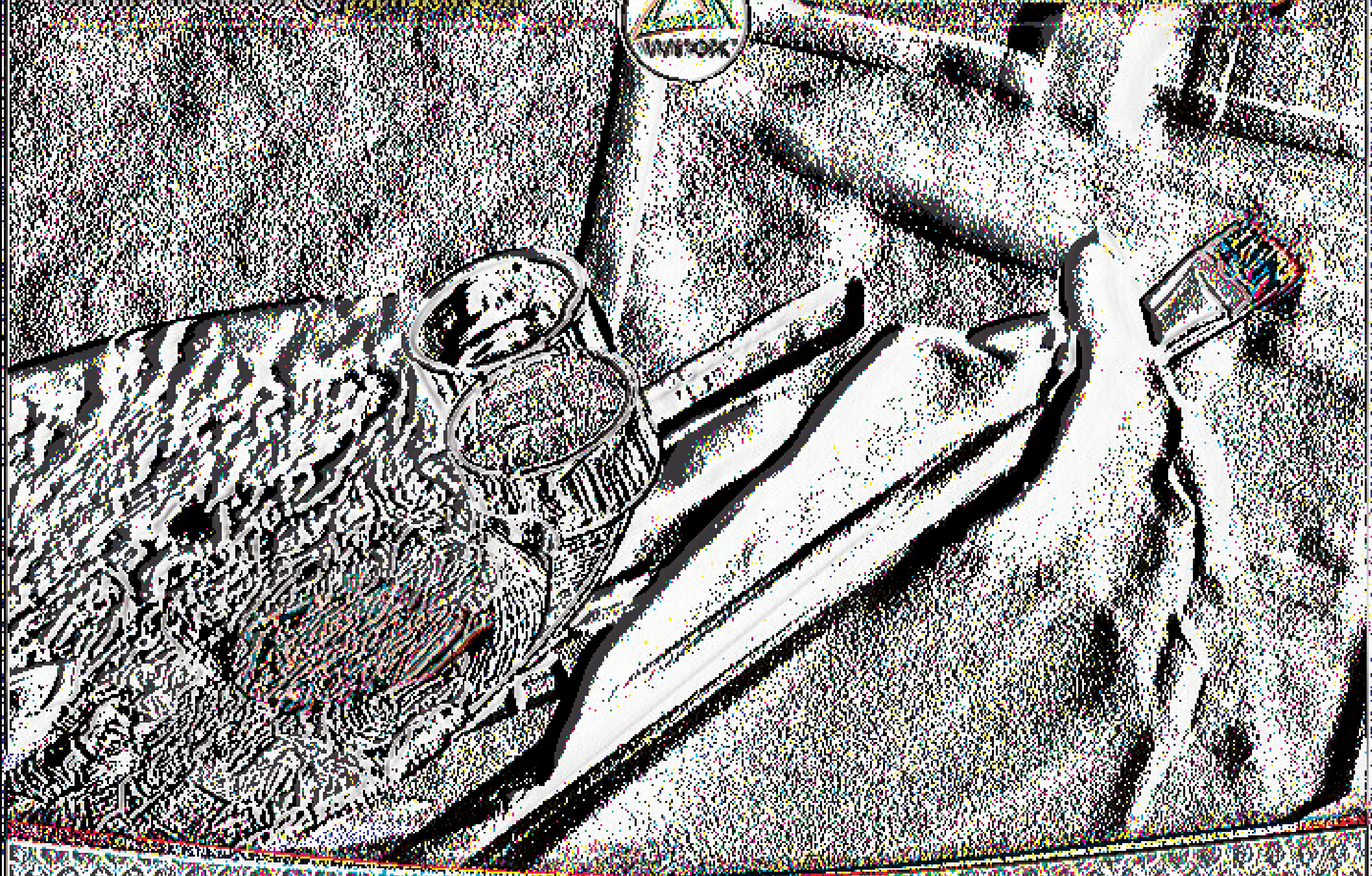
Join the discussion @ p2p.wrox.com

COVERS iOS 6 DEVELOPMENT



iPhone and iPad App 24-Hour Trainer

Abhishek Mishra, Gene Backlin

IPHONE® AND IPAD® APP 24-HOUR TRAINER

iPhone® and iPad® App 24-Hour Trainer

iPhone® and iPad® App 24-Hour Trainer

Abhishek Mishra
Gene Backlin

WILEY
John Wiley & Sons, Inc.

iPhone® and iPad® App 24-Hour Trainer

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
`www.wiley.com`

ISBN: 978-1-118-13081-0
ISBN: 978-1-118-22507-3 (ebk)
ISBN: 978-1-118-23837-0 (ebk)
ISBN: 978-1-118-26314-3 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2012930548

To my wife Sonam, for her love and support through all the years we've been together.

—Abhishek Mishra

This book is being dedicated to the family unit, past, present and future. May it never lose sight of its existence.

—Gene Backlin

ABOUT THE AUTHORS

ABHISHEK MISHRA has been developing software for over 12 years and has experience with a diverse set of programming languages and platforms. He holds a master's degree in computer science from the University of London and is a freelance consultant and trainer specializing in iOS development. He lives with his wife in London, and when not working, spends his time stargazing under Scottish skies.

GENE BACKLIN'S first calculating machine was a slide rule. If you ask him about the information revolution, his response would be "fascinating." He has developed on computers that loaded programs from paper tape, the revolutionary NeXT computer, which he still has two of, to the iPhone and iPad. Gene feels very fortunate to have not only seen an industry evolve, but also to have been an active participant in it. His childhood interest in electronics helped him break into the computer industry. He started building Heathkit walkie-talkies, leading him to build the Heathkit H-8 digital computer and H-9 video terminal, which he still has, in 1978. He taught himself programming using Extended Benton Harbor Basic and after the IBM-PC was introduced, Gene built the Heathkit H-151 PC-compatible computer, which is still running today.

Gene is owner and principal consultant of MariZack Consulting, formed in 1991 with one purpose — to help. He has been helping clients for over 30 years, including IBM, McDonnell Douglas, Waste Management, U.S. Environmental Protection Agency, Nations Bank, Bank of America, Bank One, and Sears to name a few. He is also a faculty member of DePaul University's College of Computing and Digital Media and has previously penned *Developing NeXTSTEP Applications* in 1995, and *Professional iPhone and iPad Application Development* in 2010.

TECHNICAL EDITOR

ALLAN EVANS is a veteran multimedia developer with almost 20 years of programming experience. He has turned his attention to iOS devices, currently working as a Mobile Architect for Walgreens. He has worked on iOS projects for Sears Holdings, Tribune Company and CCC Information Systems. He has spoken about iOS at SecondConf 2010 (`www.secondconf.com`) and presented "An Introduction to iOS and Xcode" to the Chicago Adobe Users Group (`www.augchicago.com`). In his spare time, he has taken a liking to physical computing (Arduino and its ilk), and has been known to write screenplays and graphic novels.

CREDITS

ACQUISITIONS EDITOR
Mary James

PROJECT EDITOR
Ed Connor

TECHNICAL EDITOR
Allan Evans

PRODUCTION EDITOR
Daniel Scribner

COPY EDITOR
Kim Cofer

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Knies

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

COMPOSITOR
Craig Woods, Happenstance Type-O-Rama

PROOFREADER
Jen Larsen, Word One

INDEXER
Ron Strauss

COVER DESIGNER
Ryan Sneed

COVER IMAGE
© webphotographeer / iStockphoto

VERTICAL WEBSITES SUPERVISING PRODUCER
Rich Graves

VERTICAL WEBSITES ASSOCIATE PRODUCER
Josh Frank

ACKNOWLEDGMENTS

THIS BOOK WOULD NOT HAVE BEEN POSSIBLE without the support of the team at John Wiley and Sons—Mary James, Ed Connor, and Kim Cofer. I would also like to thank Allan Evans for his keen eye for detail and Gene Backlin for his continued support throughout the project. It has been my privilege to work with you. Thank you.

—Abhishek Mishra

I WOULD FIRST LIKE TO THANK MY WIFE, Roseann, my son Zachary, and my daughter Marissa, for putting up with me once again while I was writing this book, and my son Ethan for keeping me young and remembering what you are to do if this idea ever comes up again, but this time really hold me to it!

I would also like to thank my mother Mary Louise, father Eugene William and daughter Hannah Angel, for giving me the gift of themselves and spirit, which I carry with me every day of my life. I miss you.

To Helen and Jerry, you will never know how much you mean to me, especially the gum!

To Allan, I am glad we were able to work together again. To Abby, it has been fun working together on this.

This book is more than just words and code. It is time and people. The pages you hold in your hand are a snapshot of events that somehow managed to come together with the very hard work of a lot of people; Ed, Mary and Kim specifically, I want to thank you all for everything that you did, it is greatly appreciated.

Finally I would like to thank Jean-Marc Krikorian especially for once again really doing nothing as he did in 1995 and 2010.

—Gene Backlin

CONTENTS

BONUS MATERIAL

INTRODUCTION

WHEN FIRST LEARNING THE IOS DEVELOPMENT ENVIRONMENT, it is natural to be overwhelmed with new concepts like view controllers and table views. While experience with previous development environments and languages is helpful, and iOS developing resources are available from Apple and forums, it is still a daunting task to become proficient.

This book is written to help someone new to iOS development come to grips with the basic concepts, and hopefully avoid making the mistakes we made when we were starting out. This book adopts a hands-on Try It approach, and you get to try out each new concept as you progress through the book.

iOS application development is a huge topic, and it is just not possible to put every single topic related to iOS application development in this book. That being said, the aim of this book is to help you get started and to understand the fundamentals of the SDK.

This book has been written for you, the reader. We hope that after reading this book, you can take your first steps into the wildly exciting world of iOS app development.

WHO THIS BOOK IS FOR

This book is for beginners with little or no prior programming experience who want to pursue a career in the exciting world of iOS development.

If you are a beginner and are looking for a book to help you get up to speed with basic concepts and start you down your journey of iOS application development, this is the book for you.

Although you do not need to have any prior programming experience, a little knowledge will help you move faster through the initial lessons. If you are a more experienced developer, this book can help you get up to speed with new concepts relating specifically to iOS 5 development.

WHAT THIS BOOK COVERS

This book covers iOS 5 application development. That includes development for both the iPhone and the iPad. The lessons in this book use Xcode 4.2.1 and make use of new iOS 5 features like storyboards and Automatic Reference Counting. Most lessons use Interface Builder to create the user interface; however, creating user interfaces programmatically is covered in Lesson 17.

The appendixes on the DVD and on the book's website (`www.wrox.com/go/iphoneipadappvideo`) contain various topics ranging from an introduction to programming with Objective-C, to deploying applications on the App Store. New iOS 5–specific topics such as storyboards, Twitter integration, iCloud document storage, and Core Image are covered in Lessons 5, 20, 25, and 32, respectively.

HOW THIS BOOK IS STRUCTURED

This book consists of 38 short lessons and 6 appendixes. Most lessons introduce a single topic and end with a step-by-step Try It section where you get to apply the concepts you've learned in the lesson to create a simple iOS application. Lessons toward the beginning of the book are simpler, and progress in complexity as you work your way through the book.

If you are an absolute beginner to programming, you should read Appendix B for an introduction to computer programming with Objective-C and then progress through the lessons from cover to cover, sequentialy.

If you have prior experience with iOS development and want to read this book for a particular topic of interest, you can jump right in to the relevant lessons. iOS development is a vast topic, and no book can attempt to cover everything related to it; this book is no exception. However, several lessons contain links to places on the Web where you can obtain additional information.

When you're finished reading the book and watching the DVD, you'll find lots of support in the P2P forums.

INSTRUCTIONAL VIDEOS

Learning is often enhanced by seeing in real-time what's being taught, which is why most lessons in the book have a corresponding video tutorial on the DVD accompanying the print book and on the book's website (`www.wrox.com/go/iphoneipadappvideo`). And, of course, it's vital that you play along at home—fire up Xcode and try out what you read in the book and watch on the videos.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

References like this one point you to the DVD to watch the instructional video that accompanies a given lesson.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show URLs and code within the text like so: `persistence.properties`.
- We present code in the following way:

```
We use a monofont type for code examples.
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. All the source code used in this book is available for download at `www.wrox.com`. When at the site, simply locate the book's title (use the Search box or one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-118-13081-0.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the Book Search Results page, click the Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

A complete book list including links to errata is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Errata page, click the Errata Form link and complete the form to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com` you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1 Hello iOS!

Hello and welcome to the exciting world of iOS application development. iOS is Apple's operating system for mobile devices; the current version as of writing this book is 5.0. It was originally developed for the iPhone (simply known as iPhone OS back then), and was subsequently extended and renamed in June 2010 to iOS to support the iPad, iPhone, and iPod Touch.

iOS at its core is Unix-based, and has its foundations in MacOS X, which is Apple's desktop operating system. In fact, both iOS and MacOS X share a common code base. As new versions of mobile operating systems have appeared, Apple has brought over more functionality from MacOS X. This is part of Apple's strategy to bridge the difference between desktop and mobile computing.

With the launch of version 5.0, Apple has once again pushed the boundaries of what is achievable on smart phones and tablet computers. iOS 5.0 has more than 200 new features and is without a doubt the most significant update to the family.

This lesson introduces you to the arena of iOS development.

IOS DEVELOPER ESSENTIALS

Before you get started on your journey of becoming an iOS developer, you will need some essential resources. This section covers these basic requirements.

A Suitable Mac

To develop apps for the iPhone and the iPad using the official set of tools provided by Apple, you will first need an Intel-based Mac running Mac OS X Lion with a minimum 2GB of RAM and at least 11GB of free space on your hard disk. You do not need a top-spec model to get started. In fact a Mac Mini or a low-end MacBook will work just fine.

Processor speed is not going to make much difference to you as a developer. You will be better off investing your money toward more RAM and hard disk space instead. These are things you can never get enough of. A large screen does help, but it is not essential.

A Device for Testing

If you are reading this book, chances are that you have used an iPhone/iPad/iPod Touch and probably even own one or more of these nifty devices.

As far as development is concerned, there aren't many differences between developing for any of these devices. When you are starting out as an iOS developer, you will test your creations on the iOS Simulator. The iOS Simulator is an application that runs on your Mac and simulates several functions of a real iOS device (more on this later).

At some point, though, you will want to test your apps on a physical device. As good as the iOS Simulator may be, you must test on a physical device before submitting your app to the App Store.

Another good reason to test on a physical device is that the processor on your Mac is much faster than that on the iPhone/iPad. Your app may appear to execute much faster on your Mac (in the iOS Simulator) than it does on the real thing.

If the app you are going to make is targeted at iPhone users, you can also use an iPod Touch as the test device. These are significantly cheaper than iPhones and for the most part offer the same functionality as their phone counterparts.

Device Differences

Though many similarities exist in developing apps for the iPhone and the iPad, there are some obvious differences between the devices.

iPhone 3GS

The iPhone 3GS was a major increment from the iPhone 3G. It included a 600Mhz ARM processor, 256MB RAM, and several enhancements to support 3D games. All iPhone models come with at least one camera.

The iPhone 3GS has a screen size of 320 × 480 units. Notice the unit of measurement is not "pixel." Starting with iOS4 and the introduction of the Retina display on the iPhone 4, Apple has introduced a new device-independent coordinate system. Application developers express sizes and positions in this new system. Depending on the physical device on which the app is executed, these device-independent coordinates are converted to device-dependent coordinates by multiplying them with a scale factor. In the case of a device that does not have a Retina display (such as the iPhone 3GS), this scale factor happens to be 1.

Thus the screen size of the iPhone 3GS (Figure 1-1) happens to be 320 × 480 pixels because 1 unit is exactly 1 pixel on this device.

iPhone 4

The iPhone 4 has a generous 512MB RAM, and a super-fast 1GHz Apple A4 processor. It also introduced a new high-resolution display called the Retina display that packs twice as many pixels in the same physical screen area.

The screen size of the iPhone4 is also 320 × 480 units; however the internal scaling factor is 2, which implies that the actual number of pixels in an iPhone 4 screen is 640 × 960 (Figure 1-2).

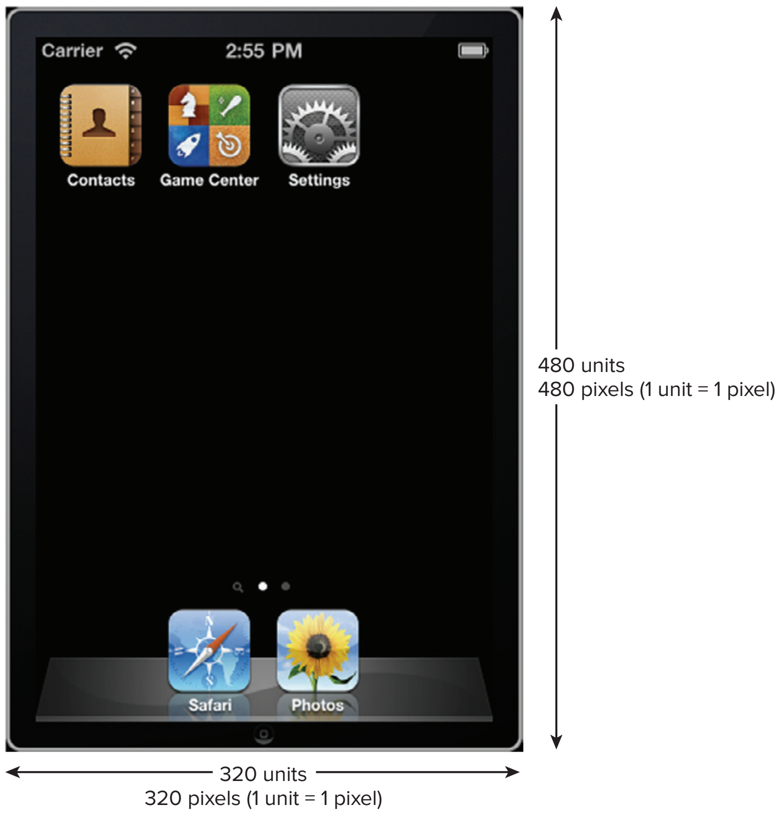


FIGURE 1-1

iPhone 4S

The iPhone 4S is the newest member of the iOS family. Like its predecessor, it also has a high-resolution Retina display. In addition, the iPhone 4S has a dual-core Apple A5 processor.

iPad

The iPad has a much larger screen size than the iPhone (768 × 1024 units); however, it does not have a camera. The first generation iPad is equipped with a 1GHz Apple A4 processor and has 256MB RAM. The iPad does not have a Retina display, hence the conversion factor from units to pixels for this device is 1. This means the number of pixels in an iPad screen is 768 × 1024 (Figure 1-3).

iPad 2

With the iPad 2 Apple has once again raised the stakes for high-end tablet computing. The iPad 2 has a much faster 1GHz Dual Core Apple A5 processor, more RAM, and two cameras, but sadly no Retina display.

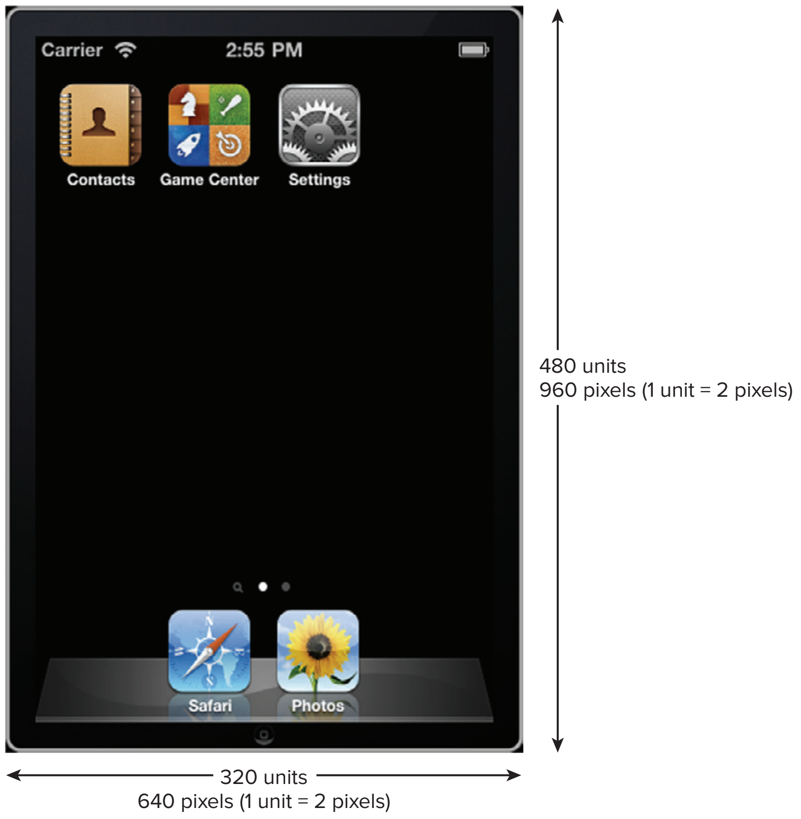


FIGURE 1-2

An iOS Developer Account

To develop your apps you will need to download the latest version of Xcode and the iOS SDK (Software Development Kit). To do this you must sign up for the Apple Developer Program to become a registered developer.

The signup process is free and you can immediately begin to develop your first apps. Limitations exist as to what you can do for free. To submit your apps to the App Store, get access to beta versions of the iOS/SDK, or test your apps on a physical device, you need to become a paying member.

Most of the concepts and apps presented in this book will work just fine with the free membership. The only exceptions would be examples that require the camera, accelerometer, and GPS for which you really do need to try the app on a physical device.

You can choose from two forms of paid membership as a registered Apple Developer: Standard and Enterprise.

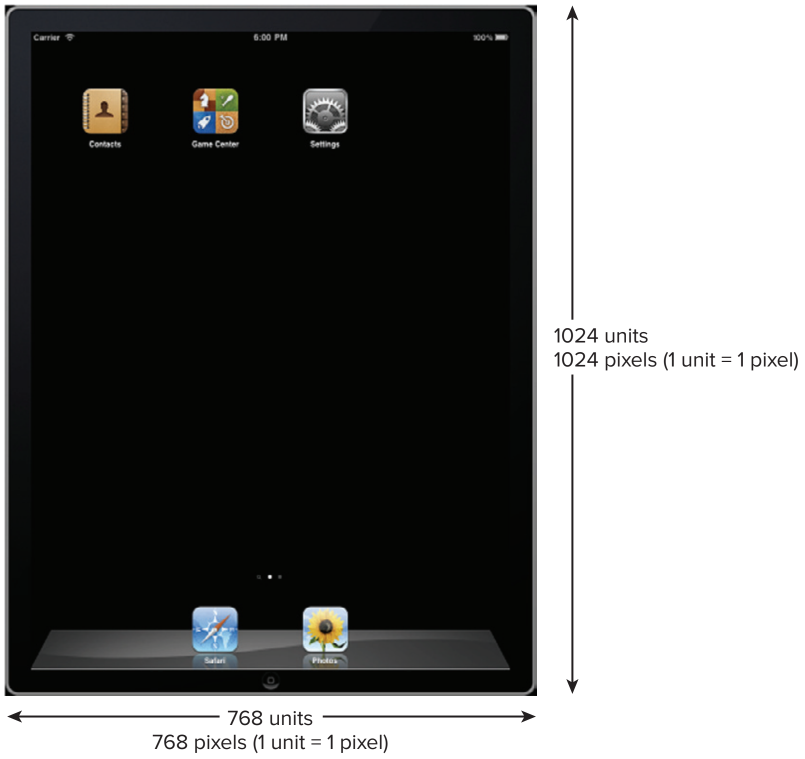


FIGURE 1-3

Standard

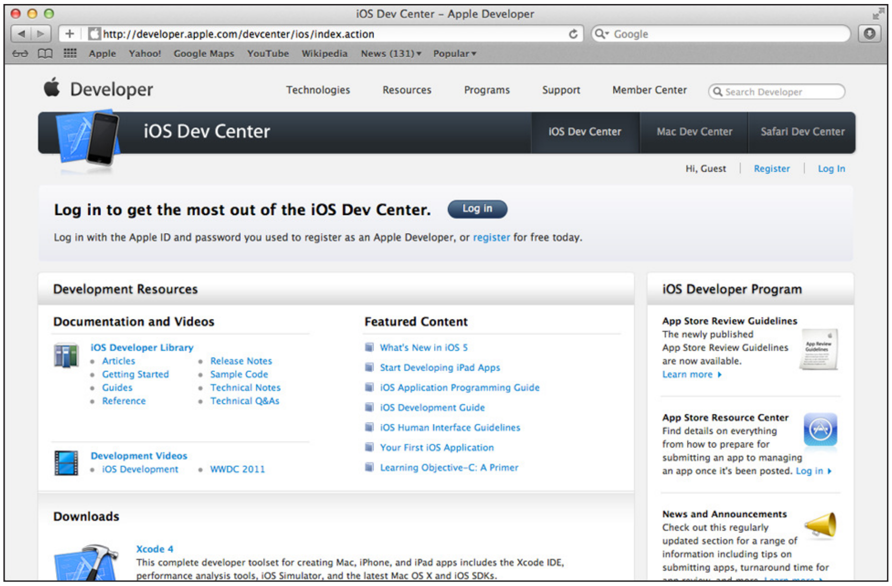
The Standard iOS Developer Program costs $99 a year and is for individuals or companies that want to develop apps that will be distributed through the App Store.

You can also test/distribute your apps on up to 100 devices without having to go through the App Store. This form of deployment (without having to submit them to the App Store) is called Ad-Hoc distribution and is a great way to submit a preview of the app to a client. This form of distribution is covered in detail in Appendix C.

Enterprise

The Enterprise iOS Developer Program costs $299 a year and is for large companies that want to develop apps for internal use and will not distribute these apps through the App Store. With the Enterprise iOS Developer Program there is no restriction to the number of devices on which your in-house application can be installed.

To start the registration process, visit the iOS Dev Center (Figure 1-4) at `http://developer.apple.com/devcenter/ios/index.action`


iOS Dev Center – Apple Developer

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

Developer — Technologies | Resources | Programs | Support | Member Center

iOS Dev Center — iOS Dev Center | Mac Dev Center | Safari Dev Center

Hi, Guest | Register | Log In

Log in to get the most out of the iOS Dev Center. Log in

Log in with the Apple ID and password you used to register as an Apple Developer, or register for free today.

Development Resources

Documentation and Videos

iOS Developer Library
- Articles
- Getting Started
- Guides
- Reference
- Release Notes
- Sample Code
- Technical Notes
- Technical Q&As

Development Videos
- iOS Development
- WWDC 2011

Featured Content
- What's New in iOS 5
- Start Developing iPad Apps
- iOS Application Programming Guide
- iOS Development Guide
- iOS Human Interface Guidelines
- Your First iOS Application
- Learning Objective-C: A Primer

Downloads

Xcode 4
This complete developer toolset for creating Mac, iPhone, and iPad apps includes the Xcode IDE, performance analysis tools, iOS Simulator, and the latest Mac OS X and iOS SDKs.

iOS Developer Program

App Store Review Guidelines
The newly published App Store Review Guidelines are now available. Learn more ›

App Store Resource Center
Find details on everything from how to prepare for submitting an app to managing an app once it's been posted. Log in ›

News and Announcements
Check out this regularly updated section for a range of information including tips on submitting apps, turnaround time for

FIGURE 1-4

The Official iOS SDK

The Apple iOS SDK (Software Development Kit) is a collection of tools and documentation that you can use to develop iOS apps. The main tools that make up the SDK are:

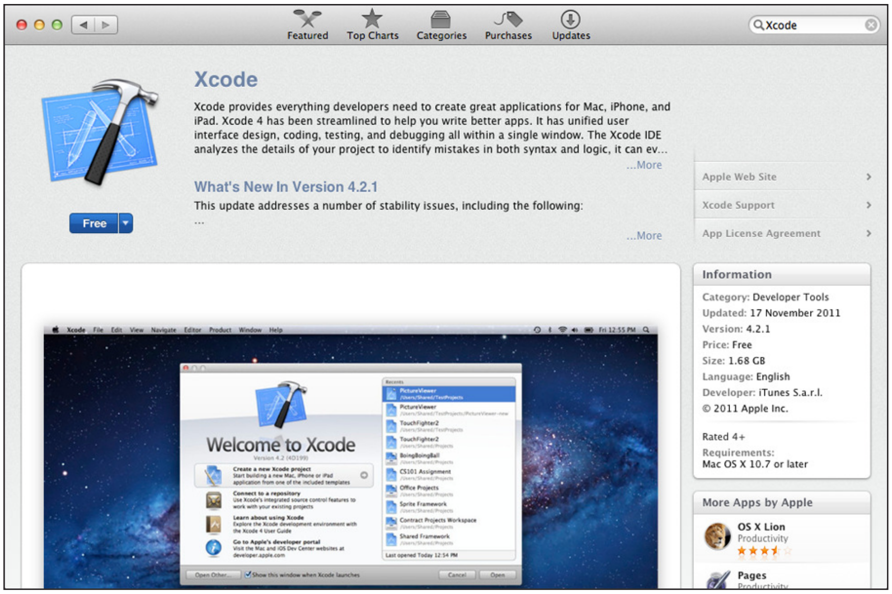
- **Xcode:** Apple's integrated development environment (IDE) that lets you manage your products, type your code, trace and fix bugs (debugging), and lots more.
- **Interface Builder:** A tool fully integrated into the Xcode IDE that lets you build your application's user interface visually.
- **iOS Simulator:** A software simulator to simulate the functions of an iPhone or an iPad on your Mac.
- **Instruments:** A tool that will help you find memory leaks and optimize the performance of your apps. Instruments are not covered in this book.

In addition to these tools, the iOS SDK also includes extensive documentation, sample code, How-To's, and access to the Apple Developer Forums.

The iOS SDK is available as a free download to registered members (registration is free). However there are benefits to paid membership, including the ability to debug tour code on an iOS device, distribute your applications, and two technical support incidents a year where Apple engineers will provide you code-level assistance.

Downloading and Installing

You can download and install Xcode 4.2.1 for Mac OS X Lion and the iOS SDK from the Mac App Store (Figure 1-5).



Xcode

Xcode provides everything developers need to create great applications for Mac, iPhone, and iPad. Xcode 4 has been streamlined to help you write better apps. It has unified user interface design, coding, testing, and debugging all within a single window. The Xcode IDE analyzes the details of your project to identify mistakes in both syntax and logic, it can ev... ...More

What's New In Version 4.2.1

This update addresses a number of stability issues, including the following:
... ...More

Information

Category: Developer Tools
Updated: 17 November 2011
Version: 4.2.1
Price: Free
Size: 1.68 GB
Language: English
Developer: iTunes S.a.r.l.
© 2011 Apple Inc.

Rated 4+
Requirements:
Mac OS X 10.7 or later

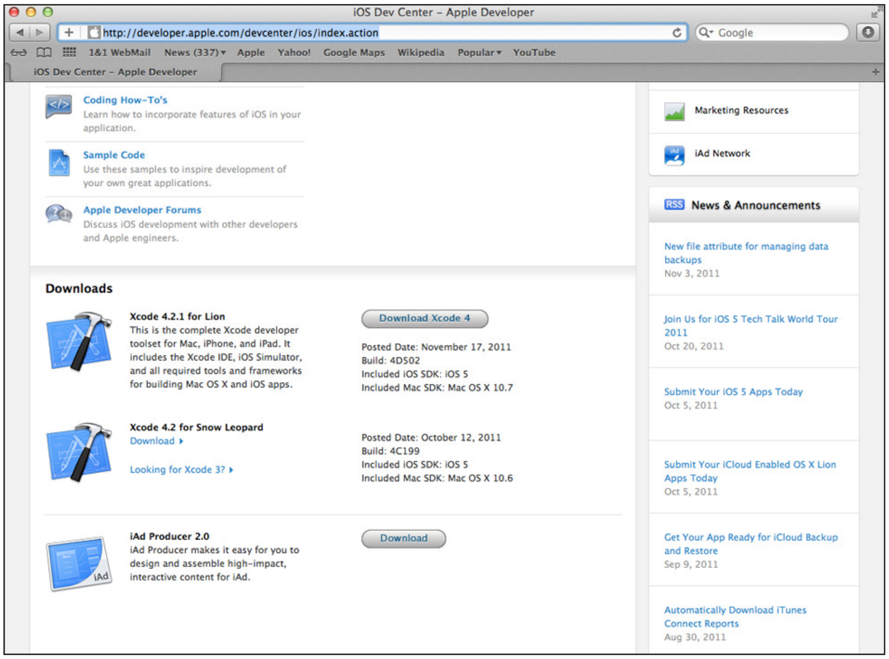
More Apps by Apple

OS X Lion
Productivity

Pages
Productivity

FIGURE 1-5

If you do not have the paid membership, you will only see a single version of Xcode and the iOS SDK to download. If you do have a paid membership you will have the option to download prior versions (Figure 1-6). This book is based on Xcode 4.2.1.


iOS Dev Center – Apple Developer
http://developer.apple.com/devcenter/ios/index.action
Google
1&1 WebMail News (337) Apple Yahoo! Google Maps Wikipedia Popular YouTube
iOS Dev Center – Apple Developer
Download Xcode 4
Download
Marketing Resources
iAd Network

Coding How-To's
Learn how to incorporate features of iOS in your application.

Sample Code
Use these samples to inspire development of your own great applications.

Apple Developer Forums
Discuss iOS development with other developers and Apple engineers.

Downloads

Xcode 4.2.1 for Lion
This is the complete Xcode developer toolset for Mac, iPhone, and iPad. It includes the Xcode IDE, iOS Simulator, and all required tools and frameworks for building Mac OS X and iOS apps.

Posted Date: November 17, 2011
Build: 4D502
Included iOS SDK: iOS 5
Included Mac SDK: Mac OS X 10.7

Xcode 4.2 for Snow Leopard
Download
Looking for Xcode 3?

Posted Date: October 12, 2011
Build: 4C199
Included iOS SDK: iOS 5
Included Mac SDK: Mac OS X 10.6

iAd Producer 2.0
iAd Producer makes it easy for you to design and assemble high-impact, interactive content for iAd.

News & Announcements

New file attribute for managing data backups
Nov 3, 2011

Join Us for iOS 5 Tech Talk World Tour 2011
Oct 20, 2011

Submit Your iOS 5 Apps Today
Oct 5, 2011

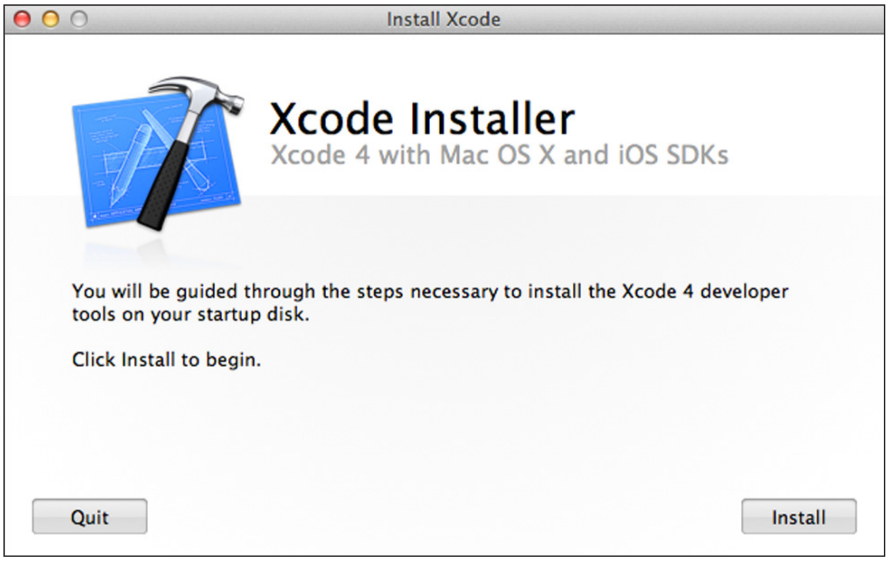
Submit Your iCloud Enabled OS X Lion Apps Today
Oct 5, 2011

Get Your App Ready for iCloud Backup and Restore
Sep 9, 2011

Automatically Download iTunes Connect Reports
Aug 30, 2011

FIGURE 1-6

After downloading Xcode and the iOS SDK, begin the installation process by double-clicking the Install Xcode icon in Launchpad (Figure 1-7). You don't have to change any of the defaults for the installer, so just read and agree to the software license and click Continue to proceed through the steps.



You will be guided through the steps necessary to install the Xcode 4 developer tools on your startup disk.

Click Install to begin.

FIGURE 1-7

The Typical App Development Process

Whether you intend to develop iOS apps yourself or manage the development of one, there is a basic sequence of steps involved in the development process (Figure 1-8). This section covers these steps briefly.

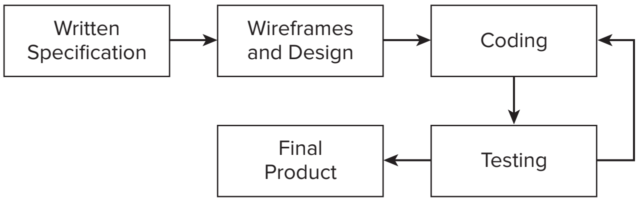


FIGURE 1-8

Writing a Specification

The development of an app begins with a concept. It is good practice to formally put this concept on paper and create a specification. You do not necessarily need to type this specification although it's a good idea to do so.

At the end of the project you should come back to the specification document to see how the final product that was created compares with the original specification.

As you build your experience developing iOS applications, this difference will become smaller. The specification must address the following points:

- ➤ A short description in 200 words or less
- ➤ The target audience/demographic of the users
- ➤ How will it be distributed (App Store, or direct to a small number of devices)
- ➤ A list of similar competing apps
- ➤ A list of apps that best illustrate the look-and-feel your app is after
- ➤ The pricing model of competing apps and potential pricing for your app

Wireframes and Design

A wireframe is large drawing that contains mockups of each screen of your app as well as lines connecting different screens that indicate the user's journey through your application.

Wireframes are important because they can help identify flaws in your design early on (before any coding has been done). They can also be used to show potential clients how a particular app is likely to look when it's completed.

There is no right or wrong way to make a wireframe. If it is for your personal use; you can just use a few sheets of paper and a pen. If it is for a client you might want to consider using an illustration package.

Coding

The actual process of creating an iOS app involves using the Xcode IDE to type your code. iOS apps are usually written in a language called Objective-C. An iOS app typically consists of several files of Objective-C code along with resource files (such as images, audio, and video).

These individual files are combined together by a process called *compilation* into a single file that is installed onto the target device. This single file is usually referred to as the *application binary* or a *build*.

Testing

It might sound obvious, but you must test your app after it has been developed. As a developer you test your code frequently as you write it. You must also perform a comprehensive test of the entire application as often as possible to ensure things that were working in the past continue to do so. This form of testing is called regression testing. It helps to make a test plan document. Such a document basically lists all the features that you want to test, and the steps required to carry out each test. The document should also clearly list which tests failed. The ones that fail will then need to be fixed and the test plan document can provide the replication procedure for the defect in question.

When your app is ready you will want to list it on the iTunes App Store. To do so involves submitting your app for review to Apple. Apple has several criteria against which it reviews applications and if your app fails one or more of these criteria it will be rejected—in which case you will need to fix the appropriate code and resubmit. It is best to test your apps thoroughly before submitting them in the first place. Distributing your apps via the App Store is covered in Appendix E.

You must always test on a real iOS device before submitting your app for the App Store review process, or giving it to a client to test. Testing on the iOS Simulator alone is not sufficient.

If you are developing for a client, you will probably need to send them a testable version of your work periodically for them to review. To do this you will need to give them something they can install on their devices. This is covered in Appendix F.

Home Screen Icon

Unless you provide an icon for your application, iOS will use a standard gray icon to represent your application in the home screen (Figure 1-9).

To replace this icon, simply create a PNG file with the appropriate dimensions, as shown in Table 1-1.

TABLE 1-1: Standard Icon Sizes

DEVICE	ICON SIZE (IN PIXELS)
iPhone	57 × 57
iPhone 4 (Retina)	114 × 114
iPad	72 × 72

These dimensions are in pixels and are different for iPhone-based and iPad-based applications.

If you are building an iPhone application that should run on both the iPhone 4 (Retina-based) and the iPhone 3GS (non-Retina–based), you will need to create two versions of each image resource that your application requires. One version of each file will be twice the size of other; your icon images are no exceptions.

You will also need to name the Retina display versions of your images using a special convention. For example, if your standard icon file is called `icon.png`, then the Retina display version of the icon should be called `icon@2x.png`.

You learn to use these icons in this lesson's Try It section.

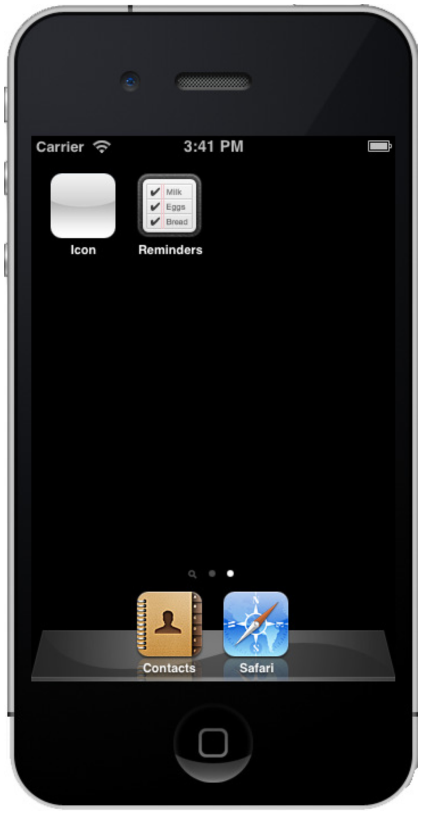


FIGURE 1-9

Application Launch Image

A launch image is a special PNG image that you may provide as part of your iOS application. When a user taps your application's icon on the home screen, iOS looks for this launch image, and if found, shows this launch image before loading the rest of the application. While this launch image is displayed, iOS proceeds to load the rest of your application in the background.

Once your application has finished loading, iOS gives it control and simultaneously hides the place-holder launch image that was displayed in its stead. The overall effect of the launch image is to give your users the perception that your application has launched quickly.

The launch image must be of a specific size; these sizes are listed in Table 1-2. Once again, these sizes are different for iPhone-based and iPad-based applications. Retina display devices (iPhone 4) require images that are twice the size of their non-Retina counterparts.

An additional requirement applies for iPad-based applications. These applications must provide a launch image for each supported orientation (landscape/portrait).

TABLE 1-2: Application Launch Image Sizes

DEVICE	IMAGE SIZE (IN PIXELS)
iPhone	320 × 480
iPhone 4 (Retina)	640 × 960
iPad (portrait orientation)	768 × 1024
iPad (landscape orientation)	1024 × 768

The usual naming convention of appending `@2x` applies to the Retina display versions of images. You learn to use launch images in this lesson's Try It.

TRY IT

In this Try It, you build a simple iPhone application using Xcode 4.2.1 that displays the text "Hello iOS" on the screen.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template
- Open a storyboard in Interface Builder
- Display the Xcode Utilities area
- Change the background color of the default scene in the storyboard
- Add a Label from the Xcode Object library
- Set up an application icon
- Setup a launch image
- Test an app in the iOS Simulator

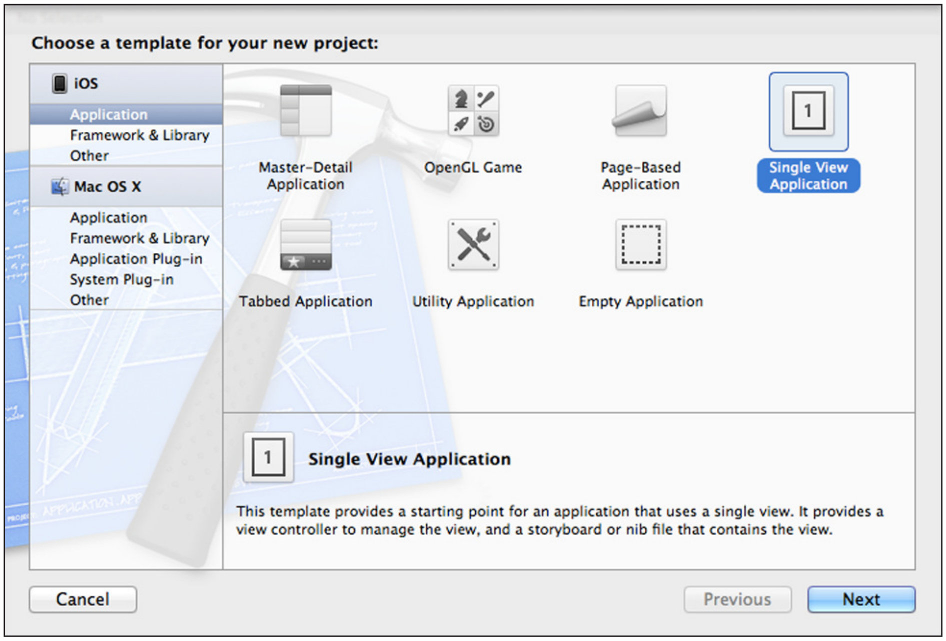
You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 1 folder in the download.*

Hints

- Download and install the latest version of Xcode and the iOS SDK on your Mac.
- Launch Xcode.

Step by Step

1. Create a Single View Application in Xcode called `HelloIOS`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application (Figure 1-10) template and click Next.



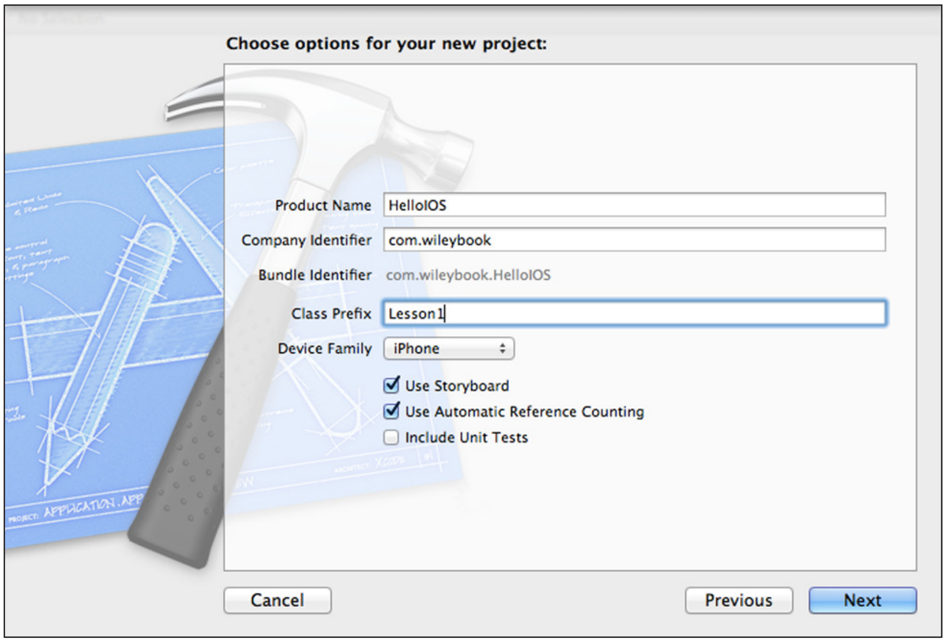
This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

FIGURE 1-10

    4. Use the following information in the project options dialog box (Figure 1-11) and click Next.
        - **Product Name:** HelloIOS
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson1
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked

- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*



Choose options for your new project:

Product Name: HelloIOS
Company Identifier: com.wileybook
Bundle Identifier: com.wileybook.HelloIOS
Class Prefix: Lesson1
Device Family: iPhone
☑ Use Storyboard
☑ Use Automatic Reference Counting
☐ Include Unit Tests

FIGURE 1-11

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Open the `MainStoryboard.storyboard` file in Interface Builder (Figure 1-12).
   1. Ensure the project navigator is visible and the `HelloIOS` project is selected and expanded. To show the project navigator, use the View ➪ Navigators ➪ Show Project Navigator menu item. To expand a project, click the triangle next to the project name in the project navigator.
   2. Click the `MainStoryboard.storyboard` file.
   3. Ensure the utilities editor is visible. To show the utilities editor, use the View Utilities ➪ Show Utilities menu item.

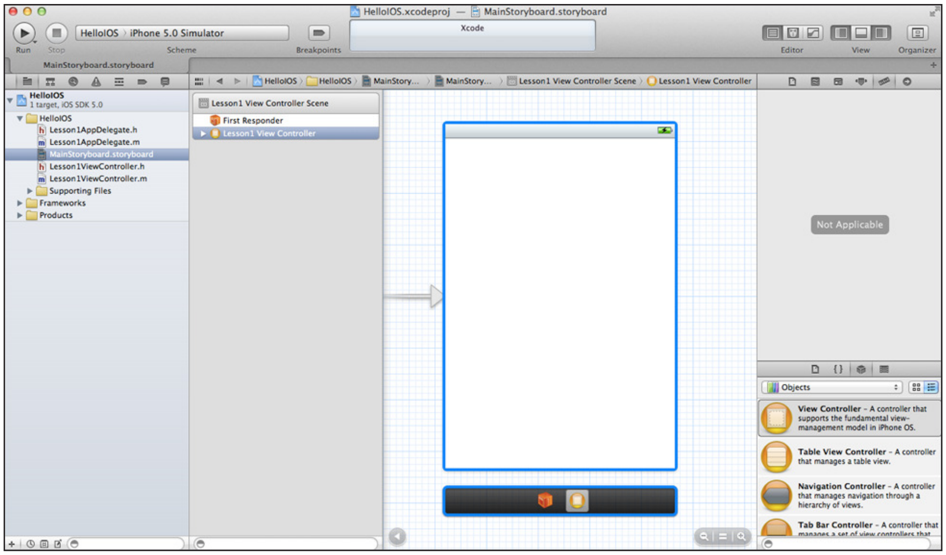

HelloIOS.xcodeproj — MainStoryboard.storyboard
Run
Stop
HelloIOS › iPhone 5.0 Simulator
Scheme
Breakpoints
Xcode
Editor
View
Organizer
MainStoryboard.storyboard
HelloIOS
1 target, iOS SDK 5.0
HelloIOS
Lesson1AppDelegate.h
Lesson1AppDelegate.m
MainStoryboard.storyboard
Lesson1ViewController.h
Lesson1ViewController.m
Supporting Files
Frameworks
Products
Lesson1 View Controller Scene
First Responder
Lesson1 View Controller
Not Applicable
Objects
View Controller - A controller that supports the fundamental view-management model in iPhone OS.
Table View Controller - A controller that manages a table view.
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Tab Bar Controller - A controller that manages a set of view controllers that

FIGURE 1-12

3. Change the background color.
    1. Click the white background area of the default scene in the storyboard and switch to the Attributes inspector by selecting the View ➪ Utilities ➪ Show Attributes Inspector menu item.
    2. Under the View section of the Attributes inspector, click once on the Background item to change the background color. This is shown in Figure 1-13.
4. Add a Label from the Xcode Object library.
    1. From the Object library, select Label and drop it onto the View (Figure 1-14).
    2. Change the text displayed in the Label to “Hello iOS” by editing the value of the Text in the Attribute inspector.
    3. Resize and reposition the Label using the mouse.
5. Set up an application icon.
    1. In Xcode, make sure the project navigator is visible.
    2. Select the `HelloIOS` project in the project navigator to bring up the project properties editor. Make sure the HelloIOS target is selected, and the Summary tab is visible (Figure 1-15).
    3. Scroll down to the App Icons section.
    4. Right-click each icon placeholder and set up an icon file using the Select File option in the popover menu. Select the `HelloIcon.png` file and the `HelloIcon@2x.png` files for the standard and Retina display icons, respectively. Both these files are located in the `resources` folder of this lesson’s `Try It` folder on the DVD.

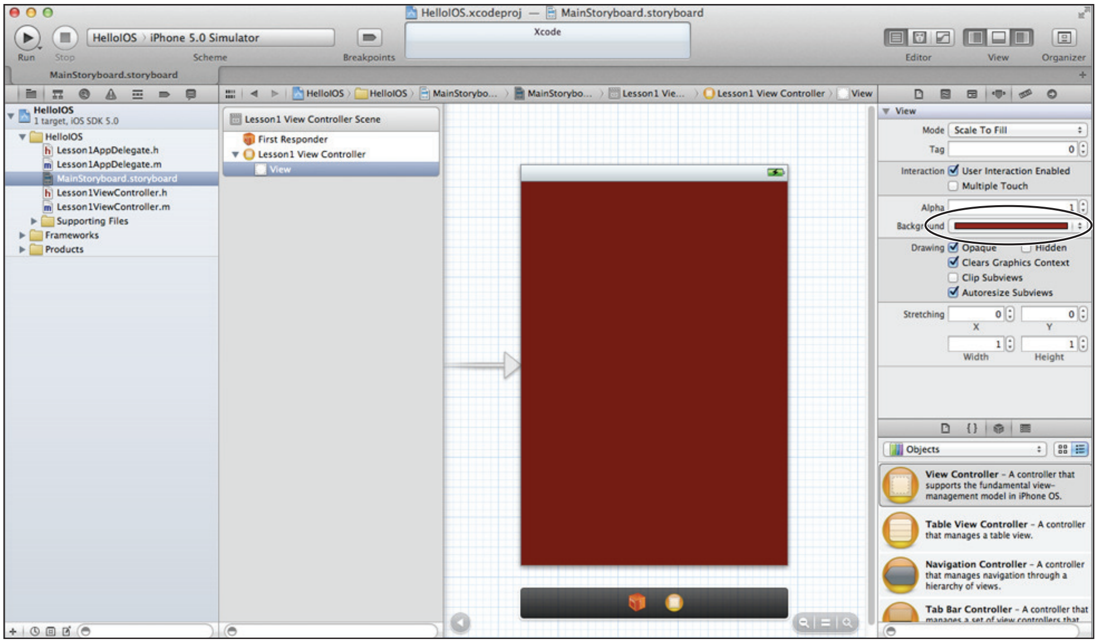

HelloIOS.xcodeproj — MainStoryboard.storyboard
Xcode
Run
Stop
Scheme
Breakpoints
Editor
View
Organizer
HelloIOS
1 target, iOS SDK 5.0
Lesson1AppDelegate.h
Lesson1AppDelegate.m
MainStoryboard.storyboard
Lesson1ViewController.h
Lesson1ViewController.m
Supporting Files
Frameworks
Products
Lesson1 View Controller Scene
First Responder
Lesson1 View Controller
View
Mode Scale To Fill
Tag 0
Interaction User Interaction Enabled
Multiple Touch
Alpha 1
Background
Drawing Opaque
Hidden
Clears Graphics Context
Clip Subviews
Autoresize Subviews
Stretching
Width
Height
Objects
View Controller – A controller that supports the fundamental view-management model in iPhone OS.
Table View Controller – A controller that manages a table view.
Navigation Controller – A controller that manages navigation through a hierarchy of views.
Tab Bar Controller – A controller that

FIGURE 1-13

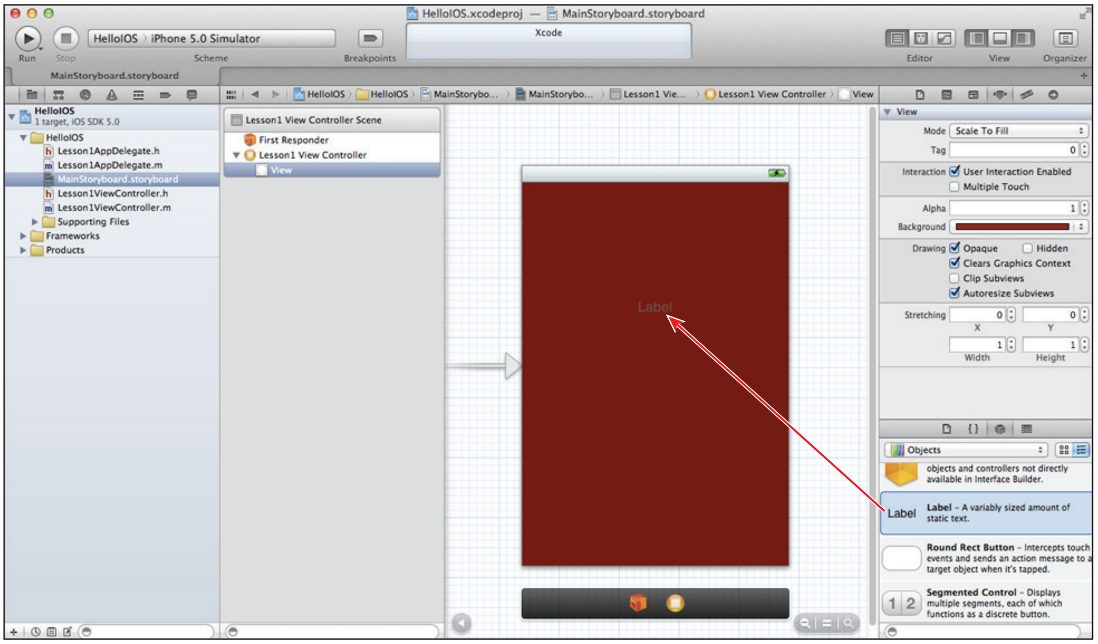
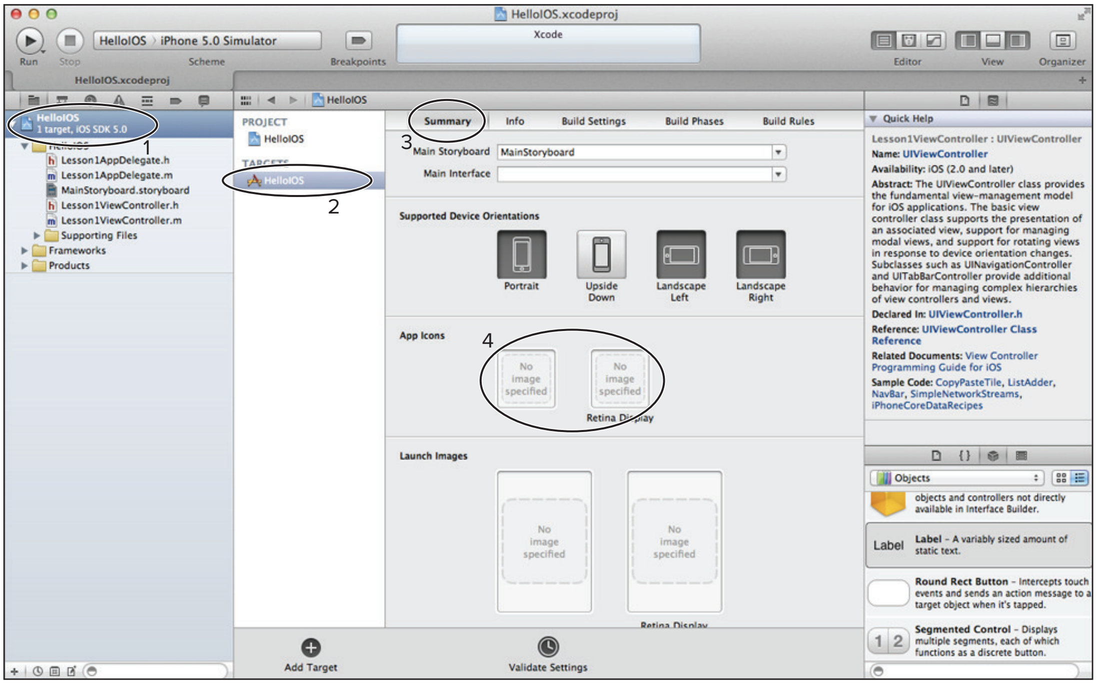

HelloIOS.xcodeproj — MainStoryboard.storyboard
Xcode
Run
Stop
Scheme
Breakpoints
Editor
View
Organizer
HelloIOS
1 target, iOS SDK 5.0
Lesson1AppDelegate.h
Lesson1AppDelegate.m
MainStoryboard.storyboard
Lesson1ViewController.h
Lesson1ViewController.m
Supporting Files
Frameworks
Products
Lesson1 View Controller Scene
First Responder
Lesson1 View Controller
View
Label
Mode Scale To Fill
Tag 0
Interaction User Interaction Enabled
Multiple Touch
Alpha 1
Background
Drawing Opaque
Hidden
Clears Graphics Context
Clip Subviews
Autoresize Subviews
Stretching
Width
Height
Objects
objects and controllers not directly available in Interface Builder.
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 1-14

6. Set up a launch image.
    1. Make sure the project navigator is visible.
    2. Select the `HelloIOS` project in the project navigator to bring up the project properties editor. Make sure the Summary tab is selected.
    3. Scroll down to the Launch Images section
    4. Right-click each placeholder and set up a launch image using the Select File option in the popover menu. Select the `HelloLaunch.png` file and the `HelloLaunch@2x.png` files for the standard and Retina display launch images, respectively. Both these files are located in the `resources` folder of this lesson's `Try It` folder on the DVD.



Lesson1ViewController : UIViewController

Name: UIViewController

Availability: iOS (2.0 and later)

Abstract: The UIViewController class provides the fundamental view-management model for iOS applications. The basic view controller class supports the presentation of an associated view, support for managing modal views, and support for rotating views in response to device orientation changes. Subclasses such as UINavigationController and UITabBarController provide additional behavior for managing complex hierarchies of view controllers and views.

Declared In: UIViewController.h

Reference: UIViewController Class Reference

Related Documents: View Controller Programming Guide for iOS

Sample Code: CopyPasteTile, ListAdder, NavBar, SimpleNetworkStreams, iPhoneCoreDataRecipes

objects and controllers not directly available in Interface Builder.

Label – A variably sized amount of static text.

Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.

Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 1-15

7. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively, you can use the Project Run menu item.

Please select Lesson 1 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

2 The iOS Simulator

The iOS Simulator is an application that runs on your Mac and allows you to test your apps without using an actual iOS device. The iOS Simulator is located in the `/Developer/Platforms/iPhoneSimulator.platform/Developer/Applications` folder and is a part of the standard iOS SDK installation. When you run your app in Xcode, you have the choice of launching it in the simulator or an actual device. If you choose to launch it in the simulator, Xcode will launch the iOS Simulator automatically.

FEATURES OF THE IOS SIMULATOR

You can use the iOS Simulator to simulate different device (iPad, iPhone 3GS, iPhone 4) and SDK versions. You can change the iOS version being simulated using the Hardware ➪ Version menu (Figure 2-1). The actual list of options you see here will depend on the different versions of the iOS SDK that you have installed on your Mac.

To switch devices use the Hardware ➪ Device menu. You can choose between an iPhone (iPhone 3GS), iPhone Retina (iPhone 4), or an iPad. Figure 2-2 shows the iPhone4 and iPad simulators.

You can rotate the simulator by using the Rotate Left or Rotate Right menu items from the Hardware menu (Figure 2-3).

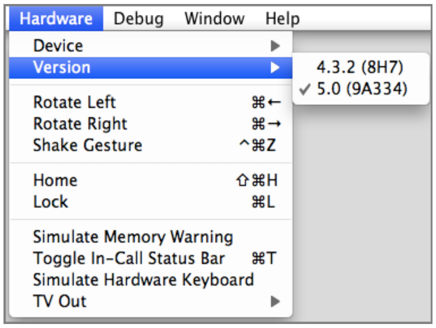


FIGURE 2-1

The iOS Simulator allows you to simulate a variety of one and two-finger multi-touch gestures. Single-finger gestures such as taps and swipes can be performed by clicking and dragging with the mouse. The only two-finger gesture that you can simulate is the pinch. To do so, hold down the Option key on your keyboard while clicking and dragging with the mouse in the simulator window. Shake gestures can be performed by using the Hardware ➪ Shake Gesture menu item.

If you are developing an app that requires location data, you can now use the iOS Simulator to simulate a test location while you are running your application within the simulator. Select the Debug ➪ Location ➪ Custom Location menu item (Figure 2-4) to specify a latitude and longitude pair. Creating location-based applications is covered in Lessons 34 and 35.

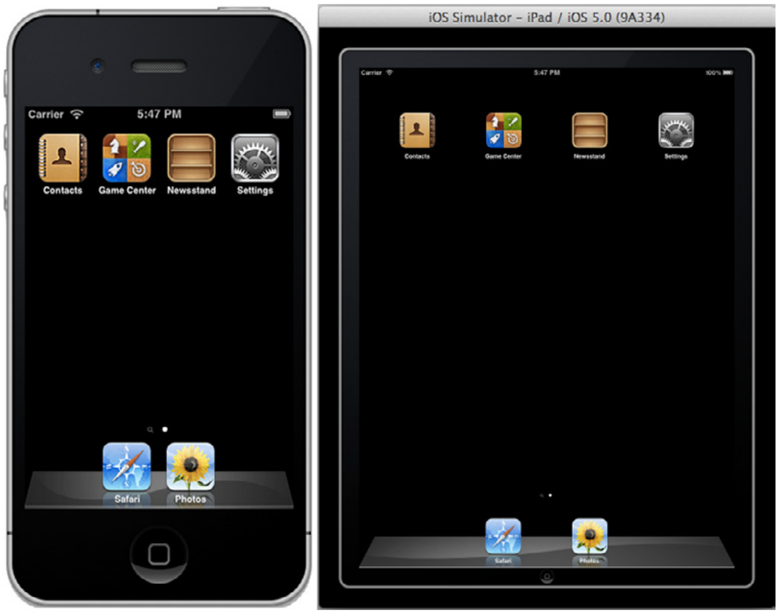


FIGURE 2-2

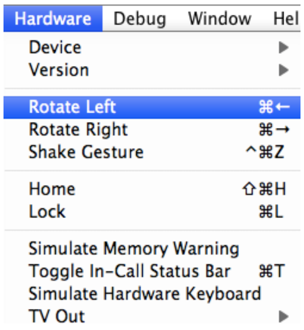


FIGURE 2-3

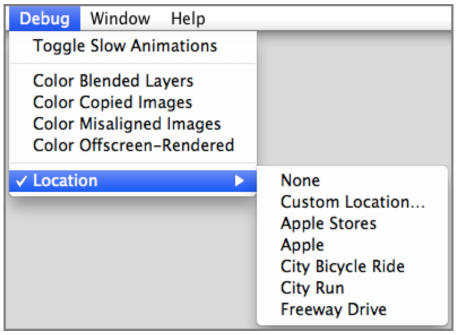


FIGURE 2-4

The simulator can also simulate changing locations. This is particularly useful if your app is designed to be used while on the move. From the Debug ➪ Location menu, you can select from a list of prerecorded location sets. The simulator will then periodically cycle between the locations in the selected set. The sets are:

- Apple Stores
- City Bicycle Ride

- City Run
- Freeway Drive

If you are developing an app that allows the user to print something and do not have an AirPrint-compatible printer, you can simulate a printer using the iOS Printer Simulator. The Printer Simulator is not loaded automatically when you switch on your Mac; to load it simply select the File ➪ Open Printer Simulator menu item.

Installing and Uninstalling Applications

To install an application to the iOS Simulator you need to open its corresponding `.xcodeproj` file in Xcode and click the Run button in the Xcode toolbar.

You cannot delete the default iOS Simulator applications (such as Photos, Settings, Game Center, Safari, and so on). To uninstall (delete) one of your applications from the iOS Simulator, click and hold the mouse button down on the icon of the app until all the icons start to wiggle. Once they start to wiggle you will notice an X button on the top-left corner of each icon (Figure 2-5).

Release the mouse button if you are still holding it down; the icons will still continue to wiggle. Click the X button on the icon of the app you want to delete. An alert window will appear asking you to confirm this action (Figure 2-6).

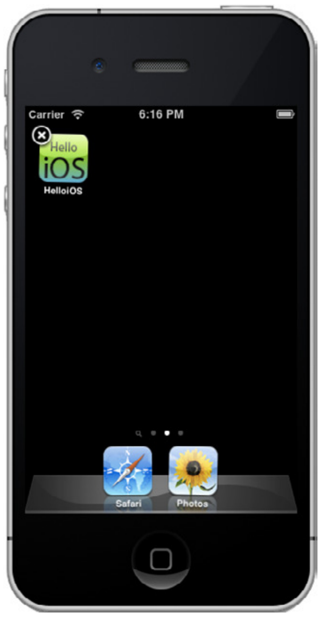
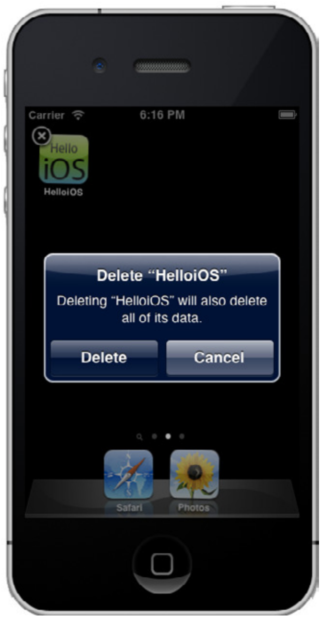


FIGURE 2-5



Delete "HelloiOS"

Deleting "HelloiOS" will also delete all of its data.

Delete | Cancel

FIGURE 2-6

LIMITATIONS OF THE IOS SIMULATOR

As good as the iOS Simulator may be, it has its limitations. For starters you cannot make calls, send or receive text messages, or install apps from the App Store.

The performance of the iOS Simulator depends on the speed of your Mac, and in certain cases your application may appear to execute much faster on your Mac (in the iOS Simulator) than it does on the real device.

Accelerometer, camera, and microphone functions are not supported in the iOS Simulator. If you are developing OpenGL/ES-based applications, you should keep in mind that several OpenGL/ES functions are not supported on the iOS Simulator.

The iOS Simulator is a useful tool to test your apps but it is definitely not a replacement for testing on a real device.

3 A Tour of Xcode

Xcode is Apple's IDE (Integrated Development Environment), which you use to create iOS applications. The word "integrated" refers to the fact that Xcode brings together several different tools into a single application.

Xcode contains several tools, but the ones you'll use most of the time are the source code editor, debugger, and Interface Builder. The current version of Xcode when this book is being written is 4.2.

In this lesson you explore various commonly used features of Xcode.

THE WELCOME SCREEN

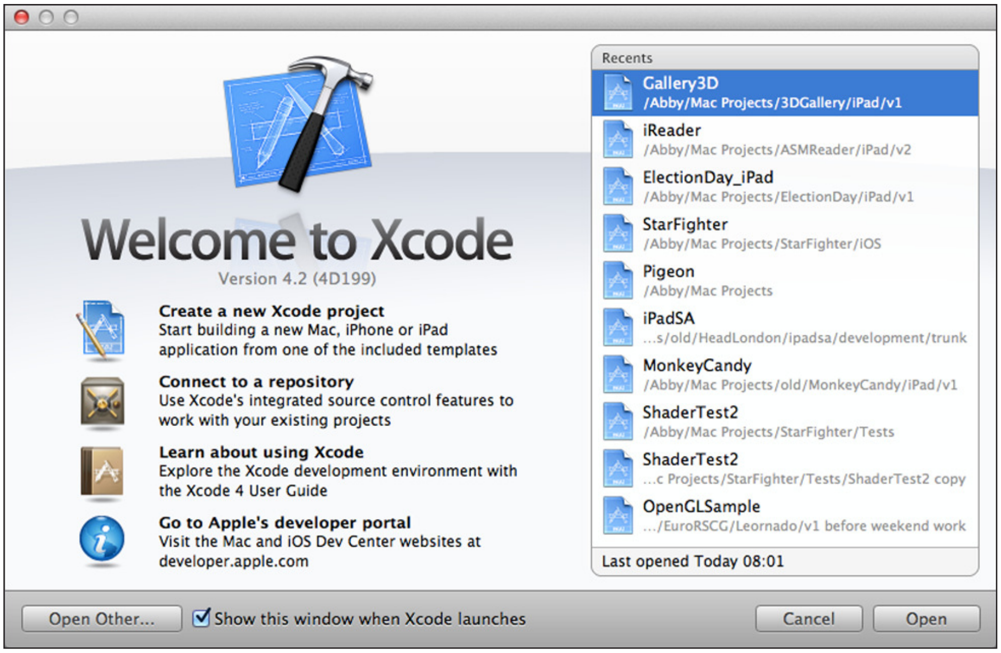
When you launch Xcode, you are presented with the welcome dialog (Figure 3-1). You can use the welcome dialog to quickly create a new project, connect to a source code repository, open a recently used project, and access documentation.

The first step toward creating an iOS application is the creation of an appropriate project in Xcode. An Xcode project has the file extension `.xcodeproj`, and tells the Xcode IDE (among other things) the name of your application, what kind of application it is (iPhone/iPad/Universal), and where to find the code files and resources required to create the application.

SELECTING A PROJECT TEMPLATE

When you create a new project in Xcode, you first need to select a template on which to base the project. Xcode templates contain files that you need to start developing a new application. Xcode provides a list of project templates to select from (Figure 3-2).

The Xcode template window has two categories of templates to choose from. In this book you create iOS applications, and hence need to make sure the iOS template category is selected.



Welcome to Xcode

Version 4.2 (4D199)

Create a new Xcode project
Start building a new Mac, iPhone or iPad application from one of the included templates

Connect to a repository
Use Xcode's integrated source control features to work with your existing projects

Learn about using Xcode
Explore the Xcode development environment with the Xcode 4 User Guide

Go to Apple's developer portal
Visit the Mac and iOS Dev Center websites at developer.apple.com

Recents

Gallery3D
/Abby/Mac Projects/3DGallery/iPad/v1

iReader
/Abby/Mac Projects/ASMReader/iPad/v2

ElectionDay_iPad
/Abby/Mac Projects/ElectionDay/iPad/v1

StarFighter
/Abby/Mac Projects/StarFighter/iOS

Pigeon
/Abby/Mac Projects

iPadSA
...s/old/HeadLondon/ipadsa/development/trunk

MonkeyCandy
/Abby/Mac Projects/old/MonkeyCandy/iPad/v1

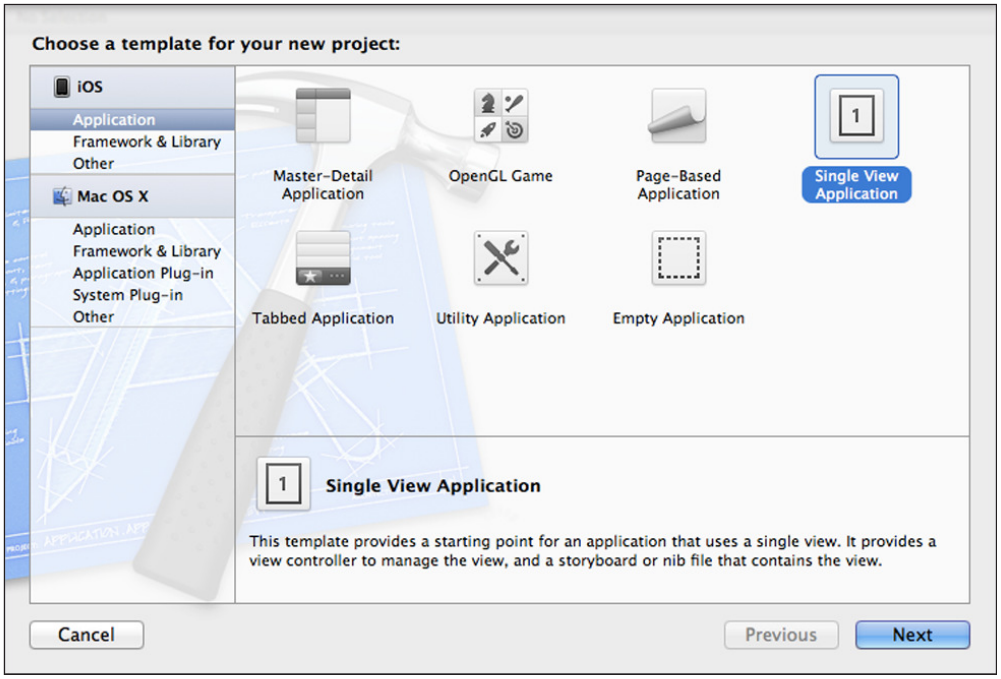
ShaderTest2
/Abby/Mac Projects/StarFighter/Tests

ShaderTest2
...c Projects/StarFighter/Tests/ShaderTest2 copy

OpenGLSample
.../EuroRSCG/Leornado/v1 before weekend work

Last opened Today 08:01

FIGURE 3-1



Choose a template for your new project:

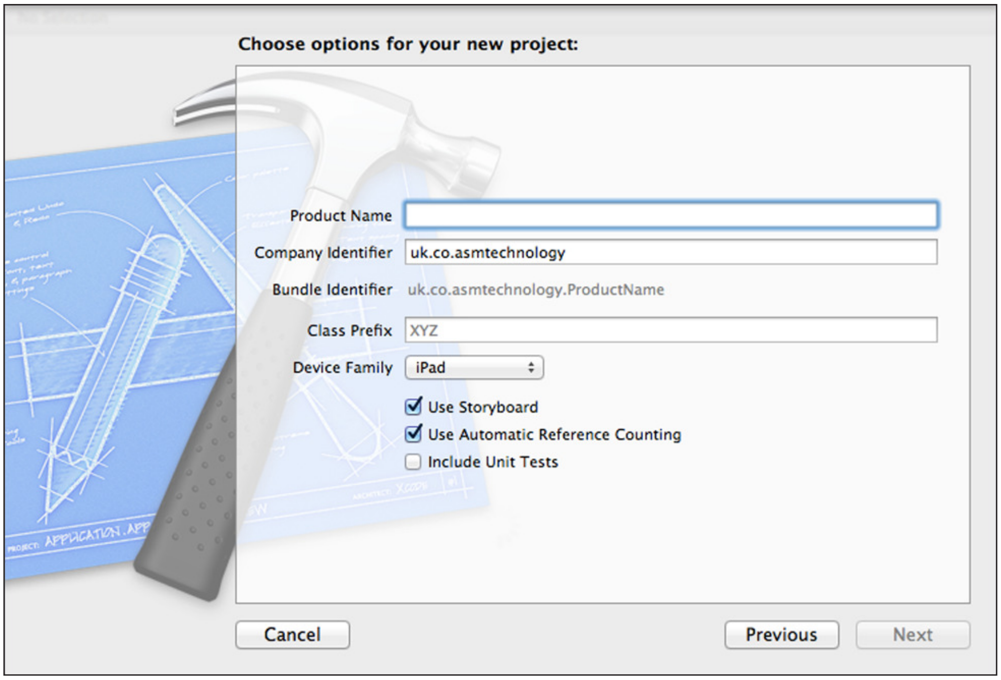
Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

FIGURE 3-2

SETTING UP PROJECT OPTIONS

After you have selected a suitable template, Xcode presents the project options dialog box (Figure 3-3).


Choose options for your new project:

Product Name:
Company Identifier: uk.co.asmtechnology
Bundle Identifier: uk.co.asmtechnology.ProductName
Class Prefix: XYZ
Device Family: iPad

☑ Use Storyboard
☑ Use Automatic Reference Counting
☐ Include Unit Tests

FIGURE 3-3

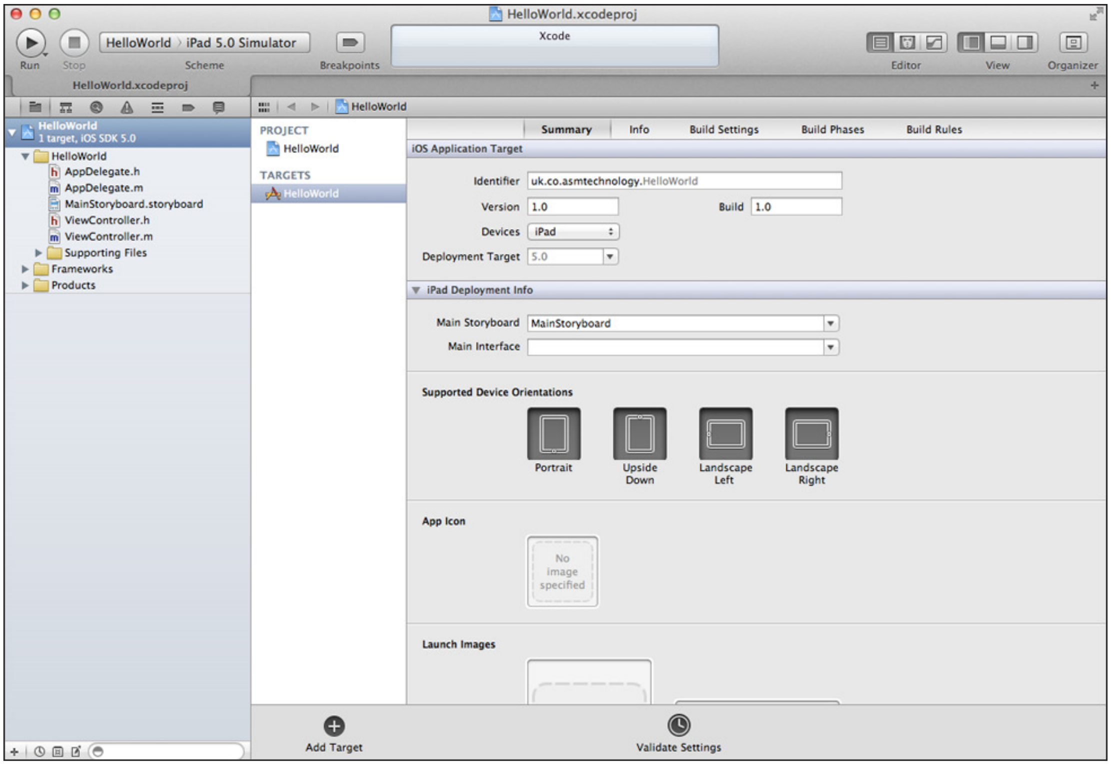
This is where you provide the name of the project, and specify the target device—iPhone, iPad, or Universal. To uniquely identify your application on the iTunes store (and on an iOS device), each project must have a unique identifier. This identifier is known as a *Bundle Identifier*, and is created by combining the name of the project along with a company identifier that you provide in the project options dialog. It is best to provide your website domain name in reversed format as the company identifier, because domain names are guaranteed to be globally unique.

Xcode can also create unit tests for your project. A unit test is a small program that is written for the sole purpose of testing another program. Unit testing is beyond the scope of this book, so just uncheck the Include Unit Tests option in the dialog box.

Ensure the Use Storyboard and Use Automatic Reference Counting checkboxes are checked before you click the Next button. The lessons in this book make use of storyboards and automatic reference counting. When you click Next, Xcode asks you to provide a location on your Mac where you would like to save the new project.

AN OVERVIEW OF THE XCODE IDE

The Xcode IDE features a single window, called the workspace window (Figure 3-4), where you get most of your work done.



iOS Application Target

Identifier: uk.co.asmtechnology.HelloWorld
Version: 1.0
Build: 1.0
Devices: iPad
Deployment Target: 5.0

iPad Deployment Info

Main Storyboard: MainStoryboard
Main Interface:

Supported Device Orientations

App Icon

Launch Images

FIGURE 3-4

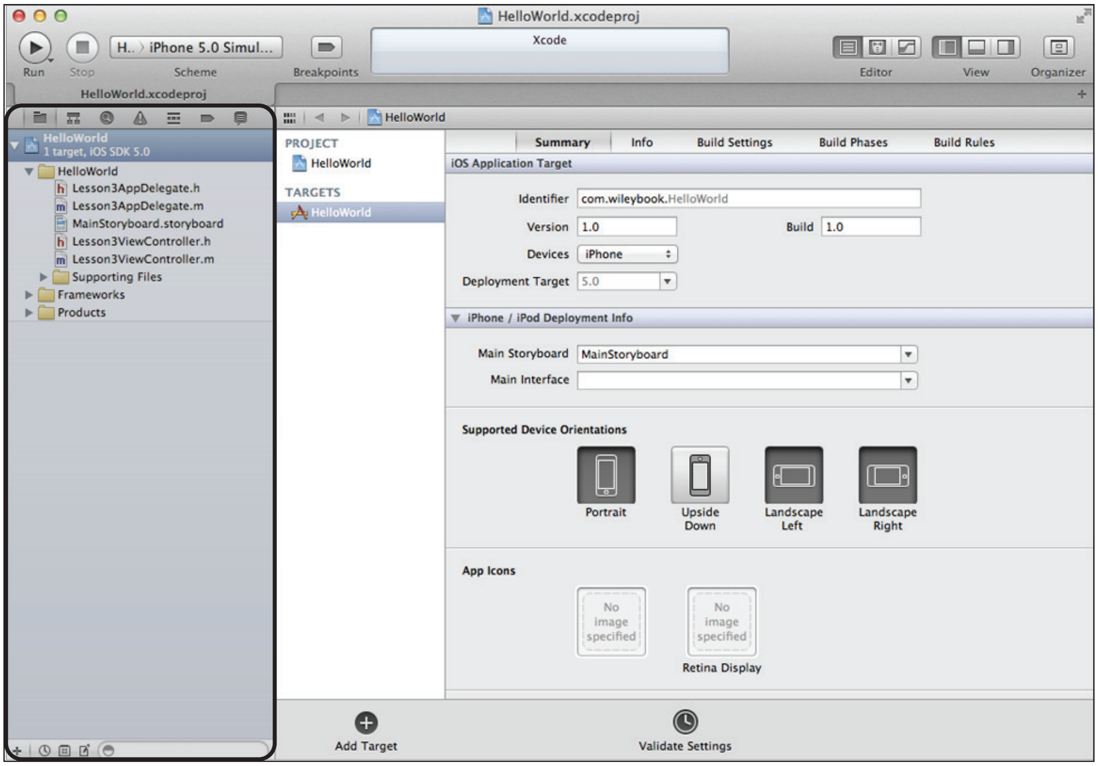
The Navigator Area

The left side of the workspace window is the navigator area (Figure 3-5).

The navigator area consists of seven tabs; each of these tabs (called navigators) shows different aspects of the same project. You can switch between navigators using the navigator selector bar at the top of the navigator area (Figure 3-6).

The Project Navigator

The project navigator (Figure 3-7) shows the contents of your project. Individual files are organized within groups that are represented as folders in a tree structure. The top-level node of this tree structure represents the project itself. These groups are purely logical and provide a convenient way to organize the contents of your project. A group may not necessarily correspond to actual folders on your hard drive.



HelloWorld
1 target, iOS SDK 5.0
HelloWorld
Lesson3AppDelegate.h
Lesson3AppDelegate.m
MainStoryboard.storyboard
Lesson3ViewController.h
Lesson3ViewController.m
Supporting Files
Frameworks
Products

PROJECT
HelloWorld
TARGETS
HelloWorld

iOS Application Target

Identifier: com.wileybook.HelloWorld
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0

iPhone / iPod Deployment Info

Main Storyboard: MainStoryboard
Main Interface:

Supported Device Orientations: Portrait, Upside Down, Landscape Left, Landscape Right

App Icons: No image specified; No image specified (Retina Display)

FIGURE 3-5

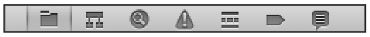
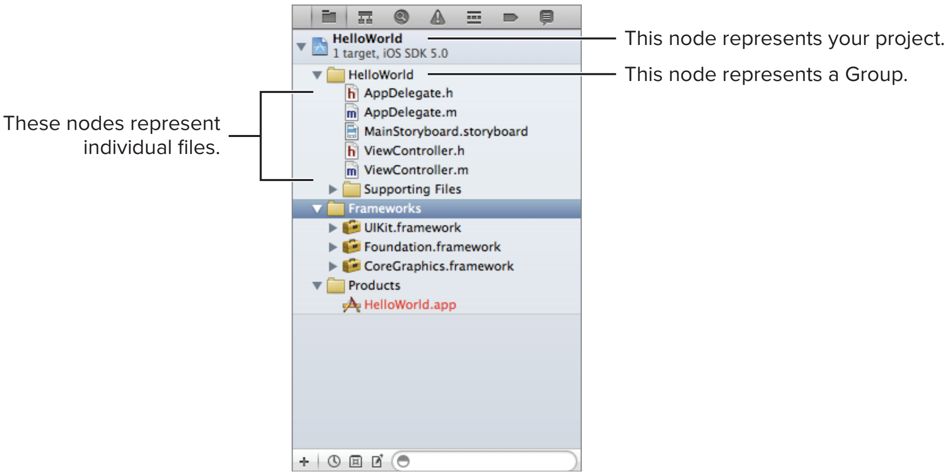


FIGURE 3-6


This node represents your project.
This node represents a Group.
These nodes represent individual files.

HelloWorld
1 target, iOS SDK 5.0
HelloWorld
AppDelegate.h
AppDelegate.m
MainStoryboard.storyboard
ViewController.h
ViewController.m
Supporting Files
Frameworks
UIKit.framework
Foundation.framework
CoreGraphics.framework
Products
HelloWorld.app

FIGURE 3-7

In most cases, you will work with a single project at a time in the Xcode workspace window; however, it is possible to open multiple projects in the workspace window using a workspace file. A workspace file has the file extension `.xcworkspace` and contains references to one or more project files. You will not be creating workspaces in this book; however, if you were to open a workspace file, the workspace window would display information on multiple projects contained within the workspace (Figure 3-8).

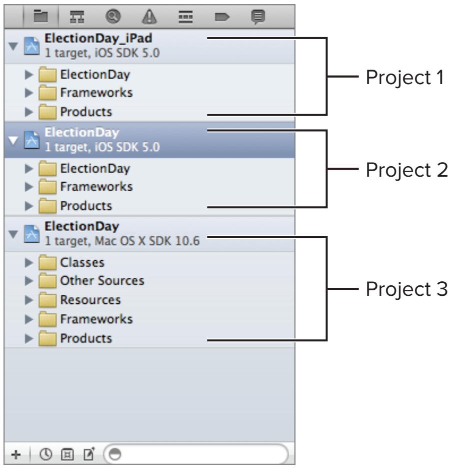


FIGURE 3-8

To create a new group, right-click an existing node in the project navigator and select New Group from the context menu. You can move files between groups by using simple drag-and-drop operations in the project navigator. If the groups in the project navigator correspond to actual folders on your Mac, then moving things around in the project navigator would not move the corresponding files into new locations on your Mac.

To delete a file, simply select the item and hit the backspace key on your keyboard. Xcode then asks you if you intended to delete the actual file from your Mac or just remove the reference from the project. The process of deleting a group is similar to that of a file; keep in mind that deleting a group deletes any files within that group.

To learn more about the project navigator read the Project Navigator Help document at `http://developer.apple.com/library/ios/#recipes/xcode_help-structure_navigator/_index.html`.

Although you are free to create any number of groups in your project, Xcode creates three groups for you by default (Figure 3-9). The first is the project group, which is usually the same name as the project itself, and this is where all your code and resources will go.

The second is the Frameworks group. Frameworks are libraries of code supplied by Apple to use in your application. The third is the Products group, which contains the actual iOS application.

At the bottom of the project navigator is a set of icons. You can use these icons to filter what is displayed in the project navigator based on certain criteria (Figure 3-10).

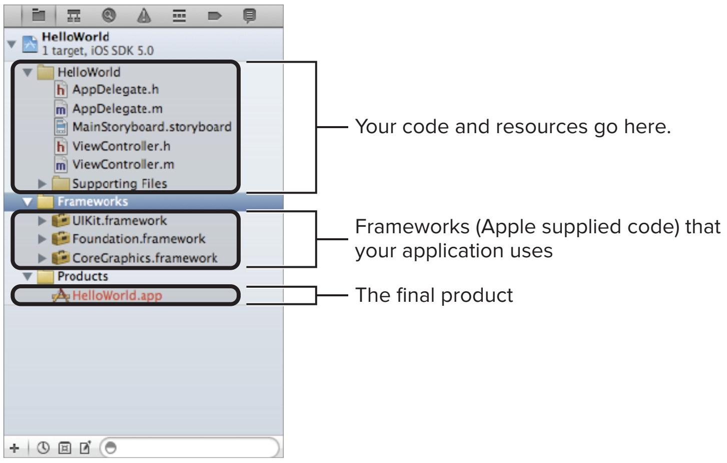


FIGURE 3-9

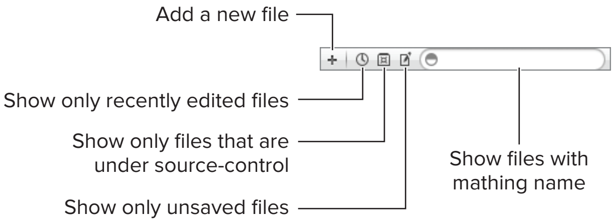

Add a new file
Show only recently edited files
Show only files that are under source-control
Show only unsaved files
Show files with mathing name

FIGURE 3-10

The Symbol Navigator

The symbol navigator (Figure 3-11) shows the classes in your project along with their methods and member variables. A top-level node in a tree-like structure represents each class. Expanding the class node reveals all its member variables and methods.

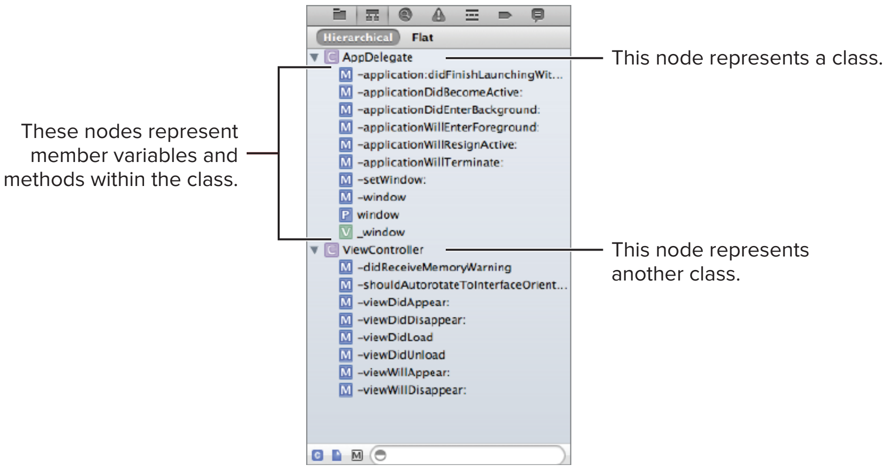
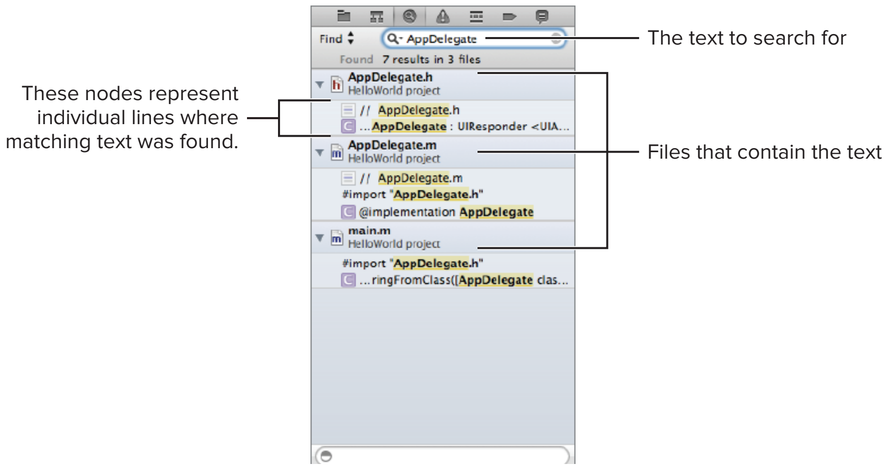

Hierarchical
Flat
AppDelegate
-application:didFinishLaunchingWit...
-applicationDidBecomeActive:
-applicationDidEnterBackground:
-applicationWillEnterForeground:
-applicationWillResignActive:
-applicationWillTerminate:
-setWindow:
-window
window
_window
ViewController
-didReceiveMemoryWarning
-shouldAutorotateToInterfaceOrient...
-viewDidAppear:
-viewDidDisappear:
-viewDidLoad
-viewDidUnload
-viewWillAppear:
-viewWillDisappear:
This node represents a class.
These nodes represent member variables and methods within the class.
This node represents another class.

FIGURE 3-11

The Search Navigator

The search navigator (Figure 3-12) lets you find all occurrences of some text, across all files of the project.


These nodes represent individual lines where matching text was found.
The text to search for
Files that contain the text

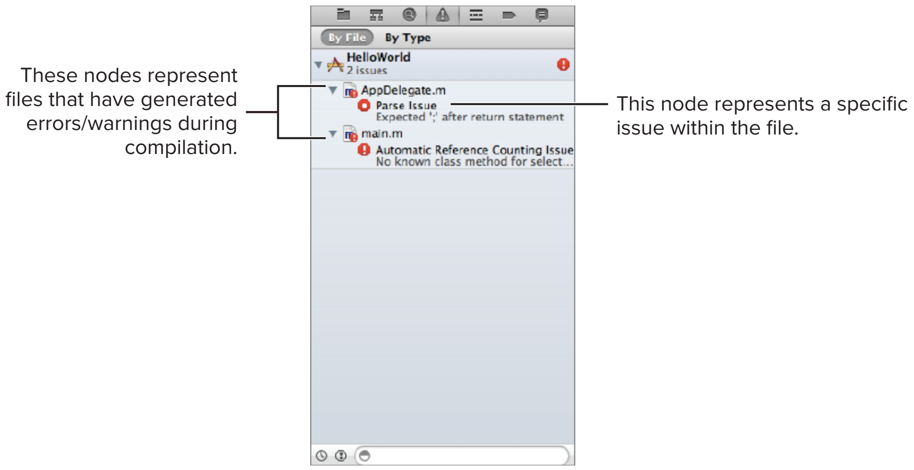
Find AppDelegate
Found 7 results in 3 files
AppDelegate.h
HelloWorld project
// AppDelegate.h
...AppDelegate : UIResponder <UIA...
AppDelegate.m
HelloWorld project
// AppDelegate.m
#import "AppDelegate.h"
@implementation AppDelegate
main.m
HelloWorld project
#import "AppDelegate.h"
...ringFromClass([AppDelegate clas...

FIGURE 3-12

A root-level node in a tree represents each file that has one or more occurrences of matching text. Expanding the node reveals the exact positions within that file where these matches were made.

The Issue Navigator

The issue navigator (Figure 3-13) lists all compile-time errors and warnings in your project. While compiling a file, Xcode raises an issue each time it finds a problem with the file. Severe show-stopping issues are flagged as errors, whereas less severe issues are flagged as warnings.


These nodes represent files that have generated errors/warnings during compilation.
This node represents a specific issue within the file.

By File By Type
HelloWorld
2 issues
AppDelegate.m
Parse Issue
Expected ';' after return statement
main.m
Automatic Reference Counting Issue
No known class method for select...

FIGURE 3-13

Each file having one or more errors/warnings is represented by a root-level node in a tree-like structure. Expanding the node reveals the exact positions within that file where these errors/warnings were encountered.

The Debug Navigator

The debug navigator is used during an active debugging session and lists the call stack for each running thread. Debugging is an advanced topic and is not covered in this book.

The Breakpoint Navigator

The breakpoint navigator lists all breakpoints in your code, and allows you to manage them. A breakpoint is an intentional pause-point that you can set in your project. When the app is being executed, Xcode interrupts the execution of the application when it encounters one of these pause-points and transfers control to the debugger. This is extremely useful when trying to figure out why a particular piece of code does not work and you want to inspect the values of variables and content of memory. Breakpoints and the debugger work only when the application is being executed in a special debug mode. Breakpoints and debugging are advanced topics, and are not covered in this book.

The Log Navigator

The log navigator shows you a history of build logs and console debug sessions. Building is the complete process of creating an executable application from your source code files. Compilation is a part of the build process. Each time you build a new executable, Xcode creates a build log that contains, among other things, a list of files that were compiled.

The Editor Area

The right side of the workspace window is the editor area (Figure 3-14). Xcode includes editors for many file types, including source code, user interface files, XML files, and project settings, to name a few.

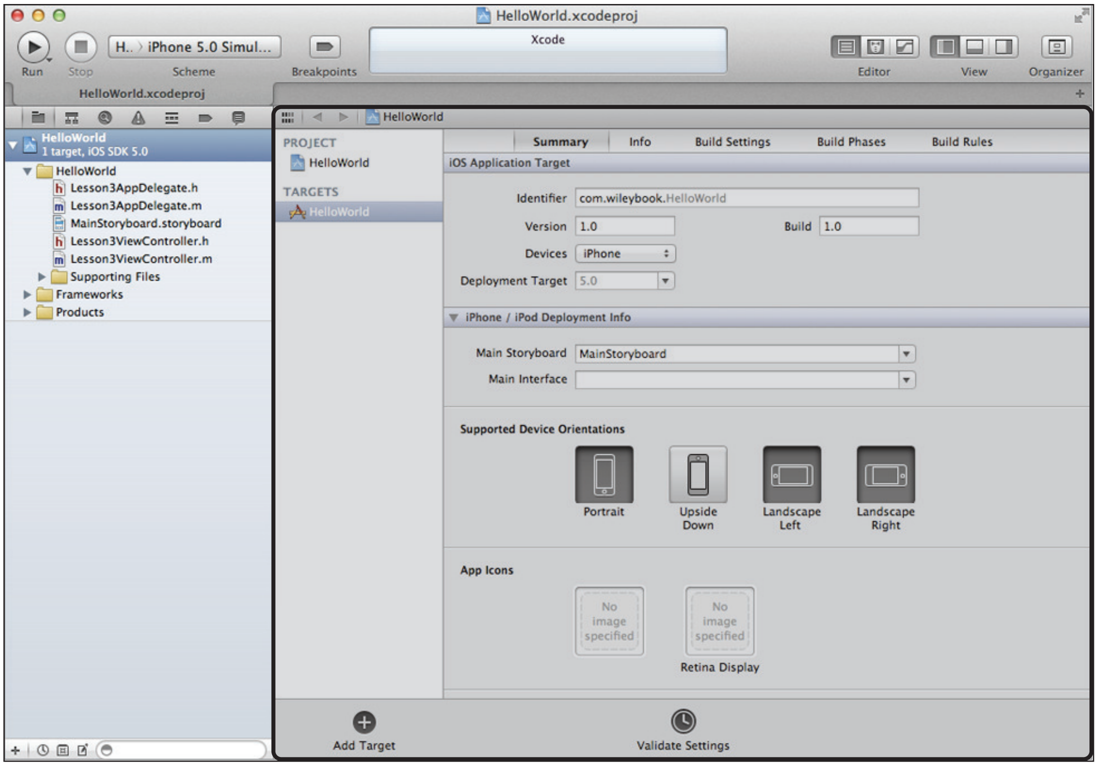
The content of the editor area depends on the current selection in the navigator area. When you select a file in the navigator area, Xcode tries to find an appropriate editor for that file type. If it can't find one, it opens the file using Quick Look (which is also used by the Finder).

Jump Bars

At the top of the editor area is the jump bar (Figure 3-15). The jump bar displays the path to the current file being edited and can be used to quickly select another file in the workspace. The jump bar also has back and forward buttons to move through a history of files edited.

Each element in the path displayed in the jump bar is a pop-up menu (Figure 3-16) that you can use to navigate around your project.

The contents of the jump bar depend on the type of file you're viewing. When editing a user interface file, for example, the jump bar enables you to navigate to individual interface elements.



iOS Application Target

Identifier: com.wileybook.HelloWorld

Version: 1.0 Build: 1.0

Devices: iPhone

Deployment Target: 5.0

iPhone / iPod Deployment Info

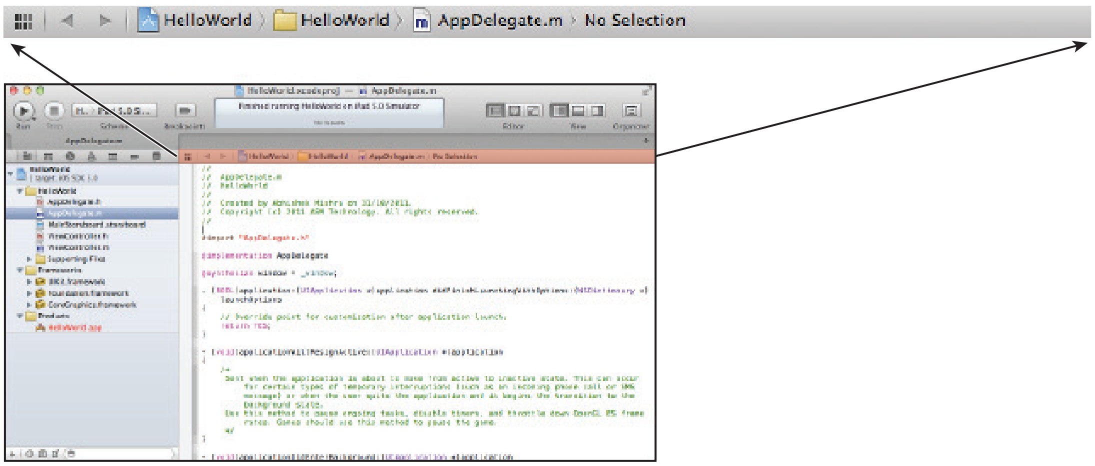
Main Storyboard: MainStoryboard

Main Interface:

Supported Device Orientations: Portrait, Upside Down, Landscape Left, Landscape Right

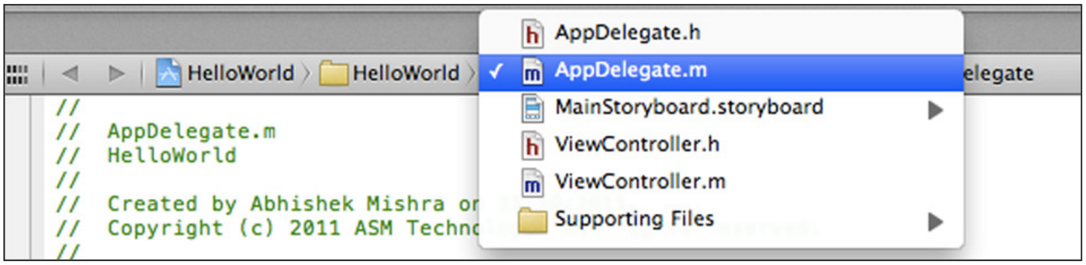
App Icons: No image specified; No image specified — Retina Display

FIGURE 3-14



```
//
//  AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra on 31/10/2011.
//  Copyright (c) 2011 ASM Technology. All rights reserved.
//

#import "AppDelegate.h"

@implementation AppDelegate

@synthesize window = _window;

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

- (void)applicationWillResignActive:(UIApplication *)application
{
    /*
     Sent when the application is about to move from active to inactive state. This can occur
         for certain types of temporary interruptions (such as an incoming phone call or SMS
         message) or when the user quits the application and it begins the transition to the
         background state.
     Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame
         rates. Games should use this method to pause the game.
     */
}

- (void)applicationDidEnterBackground:(UIApplication *)application
```

FIGURE 3-15



```
//
//  AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra or
//  Copyright (c) 2011 ASM Techno
//
```

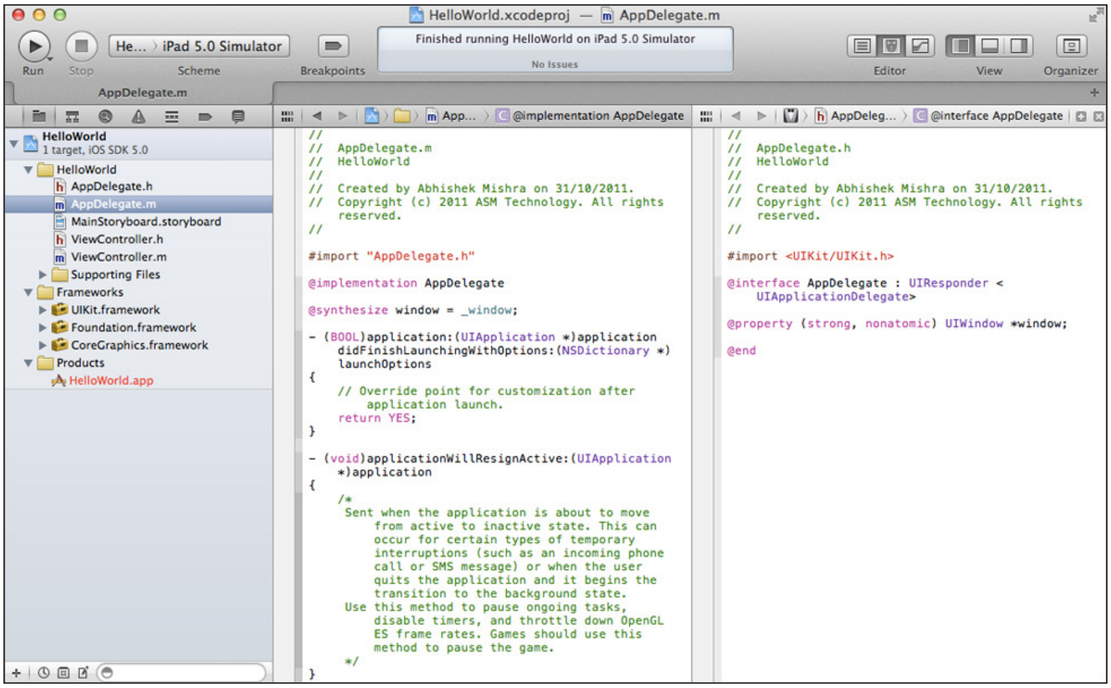
FIGURE 3-16

The Source Editor

When you select a source-code file in the navigator area, or a text/XML file, Xcode uses the source editor to open the file. This is the editor with which you will spend most of your time when you write your code. The source editor has several helpful features, such as syntax highlighting and code completion hints. You can configure individual features of the source editor using Xcode preferences.

The Assistant Editor

The assistant editor (Figure 3-17) was introduced in Xcode 4 and enables you to view multiple files side-by-side.



```
//
//  AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra on 31/10/2011.
//  Copyright (c) 2011 ASM Technology. All rights
    reserved.
//

#import "AppDelegate.h"

@implementation AppDelegate

@synthesize window = _window;

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

- (void)applicationWillResignActive:(UIApplication
    *)application
{
    /*
     Sent when the application is about to move
         from active to inactive state. This can
         occur for certain types of temporary
         interruptions (such as an incoming phone
         call or SMS message) or when the user
         quits the application and it begins the
         transition to the background state.
     Use this method to pause ongoing tasks,
         disable timers, and throttle down OpenGL
         ES frame rates. Games should use this
         method to pause the game.
     */
}
```

```
//
//  AppDelegate.h
//  HelloWorld
//
//  Created by Abhishek Mishra on 31/10/2011.
//  Copyright (c) 2011 ASM Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <
    UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

FIGURE 3-17

The assistant editor is not visible by default, and can be accessed by using the editor selector buttons (Figure 3-18) in the Xcode toolbar. Option-clicking a file in the project navigator or symbol navigator opens it in the assistant editor.

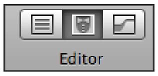
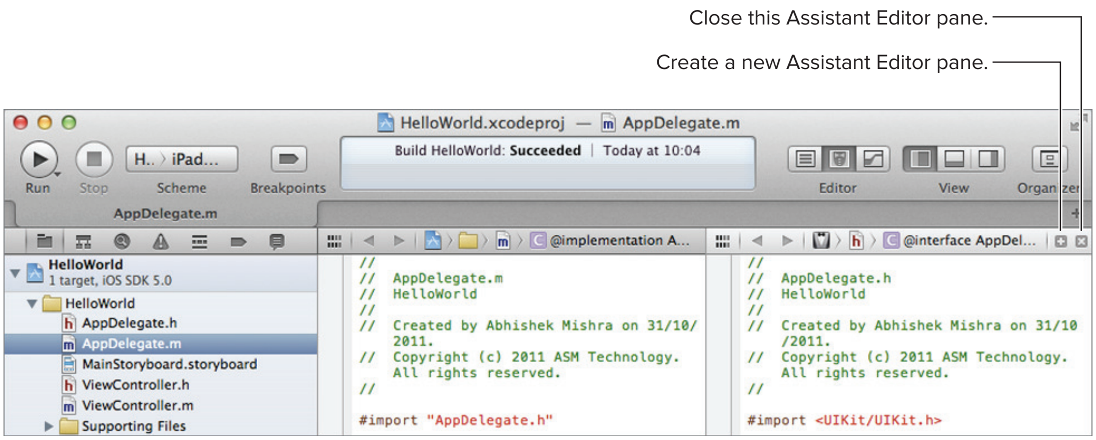


FIGURE 3-18

You can create additional assistant editor panes by using the + and x buttons (Figure 3-19). You can also use the jump bar in the additional panes to specify which file to show in each pane; however, it is often helpful to let Xcode find a related file for you. The assistant editor can find, for example, the header file that corresponds to the source code file you're viewing.



```
//
//  AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra on 31/10/
    2011.
//  Copyright (c) 2011 ASM Technology.
    All rights reserved.
//

#import "AppDelegate.h"
```

```
//
//  AppDelegate.h
//  HelloWorld
//
//  Created by Abhishek Mishra on 31/10
    /2011.
//  Copyright (c) 2011 ASM Technology.
    All rights reserved.
//

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

FIGURE 3-19

The Version Editor

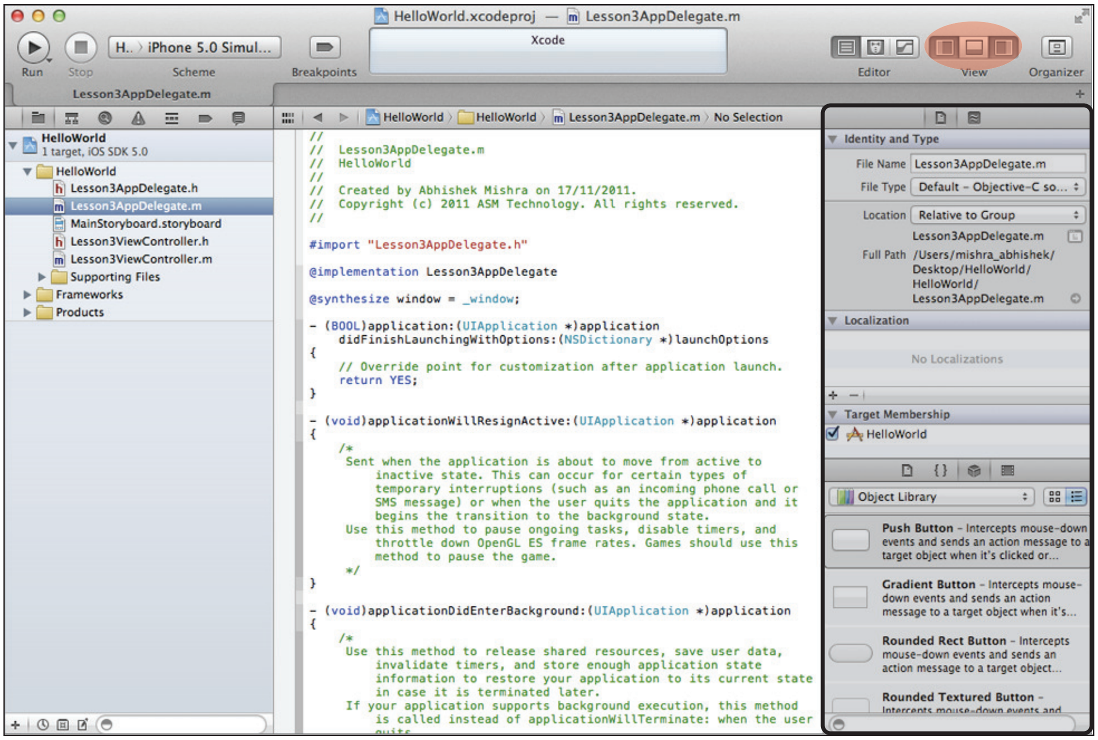
If your project is under version control, you can use the version editor to compare the current version of a file with a previous version. Like the assistant editor, the version editor is not visible by default, and can be accessed by using the editor selector buttons in the Xcode toolbar. Version control is not covered in this book.

The Utility Area

The utility area (Figure 3-20) supplements the editor area. You can display it by choosing the View Utilities ➪ Show Utilities menu item or by clicking the utility button in the toolbar.

The Inspector Area

The top portion of the utility area contains the inspector area (Figure 3-21). Like the navigator area, the inspector area also contains multiple tabs that can be switched using a selector bar at the top of the inspector area.



```
//
//  Lesson3AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra on 17/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights reserved.
//

#import "Lesson3AppDelegate.h"

@implementation Lesson3AppDelegate

@synthesize window = _window;

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

- (void)applicationWillResignActive:(UIApplication *)application
{
    /*
     Sent when the application is about to move from active to
         inactive state. This can occur for certain types of
         temporary interruptions (such as an incoming phone call or
         SMS message) or when the user quits the application and it
         begins the transition to the background state.
     Use this method to pause ongoing tasks, disable timers, and
         throttle down OpenGL ES frame rates. Games should use this
         method to pause the game.
     */
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /*
     Use this method to release shared resources, save user data,
         invalidate timers, and store enough application state
         information to restore your application to its current state
         in case it is terminated later.
     If your application supports background execution, this method
         is called instead of applicationWillTerminate: when the user
```

Identity and Type
File Name: Lesson3AppDelegate.m
File Type: Default – Objective-C so…
Location: Relative to Group
Lesson3AppDelegate.m
Full Path: /Users/mishra_abhishek/Desktop/HelloWorld/HelloWorld/Lesson3AppDelegate.m

Localization
No Localizations

Target Membership
✓ HelloWorld

Object Library

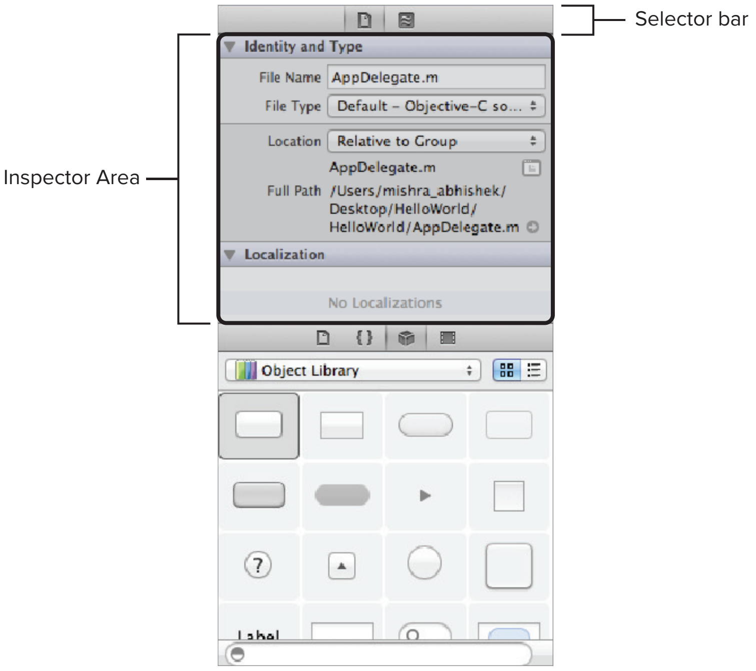
Push Button – Intercepts mouse-down events and sends an action message to a target object when it's clicked or…

Gradient Button – Intercepts mouse-down events and sends an action message to a target object when it's…

Rounded Rect Button – Intercepts mouse-down events and sends an action message to a target object…

Rounded Textured Button –

FIGURE 3-20



Identity and Type
File Name: AppDelegate.m
File Type: Default – Objective-C so…
Location: Relative to Group
AppDelegate.m
Full Path: /Users/mishra_abhishek/Desktop/HelloWorld/HelloWorld/AppDelegate.m

Localization
No Localizations

FIGURE 3-21

The number of tabs available depends on the currently selected item in the project navigator. Regardless of what is selected in the project navigator, the first two tabs are always the file inspector and the quick help inspector. The file inspector provides access to the properties of the current file. The quick help inspector provides a short description of the current file.

The Library Area

The bottom portion of the utility area contains the library area (Figure 3-22). This area contains a library of file templates, user interface objects, and code snippets that you can use in your applications.

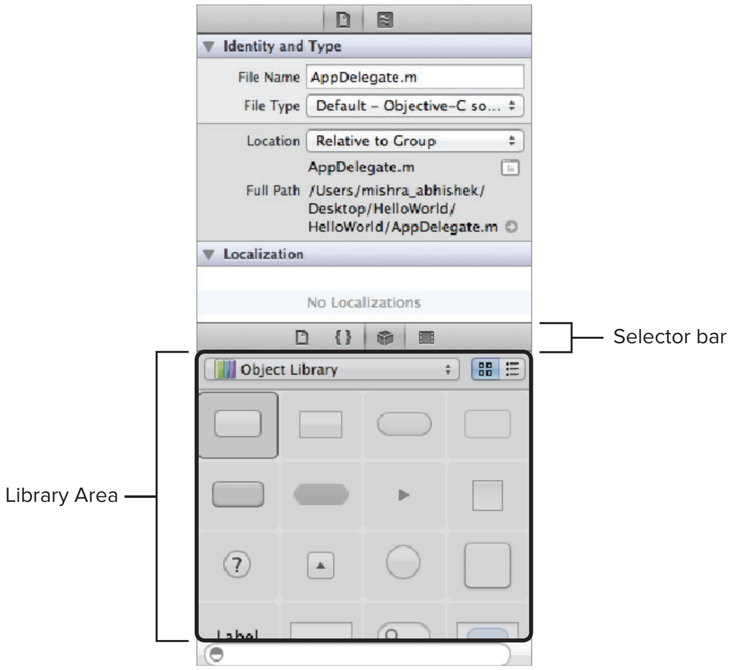

Identity and Type
File Name: AppDelegate.m
File Type: Default – Objective-C so...
Location: Relative to Group
AppDelegate.m
Full Path: /Users/mishra_abhishek/Desktop/HelloWorld/HelloWorld/AppDelegate.m
Localization
No Localizations
Object Library

FIGURE 3-22

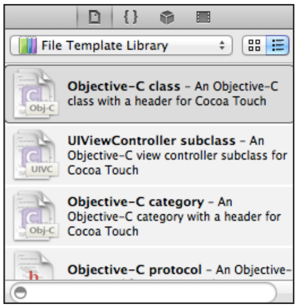
The library area also provides a convenient method to access all the media files in your project. A selector bar at the top of the library area provides access to four different library categories.

File Template Library

The File Template library (Figure 3-23) contains templates for several types of files (such as settings bundles and property lists) and subclasses of commonly used classes. To use a file template, simply drag it into the project navigator.

Code Snippet Library

The Code Snippet library (Figure 3-24) contains short pieces of code that you can use in your application. To use one, drag it directly into your source code file.



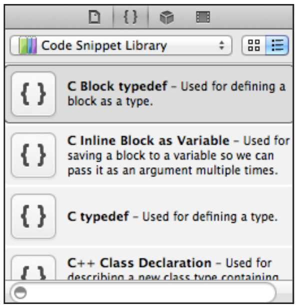
Objective-C class – An Objective-C class with a header for Cocoa Touch

UIViewController subclass – An Objective-C view controller subclass for Cocoa Touch

Objective-C category – An Objective-C category with a header for Cocoa Touch

Objective-C protocol – An Objective-

FIGURE 3-23



C Block typedef – Used for defining a block as a type.

C Inline Block as Variable – Used for saving a block to a variable so we can pass it as an argument multiple times.

C typedef – Used for defining a type.

C++ Class Declaration – Used for

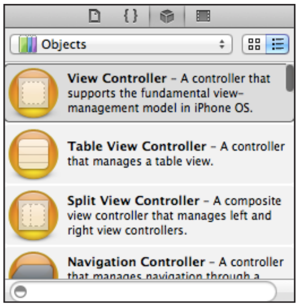
FIGURE 3-24

Object Library

The Object library (Figure 3-25) contains a collection of user interface objects that you can use in your project.

Media Library

The Media library (Figure 3-26) contains all graphics, icons, audio, and other media files that you use in your project.



View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Table View Controller – A controller that manages a table view.

Split View Controller – A composite view controller that manages left and right view controllers.

Navigation Controller – A controller

FIGURE 3-25

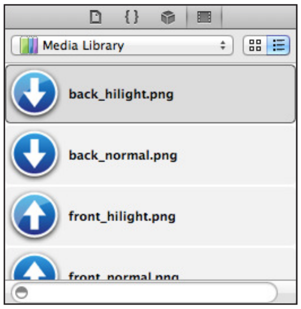


FIGURE 3-26

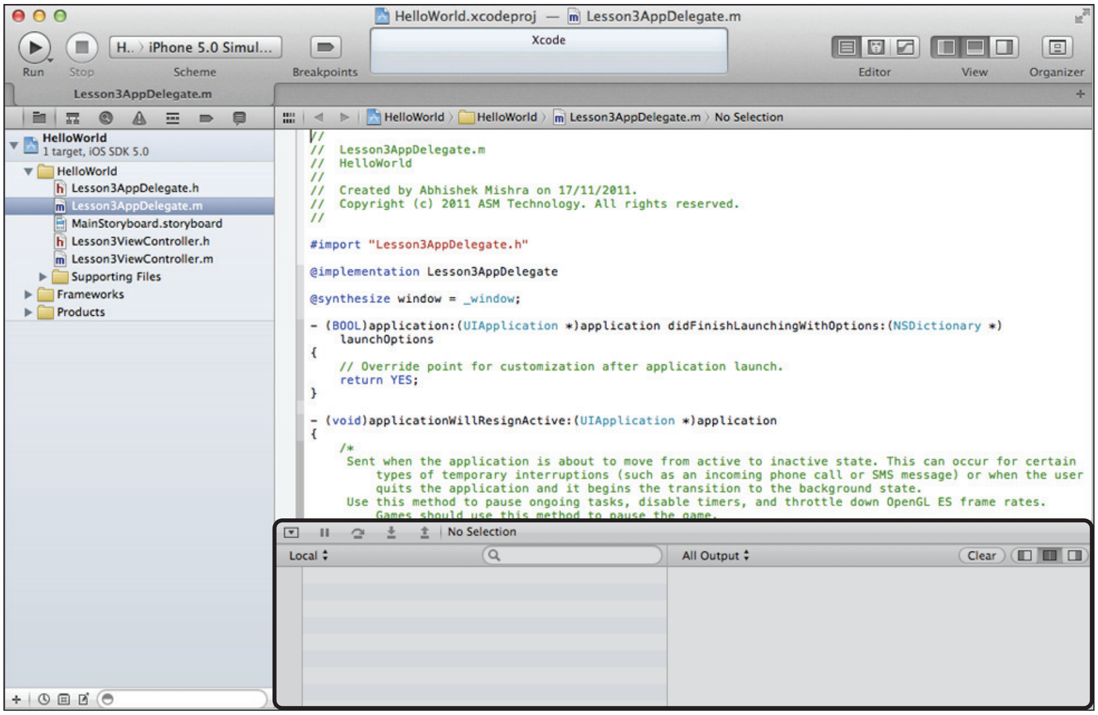
The Debugger Area

The debugger area (Figure 3-27) also supplements the editor area. You can access it by selecting the View ➪ Show Debug Area menu item or by clicking the debugger button in the toolbar.

The debugger area is used while debugging an application and to access the debug console window. You can use this area to examine the values of variables in your programs.

The Toolbar

The Xcode toolbar (Figure 3-28) is located at the top of the workspace window. Use the first two buttons on the left side to run/stop the active build scheme. Immediately following the stop button is the Scheme/Target multi-selector. When you create an iOS project, Xcode creates a scheme with the same name as the project and several build targets.



```
//
//  Lesson3AppDelegate.m
//  HelloWorld
//
//  Created by Abhishek Mishra on 17/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights reserved.
//

#import "Lesson3AppDelegate.h"

@implementation Lesson3AppDelegate

@synthesize window = _window;

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

- (void)applicationWillResignActive:(UIApplication *)application
{
    /*
     Sent when the application is about to move from active to inactive state. This can occur for certain
        types of temporary interruptions (such as an incoming phone call or SMS message) or when the user
        quits the application and it begins the transition to the background state.
     Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates.
        Games should use this method to pause the game.
```

FIGURE 3-27

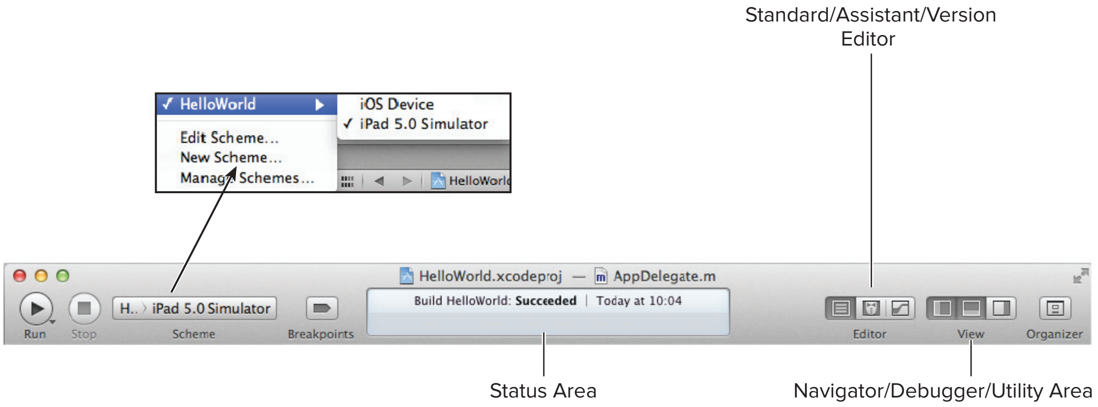


FIGURE 3-28

The build targets that are typically generated for a project include:

- iOS Device
- iPhone 5.0 Simulator (if it is an iPhone or Universal project)
- iPad 5.0 Simulator (if it is an iPad or Universal project)

You can use the Scheme/Target multi-selector to switch build targets and create/edit schemes. Managing schemes is an advanced topic beyond the scope of this book.

To the right of the Scheme/Target multi-selector is a status window. Following the status window, the toolbar contains the editor selector, utility selector, and organizer buttons. The editor and utility selector buttons were covered in the previous sections.

The organizer button enables you to access the Xcode Organizer, which you can use to manage devices, builds, and provisioning profiles.

TRY IT

In this Try It, you launch Xcode and create a new project using the Single View Application template. You then open a file in the editor area and learn to display the assistant editor, debugger, and utilities area.

Lesson Requirements

- Launch Xcode.
- Create a new project using a template.
- Open a file in the editor area.
- Show the assistant editor.
- Show the debug area.
- Show the utilities area.

Hints

- Remember to leave the Use Storyboard and Use Automatic Reference Counting checkboxes selected.

Step-by-Step

1. Create a Single View Application in Xcode called `iOSTest`.
    1. Launch Xcode.
    2. Create a new project by selecting the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** iOSTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson3

- **Define Family:** iPhone
- **Use Storyboard:** Checked
- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not checked and click Create.

2. Open the `Lesson3AppDelegate.h` file in the Xcode editor.
    1. Ensure the project navigator is visible and the `iOSTest` project is open.
    2. Click the `Lesson3AppDelegate.h` file.
3. Use the editor selector buttons on the Xcode toolbar to show the assistant editor.
4. Use the view selector buttons on the Xcode toolbar to show the debug area.
5. Use the view selector buttons on the Xcode toolbar to show the utilities area.

> *Please select Lesson 3 on the DVD that accompanies the print book, or go to* `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

4 iOS Application Basics

In Lesson 1, you created your first iOS application by dragging a few visual elements from the Object library onto your application's `.xib` file. For an app to do something useful, though, it must be able to handle user interaction. This was something missing from the `HelloiOS` app created in Lesson 1. In this lesson you learn some basic concepts involved in iOS application programming, and how to add interactivity to your apps.

iOS programming is based on an event-driven model and is all about writing code to respond to one or more events. The device generates these events every time the user does something with the application. For instance, if the user taps a button, an appropriate "touch" event is generated by the device and forwarded to the application.

Direct user interaction, although the most common reason for events, isn't the only one. For instance, events are generated when the phone is receiving a call, when a file has successfully downloaded from a server, and so on. Practically anything that happens on an iOS device ends up generating one or more events. When an event is generated, the operating system looks into your program to see if a method in one of your classes should be notified.

The key to iOS programming lies in knowing what these events are, and how to set your methods to be called when one of these events occurs. Figure 4-1 shows a simplified version of the sequence of events that occurs between the launch and termination of an iOS application. This sequence is also known as the *application life cycle*.

At key points in the application life cycle, messages are sent to objects in the application to let it know what's going on. iOS applications aren't actually terminated when the user presses the home button on the device; instead they are moved to a background "suspended" state.

APPLICATION STATES

At the heart of your code is a C-based function called `main()`. Xcode creates this function for you as part of the boilerplate code that is generated when you create a new project. Its main purpose is to serve as an entry point for your application and hand control to one of the classes in the `UIKit` framework. You should never have to modify the implementation of this function.

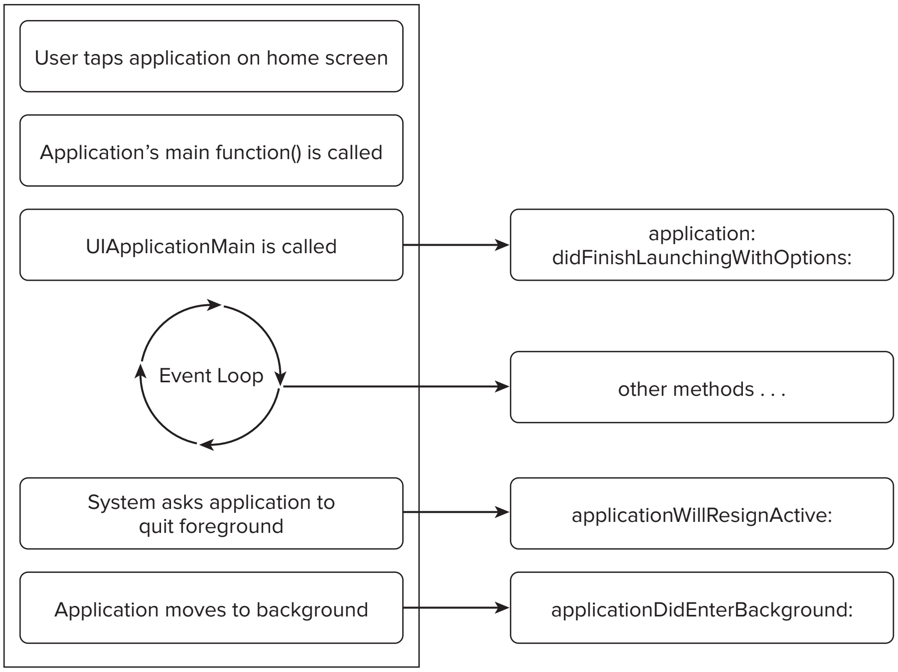

User taps application on home screen
Application's main function() is called
UIApplicationMain is called
application: didFinishLaunchingWithOptions:
Event Loop
other methods . . .
System asks application to quit foreground
applicationWillResignActive:
Application moves to background
applicationDidEnterBackground:

FIGURE 4-1

Every iOS application must have a class that implements the `UIApplicationDelegate` protocol. This class is known as the application delegate object and is responsible for monitoring the high-level behavior of your application. The application delegate object must implement a few key methods of the `UIApplicationDelegate` protocol that are used to handle critical events.

In object-oriented terminology, a delegate is an object that implements a certain set of methods. These methods are then called by another object as and when needed.

iOS applications can be in one of several states at any given point in time; these states are summarized in Table 4-1.

TABLE 4-1: iOS Application States

STATE	DESCRIPTION
Not running	The application is currently not running.
Inactive	The application is running, but not receiving any events. An application can stay briefly in this state while it transitions to another state. The only time the application remains inactive for a considerable period is when the system prompts the user to respond to some event such as an incoming phone call or SMS.
Active	The application is running and receiving events.

STATE	DESCRIPTION
Background	The application is in the background and executing code. Most applications briefly enter this state on their way to being suspended.
Suspended	The application is in the background and not executing any code.

At launch time, an application moves from the not-running state to the active or background state. During the initial startup sequence, the application delegate's `application:didFinishLaunchingWithOptions:` method is called, followed by either the `applicationDidBecomeActive:` or `applicationDidEnterBackground:` methods.

These methods are, as you might expect, part of the `UIApplicationDelegate` protocol, and your application delegate must provide implementations for these.

> *To learn more about the lifecycle of an iOS application, read the* "iOS App Programming Guide" *available at* `http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/Introduction/Introduction.html`.

Typically, when you create a new project with Xcode, basic implementations are provided for you as part of the boilerplate code in the application delegate class.

WINDOWS, VIEWS, AND VIEW CONTROLLERS

Windows and views are used to represent the application's visual content on the screen and manage the immediate interactions with that content. A window is an instance of a `UIWindow` class and is essentially an empty surface that hosts one or more views. Windows fit the entire main screen area and have no visual elements. Most iOS applications have only one window; however, some may have an additional window to support an external display.

A view is an instance of a `UIView` class and defines a rectangular region within the main application window. Views draw content in their rectangular area, have some properties that can animate to new values, and can receive touch events. Views can also have a number of subviews, thus creating a view hierarchy.

Several user interface classes that are part of the `UIKit` framework (for instance, the `UIButton` class) are subclasses of `UIView`. `UIKit` is discussed later in this lesson, but for now you need to understand that complex user interfaces can be created by putting together a hierarchy of overlapping user interface elements, as shown in Figure 4-2.

Much of iOS programming is about following conventions and creating well-structured applications. One of the most common design patterns, known as the Model-View-Controller pattern, requires the programmer to think of the individual classes in the application as belonging to one of three distinct categories: data, view, and manager.

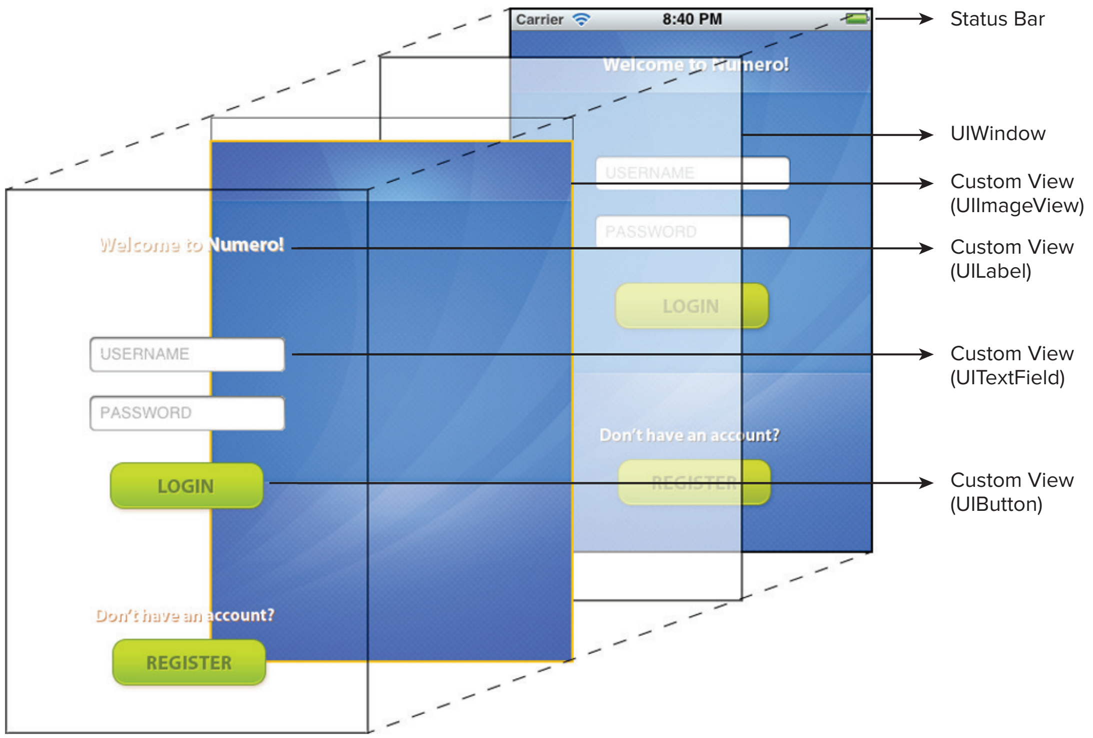


FIGURE 4-2

Data classes are responsible for storing and representing application data in a meaningful way. This data could be stock quotes downloaded from a web service, or a page of text typed in by the user. Data classes are also known as model classes.

View classes are responsible for presenting this data to the user and controller classes manage the link between data classes and view classes. More often than not, classes tend to have functionality that somewhat blurs this classification a little.

The aim of the MVC pattern is to make objects in these classes as distinct from each other as possible. A class that implements a button should not have any code to process user input when that button is pressed; similarly, a class that represents stock quotes should not have any code to draw a graph.

iOS applications in general follow this pattern, and to that end have data classes, view classes, and controller classes. An iOS application's version of a controller class is called a *view controller* class, and essentially manages the contents and user interaction with a view. Thus, while a view is responsible for presenting a user interface such as a button, the code that does something when this button is pressed sits in the view controller class.

Several controller classes are provided by the iOS SDK; however the most commonly used one is the `UIViewController` class. View controllers in iOS applications are thus usually instances of `UIViewController`.

The `UIViewController` base class defines several methods that are called when significant events occur. You can override these methods in your view controller class to do something when these significant events occur. Some of the most common methods that you can override are:

- `- didReceiveMemoryWarning`: Called when memory is low. The view controller should typically try to free up unused resources.
- `- viewDidLoad`: Called after the view has finished loading into memory. This method is a good place to set up the user interface elements in the view to initial states.

The view controller class and the user interface elements in the view class are often linked to each other using Xcode's Interface Builder. To link the two together, Interface Builder uses special variables (known as *outlets*), and methods (known as *actions*) in the view controller class. Using Interface Builder to create outlets and actions is discussed later in this lesson. It is important to keep in mind that you could create these links programmatically without using Interface Builder at all. Creating user interface elements programmatically is covered in Lesson 17.

> *To learn more about view controllers, read the* "View Controller Programming Guide" *available at* `http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007457-CH1-SW1`.

FRAMEWORKS

A *framework* is a collection of classes that you can use to write your apps. Apple provides a large number of frameworks that enforce consistent implementation of features across applications from different developers. All the familiar user interface features like navigation bars, toolbars, back buttons, and so on that we commonly use in iOS apps are, in fact, classes in one of the frameworks provided by Apple.

Although the idea of sticking to user interface elements that only appear in an Apple framework may seem limiting, it is in fact not the case. Apple's frameworks have a large number of classes; in fact, some frameworks do not have any user interface-specific classes at all. You must always try to use classes from one of the standard frameworks when possible; this will ensure that you do not spend time reinventing the wheel.

The frameworks are grouped together into layers, with frameworks in higher layers building upon frameworks found in lower layers. Figure 4-3 shows the different layers with examples of some of the frameworks they contain. In general, using a class from a framework in a lower layer requires you to write more code that using one from a higher layer.

The top-most layer is known as Cocoa Touch and contains a large number of classes distributed within multiple frameworks that handle the most common aspects of iOS applications, including but not limited to processing events, touches, gestures, multithreading, map support, and accelerometer.

Every Xcode project that is created from one of the standard iOS application templates includes three key frameworks: `CoreGraphics`, `Foundation`, and `UIKit`. Most simple apps do not need to use classes from any other framework.

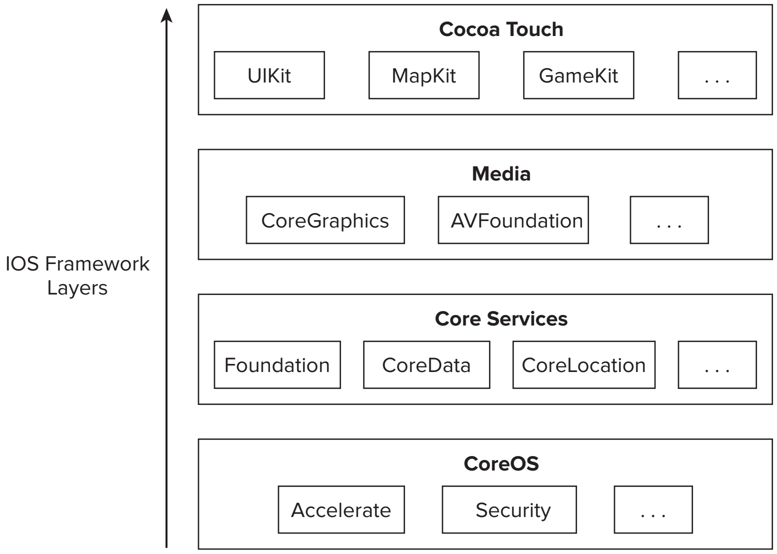


FIGURE 4-3

Of all Cocoa Touch frameworks, perhaps the most important and commonly used is `UIKit`. The name `UIKit` may lead you to conclude that it contains only user interface-specific classes. This is, however, not true. Besides user interface-specific classes, `UIKit` contains classes for handling events, touches, gestures, and general application support.

Classes that are part of `UIKit` always begin with the `UI` prefix. Thus, the `UIApplication`, `UIWindow`, `UIView`, and `UIViewController` classes that you have encountered earlier in this lesson are all part of `UIKit`.

The UIButton Class

The `UIButton` class is part of the `UIKit` framework, and encapsulates the functionality of a button on a touch screen. A `UIButton` object sends a message to a target object when it intercepts one or more touch events.

`UIButton` objects can intercept different types of touch events; some of the most common ones are briefly summarized in Table 4-2.

TABLE 4-2: UIButton Touch Events

EVENT	DESCRIPTION
Touch Up Inside	The user has lifted his finger from the touch screen inside the area of the button.

EVENT	DESCRIPTION
Touch Up Outside	The user had pressed this button but has lifted his finger outside the area of the button (that is, dragged his finger outside the button before lifting it).
Touch Down	The user has just pressed this button and hasn't yet lifted his finger, or moved it.
Touch Drag Enter	The user has pressed this button, then dragged his finger outside the button, and has just entered the area of the button again (without lifting the finger).
Touch Drag Exit	The user has pressed this button, then dragged his finger and, as a consequence of dragging, has just left the area of the button.
Touch Drag Inside	The user has pressed this button and is dragging his finger within the area of the button.
Touch Drag Outside	The user has pressed this button and is now dragging his finger outside the area of the button. The user would have had to move his finger out of the button and continued to drag without lifting his finger to receive this event.

By and large, the most common event that you will use in your code is the Touch Up Inside event.

To learn more about the UIButton class, read the "UIButton Class Reference" *documentation available at* `http://developer.apple.com/library/IOs/#documentation/UIKit/Reference/UIButton_Class/UIButton/UIButton.html`.

The UILabel Class

The `UILabel` class allows you to draw one or multiple lines of static text onto your view. The `UILabel` class does not normally generate touch events, but provides several properties that allow you to customize its appearance. The most common ones are described in Table 4-3.

TABLE 4-3: UILabel Properties

PROPERTY	DESCRIPTION
text	Sets the text displayed by the label using the current font.
numberOfLines	The maximum number of lines to be drawn.
textAlignment	Defines the horizontal alignment of text in the label. Permissible values are UITextAlignmentLeft, UITextAlignmentRight, and UITextAlignmentCenter.

continues

TABLE 4-3 *(continued)*

PROPERTY	DESCRIPTION
textColor	Sets the color used to display the text. You can set the color by providing a UIColor object. The UIColor class is discussed later in this lesson.
font	Sets the font that is used to display the text. The font is specified as a UIFont object. UIFont objects are covered in Lesson 6.

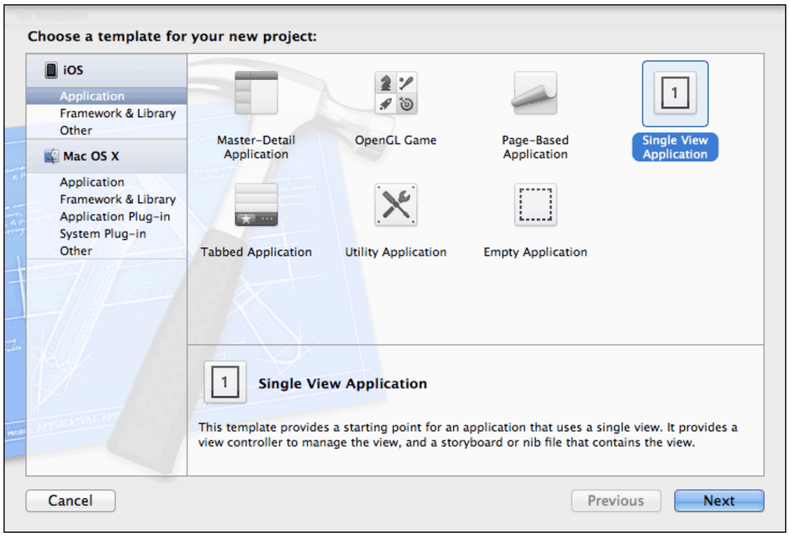
To learn more about The UILabel class, read the "UILabel Class" *reference documentation available at* `http://developer.apple.com/library/IOs/#documentation/UIKit/Reference/UILabel_Class/Reference/UILabel.html`.

CREATING USER INTERFACE ELEMENTS

`UIButton` and `UILabel` instances can be created programmatically or by using the Xcode Interface Builder. Creating `UIKit` objects programmatically is covered in Lesson 17. In this lesson, you use Interface Builder.

Before you can add a button or label, you must first create a new Xcode project. To do so, launch Xcode and select the File ➪ New ➪ New Project menu item. This brings up a dialog box (Figure 4-4) where you can select a template to use as a starting point for your development.

There are several types of iOS application templates; the most commonly used ones are described in Table 4-4.



Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

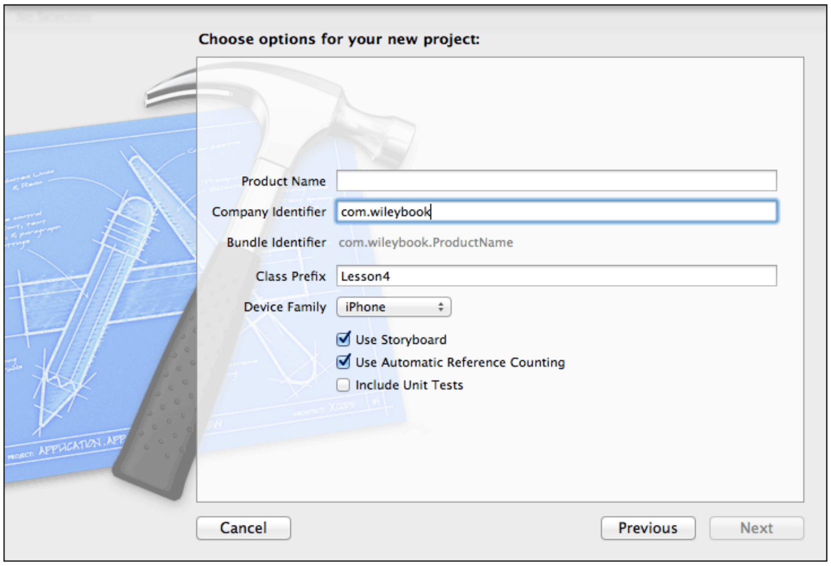
FIGURE 4-4

TABLE 4-4: Common iOS Application Templates

TEMPLATE TYPE	DESCRIPTION
Single View Application	Provides a starting point to create an application that has one view. The template provides a view controller to manage the view. The view is contained within a nib file or storyboard.
Page--Based Application	Used to create a book/magazine reader type of application in which the user swipes the screen to turn pages. An application based on this template applies a built-in page turn animation when it detects an appropriate swipe. Page-based applications are covered in Lesson 16.
OpenGL Game	Provides a starting point for an OpenGL/ES-based game. This type of project is not covered in this book.
Master Detail Application	Provides a starting point for an application that on the iPhone presents hierarchical content using tables, much like the iPhone Contacts application. On the iPad, this application will use a split view controller. This type of project is not covered in this book.
Tabbed Application	Provides a starting point for an application that uses a tab bar to present multiple views of content side by side. This is covered in Lesson 15.

Most examples in this book are based on the Single View Application template. Select this template and click Next to go to the project options dialog box shown on Figure 4-5.

The project options dialog box is where you can specify additional information for your new project. The fields in this dialog box are described in Table 4-5.


Choose options for your new project:

Product Name
Company Identifier com.wileybook
Bundle Identifier com.wileybook.ProductName
Class Prefix Lesson4
Device Family iPhone
☑ Use Storyboard
☑ Use Automatic Reference Counting
☐ Include Unit Tests

FIGURE 4-5

TABLE 4-5: Project Options

ITEM	DESCRIPTION
Product Name	This is where you type your new app's name. Avoid using spaces or starting the name with a number.
Company Identifier	Usually the reverse domain name of your website. This is a sequence of characters that is used to create the Bundle Identifier along with the product name. The lessons in this book use `com.wileybook`.
Bundle Identifier	Every app must have a unique identifier that can be used by iOS to differentiate it from other apps that may be running on your user's phone. This identifier is generated automatically for you, but you can change it later on. Combining the company identifier with the product name creates this identifier.
Class Prefix	A prefix applied to the name of each class created in the project. For the lessons in this book, the class prefix will be the word Lesson followed by the lesson number.
Device Family	Allows you to specify whether your new app is being built for the iPhone, iPad, or both device families (in which case it is called a Universal application).
Use Storyboard	Check this if you want to use a storyboard for your view instead of a nib file. All lessons in this book use storyboards. Storyboarding is covered in Lesson 5.
Use Automatic Reference Counting	Automatic Reference Counting (ARC) is a new compiler-level feature introduced with iOS5 that simplifies memory management. All lessons in this book use automatic reference counting.
Include Unit Tests	Check this if you intend to use unit testing with your application. Unit testing is not covered in this book, thus all projects you create will have this checkbox unchecked.

Name your product `ButtonTest` and specify com.wileybook as the company identifier. Specify Lesson4 as the class prefix and select iPhone from the Device Family drop-down menu. Uncheck the Include Unit Tests checkbox.

Click Next to specify a location where Xcode should create your new project files.

You can now look at some of the files generated for you by Xcode as part of the template (Figure 4-6). Take a look at the project navigator and note that Xcode has created an application delegate class and a view controller class called `Lesson4AppDelegate` and `Lesson4ViewController`, respectively.

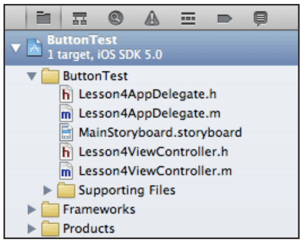


FIGURE 4-6

Your application's user interface is contained in the storyboard file called `MainStoryboard.storyboard`. Select the storyboard file to edit it with Interface Builder.

Prior to iOS5, user interfaces were contained in nib files. Storyboards are amongst the new additions to iOS 5 and are covered in Lesson 5. For now, all you need to know is that a storyboard consists of one or more scenes. Each scene contains the user interface for a single view. Most Interface Builder concepts that were applicable to nib files still apply to scenes within a storyboard.

If you are curious, you can browse through the boilerplate code generated by Xcode in the application delegate and view controller classes.

The application delegate class, Lesson4AppDelegate, contains default implementations (mostly empty) for several methods that are called at different points in the application's life cycle. Similarly, the application's view controller class, Lesson4ViewController, contains implementations for several view life cycle methods, most of which just call their base class versions.

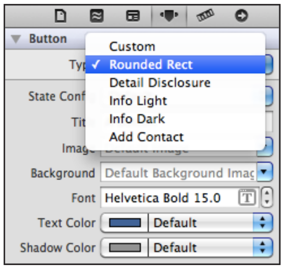
Because this application is based on the Single View Application template, the storyboard contains a single scene that represents the user interface of this view. Adding a `UIButton` to the view is a simple matter of dragging a Round Rect Button object from the Object library onto the client area of the scene. You can use the mouse to resize and position it, or specify precise values using the Size inspector (View ➪ Utilities ➪ Show Size Inspector).

You can use the Attributes inspector to set up some common properties of the new button. However, keep in mind that each of these properties can be set up using Objective-C code. If you just want to add a title to a button quickly, simply double-click the button and type in a suitable title.

The default rounded rectangle button created by Xcode is, in fact, quite boring. To make it more interesting you can change its appearance using use the Attribute inspector (View ➪ Utilities ➪ Show Attributes Inspector). You can select from common buttons types using the Type drop-down (Figure 4-7).

The standard button types are:

- **Custom:** A button without any specific appearance, invisible unless you set up an image. Typically used to create hotspots or graphical buttons.
- **Rounded Rect:** This is the default.
- **Detail Disclosure:** A button with an arrow; usually indicates that tapping it will reveal additional information.
- **Info Light:** The standard "i" icon, intended to be used over dark backgrounds.



Background Default Background Imag
Font Helvetica Bold 15.0
Text Color Default
Shadow Color Default

FIGURE 4-7

- **Info Dark:** The standard "i" icon, intended to be used over light backgrounds.
- **Add Contact:** The standard + icon.

A `UIButton` object can be in one of four states:

- **Default:** The button is visible on the screen; the user is not interacting with it.
- **Highlighted:** The user is currently pressing down the button.
- **Selected:** A UIButton object does not ordinarily move into this state as a result of user interaction, however this state can be setup programmatically.
- **Disabled:** The button is visible on the screen, but the user cannot interact with it.

For each state you can provide a different background color, title, and background image. You can use the Attribute inspector's State Config drop-down to select a state and set up attributes for that state. This is shown in Figure 4-8.

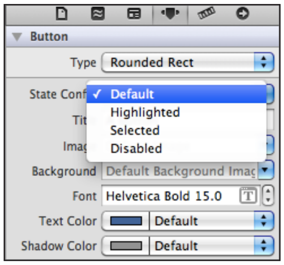


FIGURE 4-8

To assign an image for your button, you will need to create a PNG image for each state and import the images into your Xcode project. When applying an image to a button, you can assign the image to either the `Image` attribute or the `Background` attribute. There is a slight difference between the two. Assigning an image to the `Image` attribute does not hide the title of the button.

Creating Outlets

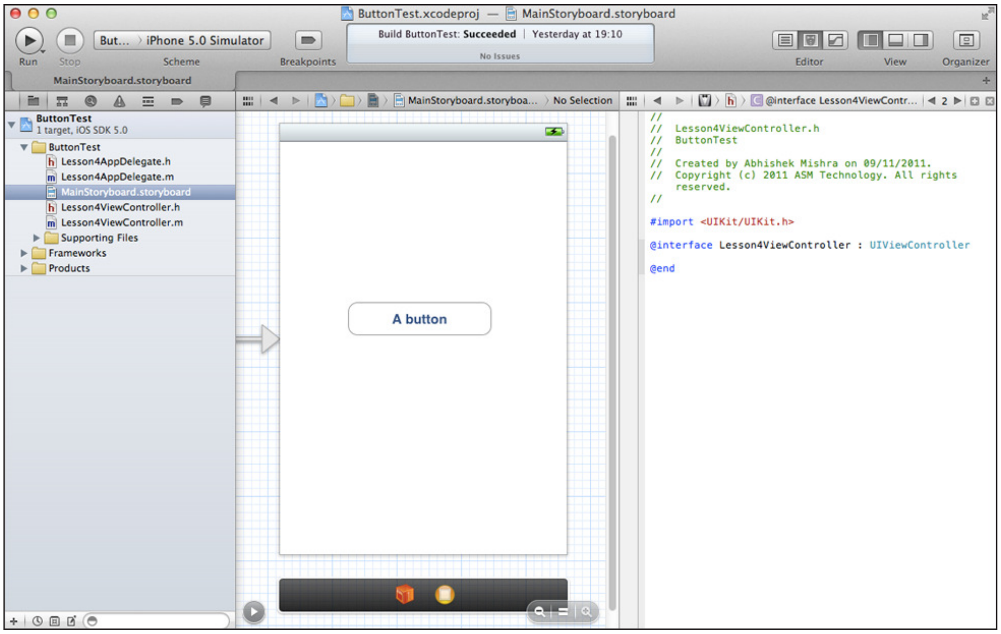
User interface elements are usually defined in storyboards, and even though you can set their properties graphically using Interface Builder, there will be times when you will need to read or change a property from your code while your application is running. To do so, you need to create an appropriate instance variable in the view controller class and connect it to the user interface element in the scene. These connections are known as outlets, and can be created quickly using the assistant editor. To display the assistant editor, use the View ➪ Assistant Editor ➪ Show Assistant Editor menu item. With the assistant editor visible, selecting a user interface element in one of the scenes of the storyboard file automatically opens the interface (`.h`) file of the corresponding view controller class. This is shown in Figure 4-9.

To create an outlet for the button object, right-click the button to bring up a context menu and drag from the circle beside the `New Referencing Outlet` line to an empty line just before the `@end` statement in the header file. This is shown in Figure 4-10.

Releasing the mouse button on an empty line in the header file pops up a dialog box that allows you to type in a name for the instance variable (Figure 4-11).

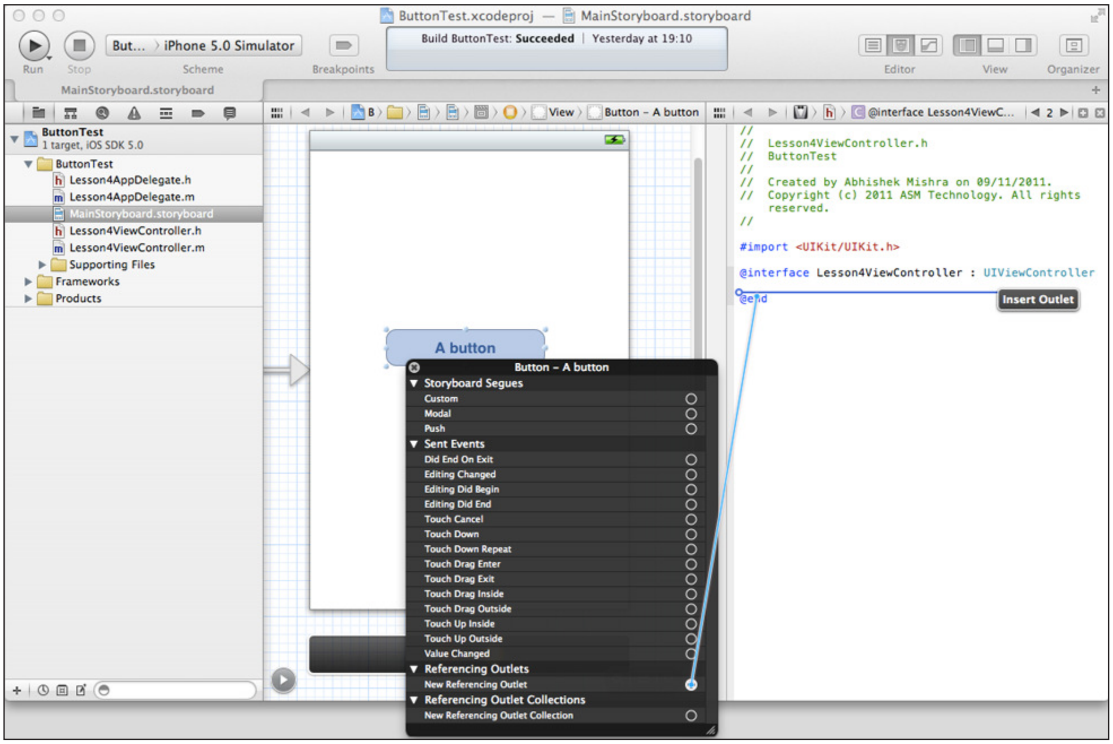
Click the Connect button in the popup dialog box to finish creating the outlet. Notice how Xcode has created a suitable `@property` declaration of type `UIButton` in your class.

```
#import <UIKit/UIKit.h>
@interface Lesson4ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIButton *someButton;
@end
```


ButtonTest.xcodeproj — MainStoryboard.storyboard
Run
Stop
But… iPhone 5.0 Simulator
Scheme
Breakpoints
Build ButtonTest: Succeeded | Yesterday at 19:10
No Issues
Editor
View
Organizer
MainStoryboard.storyboard
ButtonTest
1 target, iOS SDK 5.0
ButtonTest
Lesson4AppDelegate.h
Lesson4AppDelegate.m
MainStoryboard.storyboard
Lesson4ViewController.h
Lesson4ViewController.m
Supporting Files
Frameworks
Products
MainStoryboard.storyboa… No Selection
@interface Lesson4ViewContr…
A button

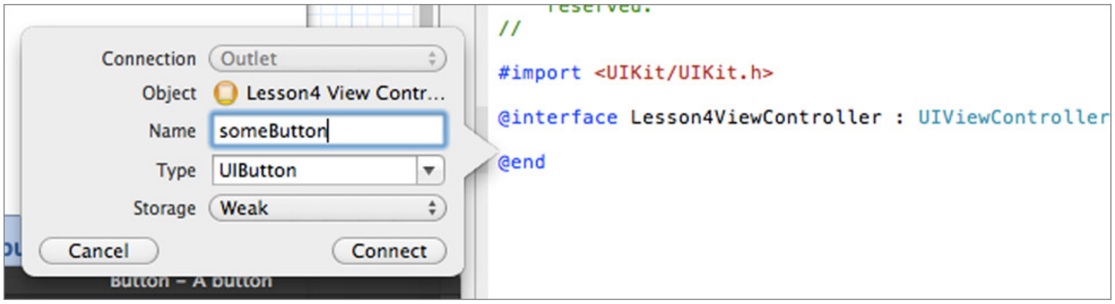
```
//
//  Lesson4ViewController.h
//  ButtonTest
//
//  Created by Abhishek Mishra on 09/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson4ViewController : UIViewController

@end
```

FIGURE 4-9



```
//
//  Lesson4ViewController.h
//  ButtonTest
//
//  Created by Abhishek Mishra on 09/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson4ViewController : UIViewController

@end
```

FIGURE 4-10



```
//

#import <UIKit/UIKit.h>

@interface Lesson4ViewController : UIViewController

@end
```

FIGURE 4-11

To signify that the variable refers to an object defined in the storyboard file, Xcode adds the `IBOutlet` keyword to the property statement:

```
    @property (weak, nonatomic) IBOutlet UIButton *someButton;
```

Creating Actions

Most user interface elements generate a variety of events as a result of user interaction. As a programmer, you will be interested in some of these events and will want your code to be executed when these events occur. To achieve this, you need to create one or more methods in the view controller class, and wire them up to appropriate events of the user interface element. These methods in the view controller class that are called as a result of an event being triggered are called actions.

As you might expect, both these steps can be performed graphically with the Interface Builder. To show a list of events that can be intercepted by a user interface object, simply right-click the user interface element in Interface Builder and browse through the entries under the Sent Events category of the context menu (Figure 4-12).

You will see all the familiar touch events listed there along with a few others. To wire up the Touch Up Inside event to a method in your class, simply drag from the circle beside the name of the event to an empty line in your view controller's interface file before the `@end` statement (Figure 4-13).

When you release the mouse on the view controller, Xcode presents a popup window in which you can provide a name for the new method. Call the new method `onButtonPressed`.

Adding a method to a class manually is a two-step process. First, a line defining the method must be added to the interface (`.h`) file, and then an implementation must be added to the corresponding implementation (`.m`) file. When you use the Interface Builder to add a method to a view controller class, both of these steps are carried out for you.

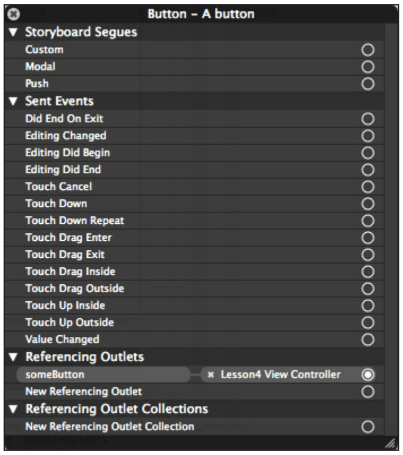
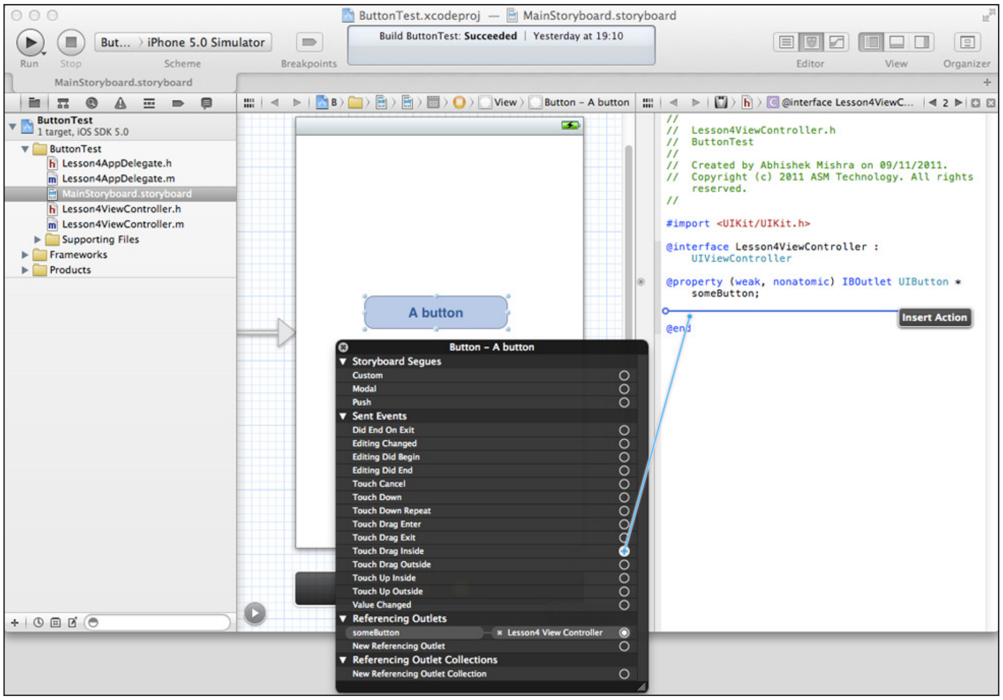


FIGURE 4-12



```
//
//  Lesson4ViewController.h
//  ButtonTest
//
//  Created by Abhishek Mishra on 09/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson4ViewController :
    UIViewController

@property (weak, nonatomic) IBOutlet UIButton *
    someButton;

@end
```

FIGURE 4-13

To verify that this is indeed the case, you can examine the `Lesson4ViewController.h` file, which now has a declaration for a method named `onButtonPressed`:

```
#import <UIKit/UIKit.h>
@interface Lesson4ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIButton *someButton;
- (IBAction)onButtonPressed:(id)sender;
@end
```

The corresponding implementation file `Lesson4ViewController.m` has a stub implementation for the method at the end of the file:

```
#import "ButtonViewController.h"
@implementation ButtonViewController
@synthesize my_button;
…
…
…
- (IBAction)onButtonPressed:(id)sender {
}
@end
```

Note that the `onButtonPressed` method takes in a single argument of type `(id)` called `sender`. This parameter always contains a reference to the object that sent this message to your view controller. In this particular case, the sender would be the user interface object that generated the corresponding event.

An alternative method to add an outlet or action to a view controller class and connect it to a user interface element in a storyboard scene is to Ctrl+drag from the element onto an empty line in the view controller's interface file before the `@end` statement. Releasing the mouse button will present a popup dialog that lets you create either an outlet or an action.

Adding Interactivity

At this point, if you were to test this application in the iOS Simulator, you would see that nothing really happens when you tap the button. This is because you haven't put any code in your action method to do something. All you've done so far is created a `UIButton`, created an action, and linked the two together.

Say you wanted to change the background color of the application's view when the button is pressed. You can do this by adding a single line of code to the `onButtonPressed` method:

```
- (IBAction)onButtonPressed:(id)sender {
    self.view.backgroundColor = [UIColor redColor];
}
```

If you run this application now, and tap the button, the background color of the view changes to red. This is shown in Figure 4-14.

The code that makes this happen demands a little explanation. The `self` variable, when used within a method, always returns a reference to the object that owns the method. In this case, that would be an instance of the `Lesson4ViewController` class.

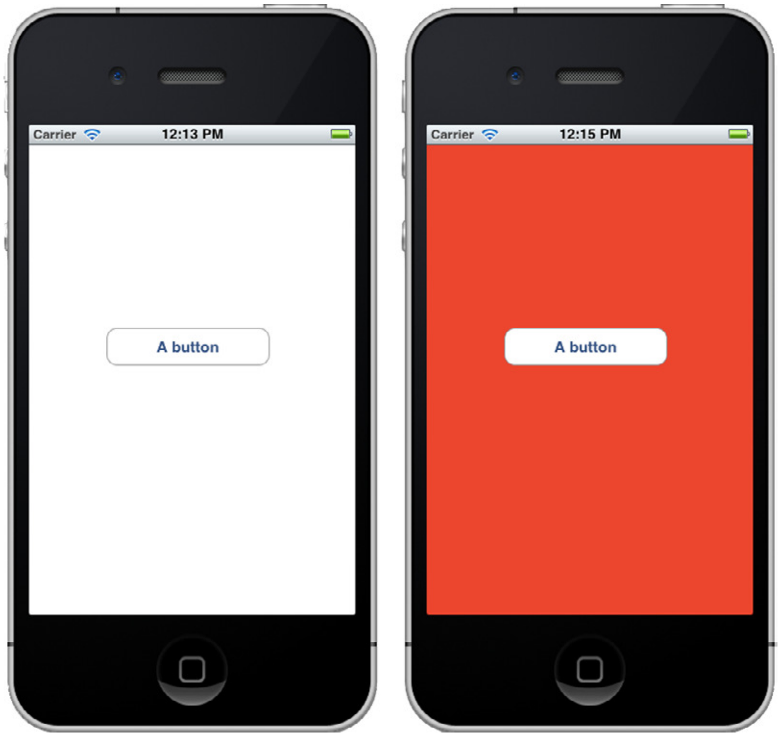


FIGURE 4-14

`Lesson4ViewController` is a view controller and inherits from `UIViewController`. This can be verified by simply examining the declaration of the `ButtonViewController` in the interface (`.h`) file:

```
@interface Lesson4ViewController : UIViewController
```

By virtue of inheritance, the `Lesson4ViewController` inherits the member variables defined in `UIViewController`, one of which is a reference to a `UIView` object called `view`. This happens to be the very same view that is being managed by the view controller—that is, your application's view.

Thus, you can get a reference to the current view by using the statement:

```
self.view
```

Once you have this, you can set any of the properties of the `UIView` object (including the `backgroundColor` property) to an appropriate value.

The `backgroundColor` property is actually an instance of a `UIColor` object. You can create `UIColor` objects either from RGB values, or from a set of predefined colors. Table 4-6 lists some of the messages that you can send the `UIColor` class to instantiate objects.

TABLE 4-6: Instantiating a UIColor Object

UICOLOR METHODS	DESCRIPTION
`+ (UIColor *)blackColor;`	Returns a UIColor object initialized to R=0.0, G=0.0, B= 0.0
`+ (UIColor *)darkGrayColor;`	Returns a UIColor object initialized to R=0.33, G=0.33, B= 0.33
`+ (UIColor *)lightGrayColor`	Returns a UIColor object initialized to R=0.66, G=0.66, B= 0.66
`+ (UIColor *)whiteColor;`	Returns a UIColor object initialized to R=1.0, G=1.0, B= 1.0
`+ (UIColor *)grayColor;`	Returns a UIColor object initialized to R=0.5, G=0.5, B= 0.5
`+ (UIColor *)redColor;`	Returns a UIColor object initialized to R=1.0, G=1.0, B= 0.0
`+ (UIColor *)greenColor;`	Returns a UIColor object initialized to R=0.0, G=1.0, B= 0.0
`+ (UIColor *)blueColor;`	Returns a UIColor object initialized to R=0.0, G=0.0, B= 1.0
`+ (UIColor *)cyanColor;`	Returns a UIColor object initialized to R=0.0, G=1.0, B= 1.0
`+ (UIColor *)yellowColor;`	Returns a UIColor object initialized to R=1.0, G=1.0, B= 0.0
`+ (UIColor *)magentaColor;`	Returns a UIColor object initialized to R=1.0, G=0.0, B= 1.0
`+ (UIColor *)clearColor;`	Returns a transparent UIColor (with alpha = 0.0)
`+ (UIColor *)colorWithRed:green:blue:alpha;`	Returns a UIColor created out of the specified RGBA values; the individual values must lie between 0 and 1

The code in the `onButtonPressed` method could just as easily change the value of any other member variable in the view controller class, or for that matter do something significantly more complicated, such as downloading data from the internet.

For example, you could create a `UILabel`, set up an appropriate outlet called `textLabel` using the Interface Builder, and change the text displayed in the label when the button is pressed using code similar to this:

```
- (IBAction)onButtonPressed:(id)sender {
    textLabel.text = @"This is some text";
}
```

Here you are changing the value of the `text` property of a `UILabel` object. This is demonstrated in the following Try It section.

TRY IT

In this Try It, you create a new Xcode project based on the Singe View Application template, called `InteractionSample`. You use the Interface Builder to create an instance of a `UIButton` and a `UILabel` class and then write code to update the text displayed in the label when the button is pressed.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template.
- Edit the storyboard with Interface Builder.
- Add a `UILabel` and a `UIButton` object to the default scene in the storyboard.
- Create and connect the `UILabel` to an outlet in the view controller class.
- Create and connect the Touch Up Inside event of the `UIButton` instance to an action method in the view controller class.
- Change the text of the label when the button is clicked.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 4 folder in the download.*

Hints

- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.
- To show the assistant editor, use the View ➪ Assistant Editor Show Assistant Editor menu item.
- To show the source editor, use the View ➪ Source Editor ➪ Show Standard Editor menu item.

Step-by-Step

1. Create a Single View Application in Xcode called `InteractionSample`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** InteractionSample
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson4
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Open the storyboard file in Interface Builder.
    1. Ensure the project navigator is visible and the `InteractionSample` project is selected and expanded. To show the project navigator, use the View ➪ Navigators ➪ Show Project Navigator menu item. To expand a project, click the triangle next to the project name in the project navigator.
    2. Click the `MainStoryboard.storyboard` file.
    3. Ensure the utilities editor is visible. To show the utilities editor, use the View ➪ Utilities ➪ Show Utilities menu item.
3. Add a label to the default scene in the storyboard
    1. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
    2. From the Object library, select Label and drop it onto the scene.

    3. Use the Size inspector to size and position the label to X=20, Y=206, W=259, H=40. You can show the Size inspector by using the View ➪ Utilities ➪ Show Size Inspector menu item.
    4. Use the Attributes inspector set the alignment property to be centered. You can show the Attributes inspector by using the View ➪ Utilities ➪ Show Attributes inspector menu item.
4. Add a button to the default scene in the storyboard
    1. From the Object library, select Round Rect Button and drop it onto the scene.
    2. Use the Size inspector to size and position the button to X=113, y=280, W=95, H=37.
    3. Double-click the button and change the text displayed in it to `Greet Me!`
5. Create an outlet in the view controller class and connect it to the label.
    1. Ensure the assistant editor is visible. To show it, use the View ➪ Assistant Editor ➪ Show Assistant Editor menu item. Ensure the `Lesson4ViewController.h` file is open in the assistant editor, if not then use the jump bars to select it.
    2. Right-click the label to show the context menu.
    3. Drag from the circle beside `New Referencing Outlet` to an empty line just before the `@end` keyword in the assistant editor.
    4. Name the new connection `textLabel` in the popup dialog that appears and click the Connect button. The code in the assistant editor should now resemble the following:

```
#import <UIKit/UIKit.h>
@interface Lesson4ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *textLabel;
@end
```

6. Create an action method in the view controller class and connect it to the Touch Up Inside event of the button.
    1. Right-click the button to show the context menu.
    2. Drag from the circle beside the Touch Up Inside event to an empty line just before the `@end` keyword in the assistant editor.
    3. Name the new method `onButtonPressed` in the popup dialog that appears and click the Connect button. The code in the assistant editor should now resemble the following:

```
#import <UIKit/UIKit.h>
@interface Lesson4ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *textLabel;
- (IBAction)onButtonPressed:(id)sender;
@end
```

7. Write code to update the text of the label when the button is pressed.

    1. Ensure the source editor is visible. To show it, use the View ➪ Standard Editor ➪ Show Standard Editor menu item.

    2. Select the `Lesson4ViewController.m` file in the project navigator to open it in the source editor.

    3. Scroll down and locate the implementation of the `onButtonPressed:` method.

    4. Replace it with the following code to change the text of the label:

    ```
    - (IBAction)onButtonPressed:(id)sender
    {
        textLabel.text = @"Greetings mighty coder!";
    }
    ```

8. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

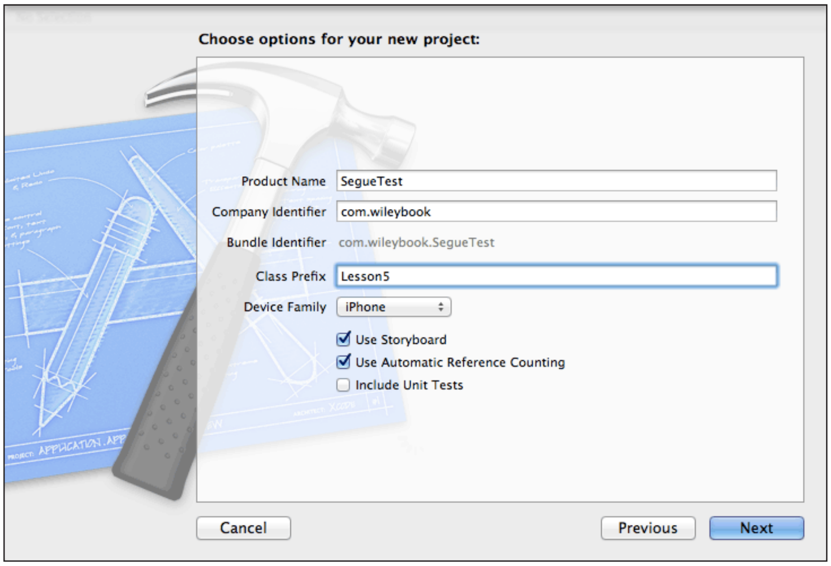
Please select Lesson 4 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

5 Introduction to Storyboards

Most iOS applications are usually made up of several screens of content, with the user typically navigating from one screen to another. A storyboard is a new feature in Xcode that lets you view all the screens as well as the connections between them in a single place.

Storyboards involve two key concepts, scenes and segues. A *scene* is defined by a view controller and is the major visual component of a storyboard. It represents one screen of content in your application.

If you have been programming iOS applications prior to iOS5, everything you know about Interface Builder applies to scenes. To use storyboarding in your application, you must select the Use Storyboard option in the project options dialog (Figure 5-1).


Choose options for your new project:

Product Name: SegueTest
Company Identifier: com.wileybook
Bundle Identifier: com.wileybook.SegueTest
Class Prefix: Lesson5
Device Family: iPhone

[✓] Use Storyboard
[✓] Use Automatic Reference Counting
[ ] Include Unit Tests

FIGURE 5-1

Figure 5-2 shows the scenes that make up the storyboard of a simple iOS application. As you can see, each scene contains familiar UIKit controls like image views, buttons, and labels. Clicking one of the scenes in the storyboard selects it. The selected scene has a green outline around it.

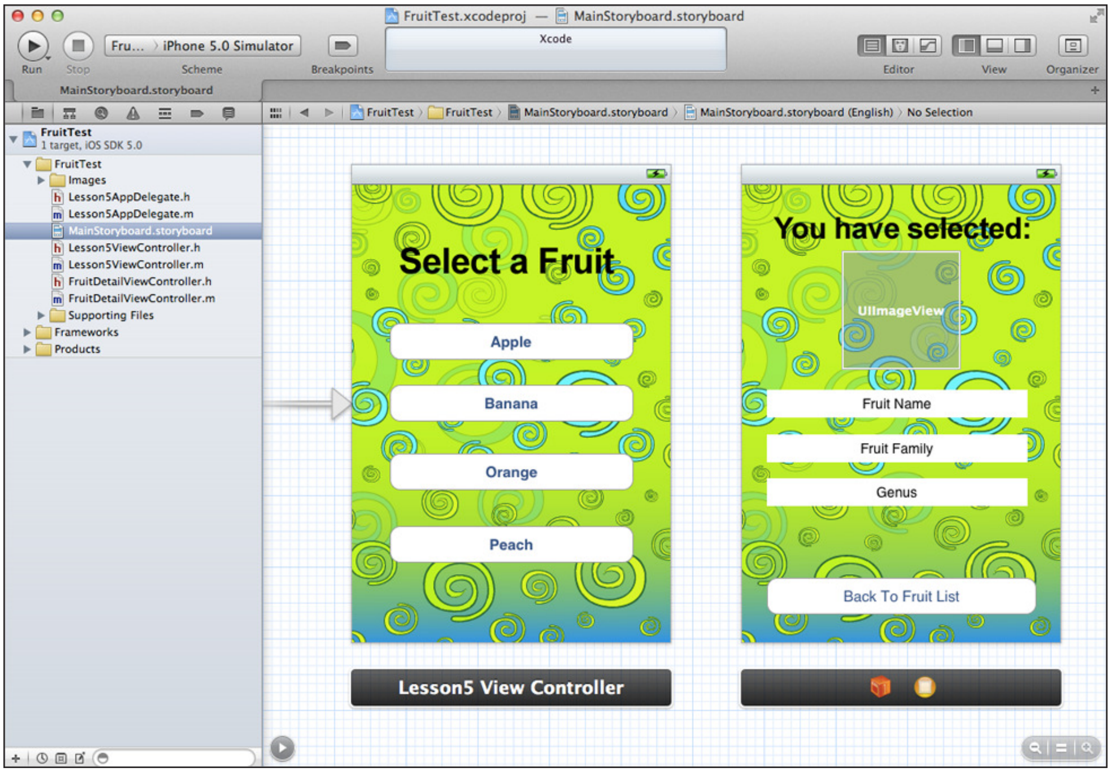


FIGURE 5-2

At the bottom of each scene is a gray rectangle, called the *dock* (Figure 5-3). When a scene is selected, the dock contains icons corresponding to the top-level objects in the scene. The two icons you see in Figure 5-3 correspond to the file's owner and the view controller. The actual user-interface elements in the view controller are not top-level objects because they are contained by the view controller, and hence do not appear in the dock.

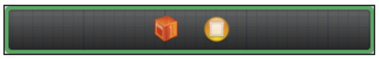


FIGURE 5-3

You can expand the dock by clicking the small triangle indicator at the bottom left of the storyboard (Figure 5-4). When expanded, the dock contains all the objects contained in each scene of the storyboard. Each scene is represented by a rounded rectangle.

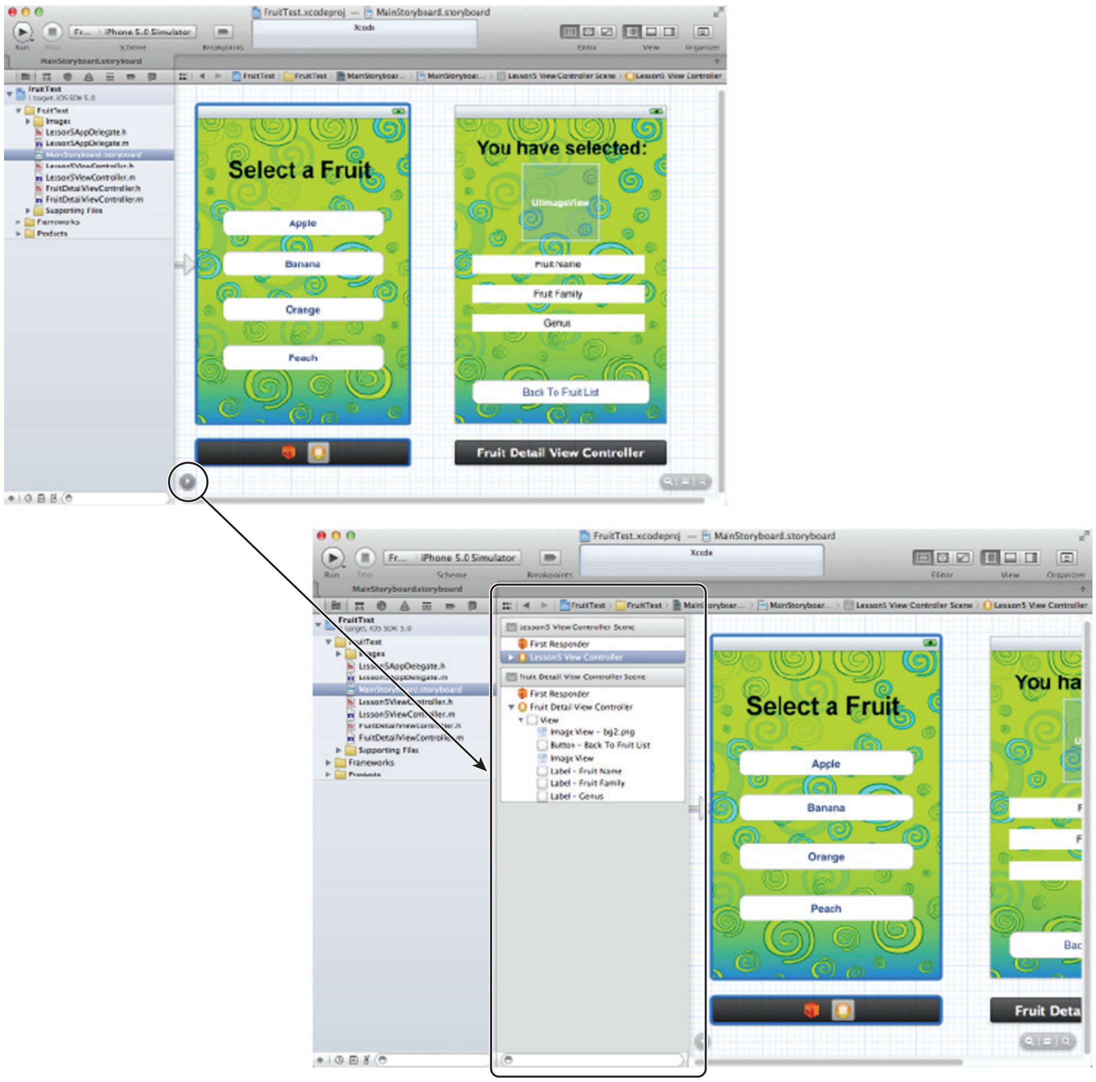


FIGURE 5-4

Objects contained within the scene are shown hierarchically (Figure 5-5). Clicking an object in the hierarchy selects it in the corresponding scene.

To view the entire storyboard at a glance, simply double-click the canvas to zoom out. Double-click a scene to select and zoom in to the storyboard.

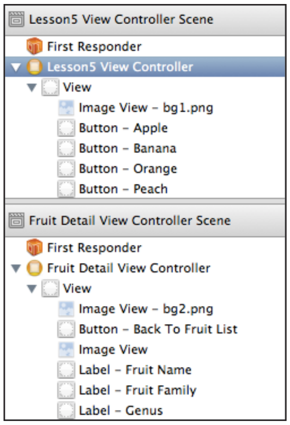
A *segue* represents the transition between one scene to another. It also represents the manner in which the new scene is presented. Segues are represented by arrows between scenes (Figure 5-6).

There are three different types of segues: Modal, Push, and Custom. Modal segues are used to present modal content; they enable you to specify a transition style, the most common of which is one where the new scene slides up from the bottom of the screen. Push segues are used in conjunction with a navigation controller to slide a new scene onto the screen. A Custom segue enables you to specify the presentation style.

You can set up the type and attributes of a segue by selecting it and using the Attributes inspector (Figure 5-7).

You can select a segue by clicking the circle in the middle of the arrow representing the segue on the storyboard (Figure 5-8). Each segue in your application must be uniquely identified by a string. This identifier can also be set up using the Attributes inspector.

When you create a new Xcode project that uses storyboards, the storyboard contains a default scene. To add a new scene to a storyboard, simply drag and drop a View Controller object from the Object library onto the canvas (Figure 5-9).



Lesson5 View Controller Scene
- First Responder
- ▼ Lesson5 View Controller
  - ▼ View
    - Image View – bg1.png
    - Button – Apple
    - Button – Banana
    - Button – Orange
    - Button – Peach

Fruit Detail View Controller Scene
- First Responder
- ▼ Fruit Detail View Controller
  - ▼ View
    - Image View – bg2.png
    - Button – Back To Fruit List
    - Image View
    - Label – Fruit Name
    - Label – Fruit Family
    - Label – Genus

FIGURE 5-5

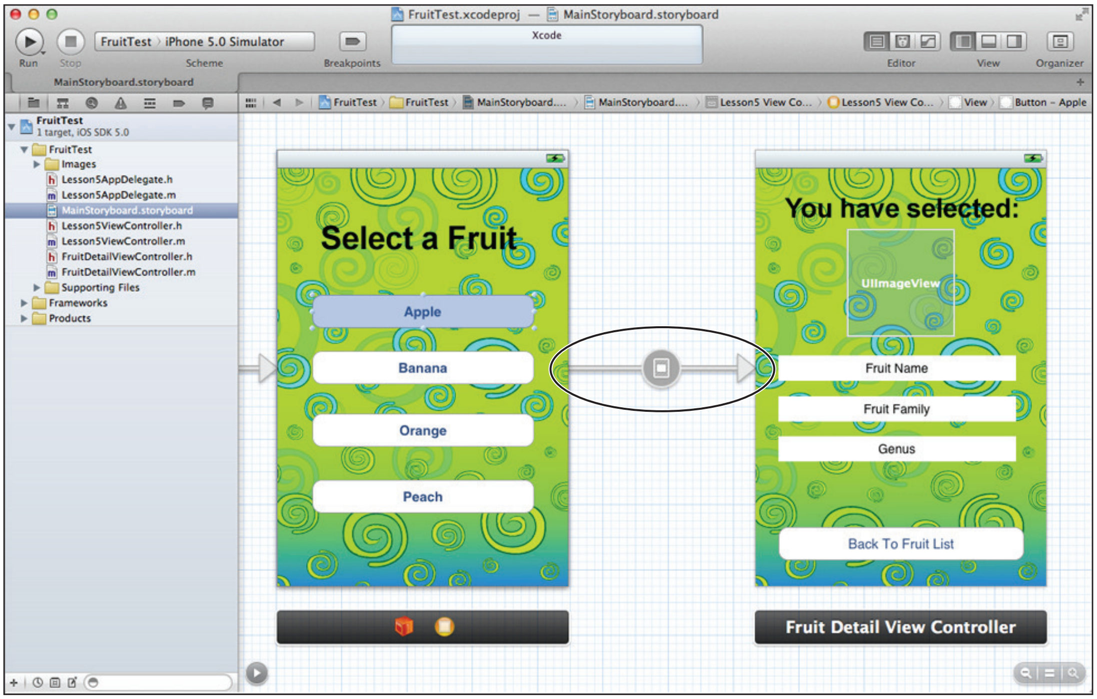
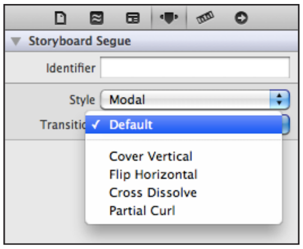

FruitTest.xcodeproj — MainStoryboard.storyboard
Xcode
Run
Stop
FruitTest 〉 iPhone 5.0 Simulator
Scheme
Breakpoints
Editor
View
Organizer
MainStoryboard.storyboard
FruitTest 〉 FruitTest 〉 MainStoryboard.... 〉 MainStoryboard.... 〉 Lesson5 View Co... 〉 Lesson5 View Co... 〉 View 〉 Button – Apple
FruitTest
1 target, iOS SDK 5.0
FruitTest
Images
Lesson5AppDelegate.h
Lesson5AppDelegate.m
MainStoryboard.storyboard
Lesson5ViewController.h
Lesson5ViewController.m
FruitDetailViewController.h
FruitDetailViewController.m
Supporting Files
Frameworks
Products
Select a Fruit
Apple
Banana
Orange
Peach
You have selected:
UIImageView
Fruit Name
Fruit Family
Genus
Back To Fruit List
Fruit Detail View Controller

FIGURE 5-6


Storyboard Segue

Identifier

Style: Modal

Transition: ✓ Default

Cover Vertical
Flip Horizontal
Cross Dissolve
Partial Curl

FIGURE 5-7

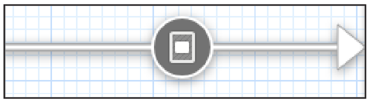


FIGURE 5-8

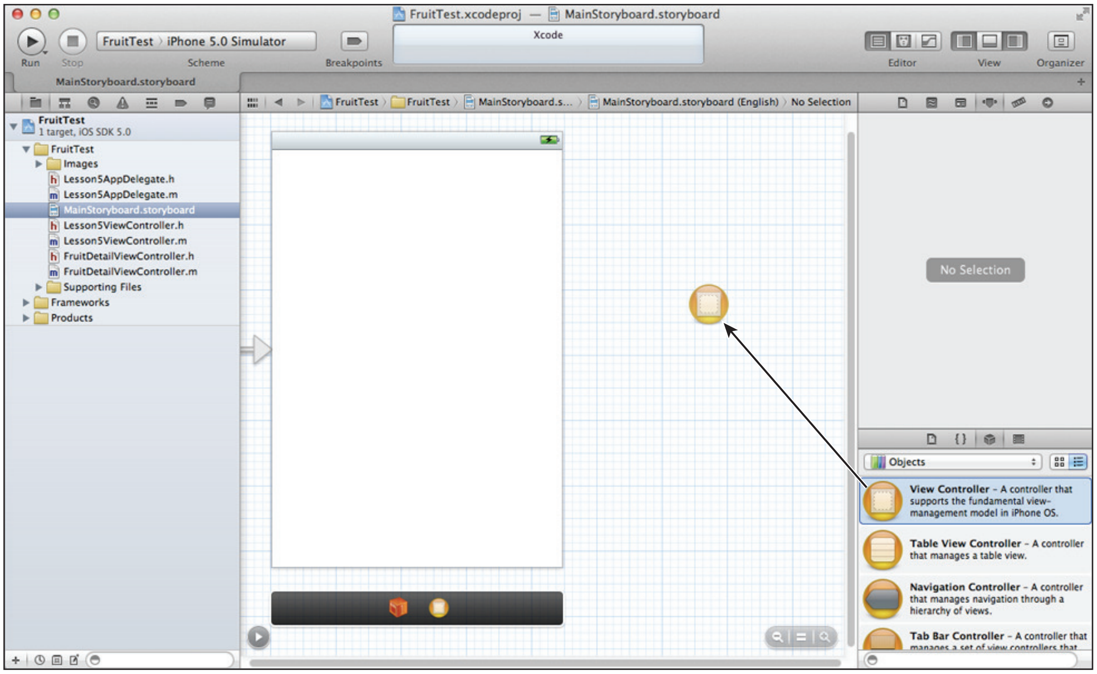
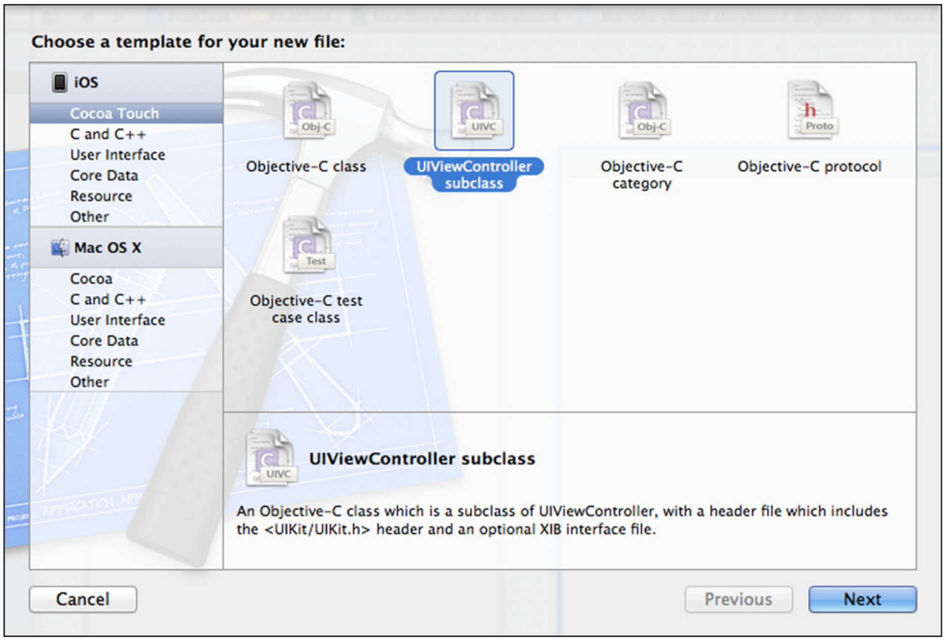

FruitTest.xcodeproj — MainStoryboard.storyboard
Xcode
Run
Stop
FruitTest › iPhone 5.0 Simulator
Scheme
Breakpoints
Editor
View
Organizer
MainStoryboard.storyboard
FruitTest › FruitTest › MainStoryboard.s... › MainStoryboard.storyboard (English) › No Selection
FruitTest
1 target, iOS SDK 5.0
FruitTest
Images
Lesson5AppDelegate.h
Lesson5AppDelegate.m
MainStoryboard.storyboard
Lesson5ViewController.h
Lesson5ViewController.m
FruitDetailViewController.h
FruitDetailViewController.m
Supporting Files
Frameworks
Products
No Selection
Objects
View Controller – A controller that supports the fundamental view-management model in iPhone OS.
Table View Controller – A controller that manages a table view.
Navigation Controller – A controller that manages navigation through a hierarchy of views.
Tab Bar Controller – A controller that manages a set of view controllers that

FIGURE 5-9

Although you can add interface elements to the new scene by simply dragging and dropping objects from the Object library, to create outlets and actions for these elements you first need to create a `UIViewController` subclass that does not have an associated XIB file, and link it to the new scene.

To create a new `UIViewController` subclass, simply right-click the project in the project navigator and select New File from the context menu. Select the UIViewController Subclass template in the dialog box that appears and click Next (Figure 5-10).

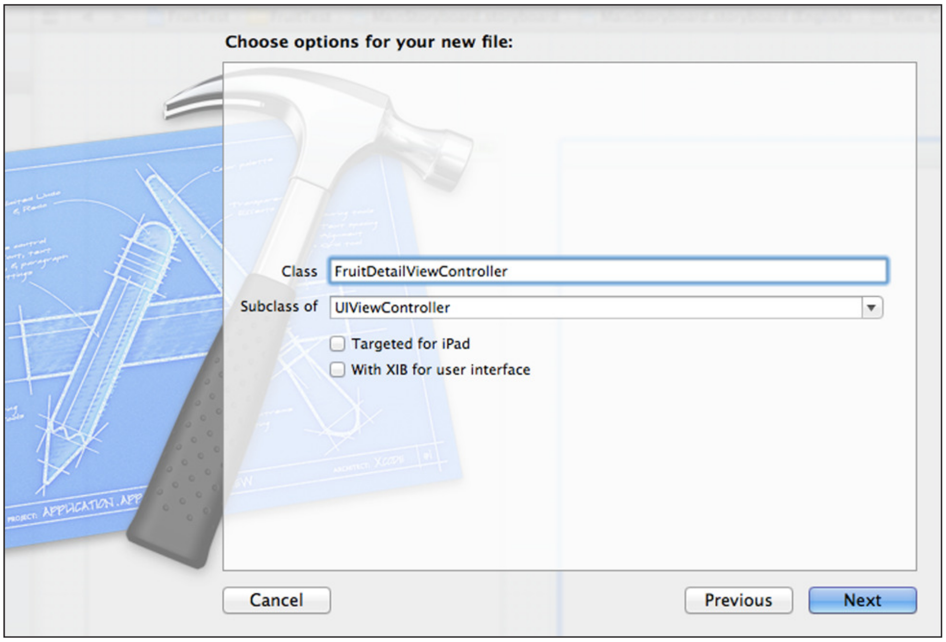


UIViewController subclass

An Objective-C class which is a subclass of UIViewController, with a header file which includes the <UIKit/UIKit.h> header and an optional XIB interface file.

FIGURE 5-10

In the file options dialog box for the `UIViewController` subclass, provide a name for the new class and ensure the With XIB for User Interface checkbox is unchecked (Figure 5-11).



Class: FruitDetailViewController
Subclass of: UIViewController
☐ Targeted for iPad
☐ With XIB for user interface

FIGURE 5-11

After you create your `UIViewController` subclass, you need to associate it with the new scene in the storyboard. To do so, simply select the scene in the storyboard, select the view controller object (the yellow box) in the dock and choose the appropriate class name in the Identity inspector (Figure 5-12).

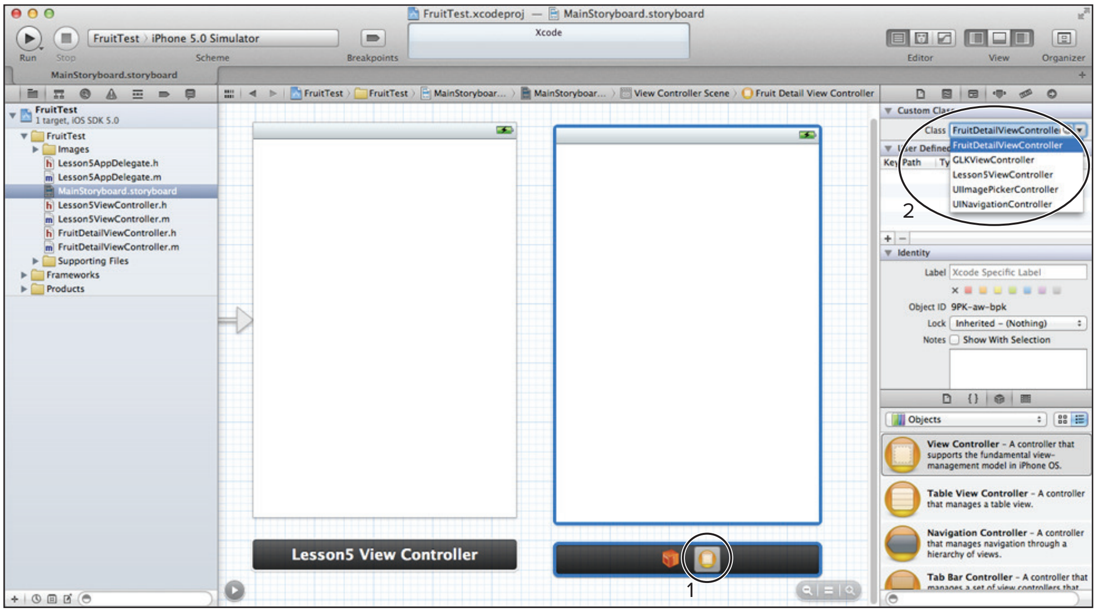


FIGURE 5-12

To create a segue from an object in one scene to another scene, simply right-click the object to display a context menu and drag from the circle beside one of the entries listed under the Storyboard Segues category to the target scene (Figure 5-13).

Alternately, you can Ctrl+drag from the object to the target scene and select an option from the context menu that appears when you release the mouse button.

Click the new segue to select it, and use the Attributes inspector to give it a unique string identifier. To perform some tasks in the source view controller when a segue is about to be performed, override the `prepareForSegue:sender:` method in the source view controller class.

You could potentially have several buttons in the source view controller, each going to different scenes of the storyboard with individual segues. If you override the `prepareForSegue:sender:` method in the source view controller, your version of this method will be called regardless of which segue is in action. Within this method you need to provide code to determine which segue is in action, and take appropriate steps.

The first argument of this method is a `UIStoryboardSegue` object that represents the segue about to be performed. The second parameter is a reference to the object that initiated the segue.

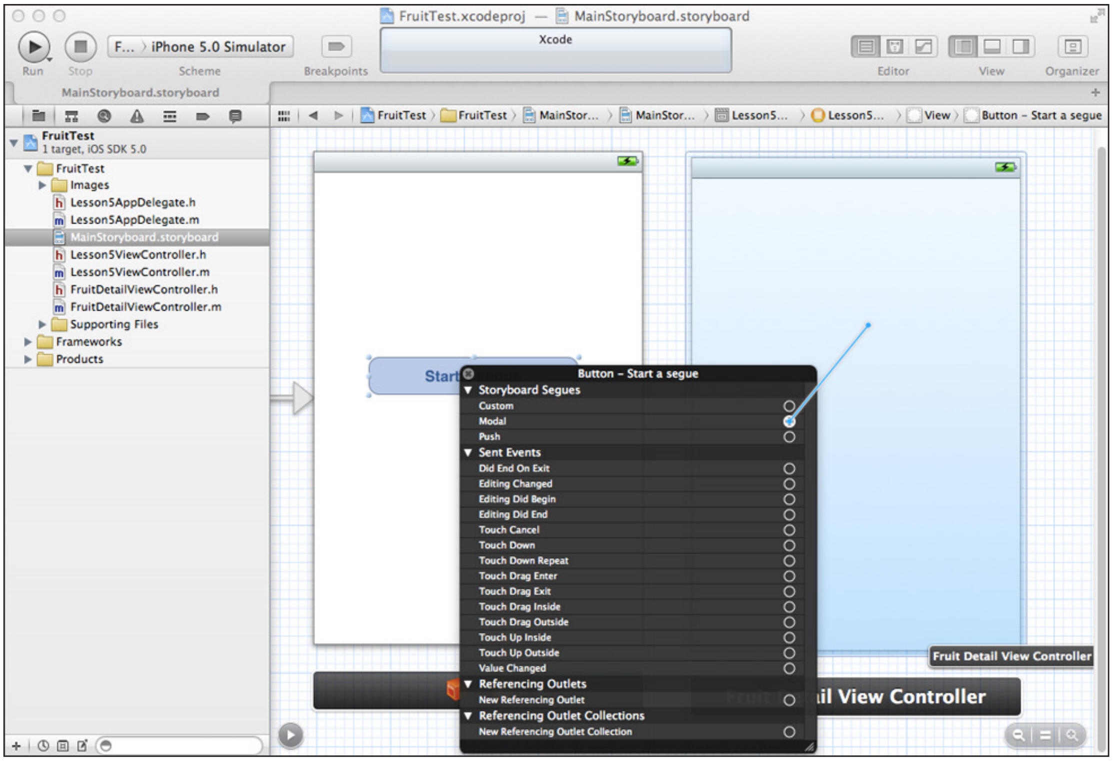


FIGURE 5-13

The `UIStoryboardSegue` object provides the `identifier` property, which contains the unique string identifier specified using the Attributes inspector. The `UIStoryboardSegue` object also provides the `sourceViewController` and `destinationViewController` properties that you can use to retrieve a reference to the source and target view controllers involved in the transition. You can use this information to set up properties in the destination view controller before it is displayed.

TRY IT

In this Try It, you create a new Xcode project based on the Single View Application called `FruitList` using storyboards. You use Interface Builder to add an additional scene to the storyboard. In the default scene you present the user with a short list of fruits, and in the second scene you show detailed information on the fruit selected in the first scene. The user will be able to get back to the first scene from the second scene.

Lesson Requirements

➤ Launch Xcode.

➤ Create a new project based on the Single View Application template.

- Add image resources to your project.
- Add a new `NSObject` subclass to your project `FruitClass`.
- Add an array to the `FruitListViewController` class, and add four instances of `FruitClass` to this array.
- Edit the storyboard with Interface Builder.
- Add four `UIButton` instances to the default scene, each containing the name of a fruit.
- Create an additional scene in the storyboard, and a new `UIViewController` subclass called `FruitDetailViewController` in the project.
- Use the Identity inspector to change the Custom Class of the new scene to `FruitDetailViewController`.
- Create segues from the four buttons in the first scene to the second scene.
- Override the `prepareForSegue:sender` method in the `Lesson5ViewController` class to pass information on the selected fruit to the second scene.
- Add user interface elements and code to the second scene to display information on a fruit.
- Add a `UIButton` to the second scene to dismiss it.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 5 folder in the download.*

Hints

- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.
- To show the assistant editor, use the View ➪ Assistant Editor Show Assistant Editor menu item.

Step-by-Step

1. Create a Single View Application in Xcode called `FruitList`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** FruitTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson5
        - **Define Family:** iPhone

- **Use Storyboard:** Checked
- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Add image resources to your project.
    1. Ensure the project navigator is visible. To show it use the View ➪ Navigators ➪ Show Project Navigator menu item.
    2. Right-click the FruitTest group in the project navigator and select Add Files to “FruitTest” from the context menu.
    3. Navigate to the `Images` folder in this chapter’s resources on the DVD.
    4. Ensure the Copy Items into Destination Group’s Folders (if needed) option is selected.
    5. Click the Add button.
3. Create an `NSObject` subclass called `FruitClass`.
    1. Ensure the project navigator is visible.
    2. Right-click the FruitTest group and select New File from the context menu.
    3. Select the Objective-C class template and click Next.
    4. Call the new class FruitClass and ensure that the new class is a subclass of `NSObject` by selecting NSObject in the drop-down combo box and click Next.
    5. Select a folder where files should be created. It is best to accept the default location provided by Xcode.
    6. Modify the `FruitClass.h` file to resemble the following:

```
#import <Foundation/Foundation.h>
@interface FruitClass : NSObject
@property (strong, nonatomic) NSString* fruitName;
@property (strong, nonatomic) NSString* imageFilename;
@property (strong, nonatomic) NSString* family;
@property (strong, nonatomic) NSString* genus;
@end
```

7. Modify the `FruitClass.m` file to resemble the following:

```
#import "FruitClass.h"
@implementation FruitClass
@synthesize fruitName;
@synthesize imageFilename;
@synthesize family;
@synthesize genus;
@end
```

4. Add an `NSArray` member to the `Lesson5ViewController` class and populate it with four `FruitClass` instances.

   1. Add the following property declaration to the `Lesson5ViewController` class:

      ```
      @property (strong, nonatomic) NSArray* arrayOfFruits;
      ```

   2. Synthesize the property in the implementation file by adding the following line:

      ```
      @synthesize arrayOfFruits;
      ```

      after the line:

      ```
      @implementation FruitTestViewController
      ```

   3. Import the definition of `FruitClass` in the `Lesson5ViewController.m` file by adding the following line to the top of the file:

      ```
      #import "FruitClass.h"
      ```

      after the line:

      ```
      #import "Lesson5ViewController.h"
      ```

   4. Instantiate the `arrayOfFruits` member variable and add data to it in the `viewDidLoad` method of the view controller class:

      ```
      - (void)viewDidLoad
      {
          [super viewDidLoad];
          FruitClass* apple = [[FruitClass alloc] init];
          apple.fruitName = @"Apple";
          apple.imageFilename = @"apple.png";
          apple.family = @"Rosaceae";
          apple.genus = @"Malus";

          FruitClass* banana = [[FruitClass alloc] init];
          banana.fruitName = @"Banana";
          banana.imageFilename = @"banana.png";
          banana.family = @"Musaceae";
          banana.genus = @"Musa";

          FruitClass* orange = [[FruitClass alloc] init];
          orange.fruitName = @"Orange";
          orange.imageFilename = @"orange.png";
          orange.family = @"Rutaceae";
          orange.genus = @"Citrus";
      ```

```
        FruitClass* peach = [[FruitClass alloc] init];
        peach.fruitName = @"Peach";
        peach.imageFilename = @"peach.png";
        peach.family = @"Rosaceae";
        peach.genus = @"Prunus";

        arrayOfFruits = [[NSArray alloc] initWithObjects:apple, banana,
                                orange, peach, nil];
    }
```

5. Add a new subclass of `UIViewController` called `FruitDetailViewController`.
    1. Ensure the project navigator is visible.
    2. Right-click the FruitTest group and select New File from the context menu.
    3. Select the UIViewController Subclass template and click Next.
    4. Call the new class FruitDetailViewController and ensure that the new class is a subclass of `UIViewController` by selecting `UIViewController` in the drop-down combo box.
    5. Ensure that With XIB for User Interface option is unchecked and click Next.
    6. Select a folder where files should be created. It is best to accept the default location provided by Xcode.
6. Open the `MainStoryboard.storyboard` file in the Xcode Interface Builder.
    1. Ensure the project navigator is visible and the FruitTest project is selected and expanded.
    2. Click the `MainStoryboard.storyboard` file.
    3. Ensure the utilities editor is visible. To show the utilities editor, use the View ➪ Utilities ➪ Show Utilities menu item.
7. Edit the first scene in the storyboard.
    1. Ensure the Media library is visible. To show it, use the View ➪ Utilities ➪ Show Media Library menu item.
    2. From the Media library, drag and drop the `bg1.png` file onto the scene.
    3. Select the image in the scene, and use the Size inspector to position it at X = 0, Y = –20. To show the Size inspector, use the View ➪ Utilities ➪ Show Size Inspector menu item.
    4. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
    5. Add four Round Rect Button instances from the Object library to the scene and size/position them as shown in Table 5-1. Instead of dragging four separate instances from the Object library, you may wish to drag just one and then duplicate it by selecting the button in the scene and Option+dragging it to a new location on the scene to duplicate it.

TABLE 5-1: Button Positions

	X	Y	W	H
First button	35	143	250	37
Second button	35	201	250	37
Third button	35	265	250	37
Fourth button	35	329	250	37

6. Double-click each button to edit the text displayed in it. Set the text in the buttons to **Apple**, **Banana**, **Orange**, and **Peach** from top to bottom (Figure 5-14).

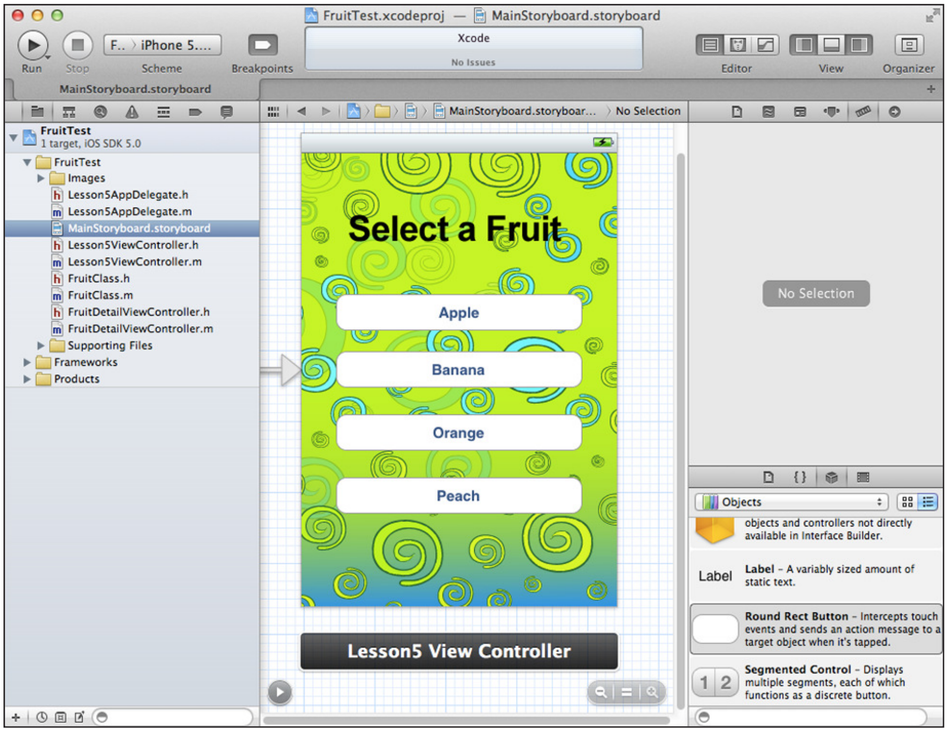

FruitTest.xcodeproj — MainStoryboard.storyboard
Run
Stop
F.. iPhone 5....
Scheme
Breakpoints
Xcode
No Issues
Editor
View
Organizer
MainStoryboard.storyboard
MainStoryboard.storyboar... No Selection
FruitTest
1 target, iOS SDK 5.0
FruitTest
Images
Lesson5AppDelegate.h
Lesson5AppDelegate.m
MainStoryboard.storyboard
Lesson5ViewController.h
Lesson5ViewController.m
FruitClass.h
FruitClass.m
FruitDetailViewController.h
FruitDetailViewController.m
Supporting Files
Frameworks
Products
Select a Fruit
Apple
Banana
Orange
Peach
Lesson5 View Controller
No Selection
Objects
objects and controllers not directly available in Interface Builder.
Label
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 5-14

8. Add a new scene to the storyboard.
    1. Drag a View Controller object from the Object library onto the storyboard canvas.
    2. Double-click the canvas to zoom out.
    3. Position the new scene alongside the original scene.

4. Select the new scene in the storyboard, select the View Controller object from the dock and use the Identity inspector to change its Custom Class to `FruitDetailViewController`. To show the Identity inspector, use the View ➪ Utilities ➪ Show Identity inspector menu item.

9. Add user interface elements to the new scene.
   1. Select the Fruit Detail View Controller scene to select it, and use the scroll bars to center it in the view area.
   2. From the Media library, drag and drop the `bg2.png` file onto the scene.
   3. Select the image in the scene, and use the Size inspector to position it at X = 0, Y = –20.
   4. Ensure the Object library is visible.
   5. Add a Round Rect Button instance to the scene, and use the Size Inspector to size/position it at X = 29, Y=403, W = 271, H=37.
   6. Double-click the button and edit the text displayed in it to Back To Fruit List.
   7. Use the Object library to add an Image View object to the scene. Use the Size inspector to size/position it at X = 104, Y=74, W=120, H=120.
   8. Use the Object library to add three Label instances. Use the Size inspector to size/position them as per Table 5-2.

TABLE 5-2: Label Positions

	X	Y	W	H
First label	29	214	262	28
Second label	29	259	262	28
Third label	29	303	262	28

   9. Double-click each label and set the text displayed, from top to bottom, to **Fruit Name**, **Fruit Family**, and **Genus**. Use the Attributes inspector to center the text in each label. The second scene in your storyboard should resemble Figure 5-15.

10. Create outlets in the `FruitDetailViewController` class and connect these outlets to user interface elements in the scene.
    1. Ensure the assistant editor is visible. To show it, use the View ➪ Editor ➪ Show Assistant Editor menu item.
    2. Ensure the `FruitDetailViewController.h` file is open in the assistant editor. If it is not use the jump bars to select it (Figure 5-16).
    3. Right-click the `UIImageView` object in the scene to display a context menu. Drag from the circle beside the New Referencing Outlet option in the context menu to an empty line in the interface of the `FruitDetailViewController` class before the `@end` statement.

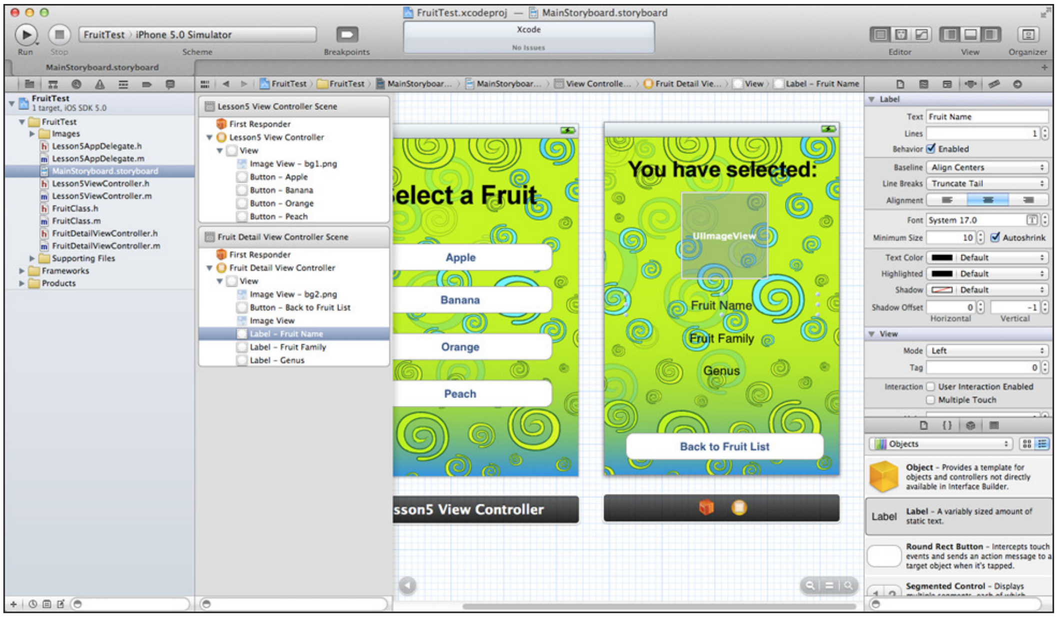
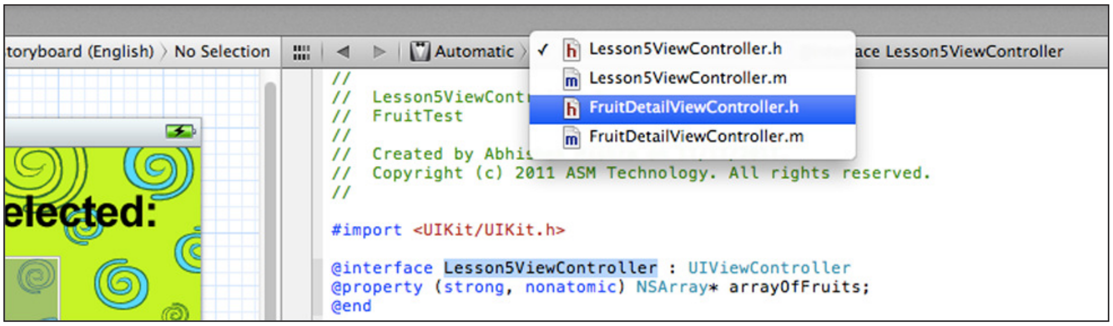


FIGURE 5-15


toryboard (English) › No Selection
Automatic
Lesson5ViewController.h
Lesson5ViewController.m
FruitDetailViewController.h
FruitDetailViewController.m
ace Lesson5ViewController

```
//
//  Lesson5ViewCont
//  FruitTest
//
//  Created by Abhis
//  Copyright (c) 2011 ASM Technology. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson5ViewController : UIViewController
@property (strong, nonatomic) NSArray* arrayOfFruits;
@end
```

FIGURE 5-16

4. This will bring up a dialog where you can specify the name of the new outlet. Name the new outlet `fruitImage`.

5. Create outlets for each of the three label objects and name them `fruitName`, `fruitFamily`, and `fruitGenus`, respectively. Your `FruitDetailViewController.h` file should now resemble:

```
#import <UIKit/UIKit.h>
@interface FruitDetailViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIImageView *fruitImage;
@property (weak, nonatomic) IBOutlet UILabel *fruitName;
@property (weak, nonatomic) IBOutlet UILabel *fruitFamily;
@property (weak, nonatomic) IBOutlet UILabel *fruitGenus;
@end
```

11. Create an action in the `FruitDetailViewController` class and connect it with the Touch Up Inside event of the `UIButton` object.
    1. Right-click the `UIButton` object in the scene to display its context menu, and drag from the circle beside the Touch Up Inside item to an empty line in the `FruitDetailViewController.h` file before the `@end` statement.
    2. Name the new action `onBackButtonPressed`.
    3. Click the `FruitDetailViewController.m` file in the project navigator to open it.
    4. Add the following line to the implementation of the `onBackButtonPressed` method:

       ```
       [self dismissModalViewControllerAnimated:YES];
       ```
12. Create segues in the storyboard.
    1. Open the `MainStoryboard.storyboard` file in Interface Builder.
    2. Double-click the canvas to zoom out. Position the two scenes sufficiently apart on the canvas by dragging them.
    3. Right-click the Apple button to bring up a context menu. Drag from the circle beside the Modal item under the Storyboard Segues category in the context menu to the Fruit Detail View Controller scene (Figure 5-17).

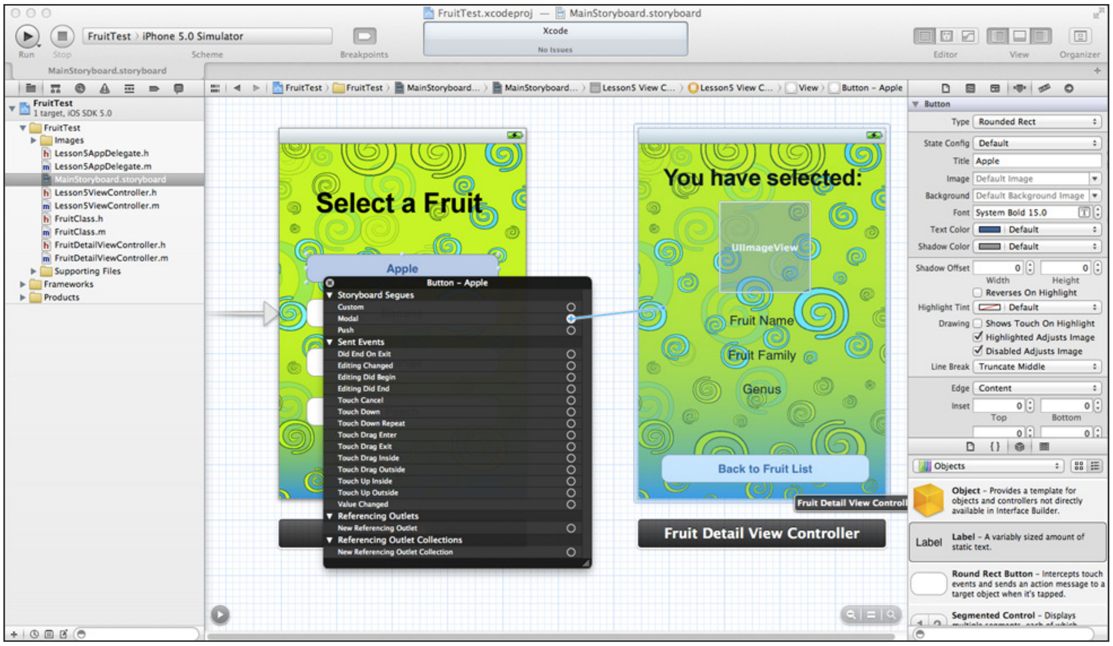
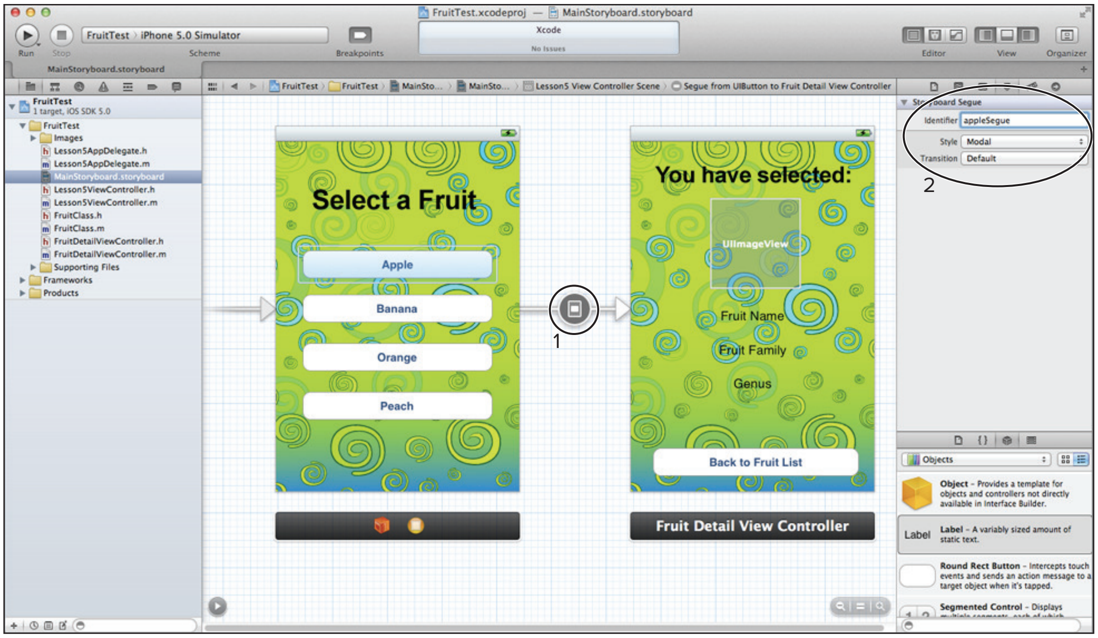


FIGURE 5-17

    4. Select the segue by clicking the circle along the line joining the two scenes and use the Attributes inspector to change the identifier to `appleSegue` (Figure 5-18).



Storyboard Segue
Identifier appleSegue
Style Modal
Transition Default

FIGURE 5-18

5. Similarly, create segues from each of the other three buttons (Banana, Orange, and Peach) in the first scene to the second scene. Name these segues `bananaSegue`, `orangeSegue`, and `peachSegue`, respectively. Your storyboard canvas should resemble Figure 5-19.

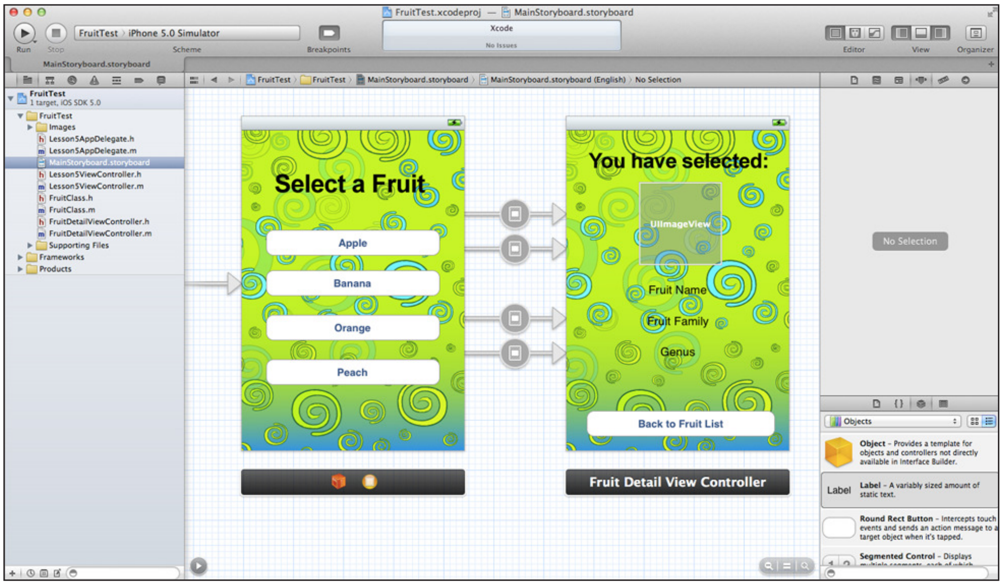


FIGURE 5-19

13. Modify the implementation of the `Lesson5ViewController` class.

    1. Add the following `#import` directive to the top of the `Lesson5ViewController.m` file:

       ```
       #import "FruitDetailViewController.h"
       ```

    2. Implement the `prepareForSegue:sender:` method in as follows:

       ```
       - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
       {
           // apple segue
           if ([segue.identifier isEqualToString:@"appleSegue"])
           {
               FruitClass* appleData = [arrayOfFruits objectAtIndex:0];
               FruitDetailViewController* detailView =
                                        segue.destinationViewController;
               detailView.dataObject = appleData;
           }

           // banana segue
           if ([segue.identifier isEqualToString:@"bananaSegue"])
           {
               FruitClass* bananaData = [arrayOfFruits objectAtIndex:1];
               FruitDetailViewController* detailView =
                                        segue.destinationViewController;
               detailView.dataObject = bananaData;
           }

           // orange segue
           if ([segue.identifier isEqualToString:@"orangeSegue"])
           {
               FruitClass* orangeData = [arrayOfFruits objectAtIndex:2];
               FruitDetailViewController* detailView =
                                        segue.destinationViewController;
               detailView.dataObject = orangeData;
           }

           // peach segue
           if ([segue.identifier isEqualToString:@"peachSegue"])
           {
               FruitClass* peachData = [arrayOfFruits objectAtIndex:3];
               FruitDetailViewController* detailView =
                                        segue.destinationViewController;
               detailView.dataObject = peachData;
           }
       }
       ```

14. Modify the interface of the `FruitDetailViewController` class.

    1. Add the following `#import` directive to the top of the `FruitDetailViewController.h` file:

       ```
       #import "FruitClass.h"
       ```

    2. Add the following property declaration:

       ```
       @property (strong, nonatomic) FruitClass* dataObject;
       ```

15. Modify the implementation of the `FruitDetailViewController` class.

    1. Synthesize the `dataObject` property by adding the following line to the top of the `FruitDetailViewController.m` file:

       ```
       @synthesize dataObject;
       ```

       after the line:

       ```
       @implementation FruitDetailViewController
       ```

    2. Implement the `viewDidLoad` method as follows:

       ```
       - (void)viewDidLoad
       {
           [super viewDidLoad];

           fruitImage.image = [UIImage imageNamed:dataObject.imageFilename];
           fruitName.text = [NSString stringWithFormat:@"Name: %@", dataObject
       .fruitName];
           fruitFamily.text = [NSString stringWithFormat:@"Family: %@", dataObject
       .family];
           fruitGenus.text = [NSString stringWithFormat:@"Genus: %@", dataObject
       .genus];
       }
       ```

16. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run nenu item.

Please select Lesson 5 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

6 Handling User Input

In Lesson 4 you were introduced to the `UILabel` class that enabled you to display static text on the screen. In this lesson you learn to use text fields and text views to accept input from users. Text fields enable users to type a single line of text and are instances of the `UITextField` class. Text views, on the other hand, enable users to type in multiple lines of text and are instances of the `UITextView` class. Both classes are part of the UIKit framework.

TEXT FIELDS

To create a text field, simply drag and drop a Text Field object from the Object library onto a view controller (Figure 6-1).

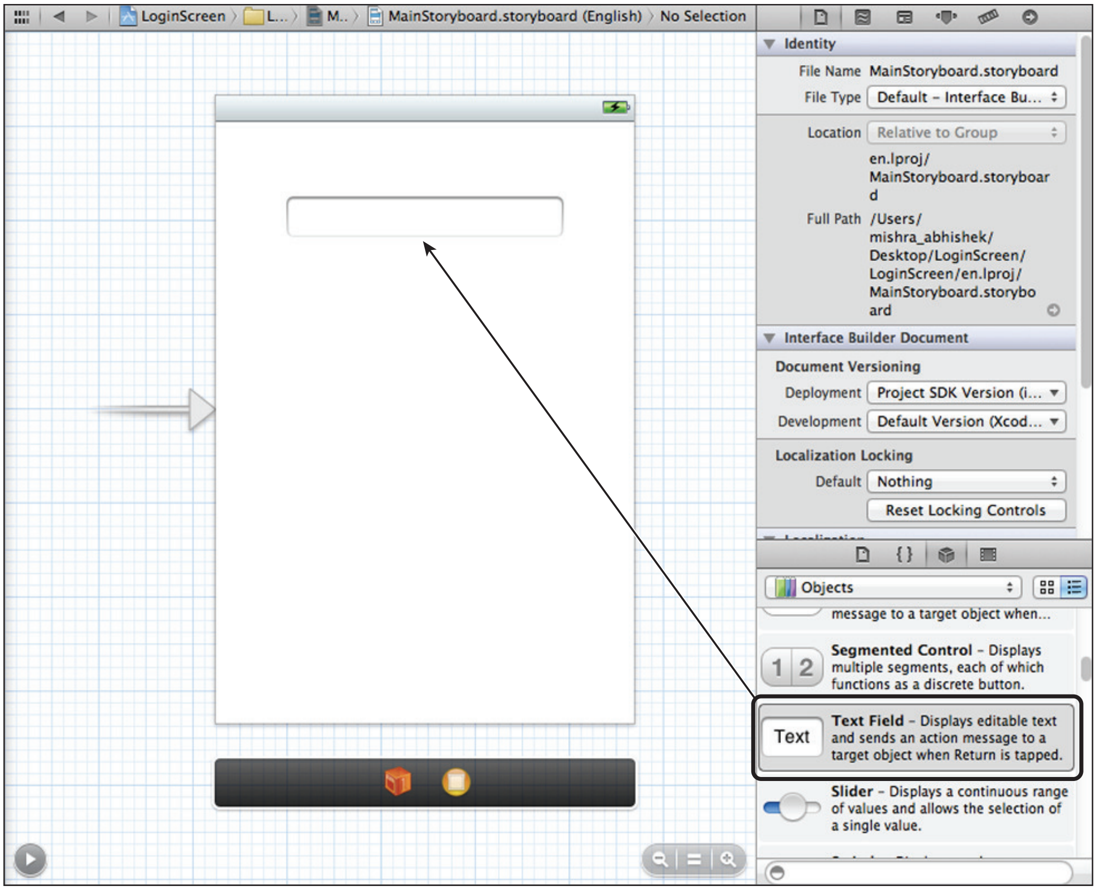
You can use the Attributes inspector to set up several attributes of the text field including the Placeholder, Alignment, Border Style, Text Color, Font Attributes, and the type of keyboard that is displayed when the user taps on the text field (Figure 6-2).

A *placeholder* is some text that is displayed in the text field when it is empty, typically prompting the user to enter some information in the field. You can choose from seven different keyboards to associate with a text field; the choice you make will depend on the type of data you expect. These keyboard styles can be selected using the Attributes inspector and are displayed in Figure 6-3.

The text displayed in a text field is an instance of an `NSString` object. The `NSString` class is defined in the Foundation framework and its instances represent sequences of characters (alphabets, numbers, punctuations) known as *strings*.

To be able to access the text displayed in a text field object from code, you first need to create an outlet in the view controller class and then read the value of the `text` property in your code. For example, if `usernameField` is an outlet created using the assistant editor, you can use the following code to get the text displayed in the field:

```
NSString* theUsername = usernameField.text;
```



Identity

File Name MainStoryboard.storyboard

File Type Default - Interface Bu...

Location Relative to Group

en.lproj/MainStoryboard.storyboard

Full Path /Users/mishra_abhishek/Desktop/LoginScreen/LoginScreen/en.lproj/MainStoryboard.storyboard

Interface Builder Document

Document Versioning

Deployment Project SDK Version (i...

Development Default Version (Xcod...

Localization Locking

Default Nothing

Reset Locking Controls

message to a target object when...

Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Slider – Displays a continuous range of values and allows the selection of a single value.

FIGURE 6-1

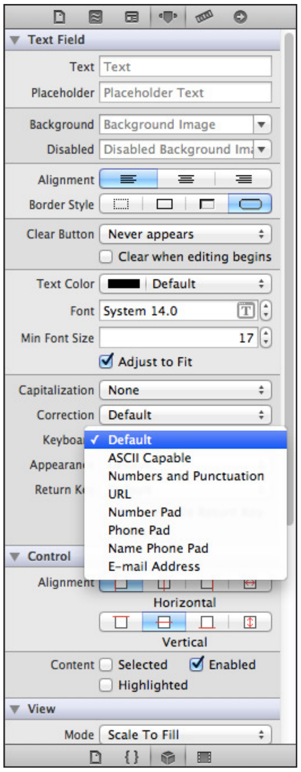
Tapping on a text field signifies that the user wants to interact with it, and as a result makes it the *active user interface element*. The active user interface element is formally known as the first responder. When a text field receives first responder status, it automatically displays a keyboard.

To dismiss a keyboard when the Done button is pressed on the keypad, you will have to use the assistant editor to create an Action method in the view controller class and connect it to the `Did End On Exit` event of the text field (Figure 6-4).

In this method you need to ask the text field to resign from first responder status. You can do this by sending it the `resignFirstResponder` message:

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

Note that the `sender` parameter will contain a reference to the source of the event that triggered this method (which will be the text field).



Text Field

Text: Text
Placeholder: Placeholder Text
Background: Background Image
Disabled: Disabled Background Im
Alignment
Border Style
Clear Button: Never appears
☐ Clear when editing begins
Text Color: Default
Font: System 14.0
Min Font Size: 17
☑ Adjust to Fit
Capitalization: None
Correction: Default
Keyboa: ✓ Default
ASCII Capable
Numbers and Punctuation
URL
Number Pad
Phone Pad
Name Phone Pad
E-mail Address
Appearan
Return K

Control

Alignment
Horizontal
Vertical
Content: ☐ Selected ☑ Enabled
☐ Highlighted

View

Mode: Scale To Fill

FIGURE 6-2

This method of dismissing the keypad works for most keyboard styles, except for the numeric keypads, which don't have a Done button. It is common practice for applications to allow the user to tap the background of the screen (outside the keypad or any other text field) to dismiss the keypad. One way to achieve this is by using a `UITapGestureRecognizer` object. Gesture recognizers are covered in detail in Lesson 29. For the moment, you can add a gesture recognizer to the view controller class by following these simple steps.

1. Add the following method declaration to the interface of the view controller class:

```
- (void) handleBackgroundTap:(UITapGestureRecognizer*)sender;
```

2. Add the following code to the `viewDidLoad` method of the view controller class:

```
UITapGestureRecognizer* tapRecognizer = [[UITapGestureRecognizer alloc]
                                         initWithTarget:self
                                 action:@selector(handleBackgroundTap:)];
tapRecognizer.cancelsTouchesInView = NO;
[self.view addGestureRecognizer:tapRecognizer];
```

3. Implement the `handleBackgroundTap:` method as follows:

```
- (void) handleBackgroundTap:(UITapGestureRecognizer*)sender
{
    [userNameField resignFirstResponder];
}
```

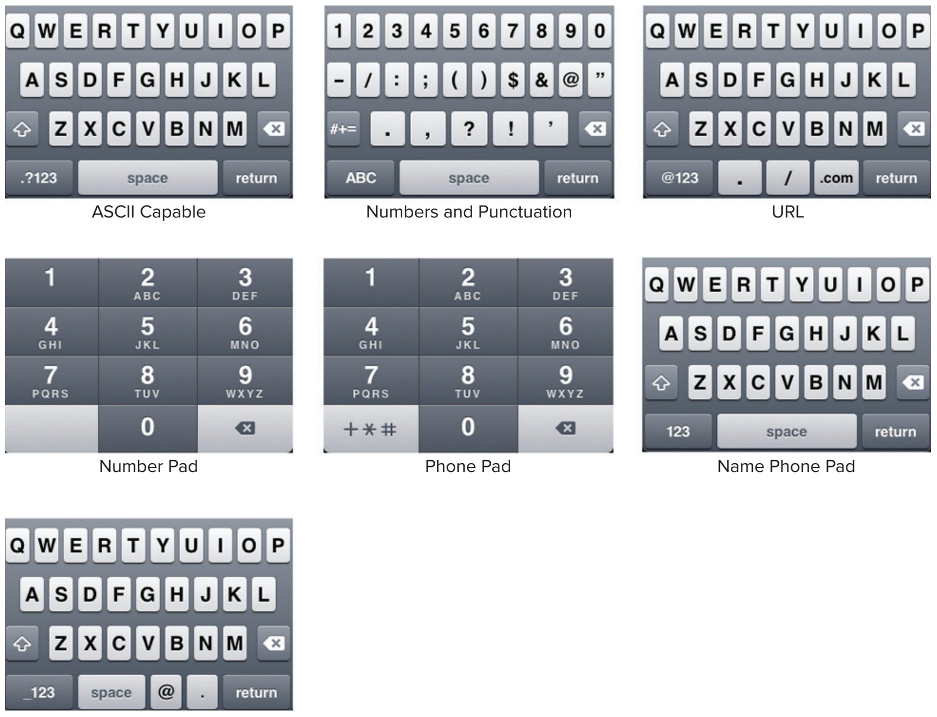
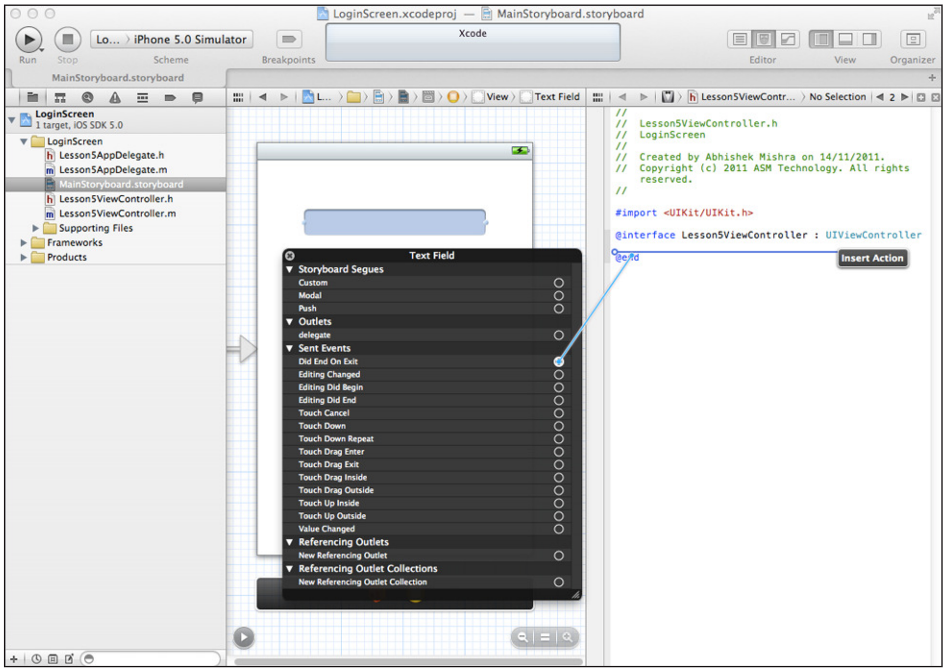


FIGURE 6-3



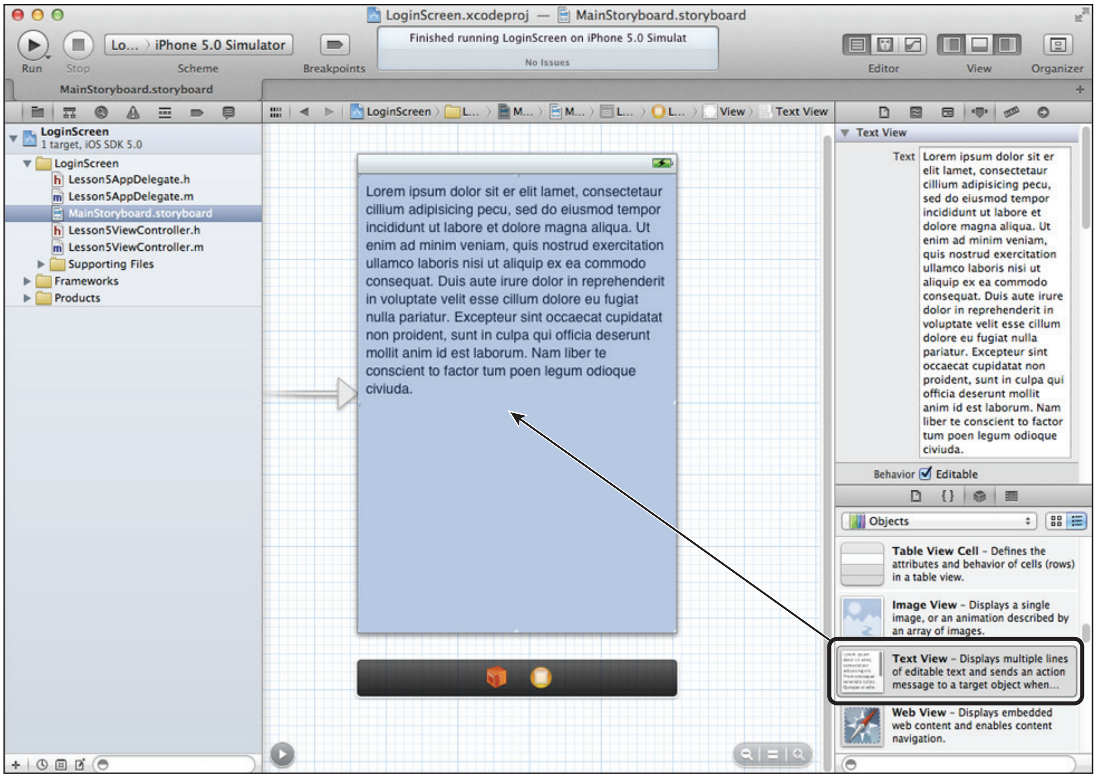
```
//
//  Lesson5ViewController.h
//  LoginScreen
//
//  Created by Abhishek Mishra on 14/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights
//  reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson5ViewController : UIViewController

@end
```

FIGURE 6-4

TEXT VIEWS

Text views are similar to text fields in many respects. The key difference, however, is that text views can handle multiple lines of text. Text views handle the scrolling of text automatically, and can also be used as a read-only view, thus providing a convenient way to display scrollable multi-line text.

To create a text view, simply drag and drop a Text View element from the Object library onto the view (Figure 6-5). By default a text view is sized to fit the entire screen, but you can resize/reposition it as needed.



LoginScreen.xcodeproj — MainStoryboard.storyboard
Run
Stop
Lo... iPhone 5.0 Simulator
Scheme
Breakpoints
Finished running LoginScreen on iPhone 5.0 Simulat
No Issues
Editor
View
Organizer
MainStoryboard.storyboard
LoginScreen
1 target, iOS SDK 5.0
LoginScreen
Lesson5AppDelegate.h
Lesson5AppDelegate.m
MainStoryboard.storyboard
Lesson5ViewController.h
Lesson5ViewController.m
Supporting Files
Frameworks
Products
LoginScreen › L... › M... › M... › L... › L... › View › Text View
Text View
Objects
Table View Cell - Defines the attributes and behavior of cells (rows) in a table view.
Image View - Displays a single image, or an animation described by an array of images.
Text View - Displays multiple lines of editable text and sends an action message to a target object when...
Web View - Displays embedded web content and enables content navigation.

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Nam liber te conscient to factor tum poen legum odioque civiuda.

Behavior [✓ Editable]

FIGURE 6-5

To create a read-only text view, simply uncheck its Editable property in the Attributes inspector. A read-only text view does not display a keypad when tapped. Editable text views also enable you to select from one of seven different keypad types that will appear when the user taps them. The keypad associated with a text view, however, does not have a Done button; instead it has a Return button that adds a new line to the text. Thus, to dismiss the keypad you will have to use the gesture recognizer technique discussed for text fields.

TRY IT

In this Try It, you create a new Xcode project based on the Single View Application template called `LoginSample` that presents a simple user interface to collect a username and password combination from the user. The user interface will also contain a Login button that displays a customized greeting to the user.

Lesson Requirements

- ➤ Launch Xcode.
- ➤ Create a new project based on the Single View Application template.
- ➤ Edit the storyboard with Interface Builder.
- ➤ Add two `UILabel` instances to the default scene, with the text `User name:` and `Password:`, respectively.
- ➤ Add two `UITextField` instances to the same scene, corresponding to the username and password fields, and create appropriate outlets in the view controller for them.
- ➤ Create an action method called `dismissKeyboard:` in the view controller class that sends the `resignFirstResponder` message to each text field, and connect the Did End On Exit event of each text field to this action method.
- ➤ Add a `UIButton` instance to the scene that when tapped, displays a message in an alert view.
- ➤ Use a tap gesture recognizer to dismiss the keyboard when the background is tapped.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 6 folder in the download.*

Hints

- ➤ When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.
- ➤ To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.

Step-by-Step

1. Create a Single View Application in Xcode called `LoginSample`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.

4. Use the following information in the project options dialog box and click Next.
    - **Product Name:** LoginSample
    - **Company Identifier:** com.wileybook
    - **Class Prefix:** Lesson6
    - **Define Family:** iPhone
    - **Use Storyboard:** Checked
    - **Use Automatic Reference Counting:** Checked
    - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This project checkbox is not selected.
7. Click Create.

2. Open the `MainStoryboard.storyboard` file in Interface Builder.
    1. Ensure the project navigator is visible and the LoginSample project is selected and expanded.
    2. Click the `MainStoryboard.storyboard` file.
    3. Ensure the utilities editor is visible. To show the utilities editor, use the View ➪ Utilities ➪ Show Utilities menu item.
3. Add two `UILabel` instances to the default scene.
    1. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
    2. From the Object library, drag and drop two Label objects onto the scene.
    3. Use the Attributes inspector to set the `text` attribute of the first label to `User name:`. To show the Attributes inspector, use the View ➪ Utilities ➪ Show Attributes Inspector menu item.
    4. Change the `text` attribute of the second label to `Password:`.
    5. Size and position the two labels as shown in Table 6-1.

TABLE 6-1: Label Positions

	X	Y	W	H
User name:	20	37	73	21
Password:	20	78	66	21

4. Add two `UITextField` instances to the scene.
    1. From the Object library, drag and drop two Text Field objects onto the scene.
    2. Use the Attributes inspector to set the Placeholder attribute of the first text field to `Enter user name`.
    3. Use the Attributes inspector to set the Placeholder attribute of the second text field to `Enter password`.
    4. Size and position the two text fields as shown in Table 6-2.

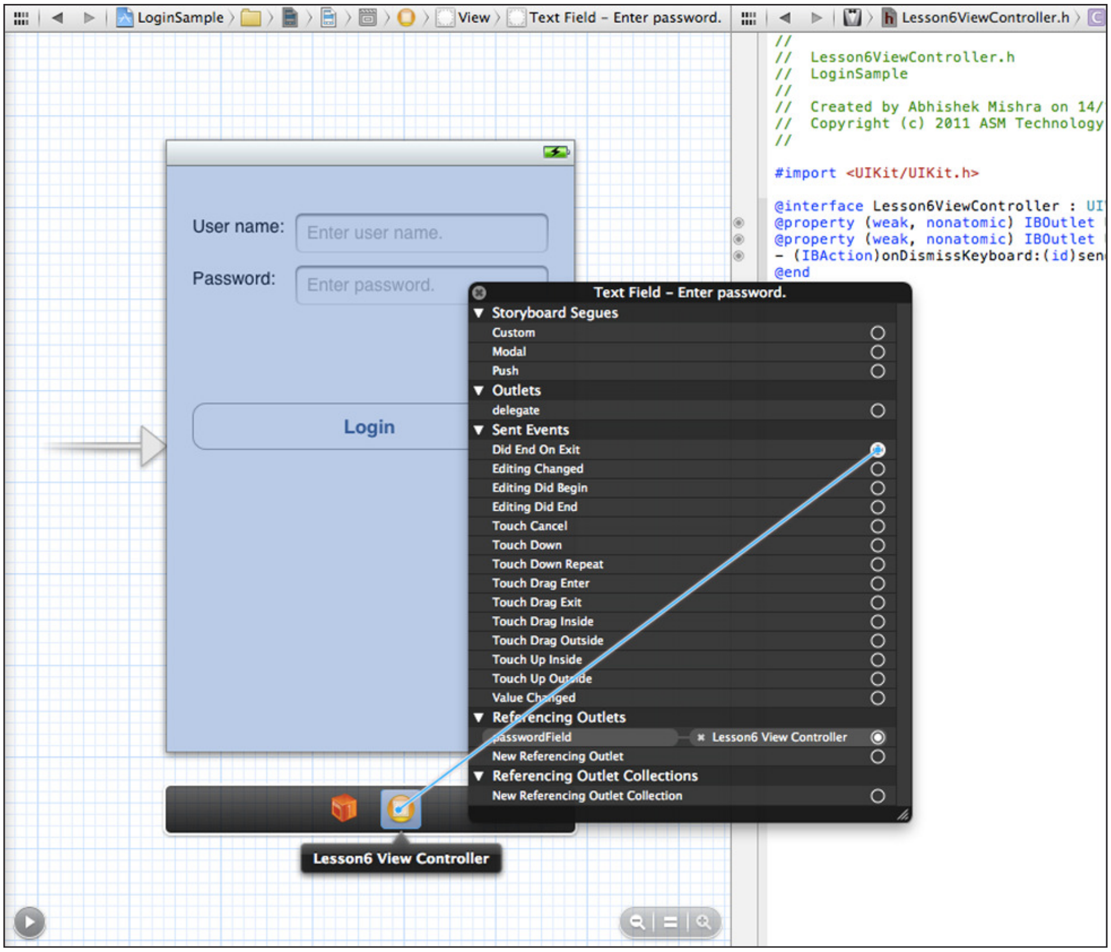
TABLE 6-2: Text Field Positions

PLACEHOLDER	X	Y	W	H
User name:	101	37	199	31
Password:	101	78	199	31

5. Add a `UIButton` instance to the scene.
    1. From the Object library, drag and drop a Round Rect Button object onto the scene.
    2. Double-click it and set the text in the button to `Login`.
    3. Size and position the button to X=20, Y=126, W=280, H=37.
6. Create outlets in the `Lesson6ViewController` class and connect these outlets to the text fields in the scene.
    1. Ensure the assistant editor is visible. To show it, use the View ➪ Editor ➪ Show Assistant Editor menu item.
    2. Right-click the `UITextField` object corresponding to the username to display a context menu. Drag from the circle beside the New Referencing Outlet option in the context menu to an empty line in the interface of the `Lesson6ViewController` class before the `@end` statement.
    3. Name the new outlet `usernameField`.
    4. Repeat this procedure for the password text field, and name the corresponding outlet `passwordField`.

7. Create an action method in the `Lesson6ViewController` class and associate it with the Did End On Exit events of the two text fields.

    1. Right-click the `UITextField` object corresponding to the username to display its context menu, and drag from the circle beside the Did End On Exit item to an empty line in the `Lesson6ViewController.h` file before the `@end` statement.
    2. Name the new Action `onDismissKeyboard`.
    3. Right-click the `UITextField` object corresponding to the password to display its context menu, and drag from the circle beside the Did End On Exit item to the icon representing the view controller in the dock (Figure 6-6).



```
//
//  Lesson6ViewController.h
//  LoginSample
//
//  Created by Abhishek Mishra on 14/
//  Copyright (c) 2011 ASM Technology
//

#import <UIKit/UIKit.h>

@interface Lesson6ViewController : UI
@property (weak, nonatomic) IBOutlet
@property (weak, nonatomic) IBOutlet
- (IBAction)onDismissKeyboard:(id)sen
@end
```

FIGURE 6-6

4. Release the mouse button over the yellow view controller icon in the dock to present a list of existing action methods in the view controller. Select the `onDismissKeyboard` method.
5. Click the `Lesson6ViewController.m` file in the project navigator to open it.
6. Add the following code to the implementation of the `onDismissKeyboard` method:

```
[usernameField resignFirstResponder];
[passwordField resignFirstResponder];
```

8. Create an action in the view controller class and connect it with the Touch Up Inside event of the login button.
    1. Select the storyboard in the project navigator.
    2. Right-click the Login button in the scene to display its context menu, and drag from the circle beside the Touch Up Inside item to an empty line in the `Lesson6ViewController.h` file before the `@end` statement.
    3. Name the new Action `onLogin`.
    4. Click the `Lesson6ViewController.m` file in the project navigator to open it.
    5. Add the following line to the implementation of the `onLogin` method:

```
[usernameField resignFirstResponder];
[passwordField resignFirstResponder];

NSString* username = usernameField.text;

int length = [username length];
if (length == 0)
    return;

NSString* alertMessage = [NSString stringWithFormat:@"Welcome %@",
                                              username];

UIAlertView* welcomeMessage = [[UIAlertView alloc]
                               initWithTitle:@"Login successful"
                               message:alertMessage
                               delegate:nil
                               cancelButtonTitle:@"Ok"
                               otherButtonTitles:nil];

[welcomeMessage show];
```

9. Add a tap gesture recognizer and use it to dismiss the keyboard when the background area of the view is tapped.
    1. Add the following method declaration to the `Lesson6ViewController.h` file:

```
- (void) handleBackgroundTap:(UITapGestureRecognizer*)sender;
```

2. Add the following code to the `viewDidLoad` method of the view controller class, after the `[super viewDidLoad]` line:

```
UITapGestureRecognizer* tapRecognizer = [[UITapGestureRecognizer alloc]
                                         initWithTarget:self
                                 action:@selector(handleBackgroundTap:)];
tapRecognizer.cancelsTouchesInView = NO;
[self.view addGestureRecognizer:tapRecognizer];
```

3. Implement the `handleBackgroundTap:` method in the `Lesson6ViewController.m` file as follows:

```
- (void) handleBackgroundTap:(UITapGestureRecognizer*)sender
{
    [usernameField resignFirstResponder];
}
```

10. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.

Please select Lesson 6 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

7 Communicating with Your Users

The user interface elements you have encountered so far have all been created by dragging and dropping from the Object library. In this lesson you are introduced to `UIAlertView` and `UIActionSheet`, two user interface elements that are created only with code.

ALERT VIEWS

An alert view is a special modal view that is used to display a short message to the user and typically enables the user to choose from a small number of options. The most common use of an alert view is to display information on success or failure of an operation; for example, on success a typical login operation may display an alert view, as shown in Figure 7-1.

When an alert view is displayed, the screen is dimmed automatically for you. You can specify a title, a message, and one or more buttons to present the user with options. One of these buttons is always designated as the cancel button, and though you can change the text displayed in it, it is always displayed at the bottom of the alert view with a small offset from the other buttons, as shown in Figure 7-2.

An alert view is an instance of the `UIAlertView` class, which is part of the UIKit framework, and is created in code as follows:

```
UIAlertView* message = [[UIAlertView alloc]
                        initWithTitle:@"This is the title"
                        message:@"This is the message text"
                        delegate:self
                        cancelButtonTitle:@"Cancel Button"
                        otherButtonTitles:@"Option 1", @"Option 2", nil];
```

The first parameter is the title of the alert view. This is followed by the message. You can provide an optional delegate object that is notified when the user clicks one of the buttons in the alert view. This delegate object must implement the `UIAlertViewDelegate` protocol, and is specified in the third parameter. The alert view is dismissed automatically when the user presses one of the buttons. If you do not specify a delegate object, you have no way to find out which button was pressed. To use the view controller class as the delegate object, specify `self` for the `delegate` parameter.

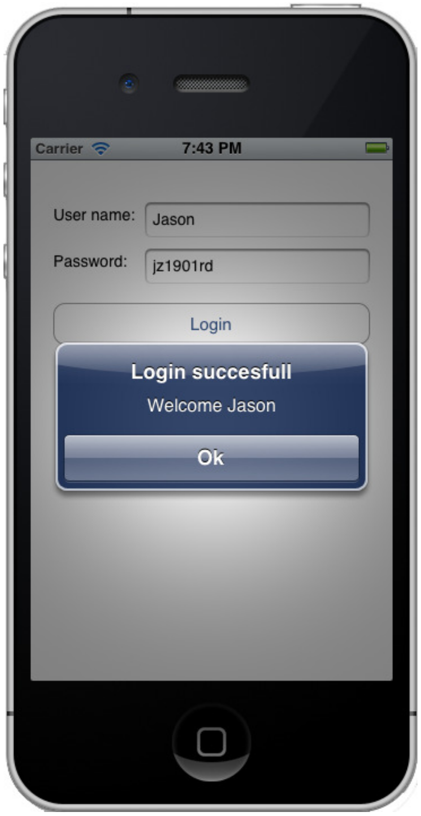


FIGURE 7-1

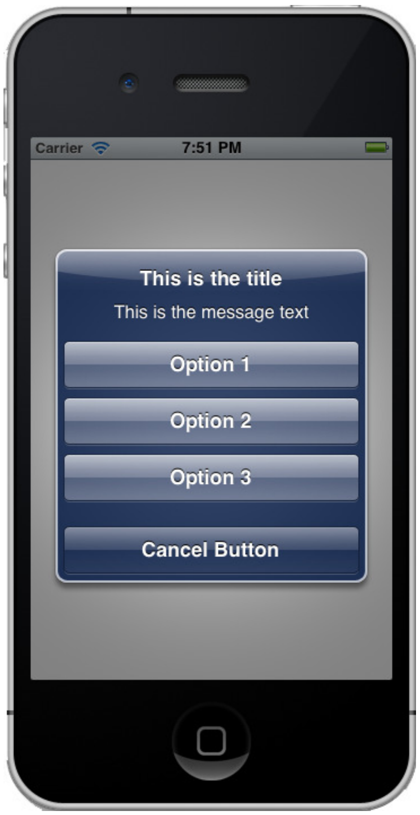


FIGURE 7-2

The fourth parameter, `cancelButtonTitle`, enables you to specify the text to be displayed in the cancel button. You can specify the titles of additional buttons in the last parameter, `otherButtonTitles`. This parameter contains a list of button titles separated by commas. The last title in the list must always be `nil`. If you want no additional buttons, simply set this parameter to `nil`.

To show the alert view, simply send it the `show` message as follows:

```
[message show];
```

To determine which button was pressed, implement the `alertView:clickedButtonAtIndex:` method in the delegate object as follows:

```
- (void)alertView:(UIAlertView *)alertView
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == alertView.cancelButtonIndex)
    {
        // cancel button pressed
    }
    else if (buttonIndex == alertView.firstOtherButtonIndex)
    {
        // first button pressed
    }
    else if (buttonIndex == alertView.firstOtherButtonIndex + 1)
    {
        // second button pressed
    }
}
```

The first parameter to this method is a reference to the alert view object itself. The second parameter, `buttonIndex`, is an integer that contains the index number of the button that was pressed. The `UIAlertView` object defines two standard properties, the value of which should be used to interpret the `buttonIndex` parameter.

The first is `cancelButtonIndex`. This stores the index position of the cancel button. Thus, to determine if the cancel button was pressed, you would use an `if` statement as follows:

```
    if (buttonIndex == alertView.cancelButtonIndex)
    {
        // cancel button pressed
    }
```

The second is `firstOtherButtonIndex`. This stores the index position of the first of the optional buttons specified while creating the alert view object. Recall that you can specify a list of optional button titles while creating the alert view object in the `otherButtonTitles` parameter. Thus, to determine if the first optional button was pressed, you would use an `if` statement as follows:

```
    if (buttonIndex == alertView.firstOtherButtonIndex)
    {
        // first button pressed
    }
```

Similarly, to determine if the second optional button was pressed, you would use the following `if` statement:

```
    if (buttonIndex == alertView.firstOtherButtonIndex + 1)
    {
        // second button pressed
    }
```

The alert view object enables you to add up to two text fields, in addition to buttons. This comes in handy when you want to collect username and password information from the user (Figure 7-3).

To do this, you can set the value of the `alertViewStyle` property of the alert view object before displaying it to the user. The value of this property can be one of the following:

- ➤ `UIAlertViewStyleDefault` — This is the default style, with no text fields.
- ➤ `UIAlertViewStyleSecureTextInput` — The alert view contains one text field, and any text typed by the user is masked.
- ➤ `UIAlertViewStylePlainTextInput` — The alert view contains one text field, and any text typed by the user is visible.
- ➤ `UIAlertViewStyleLoginAndPasswordInput` — The alert view contains two text fields, the first of which is an unmasked field and the other a masked field.

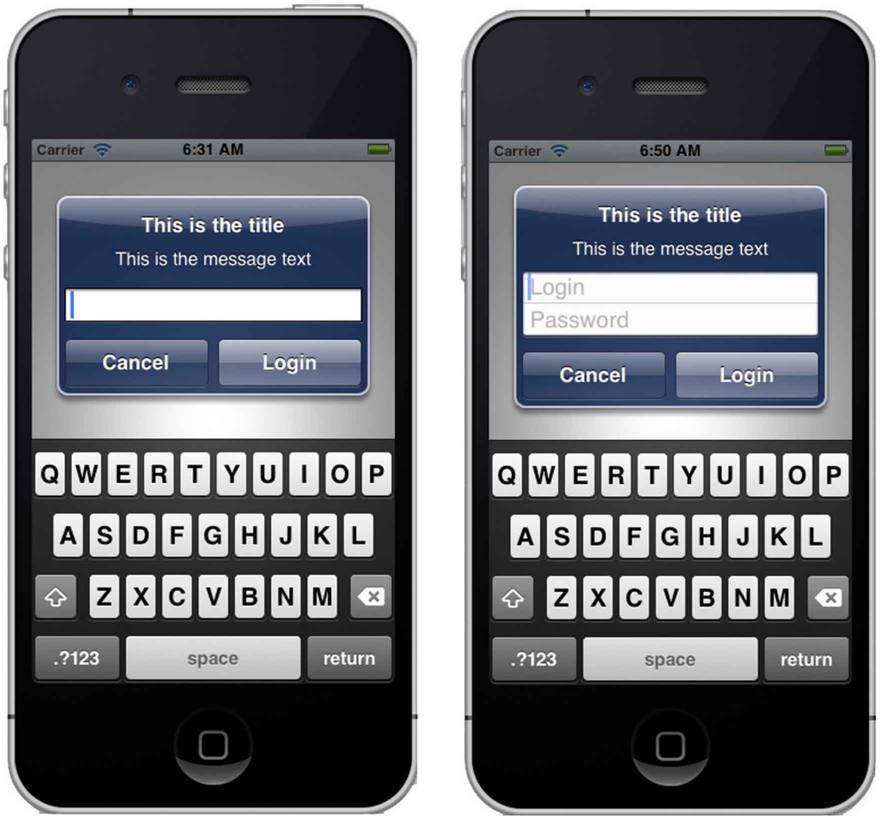


FIGURE 7-3

If you want to create an alert view with a single masked text field, you can do it as follows:

```
UIAlertView* message = [[UIAlertView alloc]
                         initWithTitle:@"This is the title"
                         message:@"This is the message text"
                         delegate:self
                         cancelButtonTitle:@"Cancel"
                         otherButtonTitles:@"Login", nil];

message.alertViewStyle = UIAlertViewStyleSecureTextInput;
[message show];
```

To retrieve the value typed by the user when the alert view is dismissed, you need to retrieve a reference to the `UITextField` object within the alert view and read its value in the `alertView:clickedButtonAtIndex:` delegate method as follows:

```
- (void)alertView:(UIAlertView *)alertView
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    UITextField* field1 = [alertView textFieldAtIndex:0];
    NSString* username = field1.text;
```

```
        if (buttonIndex == alertView.cancelButtonIndex)
        {
            // cancel button pressed
        }
        else if (buttonIndex == alertView.firstOtherButtonIndex)
        {
            // Login button pressed
        }
    }
```

ACTION SHEETS

An action sheet is another user interface component that is created through code, and can be used to present a list of choices to a user. Action sheets are similar to alert views in many respects; however, they have several important differences. To start with, action sheets look significantly different from alert views, and they look different on an iPhone and an iPad (Figure 7-4).

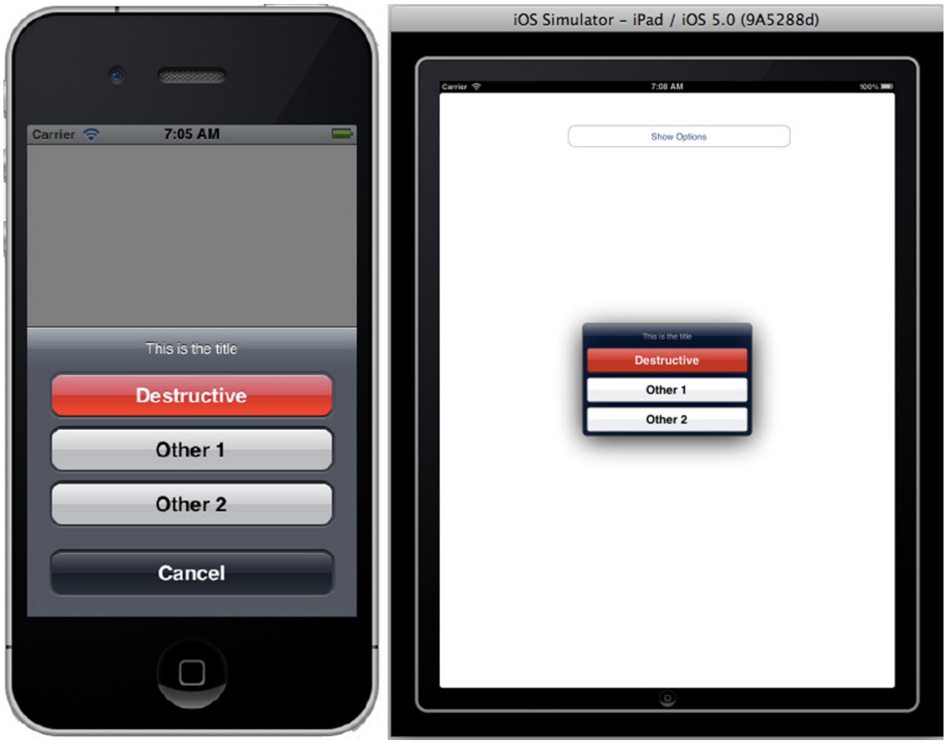


FIGURE 7-4

On an iPhone they slide up from the bottom of the screen, and on the iPad they display as popover windows. On an iPad, the cancel button is not visible. If the user taps outside the action sheet on an iPad, the action sheet is dismissed.

Action sheets enable you to highlight one of the buttons in red—this button is referred to as the *destructive* button. To create an action sheet you can use the following code:

```
UIActionSheet* message = [[UIActionSheet alloc]
                           initWithTitle:@"This is the title"
                           delegate:self
                           cancelButtonTitle:@"Cancel"
                           destructiveButtonTitle:@"Destructive"
                           otherButtonTitles:@"Other 1", @"Other 2", nil];
```

As you can see, the parameters are very similar to those of an alert view. The `destructiveButtonTitle` parameter is optional, and when specified contains the title of the destructive button. To create an action sheet without a destructive button, set this parameter to `nil`.

To show an action sheet, send it the `showInView:` message as follows:

```
[message showInView:self.view];
```

You cannot display an action sheet in the `viewDidLoad:` method of a view controller class on the iPad. Another important distinction between action sheets and alert views is that the former cannot have text fields in them.

To determine which button was pressed in an action sheet, you need to provide a delegate object that conforms to the `UIActionSheetDelegate` protocol. In most cases this delegate object is the view controller class itself. In the delegate object, you must implement `actionSheet:clickedButtonAtIndex:` as follows:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == actionSheet.cancelButtonIndex)
    {
        // cancel button pressed
    }
    else if (buttonIndex == actionSheet.destructiveButtonIndex)
    {
        // destructive button pressed
    }
    else if (buttonIndex == actionSheet.firstOtherButtonIndex)
    {
        // option 1 pressed
    }
    else if (buttonIndex == actionSheet.firstOtherButtonIndex + 1)
    {
        // option 2 pressed
    }
}
```

TRY IT

In this Try It, you create a new Xcode project based on the Single View Application template called `AlertSample` that presents an alert view prompting the user to supply a username. When the alert view is dismissed, a label on the screen is updated with the name entered in the alert view.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template.
- Edit the storyboard with Interface Builder.
- Add a `UILabel` instance to the default scene.
- Have the view controller class conform to the `UIAlertViewDelegate` protocol.
- Create a `UIAlertView` instance in the `viewDidLoad` method of the view controller class and present it to the user.
- Implement the `alertView:clickedButtonAtIndex:` delegate method in the view controller class.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 7 folder in the download.*

Hints

- Launch Xcode from the `/Developer/Applications` folder.
- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.

Step-by-Step

1. Create a Single View Application in Xcode called `AlertSample`.
    1. Launch Xcode from the `/Developer/Applications` folder.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** AlertSample
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson7
        - **Define Family:** iPhone

- **Use Storyboard:** Checked
- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`, *but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Open the `MainStoryboard.storyboard` file in the Interface Builder.
    1. Ensure the project navigator is visible and the `AlertSample` project is selected and expanded.
    2. Click the `MainStoryboard.storyboard` file.
    3. Ensure the utilities editor is visible. To show the utilities editor, use the View Utilities ➪ Show Utilities menu item.
3. Add a `UILabel` instance to the scene.
    1. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
    2. From the Object library, drag and drop a Label object onto the scene.
    3. Use the Attributes inspector to set the `text` attribute of the Label to `User name:`. To show the Attributes inspector, use the View ➪ Utilities ➪ Show Attributes inspector menu item.
    4. Size and position the label to X=68, Y=23, W=184, H=21.
    5. Use the assistant editor to create an outlet in the view controller class called `userLabel` and connect the label to it.
4. Have the view controller conform to the `UIAlertViewDelegate` protocol.
    1. Select the `Lesson7ViewController.h` file in the Project Explorer.
    2. Modify the interface of the view controller class to resemble the following:

```
@interface Lesson7ViewController : UIViewController <UIAlertViewDelegate>
@property (weak, nonatomic) IBOutlet UILabel *userLabel;
@end
```

5. Create a `UIAlertView` instance and display it.

    1. Select the `Lesson7ViewController.m` file in the Project Explorer.

    2. Replace the implementation of the `viewDidLoad` method with the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UIAlertView* message = [[UIAlertView alloc]
                            initWithTitle:@"What is your name?"
                            message:nil
                            delegate:self
                            cancelButtonTitle:@"Ok"
                            otherButtonTitles:nil];
    message.alertViewStyle = UIAlertViewStylePlainTextInput;
    [message show];
}
```

6. Implement the `alertView:clickedButtonAtIndex:` delegate method in the view controller class. Paste the following implementation into the `Lesson7ViewController.m` file:

```
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
buttonIndex
{
    UITextField* field1 = [alertView textFieldAtIndex:0];
    userLabel.text = [NSString stringWithFormat:@"User name:%@", field1.text];
}
```

7. Test your app in the iOS Simulator. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project Run menu item.

Please select Lesson 7 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

8 Adding Images to Your View

The UIKit framework provides classes that enable you to represent and display images. In this lesson, you learn how to use the `UIImage` and `UIImageView` classes.

THE UIIMAGE CLASS

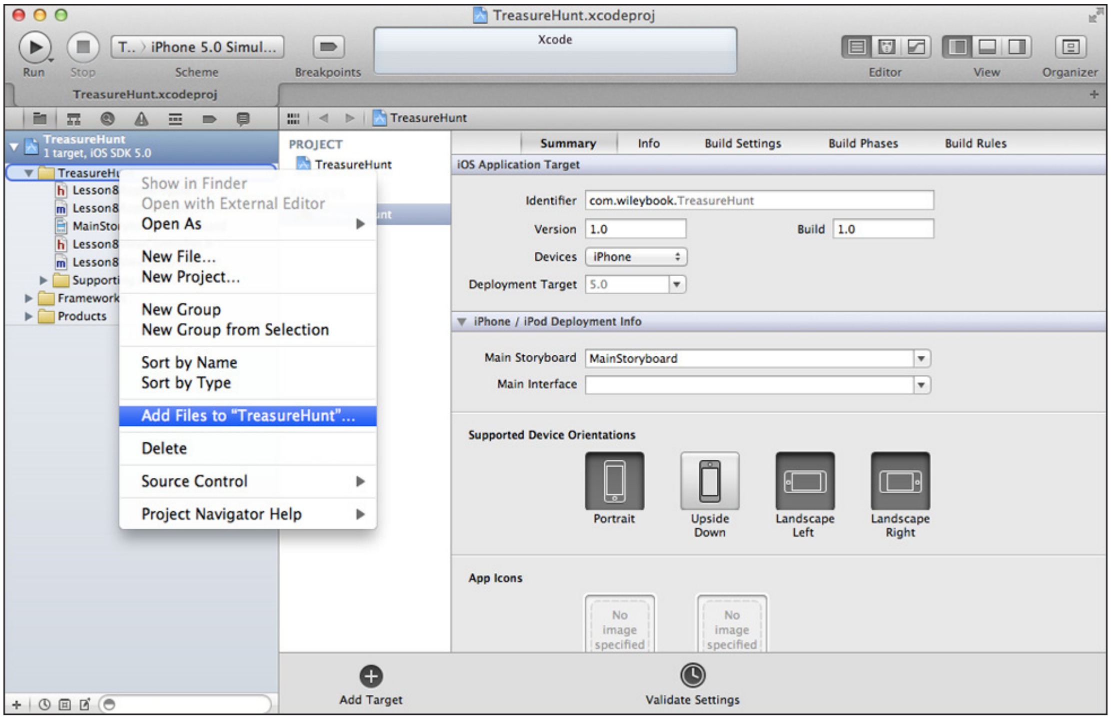
A `UIImage` object represents image data that has either been read from a file or created using Quartz primitives. Instances are immutable, thus their properties can't be changed once they have been created. `UIImage` instances do not provide access to the underlying image data, but do enable you to retrieve a PNG or JPEG image representation in an `NSData` object.

Images generally require large amounts of memory to store, and you should avoid creating image objects larger than 1024 × 1024 pixels. To load an image from a file into a `UIImage` object, you first need to ensure the file is in one of the formats listed in Table 8-1.

TABLE 8-1: UIImage Supported File Formats

DESCRIPTION	FILE EXTENSIONS
Portable Network Graphics	`.png`
Joint Photographic Experts Group	`.jpeg, .jpg`
Graphics Interchange Format	`.gif`
Windows Device Independent Bitmap	`.bmp`
Tagged Image File Format	`.tif, .tiff`

You also need to ensure that the file is part of the project. If the file is not visible in the Project Explorer, you need to add it by right-clicking an existing group and selecting the Add Files to Project option in the context menu (Figure 8-1).



Identifier: com.wileybook.TreasureHunt
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0
Main Storyboard: MainStoryboard
Main Interface:

FIGURE 8-1

Assuming you have an image file called `cat.png`, and want to load it into a `UIImage` object, use the following code:

```
UIImage* catImage = [UIImage imageNamed:@"cat.png"];
```

The `imageNamed` method is a class method of the `UIImage` class, and implements an internal system cache. Thus if you were to use this method to repeatedly load the same image file, the image data would be loaded only once and shared between the `UIImage` instances. If this code is executed on a device that has a retina display, the `imageNamed:` method first searches for a file with an `@2x` suffix appended to it. Thus, on an iPhone 4, this code would first look for a file named `cat@2x.png`. If it could not find that, it would look for the file `cat.png`.

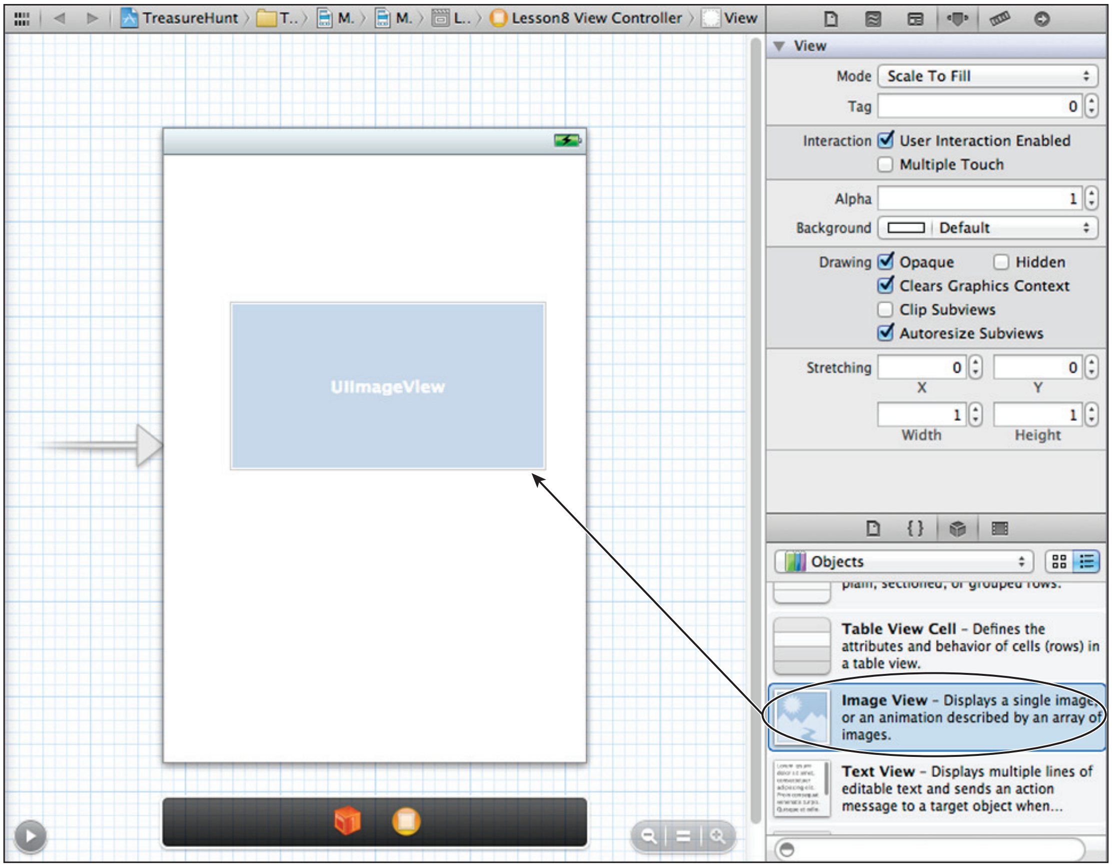
Loading images from your application bundle is not the only way to use `UIImage` objects. You can also create one from an online data source by downloading the data available at the URL into an `NSData` object and then instantiating a `UIImage` using the `imageWithData:` class method.

The following code snippet shows how to do this synchronously; however in production code you should try and download any data from the web, including images, asynchronously. Downloading images asynchronously is an advanced topic and is not covered in this book.

```
NSURL * imageURL = [ NSURL URLWithString : @"http://......" ];
NSData * imageData = [ NSData dataWithContentsOfURL :imageURL];
UIImage * image = [[ UIImage alloc ] initWithData :imageData];
```

THE UIIMAGEVIEW CLASS

A `UIImageView` object provides a container for displaying either a single `UIImage` object, or an animated series of `UIImage` objects. To add a `UIImageView` object to a view controller or storyboard scene, simply drag an Image View object from the Object library (Figure 8-2).



View
Mode: Scale To Fill
Tag: 0
Interaction: ✓ User Interaction Enabled; Multiple Touch
Alpha: 1
Background: Default
Drawing: ✓ Opaque; Hidden; ✓ Clears Graphics Context; Clip Subviews; ✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1

Table View Cell – Defines the attributes and behavior of cells (rows) in a table view.

Image View – Displays a single image, or an animation described by an array of images.

Text View – Displays multiple lines of editable text and sends an action message to a target object when...

FIGURE 8-2

To set up the default image displayed in the image view, simply select an image from the project's resources for the Image property in the Attributes inspector (Figure 8-3).

To display a `UIImage` object in an image view, you need to create an outlet for the image view in the view controller class and set up its `image` property as follows:

```
imageView.image = [UIImage imageNamed:@"cat.png"];
```

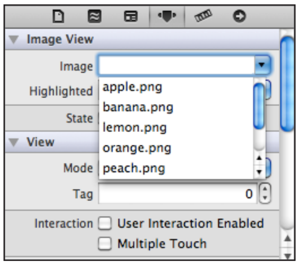


FIGURE 8-3

To use a `UIImageView` object to perform simple frame animation, simply provide an array of `UIImage` objects in its `animationImages` property as follows:

```
NSArray* frameArray = [[NSArray alloc] initWithObjects:
                           [UIImage imageNamed:@"frame1.png"],
                           [UIImage imageNamed:@"frame2.png"],
                           [UIImage imageNamed:@"frame3.png"],
                           nil];

imageView.animationImages=frameArray;
```

and send the `startAnimating` message to the image view:

```
[imageView startAnimating];
```

Specify the duration of the animation in seconds, using the `animationDuration` property:

```
imageView.animationDuration = 2;
```

TRY IT

In this Try It, you create a new Xcode project based on the Single View Application template called `TreasureHunt` that displays an image and asks the user to find an object in the image. When the user taps the object, a short congratulatory animation sequence is displayed.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template.
- Edit the storyboard with Interface Builder.
- Import image resources into the project.
- Add a `UILabel` instance to the default scene.
- Add two `UIImageView` instances to the default scene.
- Use a gesture recognizer to detect a tap on the image and display an alert view.
- If the tap occurs over a specific region of the image, display a congratulatory frame animation.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 8 folder in the download.*

Hints

- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.

Step-by-Step

1. Create a Single View Application in Xcode called `TreasureHunt`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** TreasureHunt
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson8
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    > *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Import image resources into your project.
    1. Ensure the project navigator is visible and the TreasureHunt project is selected and expanded. To show the project navigator, use the View ➪ Navigators ➪ Show Project Navigator menu item. To expand a project click the triangle next to the project name in the project navigator.
    2. Right-click the project node and select Add Files to TreasureHunt from the context menu.
    3. Select the `Images` folder in this lesson's resources on the DVD.
    4. Ensure the Copy Items to Destination Group's Folder (if needed) option is selected in the dialog box.
    5. Click the Add button.
3. Add a `UILabel` instance to the default scene.
    1. Open the `MainStoryboard.storyboard` file in Interface Builder.
    2. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.

   3. From the Object library, drag and drop a Label object onto the scene.
   4. Use the Attributes inspector to set the Text attribute of the label to `Tap the blue bead!` To show the Attributes inspector, use the View ➪ Utilities ➪ Show Attributes Inspector menu item.
   5. Size and position the label to X=102, Y=6, W=117, H=21.
4. Add two `UIImageView` instances to the default scene.
   1. From the Object library, drag and drop an Image View object onto the scene.
   2. Use the Attributes inspector to set the Image attribute of the image view to `beads.png`. To show the Attributes inspector, use the View ➪ Utilities ➪ Show Attributes Inspector menu item.
   3. Size and position the image view to X=0, Y=30, W=320, H=430.
   4. Use the assistant editor to create an outlet in the view controller class called `largeImage` and connect the image view to it.
   5. From the Object library, drag and drop a second Image View instance to the scene.
   6. Size and position the image view to X=0, Y=190, W=320, H=100.
   7. Use the assistant editor to create an outlet in the view controller class called `animatedImage` and connect the image view to it.
5. Add a tap gesture recognizer and use it to show an animated image sequence when the blue bead is tapped. Gesture recognizers are covered in detail in Lesson 29.
   1. Add the following method declaration to the `Lesson8ViewController.h` file:

      ```
      - (void) handleTap:(UITapGestureRecognizer*)sender;
      ```

   2. Update the `viewDidLoad` method of the view controller class to resemble the following:

      ```
      - (void)viewDidLoad
      {
          [super viewDidLoad];
          // install tap gesture recognizer.
          UITapGestureRecognizer* tapRecognizer = [[UITapGestureRecognizer
      alloc]
                                                   initWithTarget:self
                                                   action:@
      selector(handleTap:)];
          tapRecognizer.cancelsTouchesInView = NO;
          [self.view addGestureRecognizer:tapRecognizer];

          // setup animatedImage
          NSArray* frameArray = [[NSArray alloc] initWithObjects:
                                 [UIImage imageNamed:@"anim1.png"],
                                 [UIImage imageNamed:@"anim2.png"],
                                 [UIImage imageNamed:@"anim3.png"],
                                 [UIImage imageNamed:@"anim4.png"],
                                 [UIImage imageNamed:@"anim5.png"],
                                 [UIImage imageNamed:@"anim6.png"],
      ```

```
                                     nil];

        animatedImage.animationImages=frameArray;
        animatedImage.animationDuration = 0.5;
        animatedImage.animationRepeatCount = 1;
        animatedImage.userInteractionEnabled = NO;
        [animatedImage setHidden:YES];
    }
```

3. Implement the `handleTap:` method in the `Lesson8ViewController.m` file as follows:

```
- (void) handleTap:(UITapGestureRecognizer*)sender
{
    CGPoint startLocation = [sender locationInView:self.view];
    if ((startLocation.y >= 211) && (startLocation.y <= (211 + 104)))
    {
        [animatedImage setHidden:NO];
        [animatedImage startAnimating];
    }
}
```

6. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

Please select Lesson 8 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo, *to view the video that accompanies this lesson.*

9 Pickers

A picker view (Figure 9-1) is a user interface component that enables a user to pick a value from a set of related values using a slot machine–style interface.

Each wheel of the picker view is called a component, and it is fairly common to have picker view with multiple components. Each component can have a different number of items in it (Figure 9-2).

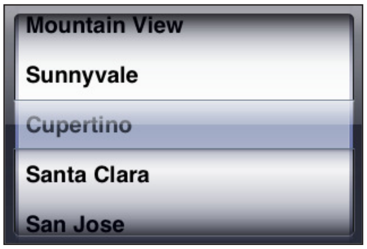


FIGURE 9-1

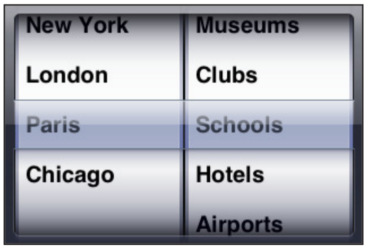


FIGURE 9-2

A picker view is encapsulated by the `UIPickerView` class, which is part of the UIKit framework. Apple provides a special picker for allowing the user to select date and time. This component is called the date picker and is covered in the next lesson.

A picker requires a data source object and a delegate object. The data source object is one that implements the `UIPickerViewDataSource` protocol and provides information on the number of components, and rows-per-component of the picker.

The delegate object implements the `UIPickerViewDelegate` protocol and has methods that are called when the current selection in a component has changed.

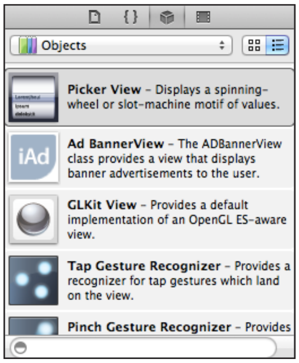
The delegate and data source objects could both be the same object, and in many cases the duties of these objects are performed by the view controller. However, it is very possible for them to be independent objects.

Creating a picker view is a simple matter of dragging the Picker View component from the Object library onto your storyboard or XIB file (Figure 9-3) and then creating an appropriate outlet in your view controller class using the assistant editor.

The delegate and data source objects can be set up using Interface Builder (Figure 9-4) or by setting up the `delegate` and `dataSource` properties in code.

The following code snippet assumes *pickerView* is an outlet that is connected to a `UIPickerView` instance and sets up the view controller to be the delegate and the data source object:

```
- (void)viewDidLoad{
    [super viewDidLoad];
    pickerView.delegate = self;
    pickerView.dataSource = self;
}
```



Picker View – Displays a spinning-wheel or slot-machine motif of values.

Ad BannerView – The ADBannerView class provides a view that displays banner advertisements to the user.

GLKit View – Provides a default implementation of an OpenGL ES-aware view.

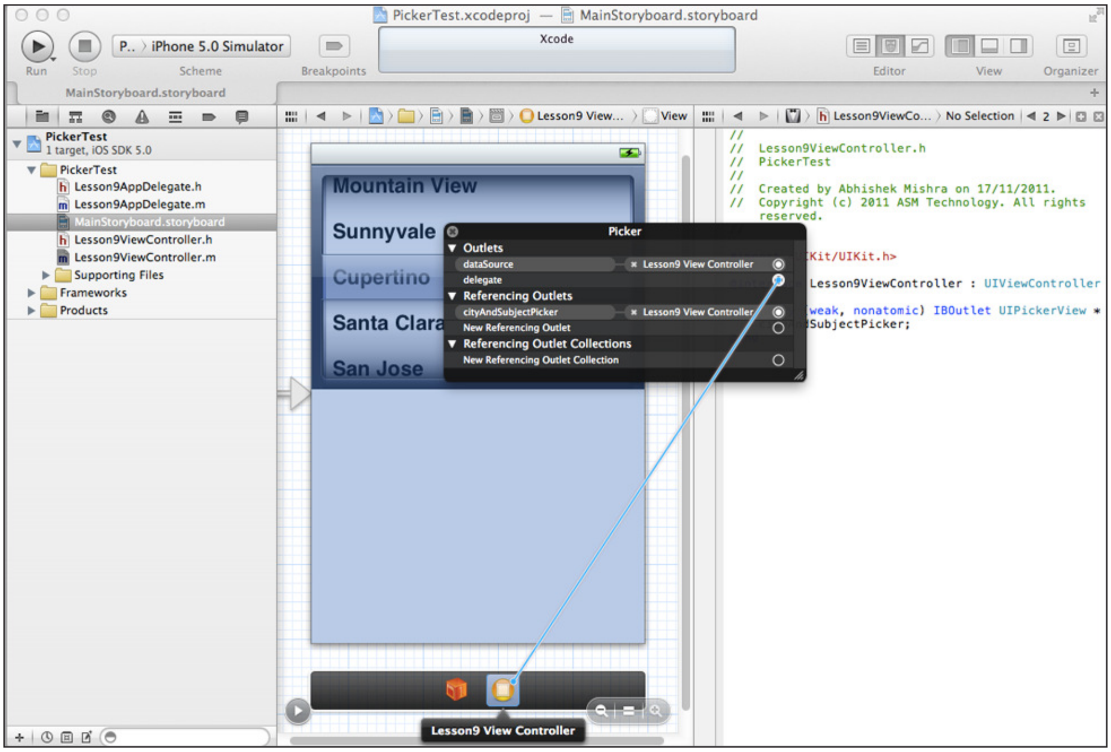
Tap Gesture Recognizer – Provides a recognizer for tap gestures which land on the view.

Pinch Gesture Recognizer – Provides

FIGURE 9-3

The `UIPickerViewDataSource` protocol defines two methods:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView;
- (NSInteger)pickerView:(UIPickerView *)pickerView
              numberOfRowsInComponent:(NSInteger)component;
```



Picker

Outlets
- dataSource — Lesson9 View Controller
- delegate

Referencing Outlets
- cityAndSubjectPicker — Lesson9 View Controller
- New Referencing Outlet

Referencing Outlet Collections
- New Referencing Outlet Collection

```
//
//  Lesson9ViewController.h
//  PickerTest
//
//  Created by Abhishek Mishra on 17/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights
reserved.

Kit/UIKit.h>

Lesson9ViewController : UIViewController

weak, nonatomic) IBOutlet UIPickerView *
SubjectPicker;
```

FIGURE 9-4

You must return the number of components in the picker view from the `numberOfComponentsInPickerView:` method. The number of rows in each component should be returned by the `pickerView:numberOfRowsInComponent:` method. For example, a two-component picker can be set up as follows:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView{
    return 2;
}
- (NSInteger)pickerView:(UIPickerView *)pickerView
              numberOfRowsInComponent:(NSInteger)component
{
    if (component == 0)
        return [cities count];

    return [placesOfInterest count];
}
```

The most commonly used `UIPickerViewDelegate` methods are:

```
- (NSString *)pickerView:(UIPickerView *)pickerView
              titleForRow:(NSInteger)row
              forComponent:(NSInteger)component;
- (void)pickerView:(UIPickerView *)pickerView
        didSelectRow:(NSInteger)row
        inComponent:(NSInteger)component;
```

The text to be displayed in each row of each component is to be returned by the `pickerView:titleForRow:forComponent:` delegate method. When the user selects a row in any component of the picker, your delegate object's `pickerView:didSelectRow:inComponent:` method will be called.

Typically, the data for each component of a picker view is stored in an array. Assuming that `cities` and `placesOfInterest` are arrays of `NSString` objects that contain the data for the two components of a picker view, the `pickerView:titleForRow:forComponent:` delegate method can be implemented as follows:

```
- (NSString *)pickerView:(UIPickerView *)pickerView
               titleForRow:(NSInteger)row
               forComponent:(NSInteger)component
{
    if (component == 0)
        return [cities objectAtIndex:row];

    return [placesOfInterest objectAtIndex:row];
}
```

ARRAYS IN OBJECTIVE-C

An array is an ordered collection of similar objects, and each object in the array has an index. The index of the first object is zero. Objective-C has two classes to represent arrays:

- `NSArray`
- `NSMutableArray`

`NSArray` instances are immutable. This means that you cannot change the contents of an `NSArray` object after you have created it. In fact, the contents of an array are set up as part of the initialization process.

`NSMutableArray` instances, on the other hand, have no such restriction. However, you must keep in mind that inserting/deleting objects from an array can be a time-consuming operation, and thus you should aim to use `NSArray` objects wherever possible.

To create an `NSArray` instance, and add four `NSString` objects to it in the same step, you can use code similar to the following:

```
cities = [[NSArray alloc]
          initWithObjects:@"New York", @"London", @"Paris", @"Chicago", nil];
```

When using the `initWithObjects:` method, the last object must always be nil.

To retrieve an object at a specific index position, you can use the `objectAtIndex:` method. Index numbers start from zero.

```
NSString* someCity = [cities ObjectAtIndex:0];
```

To retrieve the number of objects in an array, you can use the `count` method:

```
int arrayCount = [cities count];
```

`NSMutatbleArray` has all the methods provided by `NSArray` and a few more. You can create an `NSMutableArray` with a specified initial capacity as follows:

```
NSMutableArray* dynamicArray = [[NSMutableArray alloc] initWithCapacity:10];
```

The value you provide to the `initWithCapacity:` method serves merely as a hint to `NSMutableArray` to pre-allocate memory for the specified number of objects. You can have fewer or more objects in the array as required by your program.

To add an element to the back of an `NSMutableArray` instance, you can use the `addObject:` method:

```
[dynamicArray addObject:@"This in inserted at the end of the array"];
```

To remove a specific object from an `NSMutableArray`, you can use the `removeObject:` method. To remove all objects, you can use the `removeAllObjects` method.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `PickerTest` that displays a two-component picker and a button. When the button is tapped, an alert view appears, displaying the selected item in each component.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a Picker View to the default scene and create an outlet in the view controller class.

- Add a Round Rect Button to the default scene and add an action method in the view controller class that is called when the Touch Up Inside event is fired.
- Add two data arrays in the view controller class and populate them in the `viewDidLoad` method.
- Implement the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols in your view controller class.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 9 folder in the download.*

Hints

- Use an `NSArray` object to create a data array whose contents will not change.

Step-by-Step

1. Create a Single View Application in Xcode called `PickerTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** PickerTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson9
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.

2. Add a Picker View to your storyboard's default scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add a Picker View to the default scene.
    3. Use the Size inspector to resize and position it at X = 0, Y = 0.
    4. Using the assistant editor, create an outlet for the picker view called `cityAndSubjectPicker`.
    5. Set the view controller class to be the delegate and data source object for the picker view.
        1. Right-click the picker view to display a context menu.
        2. Drag from the circle beside the `delegate` entry in the context menu, to the view controller icon (yellow box) in the dock.
        3. Do the same for the `dataSource` entry in the context menu.
3. Add a Round Rect Button to the default scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add a Round Rect Button instance.
    3. Double-click the button and set its title to `Show Values`.
    4. Using the Size Inspector, resize and position it to X = 20, Y = 251, W = 280, H = 37.
    5. Using the assistant editor, create an action in the view controller class and connect it to the Touch Up Inside event of the button. Call the new method `onButtonPressed`.
4. Add two `strong` and `nonatomic` `NSArray` properties called `cities` and `placesOfInterest` to the Lesson9ViewController class.
    1. Declare the properties in the interface file:

    ```
    @property (strong, nonatomic) NSArray* cities;
    @property (strong, nonatomic) NSArray* placesOfInterest;
    ```

    2. Synthesize them in the implementation file:

    ```
    @synthesize cities;
    @synthesize placesOfInterest;
    ```

5. Instantiate and initialize the `NSArray` objects in the `viewDidLoad` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    cities = [[NSArray alloc]
              initWithObjects:@"New York", @"London", @"Paris",
                              @"Chicago", nil];

    placesOfInterest = [[NSArray alloc]
```

```
                        initWithObjects:@"Museums", @"Clubs", @"Schools",
                        @"Hotels", @"Airports", nil];
}
```

6. Have your view controller class conform to the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols by modifying its interface to the following:

```
@interface Lesson9ViewController : UIViewController<UIPickerViewDataSource,
                                                     UIPickerViewDelegate>
```

7. Implement the `numberOfComponentsInPickerView:` data source method in your view controller as follows:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 2;
}
```

8. Implement the `pickerView:numberOfRowsInComponent:` data source method in your view controller as follows:

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
             numberOfRowsInComponent:(NSInteger)component
{
    if (component == 0)
        return [cities count];

    return [placesOfInterest count];
}
```

9. Implement the `pickerView:titleForRow:forComponent:` delegate method in your view controller as follows:

```
- (NSString *)pickerView:(UIPickerView *)pickerView
              titleForRow:(NSInteger)row
              forComponent:(NSInteger)component
{
    if (component == 0)
        return [cities objectAtIndex:row];
    return [placesOfInterest objectAtIndex:row];
}
```

10. Add the following code to the implementation of the `onButtonPressed:` method in your view controller class:

```
int cityIndex = [cityAndSubjectPicker selectedRowInComponent:0];
int placeIndex = [cityAndSubjectPicker selectedRowInComponent:1];
NSString* messsageText = [[NSString alloc]
                          initWithFormat:@"Are you looking for %@ in %@?",
                          [placesOfInterest objectAtIndex:placeIndex],
                          [cities objectAtIndex:cityIndex]];

UIAlertView* message = [[UIAlertView alloc]
                        initWithTitle:@""
                        message:messsageText
                        delegate:nil
                        cancelButtonTitle:@"Yes"
                        otherButtonTitles:nil];
[message show];
```

11. Test your application in the iOS Simulator.
    1. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.
    2. Change the selection in the components of the picker and tap the Show Values button (Figure 9-5).

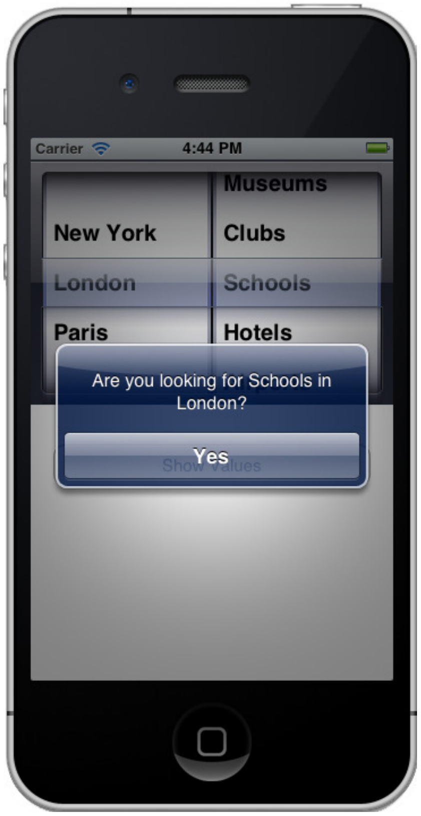


FIGURE 9-5

Please select Lesson 9 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo, *to view the video that accompanies this lesson.*

10 Date Pickers

In the previous lesson you learned about picker views. Although it is possible to create a picker view with several components to allow your user to enter a date, Apple provides a special user interface component for precisely this purpose. The date picker is a special picker that can be used to select dates and times. You can configure it to display only date, only time, or both date and time as shown in Figure 10-1.

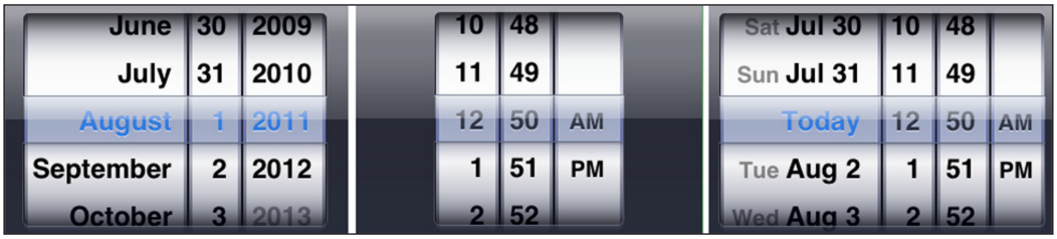
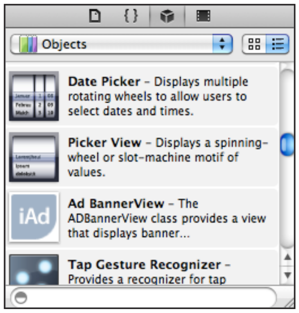


FIGURE 10-1

The `UIDatePicker` class provides the functionality of a date picker, which is part of the UIKit framework. The `UIDatePicker` class privately uses a `UIPickerView` instance, but you cannot access this instance directly.

A date picker is much simpler to use than a picker view. For starters, it does not require you to provide a delegate or data source object. Creating a date picker is a simple matter of dragging the Date Picker component from the Object library onto a scene in your storyboard (Figure 10-2) and then creating an appropriate outlet in your view controller class using the assistant editor.

The *mode* of the date picker refers to whether it displays date, time, or both date and time. You can also specify the range of values that should be displayed by the date picker. Both these tasks can be accomplished by using the assistant editor (Figure 10-3).



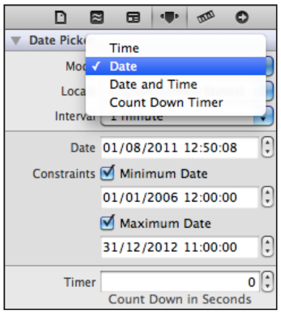
Date Picker – Displays multiple rotating wheels to allow users to select dates and times.

Picker View – Displays a spinning-wheel or slot-machine motif of values.

Ad BannerView – The ADBannerView class provides a view that displays banner...

Tap Gesture Recognizer – Provides a recognizer for tap

FIGURE 10-2



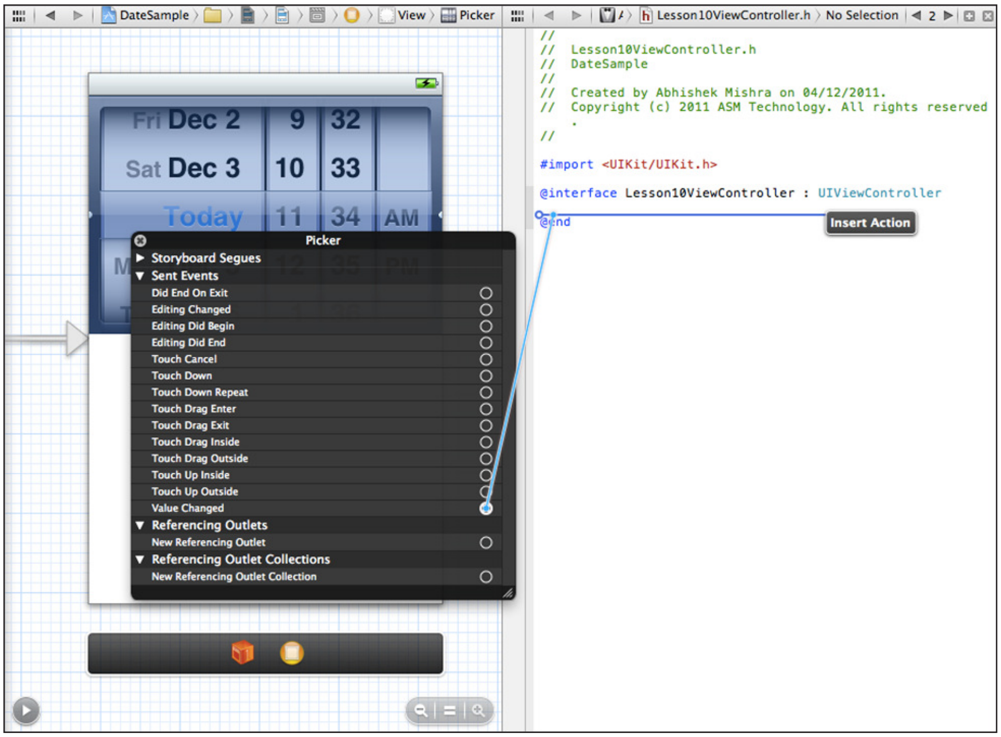
Mode: Date
Interval: 1 minute
Date 01/08/2011 12:50:08
Constraints ✓ Minimum Date
01/01/2006 12:00:00
✓ Maximum Date
31/12/2012 11:00:00
Timer 0
Count Down in Seconds

FIGURE 10-3

You can read the date currently selected in the picker by accessing the date picker's `date` property. The result is returned as an `NSDate` instance:

```
// get date from date picker
NSDate* pickerDate = datePicker.date;
```

The date picker provides a Value Changed event that is fired when the user changes the selection in the picker. You can use the assistant editor to create and associate an action method in your view controller class with this event (Figure 10-4).



```
//
//  Lesson10ViewController.h
//  DateSample
//
//  Created by Abhishek Mishra on 04/12/2011.
//  Copyright (c) 2011 ASM Technology. All rights reserved
.
//

#import <UIKit/UIKit.h>

@interface Lesson10ViewController : UIViewController

@end
```

FIGURE 10-4

DATES IN OBJECTIVE-C

Objective-C provides an `NSDate` class, instances of which represent a combined date and time value. To create an `NSDate` object that has the current date and time, use the following code:

```
NSDate* todaysDate = [[NSDate alloc] init];
```

To create an `NSDate` object dated at a specific interval of time from the current date, you can use the `initWithTimeIntervalSinceNow:` method. This method requires a single argument, which is the number of seconds in the past or future from the current date. A positive number indicates a future date.

Thus, to create an `NSDate` object exactly 24 hours from the current date, you can use the following code:

```
NSDate* tomorrowsDate = [[NSDate alloc]
                         initWithTimeIntervalSinceNow: 24 * 3600];
```

If you want to create an `NSDate` without reference to the current date, you can use the `initWithTimeIntervalSinceReferenceDate:` method to create a date that is at a specified interval from the 1st of January, 1970. The interval is specified in seconds.

`NSDate` instances also provide several useful methods to compare dates, including:

- `isEqualToDate`: Returns YES if two `NSDate` instances are equal.
- `earlierDate`: Returns the earlier of two `NSDate` objects.
- `laterDate`: Returns the later of two `NSDate` objects.

Examples that contain these methods are shown here:

```
BOOL comparisonResult = [todaysDate isEqualToDate:someOtherDate];
NSDate* firstDate = [todaysDate earlierDate:someOtherDate];
```

For information on NSDate objects, refer to the NSDate Class Reference, available at:

```
http://developer.apple.com/library/ios/#documentation/Cocoa/
Reference/Foundation/Classes/NSDate_Class/Reference/Reference
.html#//apple_ref/doc/uid/TP40003641
```

Creating a formatted representation of the contents of an `NSDate` object requires the use of another class: `NSDateFormatter`.

To use an `NSDateFormatter`, you need to first instantiate it, and use the `setDateFormat` method on the instance to specify the internal format used by the date formatter object. This internal format is specified as a string. Once a date formatter is instantiated, you can use it to create a textual representation of an `NSDate` object using the `stringFromDate` method. This is demonstrated in the following code:

```
NSDateFormatter* dateFormat = [[NSDateFormatter alloc] init];
[dateFormat setDateFormat:@"MMMM d, yyyy"];
NSString* textutalRepresentation = [dateFormat stringFromDate:todaysDate];
```

The format string consists of a series of characters that represent parts of a date and time. The characters themselves are case-sensitive, some of the most common format strings are:

- `MMMM`: The full name of the month
- `d`: The day of the month
- `YYYY`: The four-digit year
- `hh`: Two-digit hour of the day
- `mm`: Two-digit minute
- `ss`: Two-digit second
- `a`: AM
- `p`: PM

For a complete list of format strings, refer to the Date Formatting Guide, available at:

```
http://developer.apple.com/library/ios/#documentation/Cocoa/
Conceptual/DataFormatting/DataFormatting.html#//apple_ref/doc/
uid/10000029i
```

You can also use an `NSDateFormatter` instance to create an `NSDate` instance from a string representation of a date. This is done using the `dateFromString:` method of the date formatter object. If the method succeeds, the result is a valid `NSDate` object, otherwise it is `nil`. The following code snippet converts the string December 5, 2011 into an `NSDate` instance:

```
NSDateFormatter* dateFormat = [[NSDateFormatter alloc] init];
[dateFormat setDateFormat:@"MMMM d, yyyy"];
NSDate* equivalentDate = [dateFormat dateFromString:@"December 5, 2011"];
```

For more information on the NSDateFormatter class, refer to the NSDateFormatter Class Reference, available at:

```
http://developer.apple.com/library/ios/#documentation/Cocoa/
Reference/Foundation/Classes/NSDateFormatter_Class/Reference/
Reference.html#//apple_ref/doc/uid/TP40003643
```

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `DateSample` that displays a date picker and a label. The contents of the label are updated to display a formatted version of the selected date, and the number of days between the selected date and today's date.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a date picker to the default scene and create an outlet for it in the view controller class.
- Use the Attributes inspector to set the mode and range of the date picker.
- Add two `UILabel` instances and create outlets for them in the view controller class.
- Create an action method in the view controller class and connect it to the Value Changed event of the date picker.
- Write code to display the selected date in a specific format and compute the number of days between the selected date and today's date.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 10 folder in the download.*

Hints

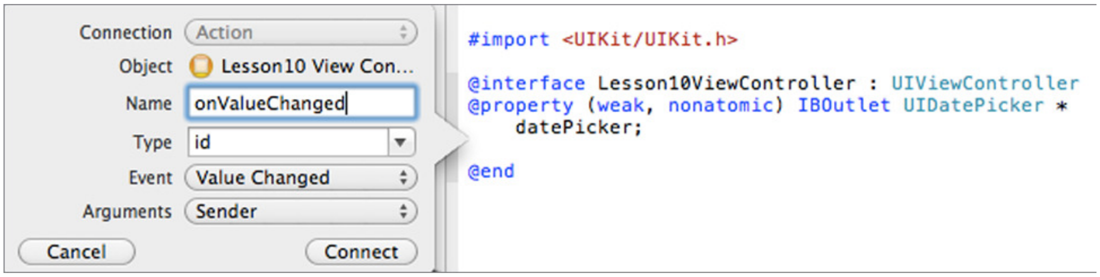
- You cannot change the height/width of a date picker.
- You can set the mode and the minimum and maximum date range using the Attributes inspector.

Step-by-Step

1. Create a Single View Application in Xcode called `DateSample`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** DateSample
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson10
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a date picker to the default scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add a date picker to the default scene.
    3. Use the Size inspector to resize and position it at X = 0, Y = 0.
    4. Using the assistant editor, create an outlet for the date picker in the view controller class called `datePicker`.
    5. Using the assistant editor, create an action method called `onValueChanged` in the view controller class and associate it with the Value Changed event of the date picker.
        1. Right-click the date picker to display a context menu.
        2. Drag from the circle beside the Value Changed event to the view controller file in the adjacent window.
        3. Name the new action `onValueChanged` (Figure 10-5).
        4. Using the Attributes inspector, set the mode of the date picker to Date.
        5. Using the Attributes inspector, set the range of dates displayed in the date picker as shown in Table 10-1.



Connection	Action
Object	Lesson10 View Con...
Name	onValueChanged
Type	id
Event	Value Changed
Arguments	Sender

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

@interface Lesson10ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIDatePicker *
    datePicker;

@end
```

FIGURE 10-5

TABLE 10-1: Custom Date Range

ITEM	VALUE
Minimum	08/01/2011
Maximum	12/31/2012

3. Add two `UILabel` instances to the scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add two `UILabel` instances.
    3. Using the Size inspector, resize and position the first label to X = 20, Y = 257, W = 280, H = 21.
    4. Using the Size inspector, resize and position the second label to X = 20, Y = 293, W = 280, H = 21.
    5. Using the Attributes inspector, set the Alignment property of both labels to be center aligned.
    6. Add outlets to the view controller class for each label. Name the outlet corresponding to the label on top as `topLabel`. Name the other one `bottomLabel`.
4. Your view controller's interface should now resemble the following:

```
#import <UIKit/UIKit.h>
@interface Lesson10ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIDatePicker *datePicker;
@property (weak, nonatomic) IBOutlet UILabel *topLabel;
@property (weak, nonatomic) IBOutlet UILabel *bottomLabel;
- (IBAction)onValueChanged:(id)sender;
@end
```

5. Modify your view of controller's `viewDidLoad` method to resemble the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    bottomLabel.text = @"";
    topLabel.text = @"Please select a date!";
}
```

6. Add the following code to the implementation of the `onValueChanged:` method in your view controller class:

```
// get todays date
NSDate* todaysDate = [[NSDate alloc] init];

// get date from date picker
NSDate* pickerDate = datePicker.date;

// difference between intervals (in days)
NSTimeInterval dateDifference = [pickerDate
                                timeIntervalSinceDate:todaysDate];
double numDays = dateDifference / (3600 * 24);
bottomLabel.text = [NSString stringWithFormat:@"Difference from
                   today (in days) = %2.0f", numDays];

// display the selected day as a string.
NSDateFormatter* dateFormat = [[NSDateFormatter alloc] init];
```

```
[dateFormat setDateFormat:@"MMMM d, yyyy"];
topLabel.text = [NSString stringWithFormat:@"Selected date:%@",
                            [dateFormat stringFromDate:pickerDate]];
```

7. Test your application in the iOS Simulator.
    1. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.
    2. Change the selection in the components of the date picker and notice how the text in each label changes.

> *Please select Lesson 10 on the DVD that accompanies the print book, or go to* `www.wrox.com/go/iphoneipadappvideo`, *to view the video that accompanies this lesson.*

11 Custom Pickers

In Lesson 9 you learned about the `UIPickerView` class. Picker views do not have to be restricted to displaying text; in fact, they can just as easily display images, or a combination of images and text. In this lesson you learn how to provide your own `UIView` subclasses for individual elements of a picker view, thus creating pickers that have images instead of text, as shown in Figure 11-1.

The key to implementing this functionality lies in three optional methods of the `UIPickerViewDelegate` protocol:

```
- (CGFloat)pickerView:(UIPickerView *)pickerView
          widthForComponent:(NSInteger)component;
- (CGFloat)pickerView:(UIPickerView *)pickerView
          rowHeightForComponent:(NSInteger)component;
- (UIView *)pickerView:(UIPickerView *)pickerView
           viewForRow:(NSInteger)row
           forComponent:(NSInteger)component
           reusingView:(UIView *)view;
```

You can customize the width of each picker component by returning an appropriate value from the `pickerView:withForComponent:` delegate method (Figure 11-2).

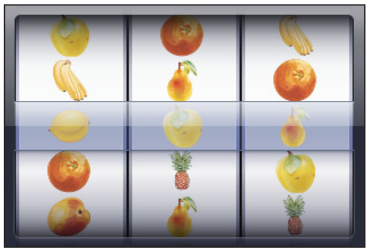


FIGURE 11-1

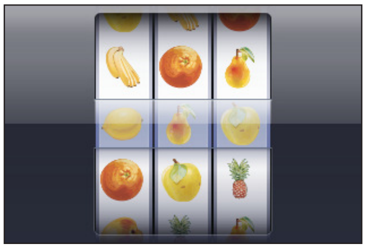


FIGURE 11-2

If you do not implement this method, the picker view distributes the available width equally between its components.

The `pickerView:rowHeightForComponent:` delegate method enables you to specify the height of each row in a given component. All rows in a component must have the same height.

You need to return a `UIView` subclass in the `pickerView:viewForRow:forComponent:resusingView:` delegate method. This method's arguments include a reference to the picker view, the row, and the component number.

The view returned by this method can be an instance of an existing UIKit class like `UIImageView` or `UILabel`. You can also provide instances of your own `UIView` subclass in which you have implemented custom drawing logic. Subclassing `UIView` is outside the scope of this book.

The last argument of this delegate method is a reference to an existing `UIView` object. If this argument is not nil, it will refer to one of the view objects provided by this method on a previous occasion. You should try to reuse it instead of creating one from scratch.

When you scroll a row in one of the components off the screen, the picker does not immediately destroy the corresponding view; instead it adds it to an internal cache of "reusable views." When it is time to display a new row in the same component the picker provides one of these cached views to your delegate method, encouraging you to reuse it instead of instantiating a fresh copy.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `CustomPickerTest` that displays a three-component custom picker view with images of fruits.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image files into the project.
- Add a Picker View and create an outlet for it in the view controller class.
- Add a `UILabel` instance and create an appropriate outlet for in the view controller class.
- Add three data arrays with the names of fruits to be displayed for each picker component in the view controller class and populate them in the `viewDidLoad` method.
- Add an `NSDictionary` object that maps names of fruits to image filenames.
- Implement the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols in your view controller class.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 11 folder in the download.*

Hints

- Use an `NSArray` object to create a data array whose contents will not change.
- An `NSDictionary` object contains a list of mappings between keys and values. Each key in a dictionary is unique.

Step-by-Step

1. Create a Single View Application in Xcode called `CustomPickerTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** CustomPickerTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson11
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    > *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Import image resources into your project.
    1. Ensure the project navigator is visible and the CustomPickerTest project is selected and expanded.
    2. Right-click the CustomPickerTest group and select Add Files to CustomPickerTest from the context menu.
    3. Select the `Images` folder in this lesson's resources on the DVD.
    4. Ensure the Copy Items to Destination Group's Folder (if needed) option is selected in the dialog box.
    5. Click the Add button.

3. Add a Picker View to your storyboard's scene.
    1. Use the Object library to add a Picker View to the default scene.
    2. Use the Size inspector to resize and position it at X = 0, Y = 0.
    3. Using the assistant editor, create an outlet for the picker view in the view controller class called `fruitPicker`.
    4. Set the view controller class to be the delegate and data source object for the picker.
        1. Right-click the picker view to display a context menu.
        2. Drag from the circle beside the `delegate` entry in the context menu to the view controller object (the yellow box) in the dock.
        3. Do the same for the `dataSource` entry in the context menu.
4. Add a label to the default scene.
    1. Use the Object library to add a `UILabel` instance.
    2. Using the Size inspector, resize and position it to X = 10, Y = 264, W = 300, H = 21.
    3. Using the Attributes inspector, set the Alignment property to be center aligned.
    4. Add an outlet to the view controller class for the label, and name it `resultLabel`.
5. Add three `strong` and `nonatomic` `NSArray` properties called `dataForComponent1`, `data-ForComponent2`, and `dataForComponent3` to the view controller class.
    1. Declare the properties in the interface file:

        ```
        @property (strong, nonatomic) NSArray* dataForComponent1;
        @property (strong, nonatomic) NSArray* dataForComponent2;
        @property (strong, nonatomic) NSArray* dataForComponent3;
        ```

    2. Synthesize them in the implementation file:

        ```
        @synthesize dataForComponent1;
        @synthesize dataForComponent2;
        @synthesize dataForComponent3;
        ```

6. Add a `strong` and `nonatomic` `NSDictionary` property called `nameToImageMapping` to the view controller class.
    1. Declare the property in the interface file:

        ```
        @property (strong, nonatomic) NSDictionary* nameToImageMapping;
        ```

    2. Synthesize it in the implementation file:

        ```
        @synthesize nameToImageMapping;
        ```

7. Have your view controller class conform to the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols by modifying its interface to the following:

    ```
    @interface Lesson11ViewController : UIViewController<UIPickerViewDelegate,
                                          UIPickerViewDataSource>
    ```

8. Your view controller's interface should now resemble the following:

```
@interface Lesson11ViewController : UIViewController<UIPickerViewDelegate,
                                    UIPickerViewDataSource>
@property (weak, nonatomic) IBOutlet UILabel *resultLabel;
@property (weak, nonatomic) IBOutlet UIPickerView *fruitPicker;
@property (strong, nonatomic) NSArray* dataForComponent1;
@property (strong, nonatomic) NSArray* dataForComponent2;
@property (strong, nonatomic) NSArray* dataForComponent3;
@property (strong, nonatomic) NSDictionary* nameToImageMapping;
@end
```

9. Add the following code to your view controller's `viewDidLoad` method to instantiate and initialize the three `NSArray` objects:

```
dataForComponent1 = [[NSArray alloc] initWithObjects:@"Apple",
                                     @"Banana", @"Lemon", @"Orange",
                                     @"Peach", @"Pear", @"Pineapple", nil];
dataForComponent2 = [[NSArray alloc] initWithObjects:@"Banana",
                                     @"Orange", @"Pear", @"Apple",
                                     @"Pineapple", @"Lemon", @"Peach", nil];
dataForComponent3 = [[NSArray alloc] initWithObjects:@"Pear",
                                     @"Peach", @"Lemon", @"Pineapple",
                                     @"Apple", @"Banana", @"Orange", nil];
```

10. Add the following code to your view controller's `viewDidLoad` method to instantiate and initialize the `NSDictionary` object:

```
nameToImageMapping = [[NSDictionary alloc]
                       initWithObjectsAndKeys:@"apple.png", @"Apple",
                                             @"banana.png", @"Banana",
                                             @"lemon.png", @"Lemon",
                                             @"orange.png", @"Orange",
                                             @"peach.png", @"Peach",
                                             @"pear.png", @"Pear",
                                             @"pineapple.png", @"Pineapple",
                                             nil];
```

11. Add the following code to your view controller's `viewDidLoad` method to set up the initial text of the `UILabel` instance `resultLabel`:

```
resultLabel.text = @"Match the fruits in each row!";
```

12. Implement the `numberOfComponentsInPickerView:` delegate method in your view controller as follows:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 3;
}
```

13. Implement the `pickerView:numberOfRowsInComponent:` data source method in your view controller as follows:

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
             numberOfRowsInComponent:(NSInteger)component
{
    if (component == 0)
        return [dataForComponent1 count];
    if (component == 1)
        return [dataForComponent2 count];

    return [dataForComponent3 count];
}
```

14. Implement the `pickerView:rowHeightForComponent:` data source method in your view controller as follows:

```
- (CGFloat)pickerView:(UIPickerView *)pickerView
           rowHeightForComponent:(NSInteger)component
{
    return 50;
}
```

15. Implement the `pickerView:viewForRow:forComponent:reusingView:` delegate method in your view controller as follows:

```
- (UIView *)pickerView:(UIPickerView *)pickerView
            viewForRow:(NSInteger)row
            forComponent:(NSInteger)component
            reusingView:(UIView *)view
{
    // get the fruit name
    NSString* keyString;
    if (component == 0)
        keyString = [dataForComponent1 objectAtIndex:row];
    else if (component == 1)
        keyString = [dataForComponent2 objectAtIndex:row];
    else if (component == 2)
        keyString = [dataForComponent3 objectAtIndex:row];

    NSString* imageFileName = [nameToImageMapping objectForKey:keyString];

    if(view == nil)
    {
        return [[UIImageView alloc] initWithImage:[UIImage imageNamed:imageFileName]];
    }

    ((UIImageView*)view).image = [UIImage imageNamed:imageFileName];
    return view;
}
```

16. Implement the `pickerView:didSelectRow:inComponent:` delegate method in your view controller as follows:

```
- (void)pickerView:(UIPickerView *)pickerView
        didSelectRow:(NSInteger)row
        inComponent:(NSInteger)component
{
    // get selected fruit in each component
    int selectedRowInComponent1 = [pickerView selectedRowInComponent:0];
    NSString* fruitInComponent1 = [dataForComponent1
                             objectAtIndex:selectedRowInComponent1];
    int selectedRowInComponent2 = [pickerView selectedRowInComponent:1];
    NSString* fruitInComponent2 = [dataForComponent2
```

```
                                   objectAtIndex:selectedRowInComponent2];
    int selectedRowInComponent3 = [pickerView selectedRowInComponent:2];
    NSString* fruitInComponent3 = [dataForComponent3
                                   objectAtIndex:selectedRowInComponent3];

    // if the same fruit is selected in
    // each row, then show a message
    if ([fruitInComponent1 isEqualToString:fruitInComponent2] &&
        [fruitInComponent2 isEqualToString:fruitInComponent3])
    {
        resultLabel.text = @"Jackpot!";
    }
    else
    {
        resultLabel.text = @"Match the fruits in each row!";
    }
}
```

17. Test your application in the iOS Simulator.

   1. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.

   2. Change the selection in the components of the picker. If you get three fruits of the same kind in the central row, you should see the Jackpot! message.

Please select Lesson 11 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo*, to view the video that accompanies this lesson.*

12 Navigation Controllers

A lot of applications for Apple's iOS devices display information on more than one view controller. On the iPad, for example, there is even a split view that displays two view controllers at once. The iPhone doesn't have that luxury, so it must provide a way to navigate through a hierarchy of view controllers.

To be able to navigate from view controller to view controller, by either drilling down into a hierarchy of views controllers or returning back to the original view controller, there has to be a process to accomplish this.

Apple has provided developers with a navigation controller that manages the presentation of these view controllers in your application. The class provided is `UINavigationController` that controls a hierarchy of `UIViewController` classes.

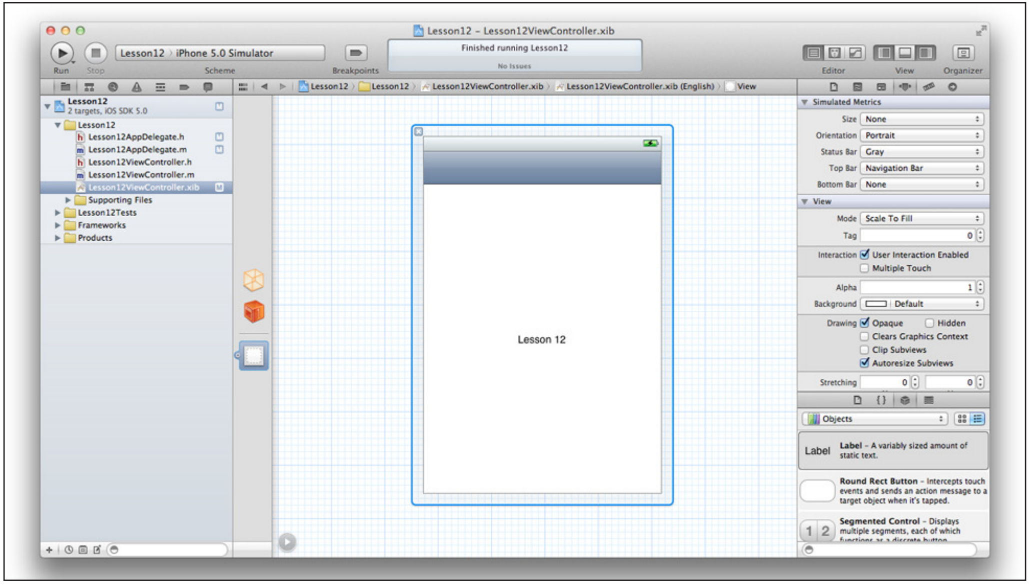
The `UIViewController` class contains a view property that contains your custom view information for user interaction. This view is what users see on their device. This custom user interface is designed using the Xcode Interface Builder editor as shown in Figure 12-1.

In this lesson you learn how to add a navigation controller to your single view application. You learn how to navigate from one view controller to another, including returning back to the original root view controller. You also learn how to transfer data from one view controller to another.

This lesson requires you to add the navigation controller manually in code. In Lesson 13, the Master-Detail Application template automatically adds the navigation controller and includes two view controllers already connected.

NAVIGATION CONTROLLER INTERFACE

To manage the presentation of your custom view controllers, the navigation controller uses custom views of its own as shown in Figure 12-2.


Lesson12 – Lesson12ViewController.xib
Run
Stop
Lesson12 › iPhone 5.0 Simulator
Scheme
Breakpoints
Finished running Lesson12
No Issues
Editor
View
Organizer
Lesson12 › Lesson12 › Lesson12ViewController.xib › Lesson12ViewController.xib (English) › View
Lesson12
2 targets, iOS SDK 5.0
Lesson12
Lesson12AppDelegate.h
Lesson12AppDelegate.m
Lesson12ViewController.h
Lesson12ViewController.m
Lesson12ViewController.xib
Supporting Files
Lesson12Tests
Frameworks
Products
Lesson 12
Objects
Label
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Simulated Metrics
Size: None
Orientation: Portrait
Status Bar: Gray
Top Bar: Navigation Bar
Bottom Bar: None

View
Mode: Scale To Fill
Tag: 0
Interaction: ✓ User Interaction Enabled; ☐ Multiple Touch
Alpha: 1
Background: Default
Drawing: ✓ Opaque; ☐ Hidden; ☐ Clears Graphics Context; ☐ Clip Subviews; ✓ Autoresize Subviews
Stretching: 0 0

FIGURE 12-1

The navigation controller interface contains the following key views:

- Navigation bar
- Navigation view
- Navigation toolbar

Navigation Bar

The navigation bar is located at the top of the view controller, and as you drill down the navigation stack, the left area of the navigation bar automatically presents a back button, enabling you to return to the previous view controller.

You can add buttons to the navigation bar that perform custom actions.

The following code adds a Done button to a navigation bar, which when tapped launches a `someAction` method in your source code:

```
UIBarButtonItem *doneButton = [[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemDone target:self
           action:@selector(someAction:)];
[[self navigationItem] setRightBarButtonItem:doneButton];
```

Navigation View

The navigation view is the view stored in the navigation controller's view property. You customize this view by adding buttons, input fields, pickers, or any other objects, depending on your application's requirements.

Navigation Toolbar

The navigation toolbar is optional, but is used for additional buttons that add more functionality to your application.

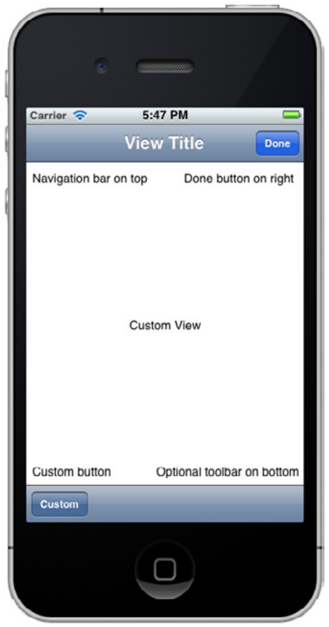


FIGURE 12-2

NAVIGATION CONTROLLER HIERARCHY

The process of navigating through view controllers within the hierarchy is stack-based, or first in, last out. The first view controller that is placed on this stack is known as the root view controller, and cannot be removed from the stack.

Each view controller is responsible for pushing the next view controller onto the navigation stack; however the navigation controller, through the use of the back button appearing on the left side of the navigation bar, controls the popping of the visible view controller, revealing the previous view controller by default.

Navigation Stack Management

The `UINavigationController` class provides methods to add and remove view controllers from the navigation stack. All management of the navigation stack originates with the initial view controller, known as the root view controller.

Table 12-1 summarizes the methods that manage the contents of the navigation stack.

TABLE 12-1: Navigation Stack Management

ACTION	DESCRIPTION
Display the next view controller	The `pushViewController:animated:` method pushes a new view controller onto the navigation stack.
Display the previous view controller	The navigation controller provides a back button to return to the previous view controller. Programatically, a call to the `popViewControllerAnimated:` method also displays the previous view controller.

continues

TABLE 12-1 *(continued)*

ACTION	DESCRIPTION
Return to the root view controller	To return to the root view controller a call to the `popToRootViewControllerAnimated:` method removes the all but the root view controller from the navigation stack.
Jump to a specific view controller	A call to the `setViewControllers:animated:` method enables immediate jumping to any view controller in the navigation stack.

xib-Based Applications

Beginning with the first iOS SDK release, the user interface was xib-based. That means that when you designed your user interface in the Interface Builder, the created file that contained all the UI information had an `.xib` extension.

Each view controller would have an associated `.xib` file that had to be loaded when it was pushed onto the navigation stack.

The following code pushes a view controller onto the navigation stack:

```
QuestionPoolViewController *controller =
     [[QuestionPoolViewController alloc] initWithStyle:UITableViewStyleGrou
ped];
[[self navigationController] pushViewController:controller animated:YES];
```

Storyboard-Based Applications

Beginning with iOS 5, the concept of storyboards was introduced. The developer now lays out the entire UI in one window as shown in Figure 12-3.

In addition to laying out your entire UI in one window, storyboarding enables the definition of all the transitions between view controllers to be defined graphically as well as all the controls that launch the transitions.

This ability to lay out your entire UI and program flow graphically significantly reduces the amount of code you need to write for your application.

Figure 12-4 illustrates how to define the transition between two view controllers in the Interface Builder by connecting the table view cell to the view controller's `performSegueWithIdentifier:sender:` method.

To pass data from one view controller to the next view controller, you implement `prepareForSegue:sender` method.

The following code passes a custom `Person` object to the next view controller:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
Person *aPerson = [[self people]
     objectAtIndex:[[[self tableView] indexPathForSelectedRow] row]];
DetailViewController *detailViewController = [segue
destinationViewController];
[detailViewController setPerson:aPerson];
}
```

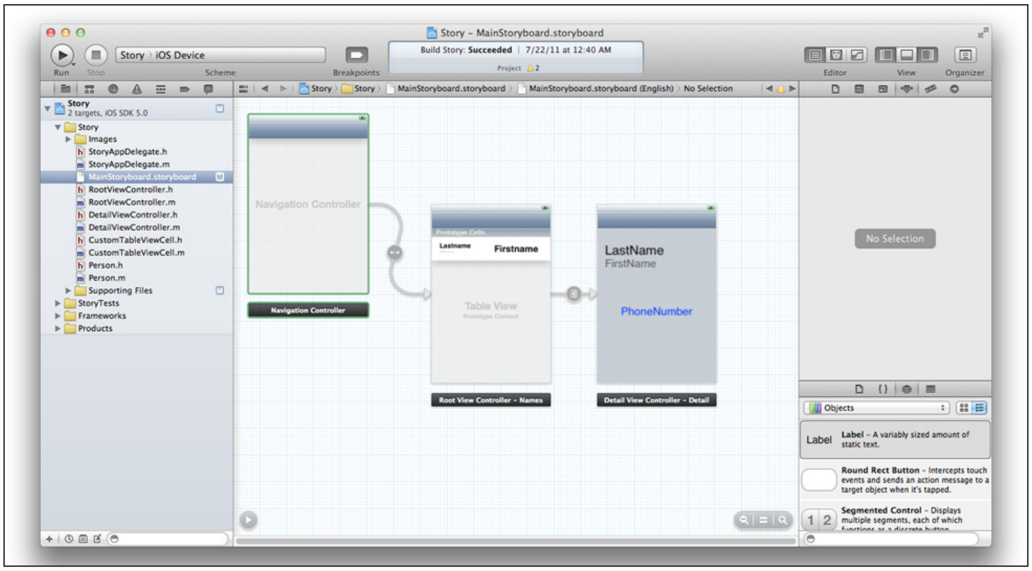


FIGURE 12-3

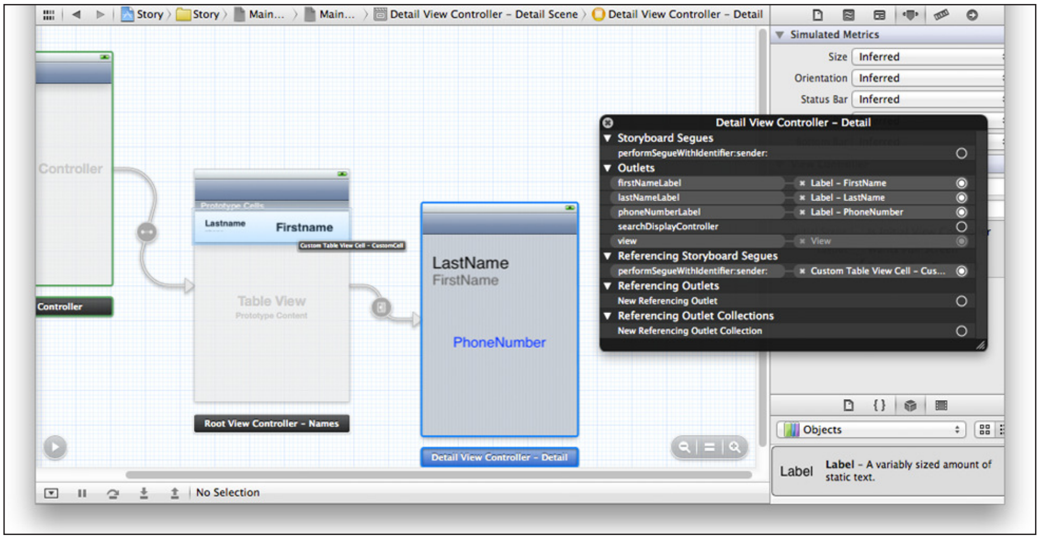

Story › Story › Main… › Main… › Detail View Controller – Detail Scene › Detail View Controller – Detail
Simulated Metrics
Size Inferred
Orientation Inferred
Status Bar Inferred
Controller
Lastname
Firstname
Custom Table View Cell – CustomCell
Table View
Root View Controller – Names
LastName
FirstName
PhoneNumber
Detail View Controller – Detail
Detail View Controller – Detail
Storyboard Segues
performSegueWithIdentifier:sender:
Outlets
firstNameLabel — Label – FirstName
lastNameLabel — Label – LastName
phoneNumberLabel — Label – PhoneNumber
searchDisplayController
view — View
Referencing Storyboard Segues
performSegueWithIdentifier:sender: — Custom Table View Cell – Cus…
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection
Objects
Label – A variably sized amount of static text.
No Selection

FIGURE 12-4

TRY IT

In this Try It, you implement two view controllers, where the root view controller passes a value to the detail view controller for display.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 12 folder in the download.*

Lesson Requirements

- Create an Xcode project using the Single View Application template.
- Create a storyboard including root and detail view controllers.
- Add a `UITextField` and `UIButton` to the root view controller.
- Add a `UILabel` to the detail view controller.
- Implement the `prepareForSegue:sender` method in the root view controller to pass the value entered in the `UITextField`.
- Display the detail view controller with the `UILabel` populated with the passed value that was entered in the root view controller's `UITextField`.

Hints

- Because this application uses storyboards instead of `.xib` files, remember to have the Use Storyboard option checked at project creation.
- The view controller transition launches when the button on the navigation bar is tapped, so the `performSegueWithIdentifier:sender` connection from the root view controller to the detail view controller will be between the button and the detail view controller.

Step-by-Step

1. Create a Single View Application.
    1. Launch Xcode.
    2. Create your new iOS project.
        a. To create a new project, select Create a New Xcode Project.
        b. On the left under iOS, select Application.
        c. Select Single View Application from the template list and click Next.

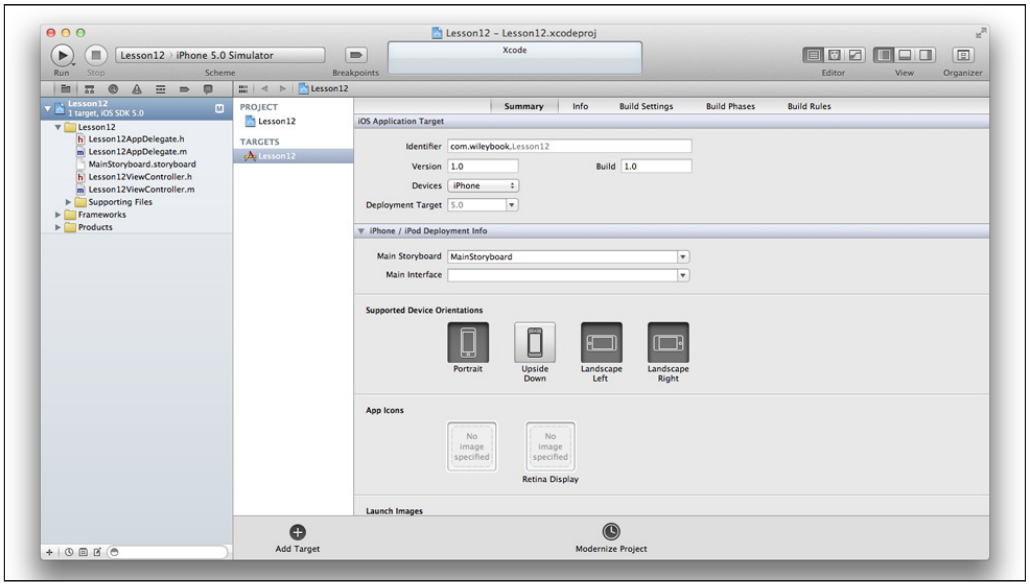
d. Choose the following options for your project:

   - **Product Name:** Lesson12
   - **Company Identifier:** com.wileybook
   - **Class Prefix:** Lesson12
   - **Device Family:** iPhone
   - **Use Storyboard:** Checked
   - **Use Automatic Reference Counting:** Checked
   - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you may use any unique identifier for your application.*

e. Select the location on your computer where the project will be saved and select Create.

f. Your Xcode project has been created as shown in Figure 12-5.



iOS Application Target

Identifier: com.wileybook.Lesson12
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0

iPhone / iPod Deployment Info

Main Storyboard: MainStoryboard
Main Interface:

Supported Device Orientations

App Icons

Launch Images

FIGURE 12-5

2. Design the user interface.
    1. On the left, select `MainStoryboard.storyboard`.
    2. On the right select the third button in the View section, to display the Utilities view as shown in Figure 12-6.

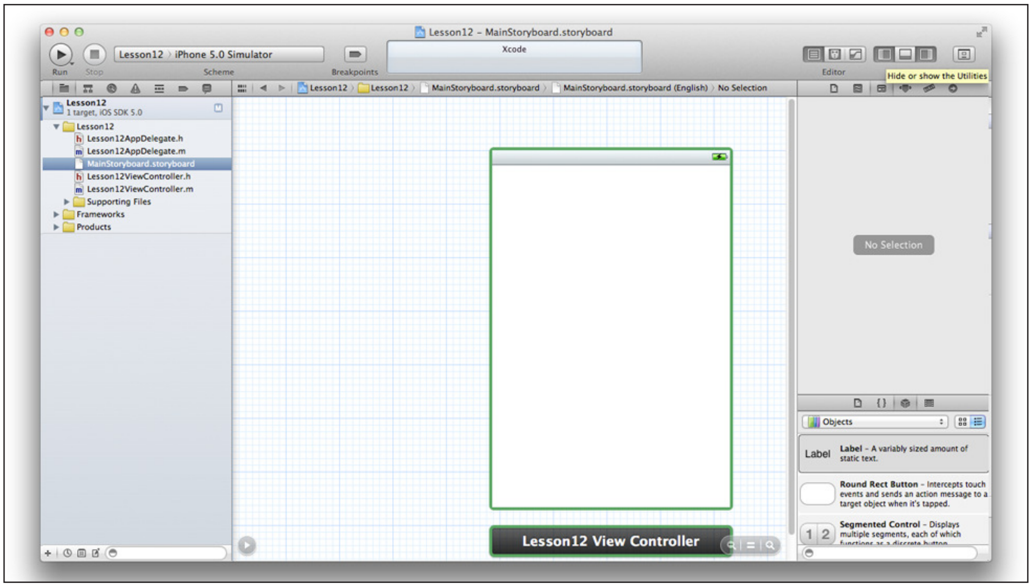


FIGURE 12-6

3. Select the bottom bar of the default view controller in the storyboard and tap the Delete key to remove it.

It is important to make sure the default view controller is completely removed from the storyboard. Your storyboard must be completely empty after step 3.

4. To create the detail view controller:
    a. Select the Lesson12 folder on the left.
    b. Select File ➪ New ➪ New File from the Xcode menu.
    c. On the left under iOS, select Cocoa Touch.
    d. Select UIViewController from the template list and click Next.
    e. Choose the following options for your new file:
        - **Class:** DetailViewController
        - **Subclass of:** UIViewController

- **Deselect:** Targeted for iPad
- **Deselect:** With XIB for user interface

f. Click Next.

g. Click Create to save the class in your project folder.

5. Select `MainStoryboard.storyboard` and on the lower right select the third icon to display the object library; scroll through the list of objects and drag a Navigation Controller on your storyboard window.

6. Complete the following to design the root view controller:

   a. Click inside the root view controller view and then click the frame of the view to select the view controller.

   b. Select the Identity Inspector from the toolbar on the right.

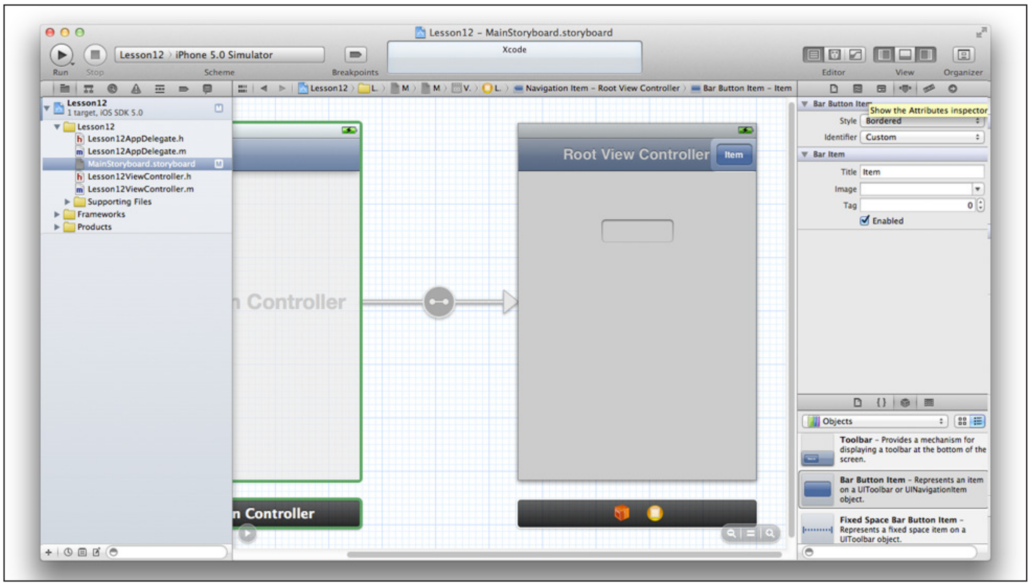
   c. In the Custom Class section of the Identity Inspector, replace `UIViewController` with your `Lesson12ViewController` class.

   d. Drag a Text Field from the objects in the lower right, and place it in the center and near the top of the root view controller window.

   e. Drag a Bar Button Item from the objects in the lower right, and place it on the right corner of the navigation bar.

   f. Select the Attributes Inspector from the toolbar on the right as shown in Figure 12-7.

   g. Select Action from the Identifier drop-down list in the Bar Button Item section of the inspector.



Bar Button Item
Style: Bordered
Identifier: Custom

Bar Item
Title: Item
Image:
Tag: 0
✓ Enabled

Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.

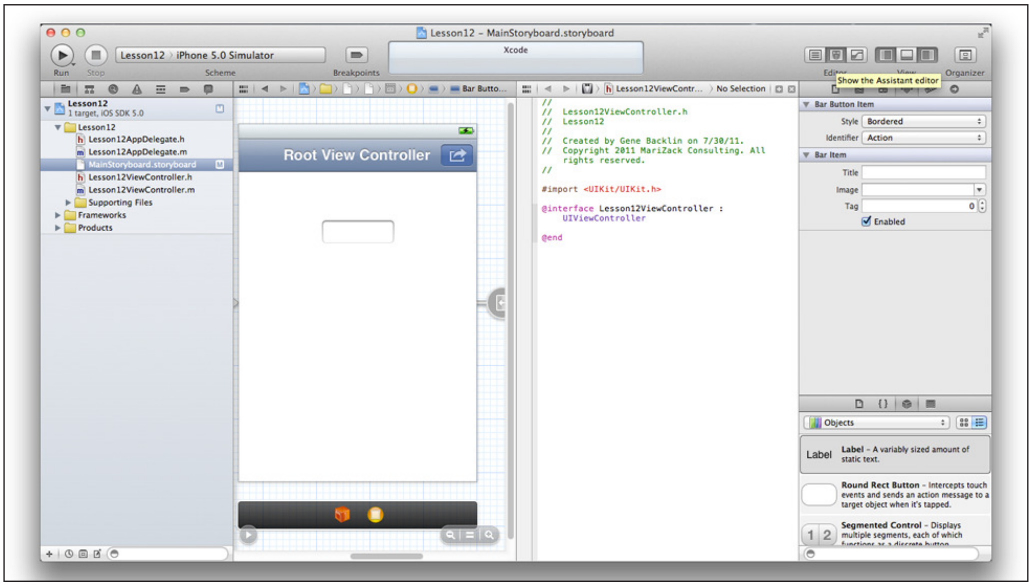
Bar Button Item – Represents an item on a UIToolbar or UINavigationItem object.

Fixed Space Bar Button Item – Represents a fixed space item on a UIToolbar object.

FIGURE 12-7

7. Complete the following to design the detail view controller:
    a. Drag a View Controller object from the objects in the lower right, and drag it on the storyboard to the right of the root view controller.
    b. From the Identity Inspector choose the `DetailViewController` class for the Custom Class.
    c. Drag a Label from the objects in the lower right, and place it in the middle of the detail view controller window.
8. Select File ➪ Save to save your project.

3. Make the connection for the navigation transition by selecting the bar button item in the navigation bar of the root view controller, Ctrl-drag to the detail view controller, and select Push, from the Storyboard Segues list..
4. Make the connection for the `UITextField`.
    1. Select the root view controller in the storyboard and from the Editor section, select the Assistant Editor button as shown in Figure 12-8.


Lesson12 – MainStoryboard.storyboard
Run
Stop
Scheme
Breakpoints
Lesson12 › iPhone 5.0 Simulator
Xcode
Editor
View
Organizer
Show the Assistant editor
Lesson12
1 target, iOS SDK 5.0
Lesson12
Lesson12AppDelegate.h
Lesson12AppDelegate.m
MainStoryboard.storyboard
Lesson12ViewController.h
Lesson12ViewController.m
Supporting Files
Frameworks
Products
Bar Butto...
Root View Controller
Lesson12ViewContr...
No Selection
Objects
Label
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

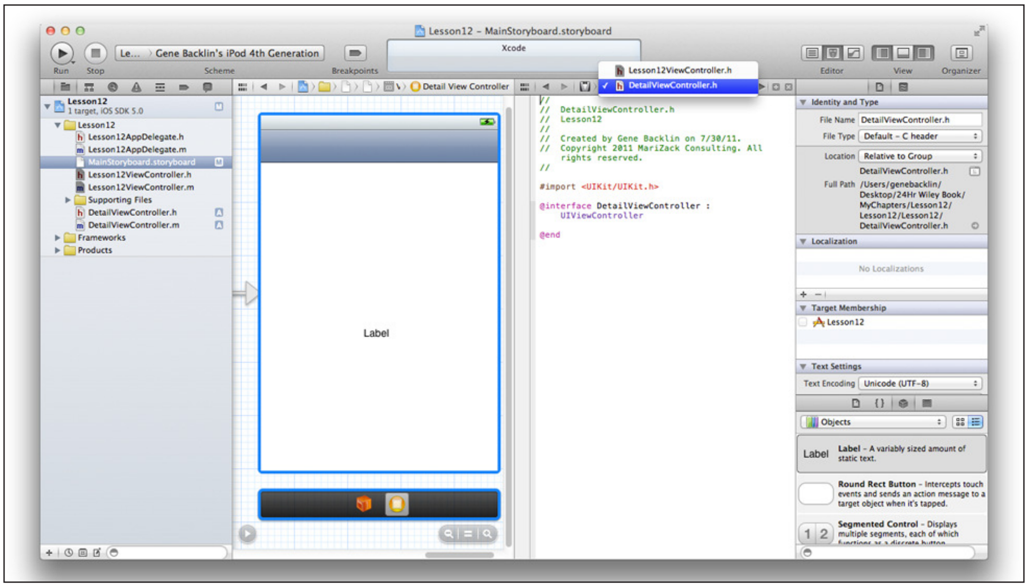
```
//
//  Lesson12ViewController.h
//  Lesson12
//
//  Created by Gene Backlin on 7/30/11.
//  Copyright 2011 MariZack Consulting. All
    rights reserved.
//

#import <UIKit/UIKit.h>

@interface Lesson12ViewController :
    UIViewController

@end
```

Bar Button Item
Style: Bordered
Identifier: Action
Bar Item
Title:
Image:
Tag: 0
[✓] Enabled

FIGURE 12-8

2. Select the text field and Ctrl-drag to the interface source code just above the `@end`.
3. Enter `entryTextField` for the outlet name and click Connect.

5. Make the connection for the `UILabel`.
   1. Select the detail view controller in the storyboard.
   2. Select `DetailViewController.h` from the list as shown in Figure 12-9.
   3. Select the label and Ctrl-drag to the interface source code just above the `@end`.
   4. Enter `displayLabel` for the outlet name and click Connect.
   5. From the Editor section, select the Standard editor to hide the `DetailViewController.h` file.



```
//
//  DetailViewController.h
//  Lesson12
//
//  Created by Gene Backlin on 7/30/11.
//  Copyright 2011 MariZack Consulting. All
    rights reserved.
//

#import <UIKit/UIKit.h>

@interface DetailViewController :
    UIViewController

@end
```

Lesson12 – MainStoryboard.storyboard
Xcode
Run
Stop
Scheme
Breakpoints
Editor
View
Organizer
Le... Gene Backlin's iPod 4th Generation
Detail View Controller
Lesson12ViewController.h
✓ DetailViewController.h
Lesson12
1 target, iOS SDK 5.0
Lesson12
Lesson12AppDelegate.h
Lesson12AppDelegate.m
MainStoryboard.storyboard
Lesson12ViewController.h
Lesson12ViewController.m
Supporting Files
DetailViewController.h
DetailViewController.m
Frameworks
Products
Label
Identity and Type
File Name DetailViewController.h
File Type Default – C header
Location Relative to Group
DetailViewController.h
Full Path /Users/genebacklin/Desktop/24Hr Wiley Book/MyChapters/Lesson12/Lesson12/Lesson12/DetailViewController.h
Localization
No Localizations
Target Membership
Lesson12
Text Settings
Text Encoding Unicode (UTF-8)
Objects
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which

FIGURE 12-9

6. Modify the view controllers.
   1. On the left, select `Lesson12ViewController.m` and add the following import at the top:

      ```
      #import "DetailViewController.h"
      ```

   2. Add the following method at the bottom above the `@end`:

      ```
      - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
          DetailViewController *detailViewController =
                [segue destinationViewController];
          [detailViewController
                setDetailText:[[self entryTextField] text]];
      }
      ```

3. Select `DetailViewController.h` and add the following property above the `@end`:

   ```
   @property (strong, nonatomic) NSString *detailText;
   ```

4. Select `DetailViewController.m` and add the following at the top:

   ```
   @synthesize detailText;
   ```

5. Uncomment the `viewDidLoad` method and add the following below `[super viewDidLoad]`:

   ```
   [self setTitle:@"Detail View Controller"];
   [[self displayLabel] setText:[self detailText]];
   ```

6. In the `viewDidUnLoad` method, add the following above `[super viewDidUnload]`:

   ```
    [self setDetailText:nil];
   ```

7. Run the application.

   1. Select the iPhone Simulator to run the application.
   2. Click the Run button from Xcode.
   3. When the application launches, enter **Hello** in the text field and click the action button on the navigation bar.
   4. The detail view controller is displayed with your Hello in the label.
   5. Click the back button to return to the root view controller.

Please select Lesson 12 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

13
Table Views

To display a list of values, the table view is one of the most common user interface elements in iOS development. The table view is more common in iPhone applications than in iPad applications because of the size of the viewing area.

In addition to displaying a scrollable list of data, you can present the list in the following ways to make viewing easier:

- As an indexed list
- As a part of related groups

Developers use the table view within a navigation controller–based application as a way to drill down through a hierarchical set of data. This style of application can be created easily by choosing the Master-Detail Application from the application template upon Xcode project creation.

The physical makeup of the table view is limited to a single-column vertical scrolling control. The list can be presented in groups along with a header and footer for each group section, and each section contains a row of data related to that specific section.

The Contacts application that is pre-installed on all iOS devices uses the indexed format to present names. The sections are grouped by the letters of the alphabet, and within each letter group, are the rows of names beginning with that letter. For example, under the group J would be a name Jones.

In this lesson you learn how to add data to a table view. You learn how to group related information into sections, and you learn how to transfer data from a selected table view cell to a separate detail view controller to display the selected list item.

TABLE VIEW WORKFLOW

To implement a table view into your application, two required elements must be in your code for the navigation workflow to be successful:

- You must display a known set of values.
- You must respond to the user's table view cell selection.

Display Values

The data to be displayed must be in list form. An `NSArray` is used frequently to store the data, as it can be referenced easily with the row index being used as the index to the array.

The number of sections represents how your data is grouped. If you have an array of related items, the number of sections will be one. If the data contains several groups, as the contacts grouped names alphabetically, the `numberOfSectionsInTableView` method returns the number of sections or groups that are present in the array.

The number of rows that are displayed usually is the count of the `NSArray`, for example. This count is returned via the `tableView:numberOfRowsInSection` method.

The actual value in the `NSArray` being displayed is a one-to-one relationship between the array index and the `indexPath`'s `row` property. The actual table view cell population occurs in the `tableView:cellForRowAtIndexPath` method.

Row Selection

Responses to a row being selected differ in nib-based versus storyboard applications.

Nib-based

When the user selects a table view cell in a nib-based application, the `tableView:didSelectRowAtIndexPath` method is called, and like the `tableView:cellForRowAtIndexPath` method, the selected `indexPath`'s `row` property is the same as the index in the array. When the row is selected, usually a view controller is pushed on to the navigation stack.

Storyboard

When the user selects a table view cell in a storyboard application that has the `performSegueWithIdentifier:sender` connection established, the `prepareForSegue:sender` method is called before the view controller is pushed on to the navigation stack.

TABLE VIEW STYLES

A table view is an instance of the `UITableView` class and has two styles of presentation. It can be a `UITableViewStyleGrouped` or `UITableViewStylePlain` style, as shown in Figure 13-1.

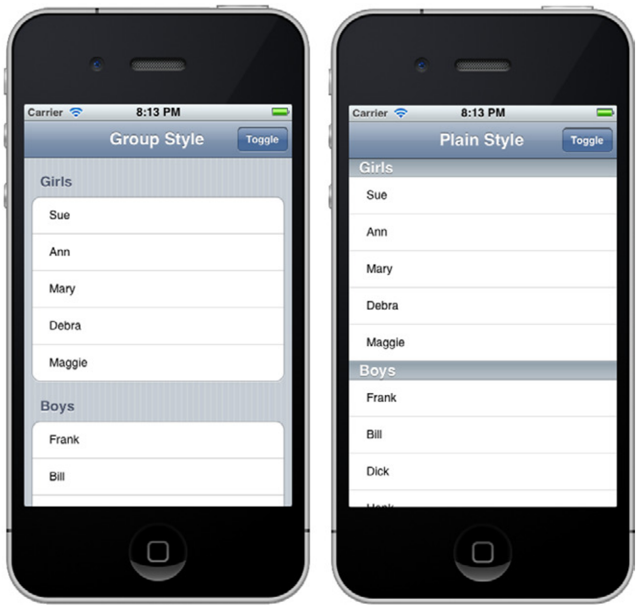


FIGURE 13-1

Each row of a table view is an instance of the `UITableViewCell` class. Custom table view cells subclass this class. The table view's appearance and source for data are handled by the following:

- Delegate methods
- Data source methods

Delegate Methods

For a view controller to indicate that it is a `UITableView` delegate, it must implement the `UITableViewDelegate` protocol. The delegate then handles managing selections, configures the section headings and footers, and assists in the deletion and reordering of table view cells.

The following are common delegate methods to implement in the view controller:

- `tableView:heightForRowAtIndexPath:`
- `tableView:willDisplayCell:forRowAtIndexPath:`
- `tableView:didSelectRowAtIndexPath:`
- `tableView:didDeselectRowAtIndexPath:`
- `tableView:commitEditingStyle:forRowAtIndexPath:`
- `tableView:canEditRowAtIndexPath:`

Data Source Methods

For a view controller to indicate that it is a `UITableView` data source, it must implement the `UITableViewDataSource` protocol. The data source provides the table view with the data items needed to populate or modify the contents of a `UITableViewCell` row.

The following are common data source methods to implement in the view controller:

- `tableView:cellForRowAtIndexPath:` (required)
- `tableView:numberOfRowsInSection:` (required)
- `numberOfSectionsInTableView:`
- `tableView:titleForHeaderInSection:`
- `tableView:titleForFooterInSection:`
- `tableView:commitEditingStyle:forRowAtIndexPath:`
- `tableView:canEditRowAtIndexPath:`
- `tableView:canMoveRowAtIndexPath:`
- `tableView:moveRowAtIndexPath:toIndexPath:`

NEW FOR IOS 5

For iOS 5, the table view has received a number of new features that can be used by the developer.

Table View Additions

Table 13-1 highlights the additions that have been added to the `UITableView` class.

TABLE 13-1: iOS 5 Table View Additions

TASKS	DESCRIPTION
`allowsMultipleSelection`	Determines if more than one row can be selected outside of editing mode.
`allowsMultipleSelectionDuringEditing`	Determines if more than one row can be selected during editing mode.
`indexPathsForSelectedRows`	The index paths of the rows that have been selected.
`moveRowAtIndexPath:toIndexPath:`	Moves a row at one index path to another row at its index path.

TASKS	DESCRIPTION
`moveSection:toSection:`	Moves a section in a table view to another section location.
`registerNib:forCellReuseIdentifier:`	Registers a custom table view cell's nib file for reuse by the table view.

Constants

When a new table view cell is added programmatically, there is a new constant, `UITableViewRowAnimationAutomatic`, that allows for the appropriate animation to be performed. The following methods can use this constant:

- ➤ `insertRowsAtIndexPaths:withRowAnimation:`
- ➤ `insertSections:withRowAnimation:`
- ➤ `deleteRowsAtIndexPaths:withRowAnimation:`
- ➤ `deleteSections:withRowAnimation:`
- ➤ `reloadRowsAtIndexPaths:withRowAnimation:`
- ➤ `reloadSections:withRowAnimation:`

When you want the `UITableView` to choose a default value for the height of the header or footer, the delegate methods `tableView:heightForHeaderInSection:` or `tableView:heightForFooterInSection:` would return the new constant `UITableViewAutomaticDimension`. The height will be sized according to the values being returned in the `tableView:titleForHeaderInSection:` or `tableView:titleForFooterInSection:` methods, so they will fit properly.

Storyboard Additions

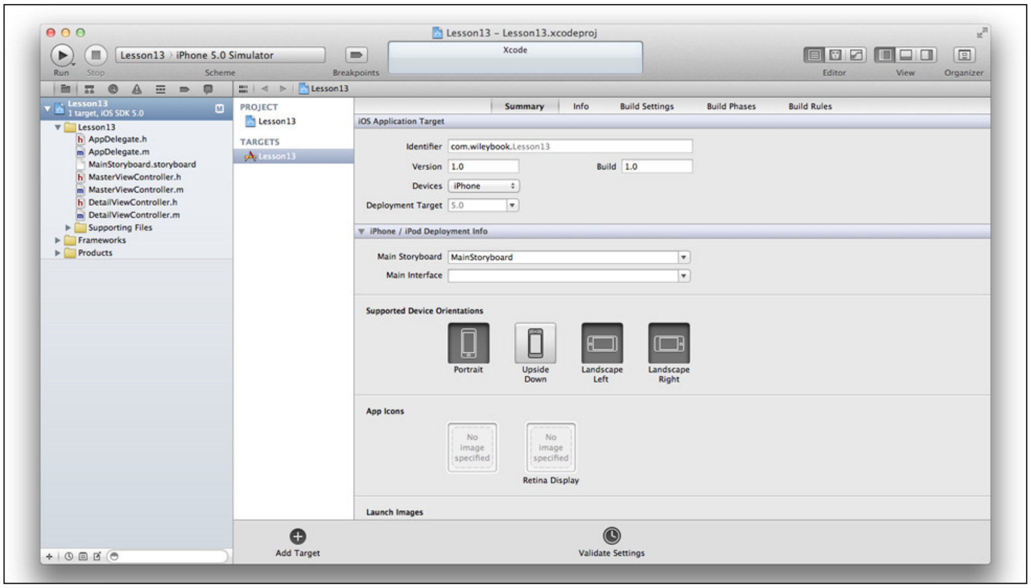
Beginning with iOS 5, the concept of storyboards was introduced. As before, with nib-based table views, the developer continues to use a dynamic prototype table view cell, but now has the ability to design a static table view.

Static Table Views

A static table view is designed directly through the Interface Builder. In addition to the table view cell itself, you can add the sections, headers, and footers. Because these are not dynamically produced, there is no data source for you to identify. See Figure 13-2.

Prototype-based Table Views

Dynamic prototype-based table views are also designed directly through the Interface Builder. They are easy to implement, and the prototype table view cell is similar to pre-storyboard custom table view cells. See Figure 13-3.



iOS Application Target

Identifier com.wileybook.Lesson13
Version 1.0 Build 1.0
Devices iPhone
Deployment Target 5.0

iPhone / iPod Deployment Info

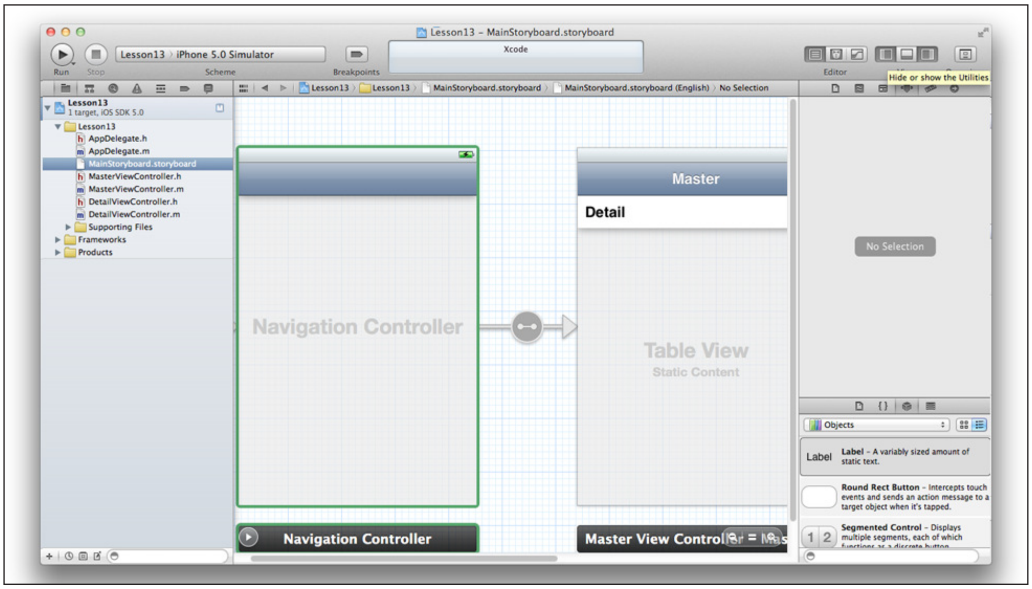
Main Storyboard MainStoryboard
Main Interface

Supported Device Orientations

App Icons

Launch Images

FIGURE 13-2



Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 13-3

Prototype table view cells must have reuse identifiers declared. They are defined in the Attributes Editor under the Table View section labeled Identifier.

TRY IT

In this Try It, you implement a dynamic-based table view using a storyboard. The table view has two sections and the application toggles the sections between each other within the table view.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson13 folder in the download.*

Lesson Requirements

- Create an Xcode project using the Master-Detail Application template.
- Create a storyboard including root and detail view controllers.
- Implement a table view using dynamic prototype content.
- Populate the header section of the table view.
- Implement the `prepareForSegue:sender` method in the root view controller to pass the values associated with the selected table view cell.
- Add a bar button item to the navigation bar that toggles the sections displayed in the table view.
- Display the detail view controller with the passed values associated with the selected table view cell.

Hints

- Because this application uses storyboards instead of `.xib` files, remember to have the Use Storyboard option checked at project creation.
- Two arrays, one for each section, will be used for table view cell population.
- A method `titleForSection` will be added to populate the title for each section in the table view.
- The bar button item on the navigation bar will launch the `toggleSections` method.
- The view controller transition will launch when a table view cell is selected, so the `performSegueWithIdentifier:sender` connection from the root view controller to the detail view controller will be between the button and the detail view controller.

Step-by-Step

1. Create a Master-Detail application.
    1. Launch Xcode.

2. Create your new iOS project.
   a. Select Create a New Xcode Project.
   b. On the left under iOS, select Application.
   c. Select Master-Detail Application from the template list and click Next.
   d. Choose the following options for your project:
      - **Product Name:** Lesson13
      - **Company Identifier:** com.wileybook
      - **Class Prefix:** leave blank
      - **Device Family:** iPhone
      - **Use Storyboard:** Checked
      - **Use Core Data:** Unchecked
      - **Use Automatic Reference Counting:** Checked
      - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

   e. Select the location on your computer where the project will be saved.
   f. Your Xcode project has been created as shown in Figure 13-4.

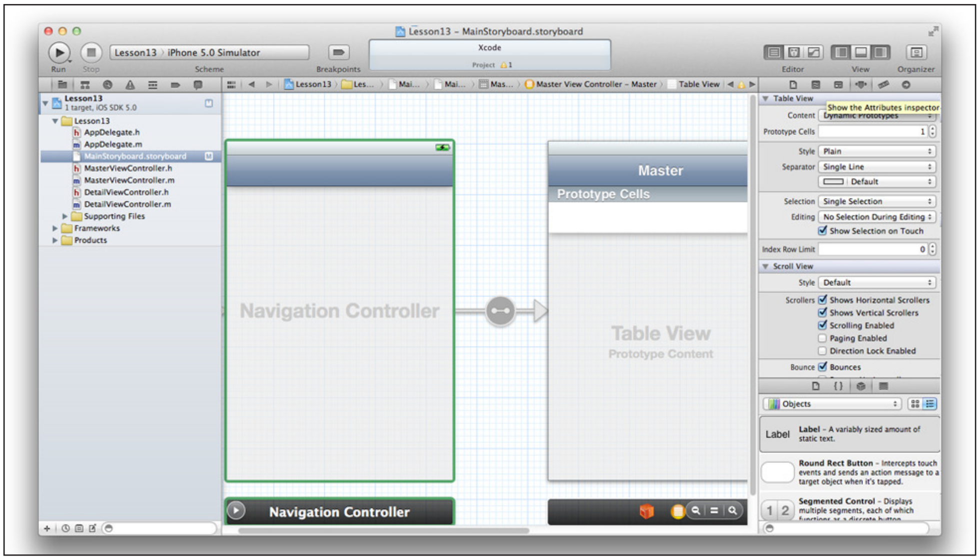


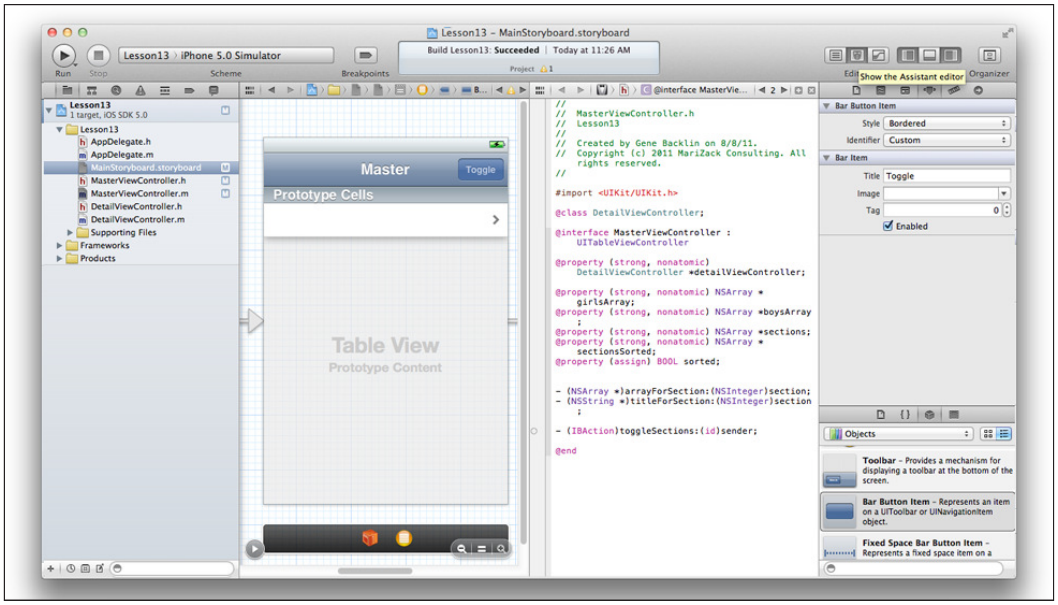
Table View
Content: Dynamic Prototypes
Prototype Cells: 1
Style: Plain
Separator: Single Line
Default
Selection: Single Selection
Editing: No Selection During Editing
☑ Show Selection on Touch
Index Row Limit: 0

Scroll View
Style: Default
Scrollers: ☑ Shows Horizontal Scrollers
☑ Shows Vertical Scrollers
☑ Scrolling Enabled
☐ Paging Enabled
☐ Direction Lock Enabled
Bounce: ☑ Bounces

Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 13-4

2. Design the user interface.
    1. On the left, select `MainStoryboard.storyboard`.
    2. On the right select the third button in the View section to display the Utilities view, as shown in Figure 13-5.



```
//
//  MasterViewController.h
//  Lesson13
//
//  Created by Gene Backlin on 8/8/11.
//  Copyright (c) 2011 MariZack Consulting. All
    rights reserved.
//

#import <UIKit/UIKit.h>

@class DetailViewController;

@interface MasterViewController :
    UITableViewController

@property (strong, nonatomic)
    DetailViewController *detailViewController;

@property (strong, nonatomic) NSArray *
    girlsArray;
@property (strong, nonatomic) NSArray *boysArray
    ;
@property (strong, nonatomic) NSArray *sections;
@property (strong, nonatomic) NSArray *
    sectionsSorted;
@property (assign) BOOL sorted;


- (NSArray *)arrayForSection:(NSInteger)section;
- (NSString *)titleForSection:(NSInteger)section
    ;

- (IBAction)toggleSections:(id)sender;

@end
```

Bar Button Item
Style: Bordered
Identifier: Custom

Bar Item
Title: Toggle
Image:
Tag: 0
☑ Enabled

Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.

Bar Button Item – Represents an item on a UIToolbar or UINavigationItem object.

Fixed Space Bar Button Item – Represents a fixed space item on a

FIGURE 13-5

3. To change the table view cell to a dynamic prototype:
    a. Select the center of the table view right over the area that has Table View Static Content.
    b. Select the fourth icon, the Attributes Inspector, from the utilities toolbar and change the Content from Static Cells to Dynamic Prototypes as shown in Figure 13-6.
    c. Select the table view cell and enter **Cell** for the Identifier in the Table View Cell section at the top of the Attributes Inspector.
4. To add a bar button item to the navigation bar:
    a. Scroll and select a bar button item from the bottom of the Objects section of the utilities, and drag it to the right side of the navigation bar.
    b. Double-click Item and enter **Toggle**.
    c. From the Editor section on the upper right of Xcode, select the Assistant Editor button as shown in Figure 13-7.

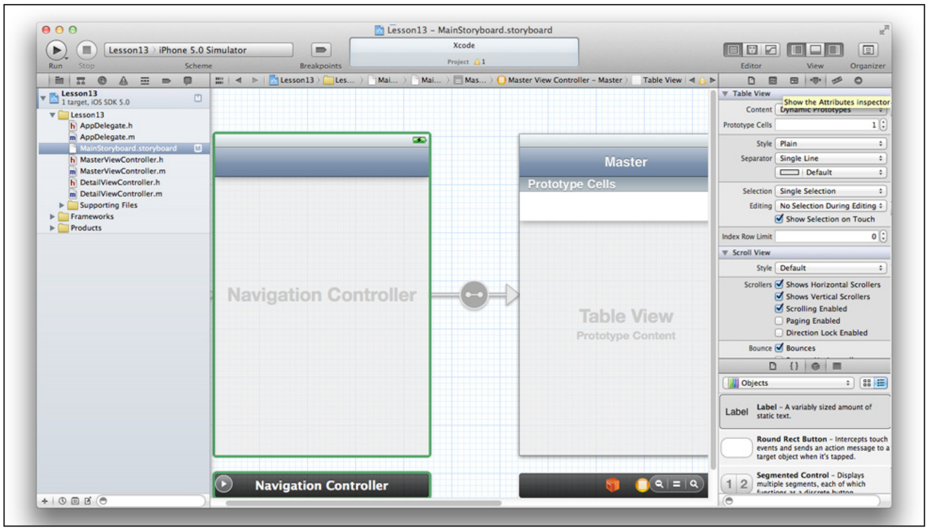
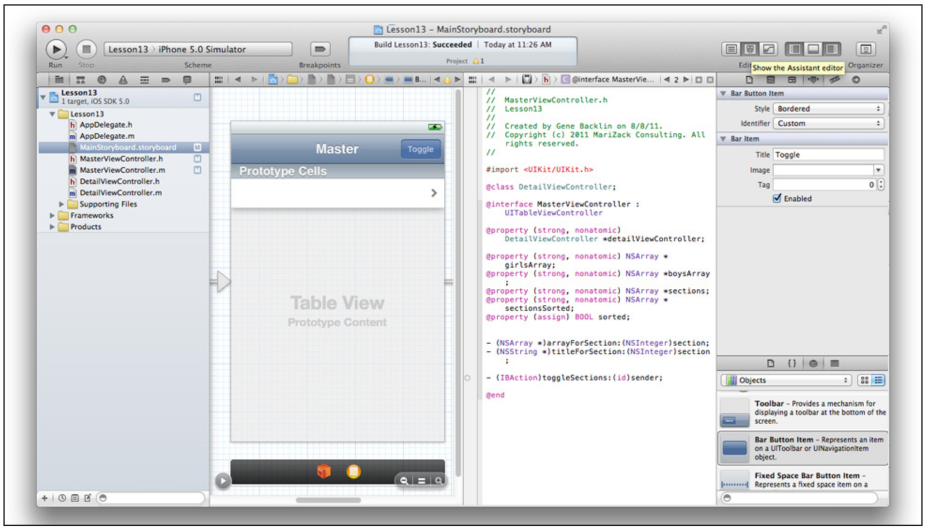


Table View
Content: Dynamic Prototypes
Prototype Cells: 1
Style: Plain
Separator: Single Line
Default
Selection: Single Selection
Editing: No Selection During Editing
✓ Show Selection on Touch
Index Row Limit: 0

Scroll View
Style: Default
Scrollers: ✓ Shows Horizontal Scrollers
✓ Shows Vertical Scrollers
✓ Scrolling Enabled
Paging Enabled
Direction Lock Enabled
Bounce: ✓ Bounces

Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

FIGURE 13-6


Lesson13 – MainStoryboard.storyboard
Build Lesson13: Succeeded | Today at 11:26 AM
Show the Assistant editor
Master
Toggle
Prototype Cells
Table View
Prototype Content
@interface MasterVie...
Objects

```
//
//  MasterViewController.h
//  Lesson13
//
//  Created by Gene Backlin on 8/8/11.
//  Copyright (c) 2011 MariZack Consulting. All
    rights reserved.
//

#import <UIKit/UIKit.h>

@class DetailViewController;

@interface MasterViewController :
    UITableViewController

@property (strong, nonatomic)
    DetailViewController *detailViewController;

@property (strong, nonatomic) NSArray *
    girlsArray;
@property (strong, nonatomic) NSArray *boysArray
    ;
@property (strong, nonatomic) NSArray *sections;
@property (strong, nonatomic) NSArray *
    sectionsSorted;
@property (assign) BOOL sorted;


- (NSArray *)arrayForSection:(NSInteger)section;
- (NSString *)titleForSection:(NSInteger)section
    ;

- (IBAction)toggleSections:(id)sender;

@end
```

Bar Button Item
Style: Bordered
Identifier: Custom

Bar Item
Title: Toggle
Image:
Tag: 0
✓ Enabled

Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.
Bar Button Item – Represents an item on a UIToolbar or UINavigationItem object.
Fixed Space Bar Button Item – Represents a fixed space item on a

FIGURE 13-7

d. Select the Toggle button and control-drag to the interface source code just above the `@end`.

e. Select Action for Connection and enter **toggleSections** for the action name and click Connect.

5. Select File ➪ Save to save your project.

3. Make the connection for the navigation transition.
    1. Select the table view cell you just labeled Cell and control-drag to the detail view controller and select `performSegueWithIdentifier:sender`.
4. Modify the view controllers.
    1. Select `MasterViewController.h` and add the following above the `@end`:

    ```
    @property (strong, nonatomic) NSArray *girlsArray;
    @property (strong, nonatomic) NSArray *boysArray;
    @property (strong, nonatomic) NSArray *sections;
    @property (strong, nonatomic) NSArray *sectionsSorted;
    @property (assign) BOOL sorted;
    - (NSArray *)arrayForSection:(NSInteger)section;
    - (NSString *)titleForSection:(NSInteger)section;
    ```

    2. Select `MasterViewController.m` and add the following right below `@implementation MasterViewController`:

    ```
    @synthesize girlsArray;
    @synthesize boysArray;
    @synthesize sections;
    @synthesize sectionsSorted;
    @synthesize sorted;
    ```

    3. In the `ViewDidLoad` method, add the following arrays for the girls and boys name, and for the section groups:

    ```
       [self setGirlsArray:[NSArray arrayWithObjects:@"Sue",
                @"Ann", @"Mary", @"Debra", @"Maggie", nil]];
        [self setBoysArray:[NSArray arrayWithObjects:@"Frank",
                @"Bill", @"Dick", @"Hank", @"Jean", nil]];
        [self setSections:[NSArray arrayWithObjects:@"Girls",
                           @"Boys", nil]];
    ```

    4. The section toggles between sorted and unsorted sections. Add the following array also in the `ViewDidLoad` method for the sorted section groups and initialize the `sorted` variable to `NO`, indicating that the sections are not initially sorted:

    ```
    [self setSectionsSorted:[[self sections]
         sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)]];
        [self setSorted:NO];
    ```

    5. To display section headers, implement the following method:

    ```
    - (NSString *)tableView:(UITableView *)tableView
            titleForHeaderInSection:(NSInteger)section {
        return [self titleForSection:section];
    }
    ```

    6. To inform the table view on the number of sections that are to be displayed, add the following:

    ```
     // Customize the number of sections in the table view.
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    ```

```
    NSInteger sectionCount = 0;

    if([self sorted]) {
        sectionCount = [[self sectionsSorted] count];
    } else {
        sectionCount = [[self sections] count];
    }
    return sectionCount;
}
```

7. For each section are the arrays containing the actual data for the associated section. To inform each section how many rows are to be displayed, add the following:

```
// Customize the number of rows in the table view.
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section {
    return [[self arrayForSection:section] count];
}
```

8. To display each row with the values in the array, you implement the `tableView:cellForRowAtIndexPath:` as follows:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSInteger row = [indexPath row];
    NSInteger section = [indexPath section];
    NSString *cellText = [[self arrayForSection:section] objectAtIndex:row];
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
          [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    // Configure the cell.
    [[cell textLabel] setText:cellText];
    return cell;
}
```

9. When the Toggle button is tapped, the `toggleSections` method is launched, which checks to see if the sorted or unsorted ordering of sections is required, and then calls the `moveSection:toSection:` method to move the sections, as implemented in the following method:

```
- (IBAction)toggleSections:(id)sender {
    NSArray *currentNames = nil;
    NSArray *newNames = nil;

    if([self sorted]) {
        currentNames = [self sectionsSorted];
        newNames = [self sections];
    } else {
        currentNames = [self sections];
        newNames = [self sectionsSorted];
    }

    [self setSorted:![self sorted]];
    [[self tableView] beginUpdates];

    NSUInteger currentIndex = 0;
    NSUInteger newIndex = 0;

    for(NSString *currentName in currentNames) {
        for(NSString *newName in newNames) {
            if([newName isEqualToString:currentName]) {
                [[self tableView] moveSection:currentIndex
toSection:newIndex];
                newIndex = 0;
                break;
            }
            newIndex++;
        }
        currentIndex++;
    }

    [[self tableView] endUpdates];
}
```

The moveSection:toSection: method is new to iOS 5.

10. To ensure the correct values are used for each section, create the `arrayForSection` method:

```
- (NSArray *)arrayForSection:(NSInteger)section {
    NSArray *selectedArray = nil;
    NSArray *sectionZero = nil;
    NSArray *sectionOne = nil;

    if([self sorted]) {
        sectionZero = [self boysArray];
        sectionOne = [self girlsArray];
    } else {
        sectionZero = [self girlsArray];
        sectionOne = [self boysArray];
    }

    switch (section) {
        case 0:
            selectedArray = sectionZero;
            break;

        case 1:
            selectedArray = sectionOne;
            break;
    }

    return selectedArray;
}
```

11. To ensure the correct section header is displayed, create the `titleForSection` method:

```
- (NSString *)titleForSection:(NSInteger)section {
    NSString *title = nil;

    if([self sorted]) {
        title = [[self sectionsSorted] objectAtIndex:section];
    } else {
        title = [[self sections] objectAtIndex:section];
    }

    return title;
}
```

12. When the table view cell is selected, `prepareForSegue:sender:` is called before the view controller transition. Here you pass the selected table view cell and associated section header title to the detail view controller for display:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    NSIndexPath *indexPath = [[self tableView] indexPathForSelectedRow];
    NSInteger row = [indexPath row];
    NSInteger section = [indexPath section];

    NSArray *names = [self arrayForSection:section];

    NSString *headerTitle = [self titleForSection:section];
    NSString *name = [names objectAtIndex:row];

    DetailViewController *detailViewController = [segue
destinationViewController];
    [detailViewController setTitle:headerTitle];
    [detailViewController setDetailItem:name];
}
```

5. Run the application.
    1. Select the iPhone Simulator to run the application.
    2. Click the Run button from Xcode.
    3. When the application launches, the two sections and their contents are displayed. Tap the Toggle button and observe the sections swap.

Please select Lesson 13 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

14 Static Table Views

For table view applications pre-iOS 5, table view cells were displayed dynamically using a prototype table view cell that would be populated dynamically prior to being displayed in the table view.

The table view cell could be customized to provide a custom look to an application; however the architecture was still using default or custom UITableViewCell and was still a dynamic list based object organized by sections and related rows.

This all changed with the release of iOS 5.

TABLE VIEW TYPES

With iOS 5 using storyboards, table views are now designed using the following table view types:

- Static
- Dynamic prototype

Static

A static table view is designed directly through the interface editor. In addition to the table view cell itself, you can add the sections, headers, and footers directly into the Attributes Inspector. Because these are not dynamically produced, there is no data source and the table view cells are populated directly. A sample static table view is shown in Figure 14-1.

Dynamic Prototype

Dynamic prototype–based table views are also designed directly through the Interface Builder. They are easy to implement, and the prototype table view cell is similar to pre-iOS 5 custom table view cells. See Figure 14-2.

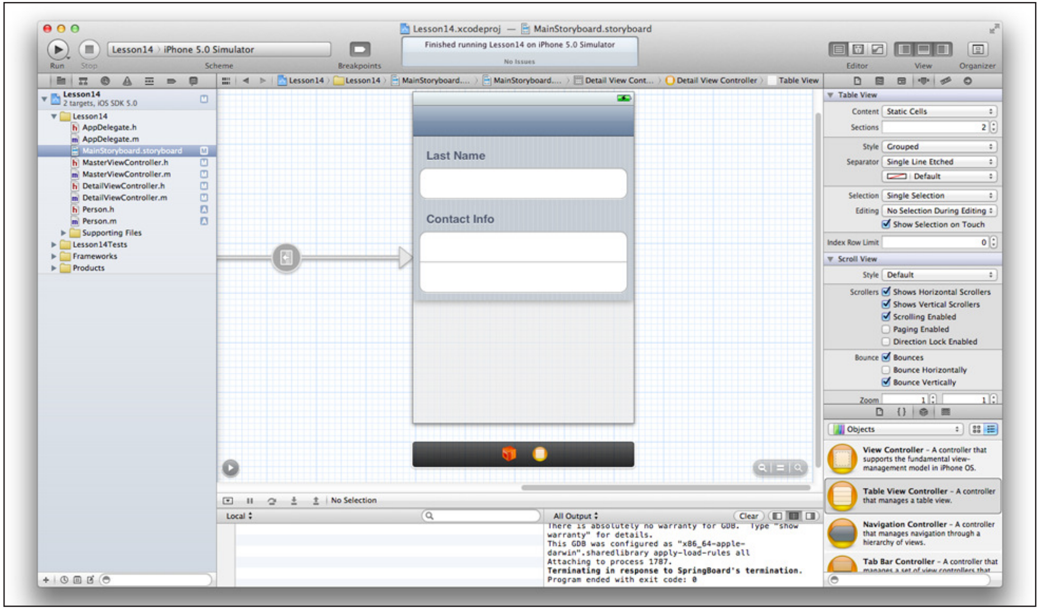


Table View

Content: Static Cells
Sections: 2
Style: Grouped
Separator: Single Line Etched
Default
Selection: Single Selection
Editing: No Selection During Editing
☑ Show Selection on Touch
Index Row Limit: 0

Scroll View

Style: Default
Scrollers: ☑ Shows Horizontal Scrollers
☑ Shows Vertical Scrollers
☑ Scrolling Enabled
☐ Paging Enabled
☐ Direction Lock Enabled
Bounce: ☑ Bounces
☐ Bounce Horizontally
☑ Bounce Vertically
Zoom: 1 1

View Controller – A controller that supports the fundamental view-management model in iPhone OS.

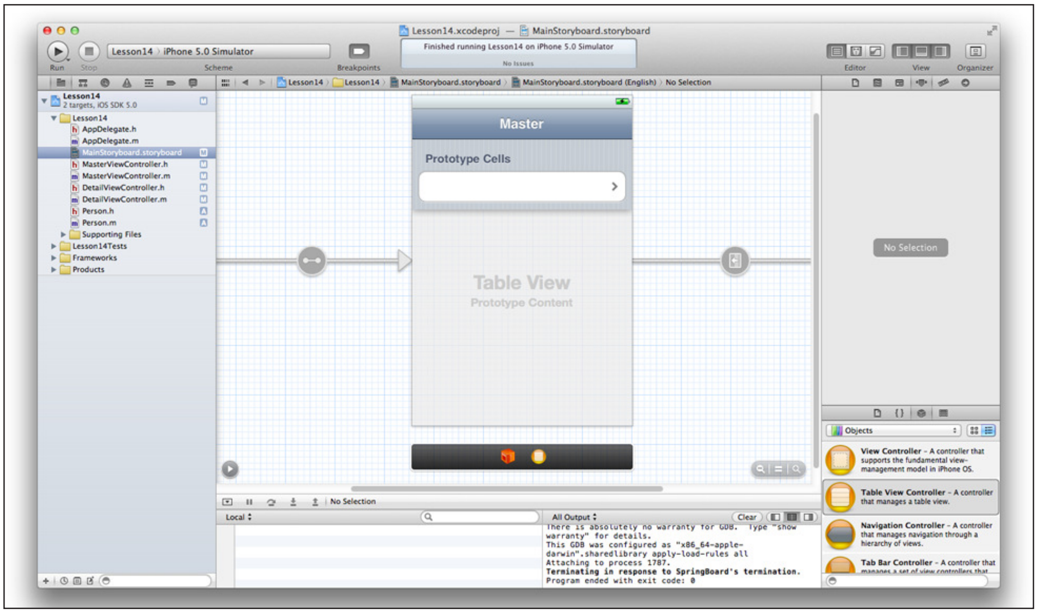
Table View Controller – A controller that manages a table view.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that

```
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "x86_64-apple-
darwin".sharedlibrary apply-load-rules all
Attaching to process 1787.
Terminating in response to SpringBoard's termination.
Program ended with exit code: 0
```

FIGURE 14-1



View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Table View Controller – A controller that manages a table view.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that

```
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "x86_64-apple-
darwin".sharedlibrary apply-load-rules all
Attaching to process 1787.
Terminating in response to SpringBoard's termination.
Program ended with exit code: 0
```

FIGURE 14-2

Prototype table view cells must have reuse identifiers declared. They are defined in the Attributes Editor under the Table View section labeled Identifier.

TABLE VIEW DESIGN

The design of the static table view is similar to the dynamic prototype style with the following design considerations:

- The designing of the cells are performed directly through the interface editor directly on the table view.
- Sections, headers, and footers are added via the Attributes Inspector.
- No data source outlet needs to be identified because the cells are directly referenced as outlets

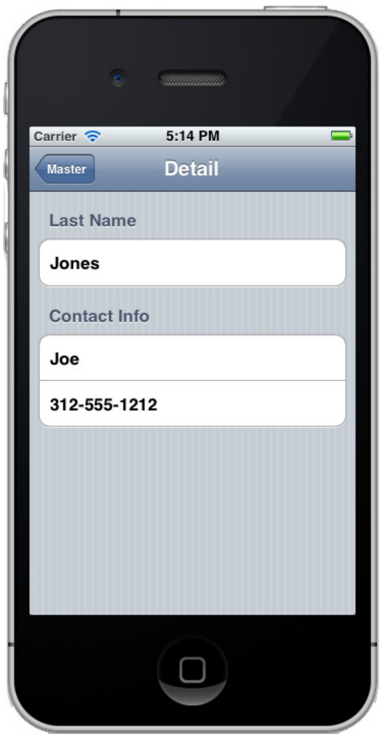


FIGURE 14-3

Display Considerations

When dynamic prototype table view cells are used, the amount of data to display is variable. Static table view cells, in contrast, display a known quantity of rows.

Data can still be grouped into sections, but the sections partition the related data to the same entity. For example, contact information can be grouped into last name and first name, phone number as shown in Figure 14-3.

TRY IT

In this Try It, you implement a Master-Detail Application that contains a list of contacts in a dynamic prototype table view. On selection of a specific contact, the contact detail is displayed in a static table view.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson14 folder in the download.*

Lesson Requirements

- Create an Xcode project using the Master-Detail Application template.
- Create a storyboard including root and detail view controllers.

- Implement a table view using a dynamic prototype and static content.
- Populate the header section of the table view.
- Implement the `prepareForSegue:sender` method in the root view controller to pass the contact values associated with the selected table view cell.
- Display the detail view controller with the passed values associated with the selected contact in a static table view.

Hints

- Because this application uses storyboards instead of `xib` files, remember to have the Use Storyboard option checked at project creation.
- One dictionary will be used for table view cell population using the keys for the cell title.
- The view controller transition will launch when a table view cell is selected, so the `performSegueWithIdentifier:sender` connection from the root view controller to the detail view controller will be between the button and the detail view controller.
- A custom `Person` object will be used to hold the contact information.

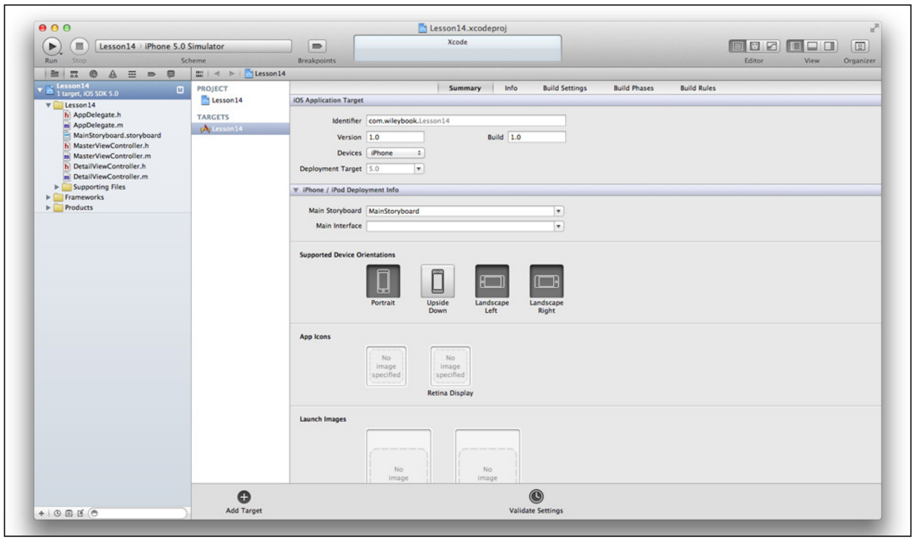
Step-by-Step

1. Create a Master-Detail Application.
    1. Launch Xcode.
    2. Create your new iOS project.
        a. To create a new project, select Create a New Xcode Project.
        b. On the left under iOS, select Application.
        c. Select Master-Detail Application from the template list and click Next.
        d. Choose the following options for your project:
            - **Product Name:** Lesson14
            - **Company Identifier:** com.wileybook
            - **Class Prefix:** leave blank
            - **Device Family:** iPhone
            - **Use Storyboard:** Checked
            - **Use Core Data:** Unchecked
            - **Use Automatic Reference Counting:** Checked
            - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

e. Select the location on your computer where the project will be saved and select Create.

f. Your Xcode project has been created as shown in Figure 14-4.



Identifier: com.wileybook.Lesson14
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0
Main Storyboard: MainStoryboard
Main Interface:

FIGURE 14-4

2. Design the user interface.

    1. On the left, select `MainStoryboard.storyboard`.

    2. On the right select the third button in the View section, to display the Utilities view as shown in Figure 14-5.

    3. To change the master table view cell to a dynamic prototype:

        a. Select the center of the table view right over the area that has Table View Static Content.

        b. Select the fourth icon, the Attributes Inspector, from the utilities toolbar and change the Content from Static Cells to Dynamic Prototype and Style from Plain to Group as shown in Figure 14-6.

        c. Select the table view cell and enter **Cell** for the Identifier in the Table View Cell section at the top of the Attributes Inspector.

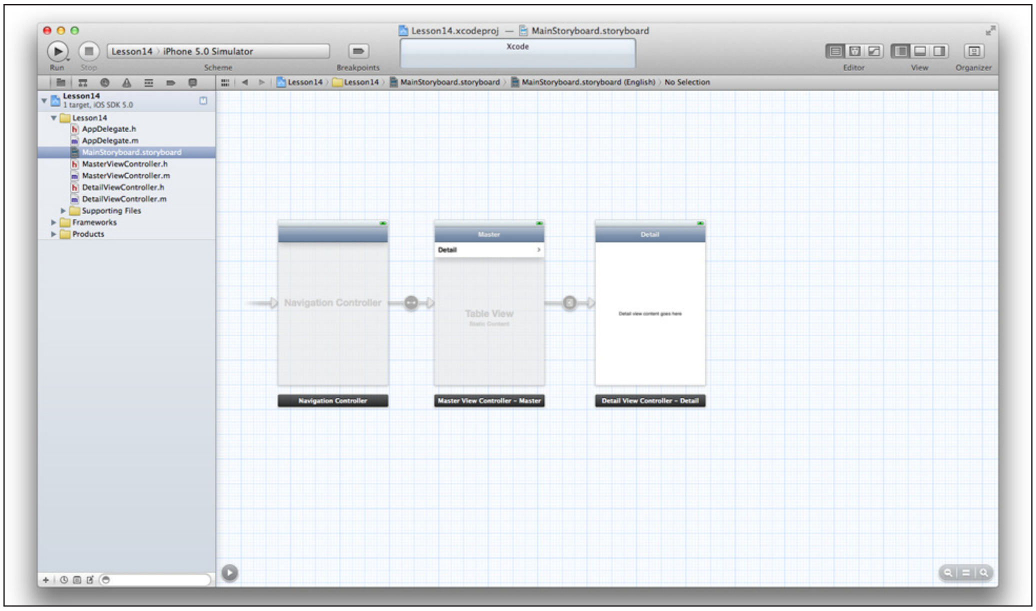


FIGURE 14-5

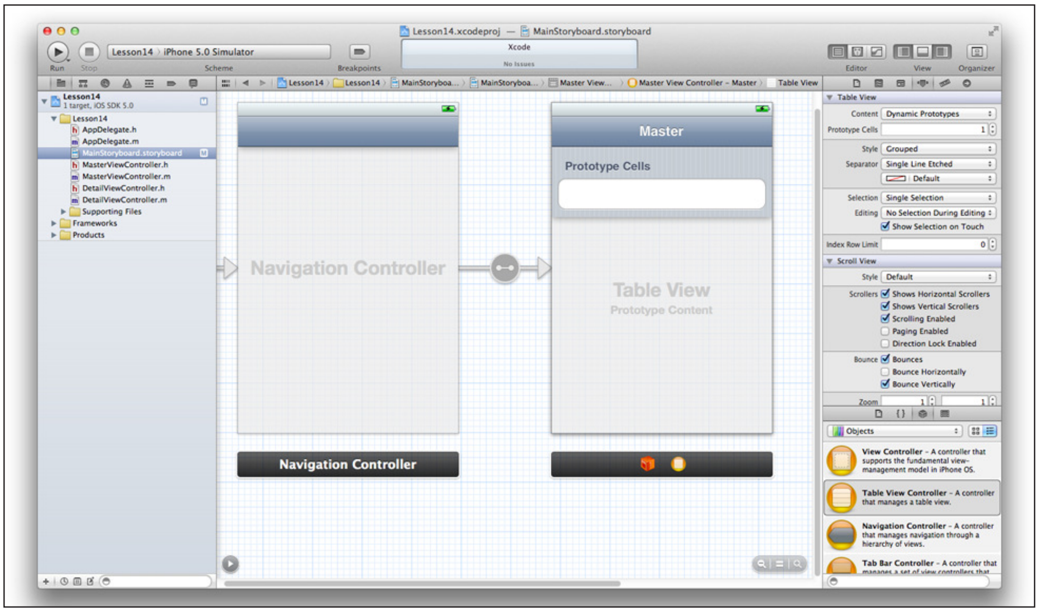


Table View

Content: Dynamic Prototypes
Prototype Cells: 1
Style: Grouped
Separator: Single Line Etched
Default
Selection: Single Selection
Editing: No Selection During Editing
[✓] Show Selection on Touch
Index Row Limit: 0

Scroll View

Style: Default
Scrollers: [✓] Shows Horizontal Scrollers
[✓] Shows Vertical Scrollers
[✓] Scrolling Enabled
[ ] Paging Enabled
[ ] Direction Lock Enabled
Bounce: [✓] Bounces
[ ] Bounce Horizontally
[✓] Bounce Vertically
Zoom: 1 1

Objects

View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Table View Controller – A controller that manages a table view.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that

FIGURE 14-6

4. To delete the detail view controller from the storyboard:

    a. Select the bottom bar of the detail view controller in the storyboard and tap the delete key to remove it.

> *It is important to make sure the default view controller is completely removed from the storyboard. Your detail view controller must be completely removed from the storyboard after this step.*

    b. Select the `DetailViewController.h` and `DetailViewController.m` files and delete and remove the files from the project.

5. To add the new detail view controller to the storyboard:

    a. Select the Lesson14 folder.

    b. Select File ➪ New ➪ New File from the Xcode menu.

    c. On the left under iOS, select Cocoa Touch.

    d. Select UIViewController from the template list and click Next.

    e. Choose the following options for your new file:

        - **Class:** DetailViewController
        - **Subclass of:** UITableViewController
        - **Deselect**: Targeted for iPad
        - **Deselect**: With XIB for user interface

    f. Click Next.

    g. Click Create to save the class in your project folder.

6. To add the detail table view controller to the storyboard:

    a. On the left, select `MainStoryboard.storyboard`.

    b. Drag a Table View Controller from the Object Library and add it to the storyboard.

    c. Select the Identity Inspector and enter **DetailViewController** for the class.

    d. Select the center of the table view right over the area that has Table View Static Content.

    e. Select the Attributes Inspector and change the Content from Dynamic Prototype to Static Cells, Sections from 1 to 2, and Style from Plain to Group.

    e. Select the frame next to the Section-1 group; change the Rows value from 3 to 1 and enter **Last Name** for the Header.

    f. Select the frame next to the Section-2 group; change the Rows value from 3 to 2 and enter **Contact Info** for the Header.

7. To create the outlets for the detail view controller table view cells:
   a. Select the Attributes Inspector to bring up the `DetailViewController.h` file.
   b. Select the first table view cell and control-drag to the interface source code just above the `@end`.
   c. Enter **lastName** for the outlet name and click Connect.
   d. Select the second table view cell and control-drag to the interface source code just above the `@end`.
   e. Enter **firstName** for the outlet name and click Connect.
   f. Select the third table view cell and control-drag to the interface source code just above the `@end`.
   g. Enter **phone** for the outlet name and click Connect.
   h. Select the Standard editor to hide the detail view controller.
8. To create the `Person` class:
   a. Select the Lesson14 folder.
   b. Select File ➪ New ➪ New File from the Xcode menu.
   c. On the left under iOS, select Cocoa Touch.
   d. Select Objective-C from the template list and click Next.
   e. Choose the following options for your new file:
      - **Class:** Person
      - **Subclass of:** NSObject
   f. Click Next.
   g. Click Create to save the class in your project folder.
9. Select File ➪ Save to save your project.

3. Make the connection for the navigation transition from the master detail view controller to the detail view controller.
   1. Select the table view cell in the master view controller and control-drag to the detail view controller below the table view cells. Select Push from Storyboard Segues list.
4. Modify the master view controller.
   1. Select `MasterViewController.h` and add the following above the `@end`:

```
@property (strong, nonatomic) NSDictionary *contacts;
```

   2. Select `MasterViewController.m` and add the following import:

```
#import "DetailViewController.h"
#import "Person.h"
```

3. Add the following synthesize variables right below the `@implementation` section:

   ```
   @synthesize contacts;
   ```

4. Uncomment the `viewDidLoad` method and add the following below `[super viewDidLoad]`:

   ```
   - (void)viewDidLoad
   {
       [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
       NSDictionary *dict = [[NSDictionary alloc] initWithObjectsAndKeys:
        [[Person alloc] initWithLastName:@"Jones"
                               firstName:@"Joe"
                                   phone:@"312-555-1212"], @"Jones",
        [[Person alloc] initWithLastName:@"Barnes"
                               firstName:@"Bill"
                                   phone:@"443-555-1212"], @"Barnes",
        [[Person alloc] initWithLastName:@"Smith"
                               firstName:@"Andy"
                                   phone:@"775-555-1212"], @"Smith",
                             nil];
       [self setContacts:dict];
   }
   ```

5. There is only one section to display the contacts. Complete the `numberOfSectionsInTableView:` method:

   ```
   - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
   {
       return 1;
   }
   ```

6. There is one row for each contact in the dictionary. Complete the `tableView:numberOfRowsInSection:` method:

   ```
   - (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
   {
       return [[[self contacts] allKeys] count];
   }
   ```

7. For each row, the last name for each contact in the list is displayed. Complete the `tableView:cellForRowAtIndexPath:` method:

   ```
   - (UITableViewCell *)tableView:(UITableView *)tableView
            cellForRowAtIndexPath:(NSIndexPath *)indexPath
   {
       static NSString *CellIdentifier = @"Cell";
         NSArray *keys = [[[self contacts] allKeys]
   sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
       NSString *key = [keys objectAtIndex:[indexPath row]];
       Person *person = [[self contacts] objectForKey:key];
       NSString *cellText = [person lastName];

       UITableViewCell *cell =
             [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
   ```

```
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
               initWithStyle:UITableViewCellStyleDefault
               reuseIdentifier:CellIdentifier];
    }

    // Configure the cell...
    [[cell textLabel] setText:cellText];

    return cell;
}
```

5. Modify the detail view controller.

   1. Select `DetailViewController.h` and add the following above the `@end`:

      ```
      @class Person;
      @interface DetailViewController : UITableViewController
      @property (strong, nonatomic) IBOutlet UITableViewCell *lastName;
      @property (strong, nonatomic) IBOutlet UITableViewCell *firstName;
      @property (strong, nonatomic) IBOutlet UITableViewCell *phone;
      @property (strong, nonatomic) Person *person;
      ```

   2. Select `DetailViewController.m` and add the following `import`:

      ```
      #import "DetailViewController.h"
      #import "Person.h"
      ```

   3. Add the following `synthesize` variables right below the `@implementation` section:

      ```
      @synthesize lastName;
      @synthesize firstName;
      @synthesize phone;
      @synthesize person;
      ```

   4. Two sections are displayed. The first section is the last name and the second section is the first name and phone number. Complete the `numberOfSectionsInTableView:` method:

      ```
      - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
      {
          return 2;
      }
      ```

   5. There is one row in the first section, and two rows in the second section. Complete the `tableView:numberOfRowsInSection:` method:

      ```
      - (NSInteger)tableView:(UITableView *)tableView
       numberOfRowsInSection:(NSInteger)section
       {
          NSInteger count = 0;

          switch (section) {
              case 0:
                  count = 1;
                  break;
      ```

```
        case 1:
            count = 2;
            break;

        default:
            break;
    }
    return count;
}
```

6. For each row, the corresponding value in the `Person` class is displayed. Complete the `tableView:cellForRowAtIndexPath:` method:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = nil;
    NSString *cellText = nil;

    switch ([indexPath section]) {
        case 0:
            cell = [self lastName];
            cellText = [[self person] lastName];
            break;

        case 1:
            switch ([indexPath row]) {
                case 0:
                    cell = [self firstName];
                    cellText = [[self person] firstName];
                    break;

                case 1:
                    cell = [self phone];
                    cellText = [[self person] phone];
                    break;

                default:
                    break;
            }
            break;

        default:
            break;
    }
    [[cell textLabel] setText:cellText];

    return cell;
}
```

6. Create the `Person` class.

1. Complete the `Person` class definition:

```
@interface Person : NSObject
@property (strong, nonatomic) NSString *lastName;
```

```
@property (strong, nonatomic) NSString *firstName;
@property (strong, nonatomic) NSString *phone;
- (id)initWithLastName:(NSString *)lastName
             firstName:(NSString *)firstName
                 phone:(NSString *)phone;
@end
```

2. Complete the `Person` class implementation:

```
@implementation Person
@synthesize lastName = _lastName;
@synthesize firstName = _firstName;
@synthesize phone = _phone;
- (id)initWithLastName:(NSString *)lastName
             firstName:(NSString *)firstName
                 phone:(NSString *)phone {
    [self setLastName:lastName];
    [self setFirstName:firstName];
    [self setPhone:phone];

    return self;
}
@end
```

7. Run the application.
    1. Select the iPhone Simulator to run the application.
    2. Click the Run button from Xcode.
    3. When the application launches, a list of contacts appears.
    4. Select a specific contact to display the details.

Please select Lesson 14 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

15

Tab Bars and Toolbars

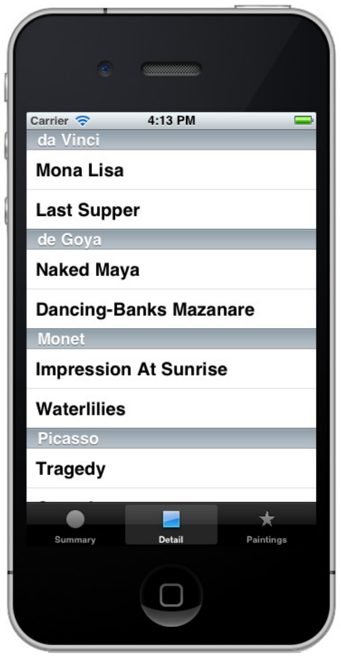
Lesson 12 illustrated navigation as one view controller pushing another view controller onto the navigation stack throughout the use of the navigation bar. Navigation bars arrange the presentation of data in a hierarchy by drilling down, and providing a path back to the root view.

This lesson presents two other navigation components:

- Tab bars
- Toolbars

TAB BARS

While not always followed in the real world of app development, Apple in its User Interface Guidelines proposes that the philosophy behind tab bar views is to provide a different perspective of the same set of data by using several view controllers, as shown in Figure 15-1.



da Vinci
Mona Lisa
Last Supper
de Goya
Naked Maya
Dancing-Banks Mazanare
Monet
Impression At Sunrise
Waterlilies
Picasso
Tragedy

FIGURE 15-1

Appearance Characteristics

Tab bars reside in a tab bar controller located at the bottom of the view, which manages the display of the multiple view controllers. The tab bar display consists of an icon and text to describe the perspective it represents. The tab itself has the ability to provide additional information that the applications may have to offer. The badge itself has the appearance of a red oval that will display a number or exclamation point. For example, when you may have any app updates from the AppStore the number of updates are displayed in a red oval.

The limitation for tab bars on the iPhone is that the tab bar cannot display more than five tabs at a time. If there are more than five tabs, the first four are displayed and the tab bar controller adds a More tab, which reveals a list of any additional tabs.

The iPad can display more than five tabs due to the larger view area the device has.

Usage Guidelines

Apple suggests the following guidelines when using a tab bar in your application:

- The tab should not give users control over any components in the displayed view, because that is the purpose of the toolbar.
- The tab bar should organize the information from an overall application functionality level.
- If a tab represents a function that is not available in your application, disable the tab rather than remove it.
- To indicate a change within an application, the badge can be updated to reflect that change. For example, if there is a tab to indicate updates, displaying a badge with the number would inform the user properly.
- Avoid overcrowding the iPad tab bar with too many tabs.
- Keep the same order of tabs in either orientation, landscape or portrait, on the iPad.

TOOLBARS

On an iPhone, toolbars are always placed at the bottom of the view, and contain bar button items that provide a number of options that act on the current view context, as shown in Figure 15-2. However, on an iPad, toolbars appear on the top of the view.

Appearance Characteristics

Toolbars are spaced equally across the width of the toolbar. There are also flexible and fixed space bar button items to aid in the proper placement.

Unlike tab bars, the number of bar button items can change from view to view, because the items are always directly related to the view that is currently displayed.

Usage Guidelines

Apple suggests the following guidelines when using a toolbar in your application:

- The button items should represent command functions that would be used on the current view.

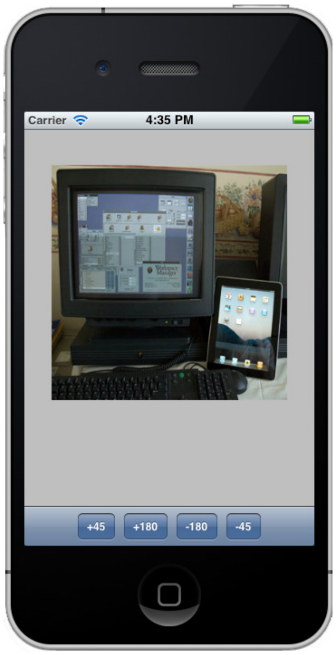


FIGURE 15-2

- Do not crowd the toolbar with too many button items, and make the button item at least 44×44 points in size.
- Use system-provided toolbar items whenever possible. For example, Done, Pause, and Play are commonly understood functions.
- Avoid mixing styles. For example, do not use bordered and borderless buttons on the same toolbar.

XIB-BASED XCODE 4.2 CHANGES

In Xcode versions prior to 4.2, the tab bar application template provided a `MainWindow.xib` file that contained the tab bar controller. New tabs were added and associated with view controllers through the Interface Builder.

With the introduction of Xcode 4.2, the `MainWindow.xib` file no longer exists, and the tab bar controller is assembled in the `AppDelegate` class.

The `AppDelegate.h` file contains the definition of the tab bar controller:

```
#import <UIKit/UIKit.h>
@interface AppDelegate : UIResponder <UIApplicationDelegate, UITabBarControllerDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) UITabBarController *tabBarController;
@end
```

In the `application:didFinishLaunchingWithOptions:` method of `AppDelegate.m`, the tab bar controller is initialized and associated view controllers are assigned, as shown in the following code:

```
- (BOOL)application:(UIApplication *)application
          didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    UIViewController *viewController1 = [[FirstViewController alloc]
                                     initWithNibName:@"FirstViewController"
                                              bundle:nil];
    UIViewController *viewController2 = [[SecondViewController alloc]
                                     initWithNibName:@"SecondViewController"
                                              bundle:nil];
    self.tabBarController = [[UITabBarController alloc] init];
    self.tabBarController.viewControllers =
             [NSArray arrayWithObjects:viewController1, viewController2, nil];
    self.window.rootViewController = self.tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

The tab titles and images are initialized in each of the view controllers `initWithNibName:bundle:` method, as shown here:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
                bundle:(NSBundle *)nibBundleOrNil
{
        self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
        if (self) {
            self.title = NSLocalizedString(@"First", @"First");
            self.tabBarItem.image = [UIImage imageNamed:@"first"];
        }
        return self;
    }
```

Remember, you have to do this only for `xib`-based tab bar applications using Xcode 4.2 or greater.

TRY IT

In this Try It, you implement a tabbed application for the iPhone that has a list of famous artists and some of their popular works. The application contains three tabs. The first tab is a summary page that indicates how many artists are available. The second tab presents each of the artists and a few of their works in a grouped table view, and the third tab lists all the paintings. When a painting is selected, an alert is presented that displays the artist and the painting.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson15 folder in the download.*

Lesson Requirements

- Create an Xcode project for iPhone using the Tabbed Application template.
- Create a storyboard including the tab bar controller and three separate view controllers associated with their respective tabs.
- The first button displays a summary page indicating how many artists are available for the detail viewing, the second button displays details about the artists and their works, and the third button displays all the paintings that are listed.

Hints

- Because this application uses storyboards instead of `.xib` files, remember to have the Use Storyboard option checked at project creation.
- Select iPhone from the Device Family list.
- Create two images, `third.png` and `third@2x.png`, with dimensions 30 × 30 72 dpi and 60 × 60 144 dpi, respectively, for the images used on the third tab.

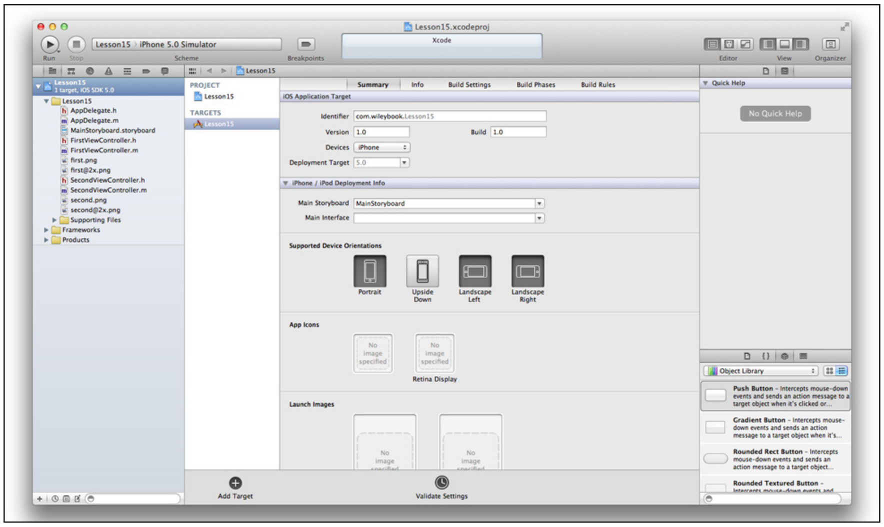
The third.png and third@2x.png images are supplied in the Try It for Lesson 15 on the DVD.

Step-by-Step

1. Create a new application using the Tabbed Application template.
    1. Launch Xcode.
    2. Create your new iOS project.
        a. To create a new project, select Create a New Xcode Project from the initial Welcome to Xcode window.
        b. On the left under iOS, select Application.
        c. Select Tabbed Application from the template list and click Next.
        d. Choose the following options for your project:
            - **Product Name:** Lesson15
            - **Company Identifier:** com.wileybook
            - **Class Prefix:** leave blank
            - **Device Family:** iPhone
            - **Use Storyboard:** Checked
            - **Use Automatic Reference Counting:** Checked
            - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

        e. Select the location on your computer where the project will be saved and click Create.
        f. Your Xcode project has been created as shown in Figure 15-3.
2. Create the `ThirdViewController` class.
    1. To add the third view controller to the storyboard:
        a. Select the Lesson15 folder.
        b. Select File ➪ New ➪ New File from the Xcode menu.
        c. On the left under iOS, select Cocoa Touch.
        d. Select UIViewController from the template list and click Next.
        e. Choose the following options for your new file:
            - **Class:** ThirdViewController
            - **Subclass of:** UITableViewController
            - **Targeted for iPad:** Unchecked
            - **With XIB for user interface:** Unchecked



Identifier com.wileybook.Lesson15
Version 1.0
Build 1.0
Devices iPhone
Deployment Target 5.0
Main Storyboard MainStoryboard
Main Interface

FIGURE 15-3

f. Click Next.

g. Click Create to save the class in your project folder.

2. To add the third view controller to the storyboard:

   a. Select the Lesson15 folder.

   b. Control-click and select Add Files to Lesson15 to add the `third.png` and `third@2x.png` images from the DVD.

3. Design the user interface.

   1. Design the `FirstViewController`.

      a. On the left, select `MainStoryboard.storyboard`.

      b. In the first view controller's view, select both the label and text view and delete them.

      c. Drag a `Label` from the list on the lower right and place it in the left center of the view. Double-click the Title and enter **Number of Artists:**.

      d. Drag a `Label` from the object list and place it just to the right of the label you added in Step c.

      e. Select the tab bar at the bottom of the view, select the Attributes Inspector, and enter **Summary** for the Title.

In the Attributes Inspector right below the Title you just entered, notice the entry for Image. This is where you would add your custom image for the tab. For this Try It, the default `first.png` *image is used.*

   2. Design the `SecondViewController`.
      a. In the second view controller's view, select both the label and text view and delete them.
      b. Drag a `Table View` from the list and place it on the entire view.
      c. Click the table view cell and from the Attributes Inspector, enter **Cell** for the Identifier.
   3. Design the `ThirdViewController`.
      a. Drag a `Table View Controller` from the object list on the lower right onto the storyboard below the tab bar controller view.
      b. From the Identity Inspector choose the `ThirdViewController` class for the Custom Class.
      c. Click the table view cell and from the Attributes Inspector, enter **Cell** for the Identifier.
   4. Select File ➪ Save to save your project.
4. Make the connections to the outlets and actions.
   1. To make the connections on the `FirstViewController`:
      a. On the storyboard, select the status bar at the top of the first view controller.
      b. From the Editor section on the top right, select the Assistant Editor.
      c. Select the second label you added. It still has the title Label. Control-drag to the interface source code just above the `@end`.
      d. Select Outlet for Connection, enter **artistCountLabel** for the outlet name, and click Connect.
   2. To make the connections on the `SecondViewController`:
      a. On the storyboard, select the status bar at the top of the popover view controller.
      b. Select in the middle of the table view and control-drag to the interface source code just above the `@end`.
      c. Select Outlet for Connection, enter **artistTableView** for the outlet name, and click Connect.
      d. Select the table view and control-drag to the Second View Controller icon on the bar just below the tab bar in the view. Release the mouse and click the dataSource outlet in the popup that is displayed (Figure 15-4).
      e. Repeat step d and click the delegate outlet in the popup that is displayed.

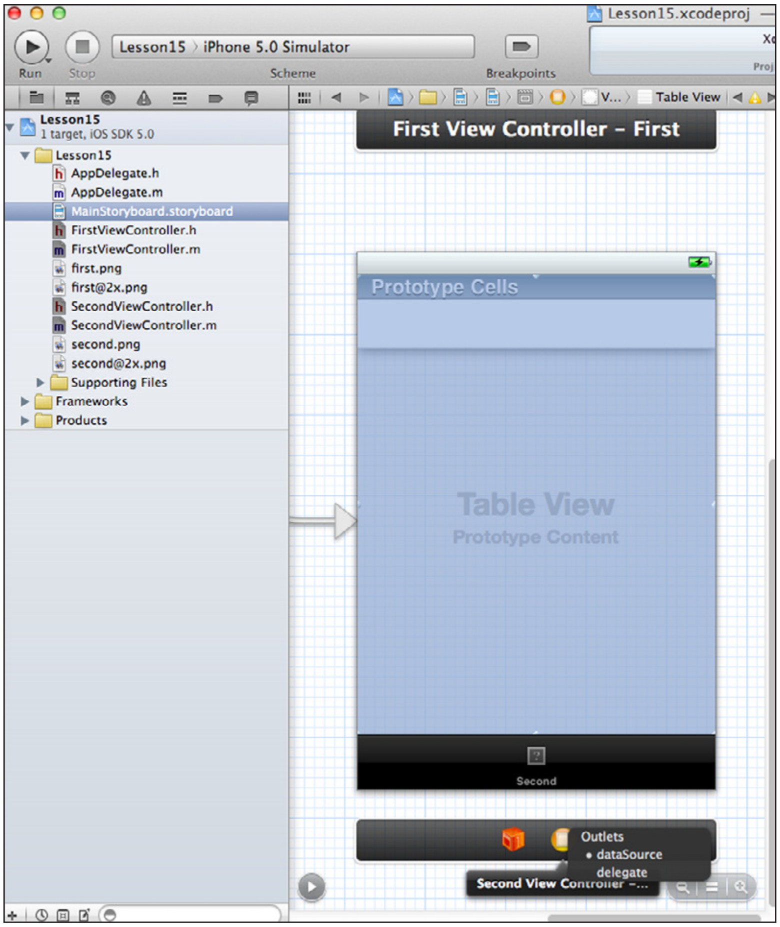


FIGURE 15-4

3. To make the connections on the `ThirdViewController`:

   a. From the main tab bar controller's view, control-drag to the center of the third view controller, release the mouse, and click Relationship – view Controllers from the options displayed.

   b. Select the tab bar on the bottom. From the Attributes Inspector, enter **Paintings** for the Title and select `third.png` for the Image.

4. Select File ➪ Save to save your project.

5. Modify the `AppDelegate` class.

   1. Select the first button for the Standard Editor in the Editors section, on the upper right of Xcode, and modify the `AppDelegate.h` file to look like the following:

```
#import <UIKit/UIKit.h>
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

```
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) NSDictionary *names;
- (NSDictionary *)createDictionaryOfArtists;
@end
```

2. Add the following synthesize variables right below the `@implementation` section to the `AppDelegate.m` file:

```
@synthesize names = _names;
```

3. Modify the `application: didFinishLaunchingWithOptions:` method to look like the following:

```
- (BOOL)application:(UIApplication *)application
          didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self setNames:[self createDictionaryOfArtists]];
    return YES;
}
```

4. Add the `createDictionaryOfArtists:` method at the bottom, before the `@end`, to look like the following:

```
- (NSDictionary *)createDictionaryOfArtists {
    NSDictionary *artists = [NSDictionary dictionaryWithObjectsAndKeys:
          [NSArray arrayWithObjects:@"Mona Lisa", @"Last Supper", nil],
                                    @"da Vinci",
          [NSArray arrayWithObjects:@"Self-Portrait", @"Starry Night", nil],
                                    @"van Gogh",
          [NSArray arrayWithObjects:@"Tragedy", @"Guernica", nil], @"Picasso",
          [NSArray arrayWithObjects:@"Naked Maya", @"Dancing-Banks Mazanare",
                                    nil], @"de Goya",
          [NSArray arrayWithObjects:@"Impression At Sunrise", @"Waterlilies",
                                    nil], @"Monet", nil];
    return artists;
}
```

6. Modify the `FirstViewController` class.

1. Modify the `FirstViewController.h` file to look like the following:

```
#import <UIKit/UIKit.h>
@interface FirstViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *artistCountLabel;
@end
```

2. Add the following imports to the `FirstViewController.m` file:

```
#import "AppDelegate.h"
```

3. Add the following synthesize variables right below the `@implementation` section:

```
@synthesize artistCountLabel = _artistCountLabel;
```

4. In the `viewDidLoad` method, add the following below `[super viewDidLoad];`

```
AppDelegate *appDelegate =
   (AppDelegate *)[[UIApplication sharedApplication] delegate];
[[self artistCountLabel] setText:[NSString stringWithFormat:@"%d",
   [[[appDelegate names] allKeys] count]]];
```

7. Modify the `SecondViewController` class.

    1. Modify the `SecondViewController.h` file to look like the following:

    ```
    #import <UIKit/UIKit.h>
    @interface SecondViewController : UIViewController
                        <UITableViewDelegate, UITableViewDataSource> {
    }
    @property (strong, nonatomic) NSDictionary *artists;
    @property (weak, nonatomic) IBOutlet UITableView *artistTableView;
    @end
    ```

> *The addition of* `<UITableViewDelegate, UITableViewDataSource>` *to the interface declaration indicates that the second table view controller is going to implement the table view delegate and data source methods, allowing for table view processing.*

    2. Add the following imports to the `SecondViewController.m` file:

    ```
    #import "AppDelegate.h"
    ```

    3. Add the following synthesize variables right below the `@implementation` section:

    ```
    @synthesize artists = _artists;
    ```

    4. In the `viewDidLoad` method and add the following below `[super viewDidLoad];`

    ```
    AppDelegate *appDelegate =
              (AppDelegate *)[[UIApplication sharedApplication] delegate];
    [self setArtists:[appDelegate names]];
    ```

    5. To let the table view know how many sections there are, add the following method:

    ```
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
    {
        return [[[self artists] allKeys] count];
    }
    ```

    6. To let the table view know how many rows per section there are, add the following method:

    ```
    - (NSInteger)tableView:(UITableView *)tableView
          numberOfRowsInSection:(NSInteger)section
    {
        NSArray *keys = [[[self artists] allKeys]
          sortedArrayUsingSelector:
          @selector(localizedCaseInsensitiveCompare:)];
       NSString *key = [keys objectAtIndex:section];
       return [[[self artists] objectForKey:key] count];
    }
    ```

    7. Each artist is the key to the dictionary, which is the section group. For each artist, there is a list of paintings, which is the row of the section. The `tableview:cellForRowAtIndexPath:` method displays the paintings for each artist:

    ```
    - (UITableViewCell *)tableView:(UITableView *)tableView
            cellForRowAtIndexPath:(NSIndexPath *)indexPath
    ```

```
{
    NSArray *keys = [[[self artists] allKeys]
      sortedArrayUsingSelector:
      @selector(localizedCaseInsensitiveCompare:)];
    NSString *key = [keys objectAtIndex:[indexPath section]];
    NSArray *paintings = [[self artists] objectForKey:key];
    NSString *cellText = [paintings objectAtIndex:[indexPath row]];

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
      [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
      if (cell == nil) {
          cell = [[UITableViewCell alloc]
               initWithStyle:UITableViewCellStyleDefault
               reuseIdentifier:CellIdentifier];
          [cell setSelectionStyle:UITableViewCellSelectionStyleNone];
      }

    [[cell textLabel] setText:cellText];

    return cell;
}
```

8. Each section's heading is displayed using the `tableView:titleForHeaderInSection:` method and is the last name of the artist:

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
NSArray *keys = [[[self artists] allKeys]
    sortedArrayUsingSelector:
    @selector(localizedCaseInsensitiveCompare:)];
return [keys objectAtIndex:section];
}
```

9. Because no navigation occurs when the table view cell is tapped, the cell is simply deselected to restore the original color. Add the following table view delegate method:

```
- (void)tableView:(UITableView *)tableView
            didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

8. Modify the `ThirdViewController` class.

   1. Modify the `ThirdViewController.h` file to look like the following:

```
#import <UIKit/UIKit.h>
@interface ThirdViewController : UITableViewController
@property (strong, nonatomic) NSArray *paintings;
- (NSArray *)getPaintingsFromDictionary:(NSDictionary *)artists;
- (void)alert:(NSString *)aMessage;
@end
```

   2. Add the following imports to the `ThirdViewController.m` file:

```
#import "AppDelegate.h"
```

3. Add the following synthesize variables right below the `@implementation` section:

```
@synthesize paintings = _paintings;
```

4. In the `viewDidLoad` method and add the following below `[super viewDidLoad];`

```
   AppDelegate *appDelegate = (AppDelegate *)[[UIApplication
sharedApplication] delegate];
   [self setPaintings:[self getPaintingsFromDictionary:
      [appDelegate names]]];
```

5. To let the table view know how many sections there are, add the following method:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
```

6. To let the table view know how many rows per section there are, add the following method:

```
- (NSInteger)tableView:(UITableView *)tableView
      numberOfRowsInSection:(NSInteger)section
{
    return [[self paintings] count];
}
```

7. Each string in the paintings array is a | character delimited string. The two parts are the painting name and the artist. The `tableview: cellForRowAtIndexPath:` method displays the paintings for each artist by parsing on the | character and selecting the painting:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *rec = [[[self paintings]
          sortedArrayUsingSelector:
          @selector(localizedCaseInsensitiveCompare:)]
            objectAtIndex:[indexPath row]];
    NSString *cellText = [[rec componentsSeparatedByString:@"|"]
                                             objectAtIndex:0];

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
         dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
               initWithStyle:UITableViewCellStyleDefault
               reuseIdentifier:CellIdentifier];
        [cell setSelectionStyle:UITableViewCellSelectionStyleNone];
    }

    [[cell textLabel] setText:cellText];

    return cell;
}
```

8. Because no navigation occurs when the table view cell is tapped, the cell's contents are displayed in an alert. Add the following table view delegate method:

```
- (void)tableView:(UITableView *)tableView
                    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    NSString *rec = [[[self paintings]
       sortedArrayUsingSelector:
       @selector(localizedCaseInsensitiveCompare:)]
          objectAtIndex:[indexPath row]];
    NSString *msg = [NSString stringWithFormat:@"%@\n%@",
     [[rec componentsSeparatedByString:@"|"] objectAtIndex:0],
     [[rec componentsSeparatedByString:@"|"] objectAtIndex:1]];

    [self alert:msg];
}
```

9. A utility method, `getPaintingsFromDictionary`, extracts the paintings from the main dictionary:

```
- (NSArray *)getPaintingsFromDictionary:(NSDictionary *)artists {
    NSMutableArray *paintingList = [NSMutableArray array];
    NSArray *keys = [artists allKeys];
    for(NSString *key in keys) {
        NSArray *list = [artists objectForKey:key];
        for(NSString *painting in list) {
            NSString *rec = [NSString stringWithFormat:@"%@|%@",
             painting, key];
            [paintingList addObject:rec];
        }
    }
    return paintingList;
}
```

10. The source for displaying the painting and artist combo is as follows:

```
- (void)alert:(NSString *)aMessage {
  UIAlertView *alert = [[UIAlertView alloc]
   initWithTitle:@"Lesson 15"
       message:aMessage
       delegate:self
   cancelButtonTitle:nil
   otherButtonTitles:@"OK", nil];
   [alert show];
}
```

9. Run the application.
    1. Select the iPhone Simulator to run the application.
    2. Click the Run button from Xcode.
    3. When the application launches, the first view controller indicates there are five artists available for display.

4. Select the Detail tab to display details about the artist and the paintings attributed to them.
5. Select the Paintings tab to display a list of all the paintings.
6. Select a painting to display the painting and artist in an alert view.

Please select Lesson 15 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

16 Creating Page-Based Applications

Apple has introduced a new view controller class to make it easy for you to create page-based applications. Applications typical to this category are e-book/magazine readers. These applications typically present content one page at a time and allow a user to flick through pages by swiping across the screen. With iOS 5 not only do you now have the page view controller class to help you create such applications, but also a new application template specifically suited to this type of application.

In this lesson you learn to use the new Page-Based Application template to create page-based applications.

THE PAGEVIEWCONTROLLER CLASS

Whether you decide to use the Page-Based Application template or an empty project to start from, you must use an instance of the new `PageViewController` class in your application. A page view controller allows users to navigate between view controllers using a specific transition. Navigation between pages generally occurs when the user performs a swipe gesture, although it can also be done programmatically.

The page view controller works in tandem with a data source and a delegate object, which must implement the `UIPageViewControllerDataSource` and `UIPageViewControllerDelegate` protocols, respectively. The data source object is called in response to gesture-based page navigation. If the value for this property is `nil`, gesture-based navigation is disabled.

Instantiation

A page view controller is typically instantiated in the `viewDidLoad` method of an existing view controller class. The view controller acts as a container for the page view controller and is often referred to as the “root” view controller. After the page view controller is instantiated and set up properly, it is added into the view hierarchy of the application by using the `addSubView:` method of the root view controller object’s view.

To instantiate a page view controller, use the `initWithTransitionStyle:navigationOrientation:options:` method as follows:

```
UIPageViewController* pageViewController = [[UIPageViewController alloc]
initWithTransitionStyle:UIPageViewControllerTransitionStylePageCurl
navigationOrientation:UIPageViewControllerNavigationOrientationHorizontal
options:nil];
```

The first parameter to this method is the transition style to be used for page turns. At the moment, the only allowed value for this parameter is `UIPageViewControllerTransitionStylePageCurl`. The second parameter controls the orientation of the transition; this can be either horizontal or vertical. The corresponding values are `UIPageViewControllerNavigationOrientationHorizontal` and `UIPageViewControllerNavigationOrientationVertical`, respectively. Figure 16-1 shows the difference between the transition orientations.

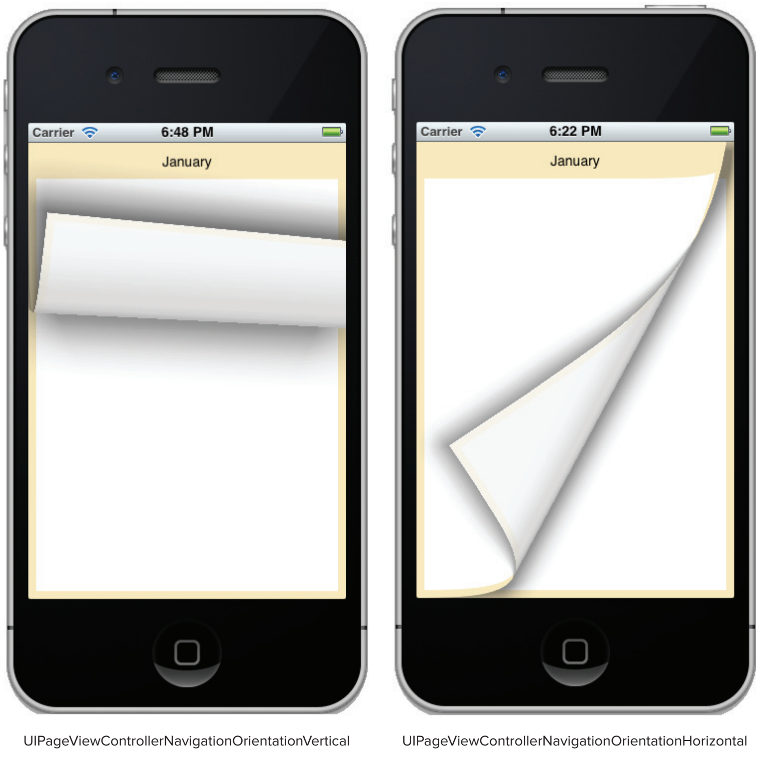

Carrier
6:48 PM
January
Carrier
6:22 PM
January
UIPageViewControllerNavigationOrientationVertical
UIPageViewControllerNavigationOrientationHorizontal

FIGURE 16-1

The third parameter is a dictionary of options. For the purposes of this lesson, this can be set to `nil`.

Delegate and Data Source

The `delegate` and `dataSource` properties of the page view controller are set up next. These must refer to objects that implement the `UIPageViewDataSource` and `UIPageViewDelegate` protocols, respectively.

The `UIPageViewControllerDataSource` protocol defines two mandatory methods that are used to provide the view controller for the previous and next pages:

```
- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController
viewControllerBeforeViewController:(UIViewController *)viewController;

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController
viewControllerAfterViewController:(UIViewController *)viewController;
```

These methods are called only in response to gesture-initiated transitions and can return `nil` to indicate that no more pages exist in the given direction.

The `UIPageViewControllerDelegate` protocol defines a couple of optional methods. The one you are most likely to implement is:

```
-(void)pageViewController:(UIPageViewController *)pageViewController
       didFinishAnimating:(BOOL)finished
       previousViewControllers:(NSArray *)previousViewControllers
       transitionCompleted:(BOOL)completed;
```

This method is called when a gesture-initiated page turn ends. The first parameter to this method is a reference to the page view controller object. The second parameter is a Boolean variable that indicates if the page turn animation has completed. The third parameter is an array of view controllers that were visible before the start of the transition, and the fourth parameter is another Boolean variable that is set to `YES` to indicate that the user completed the page turn gesture.

Preparing the Initial Page

After having instantiated the page view controller object and set up its delegate and data source, you must provide an array that contains an initial set of view controller objects for it to manage using the `setViewControllers:direction:animated:completion:` method.

The first parameter to this method is an array of view controller objects. Typically you provide an array with a single view controller object representing the first view controller, and subsequent view controllers are provided by the data source. In some situations where two pages are visible side-by-side, you will provide two view controllers in this array,

The second parameter indicates the direction of navigation; this is usually set to `UIPageViewControllerNavigationDirectionForward` but can also be `UIPageViewControllerNavigationDirectionReverse`.

The third parameter is a Boolean value that specifies if the transition should be animated. The fourth parameter is a block handler that is called when the animation has completed. When you are using this method to provide the initial set of view controllers, these are typically set to `NO` and `NULL`, respectively.

Thus, if `startingViewController` was an instance of a `UIViewController` subclass, a typical call to the `setViewControllers:direction:animated:completion:` method would be made in the `viewDidLoad:` method of the class that contains the page view controller instance, and would resemble the following:

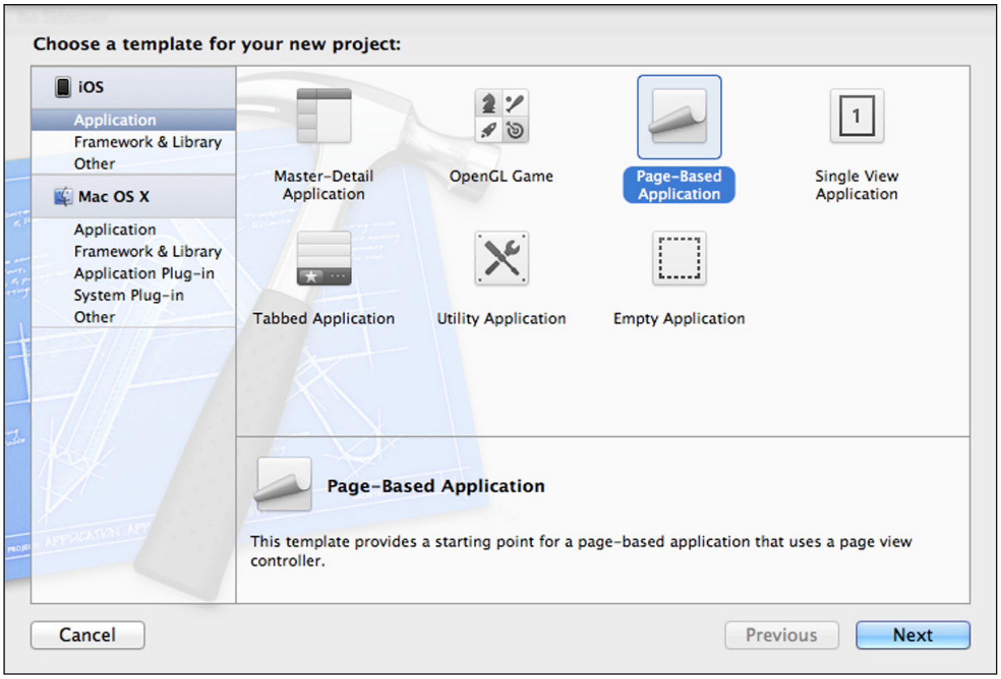
```
NSArray *viewControllers = [NSArray arrayWithObject:startingViewController];

[pageViewController setViewControllers:viewControllers
                    direction:UIPageViewControllerNavigationDirectionForward
                    animated:NO
                    completion:NULL];
```

To insert the page view controller instance into the application's view hierarchy, use the `addSubView:` method of the root view controller's view toward the end of the `viewDidLoad:` method as follows:

```
[self.view addSubview:pageViewController.view];
```

THE PAGE-BASED APPLICATION TEMPLATE

To help you get started building page-based applications, Apple has provided a new project template that includes a page view controller, data source, and delegate objects. Creating a page-based application using the new template is a simple matter of selecting the Page-Based Application template when creating a new project (Figure 16-2).



Page-Based Application

This template provides a starting point for a page-based application that uses a page view controller.

FIGURE 16-2

The template consists of several classes and a storyboard with two scenes. If you run the application in the iOS Simulator, you will see something similar to Figure 16-3. If you swipe across the screen you will be able to turn to the next/previous pages.

The main classes involved in this template are listed in Table 16-1. This table assumes that no class prefix was used while creating a project.

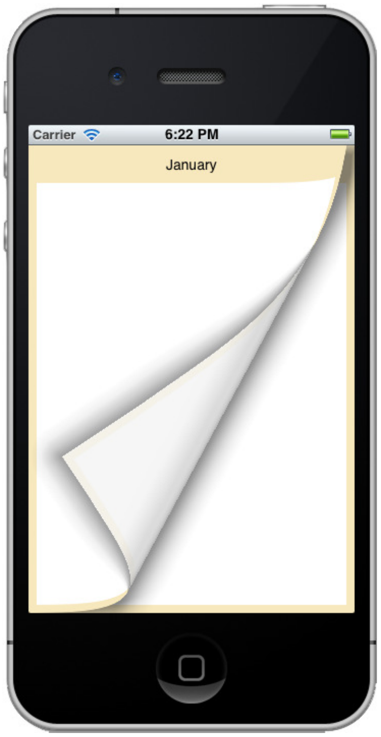


FIGURE 16-3

TABLE 16-1: Page-Based Application Template Classes

CLASS	DESCRIPTION
`AppDelegate`	The application delegate object.
`RootViewController`	Primary view controller, which contains an instance of a page view controller.
`DataViewController`	A view controller class, whose instances represent individual pages managed by the page view controller.
`ModelController`	A class that contains the data that is displayed in each page and is also responsible for instantiating `DataViewController` objects.

This template provides a clear separation between the data that appears on each page and the view that represents the actual page. Figure 16-4 depicts the various classes within this template and their relationship to one another.

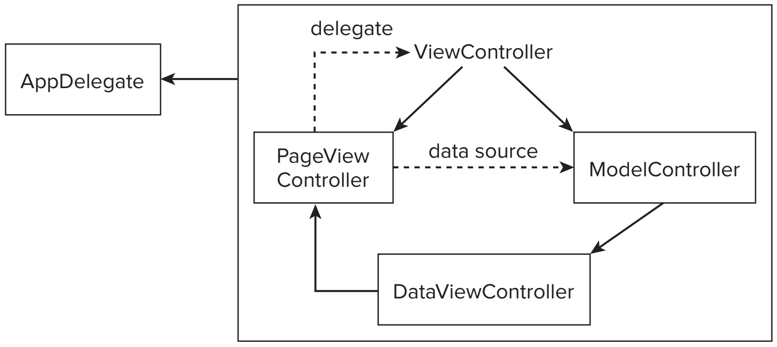


FIGURE 16-4

The primary view controller of the application is called `RootViewController` and contains an instance of the `UIPageViewController` class, an instance of the `ModelController` class, and also acts as the delegate for the page view controller. The boilerplate code to instantiate the page view controller is found in the `viewDidLoad` method.

Individual pages managed by the page view are instances of the `DataViewController` class. The interface of this class is very simple and contains a single `UILabel` outlet and a strong reference to a data object:

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

@interface DataViewController : UIViewController
@property (strong, nonatomic) IBOutlet UILabel *dataLabel;
@property (strong, nonatomic) id dataObject;
@end
```

The scene corresponding to the `DataViewController` class in the storyboard is shown in Figure 16-5. When you customize the template, you will almost certainly delete the contents of this scene and the corresponding `UILabel` outlet from the `DataViewController` class.

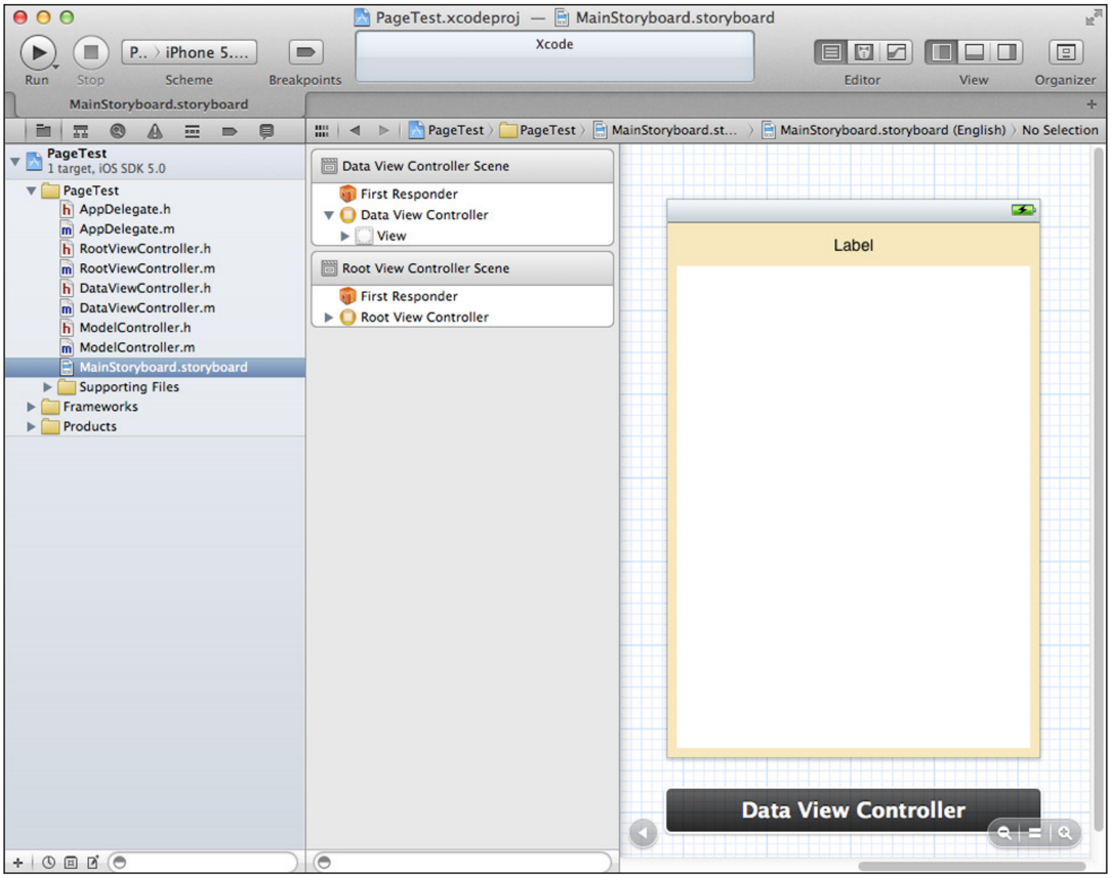


FIGURE 16-5

DataViewController instances are created by an instance of the ModelController class. The ModelController class encapsulates the data that is to be displayed on each page and also acts as the data source for the page view controller instance. The ModelController class implements the UIPageViewControllerDataSource protocol and provides a couple of additional methods:

```
- (NSUInteger)indexOfViewController:(DataViewController *)viewController
```

and

```
- (DataViewController *)viewControllerAtIndex:(NSUInteger)index
                        storyboard:(UIStoryboard *)storyboard
```

The first of these, indexOfViewController: is used to retrieve an integer page index from a given DataViewController instance. The second method is used to retrieve a DataViewController instance for a particular page index.

The data for each page is contained in an NSArray instance within the ModelController class called _pageData. The application template populates this array with month names in the init method:

```
- (id)init
{
    self = [super init];
    if (self) {
        // Create the data model.
        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        _pageData = [[dateFormatter monthSymbols] copy];
    }
    return self;
}
```

In your implementation, you are likely to populate the _pageData array differently, perhaps with instances of your own data objects. The ModelController class also defines a read-only property called pageData that allows access to the _pageData array:

```
@property (readonly, strong, nonatomic) NSArray *pageData;

@synthesize pageData = _pageData;
```

TRY IT

In this Try It, you create a simple flip book using the new Page-Based Application template. The app consists of five pages with an image on each page. You can flip through pages by swiping your finger horizontally across the screen.

Lesson Requirements

- ➤ Launch Xcode.
- ➤ Create a new project based on the Page-Based Application template.
- ➤ Modify the storyboard with Interface Builder.
- ➤ Import image resources into the project.
- ➤ Delete existing elements from one of the scenes in the storyboard.

- Add an image view to one of the scenes in the storyboard.
- Create an outlet using the assistant editor.
- Update the `Lesson16ModelController` class.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. You can find them in the Lesson 16 folder in the download.

Hints

- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.

Step-by-Step

1. Create a Page-Based Application in Xcode called `PageTest`.
    1. Launch Xcode and create a new project.
    2. Choose the Page-Based Application template and click Next.
    3. Use the following information in the project options dialog box and click Next.
        - **Product Name:** PageTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson16
        - **Define Family:** iPhone
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`, but you can use any unique identifier for your application.

    4. Select a folder where this project should be created.
    5. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    6. Click Create.
2. Import image resources into your project.
    1. Ensure the project navigator is visible and the PageTest project is selected and expanded.
    2. Right-click the PageTest group and select Add Files to PageTest from the context menu.

3. Select the `Images` folder in this lesson's resources on the DVD.
4. Ensure the Copy Items to Destination Group's Folder (if needed) option is selected in the dialog box.
5. Click Add.

3. Edit the storyboard.
   1. Open the `MainStoryboard.storyboard` file in Interface Builder
   2. Ensure the `Lesson16 Data View Controller` scene is selected (it is the scene with the yellow background), and delete the contents of the scene. Your storyboard should resemble Figure 16-6.

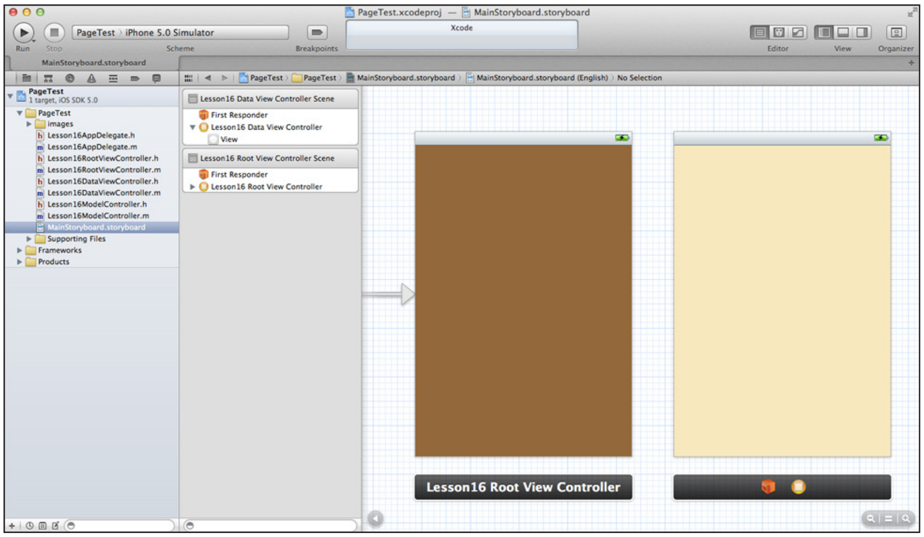

PageTest.xcodeproj — MainStoryboard.storyboard
Run
Stop
PageTest › iPhone 5.0 Simulator
Scheme
Breakpoints
Xcode
Editor
View
Organizer
MainStoryboard.storyboard
PageTest › PageTest › MainStoryboard.storyboard › MainStoryboard.storyboard (English) › No Selection
PageTest
1 target, iOS SDK 5.0
PageTest
images
Lesson16AppDelegate.h
Lesson16AppDelegate.m
Lesson16RootViewController.h
Lesson16RootViewController.m
Lesson16DataViewController.h
Lesson16DataViewController.m
Lesson16ModelController.h
Lesson16ModelController.m
MainStoryboard.storyboard
Supporting Files
Frameworks
Products
Lesson16 Data View Controller Scene
First Responder
Lesson16 Data View Controller
View
Lesson16 Root View Controller Scene
First Responder
Lesson16 Root View Controller
Lesson16 Root View Controller

FIGURE 16-6

   3. Add an Image View from the Object library onto the scene and resize/position it to X=0, Y=0, W = 320, H = 460.

4. Use the assistant editor to modify the `Lesson16DataViewController` class.
   1. Ensure the assistant editor is visible and the `Lesson16DataViewController.h` file is loaded in the editor.
   2. Delete the following line from the interface:

      ```
      @property (strong, nonatomic) IBOutlet UILabel *dataLabel;
      ```

   3. Add a new outlet to the `Lesson16DataViewController` class and connect it to the image view in the scene. Call the outlet `imageView`. The interface of the `Lesson16DataViewController` class should now resemble the following:

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

      @interface Lesson16DataViewController : UIViewController
      ```

```
@property (strong, nonatomic) id dataObject;
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@end
```

5. Update the implementation of the `Lesson16DataViewController` class.
    1. Open the `Lesson16DataViewController.m` file.
    2. Delete the following line from the top of the file:

    ```
    @synthesize dataLabel = _dataLabel;
    ```

    3. Replace the implementation of the `viewWillAppear:` method with the following:

    ```
    - (void)viewWillAppear:(BOOL)animated
    {
        [super viewWillAppear:animated];
        self.imageView.image = [UIImage imageNamed:self.dataObject];
    }
    ```

6. Update the implementation of the `Lesson16ModelController` class.
    1. Open the `Lesson16ModelController.m` file.
    2. Replace the implementation of the `init` method with the following:

    ```
    - (id)init
    {
        self = [super init];
        if (self) {
            // Create the data model.
            _pageData = [NSArray arrayWithObjects:@"image_1.png",
                            @"image_2.png", @"image_3.png", @"image_4.png",
                            @"image_5.png", nil];
        }
        return self;
    }
    ```

7. Test your app in the iOS Simulator.
    1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
    2. Flick through the images by swiping your finger across them horizontally. Note the page curl animation.

Please select Lesson 16 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

17 Creating UI Elements Programmatically

In previous lessons you learned how to use several standard user-interface objects, including buttons, image views, and labels. Using one of these objects in your application typically involves using Interface Builder to drag and drop instances onto a scene and then creating appropriate outlets/actions with the assistant editor.

However, there is another way to instantiate these objects, one that does not involve using the Interface Builder or assistant editor at all. Instead this alternate technique involves instantiating user-interface objects through Objective-C code. Any UIKit-based object that is instantiated with Interface Builder can also be instantiated programmatically.

Either method is fine; the one you choose is purely a matter of preference. Most likely, you will use the Interface Builder technique, but sometimes you may come across some code written by another programmer that creates user-interface elements programmatically. In these situations knowing how user-interface elements are created with code will be to your advantage.

Keep in mind, though, that some UIKit classes like `UIAlertView` and `UIActionSheet` cannot be instantiated with Interface Builder at all. This lesson shows you how to instantiate a few common UIKit objects with Objective-C code.

UIBUTTON

To create a `UIButton` instance programmatically, you first declare an appropriate `UIButton*` property in your view controller class:

```
@interface ViewController : UIViewController
@property (nonatomic, strong) UIButton* buttonOne;
@end
```

Next, you synthesize the property in your view controller's implementation file after the `@implementation` statement:

```
@implementation ViewController
@synthesize buttonOne;
```

Instantiate the `UIButton` object in the `viewDidLoad` method using the `buttonWithType` class method as follows:

```
buttonOne = [UIButton buttonWithType:UIButtonTypeRoundedRect];
```

The `buttonWithType:` class method takes a single argument that can be one of the following values:

- `UIButtonTypeRoundedRect`
- `UIButtonTypeCustom`
- `UIButtonTypeDetailDisclosure`
- `UIButtonTypeInfoLight`
- `UIButtonTypeInfoDark`
- `UIButtonTypeContactAdd`

To specify the size and position of the button instance, you need to provide a rectangle with the top, left, width, and height values for the button's *frame* property. For instance, to specify the button is located at left = 10, top = 20, has a width of 300 units, and a height of 40 units, use the following code:

```
buttonOne.frame = CGRectMake(10.0, 20.0, 300.0, 40.0);
```

When the user interacts with the button, one or more events are generated. To associate a method in the view controller class with an event generated by the button, use the `addTarget:action:forControlEvents:` method on the `UIButton` instance as follows:

```
[buttonOne addTarget:self
          action:@selector(onButtonPressed:)
          forControlEvents:UIControlEventTouchUpInside];
```

The first parameter is a reference to an object that contains the method to be called. In most cases this method will be implemented in the view controller class and this argument will be `self`. The second parameter contains a selector that identifies the method to be called. This method must be declared to not return a value, and takes a single argument of type `id` as shown here:

```
- (void) onButtonPressed:(id)sender;
```

To create a selector for the method, simply provide the name of the method to the `@selector` statement:

```
@selector(onButtonPressed:)
```

The third parameter identifies an event generated by the button to which you want to associate the method in question. This is usually one of the following values:

- `UIControlEventTouchDown`
- `UIControlEventTouchDownRepeat`
- `UIControlEventTouchDragInside`
- `UIControlEventTouchDragOutside`
- `UIControlEventTouchDragEnter`
- `UIControlEventTouchDragExit`
- `UIControlEventTouchUpInside`

- `UIControlEventTouchUpOutside`
- `UIControlEventTouchCancel`

To have the button appear on the screen, add it into the current view hierarchy. If you are creating your button in a view controller class, you can simply use the underlying view's `addSubView:` method as follows:

```
[self.view addSubview:buttonOne];
```

To summarize, the code to create and add a button programmatically in the `viewDidLoad` method looks like this:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    buttonOne = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    buttonOne.frame = CGRectMake(10.0, 10.0, 300.0, 40.0);

    [buttonOne addTarget:self
              action:@selector(onButtonPressed:)
              forControlEvents:UIControlEventTouchUpInside];

    [self.view addSubview:buttonOne];
}
```

If you want to use an image with the button, you need to first add the image to your project and load it into a `UIImage` object. Once the image has been loaded into an `UIImage` object, you can associate it with one or more button states by using the `setImage:forControlState:` method. The first parameter to this method is the `UIImage` instance; the second identifies one of several button states and can be one of the following values:

- `UIControlStateNormal`
- `UIControlStateHighlighted`
- `UIControlStateDisabled`

An example of using the `setImage:forControlState:` method is as follows:

```
[buttonOne setImage:[UIImage imageNamed:@"Normal.png"]
          forState:UIControlStateNormal];
```

UILABEL

Creating a `UILabel` instance is similar to creating a button. You need to create an appropriate `UILabel*` property in your view controller class, and can then instantiate the `UILabel` using the following code in the `viewDidLoad` method:

```
labelOne = [[UILabel alloc] initWithFrame:CGRectMake(10.0, 20.0,
                                                     300.0, 40.0)];
labelOne.textColor = [UIColor blackColor];
labelOne.backgroundColor = [UIColor clearColor];
labelOne.font = [UIFont fontWithName:@"Arial" size:10];
labelOne.textAlignment = UITextAlignmentCenter;
labelOne.text = @"Hello, World!";
[self.view addSubview:labelOne];
```

A `UILabel` object is instantiated by sending the `alloc` message to the `UILabel` class, and initialized with the `initWithFrame:` method. The `initWithFrame:` method takes a `CGRect` argument that specifies the position and size of the label:

```
labelOne = [[UILabel alloc] initWithFrame:CGRectMake(10.0, 20.0,
                                                     300.0, 40.0)];
```

Use the `textColor` and `backgroundColor` properties to specify the color of the text and the background of the label, respectively. The value you assign to these properties must be a `UIColor` instance.

You can instantiate a `UIColor` object with a specific RGB color by using the `colorwithRed:green:blue:alpha:` method. The arguments to the method are the values for the individual red, green, blue, and alpha components specified as numbers between 0.0 and 1.0. For example, to create a `UIColor` instance that represents red, you would use the following code:

```
UIColor* redColor = [UIColor colorWithRed:1.0 green:0.0 blue:0.0 alpha:1.0];
```

`UIColor` also defines a few class methods that allow you to create a few commonly used colors by referring to them by name instead of individual component values:

```
+ (UIColor *)blackColor;      // 0.0 white
+ (UIColor *)darkGrayColor;   // 0.333 white
+ (UIColor *)lightGrayColor;  // 0.667 white
+ (UIColor *)whiteColor;      // 1.0 white
+ (UIColor *)grayColor;       // 0.5 white
+ (UIColor *)redColor;        // 1.0, 0.0, 0.0 RGB
+ (UIColor *)greenColor;      // 0.0, 1.0, 0.0 RGB
+ (UIColor *)blueColor;       // 0.0, 0.0, 1.0 RGB
+ (UIColor *)cyanColor;       // 0.0, 1.0, 1.0 RGB
+ (UIColor *)yellowColor;     // 1.0, 1.0, 0.0 RGB
+ (UIColor *)magentaColor;    // 1.0, 0.0, 1.0 RGB
+ (UIColor *)orangeColor;     // 1.0, 0.5, 0.0 RGB
+ (UIColor *)purpleColor;     // 0.5, 0.0, 0.5 RGB
+ (UIColor *)brownColor;      // 0.6, 0.4, 0.2 RGB
+ (UIColor *)clearColor;      // 0.0 white, 0.0 alpha
```

The font used to display the text is specified in the `font` property of the label. The value of this property must be set to a `UIFont` instance. To obtain a `UIFont` instance that represents the system font in a specific point size, use the `systemFontOfSize:` class method of the `UIFont` class as follows:

```
labelOne.font = [UIFont systemFontOfSize:10];
```

To create an instance of specific font, use the `fontWithName:size:` class method:

```
[UIFont fontWithName:@"Arial" size:10];
```

`UILabel` instances allow you to set the text alignment by providing an appropriate value for the `textAlignment` property. The value specified must be one of the following:

- `UITextAlignmentLeft`
- `UITextAlignmentCenter`
- `UITextAlignmentRight`

Last but not least, use the `text` property to set up the text displayed in the label. The value of this property must be an `NSString` instance; you can also provide simple strings as shown here:

```
labelOne.text = @"Hello, World!";
```

UIIMAGEVIEW

Creating `UIImageView` instances programmatically is far simpler than creating labels and buttons. You need an appropriate `UIImageView*` property in your view controller class and can create the image view using code similar to the following:

```
UIImage* contentImage = [UIImage imageNamed:@"flag.png"];
bgView = [[UIImageView alloc] initWithImage:contentImage];
bgView.frame = CGRectMake(10, 20, 100, 150);
[self.view addSubview:bgView];
```

A `UIImageView` object is instantiated by sending the `alloc` message to the `UIImageView` class, and initialized with the `initWithImage:` method. The `initWithImage:` method requires a single argument, a `UIImage` instance that represents the image you want to display.

To resize/position the image view, provide an appropriate `CGRect` value for the `frame` property. Keep in mind that by default, the image view will resize its contents to match the dimensions provided in the `frame` property. If you do not want resizing to occur, provide the same dimensions as the original image, or specify one of the following values for the `contentMode` property to affect how images appear:

- `UIViewContentModeScaleToFill`
- `UIViewContentModeScaleAspectFit`
- `UIViewContentModeScaleAspectFill`
- `UIViewContentModeCenter`
- `UIViewContentModeTop`
- `UIViewContentModeBottom`
- `UIViewContentModeLeft`
- `UIViewContentModeRight`
- `UIViewContentModeTopLeft`
- `UIViewContentModeTopRight`
- `UIViewContentModeBottomLeft`
- `UIViewContentModeBottomRight`

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template that constructs its interface programmatically in the `viewDidLoad:` method of the view controller class. The application will create a label, an image view, and a button. Tapping on the button will present an alert view.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image files into the project.
- Add code to the view controller class to create the user interface programmatically.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 17 folder in the download.*

Hints

- When creating the user interface programmatically, you do not need to use Interface Builder.

Step-by-Step

1. Create a Single View Application in Xcode called `CodeBasedUI`.
    1. Launch Xcode and create a new project.
    2. Choose the Single View Application template and click Next.
    3. Use the following information in the project options dialog box and click Next.
        - **Product Name:** CodeBasedUI
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson17
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    4. Select a folder where this project should be created.
    5. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    6. Click Create.
2. Import image resources into your project.
    1. Ensure the project navigator is visible and the CodeBasedUI project is selected and expanded.

2. Right-click the CodeBasedUI group and select Add Files to CodeBasedUI from the context menu.
3. Select the `Images` folder in this lesson's resources on the DVD.
4. Ensure the Copy Items to Destination Group's folder (if needed) option is selected in the dialog box.
5. Click Add.

3. Edit the view controller class.

1. Add three `nonatomic`, `strong` properties to the view controller class, for a `UILabel`, `UIImageView`, and `UIButton`. Name these properties `countryName`, `flagView`, and `infoButton`, respectively.
2. Add the following method declaration to the `ViewController.h` file:

```
- (void) onShowInfo:(id)sender;
```

3. Your `Lesson17ViewController.h` file should now resemble the following:

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

@interface Lesson17ViewController : UIViewController
@property (nonatomic, strong) UIButton* infoButton;
@property (nonatomic, strong) UILabel* countryName;
@property (nonatomic, strong) UIImageView* flagView;

- (void) onShowInfo:(id)sender;
@end
```

4. Add appropriate `@synthesize` statements at the top of the `ViewController.m` file corresponding to the `@property` declarations in the `ViewController.h` file:

```
@synthesize infoButton;
@synthesize countryName;
@synthesize flagView;
```

5. Add the following code to the implementation of the `viewDidLoad` method, after the `[super viewDidLoad]` line:

```
UIImage* contentImage = [UIImage imageNamed:@"flag.png"];
flagView = [[UIImageView alloc] initWithImage:contentImage];
flagView.frame = CGRectMake(85, 20, 150, 87);
[self.view addSubview:flagView];

countryName = [[UILabel alloc] initWithFrame:CGRectMake(10.0, 120.0,
                                                  300.0, 40.0)];
countryName.textColor = [UIColor blackColor];
countryName.backgroundColor = [UIColor clearColor];
countryName.font = [UIFont systemFontOfSize:12];
countryName.textAlignment = UITextAlignmentCenter;
countryName.text = @"United Kingdom of Great Britain and Northern Ireland";
[self.view addSubview:countryName];

infoButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
infoButton.frame = CGRectMake(10.0, 200.0, 300.0, 40.0);
[infoButton setTitle:@"What is the capital city?"
```

```
                 forState:UIControlStateNormal];
[infoButton addTarget:self
                 action:@selector(onShowInfo:)
                 forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:infoButton];
```

6. Implement the `onShowInfo:` method as follows:

```
- (void) onShowInfo:(id)sender
{
    UIAlertView* alert = [[UIAlertView alloc]
                   initWithTitle:@"The capital city of this country is"
                   message:@"London"
                   delegate:nil
                   cancelButtonTitle:@"Ok"
                   otherButtonTitles:nil];
    [alert show];
}
```

4. Test your app in the iOS Simulator.
    1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
    2. Tap the button titled "What is the capital city?" You should see an alert view similar to the one in Figure 17-1.

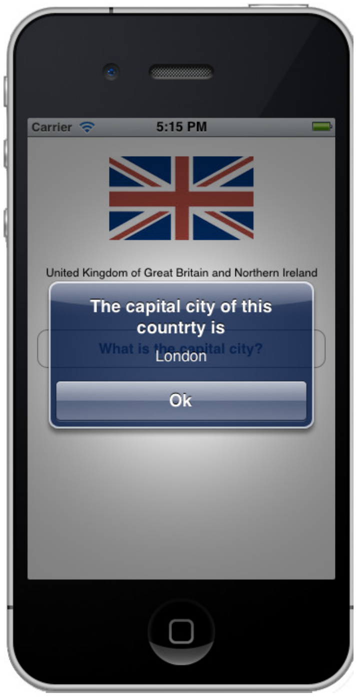

Carrier
5:15 PM
United Kingdom of Great Britain and Northern Ireland
The capital city of this countrty is
What is the capital city?
London
Ok

FIGURE 17-1

Please select Lesson 17 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`, *to view the video that accompanies this lesson.*

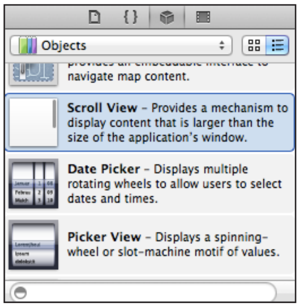
18 Creating Views That Scroll

The applications you have built in the previous lessons have one thing in common—all their content fits neatly into a single view the size of the device screen. Sometimes that is not the case, and when that happens you have two strategies to deal with the situation. You can either try to break up the content of your application and present it across multiple views using tab bars or navigation controllers, or you could still keep all the content in a single view but allow the user to scroll through the content of the view.

UIKit provides the `UIScrollView` class, specifically designed to help you create scrolling views. In this lesson you learn to use `UIScrollView` instances in your applications.

THE UISCROLLVIEW CLASS

To create a `UIScrollView` instance using the Xcode Interface Builder, simply drag and drop a Scroll View object from the Object library onto a scene, and create an outlet using the assistant editor (Figure 18-1).



navigate map content.

Scroll View – Provides a mechanism to display content that is larger than the size of the application's window.

Date Picker – Displays multiple rotating wheels to allow users to select dates and times.

Picker View – Displays a spinning-wheel or slot-machine motif of values.

FIGURE 18-1

You can add one or more instances of `UIView` subclasses as subviews of the scroll view. The collective dimensions of these subviews can be much larger than the dimensions of the scroll view itself (Figure 18-2).

The dimensions of the content managed by a scroll view can be read (or set) using the `contentSize` property. The `contentSize` property is a `CGSize` structure and contains two float members, `height` and `width`. Thus, if `scrollView` is a `UIScrollView` instance, the following code could be used to read the height and width of the content area:

```
float contentHeight = scrollView.contentSize.height;
float contentWidth = scrollView.contentSize.width;
```

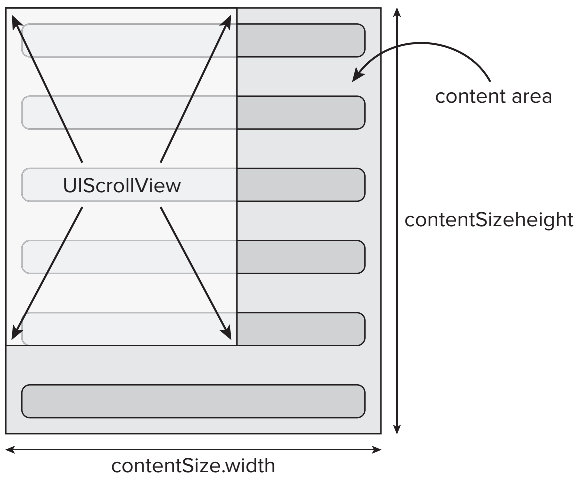


FIGURE 18-2

When you create a scroll view instance with Interface Builder, the size of the content area is exactly the same as the size of the scroll view. Thus, scroll views, by default, do not scroll. To enable the scrolling behavior, you need to set up the `contentSize` property programmatically. You can do this at any point after the scroll view is instantiated. If you created the scroll view with Interface Builder, you may want to set it up in the `viewDidLoad` method of the view controller class that contains the scroll view, using code similar to the following:

```
    scrollView.contentSize = CGSizeMake(320, 4200);
```

Another property related to the scrolling behavior is the `contentOffset` property. This property is a `CGPoint` structure and contains two float members, `x` and `y`, that represent the distance scrolled by the user along the horizontal and vertical axes (Figure 18-3).

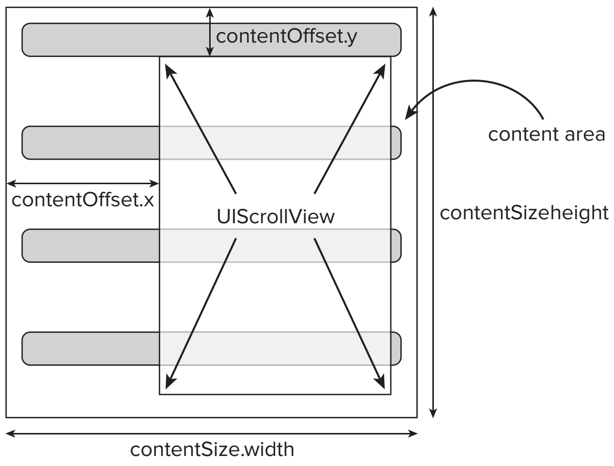


FIGURE 18-3

You can add user interface elements to a scroll view with Interface Builder by simply dragging and dropping them from the Object library onto the scroll view. Positioning elements that are not initially visible in the scroll view can be a bit tricky. One way to solve this problem is to drag and drop elements onto the scroll view and then provide precise numeric values for the X and Y positions using the Attributes inspector (Figure 18-4).

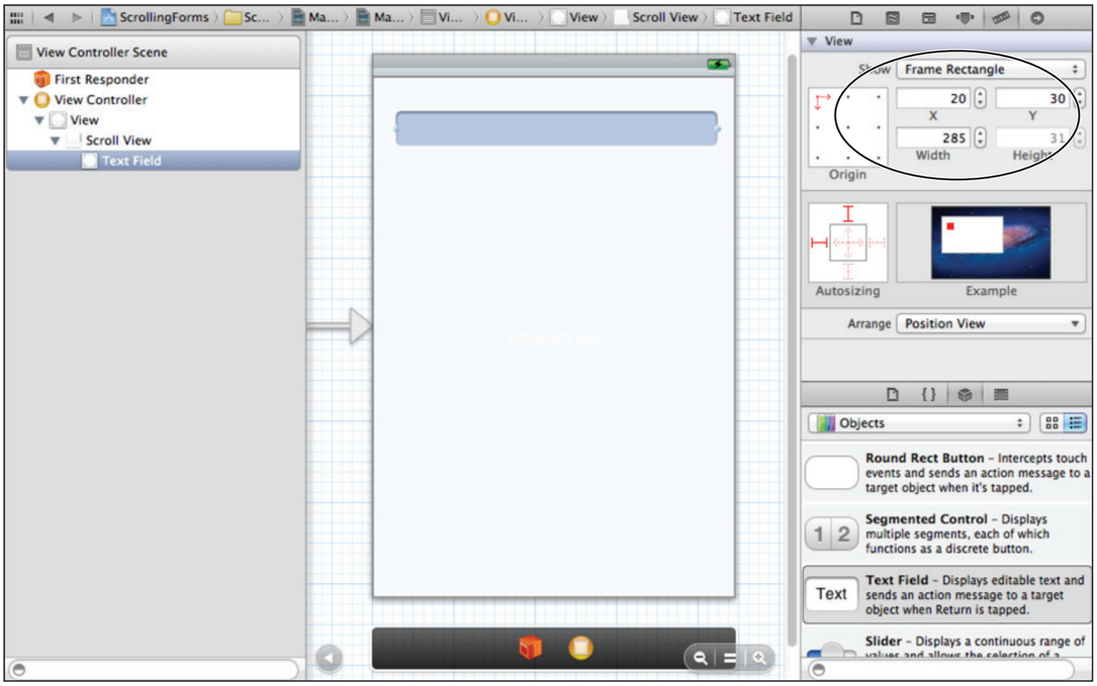

ScrollingForms
View Controller Scene
First Responder
View Controller
View
Scroll View
Text Field
View
Show Frame Rectangle
20 X
30 Y
285 Width
31 Height
Origin
Autosizing
Example
Arrange Position View
Objects
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.
Text Field – Displays editable text and sends an action message to a target object when Return is tapped.
Slider – Displays a continuous range of

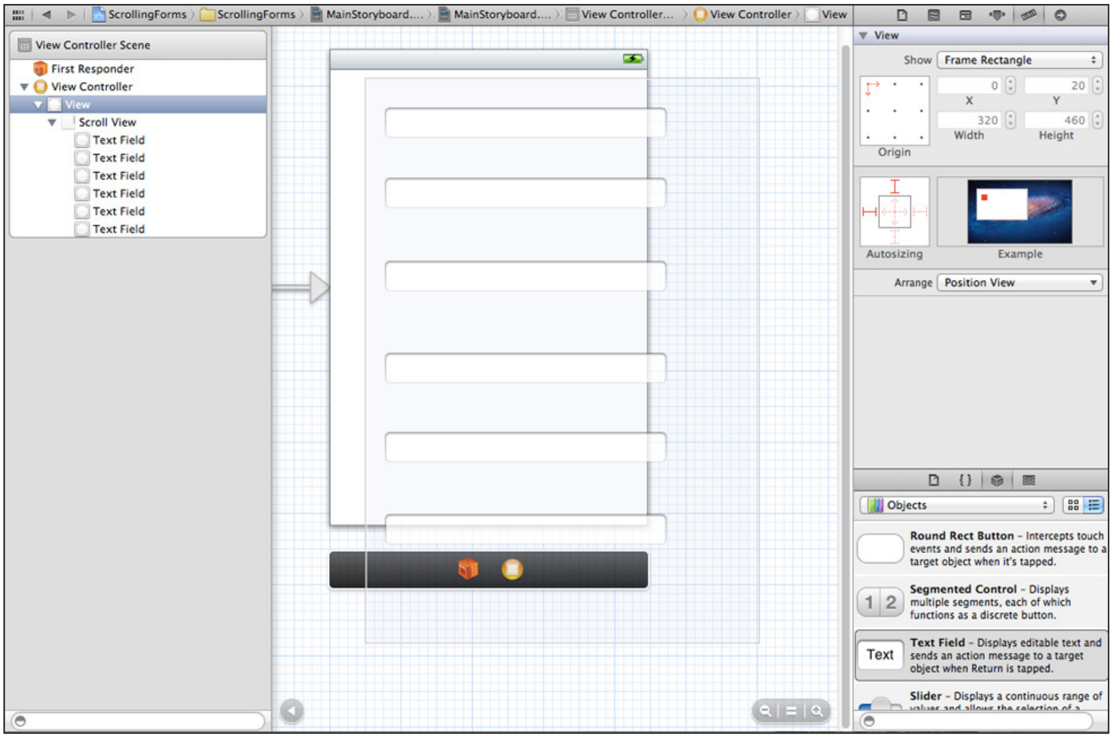
FIGURE 18-4

Another way is to resize/reposition the scroll view within the scene, and create the user interface elements visually in their correct positions. This approach requires you move the scroll view in the scene a few times until you get the results you want (Figure 18-5). Don't forget to reset the scroll view's position and size to their initial values after you are done.

You could also create the user interface elements programmatically, and insert them at the appropriate position within the scroll view. The code to add a simple `UILabel` instance into a scroll view programmatically is presented in the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UILabel* countryName = [[UILabel alloc] initWithFrame:CGRectMake(10.0,
                                                  120.0, 300.0, 40.0)];
    countryName.textColor = [UIColor blackColor];
    countryName.backgroundColor = [UIColor clearColor];
    countryName.font = [UIFont systemFontOfSize:12];
    countryName.textAlignment = UITextAlignmentCenter;
```

```
        countryName.text = @"United States of America";
        [scrollView addSubview:countryName];
    }
```



ScrollingForms
ScrollingForms
MainStoryboard....
MainStoryboard....
View Controller...
View Controller
View
View Controller Scene
First Responder
View Controller
View
Scroll View
Text Field
Text Field
Text Field
Text Field
Text Field
Text Field
View
Show Frame Rectangle
0 X
20 Y
320 Width
460 Height
Origin
Autosizing
Example
Arrange Position View
Objects
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.
Text Field – Displays editable text and sends an action message to a target object when Return is tapped.
Slider – Displays a continuous range of values and allows the selection of a

FIGURE 18-5

Regardless of which method you use, you need to set the `contentSize` property to an appropriate value to enable scrolling.

SCROLL VIEWS AND TEXT FIELDS

A common scenario in which you are likely to use a scroll view involves multiple text fields in a scene. If you tap a text field closer to the bottom of the screen, a keyboard automatically pops up and covers part of the user interface. This is illustrated in Figure 18-6; when a user taps on the Address (Line 1): field, the keyboard comes up and covers the text field, thus making it impossible for the user to see what is being typed.

Scroll views provide a simple and elegant solution to this problem; you can change the y offset of the content area within the scroll view when a specific text field is tapped, thus moving the content toward the top by a small amount. This solution is explored next, in this lesson's Try It section.

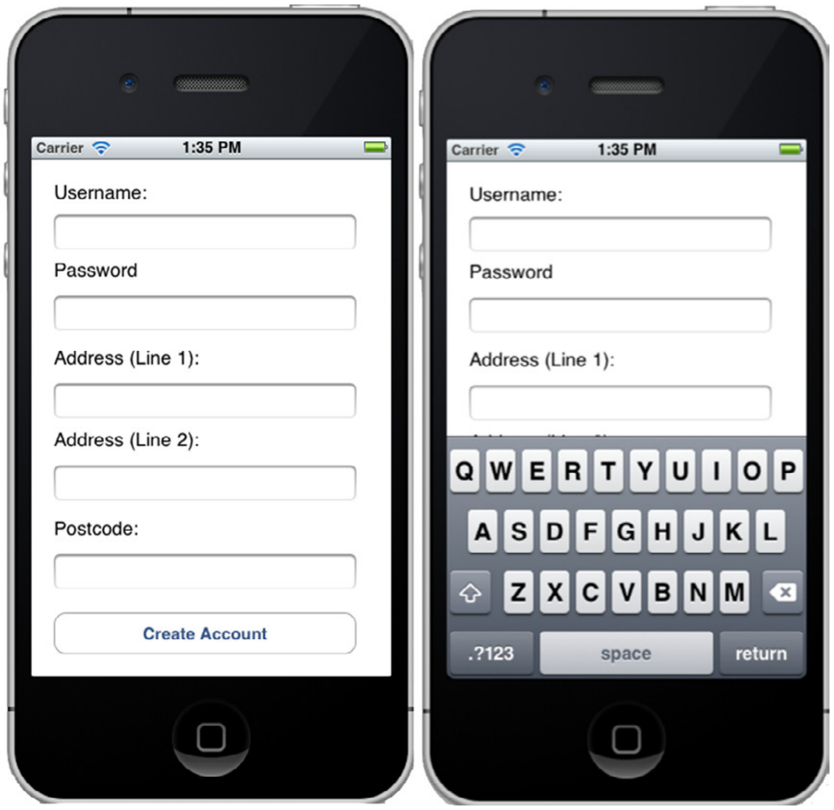


FIGURE 18-6

TRY IT

In this Try It, you create a simple application based on the Single View Application template that contains several text fields and a scroll view. When a text field is tapped on, the content of the scroll view is moved up by a small amount to ensure that the iOS keyboard will not cover the text field.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a scroll view to the default scene of the storyboard.
- Use Interface Builder to add several user interface elements to the scroll view.
- Add code to the view controller class to move the content in the scroll view when a text field is tapped, thus ensuring the text field is always visible.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 18 folder in the download.*

Hints

- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.

Step-by-Step

1. Create a Single View Application in Xcode called `ScrollingForms`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** ScrollingForms
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson18
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add user interface elements to your storyboard's scene.
    1. Add a `UIScrollView` instance to the default scene.
        1. Use the Object library to add a `Scroll View` to the default scene of the storyboard.
        2. Use the Size inspector to resize and position it at X = 0, Y = 0, Width = 320, Height = 460.
        3. Using the assistant editor, create an outlet for the scroll view in the view controller class called `scrollView`.

2. Add user interface elements to the scroll view.

    1. Use the Object library to add five Label instances and five Text Field instances to the scroll view. Position them to resemble Figure 18-7.

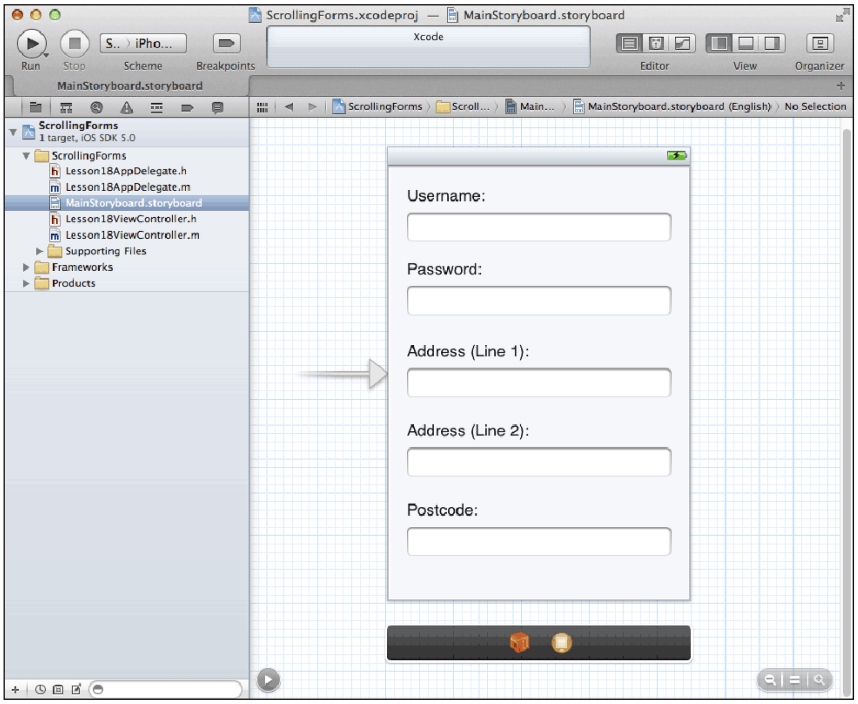


FIGURE 18-7

    2. Use the assistant editor to create outlets for each of the text fields in the view controller class. Name the outlets **`usernameField`**, **`passwordField`**, **`addressField1`**, **`addressField2`**, and **`postcodeField`**.

3. Ensure the `Lesson18ViewController` class implements the `UITextFieldDelegate` protocol.

    1. Modify the interface declaration of the `Lesson18ViewController` class from:

```
@interface Lesson18ViewController : UIViewController
```

    to

```
@interface Lesson18ViewController : UIViewController <UITextFieldDelegate>
```

4. Add additional property declarations to the `Lesson18ViewController.h` file.

    1. Add the following property declarations to the `Lesson18ViewController.h` file:

```
@property float keyboardHeight;
@property (weak, nonatomic) UITextField *currentTextField;
```

2. The code in the `Lesson18ViewController.h` file should now resemble the following:

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

@interface Lesson18ViewController : UIViewController <UITextFieldDelegate>
@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UITextField *usernameField;
@property (weak, nonatomic) IBOutlet UITextField *passwordField;
@property (weak, nonatomic) IBOutlet UITextField *addressField1;
@property (weak, nonatomic) IBOutlet UITextField *addressField2;
@property (weak, nonatomic) IBOutlet UITextField *postcodeField;

@property float keyboardHeight;
@property (weak, nonatomic) UITextField *currentTextField;

@end
```

3. Edit the implementation of the view controller class.

    1. Add the following `@synthesize` statements to the top of the `Lesson18ViewController.m` file, after the `@implementation Lesson18ViewController` line:

    ```
    @synthesize keyboardHeight;
    @synthesize currentTextField;
    ```

    2. Set up the view controller instance to be the `delegate` object for the text field instances by modifying the implementation of the `viewDidLoad:` method to the following:

    ```
    - (void)viewDidLoad
    {
        [super viewDidLoad];

        usernameField.delegate = self;
        passwordField.delegate = self;
        addressField1.delegate = self;
        addressField2.delegate = self;
        postcodeField.delegate = self;
    }
    ```

    3. You need to tell iOS to call the `keyboardDidShow:` and `keyboarDid-Hide:` methods in your view controller class when the keyboard becomes visible/hidden, respectively. To do this, you need to register these methods as observers for the `UIKeyboardDidShowNotification` and `UIKeyboardDidHideNotification` events. Modify the implementation of the `viewWillAppear:` method to the following:

    ```
    - (void)viewWillAppear:(BOOL)animated
    {
        [super viewWillAppear:animated];

        [NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidShow:)
    ```

```
        name:UIKeyboardDidShowNotification
        object:self.view.window];

        [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidHide:)
        name:UIKeyboardDidHideNotification
        object:nil];
}
```

4. You need to tell iOS that your code is not interested in the notifications previously registered by modifying the implementation of the `viewDidDisappear:` method to the following:

```
- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];

    [NSNotificationCenter defaultCenter] removeObserver:self
    name:UIKeyboardDidShowNotification
    object:nil];
    [NSNotificationCenter defaultCenter] removeObserver:self
    name:UIKeyboardWillHideNotification
    object:nil];
}
```

The above code snippet removes individual observers one by one, if you want to remove all observers in one line, you can alternately, implement the `viewDidDisappear:` *method as:*

```
- (void) viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];

    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

4. Implement the `keyboardDidShow:` method in your view controller class as follows:

```
-(void) keyboardDidShow:(NSNotification *) notification
{
    // get height of keyboard
    NSDictionary* info = [notification userInfo];
    CGRect keyboardFrame = [[info objectForKey:UIKeyboardFrameEndUserInfoKey]
                            CGRectValue];
    keyboardHeight =  keyboardFrame.size.height;

    // ensure current text field is visible, if not adjust the contentOffset
    // of the scrollView appropriately.

    float textFieldTop = currentTextField.frame.origin.y;
    float textFieldBottom = textFieldTop + currentTextField.frame.size.height;

    if (textFieldBottom > keyboardHeight)
    {
        [scrollView setContentOffset:
        CGPointMake(0, textFieldBottom - keyboardHeight)
        animated:YES];
    }
}
```

5. The preceding code snippet stores the height of the keyboard in a member variable `keyboardHeight`. It then tests to see if the currently active text field is partly or wholly covered by the keyboard. If it is, it updates the `contentOffset` property of the scroll view to rectify the situation.

6. Implement the `keyboardDidHide:` method in your view controller class as follows:

```
-(void) keyboardDidHide:(NSNotification *) notification
{
    [scrollView setContentOffset:CGPointMake(0, 0)
                      animated:YES];
}
```

7. The preceding code snippet resets the `contentOffset` property of the scroll view to X = 0, and Y = 0.

8. Implement the `textFieldShouldReturn:` method of the `UITextFieldDelegate` protocol as follows:

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

9. Implement the `textFieldDidBeginEditing:` method of the `UITextFieldDelegate` protocol as follows:

```
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    // save reference to currently-active text field
    currentTextField = textField;

    // ensure this field is visible by adjusting the contentOffset
    // property of the scrollView instance appropriately.
    float textFieldTop = currentTextField.frame.origin.y;
    float textFieldBottom = textFieldTop +
                            currentTextField.frame.size.height;

    if ((textFieldBottom > keyboardHeight) && (keyboardHeight != 0.0))
    {
        [scrollView setContentOffset:CGPointMake(0, textFieldBottom -
                                                    keyboardHeight)
                            animated:YES];
    }
}
```

10. The preceding code snippet is called when the user taps on a text field. It first saves a reference to the text field in the variable `currentTextField`. It then checks to see if the field is wholly/partially obscured by the keyboard. If this is the case, it updates the `contentOffset` property of the scroll view to rectify this situation.

5. To test your app in the iOS Simulator, click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

> *Please select Lesson 18 on the DVD that accompanies the print book, or go to* `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

19 Popovers and Modal Views

This lesson presents two ways of temporarily displaying data for user feedback:

- Popovers
- Modal views

Though the overall purpose is the same between popovers and modal views, they behave differently when implemented. For example, to dismiss a popover, you simply tap outside the bounds of the popover itself. To dismiss a modal view, you must touch a user-defined button that will dismiss it. Because of this permanence, modal views are presented to the user for immediate required feedback in order to continue the application.

POPOVERS

Popovers by definition are views that are revealed when a control is tapped. They also have the visual effect of being attached to that control as shown in Figure 19-1.

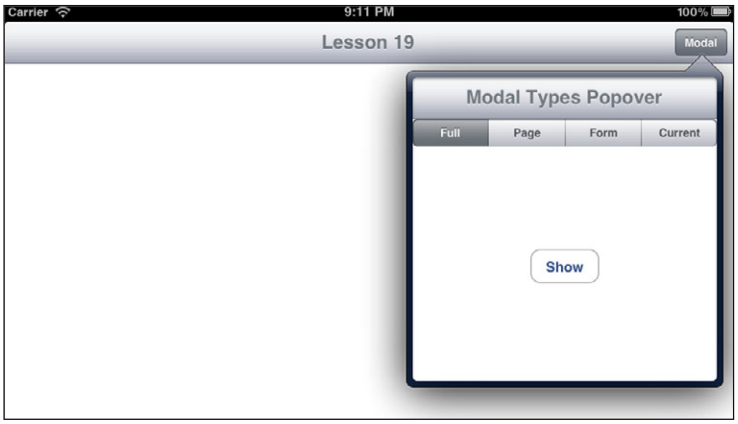


FIGURE 19-1

Usage Guidelines

The popover is an iPad-only view, and suggested uses are as follows:

- ➤ To provide a list of additional information related to the selected control.
- ➤ In a split view in portrait mode, to display the list that would appear in the left pane in landscape mode.
- ➤ To display a list of options displayed in an action sheet.

Apple User Interface Guidelines recommend that you do not provide a Done or Dismiss option in your popover. By design, a popover is dismissed by simply tapping outside the popover itself.

Presenting the Popover

Popovers can be associated with buttons on the toolbar by using the `presentPopoverFromBarButtonItem:permittedArrowDirections:animated` method. They can also be associated with a particular view throughout the `presentPopoverFromRect:inView:permittedArrowDirections:animated` method.

The following code illustrates how to present a view controller within a popover that is attached to a bar button item attached to a navigation bar:

```
PopoverViewController *controller = [[self storyboard]
          instantiateViewControllerWithIdentifier:@"PopoverView"];

UIPopoverController *popoverController =
     [[UIPopoverController alloc] initWithContentViewController:controller];
     [popoverController setPopoverContentSize:CGSizeMake(320.0, 320.0)];
     [popoverController presentPopoverFromBarButtonItem:[self modalButton]
          permittedArrowDirections:UIPopoverArrowDirectionUp animated:YES];
```

Dismissing the Popover

Apple discourages the use of a Done or Dismiss button to dismiss popovers. However, in some instances it needs to be programmatically dismissed. The following code illustrates how to dismiss the popover:

```
[popoverController dismissPopoverAnimated:YES];
```

MODAL VIEWS

Modal views are a way to manage the flow of your applications. Specifically, they will interrupt the flow to acquire vital information on how to proceed further in the application.

Usage Guidelines

Unlike popovers, modal views are not limited to the iPad. Suggested uses are as follows:

- ➤ To acquire information immediately

- To provide vital information before proceeding
- To alter application logic flow, depending on the response received

Presentation Styles

The four presentation styles for modal views are as follows:

- `UIModalPresentationFullScreen`: Takes up the full screen.
- `UIModalPresentationPageSheet`: In landscape is centered horizontally on the page and does not fill the screen.
- `UIModalPresentationFormSheet`: Appears as a self-contained form centered in the view.
- `UIModalPresentationCurrentContext`: Adopts the presentation style of its parent.

Transition Styles

The three transition styles for modal views are as follows:

- `UIModalTransitionStyleCoverVertical`: The default and used to enter/exit from the bottom of the view.
- `UIModalTransitionStyleFlipHorizontal`: Used to enter/exit by flipping horizontally between two views
- `UIModalTransitionStyleCrossDissolve`: Used to fade between two views.

Presenting the Modal View

The following code illustrates how to present a view controller modally:

```
ModalViewController *modalView = [[self storyboard]
     instantiateViewControllerWithIdentifier:@"ModalView"];
     [modalView setModalTransitionStyle:UIModalTransitionStyleCoverVertical];
     [modalView setModalPresentationStyle:UIModalPresentationFormSheet];
     [self presentModalViewController:modalView animated:YES];
```

Dismissing the Modal View

The following code illustrates how to dismiss the modal view:

```
[self dismissModalViewControllerAnimated:YES];
```

The transition style used to dismiss a modal view is the same style used to present it. If you fade the view in, when it is dismissed, it will fade out.

TRY IT

In this Try It, you implement a single view application for the iPad that has a button on the top right. Tapping this button reveals a popover that offers a choice of presentation styles that will be used for the appearance of the modal view. After choosing the presentation style, a tap on the Show button presents the modal view. Tapping the Done button on the modal view dismisses it.

> *You can download the code and resources for this Try It from the book's web page at* `www.wrox.com`*. You can find them in the Lesson 19 folder in the download.*

Lesson Requirements

- Create an Xcode project for iPad using the Single View Application template.
- Create a storyboard including just a root view controller.
- Implement a dynamic prototype table view.
- Respond to the selection of a Modal button by displaying a popover with a choice of modal presentation styles.
- Selecting a presentation style and tapping the Show button reveals a modal view.
- Tapping Done on the modal view dismisses it.

Hints

- Because this application uses storyboards instead of xib files, remember to have the Use Storyboard option checked at project creation.
- Select iPad from the Device Family list.
- The popover as well as the modal view will have to be dismissed programmatically.

Step-by-Step

1. Create a Single View Application.
    1. Launch Xcode.
    2. Create your new iOS project.
        - **a.** To create a new project, select Create a New Xcode Project, from the initial Welcome to Xcode window.
        - **b.** On the left under iOS, select Application.
        - **c.** Select Single View Application from the template list and click Next.

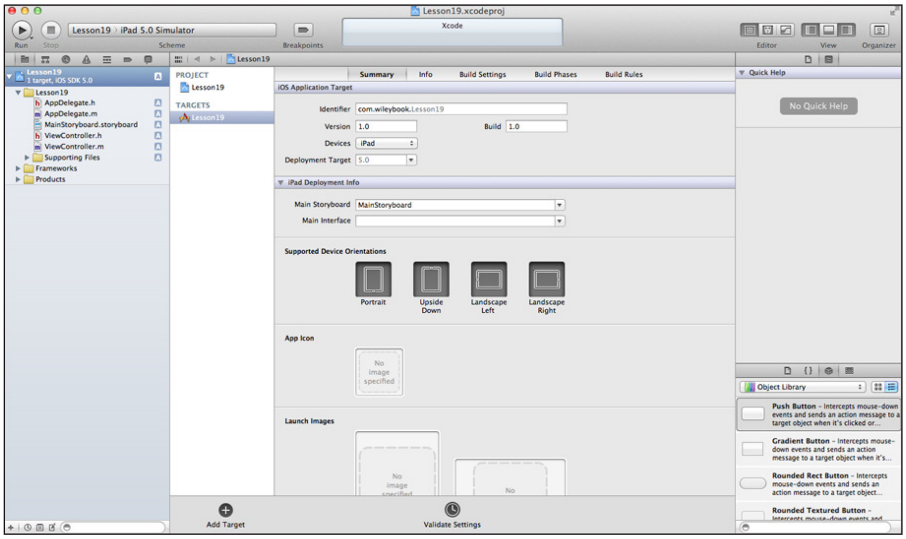
d. Choose the following options for your project:

- ➤ **Product Name:** Lesson19
- ➤ **Company Identifier:** com.wileybook
- ➤ **Class Prefix:** leave blank
- ➤ **Device Family:** iPad
- ➤ **Use Storyboard:** Checked
- ➤ **Use Automatic Reference Counting:** Checked
- ➤ **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

e. Select the location on your computer where the project will be saved and click Create.

f. Your Xcode project has been created as shown in Figure 19-2.


Lesson19.xcodeproj

Lesson19 › iPad 5.0 Simulator

iOS Application Target

Identifier com.wileybook.Lesson19
Version 1.0 Build 1.0
Devices iPad
Deployment Target 5.0

iPad Deployment Info

Main Storyboard MainStoryboard
Main Interface

FIGURE 19-2

2. Add the `PopoverViewController` and `ModalViewController`.

1. To add the popover view controller to the storyboard:

a. Select the Lesson19 folder.

b. Select File ➪ New ➪ New File from the Xcode menu.

c. On the left under iOS, select Cocoa Touch.

d. Select UIViewController from the template list and click Next.

e. Choose the following options for your new file:

➤ **Class:** PopoverViewController

➤ **Subclass of:** UIViewController

➤ **Select:** Targeted for iPad

➤ **Deselect:** With XIB for user interface

f. Click Next.

g. Click Create to save the class in your project folder.

2. To add the modal view controller to the storyboard:

a. Select the Lesson19 folder.

b. Select File ➪ New ➪ New File from the Xcode menu.

c. On the left under iOS, select Cocoa Touch.

d. Select UIViewController from the template list and click Next.

e. Choose the following options for your new file:

➤ **Class:** ModalViewController

➤ **Subclass of:** UIViewController

➤ **Select:** Targeted for iPad

➤ **Deselect:** With XIB for user interface

f. Click Next.

g. Click Create to save the class in your project folder.

3. Select File ➪ Save to save your project.

3. Design the user interface.

1. Design the `ViewController`.

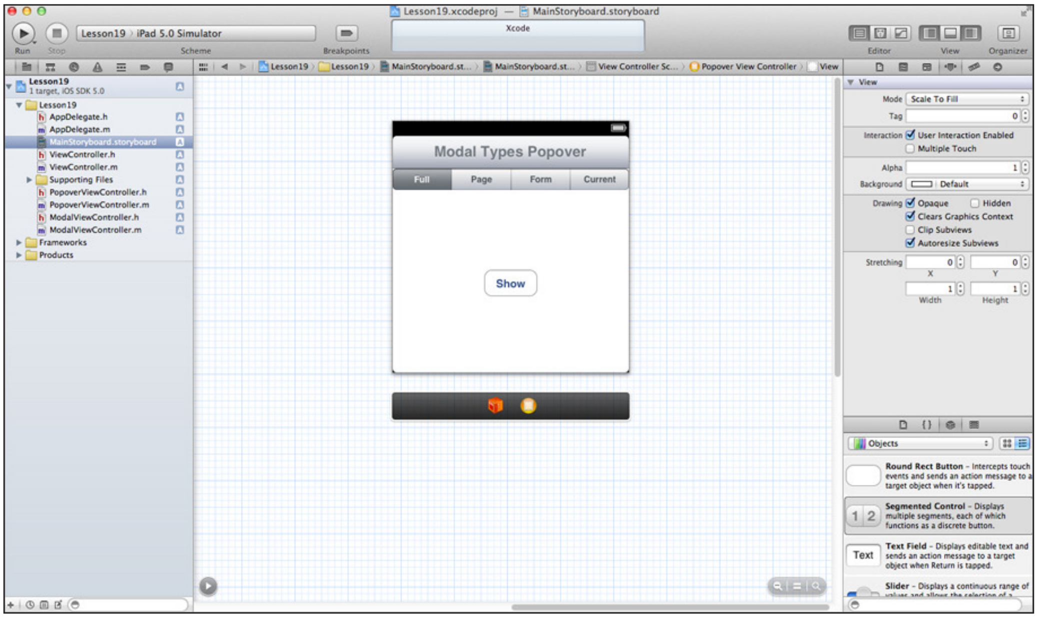
a. On the left, select `MainStoryboard.storyboard`.

b. On the right select the third button in the View section to display the Utilities view.

c. Drag a `Navigation Bar` from the list, place it on the top section of the view, double-click the Title, and enter **Lesson 19**.

d. Drag a `BarButtonItem` from the object list, place it on the right section of the navigation bar you just added, double-click Item, and enter **Modal**.

2. Design the `PopoverViewController`.
   a. On the left, select `MainStoryboard.storyboard`.
   b. Drag a `View Controller` from the list, and place it to the right of the `ViewController` on the storyboard.
   c. From the Identity Inspector, enter **PopoverViewController** for the Class.
   d. From the Attributes Inspector, select Freeform for the Size parameter and enter **PopoverView** for the identifier.
   e. Select the view and from the Size Inspector, enter **320** for both the Width and Height.
   f. Drag a `Navigation Bar` from the list, place it on the top section of the view, double-click the Title, and enter **Modal Types Popover**.
   g. Drag a `Round Rect Button` from the list, place it on the middle of the view, double-click the center, and enter **Show**.
   h. Drag a `Segmented Control` from the object list, and place it all the way on the left just attached to the bottom of the navigation bar.
   i. Select the Attributes Inspector and select Bar for Style and 4 for Segments.
   j. Center the control and add the following titles for each cell, respectively: **Full**, **Page**, **Form**, and **Current**, as shown in Figure 19-3.



View
Mode: Scale To Fill
Tag: 0
Interaction: ✓ User Interaction Enabled; ☐ Multiple Touch
Alpha: 1
Background: Default
Drawing: ✓ Opaque; ☐ Hidden; ✓ Clears Graphics Context; ☐ Clip Subviews; ✓ Autoresize Subviews
Stretching: 0 (X), 0 (Y); 1 (Width), 1 (Height)

Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.

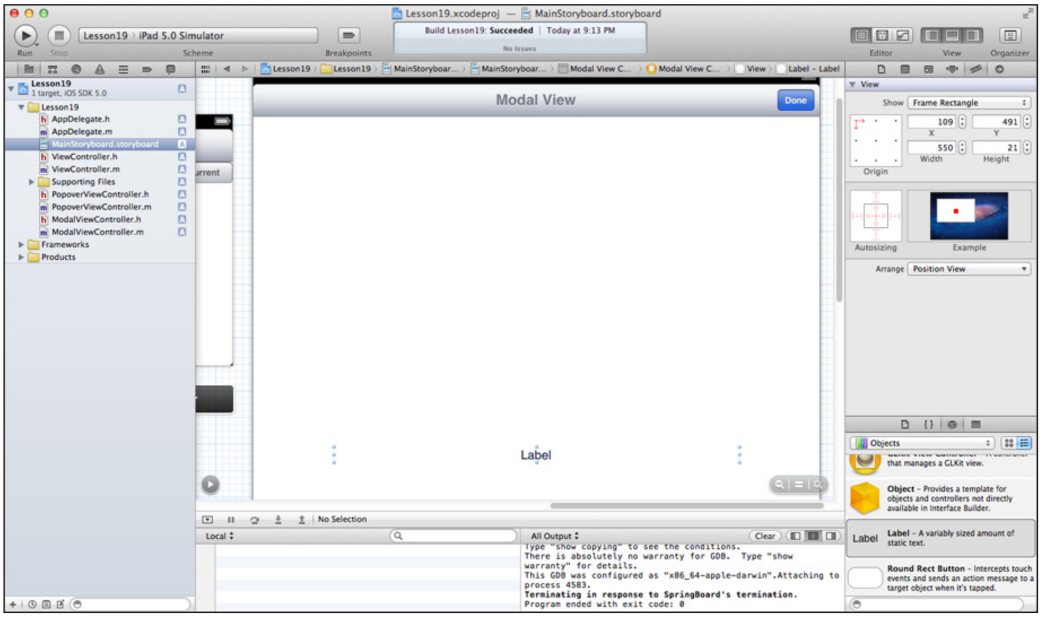
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Slider – Displays a continuous range of

FIGURE 19-3

3. Design the `ModalViewController`.
    a. On the left, select `MainStoryboard.storyboard`.
    b. Drag a `View Controller` from the list, and place it to the right of the `ViewController` on the storyboard.
    c. From the Identity Inspector, enter **ModalViewController** for the Class.
    d. From the Attributes Inspector, enter **ModalView** for the identifier.
    e. Drag a `Navigation Bar` from the list, place it on the top section of the view, double-click the Title, and enter **Modal View**.
    f. Drag a `BarButtonItem` from the object list, place it on the right section of the navigation bar you just added, and from the Attributes Inspector, select Done for the Identifier. The button should have the title Done and have a blue background.
    g. Drag a `Label` from the object list, and place it on the middle of the view. From the Attributes Inspector select the center alignment, and from the Size Inspector set the width to 550 and deselect the Autosizing anchors as shown in Figure 19-4.


Lesson19.xcodeproj — MainStoryboard.storyboard
Build Lesson19: Succeeded | Today at 9:13 PM
Modal View
Done
Label
Frame Rectangle
109
491
550
21
Width
Height
Autosizing
Example
Position View

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "x86_64-apple-darwin".Attaching to
process 4583.
Terminating in response to SpringBoard's termination.
Program ended with exit code: 0
```

FIGURE 19-4

4. Select File ➪ Save to save your project.

4. Make the Connections to the outlets and actions.

1. To make the connections on the `ViewController perform the following:`
   a. On the storyboard, select the status bar at the top of the first view controller.
   b. From the Editor section on the top right, select the Assistant Editor.
   c. Select the Modal button and control-drag to the interface source code just above the `@end`.
   d. Select Outlet for Connection, enter **modalButton** for the outlet name. and click Connect.
   e. Select the Modal button and control-drag to the interface source code just above the `@end`.
   f. Select Action for Connection, enter **selectModalType** for the action name, and click Connect.
2. To make the connections on the `PopoverViewController`, perform the following:
   a. On the storyboard, select the status bar at the top of the popover view controller.
   b. Select the segmented control and control-drag to the interface source code just above the `@end`.
   c. Select Outlet for Connection, enter **modalTypeSegmentedController** for the outlet name, and click Connect.
   d. Select the Show button and control-drag to the interface source code just above the `@end`.
   e. Select Action for Connection, enter **showModalView** for the action name, and click Connect.
3. To make the connections on the `ModalViewController`, perform the following:
   a. On the storyboard, select the status bar at the top of the first view controller.
   b. From the Editor section on the top right, select the Assistant Editor.
   c. Select the label and control-drag to the interface source code just above the `@end`.
   d. Select Outlet for Connection, enter **textLabel** for the outlet name, and click Connect.
   e. Select the Done button and control-drag to the interface source code just above the `@end`.
   f. Select Action for Connection, enter **done** for the action name, and click Connect.
4. Select File ➪ Save to save your project.

5. Modify the `ViewController` class.

    1. Modify the `ViewController.h` file to look like the following:

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

    @interface ViewController : UIViewController
    @property (strong, nonatomic) UIPopoverController
                                          *modalTypePopoverController;

    @property (weak, nonatomic) IBOutlet UIBarButtonItem *modalButton;

    - (IBAction)selectModalType:(id)sender;
    - (void)showModalView:(UISegmentedControl *)sender;
    - (void)dismissPopover:(UIPopoverController *)popoverController;

    @end
    ```

    2. Add the following imports to the `ViewController.m` file:

    ```
    #import "PopoverViewController.h"
    #import "ModalViewController.h"
    ```

    3. Add the following synthesize variables right below the `@implementation` section:

    ```
    @synthesize modalTypePopoverController = _modalTypePopoverController;
    @synthesize modalButton = _modalButton;
    ```

    4. Add the following above `[super viewDidUnload]` in the `viewDidUnload` method:

    ```
    [self setModalButton:nil];
    [self setModalTypePopoverController:nil];
    ```

    5. There is one action defined `selectModalType`:, enter the following code for this method:

    ```
    - (IBAction)selectModalType:(id)sender {
        UIPopoverController *popoverController =
                  [self modalTypePopoverController];
        if(popoverController == nil) {
            PopoverViewController *controller = [[self storyboard]
              instantiateViewControllerWithIdentifier:@"Lesson19"];
            [controller setDelegate:self];
            popoverController =
                [UIPopoverController alloc]
                     initWithContentViewController:controller];
            popoverController.popoverContentSize = CGSizeMake(320.0, 320.0);
            [self setModalTypePopoverController:popoverController];
        }
        [popoverController presentPopoverFromBarButtonItem:[self modalButton]
            permittedArrowDirections:UIPopoverArrowDirectionUp animated:YES];
    ```

    6. There is one delegate method called `showModalView:` that will be called from the `PopoverViewController` that brings up the appropriate modal view, depending on the choice selected in the popover:

    ```
    - (void)showModalView:(UISegmentedControl *)sender {
        UIPopoverController *popoverController =
            [self modalTypePopoverController];
    ```

```
    ModalViewController *modalView = [[self storyboard]
        instantiateViewControllerWithIdentifier:@"ModalView"];
    switch ([sender selectedSegmentIndex]) {
        case 0:
            [modalView setModalTransitionStyle:
                UIModalTransitionStyleCrossDissolve];
            [modalView setModalPresentationStyle:
                UIModalPresentationFullScreen];
            [modalView setText:@"UIModalPresentationFullScreen"];
        break;
        case 1:
            [modalView setModalTransitionStyle:
                UIModalTransitionStyleFlipHorizontal];
            [modalView setModalPresentationStyle:
                UIModalPresentationPageSheet];
            [modalView setText:@"UIModalPresentationPageSheet"];
        break;
        case 2:
            [modalView setModalTransitionStyle:
                UIModalTransitionStyleCoverVertical];
            [modalView setModalPresentationStyle:
                UIModalPresentationFormSheet];
            [modalView setText:@"UIModalPresentationFormSheet"];
        break;
        case 3:
            [modalView setModalTransitionStyle:
                UIModalTransitionStyleFlipHorizontal];
            [modalView setModalPresentationStyle:
                UIModalPresentationCurrentContext];
            [modalView setText:@"UIModalPresentationCurrentContext"];
        break;
    }
    [self dismissPopover:popoverController];
    [self presentModalViewController:modalView animated:YES];
}
```

7. The `dissmissPopover:` method is called after the selection has been made in preparation for the display of the modal view:

```
- (void)dismissPopover:(UIPopoverController *)popoverController {
    if (popoverController != nil) {
        [popoverController dismissPopoverAnimated:YES];
    }
       [self setModalTypePopoverController:nil];
}
```

6. Modify the `PopoverViewController` class.

   1. Modify the `PopoverViewController.h` file to look like the following;

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

@interface PopoverViewController : UIViewController

@property (strong, nonatomic) id delegate;
```

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

- (IBAction)showModalView:(id)sender;

@end
```

2. Add the following imports to the `PopoverViewController.m` file:

```
#import "ModalViewController.h"
```

3. Add the following synthesize variables right below the `@implementation` section:

```
@synthesize delegate = _delegate;
@synthesize modalTypeSegmentedController = _modalTypeSegmentedController;
```

4. Add the following above `[super viewDidUnload]` in the `viewDidUnload` method:

```
[self setModalTypeSegmentedController:nil];
```

5. There is one action defined `showModalView`:

```
- (IBAction)showModalView:(id)sender {
    [[self delegate] showModalView:[self modalTypeSegmentedController]];
}
```

7. Modify the `ModalViewController` class.

1. Modify the `ModalViewController.h` file to look like the following:

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

@interface ModalViewController : UIViewController

@property (strong, nonatomic) NSString *text;
@property (strong, nonatomic) IBOutlet UILabel *textLabel;

- (IBAction)done:(id)sender;

@end
```

2. Add the following synthesize variables right below the `@implementation` section:

```
@synthesize text = _text;
@synthesize textLabel = _textLabel;
```

3. Uncomment the `viewDidLoad` method and add the following below `[super viewDidLoad]`:

```
[[self textLabel] setText:[self text]];
```

4. Add the following above `[super viewDidUnload]` in the `viewDidUnload` method:

```
[self setText:nil];
[self setTextLabel:nil];
```

5. There is one action defined `done`:

```
- (IBAction)done:(id)sender {
    [self dismissModalViewControllerAnimated:YES];
}
```

8. Run the application.
    1. Select the iPad Simulator to run the application.
    2. Click the Run button from Xcode.
    3. When the application launches, tap the Modal button to present the popover.
    4. Select a style from the segmented control and tap Show.
    5. Tap Done to dismiss the modal view.

Please select Lesson 19 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

20 Tweeting with Twitter

Social media integration is not something that most apps can ignore. These days social media integration in apps is the norm rather than the exception. Fortunately for you, starting from iOS 5, Apple has built Twitter integration into the operating system. Sending tweets has never been easier!

In this lesson you learn to integrate the new Twitter framework in your iOS apps and allow the user to send tweets from your apps. You can build more complex Twitter clients that can access the entire Twitter API, but that topic is beyond the scope of this book.

The Twitter framework is not included in any of the standard iOS project templates that you use when creating a new project. You will need to add a reference to this framework manually. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, select the build target and switch to the Build Phases tab. Click the plus (+) button under the Link Binary With Libraries category and select Twitter.framework from the list of available frameworks (Figure 20-1).

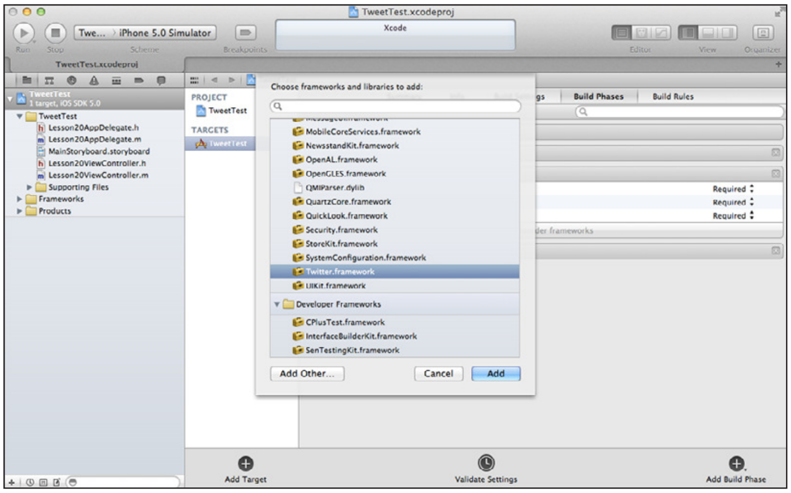


FIGURE 20-1

THE TWEET SHEET

The Twitter framework provides a tweet sheet that you should use in your apps if all you want is a simple "send tweet" feature. The tweet sheet is an instance of the `TWTwitterComposerViewController` class and provides a convenient user interface to allow the user to type a message, attach an image, and add the current location (Figure 20-2).

The keyboard is displayed automatically when the tweet sheet appears, and disappears automatically when the user presses the Send or Cancel buttons. Creating and displaying the tweet sheet is a simple matter of instantiating it and presenting it modally:

```
// create tweet sheet
TWTweetComposeViewController* tweetSheet =
              [[TWTweetComposeViewController alloc] init];

//show tweet sheet
[self presentModalViewController:tweetSheet animated:YES];
```

Typically, you will want to do this in an action method that is triggered when your user taps on a Tweet button in the user interface. Before you show the tweet sheet, you must check to see if the user has created a Twitter account on the system (Figure 20-3).

If the user has not created a Twitter account on the system, you may want to hide the Tweet button from your user interface entirely, or display an alert when the user taps it.

To check the availability of the Twitter service, use the `canSendTweet:` class method of the `TWTweetComposeViewController` class as follows:

```
BOOL serviceAvailability = [TWTweetComposeViewController canSendTweet];
```

You can set up the initial text displayed in the tweet sheet prior to displaying it by sending it the `setInitialText:` message:

```
- (BOOL) setInitialText:(NSString*)text;
```

This message takes one `NSString` argument that contains the text you want to set, and returns a Boolean value that contains the result of the operation. Common reasons why the operation may not be successful are:

1. The length of the message is longer than the 140-character limit set by Twitter.
2. You are trying to set the text in the tweet sheet after it has been displayed.

You can attach an image to the tweet sheet by sending it the `addImage:` message:

```
- (BOOL)addImage:(UIImage*)image
```

This message has one argument that is a `UIImage` object, and returns a Boolean result. The image is automatically resized and uploaded to the Twitter service by the framework. You must examine the return value to determine if the operation was successful.

To add a URL to the tweet sheet, send it the `addURL:` message:

```
- (BOOL)addURL:(NSURL*)url
```

To create an `NSURL` instance from a string, use code similar to the following:

```
NSURL *url = [NSURL URLWithString:@"http://www.asmtechnology.com "];
```

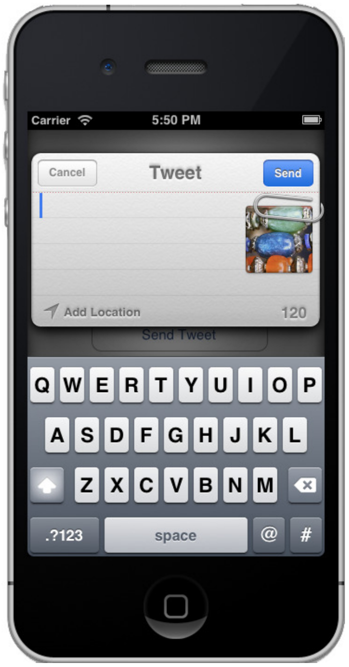


FIGURE 20-2

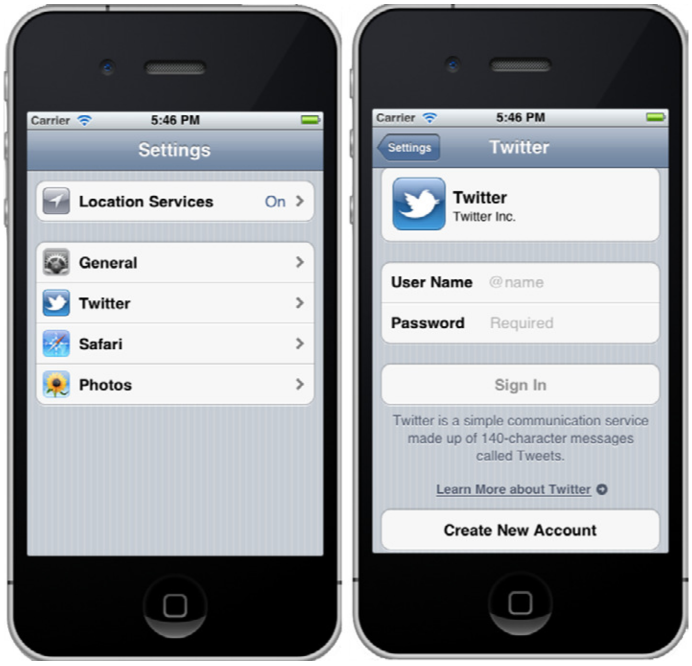


FIGURE 20-3

As with the `setInitialText:` and `addImage:` messages, the `addURL:` message returns a Boolean value indicating success or failure. It is important to note that images and URLs take up part of the 140-character limit imposed by the Twitter service.

You can provide an optional block completion handler that will be executed when the tweet has been sent. Assuming `tweetSheet` is an instance of a `TWTweetComposeViewController`, you can do this as follows:

```
tweetSheet.completionHandler = ^(TWTweetComposeViewControllerResult result)
{
        [self  dismissModalViewControllerAnimated:YES];
};
```

Within the block, you can examine the value of the `result` parameter to get more information on the result of the operation. The value of the `result` parameter depends on which button was pressed by the user, and can be either of:

- `TWTweetComposeViewControllerResultCancelled`
- `TWTweetComposeViewControllerResultDone`

You will need to dismiss the tweet sheet by sending the `dismissModalViewControllerAnimated:` message to the view controller. If you do not provide a block completion handler, the tweet sheet is dismissed automatically regardless of the result of the operation.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `TwitterTest`. The user interface of this app will consist of a single button titled Send Tweet and a label displaying service status. When the user taps the button, a tweet sheet will be displayed, with an image attached. The user can then type a message and send the tweet.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Open the storyboard in Interface Builder.
- Import image resources into the project.
- Add a `UIButton` and a `UILabel` instance to the default scene.
- Add a reference to the Twitter framework.
- Display Twitter service status in the label.
- Create a tweet sheet and attach an image to it.
- Present the tweet sheet modally.
- Create a `UIAlertView` instance in the action method and present it to the user.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 20 folder in the download.*

Hints

- To show the Object library, use the View ➪ Utilities ➪ Show Object Library menu item.
- Remember to add a reference to the Twitter framework.

Step-by-Step

1. Create a Single View Application in Xcode called `TwitterTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** TwitterTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson20

- **Define Family:** iPhone
- **Use Storyboard:** Checked
- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Import image resources into your project.
    1. Ensure the project navigator is visible and the TwitterTest project is selected and expanded.
    2. Right-click the TwitterTest group and select Add Files to TwitterTest from the context menu.
    3. Select the `Images` folder in this lesson's resources on the DVD.
    4. Ensure the Copy Items to Destination Group's Folder (if needed) option is selected in the dialog box.
    5. Click the Add button.
3. Add a `UILabel` instance to the default scene.
    1. Open the `MainStoryboard.storyboard` file in Interface Builder.
    2. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
    3. From the Object library, drag and drop a Label object onto the default scene.
    4. Use the Attributes inspector to set the Text attribute of the label to `Service Status`. To show the Attributes inspector, use the View ➪ Utilities ➪ Show Attributes Inspector menu item.
    5. Use the Attributes inspector to set the alignment of the label to Center.
    6. Size and position the label to X=35, Y=149, W=256, H=21.
    7. Using the assistant editor, create an outlet called `statusLabel` in the view controller class and connect it to the label.
4. Add a `UIButton` instance to the default scene.
    1. From the Object library, drag and drop a Round Rect Button object onto the scene.
    2. Double tap the button and set the text displayed in it to `Send Tweet`.

   3. Size and position the button to X=69, Y=202, W=190, H=37.
   4. Using the assistant editor, create a action method in the view controller class called `onSendTweet` and connect it with the Touch Up Inside event of the button.

5. Add a reference to the Twitter framework.
   1. Select the project node in the project navigator to display the settings page.
   2. On the settings page, select the build target and switch to the Build Phases tab.
   3. Click the plus (+) button under the Link Binary With Libraries category and select Twitter.framework from the list of available frameworks.
   4. Click the Add button.

6. Add the following code to the top of the `Lesson20ViewController.m` file:

   ```
   #import "Twitter/Twitter.h"
   ```

   after the line:

   ```
   #import "Lesson20ViewController.h"
   ```

7. Modify the `viewDidLoad` method in the `Lesson20ViewController.m` file to resemble the following:

   ```
   - (void)viewDidLoad
   {
       [super viewDidLoad];

       if ([TWTweetComposeViewController canSendTweet])
           statusLabel.text = @"Service Status: Can send tweets!";
       else
           statusLabel.text = @"Service Status: Unavailable!";
   }
   ```

8. Update the implementation of the `onSendTweet:` method in the `Lesson20ViewController.m` file.
   1. If the service is not available, show a suitable alert to the user by typing code similar to the following in the implementation of the `onSendTweet:` method:

      ```
      if ([TWTweetComposeViewController canSendTweet] == NO)
      {
          UIAlertView* serviceAlert = [[UIAlertView alloc]
                                       initWithTitle:@""
                                       message:@"Can't send tweets"
                                       delegate:nil
                                       cancelButtonTitle:@"Ok"
                                       otherButtonTitles:nil];
          [serviceAlert show];
          return;
      }
      ```

   2. Create the tweet sheet by typing the following code in the implementation of the `onSendTweet:` method, after the code from the previous step:

      ```
      TWTweetComposeViewController* tweetSheet =
      [[TWTweetComposeViewController alloc] init];
      ```

3. Load the `beads.png` file into a `UIImage` object and attach the image object to the tweet sheet by typing the following code after the code from the previous step:

   ```
   UIImage* attachment = [UIImage imageNamed:@"beads.png"];
   [tweetSheet addImage:attachment];
   ```

4. Finally, display the tweet sheet by typing the following code after the code from the previous step:

   ```
   [self presentModalViewController:tweetSheet animated:YES];
   ```

5. Your implementation of the `onSendTweet` method should now resemble the following:

   ```
   - (IBAction)onSendTweet:(id)sender
   {
       // display a alert if the service is not available.
       if ([TWTweetComposeViewController canSendTweet] == NO)
       {
           UIAlertView* serviceAlert = [[UIAlertView alloc]
                                 initWithTitle:@""
                                 message:@"Can't send tweets"
                                 delegate:nil
                                 cancelButtonTitle:@"Ok"
                                 otherButtonTitles:nil];
           [serviceAlert show];
           return;
       }

       // create the tweet sheet
       TWTweetComposeViewController* tweetSheet =
                 [[TWTweetComposeViewController alloc] init];

       // setup attachment
       UIImage* attachment = [UIImage imageNamed:@"beads.png"];
       [tweetSheet addImage:attachment];

       // show tweet sheet
       [self presentModalViewController:tweetSheet animated:YES];
   }
   ```

9. Test your app in the iOS Simulator.

   1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

   2. If the app indicates that the Twitter service is unavailable, create a Twitter account from the device settings app and try your app again.

Please select Lesson 20 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo, *to view the video that accompanies this lesson.*

21 Basic File Handling

Your iPhone applications always execute in a restricted environment on the device known as the application sandbox. Some of these restrictions affect where and how your application can store data. Several types of applications need the ability to store user-created data. Take, for instance, the Notes application. The notes that the user creates with this application need to be stored somewhere so that they are available when the application is restarted.

You can store data in several ways on an iOS device. In this lesson you learn to store data within files on the device.

THE IOS FILE SYSTEM

Each application is given a directory on the device's file system. The contents of this directory are private to the application, and cannot be read by other applications on the device. The information from some of Apple's applications such as Photos and Contacts can be accessed by third-party applications using frameworks in the iOS SDK.

Each application's directory has four locations into which you can store data:

- `Preferences`
- `Documents`
- `Caches`
- `tmp`

The first of these, `Preferences`, is not intended for direct file manipulation; however, the other three are. The most commonly used directories are the `Documents` and the `tmp` directories.

The `Documents` directory is the main location for storing application data. The contents of this directory can also be manipulated within iTunes (this is covered in Lesson 24). The `Caches` directory is used to store temporary files that need to persist between application launches. The `tmp` directory is used to store temporary files that do not need to persist between application launches.

Applications are responsible for cleaning up the contents of these directories, because storage space on a device is limited. The contents of the Caches and tmp directories are not backed up by iTunes.

To retrieve the path to the Documents and Caches directories, you can use a C function called NSSearchPathForDirectoriesInDomains. This function is part of the Core Foundation framework, and returns results in an array. You access only the first element of this array. For example, to get the path to the Documents directory, you would use:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory ,
                                                     NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
```

To obtain a path to the Caches directory, you should specify NSCachesDirectory as the first argument to the NSSearchPathForDirectoriesInDomains function.

To retrieve a path to the tmp directory, you need to use another Core Foundation function, NSTemporaryDirectory, as follows:

```
NSString* tmpDir = NSTemporaryDirectory();
```

Once you have a path to one of these standard directories, you can append a filename to it to refer to a specific file in that directory using the stringByAppendingPathComponent class method of the NSString class. For example, if you wanted to access the file myFile.dat in the Documents directory, you would use:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory ,
                                                     NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];

NSString* filePath = [documentsDir
                      stringByAppendingPathComponent:@"myFile.dat"];
```

INTRODUCING THE NSFILEMANAGER CLASS

The NSFileManager class provides several useful methods that allow you to manipulate files in your application's directories. NSFileManager is a singleton object—only one instance is created during the lifetime of your application. To access this one shared instance, use the defaultManager class method as follows:

```
NSFileManager* fileManager = [NSFileManager defaultManager];
```

Once you have a reference to an NSFileManager instance, you can use it to check whether a file exists by using the fileExistsAtPath: method as follows:

```
BOOL fileExists = [fileManager fileExistsAtPath:filePath];
```

To copy a file from one directory to another, use the copyItemAtPath:toPath:error: method. This method requires you to provide a source file path, a destination file path, and a variable in which detailed error information will be provided. If the operation succeeds, the method returns YES. If it fails, it returns NO, and you can get more details by examining the NSError object returned in the third parameter. The following example shows how you can use this method to copy a file from the tmp directory to the Documents directory:

```
// source file (in temporary directory)
NSString* tmpDir = NSTemporaryDirectory();
NSString* srcFilePath = [tmpDir
                        stringByAppendingPathComponent:@"myFile.dat"];

// destination file (in documents directory)
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory ,
                                                    NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
NSString* dstnFilePath = [documentsDir
                         stringByAppendingPathComponent:@"myFile.dat"];

// copy if destination file does not exist!
NSFileManager* fileManager = [NSFileManager defaultManager];
if ([fileManager fileExistsAtPath:dstnFilePath] == NO)
{
    NSError* error;
    BOOL success = [fileManager copyItemAtPath:srcFilePath
                                toPath:dstnFilePath
                                error:&error];
}
```

OBJECT SERIALIZATION

A typical application has a lot of objects in memory, and some of these objects are likely to represent the data needed by the application to perform its functions. Wouldn't it be nice if you could just save these data objects to a file, and load them the next time the app was launched? Well, the good news is that you can.

The concept of storing an object to a file is known as serialization, and the reverse is known as de-serialization. In the Cocoa universe this is also known as object archiving. Two main components are needed to perform object archiving:

- An NSCoder object that can encode and decode objects.
- Objects that can be encoded or decoded by an NSCoder instance. These must implement the NSCoding protocol.

The NSCoding protocol is very simple, and contains just two methods, encodeWithCoder: and initWithCoder:, corresponding to the encoding and decoding process, respectively.

Say you have an application that works with phone numbers and contact information. Your data class could resemble:

```
@interface ContactData : NSObject <NSCoding>
@property (strong, nonatomic) NSString* contactName;
@property (strong, nonatomic) NSString* phoneNumber;
@end
```

To archive instances of this class successfully, you will need to implement the `encodeWithCoder:` and `initWithCoding:` methods in this class. Begin with the `encodeWithCoder:` method.

In the implementation of this method you will need to first decide which variables you want to archive. Each instance variable that you encode must either be a primitive data type or an instance of an object that conforms to the `NSCoding` protocol. Fortunately, most of the Cocoa Touch objects implement `NSCoding`.

The encoding process requires a unique key for each instance variable that you want to archive. Your `encodeWithCoder:` method for this class will probably resemble:

```
- (void)encodeWithCoder:(NSCoder *)encoder {
    [encoder encodeObject:contactName forKey:@"name"];
    [encoder encodeObject:phoneNumber forKey:@"phone"];
}
```

The reverse process is carried out in the `initWithCoder:` method. In this class, decoding would be implemented as follows:

```
- (id)initWithCoder:(NSCoder *)decoder {
    if (self = [super init])
    {
        contactName = [decoder decodeObjectForKey:@"name"];
        phoneNumber = [decoder decodeObjectForKey:@"phone"];
    }
    return self;
}
```

Now that your `ContactData` class is `NSCoding` compliant, you can save an entire array of `ContactData` objects to a file. Assuming `arrayOfContacts` is an `NSArray` instance that contains a few `ContactData` objects and you want to save the entire array into a file in the `Documents` directory called `ContactData.dat`, you can use the following code:

```
[NSKeyedArchiver archiveRootObject:arrayOfContacts toFile:filePath];
```

To read back the individual `ContactData` objects into the array, you can use the following code:

```
arrayOfContacts = [NSKeyedUnarchiver unarchiveObjectWithFile:filePath];
```

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `FileTest` that saves an array of `NSCoding`-compliant objects to a file in the `Documents` directory and reads them back.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Open the storyboard in Interface Builder.
- Add a `UIButton` instance to the default scene.
- Create an Objective-C class called `ContactData` that is `NSCoding` compliant.

- Create an `NSArray` instance in the view controller class that will store `ContactData` instances.
- In the `viewDidLoad` method, load `ContactData` objects from a file, if it exists. If the file does not exist, create fresh `ContactData` instances and insert them into the array.
- When the button is pressed, archive the array of objects to a file.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 21 folder in the download.*

Hints

- To archive an object ensure the class implements the `NSCoding` protocol.

Step-by-Step

1. Create a Single View Application in Xcode called `FileTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** FileTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson21
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Create an `NSObject`-derived class called `ContactData`.
    1. In Xcode, make sure the project navigator is visible.
    2. Right-click the `FileTest` group and select New File from the popup menu.

   3. Select the Objective-C Class template for the new class.
   4. Name the class `ContactData` and make it a subclass of `NSObject`.
   5. Click the Next button, accept the default location for the file, and click Save.

3. Add member variables to the `ContactData` class.

   1. Modify the `ContactData.h` file to resemble the following:

```
@interface ContactData : NSObject <NSCoding>
@property (strong, nonatomic) NSString* contactName;
@property (strong, nonatomic) NSString* phoneNumber;
@end
```

   2. Modify the `ContactData.m` file to resemble the following:

```
@implementation ContactData
@synthesize contactName;
@synthesize phoneNumber;
@end
```

4. Ensure the `ContactData` class implements the `encodeWithCoder:` and `initWithCoder:` methods.

   1. Implement the `encodeWithCoder:` method as follows:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [encoder encodeObject:contactName forKey:@"name"];
    [encoder encodeObject:phoneNumber forKey:@"phone"];
}
```

   2. Implement the `initWithCoder:` method as follows:

```
- (id)initWithCoder:(NSCoder *)decoder
{
    if (self = [super init])
    {
        contactName = [decoder decodeObjectForKey:@"name"];
        phoneNumber = [decoder decodeObjectForKey:@"phone"];
    }
    return self;
}
```

5. Add a `UIButton` instance to the default scene.

   1. Open the `MainStoryboard.storyboard` file in Interface Builder.
   2. Ensure the Object library is visible. To show it, use the View ➪ Utilities ➪ Show Object Library menu item.
   3. From the Object library, drag and drop a Round Rect Button object onto the scene.
   4. Double tap the button and set the text displayed in it to `Save Objects To File`.
   5. Size and position the button to X=10, Y=30, W=300, H=37.
   6. Using the assistant editor, create an action method in the view controller class called `onSaveToFile:` and connect it with the Touch Up Inside event of the button.

6. Add an `NSArray` instance variable to the view controller class.

    1. Modify the interface of the view controller class to resemble the following:

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

    @interface Lesson21ViewController : UIViewController
    @property (nonatomic, strong) NSArray* arrayOfContacts;

    - (IBAction)onSaveToFile:(id)sender;
    @end
    ```

    2. Synthesize the property variable in the `Lesson21ViewController.m` file by adding the following line:

    ```
    @synthesize arrayOfContacts;
    ```

    after the line:

    ```
    @implementation Lesson21ViewController
    ```

7. Load `ContactData` objects from a file in the `viewDidLoad` method of the view controller class.

    1. Import the interface of the `ContactData` class at the top of the `Lesson21ViewController.m` file.

    2. Ensure the `viewDidLoad` method of the view controller class resembles the following:

    ```
    - (void)viewDidLoad
    {
        [super viewDidLoad];
        // target file in Documents directory
        NSArray* paths = NSSearchPathForDirectoriesInDomains
               (NSDocumentDirectory, NSUserDomainMask, YES);
        NSString *documentsDir = [paths objectAtIndex:0];
        NSString* filePath = [documentsDir
               stringByAppendingPathComponent:@"ContactData.dat"];

        // if file does not exist, show error message.
        if ([[NSFileManager defaultManager] fileExistsAtPath:filePath] == NO)
        {
            UIAlertView* errorMessage = [[UIAlertView alloc]
                            initWithTitle:@"ContactData.dat not found!"
                                         message:@"Creating objects..."
                                         delegate:nil
                                         cancelButtonTitle:@"Ok"
                                         otherButtonTitles:nil];
            [errorMessage show];

            ContactData* c1 = [[ContactData alloc] init];
            c1.contactName = @"Peter Kramer";
            c1.phoneNumber = @"44 79830 11460";

            ContactData* c2 = [[ContactData alloc] init];
            c2.contactName = @"Mark Andrews";
            c2.phoneNumber = @"44 79110 07491";

            arrayOfContacts = [NSArray arrayWithObjects:c1, c2, nil];
        }
    ```

```
        // if file exists, then show how many objects were loaded.
        else
        {
              arrayOfContacts = [NSKeyedUnarchiver
                         unarchiveObjectWithFile:filePath];

              NSString* messageText = [NSString
                         stringWithFormat:@"Loaded %d objects",
                                          [arrayofContacts count]];
            UIAlertView* message = [[UIAlertView alloc]
                                     initWithTitle:@"ContactData.dat found!"
                                     message:messageText
                                     delegate:nil
                                     cancelButtonTitle:@"Ok"
                                     otherButtonTitles:nil];

            [message show];
        }
    }
```

8. Implement the `onSaveToFile:` method in the `Lesson21ViewController.m` class as follows:

```
- (IBAction)onSaveToFile:(id)sender
{
    // target file in Documents directory
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                   NSUserDomainMask, YES);
    NSString *documentsDir = [paths objectAtIndex:0];
    NSString* filePath = [documentsDir
                      stringByAppendingPathComponent:@"ContactData.dat"];

    // save to file
    BOOL result = [NSKeyedArchiver archiveRootObject:arrayOfContacts
                                          toFile:filePath];

    if (result == YES)
    {
        UIAlertView* message = [[UIAlertView alloc]
                            initWithTitle:@"File has been saved!"
                            message:@""
                            delegate:nil
                            cancelButtonTitle:@"Ok"
                            otherButtonTitles:nil];
        [message show];
    }
    else
    {
        UIAlertView* message = [[UIAlertView alloc]
                            initWithTitle:@"Error saving to file!"
                            message:@""
                            delegate:nil
                            cancelButtonTitle:@"Ok"
                            otherButtonTitles:nil];
```

```
            [message show];
        }

    }
```

9. Test your app in the iOS Simulator.

    1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
    2. The first time you run the app, you will get an alert message stating that the `ContactData.dat` file was not found. Tap the Ok button to dismiss the alert. The app will now create two `ContactData` instances and add them to the `arrayOfContacts` array.
    3. Now tap the Save Objects To File button.
    4. Click the Stop button in the Xcode toolbar, and then run the application again by clicking the Run button.
    5. This time, you will get an alert message stating that two objects were loaded from the `ContactData.dat` file.

Please select Lesson 21 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

22 Property Lists

iOS devices have the capability to store data locally. The data is organized into a name/value relationship and uses the standard XML format. Apple recommends that the decision as to when to use property lists to persist data should be limited to small amounts, preferably less than a few hundred kilobytes. For larger amounts, Apple offers Core Data as an alternative solution.

Property lists are used quite frequently in standard Apple applications. For example, the application's user settings are stored in a property list. The Settings app requires a specific format, but the structure is a property list.

PROPERTY LIST TYPES

To convert the XML element values in a property list into an Objective C object, Apple has defined the following relationships between the Objective C object and the property list's XML element value, as shown in Table 22-1.

TABLE 22-1: Property List Types

DATA TYPE	XML TAG	IOS CLASS
array	`< array>`	`NSArray`
dictionary	`< dictionary>`	`NSDictionary`
string	`< string>`	`NSString`
data	`< data>`	`NSData`
date	`< date>`	`NSDate`
integer	`< integer>`	`NSNumber (intValue)`
floating point	`<real>`	`NSNumber (floatValue)`
Boolean	`<true/> or <false/>`	`NSNumber (boolValue) YES or NO`

CREATING PROPERTY LISTS

You can create property lists in two ways:

- Programmatically
- Using the property list editor

Programmatically

You can create a property list programmatically directly in Objective-C if all of the objects derive from the NSDictionary, NSArray, NSString, NSDate, NSData, or NSNumber class.

If they do not, the objects will have to adopt the NSCoder protocol and implement the following methods:

- encodeWithCoder
- initWithCoder

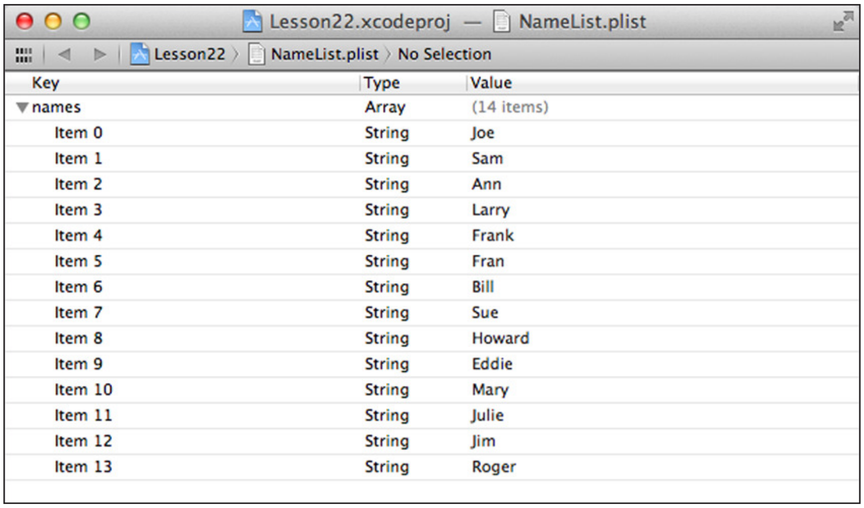
The following Transaction class contains both these methods to allow for the storage and retrieval of its data values into a property list:

```
#import "Transaction.h"

@implementation Transaction

@synthesize balance;
@synthesize items;

#pragma mark -
#pragma mark NSCoder methods

- (void)encodeWithCoder:(NSCoder *)coder {
    [coder encodeObject:[self balance] forKey:@"balance"];
    [coder encodeObject:[self items] forKey:@"items"];
}

- (id)initWithCoder:(NSCoder *)coder {
    if (self = [super init]) {
             [self setBalance:[coder decodeObjectForKey:@"balance"]];
             [self setItems:[coder decodeObjectForKey:@"items"]];
    }
    return self;
}

@end
```

Notice that the order of the variables in the encodeWithCoder *method,* balance *and* items*, is the exact order in the* initWithCoder *method. This is required to maintain data integrity.*

Property List Editor

While the plist file can be created and modified with a simple text editor, within Xcode, there is a GUI property list editor that allows for creation and modification of the file directly, with the advantage of a user friendly view. See Figure 22-1.


Lesson22.xcodeproj — NameList.plist
Lesson22 〉 NameList.plist 〉 No Selection

Key	Type	Value
▼names	Array	(14 items)
Item 0	String	Joe
Item 1	String	Sam
Item 2	String	Ann
Item 3	String	Larry
Item 4	String	Frank
Item 5	String	Fran
Item 6	String	Bill
Item 7	String	Sue
Item 8	String	Howard
Item 9	String	Eddie
Item 10	String	Mary
Item 11	String	Julie
Item 12	String	Jim
Item 13	String	Roger

FIGURE 22-1

TRY IT

In this Try It, you implement a single view application that reads a property a list of names into a dynamic prototype table view. On selection of a specific name, the name is displayed in an alert.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson22 folder in the download.*

Lesson Requirements

- Create an Xcode project using the Single View Application template.
- Create a storyboard including just a root view controller.
- Implement a dynamic prototype table view.
- Respond to the selection of a name by displaying the selected name in an alert view.

Hints

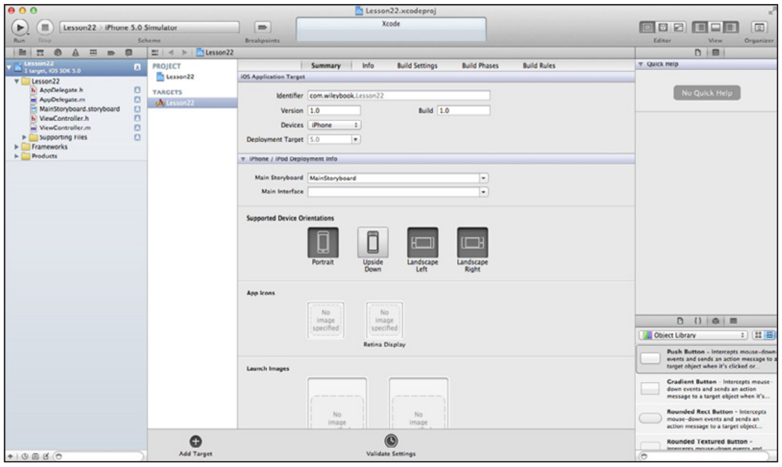
- Because this application uses storyboards instead of xib files, remember to have the Use Storyboard option checked at project creation.
- One array representing the contents of a property list will be used for table view cell population using the names stored for the cell title.
- The table view selection will be handled by the delegate method `tableView:didSelectRowAtIndexPath:`.

Step-by-Step

1. Create a Single View Application.
    1. Launch Xcode.
    2. Create your new iOS project.
        a. To create a new project, select Create a New Xcode Project.
        b. On the left under iOS, select Application.
        c. Select Single View Application from the template list and click Next.
        d. Choose the following options for your project:
            - **Product Name:** Lesson22
            - **Company Identifier:** com.wileybook
            - **Class Prefix:** leave blank
            - **Device Family:** iPhone
            - **Use Storyboard:** Checked
            - **Use Automatic Reference Counting:** Checked
            - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

        e. Select the location on your computer where the project will be saved and select Create.
        f. Your Xcode project has been created as shown in Figure 22-2.



iOS Application Target

Identifier com.wileybook.Lesson22
Version 1.0
Build 1.0
Devices iPhone
Deployment Target 5.0

iPhone / iPod Deployment Info

Main Storyboard MainStoryboard
Main Interface

Push Button – Intercepts mouse-down events and sends an action message to a target object when it's clicked or...
Gradient Button – Intercepts mouse-down events and sends an action message to a target object when it's...
Rounded Rect Button – Intercepts mouse-down events and sends an action message to a target object...
Rounded Textured Button – Intercepts mouse-down events and...

FIGURE 22-2

2. Design the user interface.
    1. On the left, select `MainStoryboard.storyboard`.
    2. On the right, select the third button in the View section to display the Utilities view.
    3. To delete the initial view controller from the storyboard:
        a. Select the bottom bar of the view controller in the storyboard and tap the Delete key to remove it.

> *It is important to make sure the default view controller is completely removed from the storyboard. Your detail view controller must be completely removed from the storyboard after this step.*

        b. Select the `ViewController.h` and `ViewController.m` files and delete and remove the files from the project.
    4. To add the new view controller to the storyboard:
        a. Select the Lesson22 folder.
        b. Select File ➪ New ➪ New File from the Xcode menu.
        c. On the left under iOS, select Cocoa Touch.
        d. Select UIViewController from the template list and click Next.
        e. Choose the following options for your new file:
            ➤ **Class:** ViewController
            ➤ **Subclass of:** UITableViewController

- ➤ **Deselect:** Targeted for iPad
- ➤ **Deselect:** With XIB for user interface

f. Click Next.

g. Click Create to save the class in your project folder.

5. To add the table view controller to the storyboard:

   a. On the left, select `MainStoryboard.storyboard`.

   b. Drag a `Table View Controller` from the Object Library and add it to the storyboard.

   c. Select the Identity Inspector and enter **ViewController** for the class.

6. To create the `NameList.plist` file:

   a. Select the Lesson22 folder.

   b. Select File ➪ New ➪ New File from the Xcode menu.

   c. On the left under iOS, select Resource.

   d. Select Property List from the template list and click Next.

   e. Enter **NameList** for the filename.

   f. Click Next.

   g. Click Create to save the class in your project folder.

7. Select File ➪ Save to save your project.

3. Create the property list editor, then select `NameList.plist`, control-click Add Row, and enter names in the main window editor as shown in Figure 22-3.

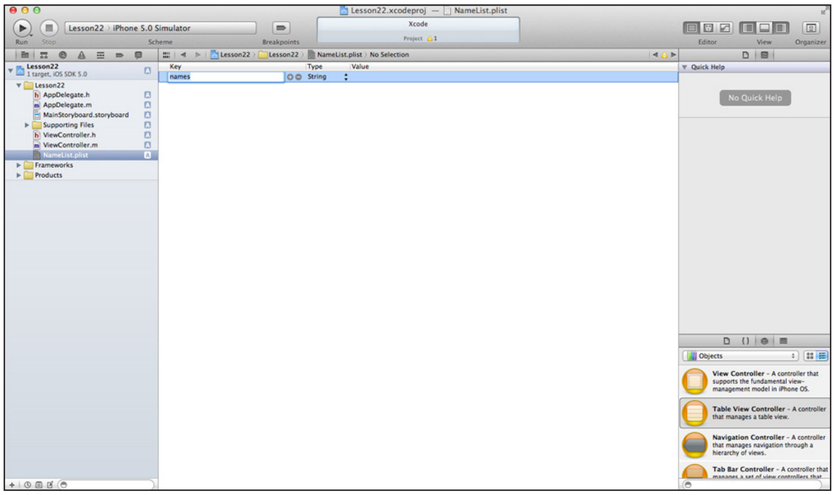

Lesson22.xcodeproj — NameList.plist
Lesson22 › iPhone 5.0 Simulator
Xcode
Lesson22
AppDelegate.h
AppDelegate.m
MainStoryboard.storyboard
Supporting Files
ViewController.h
ViewController.m
NameList.plist
Frameworks
Products
Key
Type
Value
names
String
Quick Help
No Quick Help
Objects
View Controller - A controller that supports the fundamental view-management model in iPhone OS.
Table View Controller - A controller that manages a table view.
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Tab Bar Controller - A controller that

FIGURE 22-3

4. Add names to the property list using the property list editor.
    1. Under the Type column, select Array. In the Key column, click the triangle so it points down.
    2. Hit the return key and begin entering your list of names.
    3. Hit the return key twice after entering the name to create another column to enter the next name.
    4. Continue this until 15 names are entered.
    5. Select File ➪ Save to save your project.
5. Add the following to the `ViewController.h` file before the `@end` statement:

```
@property (strong, nonatomic) NSArray *nameArray;

#pragma mark - Alert methods

- (void)alert:(NSString *)aMessage;

#pragma mark - Property List methods

- (NSArray *)readFromPropertyList:(NSString *)filename;
```

6. Modify the `ViewController.m` file using the following:
    1. Add the following synthesize variables right below the `@implementation` section:

    ```
    @synthesize nameArray;
    ```

    2. Uncomment the `viewDidLoad` method and add the following below `[super viewDidLoad];`:

    ```
     [self setNameArray:[self readFromPropertyList:@"NameList"]];
    ```

    3. Add the following above `[super viewDidUnload]` in the `viewDidUnload` method:

    ```
    [self setNameArray:nil];
    ```

    4. There is only one section to display the contacts. Complete the `numberOfSectionsInTableView:` method:

    ```
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
    {
        return 1;
    }
    ```

    5. There is one row for each contact in the dictionary. Complete the `tableView:numberOfRowsInSection:` method:

    ```
    - (NSInteger)tableView:(UITableView *)tableView
     numberOfRowsInSection:(NSInteger)section
    {
        return [[self nameArray] count];
    }
    ```

6. For each row, the name in the list is displayed. Complete the `tableView:cellForRowAtIndexPath:` method:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray *names = [[self nameArray]
        sortedArrayUsingSelector:
             @selector(localizedCaseInsensitiveCompare:)];
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    [[cell textLabel] setText:[names objectAtIndex:[indexPath row]]];
    return cell;
}
```

7. When a row is selected, the `tableView: didSelectRowAtIndexPath:` method is launched and the name from the selected row is retrieved and displayed in an alert:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    NSArray *names = [[self nameArray]
         sortedArrayUsingSelector:
            @selector(localizedCaseInsensitiveCompare:)];
    [self alert:[names objectAtIndex:[indexPath row]]];
}
```

The alert uses the UIAlertView to display the name selected:

```
- (void)alert:(NSString *)aMessage {
      UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Lesson 22"
                                                     message:aMessage
                                                     delegate:self
                                             cancelButtonTitle:nil
                                             otherButtonTitles:@"OK", nil];
      [alert show];
}
```

7. Load the property list from local storage. The `readFromPropertyList` method is called from the `viewDidLoad` method. Add this method above the `@end`:

```
- (NSArray *)readFromPropertyList:(NSString *)filename {
    NSString *errorDesc = nil;
    NSPropertyListFormat format;
    NSString *plistPath = [[NSBundle mainBundle] pathForResource:filename
                                                          ofType:@"plist"];
    NSData *plistXML = [[NSFileManager defaultManager]
                          contentsAtPath:plistPath];
    NSDictionary *temp = (NSDictionary *)[NSPropertyListSerialization
```

```
                propertyListFromData:plistXML
                    mutabilityOption:NSPropertyListMutableContainersAndLeaves
                               format:&format errorDescription:&errorDesc];
    if (!temp) {
        NSLog(@"%s at line %d with message: %@", __FUNCTION__,
                        __LINE__, errorDesc);
    }
    return [temp objectForKey:@"names"];
}
```

8. Run the application.
    1. Select the iPhone Simulator to run the application.
    2. Click the Run button from Xcode.
    3. When the application launches, a list of names appears.
    4. Select a specific contact to display the name in an alert.

Please select Lesson 22 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

23 Application Settings

Most applications that perform complex tasks will at some point need to allow users to customize the applications' operation to suit their specific needs. These customizable options are usually referred to as *application preferences* or *application settings*. iOS applications can either expose their preferences within Apple's Settings application, or provide a user interface within the application where the user can customize them appropriately.

To integrate your application's preferences with Apple's Settings application, your application must include a `Settings.bundle` file. A settings bundle file lets you declare the preferences in your application as a property list and the Settings application provides the user interface for editing those preferences.

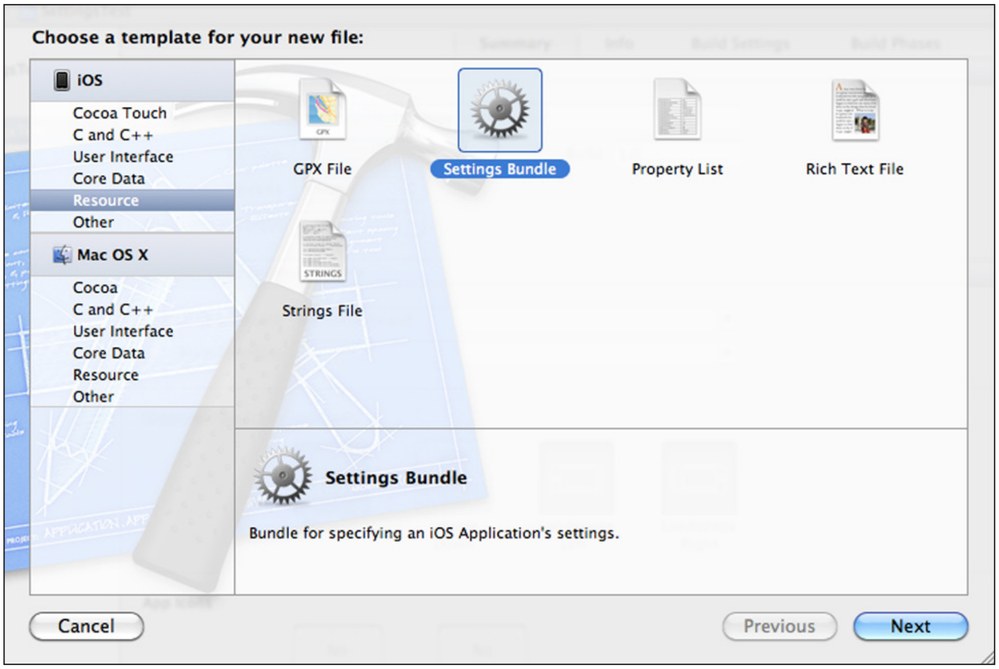
Keep in mind that to access the Settings application your users will have to first quit your application if they were using it. In this lesson, you learn to create this file and use it to expose system preferences.

ADDING A SETTINGS BUNDLE

To add a `Settings.bundle` file to your application, right-click your application's group in the project navigator and select New File from the context menu. Select the Settings Bundle file type from the iOS Resource section of the dialog box (Figure 23-1).

When the Settings application is launched on an iOS device, every third-party application is checked to see if it has a `Settings.bundle` file. For each application on the iOS device that has this file, its name and icon are added to a list on the main page of the Settings application (Figure 23-2).

Tapping on the icon will take the user to the particular application's settings page. By default, the Settings application will use an application's standard icon file when listing it. If you want to provide a custom icon to be used for your application in the Settings application, include a 29 × 29 pixel image called `Icon-Settings.png`.



Settings Bundle

Bundle for specifying an iOS Application's settings.

FIGURE 23-1

The Settings application can display application preferences in a series of hierarchical pages. Creating hierarchical settings pages is not covered in this lesson; however if you are interested in this topic you should read the "Preferences and Settings Programming Guide" available at: `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/UserDefaults/Introduction/Introduction.html#//apple_ref/doc/uid/10000059i-CH1-SW1`.

A settings bundle is actually a collection of files. To see the contents of the bundle simply click the triangle beside the `Settings.bundle` file in the project navigator (Figure 23-3).

Inside the settings bundle you will find a file named `Root.plist`. This file controls how your application's preferences will appear within the Settings application. Clicking the file opens it in the property list editor. When you do this you will see a table with three columns—`Key`, `Type`, and `Value`. This file contains two properties: an array called `Preference Items` and a string called `Strings Filename` (Figure 23-4).

Each preference that you want to expose to your users will be an entry in the `Preference Items` array. To see the contents of the `Preference Items` array, simply expand it within the property

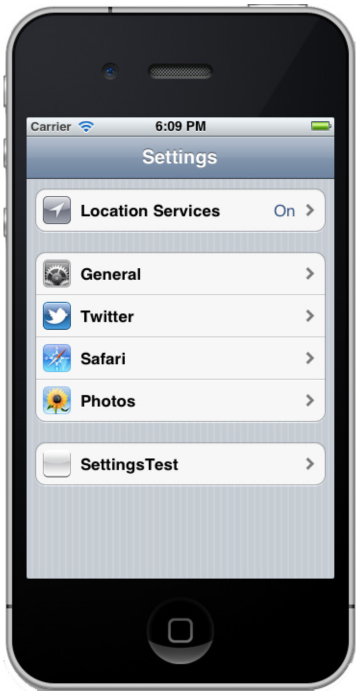


FIGURE 23-2

list editor. When you create a new settings bundle, this array contains four items by default (Figure 23-5). Each entry in the array is a dictionary of key-value pairs. Technically speaking, the `Preference Items` property is an array of dictionaries.

Each entry within the `Preference Items` array, being a dictionary, can have several key-value pairs, but you will always find four keys in each entry—`Title`, `Type`, `Identifier`, and `DefaultValue`.

The value of the `Title` key is used by the Settings application to label the preference when it is presented to the user. The value of the `Type` key determines what kind of preference value it is, and thus what user interface component will be used by the Settings application when presenting it. The value of the `Identifier` key contains a string that you can use to read the value of the preference in your Objective-C code. The value of the `DefaultValue` key contains the default value for the preference.

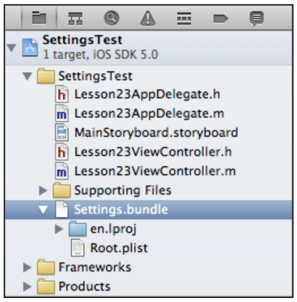
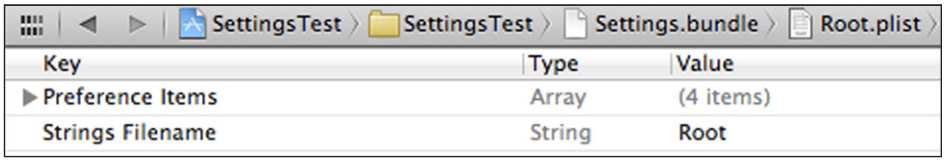
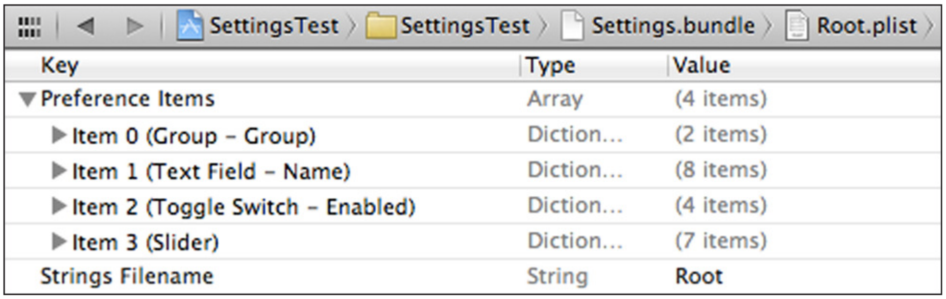


FIGURE 23-3



Key	Type	Value
▶ Preference Items	Array	(4 items)
Strings Filename	String	Root

FIGURE 23-4



Key	Type	Value
▼ Preference Items	Array	(4 items)
▶ Item 0 (Group - Group)	Diction...	(2 items)
▶ Item 1 (Text Field - Name)	Diction...	(8 items)
▶ Item 2 (Toggle Switch - Enabled)	Diction...	(4 items)
▶ Item 3 (Slider)	Diction...	(7 items)
Strings Filename	String	Root

FIGURE 23-5

The default settings bundle created by Xcode contains four entries is the `Preference Items` array:

➤ Group

➤ Text Field

➤ Toggle Switch

➤ Slider

If you were to run this app on an iOS device, and look at its settings page in the Settings application, you would see something similar to that shown in Figure 23-6.

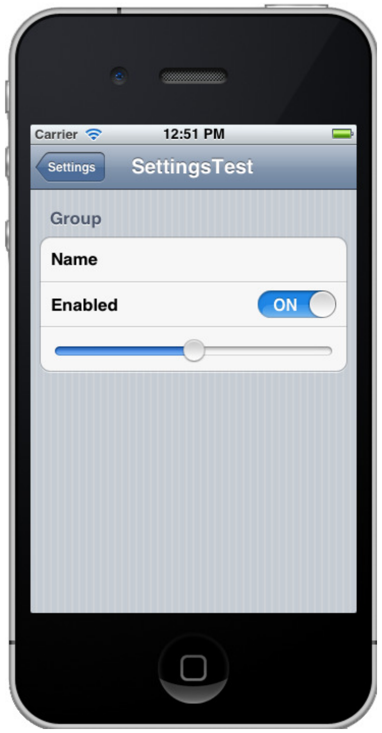


FIGURE 23-6

Table 23-1 describes the element types that can be used in the settings bundle.

TABLE 23-1: Preference Types

TYPE	DESCRIPTION
Text Field	An editable text field
Toggle Switch	On/Off toggle button
Title	A read-only text string
Slider	A slider to allow the user to select from a range of values
Multi Value	A list of values
Group	A logical group of preferences
Child Pane	Child preferences page, used to implement hierarchical preference pages

READING PREFERENCES WITH CODE

To read the value of a preference in a settings bundle from your code, you need to use an `NSUserDefaults` object. `NSUserDefaults` is part of the Core Foundation framework and provides a set of methods that allow you to manage application preferences. `NSUserDefaults` is a singleton class, and thus only one object should exist during the lifetime of an application. To get access to this one instance, use the following code:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
```

Recall that each preference within a settings bundle is represented by a dictionary of key-value pairs, and one of the four keys that each dictionary must contain is `Identifier`. To retrieve the value of a preference that has the identifier `user_name`, use the following code:

```
NSString *userName = [userDefaults stringForKey:@"user_name"];
```

This code assumes that the value being retrieved is a string. The `NSUserDefaults` class provides several methods that allow you to retrieve preference values of different data types, including:

- `boolForKey`
- `floatForKey`
- `doubleForKey`
- `integerForKey`

Although you have provided default values for the preferences in the settings bundle, these values will not be applied until the users launch the Settings application on their device after installing your application. To get around this problem, you should specify a default value for each of your preferences in code as well as the settings bundle.

You can then use methods in the `NSUserDefaults` class to ensure that the default values are applied only once regardless of whether your user launches the Settings application or your application first. To do this, you need to create a dictionary with the default values of each preference and use the `registerDefaults` and `synchronize` methods of the `NSUserDefaults` object as follows:

```
NSMutableDictionary* defaultsDict = [[NSMutableDictionary alloc] initWithCapacity:1];
[defaultsDict setObject:@"Paul Woods" forKey:@"user_name"];

[userDefaults registerDefaults:defaultsDict];
[userDefaults synchronize];
```

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `SettingsTest` that allows the user to specify a name and age value within the Settings application. Your application, when launched, will display this name and age.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a settings bundle to the application.
- Add two Text Field preferences to the settings bundle.
- Open the storyboard in the interface editor.
- Add two `UILabel` instances to the first scene.
- In the `viewDidLoad` method, read the preference values and display them in the labels.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. You can find them in the Lesson 23 folder in the download.

Hints

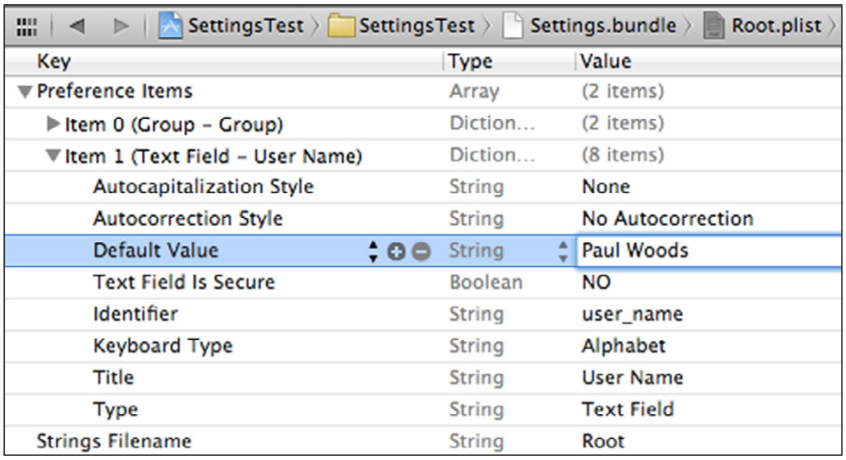
- To display your application's preferences in the Settings application, you must include a `Settings.bundle` file.
- To access the preference values specified by the user in the settings page from within your code, each preference must have a unique string identifier.

Step-by-Step

1. Create a Single View Application in Xcode called `SettingsTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** SettingsTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson23
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`, but you can use any unique identifier for your application.

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a `Settings.bundle` file to the project.
    1. Ensure the project navigator is visible.
    2. Right-click the `Settings Test` group and select New File from the context menu.
    3. Select the Settings Bundle template from the iOS Resources section. Save the file as `Settings.bundle`.
3. Edit the `Settings.bundle` file.
    1. Expand the `Settings.bundle` file in the project navigator and click the `Root.plist` file to edit it with the property editor.
    2. Expand the `Preference Items` property.
    3. Delete items 2 and 3. These are the Toggle Switch and Slider items, respectively. To delete an item, select it and hit the backspace key.
    4. Edit the Text Field preference.
        1. Expand the `Item 1 (Text Field – Name)` dictionary.
        2. Set the `Title` to **User Name**, `Identifier` to **user_name**, and `Default Value` to **Paul Woods** (Figure 23-7).



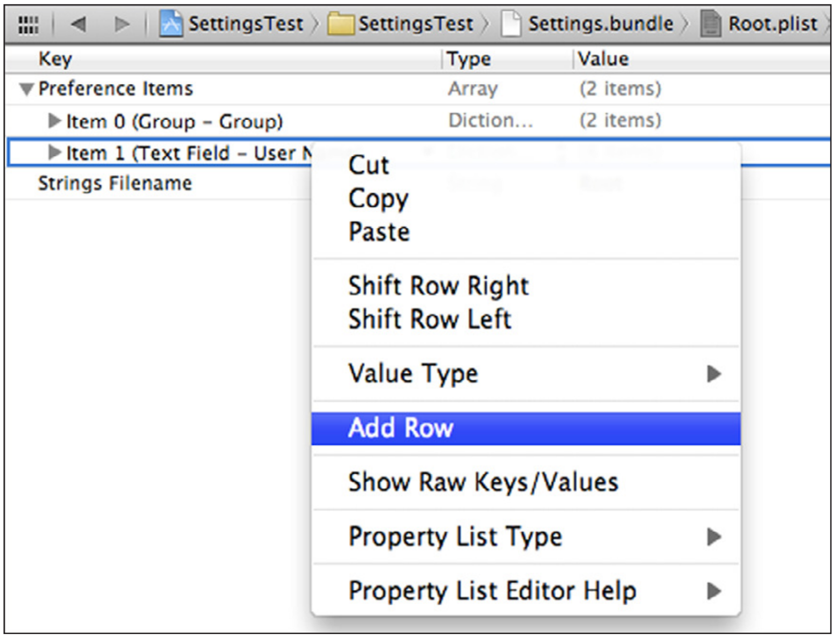
SettingsTest › SettingsTest › Settings.bundle › Root.plist ›

Key	Type	Value
▼ Preference Items	Array	(2 items)
▶ Item 0 (Group – Group)	Diction...	(2 items)
▼ Item 1 (Text Field – User Name)	Diction...	(8 items)
Autocapitalization Style	String	None
Autocorrection Style	String	No Autocorrection
Default Value	String	Paul Woods
Text Field Is Secure	Boolean	NO
Identifier	String	user_name
Keyboard Type	String	Alphabet
Title	String	User Name
Type	String	Text Field
Strings Filename	String	Root

FIGURE 23-7

    5. Add a new Text Field preference.

  1. Ensure the `Item 1 (Text Field - User Name)` dictionary is collapsed.
  2. Right-click the row corresponding to the `Item 1 (Text Field - User Name)` dictionary and select Add Row from the context menu (Figure 23-8).

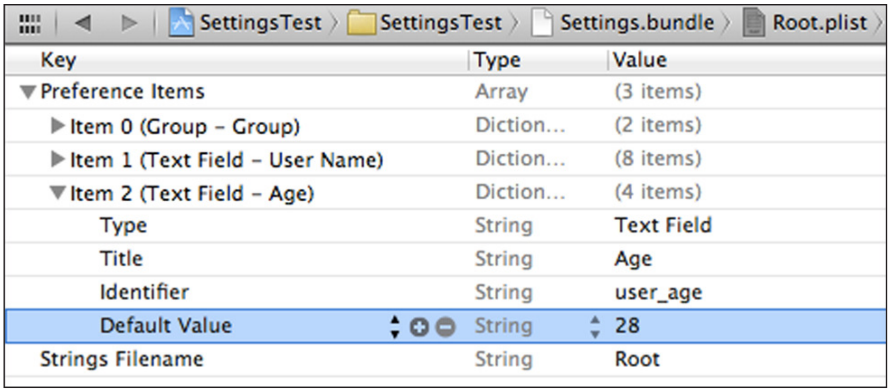


Key	Type	Value
▼ Preference Items	Array	(2 items)
▶ Item 0 (Group - Group)	Diction...	(2 items)
▶ Item 1 (Text Field - User N		
Strings Filename		

FIGURE 23-8

  3. Expand the newly added preference dictionary.
  4. Ensure the `Type` key is set to **Text Field**, `Title` is set to **Age**, and `Identifier` is set to **user_age**.
  5. Add a new key to the dictionary by right-clicking the last key (`Identifier`) and selecting Add Row from the context menu.
  6. Ensure the name of the new key is `Default Value` and the value of the key is 28 (Figure 23-9).

4. Add two `UILabel` instances to the storyboard.
   1. Open the `MainStoryboard.storyboard` file in the Xcode interface editor.
   2. From the Object library, drag and drop two `Label` objects onto the scene.
   3. Name the labels **Name** and **Age**, respectively.
   4. Size and position the Name label to X=28, Y=172, W=261, H=21.
   5. Size and position the Age label to X=28, Y=219, W=261, H=21.
   6. Using the assistant editor, create an outlet for each label in the view controller class, and name the outlets **nameLabel** and **ageLabel**, respectively.



SettingsTest › SettingsTest › Settings.bundle › Root.plist

Key	Type	Value
▼Preference Items	Array	(3 items)
▶Item 0 (Group – Group)	Diction...	(2 items)
▶Item 1 (Text Field – User Name)	Diction...	(8 items)
▼Item 2 (Text Field – Age)	Diction...	(4 items)
Type	String	Text Field
Title	String	Age
Identifier	String	user_age
Default Value	String	28
Strings Filename	String	Root

FIGURE 23-9

5. Read and display the preference values provided by the user in the Settings application by replacing the Replace the `viewDidLoad` method of the `Lesson23ViewController.m` file with the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // register defaults.
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
    NSMutableDictionary* defaultsDict = [[NSMutableDictionary alloc]
                                         initWithCapacity:2];
    [defaultsDict setObject:@"Paul Woods" forKey:@"user_name"];
    [defaultsDict setObject:@"28" forKey:@"user_age"];

    [userDefaults registerDefaults:defaultsDict];
    [userDefaults synchronize];

    // read preferences values and setup labels.
    nameLabel.text = [userDefaults stringForKey:@"user_name"];
    ageLabel.text = [userDefaults stringForKey:@"user_age"];
}
```

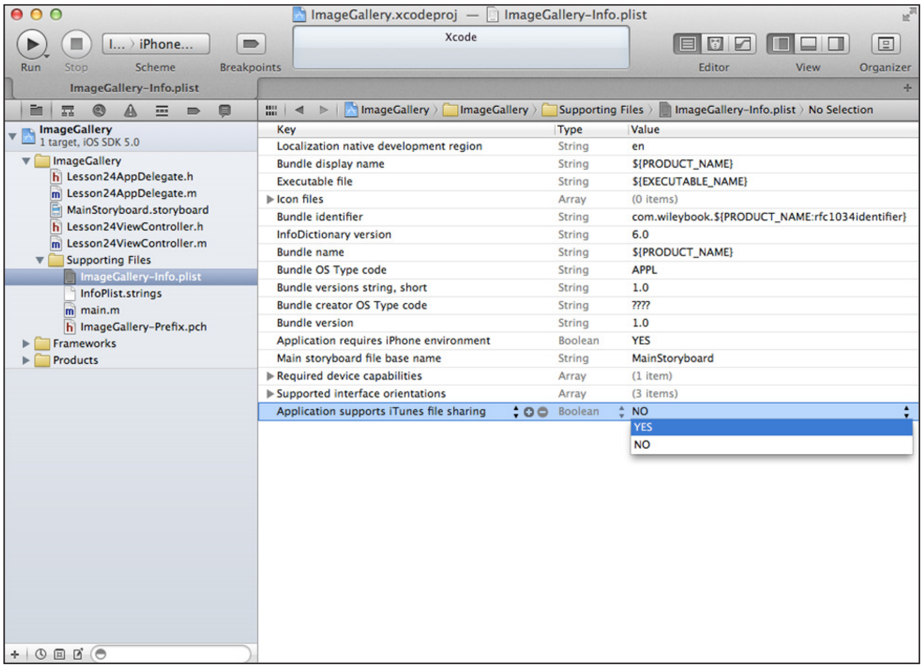
6. Test your app in the iOS Simulator.
    1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
    2. After changing preferences in the Settings application, ensure your application is not running in the background before launching it again. Building background-aware applications is covered in Lesson 38.

Please select Lesson 23 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

24 iTunes File Sharing Support

In Lesson 21 you learned to store data within files on the device. These files were created by your application in a restricted environment on the device known as the sandbox. In this lesson you learn to allow your users to modify the contents of one of the directories in your application's sandbox with iTunes.

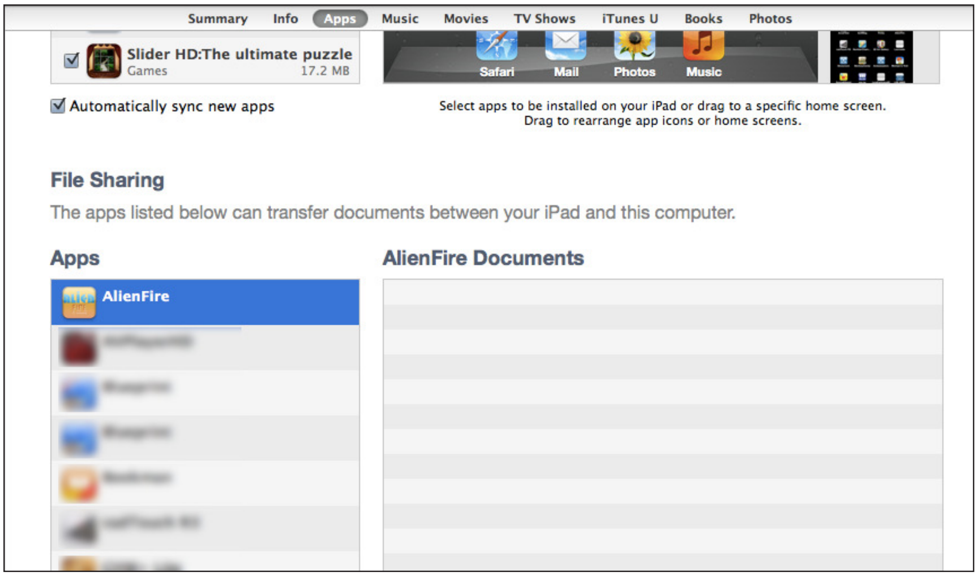
This feature is known as iTunes file sharing, and the first thing you need to do to enable it in your app is to add the `Application supports iTunes file sharing` key to the project's `info.plist` file. Set the value of this key to `YES`, as shown in Figure 24-1.



Key	Type	Value
Localization native development region	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	com.wileybook.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Main storyboard file base name	String	MainStoryboard
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
Application supports iTunes file sharing	Boolean	NO
		YES
		NO

FIGURE 24-1

When you add this key to the `info.plist` file, iTunes essentially displays the contents of your application's `Documents` directory to users when they go to the Apps section in iTunes and scroll to the bottom (Figure 24-2).



Select apps to be installed on your iPad or drag to a specific home screen.
Drag to rearrange app icons or home screens.

The apps listed below can transfer documents between your iPad and this computer.

FIGURE 24-2

In your application, you can enumerate the contents of the Documents directory using the `contentsOfDirectoryAtPath:error:` method of the `NSFileManager` class using code similar to the following:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                     NSUserDomainMask, YES);
NSString *documentsFolderPath = [paths objectAtIndex:0];

NSError* errVal;


NSArray* directoryList = [[NSFileManager defaultManager]
                     contentsOfDirectoryAtPath:documentsFolderPath
                     error:&errVal];
```

The first argument to this method is the path to the directory whose contents you want to enumerate. In this case, it would be the path to the `Documents` directory. The second parameter is used to retrieve information on any error that may have occurred in the process.

The `contentsOfDirectoryAtPath:error:` method returns an `NSArray` instance containing the filenames (excluding folder paths), which you can iterate through using a simple `for` loop:

```
for (int iX = 0; iX < [directoryList count]; iX++)
{
    // get file name
    NSString* fileName = (NSString*)[directoryList objectAtIndex:iX];
}
```

Exposing your application's `Documents` directory to your users in this way allows them to potentially drag and drop any kind of file in there, or delete anything that exists in that directory. It is unlikely that your application can handle any kind of file the user puts in the Documents directory, and thus, when processing the contents of the Documents directory it would be a good idea to check the extension of the file to determine if it is something that your application can handle. This is demonstrated in this lesson's Try It section.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `ImageGallery` that allows the user to navigate through an image gallery by using navigation buttons on the screen. You can use iTunes to modify the contents of the gallery.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Enable iTunes file sharing in the application.
- Create a simple storyboard-based user interface.
- When the application starts, read a list of image files in the `Documents` directory and display the first one.
- Implement a simple navigation strategy to allow a user to browse through the gallery using two buttons on the screen.
- Add a few images to the gallery using iTunes.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 24 folder in the download.*

Hints

- To enable iTunes file sharing in an application, you must add the `Application supports iTunes file sharing` key to the project's `info.plist` file.

Step-by-Step

1. Create a Single View Application in Xcode called `ImageGallery`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.

    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** ImageGallery
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson24
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add user interface elements to the default scene.
    1. Add a `UIImageView` instance to the default scene.
        1. Use the Object library to add an Image View to the default scene of the storyboard.
        2. Use the Size inspector to resize and position it at X = 0, Y = 0, Width = 320, Height = 460.
        3. Using the assistant editor, create an outlet in the view controller class called **`imageView`** and connect it to the scroll view.
    2. Add two `UIButton` instances to the default scene.
        1. Use the Object library to add two Round Rect Button instances to the scene, on top of the image view.
        2. Double-click the first button and set its title to `Previous`. Size and position it to X=20, Y=403, W=115, H=37.
        3. Double-click the second button and set its title to `Next`. Size and position it to X=186, Y=403, W=115, H=37.
        4. Use the assistant editor to create an action called `onPreviousImage` in the view controller class and connect it to the Touch Up Inside event of the `Previous` button.
        5. Use the assistant editor to create an action called `onNextImage` in the view controller class and connect it to the Touch Up Inside event of the `Next` button.
        6. Your scene should now resemble Figure 24-3.

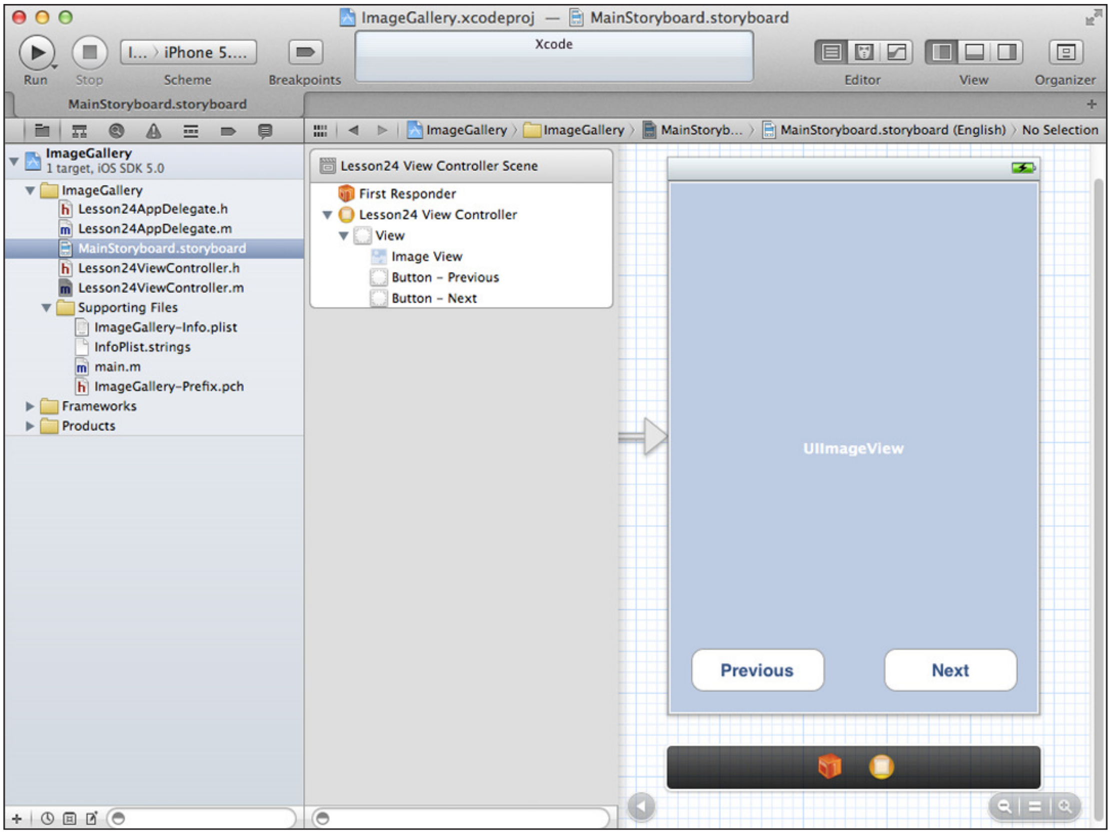


FIGURE 24-3

3. Add a new entry to the property list file called `Application supports iTunes file sharing`, and set its value to `YES`.

4. Add the following `@property` declarations to the `Lesson24ViewController.h` file.

```
@property (strong, nonatomic) NSMutableArray* imageFileNames;
@property (nonatomic) int currentImageIndex;
```

5. Update code in the `ViewController.m` file.

    1. Add the following `@synthesize` statements to the top of the file:

    ```
    @synthesize imageFileNames;
    @synthesize currentImageIndex;
    ```

    2. Replace the `viewDidLoad` method of the `Lesson24ViewController.m` file with the following code:

    ```
    - (void)viewDidLoad
    {
        [super viewDidLoad];
    ```

```
imageFileNames = [[NSMutableArray alloc] initWithCapacity:10];
currentImageIndex = 0;

// full path to Documents directory
NSArray *paths =NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES);
NSString *folderPath = [paths objectAtIndex:0];

NSError* errVal;
NSArray* directoryList = [[NSFileManager defaultManager]
                          contentsOfDirectoryAtPath:folderPath
                          error:&errVal];

for (int iX = 0; iX < [directoryList count]; iX++)
{
    // get file name
    NSString* fileName = (NSString*)[directoryList objectAtIndex:iX];

    // extract file extension
    NSArray* fileNameComponents = [fileName
                                   componentsSeparatedByString:@"."];
    if ([fileNameComponents count] < 2)
        continue;

    NSString* fileExtension = (NSString*)[fileNameComponents
    objectAtIndex:([fileNameComponents count] - 1)];

    if (([fileExtension isEqualToString:@"png"]) ||
        ([fileExtension isEqualToString:@"jpg"]))
    {
        [imageFileNames addObject:fileName];
    }
}

// show an alert that contains the number of readable
// image files found in the documents.
NSString* messageText = [NSString stringWithFormat:
@"Found %d usable files in the documents directory.",
                            [imageFileNames count]];

UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:@""
                                        message:messageText
                                               delegate:nil
                                    cancelButtonTitle:@"Ok"
                                    otherButtonTitles:nil];
[alertMessage show];

if ([imageFileNames count] > 0)
{
    NSString* imageFile = [imageFileNames
    objectAtIndex:currentImageIndex];

    NSString* sourceFile = [folderPath
stringByAppendingString:[NSString stringWithFormat:@"/%@",
                                                imageFile]];

        imageView.image = [[UIImage alloc]
        initWithContentsOfFile:sourceFile];
    }
}
```

3. Implement the `onPreviousImage:` method as follows:

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

    if ([imageFileNames count] == 0)
        return;

    if (currentImageIndex == 0)
        return;

    currentImageIndex--;

    // full path to Documents directory
    NSArray *paths = NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *folderPath = [paths objectAtIndex:0];

    // path to image file in the documents directory
    NSString* imageFile =
    [imageFileNames objectAtIndex:currentImageIndex];
NSString*  sourcefile = [folderPath stringByAppendingString:
                        [NSString stringWithFormat:@"/%@",
                        imageFile]];
    imageView.image = [[UIImage alloc] initWithContentsOfFile:sourceFile];

}
```

4. Implement the `onNextImage:` method as follows:

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

    if ([imageFileNames count] == 0)
        return;

    if (currentImageIndex == [imageFileNames count])
        return;

    currentImageIndex++;

    // full path to Documents directory
    NSArray *paths = NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *folderPath = [paths objectAtIndex:0];

    // path to image file in the documents directory
    NSString* imageFile =
    [imageFileNames objectAtIndex:currentImageIndex];
    NSString*  sourcefile = [folderPath stringByAppendingString:
                            [NSString stringWithFormat:@"/%@",
                                     imageFile]];
    imageView.image = [[UIImage alloc] initWithContentsOfFile:sourceFile];
}
```

6. Test your app on an iOS device.

    1. Connect your iOS device to your Mac, and use the Scheme/Target selector to select it (Figure 24-4).

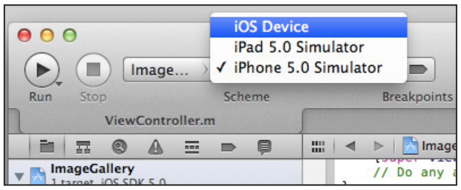


FIGURE 24-4

    2. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item. For more information on testing your apps on iOS devices, refer to Appendix D.
    3. When you run the app for the first time on the device, you will receive a message telling you that no readable images were found. The Next and Previous buttons will not work at this stage.
    4. Use iTunes to add a few images from the `Images` folder, which is included as part of this chapter's Try It on the DVD, into the application's `Documents` directory.
    5. Ensure your application is not running in the background before launching it again. Note how the application now detects the images you have added to its `Documents` directory with iTunes.

Please select Lesson 24 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`, *to view the video that accompanies this lesson.*

25 Introduction to iCloud Storage

iCloud Storage is a set of classes and services that enable you to share data between instances of your application running across different devices. In this lesson you learn to use the iCloud Storage APIs in your apps.

BASIC CONCEPTS

Apple's iCloud is a service that allows applications to synchronize data across devices. Your data is stored across a set of servers maintained by Apple and is made available to copies of your app across all iCloud-compatible devices. Changes made to this data by one instance of your application are automatically propagated to other instances.

From a developer's perspective, you need to use Apple's iCloud Storage APIs to interact with the iCloud service. These APIs enable you to store both documents and small amounts of key-value data.

This lesson does not cover key-value data storage. For more information on storing key-value data with iCloud, refer to the iCloud Storage section of the "iOS App Programming Guide," available at `http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iCloud/iCloud.html#//apple_ref/doc/uid/TP40007072-CH5-SW1`.

iCloud applications cannot be tested on the iOS Simulator, and to make the most of this lesson you should ideally have two iOS devices to test on. As of now, the iCloud Storage APIs are available to both iOS 5 and MacOS X developers.

In Lesson 21, you learned that each iOS application executes within a sandbox on the device and can store its data within subfolders of its private directory. iCloud conceptually extends this model and allows your applications to upload your data from its private directory to Apple's servers. This data then filters down to other iCloud-compatible devices on which copies of your application are running. Your application also receives notifications when a document has been created or updated by another copy of the application.

This synchronization is achieved by a background process (also known as a daemon) that runs on all iCloud-compatible devices. Figure 25-1 illustrates the iCloud architecture.

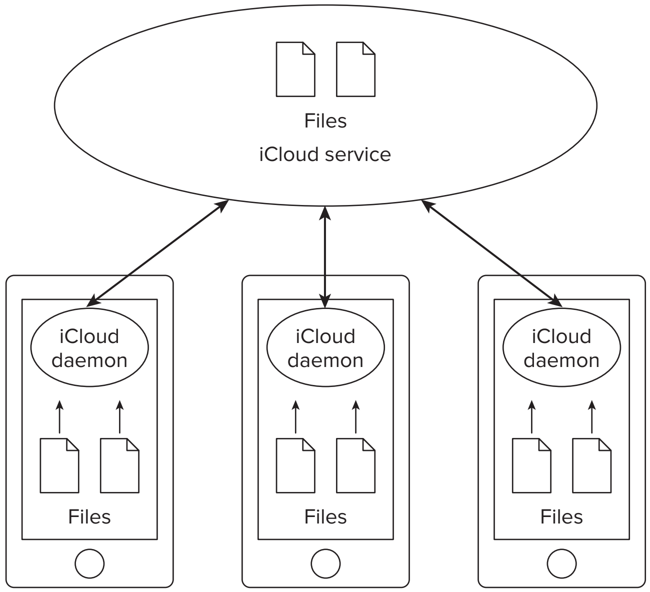


FIGURE 25-1

PREPARING TO USE THE ICLOUD STORAGE APIS

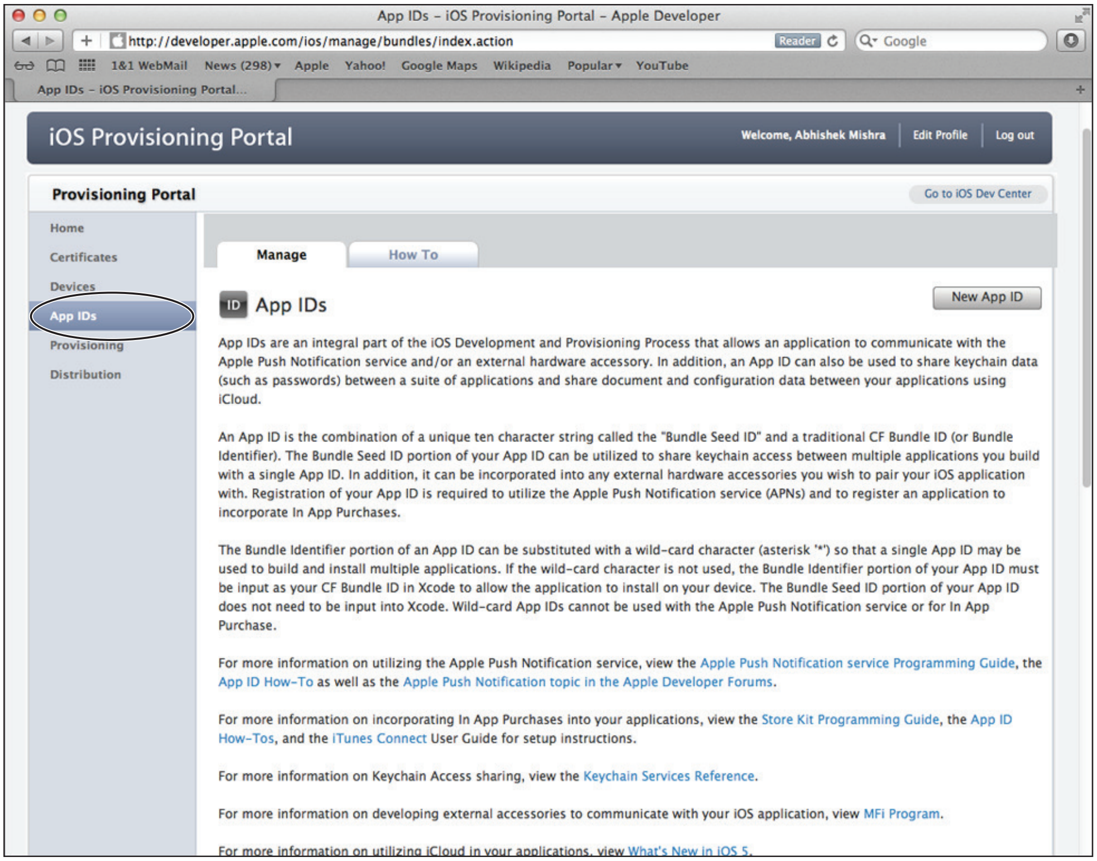
To use the iCloud Storage APIs in an application, you need to perform three steps:

1. Create an iCloud-enabled App ID.
2. Create an appropriate provisioning profile.
3. Enable appropriate entitlements in your Xcode project.

Create an iCloud-enabled App ID

To create an appropriate App ID, log in to your iOS developer account at `https://developer.apple.com/ios` and click the iOS Provisioning Portal link on the right side of the page. Within the Provisioning Portal, click the App IDs link in the menu on the left-hand side (Figure 25-2).

To create a new App ID, click the New App ID button on the top-right side. Provide a description of the new App ID in the Description field. Select Use Team ID in the Bundle Seed ID drop-down and provide a unique identifier in the Bundle Identifier field that ends in the name of the Xcode project you are going to create (or have created).



App IDs are an integral part of the iOS Development and Provisioning Process that allows an application to communicate with the Apple Push Notification service and/or an external hardware accessory. In addition, an App ID can also be used to share keychain data (such as passwords) between a suite of applications and share document and configuration data between your applications using iCloud.

An App ID is the combination of a unique ten character string called the "Bundle Seed ID" and a traditional CF Bundle ID (or Bundle Identifier). The Bundle Seed ID portion of your App ID can be utilized to share keychain access between multiple applications you build with a single App ID. In addition, it can be incorporated into any external hardware accessories you wish to pair your iOS application with. Registration of your App ID is required to utilize the Apple Push Notification service (APNs) and to register an application to incorporate In App Purchases.

The Bundle Identifier portion of an App ID can be substituted with a wild-card character (asterisk '*') so that a single App ID may be used to build and install multiple applications. If the wild-card character is not used, the Bundle Identifier portion of your App ID must be input as your CF Bundle ID in Xcode to allow the application to install on your device. The Bundle Seed ID portion of your App ID does not need to be input into Xcode. Wild-card App IDs cannot be used with the Apple Push Notification service or for In App Purchase.

For more information on utilizing the Apple Push Notification service, view the Apple Push Notification service Programming Guide, the App ID How-To as well as the Apple Push Notification topic in the Apple Developer Forums.

For more information on incorporating In App Purchases into your applications, view the Store Kit Programming Guide, the App ID How-Tos, and the iTunes Connect User Guide for setup instructions.

For more information on Keychain Access sharing, view the Keychain Services Reference.

For more information on developing external accessories to communicate with your iOS application, view MFi Program.

For more information on utilizing iCloud in your applications, view What's New in iOS 5.

FIGURE 25-2

Typically, you create this identifier by combining the reverse-domain name of your website and the name of your Xcode project. For example, the project created in this lesson is called `CloudTest` and the bundle identifier specified is `com.wileybook.CloudTest`.

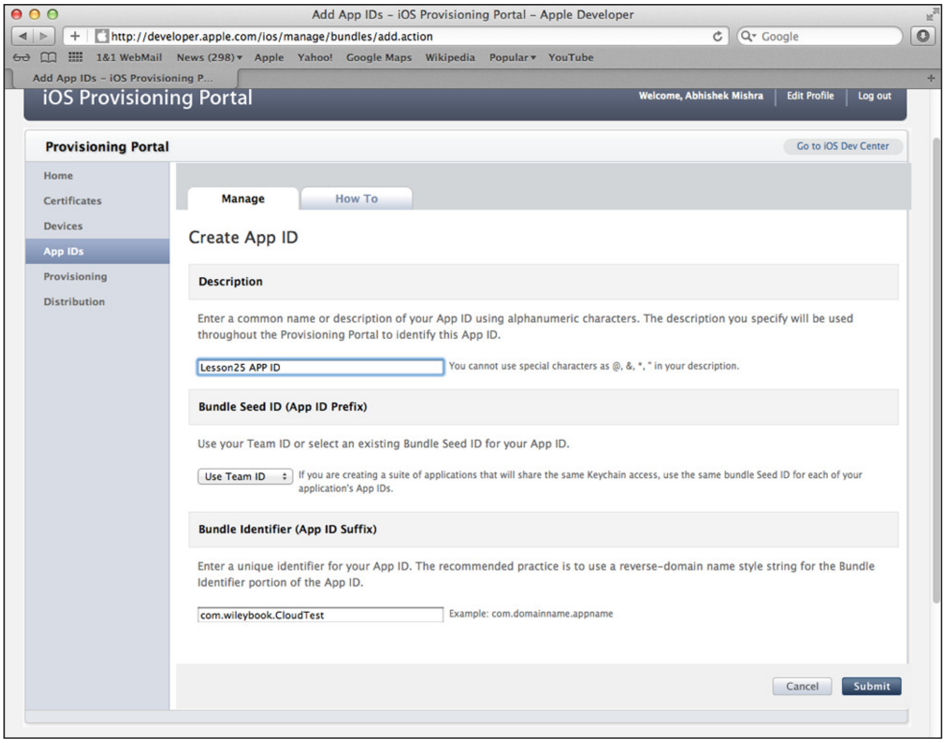
Your browser window should resemble Figure 25-3. Click the Submit button to finish creating the App ID.

Look for the new App ID in the list of App IDs and notice that, by default, it is not configured for iCloud (Figure 25-4). If your new App ID is not visible, you may need to refresh your browser window.

To configure the App ID for iCloud, click the Configure link, which takes you to the Configure App ID page. Select the Enable for iCloud checkbox (Figure 25-5).

This brings up a warning message stating that all provisioning profiles that you will create using this App ID will be iCloud enabled (Figure 25-6).

Click OK to dismiss the warning message and then click Done to finish configuring the App ID for iCloud.



Add App IDs – iOS Provisioning Portal – Apple Developer

http://developer.apple.com/ios/manage/bundles/add.action

Google
1&1 WebMail News (298) Apple Yahoo! Google Maps Wikipedia Popular YouTube
Add App IDs – iOS Provisioning P...
iOS Provisioning Portal
Welcome, Abhishek Mishra | Edit Profile | Log out
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Manage
How To

Create App ID

Description

Enter a common name or description of your App ID using alphanumeric characters. The description you specify will be used throughout the Provisioning Portal to identify this App ID.

Lesson25 APP ID — You cannot use special characters as @, &, *, " in your description.

Bundle Seed ID (App ID Prefix)

Use your Team ID or select an existing Bundle Seed ID for your App ID.

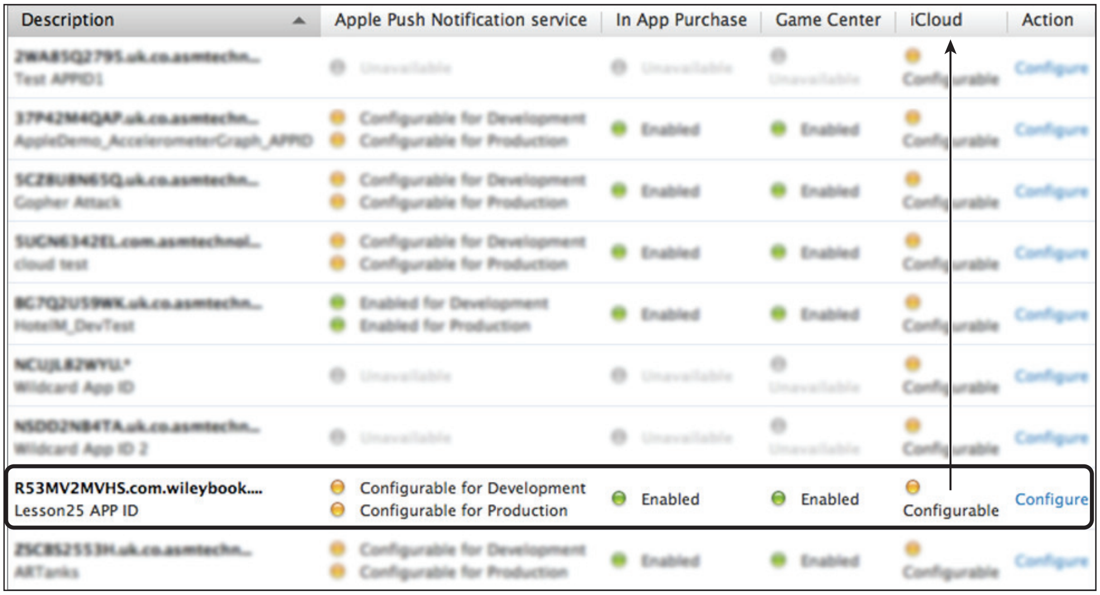
Use Team ID — If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.

Bundle Identifier (App ID Suffix)

Enter a unique identifier for your App ID. The recommended practice is to use a reverse-domain name style string for the Bundle Identifier portion of the App ID.

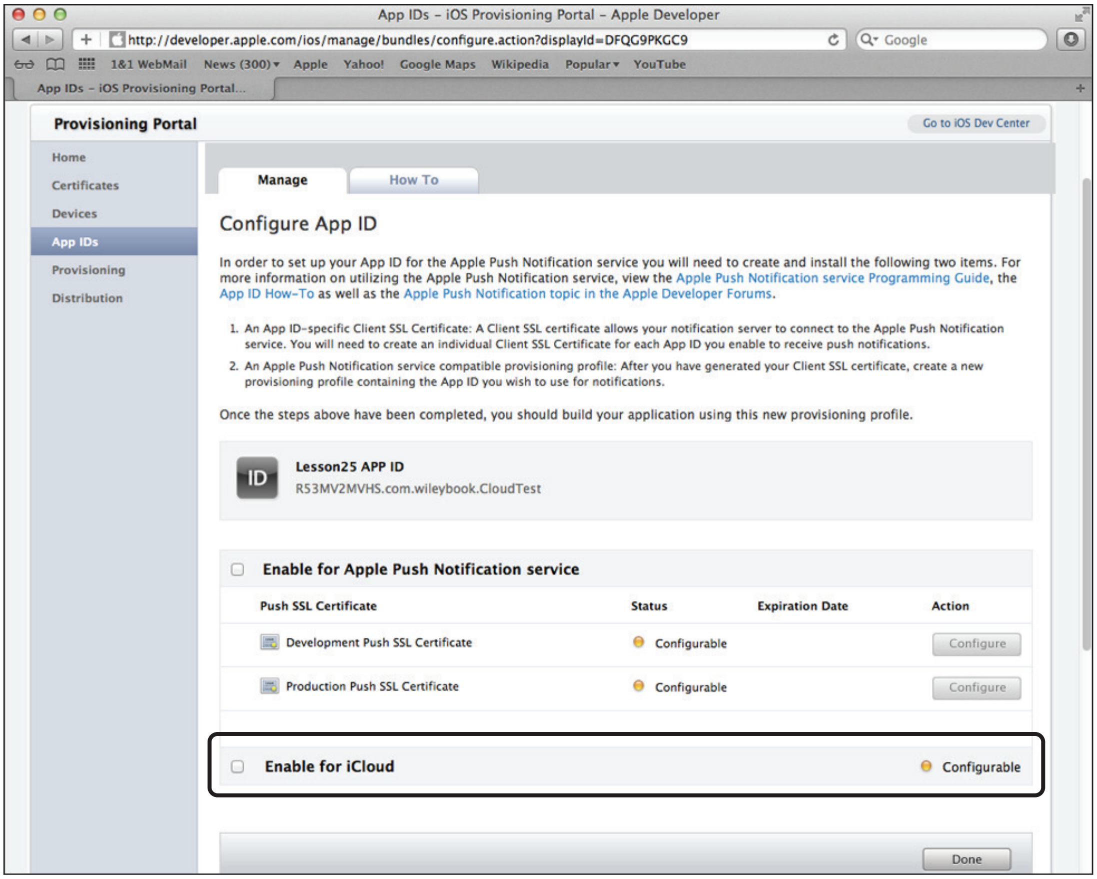
com.wileybook.CloudTest — Example: com.domainname.appname

FIGURE 25-3



Description	Apple Push Notification service	In App Purchase	Game Center	iCloud	Action
2WA85Q2795.uk.co.asmtechn... Test APPID1	Unavailable	Unavailable	Unavailable	Configurable	Configure
37P42M4QAP.uk.co.asmtechn... AppleDemo_AccelerometerGraph_APPID	Configurable for Development Configurable for Production	Enabled	Enabled	Configurable	Configure
5CZ8UBN65Q.uk.co.asmtechn... Gopher Attack	Configurable for Development Configurable for Production	Enabled	Enabled	Configurable	Configure
5UGN6342EL.com.asmtechnol... cloud test	Configurable for Development Configurable for Production	Enabled	Enabled	Configurable	Configure
8G7Q2U59WK.uk.co.asmtechn... HotelM_DevTest	Enabled for Development Enabled for Production	Enabled	Enabled	Configurable	Configure
NCUJL82WYU.* Wildcard App ID	Unavailable	Unavailable	Unavailable	Configurable	Configure
N5DD2NB4TA.uk.co.asmtechn... Wildcard App ID 2	Unavailable	Unavailable	Unavailable	Configurable	Configure
R53MV2MVHS.com.wileybook.... Lesson25 APP ID	Configurable for Development Configurable for Production	Enabled	Enabled	Configurable	Configure
Z5CB52553H.uk.co.asmtechn... ARTanks	Configurable for Development Configurable for Production	Enabled	Enabled	Configurable	Configure

FIGURE 25-4


App IDs – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/bundles/configure.action?displayId=DFQG9PKGC9
Google
1&1 WebMail News (300) Apple Yahoo! Google Maps Wikipedia Popular YouTube
App IDs – iOS Provisioning Portal...
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Manage
How To
Done

Configure App ID

In order to set up your App ID for the Apple Push Notification service you will need to create and install the following two items. For more information on utilizing the Apple Push Notification service, view the Apple Push Notification service Programming Guide, the App ID How-To as well as the Apple Push Notification topic in the Apple Developer Forums.

1. An App ID-specific Client SSL Certificate: A Client SSL certificate allows your notification server to connect to the Apple Push Notification service. You will need to create an individual Client SSL Certificate for each App ID you enable to receive push notifications.
2. An Apple Push Notification service compatible provisioning profile: After you have generated your Client SSL certificate, create a new provisioning profile containing the App ID you wish to use for notifications.

Once the steps above have been completed, you should build your application using this new provisioning profile.

Lesson25 APP ID
R53MV2MVHS.com.wileybook.CloudTest

Enable for Apple Push Notification service

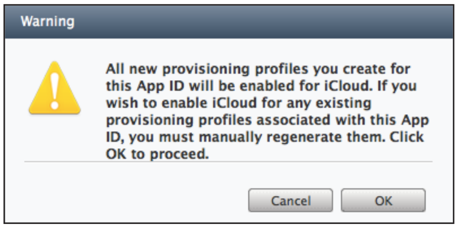
Push SSL Certificate	Status	Expiration Date	Action
Development Push SSL Certificate	● Configurable		Configure
Production Push SSL Certificate	● Configurable		Configure

Enable for iCloud ● Configurable

FIGURE 25-5

Create an Appropriate Provisioning Profile

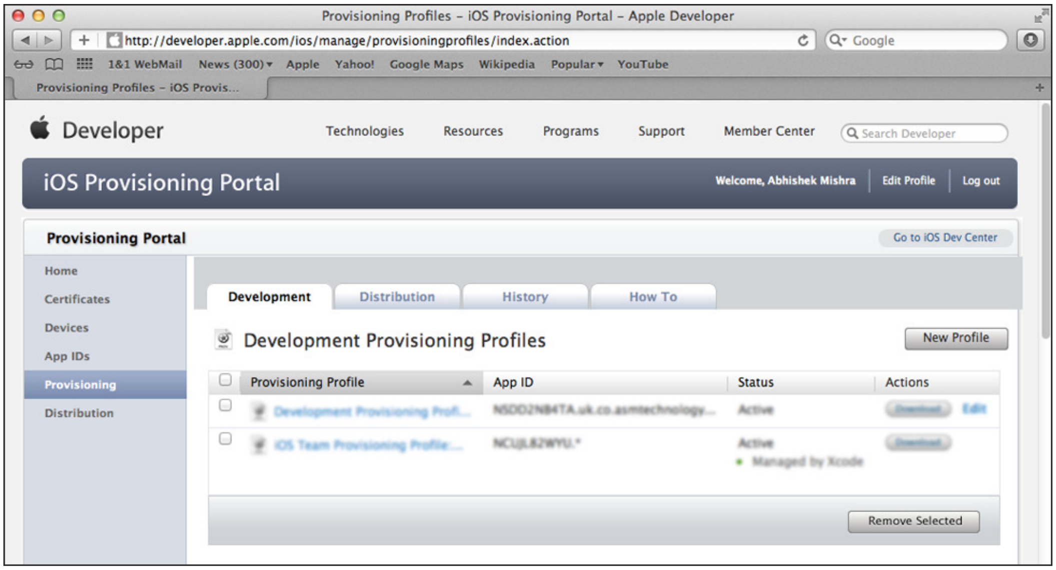
To create a provisioning profile for an iCloud-enabled App ID, click the Provisioning link in the menu on the left-hand side of the iOS Provisioning Portal window. You can create a development or distribution provisioning profile depending on whether you are testing your application on your own devices, or submitting to the App Store. This lesson focuses on developing an iCloud-enabled application, therefore, ensure the Development tab is selected and click the New Profile button (Figure 25-7).



All new provisioning profiles you create for this App ID will be enabled for iCloud. If you wish to enable iCloud for any existing provisioning profiles associated with this App ID, you must manually regenerate them. Click OK to proceed.

FIGURE 25-6

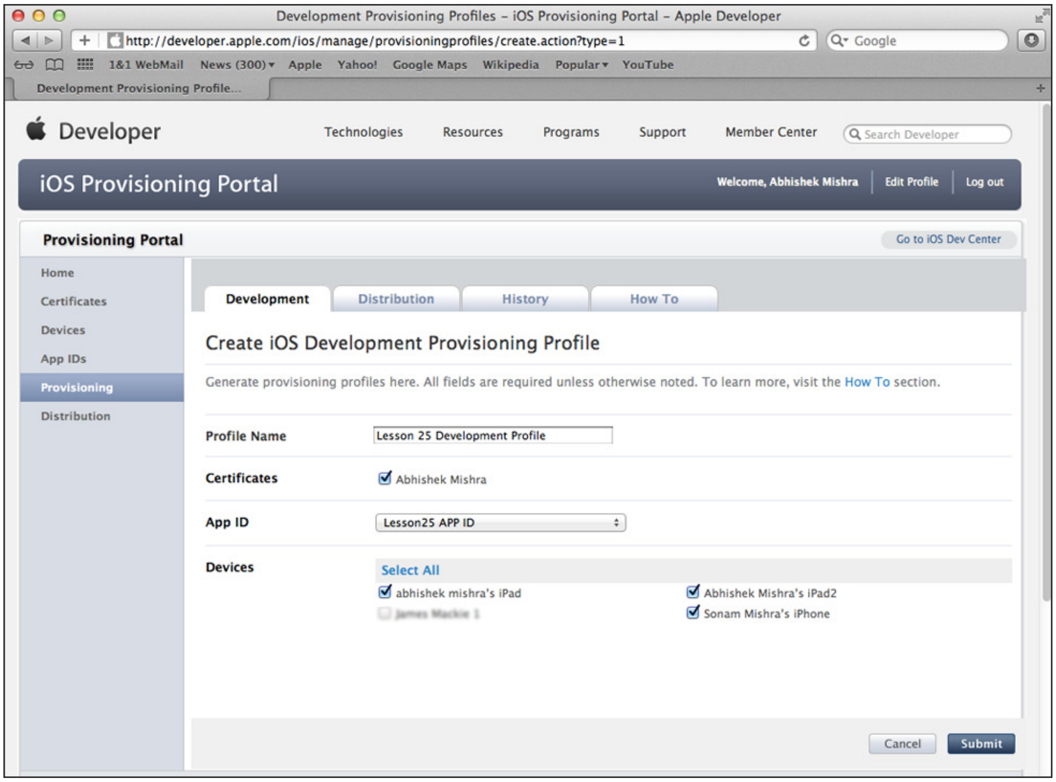
Provide a suitable name for the profile, and select your development certificate, the iCloud-enabled App ID you created in the previous step, and a list of test devices (Figure 25-8). Click the Submit button to create the provisioning profile.


Provisioning Profiles – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/provisioningprofiles/index.action
Google
1&1 WebMail News (300) Apple Yahoo! Google Maps Wikipedia Popular YouTube
Provisioning Profiles – iOS Provis...
Developer
Technologies Resources Programs Support Member Center
Search Developer
iOS Provisioning Portal
Welcome, Abhishek Mishra | Edit Profile | Log out
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Development Distribution History How To
New Profile
Remove Selected

Development Provisioning Profiles

Provisioning Profile	App ID	Status	Actions
Development Provisioning Profi...	NSDD2NB4TA.uk.co.asmtechnology...	Active	Download Edit
iOS Team Provisioning Profile:...	NCUL82WYU.*	Active ● Managed by Xcode	Download

FIGURE 25-7


Development Provisioning Profiles – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/provisioningprofiles/create.action?type=1
Google
1&1 WebMail News (300) Apple Yahoo! Google Maps Wikipedia Popular YouTube
Development Provisioning Profile...
Developer
Technologies Resources Programs Support Member Center
Search Developer
iOS Provisioning Portal
Welcome, Abhishek Mishra | Edit Profile | Log out
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Development Distribution History How To
Cancel Submit

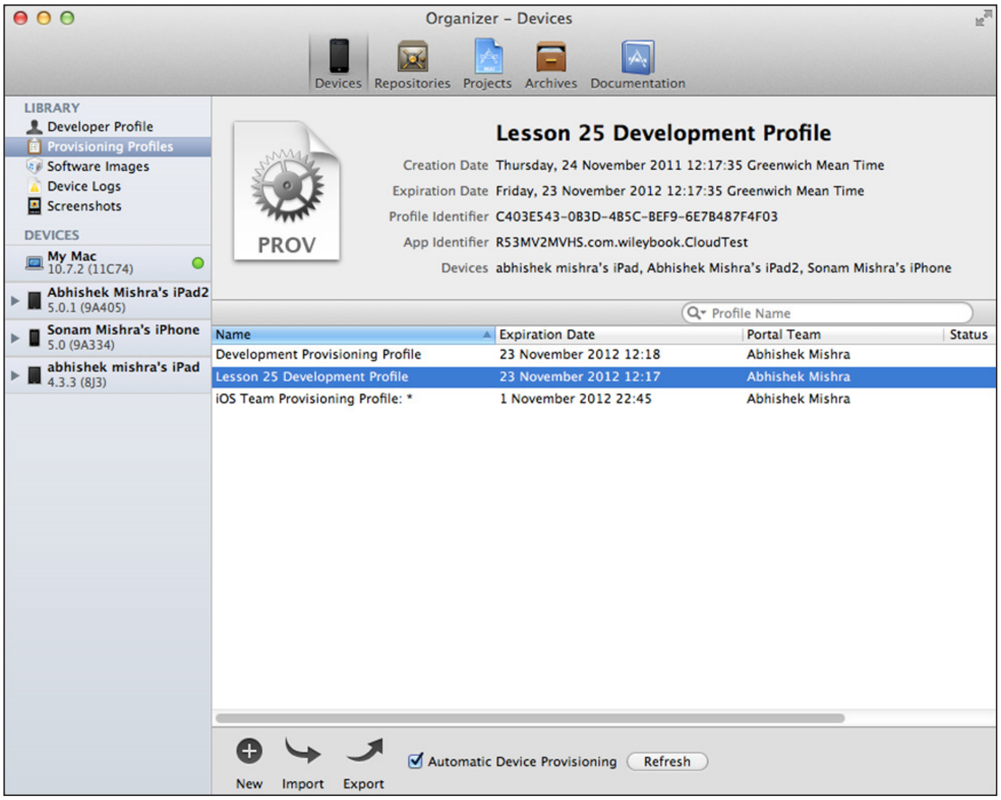
Create iOS Development Provisioning Profile

Generate provisioning profiles here. All fields are required unless otherwise noted. To learn more, visit the How To section.

Profile Name	Lesson 25 Development Profile
Certificates	✓ Abhishek Mishra
App ID	Lesson25 APP ID
Devices	Select All ✓ abhishek mishra's iPad ✓ Abhishek Mishra's iPad2 ☐ James Mackie 1 ✓ Sonam Mishra's iPhone

FIGURE 25-8

This takes you back to the previous screen and you should see an entry for the new profile in the list. Download the new provisioning profile and install it by dragging the profile from your Mac's `Downloads` folder onto the Xcode Organizer window (Figure 25-9). To show the Organizer window, launch Xcode and click the Organizer button in the toolbar.



Lesson 25 Development Profile

Creation Date	Thursday, 24 November 2011 12:17:35 Greenwich Mean Time
Expiration Date	Friday, 23 November 2012 12:17:35 Greenwich Mean Time
Profile Identifier	C403E543-0B3D-4B5C-BEF9-6E7B487F4F03
App Identifier	R53MV2MVHS.com.wileybook.CloudTest
Devices	abhishek mishra's iPad, Abhishek Mishra's iPad2, Sonam Mishra's iPhone

Name	Expiration Date	Portal Team	Status
Development Provisioning Profile	23 November 2012 12:18	Abhishek Mishra	
Lesson 25 Development Profile	23 November 2012 12:17	Abhishek Mishra	
iOS Team Provisioning Profile: *	1 November 2012 22:45	Abhishek Mishra	

FIGURE 25-9

Enable Appropriate Entitlements in Your Xcode Project

Create a new project in Xcode using one of the standard iOS application templates. In the Project Options dialog box, make sure you provide the correct value for the Product Name and Company Identifier fields so as to create the same App ID that was registered on the iOS Provisioning Portal. If, for instance, the App ID you registered was `com.wileybook.CloudTest`, use `CloudTest` for the Product Name field and `com.wileybook` for the Company Identifier field.

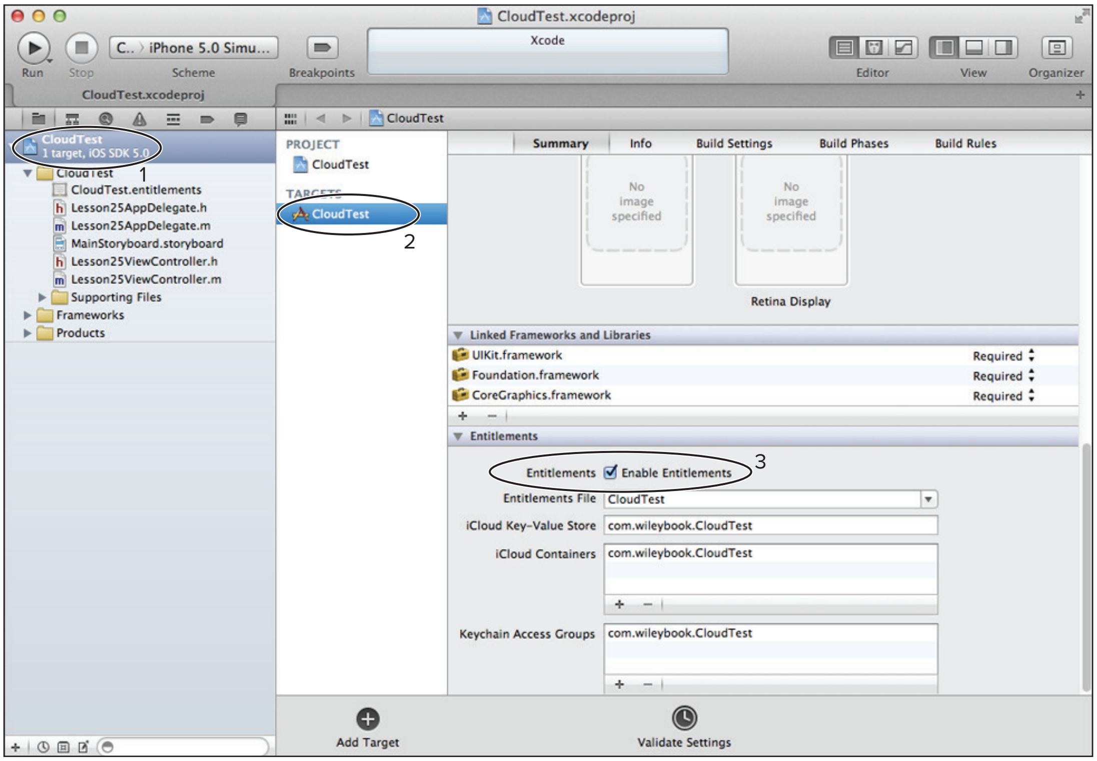
Applications that use iCloud must be signed with iCloud-specific entitlements. These entitlements ensure that only your applications can access the documents that they create. When you enable entitlements for your app target, Xcode automatically configures entitlements for both document and key-value storage.

Each entitlement is a key-value pair. The only entitlement keys allowed as of now are:

- `com.apple.developer.ubiquity-container-identifiers`
- `com.apple.developer.ubiquity-kvstore-identifier`

The value assigned to these keys consists of one or more container identifier strings. A container identifier string identifies a directory (also known as a container) on the iCloud server that your app can use to store its data. Typically, each iCloud-enabled app you create uses its own container on the server, and this container is identified uniquely by the application bundle identifier. However, applications that store documents could access multiple containers on the iCloud server.

To enable entitlements, select the project's root node in the project navigator and the appropriate build target. Ensure the Summary tab is selected. Scroll down to the Entitlements section and select the Enable Entitlements checkbox (Figure 25-10).



Entitlements ☑ Enable Entitlements
Entitlements File: CloudTest
iCloud Key-Value Store: com.wileybook.CloudTest
iCloud Containers: com.wileybook.CloudTest
Keychain Access Groups: com.wileybook.CloudTest

FIGURE 25-10

Xcode automatically fills in four fields with default values:

- **Entitlements File:** The name of a file with an `.entitlements` extension that has been added to your project. This is a standard property list file that contains all the entitlements data.

- **iCloud Key-Value Store:** The container identifier for key-value type data. There can be only one container for key-value data per application, and hence this field accepts only a single value. By default, it is the container identified by your application bundle identifier.
- **iCloud Containers:** Contains a list of container identifiers for document data. An application that uses iCloud document storage can potentially read/write to multiple containers by specifying multiple container identifier strings. By default, Xcode adds the container identifier for the directory identified by your application bundle identifier. If multiple identifiers are specified, the first string must always be the main container identifier for your application.
- **Keychain Access Groups:** Contains keys needed by applications that share keychain data. For the scope of this lesson, you should accept the default value provided by Xcode.

CHECKING FOR SERVICE AVAILABILITY

If your application intends to make use of the iCloud Storage APIs, you must ensure that the service is available to the application. This may not necessarily be the case if, for example, the user has not set up iCloud on the device.

To check for service availability, use the `URLForUbiquityContainerIdentifier:` method of the `NSFileManager` shared instance. This method requires one `NSString` parameter that specifies a container identifier that your application uses.

If this method succeeds, the return value is an `NSURL` instance that identifies the container directory. If the method fails, the return value is `nil`.

If your application uses only one container identifier, or you want to use the main container identifier for the application, pass `nil` for the parameter. If your application accesses multiple containers, you must call this method for each container identifier to ensure you have access to each container. The following code snippet shows how to use this method for the main container identifier:

```
NSURL *folderURL = [[NSFileManager defaultManager]
                   URLForUbiquityContainerIdentifier:nil];
if (folderURL != nil)
{
    // cloud access is available
}
else
{
    // cloud access is not available.
}
```

USING ICLOUD DOCUMENT STORAGE

Any file stored by your application on iCloud must be managed by a file presenter object. A file presenter is an object that implements the `NSFilePresenter` protocol. Essentially, a file presenter acts as an agent for a file. Before an external source can change the file, the file presenter for the file is notified. When your app wants to change the file, it must lock the file by making its changes through a file coordinator object. A file coordinator object is an instance of the `NSFileCoordinator` class.

The simplest way to incorporate file presenters and coordinators in your application is to have your data classes (also known as model classes) subclass `UIDocument`. The `UIDocument` class implements the methods of the `NSFilePresenter` protocol and handles all of the file-related management. At the most basic level, you will need to override two `UIDocument` methods:

```
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
                                      error:(NSError **)outError;

- (id)contentsForType:(NSString *)typeName error:(NSError **)outError;
```

The `loadFromContents:ofType:error:` method is overridden by your `UIDocument` subclass, and is called when the application needs to read data into its data model.

The first parameter of this method, `contents`, encapsulates the document data to be read. In the case of flat files, `contents` is an instance of an `NSData` object. It can also be an `NSFileWrapper` instance if the data being read corresponds to a file package. The `typeName` parameter indicates the file type of the document.

If you cannot load the document for some reason, you should create an `NSError` object encapsulating the reason for failure and return its address in the `outError` parameter. If you did not encounter problems loaded document data, ignore this parameter.

The `contentsForType:error:` method is also overridden by your `UIDocument` subclass and is called when the application saves data to a file. This method must return an `NSData` instance that will be written to the file. If you cannot return an `NSData` instance for some reason, you must return a pointer to an `NSError` object. The `NSError` object must encapsulate the reason for failure.

Listings 25-1 and 25-2 present the interface and implementation of a simple `UIDocument` subclass called `CloudTestDocument`. The example assumes that the application where this class is used has a rather simple data model consisting of a single `NSString` instance.

LISTING 25-1: CloudTestDocument.h

```
@interface CloudTestDocument : UIDocument
@property (nonatomic, strong) NSString* documentContent;
@end
```

LISTING 25-2: CloudTestDocument.m

```
@synthesize documentContent;

// Called whenever the application reads data from the file system
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
                                      error:(NSError **)outError
{
    if ([contents length] > 0)
    {
        self.documentContent = [[NSString alloc] initWithBytes:[contents bytes]
                                length:[contents length]
                                encoding:NSUTF8StringEncoding];

        [[NSNotificationCenter defaultCenter]
         postNotificationName:@"refreshDocumentPreview"
         object:self];
    }
    else
    {
        self.documentContent = @"";
    }

    return YES;
}

// Called whenever the application (auto)saves the content of a note
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError
{
    return [self.documentContent dataUsingEncoding:NSUTF8StringEncoding];
}
```

Creating a New iCloud Document

To create a new document, allocate and initialize an instance of your `UIDocument` subclass by using the `initWithFileURL:` method and then call `saveToURL:forSaveOperation:completionHandler:` on the instance.

The `initWithFIleURL:` method requires a single `NSURL` parameter that identifies the location where document data is to be written. This URL is usually composed by appending a filename in the `Documents` subdirectory to the path to an iCloud container. For instance, to create a new document on iCloud called `phoneNumber.txt`, you could use the following snippet:

```
NSURL *containerURL = [[NSFileManager defaultManager]
                       URLForUbiquityContainerIdentifier:nil];

NSURL *documentURL = [[containerURL URLByAppendingPathComponent:@"Documents"]
                       URLByAppendingPathComponent:@"phoneNumber.txt"];

CloudTestDocument * cloudDocument = [[CloudTestDocument alloc]
                                     initWithFileURL:documentURL];

[cloudDocument saveToURL:[self.cloudDocument fileURL]
                forSaveOperation:UIDocumentSaveForCreating
                completionHandler:^(BOOL success)
         {
             if (success)
             {
                 // document was create successfully.
             }
         }];
```

The `saveToURL:forSaveOperation:completionHandler:` method is described later in this lesson.

Opening an Existing Document

To open an existing document, allocate and initialize an instance of your `UIDocument` subclass and call `openWithCompletionHandler:` on the instance. For example, you could open a file called `phoneNumbers.txt` from iCloud using the following snippet:

```
NSURL *containerURL = [[NSFileManager defaultManager]
                       URLForUbiquityContainerIdentifier:nil];
NSURL *documentURL = [[containerURL URLByAppendingPathComponent:@"Documents"]
                      URLByAppendingPathComponent:@"phoneNumber.txt"];
CloudTestDocument* cloudDocument = [[CloudTestDocument alloc]
                                    initWithFileURL:documentURL];

[self.cloudDocument openWithCompletionHandler:^(BOOL success)
{
         if (success)
         {
            // cloud document opened successfully!
         }
}];
```

Saving a Document

Once you have an instance of a `UIDocument` subclass, saving it to iCloud is simply a matter of calling the `saveToURL:forSaveOperation:completionHandler:` method on it. The first parameter to this method is an `NSURL` instance that contains the target URL. You can compose this URL in the same manner as when you instantiated your `UIDocument` subclass. If, however, you want to retrieve the URL corresponding to an existing `UIDocument` subclass, simply send the `fileURL` message to your subclass. Thus, if `cloudDocument` is an instance of a `UIDocument` subclass, you can retrieve the URL used when it was instantiated using the following code:

```
NSURL *documentURL = [cloudDocument fileURL]
```

The second parameter is a constant that is used to indicate whether the document contents are being saved for the first time, or overwritten. It can be either of:

- `UIDocumentSaveForCreating`
- `UIDocumentSaveForOverwriting`

The third parameter is a block completion handler.

For more information on the `UIDocument` *class, refer to the UIDocument Class reference, available at* `http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIDocument_Class/UIDocument/UIDocument.html`.

Searching for Documents on iCloud

Often you will need to search iCloud container directories for documents. To do this, you need to create a search query using an NSMetadataQuery instance, set up an appropriate search filter, and execute the query. Queries have two phases: an initial search phase and a second live-update phase. During the live-update phase, updated results are typically available once every second. The following code snippet builds a search query:

```
NSMetadataQuery* searchQuery = [[NSMetadataQuery alloc] init];
[searchQuery setSearchScopes:[NSArray

                    arrayWithObject:NSMetadataQueryUbiquitousDocumentsScope]];
```

The setSearchScopes: method enables you to specify an array of directories over which the search should execute. To specify the iCloud container folder as the search target, you provide an NSArray instance with a single object:

```
NSMetadataQueryUbiquitousDocumentsScope
```

Before you can execute the query, you need to specify a search filter. Search filters are also known as predicates and are instances of the NSPredicate class. The following code snippet creates an NSPredicate instance that filters out a file with a specific name:

```
NSString* documentFileName = @"cloudDocument.txt";
NSPredicate *pred = [NSPredicate predicateWithFormat:@"%K == %@",
                    NSMetadataItemFSNameKey, documentFileName];
```

To apply the predicate to the search query, use the setPredicate: method on the NSMetadataQuery instance:

```
[searchQuery setPredicate:pred];
```

Search queries execute asynchronously. When the query has finished gathering results, your application will receive the NSMetadataQueryDidFinishGatheringNotification notification message. Use the following code snippet to set up a method in your code called queryDidFinish: to be called when this notification is received:

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                      selector:@selector(queryDidFinish:)
                               name:NSMetadataQueryDidFinishGatheringNotification
                                      object:searchQuery];
```

Finally, to start the query, send the startQuery message to the NSMetadataQuery instance:

```
[searchQuery startQuery];
```

When you receive the notification message, you can find out the number of results returned by the search by sending the resultCount message to the NSMetadataQuery instance:

```
int numResults = [searchQuery resultCount];
```

To retrieve an NSURL instance for each result returned by the search query, you can use a simple for loop:

```
for (int resultIndex = 0; resultIndex < numResults; resultIndex++)
{
    NSMetadataItem *item = [self.searchQuery resultAtIndex:resultIndex];
    NSURL *url = [item valueForAttribute:NSMetadataItemURLKey];
}
```

If you do not want the search query to continue returning results, use the following code snippet to stop it:

```
    [self.searchQuery disableUpdates];
    [self.searchQuery stopQuery];
```

The Try It section for this lesson contains a simple project that uses an `NSMetadataQuery` instance to find a document on iCloud and then proceeds to open it.

> *For more information on the* `NSMetadataQuery` *class, refer to the NSMetadataQuery Class Reference, available at* `http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSMetadataQuery_Class/Reference/Reference.html#//apple_ref/occ/cl/NSMetadataQuery`. *For more information on the* `NSPredicate` *class, refer to the NSPredicate Class Reference available at* `http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSPredicate_Class/Reference/NSPredicate.html#//apple_ref/doc/c_ref/NSPredicate`.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `CloudTest`. In this application, you to create a simple text document called `cloudDocument.txt` and store it on iCloud. This document can then be edited across multiple copies of the application running on different iOS devices.

Lesson Requirements

- Create a new Universal application project based on the Single View Application template.
- Register the App ID with the iOS Provisioning Portal.
- Create a development provisioning profile.
- Download and install the development provisioning profile.
- Create a simple user interface that consists of a `UIButton` instance, a `UILabel` instance, and a `UITextView` instance.
- Create a data class that subclasses `UIDocument`.
- Check iCloud service availability in the `viewDidLoad` method of the view controller class.
- Load an existing document stored on iCloud. If the document does not exist, create a new one.
- Implement code to save the document on iCloud when a button is tapped.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 25 folder in the download.*

Hints

- To make best use of this application, you will need at least two iOS devices set up to use the same iCloud account.
- You must ensure iCloud has been set up on each test device.
- This Try It requires you to create a Universal application. You should be alright following the steps listed here, but if you want more information on Universal applications, read Lesson 38.
- Testing your apps on iOS devices is covered in Appendix D.

Step-by-Step

1. Create a Single View Application in Xcode called `CloudTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the Project Options dialog box and click Next:
        - **Product Name:** CloudTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson25
        - **Define Family:** Universal
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.

2. Register an App ID with the iOS Provisioning Portal.
   1. Log in to the iOS Provisioning Portal, and register a new App ID with the following details:
      - **Description:** Lesson25 App ID
      - **Bundle Seed ID:** Use Team ID
      - **Bundle Identifier:** com.wileybook.CloudTest
   2. Enable the App ID to use with iCloud.

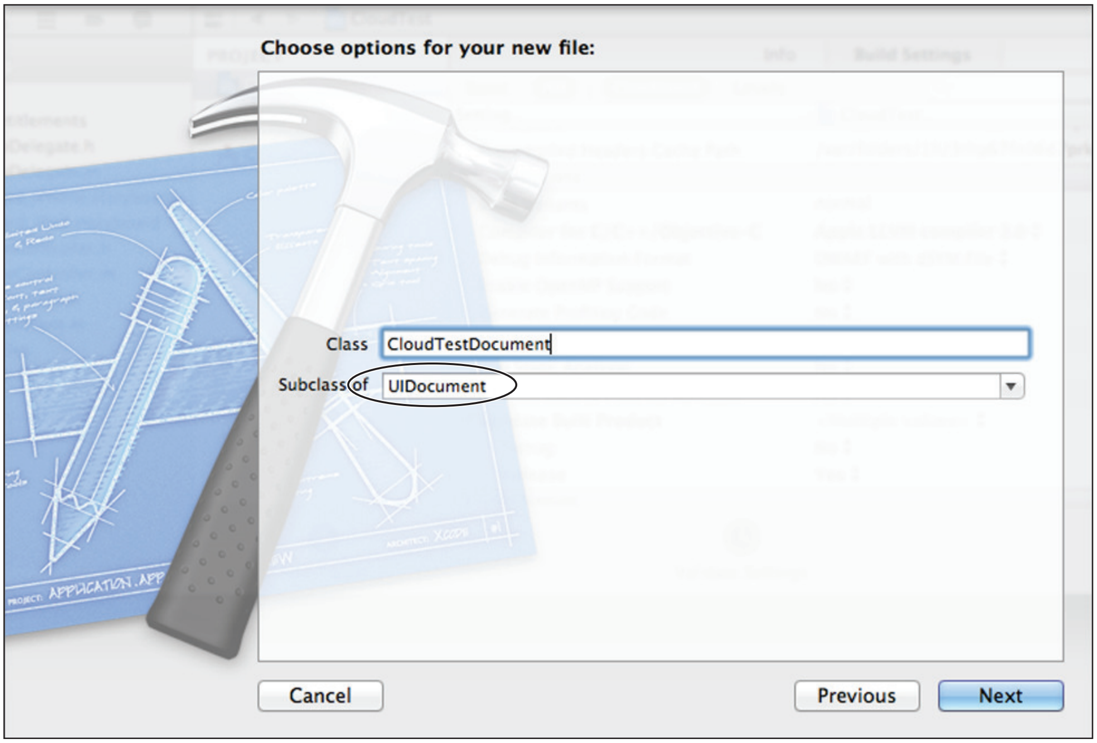
      This process is covered in the section titled "Create an iCloud-enabled App ID" earlier in this lesson.
3. Create a development provisioning profile using the App ID created in the previous step.
   1. The process of creating the provisioning profile is covered in the section titled "Creating an Appropriate Provisioning Profile" earlier in this lesson. Follow those instructions to create a development provisioning profile called Lesson 25 Development Profile.
   2. Download and install the provisioning profile in the Xcode Organizer.
4. Select the project's root node in the project navigator and select the appropriate build target. Ensure the Summary tab is selected. Scroll down to the Entitlements section and select the Enable Entitlements checkbox.
5. Create a `UIDocument` subclass.
   1. Right-click your project's root node in the project navigator and select New File from the context menu.
   2. Select the Objective-C class template and click Next.
   3. Name the class `CloudTestDocument` and specify `UIDocument` as the parent class (Figure 25-11). You will need to type `UIDocument` manually in the Subclass Of field because it is not present in the default list.
   4. Add the following property declaration to the `CloudTestDocument.h` file:

      ```
      @property (nonatomic, strong) NSString* documentContent;
      ```
   5. Add the following `@synthesize` statement to the `CloudTestDocument.m` file:

      ```
      @synthesize documentContent;
      ```
   6. Override the `loadFromContents:ofType:error:` method in `CloudTestDocument.m` by adding the following implementation:

      ```
      - (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
                                          error:(NSError **)outError
      {
          if ([contents length] > 0)
          {
              self.documentContent = [[NSString alloc]
                                      initWithBytes:[contents bytes]
                                      length:[contents length]
      ```

```
                                        encoding:NSUTF8StringEncoding];

            [[NSNotificationCenter defaultCenter]
             postNotificationName:@"refreshDocumentPreview"
             object:self];
        }
        else
        {
            self.documentContent = @"";
        }

        return YES;
    }
```



Choose options for your new file:

Class: CloudTestDocument

Subclass of: UIDocument

FIGURE 25-11

7. Recall that this method is called when a document must be loaded from a file. In the case of iCloud documents this method is also called automatically when the contents of the file have changed. This will typically happen when the file was edited by another copy of the application.

8. In the preceding implementation, in addition to loading the contents of the file into member variables of the `CloudTestDocument` class, you also send out an application-wide notification called `refreshDocumentPreview`.

9. The view controller class listens for these notifications, and treats the arrival of one as a cue to update the user interface.
10. Override the `contentsForType:error:` method in `CloudTestDocument.m` by adding the following implementation:

```
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError
{
    return [self.documentContent dataUsingEncoding:NSUTF8StringEncoding];
}
```

6. Edit the `MainStoryboard_iPhone.storyboard` file with Interface Builder.
    1. Use the Object library to add a `UILabel` instance, a `UIButton` instance, and a `UITextView` instance to the default scene.
    2. Resize/position the `UILabel` instance at X=10, Y=20, W=300, H=21.
    3. Use the Attributes inspector to set the text property of the label to **iCloud Service Status:**.
    4. Use the Attributes inspector to set the Alignment property of the label to center.
    5. Resize/position the `UIButton` instance to X=10, Y=56, W=300, H=37.
    6. Double-click the button in the scene and change its title to Save Document.
    7. Resize/position the `UITextView` instance to X=10, Y=108, W=300, H=332.
    8. Use the assistant editor to create an outlet called `serviceStatus` in the `Lesson25ViewController` class and connect it to the `UILabel` instance in the default scene.
    9. Use the assistant editor to create an outlet called `documentContentView` in the `Lesson25ViewController` class and connect it to the `UITextView` instance in the default scene.
    10. Use the assistant editor to create an action method called `onSaveDocument` in the `Lesson25ViewController` class and connect it to the `Touch Up Inside` event of the `UIButton` instance in the default scene.

    Your storyboard should resemble Figure 25-12.
7. Edit the `MainStoryboard_iPad.storyboard` file with Interface Builder.
    1. Use the Object library to add a `UILabel` instance, a `UIButton` instance, and a `UITextView` instance to the default scene.
    2. Resize/position the `UILabel` instance at X=20, Y=20, W=734, H=21.
    3. Use the Attributes inspector to set the text property of the label to **iCloud Service Status:**.
    4. Use the Attributes inspector to set the Alignment property of the label to center.
    5. Resize/position the `UIButton` instance to X=15, Y=58, W=738, H=37.

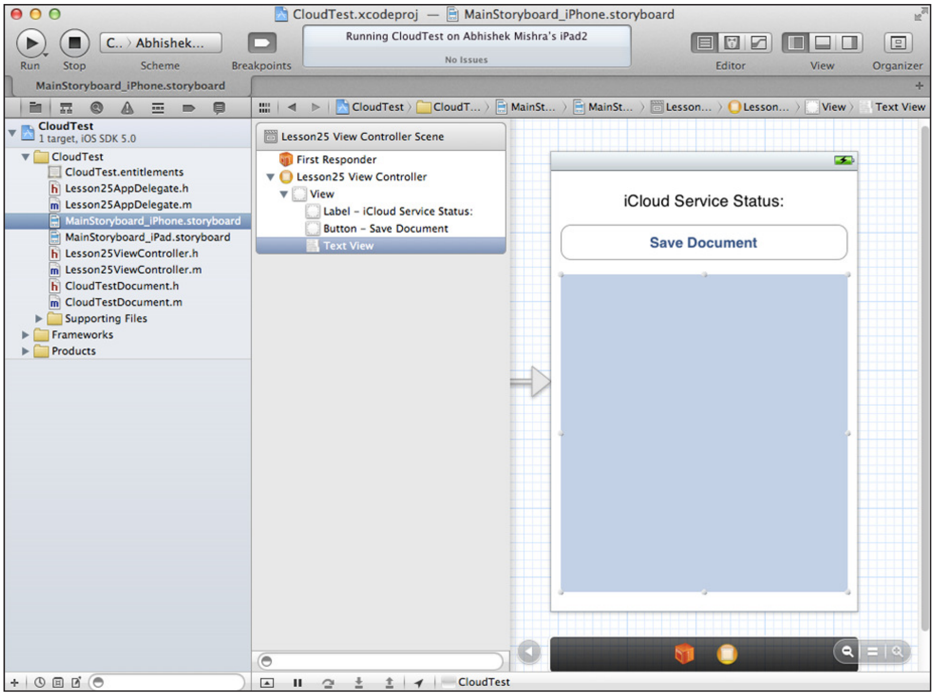
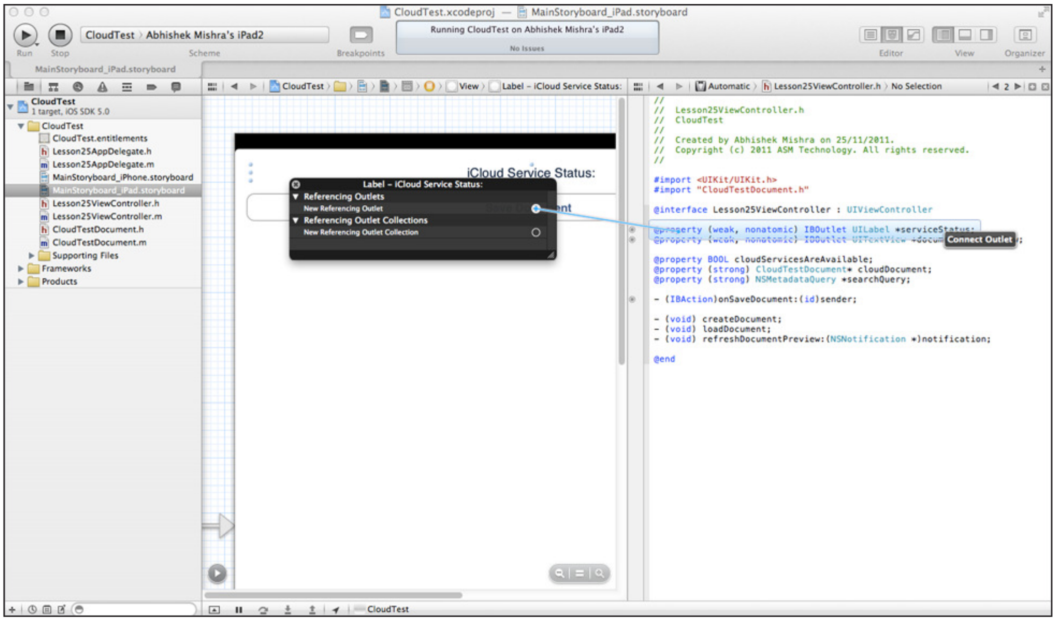


FIGURE 25-12

6. Double-click the button in the scene and change its title to Save Document.
7. Resize/position the `UITextView` instance to X=26, Y=103, W=717, H=881.
8. Use the assistant editor to connect the `UILabel` instance to the outlet called `serviceStatus` in the `Lesson25ViewController.h` file.
   1. Ensure the `Lesson25ViewController.h` file is visible in the assistant editor.
   2. Right-click the `UILabel` instance in the default scene to bring up a context menu.
   3. Drag from the circle beside the New Referencing Outlet entry of the context menu to the existing outlet called `serviceStatus` in the `Lesson25ViewController.h` file (Figure 25-13).
9. Use the assistant editor to connect the `UITextView` instance to the outlet called `documentContentView` in the `Lesson25ViewController.h` file.
10. Use the assistant editor to connect the `Touch Up Inside` event of the `UIButton` instance to the action called `onSaveDocument` in the `Lesson25ViewController.m` file.

Your storyboard should resemble Figure 25-14.


CloudTest.xcodeproj — MainStoryboard_iPad.storyboard
Running CloudTest on Abhishek Mishra's iPad2
No Issues
Label – iCloud Service Status:
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection
iCloud Service Status:
Connect Outlet
Automatic › Lesson25ViewController.h › No Selection

```
//
//  Lesson25ViewController.h
//  CloudTest
//
//  Created by Abhishek Mishra on 25/11/2011.
//  Copyright (c) 2011 ASM Technology. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "CloudTestDocument.h"

@interface Lesson25ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *serviceStatus;
@property (weak, nonatomic) IBOutlet UITextView *docum

@property BOOL cloudServicesAreAvailable;
@property (strong) CloudTestDocument* cloudDocument;
@property (strong) NSMetadataQuery *searchQuery;

- (IBAction)onSaveDocument:(id)sender;

- (void) createDocument;
- (void) loadDocument;
- (void) refreshDocumentPreview:(NSNotification *)notification;

@end
```

FIGURE 25-13

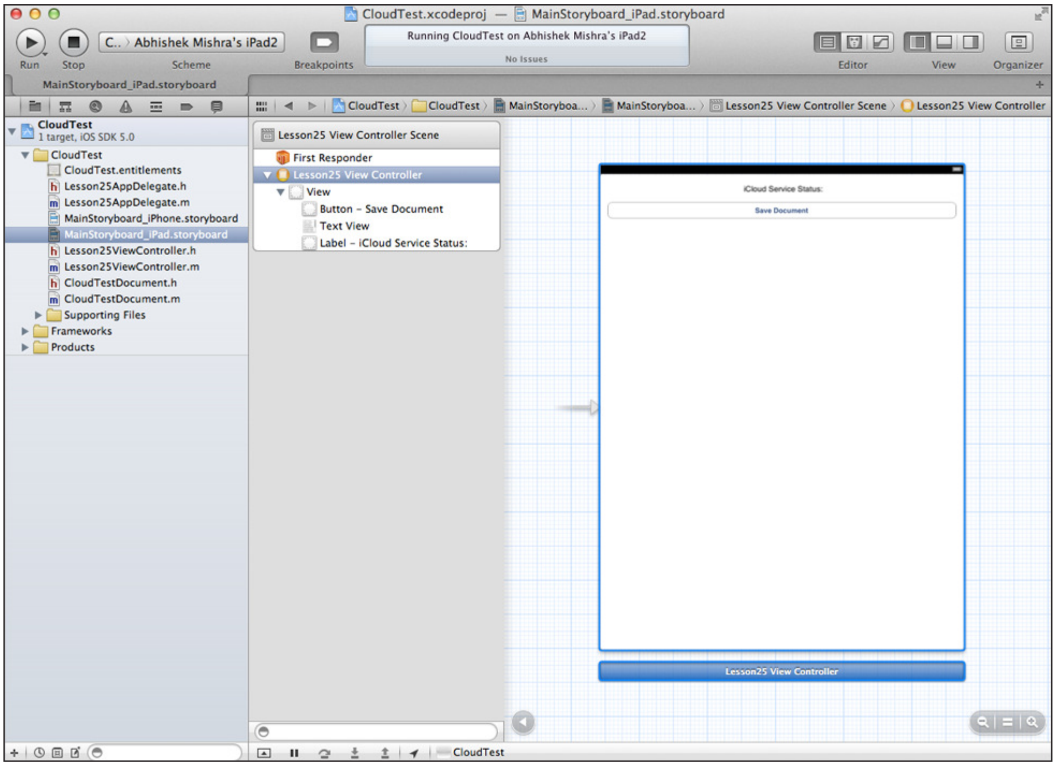

CloudTest.xcodeproj — MainStoryboard_iPad.storyboard
Running CloudTest on Abhishek Mishra's iPad2
No Issues
Lesson25 View Controller Scene
First Responder
Lesson25 View Controller
View
Button – Save Document
Text View
Label – iCloud Service Status:
iCloud Service Status:
Save Document
Lesson25 View Controller

FIGURE 25-14

8. Edit the `Lesson25ViewController.h` file.

    1. Add the following `#import` statement to the top of the file:

    ```
    #import "CloudTestDocument.h"
    ```

    2. Add the following property declarations:

    ```
    @property BOOL cloudServicesAreAvailable;
    @property (strong) CloudTestDocument* cloudDocument;
    @property (strong) NSMetadataQuery *searchQuery;
    ```

    3. Add the following method declarations:

    ```
    - (void) createDocument;
    - (void) loadDocument;
    - (void) queryDidFinish:(NSNotification *)notification;
    - (void) refreshDocumentPreview:(NSNotification *)notification;
    ```

9. Edit the `Lesson25ViewController.m` file.

    1. Add the following `@synthesize` statements:

    ```
    @synthesize cloudServicesAreAvailable;
    @synthesize cloudDocument;
    @synthesize searchQuery;
    ```

    2. Update the implementation of the `viewDidLoad` method to resemble the following:

    ```
    - (void)viewDidLoad
    {
        [super viewDidLoad];

        documentContentView.text = @"";

        // register this class as an observer for the 'refreshDocumentPreview'
        // notification, this notification is sent by the document class when
        // the contents of the document have ben updated.
        [[NSNotificationCenter defaultCenter] addObserver:self
                                   selector:@selector(refreshDocumentPreview:)
                                   name:@"refreshDocumentPreview"
                                   object:nil];

        // check if cloud services are available.
        NSURL* containerURL = [[NSFileManager defaultManager]
                              URLForUbiquityContainerIdentifier:nil];
        if (containerURL != nil)
        {
            // cloud access is available
            self.cloudServicesAreAvailable = YES;
            serviceStatus.text = @"Cloud Service Status: Available";

            // load existing document, or create a new document
            [self loadDocument];
        }
        else
        {
            // cloud access is not avaialable.
    ```

```
        self.cloudServicesAreAvailable = NO;
        serviceStatus.text = @"Cloud Service Status: Not Available";

        UIAlertView* cloudError = [[UIAlertView alloc]
                 initWithTitle:@""
                 message:@"iCloud has not been setup on this device!"
                 delegate:nil
                 cancelButtonTitle:@"Ok"
                 otherButtonTitles:nil];
        [cloudError show];
    }
}
```

In this method you check if the iCloud service is available, and if it is, then proceed to load a specific document from iCloud.

3. Add the following statements to the implementation of the `viewDidUnload` method after the existing contents of the method:

```
if (self.cloudDocument != nil)
    [self.cloudDocument closeWithCompletionHandler:nil];
self.cloudDocument = nil;

self.searchQuery = nil;

[[NSNotificationCenter defaultCenter] removeObserver:self
                                     name:@"refreshDocumentPreview"
                                     object:nil];
```

4. Implement the `loadDocument` method as follows:

```
- (void)loadDocument
{
    // search for iCloud document
    self.searchQuery = [[NSMetadataQuery alloc] init];
    [self.searchQuery setSearchScopes:[NSArray
         arrayWithObject:NSMetadataQueryUbiquitousDocumentsScope]];

    NSString* documentFileName = @"cloudDocument.txt";
    NSPredicate *pred = [NSPredicate predicateWithFormat:@"%K == %@",
                       NSMetadataItemFSNameKey, documentFileName];
    [self.searchQuery setPredicate:pred];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                selector:@selector(queryDidFinish:)
                  name:NSMetadataQueryDidFinishGatheringNotification
                                        object:nil];

    [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;

    [self.searchQuery startQuery];
}
```

These statements instantiate an `NSMetadataQuery` object to search the Documents directory in the application's iCloud container for a file called `cloudDocument.txt`. When the query is complete, the `queryDidFinish:` method of the view controller class will be called.

5. Implement the `queryDidFinish:` method as follows:

```
- (void)queryDidFinish:(NSNotification *)notification
{
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;

    // stop the query to prevent it from running constantly
    [self.searchQuery disableUpdates];
    [self.searchQuery stopQuery];

    [[NSNotificationCenter defaultCenter] removeObserver:self
                   name:NSMetadataQueryDidFinishGatheringNotification
                                          object:nil];

    // this application expects this query to return a single
    // result. If no documents were found, then inform the user.
    if ([self.searchQuery resultCount] == 0)
    {
        UIAlertView* cloudMessage = [[UIAlertView alloc]
                                 initWithTitle:@""
        message:@"Unable to find iCloud document, creating new document!"
                                 delegate:nil
                                 cancelButtonTitle:@"Ok"
                                 otherButtonTitles:nil];
        [cloudMessage show];

        self.searchQuery = nil;
        [self createDocument];
        return;
    }

    // process the result of the search
    if (self.cloudDocument == nil)
    {
        NSMetadataItem *item = [self.searchQuery resultAtIndex:0];
        NSURL *url = [item valueForAttribute:NSMetadataItemURLKey];
        self.cloudDocument = [[CloudTestDocument alloc]
initWithFileURL:url];
    }

    [self.cloudDocument openWithCompletionHandler:^(BOOL success)
     {
         if (success)
         {
             UIAlertView* cloudMessage = [[UIAlertView alloc]
                                         initWithTitle:@""
                                 message:@"iCloud document loaded!"
                                          delegate:nil
                                          cancelButtonTitle:@"Ok"
                                          otherButtonTitles:nil];
             [cloudMessage show];
         }
         else
         {
             UIAlertView* cloudMessage = [[UIAlertView alloc]
```

```
                                     initWithTitle:@""
                        message:@"Could not load iCloud document!"
                                     delegate:nil
                                     cancelButtonTitle:@"Ok"
                                     otherButtonTitles:nil];

                [cloudMessage show];
            }
        }
        ];

}
```

The preceding implementation first stops the query from running constantly. If the query did not return any results, it calls the `createDocument` method of the view controller class to create a new document on iCloud; otherwise, it loads the existing document from iCloud.

6. Implement the `onSaveDocument:` method as follows:

```
- (IBAction)onSaveDocument:(id)sender
{
    if (self.cloudDocument == nil)
        return;

    [documentContentView resignFirstResponder];
    self.cloudDocument.documentContent = documentContentView.text;

    [self.cloudDocument saveToURL:[self.cloudDocument fileURL]
                 forSaveOperation:UIDocumentSaveForCreating
                completionHandler:^(BOOL success)
     {
         if (success)
         {
             [self.cloudDocument openWithCompletionHandler:nil];
         }
     }];

}
```

This method dismisses the keypad if it is visible, and saves the `CloudTestDocument` object to the iCloud document.

7. Implement the `createDocument` method as follows:

```
- (void) createDocument
{
    if (self.cloudDocument == nil)
    {
        NSURL *containerURL = [[NSFileManager defaultManager]
                               URLForUbiquityContainerIdentifier:nil];
        NSURL* documentURL = [[containerURL
              URLByAppendingPathComponent:@"Documents]
              URLByAppendingPathComponent:@"cloudDocument.txt"];
        self.cloudDocument = [[CloudTestDocument alloc]
                                     initWithFileURL:documentURL];    }

    self.cloudDocument.documentContent = documentContentView.text;
```

```
        [self.cloudDocument saveToURL:[self.cloudDocument fileURL]
                    forSaveOperation:UIDocumentSaveForCreating
                   completionHandler:^(BOOL success)
         {
             if (success)
             {
                 [self.cloudDocument openWithCompletionHandler:nil];
             }
         }];
}
```

This method is used to create an empty file called `cloudDocument.txt` on iCloud, and is used when the `loadDocument` method could not find a document to load.

8. Implement the `refreshDocumentPreview:` method as follows:

```
- (void) refreshDocumentPreview:(NSNotification *)notification
{
    documentContentView.text = self.cloudDocument.documentContent;
}
```

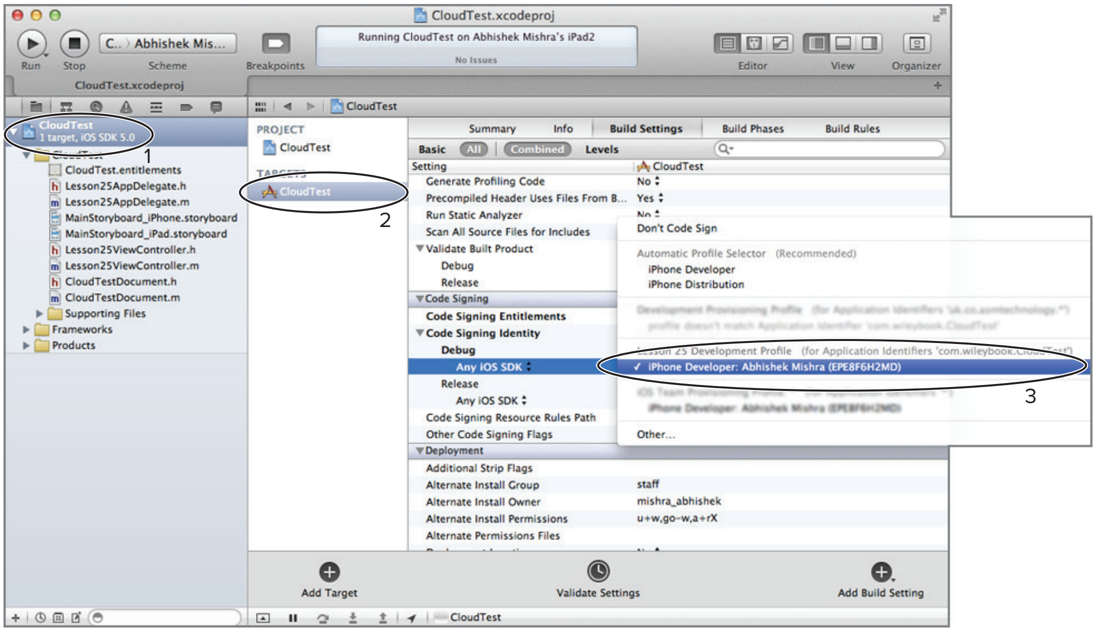
This method is received when the `CloudTestDocument` object loads data from the iCloud document `cloudDocument.txt`. Here, you simply refresh the user interface.

10. Test your app on an iOS device.

1. Connect your iOS device to your Mac.

2. Select your device from the Target/Device selector in the Xcode toolbar.

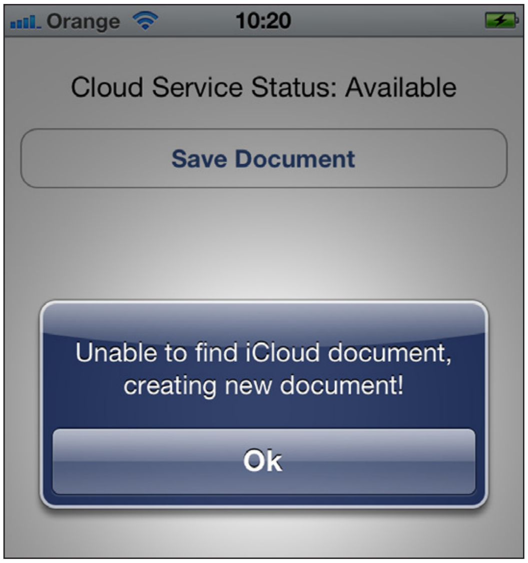
3. Ensure the correct value has been selected for the Code Signing Entity build settings of the application target (Figure 25-15).



Setting	CloudTest
Generate Profiling Code	No
Precompiled Header Uses Files From B...	Yes
Run Static Analyzer	No
Scan All Source Files for Includes	
▼Validate Built Product	
Debug	
Release	
▼Code Signing	
Code Signing Entitlements	
▼Code Signing Identity	
Debug	
Any iOS SDK	iPhone Developer: Abhishek Mishra (EPE8F6H2MD)
Release	
Any iOS SDK	
Code Signing Resource Rules Path	
Other Code Signing Flags	
▼Deployment	
Additional Strip Flags	
Alternate Install Group	staff
Alternate Install Owner	mishra_abhishek
Alternate Install Permissions	u+w,go-w,a+rX
Alternate Permissions Files	

FIGURE 25-15

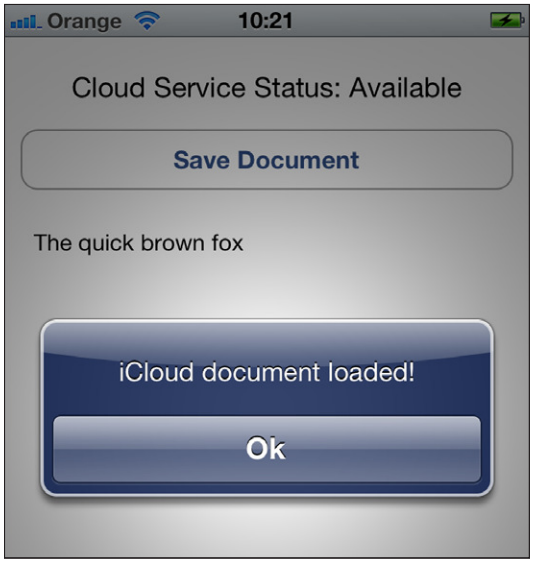
4. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
5. When you run the application for the first time, you will see a message similar to Figure 25-16, telling you that a new iCloud document is going to be created for you.
6. Type some text into the text view and tap the Save Document button.
7. If you now run this application on a different device, you will get a message similar to Figure 25-17 telling you that an existing iCloud document has been opened.



Cloud Service Status: Available

Unable to find iCloud document, creating new document!

FIGURE 25-16



Cloud Service Status: Available

The quick brown fox

iCloud document loaded!

FIGURE 25-17

Please select Lesson 25 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`, *to view the video that accompanies this lesson.*

26 Introduction to Core Data

The Core Data framework provides solutions to tasks commonly associated with managing the life-cycle of objects in your application, including object persistence. In this lesson you will learn to use Core Data to implement simple object persistence in your applications.

BASIC CONCEPTS

Core Data is based on the Model-View-Controller pattern and essentially fits in at the model stage. Core Data introduces a few new concepts and terminology, which are discussed briefly in this section.

Managed Object

A managed object is a representation of the object that you want to save to the data store. This is conceptually similar to a record in SQL and typically contains fields that correspond to properties in the object you want to save.

Managed Object Context

The managed object context is akin to a buffer between your application and the data store. It contains all your managed objects before they are written to the data store. Inside this context you can add, delete, or modify managed objects. Most of the time, when you need to read, insert, or delete objects you will call methods on the managed object context.

Persistent Store Coordinator

The persistent store coordinator represents the connection to the data store and contains low-level information like the actual name and location of the data store to be used. This class is generally used by the managed object context.

Managed Object Model

This is a class that contains definitions for each of the managed objects that you want to store in the data store. These definitions are also known as *entities*.

To use Core Data in your project, you first need to add a reference to the framework. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, switch to the Build Phases tab and click the + button under the Link Binary With Libraries category. Select CoreData.framework from the list of available frameworks (Figure 26-1).

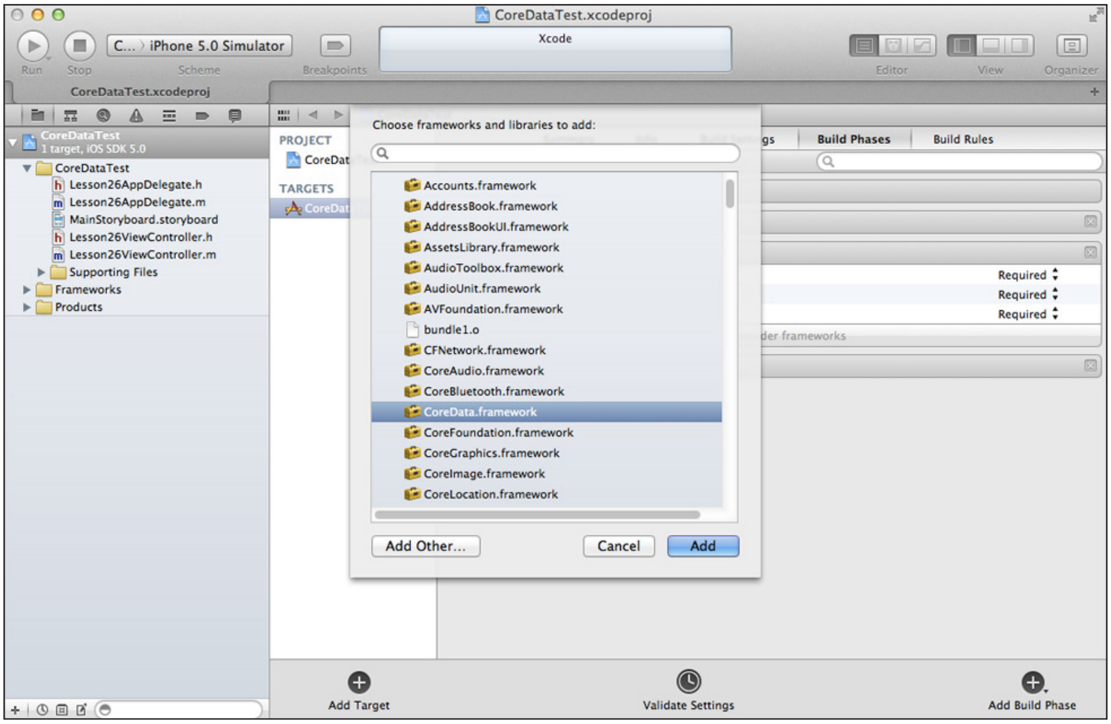


FIGURE 26-1

The next step is to create a managed object model for the project. To create an empty model file (into which you will later add entities), right-click the project group in the project navigator and select New File from the context menu. Select the Data Model template from the Core Data section and create the new file (Figure 26-2).

To open the model in the Xcode editor, simply click the file in the project navigator (the model file has the `.xcdatamodeld` extension). The new model file is initially empty (Figure 26-3), and as such is not much use to you in this state.

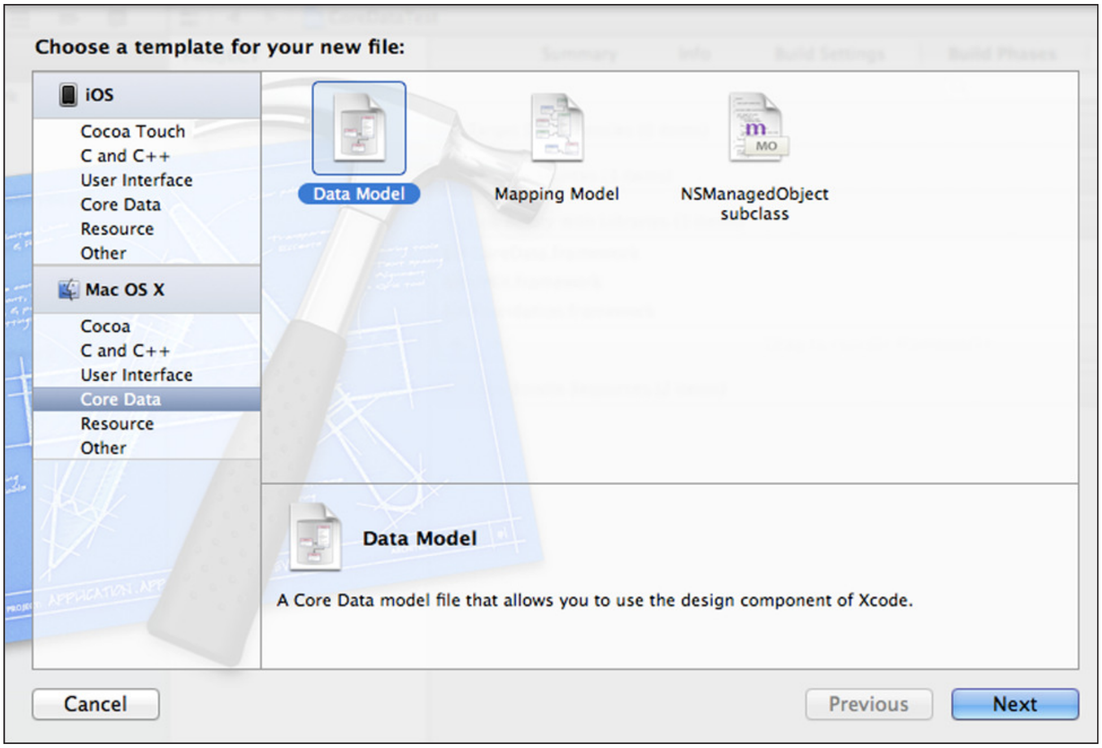


FIGURE 26-2

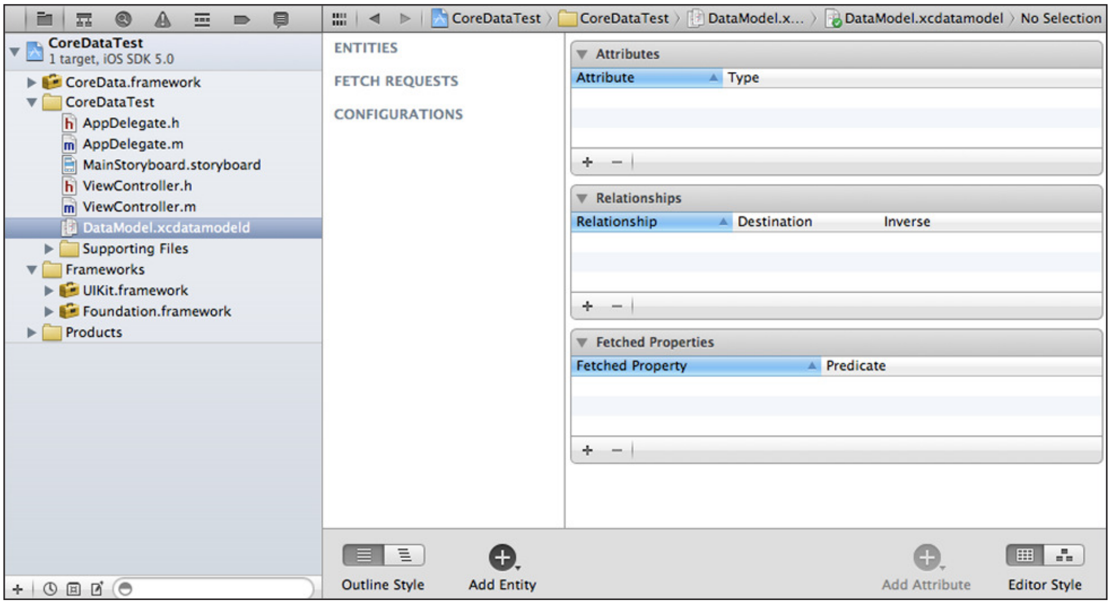


FIGURE 26-3

To persist objects into the underlying data store, you first need to define an entity in the data model for each object that you want to persist. Defining entities is trivial with the Xcode editor—to add a new entity called ContactData, select the Editor ➪ Add Entity menu item and name the new entity appropriately. You will see the new entity listed under the Entities section of the Xcode editor (Figure 26-4).

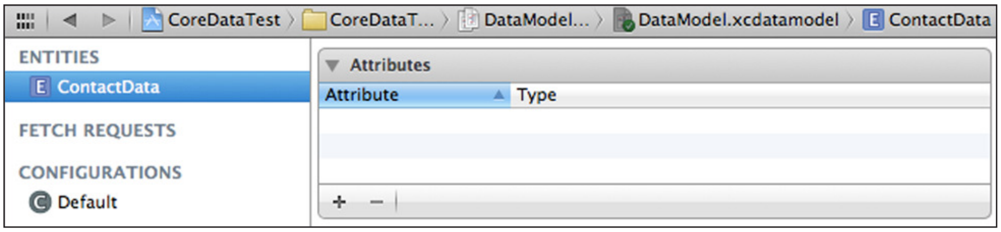


FIGURE 26-4

After you have defined an entity, you need to add attributes to it. Attributes represent the actual data fields in the entities themselves. Assuming the ContactData entity represents customer contact information, some of its attributes may be:

➤ Customer Name

➤ Phone Number

➤ Postcode

To add an attribute to the currently selected entity, select the Editor ➪ Add Attribute menu item. This adds a new row to the Attributes section of the Xcode model editor (Figure 26-5).

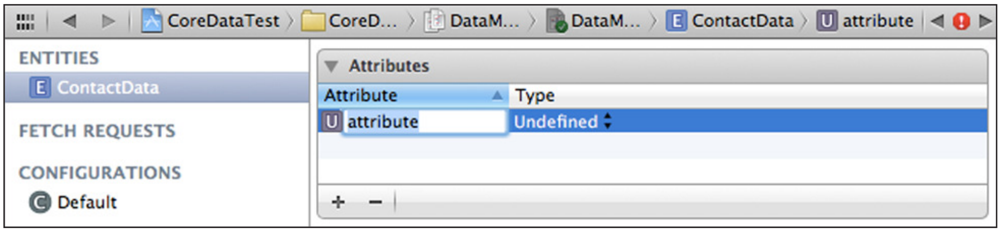


FIGURE 26-5

Type in an appropriate name for the attribute and specify the attribute type. The attribute type is similar to the data type of a variable, and determines what type of data the attribute contains. Core Data provides several data types that can be selected from a drop-down list (Figure 26-6). The type for each attribute of the ContactData entity can be String.

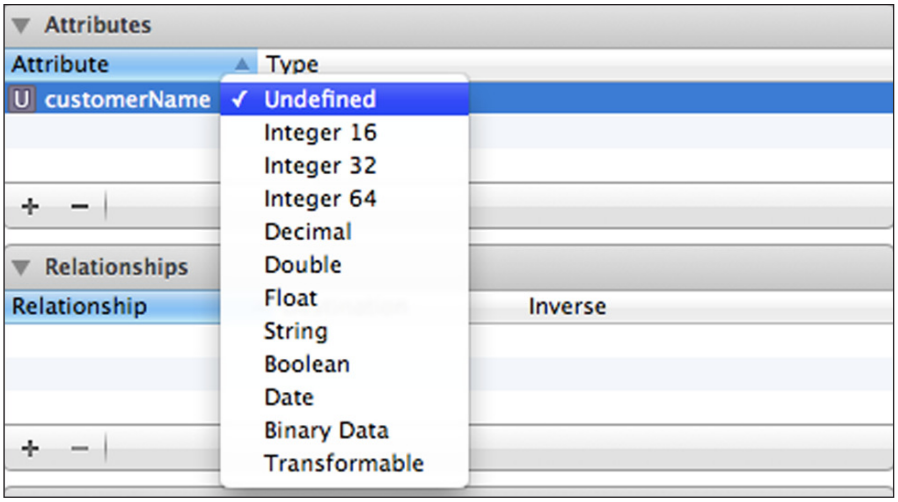


FIGURE 26-6

At this stage you have created a new data model, and added an entity to it. Now you need an actual Objective-C class that maps to the entity defined in the model. To do this, select the Editor ➪ Create NSManagedObject Subclass menu item. This presents a dialog box asking you where to save the `.h` and `.m` files for the new class. The name of the class will be the same as the name of the entity. The `ContactData` class that is created for you by Xcode is a subclass of `NSManagedObject` and maps to the entity with the same name. Its interface is listed below:

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

@interface ContactData : NSManagedObject

@property (nonatomic, retain) NSString * customerName;
@property (nonatomic, retain) NSString * phoneNumber;
@property (nonatomic, retain) NSString * postCode;

@end
```

INSTANTIATING CORE DATA OBJECTS

Before you can read or write model objects to the underlying data store, you will need to instantiate the managed object model, the managed object context, and the persistent store coordinator.

The managed object model is represented by an instance of the `NSManagedObjectModel` class, and you instantiate a single instance for all the `.xcdatamodeld` files in your project as follows:

```
NSManagedObjectModel* managedObjectModel = [NSManagedObjectModel
                                    mergedModelFromBundles:nil];
```

The `mergedModelFromBundles:` method searches the project for all files that have an `.xcdatamodeld` extension and loads all the entities into a single `NSManagedObjectModel` instance.

Once you have an `NSManagedObjectModel` instance, you can create an instance of the `NSPersistentStoreCoordinator` class, which represents the persistent store coordinator. Recall that the persistent store coordinator handles the low-level connection with underlying data stores. Individual databases are referred to as *persistent stores*.

To create an `NSPersistentStoreCoordinator` instance, use the following snippet:

```
NSPersistentStoreCoordinator* peristentStoreCoordinator =
                              [[NSPersistentStoreCoordinator alloc]
                     initWithManagedObjectModel:managedObjectModel];
```

Once you have the store coordinator, you need to give it a data store to manage. You do this by sending the `addPersistentStoreWithType:configuration:URL:options:error:` message to the store coordinator object. For instance, the following code snippet sets up a SQLite database as the data store:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory ,
                                                     NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
NSString* filePath = [documentsDir
                      stringByAppendingPathComponent:@"datastore.sqlite"];
NSURL *databaseURL = [NSURL fileURLWithPath:filePath];

NSError* error = nil;
[peristentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                           configuration:nil
                           URL:databaseURL
                           options:nil
                           error:&error];
```

Finally, with the store coordinator object in place, it is time to instantiate a managed object context. Recall that a managed object context is like a buffer where you place your managed objects before writing to (or reading from) the database. The managed object context is represented by an instance of the `NSManagedObjectContext` class and can be created as follows:

```
NSManagedObjectContext* managedObjectContext = [[NSManagedObjectContext alloc] init];
[managedObjectContext setPersistentStoreCoordinator:peristentStoreCoordinator];
```

WRITING MANAGED OBJECTS

Instantiating a managed object is slightly different from the usual `alloc` and `init` process. With managed objects, you allow Core Data to instantiate them within a managed object context. Once the object has been instantiated, you can use it as you would any other object. To instantiate a `ContactData` object, use the following code:

```
ContactData* newContact = (ContactData*)[NSEntityDescription
                insertNewObjectForEntityForName:@"ContactData"
                inManagedObjectContext:managedObjectContext];
```

Now that you have instantiated a `ContactData` object, you can set up its attributes just like you would for any object:

```
newContact.customerName = @"John Smith";
newContact.phoneNumber = @"+44 78901 78192";
newContact.postcode = @"PB2 7YK";
```

To write managed objects to the data store, simply call the `save` method of the managed object context. Doing so saves any new objects to the underlying data store (by using the persistent store coordinator). The `save` method returns a Boolean value indicating success or failure.

```
NSError* error;
BOOL result = [managedObjectContext save:&error];
```

READING MANAGED OBJECTS

Reading objects from a data store with Core Data is quite straightforward. You simply create an appropriate fetch request and ask the managed object context to execute the request. The managed object context will then return an array of objects read from the data store.

A fetch request is an instance of the `NSFetchRequest` class, and while creating one you need to specify the entity that you want to fetch. The entity has to be one that exists in the data model. To create a fetch request that retrieves all `ContactData` entities from the data store, use the following code:

```
NSEntityDescription* entityDescription = [NSEntityDescription
                                         entityForName:@"ContactData"
                          inManagedObjectContext:managedObjectContext];

NSFetchRequest* fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:entityDescription];
```

To retrieve an array of managed objects from the data store, you need to ask the managed object context to execute the fetch request, as shown in the following snippet:

```
NSError* error;
NSArray* existingCustomers = [managedObjectContext
executeFetchRequest:fetchRequest
error:&error];
```

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `CoreDataTest` that can serialize/de-serialize object data to an SQLite database using Core Data.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a reference to the Core Data framework.
- Add a Core Data model to the project.
- Add an entity to the data model.
- Create an `NSManagedObject` subclass.
- Create a simple user interface with a storyboard.
- Initialize Core Data objects.

- Save managed objects to the database with Core Data.
- Read managed objects from the database with Core Data.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 26 folder in the download.*

Hints

- To use Core Data in a project, you must add a reference to the appropriate framework.

Step-by-Step

1. Create a Single View Application in Xcode called `CoreDataTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** CoreDataTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson26
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a reference to the Core Data framework.
    1. Make sure the project navigator is visible.
    2. Click the root (project) node of the project navigator to display the project settings.
    3. Select the Build Phases tab.

    4. Expand the Link Binary With Libraries group in this tab.
    5. Click the + button at the bottom of this group and select `CoreData.framework` from the list of available frameworks.
    6. Click the Add button.
3. Create a data model for the project.
    1. Right-click the `CoreDataTest` group in the project navigator and select New File from the context menu.
    2. Select the Data Model template from the Core Data section. Save the file as `DataModel`.
4. Edit the data model file.
    1. Select the `DataModel.xcdatamodeld` file in the project navigator to open it in the Xcode editor.
    2. Select the Editor ➪ Add Entity menu item and name the new entity `ContactData`.
    3. Add attributes to the `ContactData` entity.
        1. Select the Editor ➪ Add Attribute menu item to create a new attribute. Name it `customerName` and set its type to `String`.
        2. Add two more `String` attributes, `phoneNumber` and `postCode`, to the entity.
5. Select the Editor ➪ Create NSManagedObject Subclass menu item. Accept the default file location and click Save to create a new class called `ContactData` in your project.
6. Create a simple user interface using a storyboard.
    1. Open the `MainStoryboard.storyboard` file in Interface Builder.
    2. From the Object library, drag and drop five Label objects, three Text Field objects, one Round Rect Button object, and one Table View object onto the scene.
    3. Arrange these objects to resemble Figure 26-7.
    4. Create three outlets in the view controller class corresponding to the three Text Field objects in the scene. Name the outlets `nameField`, `phoneField`, and `postcodeField`, respectively.
    5. Create an action method called `onAdd` in the view controller class and connect it to the Touch Up Inside event of the Add New Record button.
    6. Create an outlet in the view controller class corresponding to the Table View object in the scene. Name the outlet `tableOfContacts`.
7. Initialize Core Data objects.
    1. Import the `CoreData.h` header file at the top of the `Lesson26ViewController.h` file by adding this line:

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

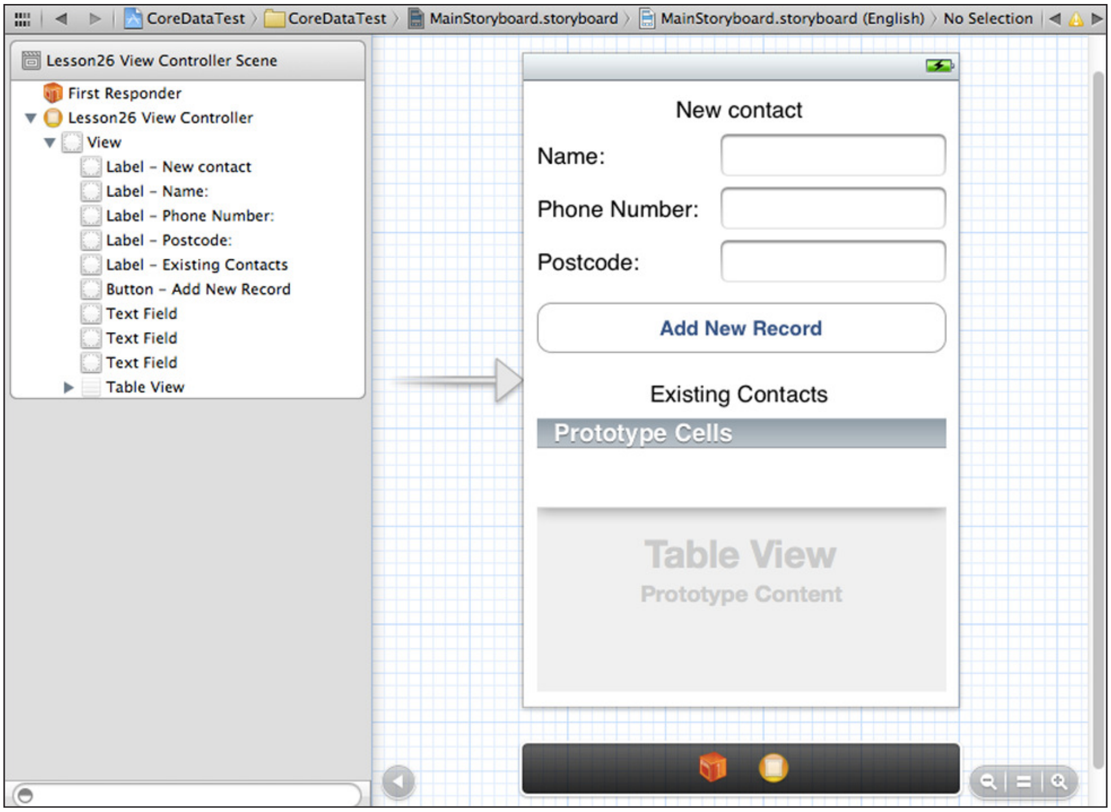


FIGURE 26-7

2. Add the following property declarations to the `Lesson26ViewController.h` file:

```
@property (nonatomic, strong) NSManagedObjectModel* objectModel;
@property (nonatomic, strong) NSPersistentStoreCoordinator* coordinator;
@property (nonatomic, strong) NSManagedObjectContext* objectContext;
```

3. Synthesize the properties in the `Lesson26ViewController.m` file.

4. Add the following code to the `viewDidLoad` method of your view controller class after the `[super viewDidLoad]` line.

```
// create managed object model
objectModel = [NSManagedObjectModel mergedModelFromBundles:nil];

// create persistent store coordinator
coordinator = [[NSPersistentStoreCoordinator alloc]
               initWithManagedObjectModel:objectModel];

// add persistent store
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                     NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
NSString* filePath = [documentsDir
```

```
                     stringByAppendingPathComponent:@"datastore.sqlite"];
NSURL *databaseURL = [NSURL fileURLWithPath:filePath];

NSError* error = nil;
[coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                        configuration:nil
                                        URL:databaseURL
                                        options:nil
                                        error:&error];

// create managed object context
objectContext = [[NSManagedObjectContext alloc] init];
[objectContext setPersistentStoreCoordinator:coordinator];
```

8. Save a managed object to the database when the Add button is tapped.

    1. Add the following code to the implementation of the `onAdd` method in the view controller class. This code creates an instance of the `ContactData` class in the managed object context, sets up its properties using values entered by the user in the text fields, and saves the object.

    ```
    NSString* newName = nameField.text;
    NSString* newPhone = phoneField.text;
    NSString* newPostcode = postcodeField.text;

    ContactData * newContact = (ContactData*)[NSEntityDescription
                          insertNewObjectForEntityForName:@"ContactData"
                          inManagedObjectContext:objectContext];

    newContact.customerName = newName;
    newContact.phoneNumber = newPhone;
    newContact.postCode = newPostcode;

    NSError* error;
    if ([objectContext save:&error])
    {
        [self fetchExistingContactData];
        [tableOfContacts reloadData];
    }

    // hide keyboard.
    [phoneField resignFirstResponder];
    [nameField resignFirstResponder];
    ```

    2. Import the definition of the `ContactData` class by adding the following line to the top of the `Lesson26ViewController.m` file:

    ```
    #import "ContactData.h"
    ```

9. Read managed objects from the database and display them in a table view.

    1. Ensure the `Lesson26ViewController` class implements the `UITableViewDataSource` and `UITableViewDelegate` protocols by changing its interface declaration to the following:

    ```
    @interface Lesson26ViewController : UIViewController
                                        <UITableViewDataSource,
                                        UITableViewDelegate>
    ```

2. Add the following property to the `Lesson26ViewController.h` file:

```
@property (strong, nonatomic) NSArray* existingContacts;
```

3. Synthesize the property in the `Lesson26ViewController.m` file.

4. Define a new method in the `Lesson26ViewController.h` file called `fetchExistingContactData`:

```
- (void) fetchExistingContactData;
```

5. Add the following lines of code to the end of the `viewDidLoad` method in the `Lesson26ViewController.m` file. These lines set up the `datasource` and `delegate` properties of the table view object and call the `fetchExistingContactData` method.

```
// setup datasource and delegate for tableView
tableOfContacts.dataSource = self;
tableOfContacts.delegate = self;

// get rows from database
[self fetchExistingContactData];
```

6. Implement the `fetchExistingContactData` method in the `Lesson26Viewcontroller.m` file as follows:

```
- (void) fetchExistingContactData
{
    NSFetchRequest* fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription* entityDescription = [NSEntityDescription
                                  entityForName:@"ContactData"

                          inManagedObjectContext:objectContext];
    [fetchRequest setEntity:entityDescription];

    NSError* error;
    existingContacts = [objectContext executeFetchRequest:fetchRequest
                                                 error:&error];
}
```

7. Implement `UITableViewDataSource` and `UITableViewDelegate` methods in the `Lesson26ViewController.m` file as follows:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)aTableView
             numberOfRowsInSection:(NSInteger)section
{
    return [existingContacts count];
}
- (UITableViewCell *)tableView:(UITableView *)aTableView
                  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [aTableView dequeueReusableCellWithIdentifier
:@"Cell"];
    if (cell == nil)
```

```
        {
            cell = [[UITableViewCell alloc]
                    initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:@"Cell"];
        }

        ContactData* data = (ContactData*)[existingContacts
                                   objectAtIndex:indexPath.row];
        [[cell textLabel] setText:data.customerName];

        return  cell;
    }
```

10. Add a tap gesture recognizer and use it to dismiss the keyboard when the background area of the view is tapped.

   1. Add the following method declaration to the `Lesson26ViewController.h` file:

   ```
   - (void) handleBackgroundTap:(UITapGestureRecognizer*)sender;
   ```

   2. Add the following code to the end of the `viewDidLoad` method of the view controller class:

   ```
   UITapGestureRecognizer* tapRecognizer = [[UITapGestureRecognizer alloc]
                                          initWithTarget:self
                                  action:@selector(handleBackgroundTap:)];
   tapRecognizer.cancelsTouchesInView = NO;
   [self.view addGestureRecognizer:tapRecognizer];
   ```

   3. Implement the `handleBackgroundTap:` method in the `Lesson26ViewController.m` file as follows:

   ```
   - (void) handleBackgroundTap:(UITapGestureRecognizer*)sender
   {
       [phoneField resignFirstResponder];
       [nameField resignFirstResponder];
       [postcodeField resignFirstResponder];

   }
   ```

11. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

Please select Lesson 26 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

27 XML Parsing with NSXMLParser

XML is an extremely popular format for data interchange and is used widely in desktop, mobile, and web applications. In this lesson you learn to parse XML documents in your applications.

XML stands for Extensible Markup Language and is a text-based markup language that lets you define the structure of a document. It is primarily used as a means to store and transfer data. Because it is text-based, XML files can be created and edited using almost any text editor capable of editing plain text files. If you decide to use TextEdit to edit XML files, be sure to use the Format ➪ Make Plain Text menu item to ensure that TextEdit treats the XML document as plan text and not RTF.

Essentially, the author of an XML document creates structure within the document by creating several tags and inserting the content of the document within these tags. This is perhaps best understood with a simple example:

```
<contact>
    <first_name>John</first_name>
    <last_name>Doe</last_name>
    <address>15 Bilton Road, Perivale</address>
</contact>
```

The preceding snippet is a simple XML document that could be used to store information about a single contact, perhaps as part of a contacts management application.

The key thing to note is that XML is just a means to store/exchange data. There is no fixed set of tags that must be used in an XML document; you can create your own tags to structure your data.

An XML document on its own does not do anything. You need one or more applications that can use the data. These applications could be desktop-based, web-based, or mobile-based.

If you are writing such applications yourself, you can use your own set of tags in the document. If you need to create an XML document that will be used with a third-party application, you will need to work with tags that the application can understand.

If you are familiar with HTML you may have noticed the similarity between XML and HTML. Although similar, these are not the same. HTML is a markup language intended to display data. XML is concerned only with storage of data.

XML FUNDAMENTALS

In this section, you learn the basics of XML. XML is conceptually a very simple language. An XML document consists of:

- Elements and tags
- Attributes
- Special characters
- Comments
- Processing instructions

Each of these items is discussed in the following sections.

Elements and Tags

XML documents are simply text documents that have been marked with tags into a series of elements. XML requires that every start tag have a matching end tag. Hence, each element consists of a start tag and an end tag. Start tags begin with < and end tags begin with `</`. For example:

```
<name>John</name>
```

XML tag names are case-sensitive, thus `<name>` and `<Name>` are not the same tag. The text between a pair of matching start and end tags is called the *content* of the element. The content is usually text, but could also be one or more elements.

When an element contains one or more elements as part of its content, the XML document structure begins to resemble a tree. The root element of the document is a special element that does not have any parent and thus contains every other element. Every other element in an XML document has a parent element. In the following example, the root element is `contact`, and it is the parent of the `first_name`, `last_name`, and `address` elements.

```
<contact>
    <first_name>John</first_name>
    <last_name>Doe</last_name>
    <address>170 Bilton Road, Perivale</address>
</contact>
```

If an element contains no content, it is called an empty element, and is represented using special syntax:

```
<empty_element/>
```

The above line is equivalent to:

```
<empty_element></empty_element>
```

Attributes

An XML element may contain attributes in the form of name-value pairs. Attributes are always specified in the start tag of an element, and have the following syntax:

```
attribute_name="value"
```

A given element can have multiple attributes; each pair separated with whitespace as shown here:

```
<name employee_id="1" department_name ="Sales">John</name>
```

Special Characters

Certain characters cannot be used as part of the content between start and end tags, or attribute values. These characters have special meaning to XML parsers and thus cannot be used as part of your data. If you need to use these characters, you will have to use a special character sequence for each. In technical terms, you will need to use a specific escape sequence for these special characters as shown in Table 27-1.

TABLE 27-1: Reserved Characters in XML

RESERVED CHARACTER	ESCAPE SEQUENCE
`>`	`&gt;`
`<`	`&lt;`
`&`	`&amp;`
`"`	`&quot;`
`'`	`&apos;`

This is an example of bad XML:

```
<company_name>Allen & Peters Plumbing</company_name>
```

The corrected version is:

```
<company_name>Allen &amp; Peters Plumbing</company_name>
```

Comments and Processing Instructions

Comments can be inserted in an XML document by placing the comment between `<!--` and `--!>`. Comments are not allowed inside attribute values, or start and end tags.

```
<!-- this is a valid xml comment --!>
```

Processing instructions are special instructions intended for specific applications. Every XML document that you want to parse using an `NSXMLParser` object must begin with the following processing instruction:

```
<?xml version= "1.0" encoding="UTF8"?>
```

THE NSXMLPARSER CLASS

The `NSXMLParser` class is part of the Cocoa framework and you can use its instances to parse XML documents. The `NSXMLParser` class is a SAX-based parser (SAX stands for Serial API for XML), and thus implements an event-based approach to XML parsing.

The `NSXMLParser` class requires you to provide a delegate object. It sequentially examines the content of the XML document and informs the delegate object as it encounters elements. It is up to the delegate object to do something with the data that has been encountered.

It is important to note that you cannot make the `NSXMLParser` object begin parsing from the middle of the XML document and stop parsing when a particular element of interest has been read. The parser will parse the entire document from start to finish, even if all you are interested in is the value of one attribute in a specific element toward the end of the document.

Parsing can take a considerable amount of time, especially if the XML document is large. For this reason it is not recommended to parse a document multiple times. In most cases you will parse the document only once, and your delegate object will populate an application-specific data model in memory with data from the XML file. For all subsequent access to the data, your code will use your application-specific data model and not parse the XML file repeatedly.

In addition to parsing the XML document and informing the delegate object, the `NSXMLParser` class also performs basic validation on the XML document to check if it is well-formed.

This check typically involves making sure that every start tag has a matching end tag, attributes have values, and that the XML document contains valid characters (certain characters are not allowed in an XML document; refer to the “Special Characters” section earlier).

If any problems are encountered, the `NSXMLParser` class calls an appropriate error method on the delegate object and stops parsing the document.

SAX and DOM Parsers

When it comes to parsing XML documents, a parsing application can take two general approaches:

- **Event-based:** Parsers that opt for this approach parse the document sequentially from start to finish, and raise appropriate events as the file is parsed. Events are typically raised when a tag begins, when a tag ends, when an error is encountered, and so on. These parsers are said to be SAX-based.
- **Tree-based:** Parsers that opt for this approach load the entire XML document into a tree-like representation in memory. A node in this tree usually corresponds to an element in the XML document. The application then has the task of traversing this tree to a specific node. These parsers are said to be DOM-based (DOM stands for document object model).

SAX-based parsers require very little memory to implement because all they really need to store is the data for the particular element that is being parsed. DOM-based parsers, on the other hand, need more memory because they need to load the entire document.

However, DOM-based parsers do allow random access to data, and the ability to verify the structure of the document to ensure that all required tags are present.

THE NSXMLPARSERDELEGATE PROTOCOL

The delegate object that you need to supply to the `NSXMLParser` instance must implement the `NSXMLParserDelegate` protocol. The protocol defines several optional methods. The four most commonly used methods that you are likely to implement are:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
  attributes:(NSDictionary *)attributeDict;

- (void)parser:(NSXMLParser *)parser didEndElement:(NSString*)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName;

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string;

- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError;
```

The `parser:didStartElement:namespaceURI:qualifiedName:attributes:` and `parser:didEndElement:namespaceURI:qualifiedName:` delegate methods are called when the parser encounters a start tag or an end tag. The first parameter to each method contains a reference to the parser. The name of the tag is supplied in the `elementName` parameter.

Attributes that may be part of the start tag are supplied as a dictionary of attribute-value pairs in the `attributeDict` parameter.

Attributes are specified only in start tags, and are thus available only in the `didStartElement:namespaceURI:qualifiedName:attributes` *delegate method. If you need to do something with the attributes in a start tag you must put relevant code in this method of your delegate object.*

The `namespaceURI` and `qualifiedName` parameters deal with the concept of XML Namespaces, which is beyond the scope of this book.

The `parser:foundCharacters:` delegate method is called by the parser when one or more characters are encountered between start and end tags. The characters themselves can be accessed through the `string` parameter.

The parser does not guarantee to pass all the characters that may be found between start and end tags in a single call. This method is usually called several times to cover all the characters that are in between a start and end tag.

For instance, if the XML document had a pair of start and end tags like this:

```
<first_name>John</first_name>
```

the parser may call this delegate method either:

- Once with the entire text `John`
- Four times with each character `J o h n`
- Any combination in between

The `parser:parseErrorOccurred:` delegate method is called by the parser if it encounters a fatal error while parsing the XML document. Parsing stops after this method is called.

There can be many reasons why the error occurred in the first place. The most common ones are:

- Start tags that do not have end tags
- Attributes that do not have values
- Invalid characters in the XML document

A SIMPLE XML FILE AND HOW IT IS PARSED

The best way to understand how to use the `NSXMLParser` class is through an example. Listing 27-1 represents a simple XML file called `contacts.xml` that could perhaps be used in an address book application to store information on various contacts.

LISTING 27-1: contacts.xml

```
<?xml version= "1.0" encoding="UTF8">
<contacts>
   <contact id="1">
 <first_name>John</first_name>
 <last_name>Doe</last_name>
 <address>170 Bilton Road, Perivale</address>
 <phone>44-798-12871</phone>
   </contact>
   <contact id="2">
 <first_name>Paul</first_name>
 <last_name>Bridges</last_name>
 <address>17A Heathfield gardens, Brent Cross</address>
 <phone>44-701-57358</phone>
   </contact>
</contacts>
```

In the listing, XML start and end tags are highlighted in boldface. The root element of the XML document is `<contacts>`, and data on each contact is stored in a `<contact>` element. The document contains two `<contact>` elements in all. For each contact the document contains the data shown in Table 27-2.

TABLE 27-2: Contact Fields

FIELD NAME	NOTES
ID	Attribute of the `<contact>` start tag
Name	Content of the `<first_name>` element

FIELD NAME	NOTES
Surname	Content of the `<last_name>` element
Address	Content of the `<address>` element
Phone number	Content of the `<phone>` element

Each `<contact>` element in the XML file will map to an instance of a data model class in the application. An interface of a data model class that could be used to encapsulate the contents of a `<contact>` element is presented next. The name of this class is `ContactInfo`.

```
@interface ContactInfo : NSObject

@property int contactId;
@property (nonatomic, strong) NSMutableString* contactName;
@property (nonatomic, strong) NSMutableString* contactSurname;
@property (nonatomic, strong) NSMutableString* contactAddress;
@property (nonatomic, strong) NSMutableString* contactPhone;

@end
```

To manage multiple `ContactInfo` instances (corresponding to multiple contact elements in the XML file), an application will typically use an `NSMutableArray` instance. In simple applications, this array could be a member of the view controller class.

Loading the XML Document into an NSData Object

The first step towards to parsing the XML document is to load the document into an `NSData` object. This step is essentially reading the file from beginning to end, character by character, and loading it into memory. At this point the application is not trying to make sense of (parse) the contents of the file while it is being loaded. It is simply creating a block of memory into which the contents of the file are placed. The following code snippet shows how `contacts.xml` could be loaded into an `NSData` instance.

```
NSBundle* bundle = [NSBundle mainBundle];
NSString* filePath = [bundle pathForResource:@"contacts" ofType:@"xml"];
NSData* xmlData = [NSData dataWithContentsOfFile:filePath];
```

Instantiating an NSXMLParser Object

The next step is to instantiate an `NSXMLParser` object, giving it a reference to the `NSData` object that contains XML content to be parsed. This is done by using the `alloc` and `initWithData:` methods as shown below:

```
NSXMLParser *xmlParser = [[NSXMLParser alloc] initWithData:xmlData];
```

You may be wondering why the XML file should be loaded into an `NSData` object at all. Why not just give the `NSXMLParser` object the path to the file straight away?

The answer is that doing so adds a level of abstraction that decouples the process of parsing XML content from the manner in which it is stored physically. As long as you can get the XML data into an `NSData` object, you can parse it with an `NSXMLParser` object.

This becomes particularly relevant when you use an NSXMLParser object to parse XML data that is downloaded from the Internet, perhaps as a response from a web service. In this case the downloaded data will be made available to your application by the relevant Cocoa framework as an NSData object.

Instantiating a Delegate Object

Having instantiated an NSXMLParser object, you will need to provide a delegate object. The delegate object must implement the NSXMLParserDelegate protocol.

In most applications you will create a separate NSObject subclass that implements the NSXMLParserDelegate protocol and use an instance of this class as the delegate. For instance, if the name of this class was XMLParserDelegate then the following code snippet could be used to set up the delegate property of the XML parser:

```
XMLParserDelegate *parserDelegate = [[XMLParserDelegate alloc] init];
[xmlParser setDelegate:parserDelegate];
```

Begin Parsing

The actual parsing begins by calling the parse method on the NSXMlParser instance:

```
[xmlParser parse];
```

At this point the NSXMLParser object will begin sequential parsing and will call one or more methods on your delegate object as XML content is encountered.

The XMLParser Delegate Methods

The code to do something when elements in the XML file are encountered is part of the delegate object. The delegate object, thus, does the bulk of the work. The XMLParserDelegate class, an instance of which is used as the delegate object to parse contacts.xml, could be defined as:

```
#import <Foundation/Foundation.h>
#import "ViewController.h"
#import "ContactInfo.h"

@interface XMLParserDelegate : NSObject <NSXMLParserDelegate>

@property (nonatomic, weak) ViewController* viewController;
@property (nonatomic, strong) ContactInfo* tmpContactInfo;
@property (nonatomic, strong) NSMutableString* currentElementValue;

@end
```

The implementation of the parser:didStartElement:namespaceURI:qualifiedName:attributes: delegate method is presented below:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qualifiedName
  attributes:(NSDictionary *)attributeDict
{
    // clear the contents of current_element_value
    if (self.currentElementValue == nil)
        self.currentElementValue = [[NSMutableString alloc] initWithString:@""];
    else
        [currentElementValue setString:@""];

    // set 'tmpContactInfo' to a new ContactInfo object each time
    // the start of a 'contact' element is been encountered
    if([elementName isEqualToString:@"contact"])
    {
        tmpContactInfo = [[ContactInfo alloc] init];

        // read the attributes of the node here.
        NSString* szID = [attributeDict objectForKey:@"id"];
        if (szID != nil)
            tmpContactInfo.contactId = [szID intValue];
    }
}
```

This method is called when the parser encounters the start of each XML element. When the `contacts.xml` file is being parsed, this method will be called 11 times, once for each XML start tag encountered, as shown in Listing 27-2.

LISTING 27-2: Start tags in contacts.xml

```
<?xml version= "1.0" encoding="UTF8">
<contacts>
   <contact id="1">
<first_name>John</first_name>
<last_name>Doe</last_name>
<address>170 Bilton Road, Perivale</address>
<phone>44-798-12871</phone>
   </contact>
   <contact id="2">
<first_name>Paul</first_name>
<last_name>Bridges</last_name>
<address>17A Heathfield gardens, Brent Cross</address>
       <phone>44-701-57358</phone>
   </contact>
</contacts>
```

The implementation of this delegate method assigns a new `NSMutableString` instance to `currentElementValue`. If `currentElementValue` is not nil, then its contents are cleared.

```
        if (self.currentElementValue == nil)
            self.currentElementValue = [[NSMutableString alloc] initWithString:@""];
        else
            [currentElementValue setString:@""];
```

The `elementName` parameter is examined next, to check if the parser is dealing with the start tag of the `<contact>` element. If this is so, then `tmpContactInfo` is assigned a reference to a new `ContactInfo` instance.

During the parsing process, `tmpContactInfo` refers to the current `ContactInfo` instance that is being set up as the document is being parsed. As the parser encounters other XML tags, appropriate member variables of `tmpContactInfo` will be set up. This will go on until the end tag of the `<contact>` element is encountered, at which point `tmpContactInfo` is considered to be ready to use.

The start tag of the `<contact>` element also happens to contain a single attribute `id`. This is read into the `contactId` member variable of `tmpContactInfo` using the following code.

```
NSString* szID = [attributeDict objectForKey:@"id"];
if (szID != nil)
   tmpContactInfo.contactId = [szID intValue];
```

Recall that attributes are only available when start tags are encountered. If the application has any interest in an attribute it must process it when its corresponding start tag is encountered.

The implementation of the `parser:foundCharacters:` delegate method is next:

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    [currentElementValue appendString:string];
}
```

This method is called when one or more characters of text are encountered between start and end tags. Unfortunately this delegate method does not provide information on the name of the current element being processed at the point this call was made. In this method, the characters that are encountered are appended to the existing value of the `currentElementValue` variable.

In essence, the `currentElementValue` variable acts like an accumulator, collecting characters as they arrive. If you clear the content of `currentElementValue` at the start of each element (in the `parser:didStartElement:namespaceURI:qualifiedName:attributes:` method), you can safely assume it will have meaningful content when the end of each element is encountered.

The implementation of the `parser:didEndElement:namespaceURI: qualifiedName:` delegate method is slightly longer, as is presented below:

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
   namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
    if([elementName isEqualToString:@"first_name"])
    {
        if (tmpContactInfo.contactName == nil)
            tmpContactInfo.contactName = [[NSMutableString alloc]
                                          initWithCapacity:25];
        [tmpContactInfo.contactName setString:currentElementValue];
    }

    if([elementName isEqualToString:@"last_name"])
    {
        if (tmpContactInfo.contactSurname == nil)
            tmpContactInfo.contactSurname = [[NSMutableString alloc]
                                            initWithCapacity:25];
        [tmpContactInfo.contactSurname setString:currentElementValue];
    }

    if([elementName isEqualToString:@"address"])
    {
        if (tmpContactInfo.contactAddress == nil)
            tmpContactInfo.contactAddress = [[NSMutableString alloc]
                                            initWithCapacity:25];
        [tmpContactInfo.contactAddress setString:currentElementValue];
    }

    if([elementName isEqualToString:@"phone"])
    {
        if (tmpContactInfo.contactPhone == nil)
            tmpContactInfo.contactPhone = [[NSMutableString alloc]
                                               initWithCapacity:25];
        [tmpContactInfo.contactPhone setString:currentElementValue];
    }

    if([elementName isEqualToString:@"contact"])
    {
        [viewController.listOfContacts addObject:tmpContactInfo];
        tmpContactInfo = nil;
    }
}
```

This method is called when the parser encounters the end of each XML element. When the `contacts.xml` file is being parsed, this method will also be called 11 times, as shown in Listing 27-3.

LISTING 27-3: End tags in contacts.xml

```
<?xml version= "1.0" encoding="UTF8">
<contacts>
   <contact id="1">
<first_name>John</first_name>
<last_name>Doe</last_name>
<address>170 Bilton Road, Perivale</address>
<phone>44-798-12871</phone>
   </contact>
   <contact id="2">
<first_name>Paul</first_name>
<last_name>Bridges</last_name>
<address>17A Heathfield gardens, Brent Cross</address>
<phone>44-701-57358</phone>
   </contact>
</contacts>
```

When the end of the `<first_name>`, `<last_name>`, `<address>`, and `<phone>` elements are encountered, the text of the element will be contained in `currentElementValue`. All you need to do then is set up appropriate member variables in the current `ContactInfo` instance `tmpContactInfo`.

When the end of the `<contact>` element is encountered, this is taken to signify that the current `ContactInfo` object `tmpContactInfo` is ready to use, and is thus added to an `NSMutableArray` in the view controller class.

Error checking has been intentionally omitted in this example to focus on the task of XML parsing. In a real-world application, at the very least you would want to examine the contents of the `listOfContacts` array to make sure it has some data in it.

TRY IT

In this Try It, you build you build a new Xcode project based on the Single View Application template called `ContactSample` that loads a list of contacts from an XML file and display the name of each contacts in a table view.

Lesson Requirements

- Create a new iPhone application project based on the Single View Application template.
- Import an XML file into the project.
- Create an `NSObject` subclass called `ContactData` that will encapsulate information on each contact.
- Create an `NSObject` subclass called `ContactDataXMLParserDelegate` that implements the `NSXMLParserDelegate` protocol.
- Add an `NSMutableArray` to the view controller class.
- Create a simple user interface consisting of a button and a table view.
- When the button is tapped, load the XML file and update the table view.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 27 folder in the download.*

Hints

- Launch Xcode from the `/Developer/Applications` folder.

Step-by-Step

1. Create a Single View Application in Xcode called `ContactSample`.
    1. Launch Xcode from the `/Developer/Applications` folder.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** ContactSample
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson27
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked

- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Copy `contacts.xml` from the resources folder of this lesson on the DVD into the Xcode project.
3. Create a new `NSObject` subclass called `ContactData`.
    1. In Xcode, make sure the project navigator is visible.
    2. Right-click the ContactSample group and select New File from the popup menu.
    3. Select the Objective-C Class template for the new class, and click Next.
    4. Name the class `ContactData`, make it a subclass of `NSObject`, and click Next.
    5. Accept the default file location suggested by Xcode and click Create.
4. Add the following `@property` declarations to the `ContactData.h` file:

```
@property int contactId;
@property (nonatomic, strong) NSMutableString* firstName;
@property (nonatomic, strong) NSMutableString* lastName;
@property (nonatomic, strong) NSMutableString* address;
@property (nonatomic, strong) NSMutableString* phone;
```

5. Add the following `@synthesize` statements to the `ContactData.m` file:

```
@synthesize contactId;
@synthesize firstName;
@synthesize lastName;
@synthesize address;
@synthesize phone;
```

6. Add user interface elements to the default scene.
    1. Select the `MainStoryboard.storyboard` file in the project navigator.
    2. Use the Object library to add a `UIButton` instance to the default scene.
    3. Resize/position the button to X=10, Y=11, W=300, H=37.
    4. Double-click the button and set its title to Load Contacts.
    5. Use the assistant editor to create an action method called `onLoadContacts` in the view controller class and connect it to the Touch Up Inside event of the button.

6. Use the Object library to add a Table View to the default scene.
7. Resize/position the table view to X=10, Y=69, W=300, H=371.
8. Use the assistant editor to create an outlet in the view controller class corresponding to the table view object in the scene. Name the outlet `tableOfContacts`.

7. Create the XML parser delegate object.

1. Create a new `NSObject` subclass called `ContactDataXMLParserDelegate` by following steps similar to those outlined in step 3 above.
2. Modify the interface of the `ContactDataXMLParserDelegate` class to implement the `NSXMLParserDelegate` protocol. The modified `@interface` declaration should now resemble:

```
@interface ContactDataXMLParserDelegate : NSObject <NSXMLParserDelegate>
```

3. Add the following `#import` directives to the top of the `ContactDataXMLParserDelegate.h` file, after the `#import <Foundation/Foundation.h>` line:

```
#import "Lesson27ViewController.h"
#import "ContactData.h"
```

4. Add the following `@property` declarations to the `ContactDataXMLParserDelegate.h` file:

```
@property (nonatomic, weak) Lesson27ViewController* viewController;
@property (nonatomic, strong) ContactData* tmpContactInfo;
@property (nonatomic, strong) NSMutableString* currentElementValue;
```

5. Add the following `@synthesize` statements to the `ContactDataXMLParserDelegate.m` file:

```
@synthesize viewController;
@synthesize tmpContactInfo;
@synthesize currentElementValue;
```

6. Implement the `parser:didStartElement:namespaceURI:qualifiedName:attributes:` delegate method in the `ContactDataXMLParserDelegate.m` file as follows:

```
- (void)parser:(NSXMLParser*)parser
           didStartElement:(NSString*)elementName
           namespaceURI:(NSString*)namespaceURI
           qualifiedName:(NSString*)qualifiedName
           attributes:(NSDictionary*)attributeDict
{
    // clear the contents of current_element_value
    if (self.currentElementValue == nil)
        self.currentElementValue = [[NSMutableString alloc]
initWithString:@""];
    else
        [currentElementValue setString:@""];

    // set 'tmpContactInfo' to a new ContactInfo object each time
    // the start of a 'contact' element is been encountered
    if([elementName isEqualToString:@"contact"])
    {
        self.tmpContactInfo = [[ContactData alloc] init];

        // read the attributes of the node here.
        NSString* szID = [attributeDict objectForKey:@"id"];
        if (szID != nil)
            self.tmpContactInfo.contactId = [szID intValue];
    }
}
```

7. Implement the `parser:foundCharacters:` delegate method in the `ContactDataXMLParserDelegate.m` file as follows:

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    [currentElementValue appendString:string];
}
```

8. Implement the `parser:didEndElement:namespaceURI:qualifiedName:` delegate method in the `ContactDataXMLParserDelegate.m` file as follows:

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
   namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
    if([elementName isEqualToString:@"first_name"])
    {
        if (tmpContactInfo.firstName == nil)
            tmpContactInfo.firstName = [[NSMutableString alloc]
                                           initWithCapacity:25];
        [tmpContactInfo.firstName setString:currentElementValue];
    }

    if([elementName isEqualToString:@"last_name"])
    {
        if (tmpContactInfo.lastName == nil)
            tmpContactInfo.lastName = [[NSMutableString alloc]
                                          initWithCapacity:25];
        [tmpContactInfo.lastName setString:currentElementValue];
    }

    if([elementName isEqualToString:@"address"])
    {
        if (tmpContactInfo.address == nil)
            tmpContactInfo.address = [[NSMutableString alloc]
                                         initWithCapacity:25];
        [tmpContactInfo.address setString:currentElementValue];
    }

    if([elementName isEqualToString:@"phone"])
    {
        if (tmpContactInfo.phone == nil)
            tmpContactInfo.phone = [[NSMutableString alloc]
                                       initWithCapacity:25];
        [tmpContactInfo.phone setString:currentElementValue];
    }
```

```
        if([elementName isEqualToString:@"contact"])
        {
            [viewController.listOfContacts addObject:tmpContactInfo];
            tmpContactInfo = nil;
        }
    }
```

8. Load and parse the XML file.
    1. Add the following `@property` declarations to the `Lesson27ViewController.h` file:

    ```
    @property (strong, nonatomic) NSMutableArray* listOfContacts;
    @property (strong, nonatomic) NSXMLParser* xmlParser;
    @property (strong, nonatomic) NSData* xmlData;
    ```

    2. Synthesize the properties in the `Lesson27ViewController.m` file.
    3. Add the following `#import` directive to the top of the `Lesson27ViewController.m` file:

    ```
    #import "ContactDataXMLParserDelegate.h"
    ```

    4. Add the following code at the end of the existing implementation of the `viewDidLoad` method:

    ```
    self.listOfContacts = [[NSMutableArray alloc] initWithCapacity:10];
    ```

    5. Add the following code to the implementation of the `onLoadContacts:` method:

    ```
    // load contacts.xml into NSData instance
    NSBundle* bundle = [NSBundle mainBundle];
    NSString* filePath = [bundle pathForResource:@"contacts" ofType:@"xml"];
    self.xmlData = [NSData dataWithContentsOfFile:filePath];

    // instantiate NSXMLParser
    self.xmlParser = [[NSXMLParser alloc] initWithData:xmlData];

    // set up parser delegate
    ContactDataXMLParserDelegate *parserDelegate =
                              [[ContactDataXMLParserDelegate alloc] init];
    parserDelegate.viewController = self;
    [xmlParser setDelegate:parserDelegate];

    // parse the file.
    [xmlParser parse];
    self.xmlParser = nil;
    self.xmlData = nil;
    ```

9. Display contact information in the table view.
    1. Modify the interface of the view controller class to implement the `UITableViewDataSource` and `UITableViewDelegate` protocols. The modified interface declaration should resemble:

    ```
    @interface Lesson27ViewController : UIViewController <UITableViewDataSource,
                                                           UITableViewDelegate>
    ```

2. Add the following code at the end of the existing implementation of the `viewDidLoad` method:

```
tableOfContacts.delegate = self;
tableOfContacts.dataSource = self;
```

3. Add the following code at the end of the existing implementation of the `onLoadContacts:` method:

```
// reload table view
    [tableOfContacts reloadData];
```

4. Implement `UITableViewDataSource` and `UITableViewDelegate` methods in the `Lesson27ViewController.m` file as follows:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    return 1;
}
- (NSInteger)tableView:(UITableView *)aTableView
 numberOfRowsInSection:(NSInteger)section
{
    return [listOfContacts count];
}
- (UITableViewCell *)tableView:(UITableView *)aTableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [aTableView
                  dequeueReusableCellWithIdentifier:@"Cell"];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:@"Cell"];
    }

    ContactData* data = (ContactData*)[listOfContacts
                             objectAtIndex:indexPath.row];

    [[cell textLabel]
     setText:[NSString stringWithFormat:@"%@ %@",
              data.firstName, data.lastName]];

    return  cell;
}
```

10. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

Please select Lesson 27 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo*, to view the video that accompanies this lesson.*

28 Consuming SOAP Web Services

In the previous lesson you learned how to parse an XML file. In this lesson you learn to communicate with web services by using XML-based messages and responses.

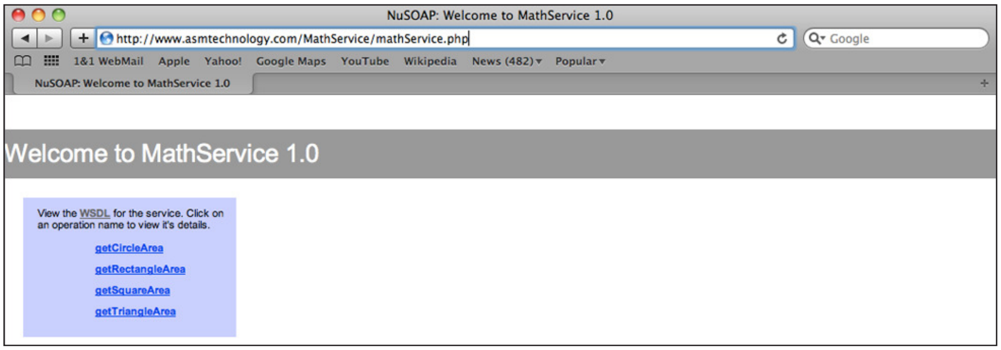
A web service is essentially a web application that runs on a web server and provides a list of methods to its users. You access the web service as you would any other website, using a URL.

You need to know the URL at which a web service is located beforehand, and more often than not, you also need to know the list of methods provided by the web service. Some web services include an additional file on the web server that provides a list of methods contained in the web service. This file is known as the WSDL file. WSDL is an acronym for Web Service Description Language.

Web services themselves can be written using one of several technologies including PHP, ASP.NET, and ColdFusion. Creating a web service is outside the scope of this book.

The examples in this lesson use a simple web service called `MathService`. This is a PHP-based web service and can be accessed at: `www.asmtechnology.com/MathService/mathservice.php`.

If you point your browser to that URL, you will see the WSDL file for the service, which contains a list of method names exposed by this web service (Figure 28-1).


NuSOAP: Welcome to MathService 1.0
http://www.asmtechnology.com/MathService/mathService.php
Google
1&1 WebMail Apple Yahoo! Google Maps YouTube Wikipedia News (482) Popular
NuSOAP: Welcome to MathService 1.0

Welcome to MathService 1.0

View the WSDL for the service. Click on an operation name to view it's details.

getCircleArea
getRectangleArea
getSquareArea
getTriangleArea

FIGURE 28-1

Table 28-1 lists the methods provided by the web service and a brief description of each.

TABLE 28-1: MathService Methods

METHOD NAME	DESCRIPTION
`getCircleArea`	Input: *radius* Output: Returns the area of a circle with specified radius.
`getRectangleArea`	Input: *length*, *breadth* Output: Returns the area of a rectangle with specified length and breadth.
`getSquareArea`	Input: *length* Output: Returns the area of a square whose sides are of specified length.
`getTriangleArea`	Input: *base*, *height* Output: Returns the area of a triangle with specified base length and height.

Broadly speaking, you have three ways to communicate with web services. You can use SOAP, HTTP GET, or HTTP POST. The precise method you use will depend on the web service in question. Most web services made using ASP.NET respond to SOAP.

SOAP is an acronym for Simple Object Access Protocol. It is an XML-based message format, which allows different applications to exchange objects with each other. For this to work, though, the systems need to know beforehand what these objects will be. A full discussion of SOAP is outside the context of this lesson.

On some platforms, developers have access to frameworks that are designed to convert objects to SOAP messages and back. Sadly as iOS developers, we do not have such a framework as of yet.

In the absence of a framework to create SOAP messages for you, the only option available is to make the SOAP message yourself, send it to the web service, and process the SOAP response.

Keep in mind that fundamentally a SOAP message is just some XML text, and all the rules of parsing XML that you learned in Lesson 27 still apply.

The SOAP message to access the `getCircleArea` method of the web service is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <ns3916:getCircleArea xmlns:ns3916="http://tempuri.org">
            <radius xsi:type="xsd:string">20.00</radius>
        </ns3916:getCircleArea>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If this is the first time you are looking at a SOAP message, you might feel intimated. If you apply the fundamental rules of XML, you can make out the root element of this XML document to be `<SOAP-ENV:Envelope>`. The `SOAP-ENV:` prefix specifies the namespace; the actual element name is `Envelope`.

XML Namespaces are not covered in this book. For now, all you need to know is that they provide a mechanism to prevent conflicts between different systems that use an element name to mean two different things.

The `Envelope` element has several attributes, and contains one child element: `<SOAP-ENV:Body>`.

Every SOAP message consists of an `Envelope` element that contains a `Body` element. The web service method that you intend to use (`getCircleArea`), along with any parameters that it may require (`radius`), forms the content of the `Body` element.

In this particular SOAP message, the `Body` element contains the following:

```
<ns3916:getCircleArea xmlns:ns3916="http://tempuri.org">
    <radius xsi:type="xsd:string">20.00</radius>
</ns3916:getCircleArea>
```

You can see that the content of the `Body` element is another element called `getCircleArea`, which happens to be the method you want to use from the web service. The parameter radius is specified as a child of the `getCircleArea` element.

You can use the `NSString` class's `stringWithFormat` convenience method to create an `NSString` object that contains the SOAP message. Assuming `the_radius` is an `NSString` instance that contains the value of the radius parameter for the web method, this can be done as follows:

```
NSString* soap_message =
[NSString stringWithFormat:@"%@%@%@%@%@%@%@%@%@%@",
 @"<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>",
 @"<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\"
    xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"
    xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
    xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
    xmlns:SOAP-ENC=\"http://schemas.xmlsoap.org/soap/encoding/\">",
 @"<SOAP-ENV:Body>",
 @"<ns3916:getCircleArea xmlns:ns3916=\"http://tempuri.org\">",
 @"<radius xsi:type=\"xsd:string\">",
 the_radius,
 @"</radius>",
 @"</ns3916:getCircleArea>",
 @"</SOAP-ENV:Body>",
 @"</SOAP-ENV:Envelope>"];
```

To send the SOAP message to the web server you will have to create a suitable `HTTP-POST` request. The `NSMutableURLRequest` class is used to create an instance of an HTTP request:

```
NSURL *url = [NSURL
URLWithString:@"http://www.asmtechnology.com/MathService/mathService.php"];
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setHTTPMethod:@"POST"];
```

An `HTTP-POST` request contains a few headers and a payload (Figure 28-2). The headers typically identify the format and size of the payload.

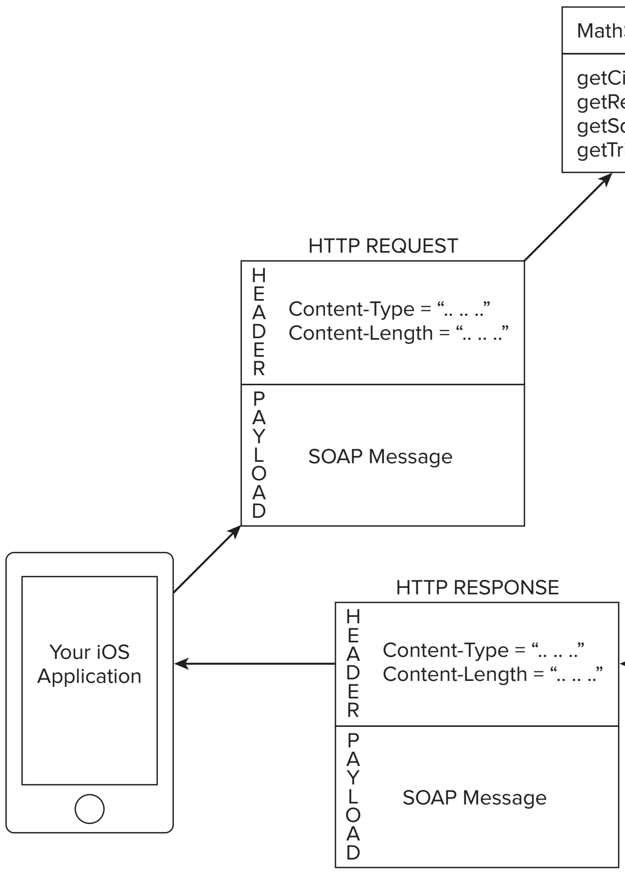


FIGURE 28-2

Table 28-2 lists the most common HTTP headers you will need to set up in order to put the SOAP message as part of the payload.

TABLE 28-2: Common HTTP Headers

HTTP MESSAGE HEADER	DESCRIPTION
`Content-Length`	The length in bytes of the payload.
`Content-Type`	A string that identifies the format of the payload. When the payload contains a SOAP message, you need to set this to be: `text/xml; charset=ISO-8859-1`

The following code can be used to set both the `Content-Type` and the `Content-Length` headers of the `HTTP-POST` request:

```
NSString *message_length =
[NSString stringWithFormat:@"%d", [soap_message length]];

[req addValue:@"text/xml; charset=ISO-8859-1"
     forHTTPHeaderField:@"Content-Type"];

[req addValue:message_length
     forHTTPHeaderField:@"Content-Length"];
```

You can set up the payload of the message as follows:

```
[req setHTTPBody:[soap_message dataUsingEncoding:NSUTF8StringEncoding]];
```

At this point, your `NSMutableURLRequest` object is all ready to be sent to the web service over an Internet connection. To connect to a web service and send the request, you need to use an `NSURLConnection` object.

An `NSURLConnection` instance manages all the low-level intricacies of connecting to a web server across the Internet. It performs this communication asynchronously, allowing your application to do something else in the meantime. The `NSURLConnection` instance also requires a delegate object that conforms to the `NSURLConnectionDelegate` protocol. Methods on this delegate object will be called to signal different events in the life cycle of a connection; for instance, when new data has been received or when a connection error has occurred.

To create an `NSURLConnection` object and send the request with your SOAP message to the web server, use the following code:

```
connection = [[NSURLConnection alloc] initWithRequest:req delegate:self];
```

This code assumes that `connection` is a variable of type `NSURLConnection*` and is defined in the class that contain this code.

The most commonly implemented `NSURLConnectionDelegate` methods are described in Table 28-3.

TABLE 28-3: Commonly used NSURLConnectionDelegate Methods

METHOD NAME	DESCRIPTION
`- connection:didReceiveResponse:`	Called before the `connection:didReceiveData:` event, and contains the HTTP response code.
`- connection:didReceiveData:`	Called multiple times as small quantities of data are received. You should concatenate each data object to build up the entire data for the URL load.
`- connection:didFailWithError:`	Called if the connection failed; error information is provided.
`- connectionDidFinishLoading:`	Called when the connection has finished loading successfully. You should use the accumulated data received over multiple calls to the `connection:didReceiveData:` method at this point.

A typical implementation of these delegate methods is provided here:

```
-(void) connection:(NSURLConnection *) connection
        didReceiveResponse:(NSURLResponse *) response
{
    if (receivedData == nil)
        receivedData = [NSMutableData data];
}

-(void) connection:(NSURLConnection *) connection didReceiveData:(NSData *) data
{
    [receivedData appendData:data];
}
-(void) connection:(NSURLConnection *) connection didFailWithError:(NSError *) error
{
    NSLog(@"%@", @"Unable to connect to web service!");
}

-(void) connectionDidFinishLoading:(NSURLConnection *) connection
{
    NSString* theResponse = [[NSString alloc]
                             initWithBytes:[received_data mutableBytes]
                             length:[received_data length]
                             encoding:NSUTF8StringEncoding];
}
```

The preceding implementation assumes that you have declared a variable named `receivedData` in your class, of type `NSMutableData`.

The sample implementation of the `connectionDidFinishLoading` delegate method converts the response that has been accumulated in the `NSMutableData` variable `receivedData` into an `NSString`. You could at this stage display the response in the Debug console with an `NSLog` statement.

This response is a SOAP message as shown here:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <ns1:getCircleAreaResponse xmlns:ns1="http://tempuri.org">
            <return xsi:type="xsd:string">1519.76</return>
        </ns1:getCircleAreaResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

To extract the result from this XML response, you will need to parse it using an `NSXMLParser` object. The result of the web method is contained within the `result` element, and is shown in boldface. Parsing XML files was covered in Lesson 27.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `SoapClient` to call a SOAP method on a web service.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template.
- Add a `UIButton` to the default scene and an appropriate action method to the view controller class.
- Add a `UITextField` to the default scene and an appropriate outlet to the view controller class.
- Add a scrolling `UITextView` to the default scene and an appropriate outlet to the view controller class.
- Dismiss the text field when the Return button is pressed on the keyboard by implementing a `UITextFieldDelegate` method.
- Send a SOAP request a web service when the `UIButton` is pressed.
- Display the SOAP response in the `UITextView`.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 28 folder in the download.*

Hints

- The `MathService` is located at `http://www.asmtechnology.com/MathService/mathService.php`.
- The `MathService` implements a web method called `getCircleArea` that requires a single parameter called `radius`.
- When composing SOAP messages, keep in mind that method names and parameter names are case sensitive.

Step-by-Step

1. Create a Single View Application in Xcode called `SoapClient`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.

4. Use the following information in the project options dialog box and click Next.
    - **Product Name:** SoapClient
    - **Company Identifier:** com.wileybook
    - **Class Prefix:** Lesson28
    - **Define Family:** iPhone
    - **Use Storyboard:** Checked
    - **Use Automatic Reference Counting:** Checked
    - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Add a `UITextField` instance to the default scene.
    1. Use the Object library to add a Text Field. Resize and reposition it at X = 20, Y = 20, W = 280, H = 31.
    2. Use the Attributes inspector to set the placeholder text to `Enter Radius`.
    3. Set the delegate property of the text field to the view controller object.
    4. Using the assistant editor, create an outlet called `radiusField` and connect it to the text field.
3. Add a button to the default scene.
    1. Add a `UIButton` instance to the default scene.
    2. Name the button `Compute area of circle` and resize/reposition it at X=66, Y=69, W=188, H=37.
    3. Using the assistant editor, create an action method called `onButtonPressed` in the view controller class and connect it to the Touch Up Inside event of the button.
4. Add a `UITextView` instance to the default scene
    1. Using the Object library, add a Text View. Resize and reposition it at X=14, Y=122, W=293, H=318.
    2. Using the Attribute inspector, uncheck the Editable checkbox.
    3. Clear the initial contents of the text view deleting the contents of the `Text` attribute.

4. Using the assistant editor, add an outlet to the view controller class called `resultView` and connect it to the text field. Your view controller should resemble Figure 28-3.

5. Dismiss the text field when the Return button is pressed on the keyboard.

   1. Ensure the `Lesson28ViewController.h` implements the `UITextFieldDelegate` protocol.

   2. Add the following code to the implementation of the view controller class:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return NO;
}
```

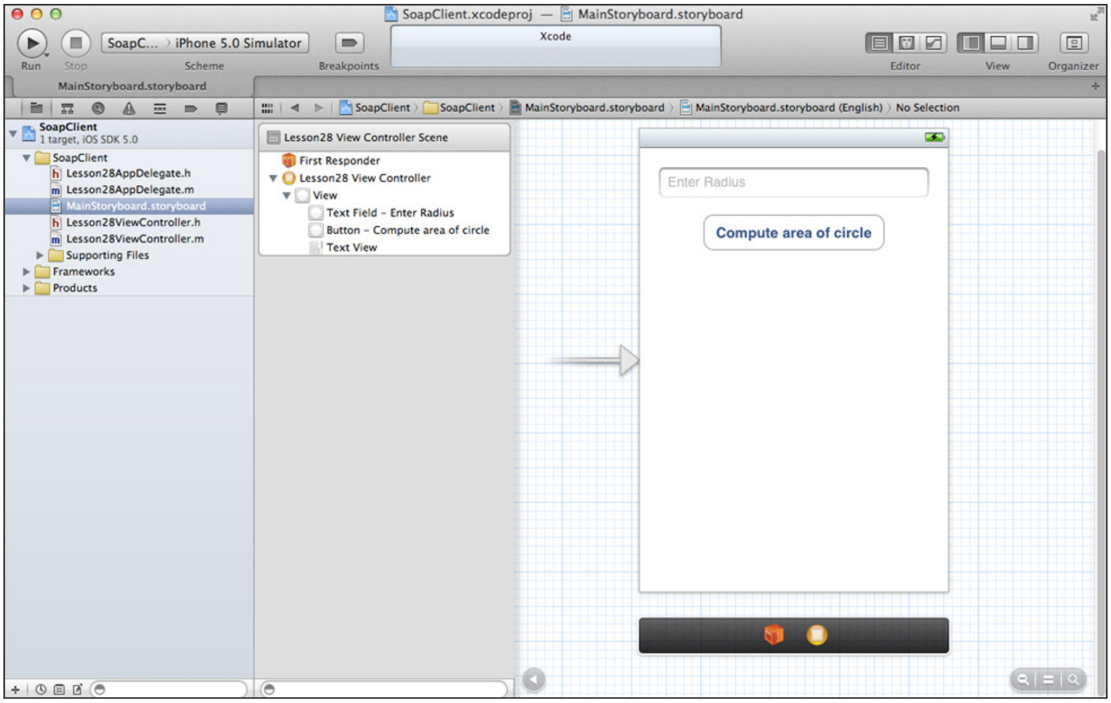

SoapClient.xcodeproj — MainStoryboard.storyboard
Run
Stop
SoapC... iPhone 5.0 Simulator
Scheme
Breakpoints
Xcode
Editor
View
Organizer
MainStoryboard.storyboard
SoapClient
1 target, iOS SDK 5.0
SoapClient
Lesson28AppDelegate.h
Lesson28AppDelegate.m
MainStoryboard.storyboard
Lesson28ViewController.h
Lesson28ViewController.m
Supporting Files
Frameworks
Products
SoapClient > SoapClient > MainStoryboard.storyboard > MainStoryboard.storyboard (English) > No Selection
Lesson28 View Controller Scene
First Responder
Lesson28 View Controller
View
Text Field - Enter Radius
Button - Compute area of circle
Text View
Enter Radius
Compute area of circle

FIGURE 28-3

6. Send a SOAP request to a web service to compute the area of a circle when the button is pressed.

   1. Add the following property declarations to the `Lesson28ViewController.h` file:

```
__strong NSURLConnection* connection;
__strong NSMutableData* receivedData;
```

2. Have the view controller conform to the `NSURLConnectionDataDelegate` protocol. The code in the `Lesson28ViewController.h` file should now resemble:

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

@interface Lesson28ViewController : UIViewController
                <NSURLConnectionDataDelegate, UITextFieldDelegate>

@property (weak, nonatomic) IBOutlet UITextField *radiusField;
@property (weak, nonatomic) IBOutlet UITextView *resultView;

@property (nonatomic, strong) NSURLConnection* connection;
@property (nonatomic, strong) NSMutableData* receivedData;
- (IBAction)onButtonPressed:(id)sender;

@end
```

3. Add the following `@synthesize` statements to the `Lesson28ViewController.m` file:

```
@synthesize connection;
@synthesize receivedData;
```

4. Add the following code to the `onButtonPressed:` method in your view controller class to dismiss the keypad if it is visible:

```
// hide keypad
if ([radiusField isFirstResponder])
    [radiusField resignFirstResponder];
```

5. Before sending a request to the web service, you need to ensure that the value of the radius parameter is valid. Add the following code to the `onButtonPressed:` method, after the code from the previous step:

```
// ensure valid radius is specified
NSString* theRadius = radiusField.text;
if ((theRadius == nil) || ([theRadius length] == 0))
{
    UIAlertView* errorMessage = [[UIAlertView alloc] initWithTitle:nil
                                    message:@"Radius not specified"
                                    delegate:nil
                                    cancelButtonTitle:@"Ok"
                                    otherButtonTitles:nil];
    [errorMessage show];

    return;
}
```

6. The next step involves creating a SOAP message. Add the following code to the `onButtonPressed:` method, after the code from the previous step, to create a SOAP message that will call the `getCircleArea` web service method. This method requires a single parameter called `radius`. Use the value entered by the user in the text field.

```
NSString* soapMessage =
[NSString stringWithFormat:@"%@%@%@%@%@%@%@%@%@%@",
 @"<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>",
 @"<SOAP-ENV:Envelope
```

```
   SOAP-ENV:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\"
   xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"
   xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
   xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
   xmlns:SOAP-ENC=\"http://schemas.xmlsoap.org/soap/encoding/\">",
 @"<SOAP-ENV:Body>",
 @"<ns3916:getCircleArea xmlns:ns3916=\"http://tempuri.org\">",
 @"<radius xsi:type=\"xsd:string\">",
 theRadius,
 @"</radius>",
 @"</ns3916:getCircleArea>",
 @"</SOAP-ENV:Body>",
 @"</SOAP-ENV:Envelope>"];
```

7. The SOAP message generated by this code is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <ns3916:getCircleArea xmlns:ns3916="http://tempuri.org">
        <radius xsi:type="xsd:string">22.0</radius>
        </ns3916:getCircleArea>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

8. Prepare an `HTTP-POST` request for the web service, setting the `Content-Type`, `SoapAction`, and `Content-Length` attributes by adding the following code to the `onButtonPressed:` method, after the code for the previous step:

```
NSURL *url = [NSURL URLWithString:
       @"http://www.asmtechnology.com/MathService/mathService.php"];

NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];

// HTTP headers
NSString *messageLength = [NSString stringWithFormat:@"%d",
                          [soapMessage length]];
[req addValue:@"text/xml; charset=ISO-8859-1"
     forHTTPHeaderField:@"Content-Type"];
[req addValue:@"" forHTTPHeaderField:@"SOAPAction"];
[req addValue:messageLength forHTTPHeaderField:@"Content-Length"];

// method = POST
[req setHTTPMethod:@"POST"];
```

9. Set the SOAP message to be the body of the HTTP request by adding the following code to the `onButtonPressed:` method, after the code from the previous step:

```
// BODY
[req setHTTPBody:[soapMessage dataUsingEncoding:NSUTF8StringEncoding]];
```

10. Finally, send the request to the server over an Internet connection by adding the following code to the end of the `onButtonPressed`: method, after the code from the previous step:

```
// send request
self.connection = [[NSURLConnection alloc] initWithRequest:req
delegate:self];
if (self.connection != nil)
{
    self.receivedData = [NSMutableData data];
}
else
{
    [[UIApplication sharedApplication]
     setNetworkActivityIndicatorVisible:NO];
}
```

7. Implement `NSURLConnectionDataDelegate` methods in your view controller class.

    1. Implement the `connection:didReceiveResponse:` method as follows:

```
-(void) connection:(NSURLConnection *) connection
        didReceiveResponse:(NSURLResponse *) response
{
    [self.receivedData setLength:0];
}
```

    2. Implement the `connection:didReceiveData`: method as follows:

```
- (void)connection:(NSURLConnection *)connection
        didReceiveData:(NSData *)data
{
    [self.receivedData appendData:data];
}
```

    3. Implement the `connection:didFailWithError:` method as follows:

```
-(void) connection:(NSURLConnection *) connection
didFailWithError:(NSError *) error
{
    [[UIApplication sharedApplication] setNetworkActivityIndicatorVisible:NO];

    UIAlertView* errorMessage = [[UIAlertView alloc] initWithTitle:nil
                                 message:@"Unable to connect to web service!"
                                 delegate:nil
                                 cancelButtonTitle:@"Ok"
                                 otherButtonTitles:nil];
    [errorMessage show];
}
```

    4. Implement the `connectionDidFinishLoading:` method as follows:

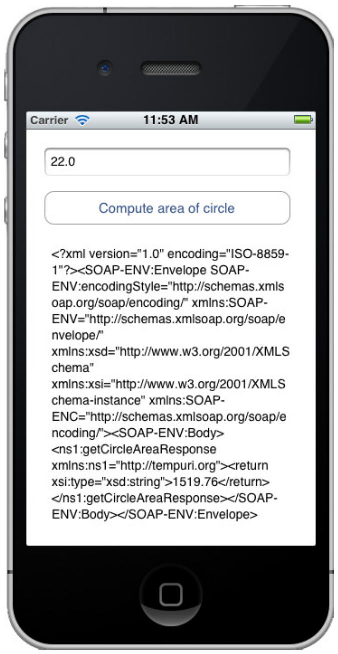
```
-(void) connectionDidFinishLoading:(NSURLConnection *) connection
{
    const void* receivedBytes = [self.receivedData mutableBytes];
    int dataLength = [self.receivedData length];
    NSString* theResponse = [[NSString alloc] initWithBytes:receivedBytes
```

```
                                              length:dataLength
encoding:NSUTF8StringEncoding];

    resultView.text = theResponse;

    NSLog(@"%@", @"Response:");
    NSLog(@"%@", theResponse);

    [[UIApplication sharedApplication]
    setNetworkActivityIndicatorVisible:NO];
}
```

8. Test your app in the iOS Simulator.

   1. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.

   2. Enter a radius between 1 and 100 and press the Compute Area of Circle button. You should get a result similar to Figure 28-4.



```
<?xml version="1.0" encoding="ISO-8859-
1"?><SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmls
oap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/e
nvelope/"
xmlns:xsd="http://www.w3.org/2001/XMLS
chema"
xmlns:xsi="http://www.w3.org/2001/XMLS
chema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/e
ncoding/"><SOAP-ENV:Body>
<ns1:getCircleAreaResponse
xmlns:ns1="http://tempuri.org"><return
xsi:type="xsd:string">1519.76</return>
</ns1:getCircleAreaResponse></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```

FIGURE 28-4

   3. Examine the contents of the Debug console to examine the SOAP message and SOAP response. The SOAP response is shown in the following code. The area of the circle is contained in the `result` element and is shown in boldface:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <ns1:getCircleAreaResponse xmlns:ns1="http://tempuri.org">
            <return xsi:type="xsd:string">1519.76</return>
        </ns1:getCircleAreaResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Please select Lesson 28 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

29 Touches and Gestures

iOS devices differ from traditional non-mobile devices. Interaction is received from a touch or gesture, rather from a keyboard or mouse. The touch or gesture is a `UIEvent` and the `UIApplication` class manages these events. The events that are most common on iOS devices are touch events interacting with views.

A touch sequence begins as a finger or fingers are placed on the touch screen, and ends when the last finger is removed from the touch screen.

Single-fingered events can be a tap, touch, and hold, or the drag and swipe. Multiple-fingered events, for example, can be a pinch, commonly used to zoom in or out on a photo.

This lesson looks at the two techniques of touch event handling:

- Use of the phase methods: `touchesBegin`, `touchesMoved`, `touchesCancelled`, and `touchesEnded`.
- Use of the `UIGestureRecognizer` class.

TOUCH EVENTS

A touch event is a `UIEvent` of the type `UIEventTypeTouches`. The touch itself is a `UITouch` object that contains the following information:

- `locationInView`: The touch coordinates
- `previousLocationInView`: Previous coordinates
- `tapCount`: Current tap count
- `timestamp`: Time of the last touch
- `phase`: The current touch phase

When a touch event occurs, it is placed on a queue that is distributed by the application to the window where the event was initiated. The event is then forwarded to a first responder. In most cases, the first responder is the view where the touch occurred.

If that view cannot handle the touch event, the event is then forwarded to the next responder in the chain—a view controller, for example.

For a view to enter the responder chain, it must perform the following:

```
[self becomeFirstResponder];
```

In addition, the following method must be added in the view controller:

```
- (BOOL)canBecomeFirstResponder { return YES; };
```

Touch Phases

Four phases make up touch events:

- Touching of the first finger:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
```

- Touch and holding of one or more fingers:

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
```

- Removing of one or more fingers:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
```

- Touch event being cancelled by system event, like a phone call:

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

Tap Counting

To obtain the tap count, retrieve the `tapCount` property of the `UITouch` class:

```
UITouch *aTouch = [touches anyObject];
int count = [aTouch tapCount];
```

GESTURE EVENTS

Since the introduction of the `UIGestureRecognizer` class in iOS SDK 3.2, gesture recognition was simplified. Six gesture recognizers were introduced:

- `UITapGestureRecognizer`: For taps
- `UIPinchGestureRecognizer`: For in, out pinching
- `UIPanGestureRecognizer`: For dragging
- `UISwipeGestureRecognizer`: For swiping
- `UIRotationGestureRecognizer`: For rotating finger in opposite direction
- `UILongPressGestureRecognizer`: For touch and hold

Gesture Handling

The process of gesture handling begins with the creation of an instance of the gesture recognizer you need, and assigning a method to handle the event as follows:

```
UITapGestureRecognizer *recognizer  = [[UITapGestureRecognizer alloc]
                        initWithTarget:self
                                action:@selector(handleTapEvent:)];
[view addGestureRecognizer:recognizer];
[recognizer setDelegate:self];
```

The `handleTapEvent` method receives the `UITapGestureRecognizer` and processes the event by centering an image over the location of the tap, as shown here:

```
- (void)handleTapEvent:(UITapGestureRecognizer *)recognizer {
    CGPoint location = [recognizer locationInView:self];

    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:0.05];
    [imageView setCenter:location];
    [UIView commitAnimations];
}
```

Gesture Recognizer Phases

Seven phases make up gesture recognizers:

- `UIGestureRecognizerStatePossible`: Has not yet recognized its gesture
- `UIGestureRecognizerStateBegan`: Has received touch objects that have been recognized as a gesture
- `UIGestureRecognizerStateChanged`: Has received touch objects that have been recognized as a change
- `UIGestureRecognizerStateEnded`: Has received touch objects that have been recognized as the end
- `UIGestureRecognizerStateCancelled`: Has received touch objects that have been recognized as a cancellation of a gesture
- `UIGestureRecognizerStateFailed`: Has received multi-touch sequence that it cannot recognize as a gesture
- `UIGestureRecognizerStateRecognized` = `UIGestureRecognizerStateEnded`: Has received multi-touch sequence that it does recognize as a gesture

TRY IT

In this Try It, you implement a Single View Application that captures gestures and displays the gesture type it has recognized.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson29 folder in the download.*

Lesson Requirements

- Create an Xcode project using the Single View Application template.
- Create a storyboard including just a root view controller.
- Create a series of gesture recognizers.
- Recognize and identify the gestures that were created.

Hints

- Because this application uses storyboards instead of xib files, remember to have the Use Storyboard option checked at project creation.
- You will create not only the gestures, but also the handlers.
- You create four swipe gesture recognizers, one for each direction.

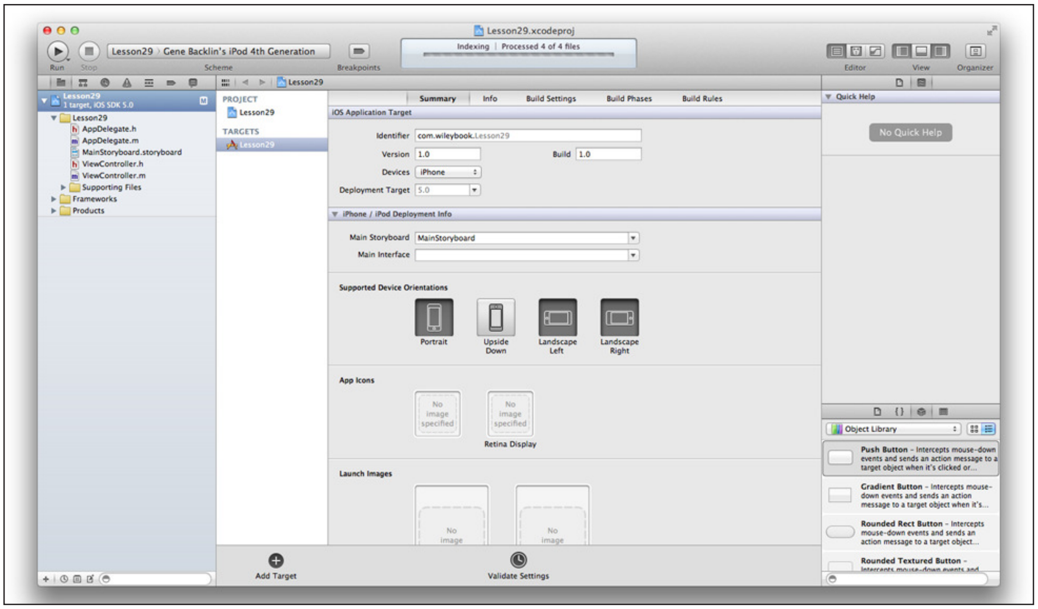
Step-by-Step

1. Create a Single View Application.
    1. Launch Xcode.
    2. Create your new iOS project.
        a. To create a new project, select Create a New Xcode Project.
        b. On the left under iOS, select Application.
        c. Select Single View Application from the template list and click Next.
        d. Choose the following options for your project:
            - **Product Name:** Lesson29
            - **Company Identifier:** com.wileybook
            - **Class Prefix:** leave blank
            - **Device Family:** iPhone
            - **Use Storyboard:** Checked
            - **Use Automatic Reference Counting:** Checked
            - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

e. Select the location on your computer where the project will be saved and select Create.

f. Your Xcode project has been created as shown in Figure 29-1.


Lesson29.xcodeproj

Lesson29 › Gene Backlin's iPod 4th Generation

Indexing | Processed 4 of 4 files

Lesson29
1 target, iOS SDK 5.0
Lesson29
AppDelegate.h
AppDelegate.m
MainStoryboard.storyboard
ViewController.h
ViewController.m
Supporting Files
Frameworks
Products

PROJECT
Lesson29
TARGETS
Lesson29

Summary | Info | Build Settings | Build Phases | Build Rules

iOS Application Target

Identifier: com.wileybook.Lesson29
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0

iPhone / iPod Deployment Info

Main Storyboard: MainStoryboard
Main Interface:

Supported Device Orientations

Run
Stop
Scheme
Breakpoints
Editor
View
Organizer
Quick Help
No Quick Help
Portrait
Upside Down
Landscape Left
Landscape Right
App Icons
No image specified
Retina Display
Launch Images
No image
Add Target
Validate Settings
Object Library
Push Button - Intercepts mouse-down events and sends an action message to a target object when it's clicked or...
Gradient Button - Intercepts mouse-down events and sends an action message to a target object when it's...
Rounded Rect Button - Intercepts mouse-down events and sends an action message to a target object...
Rounded Textured Button -

FIGURE 29-1

2. Design the user interface.
   1. On the left, select `MainStoryboard.storyboard`.
   2. On the right, select the third button in the View section, to display the Utilities view.
   3. Drag a label from the object library and place it near the top of the view and resize it to be as wide as the view.
   4. Select two more labels and resize and place them each under the other, so you have three labels arranged in a row.
   5. Select a view and drag it into the view controller in the middle.
   6. From the attributes inspector set the background to black.
   7. From the size inspector set the width and height to 100 and center it in the view as shown in Figure 29-2.

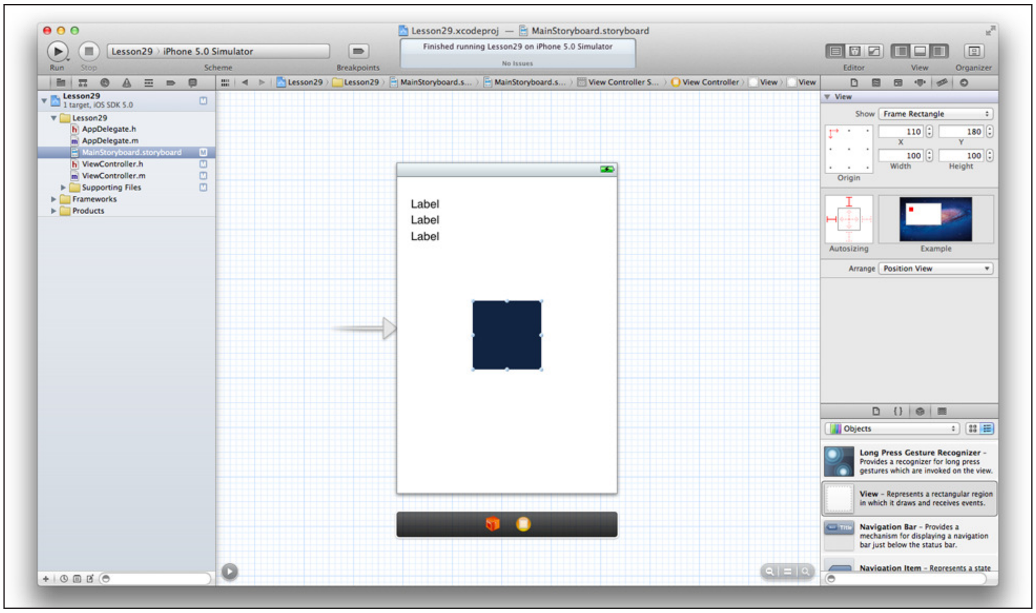

Lesson29.xcodeproj — MainStoryboard.storyboard
Lesson29 › iPhone 5.0 Simulator
Finished running Lesson29 on iPhone 5.0 Simulator
Label
Label
Label
Frame Rectangle
Position View
Objects

FIGURE 29-2

3. To create the outlets for the view controller:
    1. Select the attributes inspector to bring up the `ViewController.h` file.
    2. Select the first label and control+drag to the interface source code just above the `@end`.
    3. Enter **phaseLabel** for the outlet name and click Connect.
    4. Select the second label and control+drag to the interface source code just above the `@end`.
    5. Enter **tapCountLabel** for the outlet name and click Connect.
    6. Select the third label and control+drag to the interface source code just above the `@end`.
    7. Enter **touchCountLabel** for the outlet name and click Connect.
    8. Select the 100 × 100 view and control+drag to the interface source code just above the `@end`.
    9. Enter **touchView** for the outlet name and click Connect.
    10. Select the Standard editor to hide the detail view controller.
4. In addition to adding `<UIGestureRecognizerDelegate>` to the interface class declaration, add the following to the `ViewController.h` class before the `@end` statement:

```
#pragma mark - Create Gesture Recognizers

- (void)createGestureRecognizers;
- (void)createSingleTapRecognizer;
- (void)createDoubleTapRecognizer;
- (void)createPinchRecognizer;
- (void)createSwipeRecognizers;
- (void)createSwipeRecognizer:(UISwipeGestureRecognizerDirection)direction;

#pragma mark - Event Handlers

- (void)handleSingleTapEvent:(UITapGestureRecognizer *)recognizer;
- (void)handleDoubleTapEvent:(UITapGestureRecognizer *)recognizer;
- (void)handlePinchEvent:(UIPinchGestureRecognizer *)recognizer;
- (void)handleSwipeEvent:(UISwipeGestureRecognizer *)recognizer;

#pragma mark - Utility methods

- (void)updateDisplayWithPhase:(NSString *)phase
                      tapCount:(int)tapCount
                    touchCount:(int)touchCount;
- (void)moveImage:(UIGestureRecognizer *)recognizer;
```

5. Make the following updates to the `ViewController.m` class.

    1. In the `viewDidLoad` method add the following below the `[super viewDidLoad];` statement:

    ```
    [self becomeFirstResponder];
    [self updateDisplayWithPhase:@"" tapCount:0 touchCount:0];
    [self createGestureRecognizers];
    ```

    2. Add the following above the `@end`:

    ```
    - (BOOL)canBecomeFirstResponder {
        return YES;
    }
    ```

    3. To create the gesture recognizers, add the following above the `@end`:

    ```
    #pragma mark - Create Gesture Recognizers

    - (void)createGestureRecognizers {
        [self createSingleTapRecognizer];
        [self createDoubleTapRecognizer];
        [self createPinchRecognizer];
        [self createSwipeRecognizers];
    }

    - (void)createSingleTapRecognizer {
        UITapGestureRecognizer *recognizer =
              [[UITapGestureRecognizer alloc]
                   initWithTarget:self
                           action:@selector(handleSingleTapEvent:)];
        [[self view] addGestureRecognizer:recognizer];
        [recognizer setDelegate:self];
    }

    - (void)createDoubleTapRecognizer {
        UITapGestureRecognizer *recognizer =
              [[UITapGestureRecognizer alloc]
                   initWithTarget:self
    ```

```
                       action:@selector(handleDoubleTapEvent:)];
    [[self view] addGestureRecognizer:recognizer];
    [recognizer setDelegate:self];
    [recognizer setNumberOfTapsRequired:2];
}

- (void)createPinchRecognizer {
    UIPinchGestureRecognizer *recognizer =
          [[UIPinchGestureRecognizer alloc]
                 initWithTarget:self
                         action:@selector(handlePinchEvent:)];
    [[self view] addGestureRecognizer:recognizer];
    [recognizer setDelegate:self];
}
- (void)createSwipeRecognizers {
    [self createSwipeRecognizer:UISwipeGestureRecognizerDirectionLeft];
    [self createSwipeRecognizer:UISwipeGestureRecognizerDirectionRight];
    [self createSwipeRecognizer:UISwipeGestureRecognizerDirectionUp];
    [self createSwipeRecognizer:UISwipeGestureRecognizerDirectionDown];
}

- (void)createSwipeRecognizer:
        (UISwipeGestureRecognizerDirection)direction
{
    UISwipeGestureRecognizer *recognizer =
           [[UISwipeGestureRecognizer alloc]
              initWithTarget:self
                     action:@selector(handleSwipeEvent:)];
    [[self view] addGestureRecognizer:recognizer];
    [recognizer setDirection:direction];
}
```

4. To create the event handlers, add the following above the `@end`:

```
#pragma mark - Event Handlers

- (void)handleSingleTapEvent:(UITapGestureRecognizer *)recognizer {
    [self updateDisplayWithPhase:@"UITapGestureRecognizer"
                        tapCount:1
                      touchCount:1];
    [self moveImage:recognizer];
}

- (void)handleDoubleTapEvent:(UITapGestureRecognizer *)recognizer {
    [self updateDisplayWithPhase:@"UITapGestureRecognizer"
                        tapCount:2
                      touchCount:1];
    [self moveImage:recognizer];
}

- (void)handlePinchEvent:(UIPinchGestureRecognizer *)recognizer {
    [self updateDisplayWithPhase:@"UIPinchGestureRecognizer"
                        tapCount:1
                      touchCount:2];

    CGFloat scale = [(UIPinchGestureRecognizer *)recognizer scale];
    if ([recognizer state] == UIGestureRecognizerStateEnded) {
        [UIView beginAnimations:nil context:nil];
        [UIView setAnimationDuration:0.10];
        [[self touchView] setTransform:CGAffineTransformIdentity];
        [UIView commitAnimations];
    } else {
        [UIView beginAnimations:nil context:nil];
        [UIView setAnimationDuration:0.10];
        [[self touchView]
              setTransform:CGAffineTransformMakeScale(scale, scale)];
        [UIView commitAnimations];
    }
}

- (void)handleSwipeEvent:(UISwipeGestureRecognizer *)recognizer {
    NSString *swipeDirection = nil;
    CGPoint location = [recognizer locationInView:[self view]];
    [[self touchView] setCenter:location];

    if([recognizer direction] == UISwipeGestureRecognizerDirectionLeft) {
        swipeDirection = @"UISwipeGestureRecognizerDirectionLeft";
        location.x -= 150.0;
    } else if([recognizer direction] ==
                       UISwipeGestureRecognizerDirectionRight) {
        swipeDirection = @"UISwipeGestureRecognizerDirectionRight";
        location.x += 150.0;
    } else if(([recognizer direction] ==
                       UISwipeGestureRecognizerDirectionUp)) {
        swipeDirection = @"UISwipeGestureRecognizerDirectionUp";
        location.y -= 150.0;
    } else if(([recognizer direction] ==
                       UISwipeGestureRecognizerDirectionDown)) {
        swipeDirection = @"UISwipeGestureRecognizerDirectionDown";
        location.y += 150.0;
    }

    if(swipeDirection != nil) {
        [self updateDisplayWithPhase:swipeDirection
                            tapCount:1
                          touchCount:1];
        [UIView beginAnimations:nil context:nil];
        [UIView setAnimationDuration:0.50];
        [[self touchView] setCenter:location];
        [UIView commitAnimations];
    }
}
```

5. To update the three labels with the current phase, tap, and touch counts, add the following utility method:

```
- (void)updateDisplayWithPhase:(NSString *)phase
                      tapCount:(int)tapCount
                    touchCount:(int)touchCount {
```

```
    [[self phaseLabel] setText:[NSString
              stringWithFormat:@"Touch Phase:%@", phase]];
    [[self tapCountLabel] setText:[NSString
              stringWithFormat:@"Tap Count:%d", tapCount]];
    [[self touchCountLabel] setText:[NSString
              stringWithFormat:@"TouchCount: %d", touchCount]];
}
```

6. To move the image depending on the gesture, add the following utility method:

```
- (void)moveImage:(UIGestureRecognizer *)recognizer {
    CGPoint location = [recognizer locationInView:[self view]];

    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:0.10];
    [[self touchView] setCenter:location];
    [UIView commitAnimations];
}
```

6. Run the application.
    1. Select the iPhone Simulator to run the application.
    2. Click the Run button on the Xcode toolbar.
    3. When the application launches, tap, double tap, and swipe, and observe the display and the 100 × 100 image move about the view.

Please select Lesson 29 on the DVD that accompanies the print book, or go to www.wrox.com/go/iphoneipadappvideo, *to view the video that accompanies this lesson.*

30 Printing

Starting with iOS 4.2, applications can print their contents to local printers. Printing is a valuable feature that can be added to an existing application often with only a few lines of code. In this lesson, you learn to add printing capabilities to your apps.

From an end user's perspective, printing from an iOS device involves tapping a print button, specifying a few print options, and sending a print job to the printer. A print job is a unit of work for the iOS printing subsystem, which includes not just the content to be printed, but also additional information such as the name of the printer, print quality settings, and page orientation.

The print options user interface is provided by UIKit. On an iPad it is a popover view; on an iPhone/iPod Touch it is a sheet that can be animated to slide up from the bottom of the screen (Figure 30-1).

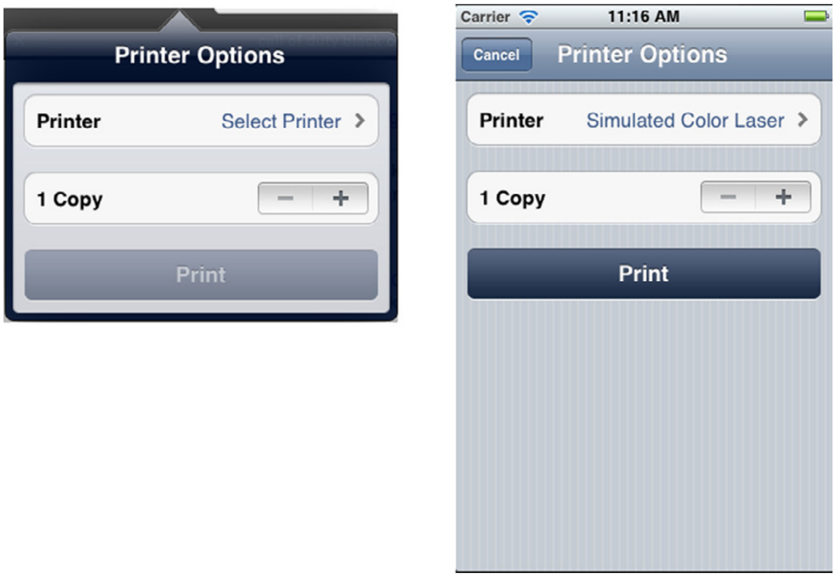
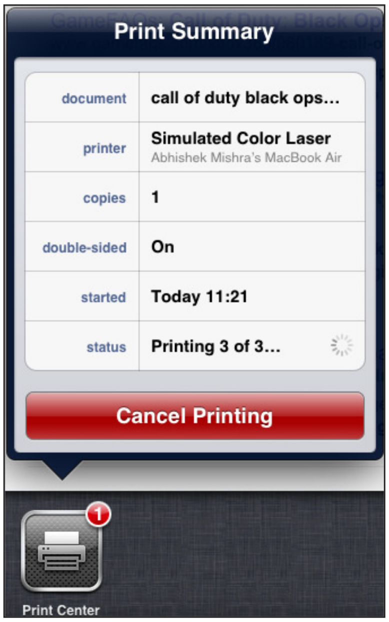


FIGURE 30-1

The print options always include a list of printers that have been discovered, and the number of copies. Sometimes the user interface can include additional options such as the range of pages to print, and duplex printing.

The printing subsystem is shared between all applications; print jobs are sent to the printer in the order in which they are received. After submitting a print job from an application, the user can monitor its status by double-tapping the home button and using the standard Print Center application (Figure 30-2). The Print Center application is available only when one or more print jobs are in progress.



Print Summary

document	call of duty black ops...
printer	Simulated Color Laser Abhishek Mishra's MacBook Air
copies	1
double-sided	On
started	Today 11:21
status	Printing 3 of 3...

FIGURE 30-2

PREPARING CONTENT FOR PRINTING

The UIKit framework contains several key classes related to printing. These are briefly described in this section.

UIPrintInfo

An instance of this class represents attributes of a print job such as the name of the printer, page orientation, output type, and duplex mode. Typically, you create a `UIPrintInfo` object by using the `printInfo` class method and set up relevant properties. The properties are shown in Table 30-1.

TABLE 30-1: UIPrintInfo Properties

PROPERTY	DESCRIPTION
`NSString* jobName`	A string representing the name of the print job. This name will be used to list the job in the Print Center while it is being printed. The default value is the name of the application.
`UIPrintInfoOrientation orientation`	Can be either `UIPrintInfoOrientationPortrait` or `UIPrintInfoOrientationLandscape`. Portrait mode is the default.
`UIPrintInfoOutputType outputType`	Specifies the output type of the print job. The value you specify here determines the default values of some of the other properties of the `UIPrintInfo` instance, and the default paper size. The `output-Type` property can be one of the following: `UIPrintInfoOutputGeneral` `UIPrintInfoOutputPhoto` `UIPrintInfoOutputGrayscale`
`UIPrintInfoDuplex duplex`	Specifies duplex-mode printing options. It can be one of the following values: `UIPrintInfoDuplexNone` `UIPrintInfoDuplexLongEdge` `UIPrintInfoDuplexShortEdge` The default value is based on the output type—it is none for photo and long edge for others.

Besides these properties, the `UIPrintInfo` object contains another property that identifies the printer. This property is filled by UIKit after the user selects a printer. The following code snippet creates a `UIPrintInfo` object suitable for printing a photo:

```
UIPrintInfo *printInfo = [UIPrintInfo printInfo];
printInfo.outputType = UIPrintInfoOutputPhoto;
printInfo.orientation = UIPrintInfoOrientationPortrait;
printInfo.jobName = @"Bird";
```

UIPrintPaper

Instances of this class specify the size of paper to use and the printable area. Most applications use the default object created by UIKit for a print job, but it is possible to provide one of your own. UIKit chooses default paper sizes based on the destination printer and the output job type specified in `UIPrintInfo` object.

UIPrintInteractionController

A shared instance of this class is the central object controlling a print job for your application. It contains information about the print job, the size of the paper, and the content to be printed. The content to be printed can be specified as a single image/PDF document, an array of images/PDF documents, a print formatter, or a page renderer. To access the shared instance, use the `sharedPrintController` class method as follows:

```
UIPrintInteractionController *pic = [UIPrintInteractionController
                                     sharedPrintController];
```

Some of the properties of the `UIPrintInteractionController` class are listed in Table 30-2.

TABLE 30-2: UIPrintInteractionController Properties

PROPERTY	DESCRIPTION
`UIPrintInfo* printInfo`	A reference to a `UIPrintInfo` object.
`Id<UIPrintInteractionControllerDelegate> delegate`	An optional delegate object that implements the `UIPrintInteractionControllerDelegate` protocol.
`BOOL showsPageRange`	Specifies whether the print options user interface should contain a page range control.
`id printingItem`	Refers to a single `NSData`, `NSURL`, or `UIImage` object to print.
`NSArray* printingItems`	Refers to an array of `NSData`, `NSURL`, or `UIImage` objects to print.
`UIPrintFormatter* printFormatter`	An instance of a `UIPrintFormatter` subclass that can be used for complex content that needs pagination.
`UIPrintPageRenderer* printPageRenderer`	An instance of a subclass of `UIPrintPageRender` that can be used for complex content that requires advanced printing features such as headers and footers.

The content to be printed can be assigned to the `printingItem`, `printingItems`, `printFormatter`, or `printPageRenderer` properties. If you are printing simple images and PDF documents you use either the `printingItem` or the `printingItems` property.

If you want to print the contents of a `UIView`, you would typically get a `UIViewPrintFormatter` object from the view and assign it to the `printFormatter` property. If you wanted total control over the content that is printed, including margins, pagination, headers, and footers, you would create an instance of a `UIPrintPageRenderer` subclass and assign it to the `printPageRenderer` property. Using print formatters and page renderers for printing is outside the scope of this book.

A UIPrintInteractionController object can have a reference to an optional delegate object that conforms to the UIPrintInteractionControllerDelegate protocol. The methods in the delegate object are invoked when printing options are presented/dismissed and when the print job begins/ends. If your application requires a special paper size, the delegate object can return an appropriate UIPrintPaper instance.

Before you can print content, you must ensure the iOS device supports printing. It is a good idea to check this once your view loads, and show/hide the print button appropriately. To determine if printing is available, use the isPrintingAvailable class method of the UIPrintInteractionController object.

```
BOOL canPrint = [UIPrintInteractionController isPrintingAvailable];
```

Once you have set up appropriate properties of the UIPrintInteractionController shared instance, you can display the print options to users by using one of three methods:

- presentFromBarButtonItem:animated:completionHandler: This is used to present the print options from a button in a navigation bar or toolbar.
- presentFromRect:inView:animated:completionHandler: This is used to present the print options from an arbitrary rectangle in the application's view.
- presentAnimated:completionHandler: This is used when presenting the print options on an iPhone or iPod Touch. It animates a sheet that slides up from the bottom.

Each of these methods requires a block handler that is called when the job is complete, or errors occur. This handler typically resembles the following:

```
void (^completionHandler)(UIPrintInteractionController *,
                          BOOL, NSError *) =
^(UIPrintInteractionController *pic, BOOL completed, NSError *error)
 {
     if (!completed && error)
     {
         // handle error here
     }
     return;
 };
```

A simple application that uses these concepts to print a UIImage object is covered in this lesson's Try It section.

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called PrintTest that prints a UIImage. To test the printing functionality, you use the iOS Printer Simulator.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image resources into the project.
- Add a UIImageView instance to the scene.

- Add a `UIButton` instance to the scene and an appropriate action method in the view controller class.
- Ensure printing features are available in the `viewDidLoad` method.
- Launch the iOS Printer Simulator.
- Print the image.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 30 folder in the download.*

Hints

- Launch the iOS Printer Simulator from the File ➪ Print Simulator menu of the iOS Simulator.

Step-by-Step

1. Create a Single View Application in Xcode called `PrintTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** PrintTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson30
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.

2. Import the `bird.png` resource from this chapter's resource folder on the DVD.
3. Add a `UIImageView` instance to the scene.
    1. Open the storyboard file and use the Media library to drag and drop `bird.png` onto the default scene. This automatically creates an image view.
    2. Size and position the image view to X=0, Y=–20, W=320, H=480.
4. Add a `UIButton` instance to the scene and create an appropriate outlet and action method in the view controller class.
    1. Drag and drop a Round Rect Button from the Object library onto the scene and position it at X=30, Y=398, W=264, H=37. Change the title of the button to `Print Image`.
    2. Use the assistant editor to create an outlet called `printButton` in the view controller class and connect it to the button.
    3. Use the assistant editor to create an action called `onPrint` in the view controller class and associate it with the Touch Up Inside event of the button.
5. Add the following code to the `viewDidLoad` method of the `Lesson30ViewController` class, after the `[super viewDidLoad]` line.

```
BOOL canPrint = [UIPrintInteractionController isPrintingAvailable];
if (canPrint == NO)
    [printButton setHidden:YES];
```

6. Print the image.
    1. Add the following code to the `onPrint` method of the `Lesson30ViewController` class:

```
UIImage* birdImage = [UIImage imageNamed:@"bird.png"];

UIPrintInteractionController *pic = [UIPrintInteractionController
                                     sharedPrintController];

UIPrintInfo *printInfo = [UIPrintInfo printInfo];
printInfo.outputType = UIPrintInfoOutputPhoto;
printInfo.orientation = UIPrintInfoOrientationPortrait;
printInfo.jobName = @"Bird Image Print";

pic.printInfo = printInfo;
pic.showsPageRange = NO;
pic.printingItem = birdImage;

void (^completionHandler)(UIPrintInteractionController*,
                                         BOOL, NSError *) =
^(UIPrintInteractionController *pic, BOOL completed, NSError *error)
{
    if (!completed && error)
    {
        UIAlertView* errorMessage = [[UIAlertView alloc]
                                     initWithTitle:@"Printing Error"
                                     message:nil
                                     delegate:nil
                                     cancelButtonTitle:@"Ok"
```

```
                                             otherButtonTitles:nil];
            [errorMessage show];
        }
        return;
    };

    [pic presentAnimated:YES completionHandler:completionHandler];
```

7. Test your app in the iOS Simulator.
    1. Launch the Printer Simulator from the File ➪ Print Simulator menu item of the iOS Simulator application on your Mac.
    2. Tap the Print Image button, select appropriate options, and tap the Print button.

Please select Lesson 30 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

31 Basic Animation with Timers

A timer is an object that waits for a time interval and then sends a specific message to a target object. When a timer sends the message, the act is generally referred to as firing, and the timer is said to have fired. You can set up a timer to be repeating or non-repeating. A non-repeating timer fires only once; a repeating timer fires and then reschedules itself to fire again.

In iOS development, timers are represented by instances of the `NSTimer` class. Timers have several uses, but in this lesson you learn to use them to create a simple animation.

Creating a repeating or non-repeating timer is simple and can be done by sending the `scheduleTimerWithTimeInterval:target:selector:userInfo:repeats:` class method to the `NSTimer` class:

```
NSTimer* animationTimer = [NSTimer scheduledTimerWithTimeInterval:0.5
                                    target:self
                                    selector:@selector(onTimerFired:)
                                    userInfo:nil
                                    repeats:YES];
```

The first parameter to this method is the time interval after which the timer should fire, expressed in milliseconds. If the timer is repeating, this is also the interval that will be used to reschedule the timer after it has fired.

The second parameter is the object to which a message should be sent when the timer fires. When used within view controllers, it is common to provide `self` for this value. The third parameter is the name of the message that should be sent to the target object. The object must implement a method that corresponds to the message. The method must be of a specific signature:

```
- (void) methodName:(NSTimer*)timer;
```

It must not return any values, and must accept a single parameter of type `NSTimer*`. The fourth parameter can be an instance of an `NSObject` subclass that contains some information you want to provide to the target object when the timer fires. This is usually set to `nil`.

The final parameter is a Boolean value that specifies whether the timer should be a repeating one. If you have created a repeating timer, you must dispose of the timer when you no longer need it. Disposing of a timer object is also known as invalidating it; once invalidated, a repeating timer will not fire. Non-repeating timers do not need to be invalidated. To invalidate a repeating timer, send the timer object an invalidate message:

```
[animationTimer invalidate];
```

If you have specified an object in the `userInfo` parameter while creating the timer, you can retrieve the object within the method that is fired in the target object as follows:

```
- (void)onTimerFired:(NSTimer*)timer
{
    id someObject = [timer userInfo];
}
```

ANIMATING UIVIEW SUBCLASSES

The size and position of a `UIView` subclass is collectively referred to as its *frame*. The position of a view is expressed by specifying the distance of the top-left corner of the view from that of its parent. If the view has no parent, the distance is measured from the top-left corner of the window (Figure 31-1).

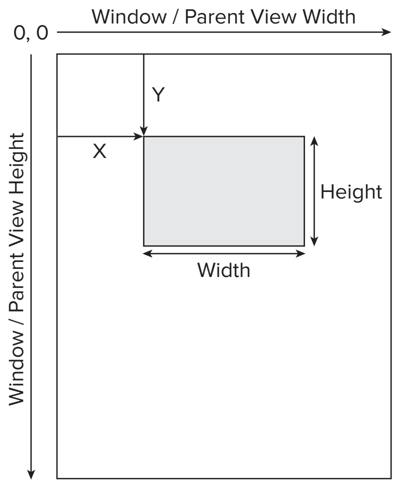


FIGURE 31-1

The size of the view is simply the number of pixels horizontally and vertically. A frame is essentially a rectangle and is represented by the `CGRect` structure. To create one, you specify the `x`, `y`, `width`, and `height` values to the `CGRectMake` function:

```
CGRect rect1 = CGRectMake(10.0, 20.0, 100.0, 100.0);
```

The `CGRect` structure internally contains two members, `origin` and `size`, which themselves are `CGPoint` and `CGSize` structures. A `CGPoint` structure has two float members, `x` and `y`, that represent the coordinate values of the point in question. A `CGSize` structure has two float members, `width` and `height`.

Thus, if `rect1` is a `CGRect` instance, you can access the individual `x`, `y`, `width`, and `height` members as:

```
rect1.origin.x
rect1.origin.y
rect1.size.width
rect1.size.height
```

`CGRect`, `CGPoint`, `CGSize`, and `CGRectMake` are all part of the Core Graphics framework. This framework is automatically included in your iOS applications when you use most of Xcode's iOS application templates.

You can reposition a `UIView` subclass by simply providing a different value for the origin of the view's frame (Figure 31-2). For instance, if `myButton` is an instance of `UIButton` with frame 100, 20, 50, 90, you can reposition it at 300, 100 by changing its frame using the following code:

```
myButton.frame = CGRectMake(300, 100, 50, 90);
```

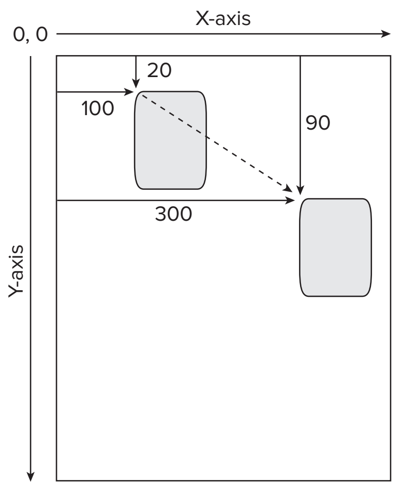


FIGURE 31-2

If you were to repeatedly reposition a UIView subclass (such as an image view) using a repeating timer, you could create simple moving objects on your screen. Animation is a complex subject and in fact Apple provides a framework called Core Animation designed specifically for this purpose. Core Animation is beyond the scope of this book; however, in this lesson's Try It section, you will move an image view across the screen using a repeating timer.

For an introduction to Core Animation, read the "Core Animation Programming Guide," available at: `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html`.

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `Bounce` that uses a timer to move a ball on the screen.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image resources into the project.
- Add a `UIImageView` instance to the scene and an appropriate outlet in the view controller file.
- Create a repeating timer in the `viewDidLoad` method.
- Move the image view across the screen by a small amount each time the timer fires.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 31 folder in the download.*

Hints

- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.
- A repeating timer must be invalidated when you do not need it any longer.

Step-by-Step

1. Create a Single View Application in Xcode called `Bounce` that uses a storyboard.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** Bounce
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson31
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Import the `ball.png` resource from this chapter's resource folder on the DVD into the project.
3. Add a `UIImageView` instance to the scene and connect it to an outlet in the view controller class.
    1. Open the storyboard file and use the Media library to drag and drop `ball.png` onto the default scene. This automatically creates an image view.
    2. Use the assistant editor to create an outlet in the view controller class and connect it to the image view. Name the outlet `ballImage`.
4. Set up a timer in the view controller class.
    1. Add the following `@property` declaration to the `Lesson31ViewController.h` file:

       ```
       @property (strong, nonatomic) NSTimer* animationTimer;
       ```

    2. Declare the method that will be called when the timer is fired in the `Lesson31ViewController.h` file as:

       ```
       -(void) onTimerFired:(NSTimer*)timer;
       ```

       Your Lesson31`ViewController.h` file should now resemble the following:

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

       @interface Lesson31ViewController : UIViewController

       @property (weak, nonatomic) IBOutlet UIImageView *ballImage;
       @property (strong, nonatomic) NSTimer* animationTimer;

       -(void) onTimerFired:(NSTimer*)timer;
       @end
       ```

    3. Add the following `@synthesize` statement to the `Lesson31ViewController.m` file:

       ```
       @synthesize animationTimer;
       ```

    4. Create a repeating timer in the `viewDidLoad` method of the view controller class using the following code:

       ```
       animationTimer = [NSTimer scheduledTimerWithTimeInterval:0.1
                                 target:self
                                 selector:@selector(onTimerFired:)
                                 userInfo:nil
                                 repeats:YES];
       ```

    5. Add the following code at the top of `viewDidUnload` method of the view controller class to invalidate the timer:

    ```
    [animationTimer invalidate];
    ```

5. Move the image view across the screen in the `onTimerFired` method.

    1. Add the following `@property` declarations to the `Lesson31ViewController.h` file:

    ```
    @property int velocityX;
    @property int velocityY;
    ```

    2. Add the following `@synthesize` statements to the `Lesson31ViewController.m` file:

    ```
    @synthesize velocityX;
    @synthesize velocityY;
    ```

    3. Add the following code to the `viewDidLoad` method, before creating the timer:

    ```
    velocityX = 10;
    velocityY = 17;
    ```

    4. Implement the `onTimerFired:` method in the view controller class as follows:

    ```
    - (void) onTimerFired:(NSTimer*)timer
    {
        // current position of ball
        int ballRadius = 34;
        int currentX = ballImage.frame.origin.x + ballRadius;
        int currentY = ballImage.frame.origin.y + ballRadius;

        // new position of ball
        int newX = currentX + velocityX;
        int newY = currentY + velocityY;

        // ensure new position is within the bounds of the screen

        // left
        if (newX < ballRadius)
        {
            newX = ballRadius;
            velocityX = velocityX * -1;
        }

        // top
        if (newY < ballRadius)
        {
            newY = ballRadius;
            velocityY = velocityY * -1;
        }

        // right
        if (newX > 320 - ballRadius)
        {
            newX = 320 - ballRadius;
            velocityX = velocityX * -1;
        }
    ```

```
            // bottom
            if (newY > 460 - ballRadius)
            {
                newY = 460 - ballRadius;
                velocityY = velocityY * -1;
            }

            // put ball in new place.
            ballImage.frame = CGRectMake(newX - ballRadius,
                                         newY - ballRadius,
                                         ballRadius * 2,
                                         ballRadius * 2);
        }
```

6. Test your app in the iOS Simulator by clicking the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.

Please select Lesson 31 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

32

Introduction to Core Image

`Core Image` is a framework for image processing built into iOS 5 that aims to provide near real-time processing of images and video by leveraging programmable graphics hardware capabilities. Prior to `Core Image`, the only way a developer could take advantage of programmable graphics hardware was to create small programs called *shaders* in a language called OpenGL Shading Language (GLSL). Not all devices have programmable graphics hardware; on these older devices (such as the iPhone 3G), `Core Image` falls back to using the main CPU for computations.

The `Core Image` framework is not included in any of the standard iOS application templates. To use this framework in your code you need to add it manually to your project.

IMAGES AND FILTERS

You need to be aware of two key concepts while working with `Core Image`:

- **Images:** An image is represented by an instance of the `CIImage` class and represents the input or output image/video frame that you want to process.
- **Filters:** The code that performs the actual image processing is known as a filter. A filter is represented by an instance of a `CIFilter` object. Though it is possible to create your own `Core Image` filters, this is an advanced topic and beyond the scope of this book. Apple provides more than 100 ready-to-use filters with the `Core Image` framework that you can use in your application.

Apple's filters are grouped into 18 categories, and each category is identified by a unique name. These names are listed in Table 32-1.

TABLE 32-1: Core Image Filter Categories

`kCICategoryBlur`	`kCICategoryColorEffect`	`kCICategoryStylize`
`kCICategoryDistortionEffect`	`kCICategoryTransition`	`kCICategorySharpen`
`kCICategoryGeometryAdjustment`	`kCICategoryTileEffect`	`kCICategoryVideo`
`kCICategoryCompositeOperation`	`kCICategoryGenerator`	`kCICategoryStillImage`
`kCICategoryHalftoneEffect`	`kCICategoryReduction`	`kCICategoryInterlaced`
`kCICategoryColorAdjustment`	`kCICategoryGradient`	`kCICategoryNonSquarePixels`

You can use the `filterNamesInCategory` class method of the `CIFilter` class to get an array that contains the names of filters in that category:

```
NSArray *array = [CIFilter filterNamesInCategory:kCICategoryColorEffect];
```

For example, the preceding code snippet returns an array with the following identifiers, corresponding to filters in the color effect category:

- ➤ `CIColorInvert`
- ➤ `CIFalseColor`
- ➤ `CISepiaTone`
- ➤ `CIVignette`

You can think of a filter as a black box (Figure 32-1), which requires a number of input parameters and returns an output. These input and output parameters differ depending on the filter.

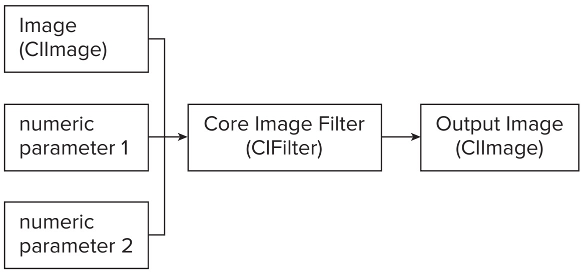


FIGURE 32-1

You can find a full list of filters in each category along with their parameters in the Core Image Filter Reference document. The output of one filter can be one of the inputs for another filter, thus allowing you to create chains of filters (Figure 32-2). Some filters can even take multiple images as input.

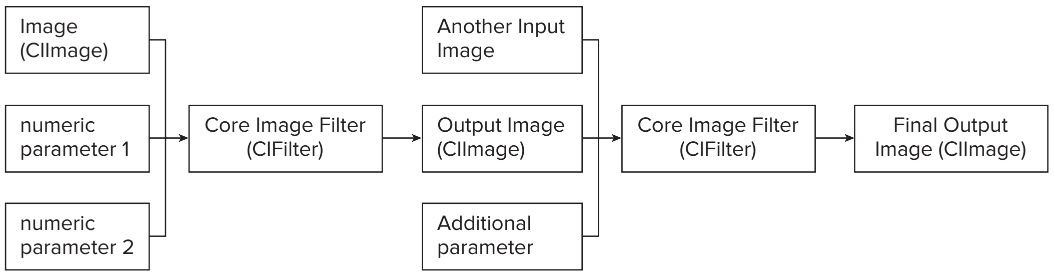


FIGURE 32-2

You can download the Core Image Filter Reference document from: `http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/CoreImaging/ci_concepts/ci_concepts.html#//apple_ref/doc/uid/TP30001185-CH202-TPXREF101`.

USING CORE IMAGE

Once you know the identifier of a filter, creating one is a simple matter of using the `filterWithName:` class method of the `CIFilter` class, as follows:

```
CIFilter* sepiaFilter = [CIFilter filterWithName:@"CISepiaTone"];
```

When you create a filter, you must send it the `setDefaults` message to initialize all its parameters to default values. These default values are specified in the Core Image Filter Reference document.

```
[sepiaFilter setDefaults];
```

Each attribute of a filter has a unique identifier. These identifiers are also known as keys. To set the value of an attribute, assuming you know its identifier, you use the `setValue:forKey:` method of the filter object:

```
[sepiaFilter setValue:[NSNumber numberWithFloat: 0.5]
             forKey:@"inputIntensity"];
```

You may be surprised to know that both the input image and the output images are considered filter attributes. The input image attribute is usually referred to with the `inputImage` key, and the output image attribute with the `outputImage` key. Thus, to specify a `CIImage` instance as input for the `CISepiaTone` filter, you would use the following code:

```
 [sepiaFilter setValue:inputCIImage forKey:@"inputImage"];
```

To access the output of the same filter, which would also be a `CIImage` instance, you would use the following code:

```
CIImage* resultCIImage = [sepiaFilter valueForKey:@"outputImage"];
```

`CIImage` objects can be created from `UIImage` objects using the following code:

```
UIImage* inputImage = [UIImage imageNamed:@"building.png"];
CIImage* inputCIImage = [CIImage imageWithCGImage:inputImage.CGImage];
```

To display the output of the filter in a `UIImageView`, you will need to convert the `CIImage` object into a `UIImage` object. You do this using the following code:

```
CIContext* context = [CIContext contextWithOptions:nil];
CGImageRef outputImageRef = [context createCGImage:resultCIImage
                             fromRect:CGRectMake(0, 0, 320, 480)];
UIImage* outputImage = [UIImage imageWithCGImage:outputImageRef];
```

The conversion involves first converting the `CIImage` object into a `CGImageRef` object and then creating the `UIImage` object from the `CGImageRef` object. `CGImageRef` is an object used to represent images in Core Graphics. When creating the `CGImageRef` object, you need to specify the dimensions of the image in a `CGRect` structure.

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `ImageFilters` that uses `Core Image` filters to perform image processing on a `UIImage` object.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image resources into the project.
- Add a `UIImageView` instance to the default scene and an appropriate outlet in the view controller file.
- Add a reference to the `Core Image` framework.
- Add two `UIButton` instances to the default scene and connect them to appropriate action methods in the view controller class.
- Perform image processing operations with `Core Image` when either button is tapped.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 32 folder in the download.*

Hints

- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.

Step-by-Step

1. Create a Single View Application in Xcode called `ImageFilters`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** ImageFilters
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson32
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Import the `building.png` resource from this chapter's resource folder on the DVD into the project.
3. Add a `UIImageView` instance to the default scene and connect it to an outlet in the view controller class.
    1. Open the storyboard file and use the Media library to drag and drop `building.png` onto the scene. This automatically creates an image view.
    2. Use the assistant editor to create an outlet in the view controller class called `imageView` and connect it to the image view.
4. Add two `UIButton` instances to the scene and connect their Touch Up Inside events to appropriate action methods in the view controller class.
    1. Set the title of the first button to `Sepia Tone` and use the Size inspector to resize/position it to X=8, Y=365, W=302, H=37.
    2. Set the title of the second button to `Hue Adjust` and use the Size inspector to resize/position it to X=8, Y=410, W=302, H=37.

3. Name the action method corresponding to the first button `onSepia`.
4. Name the action method corresponding to the second button `onHueAdjust`.

5. Add a reference to the `Core Image` framework.
6. Add the following line of code to the top of the `Lesson32ViewController.m` file:

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

7. Add the following code to the implementation of the `onSepia:` method in the `Lesson32ViewController.m` file:

```
// input image, converted from UIImage to CIImage
UIImage* inputImage = [UIImage imageNamed:@"building.png"];
CIImage* inputCIImage = [CIImage imageWithCGImage:inputImage.CGImage];

// the CIFilter, created and configured
CIFilter* sepiaFilter = [CIFilter filterWithName:@"CISepiaTone"];
[sepiaFilter setDefaults];
[sepiaFilter setValue: inputCIImage forKey: @"inputImage"];
[sepiaFilter setValue: [NSNumber numberWithFloat: 0.5]
             forKey: @"inputIntensity"];

// the result image
CIImage* resultCIImage = [sepiaFilter valueForKey: @"outputImage"];

// convert CIImage to UIImage
CIContext* context = [CIContext contextWithOptions:nil];
CGImageRef outputImageRef = [context createCGImage:resultCIImage
                            fromRect:CGRectMake(0, 0, 320, 480)];
UIImage* outputImage = [UIImage imageWithCGImage:outputImageRef];
imageView.image = outputImage;
```

8. Add the following code to the implementation of the `onHueAdjust:` method in the `Lesson32ViewController.m` file:

```
// input image, converted from UIImage to CIImage
UIImage* inputImage = [UIImage imageNamed:@"building.png"];
CIImage* inputCIImage = [CIImage imageWithCGImage:inputImage.CGImage];

// the CIFilter, created and configured
CIFilter* hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
[hueAdjust setDefaults];
[hueAdjust setValue: inputCIImage forKey: @"inputImage"];
[hueAdjust setValue: [NSNumber numberWithFloat: 2.094]
                     forKey: @"inputAngle"];

// the result image
CIImage* resultCIImage = [hueAdjust valueForKey: @"outputImage"];

// convert CIImage to UIImage
CIContext* context = [CIContext contextWithOptions:nil];
CGImageRef outputImageRef = [context createCGImage:resultCIImage
                             fromRect:CGRectMake(0, 0, 320, 480)];
UIImage* outputImage = [UIImage imageWithCGImage:outputImageRef];
imageView.image = outputImage;
```

9. Test your app in the iOS Simulator. Figure 32-3 shows the original image on the left hand side and the result of applying the sepia and hue adjustment filters to the original image.

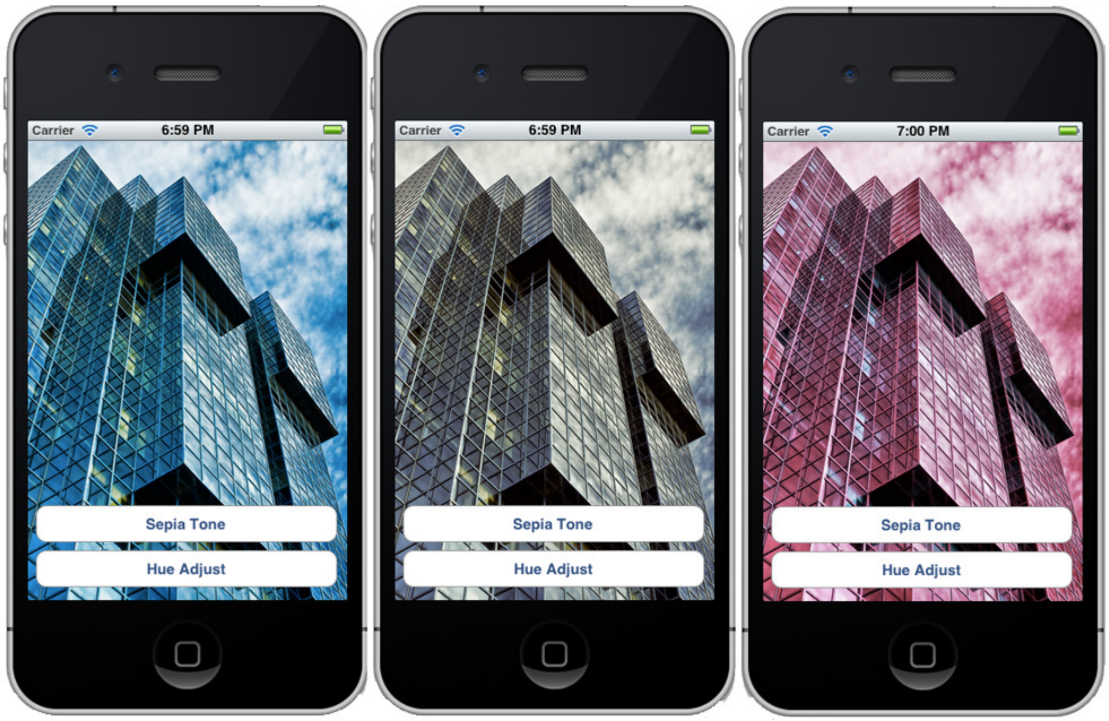


FIGURE 32-3

Please select Lesson 32 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

33 Building Universal Applications

Following up on the huge success of the iPhone and iPod Touch, Apple introduced the iPad—a device with a much larger screen size, running iOS. Developing for the iPad in most cases is similar to developing for the iPhone, except for the obvious difference in screen sizes. Certain features, like the ability to make a phone call or send an SMS, are not available on the iPad.

The latest version of the iPad when this book was written is the iPad2. It does not have a retina display and comes with two cameras. iPads may not necessarily have 3G capabilities.

If you were to run an iPhone application on an iPad, the application would appear in a small 320 × 480 window in the center of the screen, as shown in Figure 33-1. To take advantage of the extra screen space on the iPad, you need to create an application specifically for the iPad.

As an iOS developer you can create applications that are iPhone-only, iPad-only, or universal. A universal application is one that includes binaries for both iPhone and iPad in a single archive.

In most cases, if you want to create both an iPhone and iPad version of your application, you create a universal application project in Xcode. However, some developers like to keep two separate projects for the iPhone and the iPad. This is sometimes done if the iPhone and iPad versions of the application are significantly different in functionality, or simply because the developer wants to make more money by selling two copies of the application instead of one.

EXAMINING THE UNIVERSAL APPLICATION TEMPLATE

When you create a new project in Xcode, you are asked to specify the device family in the Project Options dialog box (Figure 38-2).

Select the Universal option to create a project that can run on both device families. Although you can use any Xcode template to create a universal application, this section uses the Single View Application template.

The first thing you will notice is that the project has two storyboards. This is understandable because the project targets two different device families with different screen sizes. A rather painful issue with developing a universal application is that you end up making two sets of user interfaces. If your storyboards use images, you will probably need two different sizes of each image.

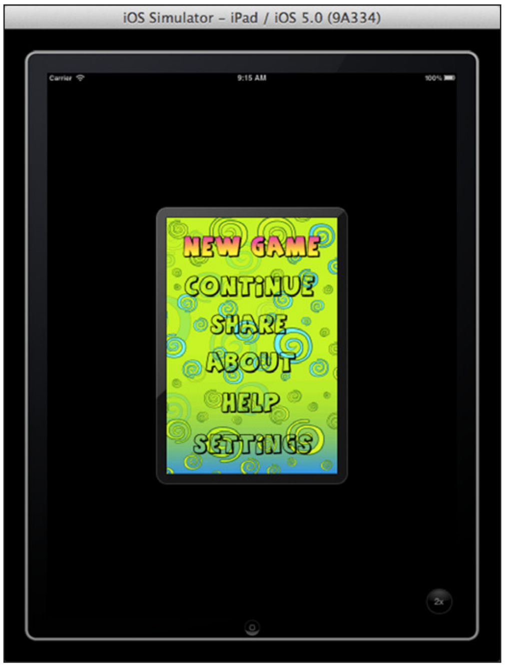


FIGURE 33-1

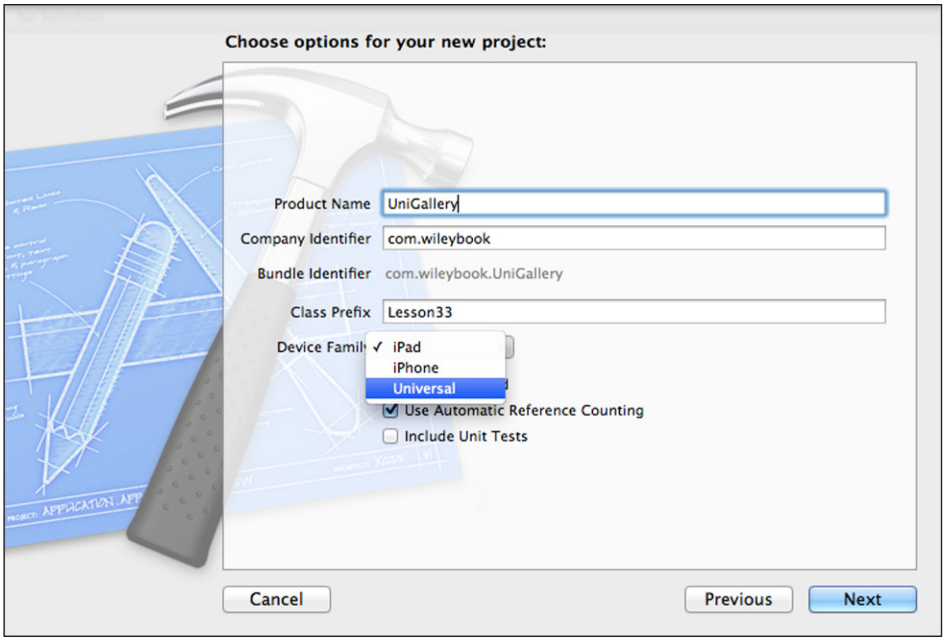
If making two applications in one is not something you are particularly fond of, you could create your user interfaces with code instead of using storyboards. Creating user interfaces programmatically was covered in Lesson 17.

As of the time when this lesson is being written, the iPad device family has a common screen size of W=768 × H=1024 units (with one unit equal to one pixel). The iPhone device family, on the other hand, has two screen sizes, one for the standard and the other for the retina-display version.

UIKit controls scale automatically on retina displays, but you will have to provide two versions of each image resource. The retina display versions will be twice the size of their standard counterparts, and their filenames must end with the `@2x` suffix.

In effect, this means that for your universal application to work well across all iOS device families, you need to provide three sizes for each image resource:

- iPhone standard
- iPhone retina
- iPad



Choose options for your new project:

Product Name: UniGallery
Company Identifier: com.wileybook
Bundle Identifier: com.wileybook.UniGallery
Class Prefix: Lesson33
Device Family: ✓ iPad / iPhone / Universal
[✓] Use Automatic Reference Counting
[ ] Include Unit Tests

FIGURE 33-2

Something else to note about the universal project template is that both storyboards share a common view controller class. Thus, it is inevitable that at some point in your Objective-C code you need to programmatically determine the device family that is executing the code. The following code snippet illustrates how to do just that:

```
if ([[UIDevice currentDevice] userInterfaceIdiom] ==
                                                  UIUserInterfaceIdiomPhone)
{
    // write iPhone specific code here
}
else
{
    // write iPad specific code here
}
```

Last but not least, you need to provide multiple versions of your application's launch image and icon, one for each device family. Table 33-1 lists the filenames and image sizes for the most commonly used images.

TABLE 33-1: Application Icon and Launch Image

FILENAME	IMAGE SIZE (PIXELS)	DESCRIPTION
`Icon.png`	57 × 57	App Store and Home screen icon on iPhone/ iPod touch
`Icon@2x.png`	114 × 114	Retina-display version of `Icon.png`
`Icon-72.png`	72 × 72	Home screen for iPad
`Default.png`	320 × 480	iPhone/iPod touch portrait launch image
`Default@2x.png`	640 × 960	Retina-display version of `Default.png`
`Default-Portrait.png`	768 × 1024	Portrait-mode IPad launch image
`Default-Landscape.png`	1024 × 768	Landscape-mode iPad launch image

TRY IT

In this Try It you build a universal application called `UniGallery` that allows the user to navigate through a gallery of images using swipe gestures. Being universal, the app will work on both the iPhone and the iPad.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image resources into the project.
- Add a `UIImageView` instance to both the iPhone and iPad storyboards, and connect the image view to an appropriate outlet in the view controller class.
- In the `viewDidLoad` method of the view controller class, populate an array with image filenames depending on the type of device that is executing the code.
- Implement swipe gesture recognizers in the view controller class.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 33 folder in the download.*

Hints

- When creating the Xcode project, set the value of the Device Family combo box to Universal.

Step-by-Step

1. Create a Single View Application in Xcode called `UniGallery`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** UniGallery
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson33
        - **Define Family:** Universal
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    > *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add an image view to the iPad storyboard.
    1. Select the `MainStoryboard_iPad.storyboard` file in the project navigator.
    2. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    3. Use the Object library to add a `UIImageView` instance and size it to fit the entire surface of the scene.
    4. Using the assistant editor, create an outlet for the image view in the view controller class and call the outlet `galleryImageView`.
3. Add an image view to the iPhone storyboard.
    1. Select the `MainStoryboard_iPhone.storyboard` file in the project navigator.
    2. Use the Object library to add a `UIImageView` instance and size it to fit the entire surface of the scene.

3. Ensure the `Lesson33ViewController.h` file is open in the assistant editor.
4. Right-click the image view instance to bring up a context menu. Make a connection from New Referencing Outlet in the context menu to the `galleryImageView` outlet in the assistant editor. While doing this step ensure you are not creating a new outlet, but instead connecting to an existing one.

4. Import image resources into the project.
    1. Ensure the project navigator is visible and the `UniGallery` project is selected and expanded. To show the project navigator, use the View ➪ Navigators ➪ Show Project Navigator menu item. To expand a project, click the triangle next to the project name in the project navigator.
    2. Right-click the Supporting Files group and select Add Files to UniGallery from the context menu.
    3. Select the `Images` folder in this lesson's resources on the DVD.
    4. Ensure the Copy Items to Destination Group's Folder (if needed) option is selected in the dialog box.
    5. Click the Add button.
5. Add an `NSArray` instance to the view controller class and populate it with a list of filenames.
    1. Add the following property declarations to the `Lesson33ViewController.h` file:

```
@property (strong, nonatomic) NSArray* imageFileNames;
@property int currentIndex;
```

    2. Synthesize the properties in the `Lesson33ViewController.m` file.
    3. Update the `viewDidLoad` method of the view controller class. In this method, create an array and populate it with filenames. The specific filenames added to the array will depend on the device on which this code is executed. Once the array is created, load the first image file to the image view. Add the following code to `viewDidLoad` method after the `[super viewDidLoad]` line:

```
if ([[UIDevice currentDevice] userInterfaceIdiom] ==
    UIUserInterfaceIdiomPhone)
{
    self.imageFileNames = [[NSArray alloc]
              initWithObjects:@"image_1.png"
              @"image_2.png",
              @"image_3.png",
              @"image_4.png",
              @"image_5.png",
              nil];
}
else
{
    self.imageFileNames = [[NSArray alloc]
              initWithObjects:@"image_1_ipad.png",
              @"image_2_ipad.png",
              @"image_3_ipad.png",
              @"image_4_ipad.png",
                @"image_5_ipad.png",
                nil];
}

// load first image
currentIndex = 0;
galleryImageView.image = [UIImage imageNamed:[imageFileNames
                         objectAtIndex:currentIndex]];
```

6. Add swipe gestures to the view controller class.

    1. Add the following method declarations to the `Lesson33ViewController.h` file:

    ```
    - (void) handleLeftSwipe:(UIGestureRecognizer *)gestureRecognizer;
    - (void) handleRightSwipe:(UIGestureRecognizer *)gestureRecognizer;
    ```

    2. Implement these methods in the `Lesson33ViewController.m` file as follows:

    ```
    - (void) handleLeftSwipe:(UIGestureRecognizer *)gestureRecognizer
    {
        if (currentIndex == ([imageFileNames count] - 1))
            return;

        currentIndex++;
        galleryImageView.image = [UIImage imageNamed:[imageFileNames

    objectAtIndex:currentIndex]];
    }

    - (void) handleRightSwipe:(UIGestureRecognizer *)gestureRecognizer
    {
        if (currentIndex == 0)
            return;

        currentIndex--;
        galleryImageView.image = [UIImage imageNamed:[imageFileNames

    objectAtIndex:currentIndex]];
    }
    ```

    3. Create two swipe gestures in the `viewDidLoad` method of the view controller class by adding the following code to the end of the method, after the code from step 5-3:

    ```
    // left swipe
    UISwipeGestureRecognizer* left_swipe = [[UISwipeGestureRecognizer alloc]
                                            initWithTarget:self
                              action:@selector(handleLeftSwipe:)];

    left_swipe.numberOfTouchesRequired = 1;
    left_swipe.cancelsTouchesInView = YES;
    left_swipe.direction = UISwipeGestureRecognizerDirectionLeft;
    [self.view addGestureRecognizer:left_swipe];

    // right swipe
    UISwipeGestureRecognizer* right_swipe = [[UISwipeGestureRecognizer
                                              alloc] initWithTarget:self
                                   action:@selector(handleRightSwipe:)];
    ```

```
        right_swipe.numberOfTouchesRequired = 1;
        right_swipe.cancelsTouchesInView = YES;
        right_swipe.direction = UISwipeGestureRecognizerDirectionRight;
        [self.view addGestureRecognizer:right_swipe];
```

7. Test your application in the iOS Simulator.
    1. Use the scheme/target multi-selector in the Xcode toolbar to ensure UniGallery ➪ iPhone 5.0 Simulator is selected.
    2. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
    3. Click the stop button in the Xcode toolbar.
    4. Now use the scheme/target multi-selector in the Xcode toolbar to ensure UniGallery ➪ iPad 5.0 Simulator is selected.
    5. Click the Run button in the Xcode toolbar.

Please select Lesson 33 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

34 Where Am I? Introducing Core Location

Core Location is a framework that allows applications to retrieve the location and heading of the device they are running on. To do this, Core Location can use a combination of a compass for heading, and either GPS, cellular radio, or WiFi technologies for location. Cellular radio and WiFi-based location is less accurate than GPS.

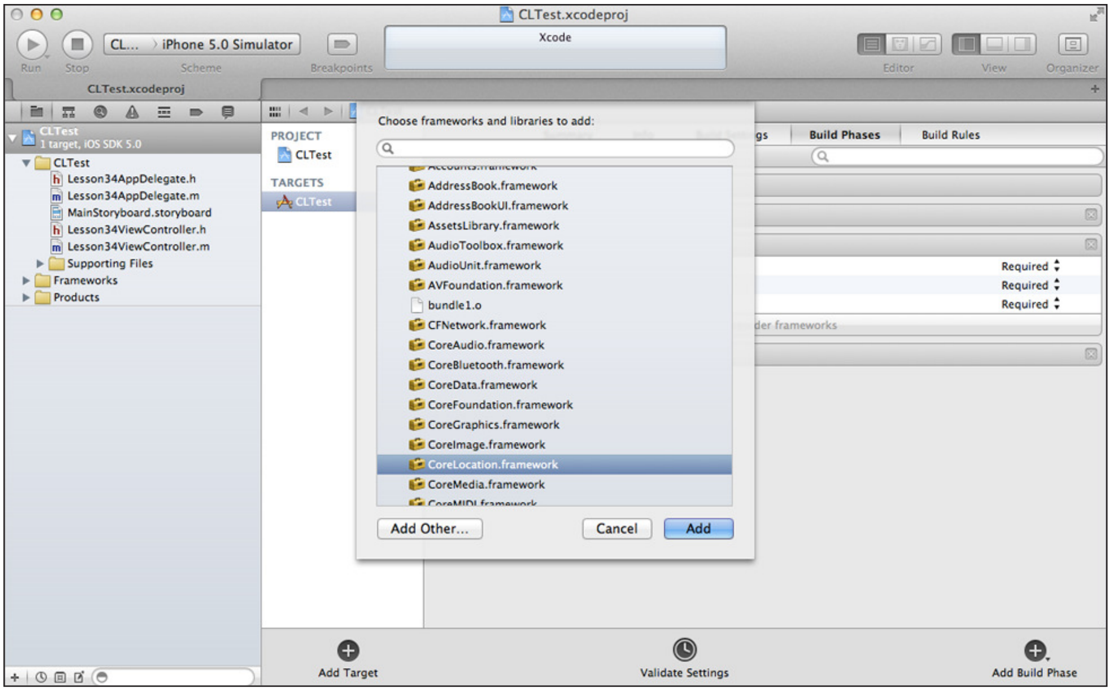
Applications cannot specify which method will be used; however, they can specify a desired level of accuracy. Depending on the desired level of accuracy, Core Location tries to use the GPS hardware, cellular radio, or WiFi in that order.

This framework is not included in any of the standard iOS application templates. To use this framework in your code you will need to add it manually to your project. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, switch to the Build Phases tab and click the + button under the Link Binary With Libraries category. Select `CoreLocation.framework` from the list of available frameworks (Figure 34-1).

Core Location defines a manager class called `CLLocationManager` that you can use to interact with the framework. It allows you to specify the desired frequency and accuracy of location information. To receive location updates in an application, you need to create an instance of the `CLLocationManager` class, and provide a delegate object to receive location updates and errors. This delegate object must implement the `CLLocationManagerDelegate` protocol.

The delegate object is often the view controller class, but could also be any other class in your application. Using location hardware can have a significant drain on the device's batteries, and hence applications need to turn on and turn off receiving location updates. The following code demonstrates the basic setup required to receive location updates:

```
// setup Core Location
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
[locationManager startUpdatingLocation];
```



Choose frameworks and libraries to add:

AddressBook.framework
AddressBookUI.framework
AssetsLibrary.framework
AudioToolbox.framework
AudioUnit.framework
AVFoundation.framework
bundle1.o
CFNetwork.framework
CoreAudio.framework
CoreBluetooth.framework
CoreData.framework
CoreFoundation.framework
CoreGraphics.framework
CoreImage.framework
CoreLocation.framework
CoreMedia.framework

Add Other... Cancel Add

FIGURE 34-1

This code assumes that `locationManager` is an instance variable defined in the class, and that the class implements the `CLLocationManagerDelegate` protocol. When your application does not want to receive location updates, it must send the `stopUpdatingLocation` message to the `CLLocationManager` instance:

```
    [locationManager stopUpdatingLocation];
```

An application can set up the `desiredAccuracy` property of the `CLLocationManager` instance to specify a desired accuracy. Core Location will try its best to achieve the desired accuracy. The more accurate a reading required, the more battery power is needed.

Applications should, in general, try to use the least accuracy possible to satisfy their requirements. The property can have the following values, listed in decreasing order of accuracy:

- `kCLLocationAccuracyBestForNavigation`
- `kCLLocationAccuracyBest`
- `kCLLocationAccuracyNearestTenMeters`
- `kCLLocationAccuracyHundredMeters`
- `kCLLocationAccuracyKilometer`
- `kCLLocationAccuracyThreeKilometers`

An application can also set up the `distanceFilter` property of the `CLLocationManager` instance to specify the minimum distance in meters. A device must move before an update is provided to the application.

The default value of this property is `kCLDistanceFilterNone`, which specifies the application wants to know of all movements.

HANDLING LOCATION UPDATES

The `CLLocationManagerDelegate` protocol defines two methods that are used by an application to handle a location update:

```
- (void)locationManager:(CLLocationManager *)manager
        didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation;

- (void)locationManager:(CLLocationManager *)manager
        didFailWithError:(NSError *)error;
```

A typical implementation of the `locationManager:didUpdateToLocation:fromLocation:` would resemble:

```
- (void)locationManager:(CLLocationManager *)manager
        didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation
{
    // lat/lon values should only be considered if
    // horizontalAccuracy is not negative.
    if (newLocation.horizontalAccuracy >= 0)
    {
        CLLocationDegrees currentLatitude = newLocation.coordinate.latitude;
        CLLocationDegrees currentLongitude =
                                   newLocation.coordinate.longitude;

        // do something with currentLatitude and currentLongitude.
    }

    // altitude values should only be considered if
    // verticalAccuracy is not negative.
    if (newLocation.verticalAccuracy >= 0)
    {
        CLLocationDegrees currentAltitude = newLocation.altitude;
        // do something with currentAltitude
    }
}
```

The `locationManager:didUpdateToLocation:fromLocation:` method's arguments are the `CLLocationManager` instance, and the current and previous locations as instances of `CLLocation` objects.

A `CLLocation` object encapsulates a location. It contains a `coordinate` property that is a structure containing a `latitude` and `longitude` member, each expressed as `CLLocationDegrees` values. `CLLocationDegrees` is an alias for a floating-point (decimal) value.

The location object also has the horizonalAccuracy property that signifies the radius of a circle centered at the coordinate property. The device can be anywhere within this circle. A larger horizontalAccuracy implies a larger circle, and thus a less accurate measurement. If the horizontalAccuracy property is negative, the reading should be discarded as being inaccurate.

The CLLocation object also provides altitude information using two properties: altitude and verticalAccuracy. A positive altitude value is a height above sea level, and a negative altitude is below sea level. A positive verticalAccuracy implies that the altitude measurement is off that amount; a negative value implies an invalid altitude measurement.

You can measure the distance between two locations using the distanceFromLocation method of the CLLocation class. The distance in meters is expressed as a CLLocationDistance value, which is also an alias for a floating-point value:

```
CLLocationDistance distanceTravelled =
                   [oldLocation distanceFromLocation:newLocation];
```

To compute the distance of a location update from a fixed point, you can instantiate a CLLocation object that represents the fixed point, and use the distanceFromLocation method as normal. For example, if you want to find out the distance of a location update from the center of London (lat = 51.5001524, lon = –0.1262362), you can use code similar to the following:

```
CLLocation *londonLocation = [[CLLocation alloc] initWithLatitude:51.5001524
                                                 longitude:-0.1262362];
CLLocationDistance distanceTravelled =
                   [londonLocation distanceFromLocation: newLocation];
```

HANDLING ERRORS AND CHECKING HARDWARE AVAILABILITY

When a user uses an application that uses Core Location for the first time, iOS will request the user for permission. The user has the option to deny the application access to location information. If this happens, or Core Location is unable to get a location fix, your delegate's locationManager:didFailWithError: method will be called. The error argument is of type NSError. Its code property can be examined to determine the reason for failure:

- kCLErrorDenied: The user has denied access to location data.
- kCLErrorLocationUnknown: Core Location has tried, but could not get a location fix.
- kCLErrorNetwork: There is no means for Core Location to get a location fix.

If the user has denied access to Core Location, then the CLLocationManager will not try to get a location fix again, and in such a case, it is best to send the stopUpdatingLocation message to the instance.

You can set up a message to be displayed to the user while asking for permission by using the purpose property of the CLLocationManager instance. This is done along with the code to create and initialize the location manager (Figure 34-2).

```
// setup Core Location
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
locationManager.purpose = @"This is a test application";
[locationManager startUpdatingLocation];
```

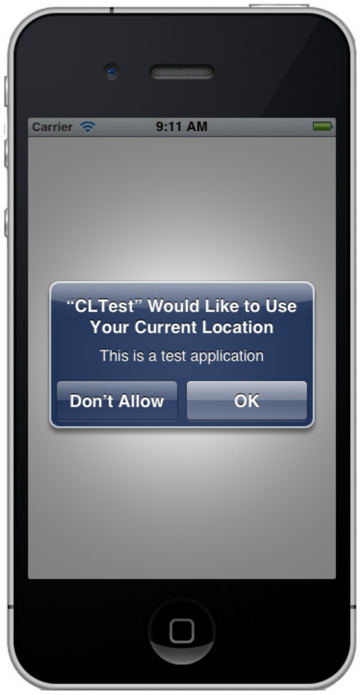

Carrier
9:11 AM
"CLTest" Would Like to Use Your Current Location
This is a test application
Don't Allow
OK

FIGURE 34-2

Some location services require the presence of specific hardware on the device. In general, you must check whether the desired service is available before attempting to use it. Table 34-1 lists some of the methods provided by the `CLLocationManager` class to test service availability.

TABLE 34-1: CLLocationManager Service Availability Methods

METHOD	DESCRIPTION
`+ (BOOL)locationServicesEnabled`	Returns YES if location services are enabled on the device. The user can disable location services from device settings.
`+ (BOOL)regionMonitoringAvailable`	Returns YES if region monitoring is supported on the current device.
`+ (BOOL)regionMonitoringEnabled`	Returns YES if region monitoring is currently enabled on the device. The user can disable region monitoring from device settings.
`+ (BOOL)headingAvailable`	Returns YES if the location manager is able to generate heading-related events.
`+ (CLAuthorizationStatus) authorizationStatus`	Returns a value indicating whether an application is authorized to use location services.

The iOS Simulator can simulate either a device at a fixed location, or a device that is moving along one of three preset routes. These features can be accessed from the Debug ➪ Location menu of the iOS Simulator.

Geocoding and Reverse Geocoding

Geocoding involves converting between a latitude/longitude coordinate pair and an address. Core Location provides the `CLGeocoder` class that provides methods to perform both forward and reverse geocoding. Forward-geocoding involves converting from an address to a latitude/longitude value. Reverse-geocoding involves converting a latitude/longitude value into an address. The result of a geocoding request is represented by a `CLPlacemark` object. A forward-geocoding request returns an array of `CLPlacemark` objects because multiple results may be returned.

You should try to use one geocoding request per action, and avoid making the same geocoding request multiple times. To perform a forward-geocoding request from an address string, you can send the geocoder a `geocodeAddressString:completionHandler:` message. This message requires you to specify an `NSString` object that contains an address string and a block handler that is called when the geocoding operation is complete. The following example converts an address string into a latitude/longitude coordinate pair:

```
CLGeocoder *localGeocoder = [[CLGeocoder alloc] init];
[localGeocoder
geocodeAddressString:@"170 Bilton Road, Perivale, UB6 7HL, United Kingdom"
completionHandler:^(NSArray *placemarks, NSError *error)
    {
         if (placemarks != nil){
             int number_of_placemarks = [placemarks count];
             CLPlacemark *firstPlacemark = [placemarks objectAtIndex:0];

             double latValue = firstPlacemark.location.coordinate.latitude;
             double lonValue = firstPlacemark.location.coordinate.longitude;
    }
}];
```

You can send the geocoder a reverse-geocoding request by sending it the `reverseGeocodeLocation:completionHandler:` message as shown here:

```
CLGeocoder *localGeocoder = [[CLGeocoder alloc] init];
CLLocation *londonLocation = [[CLLocation alloc] initWithLatitude:51.5001524
                                                 longitude:-0.1262362];
[localGeocoder reverseGeocodeLocation:londonLocation
               completionHandler:^(NSArray *placemarks, NSError *error)
               {
                 if (placemarks != nil){
                 CLPlacemark *firstPlacemark = [placemarks objectAtIndex:0];
                 NSString *countryCode = firstPlacemark.ISOcountryCode;
                 NSString *countryName = firstPlacemark.country;
                 NSString *adminArea = firstPlacemark.administrativeArea;
                 NSString *city = firstPlacemark.locality;
                 NSString *postCode = firstPlacemark.postalCode;
                 NSString *streetAddress1 = firstPlacemark.thoroughfare;
               }
 }];
```

The message requires you to provide a CLLocation object that represents a latitude/longitude coordinate pair and block handler that is called with the results of the reverse-geocoding operation. The CLLocation instance in this example is created with a fixed set of coordinates (lat=51.5001524, lon=−0.1262362) but could have just as well been obtained from a location update.

The actual geocoding operation is performed asynchronously. The results are supplied as an array of CLPlacemark objects; however in this case the array will contain just one element. If an error occurred, the array is nil and the error variable contains more information on the error.

A CLPlacemark object contains several properties that encapsulate information on an address associated with a specific coordinate. Some of the properties are:

- location: A CLLocation object that provides the coordinate pair associated with the placemark.
- ISOcountryCode: An NSString object that contains the abbreviated country code.
- country: An NSString object that contains the name of country.
- postalCode: An NSString object that contains the postal code.
- administrativeArea: An NSString object that contains the state/province.
- locality: An NSString object that contains the city.
- thoroughfare: An NSString object that contains the street address.
- subThoroughfare: An NSString object that contains additional street address information.

If the coordinates lie over an inland water body, or an ocean, this information can be accessed through the inlandWater and ocean properties, respectively, both of which are NSString objects.

OBTAINING COMPASS HEADINGS

A compass has been included as part of iPhone 3GS, iPhone 4, iPad, and iPad 2. You can determine if a compass is available on a device by sending the headingAvailable message to the location manager. If a compass is available on the device, you can use the location manager to receive heading updates. Heading updates work much like location updates. Once you have set up the CLLocationManager instance, you can send it the startUpdateHeading message to begin receiving heading updates.

The CLLocationManagerDelegate protocol defines two methods that are related to heading updates:

```
- (void)locationManager:(CLLocationManager *)manager
        didUpdateHeading:(CLHeading *)newHeading

- (BOOL)locationManagerShouldDisplayHeadingCalibration:
                              (CLLocationManager *)manager
```

Heading data is supplied as a CLHeading object to the locationManager:didUpdateHeading delegate method. The CLHeading class encapsulates the magnetic heading, the true heading, and an accuracy measure in its magneticHeading, trueHeading, and headingAccuracy properties, respectively.

The earth's geographic north pole is different from the magnetic north pole. The geographic north pole is fixed at the north pole, whereas magnetic north pole is a few hundred miles away. Make sure you know the difference between geographic north and magnetic north when you build any application that uses the compass feature.

The geographic north pole heading is contained in the `trueHeading` member of the `CLHeading` instance. Data in this member is available only if you enable both heading updates and location updates.

The `locationManagerShouldDisplayHeadingCalibration` message is sent to the delegate object when the location manager wants to display a calibration prompt to the user. If you find this prompt annoying, you can implement this method to return NO. If you were to do so, the compass would try to calibrate itself automatically but the results of the calibration process may not be accurate.

The iOS Simulator cannot simulate compass headings. You need to test applications that require this feature on an actual device.

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `CLTest` that displays the current location and the distance traveled since the last location reading was obtained.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a few `UILabel` elements that will display the location readings. Create outlets for these in the view controller class.
- Add a `UIButton` that will be used to stop/start receiving location updates. Create an appropriate outlet and action.
- Initialize Core Location when the button is pressed. Stop receiving location updates when the button is pressed a second time.
- Implement `CLLocationManagerDelegate` methods.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 34 folder in the download.*

Hints

- Remember to use the `purpose` property of the `CLLocationManager` instance to provide a description of what your application intends to do with the location data.
- Your application should send the `stopUpdatingLocation` message to the location manager when it does not require location updates.
- You need to add a reference to the Core Location framework to the project.

Step by-Step

1. Create a Single View Application in Xcode called `CLTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** CLTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson34
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a reference to the Core Location framework.
    1. In Xcode, make sure the project navigator is visible. To show it, use the View ➪ Navigators ➪ Show Project Navigator menu item.
    2. Click the root (project) node of the project navigator to display project settings.
    3. Select the Build Phases tab.
    4. Expand the Link Binary With Libraries group in this tab.
    5. Click the + button at the bottom of this group and select `CoreLocation.framework` from the list of available frameworks.
    6. Click the Add button.
3. Add six `UILabel` instances to the default scene to display location readings.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.

2. Use the Object library to add six UILabel instances to the default scene. Double-click each label in turn and change its text to `Latitude`, `Longitude`, `Distance Travelled`, `latitudeValue`, `longitudeValue`, and `distanceValue` respectively.
3. Resize/position them using the Size inspector as per Table 34-2.

TABLE 34-2: Size and Position of Labels

LABEL	X	Y	WIDTH	HEIGHT
`Latitude`	18	34	62	21
`Longitude`	18	79	77	21
`Distance Travelled`	18	123	141	21
`latitudeValue`	167	34	99	21
`longitudeValue`	167	79	113	21
`distanceValue`	167	123	107	21

4. Using the assistant editor, create outlets for the `latitudeValue`, `longitudeValue`, and `distanceValue` labels. Call these `latValue`, `lonValue`, and `distValue`, respectively.

4. Add a `UIButton` instance to start/stop receiving location updates.
   1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
   2. Use the Object library to add a `UIButton` instance.
   3. Resize and position it to X = 25, Y = 172, W = 275, H = 37.
   4. Double-click the button and set its title to Start Location Updates.
   5. Using the assistant editor, create an outlet called `toggleButton` in the `Lesson34ViewController` class and connect it to the button.
   6. Using the assistant editor, create an action method in the view controller class and connect it to the Touch Up Inside event of the button. Call the new method `onButtonPressed`.
5. Add the following line to the top of the view controller's header file, to import the Core Location framework:

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

6. Add the following `@property` declarations to the `Lesson34ViewController.h` file:

```
@property BOOL hasInitialized;
@property BOOL hasStarted;
@property (strong, nonatomic) CLLocationManager* locationManager;
```

7. Make the `Lesson34ViewController` class conform to the `CLLocationManagerDelegate` protocol by modifying its interface declaration to:

```
@interface Lesson34ViewController : UIViewController <CLLocationManagerDelegate>
```

8. Add the following `@synthesize` statements to the `Lesson34ViewController.m` file:

```
@synthesize hasInitialized;
@synthesize hasStarted;
@synthesize locationManager;
```

9. Add the following code to the `viewDidLoad` method of the view controller class after the `[super viewDidLoad]` line:

```
// setup Core Location
hasInitialized = YES;
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
locationManager.purpose = @"This application will display your current
           location and the distance travelled since the last reading.";

// setup button
hasStarted = NO;
[toggleButton setTitle:@"Start Location Updates"
              forState:UIControlStateNormal];
```

10. Add the following code to the `viewDidUnload` method of the view controller class:

```
if (hasInitialized == YES)
        [locationManager stopUpdatingLocation];
```

11. Add the following code to the `onButtonPressed:` method of the view controller class:

```
// do not do anything if location manager is not setup
if (hasInitialized == NO)
    return;

// start
if (hasStarted == NO)
{
    [toggleButton setTitle:@"Stop Location Updates"
                  forState:UIControlStateNormal];
    [locationManager startUpdatingLocation];
    hasStarted = YES;
    return;
}

// stop
else if (hasStarted == YES)
{
    [toggleButton setTitle:@"Start Location Updates"
                  forState:UIControlStateNormal];
    [locationManager stopUpdatingLocation];
    hasStarted = NO;
    return;
}
```

12. Implement the `locationManager:didFailWithError:` delegate method in the `Lesson34ViewController.m` file as follows:

```
- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
    if (error.code == kCLErrorDenied)
    {
        hasInitialized = NO;
        [locationManager stopUpdatingLocation];
    }
}
```

13. Implement the `locationManager:didUpdateToLocation:FromLocation:` delegate method in the `Lesson34ViewController.m` file as follows:

```
- (void)locationManager:(CLLocationManager *)manager
        didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation
{
    // lat/lon values should only be considered if
    // horizontalAccuracy is not negative.
    if (newLocation.horizontalAccuracy >= 0)
    {
        CLLocationDegrees currentLatitude = newLocation.coordinate.latitude;
        CLLocationDegrees currentLongitude =
                                   newLocation.coordinate.longitude;
        CLLocationDistance distanceTravelled =
                          [oldLocation distanceFromLocation:newLocation];

        latValue.text = [NSString stringWithFormat:@"%2.3f",
                                  currentLatitude];
        lonValue.text = [NSString stringWithFormat:@"%2.3f",
                                  currentLongitude];
        distValue.text = [NSString stringWithFormat:@"%2.3f",
                                   distanceTravelled];
    }
}
```

14. Test your application in the iOS Simulator.

1. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.
2. Click the Start Location Updates button.
3. Use the iOS Simulator's ability to simulate a device on the move by selecting the Debug ➪ Location ➪ City Bicycle Ride menu item.

Please select Lesson 34 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

35 Introducing Map Kit

In the previous lesson you learned how to locate a device using Core Location. In this lesson you learn how to integrate a map within your application.

The Map Kit framework provides the `MKMapView` class for adding maps into your views. Map Kit also provides additional classes for annotating the map. The Map Kit framework uses Google's map service internally. Using a class within this framework binds you to the Google Maps/Google Earth terms of service. You can find these at `http://code.google.com/apis/maps/iphone/terms.html`.

The Map Kit framework is often used in conjunction with the Core Location framework, neither of which are included in any of the standard iOS application templates. To use these frameworks in your code you need to add them manually to your project. You can do this from the Project Settings page in Xcode. Select the Project node in the project navigator to display the settings page. On the settings page, switch to the Build Phases tab and click the + button under the Link Binary With Libraries category. Select the Map Kit framework from the list of available frameworks (Figure 35-1). Repeat this step for the Core Location framework.

You can add a map view to an existing view controller or storyboard using the Object library. Simply drag an instance of a map view and create an outlet for it in the view controller class.

The map view handles zooming and scrolling automatically. You can use the Attributes inspector to choose from Map, Satellite, and Hybrid modes (Figure 35-2). You can also set up the map to use Core Location to display the user's location by checking the Shows User Location property.

You can also set up these properties programmatically by using the `mapType` property of the `MKMapView` instance to specify the map mode. The `mapType` property can take one of three values: `MKMapTypeStandard`, `MKMapTypeSatellite`, or `MKMapTypeHybrid`. To enable/disable zooming and scrolling use the `zoomEnabled` and `scrollEnabled` properties, respectively. To have the map display the user's location, set the `showsUserLocation` property to `YES`.

You can set up the initial coordinate and zoom factor of the map by defining a map region and using the `setRegion:animated` method of the `MKMapView` instance.

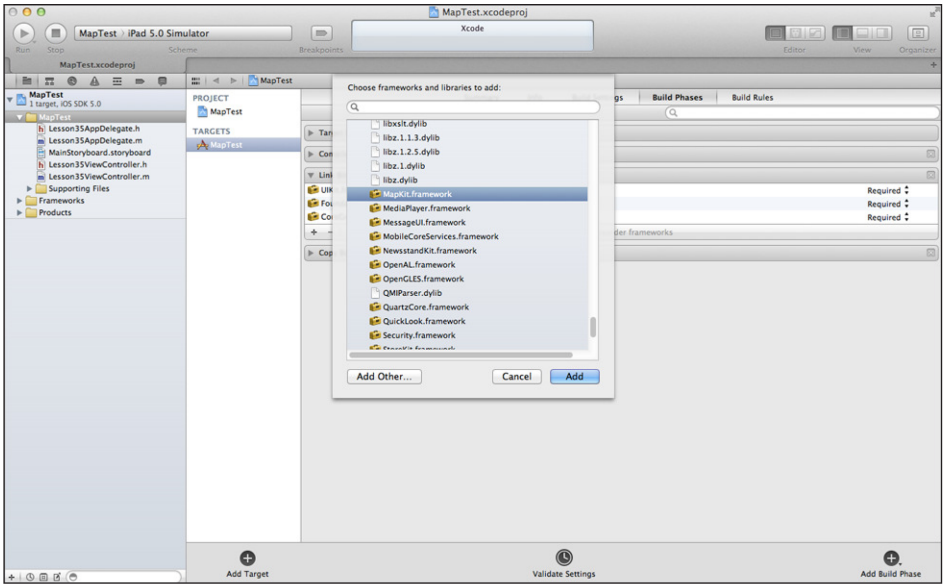
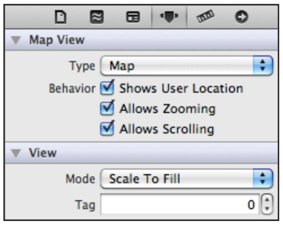


FIGURE 35-1



Map View

Type: Map

Behavior: ✓ Shows User Location
✓ Allows Zooming
✓ Allows Scrolling

View

Mode: Scale To Fill

Tag: 0

FIGURE 35-2

A region is represented by the `MKCoordinteRegion` structure and has members called `center` and `span`. The `center` member is a `CLLocationCoordinate2D` structure and has the members `latitude` and `longitude`. The `span` member is an `MKCoordinateSpan` structure and has the members `latitudeDelta` and `longitudeDelta` that specify a rectangular region around the center in degrees of latitude and longitude.

To create a region and apply it, you use code similar to the following:

```
// setup the map's location and zoom factor
MKCoordinateRegion mapRegion;
mapRegion.center.latitude=51.5001524;
mapRegion.center.longitude=-0.1262362;
mapRegion.span.latitudeDelta=0.2;
mapRegion.span.longitudeDelta=0.2;

[mapView setRegion:mapRegion animated:YES];
```

The above code snippet assumes that `mapView` is an outlet connected to the map view object created with Interface Builder.

ADDING ANNOTATIONS

The MKMapView class enables you to add custom annotations to a map. Because a map can potentially display several annotations at the same time, the designers of Map Kit decided to use separate objects to represent the data contained in an annotation and the view used to display it. The idea was that view objects could be reused with different data objects.

The data portion of an annotation is encapsulated by an instance of a class that implements the MKAnnotation protocol and contains information about the coordinates on the map and a description that is displayed in a callout.

The MKAnnotation protocol defines the coordinate, title, and subtitle properties. The coordinate property is a CLLocationCoordinate2D structure, and the title and subtitle properties are NSString objects. To conform to this protocol, your class must contain these properties. An example interface of such a class, PlacemarkClass, is shown here:

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

@interface PlacemarkClass : NSObject <MKAnnotation>

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, readonly, copy) NSString *title;
@property (nonatomic, readonly, copy) NSString *subtitle;

-(id)initWithCoordinate:(CLLocationCoordinate2D)annotCoordinate
title:(NSString*)annotTitle subtitle:(NSString*)annotSubtitle;

@end
```

Note that the class has an initializer method that enables you to specify an initial coordinate, title, and subtitle. The implementation of this class is listed next:

```
#import "PlacemarkClass.h"
@implementation PlacemarkClass

@synthesize coordinate;
@synthesize subtitle;
@synthesize title;

-(id)initWithCoordinate:(CLLocationCoordinate2D)annotCoordinate
title:(NSString*)annotTitle subtitle:(NSString*)annotSubtitle
{
    self = [super init];
    if (self)
    {
        coordinate = annotCoordinate;
        subtitle = [[NSString alloc] initWithString:annotSubtitle];
        title = [[NSString alloc] initWithString:annotTitle];
    }

    return self;
}
@end
```

To instantiate a `PlacemarkClass` object and add it as an annotation to the `mapView` object, you can use the `addAnnotation:animated:` method, as demonstrated by the following code:

```
// drop a pin on parliament square
CLLocationCoordinate2D parliamentLocation =
                       CLLocationCoordinate2DMake(51.5001524, -0.1262362);
parliamentAnnotation = [[PlacemarkClass alloc]
                        initWithCoordinate:parliamentLocation
                        title:@"Parliament Square"
                        subtitle:@"Big Ben is here!"];

[mapView addAnnotation:parliamentAnnotation];
```

The view portion of an annotation is represented by a subclass of the `MKAnnotationView` class. Apple provides a subclass called `MKPinAnnotationView` that you can use for standard pin/call-out annotations. The `MKMapView` instance requests this view from a delegate object when it is required. The delegate object must implement the `MKMapViewDelegate` protocol, which defines the `mapView:viewForAnnotation:` method.

Typically, the delegate object will be your view controller class. You can set up the delegate by using either the Interface Builder (Figure 35-3), or setting the `delegate` property of the `MKMapView` instance.

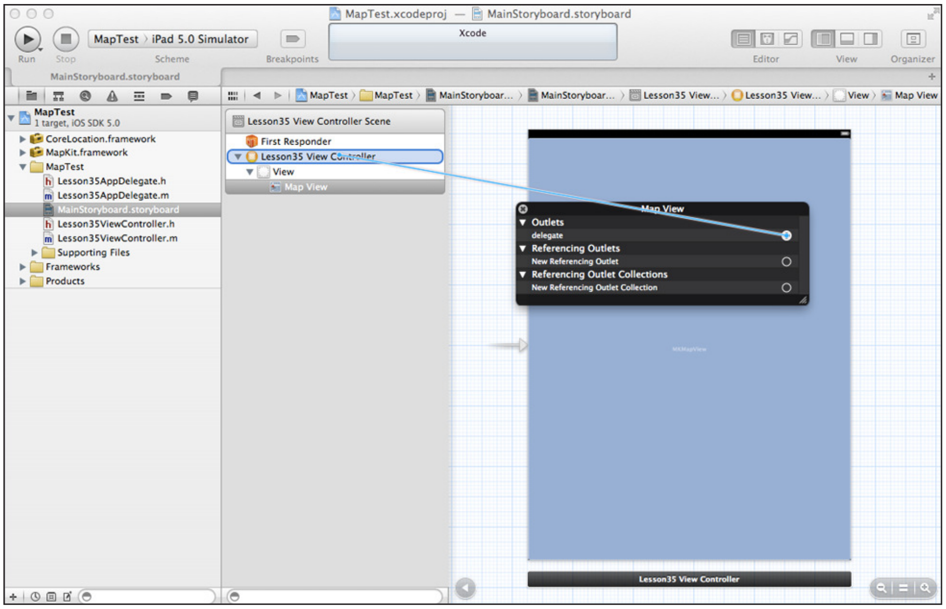


FIGURE 35-3

A typical implementation of this delegate method follows:

```
- (MKAnnotationView *) mapView:(MKMapView *)mapView
   viewForAnnotation:(id <MKAnnotation>) annotation
{
    MKPinAnnotationView *newAnnotation =
                                [[MKPinAnnotationView alloc]
                                initWithAnnotation:annotation
                                reuseIdentifier:@"annotation1"];

    newAnnotation.pinColor = MKPinAnnotationColorGreen;
    newAnnotation.animatesDrop = YES;
    newAnnotation.canShowCallout = YES;
    [newAnnotation setSelected:YES animated:YES];
    return newAnnotation;
}
```

The annotation object for which a view is required is specified in the annotation parameter. Once you have allocated an `MKPinAnnotationView` instance, you can set up its pin color using the `pinColor` property. If you want the pin to display a callout when tapped, set the `canShowCallout` property to `YES`. If you want the pin drop animation, set `animatesDrop` to `YES`.

The `pinColor` property can be one of three values:

- `MKPinAnnotationColorGreen`
- `MKPinAnnotationColorRed`
- `MKPinAnnotationColorPurple`

TRY IT

In this Try It, you build an iPad application called `MapTest` that displays the current location and the location of Big Ben on a map. The user can use a segmented control to change the map style to either standard, satellite, or hybrid.

> *Although this book does not have a lesson dedicated specifically to the segmented control, it is often used with maps. You can follow the steps outlined in this Try It to use a segmented control with a map; however, if you would like more information on the segmented control, refer to the UISegmentedControl class reference, available at:* `http://developer.apple.com/library/ios/#documentation/uikit/reference/UISegmentedControl_Class/Reference/UISegmentedControl.html`.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a map view to the default scene and create an outlet for it in the view controller class.
- Add a segmented control and add an action for it in the view controller class.
- Add a reference to the Map Kit and Core Location frameworks
- Create a subclass of `NSObject` that implements the `MKAnnotation` protocol to use as the annotation data class.
- Initialize the map view in the view controller's `viewDidLoad` method.

- Implement the `MKMapViewDelegate` protocol in your view controller class.
- Change the map style when the active segment in the segmented control is changed.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson 35 folder in the download.*

Hints

- Remember to add a reference to both the Map Kit and the Core Location frameworks.

Step-by-Step

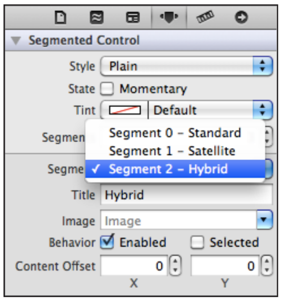
1. Create a Single View Application in Xcode called `MapTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** MapTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson35
        - **Define Family:** iPad
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

    For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a reference to the Core Location and Map Kit frameworks.
    1. In Xcode, make sure the project navigator is visible. To show it, use the View ➪ Navigators ➪ Show Project Navigator menu item.
    2. Click the root (project) node of the project navigator to display project settings.
    3. Select the Build Phases tab.

    4. Expand the Link Binary With Libraries group in this tab.
    5. Click the + button at the bottom of this group and select `CoreLocation.framework` from the list of available frameworks.
    6. Click the Add button.
    7. Click the + button at the bottom of this group and select `MapKit.framework` from the list of available frameworks.
    8. Click the Add button.

3. Add a map view to the default scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add a Map View to the default scene of the storyboard.
    3. Use the Size inspector to resize and position the map view to X = 0, Y = 0, Width = 768, Height = 900
    4. Using the assistant editor, create an outlet called `mapView` and connect it to the map view instance in the default scene.
4. Add a segmented control to the scene.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add a Segmented Control instance.
    3. Use the Attributes inspector to set the number of segments to 3.
    4. Use the Attributes inspector to name the three segments Standard, Satellite, and Hybrid, respectively (Figure 35-4).



Segmented Control
Style: Plain
State: ☐ Momentary
Tint: Default
Segment 0 – Standard
Segment 1 – Satellite
✓ Segment 2 – Hybrid
Title: Hybrid
Image: Image
Behavior: ✓ Enabled ☐ Selected
Content Offset: 0 (X) 0 (Y)

FIGURE 35-4

    5. Using the Size inspector, resize and position it to X = 223, Y = 941, W = 333, H = 44.
    6. Using the assistant editor, create an outlet in the view controller class called `mapModeSegmentControl` and connect it to the segmented control in the default scene.

7. Using the assistant editor, create an action in the view controller class and connect it to the Value Changed event of the `UISegmentedControl`. Call the new method `onSegmentChanged`.

5. Create a new Objective-C class to represent annotation data.

   1. Create a new Objective-C class by selecting the File ➪ New ➪ New File menu item.
   2. Select the Objective-C class template and click Next.
   3. Make the new class a subclass of `NSObject`.
   4. Name the new class `PlacemarkClass`.
   5. Edit the interface of the new class to resemble the following:

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

@interface PlacemarkClass : NSObject <MKAnnotation>

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, readonly, copy) NSString *title;
@property (nonatomic, readonly, copy) NSString *subtitle;

-(id)initWithCoordinate:(CLLocationCoordinate2D)annotCoordinate
title:(NSString*)annotTitle subtitle:(NSString*)annotSubtitle;

@end
```

   6. Edit the implementation of the class to resemble the following:

```
#import "PlacemarkClass.h"

@implementation PlacemarkClass

@synthesize coordinate;
@synthesize subtitle;
@synthesize title;

-(id)initWithCoordinate:(CLLocationCoordinate2D)annotCoordinate
title:(NSString*)annotTitle subtitle:(NSString*)annotSubtitle
{
    self = [super init];
    if (self)
    {
        coordinate = annotCoordinate;
        subtitle = [[NSString alloc] initWithString:annotSubtitle];
        title = [[NSString alloc] initWithString:annotTitle];
    }

    return self;
}

@end
```

6. Add the following lines to the top of the `Lesson35ViewController.h` file:

```
#import <MapKit/MapKit.h>
#import "PlacemarkClass.h"
```

7. Add the following property declaration to the `Lesson35ViewController.h` file:

```
@property (strong, nonatomic) PlacemarkClass* parliamentAnnotation;
```

8. Declare the `Lesson35ViewController` class to conform to the `MKMapViewDelegate` protocol by modifying its interface declaration to:

```
@interface Lesson35ViewController : UIViewController <MKMapViewDelegate>
```

9. Synthesize the `parliamentAnnotation` property in the `Lesson35ViewController.m` file:

```
@synthesize parliamentAnnotation;
```

10. Add the following code to the `viewDidLoad` method of the `Lesson35ViewController` class after the `[super viewDidLoad]` line:

```
// setup the map's delegate
mapView.delegate = self;

// setup the map's location and zoom factor
MKCoordinateRegion mapRegion;
mapRegion.center.latitude=51.5001524;
mapRegion.center.longitude=-0.1262362;
mapRegion.span.latitudeDelta=0.2;
mapRegion.span.longitudeDelta=0.2;

[mapView setRegion:mapRegion animated:YES];

// drop a pin on parliament square
CLLocationCoordinate2D parliamentLocation =
                       CLLocationCoordinate2DMake(51.5001524, -0.1262362);
parliamentAnnotation = [[PlacemarkClass alloc]
                        initWithCoordinate:parliamentLocation
                        title:@"Parliament Square"
                        subtitle:@"Big Ben is here!"];
mapView addAnnotation:parliamentAnnotation];
```

11. Implement the `MKMapViewDelegate` method `mapView:viewForAnnotation:` in your view controller class as follows:

```
- (MKAnnotationView *) mapView:(MKMapView *)mapView
   viewForAnnotation:(id <MKAnnotation>) annotation
{
    MKPinAnnotationView *newAnnotation = [[MKPinAnnotationView alloc]
                                          initWithAnnotation:annotation
                                          reuseIdentifier:@"annotation1"];
    newAnnotation.pinColor = MKPinAnnotationColorGreen;
    newAnnotation.animatesDrop = YES;
    newAnnotation.canShowCallout = YES;
    [newAnnotation setSelected:YES animated:YES];
    return newAnnotation;
}
```

12. Add the following code to the `onSegmentChanged:` method of the view controller class:

```
if (mapModeSegmentControl.selectedSegmentIndex == 0)
{
    mapView.mapType = MKMapTypeStandard;
}
else if (mapModeSegmentControl.selectedSegmentIndex == 1)
{
    mapView.mapType = MKMapTypeSatellite;
}
else if (mapModeSegmentControl.selectedSegmentIndex == 2)
{
    mapView.mapType = MKMapTypeHybrid;
}
```

13. Test your application in the iOS Simulator.

    1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

    2. Change the selected segment on the segmented control to switch map types (Figure 35-5).

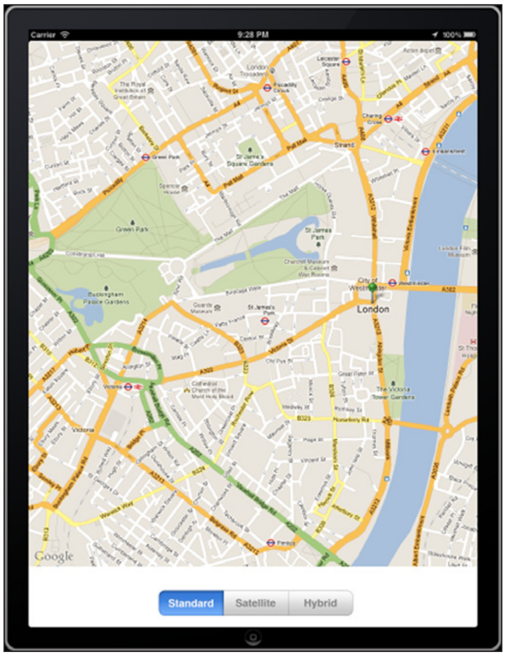


FIGURE 35-5

Please select Lesson 35 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

36 Using the Camera and Photo Library

All iOS devices, with the exception of the first-generation iPad, have at least one camera. When a user takes a picture with the camera, the image is stored in the device's photo library. This lesson shows you how to allow the user to pick an image from the photo library or take a new picture with the camera and use it in your application.

The UIKit framework contains a class called `UIImagePickerController` designed specifically to allow you to access the camera and photo library from your applications. This class presents its own user interface (Figure 36-1) that allows a user to browse through the photo library or control the camera. All you have to do is present this view controller in your application and provide a delegate method whose methods are called when the user has finished selecting an image.

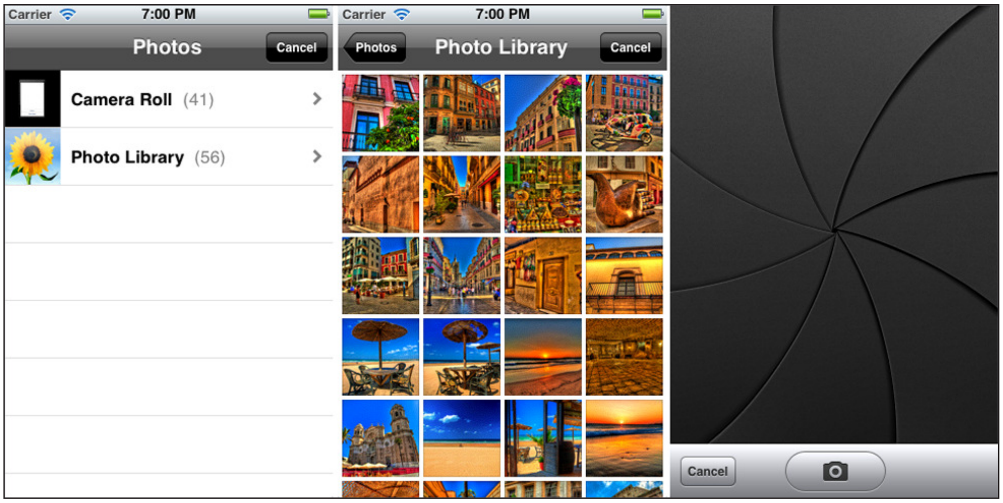


FIGURE 36-1

The image picker controller can also be used to record videos and access these recorded videos within your application. Video recording and playback is not covered in this book.

Creating an instance of the `UIImagePickerController` is a simple matter of sending it an `alloc` and `init` message:

```
UIImagePickerController* imagePicker = [[UIImagePickerController alloc]
                                         init];
```

The `UIImagePickerController` class can be used to access the contents of either the photo library, saved photos album, or the camera. You can specify the source by providing a value for the `sourceType` property. This value can be one of the following:

- `UIImagePickerControllerSourceTypePhotoLibrary`
- `UIImagePickerControllerSourceTypeCamera`
- `UIImagePickerControllerSourceTypeSavedPhotosAlbum`

The first-generation iPad and iPod touch devices do not have a camera, hence you should check to see if the device has a camera before trying to set it as a source for the image picker. To check if a particular source type is available, use the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class as follows:

```
BOOL hasCamera = [UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera];
```

When the `sourceType` is set to use the camera, you can specify which camera is to be used if your device has multiple cameras. By default, the image picker uses the rear camera. To find out if front and rear cameras are available, use the `isCameraDeviceAvailable` class method as shown in the following code snippet:

```
BOOL hasFrontCamera = [UIImagePickerController
isCameraDeviceAvailable:UIImagePickerControllerCameraDeviceFront];

BOOL hasRearCamera = [UIImagePickerController
isCameraDeviceAvailable:UIImagePickerControllerCameraDeviceRear];
```

Once you have determined that the camera you want to use is available, you can specify it using the `cameraDevice` property of the image picker instance. For instance, to use the front camera, use the following code:

```
imagePicker.cameraDevice = UIImagePickerControllerCameraDeviceFront;
```

To display the image picker as a modal sheet, use the `presentModalViewController:animated:` method on your active view controller object:

```
[self presentModalViewController:imagePicker animated:YES];
```

On an iPad, you can also display an image picker in a popover controller. The following code snippet shows how this can be done from a method in your view controller class:

```
imagePicker = [[UIImagePickerController alloc] init];
imagePicker.delegate = self;
imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
```

```
containerPopover = [[UIPopoverController alloc]
                    initWithContentViewController:imagePicker];
containerPopover.popoverContentSize = CGSizeMake(320.0, 480.0);
containerPopover.passthroughViews = nil;
containerPopover.delegate = nil;

[containerPopover presentPopoverFromRect:CGRectMake(295.0, 60.0, 30.0, 50.0)
                  inView:self.view
                  permittedArrowDirections:UIPopoverArrowDirectionLeft
                  animated:YES];
```

The above code snippet is by no means complete; it assumes that you have created appropriate properties called `imagePicker` and `containerPopover` in your view controller class. It also assumes that your view controller class implements certain protocols and implements popover controllers correctly. These protocols are described next. Popover controllers are covered in Lesson 19.

`UIImagePickerController` requires a delegate object that implements both the `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` protocols. The former defines two methods that are called when the user has selected an image, or selected the Cancel button in the image picker:

- `imagePickerControllerDidCancel:`
- `imagePickerController:didFinishPickingMediaWithInfo:`

The `imagePickerControllerDidCancel:` delegate method has one parameter that contains a reference to the image picker controller. A typical implementation of this delegate method dismisses the image picker controller if it was presented modally:

```
[picker dismissModalViewControllerAnimated:YES];
```

The `imagePickerController:didFinishPickingMediaWithInfo:` delegate method has two parameters, the first of which is a reference to the image picker. The second parameter is an `NSDictionary` object that contains a `UIImage` object corresponding to the selected image.

To access this image in this delegate method, you can use code similar to the following to retrieve the value in the dictionary that corresponds to the `UIImagePickerControllerOriginalImage` key:

```
UIImage* image = [info valueForKey: UIImagePickerControllerOriginalImage];
```

Often, you may want to save this `UIImage` instance to a file. To do that, you must first obtain an `NSData` instance that contains the pixels in the `UIImage` instance in a specific file-format. Once you have this `NSData` instance, you can write it to a file by sending it the `writeToFile:atomically:` message.

To obtain an `NSData` instance that contains the image as a PNG, use the `UIImagePNGRepresentation` function as follows:

```
NSData *imageData = UIImagePNGRepresentation(image);
```

To obtain an `NSData` instance that contains the image in JPEG format, use the `UIImageJPEGRepresentation` function as follows:

```
NSData *imageData = UIImageJPEGRepresentation(image, 1.0);
```

The first parameter to this function is the `UIImage` instance, and the second is a number between 0.0 and 1.0 that indicates the desired JPEG quality, with 0.0 representing the lowest quality and 1.0 the highest quality.

The following implementation of the `imagePickerController:didFinishPickingMediaWithInfo:` delegate method shows how to save the selected image to a PNG file in the Documents directory. Basic file handling is covered in Lesson 21.

```
- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage* image = [info valueForKey: UIImagePickerControllerOriginalImage];

    NSArray* paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                      NSUserDomainMask, YES);
    NSString* documentsDir = [paths objectAtIndex:0];
    NSString* outFile = [documentsDir
                         stringByAppendingPathComponent:@"savedImage.png"];

    NSData *imageData = UIImagePNGRepresentation(image);
    [imageData writeToFile:outFile atomically:YES];

    [picker dismissModalViewControllerAnimated:YES];
}
```

If the image picker was presented as a modal view controller, you will need to dismiss it yourself when the user has finished selecting an image by adding the following code to the end of the `imagePickerController:didFinishPickingMediaWithInfo:` delegate method:

```
[picker dismissModalViewControllerAnimated:YES];
```

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `ImagePicker` that allows the user to select an image from the photo library, or take a picture using the camera and display the image in an image view.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a `UIImageView` instance to the scene and an appropriate outlet in the view controller file.
- Add two `UIButton` instances to the scene and connect them to appropriate action methods in the view controller class.
- Allow the user to select an image from the photo library, and display the selected image in the image view.
- Allow the user to take a picture using the camera and display the image in the image view.
- Hide the camera button if the device does not have a camera.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 36 folder in the download.*

Hints

- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.

Step-by-Step

1. Create a Single View Application in Xcode called `ImagePicker`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** ImagePicker
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson36
        - **Define Family:** iPhone
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.
2. Add a `UIImageView` instance to the scene and connect it to an outlet in the view controller class.
    1. Open the storyboard file and use the Object library to drag and drop an image view onto the scene. Use the Size inspector to resize/position it to X=0, Y=0, W=320, H=460.
    2. Use the assistant editor to create an outlet in the view controller class and connect it to the image view. Name the outlet `imageView`.

3. Add two `UIButton` instances to the scene and connect their Touch Up Inside events to appropriate action methods in the view controller class.
    1. Set the title of the first button to `Camera` and use the Size inspector to resize/position it to X=8, Y=365, W=302, H=37.
    2. Set the title of the second button to `Photo Library` and use the Size inspector to resize/position it to X=8, Y=410, W=302, H=37.
    3. Name the action method corresponding to the first button `onCamera`.
    4. Name the action method corresponding to the second button `onPhotoLibrary`.
    5. Create an outlet called `cameraButton` in the view controller class and connect it to the button titled Camera in the scene.
4. Modify the declaration of the `Lesson36ViewController` class to resemble the following:

```
@interface Lesson36ViewController : UIViewController
                                    <UINavigationControllerDelegate,
                                    UIImagePickerControllerDelegate>
```

5. Add the following code to the implementation of the `onCamera:` method in the `Lesson36ViewController.m` file:

```
UIImagePickerController* imagePicker = [[UIImagePickerController alloc] init];
imagePicker.delegate = self;
imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;
[self presentModalViewController:imagePicker animated:YES];
```

6. Add the following code to the implementation of the `onPhotoLibrary:` method in the `Lesson36ViewController.m` file:

```
UIImagePickerController* imagePicker = [[UIImagePickerController alloc] init];
imagePicker.delegate = self;
imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
[self presentModalViewController:imagePicker animated:YES];
```

7. Implement `UIImagePickerControllerDelegate` methods in your view controller class.
    1. Add the following code in your `Lesson36ViewController.m` file to implement the `imagePickerControllerDidCancel:` delegate method:

    ```
    - (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
    {
        [picker dismissModalViewControllerAnimated:YES];
    }
    ```

    2. Add the following code in your `Lesson36ViewController.m` file to implement the `imagePickerController:didFinishPickingMediaWithInfo:` delegate method:

    ```
    - (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
    {
        UIImage* image = (UIImage *) [info valueForKey:
                                     UIImagePickerControllerOriginalImage];
        imageView.image = image;
        [picker dismissModalViewControllerAnimated:YES];
    }
    ```

8. Add the following code to the end of the `viewDidLoad` method of your view controller class:

```
BOOL hasCamera = [UIImagePickerController isSourceTypeAvailable:
                        UIImagePickerControllerSourceTypeCamera];
if (hasCamera == NO)
    [cameraButton setHidden:YES];
```

9. Test your application on an iPhone or iPod touch.

    1. Connect your device to your Mac and select it from the Scheme/Target selector in the Xcode toolbar.
    2. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.
    3. Tap the Photo Library button and select a photo from the contents of your device's photo library. Alternately, tap the Camera button to take a picture. After selecting the image, your device screen will resemble Figure 36-2.

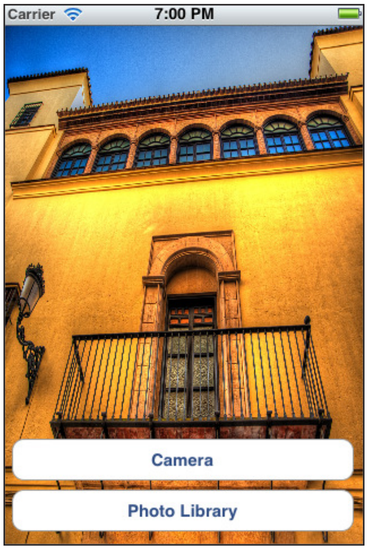


FIGURE 36-2

Please select Lesson 36 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

37 Introduction to Core Motion

Motion sensing has proven to be an effective input technique in applications, particularly games. All iOS devices have had the capability to detect motion, and most achieve this using an accelerometer. Starting with iPhone 4 and iPad 2, Apple has included a gyroscope in addition to the standard accelerometer found in its predecessors.

As far as developing motion-aware applications, accelerometer events have traditionally been available to applications through the `UIAccelerometer` class. Starting with iOS4, Apple has provided a new framework called Core Motion that exposes the functionality of both the accelerometer and the gyroscope (when available).

You can't test accelerometer and gyroscope functionality in the iOS Simulator. You will need to test any apps that require Core Motion on a real device.

ACCELEROMETERS AND GYROSCOPES

An accelerometer is a device that measures acceleration along three axes (Figure 37-1). The standard unit of acceleration is "g"(short for gravity). 1g is the force pulling down on an object that is at rest at sea level.

The only time an accelerometer will give a reading of 0g is when the device is in free fall (not recommended). Depending on how your iPhone is placed, the 1g of acceleration can be distributed differently across the three axes. An accelerometer can measure both translational acceleration and tilt (Figure 37-2).

A gyroscope, on the other hand, is a device that measures the speed with which the iOS device is spinning about the three axes (Figure 37-3). The unit of measurement in this case is radians-per-second. Two radians make a complete circle.

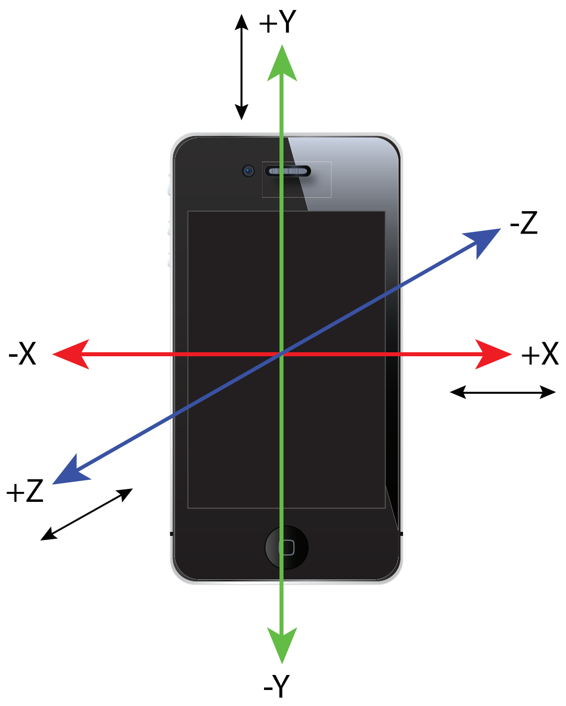


FIGURE 37-1

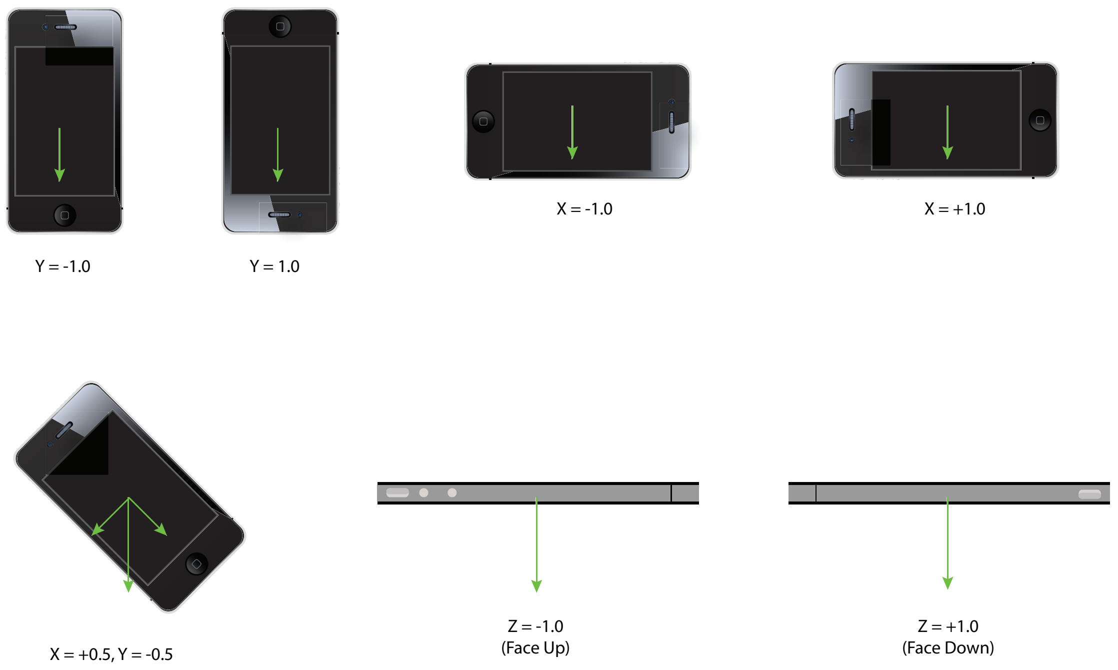


FIGURE 37-2

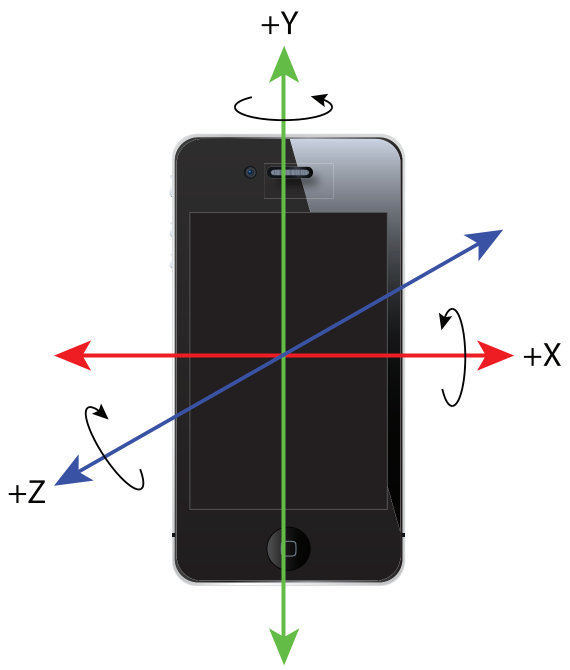


FIGURE 37-3

CORE MOTION BASICS

Core Motion is a framework that allows applications to receive data from motion sensors. This framework is not included in any of the standard iOS application templates. To use this framework in your code you need to add it manually to your project. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, switch to the Build Phases tab and click the + button under the Link Binary With Libraries category. Select `CoreMotion.framework` from the list of available frameworks (Figure 37-4).

Core Motion defines a manager class called `CMMotionManager` that you can use to interact with it. It also defines three data classes that encapsulate different types of motion data. These are described in Table 37-1.

TABLE 37-1: Core Motion Classes

CLASS	DESCRIPTION
`CMMotionManager`	A manager class that allows your apps to interact with the Core Motion framework.
`CMAccelerometerData`	Contains a measurement of device acceleration along three axes.
`CMGyroData`	Contains a biased measurement of device rotation along three axes.
`CMDeviceMotion`	Contains a combination of both accelerometer and gyroscope data.

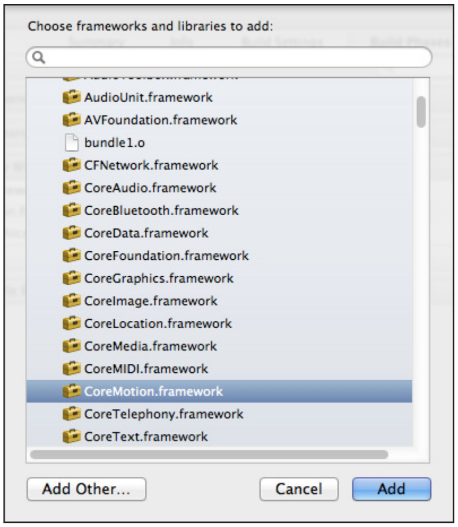


FIGURE 37-4

Each of the data classes—`CMAccelerometerData`, `CMGyroData`, and `CMDeviceMotion`—contain a timestamp that can be used by your application. For instance, you could potentially compare the timestamp between the current and previous motion events to work out the true rate at which motion data is being provided to your application.

Because the motion sensing hardware (accelerometer/gyroscope) is shared between different applications, your application should create only one instance of the `CMMotionManager` class. You can do this using the following line of code:

```
CMMotionManager* motionManager = [[CMMotionManager alloc] init];
```

An application can use the Core Motion framework in one of two modes:

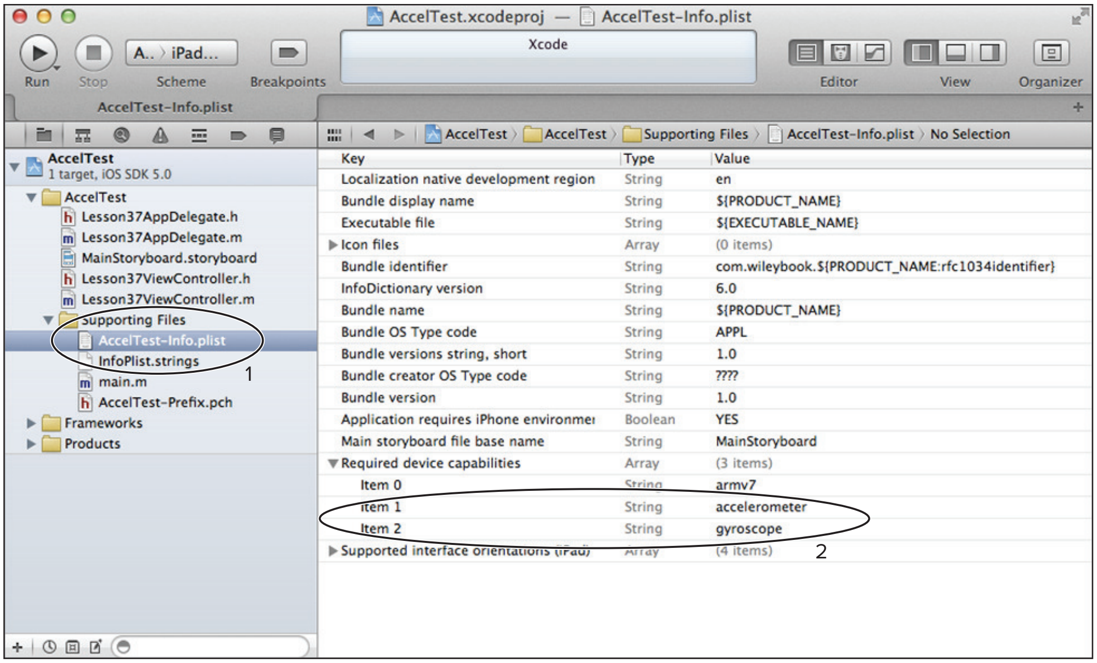
- **Push:** Your application requests an update interval (measured in seconds), and implements a block handler to process the motion data as it is available. Your application then provides both the block as well as an operation queue to the framework. Core Motion delivers each update to your block.
- **Pull:** Your application samples a property value in the `CMMotionManager` instance to read the last available acceleration/gyroscope readings. This approach is simpler to implement and requires less code. It is well suited to games that are driven by a run loop and need to poll hardware at each pass through the loop.

This lesson examines the push approach. When using this approach, your application requests motion updates at a specific frequency. The requested update interval, however, is not guaranteed.

As a developer, you will need to experiment with different update frequencies to determine what works best for your application. You may be tempted to specify a high frequency (more updates per second) than you actually need, but this can have adverse effects on overall system performance and drain the battery faster.

CHECKING HARDWARE AVAILABILITY

Because only the latest devices have gyroscopes, you need to check to see if one exists before attempting to use it. You have two ways to do this. If having a gyroscope is a key requirement for your application, and you do not want users to use your app on an older device, you can add the `UIRequiredDeviceCapabilities` key to your application's `Info.plist` file (Figure 37-5).



Key	Type	Value
Localization native development region	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	com.wileybook.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environme	Boolean	YES
Main storyboard file base name	String	MainStoryboard
▼ Required device capabilities	Array	(3 items)
Item 0	String	armv7
Item 1	String	accelerometer
Item 2	String	gyroscope
▶ Supported interface orientations (iPad)	Array	(4 items)

FIGURE 37-5

The value of this key is an array that must contain an entry for each capability that is required to run your application. The entries for motion hardware are:

- `accelerometer` (for accelerometer events)
- `gyroscope` (for motion events)

If having a gyroscope is an optional requirement, and your application can be used without one, you can test for the availability of a gyroscope using the following code:

```
// is a gyroscope available
CMMotionManager* motionManager = [[CMMotionManager alloc] init];
BOOL gyroscopeAvailable = motionManager.gyroAvailable;
```

HANDLING ACCELEROMETER EVENTS

Core Motion provides an alternate way to handle accelerometer events (the other alternative being the `UIAccelerometer` class). An accelerometer event is an instance of `CMAccelerometerData` class.

The `CMAccelerometerData` class encapsulates a `CMAcceleration` structure called *acceleration*, which in turn contains the values of acceleration along the x, y, and z axes, respectively.

The following code snippet shows how to use Core Motion to receive accelerometer updates roughly 10 times per second:

```
CMMotionManager* motionManager = [[CMMotionManager alloc] init];
motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
[motionManager startAccelerometerUpdatesToQueue:
                [NSOperationQueue currentQueue]
withHandler:^(CMAccelerometerData *accelData, NSError *error)
{
    // get acceleration values along three axes
    double accelerationX = accelData.acceleration.x;
    double accelerationY = accelData.acceleration.y;
    double accelerationZ = accelData.acceleration.z;

    // do something with the acceleration values...
}];
```

When your application does not want to receive accelerometer events any longer, it must call the `stopAccelerometerUpdates` method on the `CMMotionManager` instance:

```
[motionManager stopAccelerometerUpdates];
```

If you need to reduce the effect of sudden changes on the accelerometer data, you can modify the raw acceleration values contained in the `CMAcceleration` structure by applying a low-pass filter. The following code snippet implements a simple low-pass filter:

```
// get acceleration values along three axes
double rawX = accelData.acceleration.x;
double rawY = accelData.acceleration.y;
double rawZ = accelData.acceleration.z;

// filter the raw acceleration values using a low-pass filter
double filteredX = (rawX * 0.1) + (filteredX * 0.9);
double filteredY = (rawY * 0.1) + (filteredY * 0.9);
double filteredZ = (rawZ * 0.1) + (filteredZ * 0.9);
// do something with the filtered values...
```

The filtering is based on using 10 percent of the current, unfiltered acceleration value and 90 percent of the previous filtered value.

If you want to isolate the effect of gravity on the accelerometer data, you can modify the raw acceleration values by applying a high-pass filter as follows:

```
// get acceleration values along three axes
double rawX = accelData.acceleration.x;
double rawY = accelData.acceleration.y;
double rawZ = accelData.acceleration.z;

// filter the raw acceleration values using a high-pass filter
double filteredX = rawX - ((rawX * 0.1) + (filteredX * 0.9));
```

```
    double filteredY = rawY - ((rawY * 0.1) + (filteredY * 0.9));
    double filteredZ = rawZ - ((rawZ * 0.1) + (filteredZ * 0.9));

    // do something with the filtered values...
```

HANDLING GYROSCOPE EVENTS

A gyroscope measures the rate of rotation of a device about three axes. The gyroscope measurements provided by Core Motion are biased, meaning that the gyroscope will provide some fixed reading even where there is no change in rotation about an axis. A gyroscope event is an instance of a `CMGyroData` class.

The `CMGyroData` class encapsulates a `CMRotationRate` structure called `rotationRate`, which in turn contains the values of rotation rate along the X, Y, and Z axes, respectively. The rate of rotation is measured in radians per second.

Using Core Motion to access gyroscope data is very similar to using it with acceleration data. You need to check that the gyroscope exists before trying to access it. The following code snippet shows how to use Core Motion to receive gyroscope updates.

```
CMMotionManager* motionManager = [[CMMotionManager alloc] init];
motionManager.gyroUpdateInterval = 1.0/60.0;
if (motionManager.gyroAvailable)
{
    [motionManager startGyroUpdatesToQueue:
                  [NSOperationQueue currentQueue]
    withHandler:^(CMGyroData *rotationData, NSError *error)
    {
        // get rotation-rate values along three axes
        double rawX = rotationData.rotationRate.x;
        double rawY = rotationData.rotationRate.y;
        double rawZ = rotationData.rotationRate.z;

        // do something with the  values...
    }];
}
```

When your application does not want to receive gyroscope updates, it should send the `stopGyroUpdates` message to the `CMMotionManager` instance.

TRY IT

In this Try It, you build an iPad application called `AccelTest` based on the Single View Application template that uses Core Motion to display accelerometer and gyroscope readings and slides an image along one axis as you tilt the device.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Import image resources into the project.

- Add a background image to the default scene.
- Add a few `UILabel` elements that will display the accelerometer and gyroscope readings. Create outlets for these in the view controller class.
- Add a `UIImageView` instance to the default scene that will be moved about the screen along the X axis as the device is tilted. Create an appropriate outlet.
- Initialize Core Motion in the `viewDidLoad` method of the view controller class.
- Send appropriate methods to `CMMotionManager` to stop receiving motion updates in the `viewDidUnload` method.
- Write code to slide the `UIImageView` instance across the screen along the X axis as the device is tilted.
- The gyroscope features of this Try It require an iPad 2.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 37 folder in the download.*

Hints

- Remember to check for gyroscope availability before attempting to use it.
- You need to add a reference to the Core Motion framework to the project.

Step-by-Step

1. Create a Single View Application in Xcode called `AccelTest`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.
        - **Product Name:** AccelTest
        - **Company Identifier:** com.wileybook
        - **Class Prefix:** Lesson37
        - **Define Family:** iPad
        - **Use Storyboard:** Checked
        - **Use Automatic Reference Counting:** Checked
        - **Include Unit Tests:** Unchecked

> *For Company Identifier, we used* `com.wileybook`*, but you can use any unique identifier for your application.*

    5. Select a folder where this project should be created.
    6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
    7. Click Create.

2. Add a reference to the Core Motion framework.
    1. In Xcode, make sure the project navigator is visible. To show it, use the View ➪ Navigators ➪ Show Project Navigator menu item.
    2. Click the root (project) node of the project navigator to display project settings.
    3. Select the AccelTest target and ensure the Build Phases tab is visible.
    4. Expand the Link Binary With Libraries group in this tab.
    5. Click the + button at the bottom of this group and select `CoreMotion.framework` from the list of available frameworks.
    6. Click the Add button.
3. Disable landscape orientation for iPad applications.
    1. Switch to the Summary tab in project settings.
    2. Deselect the Landscape Left and Landscape Right orientations from the Supported Device Orientations group.
4. Copy the `gameBack.png` and `00.png` files from the resources folder of this lesson on the DVD into the Xcode project.
5. Add a background image to the view.
    1. Open the `MainStoryboard.storyboard` file in Interface Builder.
    2. Ensure the Media library is visible. You can show it by using the View ➪ Utilities ➪ Show Media Library menu item.
    3. Drag the `gameBack.png` file onto the view controller.
    4. Use the Size inspector to set the origin to X = 0 and Y = 0.
6. Add an image to the view that will be moved as the device is tilted.
    1. Drag the `00.png` file from the Media library onto the view controller.
    2. Use the Size inspector to position it at X = 267 and Y = 643.
    3. Using the assistant editor, create an outlet called `playerImage` in the `Lesson37ViewController.h` file and connect it to the image.

7. Add three `UILabel` instances to the view controller to display accelerometer readings.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object Library menu item.
    2. Use the Object library to add three `UILabel` instances. Resize and position them using the Size Inspector as per Table 37-2.

TABLE 37-2: Size and Position of Accelerometer Labels

LABEL	X	Y	WIDTH	HEIGHT
First Label	20	13	70	42
Second Label	90	13	70	42
Third Label	160	13	70	42

    3. Using the Assistant Editor, create outlets for each label named `accelX`, `accelY`, and `accelZ`, respectively, in the `Lesson37ViewController.h` file
8. Add three more `UILabel` instances to the view controller to display gyroscope readings.
    1. Ensure the Object library is visible. You can show it by using the View ➪ Utilities ➪ Show Object library menu item.
    2. Use the Object library to add three `UILabel` instances. Resize and position them using the size inspector as per Table 37-3.

TABLE 37-3: Size and Position of Gyroscope Labels

LABEL	X	Y	WIDTH	HEIGHT
First Label	543	13	70	42
Second Label	613	13	70	42
Third Label	683	13	70	42

    3. Using the assistant editor, create outlets for each label named `gyroX`, `gyroY`, and `gyroZ`, respectively.
9. Add the following line to the top of the `Lesson37ViewController.h` file, to import the Core Motion framework:

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

10. Add the following property declarations to the `Lesson37ViewController.h` file:

```
@property double filteredX;
@property double filteredY;
@property double filteredZ;
@property double xVelocity;
@property (strong, nonatomic) CMMotionManager* motionManager;
```

11. Add the following `@synthesize` statements to the `Lesson37ViewController.m` file:

```
@synthesize filteredX;
@synthesize filteredY;
@synthesize filteredZ;
@synthesize xVelocity;
@synthesize motionManager;
```

12. Add the following code to the `viewDidLoad` method of the view controller class after the `[super viewDidLoad]` line:

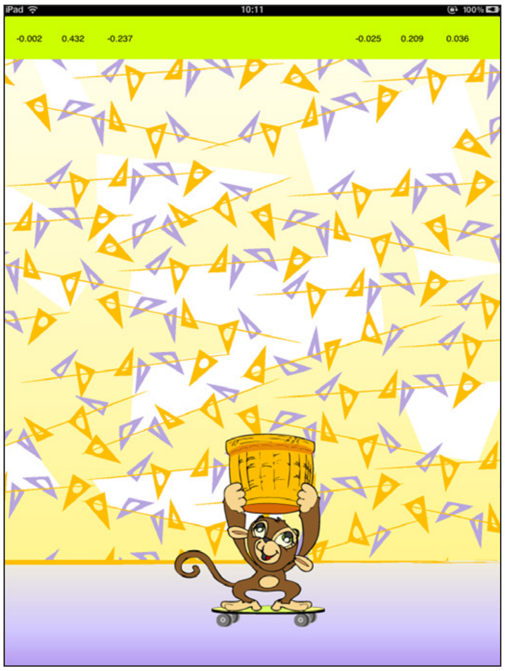
```
// instantiate CMMotionManager
self.motionManager = [[CMMotionManager alloc] init];

// initialize horizontal velocity
xVelocity = 0.0;

// set up to receive acceleration
self.motionManager.accelerometerUpdateInterval = 1.0 / 60.0;
[self.motionManager startAccelerometerUpdatesToQueue:
                    [NSOperationQueue currentQueue]
      withHandler:^(CMAccelerometerData *accelData, NSError *error)
      {
          // get acceleration values along three axes
          double rawX = accelData.acceleration.x;
          double rawY = accelData.acceleration.y;
          double rawZ = accelData.acceleration.z;

          // filter the raw acceleration values using
          // a high-pass filter
          filteredX = rawX - ((rawX * 0.1) +
                      (filteredX * 0.9));
          filteredY = rawY - ((rawY * 0.1) +
                      (filteredY * 0.9));
          filteredZ = rawZ - ((rawZ * 0.1) +
                      (filteredZ * 0.9));

          // display the values
         accelX.text = [NSString stringWithFormat:@"%2.3f",
                                 filteredX];
         accelY.text = [NSString stringWithFormat:@"%2.3f",
                                 filteredY];
         accelZ.text = [NSString stringWithFormat:@"%2.3f",
                                 filteredZ];

         // slide playerImage along the X axis
         xVelocity = xVelocity + filteredX;
         float newPlayerX = playerImage.frame.origin.x +
                                         xVelocity;
         // clamp new position between [20.0, 700.0]
         if (newPlayerX <= 20.0)
         {
              newPlayerX = 20.0;
              xVelocity = 0.0;
         }

         if (newPlayerX > 700.0)
         {
             newPlayerX = 700.0;
             xVelocity = 0.0;
         }

         playerImage.frame = CGRectMake(newPlayerX,
                                        playerImage.frame.origin.y,
                                        playerImage.frame.size.width,
                                        playerImage.frame.size.height);
      }];


    if (self.motionManager.gyroAvailable)
    {
        self.motionManager.gyroUpdateInterval = 1.0/60.0;

        [self.motionManager startGyroUpdatesToQueue:
                                [NSOperationQueue currentQueue]
             withHandler:^(CMGyroData *rotationData, NSError *error)
        {
            // get rotation-rate values along three axes
            double rawX = rotationData.rotationRate.x;
            double rawY = rotationData.rotationRate.y;
            double rawZ = rotationData.rotationRate.z;

             // display the values
             gyroX.text = [NSString stringWithFormat:@"%2.3f",
                                    rawX];
             gyroY.text = [NSString stringWithFormat:@"%2.3f",
                                    rawY];
             gyroZ.text = [NSString stringWithFormat:@"%2.3f",
                                    rawZ];
        }];
    }
    else
    {
        gyroX.text = @"--";
        gyroY.text = @"--";
        gyroZ.text = @"--";

        UIAlertView* noGyroscope = [[UIAlertView alloc]
                                    initWithTitle:@""
                                    message:@"No gyroscope detected!"
                                    delegate:nil
                                    cancelButtonTitle:@"Ok"
                                    otherButtonTitles:nil];
        [noGyroscope show];
    }
```

13. Add the following code to the end of the `viewDidUnload` method of the view controller class:

```
if (self.motionManager != nil)
{
    [self.motionManager stopAccelerometerUpdates];
    if (self.motionManager.gyroAvailable)
    {
        [self.motionManager stopGyroUpdates];
    }
}
```

14. Test your application on an iPad.

1. Connect your iPad and select it from the Scheme/Target selector in the Xcode toolbar.
2. Click the Run button in the Xcode toolbar. Alternatively you can use the Project ➪ Run menu item.
3. Tilt the iPad along its X axis to move the player character. Observe the accelerometer and gyroscope readings displayed on the top of the screen (Figure 37-6).



```
-0.002   0.432   -0.237                          -0.025   0.209   0.036
```

FIGURE 37-6

Please select Lesson 37 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

38 Building Background-Aware Applications

Starting with iOS 4, Apple included support for background applications. If you have been building the applications in the Try It sections at the end of each lesson, you may have noticed that when you quit your application and launch it again from the home screen, your application resumes from where you left off. The reason for it is that projects created with Xcode 4.2 and above are background-ready by default.

Being background-ready does not mean that your application will run in the background. It just means that your application is aware of the support for background features in iOS and can take advantage of these features. In this lesson you learn how to create applications that can perform limited functions whilst in background mode.

UNDERSTANDING BACKGROUND SUSPENSION

To support background execution, an application must be compiled against the iOS 4 SDK or higher. This means that if you purchase an application from the App Store that was created using an older version of the iOS SDK, the application will simply enter a suspended state of execution when you press the home button and will not be able to perform any background processing.

When a user quits a background-aware application by pressing the home button, the application is usually moved into a suspended state. Most applications cease to execute any code in this suspended state. Instead, these applications are preserved exactly as the user left them. When the user returns to the suspended application, it appears as if it has been running the whole time.

iOS calls several methods in your application delegate object as the application is being suspended or resumed. Typically you are responsible for cleaning up resources as the application enters the suspended state and for updating things in the application that should have changed when the application resumes.

You will typically find the following application delegate methods in a background-aware application. Default implementations of these methods are provided as part of Apple's application templates.

- `(BOOL)application didFinishLaunchingWithOptions`: Called when your application is first launched, or launched after it was manually terminated using the iOS Task Manager.
- `(void)applicationDidBecomeActive`: Called when an application returns to the foreground from the background.
- `(void)applicationWillResignActive`: Called when your application is about to become inactive. This can happen if the application is being temporarily interrupted by a phone call/SMS or is about to begin its transition to the background state.
- `(void)applicationDidEnterBackground`: Called when your application has become a background application.
- `(void)applicationWillEnterForeground`: Called when the application is in the process of becoming active from the background state.
- `(void)applicationWillTerminate`: Called when the application is about to terminate from the background state. This happens when the user decides to terminate it from the task manager or the operating system is running low on resources and terminates your application.

In most cases, you will save your application state in the `applicationDidEnterBackground:` event and restore its state in the `applicationDidBecomeActive:` event.

By default, applications created with Xcode will go into the suspended state when they are closed by the user. To override this behavior and have the application terminate instead, add the `Application Does Not Run In Background` key to the project's `Info.plist` file and set the value of the key to `YES` (Figure 38-1).

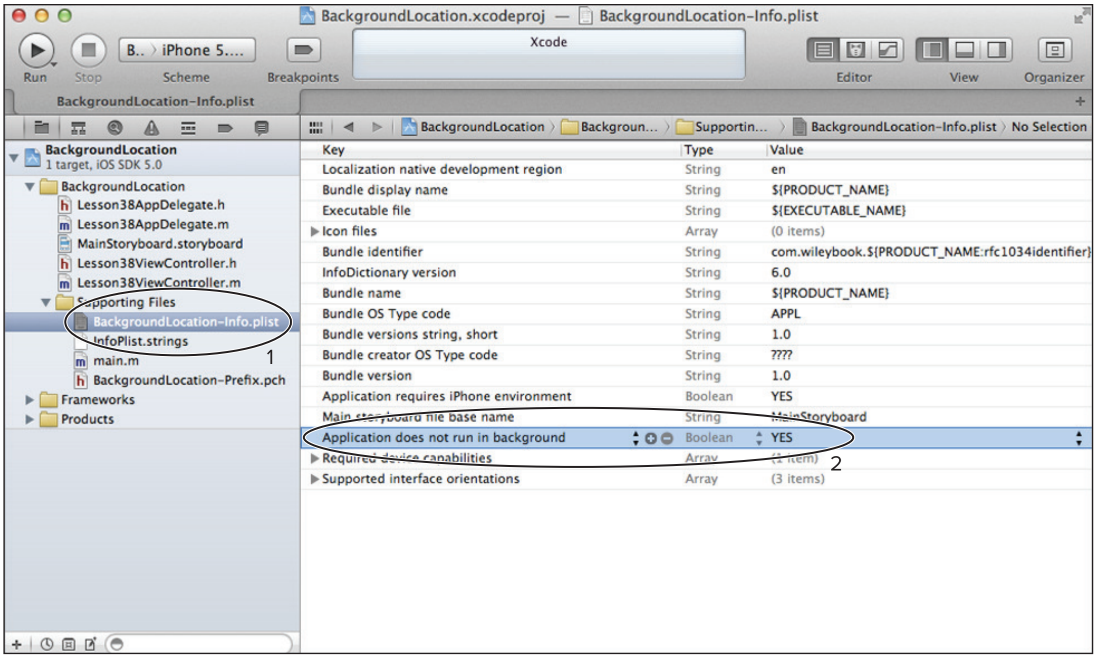
Most devices support background-aware applications; however, some older devices like the iPhone 3G may not. Use the following code snippet to determine if the device on which your application is running supports background-aware applications:

```
UIDevice *device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
    backgroundSupported = device.multitaskingSupported;
```

EXECUTING BACKGROUND CODE

Unfortunately, only applications that perform certain types of tasks can execute code when in background mode. All other applications are sent to the suspended state, and while in that state cannot execute any code. The types of applications that can execute code in the background are:

- Audio players
- Voice over IP applications
- Applications that require location updates


BackgroundLocation.xcodeproj — BackgroundLocation-Info.plist
Run
Stop
B.. 〉 iPhone 5....
Scheme
Breakpoints
Xcode
Editor
View
Organizer
BackgroundLocation-Info.plist
BackgroundLocation 〉 Backgroun... 〉 Supportin... 〉 BackgroundLocation-Info.plist 〉 No Selection
1
2

BackgroundLocation
1 target, iOS SDK 5.0
- BackgroundLocation
  - Lesson38AppDelegate.h
  - Lesson38AppDelegate.m
  - MainStoryboard.storyboard
  - Lesson38ViewController.h
  - Lesson38ViewController.m
  - Supporting Files
    - BackgroundLocation-Info.plist
    - InfoPlist.strings
    - main.m
    - BackgroundLocation-Prefix.pch
- Frameworks
- Products

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	com.wileybook.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Main storyboard file base name	String	MainStoryboard
Application does not run in background	Boolean	YES
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)

FIGURE 38-1

To declare your application as supporting one or more of these tasks, you need to add the `Required Background Modes` key to the project's `Info.plist` file and then add the values `App Plays Audio`, `App Provides Voice over IP Services`, and `App Registers for Location Updates`, respectively (Figure 38-2). In the Try It section that accompanies this lesson, you create a background-aware location-based application.

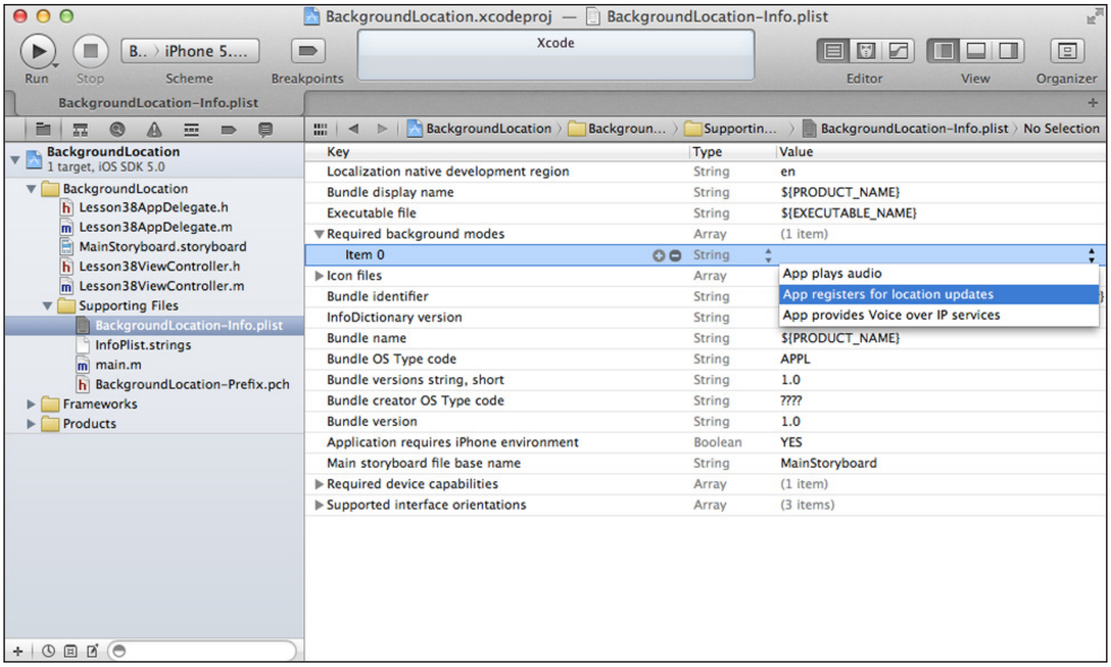
If your application has certain long-running tasks that need to be completed before the application is suspended, it can mark these tasks appropriately. Each marked task is given approximately 10 minutes to complete its actions before the application is suspended.

To mark the beginning of a long-running task, use the application's `beginBackgroundTaskWith-ExpirationHandler:` method. This method returns a task identifier and takes as an argument an Objective-C block that is called if your task does not complete in the 10-minute timeframe:

```
UIBackgroundTaskIdentifier longTask;
longTask = [[UIApplication sharedApplication]
             beginBackgroundTaskWithExpirationHandler:^{
            // write code here to handle the case that
            // your task did not complete in 10 minutes.
}];
```

Then, mark the end of the task by using the `endBackgroundTask:` method, giving it the task identifier you obtained in the previous step:

```
[[UIApplication sharedApplication] endBackgroundTask:longTask];
```


BackgroundLocation.xcodeproj — BackgroundLocation-Info.plist
Run
Stop
B.. › iPhone 5....
Scheme
Breakpoints
Xcode
Editor
View
Organizer
BackgroundLocation-Info.plist
BackgroundLocation › Backgroun... › Supportin... › BackgroundLocation-Info.plist › No Selection
BackgroundLocation
1 target, iOS SDK 5.0
BackgroundLocation
Lesson38AppDelegate.h
Lesson38AppDelegate.m
MainStoryboard.storyboard
Lesson38ViewController.h
Lesson38ViewController.m
Supporting Files
BackgroundLocation-Info.plist
InfoPlist.strings
main.m
BackgroundLocation-Prefix.pch
Frameworks
Products
App plays audio
App registers for location updates
App provides Voice over IP services

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
▼Required background modes	Array	(1 item)
Item 0	String	
▶Icon files	Array	
Bundle identifier	String	
InfoDictionary version	String	
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Main storyboard file base name	String	MainStoryboard
▶Required device capabilities	Array	(1 item)
▶Supported interface orientations	Array	(3 items)

FIGURE 38-2

CREATING LOCAL NOTIFICATIONS

All background-aware applications can receive both remote and local notifications. Remote notifications are notifications that originate from the Apple Push Notification Service and are not covered in this book. A local notification is similar in many respects to a remote notification, except that the notification is scheduled by iOS on the same device. A typical example where local notifications can be used is a to-do list application that allows the user to set a reminder for a future date.

Local notifications are instances of the `UILocalNotification` class and are created as follows:

```
UILocalNotification* futureAlert = [[UILocalNotification alloc] init];
futureAlert.alertBody = @"I need attention!";
futureAlert.alertAction = @"View ";
futureAlert.applicationIconBadgeNumber = 1;
futureAlert.fireDate = [NSDate dateWithTimeIntervalSinceNow:10];
```

When a local notification fires and your application is in the suspended state, iOS displays an alert view similar to the one shown in Figure 38-3. The `alertBody` property contains a string that is displayed in an alert view, and the `alertAction` property contains a string that represents the caption of the OK button in the notification alert.

The `fireDate` property of the local notification object is set to an `NSDate` instance that represents a point of time in the future when you want this notification to fire. To create an `NSDate` instance that refers to a point of time 10 seconds from now, use the following code:

```
[NSDate dateWithTimeIntervalSinceNow:10];
```

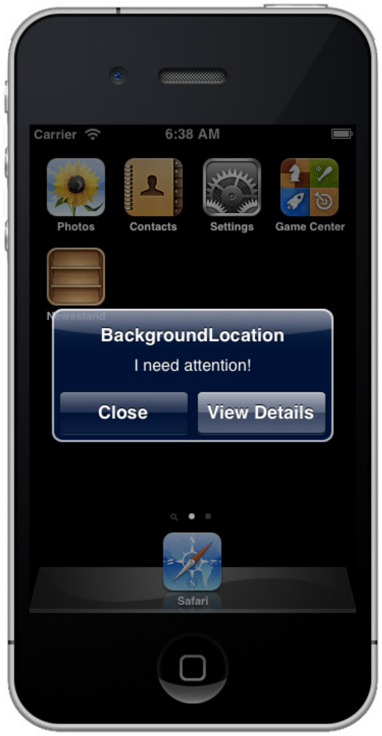


FIGURE 38-3

You can specify a number to display over the application icon in the home screen by using the `applicationIconBadgeNumber` property. You can also play a sound when the notification fires by specifying the filename of a sound resource that is part of your application bundle for the `soundName` property. To play the default system sound, set the `soundName` property to `UILocalNotificationDefaultSoundName`.

To schedule the local notification, use the `scheduleLocalNotification:` method of the application object as follows:

```
[[UIApplication sharedApplication] scheduleLocalNotification:futureAlert];
```

When the notification fires, and the user decides to activate your application by tapping on the appropriate button in the notification alert, your application delegate's `application:didReceiveLocalNotification:` method is called. This method is also called when the local notification fires while your application is active; however in this case the system will not show the notification alert view.

TRY IT

In this Try It, you build an iPhone application called `BackgroundLocation` based on the Single View Application template that uses Core Location to receive location updates. The application displays the current location, the number of location updates processed, and the distance traveled since the first location reading was obtained. When you quit the application, it will continue to execute in the background and update the current location and distance traveled.

Lesson Requirements

- Create a new project based on the Single View Application template.
- Add a few `UILabel` elements to the default scene that will display the location readings. Create outlets for these in the view controller class.
- Add a `UIButton` that will be used to stop/start receiving location updates. Create an appropriate outlet and action.
- Initialize Core Location when the button is pressed. Stop receiving location updates when the button is pressed a second time.
- Implement `CLLocationManagerDelegate` methods.
- Send appropriate methods to `CLLocationManager` to stop receiving location updates in the `viewDidUnload` method.
- Add the `Required Background Modes` key to the application's `Info.plist` file so that the application can receive location updates while it is in the background state.

You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson 38 folder in the download.*

Hints

- Remember to use the `purpose` property of the `CLLocationManager` instance to provide a description of what your application intends to do with the location data.
- Your application should send the `stopUpdatingLocation` message to the location manager when it does not require location updates.
- You need to add a reference to the Core Location framework to the project.
- This application can execute code in the background only if the `Required Background Modes` key in the `Info.plist` file is set to `App Registers for Location Updates`.

Step-by-Step

1. Create a Single View Application in Xcode called `BackgroundLocation`.
    1. Launch Xcode.
    2. To create a new project, select the File ➪ New ➪ New Project menu item.
    3. Choose the Single View Application template and click Next.
    4. Use the following information in the project options dialog box and click Next.

- **Product Name:** BackgroundLocation
- **Company Identifier:** com.wileybook
- **Class Prefix:** Lesson38
- **Define Family:** iPhone
- **Use Storyboard:** Checked
- **Use Automatic Reference Counting:** Checked
- **Include Unit Tests:** Unchecked

For Company Identifier, we used `com.wileybook`*, but you can use any unique identifier for your application.*

5. Select a folder where this project should be created.
6. Ensure the Create Local Git Repository for This Project checkbox is not selected.
7. Click Create.

2. Add a reference to the Core Location framework.
   1. In Xcode, make sure the project navigator is visible. To show it, use the View ➪ Navigators ➪ Show Project Navigator menu item.
   2. Click the root (project) node of the project navigator to display project settings.
   3. Select the `BackgroundLocation` target and ensure the Build Phases tab is visible.
   4. Expand the Link Binary With Libraries group in this tab.
   5. Click the + button at the bottom of this group and select `CoreLocation.framework` from the list of available frameworks.
   6. Click the Add button.
3. Add eight `UILabel` instances to the view controller to display location readings.
   1. Use the Object library to add eight `UILabel` instances to the default scene.
   2. Use the data in Table 38-1 to name and position these labels.

TABLE 38-1: Size and Position of Labels

LABEL	X	Y	WIDTH	HEIGHT
`Latitude`	17	34	62	21
`Longitude`	17	79	77	21
`Distance Travelled`	17	123	141	21

continues

TABLE 38-1 *(continued)*

LABEL	X	Y	WIDTH	HEIGHT
`Update Count`	17	169	104	21
`latitudeValue`	180	34	99	21
`longitudeValue`	180	79	113	21
`distanceValue`	180	123	107	21
`countValue`	180	169	86	21

3. Using the assistant editor, create outlets for the `latitudeValue`, `longitudeValue`, `distanceValue`, and `countValue` labels in the view controller class. Call these outlets `latValue`, `lonValue`, `distValue`, and `countValue`, respectively.

4. Add a `UIButton` instance to the scene.
    1. Use the Object library to add a `UIButton` instance. Resize and position it to X = 17, Y = 236, W = 275, H = 37.
    2. Using the Attributes inspector, set its `Title` for the `Default` state to Start Location Updates.
    3. Using the assistant editor, create an outlet in the view controller class called `toggleButton` and connect it to the button.
    4. Using the assistant editor, create an action in the view controller class called `onButtonPressed` and connect it to the Touch Up Inside event of the button.
    5. Your storyboard should now resemble Figure 38-4.

5. Add the following line to the top of the view controller's header file, to import the `Core Location` framework:

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

6. Add the following property declaration statements to the interface of the view controller class:

```
@property double totalDistanceTravelled;
@property long numberOfUpdatesProcessed;
@property BOOL hasInitialized;
@property BOOL hasStarted;
@property (strong, nonatomic) CLLocationManager* locationManager;
```

7. Declare the `Lesson38ViewController` class to conform to the `CLLocationManagerDelegate` protocol by modifying its interface declaration to the following:

```
@interface Lesson38ViewController : UIViewController <CLLocationManagerDelegate>
```

8. Add the following `@synthesize` statements to the `Lesson38ViewController.m` file:

```
@synthesize totalDistanceTravelled;
@synthesize numberOfUpdatesProcessed;
@synthesize hasInitialized;
@synthesize hasStarted;
@synthesize locationManager;
```

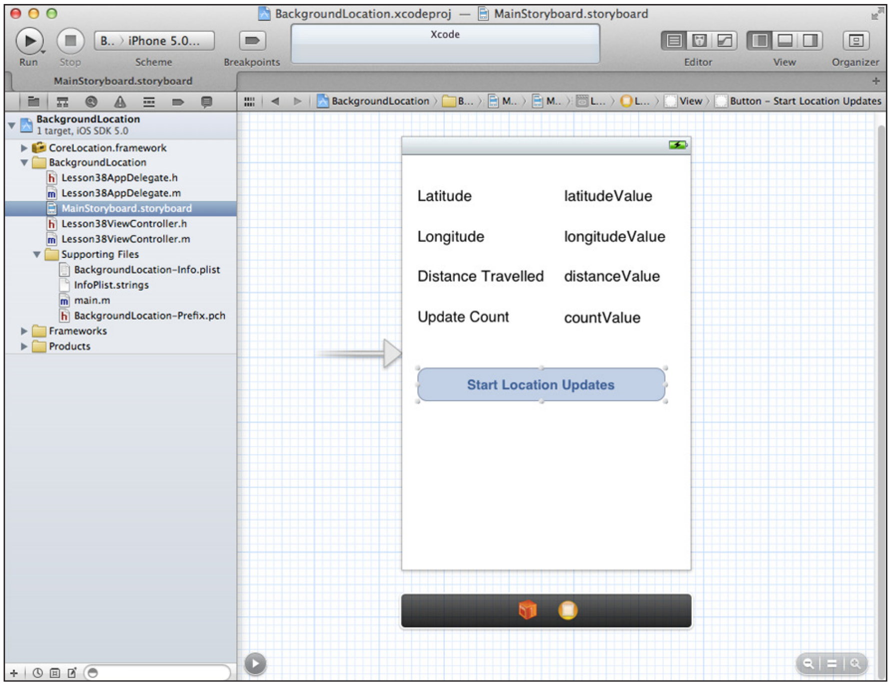


FIGURE 38-4

9. Add the following code to the `viewDidLoad` method of the view controller class after the `[super viewDidLoad]` line:

```
// setup Core Location
hasInitialized = YES;
self.locationManager = [[CLLocationManager alloc] init];
self.locationManager.delegate = self;
self.locationManager.purpose = @"This application will display your
                                 current location and the distance
                                 travelled since the first reading.";

// setup button
hasStarted = NO;
[toggleButton setTitle:@"Start Location Updates"
              forState:UIControlStateNormal];

// initialize number of updates processed
numberOfUpdatesProcessed = 0;

// initialize distance travelled
totalDistanceTravelled = 0;
distValue.text = [NSString stringWithFormat:@"%2.3f",
                  totalDistanceTravelled];
countValue.text = [NSString stringWithFormat:@"%d",
                  numberOfUpdatesProcessed];
```

10. Add the following code to the `viewDidUnload` method of the view controller class:

```
if (hasInitialized == YES)
        [self.locationManager stopUpdatingLocation];
```

11. Add the following code to the `onButtonPressed:` method of the view controller class:

```
// do not do anything if location manager is not setup
if (hasInitialized == NO)
    return;

// start
if (hasStarted == NO)
{
    [toggleButton setTitle:@"Stop Location Updates"
                  forState:UIControlStateNormal];
    [self.locationManager startUpdatingLocation];
    hasStarted = YES;
    return;
}

// stop
else if (hasStarted == YES)
{
    [toggleButton setTitle:@"Start Location Updates"
                  forState:UIControlStateNormal];
    [self.locationManager stopUpdatingLocation];
    hasStarted = NO;
    return;
}
```

12. Implement the `locationManager:didFailWithError:` delegate method as follows:

```
- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
    if (error.code == kCLErrorDenied)
    {
        hasInitialized = NO;
        [self.locationManager stopUpdatingLocation];
    }
}
```

13. Implement the `locationManager:didUpdateToLocation:FromLocation:` delegate method as follows:

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
           fromLocation:(CLLocation *)oldLocation
{
    // lat/lon values should only be considered if
    // horizontalAccuracy is not negative.
    if (newLocation.horizontalAccuracy >= 0)
    {
        CLLocationDegrees currentLatitude = newLocation.coordinate.latitude;
        CLLocationDegrees currentLongitude =
                                   newLocation.coordinate.longitude;
```

```
            // must have more than one update to compute distance travelled.
            if (numberOfUpdatesProcessed > 0)
            {
                CLLocationDistance distanceTravelled = [oldLocation
                                                  distanceFromLocation:newLocation];
                totalDistanceTravelled += distanceTravelled;
            }

            // increment numberOfUpdatesProcessed
            numberOfUpdatesProcessed++;

            // update UI
            latValue.text   = [NSString stringWithFormat:@"%2.3f",
                               currentLatitude];
            lonValue.text   = [NSString stringWithFormat:@"%2.3f",
                               currentLongitude];
            distValue.text  = [NSString stringWithFormat:@"%2.3f",
                               totalDistanceTravelled];
            countValue.text = [NSString stringWithFormat:@"%d",
                               numberOfUpdatesProcessed];
        }
    }
```

14. Add the `Required Background Modes` key to the application's `Info.plist` file.

   1. This is straightforward. Set the value of the key to `App registers for location updates`.

15. Test your application in the iOS Simulator.

   1. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.

   2. When the app is running in the simulator, click the Start Location Updates button.

   3. Use the iOS Simulator's ability to simulate a device on the move by selecting the Debug ➪ Location ➪ City Bicycle Ride menu item.

   4. Press the home button to quit the application, wait for about 10 seconds, and then launch the application by tapping its icon in the home screen. Note how the application continued to process location updates while it was in background mode.

Please select Lesson 38 on the DVD that accompanies the print book, or go to `www.wrox.com/go/iphoneipadappvideo`*, to view the video that accompanies this lesson.*

A

What's on the DVD?

This appendix provides you with information on the contents of the DVD that accompanies the print book. For the most up-to-date information please refer to the ReadMe file located at the root of the DVD. Here is what you will find in this appendix:

- Using the DVD
- What's on the DVD
- Troubleshooting

SYSTEM REQUIREMENTS

Most reasonably up-to-date computers with a DVD drive should be able to play the screencasts that are included on the DVD. You may also find an Internet connection helpful for downloading updates to this book. To follow the exercises covered in the screencasts you will need to have the iOS 5 SDK installed on your Mac. Instructions on downloading and installing the iOS 5 SDK are provided in Lesson 1.

USING THE DVD ON A PC

To access the content from the DVD, follow these steps:

1. Insert the DVD into your computer's DVD-ROM drive. The license agreement will appear.

The interface won't launch if you have autorun disabled. In that case click Start ➪ Run (For Windows Vista, Start ➪ All Programs ➪ Accessories ➪ Run). In the dialog box that appears, type **`D:\Start.exe`**. *(Replace D with the proper letter if your DVD drive uses a different letter. If you don't know the letter, see how your CD drive is listed under My Computer.) Click OK.*

2. Read through the license agreement, and click the Accept button if you want to use the DVD.
3. The DVD interface appears. Simply select the lesson video you want to view.

USING THE DVD ON A MAC

To install the items from the DVD to your hard drive, follow these steps:

1. Insert the DVD into your computer's DVD-ROM drive.
2. The DVD icon will appear on your desktop; double-click to open.
3. Double-click the Start button.
4. Read the license agreement and click the Accept button to use the DVD.
5. The DVD interface will appear. Here you can install the programs and run the demos.

WHAT'S ON THE DVD

Most lessons in the book have a corresponding screencast that illustrates examples in the lesson and provides content beyond what is covered in print.

We recommend using the following steps when reading a lesson:

1. Read the lesson's text.
2. Read the step-by-step instructions in the lesson's "Try It" section.
3. Follow these instructions to make the code sample work on your computer.
4. Watch the screencast.

You can also download all the solutions to the "Try It" sections at the book's website. If you get stuck and don't know what to do next, visit the p2p forums (`p2p.wrox.com`), locate the forum for the book, and leave a post. You can also e-mail us at `amishra@asmtechnology.com` and `gbacklin@marizack.com` and we'll try to point you in the right direction.

TROUBLESHOOTING

If you have difficulty installing or using any of the materials on the companion DVD, try the following solutions:

- **Turn off any antivirus software that you may have running:** Installers sometimes mimic virus activity and can make your computer incorrectly believe that it is being attacked by a virus. (Be sure to turn the antivirus software back on later.)

- **Close all running programs:** The more programs you're running, the less memory is available to other programs. Installers also typically update files and programs; if you keep other programs running, installation may not work properly.
- **Reference the ReadMe:** Please refer to the ReadMe file located at the root of the CD-ROM for the latest product information as of publication time.
- **Reboot if necessary:** If all else fails, rebooting your machine can often clear any conflicts in the system.

CUSTOMER CARE

If you have trouble with the CD-ROM please call the Wiley Product Technical Support phone number, (800) 762-2974. Outside the United States call 1 (317) 572-3994. You can also contact Wiley Product Technical Support at `http://support.wiley.com`. John Wiley & Sons will provide technical support only for installation and other general quality-control issues. For technical support on the applications themselves, consult the program's vendor or author.

To place additional orders or to request information about other Wiley products, please call (877) 762-2974.

INDEX

B

Introduction to Programming with Objective-C

In this appendix, you learn some of the key concepts of computer programming and the fundamentals of the Objective-C language. Objective-C is the language of choice when it comes to iOS programming. It is an object-oriented language and was invented in the early ’80s by Brad Cox.

Essentially, Objective-C is an extension of the C language designed to give it object-oriented capabilities by bringing in concepts from another popular programming language in the ’80s—Smalltalk. Objective-C development coincided with the development of the popular C++ language, thus Objective-C and C++ share many common concepts. A brief timeline of the Objective-C language is shown in Figure B-1.

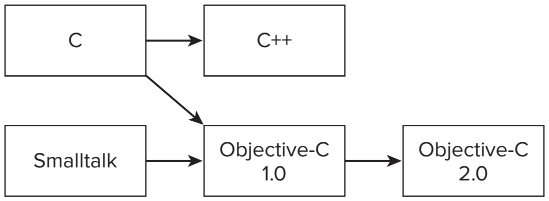


FIGURE B-1

FUNDAMENTAL PROGRAMMING CONCEPTS

To create anything more than the simplest of iOS applications requires you to know how to write programs. A program is a set of instructions to the device to carry out a specific task, and these instructions are specified using a programming language.

Fundamentally, computers are electronic devices, and only understand a language of 1s and 0s (known as *binary language*). In the early days of computer programming, programmers would have to provide their instructions in this binary language—a process that was both tedious and error prone.

As time went by, sequences of binary digits were given three-character names (like MOV and ADD) to create a higher-level language called assembly language. Programs were written using these names, and then converted to binary language before being given to the computer. These higher-level languages were easier to use by programmers.

Over the decades, several high-level, verbose languages were developed, each getting closer to instructions that began to resemble words we use in our everyday lives (like if, while, do, and return). Today almost all programming is done in one high-level language or the other (such as C, C++, Java, C#, or Objective-C). Creating an application requires a few common steps, regardless of the language in which you write your code. Sometimes one or more of these steps are combined into a single step.

Typing Your Program

The first step in creating an application is typing your code in a suitable text editor. This code that you type is essentially a series of instructions to the iOS device to perform certain tasks, and is known as *source code*.

Compiling

Objective-C is a high-level programming language. This means that the computer cannot directly understand Objective-C. A computer requires digital/binary instructions (also known as *binary language*, or *machine language*), and before your Objective-C code can run on an iOS device, it needs to be converted into machine language.

This translation (from Objective-C to machine language) is performed by an application called a *compiler*, and the process is known as *compilation*. A compiler (Figure B-2) basically takes as input your Objective-C source code and produces another file with machine language instructions (usually with an `.obj` extension).

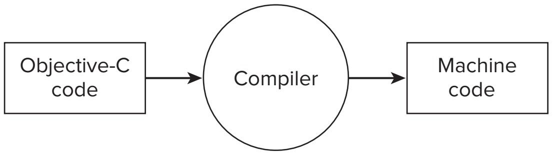


FIGURE B-2

Linking to Create an Executable

In an ideal world, you would write all the code that is required by your application to carry out its intended purpose, and in the early days of computer programming, this was how things were done. Modern application development is all about reusing code. Most of the time you will use code that has been written by someone else and you have to build on it to create your own application.

In fact, much of iOS development involves using functionality provided by code that is written by Apple engineers. Entire libraries of such code (known as *frameworks*) are installed on your hard disk when you install the iOS SDK in ready-to-use machine language form.

The final step after your code is compiled into machine language instructions is to link this compiled code with other machine language code that it depends on (Figure B-3). This process is called *linking* and the end result of this process is an executable file (that is, something that you can actually install and use on an iOS device). This executable file is also known as the *application binary*. Linking is performed by a special application called a *linker*.

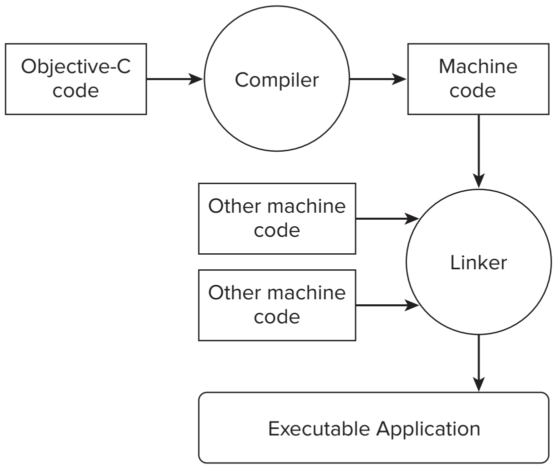


FIGURE B-3

Testing and Debugging

The testing and debugging phase of development happens after an executable has been created, and you realize that it does not do things the way you intended. An infinite number of reasons exist as to why your application does not run as expected; these faults are often called *bugs*. The most common reason for a bug in the application is poorly written source code.

The process of testing an application reveals these bugs, and the process of fixing them is called *debugging*. Debugging typically involves changing some of the source code, then compiling, linking to build a new executable, and finally testing the new executable to verify that the bug is no longer present (Figure B-4).

How Xcode Fits Into This Picture

Xcode brings together a powerful text editor, compiler, linker, and debugger into a single application. In fact, each time you click the Run button on the Xcode toolbar, you are invoking a compiler, and then a linker!

VARIABLES, STATEMENTS, AND EXPRESSIONS

The applications that you create will need to work with different types of data. This data could be anything—coordinates of a place on a map, names and addresses of your friends, and so on. In order for your application to do something useful with this data, it first needs to store it somewhere.

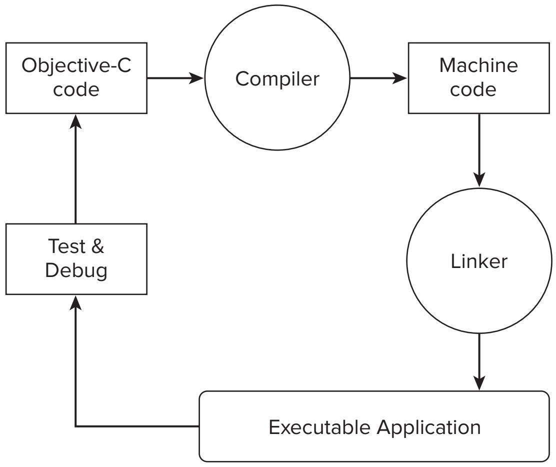


FIGURE B-4

Fundamentally two kinds of data exist—numbers (like height, weight, distance traveled) and characters (names of people, places). Your application will need a strategy to deal with either type.

Your iOS device uses RAM (random access memory) to store things while it is switched on and running. This is different from hard-disk space, which is the number commonly advertised on the back of the iOS device: 16GB, 32GB, and so on.

RAM is located inside microchips and is erased/replaced with new information frequently. Whenever you power off off your iOS device, the contents of RAM are lost for good. This is not the case with things stored on the hard disk, whose contents are available the next time the device is turned back on. Keep in mind though, that putting your iOS device to sleep is not the same thing as powering it off. When you put the device to sleep, you are essentially turning off the display and saving battery.

The amount of RAM available on a particular iOS device depends on the model, but is always significantly less than its hard-disk capacity. The amount of memory of either type (RAM or hard disk) is measured in megabytes (MB) and gigabytes (GB).

A bit stands for *binary digit* and is a fundamental unit of information storage on iOS devices. A byte is 8 bits, a kilobyte is 1024 bytes, a megabyte is 1024 kilobytes, and a gigabyte is 1024 megabytes.

RAM is organized sequentially into tiny byte-sized bins, each with a unique number (Figure B-5). A good analogy would perhaps be the locker room in your local gym. Each individual locker would represent a byte of RAM memory, and each locker has a number to distinguish it from other lockers.

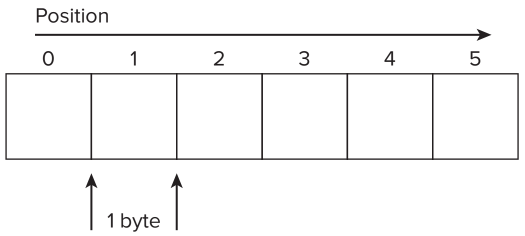


FIGURE B-5

Depending on the data that your application want to store, the computer will use one or more bytes. The more data to be stored, the more number of bytes of RAM will be required.

Variables

To store data, a computer program uses something called a *variable*, which is a named location in RAM that has a value. This value can change depending on your program's requirements.

In terms of the locker analogy, if you were to pick one of the lockers and label it with a name (perhaps by sticking a card over the locker with some tape), that named locker would now represent a variable.

When you use a variable's name in your program, you are essentially referring to the data stored in the associated location in RAM.

Declaration and Assignment

Now that you know a little about how RAM works, and what variables are, how do you go about creating a variable and storing some data in it?

It turns out this is a two-stage process. First you need to declare a variable (which in terms of the locker analogy is the act of labeling a free locker), and then you can store some data in it.

To declare a variable, you need to pick a name for the variable (more on this later), and specify the type of data the variable will store.

Computer data is essentially one of two types: number-based (numeric) or character-based (string). Consequently, variables are one of two types: numeric or string. This appendix focuses on numeric variables. The type of a data stored in a variable is also referred to as the variable's *data type*.

Since Objective-C was built upon the C programming language, you can use the standard C data types in your Objective-C code; however, Objective-C provides new data types in addition to the C data types. Table B-1 describes some of the common numeric variable data types available to Objective-C programmers.

TABLE B-1: Some Numeric Data Types in Objective-C

OBJECTIVE-C DATA TYPE	C DATA TYPE	BYTES OF STORAGE REQUIRED	NOTES
	`int`	4	Can store integers in the range –2,147,483,648 to 2,147,438,647
	`unsigned int`	4	Can store integers in the range 0 to 4,294,967,295
	`bool`	1	Can be either `true` or `false`
	`long`	8	Can store a much larger range of values than the `int` data type

continues

TABLE B-1 *(continued)*

OBJECTIVE-C DATA TYPE	C DATA TYPE	BYTES OF STORAGE REQUIRED	NOTES
	`unsigned long`	8	Can store a much larger range of values than the `unsigned int` data type
	`float`	4	Can store decimal numbers
	`double`	8	Can store decimal numbers with much greater precision than `float`
`NSInteger`		4	Equivalent to the `int` data type
`NSUInteger`		4	Equivalent to the `unsigned int` data type
`BOOL`		1	Can be either `YES` or NO
`id`		4	Can store a reference to an object
`NSDecimal`		20	Used to store decimal numbers

For a complete list of Objective-C data types available to iOS developers refer to the Foundation Data Types Reference document available at: `http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_DataTypes/Reference/reference.html`.

For example, to declare a variable `height`, whose data type is int, you write a line of code like this:

```
int height;
```

To assign a value to the variable named `height` after you have declared it, you write a line of code like this:

```
height = 10;
```

It is worth noting that no units are used while assigning a value to the variable named `height`. This is because as far as the computer is concerned, the numeric variable `height` only stores a number. Whether that number represents height measured in feet, inches, or meters depends on the context in which you use the variable.

You could combine both declaration and assignment into a single line as follows:

```
int height = 10;
```

You could also declare multiple variables (of the same data type) in a single line as follows:

```
int height, weight;
height = 10;
weight = 20;
```

or

```
int height = 10, weight = 20;
```

When it comes to naming variables, it is important to pick a name that indicates what the variable is likely to represent. Names like speed, distance, weight, and height are far more descriptive than x, p, r, and q.You need to be aware of a few rules regarding the naming of variables:

- *The name may contain digits, letters, and an underscore (_) character.*
- *The name must not begin with a number (it can begin with a letter or an underscore character).*
- *Variable names are case-sensitive, thus weight, Weight, and weiGHT represent three different variable names.*
- *You cannot use Objective-C keywords as variable names. Certain words have special meaning in Objective-C, and these cannot be used as variable names.*

Variable names need not be single words; you could create a variable name such as `heightOfStudents`*. Keep in mind, though, that whitespace cannot be part of a variable name. When a variable name consists of multiple words, it is customary to capitalize the first letter of each word except the first. This notation is commonly known as camel case notation.*

Statements

An Objective-C program consists of a series of statements; each statement is a directive to the iOS device to carry out a particular task. Most statements are written one per line, although some could span multiple lines. Statements usually end with a semicolon (`;`) character. You saw a few examples of Objective-C statements in the previous section.

Null Statements

If you type the `;` character on its own on an otherwise empty line, you create a null (do-nothing) statement. Although this is perfectly valid Objective-C code, use of null statements is not recommended.

Whitespace

When it comes to typing computer programs, the term *whitespace* refers to empty lines, tabs, or spaces in your code. In general, the Objective-C compiler ignores whitespace. Thus as far as the Objective-C compiler is concerned, the following statements are identical:

```
int height = 10;
```

and

```
int height=10;
```

and

```
int height=        10;
```

However, whitespace is required between the data type identifier (`int`) and the name of the variable (`height`). Furthermore, you cannot have whitespace as part of the name of a variable. Thus, the following code cannot be compiled by the Objective-C compiler:

```
intheight = 10;
```

and

```
int h e i ght = 10;
```

Whenever the Objective-C compiler is unable to compile one or more statements in your program, it produces a compilation error and aborts the process. Whitespace should be used to make code appear more readable.

Compound Statements

You can group together multiple statements into a compound statement by enclosing them in between `{` and `}`. Compound statements are also known as blocks.

```
{
    int iNumber1 = 10;
    int iNumber2 = 50;
}
```

It is a good idea to place the enclosing braces on a separate line each, that way it is easier to see the beginning and end of each block. You can use a block anywhere you can use a statement.

Expressions and Operators

In most cases, your Objective-C statements will consist of a combination of expressions and operators. An *expression* is basically something that evaluates to a number. The simplest expression is a numeric variable (because a numeric variable is guaranteed to evaluate to a number, that number being the value stored in the variable).

More complex expressions can be formed by connecting simpler expressions with operators, for example:

```
10 - 4
```

is a complex expression made up of two simple expressions (`10`) and (`4`), that are combined with the minus (–) operator. As you can see, the expression `10 - 4` will evaluate to `6`, which is a number.

An operator is basically a special symbol that instructs the compiler to perform some action using one or more operands. Most operators require two operands; however, some require just one, or even three operands.

Several operators are available to you as an Objective-C programmer; Table B-2 lists some of the common ones.

TABLE B-2: Objective-C Operators

SYMBOL	DESCRIPTION	EXAMPLE
/	Divides two operands; second operand must not be zero	`number1 / number2`
*	Multiplies two operands	`number1 * number2`
+	Adds two operands	`number1 + number2`
–	Subtracts two operands	`number1 – number2`
=	Assignment operator	`number1 = 300`

You can also create expressions using numeric variables just as well. Typically, you would assign an expression to a numeric variable, that is, the numeric variable would contain the number obtained by evaluating an expression, as shown here:

```
int iFirst = 100;
int iSecond = 30;
int iResult = iFirst + iSecond;
```

When an expression contains multiple operators, Objective-C applies certain rules to decide the order in which operators should be evaluated. To understand why the order is necessary, consider the expression:

```
4 - 1 * 5
```

If `4 - 1` was evaluated first, and the result multiplied by `5`, then this expression would evaluate to `15`. However, if `1 * 5` was evaluated first, and the result subtracted from `4`, this expression would evaluate to `-1`. So which is it?

It turns out that it is the latter, because Objective-C considers `*` (multiplication) to be a higher priority operator than – (subtraction). Table B-3 lists the priority of operators in increasing order.

TABLE B-3: Priority of Objective-C Operators

OPERATORS (IN INCREASING ORDER OF PRIORITY)
–
+
*
/
=

MAKING DECISIONS AND PERFORMING REPETITIVE TASKS

Besides simple assignments and arithmetic, statements can also be used to control the flow of your program. It is time to introduce you to a new type of Objective-C operator that helps you compare variables. Collectively, these operators are called the *comparison operators*. Table B-4 lists the standard Objective-C comparison operators:

TABLE B-4: OBJECTIVE-C COMPARISON OPERATORS

SYMBOL	DESCRIPTION/USAGE	EXAMPLE
`==`	Are the two operands equal?	`a == b`
`>`	Is the first operand greater than the second?	`a > b`
`<`	Is the first operand less than the second?	`a < b`
`>=`	Is the first operand greater than or equal to the second?	`a >= b`
`<=`	Is the first operand less than or equal to the second?	`a <= b`
`!=`	Are the two operands not-equal?	`a != b`

You can think of comparison operators as ones that always pose a question, and the answer to the question is `YES` or `NO`. Thus, an expression that contains a comparison operator will always evaluate to `YES` or `NO`.

The result of a comparison is always assigned to a variable of type `BOOL`. This is the Objective-C data type for *Boolean* variables. A Boolean data type is one that can take only one of two values (`YES` or `NO`) and requires exactly 1 byte of storage space in RAM. You would use a `BOOL` data type as follows:

```
BOOL result = YES;
```

It is important to note that the value assigned to a Boolean variable must be uppercase `YES` or uppercase `NO`. The result of evaluating an expression that involves comparison operators is always a Boolean variable, for example the expression:

```
BOOL comparisonTest = 10 > 4;
```

evaluates to `YES` if operand `1` (which is `10` in this case) is greater than (but not equal to) operand `2` (which is `4` in this case).

The if and if … else statements

Comparison operators are used to create control statements. Normally statements are written in a serial top-bottom order and execute one by one in the order in which they were written. A *control statement* allows you to modify the order of statements executed, execute certain statements multiple times, or execute certain statements conditionally.

The `if` statement is one such control statement. In its basic form an `if` statement executes a statement only if a specific condition is met.

```
if (condition evaluates to YES)
   statement to execute;
```

The *test condition* is usually a Boolean variable, or an expression that evaluates to a Boolean variable. If the test condition evaluates to `YES`, the following statement is executed. If the test condition evaluates to `NO`, the following statement is not executed. A simple example would be:

```
int numberOfRedMarbles = 20;
int numberOfBlueMarbles = 5;
if (numberOfRedMarbles > numberOfBlueMarbles)
        NSLog(@"%@", @"Game over, you won!");
```

In this hypothetical game example, a player is required to collect a certain number of red and blue marbles. A player wins the game if he collects more red marbles than blue ones. The test condition in this case is the expression:

```
numberOfRedMarbles > numberOfBlueMarbles
```

which evaluates to `YES` in this particular case. Note the complete `if` statement contains two lines. The first line contains the condition, and the second contains the statement to execute if the condition evaluates to `YES`. If you were to write only the first line without the second, the Objective-C compiler would consider it a syntax error.

As you learned in this appendix, a single Objective-C statement (that is, one written on a single line), can be replaced by a compound statement (a block of statements contained between `{` and `}`).

An example of an `if` statement that executes a compound statement would be:

```
int numberOfRedMarbles = 20;
int numberOfBlueMarbles = 5;
int score = 0;
if (numberOfRedMarbles > numberOfBlueMarbles)
{
    score = 10;
    NSLog(@"%@", @"Game over, you won!");
}
```

In this example, if the number of red marbles is more than the number of blue marbles, a block of statements is conditionally executed. This particular block consists of only two statements:

- Statement 1 gives the player 10 score points.
- Statement 2 writes a message to the Debug console.

However, the block could just as well have contained a few dozen statements. The key point to note is that the complete `if` statement consists of two parts: the first part is where the condition is being tested and the second is a single (or a block) statement that is to be executed when the condition evaluates to `YES`.

Any statements that follow a two-part `if` statement will execute as normal after the `if` statement has finished executing.

Thus, in this example:

```
int numberOfRedMarbles = 20;
int numberOfBlueMarbles = 5;
int score = 0;
if (numberOfRedMarbles > numberOfBlueMarbles)
{
    score = 10;
    NSLog(@"%@", @"Game over, you won!");
}

int totalMarbles = numberOfRedMarbles + numberOfBlueMarbles;
NSLog(@"%d", totalMarbles);
```

the last two statements would execute regardless of the outcome of the `if` statement, because they are separate statements in their own right. The only statements that may or may not execute are part of the `if` statement, and the precise condition that is being tested for will determine whether or not they execute.

The else Clause

A modified version of the `if` statement allows you to specify an additional statement (or a block) that is executed should the test condition fail. This additional alternate-scenario statement is completely optional and should you need to specify it, you can use the modified form of the `if` statement:

```
if (condition evaluates to YES)
       statement to execute;
else
       some other statement to execute;
```

This modified form of the `if` statement is known as the `if…else` statement. The `else` portion is optional. The statement (or block) following the `else` clause is executed only if the test condition evaluates to `NO`. A simple example follows:

```
int numberOfRedMarbles = 20;
int numberOfBlueMarbles = 5;
if (numberOfRedMarbles > numberOfBlueMarbles)
{
    NSLog(@"%@", @"Game over, you won!");
}
else
{
    NSLog(@"%@", @"You lost. Better luck next time!");
}

int totalMarbles = numberOfRedMarbles + numberOfBlueMarbles;
NSLog(@"%d", totalMarbles);
```

In this example, if `numberofRedMarbles` is greater than `numberofBlueMarbles` then

```
{
    NSLog(@"%@", @"Game over, you won!");
}
```

will be executed, otherwise

```
{
    NSLog(@"%@", @"You lost. Better luck next time!");
}
```

will be executed. Now it just so happens to be the case that 20 is greater than 5 and hence the block associated with the `else` clause will not execute in this example.

In the examples so far, the conditional expressions are trivial (such as is 20 greater than 5) and strictly speaking, the `if` statement is not being used to its true potential. In a real-world application the values of the operands in the conditional expression would be dynamic, for instance the number of times a tap is detected, or the number of alien spaceships destroyed by the player as a game proceeds. In these cases the `if` and `if...else` statements are extremely useful.

Just as with an `if` statement, statements that appear after the `if…else` statements would continue to execute regardless of what happened in the `if...else` statement.

The for Statement

The `for` statement (also known as the `for` loop), on the other hand, allows you to execute a statement (or a block) multiple times consecutively. The precise number of repetitions will depend on an integer counter variable that you control.

The `for` statement allows you to create loops in your code (a *loop* is a term used to describe a situation in computer programming when a statement or a block is executed multiple times in succession. The general form of the `for` statement is:

```
for (initial expression; termination expression; increment expression)
    loop statement;
```

Unlike the `if` and `if...else` statements, a `for` statement requires three expressions:

- **Initial expression:** This expression usually involves an assignment (where a value is assigned to a variable).
- **Termination expression:** This expression usually involves a comparison operator and evaluates to either `YES` or `NO`. If this expression evaluates to `NO` the body of the loop will not be executed.
- **Increment expression:** This expression usually adds an integer to the variable used in the initial expression.

The loop statement is any Objective-C statement (or a block of statements) and is also known as the *body* of the loop.

When a `for` loop is encountered, the following happens:

1. The initial expression is evaluated.
2. The termination expression is evaluated.
3. If the termination expression evaluates to `NO` the `for` statement terminates, and execution continues at the first statement after the loop block.

4. If the termination expression evaluates to `YES` the loop statement/block is executed once.
5. The increment expression is evaluated, and execution continues from step 2.

These steps are depicted in Figure B-6.

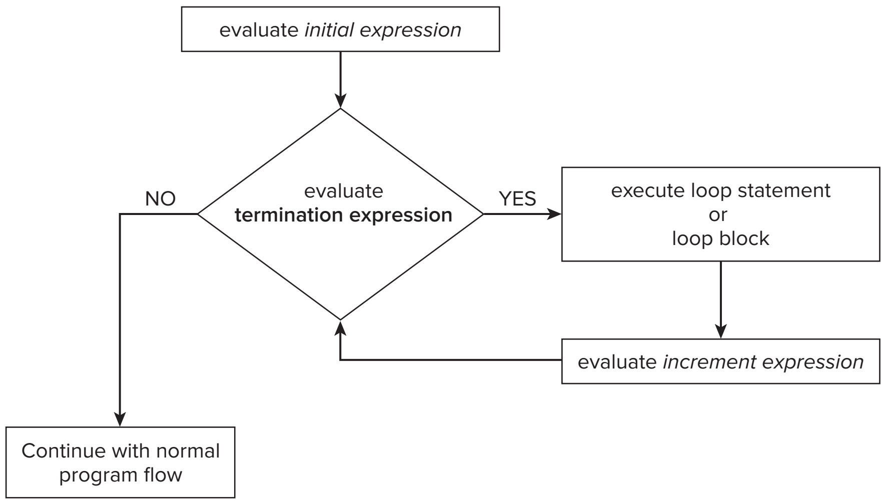

evaluate *initial expression*
NO
evaluate
termination expression
YES
execute loop statement
or
loop block
evaluate *increment expression*
Continue with normal
program flow

FIGURE B-6

As an example of a `for` statement in action, consider the following snippet:

```
int number;
for (number = 1; number <= 5; number = number + 1)
{
    NSLog(@"%@", @"This message is displayed by a loop block!");
}
```

This snippet would result in the following output:

```
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
```

In this example, the initial expression sets the value of the variable `number` to `1`:

```
number = 1
```

The termination expression is a conditional expression that evaluates to `YES` or `NO`. In this case the expression tests if the value of `iNumber` is less-than-or-equal-to `5`:

```
number <= 5
```

The increment expression adds `1` to the value of the variable `number`:

```
number = number + 1
```

Without this expression, the value of `number` would never change, and termination expression would never evaluate to `NO`. Consequently the loop would go on indefinitely.

The while Statement

The `while` statement (also known as the `while` loop), is another statement that allows you to create loops. It executes a statement (or a block of statements) as long as a specified condition holds true. The general form of the `while` statement is:

```
while (loop condition)
    loop statement;
```

The *loop condition* is typically an Objective-C expression that involves a conditional operator, and evaluates to `YES` or `NO`. The *loop statement* is any Objective-C statement (or a block of statements) and is also known as the body of the loop.

When a `while` loop is encountered, the following happens:

1. The loop condition is evaluated.
2. If the loop condition evaluates to `NO` the `while` statement terminates, and execution continues at the first statement after the loop block.
3. If the loop condition evaluates to `YES` the loop statement/block is executed once, after which execution continues from step 1.

These steps are depicted in Figure B-7.

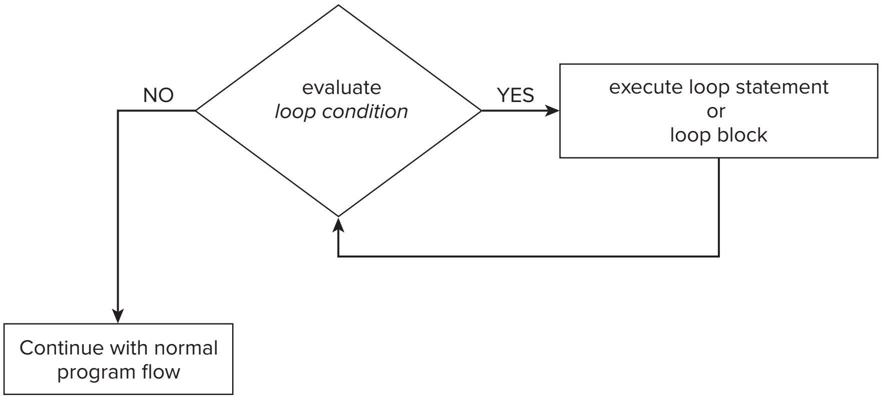


FIGURE B-7

As an example of a `while` statement in action, consider the following code snippet:

```
int number = 1;
while (number <= 5)
{
    NSLog(@"%@", @"This message is displayed by a loop block!");
    number = number + 1;
}
```

This snippet would result in the following output:

```
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
```

In this example, the loop condition is a conditional expression that evaluates to `YES` or `NO`. In this case the expression tests if the value of `number` is less-than-or-equal-to `5`:

```
number <= 5
```

It is worth noting that the value of `number` is incremented by 1 in the body of the `while` loop. If this was not done, then `number` would always equal `1` and this loop would never terminate.

The do...while Statement

The `do...while` statement (also known as the *`do...while` loop*), is another statement that allows you to create loops. It first executes a statement (or a block of statements) and then checks a specified condition to determine if the preceding statement/block should be executed again. The general form of the `do while` statement is:

```
do
    loop statement;
while (loop condition)
```

Once again, the loop condition is typically an Objective-C expression that involves a conditional operator, and evaluates to `YES` or `NO`. The loop statement is any Objective-C statement (or a block of statements) and is also known as the body of the loop.

When a `do...while` loop is encountered, the following happens:

1. The loop body is executed.
2. The loop condition is evaluated.
3. If the loop condition evaluates to `NO` the `do...while` statement terminates, and execution continues at the first statement after the loop block.
4. If the loop condition evaluates to `YES` execution continues from step 1.

As an example of a `do...while` statement in action, consider the following code snippet:

```
int number = 1;
do {
    NSLog(@"%@", @"This message is displayed by a loop block!");
    number = number + 1;
} while (number <= 5);
```

This snippet would result in the following output:

```
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
This message is displayed by a loop block!
```

A key point about the `do...while` loop is the fact that the body of the loop is guaranteed to execute at least once because the loop condition is evaluated after the loop body is executed.

Don't forget to include the semicolon (`;`) at the end of the `do...while` loop:

```
while (number <= 5);
```

The break and continue Statements

You can use the `break` statement as part of the statements that form the body of a `for`, `while`, or `do...while` loop, to end the loop prematurely. Programs that use loops generally rely on the loop to come to a natural end at some point based on some test criteria.

However, sometimes you need to break out of a loop prematurely (perhaps in response to some external factor) and in such cases you can use the `break` statement. The `break` statement is written on its own, on a single line:

```
break;
```

Any other statements in the block after the `break` statement will not be executed.

The `continue` statement, when used in the body of a loop, causes execution to skip one iteration of the loop. Any statements after a `continue` statement will be skipped for that iteration. The `continue` statement is also written on its own, on a single line:

```
continue;
```

WHEN TO USE WHAT (FOR, WHILE, DO. . .WHILE)

Objective-C provides three statements to create loops. Though it is possible to create equivalent loops using any of these statements, here are a few guidelines that should help you decide when to use them:

- *If you need to execute one or more statements a fixed number of times (and you know the number of iterations beforehand), use a* `for` *loop.*
- *If you do not know the number of iterations, and can only decide whether or not another iteration should execute within the loop body, use a* `while` *loop.*
- *If you need to guarantee the loop body is executed at least once, use a* `do...while` *loop.*

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

Over the years, computer application developers have come up with various strategies to create applications that can solve complex problems. One of the earliest approaches to problem solving was the concept of structured programming.

Structured programming (which predates object-oriented programming) centered on a divide-and-conquer philosophy. A complex program was broken down into a set of tasks, and each task subsequently into a set of simpler sub-tasks.

Blocks of code were then written to perform these simple tasks, and each block of code was given a name. Keep in mind that a block of code is just a series of statements enclosed between `{` and `}`).

The reason why names were assigned to these blocks of code was to make it possible to refer to one block of code from within another. These named code blocks were conveniently called *functions*.

Structured programming was all about breaking down a complex task into a set of simpler tasks, each to be performed by a function. If the simpler task wasn't quite so simple, it would be broken further into even simpler sub-tasks. Each of these sub-tasks would then be performed by a function that would be used by the function that represented the super task. The practice of using a function (block of code with a name) was termed *calling the function*.

Typically these functions would operate on data (variables), and conceptually an application was divided into data and a set of functions that acted on that data. A simple example will help clarify this:

Suppose an application was to be developed for a bank that would allow them to compute the net interest payable across all savings accounts in a year. Now, this is a rather complex task and can easily be broken down into finding out the net interest over each month and then summing up these results to reach the net interest payable over the year (Figure B-8).

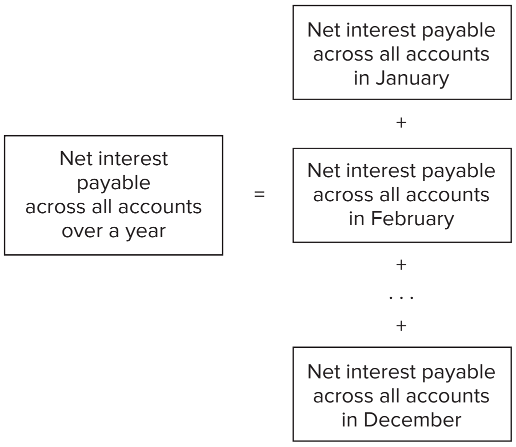

Net interest payable across all accounts over a year
=
Net interest payable across all accounts in January
+
Net interest payable across all accounts in February
+
. . .
+
Net interest payable across all accounts in December

FIGURE B-8

The task of computing the next interest payable across all savings accounts for a given month can be broken down to computing the interest payable to individual savings accounts in that month, and then summing up these results.

This division could be stopped at this point, or could be carried further down to another level. Now, each of these tasks and sub-tasks would be performed by a function (named block of code). For the purpose of this discussion assume the names `computeAnnualNetInterest`, `computeMonthlyNetInterest`, and `computeNetInterestForAccount`.

A pseudo-code version of such a program is presented in the following code snippet. Pseudo-code is an informal high-level description of a computer program, intended to illustrate concepts only.

```
computeAnnualNetInterest
{
    int total_months = 12;
    for (int month = 1; month <= total_months; month++)
    {
        computeMonthlyNetInterest(month);
    }
}

… elsewhere in the code …

computeMonthlyNetInterest
{
    int total_number_of_accounts = 5412;
    for (int account = 1; account < total_number_of_accounts; account++)
    {
        computeNetInterestForAccount(account);
    }
}
… elsewhere in the code ….
computeNetInterestForAccount
{
… do some simple computation here
}
```

At some function in the preceding example, actual computation will have to be performed, and for this some data will be needed. The data in this case could be as simple as a list of accounts and the total dollar balance in them. The point to take from this example is that structured programming requires a clear split between the data and the code that works on that data.

The Need for Object-Oriented Programming

The structured programming approach covered in the previous section is still in use in some types of applications today; however, it has a few drawbacks:

- People generally think of data (account numbers) and what they can do with it (compute balance, interest, and so on) as related concepts. It is not natural to think of them in isolation.
- Programmers were constantly reinventing the wheel, creating solutions for things that had been solved over and over again by others. Structured programming did not address the need to reuse existing functions (either written by you or someone else) conveniently. Imagine if you were manufacturing a car and had to manufacture each little part yourself without any possibility of using a seat or a tire from another manufacturer. This would become a very cost-inefficient approach. It would be far better for the automobile industry to promote re-use between manufacturers and have different manufacturers stick to making specific items. The same option, however, was just not available to software programmers who were using structured programming techniques.

A new approach to programming was created, and named *object-oriented programming (OOP)*. Essentially OOP tries to address the deficiencies in the structured programming model by:

- Providing techniques to achieve re-use of software components
- Coupling data with the functions that act on them

Core to object-oriented programming is the idea of treating data and functions that act upon them as an independent entity know as an *object*.

Key Concepts of Object-Oriented Programming

In the previous section you learned why object-oriented programming was needed. In this section you learn some of the key concepts introduced by the object-oriented programming approach.

Classes and Objects

A *class* can be thought of as a template or blueprint of an object. This is best understood by an example. If you were to go down to your local car dealer, you would likely find several cars there. Each of these cars share some common characteristics with each other; for instance each has seats, wipers, four wheels, and so on. Looking at this situation from an object-oriented perspective you can say that each of these cars is an instance of a class of objects called automobiles. The `Automobile` class (Figure B-9) could then be thought to define some characteristics that are common to each instance (like the fact that each car has four wheels).

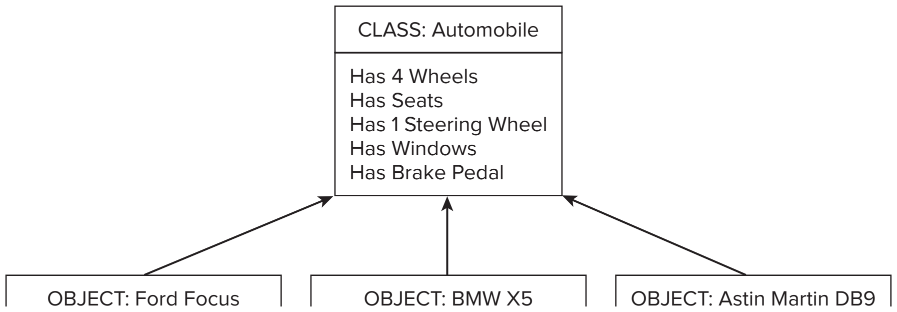


FIGURE B-9

If you were tasked with creating a software-based version of the `Automobile` class, these common characteristics between the various instances of the `Automobile` class can be easily represented using variables; for example, number of wheels could be represented by an integer variable `iWheelCount` and so on. Table B-5 lists the characteristics of the `Automobile` class and the equivalent variables that could be used to represent them.

TABLE B-5: Characteristics of the Automobile Class

CHARACTERISTIC	VARIABLE
Has 4 wheels	`int wheelCount`
Has seats	`BOOL hasSeats`
Has 1 steering wheel	`int numberOfSteeringWheels`
Has windows	`BOOL hasWindows`
Has brake pedal	`BOOL  hasBrakePedal`

However, to be compliant with the principles of object-oriented design, this `Automobile` class must also define some operations that do something with these characteristics/variables (Figure B-10). Whatever these operations may be, each concrete instance of the `Automobile` class will be able to perform them.

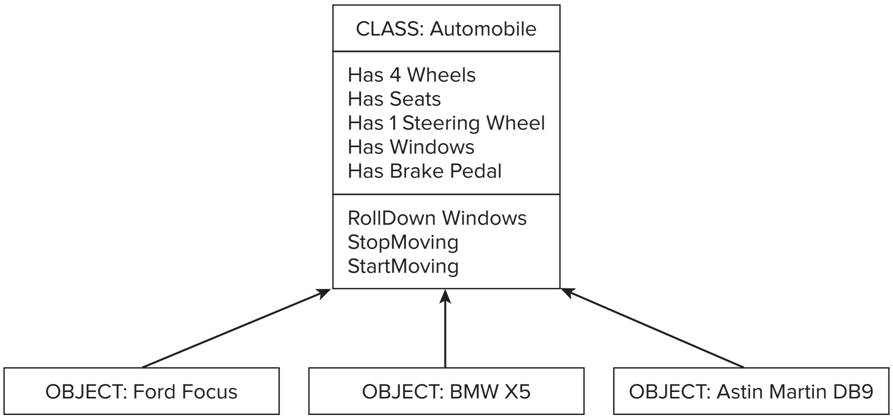


FIGURE B-10

These operations are best thought of as commands you could give to a car (instance of `Automobile` class). This is perhaps where object-oriented solutions differ from real-world situations. In the real world you can't command a car to drive itself (except in the movies); you need to drive the car. In an object-oriented world, however, the car would drive itself and all you would have to do is tell the car to start driving. Table B-6 lists a few possible operations that the `Automobile` class could define:

TABLE B-6: Operations in the Automobile Class

OPERATION	DESCRIPTION
`RollDownWindows`	The car rolls down all its windows.
`StopMoving`	The car stops moving.
`StartMoving`	The car starts moving.

Just like you use variables to represent the common characteristics, each of these operations would be represented using blocks of code (functions). These blocks of code would operate on the data (variables) within the class to achieve the desired outcome.

To put it formally: A class is simply a collection of variables and a list of functions that act on those variables. An object is a concrete instance of a class. (Recall that a function is simply a named block of code.)

The variables defined in a class are called its *instance variables* (ivars for short), and the functions that act on those variables are called *methods*.

Pay attention to the fact that when it comes to using a class, you need to instantiate it into a concrete object first. All subsequent interaction will be with the object and not the class. The `Automobile` class in this example is not an actual car; it is just the definition of what a car should be. Instantiating an Objective-C class is covered in the topic Instantiating Objects, later in this appendix.

Inheritance

When developing an application, you are likely to create more than one class. The classes you define are likely to have some relationships with each other. Object-oriented programming allows you to specify different types of relationships between classes.

The concept of *inheritance* implies that a new class can be created that inherits the functionality of an existing class. This new class will provide the functionality of the parent class and provide some additional functionality of its own. Inheriting from a base class is known as subclassing. By subclassing an existing class, the designer of an object-oriented solution is reusing the functionality present in an existing class and not duplicating it. The parent class is commonly referred to as the *base class*, and the child as the *subclass*.

As an example consider a hypothetical class `Dog` (Figure B-11). Such a class could either be created in isolation, or more likely inherit from a more general class `Mammal`. The attributes and methods present in the `Mammal` class would be a part of the `Dog` class. In addition the `Dog` class would add a few attributes and methods of its own.

When you use inheritance to create a relationship between two classes, you are essentially creating an *is-a* relationship between them. This in the above example, a `Dog` is a `Mammal`.

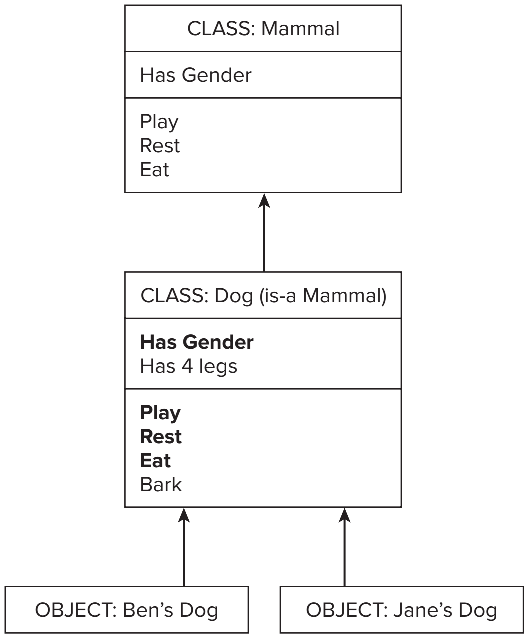


FIGURE B-11

Composition

You learned earlier that classes can contain member variables. These member variables, however, need not be restricted to `int`, `double`, and `BOOL` as you may have thought. Classes can contain objects of different classes as their member variables. This principle is called *composition* and allows you to create composite classes.

This is best explained by a simple example. Consider the `Automobile` class. This class is actually a composite class that includes within it objects of several other classes (for example, one object of the `Engine` class, four objects of the `Door` class, and so on).

If you were to try to analyze real-world objects you would come to the conclusion that almost every real-world object is an instance of a composite class. The same holds true for objects in software applications.

Composition implies a *has-a* relationship between objects. Thus an `Automobile` instance has an `Engine` instance and four `Door` instances.

Encapsulation

Encapsulation refers to a much-desired property of an object to behave like a self-contained black box. A well-designed object should hide its inner workings from the outside world. This would allow a programmer to use the object without knowing how it works internally. This is much like how you use your iPhone without knowing how its wireless receiver works.

Polymorphism

Polymorphism refers to a concept in object-oriented programming where a derived class is free to provide a specialized implementation of a method it has inherited from its parent class. This specialized method implementation in the subclass is called an *override*. Programmers thus implement polymorphism by overriding base class methods in a subclass.

Once again, consider the `Mammal` and `Dog` classes (see Figure B-11). Clearly, `Dog` is a subclass of `Mammal` and by virtue of that inherits the `Play`, `Rest`, and `Eat` methods. Now create another class, `Kangaroo`, which is also a subclass of `Mammal` (Figure B-12).

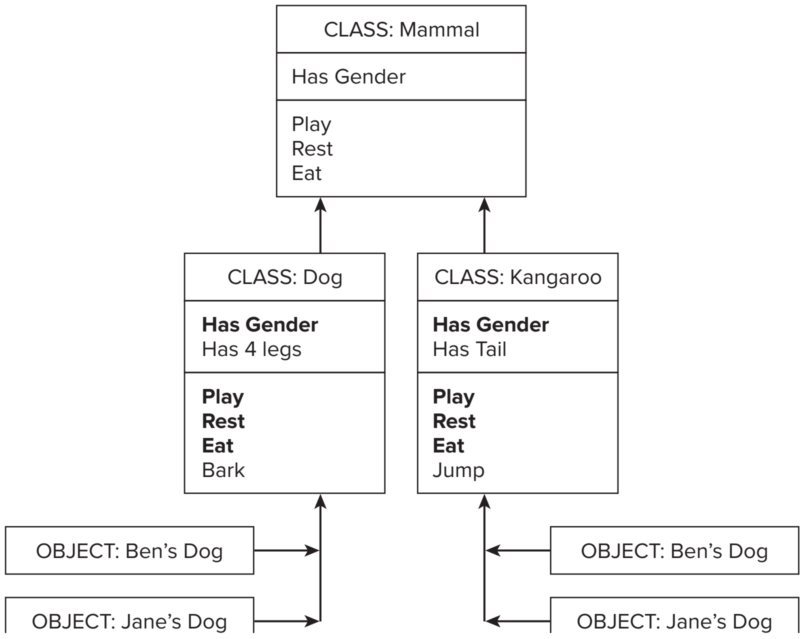


FIGURE B-12

Although both the `Dog` and `Kangaroo` classes have inherited a `Play` method, they are likely to implement it differently; after all they are very different animals.

Protocols

A *protocol* can be thought of as a contract that a class agrees to abide by. Technically speaking, the class is said to implement the protocol in question. But what form does this contract take?

This contract (protocol) is basically a list of methods. These methods can be grouped as either required or optional. Any class that wishes to conform to a protocol must provide implementations of all required methods in the protocol. A class can implement multiple protocols (Figure B-13), and multiple classes may implement a given protocol.

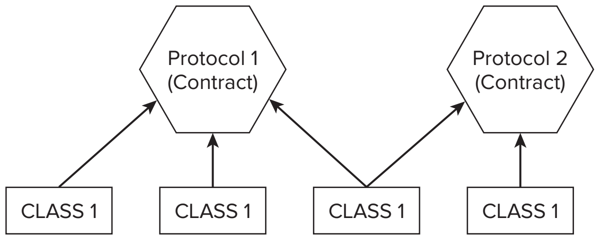


FIGURE B-13

Just because a class implements a protocol does not mean that the class cannot have additional methods of its own (in addition to the ones defined in the protocol). The manner in which protocols are used depends on the designer of the object-oriented system. In other object-oriented languages like C++, protocols are known as interfaces.

Why are protocols useful? Essentially, a protocol guarantees that an object of a conforming class will respond to a set of methods. For example, consider a hypothetical scenario where you are hired by an electronics giant to create a universal remote control. The primary requirement is that this remote control utilizes object-oriented principles and can control a wide range of home-entertainment devices from different manufacturers.

To simplify things a little and to keep this example within the scope of the book, assume that the various devices that your remote control will interact with are all objects of one kind or the other (television, radio, DVD player, and so on).

Being objects, each of these devices will provide a few methods that can be used to control them. However, because these devices are of different kinds, and are probably made by different manufacturers, each implements a slightly different set of methods. Table B-7 lists the methods defined by each device.

TABLE B-7: Methods provided by different devices

TELEVISION	RADIO	DVD PLAYER
`switchOn`	`turnOn`	`powerOn`
`jumpToChannel`	`tuneToChannel`	`skipToTrack`

You could try to design a remote control that somehow detects the type of device it is interacting with, and then uses specific methods for that device.

The alternative is that you define a protocol called `UniversalRemoteDelegate` that implements two methods—`switchOnDevice` and `changeChannel`—and have the manufacturers of each of those devices conform to your protocol. In order to declare themselves compliant with your standard, they would have to implement these two methods in their devices regardless of whatever other methods they implement.

Remember, you happen to be hired by an electronics giant, who presumably has the necessary influence to make this happen. But, if this were to happen your task as a remote control designer is quite simple because you know for certain that every controllable device will have two methods that you can use.

This example illustrates the power of interfaces; applications can be built that can work with objects of different types, created by different programmers.

Creating an Objective-C Class

Creating a class in Objective-C is a two-step process. First you need to declare the class (in a header file) and then implement its methods (in an implementation file). Thus the definition of an Objective-C class spans two physical files as shown in Figure B-14.

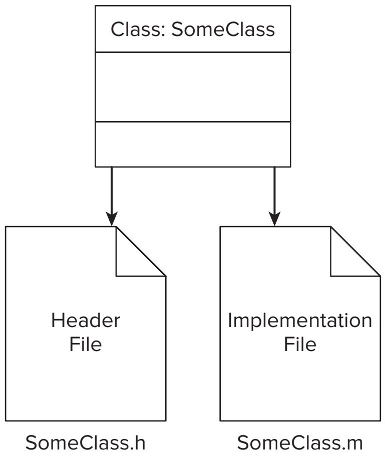


FIGURE B-14

The declaration of the class (header file) includes, among other things, a list of the member variables and functions (methods) provided by the class. The implementation file is where you would type in the code that actually makes these methods work.

The following code snippet shows the declaration of a `Planet` class:

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

@interface Planet : NSObject
{
    float surfaceTemperature;
}

- (void)spinOnAxis;
- (float)distanceFromSun;
- (void)rotatePlanetByAngle:(float)angle;

@end
```

The #import Directive

The first line of the header file usually starts with the `#import` directive. A directive is an instruction to Xcode and begins with the `#` character:

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

This particular directive (`#import`) tells Xcode to include the contents of the file referred to in the angular brackets at that position, provided it has not been included earlier.

As it turns out, Apple provides hundreds of classes that you can use in your code. For instance, the slot machine–style date picker that you commonly encounter in many apps is actually a class provided by Apple. Much of iOS programming has to do with using the methods of one or more classes provided by Apple.

Classes that provide related functionality are grouped together in what is known as a *framework*. For example, all classes that deal with video playback are packaged into a single framework called Media Player.

If you need to include a file that is part of a standard framework provided by Apple, you enclose the name of the file in angular brackets, as seen in the preceding `import` statement. However, if you need to include a file that is part of your own code, you use double quotes instead of angular brackets. You see an example of this shortly.

Earlier in this appendix, you also learned that classes inherit from other classes, thus creating tree-like hierarchies. You may be wondering if there is some root class that sits at the base of all Objective-C classes. There is, and it is called `NSObject`. This is shown in Figure B-15.

All Objective-C classes, including your `Planet` class, can be traced down to `NSObject` in their inheritance tree. `NSObject` is a class provided by Apple and is part of the Foundation framework. This is why the definition of the class includes the `Foundation/Foundation.h` file at the top.

The @interface and @end Keywords

The definition of a class in Objective-C is also called its *interface* and is basically everything between the `@interface` and `@end` keywords in the header file:

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

@interface Planet : NSObject
…
…
…

@end
```

The name of the class immediately follows the `@interface` keyword:

```
@interface Planet : NSObject
```

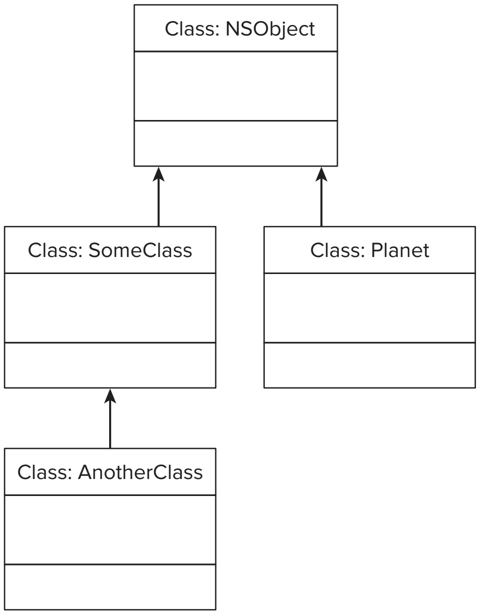


FIGURE B-15

Following the name of the class is the name of the parent class, separated by a colon (`:`). In most cases the classes that you create will inherit directly from `NSObject`. However sometimes you will derive your new class from an existing class, which in turn will inherit from `NSObject` somewhere in its inheritance tree.

Instance Variables and Methods

Instance variables are declared between the pair of curly braces immediately following the name of the class:

```
@interface Planet : NSObject
{
   …
   instance variables can be declared here
   …
}

@end
```

Methods are declared after the closing brace that follows member variable definitions, and before the `end` statement:

```
@interface Planet : NSObject
{
}

…
methods are declared here
…

@end
```

Methods never return more than one value; they can however take any number of input parameters. When you declare a method in a class, you will need to specify how many inputs it requires, the data types of these inputs, and whether or not the method will return a value.

For example, a method that does not require any input parameters and does not return a value can be declared as:

```
- (void)spinOnAxis;
```

Figure B-16 dissects this method declaration into its constituent parts. Every method begins with a – or a + character. A minus character indicates that the method can only be called on an object (instance of the class). This makes sense because the class is just a blueprint anyway that defines what an object should be. Methods that start with a – sign are also known as *instance methods*.

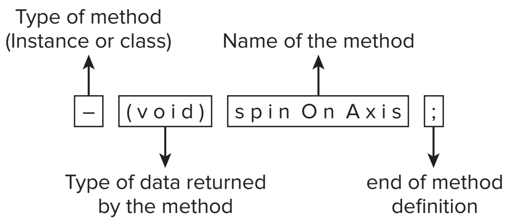


FIGURE B-16

Thus, if the `Planet` class defined a method `SpinOnAxis` you would need an actual object on which to call the method. In effect you would be asking the object to do something.

A + character before a method definition indicates that the method can be called on the class, and must not be called on an instance. Class methods have their uses, but they are beyond the scope of this appendix. It is safe to assume that as a beginner, all the methods that you will create in your classes will start with a – sign.

Following the – (or +) sign, a method definition specifies the type of information returned by the method in parentheses. If the method does not return anything, its return type is set as `void`.

Next comes the name of the method. The declaration of the method is complete by adding a `;` at the end of the line.

What if your method were to return a value? Say, a method that provides the distance of the planet from its sun. Such a method can be declared as:

```
- (float)distanceFromSun;
```

Note the return type is not `void` anymore! It is a numeric data type. This indicates that the method will return a numeric value, which in this case is a distance.

What if you wanted to create a method that took one input, perhaps a method that would allow a `Planet` object to be rotated about its axis by a specified angle? Such a method can be declared as:

```
- (void)rotatePlanetByAngle:(float)angle;
```

This method definition looks slightly different from the ones before it. That is because this method takes a single input value. Figure B-17 dissects this method.

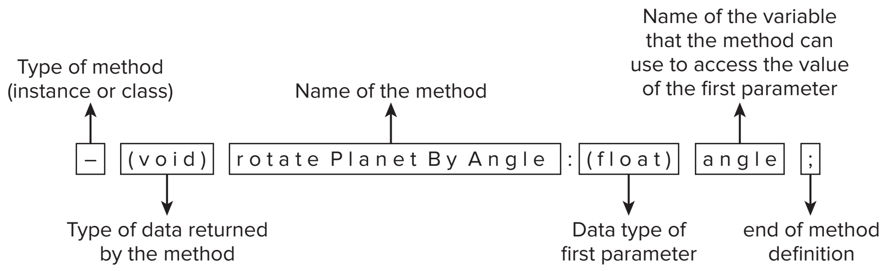

Type of method (instance or class)
Name of the method
Name of the variable that the method can use to access the value of the first parameter
–
(void)
rotate Planet By Angle
:
(float)
angle
;
Type of data returned by the method
Data type of first parameter
end of method definition

FIGURE B-17

If a method takes an input parameter, the name of the method is followed by a colon, followed by the data type of the input parameter in parentheses, followed by a variable name that will be used to refer to that parameter within the implementation of the method. When the `rotatePlanetByAngle` method is used, Objective-C will automatically create the variable named `angle` as specified in the method declaration, copy the value into it, and make it available for the body of the method to use.

Methods with Multiple Parameters

An Objective-C method is not restricted to taking a single input parameter. Methods can (and commonly do) require multiple input parameters. In this case, for each parameter, a short descriptive label is added. This label is followed by another colon, data type, and variable name.

The name of the method is then the name formed by collecting the labels of each parameter together.

It is best to illustrate this with an example. Suppose you wanted to add a method to the `Planet` class to rotate the planet by specific amounts along three orthogonal axes x, y, and z. Such a method would look like:

```
- (void)rotatePlanetAboutX:(float)xAngle aboutY:(float)yAngle aboutZ:(float)zAngle;
```

The name of this method is `rotatePlanetAboutX:aboutY:aboutZ:`. In this method declaration, the first parameter is labeled `rotatePlanetAboutX`, the second is `aboutY`, and the third is `aboutZ`. The values assigned to these parameters can be accessed within the body of the method by the variables named `xAngle`, `yAngle`, and `zAngle` respectively. This is depicted in Figure B-18.

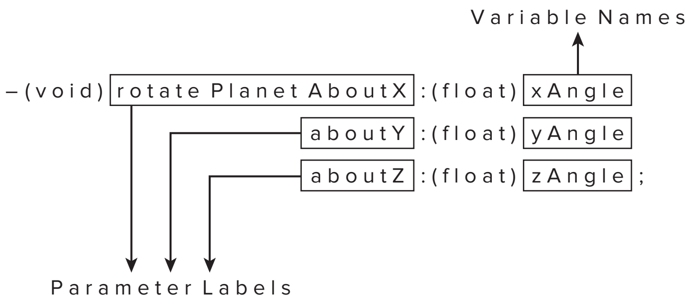


FIGURE B-18

OBJECTIVE-C CATEGORIES

A category allows you to add methods to an existing class, even if you do not have the source to the class. Using categories you can extend the functionality of a class without subclassing. For more information on categories refer to "The Objective-C Programming Language Guide" available at: `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocCategories.html#//apple_ref/doc/uid/TP30001163-CH20-SW1`.

Implementing the Class

The implementation file ends with the extension `.m` and is where you type the code for the methods of the class. The implementation file of the `Planet` class is listed here:

```
#import "Planet.h"

@implementation Planet

- (void)spinOnAxis
{

}

- (float)distanceFromSun
{

}

- (void)rotatePlanetByAngle:(float)angle
{

}

@end
```

With the introduction of iOS SDK 5.0, it is now possible to declare member variables in the implementation file (`.m`*) instead of the interface (*`.h`*) file. The benefit of doing this is that the* `.h` *file truly hides the implementation details, and only exposes the methods made available by the class. The member variables of a class, while necessary, are just implementation details. The principle of encapsulation dictates that classes should try as much as possible to hide implementation details from their users. To this end you could move the declaration of instance variable* `surfaceTemperature` *into the* `.m` *file. If you do so, then your* `Planet.h` *file will look like this:*

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

@interface Planet : NSObject

- (void)spinOnAxis;
- (float)distanceFromSun;
- (void)rotatePlanetByAngle:(float)angle;

@end
```

The corresponding implementation file `Planet.m` *will now contain the declaration of the instance variable and will look like this:*

```
#import "Planet.h"

@implementation Planet
{
    float surfaceTemperature;
}

- (void)spinOnAxis
{

}

- (float)distanceFromSun
{

}

- (void)rotatePlanetByAngle:(float)angle
{

}

@end
```

Whether you should opt to do things thing way or not is a matter of personal choice. This is a new recommendation and a lot of code that you are likely to encounter in the near future will not have adapted it.

Furthermore, you do not need to declare instance variables for synthesized properties. Synthesized properties are discussed later in this appendix.

The implementation file also begins with the `#import` directive. An implementation file must always include the corresponding header file.

Implementations for each method of the class are placed between the `@implementation` and `@end` keywords:

```
#import "Planet.h"

@implementation Planet
….
….
@end
```

The implementation of a method looks similar to its declaration in the header file, except that the semicolon at the end of the declaration is replaced by a pair of curly braces, `{` and `}`.

Sending Messages to Objects

In Objective-C terminology, the act of using a method on an object is known as sending a message to the object. Thus you don't call a method, you send the object a message.

If `secondPlanet` was an instance of the `Planet` class (we will not consider the problem of how to instantiate the object just yet), then to use the `spinOnAxis` method, you would send it the `spinOnAxis` message as shown in the following code snippet:

```
.. assume secondPlanet is an instance of Planet
...

[secondPlanet spinOnAxis];
```

An Objective-C statement that sends a message to an object is enclosed in square brackets. The recipient of the message (which is usually an object) comes first, followed by a space, followed by the name of the method. This is depicted in Figure B-19.

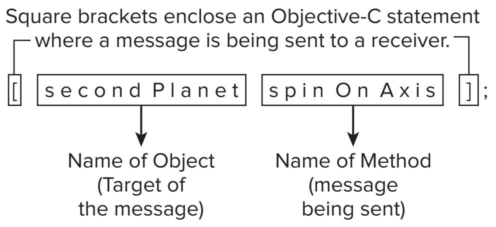

Square brackets enclose an Objective-C statement where a message is being sent to a receiver.
[second Planet spin On Axis];
Name of Object (Target of the message)
Name of Method (message being sent)

FIGURE B-19

If the message being sent (method being called) returned a value, you will need to collect this value in a variable of the appropriate type:

```
float result = [secondPlanet distanceFromSun];
```

If the message requires you to include an input value (parameter), you will need to specify an appropriate value (depending on the data type of the parameter) after the method name. This can be done as follows:

```
[secondPlanet rotatePlanetByAngle:16.88];
```

Compare this to the definition of the `rotatePlanetByAngle` method:

```
- (void)rotatePlanetByAngle:(float)angle;
```

You will see that the value `16.88` will be copied into a variable named `angle`. This variable `angle` will be used within the block of code that makes up the function.

You can insert a comment line in your code by starting the line with `//`. The Objective-C compiler ignores everything in a line that begins with `//`.If you want your comments to span across multiple lines, you can either place `//` in front of each line, or create a block comment by putting your comments between a pair of `/*` and `*/` characters, as shown here:

```
/*
this is a comment that spans
multiple lines, and will be ignored by
the Objective-C complier.
*/
```

Instantiating Objects

To instantiate an object of an existing class, you use the `alloc` and `init` methods as shown in the following code snippet.

```
Planet* newPlanet = [[Planet alloc] init];
```

The `alloc` and `init` methods are defined in `NSObject` as:

```
+ (id)alloc;
- (id)init;
```

It is common for classes to override the init method. If a class does not specifically override the init method, the base class' version will be used instead.

Objective-C classes also provide overloaded versions of the `init` method that accept one or more arguments. These overloaded methods internally call the `init` method of the parent class. As an example, the `NSString` class provides several overloaded versions of the `init` method including `initWithString:`, `initWithFormat:` and others. The following code shows how a new instance of an `NSString` object is created using the `alloc` and `initWithString:` methods:

```
NSString* userName = [[NSString alloc] initWithString:@"Roy Wicks"];
```

For more information on instantiating objects, refer to "The Objective-C Programming Language Guide" available at: `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocCategories.html#//apple_ref/doc/uid/TP30001163-CH20-SW1`.

Objective-C Properties

The principle of encapsulation requires that in order to use an object, you should not need to know how the object works. If a class were well designed, then in order to use it, you only need to know of a list of methods provided by the object.

Member variables, although an essential component of an object, are part of the object's internal details and thus are not accessible to users directly. The interface of the `Planet` class that has been used in the previous sections of this appendix is presented once again:

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

@interface Planet : NSObject
{
    float surfaceTemperature;
}

- (void)spinOnAxis;
- (float)distanceFromSun;
- (void)rotatePlanetByAngle:(float)angle;

@end
```

In order to give other classes (including subclasses) the ability to read/change the value of the instance variable `surfaceTemperature`, you will need to add a pair of accessor methods to the class. A method that is used to read the value of an instance variable is known as a getter, and a method used to change an instance variable is called a setter. It turns out that accessor methods are created quite frequently, and in order to simplify the process of creating these accessors, Objective-C provides the concept of *synthesized properties*.

Essentially, you declare a property in your class to expose a instance variable to your users, and the compiler generates the required getter/setter method for you.

Thus you can expose the `surfaceTemperature` variable to your users by adding this statement to your class declaration, between the closing `}` and the `@end` statement:

```
@property float surfaceTemperature;
```

You can think of adding the above `@property` declaration statement to be equivalent to adding two methods:

```
- (float)surfaceTemperature;
- (void)setSurfaceTemperature:(float)newValue;
```

The updated `Planet.h` file, with a property called `surfaceTemperature` will look like this:

```
@interface Planet : NSObject
{
    float surfaceTemperature;
}

@property float surfaceTemperature;
- (void)spinOnAxis;
- (float)distanceFromSun;
- (void)rotatePlanetByAngle:(float)angle;

@end
```

You also need to add an `@synthesize` statement in the `.m` file for each corresponding property you have added to the `.h` file. The `@synthesize` statement is added at the top of the implementation (`.m`) file, immediately after the `@implementation` line as shown here:

```
@implementation Planet

@synthesize surfaceTemperature;

// other method declarations follow …

@end
```

The net effect of a pair of `@property` and `@synthesize` statements is to create both a getter and a setter method for the member variable in question. The Objective-C language also allows you to use a few modifiers with `@property` statements. When used, these modifiers create slightly different setter/getter methods.

The most common `@property` modifiers are `nonatomic` and `readonly`. Using the `nonatomic` modifier results in code that is slightly faster. A `readonly` property is one that can only be read, thus only a getter method is created for one of these. Objective-C properties are not read-only by default.

Applying these modifiers to a property is a simple matter of including the appropriate keywords in the `@property` statement:

```
@property (readonly) float surfaceTemperature;
```

It is common practice, when exposing an instance variable using synthesized properties, to not declare the instance variable in the `.h` file at all. This is because when the Objective-C compiler comes across an `@property` declaration without a matching instance variable declaration, it will create an instance variable automatically. Thus, the final version of the `Planet.h` file would resemble:

```
@interface Planet : NSObject
@property float surfaceTemperature;
- (void)spinOnAxis;
- (float)distanceFromSun;
- (void)rotatePlanetByAngle:(float)angle;

@end
```

Last but not least, it is worth mentioning that you could override the default getter/setter method generated by the compiler with one of your own. Thus, if you wanted to use the standard compiler-generated setter method, but override the getter method, you will need to add an implementation for the `setSurfaceTemperature:` method in the `.m` file:

```
- (void)setSurfaceTemperature:(float)newValue
{
    // do something with newValue
}
```

This concludes the introduction to Objective-C. For more information you are encouraged to read "The Objective-C Programming Language Guide" available at: `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html`.

C

Introduction to ARC

Automatic Reference Counting (ARC) is a feature of the new LLVM 3.0 compiler that does away with manual memory management of Objective-C objects. From a programmer's perspective, enabling ARC in your project implies you do not need to call `retain`, `release`, or `autorelease` on your Objective-C objects. In fact, you will get a compile-time error if you attempt to do so. Instead, the compiler evaluates the lifetime of each object and inserts appropriate method calls for you. The compiler also generates appropriate `dealloc` methods for you.

It is important to keep in mind that ARC is not an addition to the Objective-C language. It is a compile-time feature. All ARC does is insert appropriate `release` and `retain` calls in your code where you would have. Another point worth mentioning is that ARC is not garbage collection. Garbage collection is an Objective-C language-level feature and is not supported on iOS.

In this appendix, you learn how to use ARC in your projects.

OBJECT OWNERSHIP

To use ARC in your projects you need to move away from the retain-release model and think in terms of object ownership. This is best explained with an example. The following code allocates memory for and initializes an `NSString` object:

```
NSString* firstName = @"Andrew";
```

When the preceding statement is executed, the variable `firstName` points to the location in memory where the `NSString` instance is located, and effectively owns the contents of that memory location. In fact, `firstName` is the only reference to that memory location, and keeps the memory "alive." This is illustrated in Figure C-1.

This kind of reference (which implies ownership) is called a *strong reference*, and is the default type of reference for all instance variables, local variables, and property declarations.

To specify a strong reference, you can apply the `__strong` prefix at the beginning of an assignment statement, although this is not strictly necessary because assignments are, by default, strong references. The equivalent statement with the `__strong` prefix would have been:

```
__strong NSString* firstName = @"Andrew";
```

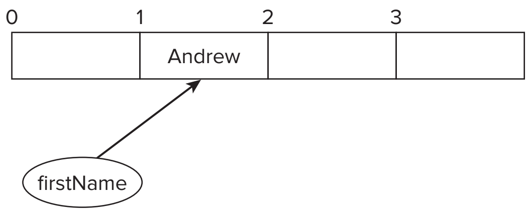


FIGURE C-1

It is possible (though not particularly useful in this case) to create another variable that points to the same memory as `firstName`. You can do this by using the following statement:

```
NSString* anotherName = firstName;
```

The situation is depicted in Figure C-2. At this point there are two strong references to the same memory location. ARC-compliant code can have multiple strong references to the same memory.

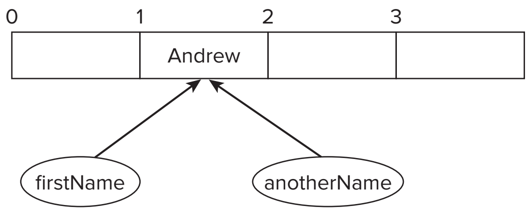


FIGURE C-2

If you were to now reassign `firstName` using the following statement:

```
firstName = @"Johnson";
```

the situation in memory changes, and is depicted in Figure C-3. Note that both `firstName` and `anotherName` are now strong references to different blocks of memory. The memory for the block containing the string Andrew was not freed because there is still a strong reference to the block (in the variable `anotherName`).

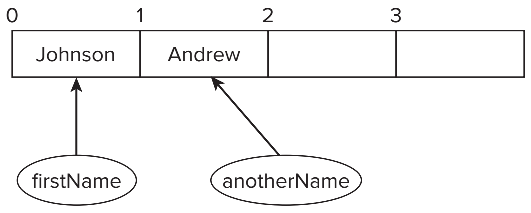


FIGURE C-3

This is important to note. Memory will not be freed as long as there is a single strong reference to that block. If you were to once again reassign `firstName` using the following code:

```
firstName = @"Kirk";
```

the block of memory containing the string Johnson would indeed be freed because there is no longer a strong reference to it. This is depicted in Figure C-4.

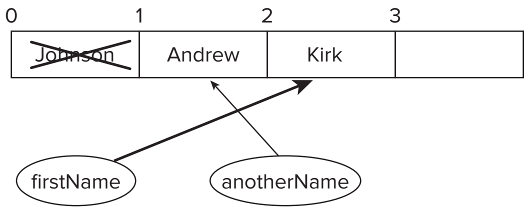


FIGURE C-4

ARC also introduces the concept of a weak reference. A weak reference does not signify ownership. It is merely a reference and does not count toward keeping a block of memory "alive." You create a weak reference by prefixing an assignment with the `__weak` qualifier:

```
__weak NSString* weakReference = firstName;
```

When the block of memory referenced by a weak reference is destroyed (because there are no strong references keeping the block alive), weak references are automatically set to `nil`. Such a reference is referred to as a *zeroing weak reference* in ARC terminology.

Weak references aren't used very often; they are mostly used when two objects have a parent-child relationship. The parent holds a strong reference to the child, thus keeping the child object from being deallocated. The child, on the other hand, keeps a weak reference to the parent. A common example of this situation would be when you create a `UITextField` object in your view controller class and set its delegate property to `self`. Your view controller holds a strong reference to the `UITextField` instance; however, the text field holds a weak reference to the view controller (via its `delegate` property). This situation is depicted in Figure C-5.

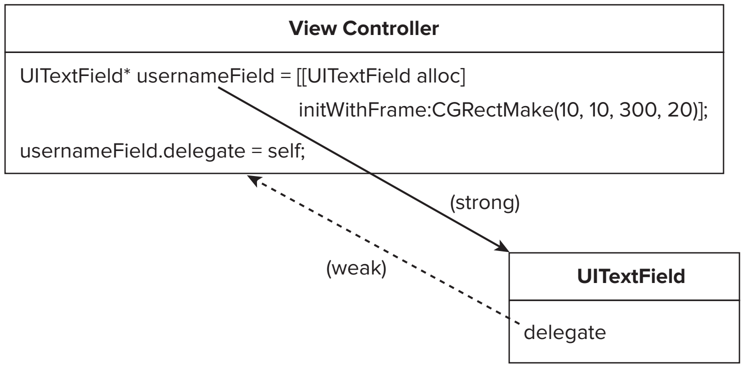

View Controller
UITextField* usernameField = [[UITextField alloc]
initWithFrame:CGRectMake(10, 10, 300, 20)];
usernameField.delegate = self;
(strong)
(weak)
UITextField
delegate

FIGURE C-5

Just like variables, properties can also be strong or weak. For example:

```
@property (nonatomic, strong) NSString* strongReference;
@property (nonatomic, weak) NSString* weakReference;
```

When creating outlets, the properties added to your class by Interface Builder are weak, so that they will automatically be set to `nil` when the objects they reference go out of scope.

ARC is a great feature and helps reduce clutter in your code. However, it does have its limitations, the most important being that ARC works only with Objective-C objects.

If your code uses Core Foundation or allocates memory with the C-function `malloc`, you are still responsible for managing that memory. Another point to remember is to set strong references to `nil` when you do not need the references. This is because if you keep holding on to all the objects you have allocated with strong references, ARC will never be able to free the memory, and eventually your application may run out of memory.

CONVERTING PROJECTS TO ARC

When you create a new project in Xcode, you have the option to enable the use of Automatic Reference Counting in the Project Options dialog box. However, there may be instances when you want to enable ARC in an existing project that is not currently ARC enabled. To do this, you could either use the automatic conversion tool included with Xcode or attempt to convert the project manually. This section presents a simple project that is made ARC compliant using the automatic conversion tool. The project is a simple view-based application called `FruitTap` that displays images of four fruits and an alert view with the name of the fruit when the user taps one of them. Figure C-6 shows this simple application in action.

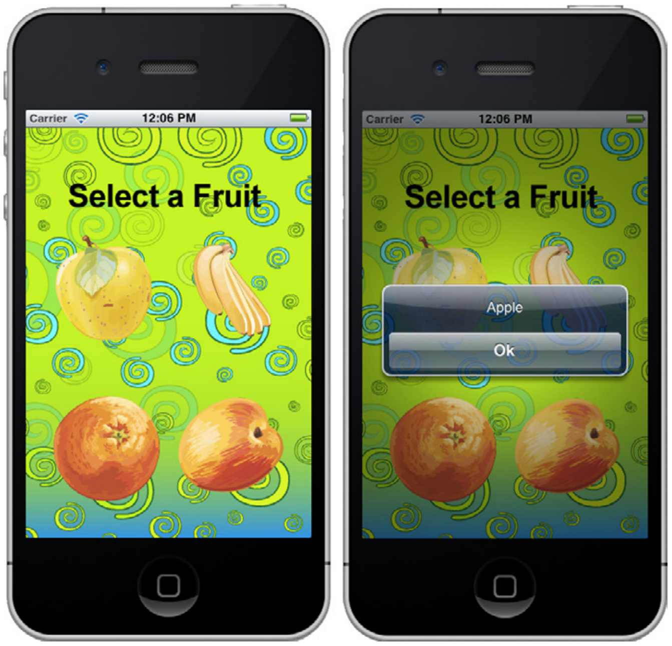


FIGURE C-6

The user interface for the application is created programmatically in the `viewDidLoad` method of the view controller class. Listings C-1 to C-4 contain the source code for the application before it is converted to use ARC.

You can download the code and resources for this appendix from the book's web page at `www.wrox.com`*. You can find them in the Appendix C folder in the download.*

LISTING C-1: The AppDelegate.h file

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

@class ViewController;

@interface AppDelegate :  NSObject <UIApplicationDelegate>

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

@end
```

LISTING C-2: The AppDelegate.m file

```
#import "AppDelegate.h"
#import "ViewController.h"

@implementation AppDelegate

@synthesize window;
@synthesize viewController;

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

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // add the view controller's view.
    [window addSubview:[viewController view]];
    [window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
```

continues

LISTING C-2 *(continued)*

```
{

}
- (void)applicationDidEnterBackground:(UIApplication *)application
{

}
- (void)applicationWillEnterForeground:(UIApplication *)application
{

}

- (void)applicationDidBecomeActive:(UIApplication *)application
{

}

- (void)applicationWillTerminate:(UIApplication *)application
{

}

@end
```

LISTING C-3: The ViewController.h file

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

@interface ViewController : UIViewController

@property (nonatomic, retain) UIImageView* backgroundImage;
@property (nonatomic, retain) UIImageView* appleImage;
@property (nonatomic, retain) UIImageView* bananaImage;
@property (nonatomic, retain) UIImageView* orangeImage;
@property (nonatomic, retain) UIImageView* peachImage;
@property (nonatomic, retain) UITapGestureRecognizer* tapRecognizer;

- (void) handleTap:(id)sender;

@end
```

LISTING C-4: The ViewController.m file

```
#import "ViewController.h"

@implementation ViewController

@synthesize backgroundImage = _backgroundImage;
@synthesize appleImage = _appleImage;
```

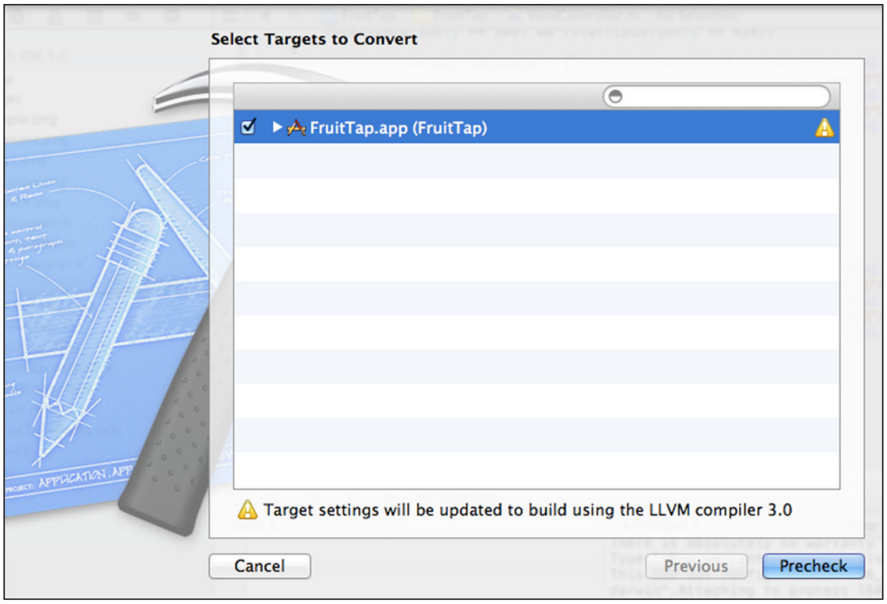
```
@synthesize bananaImage = _bananaImage;
@synthesize orangeImage = _orangeImage;
@synthesize peachImage = _peachImage;
@synthesize tapRecognizer = _tapRecognizer;

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}

#pragma mark - View lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];

    _backgroundImage = [[UIImageView alloc] initWithImage:[UIImage
                                           imageNamed:@"bg1.png"]];
    _backgroundImage.frame = CGRectMake(0, -20, 320, 480);
    [self.view addSubview:_backgroundImage];

    _appleImage = [[UIImageView alloc] initWithImage:[UIImage
                                         imageNamed:@"apple.png"]];
    _appleImage.frame = CGRectMake(30, 120, 120, 120);
    [self.view addSubview:_appleImage];

    _bananaImage = [[UIImageView alloc] initWithImage:[UIImage
                                    imageNamed:@"banana.png"]];
    _bananaImage.frame = CGRectMake(170, 120, 120, 120);
    [self.view addSubview:_bananaImage];

    _orangeImage = [[UIImageView alloc] initWithImage:[UIImage
                                    imageNamed:@"orange.png"]];
    _orangeImage.frame = CGRectMake(30, 300, 120, 120);
    [self.view addSubview:_orangeImage];

    _peachImage = [[UIImageView alloc] initWithImage:[UIImage
                                     imageNamed:@"peach.png"]];
    _peachImage.frame = CGRectMake(170, 300, 120, 120);
    [self.view addSubview:_peachImage];

    _tapRecognizer = [[UITapGestureRecognizer alloc]
                                             initWithTarget:self
                                             action:@selector(handleTap:)];
    _tapRecognizer.cancelsTouchesInView = NO;
    [self.view addGestureRecognizer:_tapRecognizer];
}

- (void) dealloc
{
    [_backgroundImage release];
    [_appleImage release];
    [_bananaImage release];
    [_orangeImage release];
    [_peachImage release];
```

continues

LISTING C-4 *(continued)*

```
    [_tapRecognizer release];

    [super dealloc];
}
- (void)viewDidUnload
{
    [super viewDidUnload];
    self.backgroundImage = nil;
    self.appleImage = nil;
    self.bananaImage = nil;
    self.orangeImage = nil;
    self.peachImage = nil;
    self.tapRecognizer = nil;
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
}
- (void)viewDidDisappear:(BOOL)animated
{
   [super viewDidDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}

- (void) handleTap:(id)sender
{
    CGPoint startLocation = [sender locationInView:self.view];


    if ((startLocation.x >= 30) && (startLocation.x <= 150) &&
        (startLocation.y >= 120) && (startLocation.y <= 240))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                         message:@"Apple"
                                                         delegate:nil
                                                         cancelButtonTitle:@"Ok"
                                                         otherButtonTitles:nil];
        [alertMessage show];
        [alertMessage release];
        return;
    }


    if ((startLocation.x >= 170) && (startLocation.x <= 290) &&
        (startLocation.y >= 120) && (startLocation.y <= 240))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                        message:@"Banana"
                                                       delegate:nil
                                              cancelButtonTitle:@"Ok"
                                              otherButtonTitles:nil];
        [alertMessage show];
        [alertMessage release];
        return;
    }


    if ((startLocation.x >= 30) && (startLocation.x <= 150) &&
        (startLocation.y >= 300) && (startLocation.y <= 420))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                        message:@"Orange"
                                                       delegate:nil
                                              cancelButtonTitle:@"Ok"
                                              otherButtonTitles:nil];
        [alertMessage show];
        [alertMessage release];
        return;
    }

    if ((startLocation.x >= 170) && (startLocation.x <= 290) &&
        (startLocation.y >= 300) && (startLocation.y <= 420))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                        message:@"Peach"
                                                       delegate:nil
                                              cancelButtonTitle:@"Ok"
                                              otherButtonTitles:nil];
        [alertMessage show];
        [alertMessage release];
        return;
    }

}

@end
```

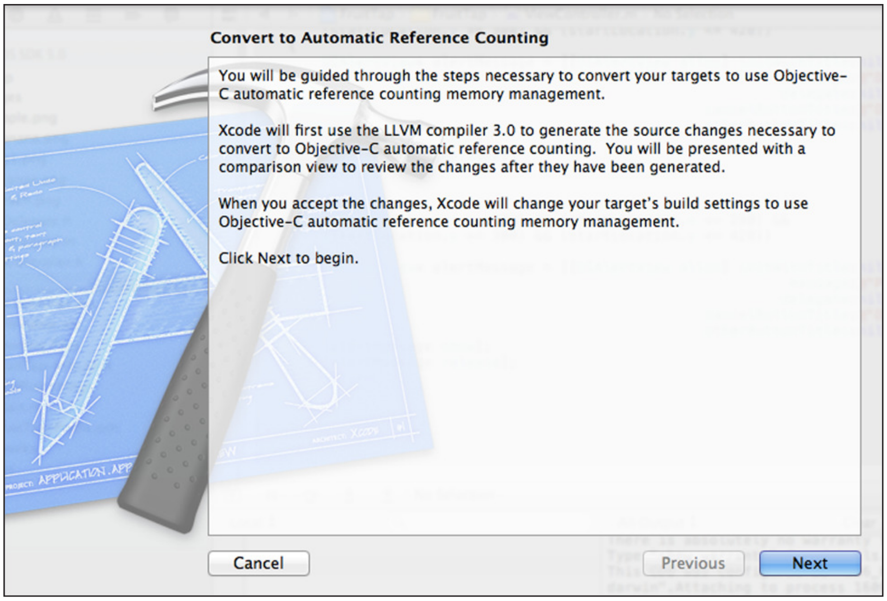
Before you begin converting the project to use ARC memory management, be sure to make a backup of the original project in case you need to start over. To begin the conversion process, select the Edit ➪ Refactor ➪ Convert to Objective-C ARC menu item in Xcode. Xcode will prompt you to select one or more build targets that need to be converted (Figure C-7). Once you have selected the appropriate targets, click the Precheck button.



Target settings will be updated to build using the LLVM compiler 3.0

FIGURE C-7

Xcode goes through the project's source files and looks for any statements in your code that it cannot convert. If no such statements are found, Xcode presents a dialog box that requires you to confirm your intent to upgrade the selected targets to use ARC memory management (Figure C-8). Click the Next button.



You will be guided through the steps necessary to convert your targets to use Objective-C automatic reference counting memory management.

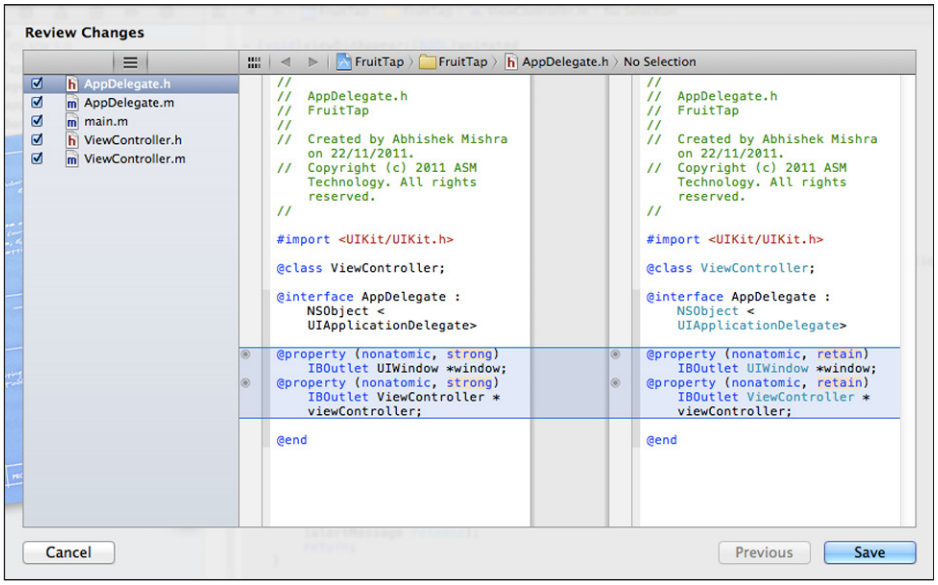
Xcode will first use the LLVM compiler 3.0 to generate the source changes necessary to convert to Objective-C automatic reference counting. You will be presented with a comparison view to review the changes after they have been generated.

When you accept the changes, Xcode will change your target's build settings to use Objective-C automatic reference counting memory management.

Click Next to begin.

FIGURE C-8

Once the conversion has been performed, Xcode lets you review the changes that will be made to each file (Figure C-9). If you are not happy with the changes, you can click the Cancel button and your project will remain in its original, unconverted state.

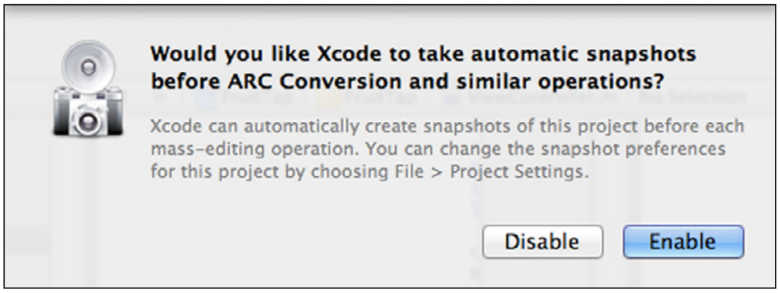


```
//
//  AppDelegate.h
//  FruitTap
//
//  Created by Abhishek Mishra
    on 22/11/2011.
//  Copyright (c) 2011 ASM
    Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@class ViewController;

@interface AppDelegate :
    NSObject <
    UIApplicationDelegate>

@property (nonatomic, strong)
    IBOutlet UIWindow *window;
@property (nonatomic, strong)
    IBOutlet ViewController *
    viewController;

@end
```

```
//
//  AppDelegate.h
//  FruitTap
//
//  Created by Abhishek Mishra
    on 22/11/2011.
//  Copyright (c) 2011 ASM
    Technology. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@class ViewController;

@interface AppDelegate :
    NSObject <
    UIApplicationDelegate>

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

@end
```

FIGURE C-9

If you are satisfied with the changes Xcode will make, click the Save button. Xcode asks you to take a snapshot of the project before proceeding; this is in general a good idea, and you should click the Enable button (Figure C-10). If you ever need to go back to the original version of the project before the conversion process, you can find the snapshot in the Xcode Organizer.



Would you like Xcode to take automatic snapshots before ARC Conversion and similar operations?

Xcode can automatically create snapshots of this project before each mass-editing operation. You can change the snapshot preferences for this project by choosing File > Project Settings.

FIGURE C-10

In the `FruitTap` example, the conversion process has no impact on the `AppDelegate.m` file. Listings C-5 to C-7 contain the source code for the `AppDelegate.h`, `ViewController.h`, and `ViewController.m` files after they have been converted to use ARC.

LISTING C-5: The AppDelegate.h file after ARC conversion

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

@class ViewController;

@interface AppDelegate :  NSObject <UIApplicationDelegate>

@property (nonatomic, strong) IBOutlet UIWindow *window;
@property (nonatomic, strong) IBOutlet ViewController *viewController;

@end
```

LISTING C-6: The ViewController.h file after ARC conversion

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

@interface ViewController : UIViewController

@property (nonatomic, strong) UIImageView* backgroundImage;
@property (nonatomic, strong) UIImageView* appleImage;
@property (nonatomic, strong) UIImageView* bananaImage;
@property (nonatomic, strong) UIImageView* orangeImage;
@property (nonatomic, strong) UIImageView* peachImage;
@property (nonatomic, strong) UITapGestureRecognizer* tapRecognizer;

- (void) handleTap:(id)sender;

@end
```

LISTING C-7: The ViewController.m file after ARC conversion

```
#import "ViewController.h"

@implementation ViewController

@synthesize backgroundImage = _backgroundImage;
@synthesize appleImage = _appleImage;
@synthesize bananaImage = _bananaImage;
@synthesize orangeImage = _orangeImage;
@synthesize peachImage = _peachImage;
@synthesize tapRecognizer = _tapRecognizer;

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}

#pragma mark - View lifecycle
- (void)viewDidLoad
```

```
{
    [super viewDidLoad];

    _backgroundImage = [[UIImageView alloc]
          initWithImage:[UIImage imageNamed:@"bg1.png"]];
    _backgroundImage.frame = CGRectMake(0, -20, 320, 480);
    [self.view addSubview:_backgroundImage];

    _appleImage = [[UIImageView alloc]
         initWithImage:[UIImage imageNamed:@"apple.png"]];
    _appleImage.frame = CGRectMake(30, 120, 120, 120);
    [self.view addSubview:_appleImage];

    _bananaImage = [[UIImageView alloc]
         initWithImage:[UIImage imageNamed:@"banana.png"]];
    _bananaImage.frame = CGRectMake(170, 120, 120, 120);
    [self.view addSubview:_bananaImage];

    _orangeImage = [[UIImageView alloc]
         initWithImage:[UIImage imageNamed:@"orange.png"]];
    _orangeImage.frame = CGRectMake(30, 300, 120, 120);
    [self.view addSubview:_orangeImage];

    _peachImage = [[UIImageView alloc]
         initWithImage:[UIImage imageNamed:@"peach.png"]];
    _peachImage.frame = CGRectMake(170, 300, 120, 120);
    [self.view addSubview:_peachImage];

    _tapRecognizer = [[UITapGestureRecognizer alloc]
                                        initWithTarget:self
                                        action:@selector(handleTap:)];
    _tapRecognizer.cancelsTouchesInView = NO;
    [self.view addGestureRecognizer:_tapRecognizer];
}
- (void)viewDidUnload
{
    [super viewDidUnload];
    self.backgroundImage = nil;
    self.appleImage = nil;
    self.bananaImage = nil;
    self.orangeImage = nil;
    self.peachImage = nil;
    self.tapRecognizer = nil;
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
}
```

continues

LISTING C-7 *(continued)*

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];

}

- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}

- (void) handleTap:(id)sender
{
    CGPoint startLocation = [sender locationInView:self.view];


    if ((startLocation.x >= 30) && (startLocation.x <= 150) &&
        (startLocation.y >= 120) && (startLocation.y <= 240))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                 message:@"Apple"
                                                 delegate:nil
                                                 cancelButtonTitle:@"Ok"
                                                 otherButtonTitles:nil];
        [alertMessage show];
        return;
    }


    if ((startLocation.x >= 170) && (startLocation.x <= 290) &&
        (startLocation.y >= 120) && (startLocation.y <= 240))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                       message:@"Banana"
                                                      delegate:nil
                                             cancelButtonTitle:@"Ok"
                                             otherButtonTitles:nil];
        [alertMessage show];
        return;
    }


    if ((startLocation.x >= 30) && (startLocation.x <= 150) &&
        (startLocation.y >= 300) && (startLocation.y <= 420))
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                       message:@"Orange"
```

```
                                                                    delegate:nil
                                                           cancelButtonTitle:@"Ok"
                                                           otherButtonTitles:nil];
            [alertMessage show];
            return;
        }

        if ((startLocation.x >= 170) && (startLocation.x <= 290) &&
            (startLocation.y >= 300) && (startLocation.y <= 420))
        {
            UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                                     message:@"Peach"
                                                                    delegate:nil
                                                           cancelButtonTitle:@"Ok"
                                                           otherButtonTitles:nil];
            [alertMessage show];
            return;
        }

    }

    @end
```

When using ARC memory management, you cannot manage autorelease pools with the `NSAutoReleasePool` class. Instead, ARC introduces a new statement construct to the Objective-C language called `@autoreleasepool`. One place in your application where this new construct is used is the `main.m` file. The modified version of this file is listed here:

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

int main(int argc, char *argv[])
{
    @autoreleasepool {
        int retVal = UIApplicationMain(argc, argv, nil, nil);
        return retVal;
    }
}
```

The conversion process doesn't always go smoothly. In many cases you will need to tweak your code and try the automatic conversion process again. Some of the more common conversion errors you are likely to encounter are:

➤ ARC forbids Objective-C objects in structs or unions.

  You are getting this error because you have a C-style struct that contains Objective-C pointers. You will need to update these structures to Objective-C classes. An example of code that will cause this error is:

```
typedef struct
{
    NSString *firstName;
    NSString *lastName;
}
ContactDetails;
```

- Switch case is in protected scope.

  You are getting this error because your code is creating a new pointer variable inside a case statement. An example of this kind of code is:

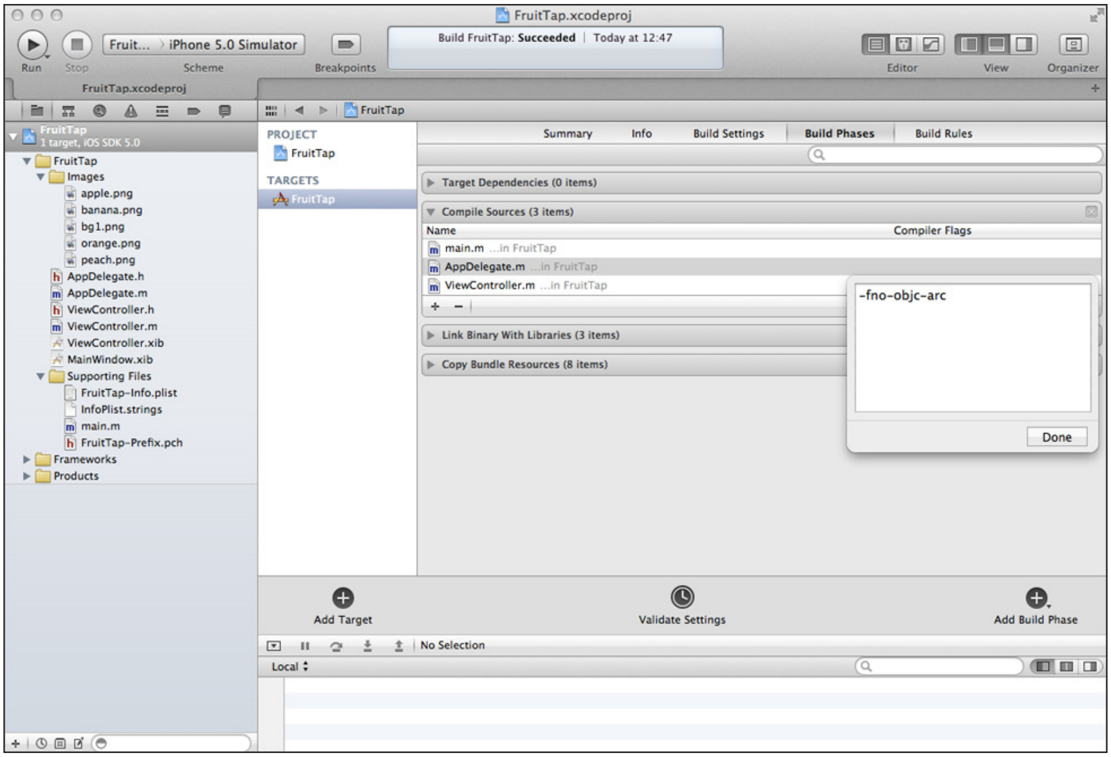
```
switch (numberOfTaps)
{
    case 1:
        UIAlertView* alertMessage = [[UIAlertView alloc]
                                     initWithTitle:nil
                                     message:messageText
                                     delegate:nil
                                     cancelButtonTitle:@"Ok"
                                     otherButtonTitles:nil];
            [alertMessage show];
            break;
        default:
            break;
    }
 }
```

  To correct this problem you will need to enclose the body of the case statements in its own block as shown here:

```
switch (numberOfTaps)
{
    case 1:
    {
        UIAlertView* alertMessage = [[UIAlertView alloc] initWithTitle:nil
                                                      message:messageText
                                                      delegate:nil
                                                  cancelButtonTitle:@"Ok"
                                                  otherButtonTitles:nil];
        [alertMessage show];
        break;
    }
    default:
        break;
}
```

If one or more files throw a lot of errors during the conversion process, you can always disable ARC memory management for specific files in your project. To do so, select the project node in the project navigator and switch to the Build Phases tab. Open the Compile Sources group to reveal the source file list. Double-click the file for which you want to disable ARC and type `-fno-objc-arc` in the popup window (Figure C-11).

When you create new projects you should try to use ARC memory management right from the start. However, because ARC is a new concept, you may have to use code that is not ARC ready every now and then. Hopefully this appendix will give you the basic knowledge to use ARC memory management in your projects.



-fno-objc-arc

FIGURE C-11

For more information on how to handle ARC conversion issues, read the Transitioning to ARC Release Notes available at `http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/_index.html`.

D

Testing on an iOS Device

The iOS Simulator is a handy tool for testing your application as you are developing it. However, it is no substitute for testing on an actual device—certain features, like the accelerometer, cannot be tested on the simulator at all.

Testing your application on your device is slightly different from giving it to a small number of users for beta testing. When it is your own device, you can physically connect it to your Mac and use Xcode to test/debug your app while it executes on the device. Distributing your app to a few users for beta testing is achieved through Ad Hoc distribution—a process covered in detail in Appendix E.

Before you can test your app on a device, you need to prepare the device for testing and configure a few options in Xcode. The process itself can seem quite complicated at first. This appendix goes through the various steps required to test your apps on a device with Xcode.

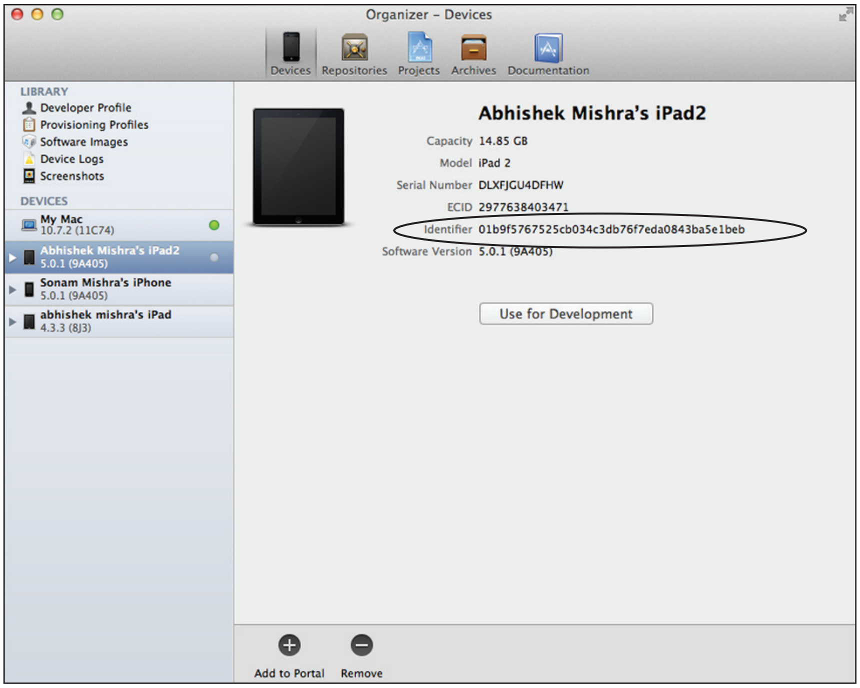
OBTAINING AND REGISTERING UDIDS

Each iOS device has a unique 40-digit identifier, commonly referred to as the device UDID. Before you can test your app on a device with Xcode, you will need to register the UDID of that device with the iOS Provisioning Portal. You can obtain this UDID through the Xcode Organizer.

To obtain the UDID for a device, simply connect it to your Mac and access the Organizer from the Xcode toolbar. Click the device in the list on the left-hand side and note the value of the Identifier field (Figure D-1).

To register a device for development, simply click the Use for Development button in the Organizer. You will be asked to provide the Apple ID and password you used to register as an iOS developer.

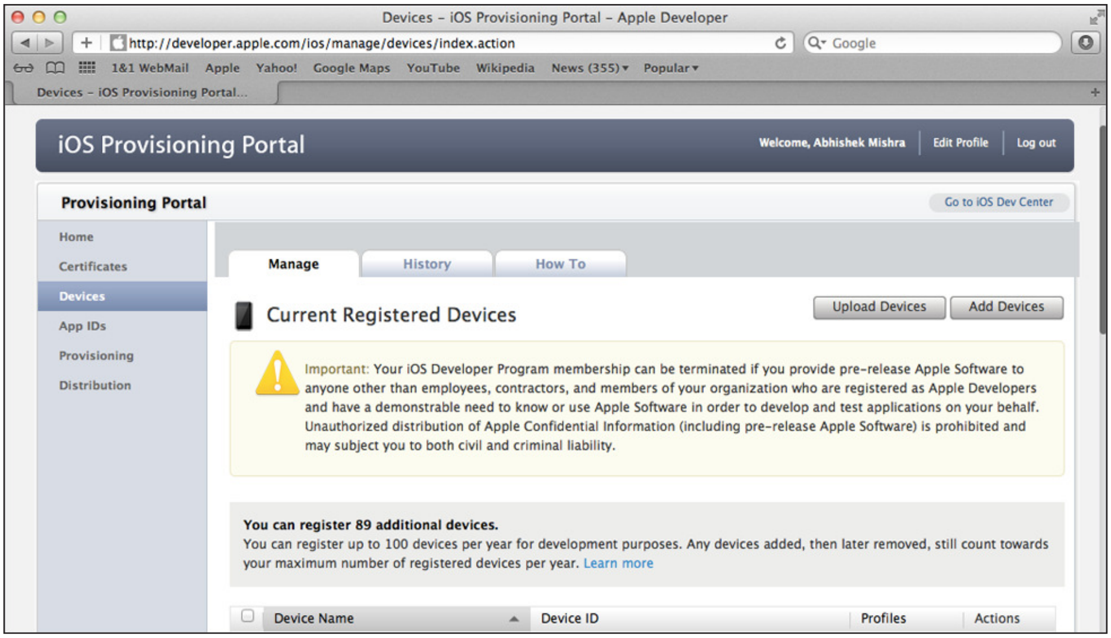
You can also register UDIDs manually. To do this you must log in to your iOS developer account at `https://developer.apple.com/ios` and click the iOS Provisioning Portal link on the right side of the page. Within the Provisioning Portal, click the Devices link on the left side (Figure D-2).



Abhishek Mishra's iPad2

Capacity 14.85 GB
Model iPad 2
Serial Number DLXFJGU4DFHW
ECID 2977638403471
Identifier 01b9f5767525cb034c3db76f7eda0843ba5e1beb
Software Version 5.0.1 (9A405)

FIGURE D-1



Important: Your iOS Developer Program membership can be terminated if you provide pre-release Apple Software to anyone other than employees, contractors, and members of your organization who are registered as Apple Developers and have a demonstrable need to know or use Apple Software in order to develop and test applications on your behalf. Unauthorized distribution of Apple Confidential Information (including pre-release Apple Software) is prohibited and may subject you to both civil and criminal liability.

You can register 89 additional devices.
You can register up to 100 devices per year for development purposes. Any devices added, then later removed, still count towards your maximum number of registered devices per year. Learn more

Device Name	Device ID	Profiles	Actions

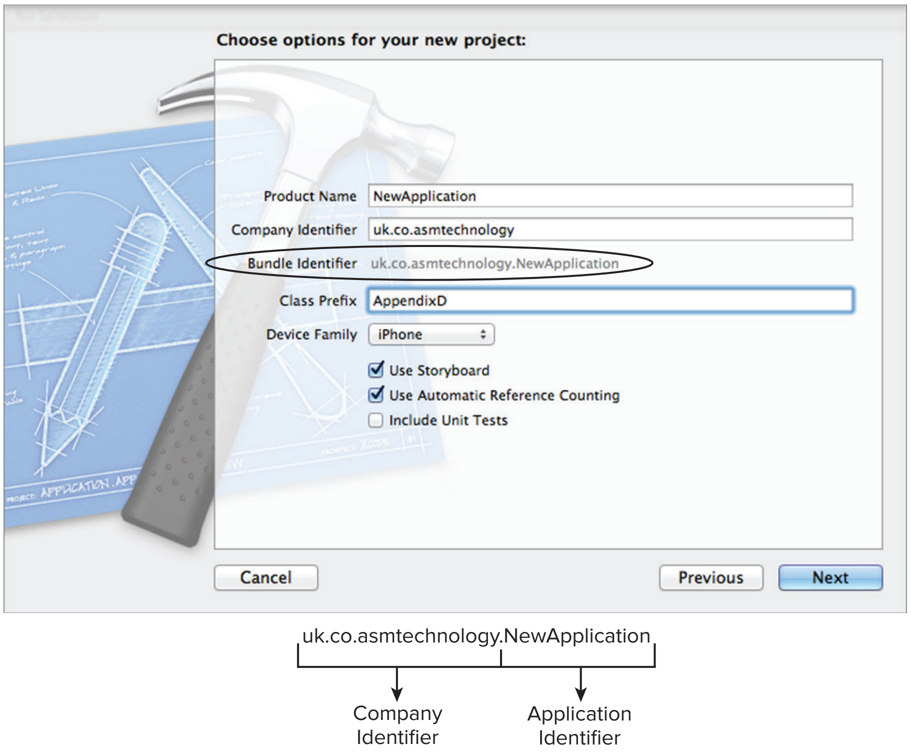
FIGURE D-2

The Devices screen shows you a list of devices registered to your account. You can register up to 100 devices a year (note that deleting a device does not count toward this limit). Click the Add Devices link and fill in the UDID of the device along with a name with which you would like to refer to the device. Click Submit to add the UDID to the device list. This list can be reset once a year, when you renew your paid membership.

CREATING AN APP ID (BUNDLE IDENTIFIER)

The next step involves creating and registering a unique identifier for your app; this is known as the App ID or Bundle Identifier. In addition to uniquely identifying your application, it allows your application to receive remote notifications, communicate with external accessories, or share keychain data with other applications in a suite.

A Bundle Identifier consists of company identifier and an application identifier (Figure D-3). When you create a new project in Xcode, you are asked to provide a company identifier, and the Bundle Identifier is generated for you by appending the name of the project to the company identifier. To distribute the application through the App Store, or through Ad Hoc distribution, the identifier you provide must match the one you register in the Provisioning Portal. You can always change the Bundle Identifier for an existing application by editing the Bundle Identifier key in the project's `info.plist` file.

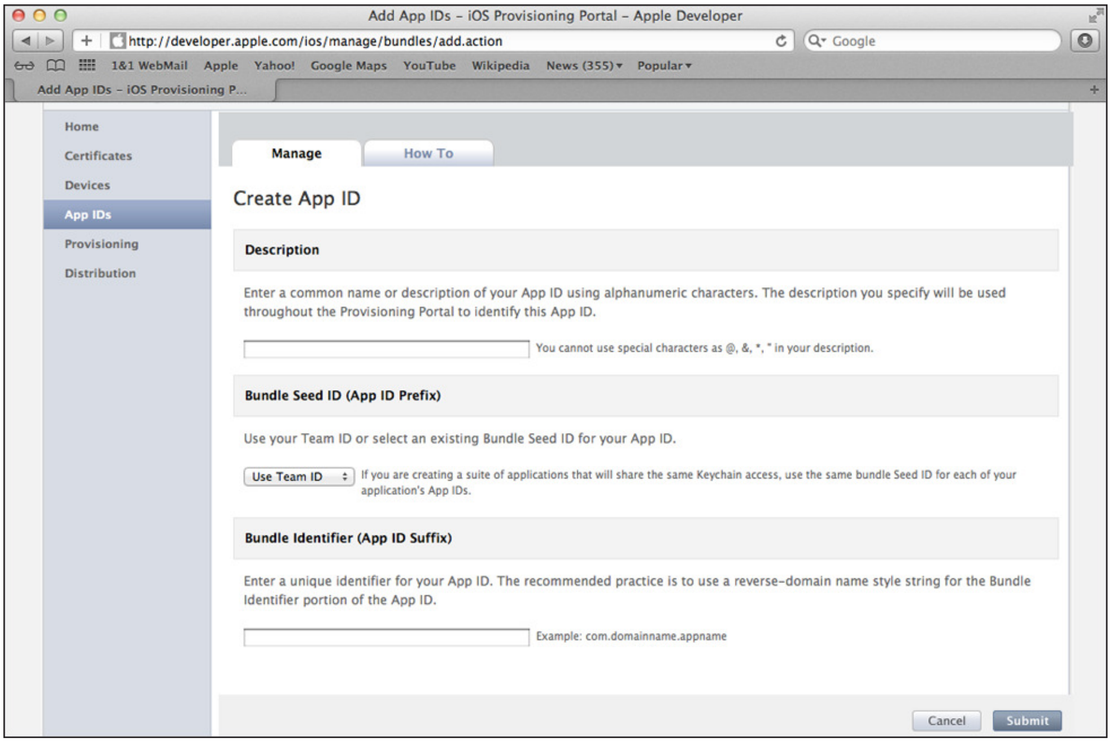


Choose options for your new project:

Product Name: NewApplication
Company Identifier: uk.co.asmtechnology
Bundle Identifier: uk.co.asmtechnology.NewApplication
Class Prefix: AppendixD
Device Family: iPhone
☑ Use Storyboard
☑ Use Automatic Reference Counting
☐ Include Unit Tests

FIGURE D-3

To register an App ID within the iOS Provisioning Portal, log in to the portal and click the App IDs link on the left-hand side. Click New App ID to access the App IDs creation screen (Figure D-4).


Add App IDs – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/bundles/add.action
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Manage
How To
Use Team ID
Cancel
Submit

Create App ID

Description

Enter a common name or description of your App ID using alphanumeric characters. The description you specify will be used throughout the Provisioning Portal to identify this App ID.

You cannot use special characters as @, &, *, " in your description.

Bundle Seed ID (App ID Prefix)

Use your Team ID or select an existing Bundle Seed ID for your App ID.

If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.

Bundle Identifier (App ID Suffix)

Enter a unique identifier for your App ID. The recommended practice is to use a reverse-domain name style string for the Bundle Identifier portion of the App ID.

Example: com.domainname.appname

FIGURE D-4

On this screen, provide a description for the App ID, choose Generate New for the Bundle Seed ID, and provide a unique value for the Bundle Identifier. The standard convention encouraged by Apple is to use a reverse-domain name type string of the form `com.domainname.appname`. If you do not mind your apps sharing data between them, you can use an asterisk instead of the `appname`, thus creating a string of the form `com.domainname.*`.

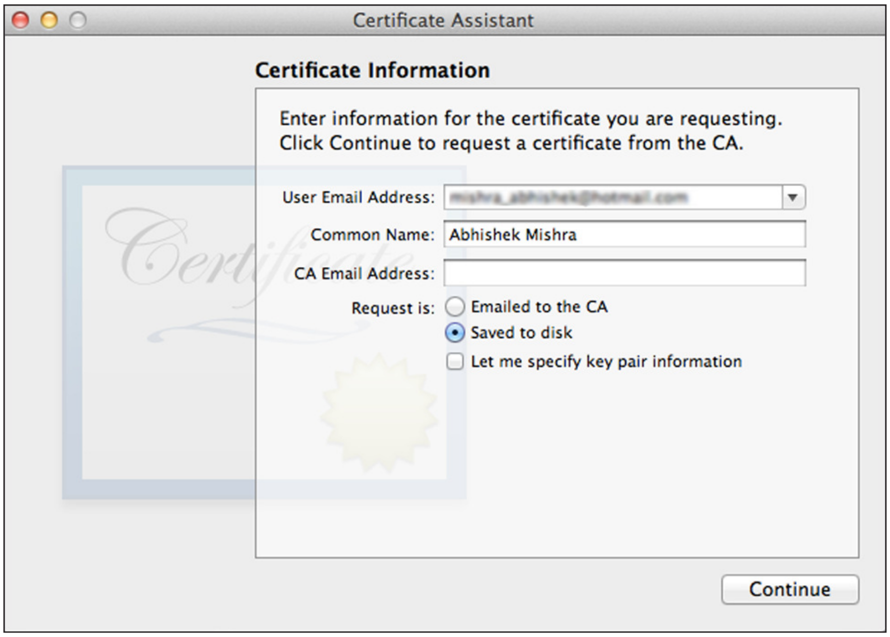
Such an App ID is called a *wildcard* App ID and can be used repeatedly across multiple applications. The downside of wildcard App IDs is that certain features such as Remote Push Notifications are not available. After you have filled up all the values on this screen, click Submit to create the new App ID.

CREATING A DEVELOPMENT CERTIFICATE

The next step involves creating and installing a development certificate. Creating a development certificate involves creating an appropriate certificate request and submitting this request to the iOS Provisioning Portal. Once the certificate is ready, you will be able to download and install it on your Mac.

To create a certificate request, launch the Keychain Access utility from the `Applications` folder on your Mac. When the Keychain Access utility is running, choose Certificate Assistant ➪ Request a Certificate from a Certificate Authority menu item.

In the Certificate Assistant dialog (Figure D-5), specify the e-mail address and account name used to access the iOS Developer Program, and ensure the Saved to Disk radio button is selected. Click the Continue button to save the certificate request as a file on your Mac.



Certificate Information

Enter information for the certificate you are requesting.
Click Continue to request a certificate from the CA.

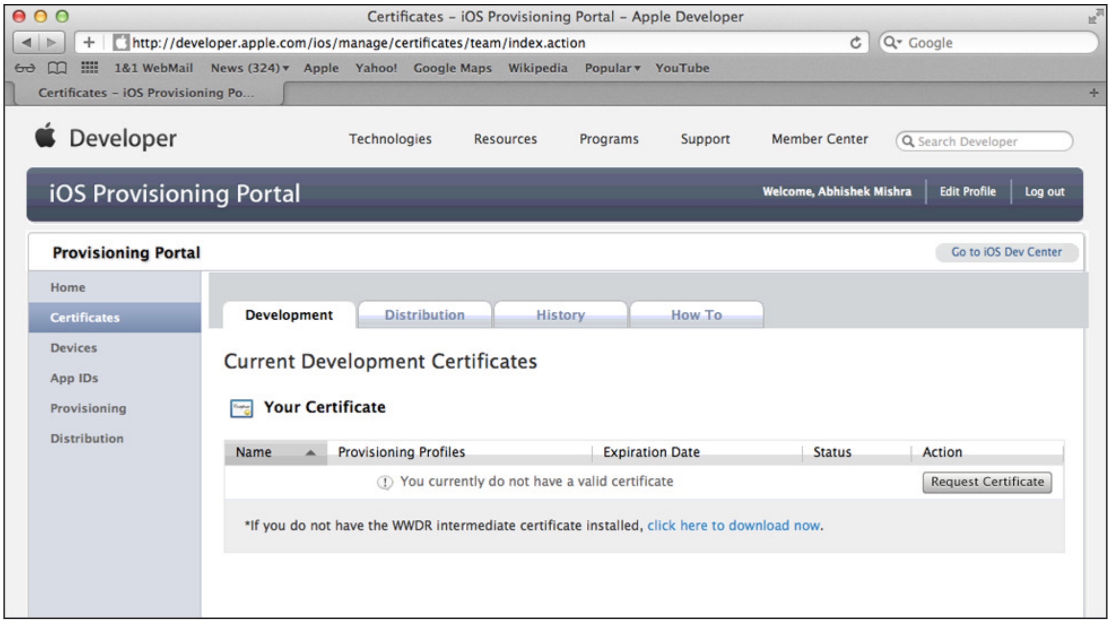
User Email Address: [redacted]
Common Name: Abhishek Mishra
CA Email Address:
Request is: Emailed to the CA [✓ Saved to disk]
☐ Let me specify key pair information

FIGURE D-5

Log in to the iOS Provisioning Portal, click the Certificates category on the left-hand side, and select the Development tab (Figure D-6). Click the Request Certificate button and then the Choose File button on the following screen. Select the certificate request file that you saved on your Mac. Click the Submit button to submit the certificate request.

If you are not part of a team, and are solely responsible for handling your iOS Developer account, your certificate is issued automatically and available to download in a few minutes. You may need to refresh your browser window. If you are part of a team, your team agent will need to first approve the certificate request. When your certificate is ready to download, you will see its status listed as Issued, and a Download link will be available.

Download the certificate and save it to your Mac; by default, the certificate should be saved to your `Downloads` folder. If you haven't done so already, download the WWDR Intermediate certificate from the same page of the iOS Provisioning Portal by using the link below the certificate. Simply double-click the certificates to install them onto your Mac.



Current Development Certificates

Your Certificate

Name	Provisioning Profiles	Expiration Date	Status	Action
	You currently do not have a valid certificate			Request Certificate

*If you do not have the WWDR intermediate certificate installed, click here to download now.

FIGURE D-6

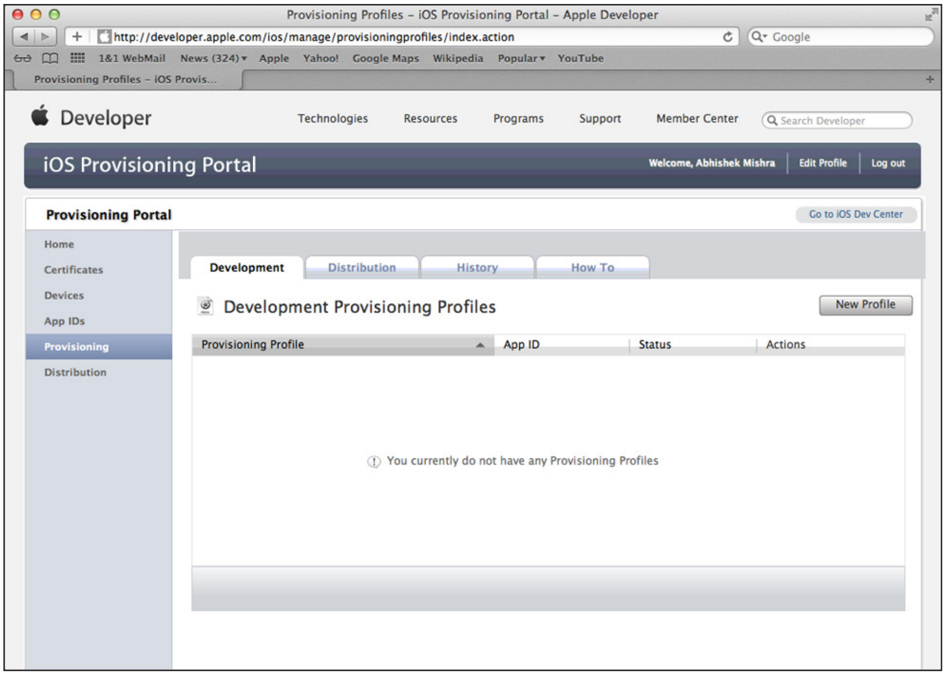
CREATING A PROVISIONING PROFILE

After having registered your device UDID, App ID, and creating a development certificate, you will need to create a development provisioning profile. A provisioning profile associates an App ID, a Certificate, and some device-specific information within Xcode. In this case the certificate in question would be the development certificate you just generated in the previous section, and the device-specific information would be a list of UDIDs on which you want to debug your application.

To create a development provisioning profile, log in to the iOS Provisioning Portal, click the Provisioning category on the left side, and select the Development tab (Figure D-7). Here you will see a list of existing development provisioning profiles. To create a new one, click the New Profile button.

Type in a name for the profile and select the development certificate you created earlier (Figure D-8). Select an App ID and a list of target devices. Click the Submit button to create the provisioning profile.

Your provisioning profile is issued automatically and available to download in a few minutes. You may need to refresh your browser window. Download the development provisioning profile onto your Mac and install it by dragging the `.mobileprovision` file onto Xcode in your dock.



Provisioning Profiles – iOS Provisioning Portal – Apple Developer

http://developer.apple.com/ios/manage/provisioningprofiles/index.action

1&1 WebMail News (324) Apple Yahoo! Google Maps Wikipedia Popular YouTube

Provisioning Profiles – iOS Provis...

Developer Technologies Resources Programs Support Member Center Search Developer

iOS Provisioning Portal — Welcome, Abhishek Mishra | Edit Profile | Log out

Provisioning Portal — Go to iOS Dev Center

Home · Certificates · Devices · App IDs · Provisioning · Distribution

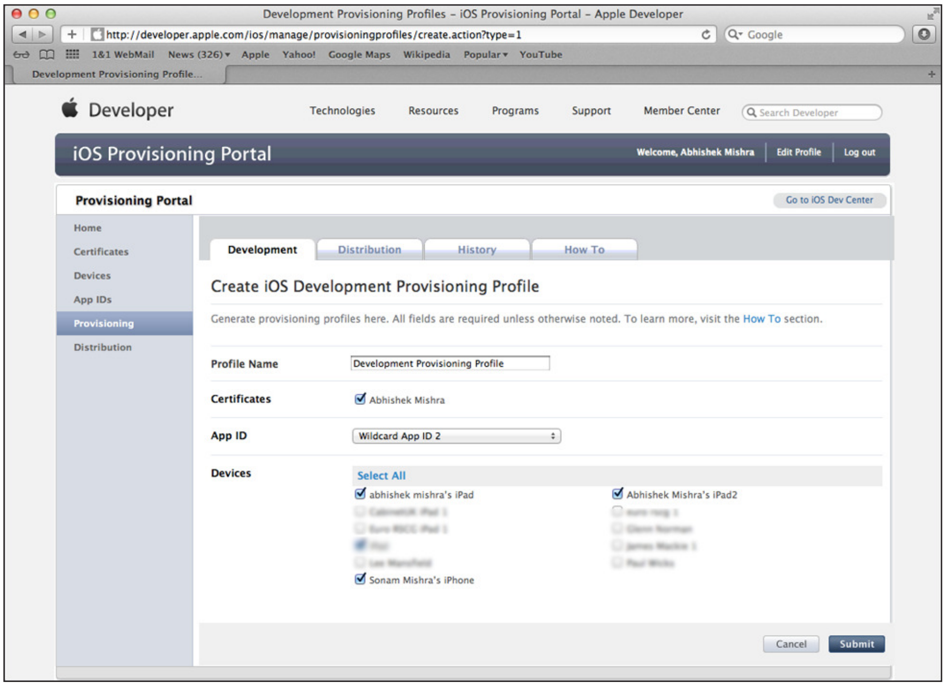
Development | Distribution | History | How To

Development Provisioning Profiles — New Profile

Provisioning Profile	App ID	Status	Actions

You currently do not have any Provisioning Profiles

FIGURE D-7



Development Provisioning Profiles – iOS Provisioning Portal – Apple Developer

http://developer.apple.com/ios/manage/provisioningprofiles/create.action?type=1

1&1 WebMail News (326) Apple Yahoo! Google Maps Wikipedia Popular YouTube

Development Provisioning Profile...

Developer Technologies Resources Programs Support Member Center Search Developer

iOS Provisioning Portal — Welcome, Abhishek Mishra | Edit Profile | Log out

Provisioning Portal — Go to iOS Dev Center

Home · Certificates · Devices · App IDs · Provisioning · Distribution

Development | Distribution | History | How To

Create iOS Development Provisioning Profile

Generate provisioning profiles here. All fields are required unless otherwise noted. To learn more, visit the How To section.

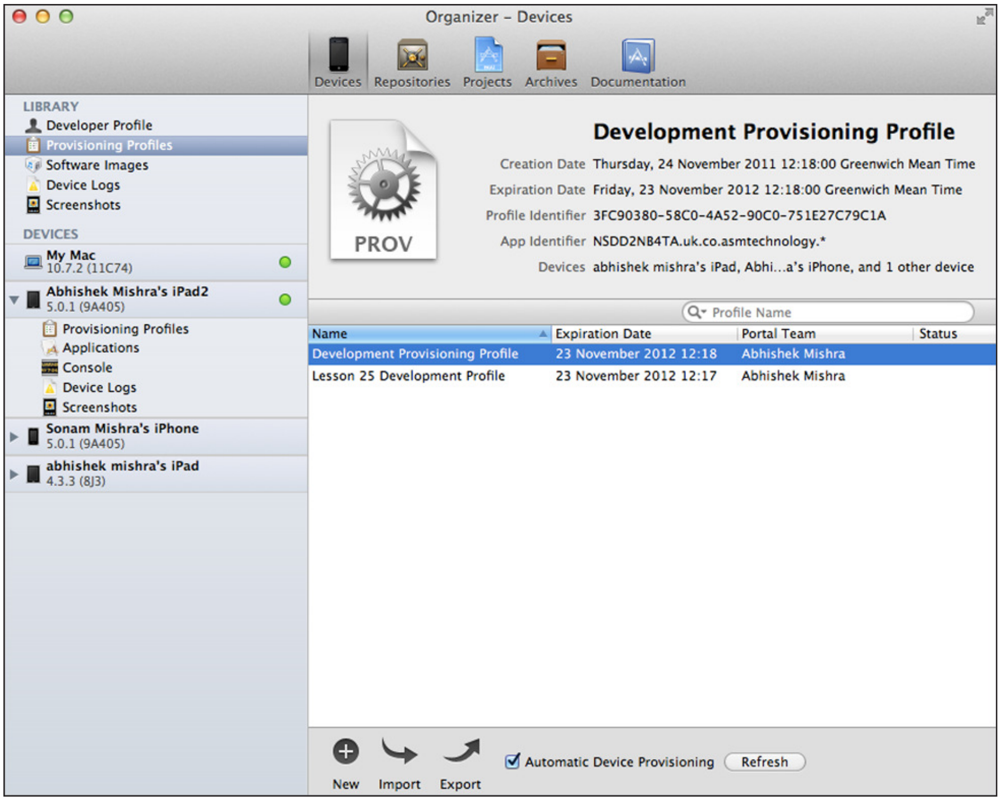
Profile Name	Development Provisioning Profile
Certificates	✓ Abhishek Mishra
App ID	Wildcard App ID 2
Devices	Select All
	✓ abhishek mishra's iPad · ✓ Abhishek Mishra's iPad2
	[redacted] · [redacted]
	[redacted] · [redacted]
	[redacted] · [redacted]
	[redacted] · [redacted]
	✓ Sonam Mishra's iPhone

Cancel Submit

FIGURE D-8

CONFIGURING YOUR PROJECT

The final step in the process involves setting up your Xcode project and preparing an appropriate build. Before you begin, make sure you have installed both your development certificate and development provisioning profile. To check, simply bring up the Xcode Organizer by clicking the last button in the Xcode toolbar, and select Provisioning Profiles in the Organizer menu (Figure D-9).



LIBRARY
Developer Profile
Provisioning Profiles
Software Images
Device Logs
Screenshots

DEVICES
My Mac
10.7.2 (11C74)
Abhishek Mishra's iPad2
5.0.1 (9A405)
Provisioning Profiles
Applications
Console
Device Logs
Screenshots
Sonam Mishra's iPhone
5.0.1 (9A405)
abhishek mishra's iPad
4.3.3 (8J3)

Development Provisioning Profile

Creation Date	Thursday, 24 November 2011 12:18:00 Greenwich Mean Time
Expiration Date	Friday, 23 November 2012 12:18:00 Greenwich Mean Time
Profile Identifier	3FC90380-58C0-4A52-90C0-751E27C79C1A
App Identifier	NSDD2NB4TA.uk.co.asmtechnology.*
Devices	abhishek mishra's iPad, Abhi...a's iPhone, and 1 other device

Profile Name

Name	Expiration Date	Portal Team	Status
Development Provisioning Profile	23 November 2012 12:18	Abhishek Mishra	
Lesson 25 Development Profile	23 November 2012 12:17	Abhishek Mishra	

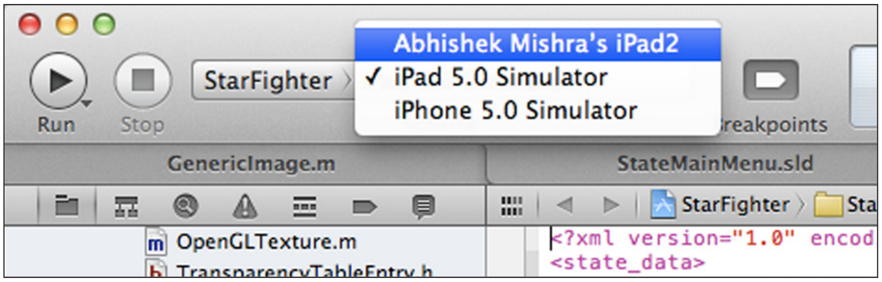
✓ Automatic Device Provisioning

FIGURE D-9

If you have successfully installed your development provisioning profile and development certificate, your profile should be listed on this screen without any error messages in the Status column.

Once you have verified that no errors have been reported by the Organizer, close it and load the project that you want to prepare for Ad Hoc distribution. If the project's App ID is different from what has been registered with the iOS Provisioning Portal, edit the value of the Bundle identifier key in the project's `info.plist` file to match.

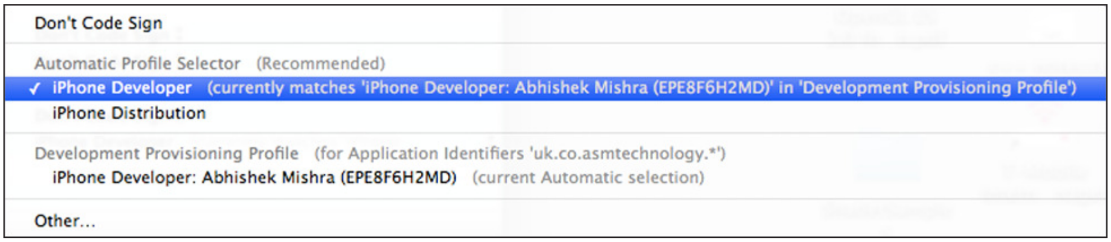
Save the `info.plist` file if you have edited it, then connect one of the provisioned iOS devices to your Mac and ensure that the Scheme/Target selector in the Xcode toolbar is set to build for an iOS Device (Figure D-10).



```
<?xml version="1.0" encod
<state_data>
```

FIGURE D-10

Access the project's properties by selecting the root project node in the project navigator. Select the build target and then switch to the Build Settings tab. Scroll down to the Code Signing section. Locate the node that corresponds to the Debug configuration. Under this node, you will find a node labeled Any iOS SDK. Ensure the value of this node is set to be the development provisioning profile you created and installed earlier. In most cases the default value selected should be okay, but it helps to verify this (Figure D-11).



Don't Code Sign

Automatic Profile Selector (Recommended)

✓ iPhone Developer (currently matches 'iPhone Developer: Abhishek Mishra (EPE8F6H2MD)' in 'Development Provisioning Profile')

iPhone Distribution

Development Provisioning Profile (for Application Identifiers 'uk.co.asmtechnology.*')

iPhone Developer: Abhishek Mishra (EPE8F6H2MD) (current Automatic selection)

Other...

FIGURE D-11

At this point you are ready to test/debug your application on the iOS device. Simply click the Run button on the Xcode toolbar to begin.

E

Ad Hoc Distribution

As an iOS application developer, there will be times when you need to try out your apps on one or more test devices before submitting it to Apple for the App Store approval process. In cases where these devices are your own, you can always set up the device for development and use Xcode to debug applications on the device. In many cases, however, these devices may not be physically accessible; for example, if a client asked you to provide a preview of your app for them to try out on their devices. In these cases you will need to prepare your app for Ad Hoc distribution.

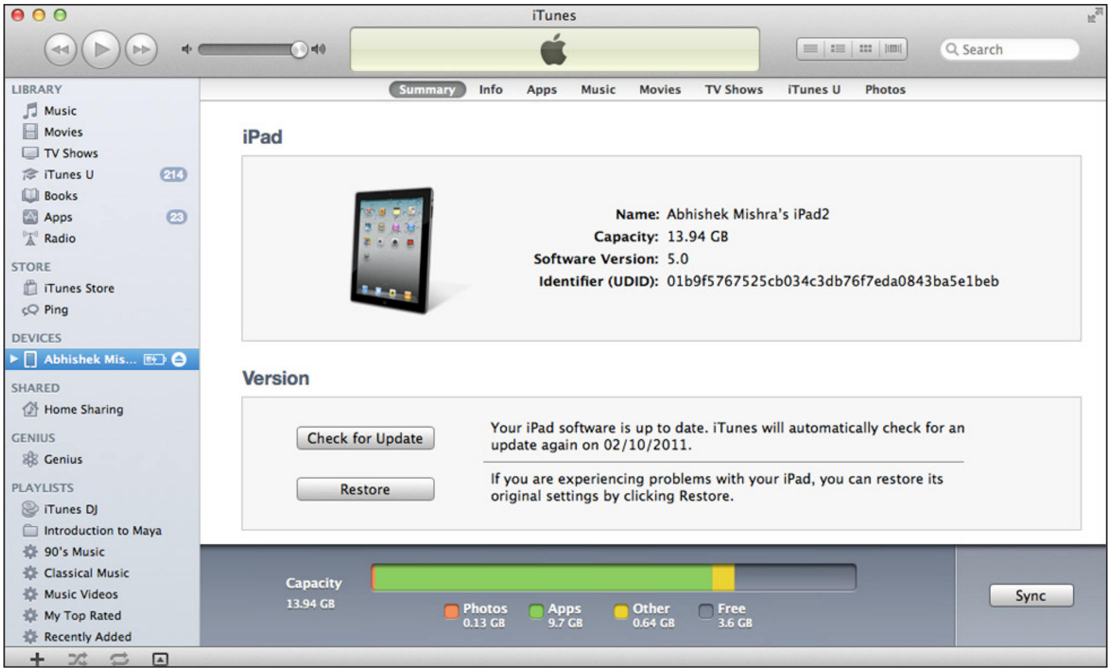
Using Ad Hoc distribution, you can distribute your application to a limited number of devices outside the App Store. The standard iOS developer account enables you to specify up to 100 devices each year, which can be used for development or Ad Hoc distribution. It is important to remember that Ad Hoc distribution is not a replacement for App Store distribution. Apps that are distributed in this manner will eventually expire, at which point they can't be run on the test devices.

OBTAINING AND REGISTERING UDIDS

Each iOS device has a unique 40-digit identifier, commonly referred to as the device UDID. Before you can use Ad Hoc distribution for your apps, you must obtain and register the UDIDs of each device with the iOS Provisioning Portal.

You can obtain the UDID with Xcode or iTunes. To obtain the UDID for a device with iTunes, simply connect the device to a computer with iTunes installed (keep in mind that your clients may have PCs that do not ship with iTunes installed). Launch iTunes, and select the device from the device list on the left side of the iTunes window. The UDID is hidden by default, and to reveal it you need to click the Serial Number label, which then changes to read Identifier (UDID) (Figure E-1).

Obtaining the UDID with Xcode and registering it with iOS Provisioning Portal has been covered in Appendix D.



iPad

Name: Abhishek Mishra's iPad2
Capacity: 13.94 GB
Software Version: 5.0
Identifier (UDID): 01b9f5767525cb034c3db76f7eda0843ba5e1beb

Version

Your iPad software is up to date. iTunes will automatically check for an update again on 02/10/2011.

If you are experiencing problems with your iPad, you can restore its original settings by clicking Restore.

FIGURE E-1

CREATING AN APP ID (BUNDLE IDENTIFIER)

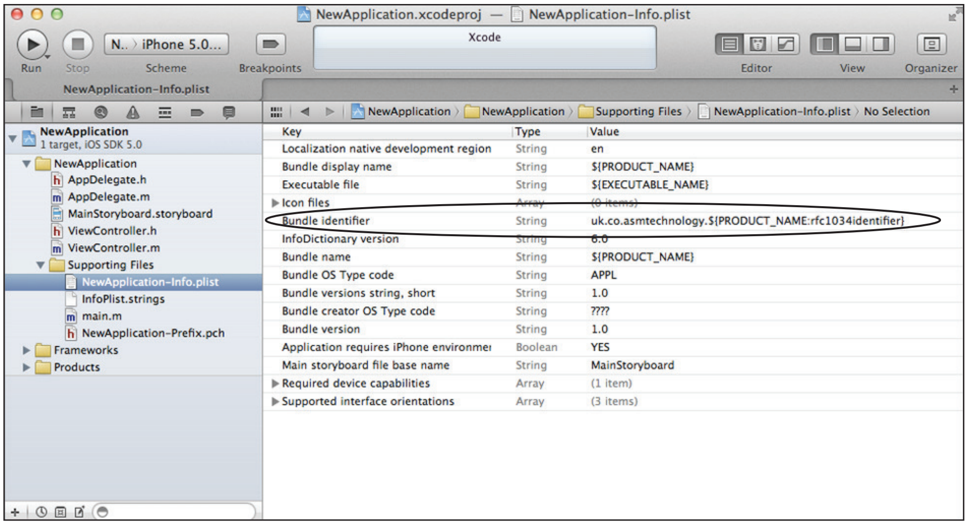
The next step involves creating and registering a unique identifier for your app; this is known as the App ID or Bundle Identifier. In addition to uniquely identifying your application, it allows your application to receive remote notifications, communicate with external accessories, or share keychain data with other applications in a suite.

When you create a new project in Xcode, you are asked to provide a company identifier, and the Bundle Identifier is generated for you by appending the name of the project to the company identifier. To distribute the application through the App Store, or through Ad Hoc distribution, the identifier you provide must match the one you register in the Provisioning Portal. You can always change the Bundle Identifier for an existing application by editing the Bundle Identifier key in the project's `info.plist` file (Figure E-2).

Creating a Bundle Identifier has been covered in Appendix D.

CREATING A DISTRIBUTION CERTIFICATE

To distribute code to iOS devices, you must have a valid distribution certificate installed on your Mac. Creating a distribution certificate involves creating an appropriate certificate request and submitting this request to the iOS Provisioning Portal. Once the certificate is ready, you will be able to download and install it on your Mac.



NewApplication.xcodeproj — NewApplication-Info.plist

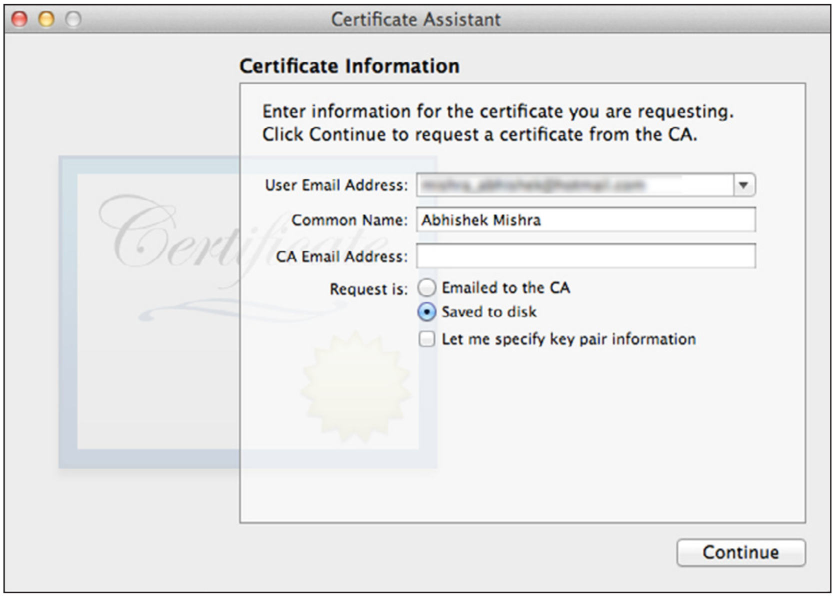
NewApplication > NewApplication > Supporting Files > NewApplication-Info.plist > No Selection

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	uk.co.asmtechnology.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Main storyboard file base name	String	MainStoryboard
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)

FIGURE E-2

To create a certificate request, launch the Keychain Access utility from the `Applications` folder on your Mac. When the Keychain Access utility is running, choose the Certificate Assistant ➪ Request a Certificate from a Certificate Authority menu item.

In the Certificate Assistant dialog (Figure E-3), specify the e-mail address and account name used to access the iOS Developer Program, and ensure the Saved to Disk radio button is selected. Click the Continue button to save the certificate request as a file on your Mac.



Certificate Assistant

Certificate Information

Enter information for the certificate you are requesting.
Click Continue to request a certificate from the CA.

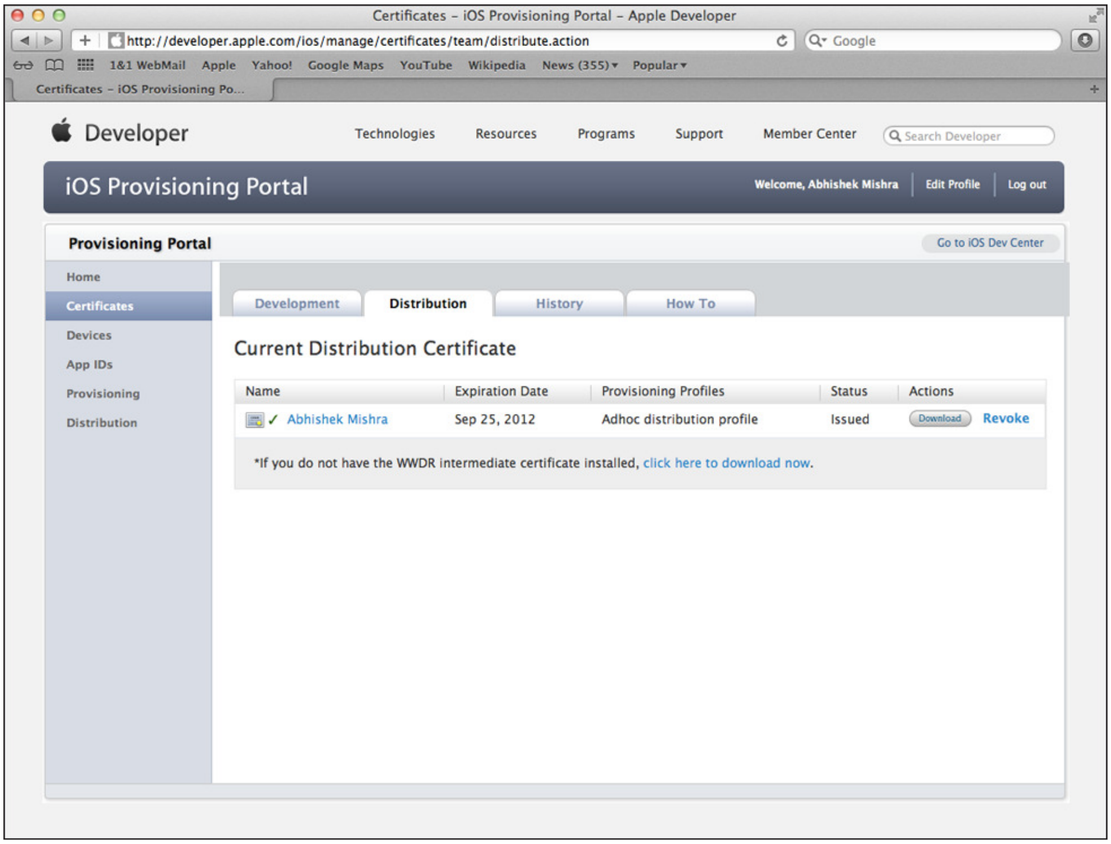
User Email Address: [redacted]
Common Name: Abhishek Mishra
CA Email Address:
Request is: ○ Emailed to the CA [✓ Saved to disk]
☐ Let me specify key pair information

FIGURE E-3

Log in to the iOS Provisioning Portal, click the Certificates category on the left side, and then click the Distribution tab. Click the Choose File button and select the certificate request file that you had saved on your Mac. Finally, click the Submit button to submit the certificate request.

If you are not part of a team, and are solely responsible for handling your iOS Developer account, your certificate is issued automatically and available to download in a few minutes. You may need to refresh your browser window. If you are part of a team, your team agent will need to first approve the certificate request. When your certificate is ready to download, you will see its stats listed as Issued, and a Download link will be available as shown in Figure E-4.

Download the certificate and save it to your Mac. By default, the certificate should be saved to your `Downloads` folder. If you haven't done so already, download the WWDR Intermediate certificate from the same page of the iOS Provisioning Portal by using the link below the certificate. Simply double-click the certificates to install them onto your Mac.


Certificates – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/certificates/team/distribute.action
Google
1&1 WebMail Apple Yahoo! Google Maps YouTube Wikipedia News (355) Popular
Certificates – iOS Provisioning Po...
Developer
Technologies Resources Programs Support Member Center
Search Developer
iOS Provisioning Portal
Welcome, Abhishek Mishra
Edit Profile
Log out
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Development
Distribution
History
How To

Current Distribution Certificate

Name	Expiration Date	Provisioning Profiles	Status	Actions
✓ Abhishek Mishra	Sep 25, 2012	Adhoc distribution profile	Issued	Download Revoke

*If you do not have the WWDR intermediate certificate installed, click here to download now.

FIGURE E-4

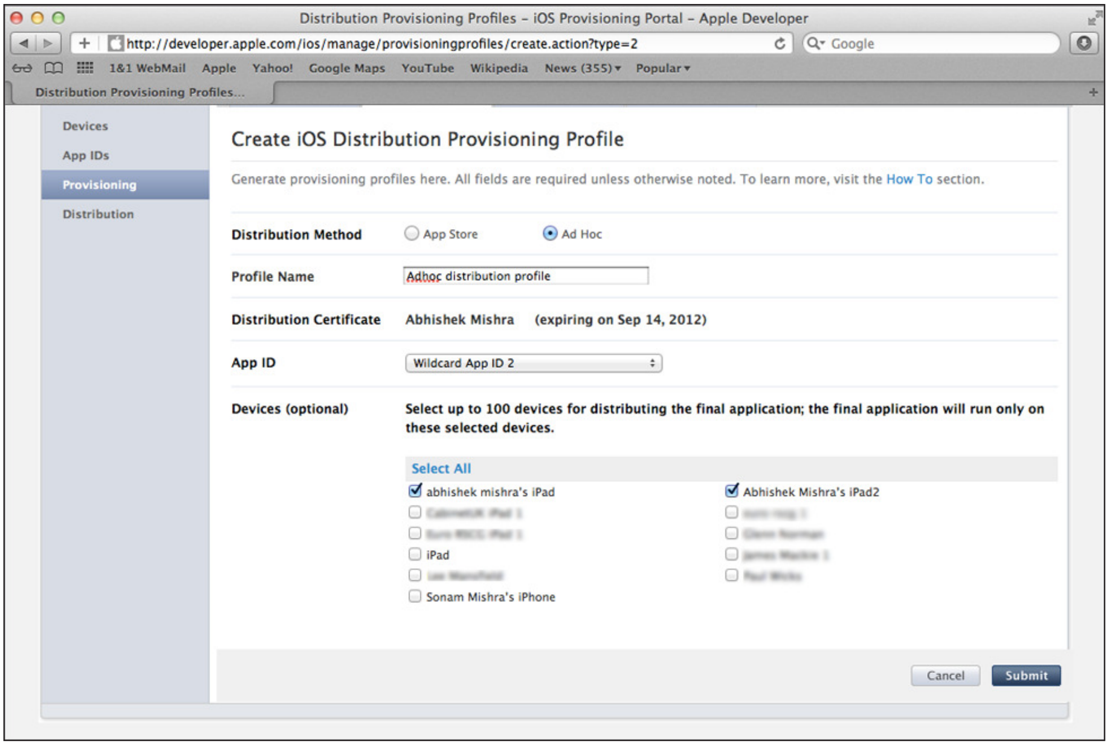
CREATING A PROVISIONING PROFILE

After registering your device UDID, App ID, and creating a distribution certificate, you will need to create a distribution provisioning profile. A Provisioning Profile associates an App ID, a certificate, and some distribution-specific information within Xcode. In the case of Ad Hoc distribution, the certificate in question would be the distribution certificate you just generated in the previous section, and the distribution-specific information would be a list of UDIDs to which you want to distribute.

To create a distribution provisioning profile, log in to the iOS Provisioning Portal, click the Provisioning category on the left side, and select the Distribution tab. Here you will see a list of existing Distribution Provisioning Profiles. To create a new one, click the New Profile button.

Set the distribution method to Ad Hoc (Figure E-5) and provide a descriptive name for the new profile. Select an App ID and a list of target devices. Click the Submit button to create the provisioning profile.

Your provisioning profile is issued automatically and available to download in a few minutes. You may need to refresh your browser window. Download the distribution provisioning profile onto your Mac and install it by dragging the `.mobileprovision` file onto Xcode in your dock.


Distribution Provisioning Profiles – iOS Provisioning Portal – Apple Developer
http://developer.apple.com/ios/manage/provisioningprofiles/create.action?type=2
Google
1&1 WebMail
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News (355)
Popular
Distribution Provisioning Profiles...
Devices
App IDs
Provisioning
Distribution
Cancel
Submit

Create iOS Distribution Provisioning Profile

Generate provisioning profiles here. All fields are required unless otherwise noted. To learn more, visit the How To section.

Distribution Method ○ App Store [✓ Ad Hoc]

Profile Name Adhoc distribution profile

Distribution Certificate Abhishek Mishra (expiring on Sep 14, 2012)

App ID Wildcard App ID 2

Devices (optional) **Select up to 100 devices for distributing the final application; the final application will run only on these selected devices.**

Select All

☑ abhishek mishra's iPad	☑ Abhishek Mishra's iPad2
☐ [illegible]	☐ [illegible]
☐ [illegible]	☐ [illegible]
☐ iPad	☐ [illegible]
☐ [illegible]	☐ [illegible]
☐ Sonam Mishra's iPhone	

FIGURE E-5

CONFIGURING YOUR PROJECT FOR DISTRIBUTION

The final step in the process involves setting up your Xcode project and preparing an appropriate build. Before you begin, make sure you have installed both your distribution certificate and distribution provisioning profile. To check, simply bring up the Xcode Organizer by clicking the last button in the Xcode toolbar and select Provisioning Profiles in the Organizer menu.

If you have successfully installed your distribution provisioning profile and distribution certificate, your profile should be listed on this screen without any error messages in the Status column.

Once you have verified that no errors are reported by the Organizer, close it and load the project that you want to prepare for Ad Hoc distribution. If the project's App ID is different from what has been registered with the iOS Provisioning Portal, edit the value of the Bundle Identifier key in the project's `info.plist` file to match.

Save the file, and then ensure that the Scheme/Target selector in the Xcode toolbar is set to build for an iOS Device (Figure E-6).

Access the project's properties by selecting the root project node in the project navigator and select the Info tab (Figure E-7).

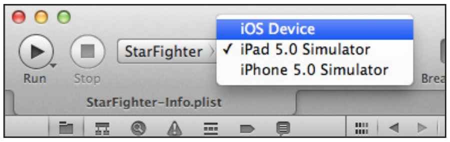
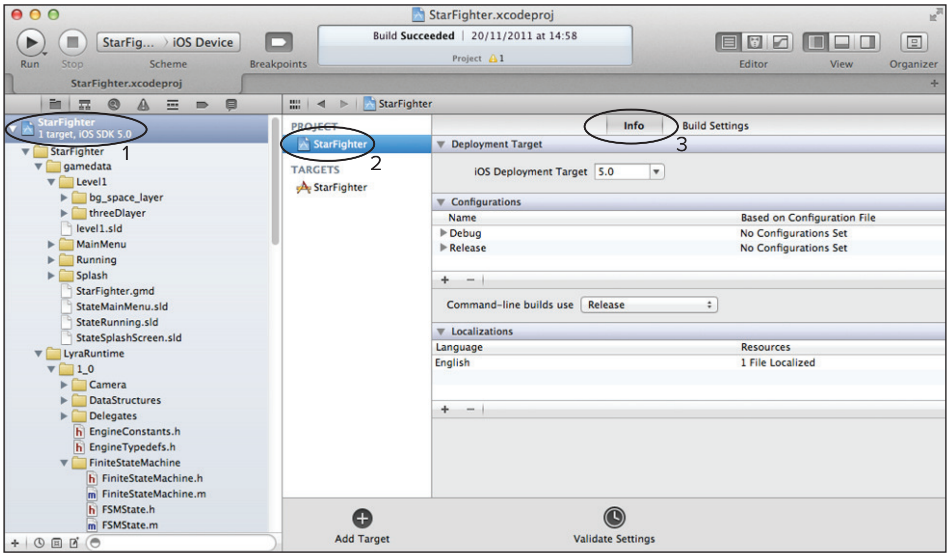


FIGURE E-6



Deployment Target

iOS Deployment Target 5.0

Configurations

Name	Based on Configuration File
Debug	No Configurations Set
Release	No Configurations Set

Command-line builds use Release

Localizations

Language	Resources
English	1 File Localized

FIGURE E-7

Add a new build configuration by clicking the + button below the list of configurations. This brings up a popup menu. Select the Duplicate "Release" Configuration option to create a duplicate of the Release build configuration (Figure E-8). Name this duplicate Ad Hoc Release.

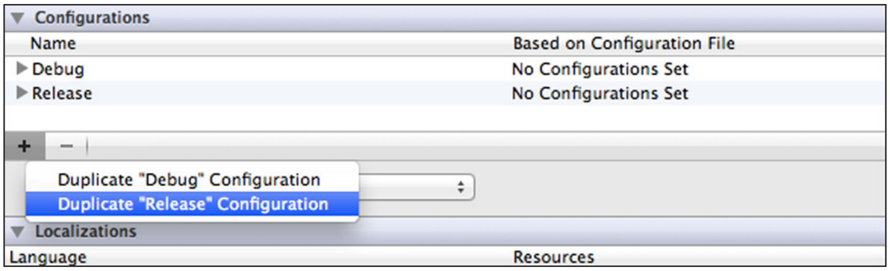


FIGURE E-8

Select the build target and then switch to the Build Settings tab. Scroll down to the Code Signing section and locate the node that corresponds to the Ad Hoc Release configuration you just created. Under this node, you will find a node labeled Any iOS SDK. Ensure the value of this node is set to be the distribution provisioning profile you created and installed earlier. In most cases the default value selected should be okay, but it helps to verify this (Figure E-9).

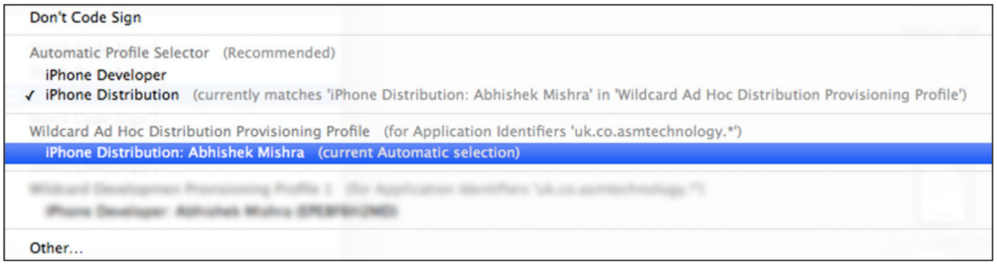

Don't Code Sign
Automatic Profile Selector (Recommended)
iPhone Developer
✓ iPhone Distribution (currently matches 'iPhone Distribution: Abhishek Mishra' in 'Wildcard Ad Hoc Distribution Provisioning Profile')
Wildcard Ad Hoc Distribution Provisioning Profile (for Application Identifiers 'uk.co.asmtechnology.*')
iPhone Distribution: Abhishek Mishra (current Automatic selection)
Other...

FIGURE E-9

Select the Edit Scheme menu from the Scheme/Target multi-selector in the Xcode toolbar (Figure E-10).

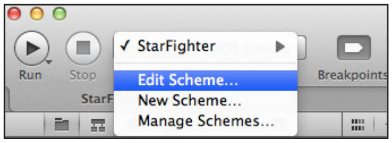
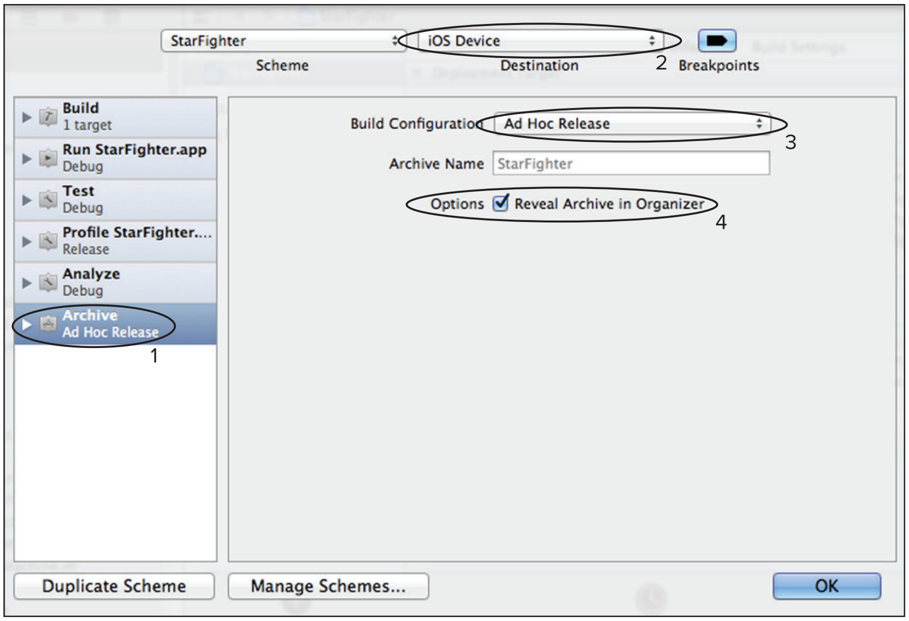


FIGURE E-10

In the Edit Scheme dialog box, select Archive from the left side menu to bring up archive-specific options (Figure E-11).

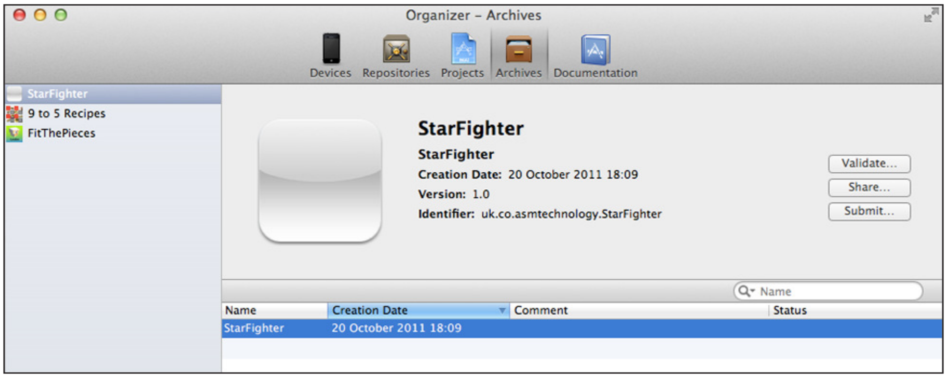
Ensure the Reveal Archive in Organizer option is checked and the Build Configuration is set to Ad Hoc Release. Click OK to dismiss this dialog.

At this point you are ready to prepare an archive that can be distributed to your clients/beta-testers. To prepare an archive, simply select the Product ➪ Archive menu item in Xcode. This builds your project for Ad Hoc distribution. During the build process, Xcode may ask you to allow access to your development certificate; if it does, click the Allow button. When the archive is successfully built, the Organizer opens automatically, revealing the archive (Figure E-12).



Build Configuration: Ad Hoc Release
Archive Name: StarFighter
Options: ✓ Reveal Archive in Organizer

FIGURE E-11



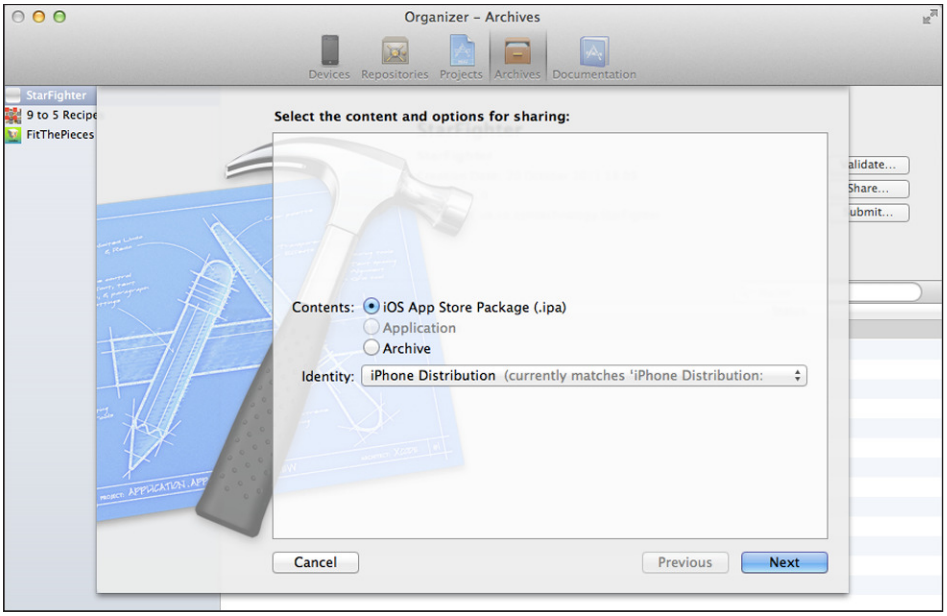
StarFighter
StarFighter
Creation Date: 20 October 2011 18:09
Version: 1.0
Identifier: uk.co.asmtechnology.StarFighter

Name	Creation Date	Comment	Status
StarFighter	20 October 2011 18:09		

FIGURE E-12

The Archive tab in the Xcode Organizer lists all archives created across all projects. To share an archive with your client/beta-tester, ensure the relevant archive is selected and click the Share button. The Organizer presents a simple options sheet (Figure E-13). Ensure the iOS App Store Package option is selected and click Next.

Once again, you may be prompted to allow access to your Distribution Certificate. If so, click Allow. In a few seconds you are presented with the standard Save File dialog box. Select a suitable location on your Mac to save the packaged archive. Send the packaged archive `.ipa` file to your clients/beta-testers. To install the app, they will need to connect their iOS device to their Mac, bring up iTunes, drag the `.ipa` file onto the iTunes Library menu, and sync the iOS device.



Select the content and options for sharing:

Contents: [✓ iOS App Store Package (.ipa)] Application Archive

Identity: iPhone Distribution (currently matches 'iPhone Distribution:

FIGURE E-13

F

App Store Distribution

In most cases, after your app is ready and tested, you will want to list it on the App Store. Regardless of your pricing strategy (free or paid), every application that is submitted to Apple for distribution via the App Store is subject to an approval process. The approval process takes about a week. Updated versions of an existing application also need to go through an approval process.

To distribute your application via the App Store, you will need a standard, paid, iOS developer account. If you have an enterprise iOS developer account, you cannot distribute your applications through the App Store. Submitting an application to Apple for inclusion in the App Store is a two-stage process. First, you need to create an Application Profile on the iTunes Connect portal, and then you need to submit your application binary with the Xcode Organizer.

Most of the steps involved in the second stage of creating and submitting the application binary are similar to the Ad Hoc distribution process, and these steps are covered only briefly in this appendix.

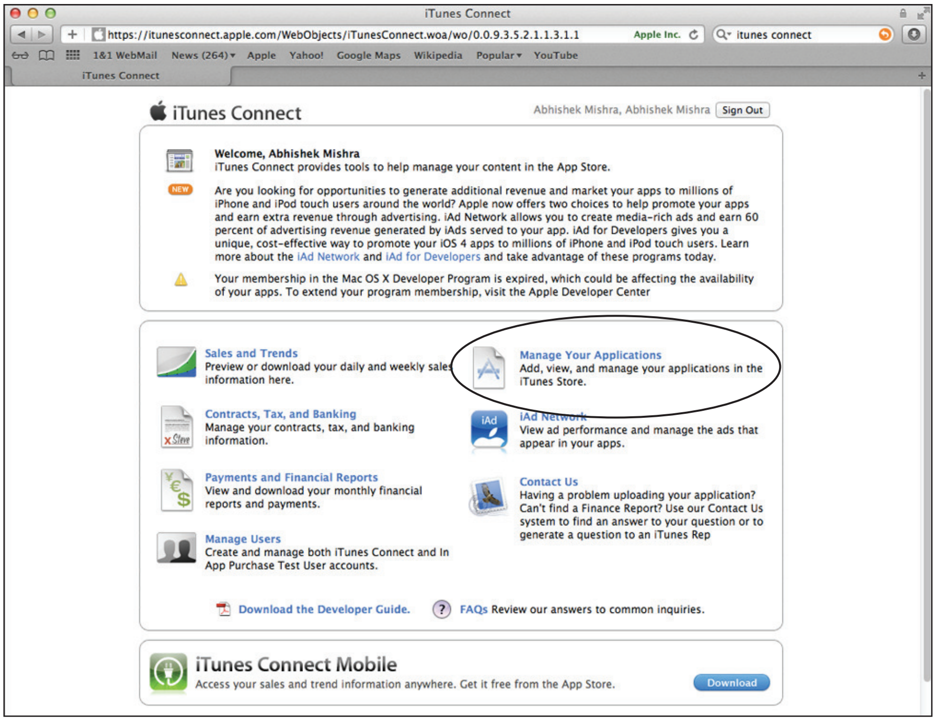
CREATING AN APPLICATION PROFILE

To start the App Store submission process, log in to the iTunes Connect portal at `https://itunesconnect.apple.com/` with your iOS developer account credentials. Once you have logged in to the portal, click the Manage Your Applications link (Figure F-1).

On this screen you will see all your iOS and MacOS applications. You can either add a new application, or manage one of the existing ones. To create a new Application Profile, click the Add New App button on the top-left corner of the window (Figure F-2).

You will be asked to select the type of application you want to submit (iOS or MacOS). Selecting the iOS App option will take you to the Application Information screen (Figure F-3).

On this screen you need to specify basic information on the app. including an application name, a SKU code, and an application Bundle Identifier. Creating and registering Bundle Identifiers was covered in Appendix D.


iTunes Connect
https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa/wo/0.0.9.3.5.2.1.1.3.1.1
itunes connect
1&1 WebMail News (264) Apple Yahoo! Google Maps Wikipedia Popular YouTube
Abhishek Mishra, Abhishek Mishra
Sign Out
NEW
Download

iTunes Connect

Welcome, Abhishek Mishra
iTunes Connect provides tools to help manage your content in the App Store.

Are you looking for opportunities to generate additional revenue and market your apps to millions of iPhone and iPod touch users around the world? Apple now offers two choices to help promote your apps and earn extra revenue through advertising. iAd Network allows you to create media-rich ads and earn 60 percent of advertising revenue generated by iAds served to your app. iAd for Developers gives you a unique, cost-effective way to promote your iOS 4 apps to millions of iPhone and iPod touch users. Learn more about the iAd Network and iAd for Developers and take advantage of these programs today.

Your membership in the Mac OS X Developer Program is expired, which could be affecting the availability of your apps. To extend your program membership, visit the Apple Developer Center

Sales and Trends
Preview or download your daily and weekly sales information here.

Manage Your Applications
Add, view, and manage your applications in the iTunes Store.

Contracts, Tax, and Banking
Manage your contracts, tax, and banking information.

iAd Network
View ad performance and manage the ads that appear in your apps.

Payments and Financial Reports
View and download your monthly financial reports and payments.

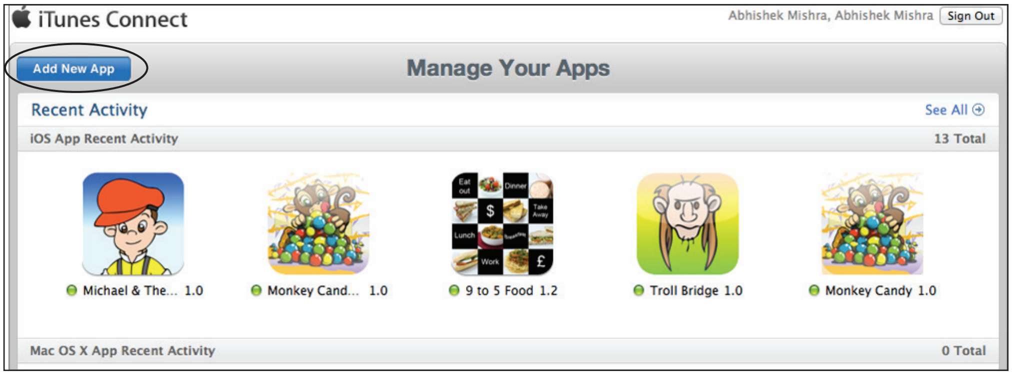
Contact Us
Having a problem uploading your application? Can't find a Finance Report? Use our Contact Us system to find an answer to your question or to generate a question to an iTunes Rep

Manage Users
Create and manage both iTunes Connect and In App Purchase Test User accounts.

Download the Developer Guide. FAQs Review our answers to common inquiries.

iTunes Connect Mobile
Access your sales and trend information anywhere. Get it free from the App Store.

FIGURE F-1



Manage Your Apps

Recent Activity

iOS App Recent Activity 13 Total

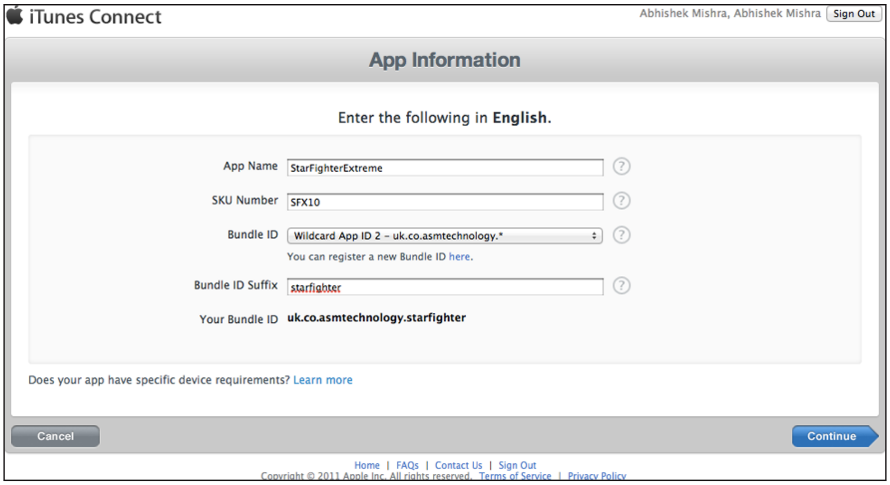
Michael & The... 1.0 | Monkey Cand... 1.0 | 9 to 5 Food 1.2 | Troll Bridge 1.0 | Monkey Candy 1.0

Mac OS X App Recent Activity 0 Total

FIGURE F-2

You will also need to enter a Bundle ID Suffix and verify that the bundle identifier specified in your Xcode project's `info.plist` file matches what is specified on this screen.

The Bundle ID Suffix is typically the name you used while creating the Xcode project. The SKU code is not used by Apple, but is used to identify the application on the monthly financial report provided by Apple. Click Continue when you have finished entering this information.



iTunes Connect

Abhishek Mishra, Abhishek Mishra Sign Out

App Information

Enter the following in **English**.

App Name	StarFighterExtreme
SKU Number	SFX10
Bundle ID	Wildcard App ID 2 – uk.co.asmtechnology.*
	You can register a new Bundle ID here.
Bundle ID Suffix	starfighter
Your Bundle ID	uk.co.asmtechnology.starfighter

Does your app have specific device requirements? Learn more

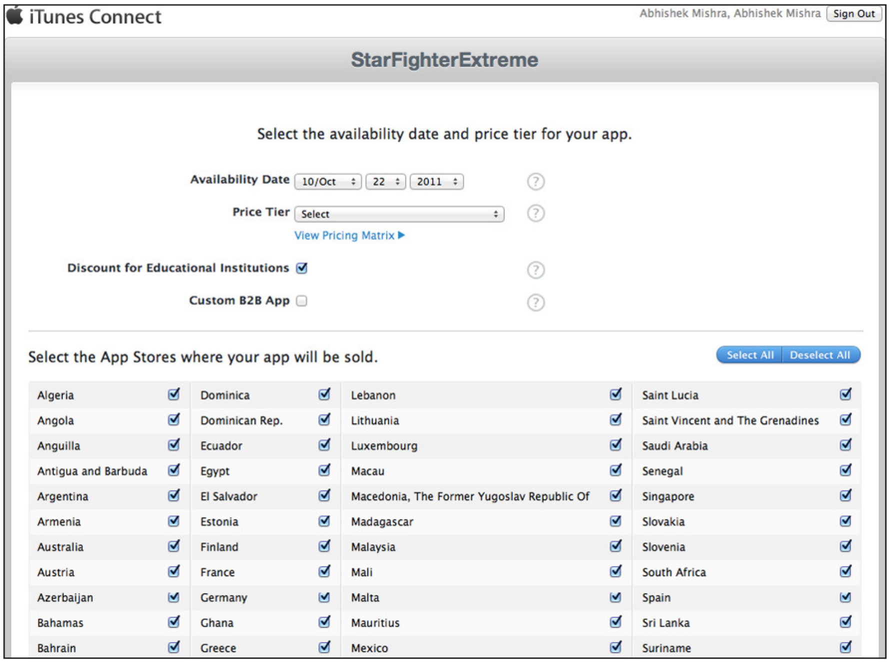
Cancel Continue

Home | FAQs | Contact Us | Sign Out

Copyright © 2011 Apple Inc. All rights reserved. Terms of Service | Privacy Policy

FIGURE F-3

The next part of the process requires you to specify availability date and pricing information. The pricing category you select will affect the price your end users pay for your application. If you do not want your application to be available across all App Stores worldwide, you can select specific countries on this screen (Figure F-4) by clicking the Specific Stores link and then selecting individual stores.



iTunes Connect

Abhishek Mishra, Abhishek Mishra Sign Out

StarFighterExtreme

Select the availability date and price tier for your app.

Availability Date	10/Oct 22 2011
Price Tier	Select
	View Pricing Matrix ▶
Discount for Educational Institutions	✓
Custom B2B App	☐

Select the App Stores where your app will be sold. Select All Deselect All

Algeria	✓	Dominica	✓	Lebanon	✓	Saint Lucia	✓
Angola	✓	Dominican Rep.	✓	Lithuania	✓	Saint Vincent and The Grenadines	✓
Anguilla	✓	Ecuador	✓	Luxembourg	✓	Saudi Arabia	✓
Antigua and Barbuda	✓	Egypt	✓	Macau	✓	Senegal	✓
Argentina	✓	El Salvador	✓	Macedonia, The Former Yugoslav Republic Of	✓	Singapore	✓
Armenia	✓	Estonia	✓	Madagascar	✓	Slovakia	✓
Australia	✓	Finland	✓	Malaysia	✓	Slovenia	✓
Austria	✓	France	✓	Mali	✓	South Africa	✓
Azerbaijan	✓	Germany	✓	Malta	✓	Spain	✓
Bahamas	✓	Ghana	✓	Mauritius	✓	Sri Lanka	✓
Bahrain	✓	Greece	✓	Mexico	✓	Suriname	✓

FIGURE F-4

Click Continue to go to the next screen, where you will provide detailed information on the application. This is a large screen and consists of the following sections:

- Metadata
- Rating
- EULA
- Uploads

The Metadata section resembles Figure F-5. Here you need to specify the following information:

- **Version Number:** This must match the value set in the Xcode project.
- **Description:** This is the description as you want it to appear on the App Store. It can be no more than 4000 characters.
- **Primary Category:** Select a category that best describes the app you are adding from a list of available categories.
- **Subcategory:** Some categories (like Games) allow you to specify up to two subcategories.
- **Secondary Category:** An additional category that further describes the app you are adding.
- **Keywords:** One or more keywords that describe the app you are adding. When users search the App Store, the terms they enter are matched with these keywords.
- **Copyright:** The name of the person or entity that owns the copyright to the app.
- **Contact Email Address:** An email address where Apple engineers can contact you if there are problems with your app.
- **Support URL:** A URL that links to the application's support site.
- **App URL:** An optional URL that links to the application's website.
- **Privacy Policy URL:** An optional URL that links to the application's privacy policy page.
- **Reviewer Notes:** Optional notes intended for the person reviewing the application at Apple. If your application uses any online services and requires its users to provide credentials to access these services, you can provide a set of test credentials here for the reviewer to use.

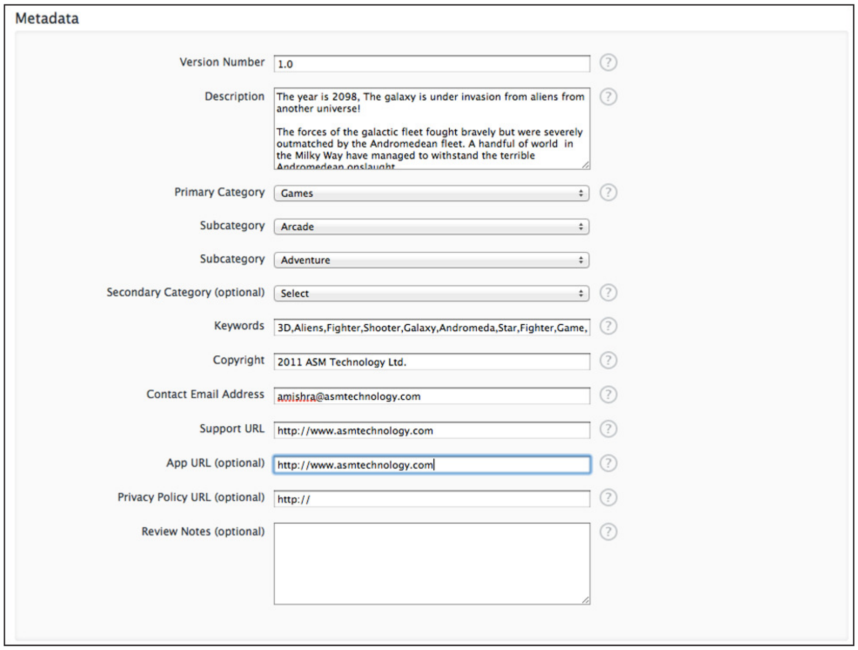
The Rating section consists of a series of questions, the answers to which determine a rating category for your application (Figure F-6). The rating determines the parental controls that will apply to your application. As you change the answers to these questions, the age limit will change.

The EULA section allows you to provide a specific end-user license agreement for specific countries. If you do not provide one, the standard license agreement will apply.

The Uploads section allows you to upload a few images for your application (Figure F-7). These images are:

- **Large 512 × 512 Icon:** A 512 × 512 pixel image that is similar to the application icon and will be used while listing your application on the App Store.

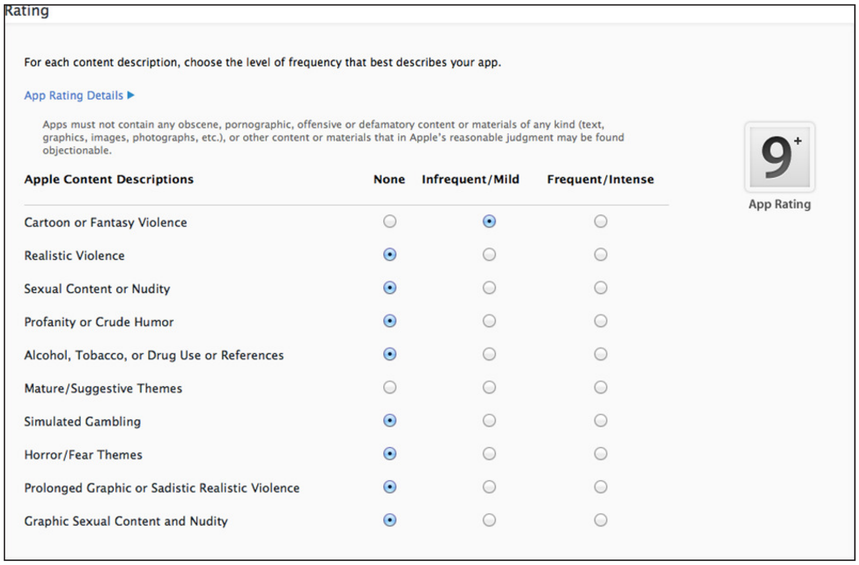
- **iPhone and iPod Touch Screenshots:** You can upload up to five images. These must be 320 × 480 pixels for portrait applications and 480 × 320 pixels for landscape applications.
- **iPad Screenshots:** You can upload up to five images. These must be 768 × 1024 pixels for portrait applications and 1024 × 768 pixels for landscape applications.



Metadata

Version Number	1.0
Description	The year is 2098, The galaxy is under invasion from aliens from another universe! The forces of the galactic fleet fought bravely but were severely outmatched by the Andromedean fleet. A handful of world in the Milky Way have managed to withstand the terrible Andromedean onslaught
Primary Category	Games
Subcategory	Arcade
Subcategory	Adventure
Secondary Category (optional)	Select
Keywords	3D,Aliens,Fighter,Shooter,Galaxy,Andromeda,Star,Fighter,Game,
Copyright	2011 ASM Technology Ltd.
Contact Email Address	amishra@asmtechnology.com
Support URL	http://www.asmtechnology.com
App URL (optional)	http://www.asmtechnology.com
Privacy Policy URL (optional)	http://
Review Notes (optional)	

FIGURE F-5



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	◉	○	○

9+ App Rating

FIGURE F-6

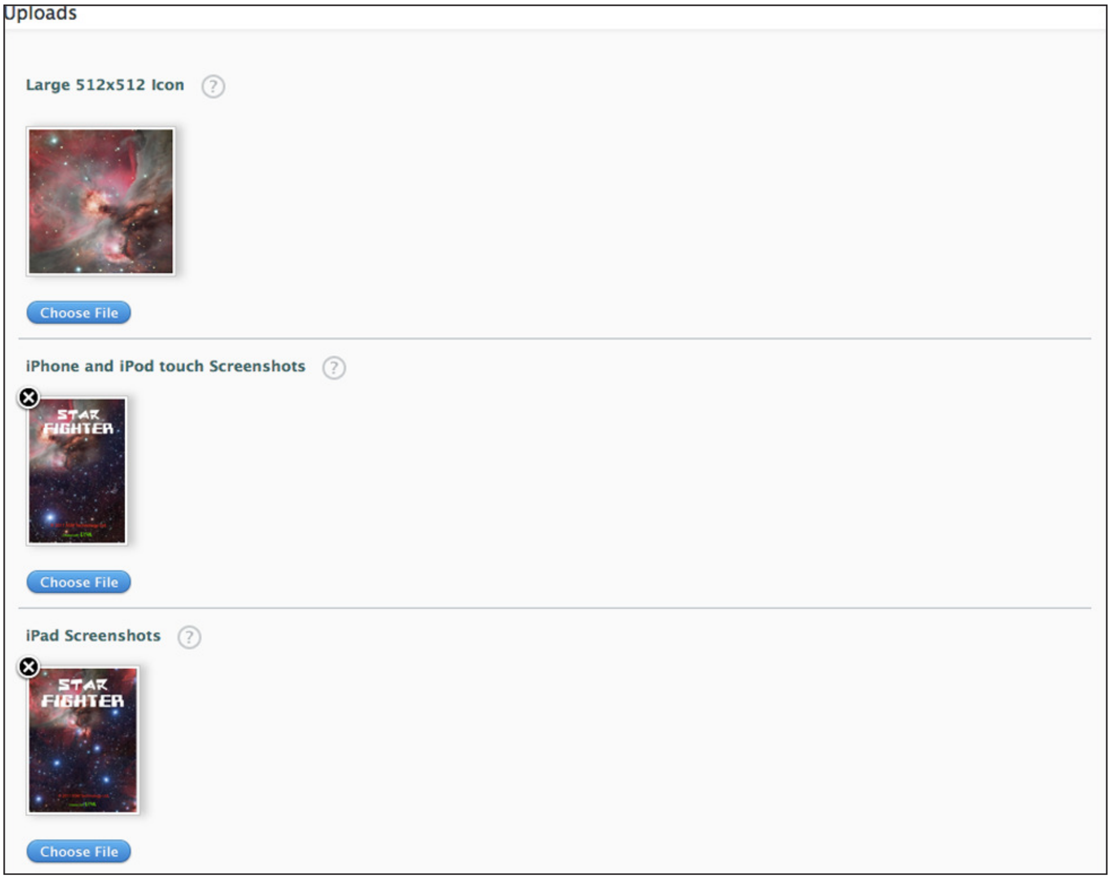


FIGURE F-7

If you are submitting a universal application, you will need to provide both iPhone and iPad screenshots. Click the Next button at the bottom of the screen to finish creating the Application Profile. You are then taken to the App Summary screen, where you should review the information presented and click Done.

PREPARING AND UPLOADING THE APPLICATION BINARY

Once you have created the Application Profile, the next step involves preparing and uploading the actual binary. You can do this immediately after creating the Application Profile or later on. If you decide to do it later on, you will need to log in to the iTunes Connect portal and click the Manage Your Applications link to come to the screen that lists all your applications (Figure F-8). If you've decided to proceed immediately after creating the Application Profile, you should be at this screen now.

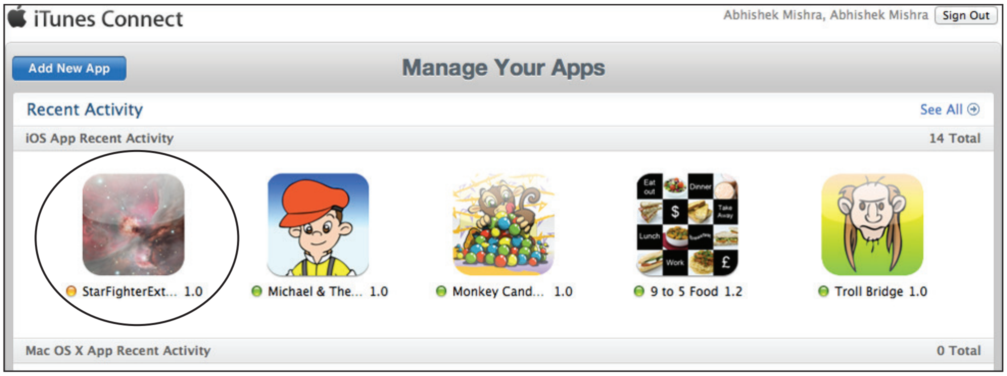
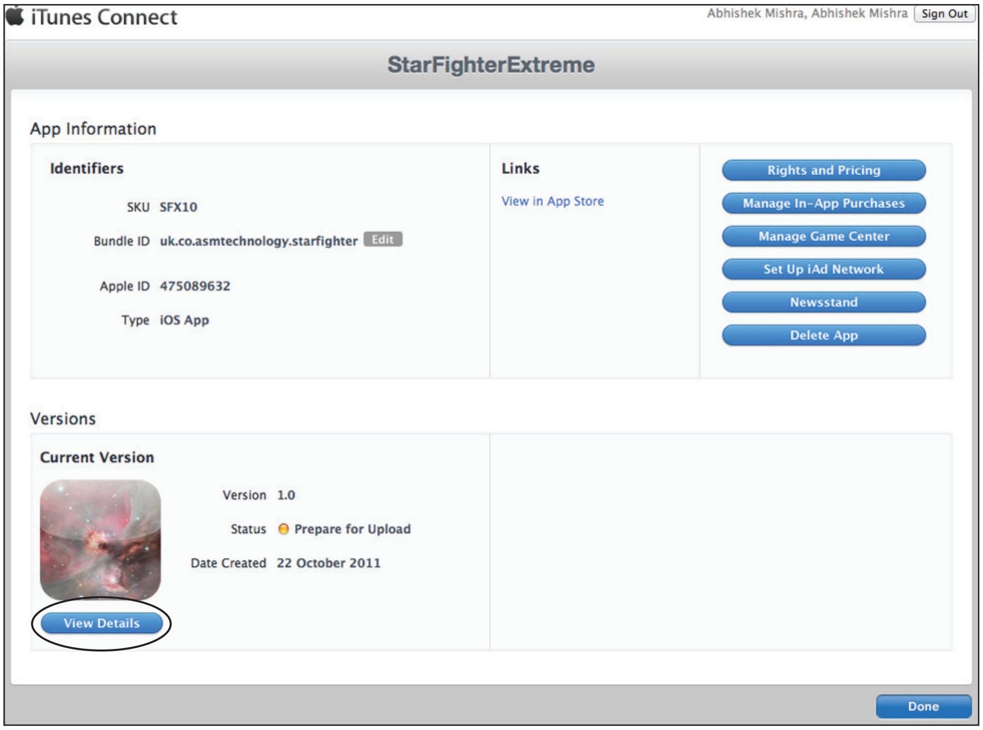


FIGURE F-8

You should find your new application listed there. Click your application's icon to access the App Summary screen. The App Summary screen enables you to modify rights and pricing information, configure In-App purchases, Game Center, News Stand, and iAd Network settings. You can also delete the app from this screen.

Toward the bottom of the screen you will see a list of versions for the application, along with status information. Because this is a new application, there is only one entry in this list corresponding to version 1.0. Click the View Details link to go to the details page for this version (Figure F-9).



SKU SFX10
Bundle ID uk.co.asmtechnology.starfighter Edit
Apple ID 475089632
Type iOS App

Version 1.0
Status Prepare for Upload
Date Created 22 October 2011

FIGURE F-9

On the Version Details screen you can edit metadata, update screenshots, configure localization options for your application, and update its status. Click the Ready to Upload Binary button on this screen (located in the top-right corner).

You will be asked to verify that you are not exporting encryption software, and if you are you will need to provide answers to additional questions. Select the appropriate options and click the Save button.

Click the Continue button on the next screen to go back to the Version Details screen, and note that your application's status has changed to "Waiting for Upload." You can sign out of the iTunes Connect portal at this point if you wish. The rest of the process involves building the application binary and uploading it to the iTunes Connect portal using the Xcode Organizer.

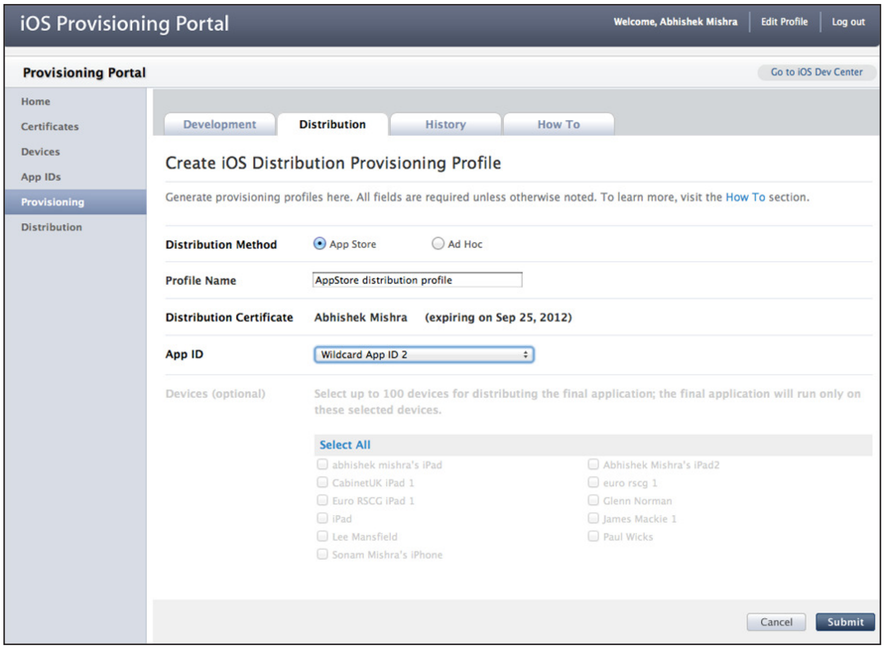
If you haven't done so already, you will need to create and install a distribution certificate. This process is covered in detail in Appendix D.

Creating an App Store Distribution Provisioning Profile

If this is the first application you are submitting to the App Store, you will need to create an App Store distribution provisioning profile. A provisioning profile associates an App ID, a certificate, and some distribution-specific information within Xcode.

To create an App Store distribution provisioning profile, log in to the iOS Provisioning Portal, click the Provisioning category on the left side, and select the Distribution tab. Here you will see a list of existing distribution provisioning profiles. To create a new one, click the New Profile button.

Set the distribution method to App Store (Figure F-10) and provide a descriptive name for the new profile. Select an App ID and click the Submit button to create the provisioning profile.



Create iOS Distribution Provisioning Profile

Generate provisioning profiles here. All fields are required unless otherwise noted. To learn more, visit the How To section.

Distribution Method	[✓ App Store] Ad Hoc
Profile Name	AppStore distribution profile
Distribution Certificate	Abhishek Mishra (expiring on Sep 25, 2012)
App ID	Wildcard App ID 2
Devices (optional)	Select up to 100 devices for distributing the final application; the final application will run only on these selected devices.

abhishek mishra's iPad	Abhishek Mishra's iPad2
CabinetUK iPad 1	euro rscg 1
Euro RSCG iPad 1	Glenn Norman
iPad	James Mackie 1
Lee Mansfield	Paul Wicks
Sonam Mishra's iPhone	

FIGURE F-10

Your provisioning profile is issued automatically, and is available to download in a few minutes. You may need to refresh your browser window. Download the distribution provisioning profile onto your Mac and install it by dragging the `.mobileprovision` file onto Xcode in your dock.

Configuring Your Project for App Store Distribution

The final step in the process involves setting up your Xcode project and preparing an appropriate build. Before you begin, make sure you have installed both your distribution certificate and App Store distribution provisioning profile. To check, simply bring up the Xcode Organizer by clicking the last button in the Xcode toolbar and select Provisioning Profiles in the Organizer menu.

If you have successfully installed your App Store distribution provisioning profile and distribution certificate, your profile should be listed in the Organizer without any error messages in the Status column.

Once you have verified that no errors are reported by the Organizer, close it and load the project that you want to prepare for App Store distribution. If the project's App ID is different from what has been registered with the iOS Provisioning Portal, edit the value of the Bundle Identifier key in the project's `info.plist` file to match.

Save the `info.plist` file if you have edited it, and then ensure that the Scheme/Target selector in the Xcode toolbar is set to build for an iOS Device.

Access the project's properties by selecting the root project node in the project navigator. Select the build target and then switch to the Build Settings tab. Scroll down to the Code Signing section and locate the node that corresponds to the Release configuration.

Under this node, you will find a node labeled Any iOS SDK. Ensure the value of this node is set to be the App Store distribution provisioning profile you created and installed earlier (Figure F-11).

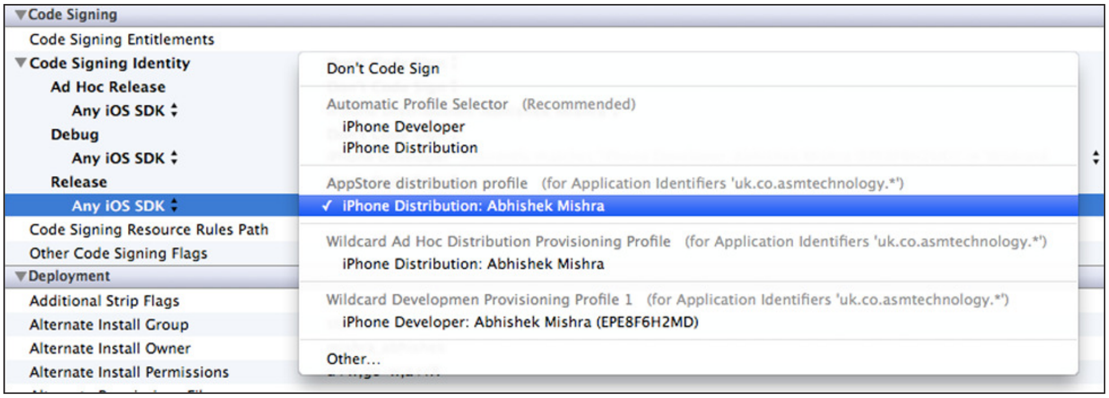

Code Signing
Code Signing Entitlements
Code Signing Identity
Ad Hoc Release
Any iOS SDK
Debug
Any iOS SDK
Release
Any iOS SDK
Code Signing Resource Rules Path
Other Code Signing Flags
Deployment
Additional Strip Flags
Alternate Install Group
Alternate Install Owner
Alternate Install Permissions
Don't Code Sign
Automatic Profile Selector (Recommended)
iPhone Developer
iPhone Distribution
AppStore distribution profile (for Application Identifiers 'uk.co.asmtechnology.*')
✓ iPhone Distribution: Abhishek Mishra
Wildcard Ad Hoc Distribution Provisioning Profile (for Application Identifiers 'uk.co.asmtechnology.*')
iPhone Distribution: Abhishek Mishra
Wildcard Developmen Provisioning Profile 1 (for Application Identifiers 'uk.co.asmtechnology.*')
iPhone Developer: Abhishek Mishra (EPE8F6H2MD)
Other...

FIGURE F-11

Select the Edit Scheme menu from the Scheme/Target multi-selector in the Xcode toolbar (Figure F-12).

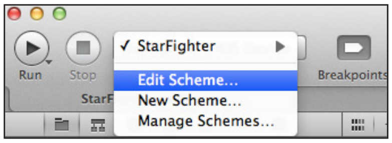
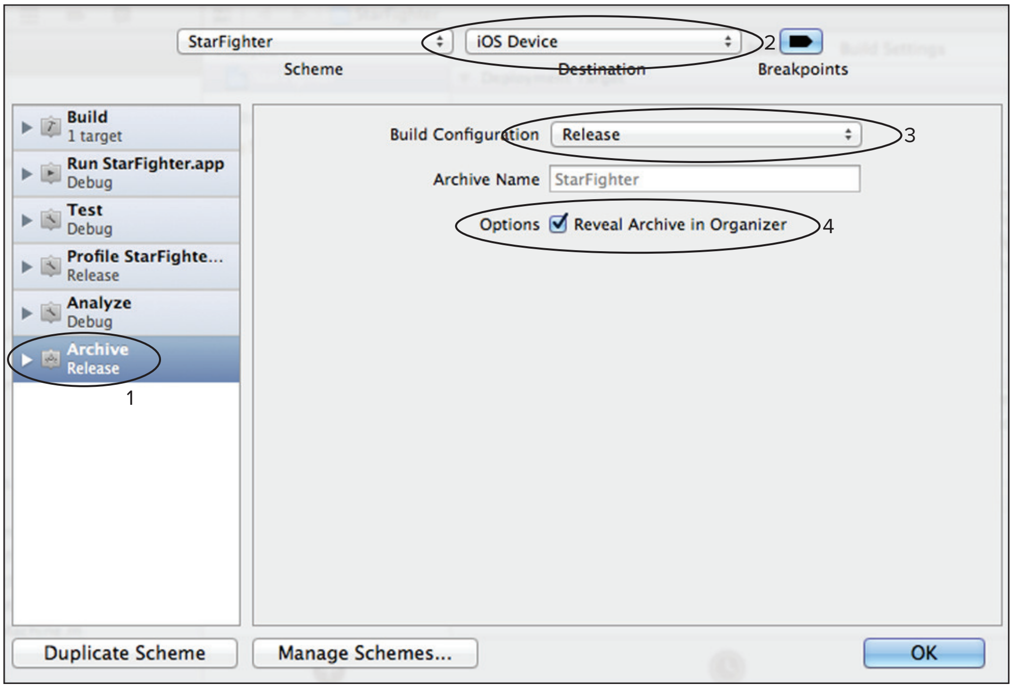


FIGURE F-12

In the Edit Scheme dialog box, select Archive from the left menu to bring up archive-specific options. Ensure the Reveal

Archive in Organizer option is checked and the Build Configuration is set to Release (Figure F-13). Click OK to dismiss this dialog.



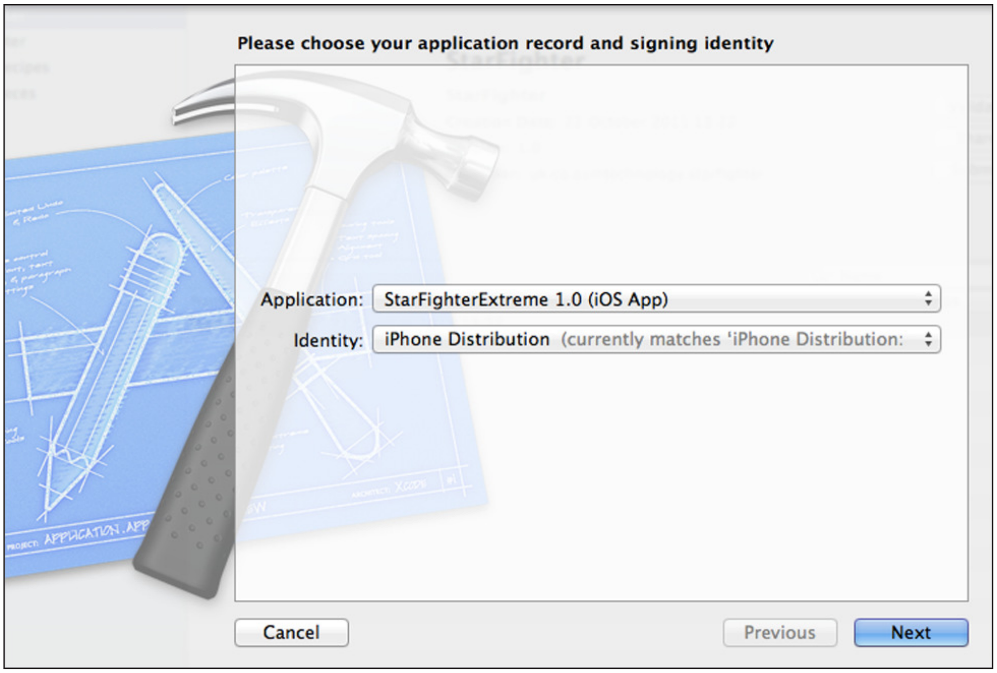
Build Configuration: Release
Archive Name: StarFighter
Options: ☑ Reveal Archive in Organizer

FIGURE F-13

At this point you are ready to prepare an archive that can be distributed to your clients/beta-testers. To prepare an archive, simply select the Product ➪ Archive menu item in Xcode. This builds your project for App Store distribution. During the build process, Xcode may ask you to allow access to your development certificate. If it does, click the Allow button. When the archive is successfully built, the Organizer opens automatically, revealing the archive.

To submit the archive to the iTunes Connect portal, ensure the relevant archive is selected, and click the Submit button. The Organizer will ask you for your iTunes Connect login credentials, and present a simple list of applications in the "Waiting for Upload" state (Figure F-14).

Ensure the correct Application Profile is selected, and click Next. Once again, you may be prompted to allow access to your Distribution Certificate; if so, click Allow. In a few seconds you are presented with a message confirming that the archive has been submitted for approval.


Please choose your application record and signing identity

Application: StarFighterExtreme 1.0 (iOS App)

Identity: iPhone Distribution (currently matches 'iPhone Distribution:

FIGURE F-14