Kyle Richter
Joe Keeley

Completely updated for **IOS 7** and **Xcode 5**

iOS Components and Frameworks

Understanding the Advanced Features of the iOS SDK

Developer's Library

FREE SAMPLE CHAPTER

iOS Components and Frameworks

This page intentionally left blank

iOS Components and Frameworks

Understanding the Advanced Features of the iOS SDK

Kyle Richter
Joe Keeley

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2013944841

ISBN-13: 978-0-321-85671-5
ISBN-10: 0-321-85671-6

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.

First printing: October 2013

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Thomas Cirtin

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Cheri Clark

Indexer
Brad Herriman

Proofreader
Debbie Williams

Technical Reviewers
Collin Ruffenach
Dave Wood

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

I would like to dedicate this book to my co-workers who continually drive me to never accept the first solution.

—Kyle Richter

I dedicate this book to my wife, Irene, and two daughters, Audrey and Scarlett. Your boundless energy and love inspire me daily.

—Joe Keeley

❖

Contents

Foreword

I have been working with the iPhone SDK (now iOS SDK) since the first beta released in 2008. At the time, I was focused on writing desktop apps for the Mac and hadn't thought much about mobile app development.

If you chose to be an early adopter, you were on your own. In typical Apple fashion, the documentation was sparse, and since access to the SDK required an NDA—and apparently, a secret decoder ring—you were on your own. You couldn't search Google or turn to StackOverflow for help, and there sure as hell weren't any books out yet on the SDK.

In the six years (yes, it really has only been six years) since Apple unleashed the original iPhone on the world, we've come a long way. The iPhone SDK is now the iOS SDK. There are dozens of books and blogs and podcasts and conferences on iOS development. And ever since 2009, WWDC has been practically impossible to get into, making it even harder for developers—old and new—to learn about the latest features coming to the platform. For iOS developers, there is so much more to learn.

One of the biggest challenges I have as an iOS developer is keeping on top of all the components and frameworks available in the kit. The iOS HIG should help us with that, but it doesn't go far enough—deep enough. Sure, now I can find some answers by searching Google or combing through StackOverflow but, more often than not, those answers only explain the how and rarely the why, and they never provide the details you really need.

And this is what Kyle and Joe have done with this book—they're providing the detail needed so you can fully understand the key frameworks that make up the iOS SDK.

I've had the pleasure of knowing Kyle and Joe for a number of years. They are two of the brightest developers I have ever met. They have each written some amazing apps over the years, and they continuously contribute to the iOS development community by sharing their knowledge—speaking at conferences and writing other books on iOS development. If you have a question about how to do something in iOS, chances are good that Kyle and Joe have the answer for you.

But what makes these guys so awesome is not just their encyclopedic knowledge of iOS, it's their willingness to share what they know with everyone they meet. Kyle and Joe don't have competitors, they have friends.

Kyle and Joe's in-depth knowledge of the iOS SDK comes through in this book. It's one of the things I like about this book. It dives into the details for each component covered at a level that you won't always find when searching online.

I also like the way the book is structured. This is not something that you'll read cover to cover. Instead, you'll pick up the book because you need to learn how to implement a collection view or sort out some aspect of running a task in a background thread that you can't quite wrangle. You'll pick up the book when you need it, find the solution, implement it in your own code, and then toss the book back on the floor until you need it again. This is what makes

iOS Components and Frameworks an essential resource for any iOS developer—regardless of your experience level. You might think you're a master with Core Location and MapKit, but I reckon you'll find something here that you never knew before.

Kyle and Joe don't come with egos. They don't brag. And they sure don't act like they are better than any other developer in the room. They instill the very spirit that has made the Mac and iOS developer community one of the friendliest, most helpful in our industry, and this book is another example of their eagerness to share their knowledge.

This book, just like the seminal works from Marks and LaMarche or Sadun, will always be within arm's reach of my desk. This is the book I wish I had when I first started developing iOS apps in 2008. Lucky you, it's here now.

—Kirby Turner,

Chief Code Monkey at White Peak Software, author of *Learning iPad Programming, A Hands on Guide to Building Apps for the iPad, Second Edition* (Addison-Wesley Professional), and Cocoa developer community organizer and conference junkie

August 28, 2013

Preface

Welcome to *iOS Components and Frameworks: Understanding the Advanced Features of the iOS SDK*!

There are hundreds of "getting started with iOS" books available to choose from, and there are dozens of advanced books in specific topics, such as Core Data or Security. There was, however, a disturbing lack of books that would bridge the gap between beginner and advanced niche topics.

This publication aims to provide development information on the intermediate-to-advanced topics that are otherwise not worthy of standalone books. It's not that the topics are uninteresting or lackluster, it's that they are not large enough topics. From topics such as working with JSON to accessing photo libraries, these are frameworks that professional iOS developers use every day but are not typically covered elsewhere.

Additionally, several advanced topics are covered to the level that many developers need in order to just get started. Picking up a 500-page Core Data book is intimidating, whereas Chapter 13 of this book provides a very quick and easy way to get started with Core Data. Additional introductory chapters are provided for debugging and instruments, TextKit, language features, and iCloud.

Topics such as Game Center leaderboards and achievements, AirPrint, music libraries, Address Book, and Passbook are covered in their entirety. Whether you just finished your first iOS project or you are an experienced developer, this book will have something for you.

The chapters have all been updated to work with iOS 7 Beta 4. As such, there were several iOS 7 features that were still in active development that might not work the same as illustrated in the book after the final version of iOS 7 is released. Please let us know if you encounter issues and we will release updates and corrections.

If you have suggestions, bug fixes, corrections, or anything else you'd like to contribute to a future edition, please contact us at icf@dragonforged.com. We are always interested in hearing what would make this book better and are very excited to continue refining it.

—Kyle Richter and Joe Keeley

Prerequisites

Every effort has been made to keep the examples and explanations simple and easy to digest; however, this is to be considered an intermediate to advanced book. To be successful with it, you should have a basic understanding of iOS development, Objective-C, and C. Familiarity of the tools such as Xcode, Developer Portal, iTunes Connect, and Instruments is also assumed. Refer to *Programming in Objective-C,* by Stephen G. Kochan, and *Learning iOS Development,* by Maurice Sharp, Rod Strougo, and Erica Sadun, for basic Objective-C and iOS skills.

What You'll Need

Although you can develop iOS apps in the iOS simulator, it is recommended that you have at least one iOS device available for testing:

- **Apple iOS Developer Account:**The latest version of the iOS developer tools including Xcode and the iOS SDKs can be downloaded from Apple's Developer Portal (http://developer.apple.com/ios). To ship an app to the App Store or to install and test on a personal device, you will also need a paid developer account at $99 per year.
- **Macintosh Computer:**To develop for iOS and run Xcode, you will need a modern Mac computer capable of running the latest release of OS X.
- **Internet Connection:**Many features of iOS development require a constant Internet connection for your Mac as well as for the device you are building against.

How This Book Is Organized

With few exceptions (Game Center and Core Data), each chapter stands on its own. The book can be read cover to cover but any topic can be skipped to when you find a need for that technology; we wrote it with the goal of being a quick reference for many common iOS development tasks.

Here is a brief overview of the chapters you will encounter:

- **Chapter 1, "UIKit Dynamics":** iOS 7 introduced UI Kit Dynamics to add physics-like animation and behaviors to UIViews. You will learn how to add dynamic animations, physical properties, and behaviors to standard objects. Seven types of behaviors are demonstrated in increasing difficulty from gravity to item properties.
- **Chapter 2, "Core Location, MapKit, and Geofencing":** iOS 6 introduced new, Apple-provided maps and map data. This chapter covers how to interact with Core Location to determine the device's location, how to display maps in an app, and how to customize the map display with annotations, overlays, and callouts. It also covers how to set up regional monitoring (or geofencing) to notify the app when the device has entered or exited a region.
- **Chapter 3, "Leaderboards":** Game Center leaderboards provide an easy way to add social aspects to your iOS game or app. This chapter introduces a fully featured iPad game called Whack-a-Cac, which walks the reader through adding leaderboard support. Users will learn all the required steps necessary for implementing Game Center leaderboards, as well as get a head start on implementing leaderboards with a custom interface.
- **Chapter 4, "Achievements":** This chapter continues on the Whack-a-Cac game introduced in Chapter 3. You will learn how to implement Game Center achievements in a fully featured iPad game. From working with iTunes Connect to displaying achievement progress, this chapter provides all the information you need to quickly get up and running with achievements.
- **Chapter 5, "Getting Started with Address Book":** Integrating a user's contact information is a critical step for many modern projects. Address Book framework is one of the oldest available on iOS; in this chapter you'll learn how to interact with that framework. You will learn how to use the people picker, how to access the raw address book data, and how to modify and save that data.
- **Chapter 6, "Working with Music Libraries":** This chapter covers how to access the user's music collection from a custom app, including how to see informational data about the music in the collection, and how to select and play music from the collection.
- **Chapter 7, "Working with and Parsing JSON":** JSON, or JavaScript Object Notation, is a lightweight way to pass data back and forth between different computing platforms and architectures. As such, it has become the preferred way for iOS client apps to communicate complex sets of data with servers. This chapter describes how to create JSON from existing objects, and how to parse JSON into iOS objects.

- **Chapter 8, "Getting Started with iCloud":** This chapter explains how to get started using iCloud, for syncing key-value stores and documents between devices. It walks though setting up an app for iCloud, how to implement the key-value store and document approaches, and how to recognize and resolve conflicts.
- **Chapter 9, "Notifications":** Two types of notifications are supported by iOS: local notifications, which function on the device with no network required, and remote notifications, which require a server to send a push notification through Apple's Push Notification Service to the device over the network. This chapter explains the differences between the two types of notifications, and demonstrates how to set them up and get notifications working in an app.
- **Chapter 10, "Bluetooth Networking with Game Kit":** This chapter will walk you through creating a real-time Bluetooth-based chat client, enabling you to connect with a friend within Bluetooth range and send text messages back and forth. You will learn how to interact with the Bluetooth functionality of Game Kit, from finding peers to connecting and transferring data.
- **Chapter 11, "AirPrint":** An often underappreciated feature of the iOS, AirPrint enables the user to print documents and media to any wireless-enabled AirPrint-compatible printer. Learn how to quickly and effortlessly add AirPrint support to your apps. By the end of this chapter you will be fully equipped to enable users to print views, images, PDFs, and even rendered HTML.
- **Chapter 12, "Core Data Primer":** Core Data can be a vast and overwhelming topic. This chapter tries to put Core Data in context for the uninitiated, and explains when Core Data might be a good solution for an app and when it might be overkill. It also explains some of the basic concepts of Core Data in simple terminology.
- **Chapter 13, "Getting Up and Running with Core Data":** This chapter demon-strates how to set up an app to use Core Data, how to set up a Core Data data model, and how to implement many of the most commonly used Core Data tools in an app. If you want to start using Core Data without digging through a 500-page book, this chapter is for you.
- **Chapter 14, "Language Features":** Objective-C has been evolving since iOS was introduced. This chapter covers some of the language and compiler-level changes that have occurred, and explains how and why a developer would want to use them. It covers the new literal syntaxes for things like numbers, array, and dictionaries; it also covers blocks, ARC, property declarations, and some oldies but goodies including dot notation, fast enumeration, and method swizzling.
- **Chapter 15, "Integrating Twitter and Facebook Using Social Framework":** Social integration is the future of computing and it is accepted that all apps have social features built in. This chapter will walk you through adding support for Facebook and Twitter to your app using the Social Framework. You will learn how to use the built-in composer to create new Twitter and Facebook posts. You will also learn how to pull down feed information from both services and how to parse and interact with that data. Finally, using the frameworks to send messages from custom user interfaces is covered. By the

end of this chapter, you will have a strong background in Social Framework as well as working with Twitter and Facebook to add social aspects to your apps.

- **Chapter 16, "Working with Background Tasks":** Being able to perform tasks when the app is not the foreground app was a big new feature introduced in iOS 4, and more capabilities have been added since. This chapter explains how to perform tasks in the background after an app has moved from the foreground, and how to perform specific background activities allowed by iOS.
- **Chapter 17, "Grand Central Dispatch for Performance":** Performing resource-intensive activities on the main thread can make an app's performance suffer with stutters and lags. This chapter explains several techniques provided by Grand Central Dispatch for doing the heavy lifting concurrently without affecting the performance of the main thread.
- **Chapter 18, "Using Keychain to Secure Data":** Securing user data is important and an often-overlooked stage of app development. Even large public companies have been called out in the news over the past few years for storing user credit card info and passwords in plain text. This chapter provides an introduction to not only using the Keychain to secure user data but developmental security as a whole. By the end of the chapter, you will be able to use Keychain to secure any type of small data on users' devices and provide them with peace of mind.
- **Chapter 19, "Working with Images and Filters":** This chapter covers some basic image-handling techniques, and then dives into some advanced Core Image techniques to apply filters to images. The sample app provides a way to explore all the options that Core Image provides and build filter chains interactively in real time.
- **Chapter 20, "Collection Views":** Collection views, a powerful new API introduced in iOS6, give the developer flexible tools for laying out scrollable, cell-based content. In addition to new content layout options, collection views provide exciting new animation capabilities, both for animating content in and out of a collection view, and for switching between collection view layouts. The sample app demonstrates setting up a basic collection view, a customized flow layout collection view, and a highly custom, nonlinear collection view layout.
- **Chapter 21, "Introduction to TextKit":** iOS 7 introduced TextKit as an easier-to-use and greatly expanded update to Core Text. TextKit enables developers to provide rich and interactive text formatting to their apps. Although TextKit is a very large subject, this chapter provides the basic groundwork to accomplish several common tasks, from adding text wrapping around an image to inline custom font attributes. By the end of this chapter, you will have a strong background in TextKit and have the groundwork laid to explore it more in depth.
- **Chapter 22, "Gesture Recognizers":** This chapter explains how to make use of gesture recognizers in an app. Rather than dealing with and interpreting touch data directly, gesture recognizers provide a simple and clean way to recognize common gestures and respond to them. In addition, custom gestures can be defined and recognized using gesture recognizers.

- **Chapter 23, "Accessing Photo Libraries":** The iPhone has actually become a very popular camera, as evidenced by the number of photos that people upload to sites such as Flickr. This chapter explains how to access the user's photo library, and handle photos and videos in a custom app. The sample app demonstrates rebuilding the iOS 6 version of Photos.app.
- **Chapter 24, "Passbook and PassKit":** With iOS6, Apple introduced Passbook, a standalone app that can store "passes," or things like plane tickets, coupons, loyalty cards, or concert tickets. This chapter explains how to set up passes, how to create and distribute them, and how to interact with them in an app.
- **Chapter 25, "Debugging and Instruments":** One of the most important aspects of development is to be able to debug and profile your software. Rarely is this topic covered even in a cursory fashion. This chapter will introduce you to debugging in Xcode and performance analysis using Instruments. Starting with a brief history of computer bugs, the chapter walks you through common debugging tips and tricks. Topics of breakpoints and debugger commands are briefly covered, and the chapter concludes with a look into profiling apps using the Time Profiler and memory analysis using Leaks. By the end of this chapter, you will have a clear foundation on how to troubleshoot and debug iOS apps on both the simulator and the device.

About the Sample Code

Each chapter of this book is designed to stand by itself; therefore, each chapter with the exception of Chapter 25, “Debugging and Instruments,” Chapter 12, “Core Data Primer,” and Chapter 14, “Language Features,” has its own sample project. Chapter 3, “Leaderboards,” and Chapter 4, “Achievements,” share a base sample project, but each expands on that base project in unique ways. Each chapter provides a brief introduction to the sample project and walks the reader through any complex sections of the sample project not relating directly to the material in the chapter.

Every effort has been made to create simple-to-understand sample code, which often results in code that is otherwise not well optimized or not specifically the best way of approaching a problem. In these circumstances the chapter denotes where things are being done inappropriately for a real-world app. The sample projects are not designed to be standalone or finished apps; they are designed to demonstrate the functionality being discussed in the chapter. The sample projects are generic with intention; the reader should be able to focus on the material in the chapter and not the unrelated sample code materials. A considerable amount of work has been put into removing unnecessary components from the sample code and condensing subjects into as few lines as possible.

Many readers will be surprised to see that the sample code in the projects is not built using Automatic Reference Counting (ARC); this is by design as well. It is easier to mentally remove the memory management than to add it. The downloadable sample code is made available to suit both tastes; copies of ARC and non-ARC sample code are bundled together. The sample code is prefixed with “ICF” and most, but not all, sample projects are named after the chapter title.

When working with the Game Center chapters, the bundle ID is linked to a real app, which is in our personal Apple account; this ensures that examples continue to work. Additionally, it has the small additional benefit of populating multiple users’ data as developers interact with the sample project. For chapters dealing with iCloud, Push Notifications, and Passbook, the setup required for the apps is thoroughly described in the chapter, and must be completed using a new App ID in the reader’s developer account in order to work.

Getting the Sample Code

You will be able to find the most up-to-date version of the source code at any moment at https://github.com/dfsw/icf. The code is publicly available and open source. The code is separated into two folders, one for ARC and one running non-ARC. Each chapter is broken down into its own folder containing an Xcode project; there are no chapters with multiple projects. We encourage readers to provide feedback on the source code and make recommendations so that we can continue to refine and improve it long after this book has gone to print.

Installing Git and Working with GitHub

Git is a version control system that has been growing in popularity for several years. To clone and work with the code on GitHub, you will want to first install Git on your Mac. A current installer for Git can be found at http://code.google.com/p/git-osx-installer. Additionally, there are several GUI front ends for Git, even one written by GitHub, which might be more appealing to developers who avoid command-line interfaces. If you do not want to install Git, GitHub also allows for downloading the source files as a Zip.

GitHub enables users to sign up for a free account at https://github.com/signup/free. After Git has been installed, from the terminal's command line $git clone git@github.com:dfsw/icf.git will download a copy of the source code into the current working directory. You are welcome to fork and open pull requests with the sample code projects.

Contacting the Authors

If you have any comments or questions about this book, please drop us an e-mail message at icf@dragonforged.com, or on Twitter at @kylerichter and @jwkeeley.

Acknowledgments

This book could not have existed without a great deal of effort from far too many behind-the-scenes people; although there are only two authors on the cover, dozens of people were responsible for bringing this book to completion. We would like to thank Trina MacDonald first and foremost; without her leadership and her driving us to meet deadlines, we would never have been able to finish. The editors at Pearson have been exceptionally helpful; their continual efforts show on every page, from catching our typos to pointing out technical concerns. The dedicated work of Dave Wood, Olivia Basegio, Collin Ruffenach, Sheri Cain, Tom Cirtin, Elaine Wiley, and Cheri Clark made the following pages possible.

We would also like to thank Jordan Langille of Langille Design (http://jordanlangille.com) for providing the designs for the Whack-a-Cac game featured in Chapters 3 and 4. His efforts have made the Game Center sample projects much more compelling.

The considerable amount of time spent working on this book was shouldered not only by us but also by our families and co-workers. We would like to thank everyone who surrounds us in our daily lives for taking a considerable amount of work off of our plates, as well as understanding the demands that a project like this brings.

Finally, we would like to thank the community at large. All too often we consulted developer forums, blog posts, and associates to ask questions or provide feedback. Without the hard efforts of everyone involved in the iOS community, this book would not be nearly as complete.

About the Authors

Kyle Richter is the founder of Dragon Forged Software, an award-winning iOS and Mac Development Company, and co-founder of Empirical Development, a for-hire iOS shop. Kyle began writing code in the early 1990s and has always been dedicated to the Mac platform. He has written several books on iOS development, as well as articles on many popular developer blogs and websites. He manages a team of more than 20 full-time iOS developers and runs day-to-day operations at three development companies. Kyle travels the world speaking on development and entrepreneurship; currently he calls Key West his home, where he spends his time with his border collie Landis. He can be found on Twitter at @kylerichter.

Joe Keeley is the CTO of Dragon Forged Software, and Project Lead at Empirical Development. Joe works on Resolve and Slender, and has led a number of successful client projects to completion. He has liked writing code since first keying on an Apple II, and has worked on a wide variety of technology and systems projects in his career. Joe has presented several different technical topics at iOS and Mac conferences around the U.S. Joe lives in Denver, Colorado, with his wife and two daughters, and hopes to get back into competitive fencing again in his spare time. He can be reached on Twitter at @jwkeeley.

This page intentionally left blank

13

Getting Up and Running with Core Data

At first glance, Core Data can look difficult and overwhelming. There are several books devoted solely to Core Data, and the official Apple documentation is lengthy and challenging to get through since it covers the entire breadth and depth of the topic. Most apps do not require all the features that Core Data has to offer. The goal of this chapter is to get you up and running with the most common Core Data features that apps need.

This chapter describes how to set up a project to use Core Data, and illustrates how to implement several common use cases with the sample app. It covers how to set up your data model, how to populate some starting data, and how to display data in a table using a fetched results controller. This chapter also demonstrates how to add, edit, and delete data, how to fetch data, and how to use predicates to fetch specific data. With this knowledge, you will have a good foundation for implementing Core Data quickly in your apps.

Sample App

The sample app for this chapter is called MyMovies. It is a Core Data–based app that will keep track of all your physical media movies and, if you have loaned a movie to someone, who you loaned it to and when (as shown in Figure 13.1).

The sample app has three tabs: Movies, Friends, and Shared Movies. The Movies tab shows the whole list of movies that the user has added and tracked in a table view. There are two sections in the table view demonstrating how data can be segregated with a fetched results controller. Users can add new movies from this tab, and can edit existing movies. The Friends tab lists the friends set up to share movies with, shows which friends have borrowed movies, and allows the user to add and edit friends. The Shared Movies tab displays which movies have currently been shared with friends.

Figure 13.1 Sample App: Movies tab.

Starting a Core Data Project

To start a new Core Data project, open Xcode and select File from the menu, New, and then Project. Xcode will present some project template options to get you started (see Figure 13.2).

The quickest method to start a Core Data project is to select the Master-Detail template. Click Next to specify options for your new project, and then make sure that Use Core Data is selected (see Figure 13.3). This ensures that your project has the Core Data plumbing built in.

When Next is clicked, Xcode creates the project template. The project template includes a "master" view controller, which includes a table view populated by an `NSFetchedResultsController`, a specialized controller that makes pairing Core Data with a table view a snap. The project template includes a "detail" view to display a single data record. In the sample app, the master and detail views have been renamed to fit the project.

> **Note**
>
> To add Core Data to an existing project quickly, create an empty template project with Core Data support as described, and then copy the elements described in the following section, "Core Data Environment," into the existing project. Add a new managed object model file to the project, and be sure to add the Core Data framework to the existing project as well.

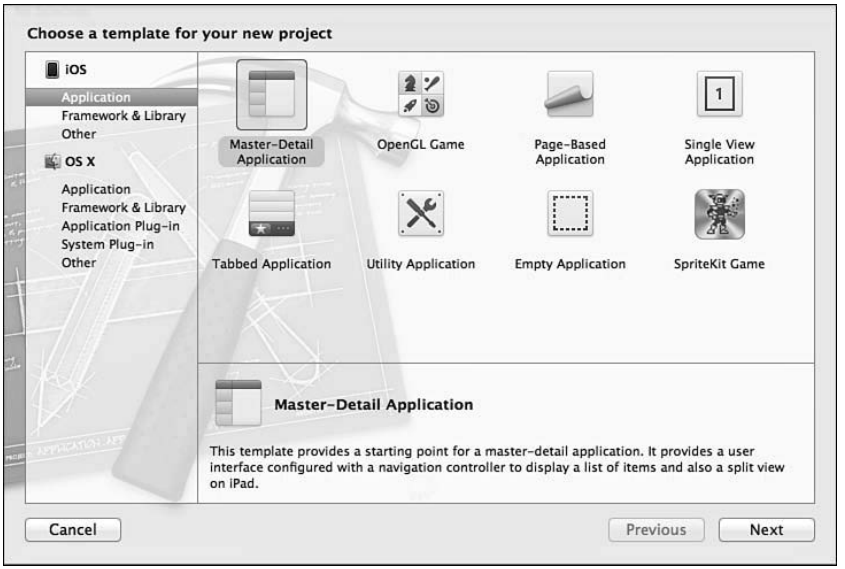

Figure 13.2 Xcode new project template choices.

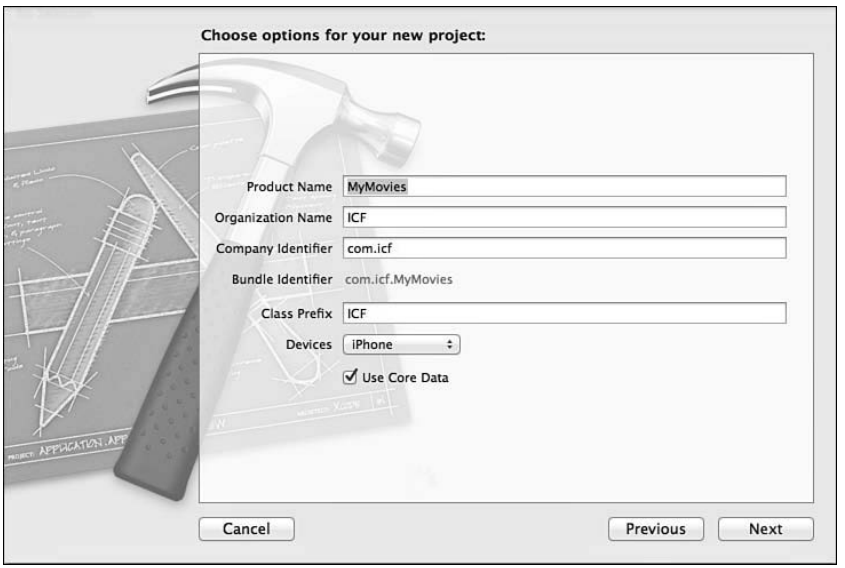

Figure 13.3 Xcode new project options.

Core Data Environment

The project template sets up the Core Data environment for the project in the class that implements the `UIApplicationDelegate` protocol; in the sample app this is `ICFAppDelegate`. The project template uses a lazy-loading pattern for each of the properties needed in the Core Data environment, so each is loaded when needed. For more information about the Core Data environment, refer to Chapter 12, "Core Data Primer."

The process of loading the Core Data environment is kicked off the first time the managed object context is referenced in the app. The managed object context accessor method will check to see whether the managed object context instance variable has a reference. If not, it will get the persistent store coordinator and instantiate a new managed object context with it, assign the new instance to the instance variable, and return the instance variable.

```
- (NSManagedObjectContext *)managedObjectContext
{
  if (__managedObjectContext != nil)
  {
    return __managedObjectContext;
  }

  NSPersistentStoreCoordinator *coordinator =
   [self persistentStoreCoordinator];

  if (coordinator != nil)
  {
    __managedObjectContext =
     [[NSManagedObjectContext alloc] init];

    [__managedObjectContext
     setPersistentStoreCoordinator:coordinator];
  }
  return __managedObjectContext;
}
```

The persistent store coordinator is the class that Core Data uses to manage the persistent stores (or files) where the data for the app is stored. To instantiate it, an instance of `NSManagedObjectModel` is needed so that the persistent store coordinator knows what object model the persistent stores are implementing. The persistent store coordinator also needs a URL for each persistent store to be added; if the file does not exist, Core Data will create it. If the persistent store doesn't match the managed object model (Core Data uses a hash of the managed object model to uniquely identify it, which is kept for comparison in the persistent store), then the template logic will log an error and abort. In a shipping application, logic would be added to properly handle errors with a migration from the old data model to the new one; in development having the app abort can be a useful reminder when the model changes to retest with a clean installation of the app.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
  if (__persistentStoreCoordinator != nil)
  {
    return __persistentStoreCoordinator;
  }

  NSURL *storeURL =
  [[self applicationDocumentsDirectory]
   URLByAppendingPathComponent:@"MyMovies.sqlite"];

  NSError *error = nil;
  __persistentStoreCoordinator =
  [[NSPersistentStoreCoordinator alloc]
   initWithManagedObjectModel:[self managedObjectModel]];

  if (![__persistentStoreCoordinator
     addPersistentStoreWithType:NSSQLiteStoreType
     configuration:nil URL:storeURL options:nil
     error:&error])
  {
    NSLog(@"Unresolved error %@, %@", error,
       [error userInfo]);

    abort();
  }

  return __persistentStoreCoordinator;
}
```

The managed object model is loaded from the app's main bundle. Xcode will give the managed object model the same name as your project.

```
- (NSManagedObjectModel *)managedObjectModel
{
  if (__managedObjectModel != nil)
  {
    return __managedObjectModel;
  }

  NSURL *modelURL =
  [[NSBundle mainBundle] URLForResource:@"MyMovies"
              withExtension:@"momd"];

  __managedObjectModel =
  [[NSManagedObjectModel alloc]
   initWithContentsOfURL:modelURL];

  return __managedObjectModel;
}
```

Building Your Managed Object Model

With the project template, Xcode will create a data model file with the same name as your project. In the sample project this file is called `MyMovies.xdatamodeld`. To edit your data model, click the data model file, and Xcode will present the data model editor (see Figure 13.4).

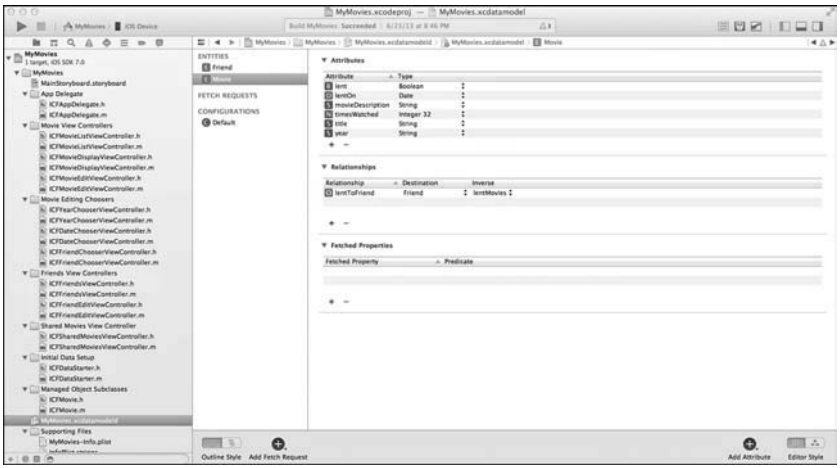

Figure 13.4 Xcode data model editor, Table style.

Xcode has two styles for the data model editor: Table and Graph. The Table style presents the entities in your data model in a list on the left. Selecting an entity will display and allow you to edit the attributes, relationships, and fetched properties for that entity.

To change to Graph style, click the Editor Style Graph button in the lower-right corner of the data model editor (see Figure 13.5). There will still be a list of entities on the left of the data model editor, but the main portion of the editor will present an entity relationship diagram of your data model. Each box presented in the diagram represents an entity, with the name of the entity at the top, the attributes listed in the middle, and any relationships listed in the bottom. The graph will have arrows connecting entities that have relationships established, with arrows indicating the cardinality of the relationship.

When working with your data model, it is often convenient to have more working space available and to have access to additional detail for selected items. Use Xcode's View options in the upper-right corner of the window to hide the Navigator panel and display the Utilities panel (see Figure 13.5).

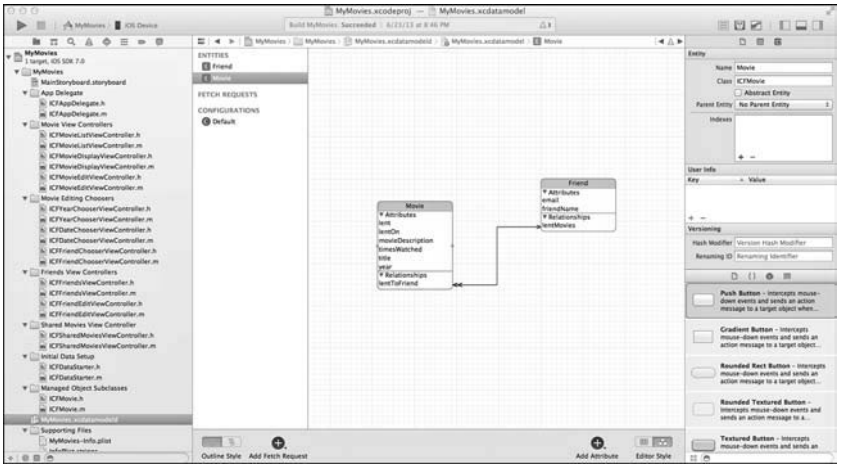

Figure 13.5 Xcode data model editor, Graph style.

Creating an Entity

To create an entity, click the Add Entity button. A new entity will be added to the list of entities, and if the editor is in Graph style, a new entity box will be added to the view. Xcode will highlight the name of the entity and allow you to type in the desired name for the entity. The name of the entity can be changed at any time by just clicking twice on the entity name in the list of entities, or by selecting the desired entity and editing the entity name in the Utilities panel.

Core Data supports entity inheritance. For any entity, you can specify a parent entity from which your entity will inherit attributes, relationships, validations, and custom methods. To do that, ensure that the entity to be inherited from has been created, and then select the child entity and choose the desired parent entity in the Utilities panel.

> **Note**
>
> If you are using SQLite as your persistent store, Core Data implements entity inheritance by creating one table for the parent entity and all child entities, with a superset of all their attributes. This can obviously have unintended performance consequences if you have a lot of data in the entities, so use this feature wisely.

Adding Attributes

To add attributes to an entity, first select the entity in either the graph or the list of entities. Then click the Add Attribute button in the lower part of the editor, just to the left of the Editor Style buttons. Xcode will add an attribute called `attribute` to the entity. Select a Type for the

attribute. (See Table 13.1 for supported data types.) Note that Core Data treats all attributes as Objective-C objects, so if Integer 32 is the selected data type, for example, Core Data will treat the attribute as an `NSNumber`.

One thing to note is that Core Data will automatically give each instance a unique object ID, called `objectID`, which it uses internally to manage storage and relationships. You can also add a unique ID or another candidate key to the entity and add an index to it for quick access, but note that Core Data will manage relationships with the generated object ID.

`NSManagedObjects` also have a method called `description`; if you want to have a `description` attribute, modify the name slightly to avoid conflicts. In the sample app, for example, the Movie entity has a `movieDescription` attribute.

Table 13.1 **Core Data Supported Data Types**

Data Types	Objective-C Storage
Integer 16	`NSNumber`
Integer 32	`NSNumber`
Integer 64	`NSNumber`
Decimal	`NSNumber`
Double	`NSNumber`
Float	`NSNumber`
String	`NSString`
Boolean	`NSNumber`
Date	`NSDate`
Binary Data	`NSData`
Transformable	Uses value transformer

Establishing Relationships

Having relationships between objects can be a very powerful technique to model the real world in an app. Core Data supports one-to-one and one-to-many relationships. In the sample app, a one-to-many relationship between friends and movies is established. Since a friend might borrow more than one movie at a time, that side of the relationship is “many,” but a movie can be lent to only one friend at a time, so that side of the relationship is “one.”

To add a relationship between entities, select one of the entities, and then Ctrl-click and drag to the destination entity. Alternatively, click and hold the Add Attribute button, and select Add Relationship from the menu that appears. Xcode will create a relationship to the destination entity and will call it “relationship.” In the Utilities panel, select the Data Model inspector, and change the name of the relationship. In the Data Model inspector, you can do the following:

- Indicate whether the relationship is transient.
- Specify whether the relationship is optional or required with the Optional check box.
- Specify whether the relationship is ordered.
- Establish an inverse relationship. To do this, create and name the inverse relationship first, and then select it from the drop-down.
- Specify the cardinality of the relationship by checking or unchecking the Plural check box. If checked, it indicates a to-many relationship.
- Specify minimum and maximum counts for a relationship.
- Set up the rule for Core Data to follow for the relationship when the object is deleted. Choices are No Action (no additional action taken on delete), Nullify (relationship set to `nil`), Cascade (objects on the other side of the relationship are deleted too), and Deny (error issued if relationships exist).

Custom Managed Object Subclasses

A custom `NSManagedObject` subclass can be useful if you have custom logic for your model object, or if you would like to be able to use dot syntax for your model object properties and have the compiler validate them.

Xcode has a menu option to automatically create a subclass for you. To use it, ensure that you have completed setup of your entity in the data model editor. Select your entity (or multiple entities) in the data model editor, select Editor from the Xcode menu, and then select Create NSManagedObject Subclass. Xcode will ask where you want to save the generated class files. Specify a location and click Create, and Xcode will generate the header and implementation files for each entity you specified. Xcode will name each class with the class prefix specified for your project concatenated with the name of the entity.

In the generated header file, Xcode will create a property for each attribute in the entity. Note that Xcode will also create a property for each relationship specified for the entity. If the relationship is to-one, Xcode will create an `NSManagedObject` property (or `NSManagedObject` subclass if the destination entity is a custom subclass). If the relationship is to-many, Xcode will create an `NSSet` property.

In the generated implementation file, Xcode will create `@dynamic` instructions for each entity, rather than `@synthesize`. This is because Core Data dynamically handles accessors for Core Data managed attributes, and does not need the compiler to build the accessor methods.

> **Note**
>
> There is a project called mogenerator that will generate two classes per entity: one for the attribute accessors and one for custom logic. That way, you can regenerate classes easily when making model changes without overwriting your custom logic. Mogenerator is available at http://rentzsch.github.com/mogenerator/.

Setting Up Default Data

When a Core Data project is first set up, there is no data in it. Although this might work for some use cases, frequently it is a requirement to have some data prepopulated in the app for the first run. In the sample app there is a custom data setup class called `ICFDataStarter`, which illustrates how to populate Core Data with some initial data. A `#define` variable is set up in `MyMovies-Prefix.pch` called `FIRSTRUN`, which can be uncommented to have the app run the logic in `ICFDataStarter`.

Inserting New Managed Objects

To create a new instance of a managed object for data that does not yet exist in your model, a reference to the managed object context is needed. The sample app uses a constant to refer to the `ICFAppDelegate` instance, which has a property defined for the managed object context:

```
NSManagedObjectContext *moc =
 [kAppDelegate managedObjectContext];
```

To insert data, Core Data needs to know what entity the new data is for. Core Data has a class called `NSEntityDescription` that provides information about entities. Create a new instance using `NSEntityDescription`'s class method:

```
NSManagedObject *newMovie1 =
 [NSEntityDescription insertNewObjectForEntityForName:@"Movie"
                inManagedObjectContext:moc];
```

After an instance is available, populate the attributes with data:

```
[newMovie1 setValue:@"The Matrix" forKey:@"title"];
[newMovie1 setValue:@"1999" forKey:@"year"];

[newMovie1 setValue:@"Take the blue pill."
       forKey:@"movieDescription"];

[newMovie1 setValue:@NO forKey:@"lent"];
[newMovie1 setValue:nil forKey:@"lentOn"];
[newMovie1 setValue:@20 forKey:@"timesWatched"];
```

Core Data uses key-value coding to handle setting attributes. If an attribute name is incorrect, it will fail at runtime. To get compile-time checking of attribute assignments, create a custom `NSManagedObject` subclass and use the property accessors for each attribute directly.

The managed object context acts as a working area for changes, so the sample app sets up more initial data:

```
NSManagedObject *newFriend1 =
 [NSEntityDescription insertNewObjectForEntityForName:@"Friend"
                inManagedObjectContext:moc];
```

```
[newFriend1 setValue:@"Joe" forKey:@"friendName"];
[newFriend1 setValue:@"joe@dragonforged.com" forKey:@"email"];
```

The last step after setting up all the initial data is to save the managed object context.

```
NSError *mocSaveError = nil;

if ([moc save:&mocSaveError])
{
  NSLog(@"Save completed successfully.");
} else
{
  NSLog(@"Save did not complete successfully. Error: %@",
     [mocSaveError localizedDescription]);
}
```

After the managed object context is saved, Core Data will persist the data in the data store. For this instance of the app, the data will continue to be available through shutdowns and restarts. If the app is removed from the simulator or device, the data will no longer be available. One technique to populate data for first run is to copy the data store from the app's storage directory back into the app bundle. This will ensure that the default set of data is copied into the app's directory on first launch and is available to the app.

Other Default Data Setup Techniques

There are two other default data setup techniques that are commonly used: data model version migrations and loading data from a Web service or an API.

Core Data managed object models are versioned. Core Data understands the relationship between the managed object model and the current data store. If the managed object model changes and is no longer compatible with the data store (for example, if an attribute is added to an entity), Core Data will not be able to initiate the persistent store object using the existing data store and new managed object model. In that case, a migration is required to update the existing data store to match the updated managed object model. In many cases Core Data can perform the migration automatically by passing a dictionary of options when instantiating the persistent store; in some cases, additional steps need to be taken to perform the migration. Migrations are beyond the scope of this chapter, but be aware that migrations can be used and are recommended by Apple to do data setup.

The other approach is to pull data from a Web service or an API. This approach is most applicable when an app needs to maintain a local copy of a subset of data on a Web server, and Web calls need to be written for the app to pull data from the API in the course of normal operation. To set up the app's initial data, the Web calls can be run in a special state to pull all needed initial data and save it in Core Data.

Displaying Your Managed Objects

To display or use existing entity data in an app, managed objects need to be fetched from the managed object context. Fetching is analogous to running a query in a relational database, in that you can specify what entity you want to fetch, what criteria you want your results to match, and how you want your results sorted.

Creating Your Fetch Request

The object used to fetch managed objects in Core Data is called `NSFetchRequest`. Refer to `ICFFriendChooserViewController` in the sample app. This view controller displays the friends set up in Core Data and allows the user to select a friend to lend a movie to (see Figure 13.6).

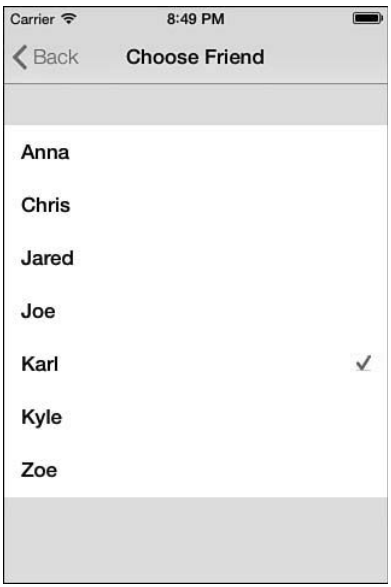

Figure 13.6 Sample App: Friend Chooser.

To get the list of friends to display, the view controller performs a standard fetch request when the view controller has loaded. The first step is to create an instance of `NSFetchRequest` and associate the entity to be fetched with the fetch request:

```
NSManagedObjectContext *moc = kAppDelegate.managedObjectContext;

NSFetchRequest *fetchReq = [[NSFetchRequest alloc] init];

NSEntityDescription *entity =
```

```
 [NSEntityDescription entityForName:@"Friend"
       inManagedObjectContext:moc];

[fetchReq setEntity:entity];
```

The next step is to tell the fetch request how to sort the resulting managed objects. To do this, we associate a sort descriptor with the fetch request, specifying the attribute name to sort by:

```
NSSortDescriptor *sortDescriptor =
 [[NSSortDescriptor alloc] initWithKey:@"friendName"
              ascending:YES];

NSArray *sortDescriptors =
 [NSArray arrayWithObjects:sortDescriptor, nil];

[fetchReq setSortDescriptors:sortDescriptors];
```

Since the friend chooser should show all the available friends to choose from, it is not necessary to specify any matching criteria. All that remains is to execute the fetch:

```
NSError *error = nil;

self.friendList = [moc executeFetchRequest:fetchReq
                                     error:&error];

 if (error)
{
  NSString *errorDesc =
   [error localizedDescription];

  UIAlertView *alert =
   [[UIAlertView alloc] initWithTitle:@"Error fetching friends"
                message:errorDesc
                delegate:nil
           cancelButtonTitle:@"Dismiss"
           otherButtonTitles:nil];
  [alert show];
}
```

To execute a fetch, create an instance of `NSError` and set it to `nil`. Then have the managed object context execute the fetch request that has just been constructed. If an error is encountered, the managed object context will return the error to the instance you just created. The sample app will display the error in an instance of `UIAlertView`. If no error is encountered, the results will be returned as an `NSArray` of `NSManagedObjects`. The view controller will store those results in an instance variable to be displayed in a table view.

Fetching by Object ID

When only one specific managed object needs to be fetched, Core Data provides a way to quickly retrieve that managed object without constructing a fetch request. To use this method, you must have the `NSManagedObjectID` for the managed object.

To get the `NSManagedObjectID` for a managed object, you must already have fetched or created the managed object. Refer to `ICFMovieListViewController` in the sample app, in the `prepareForSegue:sender:` method. In this case, the user has selected a movie from the list, and the view controller is about to segue from the list to the detail view for the selected movie. To inform the detail view controller which movie to display, the `objectID` for the selected movie is set as a property on the `ICFMovieDisplayViewController`:

```
if ([[segue identifier] isEqualToString:@"showDetail"])
{

  NSIndexPath *indexPath =
   [self.tableView indexPathForSelectedRow];

  ICFMovie *movie =
   [[self fetchedResultsController]
    objectAtIndexPath:indexPath];

  ICFMovieDisplayViewController *movieDispVC =
   (ICFMovieDisplayViewController *)
   [segue destinationViewController];

  [movieDispVC setMovieDetailID:[movie objectID]];
}
```

When the `ICFMovieDisplayViewController` is loaded, it uses a method on the managed object context to load a managed object using the `objectID`:

```
ICFMovie *movie = (ICFMovie *)[kAppDelegate.managedObjectContext
             objectWithID:self.movieDetailID];

[self configureViewForMovie:movie];
```

When this is loaded, the movie is available to the view controller to configure the view using the movie data (see Figure 13.7).

It is certainly possible to just pass the managed object from one view controller to the next with no problems, instead of passing the `objectID` and loading the managed object in the destination view controller. However, there are cases when using the `objectID` is highly preferable to using the managed object:

- If the managed object has been fetched or created on a different thread than the destination view controller will use to process and display the managed object—this approach must be used since managed objects are not thread safe!

- If a background thread might update the managed object in another managed object context between fetching and displaying—this will avoid possible issues with displaying the most up-to-date changes.

Figure 13.7 Sample App: Movie Display view.

Displaying Your Object Data

After managed objects have been fetched, accessing and displaying data from them is straightforward. For any managed object, using the key-value approach will work to retrieve attribute values. As an example, refer to the `configureCell:atIndexPath` method in `ICFFriendsViewController` in the sample app. This code will populate the table cell's text label and detail text label.

```
NSManagedObject *object =
 [self.fetchedResultsController objectAtIndexPath:indexPath];

cell.textLabel.text = [object valueForKey:@"friendName"];

NSInteger numShares = [[object valueForKey:@"lentMovies"] count];

NSString *subtitle = @"";

switch (numShares)
```

```
{
  case 0:
    subtitle = @"Not borrowing any movies.";
    break;

  case 1:
    subtitle = @"Borrowing 1 movie.";
    break;

  default:
    subtitle =
     [NSString stringWithFormat:@"Borrowing %d movies.",
     numShares];

    break;
}

cell.detailTextLabel.text = subtitle;
```

To get the attribute values from the managed object, call `valueForKey:` and specify the attribute name. If the attribute name is specified incorrectly, the app will fail at runtime.

For managed object subclasses, the attribute values are also accessible by calling the property on the managed object subclass with the attribute name. Refer to the `configureViewForMovie:` method in `ICFMovieDisplayViewController` in the sample app.

```
- (void)configureViewForMovie:(ICFMovie *)movie

{
  NSString *movieTitleYear = [movie yearAndTitle];

  [self.movieTitleAndYearLabel
   setText:movieTitleYear];

  [self.movieDescription setText:[movie movieDescription]];

  BOOL movieLent = [[movie lent] boolValue];

  NSString *movieShared = @"Not Shared";
  if (movieLent)
  {
    NSManagedObject *friend =
    [movie valueForKey:@"lentToFriend"];

    NSDateFormatter *dateFormatter =
     [[NSDateFormatter alloc] init];

    [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
```

```
    NSString *sharedDateTxt =
    [dateFormatter stringFromDate:[movie lentOn]];

    movieShared =
     [NSString stringWithFormat:@"Shared with %@ on %@",
     [friend valueForKey:@"friendName"],sharedDateTxt];
  }

  [self.movieSharedInfoLabel setText:msh];
}
```

If the property-based approach to get attribute values from managed object subclasses is used, errors will be caught at compile time.

Using Predicates

Predicates can be used to narrow down your fetch results to data that match your specific criteria. They are analogous to a `where` clause in an SQL statement, but they can be used to filter elements from a collection (like an `NSArray`) as well as a fetch request from Core Data.
To see how a predicate is applied to a fetch request, refer to method `fetchedResults Controller` in `ICFSharedMoviesViewController`. This method lazy loads and sets up an `NSFetchedResultsController`, which helps a table view interact with the results of a fetch request (this is described in detail in the next section). Setting up a predicate is simple, for example:

```
NSPredicate *predicate =
 [NSPredicate predicateWithFormat:@"lent == %@",@YES];
```

In the format string, predicates can be constructed with attribute names, comparison operators, Boolean operators, aggregate operators, and substitution expressions. A comma-separated list of expressions will be substituted in the order of the substitution expressions in the format string. Dot notation can be used to specify relationships in the predicate format string. Predicates support a large variety of operators and arguments, as shown in Table 13.2.

Table 13.2 **Core Data Predicate Supported Operators and Arguments**

Type	Operators and Arguments
Basic Comparisons	`=`, `==`, `>=`, `=>`, `<=`, `=<`, `>`, `<`, `!=`, `<>`, `BETWEEN {low,high}`
Boolean	`AND`, `&&` `OR`, `||` `NOT`, `!`
String	`BEGINSWITH`, `CONTAINS`, `ENDSWITH`, `LIKE`, `MATCHES`
Aggregate	`ANY`, `SOME`, `ALL`, `NONE`, `IN`
Literals	`FALSE`, `NO`, `TRUE`, `YES`, `NULL`, `NIL`, `SELF`. Core Data also supports string and numeric literals.

Tell the fetch request to use the predicate:

```
[fetchRequest setPredicate:predicate];
```

Now the fetch request will narrow the returned result set of managed objects to match the criteria specified in the predicate (see Figure 13.8).

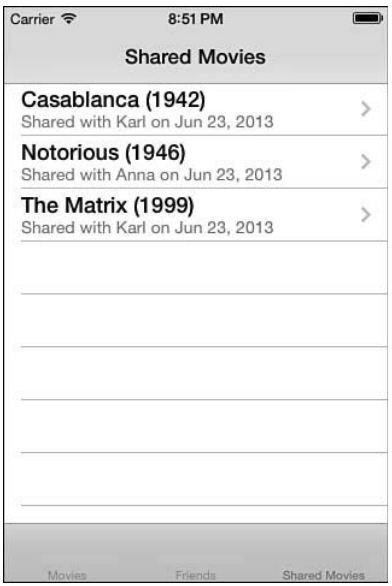

Figure 13.8 Sample App: Shared Movies tab.

Introducing the Fetched Results Controller

A fetched results controller (`NSFetchedResultsController`) is a very effective liaison between Core Data and a `UITableView`. The fetched results controller provides a way to set up a fetch request so that the results are returned in sections and rows, accessible by index paths. In addition, the fetched results controller can listen to changes in Core Data and update the table accordingly using delegate methods.

In the sample app, refer to `ICFMovieListViewController` for a detailed example of a fetched results controller in action (see Figure 13.9).

Preparing the Fetched Results Controller

When the "master" view controller is set up using Xcode's Master Detail template, Xcode creates a property for the fetched results controller, and overrides the accessor method

(`fetchedResultsController`) to lazy load or initialize the fetched results controller the first time it is requested. First the method checks to see whether the fetched results controller has already been initialized:

```
if (__fetchedResultsController != nil)
{
  return __fetchedResultsController;
}
```

Figure 13.9 Sample App: Movie list view controller.

If the fetched results controller is already set up, it is returned. Otherwise, a new fetched results controller is set up, starting with a fetch request:

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
```

The fetch request needs to be associated with an entity from the managed object model, and a managed object context:

```
NSManagedObjectContext *moc = kAppDelegate.managedObjectContext;

NSEntityDescription *entity =
 [NSEntityDescription entityForName:@"Movie"
       inManagedObjectContext:moc];

[fetchRequest setEntity:entity];
```

A batch size can be set up to prevent the fetch request from fetching too many records at once:

```
[fetchRequest setFetchBatchSize:20];
```

Next, the sort order is established for the fetch request using `NSSortDescriptor` instances. An important point to note is that the attribute used for sections needs to be the first in the sort order so that the records can be correctly divided into sections. The sort order is determined by the order of the sort descriptors in the array of sort descriptors attached to the fetch request.

```
NSSortDescriptor *sortDescriptor =
 [[NSSortDescriptor alloc] initWithKey:@"title"
               ascending:YES];

NSSortDescriptor *sharedSortDescriptor =
 [[NSSortDescriptor alloc] initWithKey:@"lent" ascending:NO];

NSArray *sortDescriptors = [NSArray arrayWithObjects:
              sharedSortDescriptor,sortDescriptor,
              nil];

[fetchRequest setSortDescriptors:sortDescriptors];
```

After the fetch request is ready, the fetched results controller can be initialized. It requires a fetch request, a managed object context, a key path or an attribute name to be used for the table view sections, and a name for a cache (if `nil` is passed, no caching is done). The fetched results controller can specify a delegate that will respond to any Core Data changes. When this is complete, the fetched results controller is assigned to the view controller's property:

```
NSFetchedResultsController *aFetchedResultsController =
 [[NSFetchedResultsController alloc]
 initWithFetchRequest:fetchRequest
 managedObjectContext:moc
  sectionNameKeyPath:@"lent"
       cacheName:nil];

aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;
```

Now that the fetched results controller has been prepared, the fetch can be executed to obtain a result set the table view can display, and the fetched results controller can be returned to the caller:

```
NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error])
{
  NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  abort();
}

return __fetchedResultsController;
```

Integrating Table View and Fetched Results Controller

Integrating the table view and fetched results controller is just a matter of updating the table view's datasource and delegate methods to use information from the fetched results controller. In `ICFMovieListViewController`, the fetched results controller tells the table view how many sections it has:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
  return [[self.fetchedResultsController sections] count];
}
```

The fetched results controller tells the table view how many rows are in each section, using the `NSFetchedResultsSectionInfo` protocol:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
   [[self.fetchedResultsController sections]
   objectAtIndex:section];

  return [sectionInfo numberOfObjects];
}
```

The fetched results controller provides section titles, which are the values of the attribute specified as the section name. Since the sample app is using a Boolean attribute for the sections, the values that the fetched results controller returns for section titles are not user-friendly titles: `0` and `1`. The sample app looks at the titles from the fetched results controller and returns more helpful titles: `Shared` instead of `1` and `Not Shared` instead of `0`.

```
- (NSString *)tableView:(UITableView *)tableView
 titleForHeaderInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
   [[self.fetchedResultsController sections]
   objectAtIndex:section];

  if ([[sectionInfo indexTitle] isEqualToString:@"1"])
  {
    return @"Shared";
  }
  else
  {
    return @"Not Shared";
  }
}
```

To populate the table cells, the sample app dequeues a reusable cell, and then calls the `configureView:` method, passing the `indexPath` for the cell:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell =
   [tableView dequeueReusableCellWithIdentifier:@"Cell"];

  [self configureCell:cell atIndexPath:indexPath];

  return cell;
}
```

The fetched results controller knows which movie should be displayed at each index path, so the sample app can get the correct movie to display by calling the `objectAtIndexPath:` method on the fetched results controller. Then, it is simple to update the cell with data from the movie instance.

```
- (void)configureCell:(UITableViewCell *)cell
     atIndexPath:(NSIndexPath *)indexPath
{
  ICFMovie *movie =
   [self.fetchedResultsController objectAtIndexPath:indexPath];

  cell.textLabel.text = [movie cellTitle];

  cell.detailTextLabel.text = [movie movieDescription];
}
```

The last table-view integration detail would typically be handling table cell selection in the `tableView:didSelectRowAtIndexPath:` method. In this case, no integration in that method is needed since selection is handled by storyboard segue. In the `prepareForSegue:sender:` method, selection of a table cell is handled with an identifier called `showDetail`:

```
if ([[segue identifier] isEqualToString:@"showDetail"])
{
NSIndexPath *indexPath =
     [self.tableView indexPathForSelectedRow];

    ICFMovie *movie =
     [[self fetchedResultsController]
       objectAtIndexPath:indexPath];

    ICFMovieDisplayViewController *movieDisplayVC =
     (ICFMovieDisplayViewController *)
      [segue destinationViewController];

    [movieDisplayVC setMovieDetailID:[movie objectID]];
}
```

This method gets the index path of the selected row from the table view, and then gets the movie instance from the fetched results controller using the index path. The method then sets the `movieDetailID` of the `ICFMovieDisplayViewController` instance with the movie instance's `objectID`.

Responding to Core Data Changes

For the fetched results controller to respond to Core Data changes and update the table view, methods from the `NSFetchedResultsControllerDelegate` protocol need to be implemented. First the view controller needs to declare that it will implement the delegate methods:

```
@interface ICFMovieListViewController : UITableViewController
 <NSFetchedResultsControllerDelegate>
```

The fetched results controller delegate will be notified when content will be changed, giving the delegate the opportunity to animate the changes in the table view. Calling the `beginUpdates` method on the table view tells it that all updates until `endUpdates` is called should be animated simultaneously.

```
- (void)controllerWillChangeContent:
  (NSFetchedResultsController *)controller
{
  [self.tableView beginUpdates];
}
```

There are two delegate methods that might be called based on data changes. One method will tell the delegate that changes occurred that affect the table-view sections; the other will tell the delegate that the changes affect objects at specified index paths, so the table view will need to update the associated rows. Because the data changes are expressed by type, the delegate will be notified if the change is an insert, a delete, a move, or an update, so a typical pattern is to build a `switch` statement to perform the correct action by change type. For sections, the sample app will only make changes that can insert or delete a section (if a section name is changed, that might trigger a case in which a section might move and be updated as well).

```
- (void)controller:(NSFetchedResultsController *)controller
 didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
      atIndex:(NSUInteger)sectionIndex
   forChangeType:(NSFetchedResultsChangeType)type
{
  switch(type)
  {
    case NSFetchedResultsChangeInsert:
      ...
      break;

    case NSFetchedResultsChangeDelete:
      ...
      break;
  }
}
```

Table views have a convenient method to insert new sections, and the delegate method receives all the necessary information to insert new sections:

```
[self.tableView
  insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
 withRowAnimation:UITableViewRowAnimationFade];
```

Removing sections is just as convenient:

```
[self.tableView
  deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
 withRowAnimation:UITableViewRowAnimationFade];
```

For object changes, the delegate will be informed of the change type, the object that changed, the current index path for the object, and a "new" index path if the object is being inserted or moved. Using `switch` logic to respond by change type works for this method as well.

```
- (void)controller:(NSFetchedResultsController *)controller
  didChangeObject:(id)anObject
    atIndexPath:(NSIndexPath *)indexPath
   forChangeType:(NSFetchedResultsChangeType)type
   newIndexPath:(NSIndexPath *)newIndexPath
{
  UITableView *tableView = self.tableView;

  switch(type)
  {
    case NSFetchedResultsChangeInsert:
      ...
      break;

    case NSFetchedResultsChangeDelete:
      ...
      break;

    case NSFetchedResultsChangeUpdate:
      ...
      break;

    case NSFetchedResultsChangeMove:
      ...
      break;
  }
}
```

Table views have convenience methods to insert rows by index path. Note that the `newIndexPath` is the correct index path to use when inserting a row for an inserted object.

```
[tableView
 insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
    withRowAnimation:UITableViewRowAnimationFade];
```

To delete a row, use the `indexPath` passed to the delegate method.

```
[tableView
 deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationFade];
```

To update a row, call the `configureCell:atIndexPath:` method for the current `indexPath`. This is the same configure method called from the table view delegate's `tableView:cellForRowAtIndexPath:` method.

```
[self configureCell:[tableView cellForRowAtIndexPath:indexPath]
    atIndexPath:indexPath];
```

To move a row, delete the row for the current `indexPath` and insert a row for the `newIndexPath`.

```
[tableView
 deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationFade];

[tableView
 insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
    withRowAnimation:UITableViewRowAnimationFade];
```

The fetched results controller delegate will be notified when the content changes are complete, so the delegate can tell the table view there will be no more animated changes by calling the `endUpdates` method. After that method is called, the table view will animate the accumulated changes in the user interface.

```
- (void)controllerDidChangeContent:
  (NSFetchedResultsController *)controller
{
  [self.tableView endUpdates];
}
```

Adding, Editing, and Removing Managed Objects

Although it is useful to be able to fetch and display data, apps often need to add new data, edit existing data, and remove unneeded data at the user's request.

Inserting a New Managed Object

In the sample app, view the Movies tab. To insert a new movie, the user can tap the Add button in the navigation bar. The Add button is wired to perform a segue to the

`ICFMovieEditViewController`. In the segue logic, a new movie managed object is inserted into Core Data, and the new movie's object ID is passed to the edit movie view controller. This approach is used in the sample app to prevent having logic in the edit view controller to handle both creating new managed objects and editing existing managed objects; however, it would be perfectly acceptable to create the new movie managed object in the edit view controller if that makes more sense in a different app.

To create a new instance of a movie managed object, a reference to the managed object context is needed.

```
NSManagedObjectContext *moc =
 [kAppDelegate managedObjectContext];
```

To insert data, Core Data needs to know what entity the new data is for. Core Data has a class called `NSEntityDescription` that provides information about entities. Create a new instance using `NSEntityDescription`'s class method:

```
ICFMovie *newMovie = [NSEntityDescription
 insertNewObjectForEntityForName:@"Movie"
     inManagedObjectContext:moc];
```

Populate the new movie managed object's attributes with data:

```
[newMovie setTitle:@"New Movie"];
[newMovie setYear:@"2012"];
[newMovie setMovieDescription:@"New movie description."];
[newMovie setLent:@NO];
[newMovie setLentOn:nil];
[newMovie setTimesWatched:@0];
```

Prepare an `NSError` variable to capture any potential errors, and save the managed object context.

```
NSError *mocSaveError = nil;

if (![moc save:&mocSaveError])
{
  NSLog(@"Save did not complete successfully. Error: %@",
     [mocSaveError localizedDescription]);
}
```

After the managed object context has been successfully saved, the fetched results controller will be notified if the save affects the results of the controller's fetch, and the delegate methods described earlier in the chapter will be called.

Removing a Managed Object

On the Movies tab in the sample app, the user can swipe on the right side of a table cell, or can tap the Edit button to reveal the delete controls for each table cell. When Delete is tapped

on a cell, the table view delegate method `tableView:commitEditingStyle:forRowAtIndexPath:` is called. The method checks whether the editing style is delete. If so, the method gets a reference to the managed object context from the fetched results controller. The fetched results controller keeps a reference to the managed object context it was initialized with, which is needed to delete the object.

```
NSManagedObjectContext *context =
 [self.fetchedResultsController managedObjectContext];
```

The method determines which managed object should be deleted, by asking the fetched results controller for the managed object at the specified index path.

```
NSManagedObject *objectToBeDeleted =
 [self.fetchedResultsController objectAtIndexPath:indexPath];
```

To delete the managed object, the method tells the managed object context to delete it.

```
[context deleteObject:objectToBeDeleted];
```

The deletion is not permanent until the managed object context is saved. After it is saved, the delegate methods described earlier in the chapter will be called and the table will be updated.

```
NSError *error = nil;
if (![context save:&error])
{
  NSLog(@"Error deleting movie, %@", [error userInfo]);
}
```

Editing an Existing Managed Object

On the Movies tab in the sample app, the user can tap a movie to see more detail about it. To change any of the information about the movie, tap the Edit button in the navigation bar, which will present an instance of `ICFMovieEditViewController`. When the view is loaded, it will load an instance of `ICFMovie` using the `objectID` passed in from the display view or list view, will save that instance into the property `editMovie`, and will configure the view using information from the movie managed object.

If the user decides to edit the year of the movie, for example, another view controller will be presented with a `UIPickerView` for the user to select a new year. The `ICFMovieEditViewController` is set up as a delegate for the year chooser, so when the user has selected a new year and taps Save, the delegate method `chooserSelectedYear:` is called. In that method, the `editMovie` is updated with the new date and the display is updated.

```
- (void)chooserSelectedYear:(NSString *)year
{
    [self.editMovie setYear:year];
    [self.movieYearLabel setText:year];
}
```

Note that the managed object context was not saved after `editMovie` was updated. The managed object `editMovie` can keep updates temporarily until the user makes a decision about whether to make the changes permanent, indicated by tapping the Save or Cancel button.

Saving and Rolling Back Your Changes

If the user taps the Save button, he has indicated his intention to keep the changes made to the `editMovie`. In the `saveButtonTouched:` method, the fields not updated with delegate methods are saved to the `editMovie` property:

```
NSString *movieTitle = [self.movieTitle text];
[self.editMovie setTitle:movieTitle];

NSString *movieDesc = [self.movieDescription text];
[self.editMovie setMovieDescription:movieDesc];

BOOL sharedBool = [self.sharedSwitch isOn];
NSNumber *shared = [NSNumber numberWithBool:sharedBool];
[self.editMovie setLent:shared];
```

Then the managed object context is saved, making the changes permanent.

```
NSError *saveError = nil;
[kAppDelegate.managedObjectContext save:&saveError];
if (saveError)
{

  UIAlertView *alert =
   [[UIAlertView alloc]
    initWithTitle:@"Error saving movie"
         message:[saveError localizedDescription]
        delegate:nil
    cancelButtonTitle:@"Dismiss"
    otherButtonTitles:nil];

  [alert show];
}
else
{
  NSLog(@"Changes to movie saved.");
}
```

If the user decides that the changes should be thrown away and not be made permanent, the user will tap the Cancel button, which calls the `cancelButtonTouched:` method. That method will first check whether the managed object context has any unsaved changes. If so, the method will instruct the managed object context to roll back or throw away the unsaved changes. After that is completed, the managed object context will be back to the state it was in

before any of the changes were made. Rather than the user interface being updated to reflect throwing away the changes, the view is dismissed.

```
if ([kAppDelegate.managedObjectContext hasChanges])
{
  [kAppDelegate.managedObjectContext rollback];
  NSLog(@"Rolled back changes.");
}

[self.navigationController.presentingViewController
 dismissModalViewControllerAnimated:YES];
```

Summary

This chapter described how to set up a new project to use Core Data and how to set up all the Core Data environment pieces. The chapter detailed how to create a managed object model, including how to add a new entity, add attributes to an entity, and set up relationships between entities. It also described why an `NSManagedObject` subclass is useful and how to create one.

This chapter explained how to set up some initial data for the project, and demonstrated how to insert new managed objects. Alternative techniques for initial data setup were discussed.

This chapter then detailed how to create a fetch request to get saved managed objects, and how to fetch individual managed objects using an `objectID`. It described how to display data from managed objects in the user interface of an app. It explained how to use predicates to fetch managed objects that match specific criteria.

This chapter introduced the fetched results controller, a powerful tool for integrating Core Data with the `UITableView`; described how to set up a `UITableView` with a fetched results controller; and explained how to set up a fetched results controller delegate to automatically update a table view from Core Data changes.

Lastly, this chapter explained how to add, edit, and delete managed objects, and how to save changes or roll back unwanted changes.

With all of these tools, you should now have a good foundation for using Core Data effectively in your apps.

Exercises

1. For the `Friend` managed object, there is an attribute specified for `email`, but the `ICFFriendEditViewController` does not support editing the email address. Update that view controller to display the friend's email address when the view is displayed, allow the user to edit the email address, and ensure that the email address is saved when the Save button is tapped. Test tapping the Cancel button to make sure that

changes to the email address are thrown away. Then add a Remind Friend button to the `ICFMovieDisplayViewController` that will present an email message composer with the friend's email address.

2. Currently, the friends list is not very helpful, because the user can see which friends have borrowed movies but cannot see which movies the friends have borrowed. Change the accessory selection to edit the friend record instead of row selection, and add a new view controller to display all the movies a friend has borrowed when the row for a friend is selected. Hint: Use a table view controller backed with a fetched results controller to display the list of movies borrowed by a friend.
3. Change the sort order in `ICFMovieListViewController` to sort by year instead of title, without messing up the Shared/Not Shared section headers. Hint: This is a very simple change.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V-W

X-Z