidLoad];  
}
```

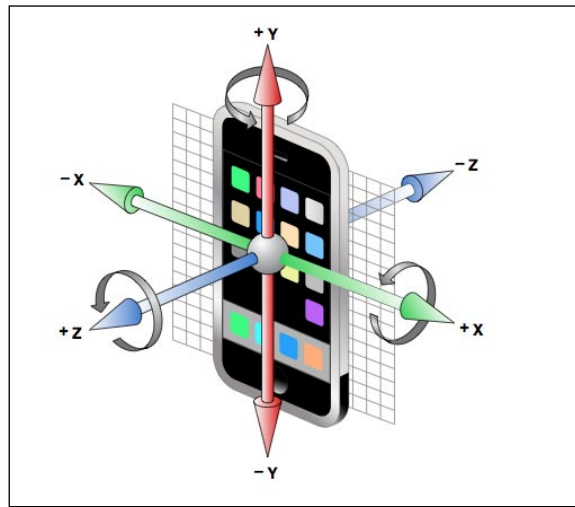
6. Our final step is to release the memory used by our view controller objects which we declared in our `AccelGyroExampleViewController.h` interface file, located within the `Classes` folder of your project. Add the following highlighted code as shown in the code snippet below to your **dealloc** method:

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

7. We have finally made it to the end of the chapter and now we are ready to Compile, Build and Run our **AccelGyroExample** application.

 Since the iOS Simulator does not support the accelerometer and Gyroscope features, you will need to deploy this example to your iPhone device in order to see this working.

The screenshot below shows how the iPhone responds to changes on its three axes when the iPhone is tilted. Under normal gravity, each of these values will be between -1 and +1 with a value of 0 being the middle centre point. Moving the phone in a rapid motion will increase these values:



What just happened?

What we have done in this section is implement the `UIAccelerometerDelegate` protocol so that we can use this within our `AccelGyroExampleViewController.m` implementation file. We then declare an instance `CMMotionManager` variable **motionManager** which will enable us to use the accelerometer and gyroscope features. We then need to synthesize our property which we declared within our `AccelGyroExampleViewController.h` interface file. If we don't declare this, we will receive warning error messages which can cause unexpected application errors. We have also mentioned this in previous chapters as this is common practice to release the memory used by these objects once you have finished using them.

Next, we declare our methods which will handle the accelerometer and the gyroscope features. In the first part, we declare a delegate to the `UIAccelerometer` class, and then derive the values for the x, y, and z axes, which will be used to determine the current device orientation; then we set the background color accordingly. As a final step, we set the background alpha property from transparent to opaque depending on whether the value is within the range 0.0 to 1.0, where 0.0 represents totally transparent, and 1.0 represents opaque.



When you set the alpha property of a view, it only affects the current view and does not affect any of its embedded subviews. The `fabs` function is a C/C++ library function which returns the absolute value of X.

In our next step, we determine by using the `#ifdef __IPHONE_4_0` directive if the device currently in use is an iPhone 4. If this is the case, it then checks to see if the device supports the gyroscope feature and a Boolean status **YES** is returned; otherwise **NO** is returned.

Next, we set up our `UIAccelerometer` delegate and update the interval to be twice per second in order to request updates. We then make a call to our `isGyroscopeAvailable` function to check to see if the gyroscope feature is supported. We then set up a call to the `startGyroUpdatesToQueue` function and add a handler to call our `doGyroRotation` function which then updates the alpha blend color of our view. If no gyroscope feature is supported, this is logged out to the debug window.



To start receiving and handling rotation-rate data for the gyroscope feature, you need to create an instance of the `CMMotionManager` class and call one of the following methods to it.

The following table explains each of the method calls relating to the `CMMotionManager` class:

CMMotionManager Methods	DESCRIPTION
<code>startGyroUpdates</code>	When this method is called, Core Motion kicks in and continuously updates the <code>gyroData</code> property of the <code>CMMotionManager</code> class with the latest measurement of activity.
<code>startGyroUpdatesToQueue:withHandler</code>	Before calling this method, you need to ensure that you have set the update interval of the <code>gyroUpdateInterval</code> property. When this method is called, it creates an <code>NSOperationQueue</code> event which queues the gyroscope event which then fires when the update interval has been reached, then calls the function and passes it the latest gyroscope data.
<code>stopGyroUpdates</code>	This method turns off the core motion sensors and stops all updates of motion data. It is a good idea to always stop gyro updates as this will save battery power.

Summary

In this chapter, we learned about the MultiTouch Architecture and how to go about detecting taps, swipes, pinching and device shaking and tilting. We also looked into one of the new features which come as part of the iPhone 4, that being the Gyroscope feature.

Now that we have learned about the various ways in which we can handle and respond to device Multi-Touch features, we are ready to learn how to debug projects and explore the new debugging feature improvements and code analysis to help you eliminate bugs within your applications.

In the next chapter, we will be taking a look into how to go about **Debugging your Xcode Projects**, and taking a look at the new and improved debugger, as well as learning how to go about creating a new debugging project. We will also look at how we can use Fix-It to correct syntax errors and code as you type. We will also learn about breakpoints, and how to define schemes within your project workspace, as well as using static analysis to show code flow.

8

Debugging Xcode Projects

In this chapter, we will look at how we go about debugging our projects through the use of the various debugging tools that Xcode provides.

We will look at the following methods to debug code:

- ◆ *Fix-it, which corrects code as you type and provides you with some great coding alternatives*
- ◆ *Static Analysis, which shows you potential coding errors, that is, memory leaks and dead or unreachable code*

In this chapter, we will be covering the following topics:

- ◆ Introducing the new and improved debugger
- ◆ Creating, running, and debugging projects
- ◆ Setting up project schemes using the Scheme Editor
- ◆ Navigating through threads and stacks within the Debugger
- ◆ Finding potential coding errors by using Fix-it and the Static Analyzer

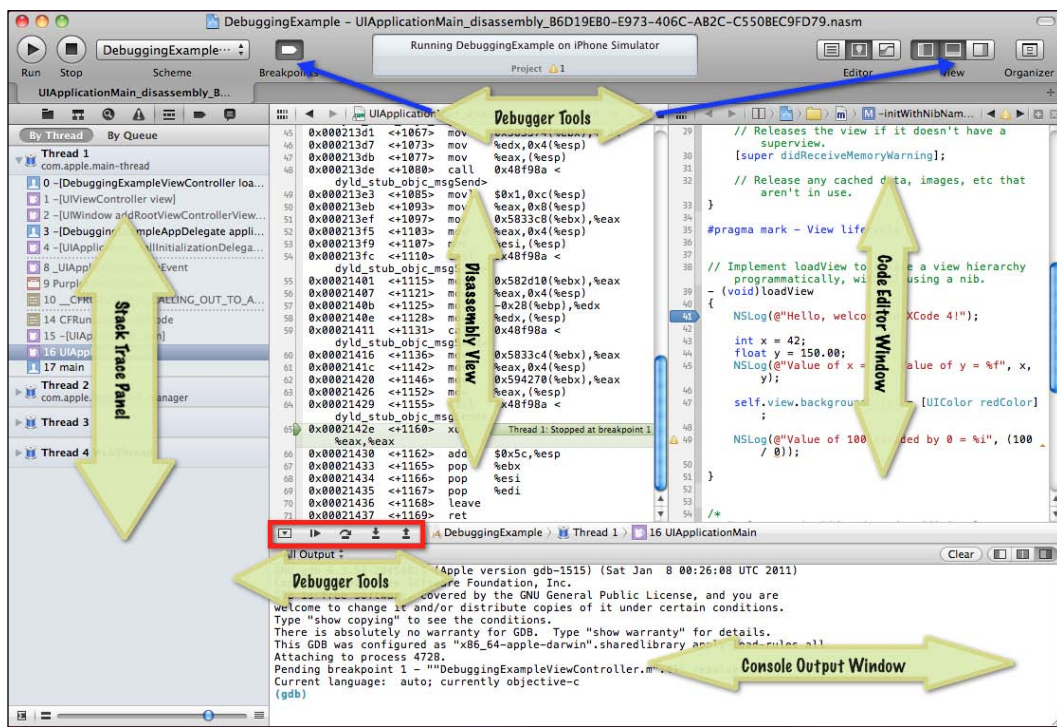
We have got quite a bit to cover, so let's get started.

Introducing the new and improved Debugger

The **Xcode Debugger** is a collection of sophisticated tools that allows you to monitor your application as it runs, and provides you with information about the current state of the program line by line.

This has been fully integrated into the Xcode 4 workspace environment and any errors or issues will be automatically displayed within the Project Navigator. Through effectively using the Debugger, it will give insights into how your code is performing and, most importantly, helpful pointers to where it may be going wrong.

The following screenshot shows the various parts of the Xcode Debugger:



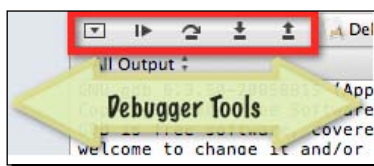
The Xcode Debugger contains a console pane, which is used to log output messages as well as variables, and a register pane, which can display the values of variables on the stack, as well as the methods.

The Xcode Debugger window consists of the following parts; these are explained in greater detail in the following sections.

Debugger toolbar

The Debugger toolbar consists of a number of items for you to be able to step into your code line by line to determine where an error has occurred. This also enables you to check the value of a particular variable to ensure that it is being assigned a value correctly. These options become available when your execution stops at a breakpoint within your code.

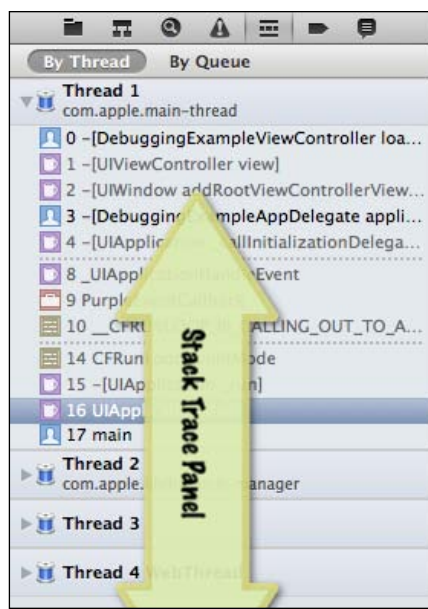
The screenshot below displays the debugging toolbar which becomes available when your project is in debugging mode:



Stack trace panel

The Stack Panel window enables you to look at each file running within the stack. This view is particularly useful if your code stops at a breakpoint, and you want to see a list of all threads that were running up to the point the program halted.

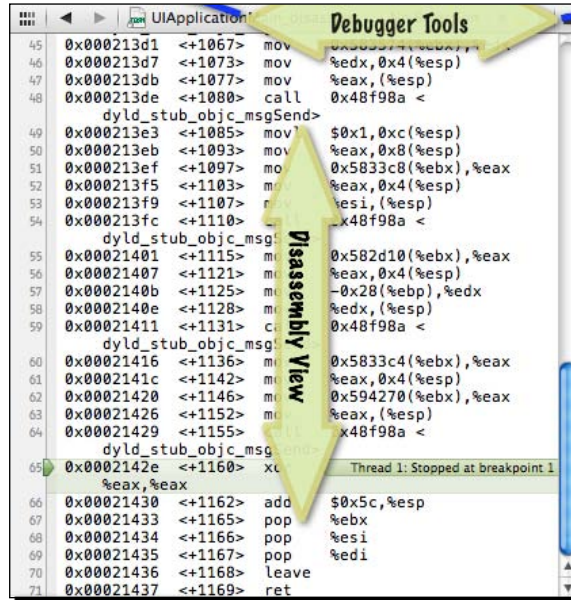
The screenshot below displays the stack trace panel toolbar, displaying all of the associated threads that your application is running when a breakpoint has been encountered within your code:



Disassembly view

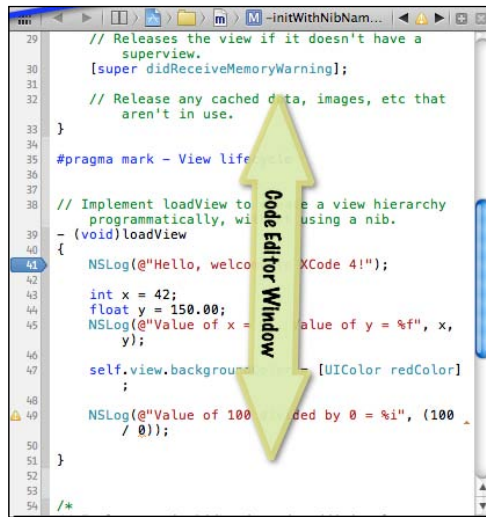
The Disassembly window enables you to trace through a stack dump within your program logic to determine where the crash occurred. You are also able to examine program variables and registers of all files within your project.

The screenshot below displays the Xcode Disassembly view whenever your program causes a crash, due to a piece of faulty code that has been detected within your code:



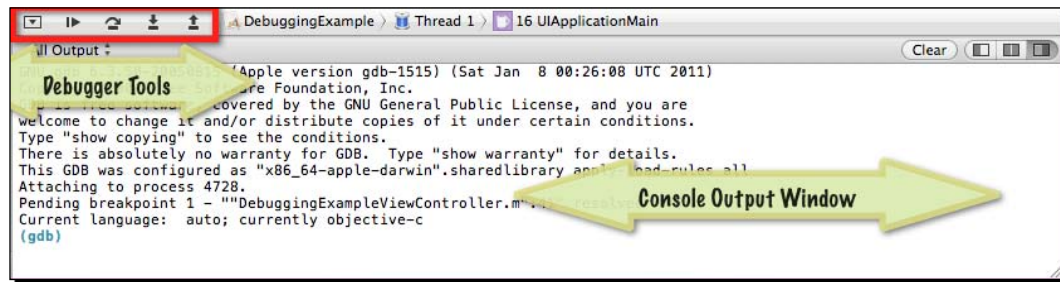
Code Editor window

The Code Editor window shown below is where you start to provide the instructions to build your application. This editor is where you can navigate to other files within your workspace, as well as enabling you to debug your projects:



Console output window

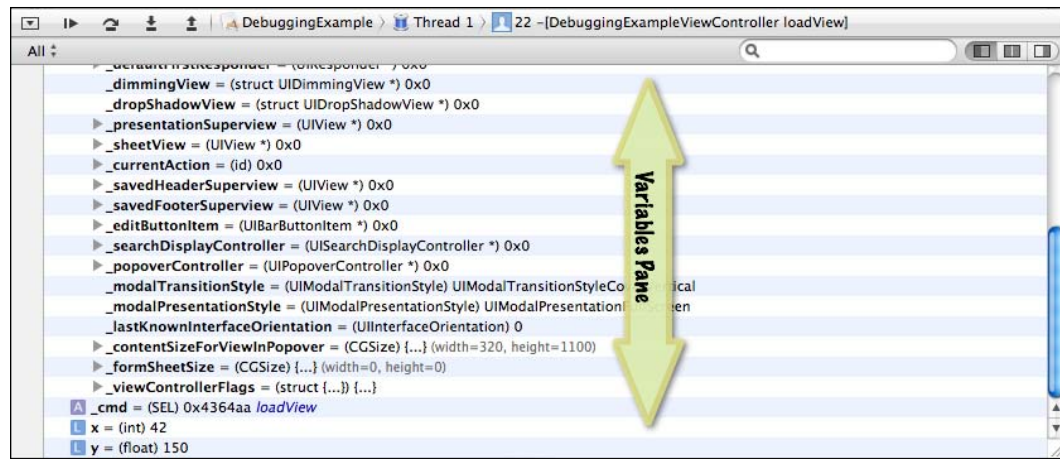
The console output window as shown below displays all compiler messages as well as any messages logged using the NSLog function:



You can specify the type of output the console displays by using the pop up menu in the top-left corner of the console pane. The descriptions of each of these types are given below:

CONSOLE OUTPUT TYPES	DESCRIPTION
All Output	Displays both Target and Debugger Output. This is the default view for this window.
Debugger Output	Only displays output information relating to the Debugger.
Target Output	Only displays output information relating to the Target.

The following screenshot displays a list of all the variables and registers that are currently used by your application:



You are able to specify which items you would like to display by using the pop up menu item within the variables pane. A description of each of these types is given below:

VARIABLES PANE TYPES	DESCRIPTION
Auto	Displays a listing of all recently accessed variables.
Local	Displays only locally accessible variables.
All	Displays both Variables and Registers. This is the default view for this window.

The default setting for both of these views is **All** or **All Output**. Don't worry; we will be using these when we start to debug our sample application in the section *Running and Debugging the Project*.

Creating a new debugging project

Before you can start to use the Debugger, you will need to create an application to debug. We will create a small simple application, which will highlight this. The program will set the background color of the view to red and display some text in the console view window.

Time for action – creating the DebuggingExample project

Before we can proceed, we first need to create the `DebuggingExample` project. To refresh your memory, you can refer to the section *Creating your first iPhone application* which we covered in *Chapter 2, Introducing the Xcode 4 Workspace*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. You will be prompted to choose a project type and template.
4. From the Project Template Dialog, select the **View-based Application** project type.
5. Ensure that you have selected **iPhone** from the Device Family drop-down as the type of View to create.
6. Click on the **Next** button to proceed to the next step.
7. Specify a name for the project that you want to create.
8. Enter *DebuggingExample* and then click on the **Next** button to proceed to the next step in the wizard. You will then be asked to choose a location where you would like to save the project.

Once our project has been created, you will be presented with the Xcode interface, along with the project files that the template created for you within the Project Navigator window.

We now need to implement the code, which will be used to set the background color of our view, as well as being able to log messages out to the Debugger console window:

1. Next, open the `DebuggingExampleViewController.m` implementation file.
2. Then scroll down and locate the `viewDidLoad` method, and enter the following code snippet:

```
// Implement viewDidLoad to do additional setup after loading
// the view, typically from a nib.
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSLog(@"Hello, welcome to XCode 4!");

    int x = 42;
    float y = 150.00;
    NSLog(@"Value of x = %i, Value of y = %f", x, y);

    self.view.backgroundColor = [UIColor redColor];
    NSLog(@"Value of 100 divided by 0 = %i", (100 / 0));
}
```

What just happened?

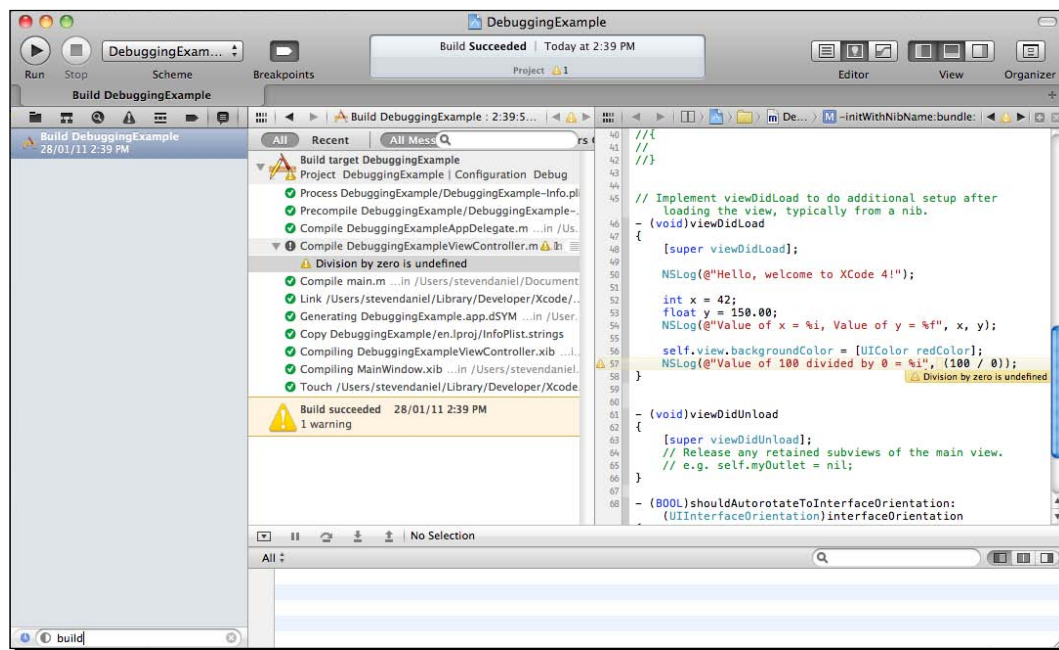
In the above section, we looked at the steps involved in creating our Debugging example project. We then inserted some code in our **viewDidLoad** method and declared two variables **int** and **float** and assigned each a value, before logging messages and the values of these variables out to our Debugger Console Output window using the **NSLog** method.

We then proceeded to set the background color of our view to be red using the `UIColor` class, before finally logging the value of what 100 divided by 0 is (which as we know will give us a runtime error). In the next part, we will look at how we are able to Build and Run our application.

Running and debugging the project

Now you are ready to Build and Run your application. Click on the **Run** button within the Xcode 4 IDE, or select the **Product | Run DebuggingExample**.

You will notice that when you build the application, there is an immediate problem and the build fails. The following screenshot shows the Build Log results tab with the resultant error, as well as the line of code that the compiler has found to be causing the problem:



For more information on the Debugger, refer to the Apple Developer Documentation at: <http://developer.apple.com/library/mac/#documentation/toolslanguages/conceptual/xcode4userguide/debugging/debugging.html>.

Handling errors

No software application is ever perfect. There will be times when you will need to revisit the code to correct a coding problem, or to change the way the code is performing due to a change in user requirements. Xcode provides you with the tools to make your life easier, and prevent coding issues from happening. There are three common forms of error, which we will see in the following sections.

Runtime errors

These types of error cause your program to stop executing. It can be caused by an unhandled exception due to an out of memory issue, or if you are writing some data to a database, you may have exceeded the allowable size that the field can handle.

In most cases, you will receive an error message, but in some cases your application will likely crash or hang.

Syntax errors

These types of errors are the most obvious, simply because your program will not compile (and therefore won't run) until all of them are fixed. Generally, syntax errors come from typographical errors.

The Objective-C compiler in Xcode is case-sensitive, which means that `UIColor` and `UIColor` are treated differently. For example, in Objective-C, the compiler can understand the following:

```
self.view.backgroundColor = [UIColor redColor];
```

But if you type in:

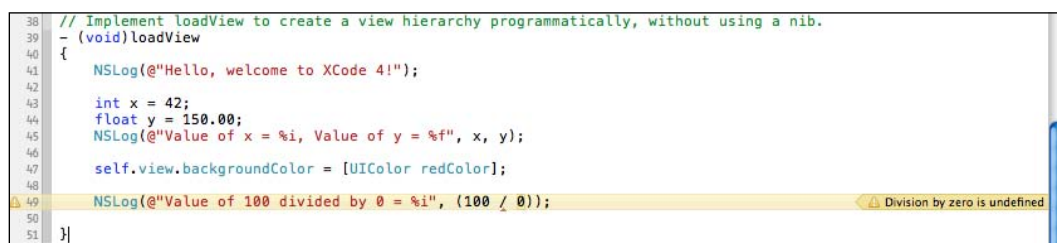
```
self.view.backgroundColor = [UIColor redcolor];
```

The compiler will call it a syntax error, because you've specified language-specific (syntax) that it can't recognize. Whilst this may be a small error to you, it is phenomenal to the computer. Even forgetting the semicolon at the end of that line would give you a syntax error. Syntax errors are very easy to identify, since the compiler recognizes them on its own.

Logic errors

These types of errors occur when you have created a piece of code logic which is not performing correctly and producing the correct results. It is worth mentioning here, that the program will do what you tell it to do, the classic catch phrase, *Garbage-In-Garbage-Out (GIGO)*.

A classic example of a logic error would be when you are dividing a value by zero, but not checking to see if the denominator is zero first. During the building phase of your project, Xcode will bring your attention to this error, but will still go ahead and build and run the application:



```
38 // Implement viewDidLoad to create a view hierarchy programmatically, without using a nib.
39 - (void)viewDidLoad
40 {
41     NSLog(@"Hello, welcome to Xcode 4!");
42
43     int x = 42;
44     float y = 150.00;
45     NSLog(@"Value of x = %i, Value of y = %f", x, y);
46
47     self.view.backgroundColor = [UIColor redColor];
48
49     NSLog(@"Value of 100 divided by 0 = %i", (100 / 0));
50
51 }
```

As you can see from the previous screenshot, Xcode has cleverly detected that the piece of code within the `NSLog` statement can potentially cause your application to stop executing and crash your application. A better approach would be to store these values in separate variables and check to see if the value is greater than zero prior to executing the statement.

Using Fix-it to correct code as you type

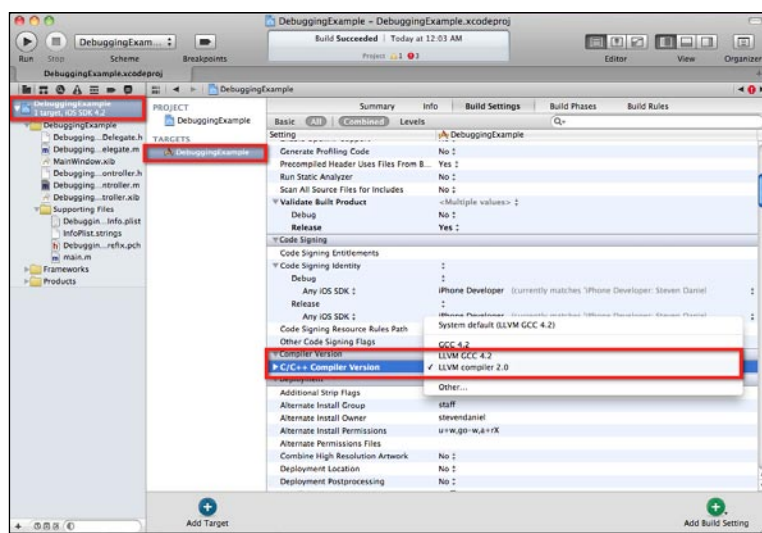
When your target is set to use the LLVM compiler, a new feature called Fix-it scans your source code as you type. Fix-it marks syntax errors and misspelt words with a red line at the position where the error was located and a red exclamation mark symbol is placed within the gutter area of your project workspace.

Time for action – setting up the LLVM compiler

To set up your project to use the LLVM Compiler, you will need to adjust the settings for your project workspace by following these simple steps:

1. In the project navigator, select and highlight your project.
2. Select and click on your Target.
3. Select the **Build Settings** tab.
4. Scroll down to the **C/C++ Compiler Version**.
5. Select the **LLVM compiler 2.0** option from the list.

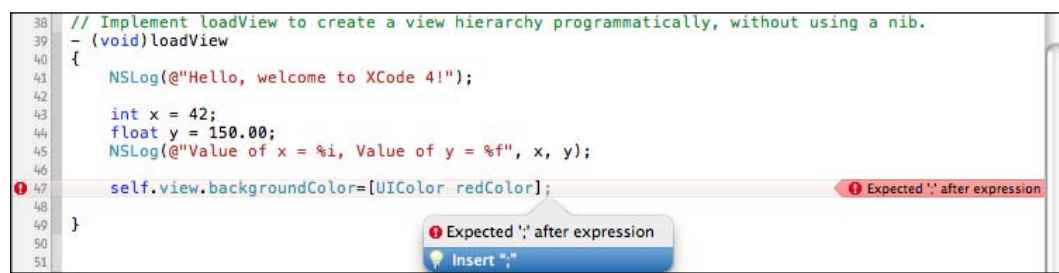
The following screenshot shows you how to go about setting the project target to use the LLVM 2.0 Compiler:



Now that you have changed the default compiler to use the LLVM Compiler 2.0, your project will start to use and compile all code using this new version.

Take for example, the following piece of code as shown in the screenshot below where we are setting the background color of our view to red.

We have obviously mistakenly forgotten to add a semicolon to the end of our method and the LLVM compiler has detected this for us and has provided us with some suggestions. To accept one of the proposed changes given by the compiler, click on the item and watch your code update automatically. Clicking on the symbols located in the gutter bar displays an error message describing the possible syntax error and in some cases offers to repair this for you automatically:

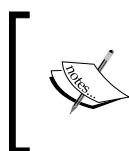


What just happened?

In the above section, we looked at the steps required to configure our project to use the LLVM 2.0 Compiler. We also took a look at the use of Fix-it to correct coding pitfalls when typing directly into the Xcode 4 IDE.

As you can see, Fix-it is a great tool for finding and fixing your code on-the-fly and is a great companion to the rigorous testing performed by the static analyzer tool, which will walk through thousands of potential code paths, looking for places where code, while perfectly valid, could potentially cause unexpected results.

Some of the common programming pitfalls are due to allocating memory that is never released, improperly constructed code loops, or `case` statements that have not properly been assigned a default value, resulting in them never falling into a condition.



For more information on Fix-it, please refer to the Apple Developer Documentation at: <http://developer.apple.com/library/mac/#documentation/toolslanguages/conceptual/xcode4userguide/debugging/debugging.html>.

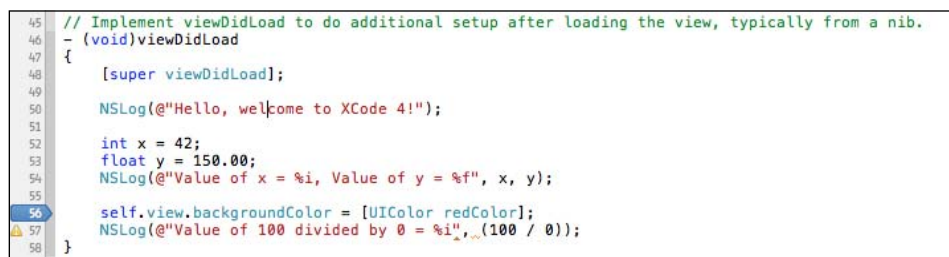
Debugging with breakpoints

Although you can use the Debugger to pause execution of your program at any time and view the state of the running code, it's usually helpful to set breakpoints before running your executable so you can stop at known points and view the values of variables in your source code.

A breakpoint is basically just an instruction in code that tells the application to "stop" when the breakpoint is reached, and the execution of the program pauses, waiting for further instructions as to what to do next. During this phase, you have the opportunity to either inspect the current values of any of the properties, or step through the code.

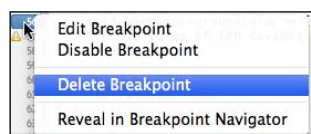
Adding a breakpoint is easy and is done by following these simple steps:

1. Click within the grey gutter area to the left of the code editor using your mouse.
2. You will notice that a blue arrow appears at the line which you have selected:



Removing Breakpoints are a breeze too and can be done as follows:

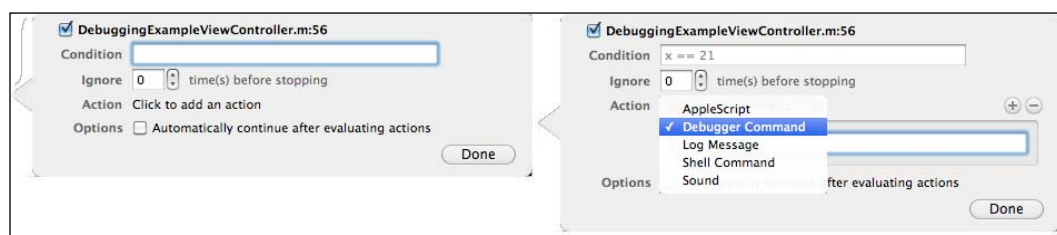
1. Right-click on the breakpoint arrow.
2. Then select **Delete Breakpoint** from the list of options or simply drag it off from the gutter area:



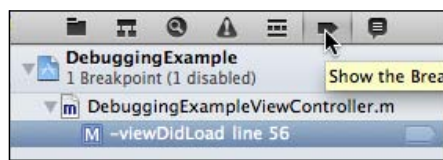
The Breakpoints menu includes a variety of other useful debugging options, which are explained below:

- ◆ **Edit Breakpoint:** Several options can be set for each breakpoint, such as setting the number of times to pass through the breakpoint before it is triggered, or specifying a conditional breakpoint.

There is also an option of having it perform an action, such as logging a message to the Debugger console window, or executing a shell command, or even playing a sound:



- ◆ **Disable Breakpoint:** This sets the color of the breakpoint to more of a washed out transparent blue. The breakpoint will still be displayed within the gutter bar, but will no longer be active and your program will not stop at this location.
- ◆ **Reveal in Breakpoint Navigator:** This will display a list of all breakpoints (*with this option set*) within the Breakpoint Navigator pane. It displays the module name, function, and line number at which the breakpoint occurs as shown in the screenshot below:



Using NSLog to track changing properties

There are times when you want to place your own form of debugging into your projects. These could be, for instance, to check on the status of a variable or to check why an event is not firing inside your code. This is where the `NSLog` function comes in.

Any statements that you log using the `NSLog` function will be echoed out to the Xcode's Debugger Console window, which appears when you build and run your application. The console window can be displayed by pressing the `Command + Shift + R` key combinations.

The `NSLog` function takes an `NSString` argument that can optionally contain what are called string format specifiers. The table below provides a list of some of these types:

STRING FORMAT SPECIFIER	DESCRIPTION
<code>%@</code>	An Objective-C object using the <code>description</code> or <code>descriptionWithLocale</code> : results.
<code>%d</code>	Displays the result as a signed integer (32-bit).
<code>%f</code>	Displays the result as a floating point value.
<code>%c</code>	Displays the result as an unsigned character.
<code>%s</code>	Displays the result as a null-terminated character string array.
<code>%x</code>	Displays the result as an unsigned hexadecimal value.



The table above only lists the most commonly used string formats. If you are interested in seeing the full list, you can access these at: <http://en.wikipedia.org/wiki/Printf>.

The `NSLog` function takes a variable number of arguments which are inserted into the string at the location of the specifiers, which is quite similar to the `printf()` function as found within the Standard C library.

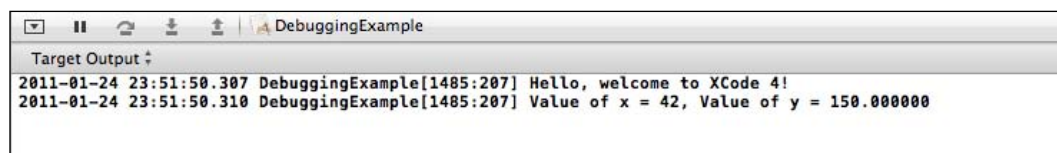
The string format specifier is basically displayed as a percent (%) symbol, which is then followed by one or two characters indicating the type of the variable that will be displayed.

Consider the following code snippet below. It has been written using the C-Notation:

```
int    x = 42;
float  y = 150.00

printf("%s", "Hello, welcome to XCode 4!");
NSLog(@"Hello, welcome to XCode 4!");
NSLog(@"Value of x = %i, Value of y = %f", x, y);
```

When you Build and Run the above code output, the text is logged out to the Xcode Debugger output window as shown in the screenshot below:



It is worth mentioning that the **NSLog** function is merely a means of logging output to the window when you want to see if an event is being fired or the value of a variable. It is highly recommended that prior to releasing your product for distribution, you remove or comment out all references made to **NSLog** from within your project as this will end up writing out to the filesystem, and will take up unnecessary space on the user iOS device.

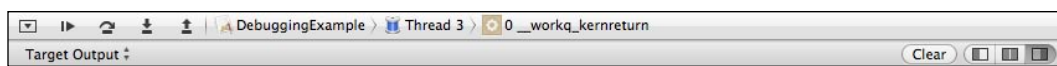
Exploring the new Debugger

Given the flexibility of the Xcode Developer Tools you have come to expect, there are several ways to use the debugging features within the Xcode development environment. You have several ways in which you can debug your projects. These are listed below:

- ◆ The Code Editor
- ◆ The Debugger window
- ◆ The Console view

Debugging features in the Code Editor

The debugging features only apply to the code editor within the Xcode development workspace. If you set breakpoints in your code, you will see the Debugger toolbar when your program stops at the breakpoint in your code. The screenshot below shows the Debugger toolbar:



The Debugger toolbar is context-sensitive, which means that it will only be visible while you are in the middle of debugging. When your debugging session completes, the normal Code Editor Toolbar is restored. The buttons on the debugging toolbar enable you to:

- ◆ Open or close the Debugger area window
- ◆ Pause or resume execution of your code
- ◆ Step over a method
- ◆ Step into a method
- ◆ Step out of the current method
- ◆ See exception and other messages, such as the current method name and the executing threads



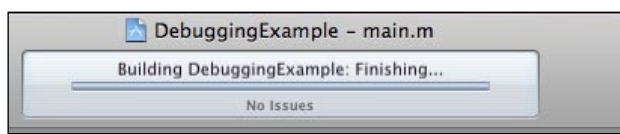
It is worth mentioning that if you hold down the *Option* or *Option + Shift* buttons when clicking on the *step over* or *step into* controls, this will provide you with different ways in which you can step through your code.

You can step through your code using the assembly language instructions view, instead of by statements; or if you prefer, you can step directly into the currently active thread.

The Activity Viewer/Progress window

We touched on this briefly in *Chapter 2, Introducing the Xcode 4 workspace*, so this section will just be a refresher in case you may have forgotten. Basically, the Activity Viewer (or Progress window) shows you the progress of tasks which are currently executing. These tasks may fall along the lines of building your project, or you may be using the Static Analysis feature to analyze your project for syntax or code errors.

The Activity Viewer can also show you any compiler warnings or syntax errors and any information relating to project builds:



Defining a scheme for project builds using the Scheme Editor

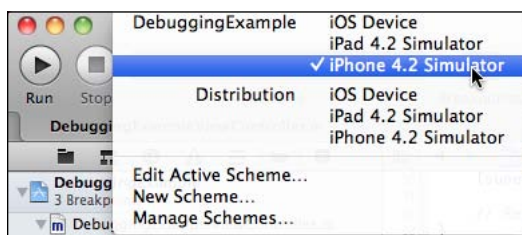
Schemes are not new to Xcode 4; they have existed since the release of Xcode 3. In previous releases of Xcode, you had to configure each of the items separately when setting an active target, a build configuration, and an executable. This posed many issues as all of these were linked to one another. This is where Schemes come in.

Schemes can be thought of as separate configurations, meaning that you can create a scheme to specify which targets to build, what configuration build to use, and what executable environment to use when the product specified by the target is launched. This could be if you wanted to target a specific iOS version, or if you wanted to have the application launch within the simulator.

Time for action – using the Scheme Editor to define a Scheme

In Xcode 4, whenever you open an existing Xcode project, or create a new one, Xcode 4 automatically creates a default scheme for you. This scheme allows you to either test your application within the iOS Simulator or have it deployed to an iOS device. Additional schemes can be created, and in this section we will see how this can be done:

1. To select a **scheme**, you can use the Scheme popup menu, which is located in the upper-left corner of the Xcode workspace:



2. In order to create a new scheme, choose the **New Scheme...** option. Alternatively, if you wanted to edit the active scheme, you would choose the **Edit Active Scheme...** menu option. These options are also available under the **Product** menu bar.

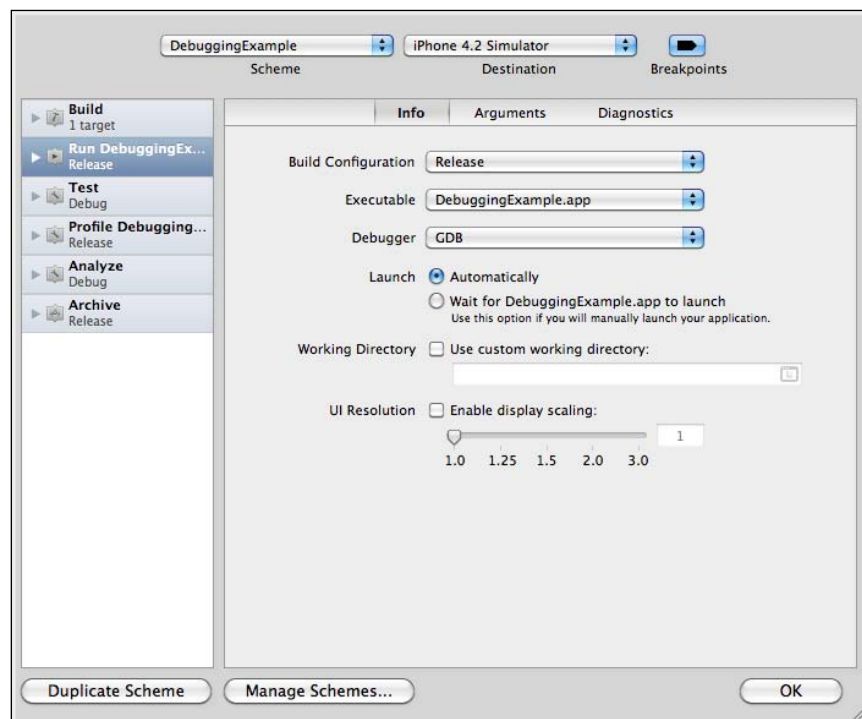
Each scheme can be set to do a specific task; for instance, you may have a scheme to do a Debug build, and one to handle the Release or Distribution. Various types of build options are available for building, testing, running, profiling (using instruments), and archiving your products.

There is no limit on the number of schemes that you can define. However, only one scheme can be active at a time.

3. Schemes can also be managed by choosing the **Manage Schemes...** option from the popup menu, or similarly from the Product menu.

You can specify whether schemes should be stored per project, in which case it will be made available to every workspace that includes that project; or be stored within the workspace environment it's currently in.

The following screenshot shows you how you can go about customizing the active scheme:



You can specify the type of Build Configuration to use, the type of Debugger, and the current working directory to use. You can also choose to have your product run at a higher resolution. This enables you to simulate your application running at different display resolutions.

What just happened?

In this section, we looked at how we are able to define and manage schemes in Xcode 4 using the Scheme Editor. We also looked at the ways in which we can use the Scheme Editor to define a separate scheme for Debug, Release, and Distribution.

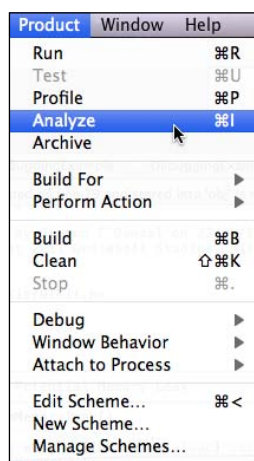
Viewing the Static Analysis results

There may be times when you want to examine the syntax of your code for bugs. This is where the Static Analyzer comes in. It was first introduced in Xcode 3, which opened and displayed the build results within a new window. Xcode 4 lets you perform the analysis, examine the results, and apply the fixes to your source files all within the Xcode 4 workspace.

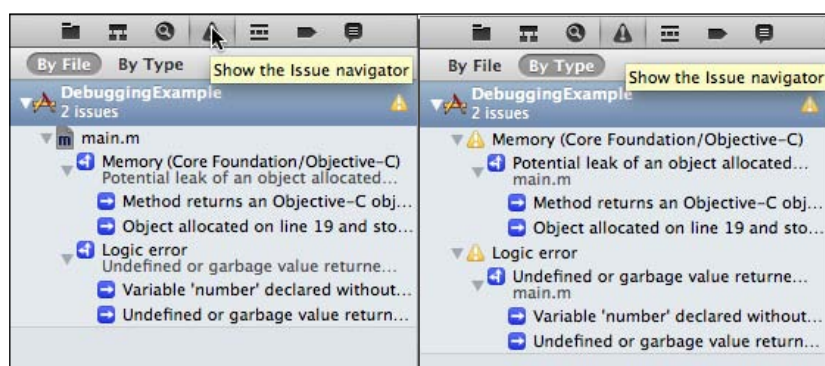
Time for action – running the Static Analyzer

To run the Static Analyzer, follow these simple steps:

1. Select your project from the project Navigator.
2. From the **Product** menu, select **Analyze** or alternatively, hold down the *Command + I* key combinations:



When the analyzer finishes checking your code for problems, the issues navigator opens automatically showing you a list of issues that were found with your project as shown in the following screenshot:



Clicking on an issue within this view will open the file in question, and display the problem which has been marked with a blue triangle. Clicking on this triangle will display the faulty flow of logic that has been identified and detected by the analyzer.

What just happened?

In this section, we looked at how we can use the Static Analyzer to validate our program to ensure that it is free from problems and how we are able to use the Issues Navigator to jump directly into the code file at the location where the faulty code that was causing the issue was found.

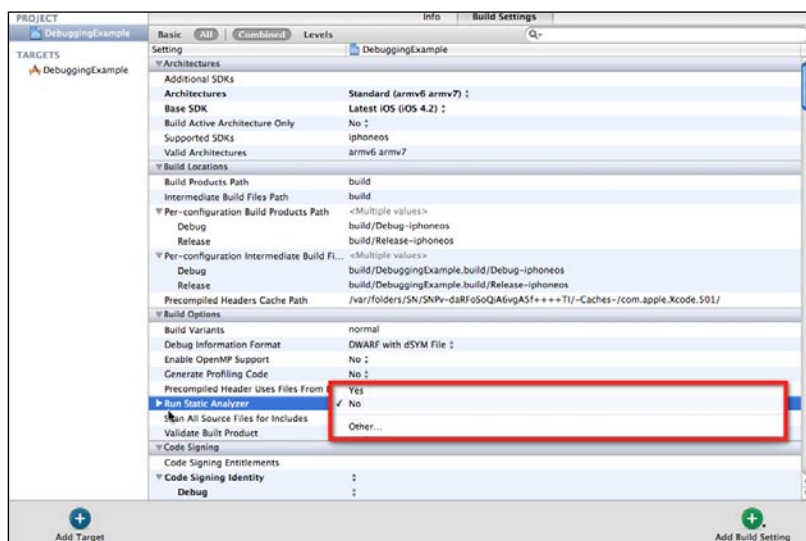
Time for action – configuring your project to perform automatic Static Analysis

Xcode offers you an option to automatically analyze your code whenever your project is built. In this section, we will look at how we can configure our project by following these simple steps.

To set up your project to use the Static Analyzer, you will need to adjust the settings for your project workspace as shown below:

1. In the project navigator, select and highlight your project.
2. Select and click on your target.
3. Select the **Build Settings** tab.
4. Scroll down to the **Run Static Analyzer**.
5. Select the **Yes** option from the list. The default option is **No**.

Now that you have changed the default compiler to use the Static Analyzer, your project will start to use this when you compile and build your application:



You have successfully updated the build configuration for your project. Now when you build your application, it will also run the **Static Analyzer**.

What just happened?

In this section, we looked at how we are able to configure a project to perform automatic static analysis when the project is built, by following simple steps to modify the configuration of the project build.

Time for action – Detecting a memory leak

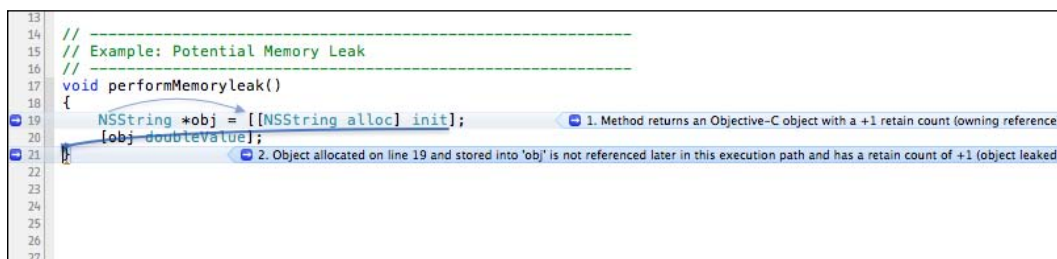
Memory leaks are a very serious type of bug that the static analyzer can help you discover. As you can see in the following screenshot, the analyzer has detected a potential memory leak for the instance of `NSString`.

If your project does not release objects which have been declared from memory, you are opening the doors to some serious issues which can affect your application, and make it run out of memory very quickly which may result in your application crashing. We will take a look at an example and add some code to our project to show you how memory leaks occur:

1. Firstly, open the file `main.m` located within the `Supporting Files` folder of your project, and add the following code as shown in the snippet below:

```
// -----  
// Example: Potential Memory Leak  
// -----  
void performMemoryleak()  
{  
    NSString *obj = [[NSString alloc] init];  
    [obj doubleValue];  
  
    // Add this to prevent memory leaks.  
    [obj release];  
}
```


You will notice that when the Static Analyzer ran through our project, it picked up some serious issues with the code which resulted in it flagging that a potential memory leak is being performed. The blue arrows in the screenshot below show the program flow of the object that is being allocated memory and it has been determined that the object has not been freed:



What just happened?

In the above code example, we declared a variable object **obj** of type `NSString` and allocated memory to this object. We then called the `doubleValue` method on this object, before finally exiting the function. Since we are not using garbage collection to release this object from the stack, it will result in a serious case of memory leakage.

Time for action – detecting an instance of an uninitialized variable

Another type of common error made by developers is variables which have not been initialized upon being declared. Fortunately, the Static Analyzer can also catch these types of bugs.

To demonstrate how the Static Analyzer helps detect instances of uninitialized variables, we will use the following code:

1. Open the file `main.m` located within the `Supporting Files` folder of your project.
2. Enter the following piece of code as shown in the snippet below:

```
// -----  
// Example: Uninitialized Variable being declared  
// -----  
int setReturnValue(int _varX)  
{  
    int number;  
  
    if (_varX > 100)
```

```

    {
        number = _varX * 2;
    }
    else if (_varX == 100)
    {
        number = _varX - 50;
    }
    return number;
}

```

In the following screenshot, you will notice that the static analyzer has flagged the variable `number` as a potential error. This is due to the fact that the variable was uninitialized and is returned with some random value. This is due to the fact that the `number` was not initialized upon declaration.

If the **if-else** clause fails due to the value of `_varX` resulting in a negative value, the variable `number` will remain unassigned, resulting in your application producing unexpected random results.

You will also notice that the analyzer provides additional detail when you click on the message bubbles and displays the control-flow (as shown by the blue arrows), and a set of **events** that give a full diagnosis of the bug. Many of the issues that are reported by the Static Analyzer tool have this information, and this makes analysis and fixing of these errors much easier:



What just happened?

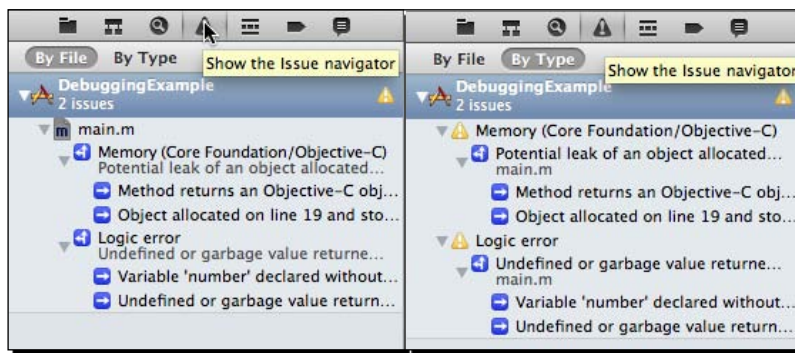
In the above section, we looked at how we can use the power of the Static Analyzer to detect variables which have not yet been initialized within our application. We also looked at how the static analyzer uses control-flows (highlighted by the blue arrows) to show the execution path of your code. This helps prevent unexpected results from happening within code that should execute, but doesn't.

Viewing the Issues Navigator

During the building of your application, the compiler checks your code to ensure that it is syntactically correct and that no errors exist. However, if the compiler finds any problems during the building of your project, the issues navigator window opens and displays all potential errors and warnings that were found.

The issues navigator can be changed to show problems broken down By File or By Type. If you select any of the errors or warning messages that are displayed within the list, they will automatically display within the source code editor at the line where the error was discovered.

The following screenshot displays potential errors that were found within the project. This list can be broken down to display by File or By File Type:

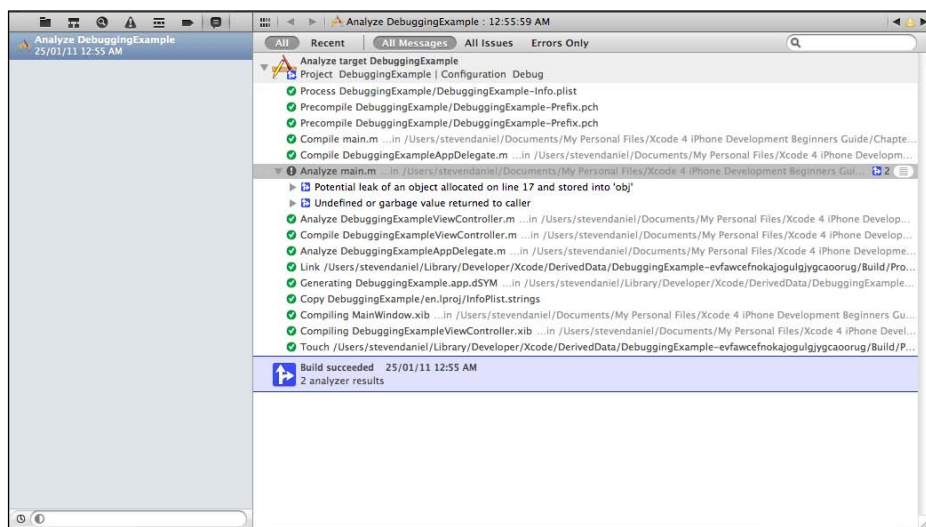


Viewing the Program Build log

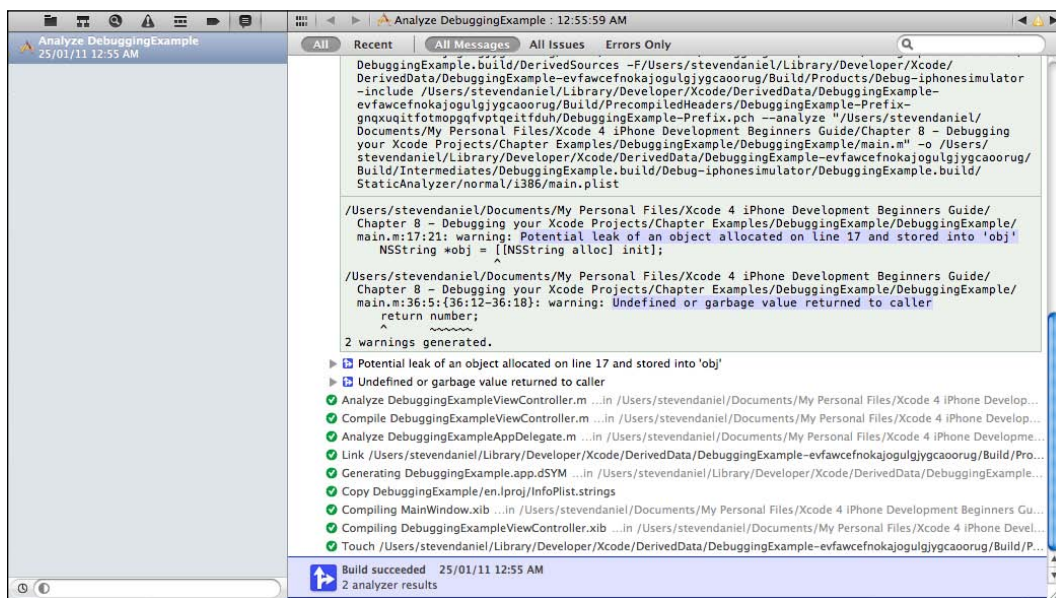
The **Build log** contains a history of all project builds that are contained within your project workspace. This list contains all warnings and error messages that were found during each stage of the project build process, and contains the date and timestamp. When you select one of the project builds in the build log, the results are displayed within the editor window area.

Double-clicking on a warning or error message will open the source file and jump to the line containing the error. The build log window can display the results in a compact/normal view (default) as shown in the screenshot below, or in a more verbose type (full build) form, as shown in the second screenshot.

The screenshot shown below displays the Build Log in Compact View. This view is more of a condensed view, which unlike the verbose mode view, shows minimal information:



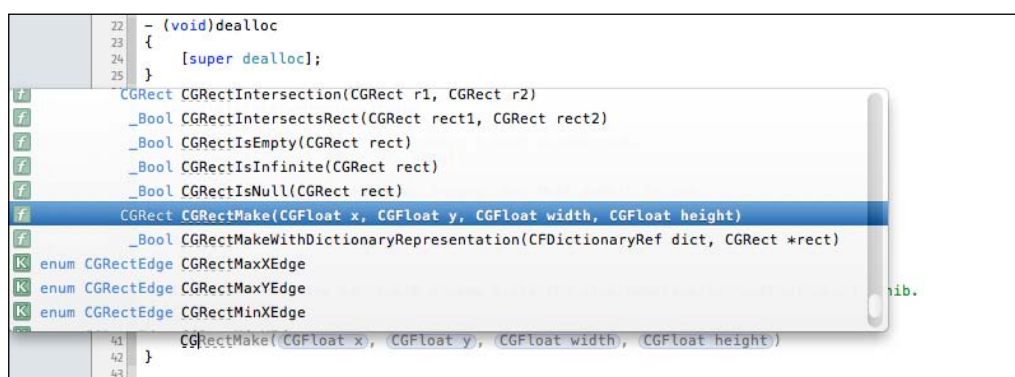
The screenshot below displays the Build Log in the verbose view, which provides us with a bit more information about the types of errors we are receiving. It provides the module name, line number(s), and the piece of code which is potentially causing the problem:



Understanding and using code completion

Code completion makes your development experience a lot easier by having Xcode provide you with suggestions when you have entered enough letters for Xcode to make a reasonable guess as to what you are intending to type and it will then display the suggestion as dimmed text.

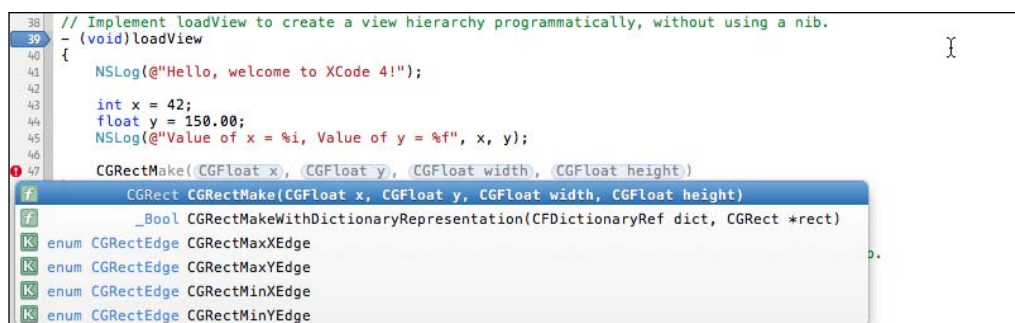
The following screenshot displays a list of inline suggestions for completing the symbol name that is being entered, as well as a list of all its possibilities. If there is no common prefix, code completion shows the dotted underline up to the next uppercase letter in the symbol:



Time for action – working with code completion

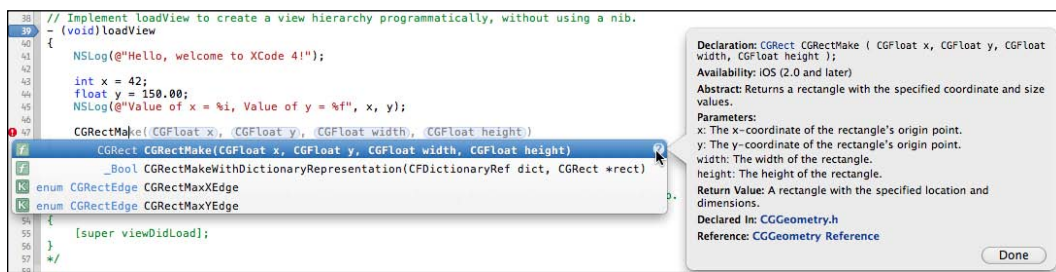
To accept the auto completion suggestion, follow these simple steps:

1. Press the *Tab* key and the code will be inserted just as if you had typed in the whole thing.
2. Press *Return* to accept the entire auto completion suggestion. You will notice from the screenshot below that Xcode is smart enough to work out what you intended to type in. As soon as you get to a point in each line, it will display an autocompleted version of the code:



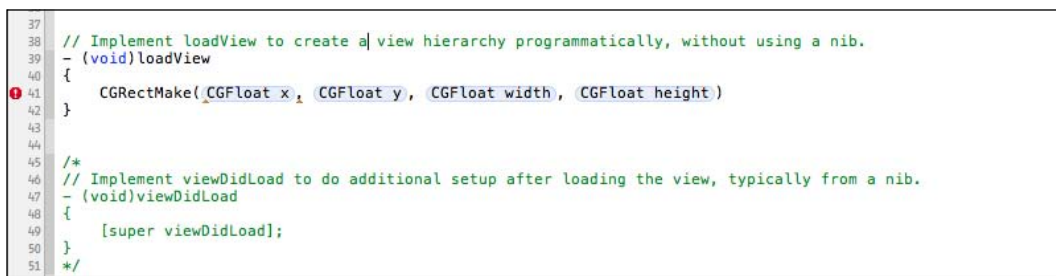
A Quick Help popup is also available from code completion, even when the Quick Help inspector is not open. Hover over the code completion option you're interested in until a question mark icon appears:

3. Click the **question mark** or use the *Command + Shift + Control + ?* keyboard short-cut to display the Quick-Help for the method.
4. When you have finished, click on the **Done** button in the Quick Help popup to cancel the operation:



5. Press the *Tab* key to accept only the sub-word or *Return* to accept the entire suggestion. Xcode will even try to complete method names, variables that you have declared, as well as anything else related to the project that it might recognize.

If you look at the example where we type in the `CGRectMake` method, Xcode displays an auto completed version of the code as shown in the screenshot below:



6. Press *Control + Space bar* to toggle the completion suggestion on or off. That is, if the inline suggestion and list are being displayed, pressing *Control + Space bar* cancels the code completion operation. If there is no suggestion displayed, place the cursor at the end of a partially typed symbol and press *Control + Space bar* to get completion suggestions.



In Xcode 4, pressing the *Esc* key cancels the operation and pressing the *Delete* key always deletes the preceding character.

What just happened?

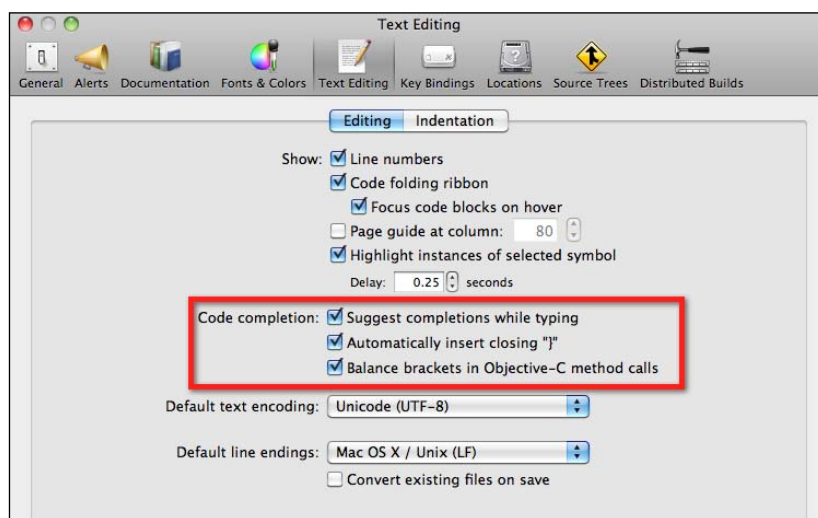
In this section, we looked at how we can use the code completion features of the Xcode 4 IDE to enable developers to be more productive. The editor is intelligent enough to somehow work out what the developer is trying to do and therefore provides a list of suggestions for the developer to choose from.

Time for action – stopping Xcode from alerting you to problems

If for some reason you would like to stop Xcode from alerting you and correcting your code for you, you can turn off this feature by following these simple steps:

1. Click on the **Xcode** menu item or alternatively, press the *Command + ,* key.
2. Next, select the **Preferences** menu option.
3. Then click on the **Text Editing** Page.
4. Locate the **Code Completion** section.
5. Check or Uncheck the **Suggest Completions while typing** option.

The screenshot below shows the Xcode Preferences page and the Code Completion section highlighted by a red rectangle to show how to turn this feature on/off:



As you can see from the above screenshot, you can also stop Xcode from automatically adding the closing brace. Various other options are also made available to hide or show line numbers and code folding.

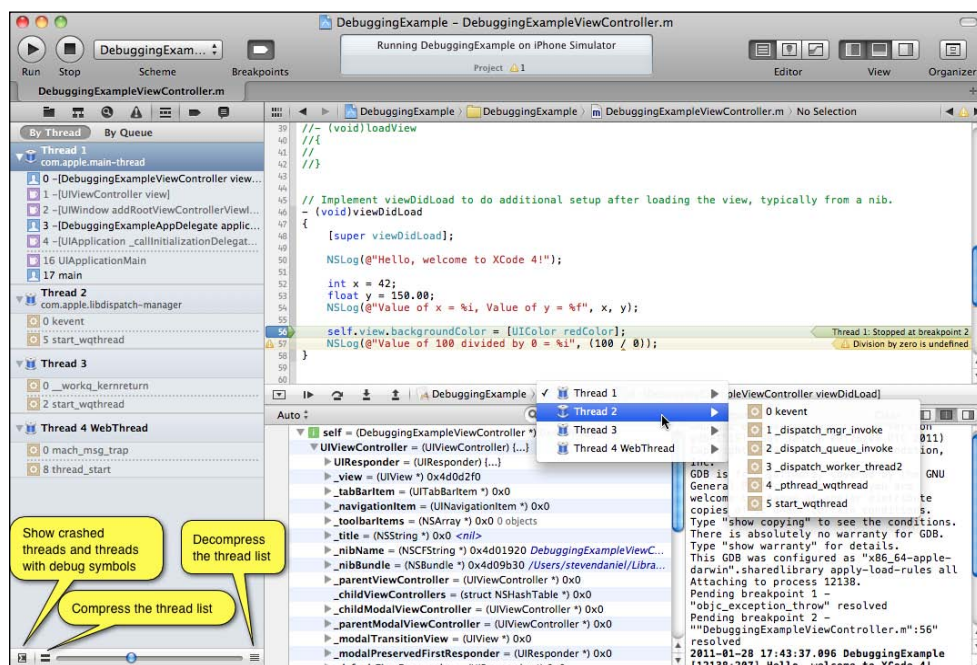
What just happened?

In this section, we looked at how we can configure the Xcode preferences to turn features on and off and how we would like the IDE to alert the developer to problems. We also looked at how we can turn code completion on and off, as well as looking at some of the other options made available on this tab, for instance, displaying line numbers within the code editor and code folding.

Navigating through threads and stacks in the Debugger

Whenever you pause execution of your code or the running code stops at a breakpoint located within your code, Xcode opens the debug navigator window, and displays the threads that were running when program execution halted.

Under each of the threads is the program stack at that point in the program execution. Select a stack frame to see the corresponding source file in the source editor or disassembled object code. At the bottom of the debug navigator window is a slider control. This controls how much stack information the Threads and Stacks navigator is to display within the window:



Located at the left end of the slider control is a feature to display only the top frame threads of each stack. The option at the right end of the control bar shows all stack frames. The button towards the last left of the slider control displays all active threads or only those threads that have your code in them, as opposed to only including the system library code.

Have a go hero – Static Analyzer and debugging features

Now that you have a good working knowledge of the debugging features within Xcode, your task will be to apply some breakpoints and use the static analyzer on your `GetUsersAttention` example that we created in *Chapter 6, Displaying notification messages*:

1. Open Xcode 4 and load the **GetUsersAttention** example program.
2. Modify the scheme to ensure that the option to run the Static Analyzer has been set when the application is run. You can refer to the section *Configuring your project to perform automatic Static Analysis* located in this chapter.
3. Next, place a breakpoint in the function that handles the displaying of when the activity indicator button is pressed. You can refer to the section *Debugging with Breakpoints* located in this chapter.
4. Run the application and analyse the results of any errors or warnings reported by the static analyser. You can refer to the section *Viewing the Issues Navigator* located in this chapter.
5. Next, click on the button to display the activity indicator. When your application stops at the breakpoint, use the *step over* and *step into* buttons. You can refer to the section *Debugging features in the Code Editor* located in this chapter.
6. Familiarise yourself with the Stack Trace Panel, Disassembly View, and Variables Pane to see what types of information are being displayed. You can refer to the section *Introducing the New and Improved Debugger* located in this chapter.
7. Once you have familiarised yourself with the information that has been displayed, click on the Resume button located within the debugging toolbar to resume execution of the application. You can refer to the section *Debugging features in the Code Editor* located in this chapter.
8. Close down the iOS simulator and return back to the Xcode IDE.
9. Remove all breakpoints that you have added to the code. You can refer to the section *Debugging with Breakpoints* located in this chapter.

Once you have followed the above steps correctly, you would have successfully set up your project to use the Static Analyzer, setting breakpoints within your code as well as using the debugging features within the Xcode IDE.

Pop quiz – all about debugging projects

1. What option within the Variables Pane allows you to display a list of all recently accessed variables?
 - a. Local
 - b. Auto
 - c. All
2. What option in the Console Pane allows you to show information relating to the Debugger?
 - a. All Output
 - b. Target Information
 - c. Target Output
3. What is the default view of the Variables and Console Pane?
 - a. Auto
 - b. Local
 - c. All
 - d. All Output
4. What is the purpose of the Issues Navigator?
 - a. Displays all flow issues
 - b. Displays a list of all issues found within your project
 - c. All of the above
5. What are the two ways in which you can run the Static Analyzer?
 - a. Select **Analyze** from the **Product** Menu
 - b. Press *Command + A*
 - c. Hold down the *Command + I* key

Summary

In this chapter, we learned how we go about debugging our projects through the use of the various debugging tools that Xcode provides us. We looked at how we can use Fix-it to correct code as we type, Static Analysis which showed us potential coding errors, that is, Memory leaks, Dead code, or unreachable code, as well as using the Debugger to navigate through threads and stacks within our project. You will see that by using these two great companion tools, Fix-it and the static analyzer, you will ensure that your code is free from bugs long before your users find them.

We have learned about the various debugging features that Xcode offers. We looked at how we can go about detecting and preventing our application from giving unexpected errors. We are now ready to move on to and learn about Source Code Management (SCM) with the Version Editor.

In the next chapter, we will learn about the Source Code Management (SCM) features of Xcode, by creating, configuring, and adding items to new and existing code repositories. We will also talk about the Version Editor and how we are able to compare different versions of a file side-by-side, all within the Xcode IDE.

We will also learn about the Track Blame feature of the version editor to check past check-ins of files to see which person made the last revision, and finally we will learn how to use Git and Subversion to manage multiple projects.

9

Source Code Management with the Version Editor

In this chapter, we will focus on the new features of the Xcode Version Editor that has been integrated directly within the Xcode 4 IDE and provides you with an easy way to manage your source code. By using this tool, it allows you to travel back through your revisions to compare previous changes made throughout the life cycle of the file.

In this chapter, we will be covering the following topics:

- ◆ Introducing the new Xcode Version Editor
- ◆ Introduction to Subversion
- ◆ Learning how to create and configure Repositories in Xcode
- ◆ Learning how to add items into an existing Repository
- ◆ Learning how to compare different versions of a file using the Version Editor
- ◆ Learning how to use the Track Blame feature to check on past source code check-ins
- ◆ Learning how to use both Subversion and Git to manage multiple projects

We have got quite a bit to cover, so let's get started.

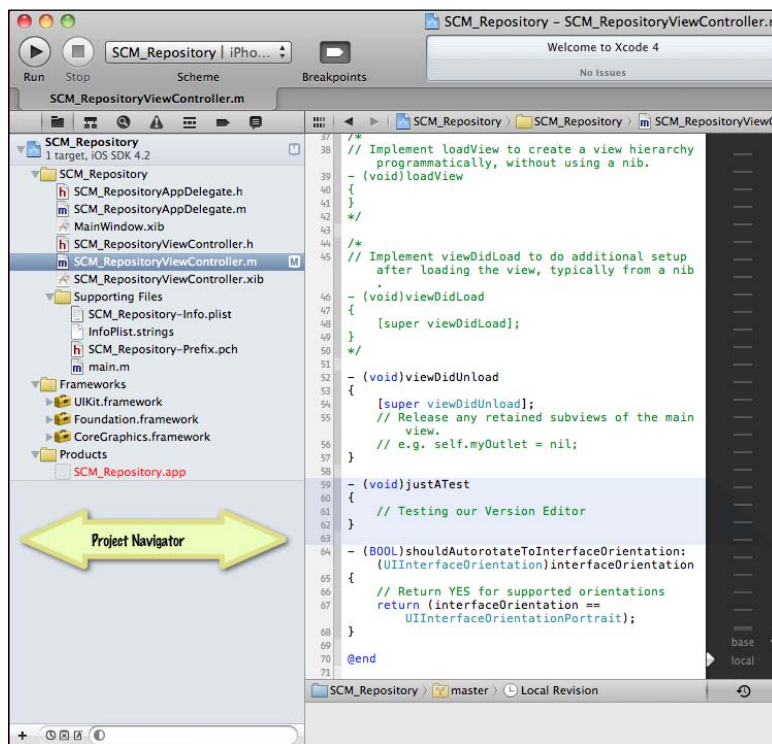
Introducing the new Version Editor

The new Version editor in Xcode 4 makes it easy to see any two versions of your source code, side by side, all within the IDE. By having the Version editor integrated directly into the IDE development environment, it sets a different direction in the way we think of source-control management.

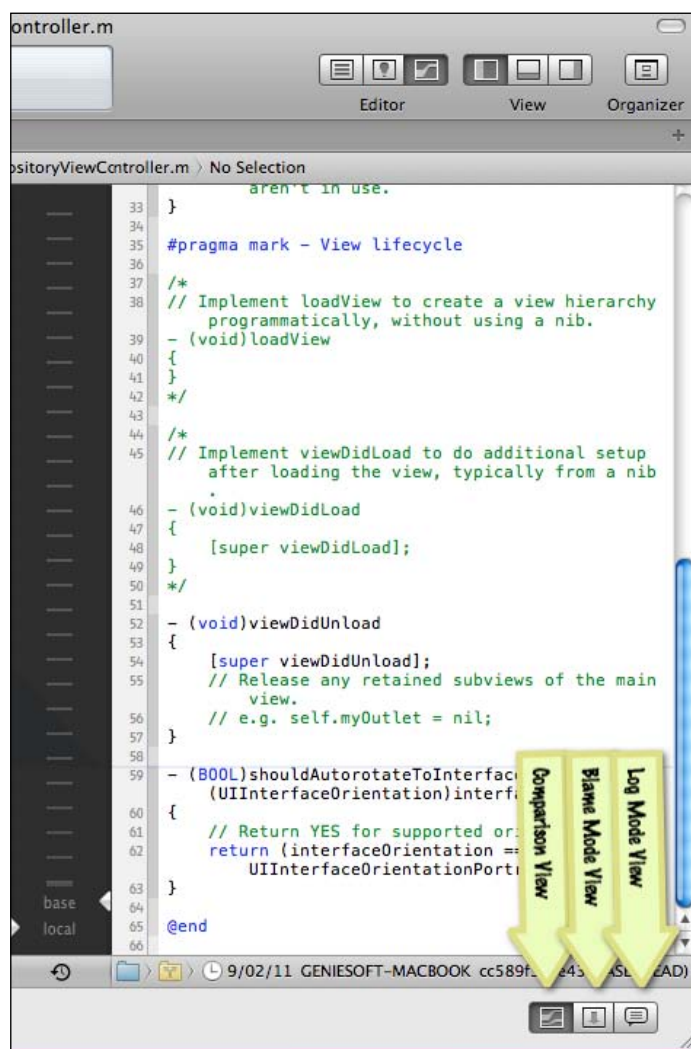
The Version Editor provides you with a comparison view of each of your files within the repository to what is stored locally via a timeline. Dragging the slider in the middle of this timeline enables you to travel back in time through your project, thus comparing any two versions of the same file.

The Version editor can also show you a detailed log of past events, and track blame for past check-ins. You don't need to worry about anything else as all of the complex source code management (**SCM**) commands are managed for you behind the scenes. It is even possible to manage multiple projects within a single Xcode 4 workspace, one project managed in Subversion, and the other in Git, with all being updated for you automatically.

In the screenshot on the following page, are displayed the contents of a local revision of the file showing the piece of code that has been added, but has not yet been checked into Source-Control:



The next section of the image shown in the following screenshot shows the version of the file that is within source-control being compared with a local version of the file. It also displays the various options available to you when in this mode; that is, Comparison view, Blame mode, and Log mode:



The Project Navigator contains badges, which are for SCM and pertain to Subversion and repositories. If you need to refresh your memory on what these status codes mean, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section *Listing files in a project*.

Introducing Subversion

You may be asking yourself? What is Subversion? Subversion is a version control system, that handles keeping track of the changes made to a file and who made the changes. While the use of version control is not limited to source code files (I could use version control to keep track of the changes I make to the articles and book chapters I write), the primary users of version control are software developers.

Version control is especially helpful on large projects with multiple developers. Each developer can add code to a file, and the version control system records the code each developer added along with their name. Even if you're working on a project by yourself, version control can help you. If you've ever mistakenly saved a file and wished you could go back to the way the file was before you saved it, you'll appreciate version control. With version control, you can go back to an older version of a file.

You may have worked with and used other version control software such as Microsoft Visual Source-Safe, Tortoise SVN, or Concurrent Versions System (CVS), which is a popular version control system.

Apart from the ones mentioned above, the use of Subversion has several other advantages too:

- ◆ Subversion can handle the renaming of files.
- ◆ Subversion can track the changes made to directories.
- ◆ Subversion handles atomic commits. Atomic commits allow you to commit changes in multiple files while making sure all the changes get committed.

Installing a local Subversion server

In this section, we are going to look at how we go about installing the server components onto the same computer, as you will be using for development. In the real world, this will not be the case as you will be connecting to another computer (so as to avoid an unexpected system crash), which will be periodically backed up). But for the purposes of learning, this is the easiest way.

Firstly, we will check the current version of Subversion that you currently have on your Mac computer. This is to confirm that you are using the latest version of the software. Open a Terminal window session and type `svn --version`:

```
$ svn --version
```

You should eventually see the following text appear on your screen:

```
svn, version 1.6.5 (r38866)
   compiled Jun 24 2010, 17:16:45
```

Copyright (C) 2000-2009 CollabNet.

Subversion is open source software, see <http://subversion.tigris.org/>

This product includes software developed by CollabNet
(<http://www.Collab.Net/>).

The following repository access (RA) modules are available:

- * **ra_neon** : Module for accessing a repository via WebDAV protocol using Neon.
 - handles 'http' scheme
 - handles 'https' scheme
- * **ra_svn** : Module for accessing a repository using the svn network protocol.
 - handles 'svn' scheme
- * **ra_local** : Module for accessing a repository on local disk.
 - handles 'file' scheme

\$

In the unlikely event that you don't happen to see the above displayed on your screen, you will have to download and install the Subversion software onto your computer. You can obtain the binary distributions from the following location <http://subversion.apache.org/packages.html> and then choose the Mac OS X option from the top section of the main page. From the list of available options, select openCollabNet, make your selection between **Universal Subversion 1.6.16 Binaries for Leopard (Mac OS X 10.5)** or **Universal Subversion 1.6.16 Binaries for Snow Leopard (Mac OS X 10.6)** depending on your system. It is a requirement that before downloading either version, you will need to sign-up and register your details. This process is completely free.

Creating a repository

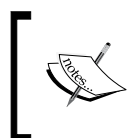
Once you have installed the Subversion software, you will be able to create your repository. We will be creating a single repository for this chapter and then look at how we can use Xcode and Subversion and Git to manage multiple projects.

Before we start to look at how we create a repository, it is worthwhile to mention the types of repositories that Subversion supports. Subversion enables you to create two types of repositories: Local and Remote. These are explained in the table below:

REPOSITORY TYPES	DESCRIPTION
Local Repositories	These types of repositories reside on your computer and will only be used by you.
Remote repositories	These allow other people to access the repository and check out the files in the repository from their computers.

Subversion is a complex, powerful, and sophisticated product that you can configure and use in many ways. We will be covering the essential areas of Subversion to get you started. If you want to learn more about Subversion, I would thoroughly recommend reading the free online book *Version Control with Subversion* that can be accessed at <http://svnbook.red-bean.com/>.

Even though Subversion is included with the installation of Xcode 4, it is possible to install a local version of Subversion manually. In the next section, we will look at how this can be done.



In this chapter, we will be covering local repositories. Apple provides documentation on their developer site that shows you how to set up remote Subversion repositories at the following location <http://developer.apple.com/>.

Time for action – setting up a local Subversion repository

In this section, we will be looking at how we can use the command line to set up a Subversion repository. We will now start to create a repository, that will be used to house our project revisions:

1. In your documents folder, create two folders: **Master_Projects** and **Working_Copy** by following the commands below:

```
cd ~/Documents  
mkdir -p Repositories/Master_Projects  
mkdir -p Repositories/Working_Copy
```
2. Next, we will use one of the examples from a previous chapter. Locate the project `TapExample` from *Chapter 7, Exploring the MultiTouch Interface* and copy it to the **Master_Projects** folder.
3. Open the Terminal utility application using *Shift + Command + U*. At the command prompt, type in the following command to create a subversion repository:

```
svnadmin create ~/Documents/Repositories/svn
```

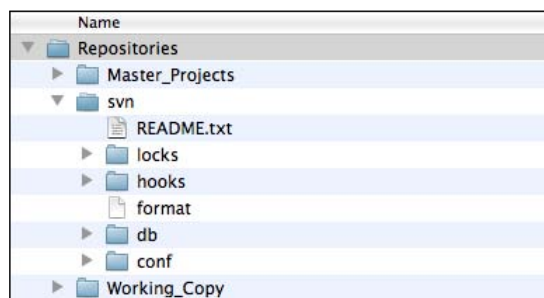
4. Our next step is to use the `verify` command to ensure that our command which we executed above has successfully created our Subversion repository to house our projects. At the command prompt, enter in the following command:

```
svnadmin verify ~/Documents/Repositories/svn  
* Verified revision 0.
```

Notice that when you run the above command, it displays **revision 0**. This is because we have not created any revisions or imported any content.



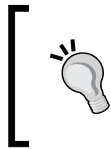
When creating Subversion repositories, it is common to create three top-level folders, called branches, tags, and trunk, below the repository name. The **trunk** folder holds the main folder structure, while the branches and tags hold the information that is contained within your revision history that you assign to specific folders within a revision by giving them friendly easy to identify names.



What just happened?

In this section, we looked at how to manually create the Subversion repositories using the command line tool `svnadmin`. We then verified to ensure that our repository got created. In the next section, we will look at how we can use Xcode to configure our subversion repository.

This folder structure, that we just created, contains the configuration files that are required for the repository and the database files that are used by the Subversion server to manage and handle the project revisions.



Never make changes directly into this folder. This could corrupt the database that holds all of your code history and there is no way to get this back. It is best to leave this up to Subversion and Xcode to automatically handle for you.

Configuring the repository in Xcode

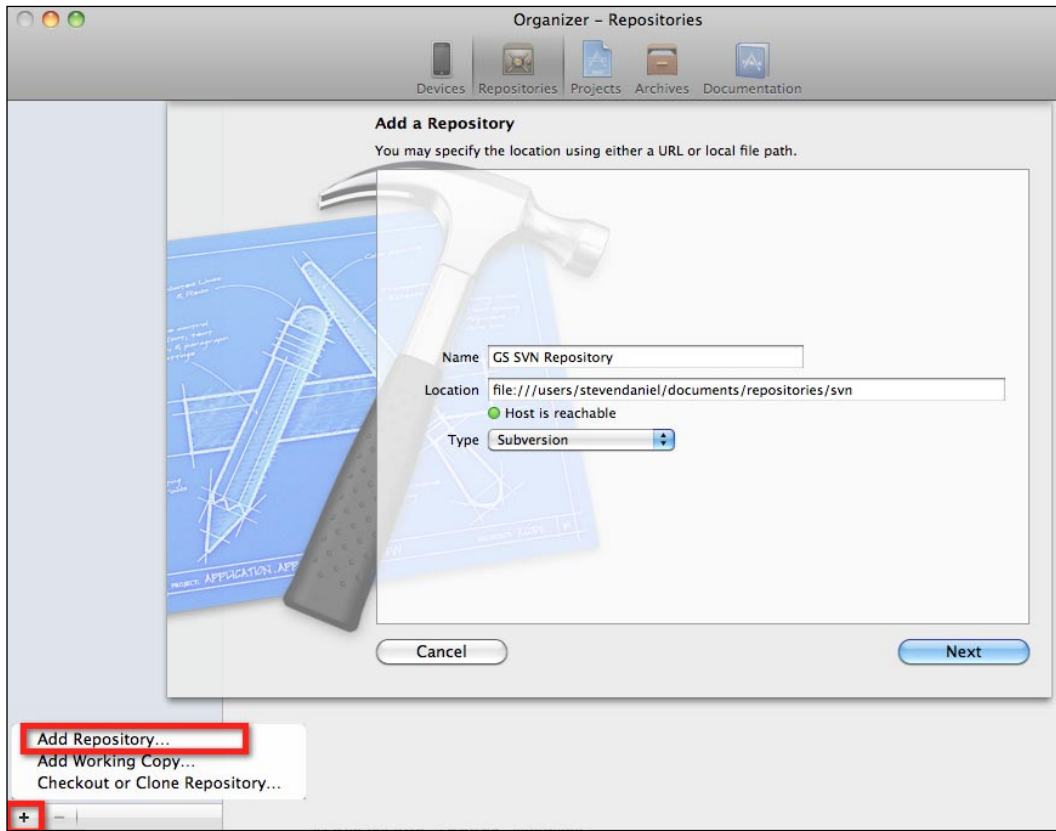
In this section, we will look at the necessary steps into what is involved with setting up and configuring a source code repository using Subversion within Xcode 4. We will be creating the necessary branches for each section to ensure that it has been configured correctly.

Time for action – configuring the Subversion repository

Now that you have successfully created a Subversion repository, it is time for us to add our project to source code management:

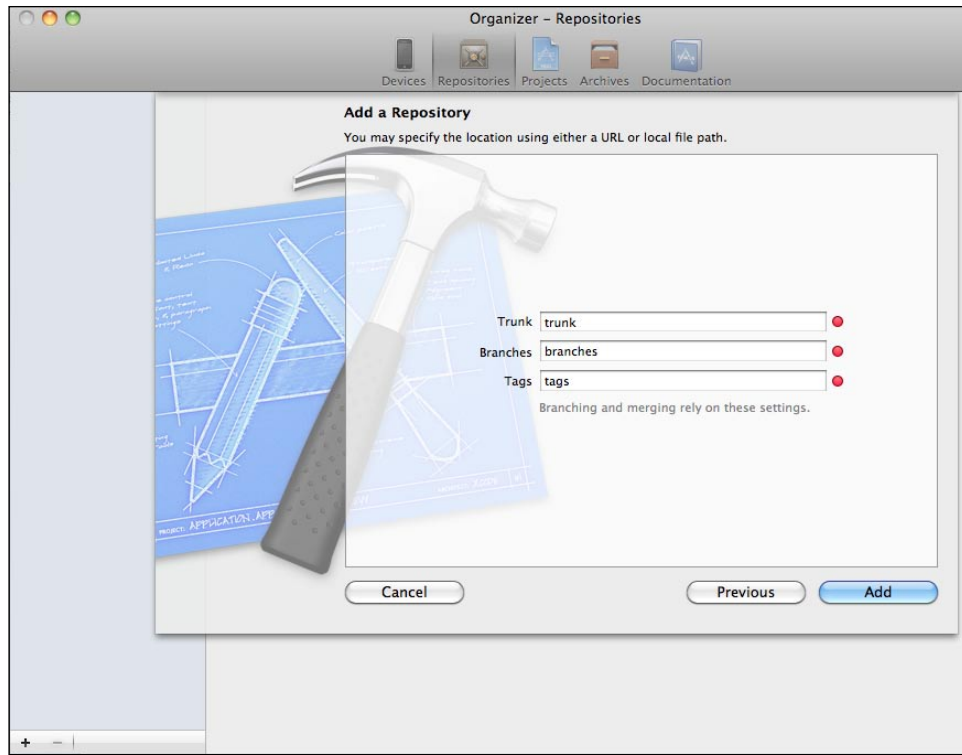
1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Select Organizer from the Window menu or press *Shift + Command + 2*.
3. Click on the **Add (+)** button and then select **Add Repository...** to create a new repository. This will bring up the Add a Repository dialog window from where you specify the name of the repository configuration that you would like to create.
4. Enter in a suitable name for the repository. I have chosen **GS SVN Repository**.
5. In the Location box, type the location of the repository, which we created previously. This will need to be in URI format, which is the folder location. Enter in: `file:///users/stevendaniel/documents/repositories/svn`.
6. Make sure that you select **Subversion** as the type of repository to set up.

7. Click **Next** once you have filled in all of the fields:



One thing that you will notice when filling in the location field, is that a green light is displayed. What Xcode is doing is testing the configuration to ensure that it can connect properly to the repository and confirms that you have a valid connection.

Our next step is to add the branches, tags, and trunk folders. However, since we are creating a Subversion repository, this is not a mandatory step to add a project to a Subversion repository:



I would personally recommend creating these folders, as this will make your life easier later on should you decide to branch out your code. An example of this could be, say, that you currently have a Mac OS X application, and then later on you decide that you want to create an iPhone/iPad application.

You could in theory use the same Subversion repository and create three separate branches: one for Mac OS X, iPhone, and finally one for iPad. Creating these folders now for branches, tags, and trunk will not do any harm.

What just happened?

In this section we looked at how we can use Xcode to configure our subversion repository that we created in the previous section. We provided the Name, Location, and Type of repository to configure. We also specified the folder names to create for the Trunk, Branches, and Tags. In the next section, we will look at how we go about adding an existing project into our repository using Xcode 4.

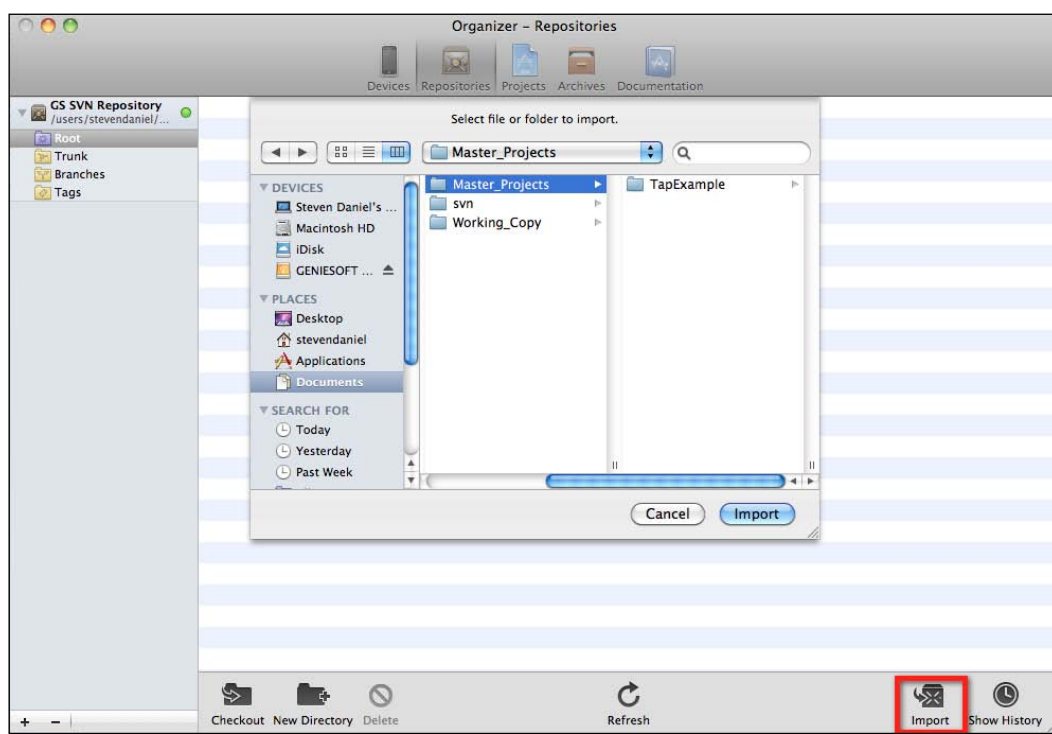
Adding items to an existing repository

In this section, we will look at the necessary steps involved in how we go about adding an existing project and its accompanying files into source control using Subversion and Xcode 4.

Time for action – adding our TapExample project to the repository

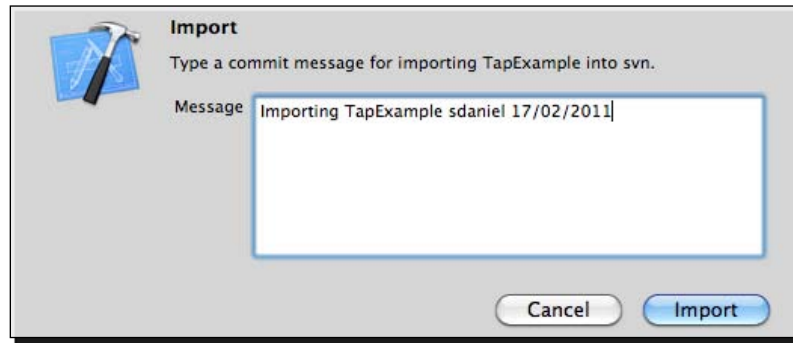
Now that we have configured and set up our SCM repository, Xcode is aware of this, and we can start to proceed with importing our **TapExample** example project:

1. Click on the **Import** button as shown in the screenshot below, and use the file browser to navigate to the **Master_Projects** folder and select the **TapExample** example project:

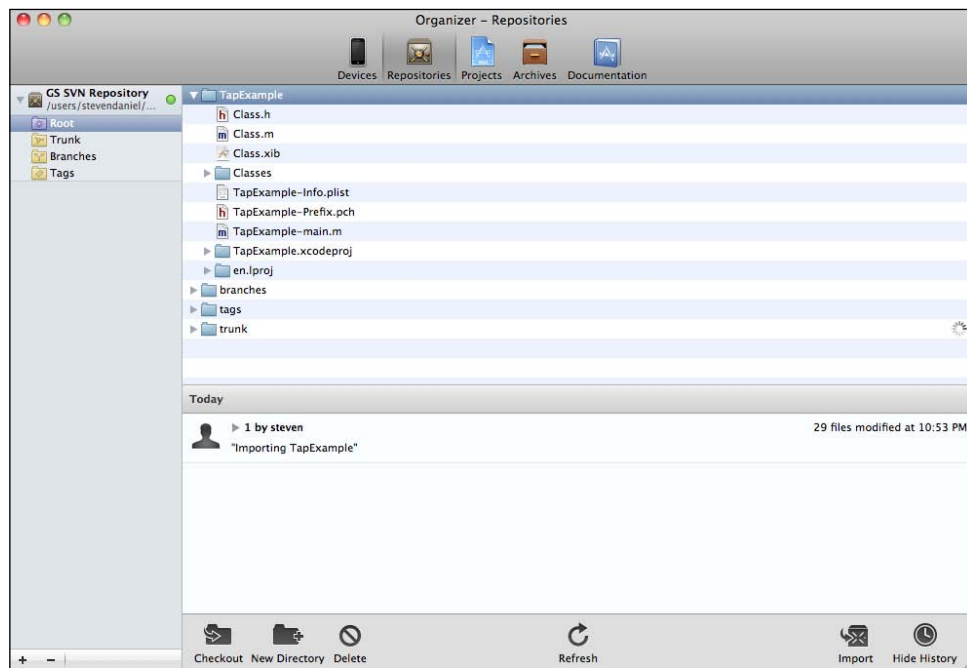


2. Once you have successfully navigated to the **Master_Projects** folder and selected the **TapExample** project, click on the **Import** button.
3. This will display a popup dialog box where you must submit a descriptive message before proceeding to have your project added to Subversion.

I usually tend to add a fairly descriptive message along with my initials and date that the project was added to the repository:



Once you have clicked on the **Import** button, Xcode will begin to start importing your project. Since this is being imported into a local repository, this process will complete very quickly:



Now that the project has successfully been added into Subversion, it is safe to delete the version located within the `Master_Projects` folder. You will notice that when you import your project into the Subversion Repository, it creates a project revision number along with the name of the user who added the project and the associated comment. This is located within the middle windowpane of the SCM Organizer window.

If you decide to add the branches, tags, and trunk folders to your project folder, it is important that you move the files from within the project folder to the trunk folder.

What just happened?

In this section, we looked at how we are able to use Xcode to add items into a previously configured repository, which we did in the previous section. We imported our **TapExample** project into this repository and provided a comment entry. In the next section, we will look at how we can check-out a working copy of an existing project from the repository.

Getting a working copy of the project out of the repository

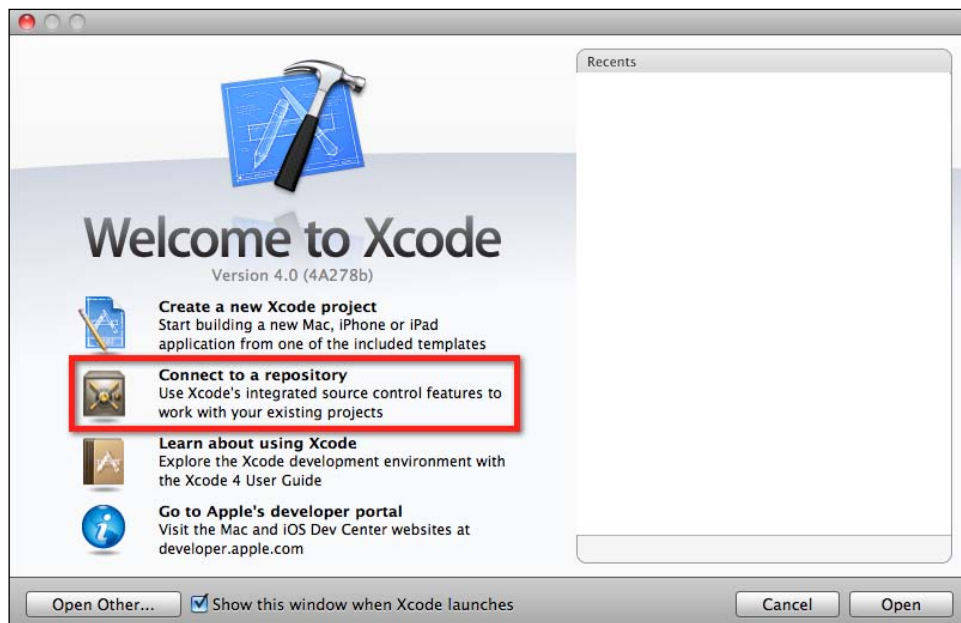
In this section, we will look at how we go about checking out the project from the repository so that we have a version that we can work with and apply the necessary changes, then check these changes back into the repository.

Time for action – checking out the project from the repository

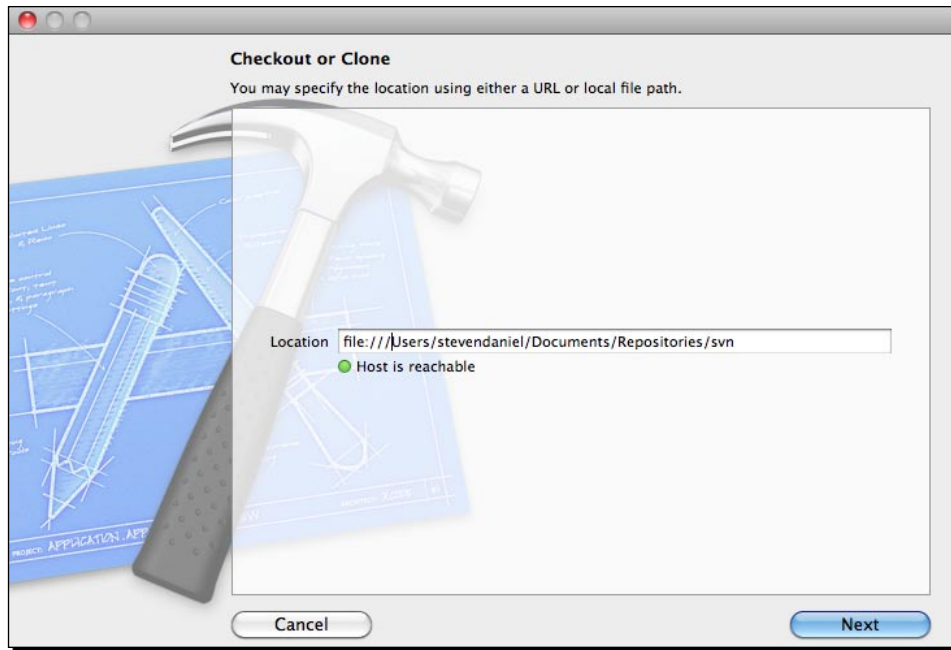
We will look at two different ways in which we can check-out and get a working copy of a project within an Xcode repository.

The first and easiest option is to:

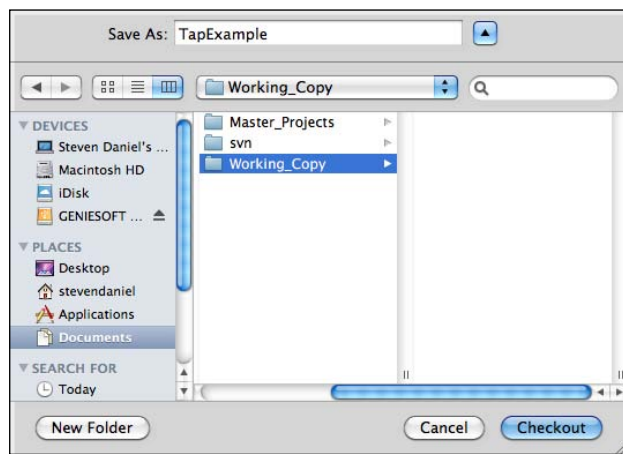
1. Select the **Connect to a repository** option from the **Welcome to Xcode** screen:



2. When this option is selected, it will display the Checkout or Clone dialog where you will need to specify the file location where your repository is located. This is shown in the screenshot below:



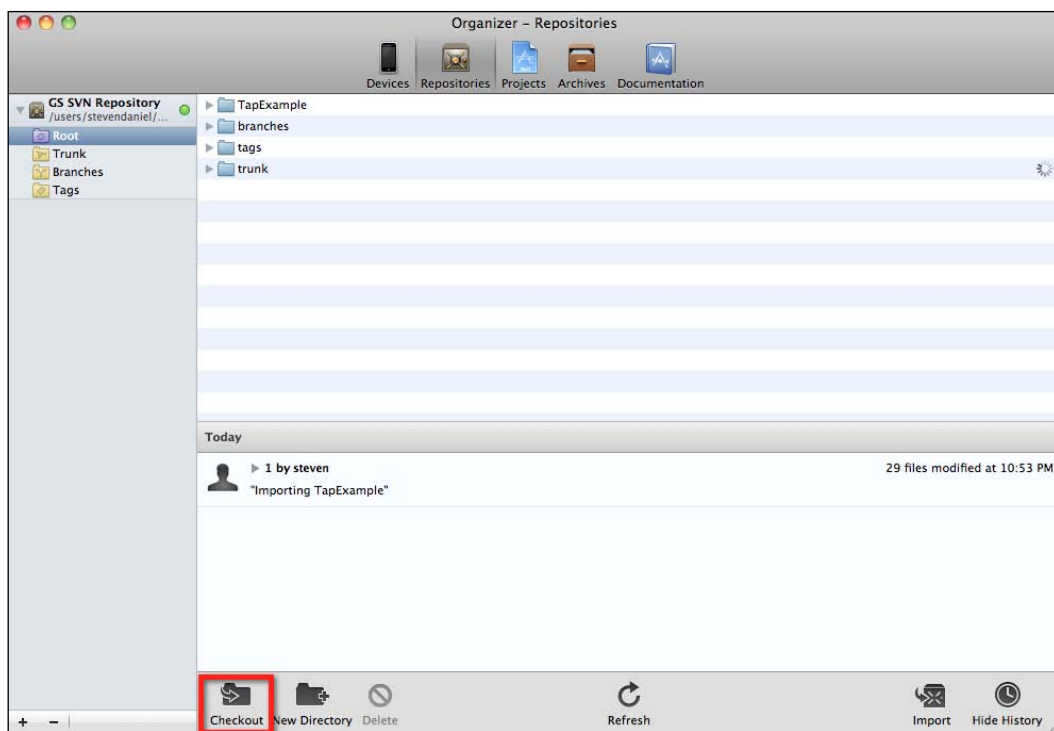
3. Click on the **Next** button to proceed to the next screen, which will display the file browser window and allow you to specify the folder location to where you would like to checkout your project. I have decided to checkout this project to the **Working_Copy** folder location:



4. Once you have selected your folder location, click on the **Checkout** button. This will create a Subversion-managed copy of the `TapExample` project folder and its contents in the `Working_Copy` folder or the folder which you have specified.
5. Once Xcode has successfully completed checking out the project, you will see the message box displayed as shown in the screenshot below. Click on the **Open** button to have the project open within the Xcode environment:



The second option to check-out the project is to click on the **CheckOut** button located within the SCM Repository organizer window, as shown in the following screenshot:



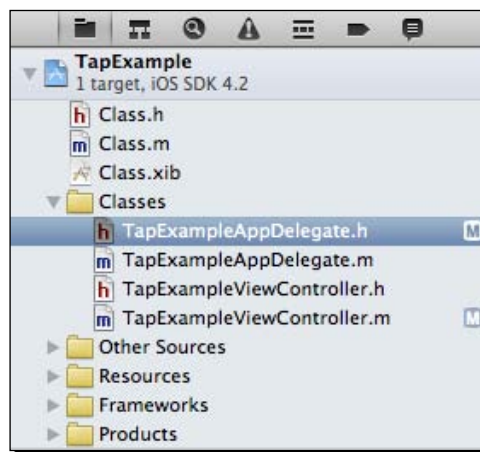
This will display the same dialogs as shown in the previous screenshots and allow you to specify the folder to checkout your project to.

What just happened?

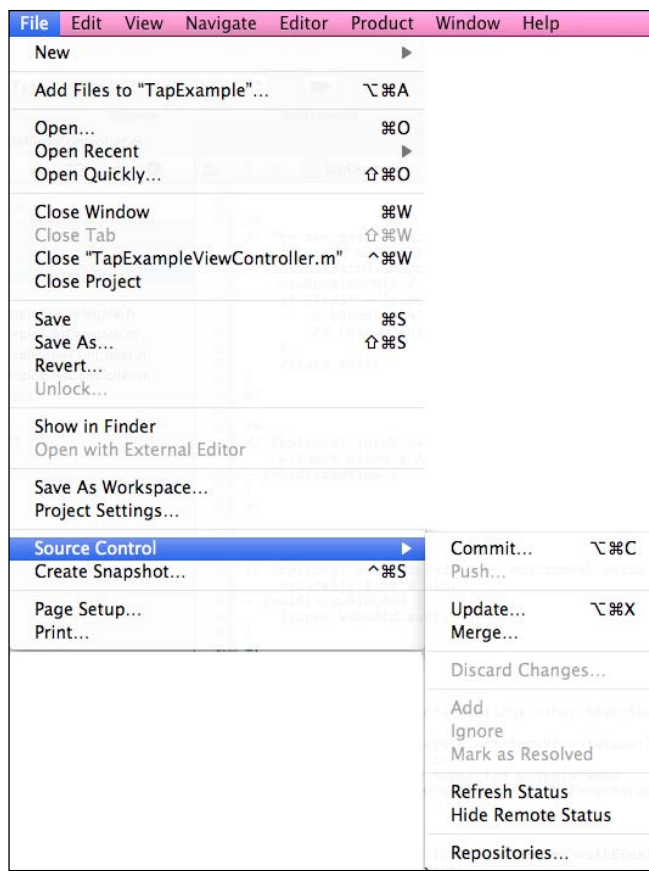
In this section, we looked at how we are able to use Xcode to connect to an existing repository to check out the master copy of our project and have it save to another folder location on disk, prior to having it opening directly within the Xcode IDE. In the next section, we will look at how we are able to use the Source Control menu within the Xcode 4 IDE, to commit, merge, or update our changes.

Xcode source-control features and file statuses

The screenshot below displays the file statuses, which have been modified, added, or deleted from a Subversion repository. These badges propagate up to the highest container so you can see the source control status of the whole project workspace:



The Source Control Management (SCM) menu shown below is where you can Add, Update, Merge, and Commit your changes made to your project into your repository. In the section below, each of these main options are explained:



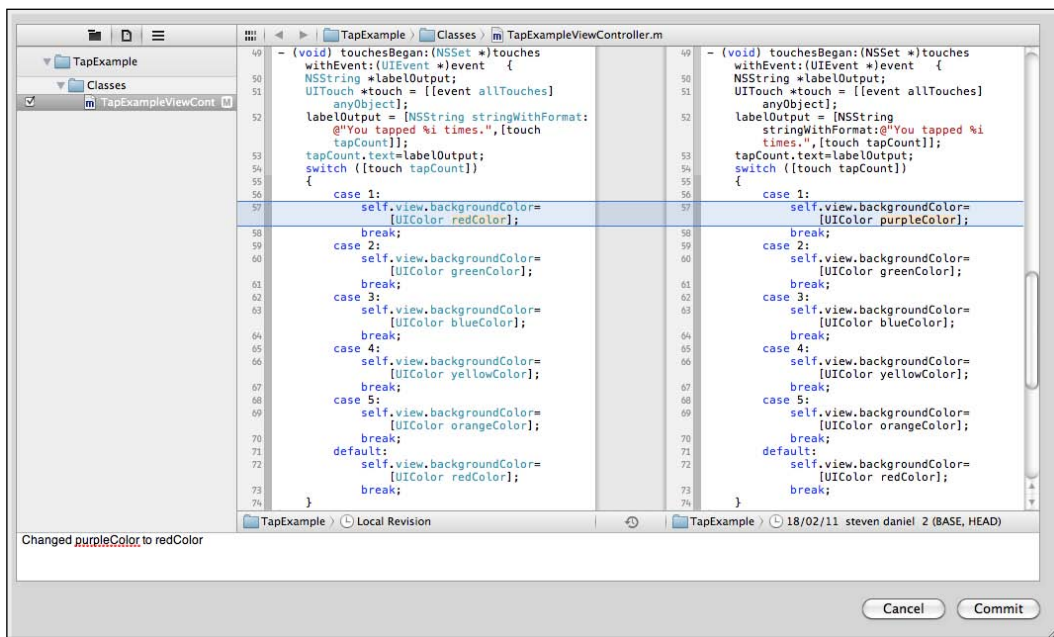
Whenever you make changes to a file within your project, these are changed and stored locally and are not included as part of the source code control repository. Before you can add those changes to the repository, you must first commit changes made to the file. Saving changes to a file is not the same as committing it into the repository.

In order to see what files have been modified since the last checkout from the repository, look for the **M** badge next to the filename within the Project navigator. Any new files which are added to your project will have an **A** badge beside their name. Any files that are not under source control will have a question mark badge.

When you choose the **Commit...** option from the Source Control menu, a confirmation dialog is displayed that you can use to make sure that the changes which you are committing are what you intended.

Within this comparison view, any changes that you make to the file locally can be compared with any previous version stored within the repository. Only those files that you select will be added into the repository. In order to proceed committing your changes, a comment must be provided before this can happen. Any last minute changes to the file can also be applied and are saved back into your project.

Make sure that the comment you provide is informative enough, so that anyone else using the repository can see what changes were actually done to the file. If the changes made to the file was a bug fix, provide the ID. If the change was an enhancement, provide the change request number:



The **Merge** facility helps you to reconcile different branches and allows you to merge the code in a separate branch back into the main branch, or when you want to combine the code in any two branches to reconcile differences between the branches.

The **Update** or **Pull** commands update your working copy from the repository when two or more people are working on the same project. From time to time, you will need to synchronize your working copy with changes made in the local or remote repository.

When using **Git**, you can use the **Pull** command or the **Update** command in Subversion to do so.

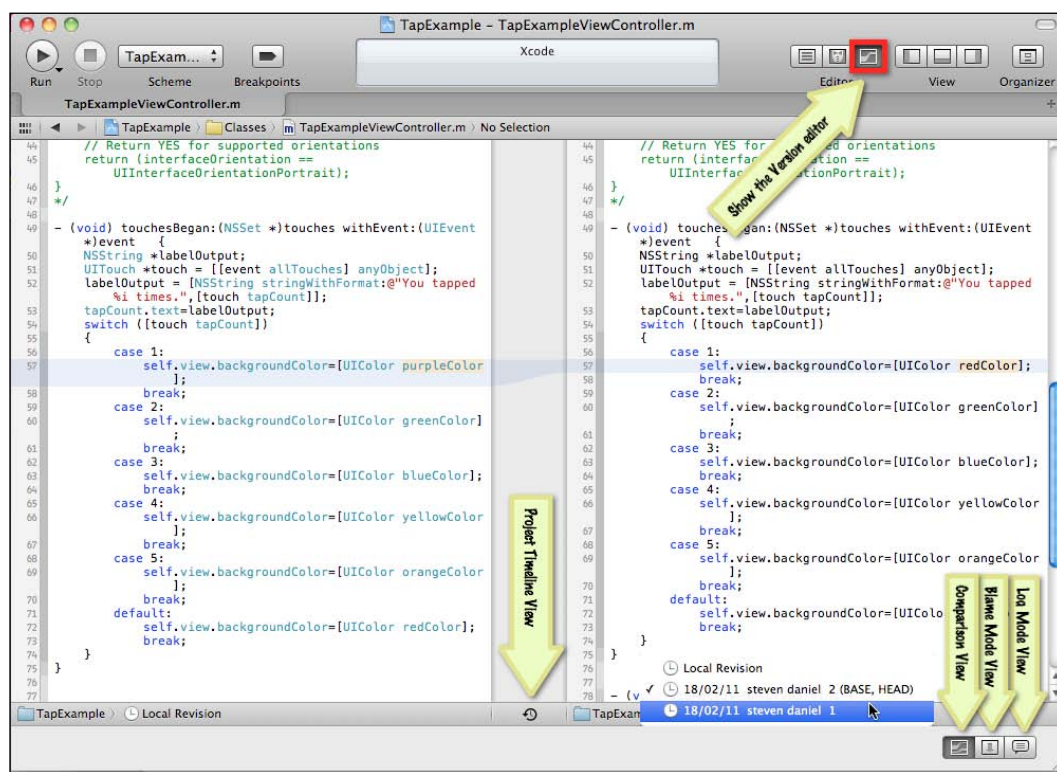
If you are using a remote or local Git repository and you want to share your work with the other members of your team, you can use the **Push** command to push your files back to the repository.

Before you can perform a Push, you will need to save and commit any changes made to your project files and then execute the Pull command to reconcile any differences between your version and the one stored within the repository, before finally selecting the Push command.

Comparing different versions of a file side-by-side

There may be times when you want to compare a file which you are working on with another one located within the repository. It could be that someone else in your team may have applied some changes to one of the code modules within your project and it is causing some issues and you need to determine what the previous change was.

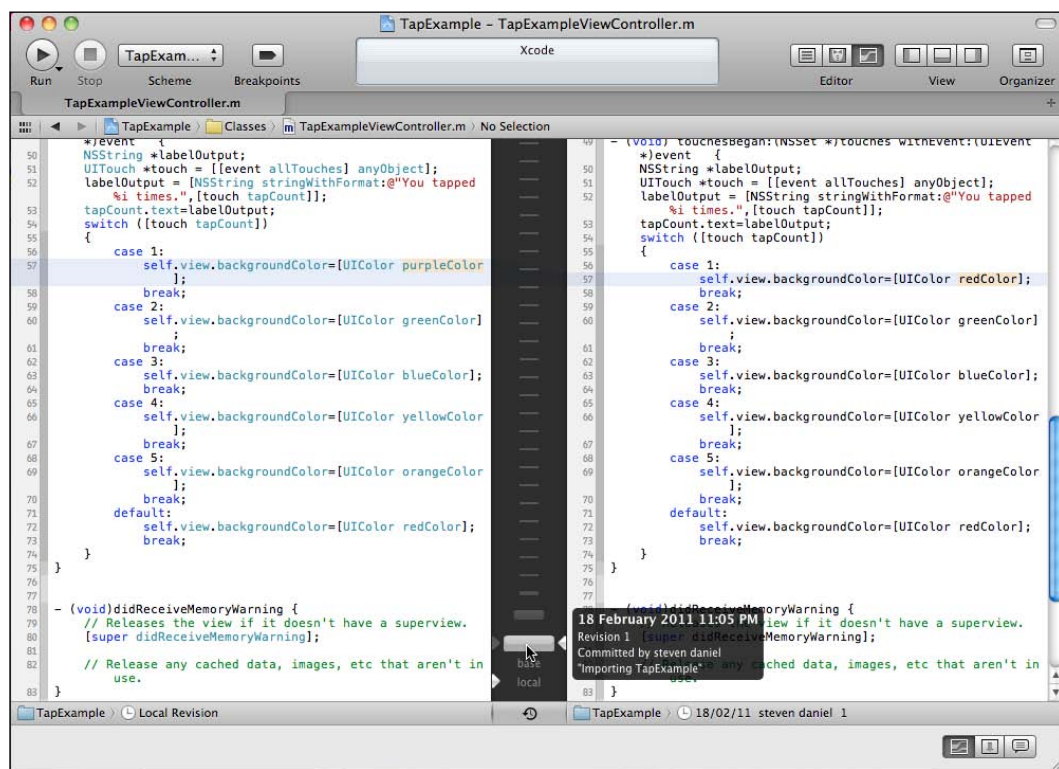
This is where the version editor comes in and it is particularly useful to find out what has been changed and why. In order to compare any two versions of the same file under source control, select the file from within the Project Navigator window and then click on the Show the Version editor button and Comparison View button, as shown in the screenshot below:



Use the jump bar underneath each of the editor panes to select the version of the file to compare with the one in the other pane. It is possible to select any committed version of that file from any revision within the repository and merge the code changes from the version editor for the revision being viewed directly into the current working copy of the file.

Using Timeline to select and compare revisions

Another great feature of the Version editor which you will find very useful is the ability to select any revision within the repository to compare with the working copy of the file. This can be very useful if you want to track what changes have been made to this file. However, there is a much simpler way to achieve this and we will take a look at this in the section *Using Track Blame to check past check-ins*:



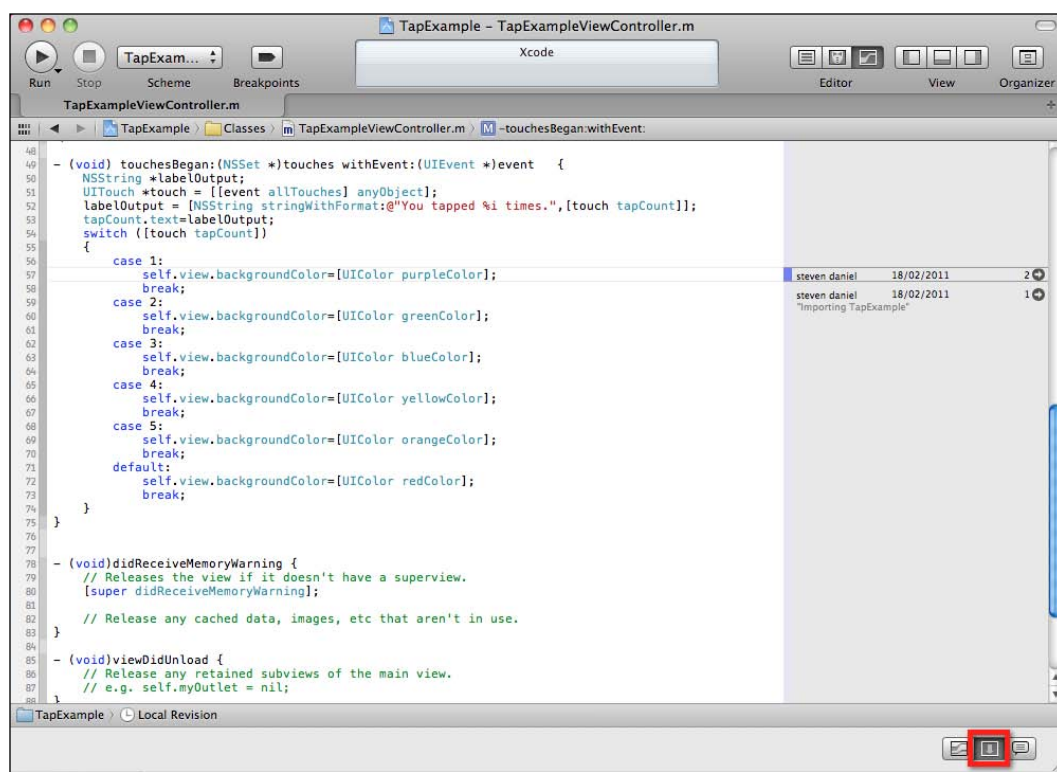
In order to use the version timeline to choose file versions to compare, click on the Timeline icon in the center column to display a visual timeline of all repository revision versions.

Use the sliders to control which version of the file should be displayed in each windowpane; you can use the *Up* and *Down* arrow keys to cycle through the available versions. Versions are listed in chronological order, with a line for each version. Newer versions of the file are at the bottom of the timeline.

Each major division within the timeline group's revisions is submitted within a twenty-four hour period. As you reach each version in the timeline, additional information for that version is displayed in an annotation. When you find the version you want, click the left or right indicator triangle to display that version in the corresponding editor pane.

Using Track Blame to check past check-ins

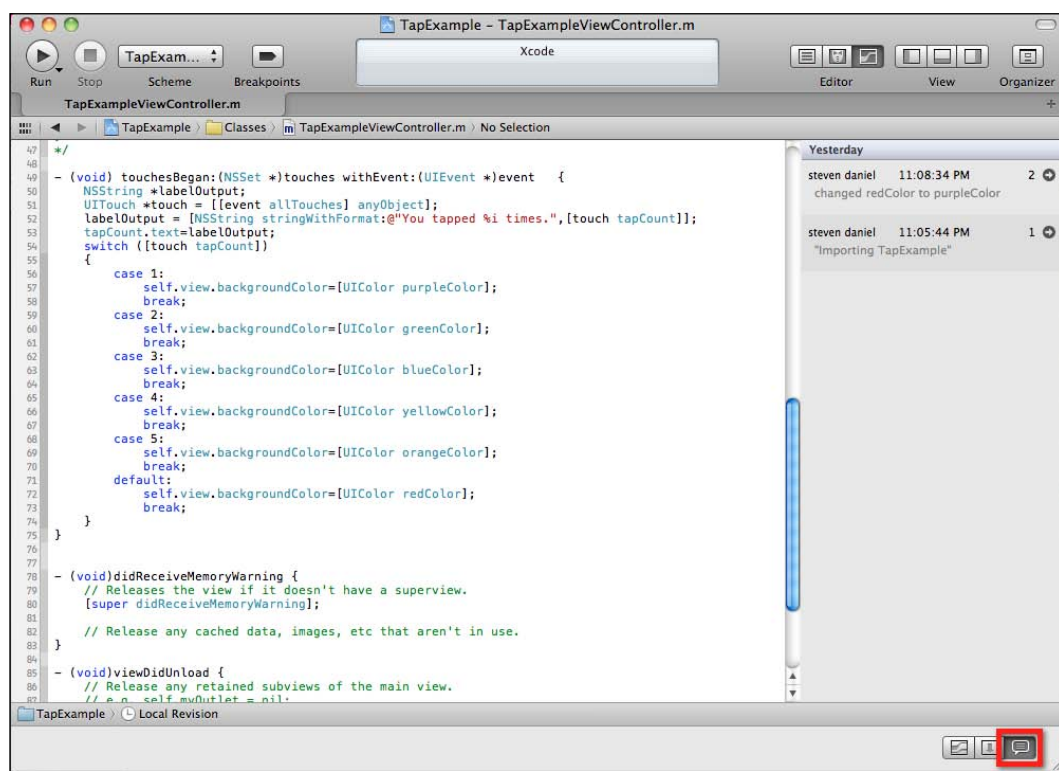
The Version editor also includes a feature called **Blame mode**. In Blame mode, the current revision of the file is displayed along with the last revision that modified each line of the file and it is even possible to see which person is responsible for submitting the last change made to the last revision of the line of code:



To display the Blame Mode screen, click on the **Blame** button, which is shown in the screenshot above. Each log entry is aligned with the line where the change was made. The log entry includes the name of the person who committed the change, the date it was committed, and the ID of the committed record. Next to the entry is an arrow button which, when clicked will open the file and let you see the change associated with it.

Using Log Mode to list all revisions chronologically

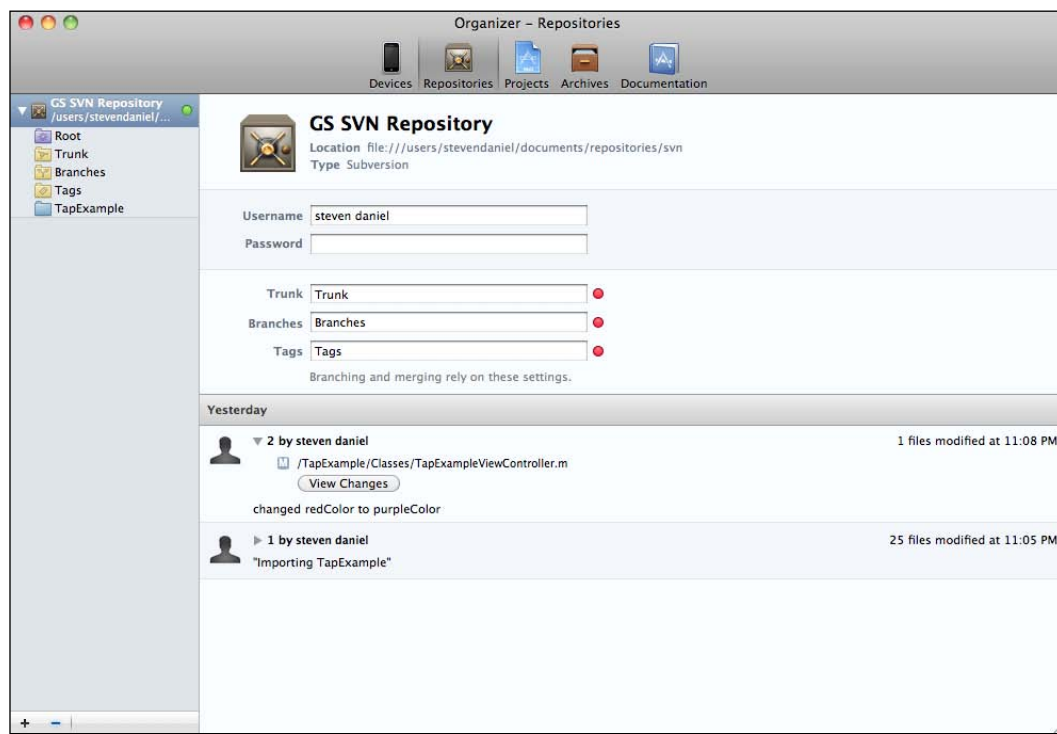
One other great feature that the Version editor comes with is the ability to review all revisions on a file. Each change made to each file is listed individually in chronological order and displays the name of the person who committed the change, the date, the ID of the commit, and the commit comments. Clicking on the arrow next to the log entry will open the file and allow you to see what changes were made to the file:



Using the Repository Organizer to keep track of your files

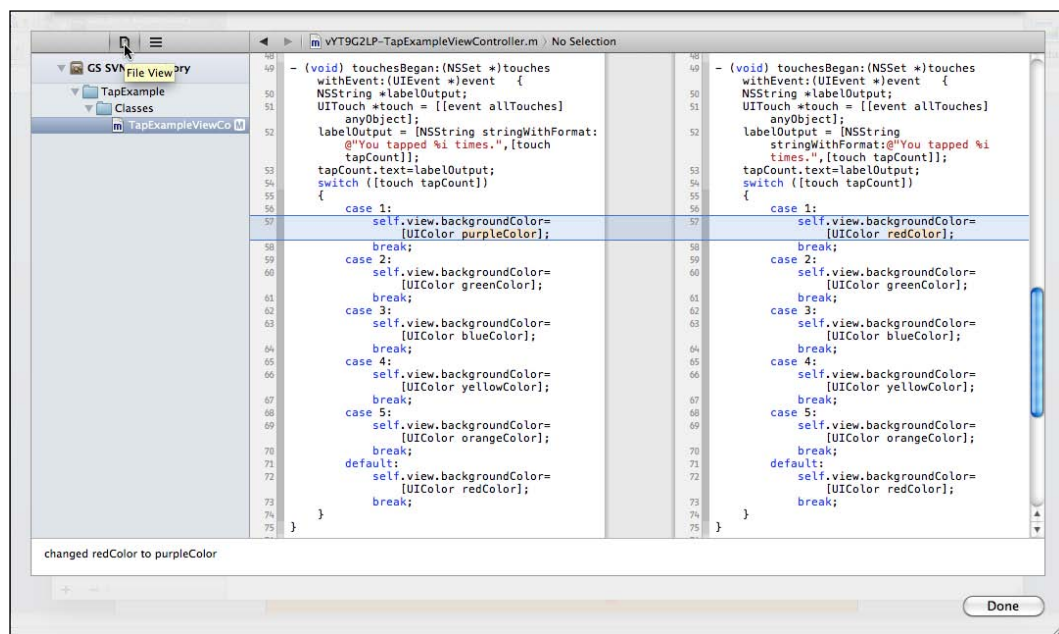
Another way in which we are able to keep track of all revision changes made to our projects contained within the repository, is to use the Repository Organizer.

This can be accessed from the **Window | Organizer** option or by clicking on the **Organizer** button from within the Xcode IDE:



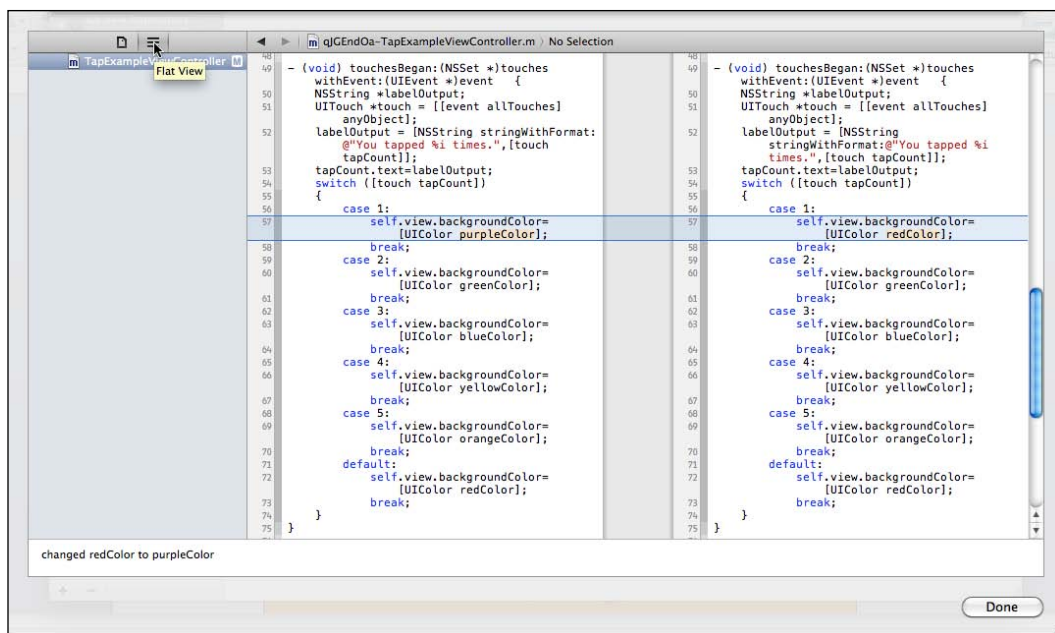
As you can see from the above screenshot it shows you all revisions made to the project, with the most recent revision located at the top of the hierarchy. It also displays the total number of files that were modified and the time the change was made. If you expand the revision node, you are able to see what files are contained within this release.

Clicking on the **View Changes** button will display the version editor in a read-only comparison view to show you what was changed:



When the version editor comparison view is displayed, you can decide to view the files in the repository which have been modified in either File View or Flat View. In File View, all files are displayed and broken down in a tree-like structure, and each of the files that have been modified are located within their respective folder(s), as shown in the above screenshot.

In Flat View, as the name suggests, all project files are displayed in a flat one-dimensional view enabling you to select each of the files and see their associated changes within each of the file panes. You can see this in the screenshot below:



Using Git to manage multiple projects

When working on a software project, it can be very useful to use source control management (SCM) to keep track of changes made to code. When using an SCM system, it saves multiple versions of each file on disk, storing metadata about each version of each file in a location, that is known as the SCM repository. Xcode supports two SCM systems: Subversion and Git.

When using **Subversion**, it is always better to have this stored on a remote computer which is backed up on a daily basis, but it can be stored locally as we have seen in the examples above.

Git on the other hand, can be purely used as a local repository, or like Subversion, can be installed as a Git server on a remote machine in order to share your files amongst your team members. Xcode provides a consistent user interface and workflow for users who use either Subversion or Git.

Time for action – creating a new Xcode project using Git

Before we can proceed, we first need to create the `UsingGitExample` project. To refresh your memory, you can refer to the section which we covered in *Chapter 2, Introducing the Xcode 4 Workspace* under the section *Creating your first iPhone application*:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project** or **File | New Project**.
3. Select the View-based Application template from the list of available templates.
4. Click on the **Next** button to proceed to the next step in the wizard.
5. Enter in **UsingGitExample** as the name of the Product to create.
6. Select **iPhone** from under the Device Family dropdown.
7. Click on the **Next** button to proceed to the next step in the wizard.
8. Specify the location where you would like to save your project.
9. Ensure that you have checked **Create local git repository for this project** from under the Source Control section.
10. Click on the **Create** button to continue and display the Xcode workspace environment.

What just happened?

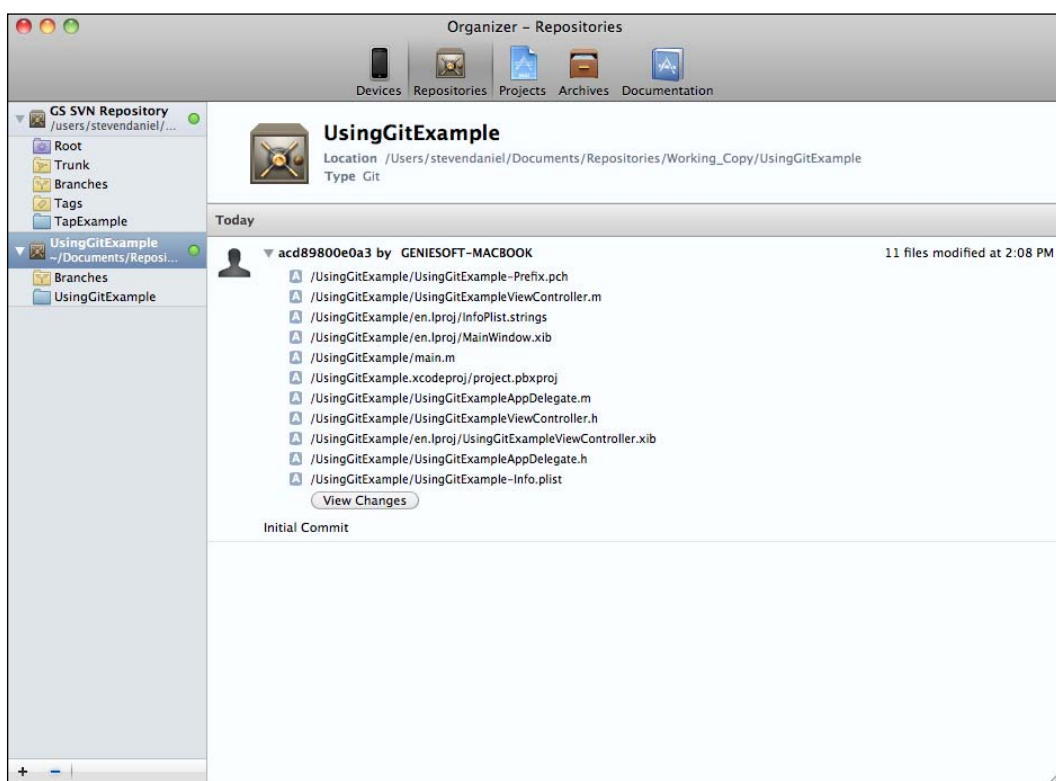
In this section, we looked at how we can use Xcode to create a new project using Git to manage multiple projects and having this added automatically for us under Source-Control.

As you can see, creating Git projects is much easier than creating Subversion projects and as you begin to start using them you will come to understand why it is the preferred choice of many developers and is highly recommended by Apple. In the next section, we will look at how we are able to use the Organizer screen to assign address book information for each person and are able to examine changes made to files when they have been committed back into the repository.

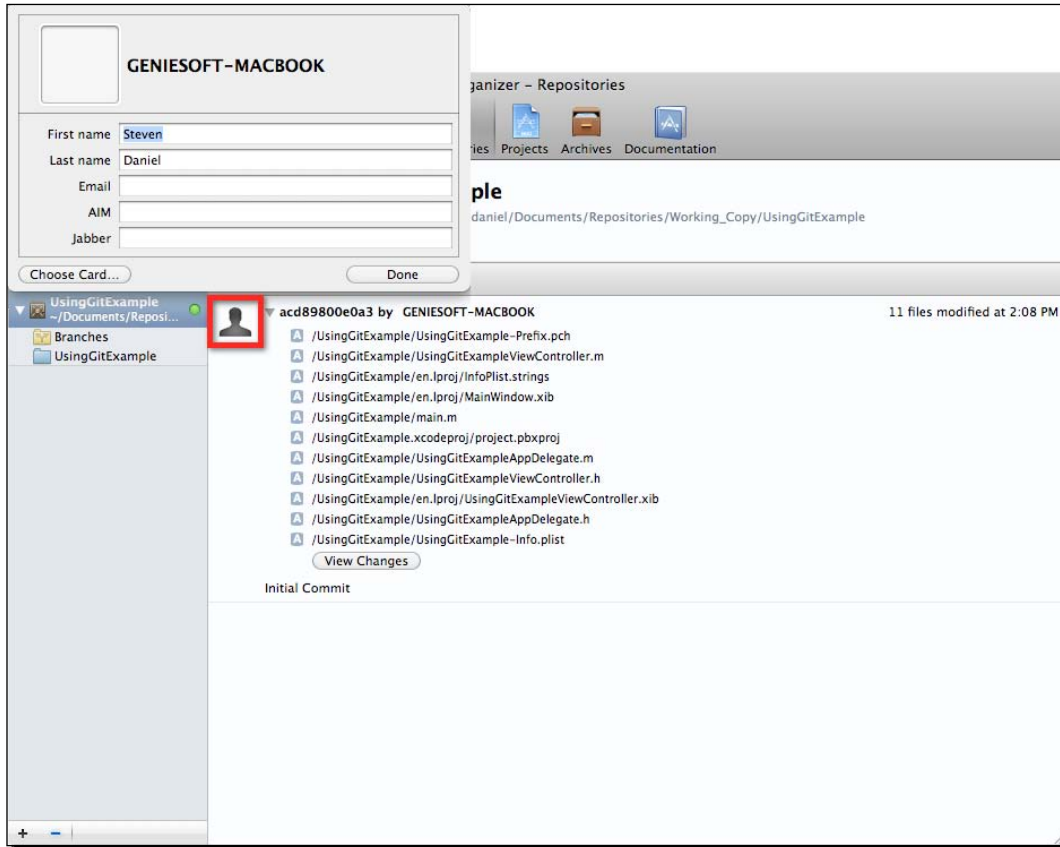
Time for action – assigning address book identities within the organizer

Now that we have created our project, we will look at how we can assign address book information to the person who committed each version listed into the repository and let you examine the changes made to each of the files at each commit:

1. Open the Repository Organizer.
2. Select our `UsingGitExample` repository in the navigation pane of the repositories organizer:



3. Click on the icon next to the commit ID number:

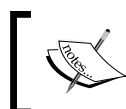


4. Fill in the information about the person who executed the commit. If the person is in your address book, you can click on the **Choose Card...** button and select the person's card in your address book. If your address book has an associated picture, it will also be displayed.
5. To see what changes were committed for this release or to see what changes were made on a specific file, click the **View Changes** button.

What just happened?

In this section, we looked at how we can add information applying to the user that committed the changes to the Git Repository. Comparing Subversion and Git, you can obviously see why Git is the perfect way to go when creating your projects. While Subversion is good, it was designed to replace Concurrent Versions System (or CVS) to save and retrieve multiple versions of source code.

Git on the other hand is a *distributed version control system*, that was designed to handle everything from small to very large projects, and was designed for speed and efficiency.



If you are interested in reading up on and learning more about Git, please check out the provided links: [http://en.wikipedia.org/wiki/Git_\(software\)](http://en.wikipedia.org/wiki/Git_(software)) and <http://git-scm.com/>

Have a go hero – adding a project to a Subversion repository

Now you have a good knowledge of Subversion and understand how it all works, and are familiar with how to go about creating a repository. The task that you will be performing will be to add the `MoviePlayer` example, that we created in *Chapter 4, Working with the Xcode Frameworks* to a repository:

1. Load up the Xcode 4 Organizer
2. Next, locate the `MoviePlayer` project and then import this into the repository.
3. Provide a descriptive message for your project prior to importing.
4. Once the project has been imported, verify that you can see the `MoviePlayer` folder added to the navigational pane, under the repository to which you imported.
5. Exit from the Organizer window to return back to the Xcode IDE.

Once you have followed the above steps correctly, you would have successfully added a project into an SVN Repository..

Pop quiz – Subversion / Version Editor

1. What command and switch would you use to set up a local subversion repository?
 - a. `svn --version`
 - b. `svnadmin create`
 - c. `svnadmin --create`
 - d. `svnadmin verify`
 - e. `svnadmin /create`
2. What command would you use to check the version of svn installed on your machine?
 - a. `svn`
 - b. `svn --version`
 - c. `svn --version`
3. Where would you find the import and checkout buttons?
 - a. File | Source Control
 - b. Editor menu
 - c. **Organizer | Repositories**
4. What is the purpose of the Timeline feature of the Version Editor?
 - a. **Transport the file back in time**
 - b. Warp speed engage
 - c. Provides the ability to select any revision of the file within a repository to compare with the local copy of the file to track changes made
 - d. I don't know
5. What option displays a list of all revisions made to a file in chronological order?
 - a. **Track Blame**
 - b. Comparison View
 - c. Log Mode
 - d. A Library
6. Which Source Control Status specifies that the file(s) have been modified locally, with regards to SCM?
 - a. *
 - b. D
 - c. ?
 - d. M

Summary

In this chapter, we focused on the new features of the Xcode Version Editor and how to go about creating, configuring, and adding items to existing source code repositories.

We also spent some time looking at how we can use the Xcode Version Editor to compare different versions of the same source file and the Track Blame feature to check on what changes were made by other developers, before finishing up learning how we can use Subversion and Git together to manage multiple projects.

Now that we have learned about the features and capabilities of the Version Editor, how to manage source code repositories, and gained some insight into what Subversion and Git are, we are now ready to get stuck in and focus on how we can use Xcode Instruments to track down memory leaks and how we can profile our application to ensure that it is running smoothly.

In the next chapter, we will be taking a look into how to go about *Making your applications run smoothly*, and taking a look at the new features that come with Instruments, and how to track down iPhone graphics performance using OpenGL ES.

10

Making your Applications Run Smoothly

In this chapter, we will focus on how we can effectively use Instruments within our applications to track down memory leaks and bottlenecks within our applications. These types of issues could potentially cause our application to crash on the user's iOS device.

We will take a look into each of the different types of built-in instruments, that come as part of the Instruments application and how we can use the Leaks instruments to help track down and determine where memory leaks are happening within our code. We will look at how we can configure instruments to display data differently within the trace document that is being reported.

In this chapter, we will be covering the following topics:

- ◆ Introducing the Instruments environment
- ◆ Learning how to add and profile against different instrument sets
- ◆ Learning how to track down and fix memory leaks
- ◆ Introducing other components of the Instruments family
- ◆ Introducing the new Instruments that are included with Xcode 4

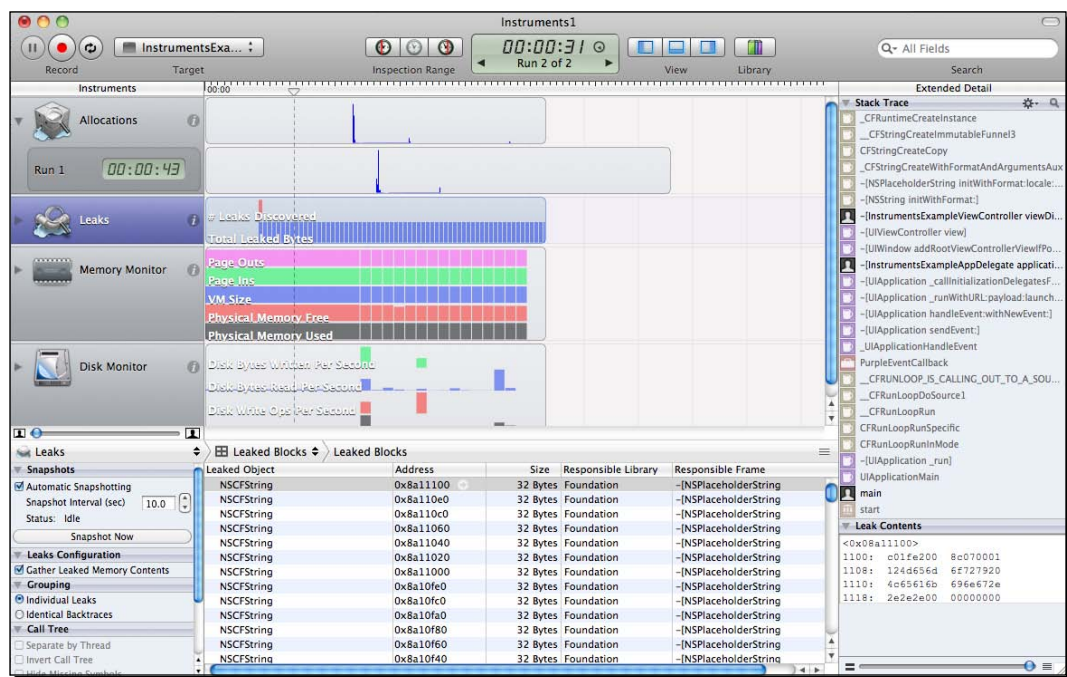
We have got quite a bit to cover, so let's get started.

Introducing Instruments

The Instruments application is a powerful tool that enables you to collect information about the performance of your application over time. Instruments lets you gather information based on a variety of different types of data and view them side by side at the same time. This lets you spot trends that would be hard to spot otherwise and this can be used to see code running by your program along with the corresponding memory usage.

The instruments application includes a standard library, that you can use to examine various aspects of your code. You can configure instruments to gather data about the same process or about different processes on the system.

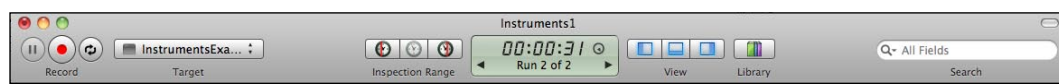
Each instrument collects and displays different types of information relating to file access, memory usage (leaks and allocation), and so forth. As you can see from the screenshot below, it shows the Instruments application profiling against our `InstrumentsExample`, using a number of different types of instruments to monitor how much memory is being allocated but not freed, what parts are causing memory leaks, number of disk reads and writes, and how much memory is being consumed:



The information in the table below outlines each feature of the Instruments application, and provides a description about what each part covers:

INSTRUMENTS FEATURE	DESCRIPTION
Instruments Pane	This section lists all of the instruments which have been added for those that you want to profile against. New instruments can be added by selecting and then dragging each one from the instruments library into this pane. Items within this pane can also be deleted.
Track Pane	This section displays a graphical summary of the data returned by the current instruments. Each instrument has its own track, which provides a chart of the data that is collected by that instrument. The information within this pane is read-only.
Detail Pane	This section shows the details of the data collected by each of the instruments. It displays the set of events gathered and is used to create the graphical view in the track pane. Depending on the type of instrument, information that is represented within this pane can be customized to represent the data differently.
Extended Detail Pane	This section shows you detailed information about the item that is currently selected in the Detail pane. This pane displays the complete stack trace, timestamp, and other instrument-specific data gathered for the given event.
Navigation Bar	This shows you where you are and the steps you took to get there. It includes two menus—the active instrument menu and the detail view menu. You can click on the entries within the navigation bar to select the active instrument and the level and type of information in the detail view.

The instruments trace document toolbar allows you to add and control instruments, open view, and configure the track pane:



In the table below, an explanation is given for each of the different controls on the toolbar:

TOOLBAR ITEM	DESCRIPTION
Pause/Resume Button	Pauses the gathering of trace data during a recording. Selecting this option does not actually stop the recording; it just simply stops the instruments from gathering data while a recording is in progress. When the pause button has been pressed, in the track pane, it will show a gap in the trace data to highlight this.
Record/Stop Button	Starts or stops the recording process. You use this button to begin gathering trace data for your application.
Loop Button	Enables you to set whether the recorder should loop during playback to repeat the recorded steps continuously. This can be useful if you want to gather multiple runs for a given set of steps.

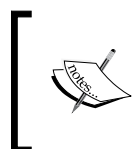
TOOLBAR ITEM	DESCRIPTION
Target Menu	Selects the trace target for the document. This is the process for which data is gathered.
Inspection Range Control	This enables you to select a time range in the track pane. When this has been set, the instruments display only the data collected within the specified time period. Using the buttons with this control enable you to set the start and ending points of the inspection range and to clear the current range.
Time/Run Control	Shows the time elapsed by the current document trace. If the trace document contains multiple data runs associated with it, you can use the arrow controls to choose which run data you want to display in the track pane.
View Control	Hides or shows the Instruments Pane, Detail Pane, and Extended View Pane. This control makes it easier to only focus on the area in which you are interested.
Library Button	Hides or shows the instrument library window.
Search Field	This option filters information within the Detail pane, based on a search term that you enter.

The Instruments application comes as part of the **Xcode 4 Tools** installation and can be found located within the <Xcode>/Developer/Applications folder; where <Xcode> is the installation folder where Xcode 4 is installed on your system.

Tracking down and fixing memory leaks

One common use for Instruments is to detect memory leaks within an application. A memory leak occurs when memory is allocated by an application, but is never released. One of the instruments that come with the Instruments application is the leak detector, called Leaks.

This nifty instrument tracks all memory that is allocated by the application and tracks all of the pointers made to that memory location. So how does the Leak Instrument know when a leak has happened? Well, this occurs when the application no longer has a valid pointer to the memory it has allocated and the Leak instrument knows that the application can never free the memory.



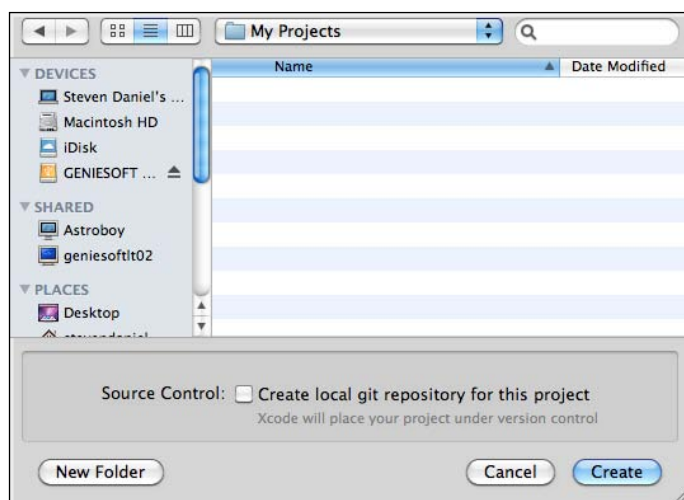
An application can use too much memory by never freeing the memory that it allocates, even after it no longer needs it. You can use the Static Analyzer to detect for potential memory leaks, or the Object Allocations instrument to detect this.

To show the use of the Leaks Instrument, we will create an example application in Xcode to show how to purposely leak memory. There are many ways in which you can start the Instruments application; you can run Instruments and then have it launch the iPhone application, or you can use the tools under the Product menu from within Xcode. Let's start by creating our sample application.

Time for action – creating the InstrumentsExample project

Before we proceed with creating our `InstrumentsExample` project, we must first launch the **Xcode** development environment. This can be located in the `/Xcode4/Applications` folder. Alternatively, you can use spotlight to search for Xcode by typing **Xcode** into the search box window.

1. Choose **Create a new Xcode project**, or **File | New Project**
2. Select the **View-based Application** template from the list of available templates
3. Click on the **Next** button to proceed to the next step in the wizard
4. Enter in **InstrumentsExample** as the name of the Product to create
5. Select **iPhone** from under the **Device Family** dropdown
6. Click on the **Next** button to proceed to the next step in the wizard
7. Specify the location where you would like to save your project
8. Ensure that the **Create local git repository for this project** is unchecked from under the **Source Control** section
9. Click on the **Create** button to continue and display the Xcode workspace environment:



We now need to start implementing the code, that will be used to perform our Memory Leak.

Open the `InstrumentsExampleViewController.m` implementation file, then scroll down and locate the `viewDidLoad` method, and enter in the following code snippet:

```
// Implement viewDidLoad to do additional setup after loading
// the view, typically from a nib.
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSLog(@"Starting....");

    // Loop for 5000 times
    for (int i = 1; i <= 5000; i++){
        NSString *MemStatus = [[NSString alloc] initWithFormat:@"Memory
            Leaking...."];
        NSLog(@"Value of i: - %i and status - %@", i, MemStatus);
    }

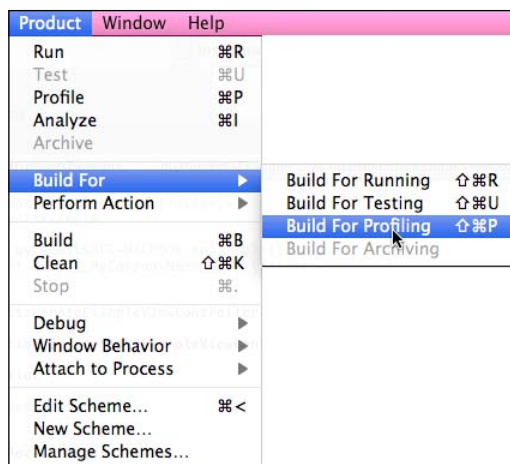
    NSLog(@"Completed...");
}
```

What just happened?

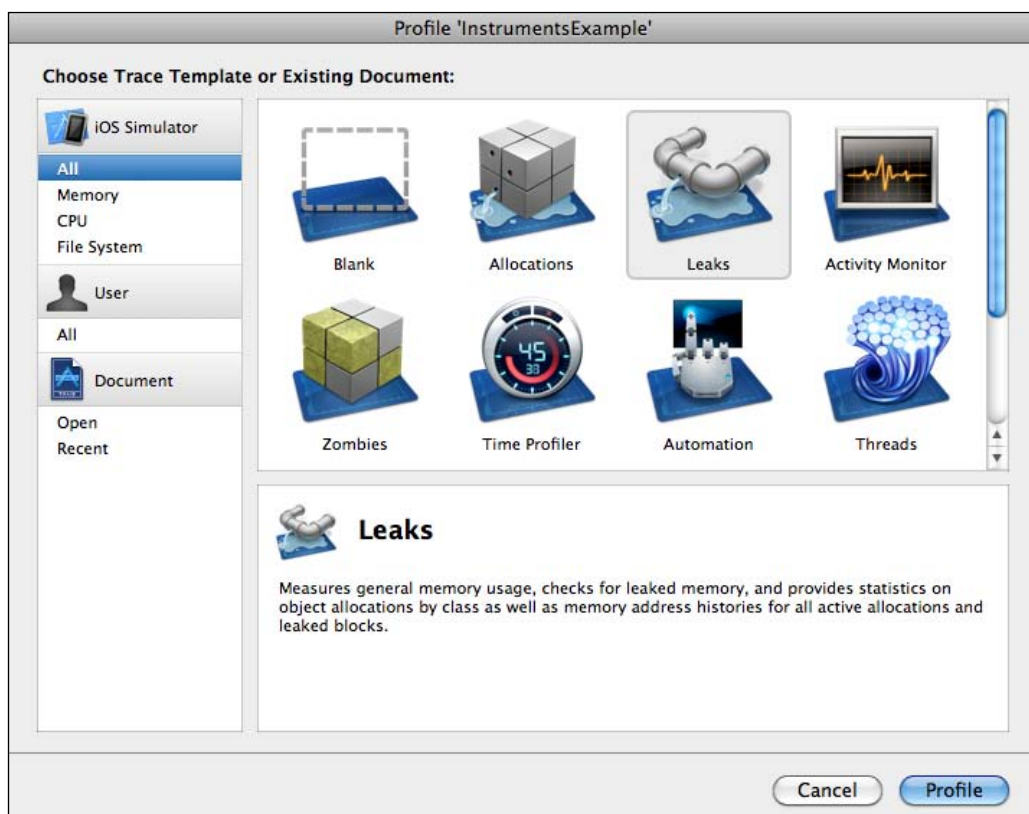
In this section, we created a simple project within Xcode and allocated 5,000 strings inside a loop to demonstrate ways of how memory leaks can happen. We then implemented some code within our `viewDidLoad` method. What this code does is that it allocates memory for a new string `MemStatus` each time through the loop, and lets the pointer to each string that we allocate go out of scope as we progress through the loop. In the next section, we will Build, Run, and Profile our `InstrumentsExample` application using the Leaks instrument.

Time for action – running and Profiling the project

We are now ready to Build and Run our application. To run the instruments application from within the Xcode environment, you can either use the *Command + P* option or the **Build For Profiling** option under the **Build For** menu. You can access this using the keyboard shortcut *Shift + Command + P*:



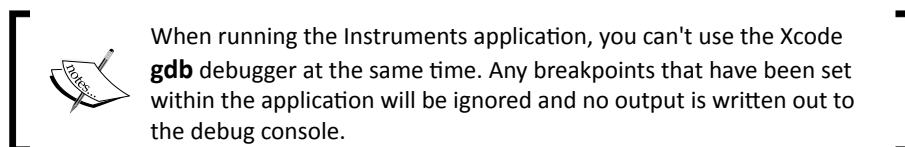
Once this option has been selected, you will eventually see the Instruments application window display on your screen. This is shown in the screenshot below:



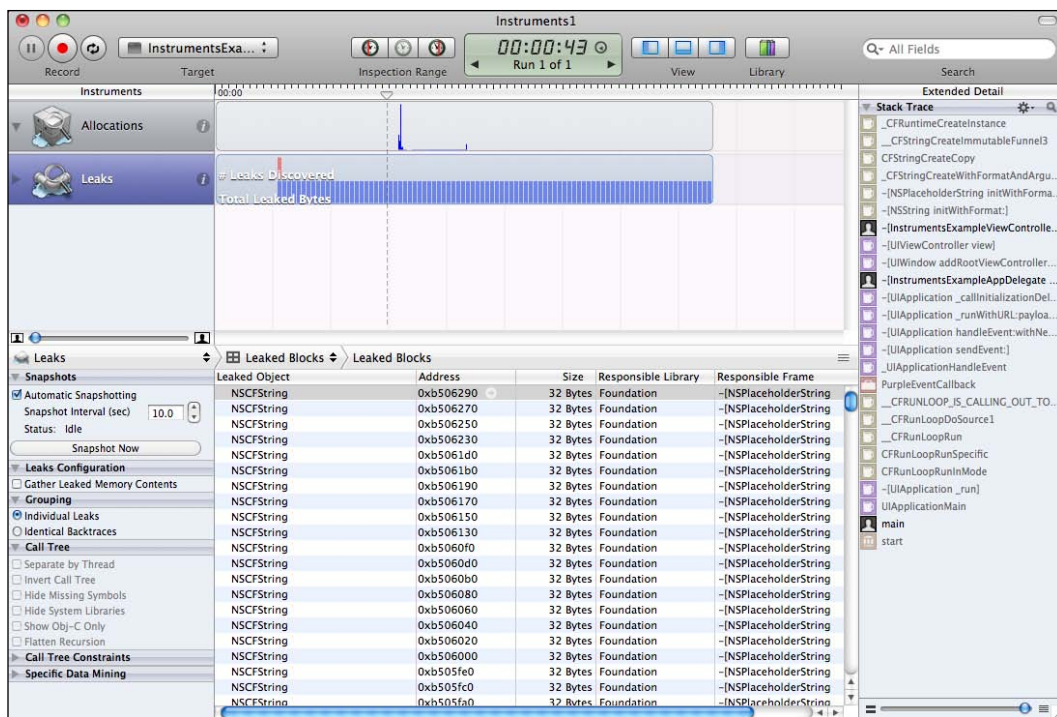
The table below gives an overview of each of the templates that are available and required for iPhone development:

INSTRUMENT TEMPLATE	DESCRIPTION
Blank Template	Creates an empty trace document to which you can add your own combinations of instruments.
Allocations	Monitors memory and object-allocation patterns within your program.
Activity Monitor	Monitors overall CPU, memory, disk, and network activity.
Leaks	Detects memory leaks within your application.
Zombies	Measures memory usage and detection of over-released objects.
Time Profiler	Performs low-overhead time-based sampling of one or all processes.
Automation	Automates User interface tests within your application.
Threads	Analyzes thread state transitions within a process, including running and terminated threads, thread state, and associated back traces.
File Activity	Monitors an application's interaction with the file system.

The type of Instrument that we want to use for this example is the Leaks Instrument. Select the Leaks option and then click on the **Profile** button to proceed to load the Instruments Trace Document window and start profiling our **InstrumentsExample** application.

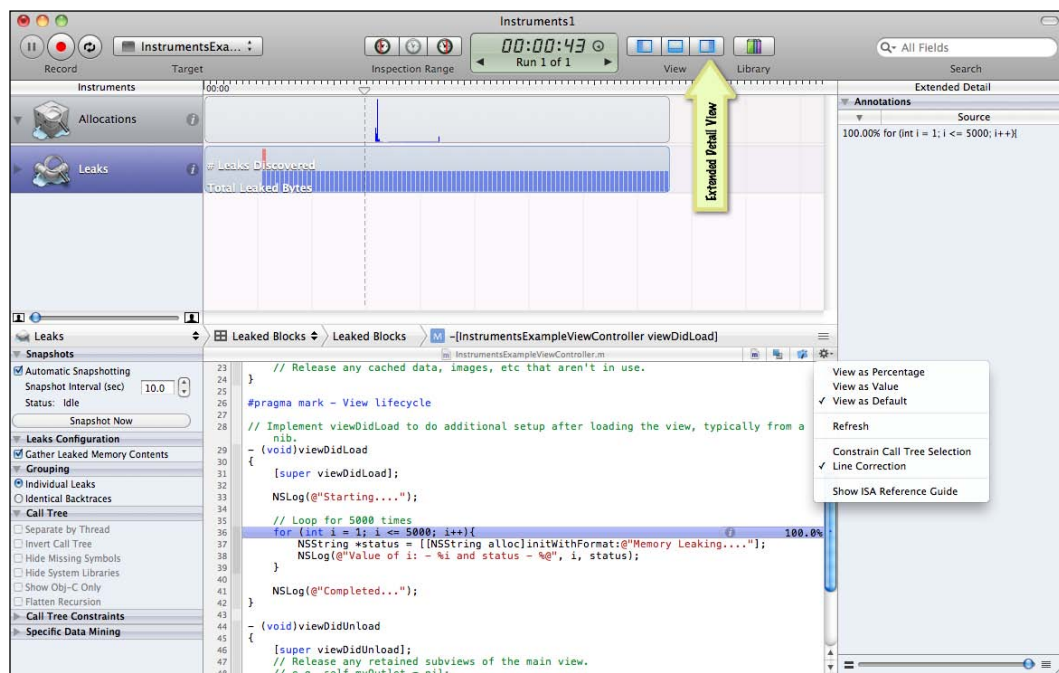


You will notice that after a number of seconds have passed, you will see a red spike appear which tells us that our application contains a leak and represents the 5,000 strings which we have allocated. You can stop the application from profiling by clicking on the red record button. Since the Instruments application has detected that our application contains leaks, it knows exactly where in our application we have allocated the memory that was leaked:



The Extended Detail portion of the Instruments window shows a color-coded stack trace. Each of the colors within this view indicate which library each method belongs to, and our **InstrumentsExample** code is highlighted in purple, with the methods highlighted in a black-grey color and we are able to see the method which allocated and leaked the memory.

If you double-click on the `InstrumentsExample:viewDidLoad` method within the list, it will open up the code module, and point you to the section where the leaked occurred:



As you can see from the screenshot above, the Instruments application has cleverly highlighted the line within the module where the leak has occurred and has provided us with the amount of processing time that this has taken up. The value can be changed to **View as Percentage** or **View as Value**. The default option for this view is **View as Default**.

What just happened?

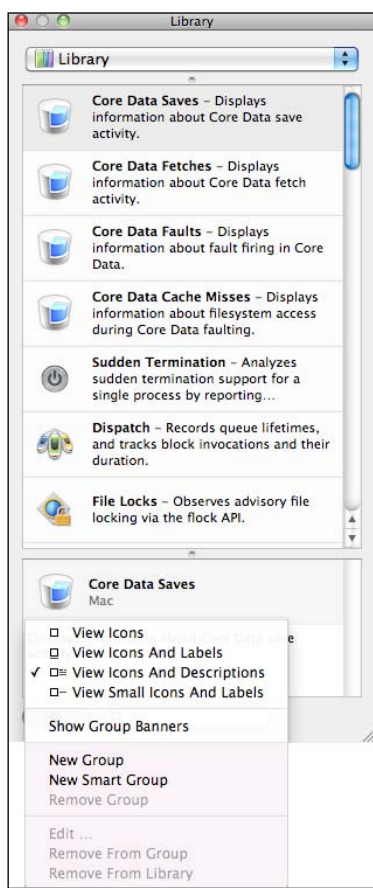
In this section, we looked at how to run and profile our example project using the Instruments application to help track down, locate, and fix memory leaks within our application, using the Leaks instrument. We looked at the different views available within the Instruments application, which displays a color-coded stack trace to indicate which library each method belongs to, with the color-code of purple indicating our code. We then saw that if we clicked on a method `viewDidLoad`, it would take us directly into the code module, and section where the leaked occurred, with the line highlighted. Through the use of instruments, this helps highlight bottle-necks within our code, that could eventually make our application crash on the user's iOS device.

Adding and configuring Instruments

The instruments application comes with a wide-range of built-in instruments to make your job easier by using them to gather data from one or more processes. Most of these instruments require little configuration to use and are simply added to your trace document to start gathering trace data. We will look at how we add and configure instruments into an existing trace document.

Using the Instruments Library

The instruments library displays all instruments that you can use and add to your trace document. The library contains all of the built-in instruments that come with the installation of Xcode 4, as well as any custom instruments that you have already created. To open the Instruments window, click on the Library button from within your trace document window or choose **Window | Library** from the menu bar. Alternatively, you can use the *Command + L* keyboard shortcut:



As you can see from the above screenshot, the Instruments Library list contains a massive number of instruments, that can grow over time especially when you start adding your own custom built instruments. The library list provides several options for organizing and finding the instrument that you are looking for by using the different view modes.

View modes help you to decide the amount of information that should be displayed at any one time and the amount of space you want that instrument group to occupy. In the table below, we describe the following view modes supported by the Instruments Library:

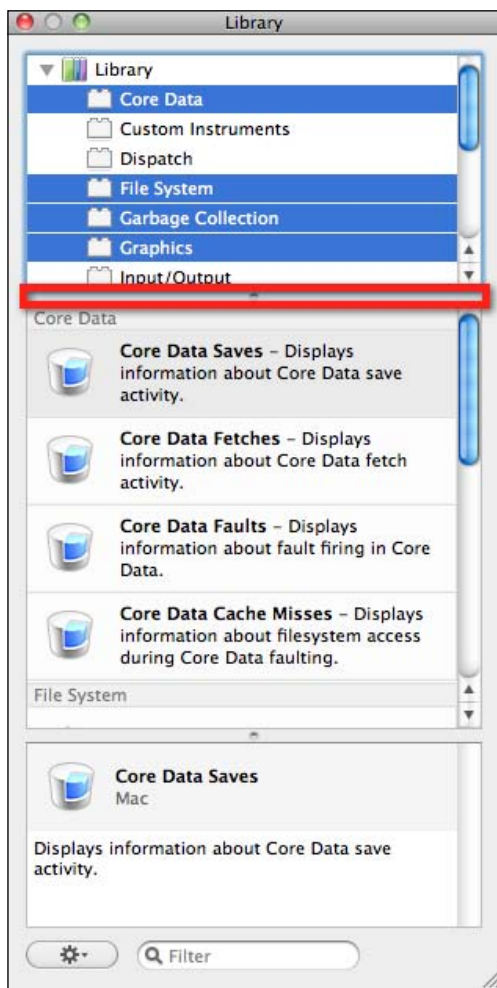
VIEW MODE TYPES	DESCRIPTION
View Icons	This setting displays only the icon representing each instrument.
View Icons and Labels	This setting displays the icon with the name of the instrument.
View Icons and Descriptions	This setting displays the icon, name, and full description of each of the instruments
View Small Icons and Labels	This setting displays the name of the instrument with a small version of its icon.

In addition to setting the view mode of the Instruments Library, Instruments can be organized into groups that make it easier to identify which instrument relates to which group. This is shown in the screenshot above.

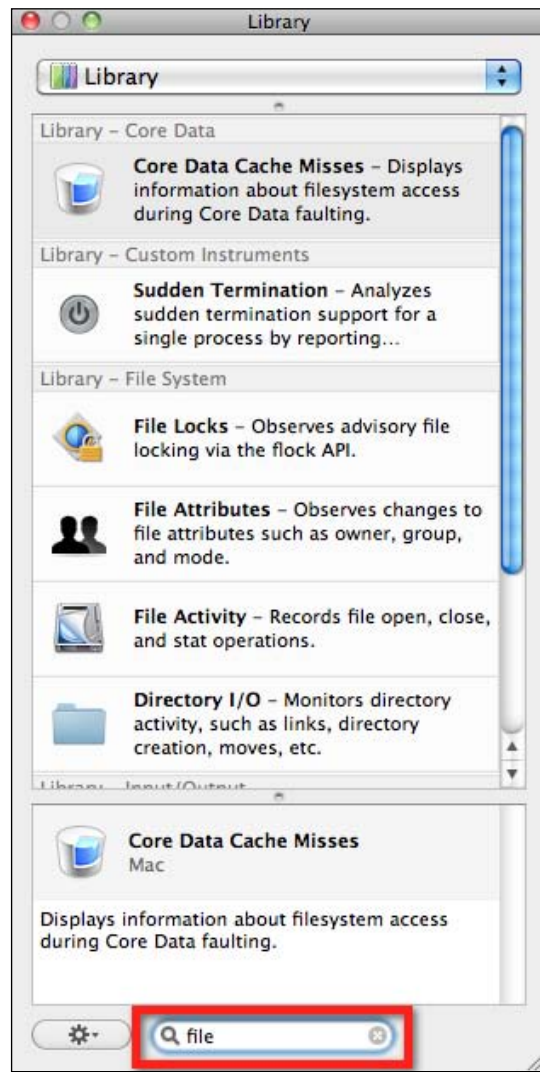
Locating an Instrument within the Library

There are two ways to locate an instrument within the Instrument Library. One common way is to use the group selection criteria controls, which are located at the top of the Library window and can be used to select one or more groups to limit the amount of instruments that are displayed within the Library window.

If you drag the split bar between the pop-up menu and the instrument pane downwards, you will notice that the pop-up menu changes from a single selection to an outline view, so that you can select multiple groups by holding down the *Command + Shift* key combinations and then selecting the desired groups to display with your mouse as shown in the screenshot below:



Another way to filter the contents of the Instruments Library window is to use the search field, that is located at the bottom of the Library window. By using this search field, you can quickly narrow down and display only those instruments that have the search keyword within their name, description, category, list, or keywords. In the screenshot below, all instruments that contain the search string **file** are displayed:

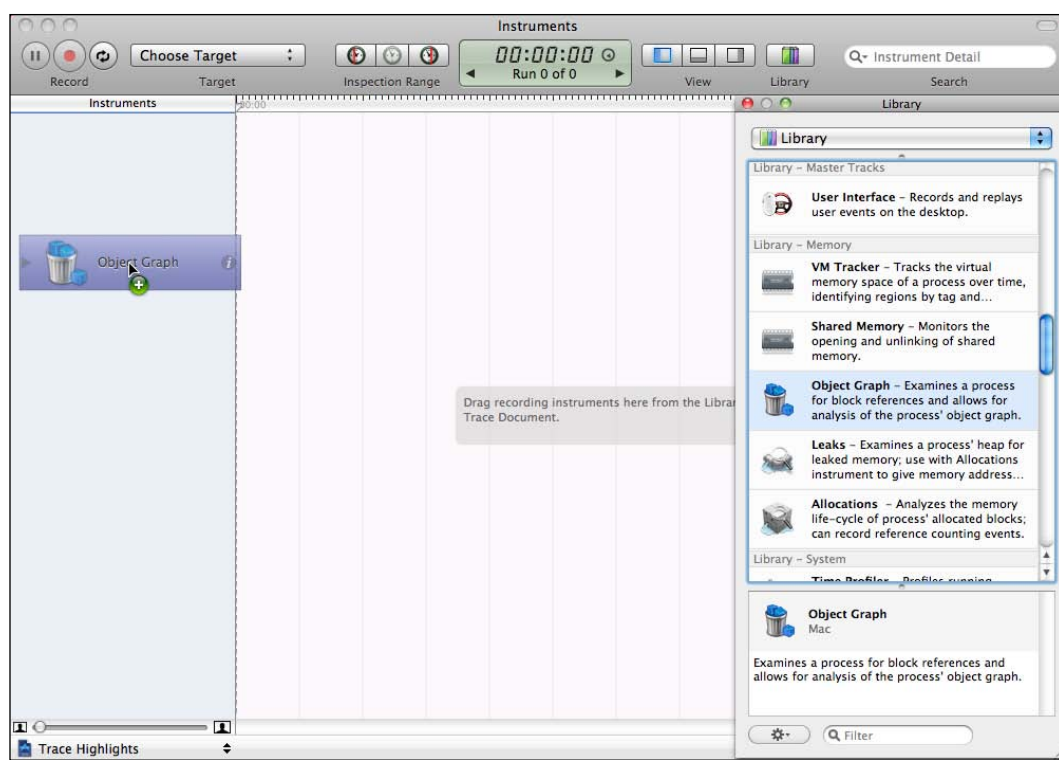


Adding and removing Instruments

There will be times when you want to trace your application against other instruments within the Instruments library. This could be when you want to check to see how your application is performing on the device and how much battery is being consumed by your application.

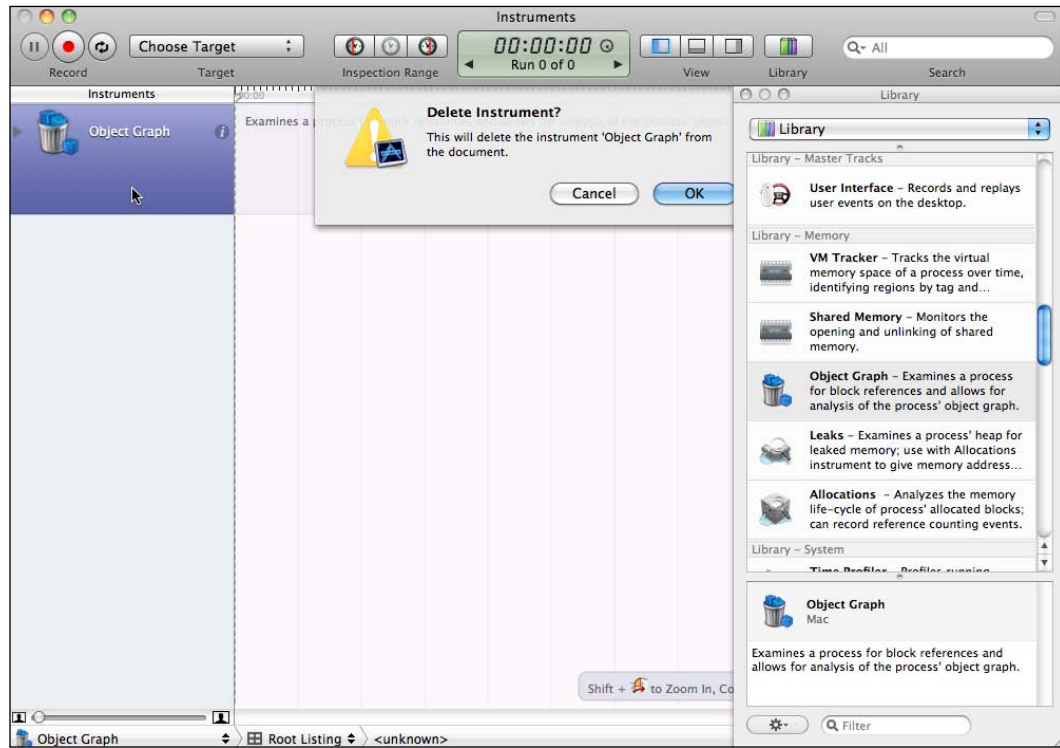
You can add as many instruments as you like to your trace document, but be aware that not all instruments included in the library are capable of tracking a wide range of system processes, as you will find that some can only track a single process. But to get around this, you can add multiple instances of the instrument, and assign each one to a different process. By doing it this way, you gather similar information for multiple programs running simultaneously.

To add an instrument to the trace document, select the instrument from the Instrument library and then drag it either to the Instruments pane or directly onto the track pane of your trace document as shown in the screenshot below:



To remove an instrument from the trace document, select the instrument that you would like to remove from the Instruments pane and then press the *Delete* key on your keyboard.

You will then receive a confirmation message. Click on the **OK** button to proceed:

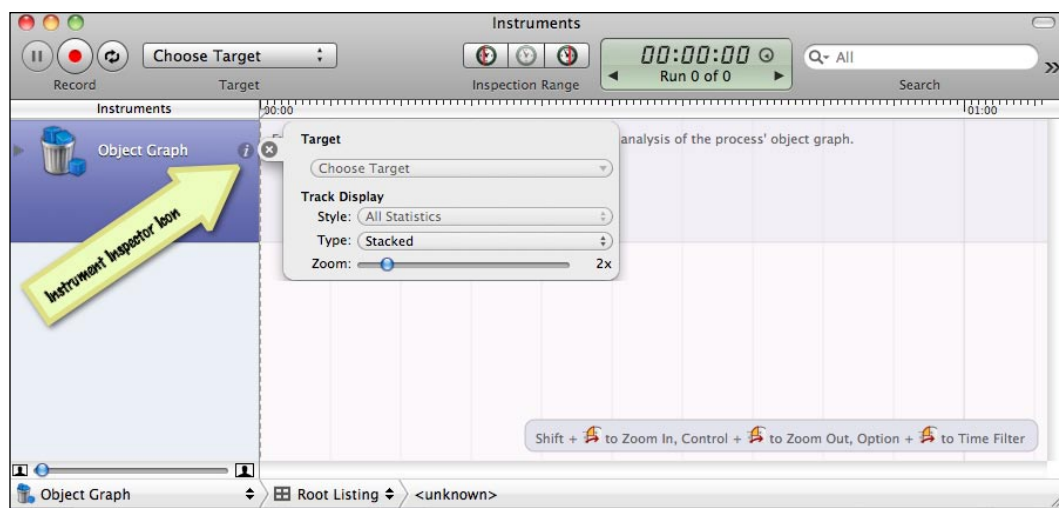


In the next section, we will look at how we go about configuring an instrument that you have added to your trace document.

Configuring an Instrument

You will find that most of the instruments that you add to your trace document are ready to use, out of the box. However, some instruments can be configured using the Instruments Inspector and this varies depending on the type of instrument that is being configured. You will notice that most instruments will contain options for configuring the contents of the track pane, while only a small handful contain additional functionality for determining what type of information is gathered by the instrument.

To configure an instrument, select the instrument from the Instruments pane and then click on the Inspector icon, which is located to the right of the instrument. This is shown in the screenshot below:



When the Instrument Inspector icon is clicked, it displays the inspector configuration dialog next to the instrument name. To dismiss the inspector, click on the close button highlighted by an X. You can use the *Command + I* and **File | Get Info** commands to close this window also.

Depending on the type of instrument that is being configured, they can either be configured before, during, or after the data within your trace document has been recorded.

The Zoom control can be found in most of the inspector controls for those instruments which you configure. This feature controls the magnification of the trace data that is displayed within the track pane and adjusts the height of the instrument within the track pane. Alternatively, you can use the **View | Decrease Deck Size** and **View | Increase Deck Size** menu options to do the same thing.

Other components of the Instruments family explained

There are other instruments which come with the Instruments application, apart from tracking down Memory Leaks and Allocation Objects. Although not every instrument works with iPhone applications, the list of Instruments pertaining to each type is explained in the table below:

INSTRUMENT TYPE	PLATFORM TYPE	DESCRIPTION
Activity Monitor	iPhone / Simulator	Correlates the system workload with the virtual memory size.
Allocations	iPhone/ Simulator	This can be used to take snapshots of the heap as apps perform their tasks. If taken at two different points in time, it can be used to identify situations where memory is being lost, not leaked. The test case would be to take a snapshot, do something in the app, and then undo that something, returning the state of the app to its prior point. If the memory allocated in the heap is the same, no worries. It's a simple and repeatable test scenario of performing a task, and returning the app to its state prior to performing the task.
Automation	iPhone/ Simulator	Used to automate user interface tests in your iOS application.
Core Animation	iPhone	Measures the number of Core Animation frames per second in a process running on an iOS device through visual hints that help you understand how content is rendered on the screen.
CPU Sampler	iPhone/ Simulator	Correlates the overall system workload with the work being done specifically by your application.
Energy Diagnostics	iPhone	Displays diagnostics information regarding the amount of energy being used on the device for GPU Activity, Display brightness, Sleep/Wake, Bluetooth, WiFi, and GPS.
File Activity	Simulator	Examines file usage patterns in the system by monitoring when files open, close, read, and write operations to files. It also monitors changes in the file system itself relating to permission and owner changes.
Leaks	iPhone/ Simulator	This instrument looks for situations where memory has been allocated, but is no longer able to be used. These memory leaks can lead to the application crashing or being shut down.

INSTRUMENT TYPE	PLATFORM TYPE	DESCRIPTION
OpenGL ES Driver	iPhone	Determines how efficiently you are using Open GL and the GPU on iOS devices.
System Usage	iPhone	Records calls to functions that operate on files within a process on the iOS device.
Threads	Simulator	Analyzes state transitions within a process, including both running and terminating threads, thread state, and associated back traces.
Time Profiler	iPhone/ Simulator	Performs low-overhead time-based sampling of one or all processes.
Zombies	Simulator	The Zombies instrument keeps an empty or 'dead' object alive (in a sense) in place of objects that have already been released. These 'dead' objects are later accessed by the faulty application logic and halt execution of the app without crashing. The 'zombie' objects receive the call and point the instrument to the exact location where the app would normally crash.

New Instruments in Xcode 4

The instruments application that comes with Xcode contains a wide range of built-in instruments to make your job easier and to gather and display data for one or more processes. In Xcode 4, a collection of new instruments has been added and these are explained below.

Automated Testing

The Automation instrument allows you to automate user interface tests of your iOS application that are guided by your test scripts which exercise the user interface elements of your application, allowing you to log the results for your analysis at a later time.

As you can see that using this fantastic Automation instrument, can simulate the many user actions supported by the devices that support multitasking and are running iOS 4.0 or later. Any test scripts that you create can be run on the iOS device or within the iOS simulator without any modifications being made to your scripts.

Performance and Power Analysis

The Performance and Power Analysis Instrument allows you to collect performance data and track the power usage of your application through the use of the Time Profiler and Energy Diagnostics Instruments for iOS.

Time Profiler

The Time Profiler Instrument illustrates how much time is being spent in each code segment. This allows developers to prioritize which bit of logic needs to be refactored prior to release. Although this can be run using the iOS Simulator, it is recommended to run this on the iOS devices, as the performance will vary greatly between the two.

Energy Diagnosis

The Energy Diagnostics instrument is the most exciting tool that Apple gave to developers.

This instrument will help you identify optimum use of the iOS device resources by enabling you to test your application as close to real world scenarios as possible. The data that is collected can later be analyzed to see how much of the device's battery life each function consumes and it will tell the developer how long each of the device's various components are used.

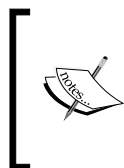
If you need to know the user's location, it will tell you which devices were turned on and for how long. GPS is a resource hog and consumes much of the device's battery life. Turning off location services once a location has been obtained is ideal.

Tracking iPhone graphics performance using OpenGL ES Driver

The OpenGL ES Driver instrument queries the GPU (*Graphics Processing Unit*) driver on an iOS device to sample OpenGL statistics for a given single process. This instrument helps you determine how efficiently your device is using OpenGL and the GPU on your device.

The GPU hardware comes with two components: Tiler and Renderer. The scene is tiled and then it is rendered. Both the Tiler and Renderer components often work on different scenes and the utilization of each component can reach up to 100%.

Using both the Tiler and Renderer utilization can be helpful in determining where bottlenecks exist. Low renderer utilization might mean that the process is stuck waiting for tiling, which would suggest decreasing the complexity of the scene that is being drawn. A low tiler and renderer utilization can suggest a CPU bottleneck somewhere else in the application.



If you are interested in reading more about Instruments, check out the Apple Developer Connection documentation at the following location: <http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html>.

Have a go hero – adding Instruments to your project

Now you have a good working knowledge of the Xcode Instruments and how to go about using these to profile your project. The task will be to profile our `MoviePlayer` example that we created in *Chapter 4, Working with the Xcode Frameworks* and add the Activity Monitor and Thread instruments to monitor each process:

1. Open Xcode 4 and load the `MoviePlayer` example program.
2. Start profiling the `MoviePlayer` example, which will launch the Instruments application. You can refer to the section *Running and Profiling the Project* located in this chapter.
3. Locate and add the **Blank** template from the **Trace Templates or Existing Document** dialog.
4. From the Instruments Library, add the Activity Monitor to the Instruments Pane. You can refer to the section *Adding and Removing Instruments* located in this chapter.
5. Next, drag the Threads instrument to the instruments Pane. You can refer to the section *Adding and Removing Instruments* located in this chapter.
6. Start profiling the `MoviePlayer` application. You can refer to the section *Running and Profiling the Project* located in this chapter.
7. You will notice that your application will compile and run within the iOS simulator, and then profile your application, monitoring the Activity and Threads being used.
8. Exit from the Instruments application to return back to the Xcode IDE.

Once you have followed the above steps correctly, you would have successfully added instruments to a blank template, as well as successfully profiling an existing application.

Pop quiz – playing with Instruments

1. What instrument helps you identify optimum use of the iOS device resources to test your application as close to real world scenarios?
 - a. Automated Testing
 - b. Performance and Power Analysis
 - c. Energy Diagnosis
2. What instrument queries the GPU (Graphics Processing Unit) driver on an iOS device?
 - a. OpenGL
 - b. Energy Diagnosis
 - c. OpenGL ES
 - d. All of the above

3. What are the ways in which you can configure an instrument?
 - a. Click on the Inspector icon
 - b. **View | Config**
 - c. **File | Get Info**
 - d. *Command + I*
 - e. All of the above
4. When profiling a project, where would you find the **Build for Profiling** option?
 - a. Under the **Product | Build For** menu
 - b. Under the **Window** menu
 - c. Under the **Product | Profile** menu
 - d. I don't know
5. What two methods can you use to find potential memory leaks for memory that has already been allocated?
 - a. Static Analyzer
 - b. Leaks Instrument
 - c. Object Allocations
 - d. All of the above

Summary

In this chapter, we focused on the new features of the Xcode Instruments application and how we can use this brilliant tool to ensure that our application runs smoothly, and that it is free from memory leaks and bottlenecks to avoid having our application crash on the users' iOS device.

We took a look into each of the different types of built-in instruments that come as part of the instruments application, in particular the Leaks instrument to help to track down and determine where memory leaks are occurring within the code in our applications. We ended the chapter by looking at how we can configure instruments to represent data differently within the trace document.

Now that we have learned about the power and features of the different types of instruments that are included with Xcode 4, we are ready to embark on our final chapter, *Building, Packaging, and Distributing your application*.

In this chapter, we will be taking a look at how to create Build configurations for Debugging stage to Release and learn how to go about obtaining provisioning profiles for Testing and Submission and how to register devices to be used for testing. Finally, we will look at the steps involved in making our application ready for submission to the Apple AppStore and join the other thousands of developers out there.

11

Distributing your Application

Well done for making it to the final chapter of this book. This is where you start to submit your application to the Apple App Store and share your creation with the rest of the community.

So, you've finally done it. You have successfully built your application and now you are ready to release it to the rest of the world. All you now need to do is decide how to deploy it and market it. In this chapter, we will look at step-by-step instructions on how to go about submitting your application to the Apple App Store.

In this chapter, we will be covering the following topics:

- ◆ How to create the different build configurations for debug and release
- ◆ Setting up your profile for testing and submitting of apps
- ◆ Setting up your iPhone development team and certificate
- ◆ Creating application IDs and how to register iOS devices for testing
- ◆ Conforming to the iPhone Human Interface Guidelines
- ◆ How to price your app and avoid it being rejected
- ◆ How to market and promote your application

We have got quite a bit to cover, so let's get started.

Build configurations – debug to release

Since the beginning of this book, the default build configuration that we have been dealing with has been the Debug Build Configuration. The screenshot below shows the currently active build configuration within the Xcode workspace toolbar:

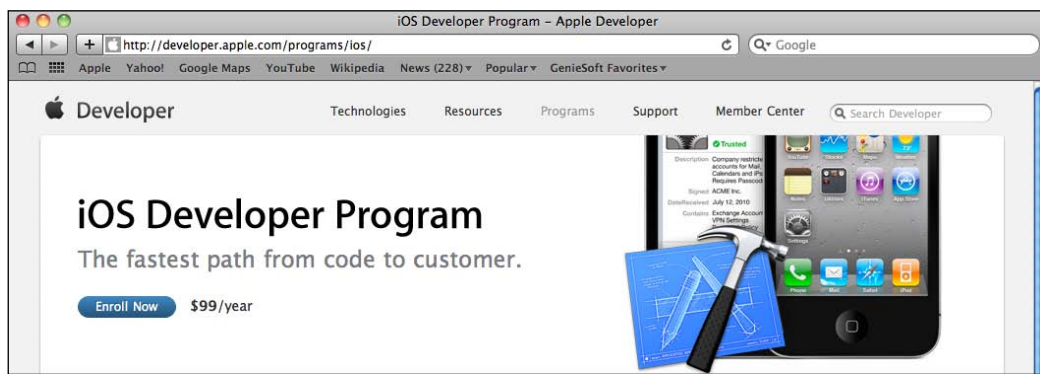


When the Debug configuration is selected, it causes your application to be compiled along with the debug symbols. On the other hand, when the Release configuration is selected, this removes the debug symbols and also carries out some optimization of the code during compilation. You can also create custom build configurations using the **Manage Schemes...** option as shown above. As a refresher, you can refer to *Chapter 8, Debugging your Xcode Projects*.

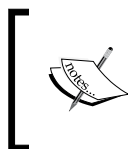
The iPhone Developer Program

The iPhone Developer Program provides you with a means of being able to share your applications with the rest of the iOS community. In order for you to test, submit, or give your apps to your friends to test, you will need to join up with the iPhone Developer Program.

To sign up, you will need to go to <http://developer.apple.com/programs/ios> and then click on the **Enroll Now** button to proceed.



For most developers wanting to release their applications to the App Store, they can simply sign up for the Standard program, that costs US\$99, or US\$299 for Enterprise users. Prices vary depending on which path you want to take—either register as an Individual/Company or Enterprise Developer.



If you are interested in learning more about the differences between the Standard and Enterprise programs, you can find more information on what these entail at <http://developer.apple.com/programs/ios/enterprise/#compare>.

When you become a member, you will have access to numerous resources to help you get started. Below is a list of some of the things that you will be able to access upon becoming a member:

- ◆ Getting started guides to help you get up and running
- ◆ Helpful tips which show you how to submit your apps to the App Store
- ◆ Access to programming guides for various areas of iOS development
- ◆ Access to sample code – TableViews, CoreData, OpenGL ES, and so on
- ◆ Ability to download current releases of the software
- ◆ Preview/beta releases of the iOS and iOS SDK
- ◆ Access to the Apple developer forums
- ◆ Access Developer videos on iOS development and WWDC 2010 (World Wide Developer Conference)

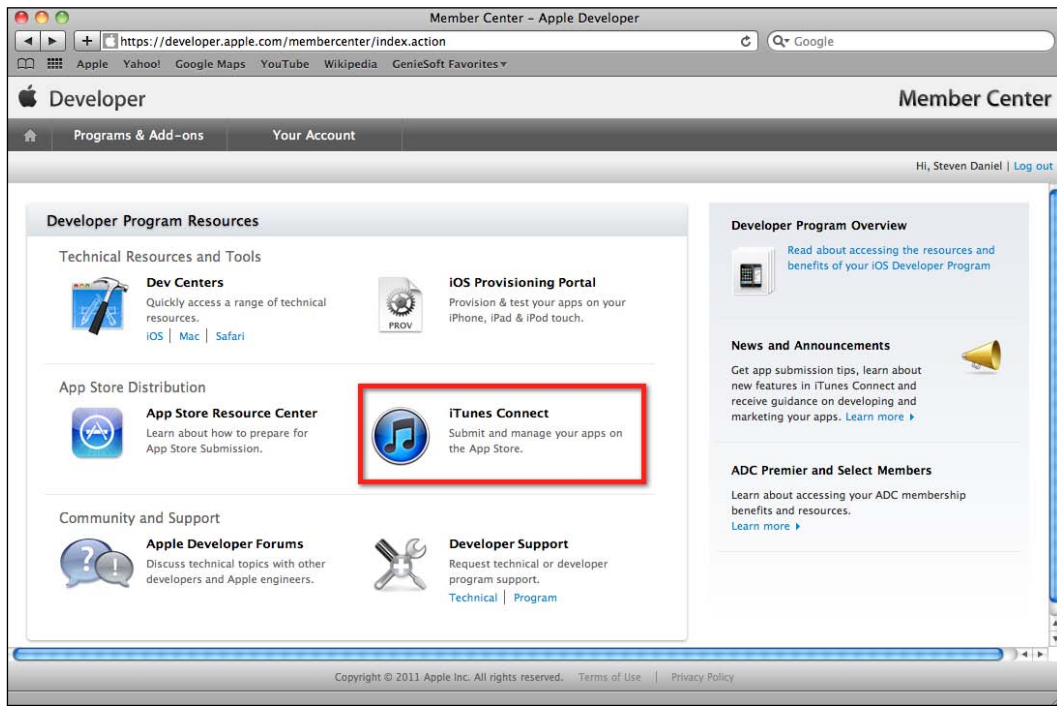
Setting up your iPhone development team

Before you can submit your application to the Apple App Store for approval, you will need to set up your iPhone development team. This enables you or the people within your organisation to log into the iOS Developer portal website to test apps on the iOS devices they are being deployed to, add additional iOS devices to the account to be used for testing, and so on.

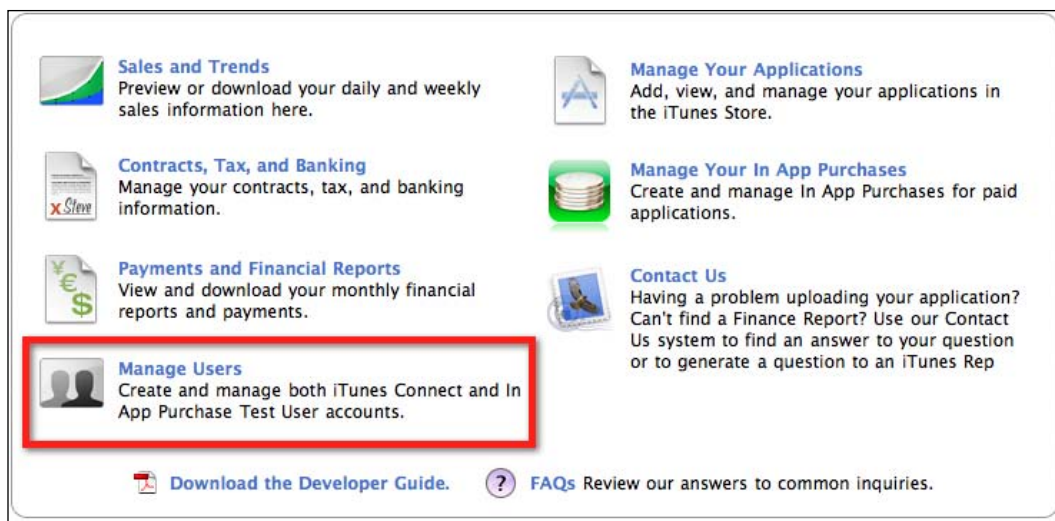
Time for action – setting up the team

In order to set up your team for iPhone development, follow these steps:

1. Log into the iOS Developer Portal website and click on the **Member Center** link which is located right at the top. This will then display the **Developer Program Resources** page, which is shown below:



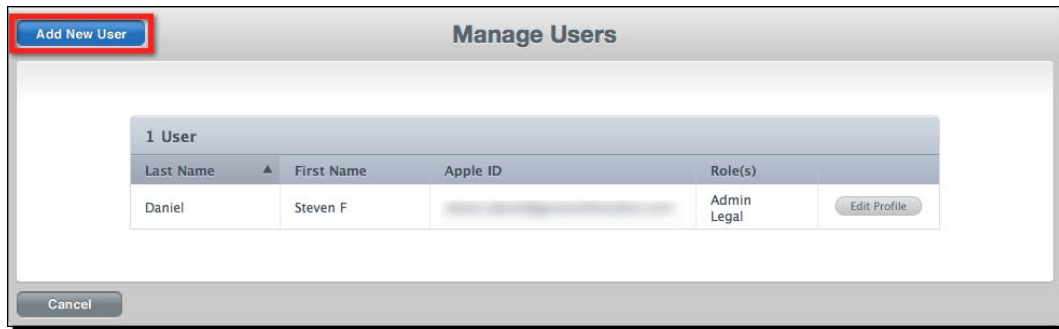
2. Next, click on the **iTunes Connect** button as highlighted in the screenshot above. This will display the iTunes Connect page where you have the ability to check on various things like **Sales and Trends** as well as **Manage your In App Purchases**:



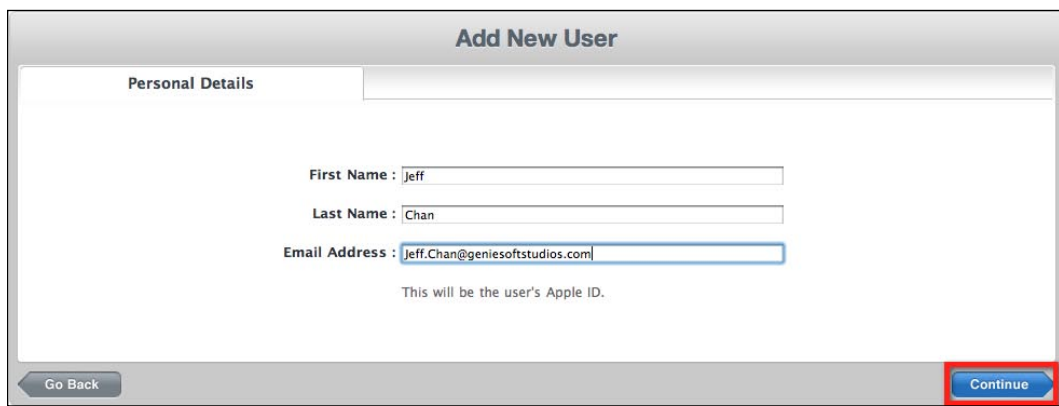
3. Click on the **Manage Users** button to add yourself or the people within your organization who will be able to log into the iOS Developer Program Portal, test apps on iOS devices, add iOS devices to the account for testing, and so on:



4. Select the **iTunes Connect User** option as highlighted above. This will bring up the **Add New User** option pane from where you can add a new user as highlighted below:



5. The list above shows a list of any existing users, that you have set up previously, along with their details and **Roles** that they have been set up with and have access to. To continue, click on the **Add New User** button:



6. Fill in the Personal details for the person that you will be adding to your development team. Once all details have been filled in, click the **Continue** button. In the next step, we will need to assign which roles the user will take on:

	☒ Admin	☐ Technical	☐ Sales	☐ Finance
Manage Users*	✓	✓	✓	✓
Manage Your Apps	✓	✓		
Manage Your In App Purchases	✓	✓		
Manage Test Users	✓	✓		
Request Promotional Codes				
Sales and Trends	✓		✓	✓
Contracts, Tax, & Banking**	✓			✓
Payments and Financial Reports	✓			✓
Contact Us	✓	✓	✓	✓

*All users can edit their own Personal Details.
 **Only users with the Legal role will be able to enter into contracts. Admin and Finance users without the Legal role will not be able to enter into contracts.

✓ Role has read and write access to this module.
 ✓ Role has read-only access to this module.

Go Back Continue

7. Select from one of the four options as shown in the screenshot above, and click on the **Continue** button to proceed to the final step in the wizard where we will be assigning the relevant notification types and territories that will be assigned to the user:

Territory	All Reports	Contract	Financial Report	App Status	Payment
Worldwide	☐	☐	☐	☐	☐
United States	☐	☐	☐	☐	☐
Canada	☐	☐	☐	☐	☐
Mexico	☐	☐	☐	☐	☐
Europe	☐	☐	☐	☐	☐
Japan	☐	☐	☐	☐	☐
Australia	☐	☐	☐	☐	☐
New Zealand	☐	☐	☐	☐	☐

See Notification description ⓘ

Go Back Save Changes

8. Once you have finished specifying the different types of notification methods for each territory, click on the **Save Changes** button. The new user account will then be created, and a confirmation e-mail will be sent to the user's account for them to activate their account.

What just happened?

In this section, we looked at how to go about setting up our iOS development team, and how to go about creating and assigning roles to users, as well as which user roles are allowed to log into the iOS Developer portal to manage users, view sales or trends, payments and financial reports, or that have the ability to add new devices in order to test apps on the iOS devices. We ended the section by looking at how we can assign the different types of notification methods for each territory to each user.

The table below explains each of the different types of notifications that are shown in the screenshot above.

NOTIFICATION	DESCRIPTION
App Status	Provides e-mail alerts with app status updates.
Contract	Provides e-mail alerts with contract status updates (for example, contract expiration warnings) or if iTunes needs more contract information.
Financial Report	Provides e-mail alerts when finance reports are available for download on iTunes Connect.
Payment	Provides e-mail alerts when payment(s) to your bank are returned.

Getting an iOS development certificate

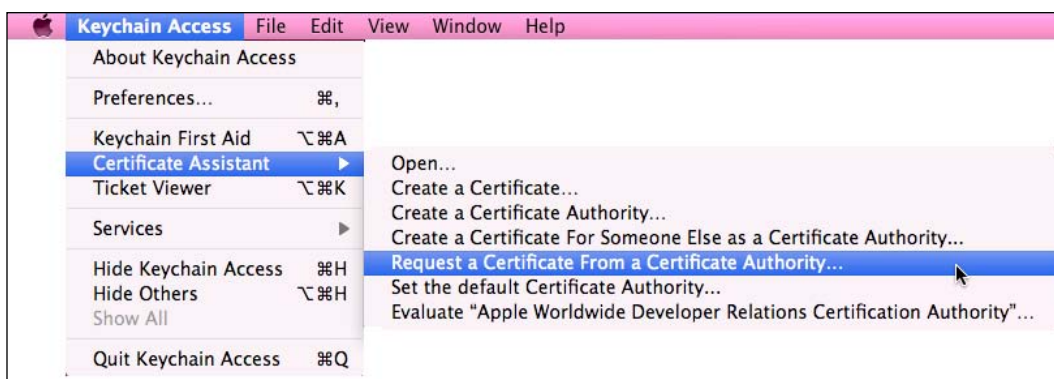
The first steps that are required before we can start to generate our iOS development certificate, and registering the iOS devices that will be used for both, development and distribution, will be to generate a Certificate Request file. This file will enable you to request the development certificate that will be used for code signing your application.

Time for action – generating a Certificate Request

In this section, we will be taking a look at the steps involved in generating an iOS development certificate. This certificate is encrypted and serves as your digital identification and you must sign your app using this certificate before you can run and test any applications that you develop on your iOS device.

In order to generate a certificate request for iOS development, you must first generate a **Certificate Signing Request (CSR)** using the pre-installed Mac OS X Keychain Access application following these steps:

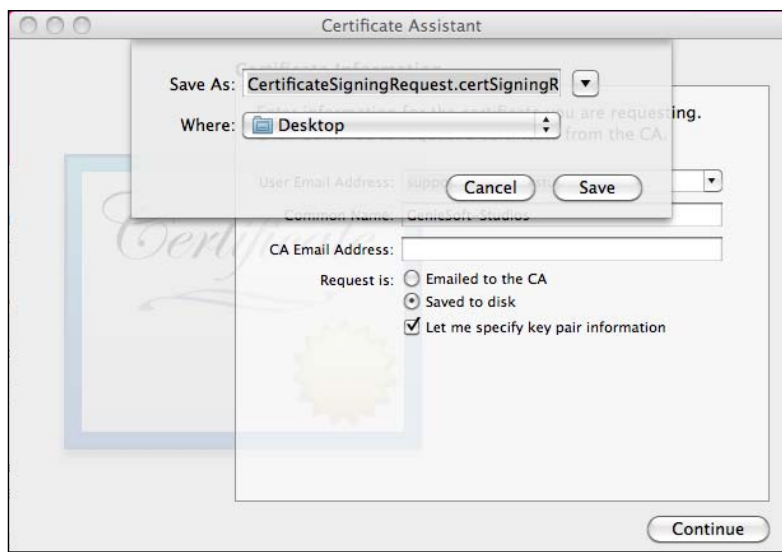
1. To begin, launch the Keychain Access application located within your /Applications/Utilities folder, or which can be accessed via the **Keychain Access | Certificate Assistant** menu and selecting the **Request a Certificate From a Certificate Authority...** option:



2. Next, we need to provide some information before the certificate can be generated. Enter the required information as shown in the screenshot below, ensuring that you have selected the **Saved to disk** and the **Let me specify key pair information** options:



3. Once all information has been filled out, click on the **Continue** button. You will be asked to specify a name for the certificate; accept the default suggested name and click on the **Save** button:



4. At this point, the certificate is being created at the location specified. You will be asked to specify the **Key Size** and **Algorithm** to use. Accept the default bits of **2048** and the **RSA** Algorithm and then click on the **Continue** button. Click on **Done** when the final screen appears:



What just happened?

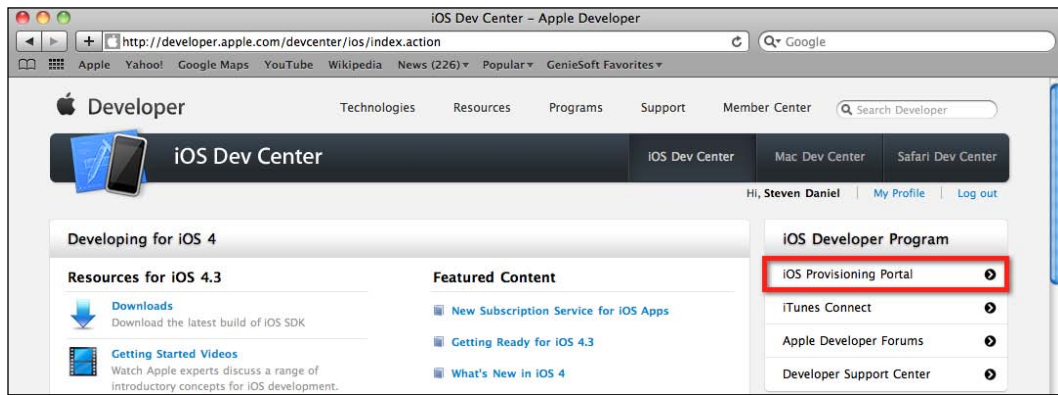
In this section, we looked at the steps involved in generating a certificate and how we can specify the type of Key pair to use for both development and distribution by using the Keychain Access Application. The Keychain Access Application is a great utility for generating certificate requests that you can then send to Apple later to request the iOS development certificates.

Time for action – getting the certificate

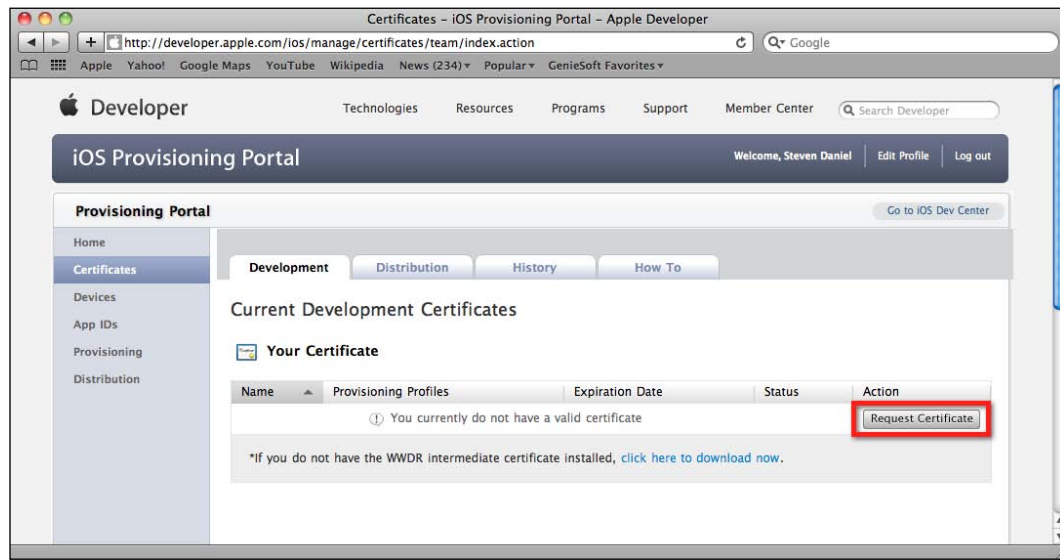
Once a certificate request has been generated you will need to use it to request what is called *A Development Certificate* from Apple. This development certificate will be used for code signing your applications in order to deploy your applications onto the real device.

To generate a development request, follow these steps:

1. Sign in to the iOS Developer Program at <http://developer.apple.com/devcenter/ios> and click on the iOS Provisioning Portal which is located on the right-hand side of the page:



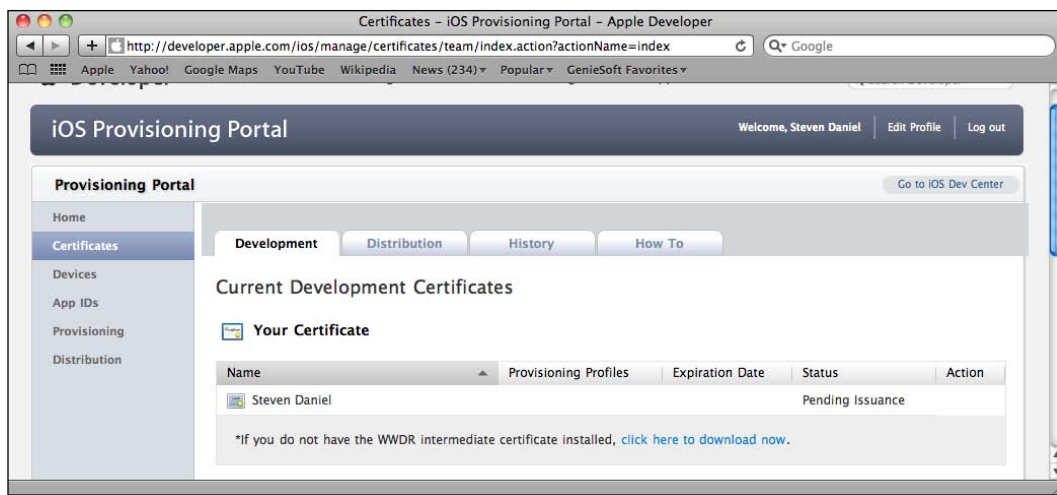
2. You will then see the **Welcome to the iPhone Provisioning Portal** page appear. Click on the **Certificates** tab, which is located on the left-hand side of the panel and then click on the **Development** tab. The **Current Development Certificates** window appears allowing you to request a certificate:



3. Click on the **Request Certificate** button; the **Create iOS Development Certificate** screen appears.
4. Click on the **Choose File** button and then select the certificate request file that you created in the previous section. Click on the **Submit** button once done:



At the point, you should see that the provisioning profile will be showing **Pending issuance** status. This is shown in the screenshot below:



- After a few seconds, the page will refresh (if this does not happen, click on the Refresh button within your browser) and the certificate will be ready and you will be able to download it. Once it is downloaded, double-click the file to install it within the Keychain Access application, as shown in the screenshot below:



What just happened?

In this section, we covered how to go about obtaining an iOS Development Certificate that you need to code-sign and deploy your applications to a real iOS device. The certificate that is installed within the Keychain Access application contains the public and private key pairs.

Next, we looked at how we can use the iOS Provisioning Portal to create a certificate for development by using the keychain file that we created in the section *Getting an iOS Development Certificate*. Finally we looked at how to download and install the certificate within our Keychain Access application.

Registering devices for testing

Before you can start to test your iOS applications on your devices for distribution and testing, you will need to register the devices to support your mobile provisioning profile. To do this, you will need the unique device identifier (UDID) for each of those devices.

Once you have obtained the device identifiers, you will need to obtain each of the users' devices in the same way you obtained your own.

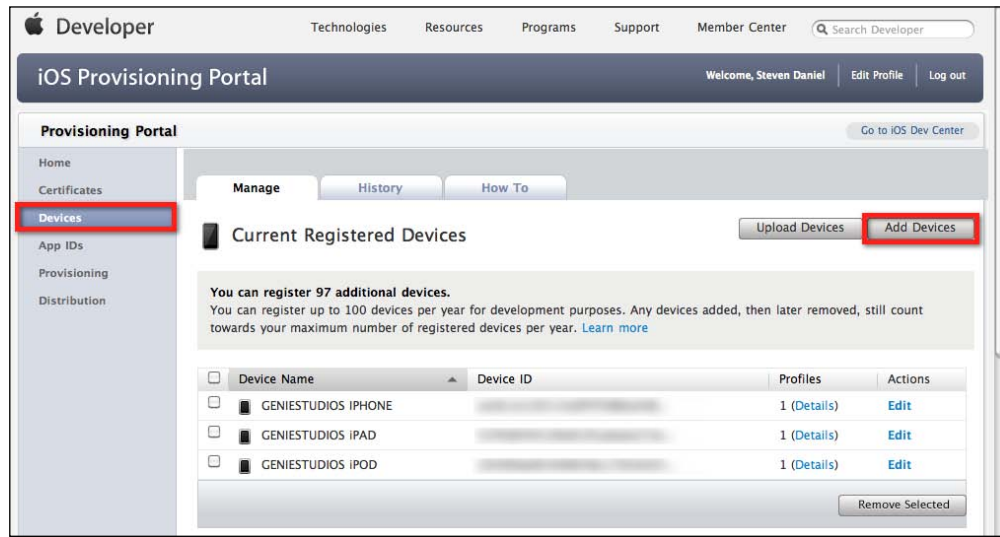
Time for action – registering devices

In this section, we will look at how to register our iOS device so that we will be able to test the applications that we develop. You can register up to 100 iOS devices for you to test your applications on. To register each device, follow these simple steps:

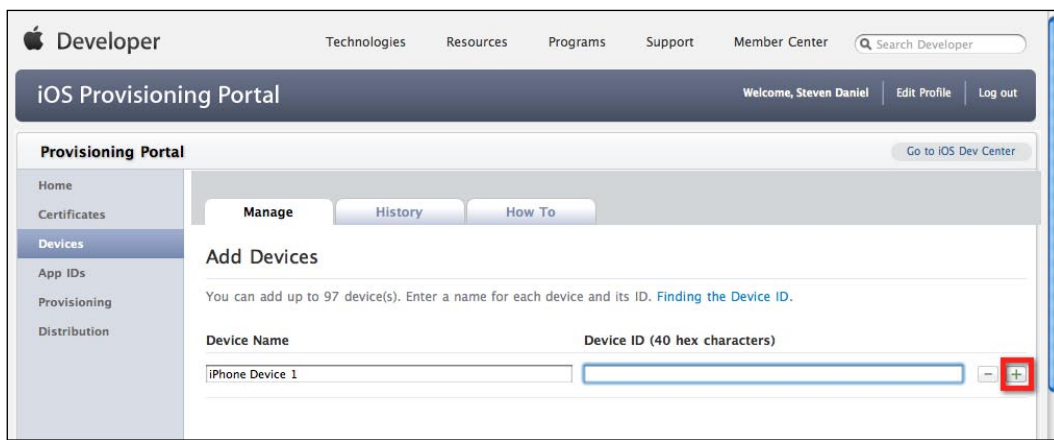
1. First, we need to obtain the 40-character identifier that uniquely identifies your iOS device. To do this, connect your device to your Mac and launch Xcode. Select the **Window | Organizer** menu item to launch the organizer application.
2. The screenshot below shows the Organizer window and the identifier of the currently connected iOS device. Copy this UDID identifier and save it somewhere, as we will be using this in the next part:



3. Log back into the Apple iOS Developer Center page and click on the **iOS Provisioning Portal** link on the right-hand side of the page.
4. Next, from the iOS Provisioning Portal page, click on the **Devices** tab, then click on the **Manage** tab and then click on the **Add Devices** button. The screenshot below shows you a listing of all iOS devices that have been registered:



5. The next step is to enter a suitable and meaningful name for each of the devices that will be used for testing. Provide the **Device Name** and UDID for the **Device ID** into each appropriate box and then click on the **Submit** button to save your changes:



Clicking on the + button will allow you to add additional devices at once. Upon clicking on the **Submit** button, you will have successfully registered each of the devices you provided. You will need to go through the same process if you intend to deploy to additional devices.

What just happened?

In this section, we looked at what is involved in registering devices to be used within iOS development within Xcode. We looked at how to register new devices using their UDIDs (Unique Device Identifier) that will be added to the iOS Provisioning Portal profile so that it can later be deployed to only those devices that have been specified.

In the next section, we will be taking a look at how to go about creating the Application IDs and development certificates so that we can use these to deploy applications to test on our iOS devices.

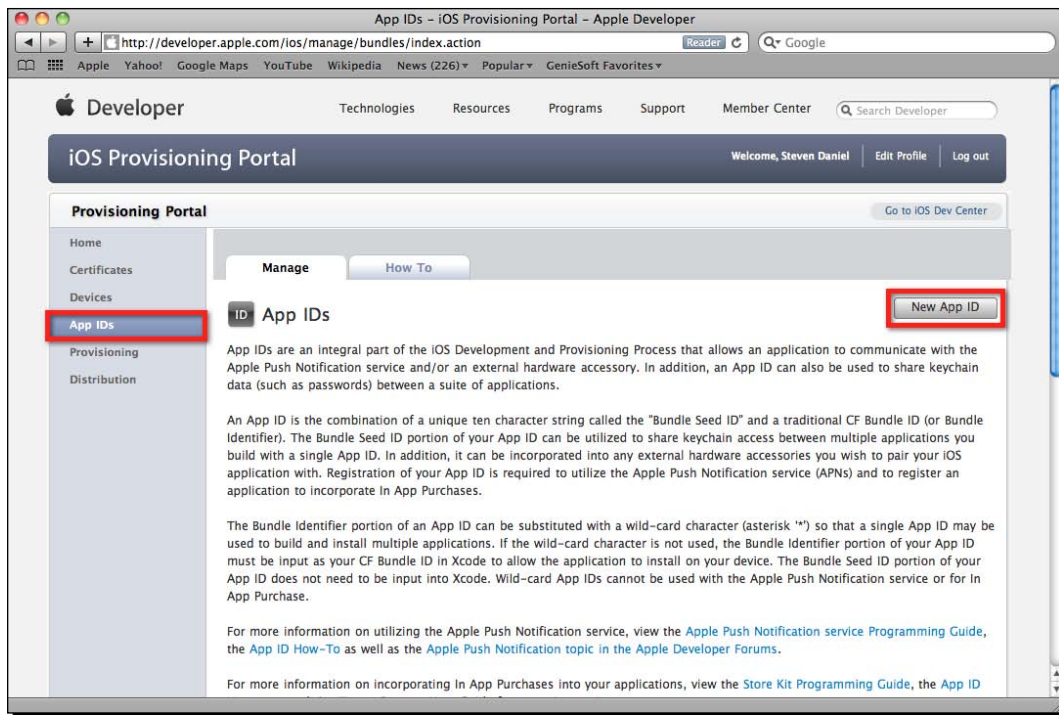
Creating application IDs

Each iOS application that you create must have a unique application ID that identifies itself. The **App ID** is part of the provisioning profile and identifies an app or a suite of related applications. It is used when your applications communicate with the iOS hardware accessories and the Apple Push Notification service and when sharing data between your applications.

Time for action – creating the application ID

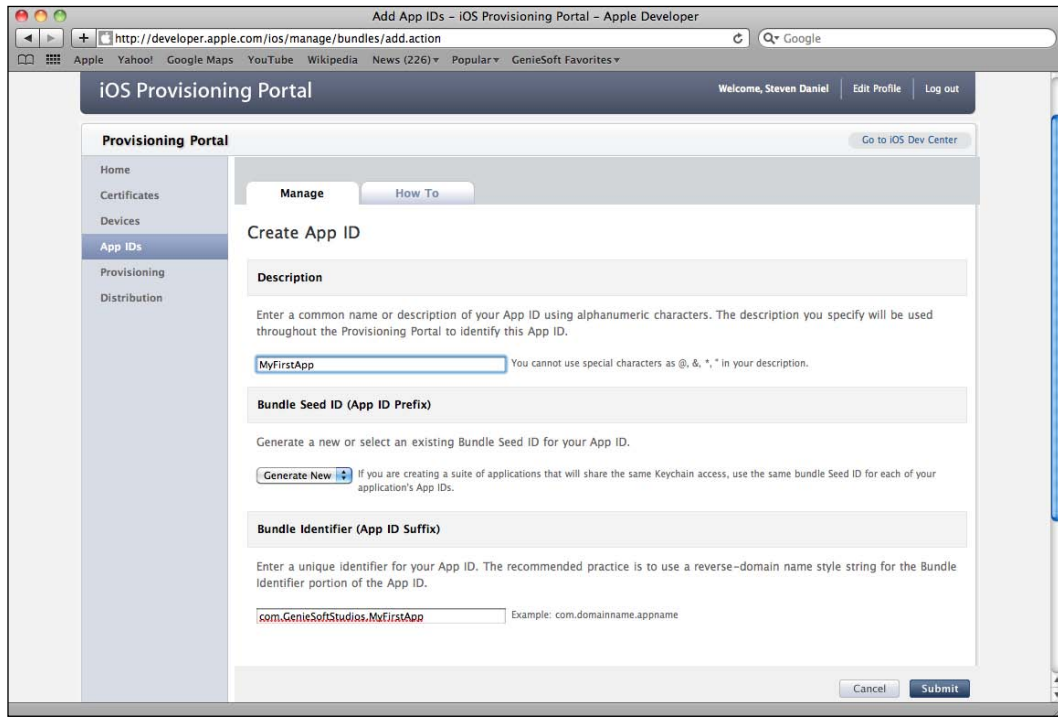
In the following steps, we will take a look at how to create an application ID:

- 1.** Log back into the Apple iOS Developer Center page and click on the **iOS Provisioning Portal** link on the right-hand side of the page.
- 2.** Next, from the iOS Provisioning Portal page, click on the **App IDs** tab, then click on the **New App ID** button:



3. You will then be asked to create an App ID for your application. This App ID is a series of characters, which will be used to uniquely identify any application, that you create on your iPhone. It is only necessary to create an App ID once per application. If you are implementing iAds within your applications, then the App ID must be unique. Provide a friendly name for your Application ID and then click on the **Continue** button.
4. Next, you will need to provide a description that will be used to identify your application and then click on the **Generate New** for the **Bundle Seed ID (App ID Prefix)**. Ensure that you provide a suitable name for your **Bundle Identifier (App ID Suffix)**.

5. Once all details have been filled in, click on the **Submit** button:



In the screenshot below, you should now see the newly created App ID that you created in the previous step, together with those you may have previously created:

Description	Apple Push Notification service	In App Purchase	Game Center	Action
X49LD83VK2.com.GenieSoftS... MyFirstApp	Configurable for Development Configurable for Production	Enabled	Enabled	Configure

What just happened?

In this section, we looked at how we can use the iOS Provisioning Portal to create the App IDs that will be used when we start to distribute our App to the App Store. This enables you to associate an individual App ID to each iOS application that you develop and is particularly useful when communicating with the iOS device hardware accessories and the Apple Push Notifications especially when sharing data between suites of applications that you develop. We then looked at how to create the Bundle Seed ID and Bundle Identifier for our App which needs to be unique. When creating the Bundle Identifier, Apple recommends using the reverse-domain style (for example, *com.DomainName.AppName*).

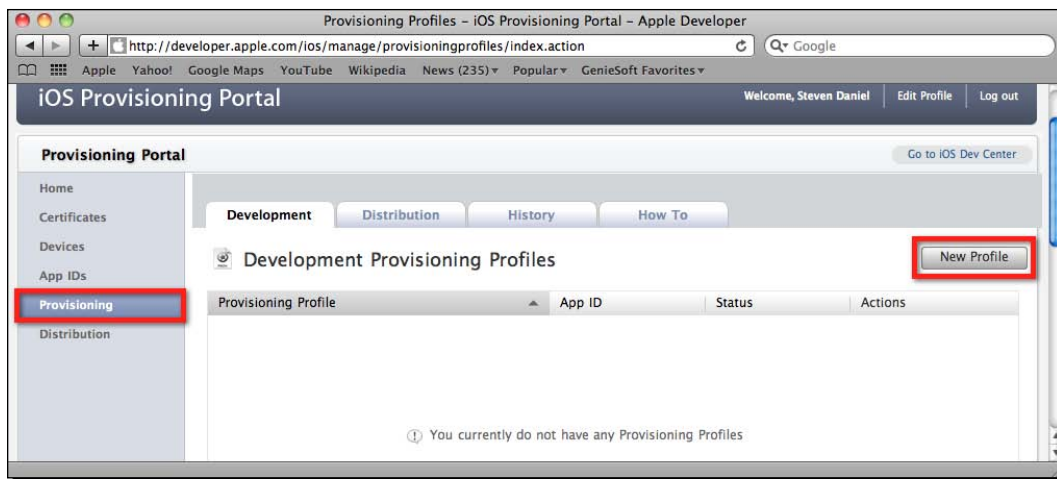
Creating a Provisioning Profile

In this section, we will look at how we go about creating a Provisioning Profile so that your application can be installed onto a real iOS device. Creating provisioning profiles gives you the ability to assign team members who are authorized to install and test an application on their iOS devices. When it is installed, it contains the iOS Development Certificates for each team member, as well as the UDID (Unique Device Identifier) and the App ID.

Time for action – creating the profile

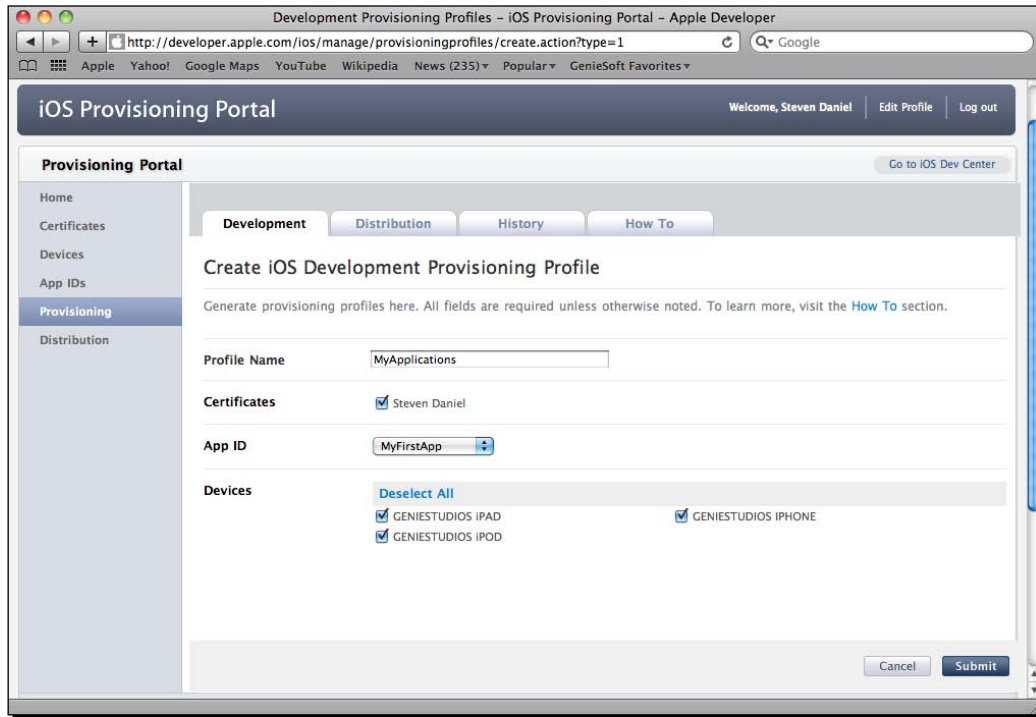
In order to create a provisioning profile, follow the simple steps below:

1. In the iPhone Provisioning Profile, click on the **Provisioning** tab and then click on the **New Profile** button:



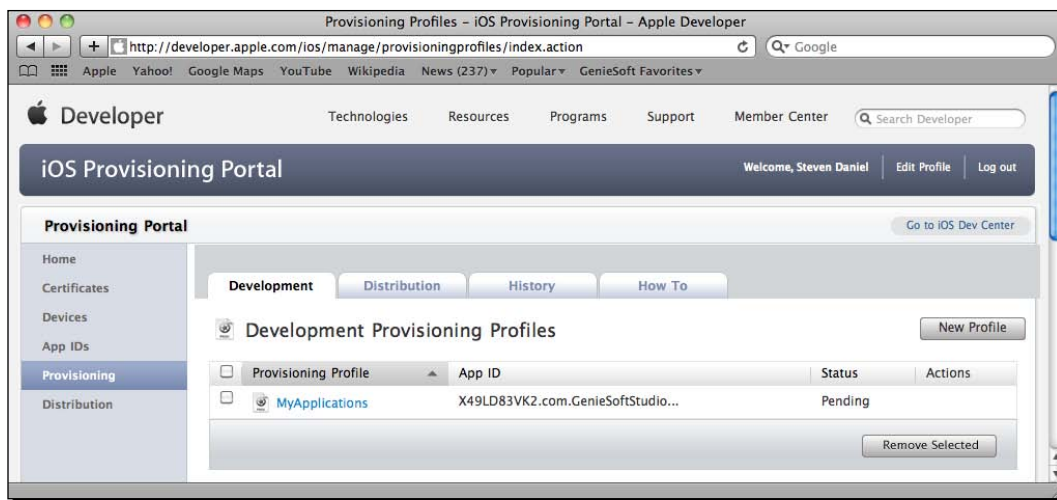
2. Once the **New Profile** button has been clicked, the provisioning profile screen appears which is shown in the screenshot below. This screen allows you to associate one or more development certificates with one or more devices, using the App ID so that you can install your signed iOS applications onto a real iOS device. From this screen, enter in **MyApplications** as the **Profile Name**, ensuring that you select all certificates that you would like to associate with this provisioning.
3. Select **MyFirstApp** as the **App ID** to use.

4. Finally, check all the devices that you would like to provision and then click on the **Submit** button once finished:

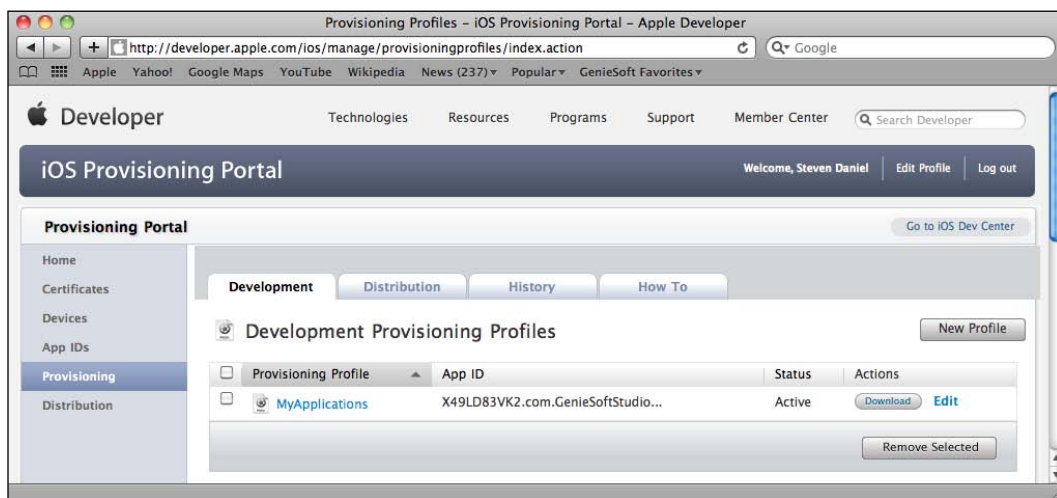


 You can choose to register additional devices using the iOS Provisioning Portal by clicking on the **Devices** tab and then follow the steps outlined in section *Registering devices for testing* located within this document.

At this point, the provisioning profile will be shown as pending approval status. This is shown in the screenshot below:



After a few seconds, you should see that the status changes from **Pending** to **Active**. If this does not happen, you may need to refresh your browser. At this point, you will be able to download your mobile provisioning file:



Click on the **Download** button to download your provisioning certificate profile. You will notice that when you download the mobile provisioning file, the file will be named as **MyApplications.mobileprovision**.

What just happened?

In this section, we looked at how to create a provisioning profile certificate so that a signed iOS application can be deployed to a real iOS device to see how it will perform in a real environment. We then saw how we can associate a development certificate with one or more devices using an Application (App ID) and Unique Device Identifier (UDID). To end the section, we learned how to download our mobile provisioning certificate. In the next section, we will look at how to deploy an application to an iOS device using this certificate.

Using the Provisional Profile to install an App on an iOS device

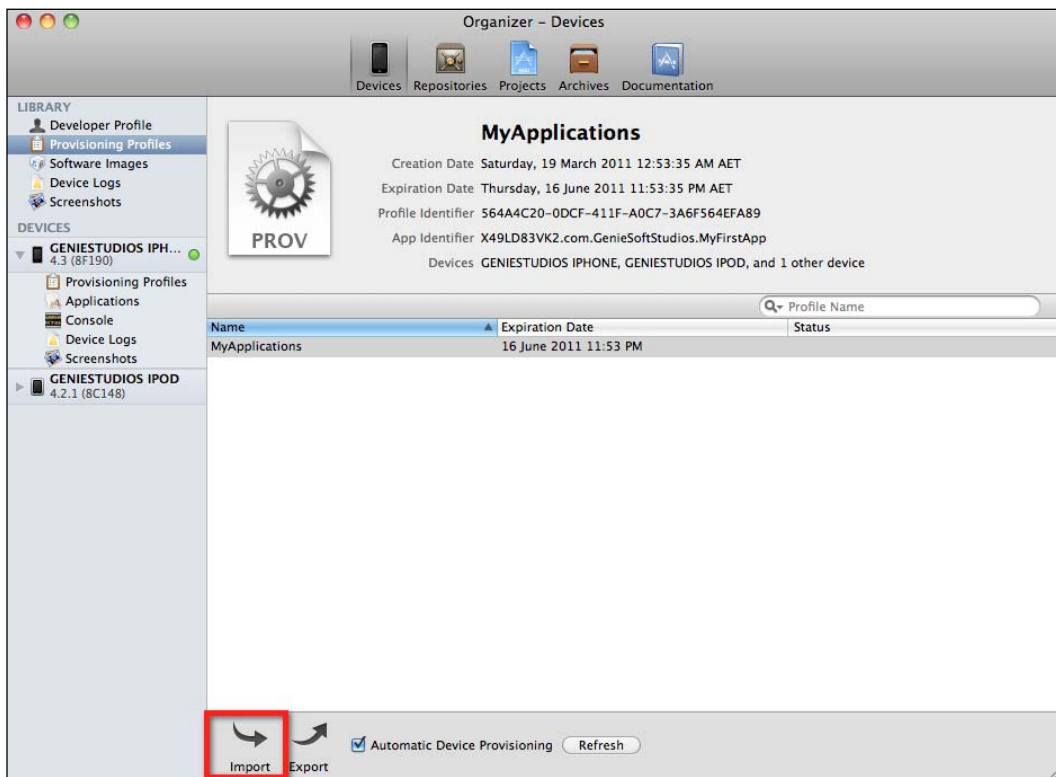
Now that we have generated our Development Certificate and have created our Provisioning Profiles, the final thing that we need to do is create a version of our application that can be deployed onto an iOS device. What we are going to do for this example is create a simple Open GL ES application. We won't be doing any coding as this application will run and show a multi-colored box bouncing up and down the screen.

Time for action – creating and deploying the app to an iOS device

Before we proceed with creating our `MyFirstApp` project, we must first launch the **Xcode** development environment. This can be located in the `/Developer/Applications` folder. Alternatively, you can use the spotlight to search for Xcode by typing **Xcode** into the search box window:

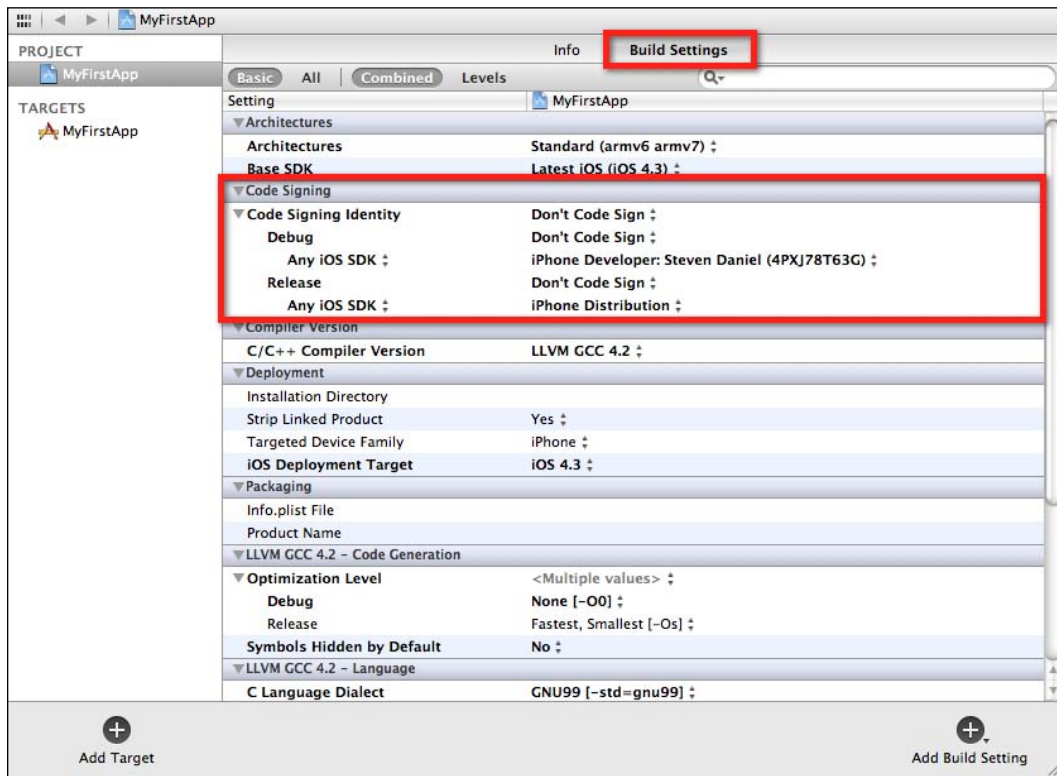
- 1.** Choose Create a new Xcode project, or **File | New Project**.
- 2.** Select the OpenGL ES Application template from the list of available templates.
- 3.** Click on the **Next** button to proceed to the next step in the wizard.
- 4.** Enter **MyFirstApp** as the name of the Product to create.
- 5.** Select **iPhone** from under the **Device Family** dropdown.
- 6.** Click on the **Next** button to proceed to the next step in the wizard.
- 7.** Specify the location where you would like to save your project.

8. Ensure that the **Create local git repository for this project** is unchecked from under the Source Control section.
9. Click on the **Create** button to continue and display the Xcode workspace environment.
10. The next step that we need to do is add the Mobile Provisioning profile that we created in the previous sections of this chapter. Open the Organizer Window by pressing *Shift + Command + 2*:

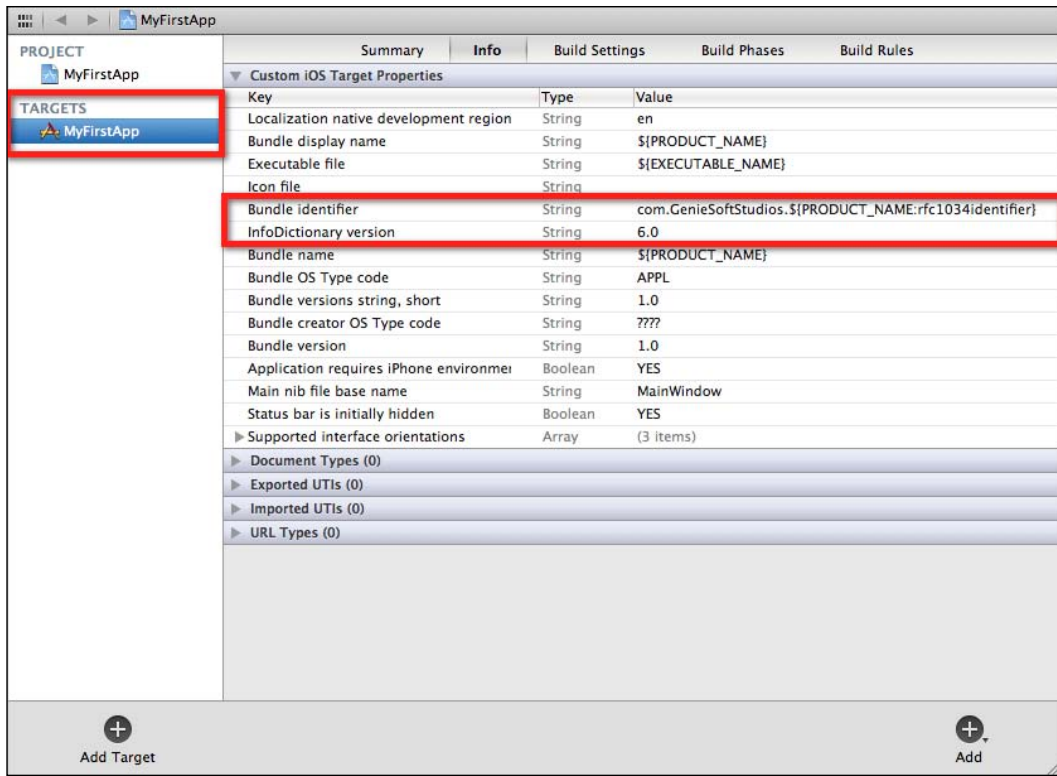


11. Click on the **Import** button and select the file **MyApplications.mobileprovision**.

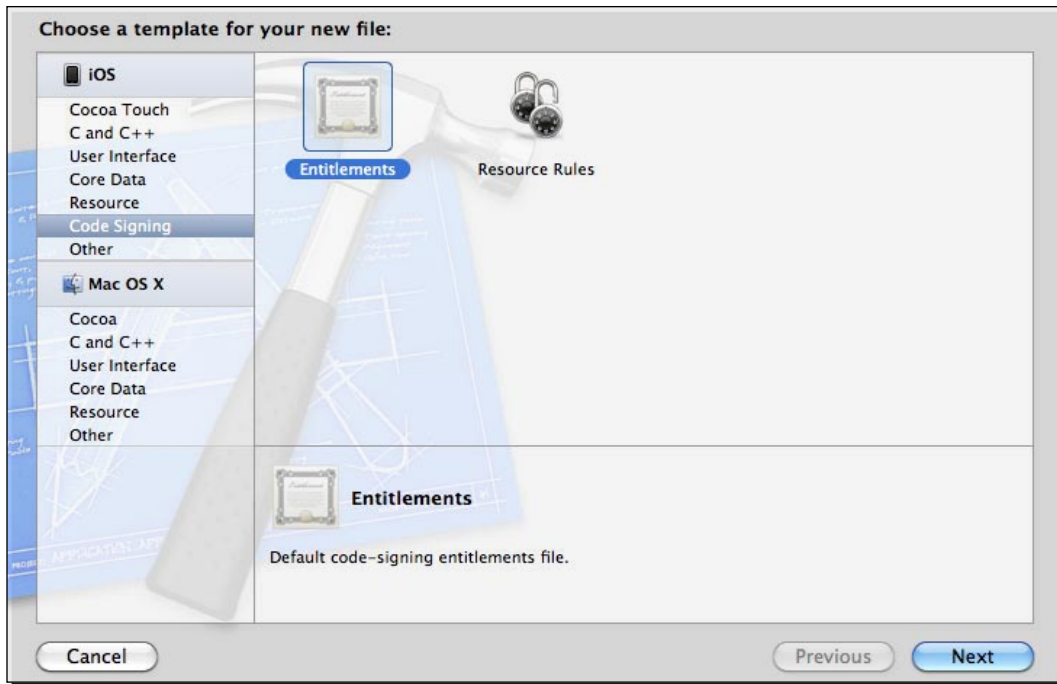
- 12.** Next, we need to associate our distribution certificate and provisioning profile with the build. From the **Build Settings** tab, select the **Code Signing** section, select **Any iOS SDK**, and choose your certificate from the drop-down list. Your certificate will be in bold, with your provisioning profile in gray. Without a valid certificate, you cannot deploy or upload your applications to the App Store:



- 13.** Next, display the **Info** tab so that you can specify the bundle identifier for your application. This bundle identifier needs to be the same as the App ID you used to register with the Developer Portal:



14. Now you can create an `entitlements.plist`. This file provides the code signing for the application. Choose **File | New | New File | Code Signing** and then select the **Entitlements** option:



15. Click on the **Next** button to proceed to the next step and save the file. Most people name the entitlement file **Entitlements.plist** for ease of use. This file is saved to the root folder of your application. This file has one property called **get-task-allow** and has the Boolean value of this property set to YES.
16. Change your Active Configuration to Release and Run your application to see if all works well. You will be asked to grant access to the certificate. Click **Always Allow**:



What just happened?

In this section, we created a simple OpenGL ES application and added our previously created Mobile Provisioning Profile that we created from the iOS Provisioning Portal. We also looked at how we go about Code Signing our application and creating an `Entitlements.plist` file in order for us to be able to deploy the project out to an iOS device.

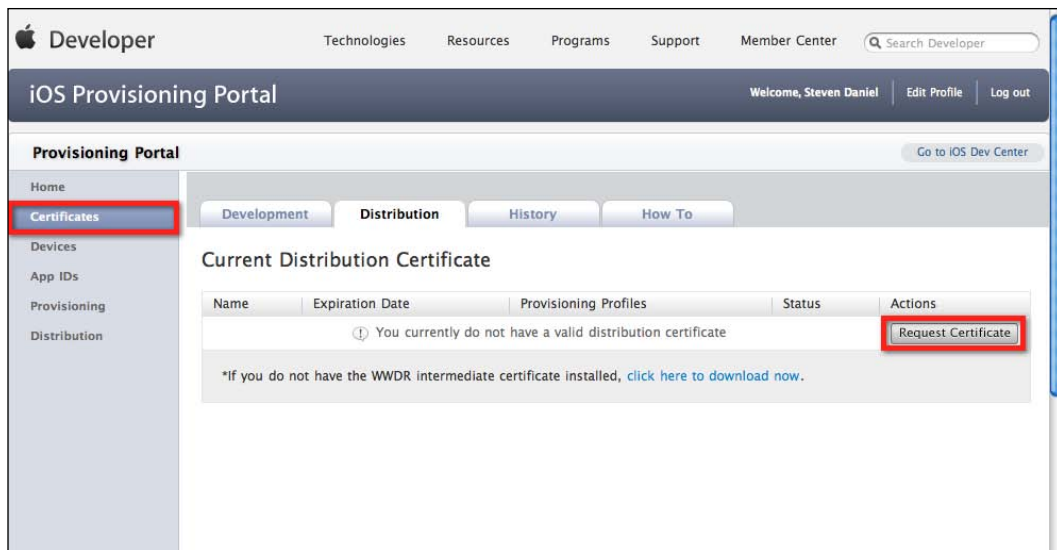
Getting a Distribution Certificate for your app

Before you can submit your application for Distribution, you will need to create a Distribution Certificate. This certificate enables the person responsible for submitting the final applications to the iTunes store, and is referred to as the Team agent. In order for the Team agent to submit any solutions, they must have an approved iOS Distribution Certificate. The section below, explains how to obtain this certificate.

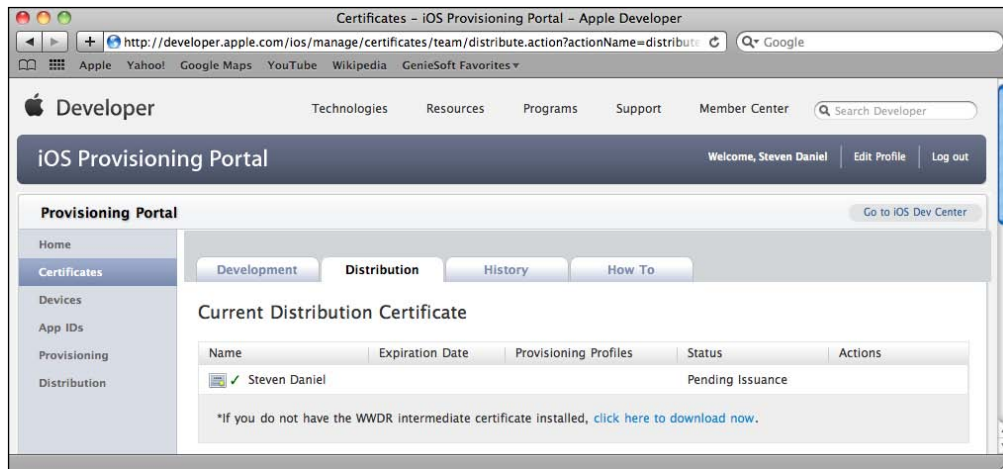
Time for action – getting the Distribution Certificate

In order to obtain the iOS Distribution Certificate for your application, follow these simple steps:

1. Log in to the iOS Provisioning Portal and click on the **Certificates** tab, which is located on the left-hand side of the panel and then click on the **Distribution** tab. The Current Distribution Certificate window appears, allowing you to request a certificate:



2. Click on the **Request Certificate** button. The **Create iOS Distribution Certificate** screen appears.
3. Click on the **Choose File** button and then select the certificate request file that you created in the previous section. Click on the **Submit** button once done. At this point, the provisioning profile will be shown as Pending Issuance status. This is shown in the screenshot below:



4. Our next step is to refresh the page and after a few seconds, the certificate will be ready and you will be able to download it. Once the certificate is downloaded, double-click on it to install it in the Keychain Access application as shown in the screenshot below:



What just happened?

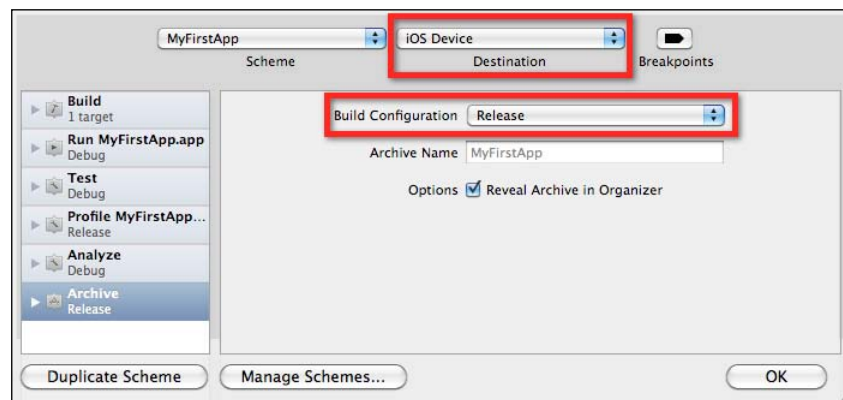
In the above section, we looked at how to request and obtain a distribution certificate in order to deploy our apps to the App Store. We looked at how we can use the iOS Provisioning Portal to create our distribution certificate from our Keychain which we created in the section *Getting an iOS Development Certificate*. Finally, we looked at how to download and install the certificate within our Keychain Access application.

Archiving and submitting Apps using Xcode 4

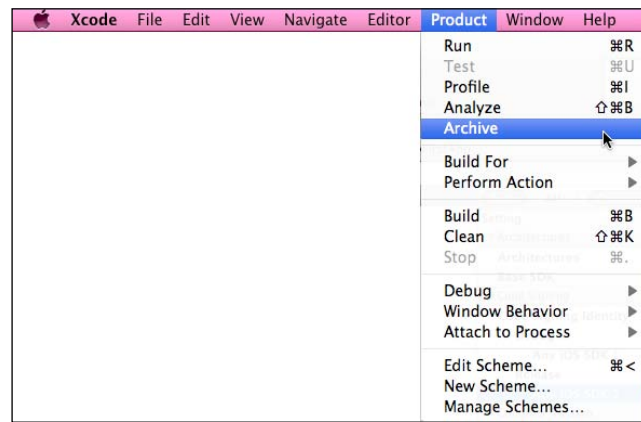
Before archiving your application, ensure that the binary is self-contained. What this means is that, if the binary relies on any static libraries, it ensures that those libraries are part of the application binary by setting the **Skip Install** build setting to **Yes** within the **Build Settings** section of the target that builds and archives the application:



Our next step is to make sure the Archive action of the scheme has the appropriate destination set for the type of application to archive. Ensure that the **Destination** has been set to use **iOS Device**, and the **Build Configuration** has been set to use the **Release** scheme:



Before submitting an application to the App Store, or sharing it with others, you create an application archive which will enable you to share your application archive with other developers and testers, or distribute it to users. Select the **Product | Archive** option from the Xcode menu and it will begin to create the application archive. Eventually, the archives organizer window will be displayed:



In order to have your application considered for inclusion on the Apple App store, you must submit the archive to iTunes Connect. This is to ensure that your application archive passes the essential iTunes Connect validation tests. Xcode can validate this for you before you submit it:





For more information on other ways of distributing your applications check out the provided link made available through the Apple Developer Connection website at: <http://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/Introduction/Introduction.html>.

iOS Human Interface Guidelines

When you begin creating iOS applications, it is important to keep in mind and follow the iOS Human Interface Guidelines document that Apple provides. This document describes the guidelines and principles that help you design a consistent user interface and user experience for your iOS app.

This document provides you with the guidelines needed for developing applications that run efficiently and effectively on the iOS platform. This involves considering the screen size of your window, the memory limitations, and the ease of use.

Other areas are covered to ensure the consistency of your application as you navigate from screen to screen, as well as principles for developing good user interfaces. This document also includes information on how to go about providing status updates when the network is down, or other feedback, through using messages if an error occurred or if a field cannot be left blank.

There is also information relating to the proper use and appearance of views and controls for Navigation and Toolbars, Alerts, Table Views, Buttons, and icons, as well as the creation of custom icons and images.



To obtain further information about what these guidelines consist of, it is worthwhile checking out the iOS Human Interface Guidelines document at the following location: <http://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/MobileHIG.pdf>.

Testing your application

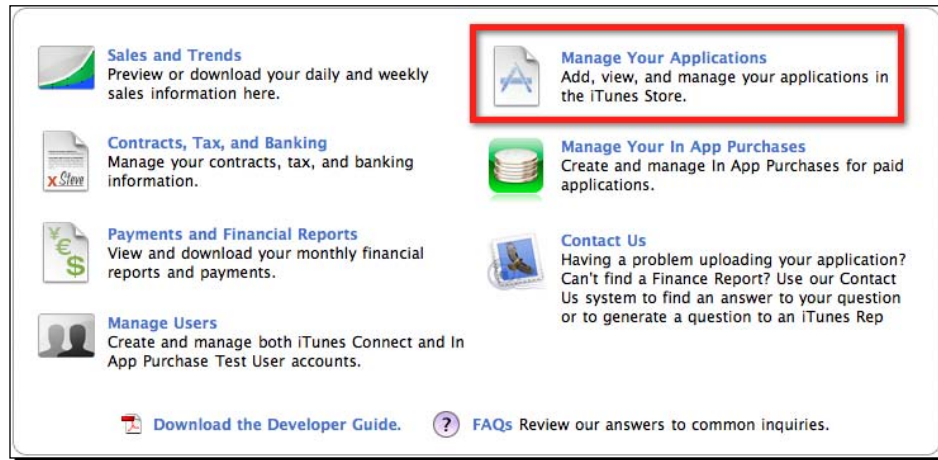
Before submitting your application for approval to the Apple App Store, you need to ensure that it works properly and is free from problems. The iOS simulator is a good place to start, and although not everything can be tested within the simulator, it proves a good starting point. Whilst your application may run perfectly within the simulator, problems may still exist when it has been deployed to the iOS device; it is always best to deploy it to a real iOS device running the latest OS 4.x.

You can also make use of the Instruments application to ensure that no memory leaks exist within your application and avoid having your app crash on the user's device. If your app crashes, it could also prevent your application from being successfully approved and being displayed on the App Store by Apple.

Preparing your App for submission through iTunes Connect

When you have tested your application to ensure that it all works and is free from problems and you have set up all of your accounts, you will want to start uploading your application to the App Store. Please bear in mind that only a release version of your application can be uploaded.

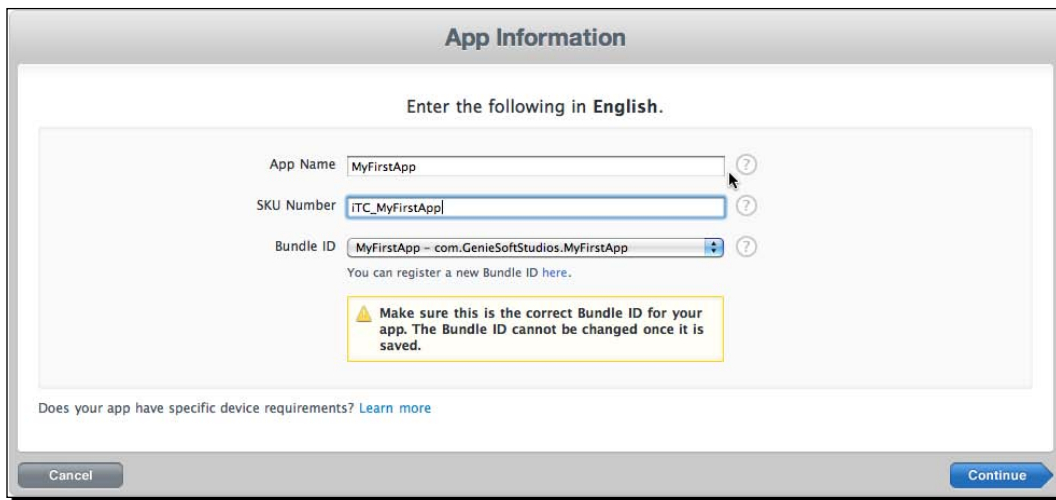
First, log into iTunes Connect and then click on the **Manage Your Applications** button as shown in the screenshot below:



Click on the **Add New App** link to begin adding your application to the App Store as shown in the screenshot below:



Next, we need to enter the Application details for the application we are uploading. Enter the Application details, and then click on the **Continue** button to proceed to the next step. The SKU Number is a unique identifier that you create for your app:



App Information

Enter the following in **English**.

App Name ?

SKU Number ?

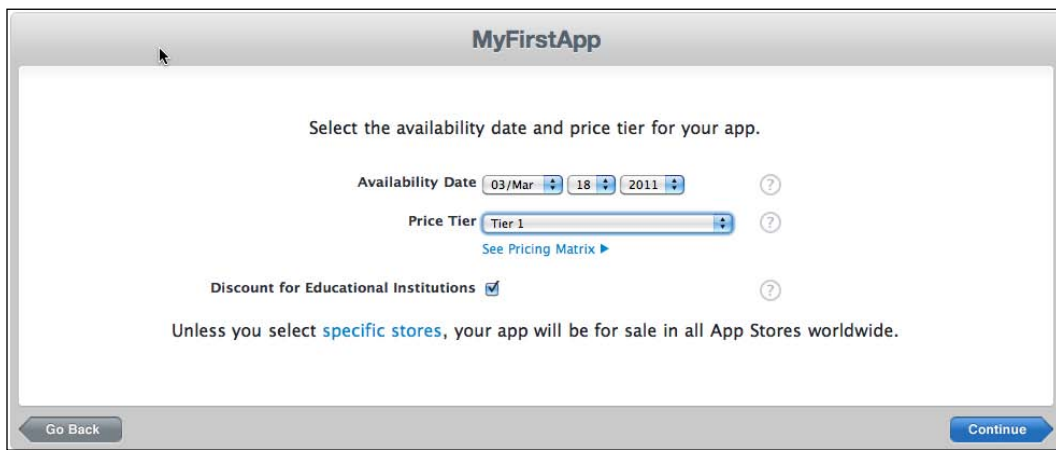
Bundle ID ?

You can register a new Bundle ID [here](#).

⚠ Make sure this is the correct Bundle ID for your app. The Bundle ID cannot be changed once it is saved.

Does your app have specific device requirements? [Learn more](#)

In the next step, we specify the availability date and pricing tier for when the application should be made available for download:



MyFirstApp

Select the availability date and price tier for your app.

Availability Date ?

Price Tier ?

[See Pricing Matrix](#) ▶

Discount for Educational Institutions ☒ ?

Unless you select [specific stores](#), your app will be for sale in all App Stores worldwide.

There are up to 85 pricing tiers to choose from, including an option for selling your application for free. The screenshot below shows a small snapshot of the pricing matrix that Apple provides:

App Store Pricing Matrix												
	U.S. – US\$		Canada – CAD		Europe – Euro		U.K. – GBP		Japan – Yen		Australia – AUS	
Tier	Customer Price	Your Proceeds	Customer Price	Your Proceeds	Customer Price	Your Proceeds	Customer Price	Your Proceeds	Customer Price	Your Proceeds	Customer Price	Your Proceeds
Tier 1	0.99	0.70	0.99	0.70	0.79	0.48	0.59	0.36	115	81	1.19	0.76
Tier 2	1.99	1.40	1.99	1.40	1.59	0.97	1.19	0.72	230	161	2.49	1.58
Tier 3	2.99	2.10	2.99	2.10	2.39	1.45	1.79	1.09	350	245	3.99	2.54
Tier 4	3.99	2.80	3.99	2.80	2.99	1.82	2.39	1.45	450	315	4.99	3.18

On clicking on the **Continue** button, you will be directed to the **Metadata** screen, where you are required to fill in the information pertaining to your application. The fields in that screen along with their description are listed in the table below:

SCREEN FIELD	DESCRIPTION
Version Number	This can be anything that you like. It is preferable to start at 1.0.
Description	The Application Description that can contain up to 4,000 characters.
Primary Category	These contain up to 20 different categories to choose from, including Games, Entertainment, Business, Books, and so on.
Secondary Category (Optional)	You can choose from a Secondary Category.
Keywords	These help return results faster when a customer is searching for an application within iTunes.
Copyright	The name of the person or entity that owns the exclusive rights to the app, preceded by the year the rights were obtained (for example, "2011 GenieSoft Studios").
Contact Email Address	An e-mail address where users can contact you if there are problems with your app.
Support URL	A URL that provides support for the app you are adding. This will be visible to customers on the App Store.
App URL (Optional)	A URL with information about the app you are adding. If provided, this will be visible to customers on the App Store.
Review Notes (Optional)	Additional information about your app and/or your in-app purchases. Review Notes cannot be longer than 4,000 bytes.

When Apple released their iOS 3.0 release, they included a rating scheme that allowed parents to control which applications their children could download. This screen is compulsory and you must complete this before you can submit your application. The age limit will change depending on how you go about rating your application. Ensure that you rate this correctly as Apple uses this information during their internal process and reviews how you score your application:

Rating

For each content description, choose the level of frequency that best describes your app.

[App Rating Details ▶](#)

Apps must not contain any obscene, pornographic, offensive or defamatory content or materials of any kind (text, graphics, images, photographs, etc.), or other content or materials that in Apple's reasonable judgment may be found objectionable.

Apple Content Descriptions	None	Infrequent/Mild	Frequent/Intense
Cartoon or Fantasy Violence	☐	☐	☐
Realistic Violence	☐	☐	☐
Sexual Content or Nudity	☐	☐	☐
Profanity or Crude Humor	☐	☐	☐
Alcohol, Tobacco, or Drug Use or References	☐	☐	☐
Mature/Suggestive Themes	☐	☐	☐
Simulated Gambling	☐	☐	☐
Horror/Fear Themes	☐	☐	☐
Prolonged Graphic or Sadistic Realistic Violence	☐	☐	☐
Graphic Sexual Content and Nudity	☐	☐	☐

Avoiding rejection of your App

As with any application, there are many reasons why your application may be rejected by Apple and can be as simple as the application crashes or does not conform to the iOS Human Interface Guidelines. The list below outlines the various possibilities to watch out for to avoid your application being rejected:

- ◆ Application does not run on the latest iOS.
- ◆ Application fails to comply with the guidelines outlined in the iOS Human Interface Guidelines.
- ◆ Has a similarity to existing features found in iOS applications.
- ◆ Functionality does not work as outlined.
- ◆ Application crashes on the iOS device.

- ◆ Differences between your application icons, that is, large and small icons are different.
- ◆ Failure to inform the user through messages, that is, Network Unavailable or if an error has occurred.
- ◆ Collecting of personal user data without receiving their permission.
- ◆ The use of Apple's private APIs (such as the built-in text-to-speech functionality) is strictly prohibited. If they determine that your application makes use of such functionality, it will be rejected immediately.



If you are interested in learning more about the different ways in which Apple can reject your application, you can check out the following link provided which breaks down each of these into the various categories: <http://developer.apple.com/appstore/resources/approval/guidelines.html>.

Pricing your app

How much you should charge for your application is a tough question for all developers and companies selling their applications on the App Store. One of the tactics that I have found that many developers have seemed to adopt is to start selling their application at \$6.99 and then, shortly after release, they temporarily drop the price by a couple of dollars, or sometimes offer 50-80% off the price for a limited time.

By reducing the price, this will create a surefire sale, encouraging people to purchase before the limited time expires which can also increase the number of sales for your Application within the iTunes App Store hence getting into the top 10 or even to number 1.

Adding your App to iTunes Connect

This section shows you how you can upload your application. Before this can happen, you will need to provide the iTunes artwork and application icon and have a screenshot of the application. Apple prefers that you create all artwork in the `PNG` file format. The final images should be 72 pixels per inch with no transparency or layers.

The artwork to be used as the main artwork for your application should be 512x512 pixels in size and exported in `PNG` file format, and must be named as **iTunesArtwork**, with no file extension present.

The icon used to distinguish your application on the iOS device will need to be 57x57 pixels, with no transparency or layers. To help identify your applications icon image, it is best to save this file as **icon.png**. Prior to iOS4, iPhone app icons were 57x57-pixel `PNG` files. For the iPhone 4's retina display, you need one that's double that size: 114x114 pixels.



For more information on creating high resolution icons, check out the Apple Developer Documentation at: http://developer.apple.com/library/ios/#documentation/xcode/conceptual/iphone_development/115-Configuring_Applications/configuring_applications.html.

As well as providing the iTunes artwork, you also need to provide up to four additional screenshots to show the different parts of the application each of 320x460 pixels in size, in order to allow the image to fit on your screen when you view the image via an iOS device or the App Store. Within iOS4 and the new iPhone 4's retina display, you can create application graphics that look sharp and crisp on the **new retina display**. You will need to create high-resolution versions of your application images. These are double the size (width x height) of the original images, and are named **imageName@2x.png**.



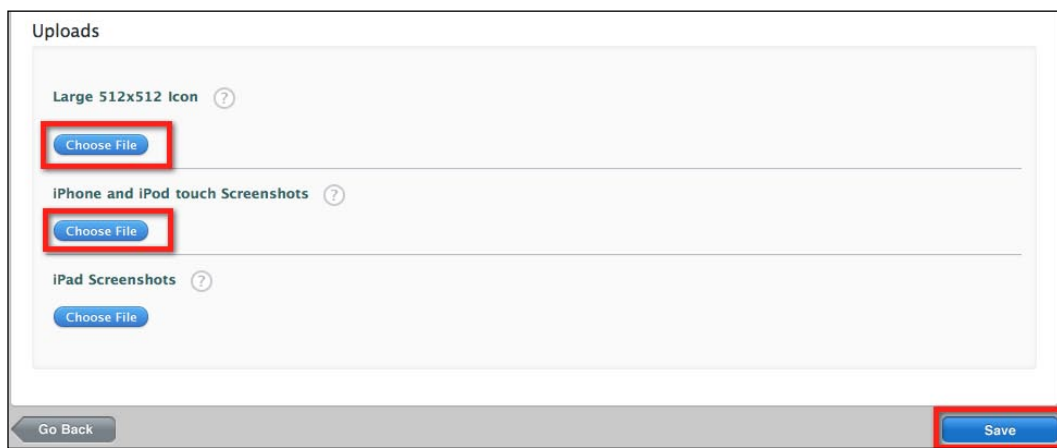
For more information on creating high resolution images, you can check out the Apple Developer Documentation at: <http://developer.apple.com/library/ios/#documentation/2DDrawing/Conceptual/DrawingPrintingiOS/SupportingHiResScreens/SupportingHiResScreens.html>.

Time for action – uploading the application icon and screenshot images

In the section below, we will take a look at how we can upload the application icon and screenshot images for our application. You will only need to add the ones highlighted by a red rectangle. To upload the images and icon files, follow these simple steps:

1. First, we need to add the icon image that will be used on the App Store. This is a large version of your app icon that will be used on the App Store. It must be at least 72 DPI and a minimum of 512x512 pixels (it cannot be scaled up). It must be flat artwork without rounded corners.
2. Navigate to the **iTunes Connect** page under the section **Uploads** and click on the **Choose File** button as shown under the **Large 512x512 icon** section heading.
3. Next, we need to add the iPhone screenshots and these must be .jpeg, .jpg, .tif, .tiff, or .png files, that is 960x640, 960x600, 640x960, 640x920, 480x320, 480x300, 320x480 or 320x460 pixels, at least 72 DPI, and in the RGB color space. Apple recommends that you include at least three screenshots with a maximum of five.

4. Click on the Choose File button, located under the **iPhone and iPod touch Screenshots**.
5. Finally, this option is purely for iPad development (*which this book does not cover*), basically for iPad screenshots. These must be .jpeg, .jpg, .tif, .tiff, or .png files, that is 1024x768, 1024x748, 768x1024 or 768x1004 pixels, at least 72 DPI, and in the RGB color space. If you wanted to add iPad screenshots, click on the Choose File button, located under **the iPad Screenshots** section:



The screenshot shows the 'Uploads' section of the Apple Developer portal. It contains three main sections for uploading assets:

- Large 512x512 Icon**: Includes a 'Choose File' button, which is highlighted with a red rectangle.
- iPhone and iPod touch Screenshots**: Includes a 'Choose File' button, which is highlighted with a red rectangle.
- iPad Screenshots**: Includes a 'Choose File' button.

At the bottom of the form, there are two buttons: 'Go Back' and 'Save'. The 'Save' button is highlighted with a red rectangle.

6. Once you have finished providing this information, click on the **Save** button to submit your changes.

What just happened?

In this section, we looked at the steps involved in uploading the images and icons for our application and learned about the different types of acceptable file formats expected by Apple, the resolution that these images need to be in, as well as the recommended number of images that Apple expects you to submit.

Apple takes up to a week to review your application before it appears on the App Store. You will receive numerous e-mails from Apple when your application changes state.

Using iTunes Connect to manage your Apps

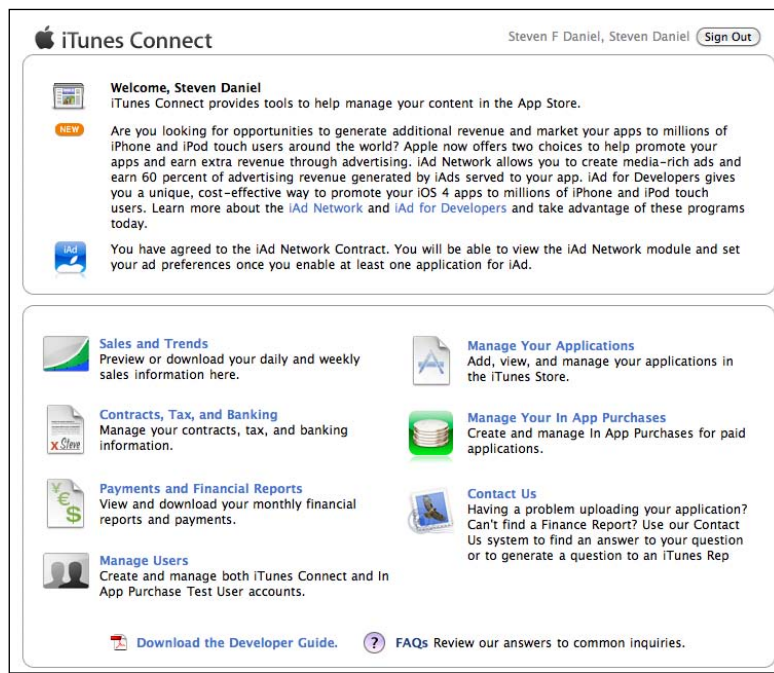
The iTunes Connect application is a fantastic tool, which you use to upload and manage your application in the Apple App Store. By using iTunes Connect, you can achieve the following, as described below:

- ◆ Manage and upload your applications to the App Store

- ◆ Assess sales/trends for each of your applications
- ◆ Review your contracts, tax, and banking information
- ◆ Download financial reports
- ◆ The ability to manage your users, for example, iTunes Connect Users or General Users
- ◆ Include an internal Application Store from within your application, which allows you to sell optional content for your app
- ◆ Request promotional codes to enable people that you know to get free copies of your applications
- ◆ Contact Apple support with any questions that you may have

 You will only be able to upload a release version of your application to iTunes Connect and the App Store.

The screenshot below shows the iTunes Connect tool which you can use to upload and manage your applications within the App Store:



Once you have logged into iTunes Connect, it is a good idea to ensure that your banking detail information is correct and up-to-date. Since you are selling something, you want to ensure that you are receiving the funds and getting paid. After you have validated and ensured that all of your banking information is up-to-date, you will need to set up accounts for those people in your company who will need access to iTunes Connect.

The iTunes Connect website allows three different user types that can be set up to enable access to iTunes Connect, in addition to the original person who set up the account (this is sometimes referred to as a Legal Account):

- ◆ The Admin account has the right to view, add, and delete additional accounts as well as managing the whole iTunes Connect environment
- ◆ The Finance accounts give the user access to all financial reports and contracts, as well as tax and banking information, and sales/trend report modules
- ◆ The Technical Account allows the user to manage applications and manage the user modules

Marketing and promoting your app

So, you have submitted and installed your application on the Apple App Store, and you are really excited to tell the world about your latest creation. Although you can make a lot of money from selling your application, you must also be willing to put in a bit of effort to promote your application.

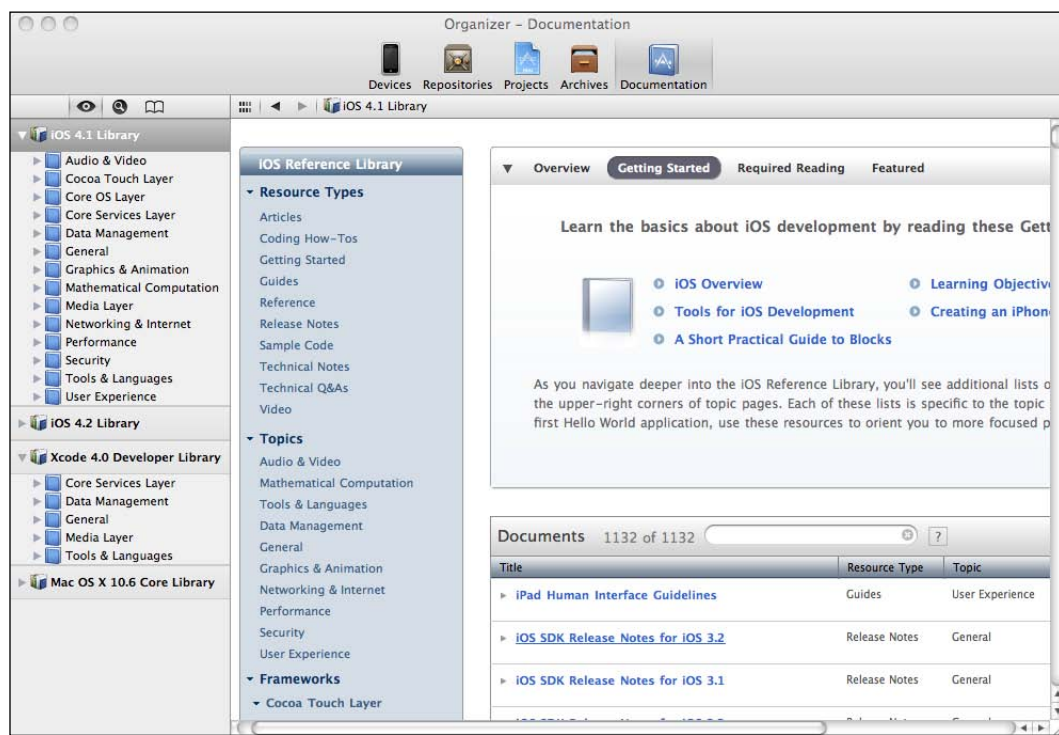
In order to sell any piece of software, using the traditional ways is always the most effective. Some of the ways are explained below:

- ◆ Using the Apple's iTunes Connect to monitor/manage sales
- ◆ Make use of websites, brochures, and flyers
- ◆ Networking at social gatherings
- ◆ Advertise in computer magazines, for example, Mac User
- ◆ Update your application and change your price often
- ◆ Creating and making use of application promotional codes to give to your testers

Using one or more of the items shown above, can help you gain an interest in your application and drive sales of your application.

iOS Developer Documentation

The Xcode development suite is a good place to start, and contains a great wealth of information for developing on iOS devices. This can be accessed from within the Xcode development environment. Select **Shift | Command | 2** to bring up the **Window Organizer** window and then click on the **Documentation** icon:



The Apple website contains a number of freely available and downloadable reference materials for the iPhone. If you require additional documentation, it can be found on the Apple Developer Connection website using the following URL: <http://developer.apple.com/index.html>.

Have a go hero – creating App IDs and submitting your App

Now that you have a good working knowledge on how to go about creating an application ID for your application, the task will be to build a deployable version of your application and send it to some of your closest friends to have them test it on their iOS devices. When you are confident that the application is good enough to be published, submit it to the iTunes Connect App Store:

1. You will need to build a full release version of your application. You can refer to the section *Build Configurations – debug to release* located within this chapter.
2. Next, you will need to obtain each person's UDID and add this to your provisioning profile. You can refer to the section(s) *Registering Devices for Testing*, *Creating Application IDs* and *Creating a Provisioning Profile*, again all located within this chapter.
3. When you are satisfied that the application is free from bugs and are ready to submit your application to the Apple App Store, you will need to create and obtain a Distribution Certificate. For help on this area, refer to the section *Getting a Distribution Certificate for your app*.

Once you have a working application, you will have mastered how to create application IDs for your application as well as know how to create Provisioning profiles and distribution certificates for submission to iTunes Connect and the Apple App Store.

Pop quiz – distribution of your App

1. What does iTunes Connect do?
 - a. Allows you to manage your application after you have uploaded it to the iTunes App Store
 - b. Enables you to connect to iTunes to play music
 - c. I don't know
2. Where would you find the UDID for your iOS device?
 - a. From the iOS Provisioning Portal
 - b. The Organizer application located under the **Window | Organizer** menu item
 - c. All of the above

3. What is the purpose of creating Provisioning Profiles?
 - a. It allows you to deploy and test your app on an iOS device
 - b. Enables you to have your app run on multiple devices
 - c. Allows you to give your a copy of your App to your friends to test
 - d. All of the above
4. Where can you modify the settings for iPhone icons?
 - a. This can be modified in the `plist` file and by adding the setting `UIPrerenderedIcon` and setting the option to `True`
 - b. This can be done via the iOS Provisioning portal
 - c. I don't know
5. When building your application, what do we mean by *release*?
 - a. Releases your application to the world and breaks free
 - b. The release step builds a version of your application with the extension `.app` that will be sent to the iTunes App Store or ad hoc via e-mail
 - c. All of the above

Summary

In this final chapter, we looked at the steps required to submit applications to the iTunes App Store and how we can use Xcode to create separate build configurations for debug and release and how we go about signing up to the iOS Development program.

We also looked at how we go about creating provisioning profiles, Application IDs, and how to register devices required for testing. To finish up, we looked at what are the best ways to go about pricing out Apps, and the ways in which we can avoid our application from being rejected by Apple by following the iOS Human Interface Guidelines.

We have now come to the end of the book. I hope you have enjoyed reading it as much as I did writing it. This is certainly not the end of the road for you. There is a lot of stuff to explore in the world of Xcode and iPhone. Don't worry; you won't be on your own. There are many developers out there who are more than willing to help you in case you get stuck at any point of time at <http://developer.apple.com/devforums/>.

Good luck with your Xcode journey. I hope to see your app on the Apple App Store soon!

Pop Quiz Answers

Chapter 3

Actions and Rotatable Interfaces

1.	2.	3.
d.	a., b., c., d., e.	a.

Chapter 4

Core Data/Media Playback and Core Location

1.	2.	3.	4.	5.	6.
c.	b., d.	b.	c.	a.	b.

Chapter 5

Table Views/repositioning Controls

1.	2.	3.	4.
b.	c.	c.	b.

Chapter 6

Frameworks

1.	2.	3.
b.	d.	d.

Activity Indicators

1.	2.	3.
c.	b.	c.

Alert Dialogs and Button Indexes

1.	2.
c.	c.

Sounds and vibrations

1.	2.	3.
c.	c.	c.

Chapter 7

Tap counts

1.	2.
a.	c.

Tracking and identifying swipes

1.	2.
d.	c.

Pinches and transformations

1.	2.	3.
c.	b.	b.

Motion events

1.	2.
a., d., e.	c.

Sensing orientation

1.	2.
c.	b.

Chapter 8

All about debugging projects

1.	2.	3.	4.	5.
b.	a.	c., d.	b.	a., c.

Chapter 9

Subversion/Version Editor

1.	2.	3.	4.	5.	6.
b.	c.	c.	c.	c.	d.

Chapter 10

Playing with Instruments

1.	2.	3.	4.	5.
c.	c.	a., c., d.	a.	a., b.

Chapter 11

Distribution of your App

1.	2.	3.	4.	5.
a.	b.	a.	a.	b.

Index

Symbols

- #import statement 21
- + Add Attribute button 106
- + Add Entity button 106
- .h file type 20
- [moviePlayerController play] statement 129
- @implementation compiler directive 22
- @implementation directive 22
- @interface directive 21
- @private, access privilege 22
- @property directive 71, 153, 154
- @property method 46
- @protected, access privilege 22
- @public, access privilege 22
- @synthesize directive 153, 154

A

- accelerometer
 - didAccelerate method 249
- accelerometer, exploring 249
- AccelGyroExample application 258
- AccelGyroExampleViewController.h
 - implementation file 255
- AccelGyroExampleViewController.h interface file 258, 259
- AccelGyroExampleViewController.m
 - implementation file 256, 257
- Actions (IBActions) 145
- action sheet button presses
 - responding to 215-217
- actionSheet property 215

action sheets

- alerts, handling via sounds 217-220
- alerts, handling via vibrations 217-220
- appearance, changing 220
- associating, with view 214
- customizing 217
- Display Action Sheet method, implementing 214
- items, adding 220
- parameters 215

action sheets, parameters

- cancelButtonTitle parameter 215
- delegate parameter 215
- destructiveButtonTitle parameter 215
- initWithTitle parameter 215
- OtherButtonTitles parameter 215

active/inactive breakpoints, unified navigation UI 54

activityindicator object 209

activity monitor 332, 342

Add New User button 352

Add New User option pane 352

AddressBook.Framework 101

address book component 13

address book identities

- assigning, within organizer 319-321

AddressBookUI.Framework 101

afterDelay property 209

alert dialog button presses

- additional buttons, adding 213
- events, creating 213
- responding to 211, 213

alerts

- AudioToolbox Framework, adding to application 203, 204
- dialog button presses, responding to 211-213
- generating 202
- GetUsersAttention application, creating 202, 203

alerts, handling

- via sounds 217-220
- via vibrations 217-220

alerts component 15

alertView property 211

allocations 332, 342

alloc keyword 22

APIs

- using, in iPhone development 102

app

- adding, to iTunes Connect 382
- archiving, XCode 4 used 375-377
- icon, uploading 383, 384
- managing, iTunes Connect used 384, 386
- marketing 386
- preparing for submission, through iTunes Connect 378-381
- pricing 382
- promoting 386
- rejection, avoiding 381, 382
- screenshot images, uploading 383, 384
- submitting, XCode 4 used 375-377
- testing 377

AppKit.Framework 101

Apple Developer Documentation

- URL 271, 383

Apple Human Interface Guidelines

- URL 78

Apple website

- URL 8

application. *See* app

Application (App ID) 368

applicationDidFinishLaunching\$ method 74

application ID's

- creating 362-364

application life cycle

- about 73, 74

application status bar / activity window, Xcode 4 workspace environment 49

Application ToolBar, Xcode 4 workspace environment 48, 49

applicationWillTerminate\$ method 74

App Status, notification 354

App URL (Optional) 380

arrVegetables array 161

Assistant button 46

audio file

- playing 118, 119

AudioServicesPlaySystemSound method 218

AudioToolBox.Framework 101

AudioToolbox Framework

- adding, to application 203, 204

AudioUnit.Framework 101

automated testing, instruments 343

automation 332, 342

AVAudioSession class

- about 117

- application, creating to play audio file 119

- audio file, playing 117-119

- AVAudioPlayer, framework 118

- AVAudioRecorder, framework 118

- MoviePlayer application, creating 125-131

- movie playing, media player used 118, 125

- MusicPlayer application, creating 119-124

AVFoundation.Framework 101

AV foundation frameworks 117

B

backgroundColor method 231

behaviors preference pane, Xcode workspace preferences 61

blank template 332

bonjour component 12

breakpoints

- debugging with 272

- disabling 273

- editing 272

- navigator, revealing 273

BSD (Berkeley Standard Distribution) component 12

button controls 192

buttons

- applications, creating 193-195

- using 193

C

- calculatePurchase method** 152
- cancelButtonTitle, alert parameter** 211
- cancelButtonTitle parameter** 215
- case statements** 231
- cellForRowAtIndexPath method** 198
- central search interface, unified navigation UI**
 - listing, in project 52, 53
- certificates component** 12
- Certificate Signing Request (CSR)** 355
- CFNetwork.Framework** 101
- CFURLRef parameter** 219
- CGAffineTransformRotate function** 241
- CGPoint class** 241
- CGRect object** 46
- CGRect variable** 241
- clickedButtonIndex method** 211, 215
- CLLocationManagerDelegate protocol** 133
- CLLocationManager instance** 131
- CLLocationManager method** 131
- CLLocation method** 131
- CLLocation object** 131
- CMAccelerometerData, CORE MOTION CLASS** 250
- CMAttitude, CORE MOTION CLASS** 250
- CMDeviceMotion, CORE MOTION CLASS** 250
- CMGyroData, CORE MOTION CLASS** 250
- CMMotionManager, CORE MOTION CLASS** 250
- CMMotionManager class** 259
- Cocoa**
 - about 16
 - and Cocoa touch, differences 16
 - design patterns 16
 - Model-View-Controller (MVC) design pattern 16
- cocoa-Accelerometer component** 15
- cocoa-touch layer, iOS architecture layer**
 - Accelerometer component 15
 - alerts component 15
 - controllers component 15
 - Geographical component 15
 - Gyroscope component 15
 - image picker component 15
 - Localization component 15
 - multi-touch controls component 15
 - multi-touch events component 15
 - people picker component 15
 - view hierarchy component 15
 - web views component 15
- Cocoa touch**
 - and Cocoa, differences 16
- code assistants, unified navigation UI** 55
- code completion**
 - about 286
 - accepting, steps 286-288
- code editor**
 - debugging features 275, 276
- code editor window, Xcode Debugger** 264
- code snippets library, unified navigation UI** 57, 58
- collections component** 13
- components**
 - making, to work 81
- console output window, Xcode Debugger** 265
- Contact Email Address** 380
- Continue button** 352, 353
- contract, notification** 354
- Controller** 17, 145
- controllers component** 15
- control objects**
 - binding 82, 84
- controls**
 - adding, to user interface 67
 - adding, to view 205-207
 - relocating, within view on rotation 80, 81
 - repositioning 84, 86
- Copyright** 380
- core animation** 342
- core application architecture**
 - layer 72
- CoreAudio.Framework** 101
- core data**
 - example 141
- CoreData.Framework** 101
- core data application**
 - + Add Attribute button 106
 - + Add Entity button 106
 - application delegate and view controller, connecting 111
 - CoreDataExampleAppDelegate.h interface file 108

- CoreDataSample.xcdatamodel file 106
- coreDataViewController.h interface file 112
- coreDataViewController.m implementation file 113
- coreDataViewController class 110
- creating 104
- dealloc() method 116
- Next button 105
- Save button 106
- saveData() method 114-116
- searchData() method 115
- setValue method 115
- valueForKey method 116
- viewController object 109
- viewDidLoad() method 116
- viewDidUnload method 116
- With XIB for user interface option 107
- CoreDataExampleAppDelegate.h interface file 108**
- core data frameworks 103, 104**
- CoreDataSample.xcdatamodel file 106**
- coreDataViewController.h interface file 112**
- coreDataViewController.m implementation file 113**
- coreDataViewController class 110**
- CoreFoundation.Framework 101**
- CoreGraphics.Framework 101**
- CoreLocation.Framework 101**
- core location component 13**
- core location framework**
 - about 131
 - adding, to project 132, 133
 - application, making location aware 131, 132
 - CLLocationManagerDelegate protocol 133
 - description method 134
 - didFailWithError method 135
 - didUpdateToLocation method 134, 135
 - locationManager:didUpdateToLocation method 134
 - NSLog function 134
 - stopUpdatingLocation method 134, 135
- core motion framework 249**
- core OS layer, iOS architecture layer**
 - about 11
 - bonjour component 12
 - BSD (Berkeley Standard Distribution) component 12
 - certificates component 12
 - file system component 12
 - keychain component 12
 - Mach 3.0 component 12
 - OS X Kernel component 12
 - power management component 12
 - security component 12
 - sockets component 12
- core services layer, iOS architecture layer**
 - address book component 13
 - collections component 13
 - core location component 13
 - file access component 13
 - net services component 13
 - networking component 13
 - preferences component 13
 - SQLite component 13
 - threading component 13
 - URL Utilities component 13
- count property 162**
- CPU sampler 342**
- Custom Picker project**
 - controls binding, actions used 188-191
 - controls binding, outlets used 188-191
 - creating 186-188

D

- DashCode component 8**
- data, unified navigation UI**
 - debugging, with compressionable stack traces 54
- data hiding 17**
- dataOfType method 95**
- DatePickersExampleViewController.m implementation file 184**
- dealloc function 44**
- dealloc() method 116, 179, 184, 240, 241, 258**
- debugger toolbar, Xcode Debugger 262**
- DebuggingExample project, Xcode Debugger**
 - breakpoints, debugging with 272, 273
 - creating 266, 267
 - debugging 268
 - errors, handling 268
 - fix-it, using to correct code 270
 - LLVM compiler, setting up 270, 271
 - logic errors 269, 270

- NSLog, using to track changing properties 273-275
- running 268
- runtime errors 269
- syntax errors 269
- delayFactor**
 - adjusting 235
- delegate, alert parameter 211**
- delegate parameter 215**
- Description 380**
- description method 134**
- design pattern 16, 144**
- destructiveButtonTitles parameter 215**
- devices**
 - registering, for testing 360-362
- device tilting**
 - AccelGyroExample project, creating 254-258
 - detecting 254
- didFailWithError method 135**
- didUpdateToLocation method 134, 135**
- disassembly window, Xcode Debugger 263**
- Display Action Sheet button**
 - implementing 214
- displayActionSheet method 216, 220**
- displayAlertDialog method**
 - about 210-213
 - implementing 210, 211
- distanceBetweenPoints function 241**
- distanceBetweenPoints method 238**
- distributed builds, Xcode workspace preferences 63**
- Document-based application**
 - creating 90-93
 - file, loading 95-97
 - file, saving 95-97
- documentation, Xcode workspace preferences 62**
- Dog class 18**
- doGyroRotation function 259**
- doubleValue method 282**
- Download button 367**

E

- energy diagnosis, instruments 344**
- energy diagnostics 342**
- Enroll Now button 348**

errors

- handling 268
- logic errors 269, 270
- runtime errors 269
- syntax errors 269

events

- activity indicators 210
- creating 207
- Display Alert Dialog method, implementing 210, 211
- second activity indicator, adding 209, 210
- Show Activity Indicator method, implementing 207-209

ExternalAccessory.Framework 101

F

FavoriteColor application

- creating 155, 156

FavoriteColorViewController.h interface file 158

file

- versions, comparing 311, 312

file access component 13

file activity 332, 342

files, tracking

- repository organizer used 315, 316

files, unified navigation UI

- listing, in project 50, 51

file statuses, Xcode

- features 308-311

file system component 12

file templates library, unified navigation UI 56

financial report, notification 354

fix-it method

- about 261, 288

- LLVM compiler, setting up 270, 271

- using, to correct code 270

fonts & colors preference, Xcode workspace preferences 61

Foundation.Framework 101

framework APIs

- calendar access 140

- core data, example 141

- In-App SMS 140

- photo library access 140

- quick look 140

frameworks

- about 100, 102
- using, in iPhone development 102

G

GameKit.Framework 102

general button, Xcode workspace preferences 61

generateTextView method 196, 197

Geographical component 15

getter 153

GetUsersAttention application 203

creating 202, 203

GetUsersAttentionViewController.h interface file 207

GetUsersAttentionViewController.m implementation file 212

Git

- address book identities, assigning within organizer 319-321
- new Xcode project, creating 318, 321
- project, adding to subversion repository 321
- using, to manage multiple projects 317

GUI (Graphical User Interface) 16

gyroscope, exploring 249

Gyroscope component 15

H

hasOrientationChanged method 253

header files 20

HelloXcode4_GUI application

creating 67-69

HelloXcode4_GUI application, architecture

about 70

HelloXcode4_GUIAppDelegate.h 71

HelloXcode4_GUIAppDelegate.m 71

Main.m 70

hideKeyboard method 88

hideWhenStopped property 209

I

IB

- about 66
- controls, adding to user interface 67
- HelloXcode4_GUI application, architecture 70

- HelloXcode4_GUI application, creating 67-69
- used, for implementing MVC 145

icon.png 382

id directive 153

if-else clause 283

imageName@2x.png 383

image picker component 15

implementation files 20

initWithFormat function 84

initWithTitle, alert parameter 211

initWithTitle parameter 215

inspection range control 328

instruments

- about 326
- adding 335, 339, 340
- components 342, 343
- configuring 340, 341
- features 327
- inspection range control 328
- InstrumentsExample project, creating 329, 330
- instruments library, using 335, 336
- library button 328
- locating, with library 336-338
- loop button 327
- memory leaks, fixing 328, 329
- pause/resume button 327
- project, profiling 330-334
- project, running 330-334
- record/stop button 327
- removing 339, 340
- search field 328
- target menu 328
- time/run control 328
- view control 328

instruments, in Xcode 4

- adding, to project 345
- automated testing 343
- energy diagnosis 344
- iPhone graphics performance 344
- performance and power analysis 343
- time profiler 344

instruments, template

- activity monitor 332
- allocations 332
- automation 332
- blank template 332
- file activity 332

- leaks 332
- threads 332
- time profiler 332
- zombies 332
- instruments, types**
 - activity monitor 342
 - allocations 342
 - automation 342
 - core animation 342
 - CPU sampler 342
 - energy diagnostics 342
 - file activity 342
 - leaks 342
 - OpenGL ES Driver 343
 - system usage 343
 - threads 343
 - time profiler 343
- instruments component 9**
- InstrumentsExample project**
 - creating 329, 330
- instruments library**
 - using 335, 336
 - view icons, view mode types 336
 - view icons and descriptions, view mode types 336
 - view icons and labels, view mode types 336
 - view small icons and labels, view mode types 336
- Integrated Development Environment (IDE) 10**
- 9**
- Interface Builder. *See* IB**
- Interface Builder component 9**
- interface files 20**
- IOKit.Framework 102**
- iOS4 SDK**
 - features 28
- iOS applications**
 - developing, MVC design used 144
- iOS architecture, layers**
 - about 11
 - Cocoa services layer 13
 - Cocoa-touch layer 14, 15
 - core OS layer 11, 12
 - core services layer 13
 - media layer 14
- iOS cocoa-touch layer**
 - URL 15
- iOS developer**
 - URL 8
- iOS Developer Documentation**
 - about 387
 - app, submitting 388
 - app IDs, creating 388
- iOS Developer Program 357**
- iOS development certificate**
 - getting 357-360
 - request, generating 354-357
- iOS distribution certificate**
 - obtaining, steps 373-375
- iOS Human Interface Guidelines 377**
- iOS native maps application 139**
- iOS SDK (software development kit)**
 - about 8
 - installing 33, 34
 - URL 32
 - Xcode developer tools, removing 35
- iPhone application**
 - enhancing 87
 - HelloXcode4 example, hiding 88, 89
 - keyboard, hiding 87, 88
 - writing 38
- iPhone Developer Program 348, 349**
- iPhone development**
 - APIs, using 102
 - frameworks, using 102
- iPhone development team**
 - setting up 349
 - setting up, steps 350-354
- iPhone graphics performance**
 - tracking, OpenGL ES Driver used 344
- iPhone OS4 SDK, features**
 - apps folder 29
 - game center 29
 - iAd 29
 - multi-tasking 28
- iPhone SDK core components**
 - about 8
 - UserDefaults component 8
 - instruments component 9
 - interface builder component 9
 - iPhone Simulator component 8
 - Xcode component 8

iPhone Simulator

about 8, 10
features 25

isEqualToString method 213

isOn property 176

issues, unified navigation UI
tracking 53

issues navigator
viewing 284

items

adding, to existing repository 303

iTunes Connect

app, adding 382
app, preparing for submission 378-380
using, to manage apps 384-386

iTunes Connect User option 352

J

jump bar, unified navigation UI
collection 55

K

key bindings, Xcode workspace preferences 62
keychain component 12
Keywords 380

L

labelOutput string 229
labels 193
landscape left, orientation method 79
landscape right, orientation method 79
lblColor control 157
leaks 332, 342
library button 328
LLVM (Low Level Virtual Machine) 9
LLVM Compiler 2.0, unified navigation UI 55
Localization component 15
local repositories 298
local subversion repository
setting up 298-300
locationManager
didUpdateToLocation method 134
locations, Xcode workspace preferences
about 62

archives location 63
build location 63
derived data 62
snapshots location 63

logic errors 269, 270

Log Mode

using, to list revisions chronologically 314

logs, unified navigation UI

logs, unified navigation UIcollection 55

loop button 327

M

Mach 3.0 component 12

main function 40, 73

MainWindow.xib file 71, 72

managed object 104

managed object context 104

managed object model 104

Manage Schemes... option 348

map kit framework

about 135, 136
simple geographical application, creating
136-139

mapView object 138

media layer, iOS architecture layer

about 14
audio mixing component 14
audio recording component 14
core animations component 14
core audio component 14
image formats component 14
OpenGL component 14
OpenGL ES component 14
PDF component 14
quartz component 14
video playback component 14

media library, unified navigation UI 59

media player

used, for playing movie 125

MediaPlayer.Framework 102, 203

memory leaks

detecting 281, 282
fixing 328

message, alert parameter 211

MessageUI.Framework 102

MKMapTypeHybrid, map type constant 139

- MKMapTypeSatellite**, map type constant 139
- MKMapTypeStandard** map type constant 139
- MobileCoreServices.Framework** 102
- Model** 17, 145
- Model-View-Controller**. *See* MVC
- motionBegan**
 - motion:withEvent\$ method 242
- motionCancelled**
 - motion:withEvent\$ method 242
- motionCancelled** event 246
- motionEnded**
 - motion:withEvent\$ method 242
- MoviePlayer** application
 - creating 125-131
- moviePlayerController** object 129, 130
- MoviePlayer** project 125
- MoviePlayerViewController.m** implementation
 - file class 127
- MPMediaItemCollection** framework 119
- MPMediaItem** framework 118
- MPMediaPickerController** framework 118
- MPMoviePlayerController** class 131
- MPMoviePlayerController** framework 118
- MPMoviePlayerController** object 129
- MPMoviePlayerController** pointer 130
- MPMusicPlayerController** framework 119
- multi-touch** controls component 15
- multi-touch** events component 15
- multiple** projects
 - managing, Git used 317
- MultiTouch** architecture 224-226
- MultiTouch** gesture 236
- MusicPlayer** application
 - creating 119-125
- MVC**
 - about 16, 17, 145
 - controls building, actions used 147-152
 - controls building, outlets used 147-152
 - design pattern 144
 - implementing, interface builder used 145
 - implementing, XCode used 145
 - input field, declaring as property 154
 - Pizza order application, building 145-147
 - view controllers, implementing 152, 153
 - views, implementing 152
- MyDocument.h** interface file 92

N

- navigation-based** applications 168-169
- net** services component 13
- networking** component 13
- New Profile** button 365
- Next** button 105
- notification** methods
 - exploring 201, 202
- NSArchiver** class 97
- NSAttributedString** class 96
- NSDateFormatter** object 185
- NSLog**
 - using, to track changing properties 273-275
- NSLog** function 134, 273, 275
- NSLog** statement 270
- NSObject** class 21
- NSString** argument 274
- NSString** buttonTitle 213, 217
- NSString** variable buttonText 216
- NSUnarchiver** class 97
- numberOfRowsInSection** method 198
- numberOfSectionsInTableView** method 165, 198

O

- Object-Oriented** programming
 - about 17
 - data hiding 17-19
- object** controls
 - adding, to view 74-78
- Objective-C**
 - .h, file type 20
 - .m, file type 20
 - about 20
 - directives 21
 - files, type 20
- Objective-C** classes
 - @implementation directive 22
 - @interface directive 21
 - about 21
 - access privileges 22
 - instantiation 22
- object** library, unified navigation UI 58
- object** model
 - managed object 104

- managed object context 104
- managed object model 104
- ofType method 219**
- OpenAL.Framework 102**
- OpenGL ES Driver 343**
 - used, for tracking iPhone graphics performance 344
- orientation**
 - sensing 250, 253
- OrientationExample application**
 - modifying 253
- OrientationExample project 253**
 - creating 250-253
- OS X Kernel component 12**
- OtherButtonTitles, alert parameter 211**
- OtherButtonTitles parameter 215**
- otherButtonTitles property 213, 248**
- Outlets (IBOutlets) 145**

P

- past check-ins**
 - checking, Track Blame used 313
- pause/resume button 327**
- payment, notification 354**
- people picker component 15**
- performance and power analysis, instruments 343**
- pickers**
 - about 181
 - controls binding, actions used 183-185
 - controls binding, outlets used 183-185
 - Custom Picker project, creating 186-188
 - custom pickers 186
 - Date Picker project, creating 182
 - date pickers 181, 182
- pinches**
 - detecting 236
 - multiple fingers, handling 241
 - PinchExample project, creating 236-240
- PinchExample application 240**
- PinchExample project**
 - creating 236-240
- PinchExampleViewController.h interface file 237, 240**

- PinchExampleViewController.m implementation file 237-239**

Pizza order application

- building 145-147
- controls binding, actions used 147-152
- controls binding, outlets used 147-152

- PizzaOrdersViewController.h interface file 147, 151**

- PizzaOrdersViewController.m implementation file 148, 152**

- playAlertSound method 218**

- playMovie function 128**

- portrait, orientation method 79**

- Portrait upside-down, orientation method 79**

- power management component 12**

- pre-processor directive 21**

- preferences component 13**

- Primary Category 380**

- program build log**

- viewing 284, 285

project

- breakpoints, debugging with 272, 273
 - debugging 268
 - errors, handling 268
 - fix-it, used for correcting code 270
 - LLVM compiler, setting up 270, 271
 - logic errors 269
 - properties tracking, NSLog used 273-275
 - running 268
 - runtime errors 269
 - syntax errors 269

provisioning profile

- app, creating to iOS device 368-373
 - app, deploying to iOS device 368-373
 - creating 365-368
 - using, to install app on iOS device 368

- push 168**

Q

- QuartzCore.Framework 102**

R

- readFromData method 96, 97**

- record/stop button 327**

- remote repositories 298**

- repository**
 - creating 297
 - in Xcode, configuring 300
 - items, adding 303
 - local repositories 298
 - local subversion repository, setting up 298-300
 - project, checking 305-308
 - remote repositories 298
 - subversion repository, configuring 300-302
 - TapExample project, adding 303-305
 - types 298
- repository, in Xcode**
 - configuring 300
 - subversion repository, configuring 300-302
- repository organizer**
 - used, to tracking files 315, 316
- resignFirstResponder message 88**
- resignFirstResponder method 87**
- resizable interface 79**
- Review Notes (Optional) 380**
- revisions**
 - comparing, Timeline used 312, 313
 - listing chronologically, log mode used 314
 - selecting, Timeline used 312, 313
- RootViewController.m 160**
- RootViewController.h 160**
- RootViewController.m implementation file 163**
- RootViewController.xib 160**
- rotatable interface**
 - about 79
 - controls, relocating within view 80, 81
 - enabling 79, 80
- runtime errors 269**

S

- Save button 106, 356**
- Save Changes button 354**
- saveData() method 114-116**
- scalingMode property 129**
- scheme**
 - defining, scheme editor used 277, 278
 - defining for project builds, scheme editor used 276
- scheme editor**
 - used, for defining scheme 277, 278
- ScrollingViews application 180**

- ScrollingViews project**
 - controls binding, actions used 179, 180
 - controls binding, outlets used 179, 180
 - creating 177-179
- ScrollingViewsViewController.m implementation file 179**
- searchData() method 115**
- search field 328**
- Secondary Category (Optional) 380**
- Security.Framework 102**
- security component 12**
- segmented controls 169**
- Sender directive 153**
- setter 153**
- setValue method 115**
- ShakeExample application 246**
 - modifying 248
- ShakeExample project**
 - creating 243, 244
- ShakeExampleViewController.m implementation file 243-245**
- shakes**
 - detecting 242
 - motionBegan method, implementing 245-247
 - motionCancelled method, implementing 245-247
 - motionEnded method, implementing 245-247
 - motion events 248, 249
 - ShakeExample application, modifying 248
 - ShakeExample project, creating 243, 244
- shouldAutorotateToInterfaceOrientation\$ message 79**
- Show Activity Indicator method**
 - implementing 207-209
- showInView\$self.view method 215**
- showProgressDismiss method 209**
- showProgress event 209**
- simple database application**
 - building 104
 - core data application 104-108
- sliders 169**
- SliderValue label 176**
- sockets component 12**
- Software Development Kit (SDK) 31**
- sorted symbols, unified navigation UI**
 - listing, in project 51, 52
- SoundID variable 220**

- source-code management (SCM) 294
- source-control, Xcode
 - features 308-311
- Source Configuration Management (SCM) 10
- Source Control Management (SCM) 308, 317
- SQLite component 13
- square_root function 20
- stack trace panel, Xcode Debugger 263
- startAnimating method 209
- startGyroUpdates, CMMotionManager Method 260
- startGyroUpdatesToQueue function
 - about 259
 - withHandler, CMMotionManager Method 260
- startUpdatingLocation method 131
- static analysis 261, 288
- static analysis result
 - Static Analyzer, running 279
 - viewing 278
- stopAnimating method 209
- stopGyroUpdates, CMMotionManager Method 260
- stopUpdatingLocation method 134, 135
- StoreKit.Framework 102
- Submit button 362
- subversion
 - about 296
 - advantages 296
 - local subversion, installing 296, 297
- subversion repository
 - project, adding 321
 - project, adding to 321
- Support URL 380
- SwipeExample project
 - creating 232-235
- swipes
 - background, changing 235
 - delayFactor, adjusting 235
 - detecting 231
 - identifying 236
 - SwipeExample project, creating 232-234
 - tracking 236
- switches 169
- SwitchesSlidersSegments project
 - creating 170-172
- SwitchesSlidersSegmentsViewController.m
 - implementation file 173, 176

- switch statement 167
- syntax errors 269
- SystemConfiguration.Framework 102
- System Sound Services class 202
- system usage 343

T

- tableView
 - cellForRowAtIndexPath method 161
- tableView class 162
- Table View example
 - modifying 198
- TableViewExample application 166, 168
- table views
 - implementing 159
 - row items, grouping in TableViewExample application 163-168
 - Table view application, creating 159-162
- tapCountLabel control 230
- TapExample application 228, 230
- TapExample project 226
 - adding, to repository 303-305
 - creating 226-228
- TapExampleViewController.h interface file 228
- TapExampleViewController.m implementation
 - file 228, 229
- taps
 - controls, binding 228-230
 - detecting 226
 - program, modifying to change background 231
 - TapExample project, creating 226-228
- target menu 328
- text editing, Xcode workspace preferences 62
- text fields
 - about 192
 - applications, creating 193-195
 - using 193
- text property 193
- textStorage property 97
- TextView control 195
- text views
 - about 192
 - applications, creating 193-195
 - using 193
- TextViewsandButtonsViewController.h interface
 - file 197

- TextViewsandButtonsViewController.m**
 - implementation file 196
- threading component** 13
- threads** 332, 343
- time/run control** 328
- Timeline**
 - used, for comparing revisions 312, 313
 - used, for selecting revisions 312, 313
- time profiler** 332, 343
- time profiler, instruments** 344
- titleForHeaderInSection** method 198
- touchesBegan**
 - touches method 229
- touchesMoved** method 239, 241
- Track Blame**
 - using, to check past check-ins 313
- transformWithRotation** method 239
- transformWithScale** method 238
- txtUsername** control 84

U

- UIAcceleration** class 249
- UIAccelerometer** class 249, 259
- UIAccelerometerDelegate** protocol 259
- UIActionSheet** class 202, 215
- UIActivityIndicator** class 209
- UIActivityIndicatorView** class 207
- UIActivityIndicatorViewStyleWhiteLarge** style 208
- UIAlertView** alert class 209, 211
- UIAlertView** class 202, 207, 211
- UIAlertView** class variable 248
- UIAlertView** control 208
- UIAlertView** object 212, 216
- UIApplication** class 70
- UIApplication** instance 224
- UIApplicationMain()** function 70
- UIApplication** object 40, 70, 71, 73
- UIBarStyle** class 217
- UIButton** button 67, 192
- UIColor** class 267
- UIDeviceOrientation** class 253
- UIDeviceOrientationDidChangeNotification**
 - notification method 250
- UIEvent** events 224
- UIKit.Framework** 102
- UILabel** 75
- UILabel** controls 147
- UILabel** label control 67
- UILabel** objects 113
- UIResponder** class 224, 225
- UIResponder** event 226
- UIResponder** motion 244
- UIResponder** responder chain 244
- UIScrollView** class 177
- UITextField** control 146, 152
- UITextField** field 67
- UITouch** class 224, 241
- UITouchPhaseBegan**, **UITOUCH PHASE EVENT** 226
- UITouchPhaseCancelled**, **UITOUCH PHASE EVENT** 226
- UITouchPhaseEnd**, **UITOUCH PHASE EVENT** 226
- UITouchPhaseMoved**, **UITOUCH PHASE EVENT** 226
- UITouchPhaseStationary**, **UITOUCH PHASE EVENT** 226
- UIViewController** class 225
- UIView** object 42
- UIView** variable 241
- UIWindow** instance 224
- unified navigation UI**
 - about 50
 - active/inactive breakpoints 54
 - central search interface 52, 53
 - code assistants, using 55
 - code snippets library 57, 58
 - data, debugging with compresssionable stack traces 54
 - files in project, listing 50, 51
 - file templates library 56, 57
 - issues, tracking 53
 - jump bar 55
 - LLVM Compiler 2.0 55
 - logs, collection 55
 - media library 59
 - object library 58
 - static analysis, using to find potential problems 53
 - symbol navigator 51, 52
 - version editor 56
 - Xcode's development environment settings, resetting 60

unique device identifier (UDID) 360, 368

URL Utilities component 13

user interface

building 205

controls, adding to view 205-207

userName 152

UsingGitExample project 318

V

valueForKey method 116

version editor

in Xcode 4 294, 295

local subversion server, installing 296, 297

subversion 296

version editor, unified navigation UI 56

Version Number 380

vfiw control 328

vibratePhone method 218

View 17

view

about 17, 145

action sheets, associating 214

implementing 152

view based application template

application, creating with buttons 193, 194

application, creating with text fields 193, 194

application, creating with text view 193, 194

button controls 192

controls binding, actions used 156, 158,
173-197

controls binding, outlets used 156, 158,
173-197

creating 154

Custom Picker project, creating 186-188

custom pickers 186

Date Picker project, creating 182, 183

date pickers 181

FavoriteColor application, creating 155, 156

labels 193

navigation-based applications 168, 169

pickers 181

row items, grouping in TableViewExample
application 163-167

row items, grouping into sections 163

ScrollingViews project, creating 177-179

segmented controls 169, 170

sliders 169, 170

switchers 169, 170

SwitchesSlidersSegments project, creating
170-172

Table view application, creating 159-162

Table View example, modifying 198

table views, implementing 159

text fields 192

text view 192

user input, handling 192

user output, handling 192

web views 169, 170

viewController object 109

view controllers

implementing 152, 153

input field, declaring as view controller property
154

viewDidAppear method 244

viewDidLoad() method 116, 138, 175, 188, 330,
334

viewDidLoad method 116

view hierarchy component 15

view icons, view mode types 336

view icons and descriptions, view mode types
336

view icons and labels, view mode types 336

view property 42

view small icons and labels, view mode types
336

W

web views 169, 170

web views component 15

With XIB for user interface option 107

workspace 36

**workspace settings, Xcode 4 workspace
environment** 49

X

Xcode

about 10

file statuses 308-311

repository, configuring 300

- source-control features 308-311
- used, for implementing MVC 145
- Xcode's development environment settings, unified navigation UI 60**
- Xcode 4**
 - about 9
 - instruments 343
 - used, for archiving apps 375-377
 - used, for submitting apps 375-377
 - version editor 294, 295
- Xcode 4 workspace environment**
 - about 48
 - application status bar/activity window 49
 - Application Toolbar 48, 49
 - workspace, settings 49
- Xcode component 8**
- Xcode Debugger**
 - about 261, 262
 - activity viewer 276
 - code completion, using 286-288
 - code editor, debugging features 275, 276
 - code editor window 264
 - console output window 265
 - debugger toolbar 262
 - disassembly window 263
 - exploring 275
 - issues navigator, viewing 284
 - memory leak, detecting 281, 282
 - progress window 276
 - program build log, viewing 284, 285
 - project, configuring for automatic Static Analysis 280, 281
 - scheme defining for project builds, scheme editor used 276
 - schemes defining, scheme editor used 277, 278
 - stacks, navigating through 289
 - stack trace panel window 263
 - static analysis results, viewing 278
 - static analyzer 290
 - Static Analyzer, running 279, 280
 - threads, navigating through 289
 - uninitialized variable, instance detecting 282, 283
 - Xcode, stopping from alerting 288, 289
- Xcode developer set of tools**
 - about 23
 - interface builder 23
 - iPhone simulator 23
 - Xcode instruments 26, 27, 28
 - Xcode Integrated Development Environment (IDE) 23
- Xcode developer tools**
 - removing 35
- Xcode development environment**
 - about 35, 36, 67, 368
 - iPhone application, writing 38-44
 - new project, creating 37
 - new Xcode Assistant, working with 46, 48
 - older projects, migrating into new environment 37
 - single-development environment, working within 36
- Xcode Integrated Development Environment (IDE) 24**
- Xcode screen 23**
- Xcode workspace preferences**
 - about 60
 - behaviors preference pane 61
 - distributed builds preference 63
 - documentation preference 62
 - fonts & colors preference 61
 - general button 61
 - key binding preference 62
 - locations preference 62, 63
 - source trees preference 63
 - text editing preference 62
- xib file 66**

Z

- zombies 332, 343**



**Thank you for buying
Xcode 4 iOS Development Beginner's Guide**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



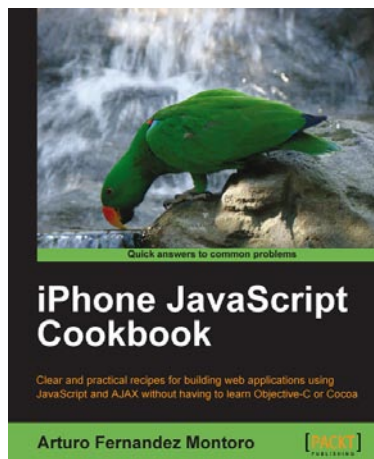
Core Data iOS Essentials

ISBN: 978-1-849690-94-2

Paperback: 340 pages

A fast-paced, example-driven guide to data-driven iPhone, iPad, and iPod Touch applications

1. Covers the essential skills you need for working with Core Data in your applications
2. Particularly focused on developing fast, light weight data-driven iOS applications
3. Builds a complete example application. Every technique is shown in context
4. Completely practical with clear, step-by-step instructions



iPhone JavaScript Cookbook

ISBN: 978-1-849691-08-6

Paperback: 328 pages

Clear and practical recipes for building web applications using JavaScript and AJAX without having to learn Objective-C or Cocoa

1. Build web applications for iPhone with a native look feel using only JavaScript, CSS, and XHTML
2. Develop applications faster using frameworks.
3. Integrate videos, sound, and images into your iPhone applications
4. Work with data using SQL and AJAX

Please check www.PacktPub.com for information on our titles

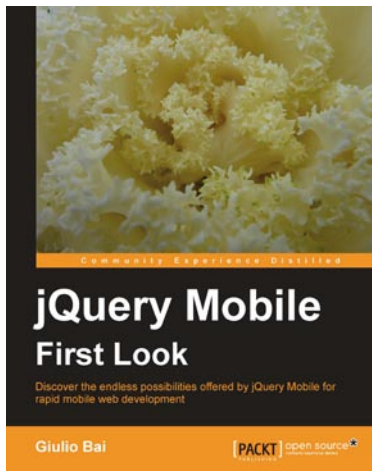


Cocos2d for iPhone 0.99 Beginner's Guide

ISBN: 978-1-849513-16-6 Paperback: 368 pages

Make mind-blowing 2D games for iPhone with this fast, flexible, and easy-to-use framework!

1. A cool guide to learning cocos2d with iPhone to get you into the iPhone game industry quickly
2. Learn all the aspects of cocos2d while building three different games
3. Add a lot of trendy features such as particles and tilemaps to your games to captivate your players
4. Full of illustrations, diagrams, and tips for building iPhone games, with clear step-by-step instructions and practical examples



jQuery Mobile First Look

ISBN: 978-1-849515-90-0 Paperback: 216 pages

Discover the endless possibilities offered by jQuery Mobile for rapid Mobile Web Development

1. Easily create your mobile web applications from scratch with jQuery Mobile
2. Learn the important elements of the framework and mobile web development best practices
3. Customize elements and widgets to match your desired style
4. Step-by-step instructions on how to use jQuery Mobile

Please check www.PacktPub.com for information on our titles