

iOS5  
Compliant

Store and retrieve your app data  
accurately and efficiently



# Pro Core Data for iOS

SECOND EDITION

Michael Privat | Robert Warner

Apress®

[www.it-ebooks.info](http://www.it-ebooks.info)

---

# Contents at a Glance

|   |                                    |
|---|------------------------------------|
| <b>Contents.....</b>  | <b><u><a href="#">v</a></u></b>    |
| <b>About the Authors.....</b>   | <b><u><a href="#">x</a></u></b>    |
| <b>About the Technical Reviewer .....</b>   | <b><u><a href="#">xi</a></u></b>   |
| <b>Acknowledgments .....</b>  | <b><u><a href="#">xii</a></u></b>  |
| <b>Introduction .....</b>   | <b><u><a href="#">xiii</a></u></b> |
| <b>■ Chapter 1: Getting Started.....</b>  | <b><u><a href="#">1</a></u></b>    |
| <b>■ Chapter 2: Understanding Core Data .....</b>                                 | <b><u><a href="#">27</a></u></b>   |
| <b>■ Chapter 3: Storing Data: SQLite and Other Options .....</b>                  | <b><u><a href="#">59</a></u></b>   |
| <b>■ Chapter 4: Creating a Data Model.....</b>                                    | <b><u><a href="#">111</a></u></b>  |
| <b>■ Chapter 5: Working with Data Objects .....</b>                               | <b><u><a href="#">133</a></u></b>  |
| <b>■ Chapter 6: Refining Result Sets.....</b>                                     | <b><u><a href="#">187</a></u></b>  |
| <b>■ Chapter 7: Tuning Performance and Memory Usage.....</b>                      | <b><u><a href="#">209</a></u></b>  |
| <b>■ Chapter 8: Versioning and Migrating Data .....</b>                           | <b><u><a href="#">253</a></u></b>  |
| <b>■ Chapter 9: Managing Table Views Using a Fetched Results Controller .....</b> | <b><u><a href="#">285</a></u></b>  |
| <b>■ Chapter 10: Using Core Data in Advanced Applications .....</b>               | <b><u><a href="#">307</a></u></b>  |
| <b>Index .....</b>  | <b><u><a href="#">367</a></u></b>  |

# Pro Core Data for iOS

Data Access and Persistence Engine for iPhone, iPad,  
and iPod touch

Second Edition



**Michael Privat and  
Robert Warner**

**Apress®**

## **Pro Core Data for iOS: Data Access and Persistence Engine for iPhone, iPad, and iPod touch Second Edition**

Copyright © 2011 by Michael Privat and Robert Warner

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-3656-6

ISBN-13 (electronic): 978-1-4302-3567-3

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning

Lead Editor: Steve Anglin

Development Editors: Matthew Moodie and Douglas Pundick

Technical Reviewer: Robert Hamilton

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel,

Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman,

James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick,

Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh

Coordinating Editor: Jennifer L. Blackwell

Copy Editor: Mary Behr

Compositor: MacPS, LLC

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit [www.springeronline.com](http://www.springeronline.com).

For information on translations, please e-mail [rights@apress.com](mailto:rights@apress.com), or visit [www.apress.com](http://www.apress.com).

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at [www.apress.com/bulk-sales](http://www.apress.com/bulk-sales).

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Any source code or other supplementary materials referenced by the author in this text is available to readers at [www.apress.com](http://www.apress.com). For detailed information about how to locate your book’s source code, go to <http://www.apress.com/source-code/>.



*To my loving wife, Kelly, and our children, Matthieu and Chloé.*

*—Michael Privat*

*To my beautiful wife, Sherry, and our wonderful children:  
Tyson, Jacob, Mallory, Camie, and Leila.*

*—Rob Warner*

---

# Contents

|   |                    |
|---|--------------------|
| <b>Contents at a Glance .....</b>                 | <b><u>iv</u></b>   |
| <b>About the Authors .....</b>                    | <b><u>x</u></b>    |
| <b>About the Technical Reviewer .....</b>         | <b><u>xi</u></b>   |
| <b>Acknowledgments .....</b>                      | <b><u>xii</u></b>  |
| <b>Introduction .....</b>                         | <b><u>xiii</u></b> |
| <br>  |                    |
| <b>■ Chapter 1: Getting Started .....</b>         | <b><u>1</u></b>    |
| What Is Core Data? .....                          | <u>1</u>           |
| History of Persistence in iOS .....               | <u>2</u>           |
| Creating a Basic Core Data Application .....      | <u>3</u>           |
| Understanding the Core Data Components .....      | <u>3</u>           |
| Creating a New Project .....                      | <u>5</u>           |
| Running Your New Project .....                    | <u>6</u>           |
| Understanding the Application's Components .....  | <u>9</u>           |
| Fetching Results .....                            | <u>10</u>          |
| Inserting New Objects .....                       | <u>13</u>          |
| Initializing the Managed Context .....            | <u>14</u>          |
| Adding Core Data to an Existing Project .....     | <u>16</u>          |
| Adding the Core Data Framework .....              | <u>16</u>          |
| Creating the Data Model .....                     | <u>19</u>          |
| Initializing the Managed Object Context .....     | <u>22</u>          |
| Summary .....                                     | <u>25</u>          |
| <br>  |                    |
| <b>■ Chapter 2: Understanding Core Data .....</b> | <b><u>27</u></b>   |
| Core Data Framework Classes .....                 | <u>27</u>          |
| The Model Definition Classes .....                | <u>30</u>          |
| The Data Access Classes .....                     | <u>38</u>          |
| Key-Value Observing .....                         | <u>43</u>          |
| The Query Classes .....                           | <u>44</u>          |
| How the Classes Interact .....                    | <u>47</u>          |
| SQLite Primer .....                               | <u>53</u>          |
| Reading the Data Using Core Data .....            | <u>55</u>          |
| Summary .....                                     | <u>57</u>          |

|  |            |
|--|------------|
| <b>Chapter 3: Storing Data: SQLite and Other Options .....</b>   | <b>59</b>  |
| Visualizing the User Interface .....                             | 60         |
| Using SQLite as the Persistent Store .....                       | 63         |
| Configuring the One-to-Many Relationship .....                   | 67         |
| Building the User Interface .....                                | 69         |
| Configuring the Table .....                                      | 72         |
| Creating a Team .....  | 72         |
| The Player User Interface .....                                  | 81         |
| Adding, Editing, and Deleting Players .....                      | 84         |
| Seeing the Data in the Persistent Store .....                    | 89         |
| Using an In-Memory Persistent Store .....                        | 92         |
| Creating Your Own Custom Persistent Store .....                  | 94         |
| Initializing the Custom Store .....                              | 95         |
| Mapping Between NSManagedObject and NSAtomicStoreCacheNode ..... | 98         |
| Serializing the Data .....                                       | 101        |
| Using the Custom Store .....                                     | 106        |
| What About XML Persistent Stores? .....                          | 107        |
| Summary .....  | 110        |
| <b>Chapter 4: Creating a Data Model.....</b>                     | <b>111</b> |
| Designing Your Database .....                                    | 111        |
| Relational Database Normalization .....                          | 112        |
| Using the Xcode Data Modeler .....                               | 113        |
| Viewing and Editing Attribute Details .....                      | 119        |
| Viewing and Editing Relationship Details .....                   | 120        |
| Using Fetched Properties .....                                   | 121        |
| Creating Entities.....   | 123        |
| Creating Attributes.....   | 125        |
| Creating Relationships.....                                      | 127        |
| Name .....   | 128        |
| Destination and Inverse .....                                    | 129        |
| Transient.....   | 129        |
| Optional.....  | 129        |
| To-Many Relationship .....                                       | 130        |
| Count (Minimum and Maximum) .....                                | 130        |
| Delete Rule.....   | 130        |
| Summary .....  | 131        |
| <b>Chapter 5: Working with Data Objects .....</b>                | <b>133</b> |
| Understanding CRUD.....  | 133        |
| Creating the Shape Application Data Model .....                  | 137        |
| Building the Shape Application User Interface .....              | 145        |
| Enabling User Interactions with the Shapes Application .....     | 154        |
| Generating Classes .....   | 156        |
| Modifying Generated Classes .....                                | 164        |
| Using the Transformable Type .....                               | 169        |
| Validating Data.....   | 173        |
| Custom Validation .....  | 175        |
| Invoking Validation .....  | 179        |

|   |            |
|---|------------|
| Default Values .....  | 179        |
| Undoing and Redoing .....                                   | 180        |
| Undo Groups .....   | 181        |
| Limiting the Undo Stack .....                               | 181        |
| Disabling Undo Tracking .....                               | 182        |
| Adding Undo to Shapes .....                                 | 182        |
| Summary .....   | 185        |
| <b>Chapter 6: Refining Result Sets .....</b>                | <b>187</b> |
| Building the Test Application .....                         | 187        |
| Creating the Org Chart Data .....                           | 188        |
| Reading and Outputting the Data .....                       | 191        |
| Filtering .....   | 192        |
| Expressions for a Single Value .....                        | 193        |
| Expressions for a Collection .....                          | 194        |
| Comparison Predicates .....                                 | 195        |
| Compound Predicates .....                                   | 198        |
| Subqueries .....  | 200        |
| Aggregating .....   | 203        |
| Sorting .....   | 204        |
| Returning Unsorted Data .....                               | 204        |
| Sorting Data on One Criterion .....                         | 205        |
| Sorting on Multiple Criteria .....                          | 206        |
| Summary .....   | 207        |
| <b>Chapter 7: Tuning Performance and Memory Usage .....</b> | <b>209</b> |
| Building the Application for Testing .....                  | 209        |
| Creating the Core Data Project .....                        | 210        |
| Creating the Data Model and Data .....                      | 213        |
| Creating the Testing View .....                             | 215        |
| Building the Testing Framework .....                        | 218        |
| Adding the Testing Framework to the Application .....       | 220        |
| Running Your First Test .....                               | 222        |
| Faulting .....  | 223        |
| Firing Faults .....   | 224        |
| Faulting and Caching .....                                  | 225        |
| Refaulting .....  | 225        |
| Building the Faulting Test .....                            | 226        |
| Taking Control: Firing Faults on Purpose .....              | 229        |
| Prefetching .....   | 231        |
| Caching .....   | 233        |
| Expiring .....  | 236        |
| Memory Consumption .....                                    | 236        |
| Brute-Force Cache Expiration .....                          | 236        |
| Expiring the Cache Through Faulting .....                   | 237        |
| Uniquing .....  | 237        |
| Improve Performance with Better Predicates .....            | 241        |
| Using Faster Computers .....                                | 241        |
| Using Subqueries .....                                      | 242        |

|   |            |
|---|------------|
| Analyzing Performance .....   | 245        |
| Launching Instruments .....   | 245        |
| Understanding the Results .....   | 249        |
| Summary .....   | 251        |
| <b>Chapter 8: Versioning and Migrating Data .....</b>                           | <b>253</b> |
| Versioning .....  | 254        |
| Lightweight Migrations .....  | 257        |
| Migrating a Simple Change .....   | 258        |
| Migrating More Complex Changes .....  | 259        |
| Renaming Entities and Properties .....  | 260        |
| Creating a Mapping Model .....  | 262        |
| Understanding Entity Mappings .....   | 263        |
| Understanding Property Mappings .....   | 264        |
| Creating a New Model Version That Requires a Mapping Model .....                | 266        |
| Creating a Mapping Model .....  | 270        |
| Migrating Data .....  | 276        |
| Running Your Migration .....  | 277        |
| Custom Migrations .....   | 278        |
| Making Sure Migration Is Needed .....   | 280        |
| Setting Up the Migration Manager .....  | 281        |
| Running the Migration .....   | 282        |
| Summary .....   | 284        |
| <b>Chapter 9: Managing Table Views Using a Fetched Results Controller .....</b> | <b>285</b> |
| Understanding NSFetchedResultsController .....                                  | 285        |
| The Fetch Request .....   | 286        |
| The Managed Object Context .....  | 286        |
| The Section Name Key Path .....   | 286        |
| The Cache Name .....  | 287        |
| Understanding NSFetchedResultsController Delegates .....                        | 287        |
| Using NSFetchedResultsController .....  | 288        |
| Implementing NSFetchedResultsController .....                                   | 288        |
| Implementing the NSFetchedResultsController .....                               | 293        |
| Implementing the NSFetchedResultsControllerDelegate Protocol .....              | 298        |
| Indexing Your Table .....   | 298        |
| Responding to Data Change .....   | 302        |
| Summary .....   | 305        |
| <b>Chapter 10: Using Core Data in Advanced Applications .....</b>               | <b>307</b> |
| Creating an Application for Note and Password Storage and Encryption .....      | 307        |
| Setting Up the Data Model .....   | 309        |
| Setting Up the Tab Bar Controller .....   | 310        |
| Adding the Tab .....  | 311        |
| Incorporating NSFetchedResultsController into MyStash .....                     | 316        |
| Creating the Interface for Adding and Editing Notes and Passwords .....         | 322        |
| Splitting Data Across Multiple Persistent Stores .....                          | 335        |
| Using Model Configurations .....  | 336        |
| Adding Encryption .....   | 340        |
| Persistent Store Encryption Using Data Protection .....                         | 340        |

|  |                   |
|--|-------------------|
| Data Encryption.....                         | 342               |
| Sending Notifications When Data Changes..... | 347               |
| Registering an Observer .....                | 348               |
| Receiving the Notifications .....            | 349               |
| Seeding Data.....                            | 349               |
| Adding Categories to Passwords .....         | 350               |
| Creating a New Version of Seeded Data .....  | 353               |
| Error Handling .....                         | 353               |
| Handling Core Data Operational Errors .....  | 355               |
| Handling Validation Errors .....             | 357               |
| Handling Validation Errors in MyStash.....   | 360               |
| Summary .....                                | 365               |
| <b>Index .....</b>                           | <b><u>367</u></b> |

---

# About the Authors



**Michael Privat** is the President and CEO of Majorspot, Inc., developer of the following iPhone and iPad apps:

- Ghostwriter Notes
- My Spending
- iBudget
- Chess Puzzle Challenge

He is also an expert developer and technical lead for Availity, LLC, based in Jacksonville, Florida. He earned his Master's degree in Computer Science from the University of Nice in Nice, France. He moved to the United States to develop software in artificial intelligence at the Massachusetts Institute of Technology. Coauthor of *Beginning OS X Lion Apps Development* (Apress, 2011), he now lives in Jacksonville, Florida, with his wife, Kelly, and their two children.



**Rob Warner** is a senior technical staff member for Availity, LLC, based in Jacksonville, Florida, where he works with various teams and technologies to deliver solutions in the health care sector. He coauthored *Beginning OS X Lion Apps Development* (Apress, 2011) and *The Definitive Guide to SWT and JFace* (Apress, 2004), and he blogs at [www.grailbox.com](http://www.grailbox.com). He earned his Bachelor's degree in English from Brigham Young University in Provo, Utah. He lives in Jacksonville, Florida, with his wife, Sherry, and their five children.

---

# About the Technical Reviewer



**Robert Hamilton** is a seasoned information technology director for Blue Cross Blue Shield of Florida. He is experienced in developing applications for the iPhone and iPad; his most recent project was Ghostwriter Notes.

Before entering his leadership role at BCBSF, Robert excelled as an application developer, having envisioned and created the first claims status application used by their providers through Availity.

A native of Atlantic Beach, Florida, Robert received his B.S. in Information Systems from the University of North Florida. He supports The First Tee of Jacksonville and the Cystic Fibrosis Foundation. He is the proud father of two daughters.



---

# Acknowledgments

There is no telling how many books never had a chance to be written because the potential authors had other family obligations to fulfill. I thank my wife, Kelly, and my children, Matthieu and Chlo  , for allowing me to focus my time on this book for a few months and accomplish this challenge. Without their unconditional support and encouragement, I would not have been able to contribute to the creation of this book.

Working on this book with Rob Warner has been enlightening. I have learned a lot from him throughout this effort. His dedication to getting the job done correctly carried me when I was tired. His technical skills got me unstuck a few times when I was clueless. His gift for writing so elegantly and his patience have made my engineer jargon sound like nineteenth century prose.

I also thank the friendly and savvy Apress team who made the whole process work like a well-oiled machine. Jennifer Blackwell helped us through the entire project, guiding us through all the tasks that are required of authors. Douglas Pundick shared his editorial wisdom to keep this work readable, well organized and understandable; Steve Anglin, Matthew Moodie, Mary Behr, and the rest of the Apress folks were always around for us to lean on.

Robert Hamilton was once again a reliable watchdog to correct our technical mistakes. I'd also like to thank Brian Kohl for saving us from shaming ourselves at times with overly complicated code.

Finally, I thank the incredibly talented people of Availity who were supportive of this book from the very first day and make this company a great place to work at. Trent Gavazzi, Ben Van Maanen, Taryn Tresca, Herve Devos, and all the others offered friendship and encouragement. The last bit of thanks goes to Geoff Packwood for calling in regularly to check on the progress.

—Michael Privat

What a privilege it's been to write a second edition of *Pro Core Data for iOS*! I thank Apress for the opportunity, particularly Steve Anglin, Jennifer Blackwell, Douglas Pundick, Matthew Moodie, Mary Behr, and Robert Hamilton. It's good to get a second crack at an intriguing topic.

Thanks to everyone who read the first edition, provided feedback, posted reviews, e-mailed thanks and questions, and generally made us feel that all our efforts made a dent. We've tried to incorporate your feedback into this edition, and we welcome any praise, criticism, and questions.

I thank my wife, Sherry, and my children (Tyson, Jacob, Mallory, Camie, and Leila) for their support and encouragement. I promise to take some downtime now, at least for awhile.

Working with Michael both enlightened and humbled me. I learned so much, yet was reminded often of how much I have to learn. I thank Michael for his persistence and dedication.

Thanks also to my employer, Availity, for providing opportunities to keep my mind nimble and engaged. Naming names creates the dilemma of knowing where to stop, so I'll keep this purposely short: thanks to Trent for all the challenges, opportunities, and support. Thanks to Jon for letting me contribute to the Innovation Center. And thanks to Brian Kohl for the Code Jams! Finally, thanks to Mom, Dad, and my siblings and in-laws for asking, "How's the book coming?" and then listening to me describe all the details. Or at least pretending to.

---

# Introduction

Interest in developing apps for Apple's iOS platform continues to rise, and more great apps appear in Apple's App Store every day. As people like you join the app-creation party, they usually discover that their apps must store data on iOS devices to be useful. Enter *Pro Core Data for iOS*, written for developers who have learned the basics of iOS development and are ready to dive deeper into topics surrounding data storage to take their apps from pretty good to great. Core Data, Apple's technology for data storage and retrieval, is both easy to approach and difficult to master. This book spans the gamut, starting you with the simple and taking you through the advanced. Read each topic, understand what it means, and incorporate it into your own Core Data apps.

## Why a Second Edition?

Since the publication of the first edition of *Pro Core Data for iOS*, Apple has released Xcode 4, a major overhaul of their programming tool. Everything has moved or changed somehow, so the descriptions and tutorials from the first edition of this book, which used Xcode 3, no longer apply. All the descriptions and screenshots have been updated to the new interface.

We didn't stop at updating the book for Xcode 4, however. We broke the discussion of `NSFetchedResultsController` into its own chapter, giving it more treatment and coverage. We dug deeper into the tricky topic of migrations. We took a new approach to the section on data encryption, based on feedback from Brian Kohl. We responded to feedback we've received via reviews and e-mail. We think both new readers and people who have already read the first edition will profit from reading this edition.

## What You'll Need

To follow along with this book, you need an Intel Mac running Snow Leopard or Lion, and you need Xcode 4, which is available from the Mac App Store or from [developer.apple.com](http://developer.apple.com) for registered Apple developers. You'll also do better if you have at least a basic understanding of Objective-C, Cocoa Touch, and iOS development.

## What You'll Find

This book starts by setting a clear foundation for what Core Data is and how it works, and then it takes you step-by-step through how to get the results you need from this powerful framework. You'll learn about the components of Core Data and how they interact, how to design your data model, how to filter your results, how to tune performance, how to migrate your data across data model versions, and many other topics around and between these that will separate your apps from the crowd.

This book combines theory and code to teach its subject matter. Although you can take the book to your Barcalounger and read it from cover to cover, you'll find the book is more effective if you're in front of a computer, typing in and understanding the code it explains. We also hope that, after you read the book and work through its code, you'll keep it handy as a reference, turning to it often for answers and clarification.

## How This Book Is Organized

We've tried to arrange the material so that it builds from beginning topics to advanced, at least in a general sense, as the book progresses. The topics tend to build on each other, so you'll likely benefit most by working through the book front to back, rather than skipping around. If you're looking for guidance on a specific topic—versioning and migrating data, say, or tuning performance and memory usage—skip ahead to that chapter. Most chapters focus on a single topic, indicated by that chapter's title. The final chapter covers an array of advanced topics that don't fit neatly anywhere else.

## Source Code and Errata

You can and should download the source code for this book from the Apress web site at [www.apress.com](http://www.apress.com). Feel free to use it in your own projects, whether personal or commercial. We'll post any corrections to code as they're uncovered. We'll also post book corrections in the errata section.

## How to Contact Us

We'd love to hear from you, whether it's questions, concerns, better ways of doing things, or triumphant announcements of your Core Data apps landing on the App Store. You can find us here:

Michael Privat  
E-mail: [mprivat@mac.com](mailto:mprivat@mac.com)  
Twitter: [@michaelprivat](https://twitter.com/michaelprivat)  
Blog: <http://michaelprivat.com>

Rob Warner  
E-mail: [rwarnier@grailbox.com](mailto:rwarnier@grailbox.com)  
Twitter: [@hoop33](https://twitter.com/hoop33)  
Blog: <http://grailbox.com>

## Getting Started

If you misread this book's title, thought it discussed and deciphered core dumps, and hope it will help you debug a nasty application crash, you got the wrong book. Get a debugger, memory tools, and an appointment with the optometrist. Otherwise, you bought, borrowed, burglarized, or acquired this book somehow because you want to better understand and implement Core Data in your iOS applications. You got the right book.

You might read these words from a paper book, stout and sturdy and smelling faintly of binding glue. You might digitally flip through these pages on a nook, iPad, Kindle, Sony Reader, Kobo eReader, or some other electronic book reader. You might stare at a computer screen, whether on laptop, netbook, or monitor, reading a few words at a time while telling yourself to ignore your Twitter feed rolling CNN-like along the screen's edge. As you read, you know that not only can you stop at any time but that you can resume at any time. Any time you want to read this book, you can pick it up. If you marked the spot where you were last reading, you can even start from where you last stopped. We take this for granted with books.

Users take it for granted with applications.

Users expect to find their data each time they launch their applications. Apple's Core Data framework helps you ensure that they will. This chapter introduces you to Core Data, explaining what it is, how it came to be, and how to build simple Core Data-based applications for iOS. This book walks through the simplicity and complexities of Core Data. Use the information in the book to create applications that store and retrieve data reliably and efficiently so that users can depend on their data. Code carefully, though—you don't want to write buggy code and have to deal with nasty application crashes.

### What Is Core Data?

When people use computers, they expect to preserve any progress they make toward completing their tasks. Saving progress, essential to office software, code editors, and games involving small plumbers, is what programmers call *persistence*. Most software requires persistence, or the ability to store and retrieve data, so that users don't have to

reenter all their data each time they use their applications. Some software can survive without any data storage or retrieval; calculators, carpenter's levels, and apps that make annoying or obscene sounds spring to mind. Most useful applications, however, preserve some state, whether configuration-oriented data, progress toward achieving some goal, or mounds of related data that users create and care about. Understanding how to persist data to iDevices is critical to most useful iOS development.

Apple's Core Data provides a versatile persistence framework. Core Data isn't the only data storage option, nor is it necessarily the best option in all scenarios, but it fits well with the rest of the Cocoa Touch development framework and maps well to objects. Core Data hides most of the complexities of data storage and allows you to focus on what makes your application fun, unique, or usable.

Although Core Data can store data in a relational database (such as SQLite), it is not a database engine. It doesn't even have to use a relational database to store its data. Though Core Data provides an entity-relationship diagramming tool, it is not a data modeler. It isn't a data access layer like Hibernate, though it provides much of the same object-relational mapping functionality. Instead, Core Data wraps the best of all these tools into a data management framework that allows you to work with entities, attributes, and relationships in a way that resembles the object graphs you're used to working with in normal object-oriented programming.

Early iPhone programmers didn't have the power of the Core Data framework to store and retrieve data. The next section shows you the history behind persistence in iOS.

## History of Persistence in iOS

Core Data evolved from a NeXT technology called Enterprise Objects Framework (EOF) by way of WebObjects, another NeXT technology that still powers parts of Apple's web site. It debuted in 2005 as part of Mac OS X 10.4 ("Tiger"), but didn't appear on iPhones until version 3.0 of the SDK, released in June 2009. Before Core Data, iPhone developers had the following options in terms of persistence:

- Use property lists, which contain nested lists of key/value pairs of various data types.
- Serialize objects to files using the SDK's NSCoding protocol.
- Take advantage of the iPhone's support for the relational database SQLite.
- Persist data to the Internet cloud.

Developers used all these mechanisms for data storage as they built the first wave of applications that flooded Apple's App Store. Each one of these storage options remains viable, and developers continue to employ them as they build newer applications using newer SDK versions.

None of these options, however, compares favorably to the power, ease of use, and Cocoa-fitness of Core Data. Despite the invention of frameworks like FMDatabase or

ActiveRecord to make dealing with persistence on iOS easier in the pre-Core Data days, developers gratefully leapt to Core Data when it became available.

Although Core Data might not solve all persistence problems best and you might solve some of your persistence scenarios using other means like the options listed earlier, you'll turn to Core Data more often than not. As you work through this book and learn the problems that Core Data solves and how elegantly it solves them, you'll likely use Core Data any time you can. As new persistence opportunities arise, you won't ask yourself, "Should I use Core Data for this?" but rather, "Is there any reason *not* to use Core Data?"

The next section shows you how to build a basic Core Data application using Xcode's project templates. Even if you've already generated an Xcode Core Data project and know all the buttons and check boxes to click, don't skip the next section. It explains the Core Data-related sections of code that the templates generate and forms a base of understanding on which the rest of the book builds.

## Creating a Basic Core Data Application

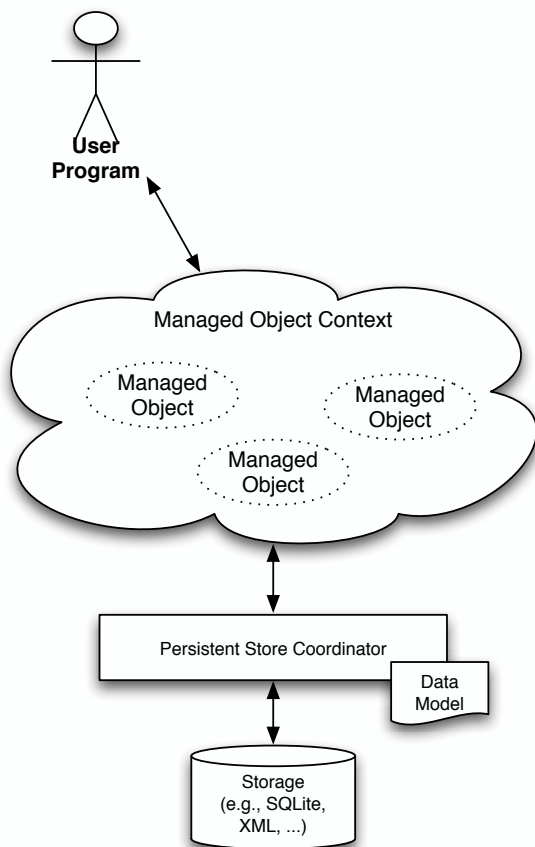
The many facets, classes, and nuances of Core Data merit artful analysis and deep discussions to teach you all you need to know to gain mastery of Core Data's complexities. Building a practical foundation to support the theory, however, is just as essential to mastery. This section builds a simple Core Data-based application using one of Xcode's built-in templates and then dissects the most important parts of its Core Data-related code to show what they do and how they interact. At the end of this section, you will understand how this application interacts with Core Data to store and retrieve data.

## Understanding the Core Data Components

Before building this section's basic Core Data application, you should have a high-level understanding of the components of Core Data. Figure 1–1 illustrates the key elements of the application you will build in this section. Review this figure for a bird's-eye view of what this application accomplishes, where all its pieces fit, and why you need them.

As a user of Core Data, you should never interact directly with the underlying persistent store. One of the fundamental principles of Core Data is that the persistent store should be abstracted from the user. A key advantage of that is the ability to seamlessly change the backing store in the future without having to modify the rest of your code. You should try to picture Core Data as a framework that manages the persistence of objects rather than thinking about databases. Not surprisingly, the objects managed by the framework must extend `NSObject` and are typically referred to as, well, managed objects. Don't think, though, that the lack of imagination in the naming conventions for the components of Core Data reveals an unimaginative or mundane framework. In fact, Core Data does an excellent job at keeping all the object graph interdependencies, optimizations, and caching in a predictable state so that you don't have to worry about

it. If you have ever tried to build your own object management framework, you understand all the intricacies of the problem Core Data solves for you.

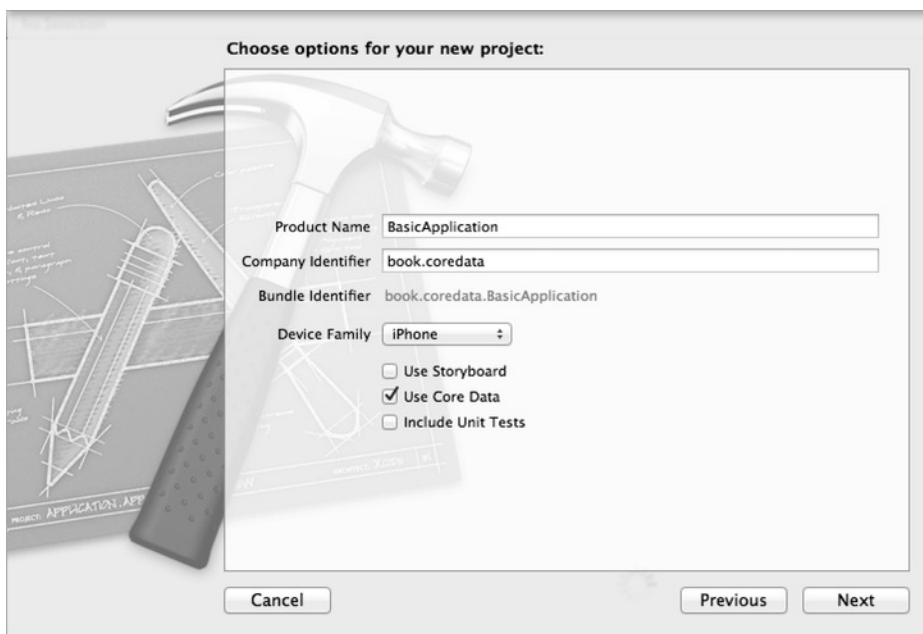


**Figure 1–1.** Overview of Core Data's components

Much like we need a livable environment to subsist, managed objects must live within an environment that's livable for them, usually referred to as a *managed object context*, or simply *context*. The context keeps track of the states of not only the object you are altering but also all the objects that depend on it or that it depends on. The `NSManagedObjectContext` object in your application provides the context and is the key property that your code must always be able to access. You typically accomplish exposing your `NSManagedObjectContext` object to your application by having your application delegate initialize it and expose it as one of its properties. Your application context will often give the `NSManagedObjectContext` object to the main view controller as well. Without the context, you will not be able to interact with Core Data.

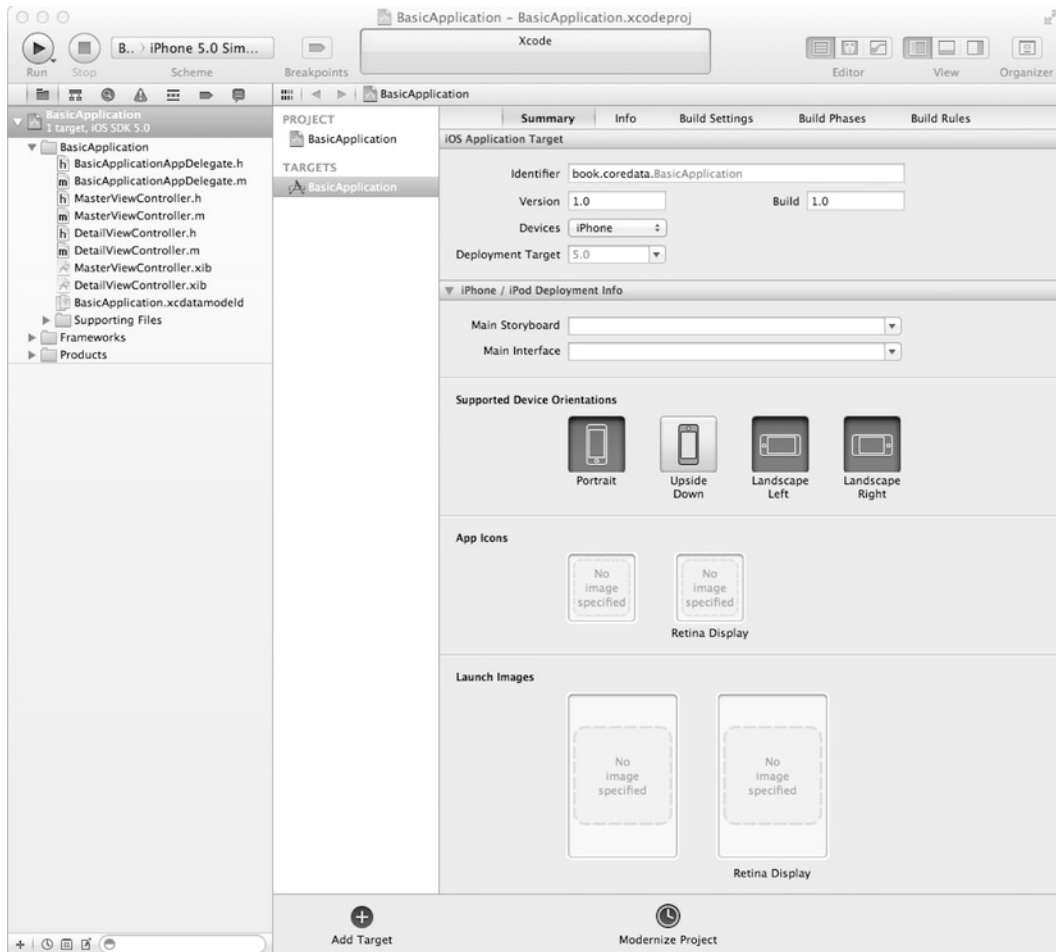
## Creating a New Project

To begin, launch Xcode, and create a new project by selecting **File > New > New Project** from the menu. Note that you can also create a new project by pressing  $\hat{+}+\mathbb{A}+N$ . From the list of application templates, select the Application item under iOS on the left, and pick Master-Detail Application on the right. Click Next, and on the next screen type **BasicApplication** in the Product Name field, **book.coredata** in the Company Identifier field, uncheck Use Storyboard and check Use Core Data. See Figure 1–2. Click the Next button, choose the parent directory where Xcode will create the BasicApplication directory and project, and click Create. Xcode creates your project, generates the project's files, and opens its IDE window with all the files it generated, as Figure 1–3 shows.



**Figure 1–2.** Creating a new project with Core Data





**Figure 1–3.** Xcode showing your new project

## Running Your New Project

Before digging into the code, run it to see what it does. Launch the application by clicking the Run button. The iPhone Simulator opens, and the application presents the navigation-based interface shown in Figure 1–4, with a table view occupying the bulk of the screen, an Edit button in the top-left corner, and the conventional Add button, denoted by a plus sign, in the upper-right corner. The application's table shows an empty list indicating that the application isn't aware of any events, which is what the generated Xcode Core Data project stores. Create a new event stamped with the current time by clicking the plus button in the top-right corner of the application.



**Figure 1–4.** *The basic application with a blank screen*

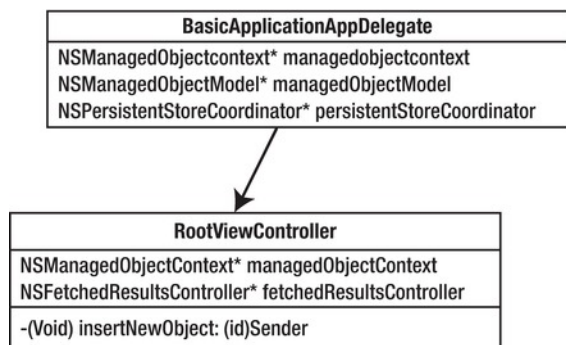
Now, stop the application by clicking the Stop button in the Xcode IDE. If the application hadn't used Core Data persistence, it would have lost the event you just created as it exited. Maintaining a list of events with this application and no persistence would be a Sisyphean task—you'd have to re-create the events each time you launched the application. Because the application uses persistence, however, it stored the event you created using the Core Data framework. Relaunching the application shows that the event is still there, as Figure 1–5 demonstrates.



**Figure 1–5.** *The basic application with a persisted event*

## Understanding the Application's Components

The anatomy of the application is relatively simple. It has a data model that describes the entities in the data store, a view controller that facilitates interactions between the view and the data store, and an application delegate that helps initialize and launch the application. Figure 1–6 shows the classes involved and how they relate to each other.



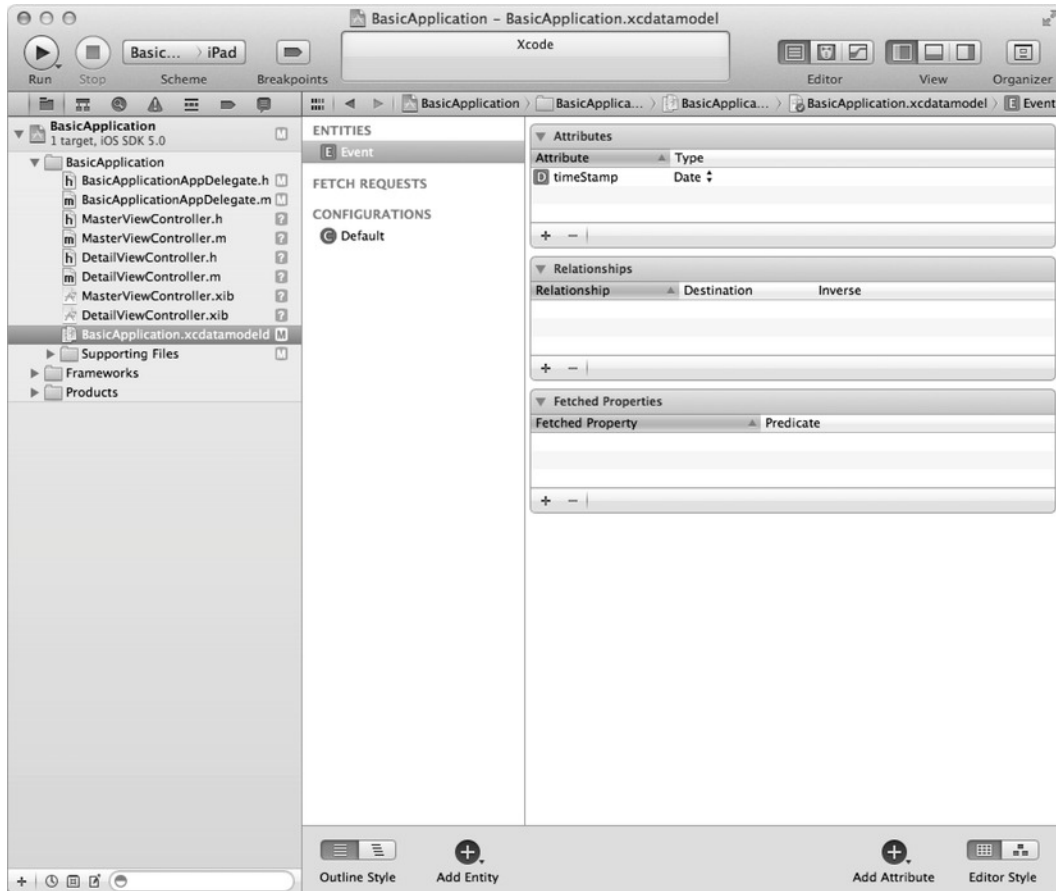
**Figure 1–6.** *Classes involved in the BasicApplication example*

Note how the MasterViewController class, which is in charge of managing the user interface, has a handle to the managed object context so that it can interact with Core Data. As you go through the code, you'll see that the MasterViewController class obtains the managed object context from the application delegate. This happens in the controller's `initWithNibName:bundle:` method, shown here:

```

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Master", @"Master");
        id delegate = [[UIApplication sharedApplication] delegate];
        self.managedObjectContext = [delegate managedObjectContext];
    }
    return self;
}
  
```

The entry called `BasicApplication.xcdatamodeld`, which is actually a directory on the file system, contains the data model, `BasicApplication.xcdatamodel`. The data model is central to every Core Data application. This particular data model defines only one entity, named `Event`, for the application. Events are defined as entities that contain only one attribute named `timeStamp` of type `Date`, as shown in Figure 1–7.



**Figure 1–7.** *The Xcode-generated data model*

The Event entity is of type `NSManagedObject`, which is the basic type for all entities managed by Core Data. Chapter 2 explains the `NSManagedObject` type in more detail.

## Fetching Results

The next class of interest is the `MasterViewController`. Opening its header file (`MasterViewController.h`) reveals two properties:

```
@property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;
```

These properties are defined using the same syntax as the definitions of any Objective-C class properties. The `NSFetchedResultsController` is a type of controller provided by the Core Data framework that helps manage results from queries. `NSManagedObjectContext` is a handle to the application's persistent store that provides a context, or environment, in which the managed objects can exist.

The implementation of the `MasterViewController`, found in `MasterViewController.m`, shows how to interact with the Core Data framework to store and retrieve data. The `MasterViewController` implementation provides an explicit getter for the `fetchResultsController` property that preconfigures it to fetch data from the data store.

The first step in creating the fetch controller consists of creating a request that will retrieve Event entities, as shown in this code from the `fetchResultsController` accessor:

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
    inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];
```

The result of the request can be ordered using the sort descriptor from the Cocoa Foundation framework. The sort descriptor defines the field to use for sorting and whether the sort is ascending or descending. In this case, you sort by descending chronological order, like so:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:
@"timeStamp" ascending:NO];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
[fetchRequest setSortDescriptors:sortDescriptors];
```

Once you define the request, you can use it to construct the fetch controller. Because the `MasterViewController` implements `NSFetchedResultsControllerDelegate`, it can be set as the `NSFetchedResultsController`'s delegate so that it is automatically notified as the result set changes and so that it updates its view appropriately. You could get the same results by invoking the `executeFetchRequest` of the managed object context, but you would not benefit from the other advantages that come from using the `NSFetchedResultsController` such as the seamless integration with the `UITableView`, as you'll see later in this section and in Chapter 9. Here is the code that constructs the fetch controller:

```
NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
    alloc] initWithFetchRequest:fetchRequest managedObjectContext:
self.managedObjectContext sectionNameKeyPath:nil cacheName:@"Master"];
aFetchedResultsController.delegate = self;
self.fetchResultsController = aFetchedResultsController;
```

**NOTE:** You may have noticed that the `initWithFetchRequest` shown earlier uses a parameter called `cacheName`. You could pass `nil` for the `cacheName` parameter to prevent caching, but naming a cache indicates to Core Data to check for a cache with a name matching the passed name and see whether it already contains the same fetch request definition. If it does find a match, it will reuse the cached results. If it finds a cache entry by that name but the request doesn't match, then it is deleted. If it doesn't find it at all, then the request is executed, and the cache entry is created for the next time. This is obviously an optimization that aims to prevent executing the same request over and over. Core Data manages its caches intelligently so that if the results are updated by another call, the cache is removed if affected.

Finally, you tell the controller to execute its query to start retrieving results. To do this, use the `performFetch` method.

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

You can see the entire getter method for `fetchedResultsController` in Listing 1–1.

**Listing 1–1. The Entire Getter Method for `fetchedResultsController`**

```
- (NSFetchedResultsController *)fetchedResultsController
{
    if (__fetchedResultsController != nil)
    {
        return __fetchedResultsController;
    }

    /*
     Set up the fetched results controller.
     */
    // Create the fetch request for the entity.
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    // Edit the entity name as appropriate.
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
    inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];

    // Set the batch size to a suitable number.
    [fetchRequest setFetchBatchSize:20];

    // Edit the sort key as appropriate.
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"timeStamp"
    ascending:NO];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];

    [fetchRequest setSortDescriptors:sortDescriptors];

    // Edit the section name key path and cache name if appropriate.
    // nil for section name key path means "no sections".
    NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
    alloc] initWithFetchRequest:fetchRequest managedObjectContext:self.managedObjectContext
    sectionNameKeyPath:nil cacheName:@"Master"];
    aFetchedResultsController.delegate = self;
    self.fetchedResultsController = aFetchedResultsController;

    NSError *error = nil;
    if (![self.fetchedResultsController performFetch:&error])
    {
        /*
         Replace this implementation with code to handle the error appropriately.

         abort() causes the application to generate a crash log and terminate. You should
         not use this function in a shipping application, although it may be useful during
         development. If it is not possible to recover from the error, display an alert panel
         that instructs the user to quit the application by pressing the Home button.
```

```

    */
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

return __fetchedResultsController;
}

```

NSFetchedResultsController behaves as a collection of managed objects, similar to an NSArray, which makes it easy to use. In fact, it exposes a read-only property called `fetchedObjects` that is of type NSArray to make things even easier to access the objects it fetches. The `MasterViewController` class, which also extends `UITableViewController`, demonstrates just how suited the `NSFetchedResultsController` is to manage the table's content.

## Inserting New Objects

A quick glance at the `insertNewObject:` method shows how new events (the managed objects) are created and added to the persistent store. Managed objects are defined by the entity description from the data model and can live only within a context. The first step is to get a hold of the current context as well as the entity definition. In this case, instead of explicitly naming the entity, you reuse the entity definitions that are attached to the fetched results controller:

```

NSManagedObjectContext *context = [self.fetchedResultsController managedObjectContext];
NSEntityDescription *entity = [[self.fetchedResultsController fetchRequest] entity];

```

Now that you've gathered all the elements needed to bring the new managed object to existence, you create the Event object and set its `timeStamp` value.

```

NSManagedObject *newManagedObject = [NSEntityDescription
    insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];
[newManagedObject setValue:[NSDate date] forKey:@"timeStamp"];

```

The last step of the process is to tell Core Data to save changes to its context. The obvious change is the object you just created, but keep in mind that calling the `save` method will also affect any other unsaved changes to the context.

```

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

```

The complete method for inserting the new Event object is shown in Listing 1–2.

### **Listing 1–2.** *The Complete Method for Inserting the New Event Object*

```

- (void)insertNewObject {
    // Create a new instance of the entity managed by the fetched results controller.
    NSManagedObjectContext *context = [self.fetchedResultsController
    managedObjectContext];
    NSEntityDescription *entity = [[self.fetchedResultsController fetchRequest] entity];
    NSManagedObject *newManagedObject = [NSEntityDescription
    insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];
}

```



```

// If appropriate, configure the new managed object.
// Normally you should use accessor methods, but using KVC here avoids the need to add
a custom class to the template.
[newManagedObject setValue:[NSDate date] forKey:@"timeStamp"];

// Save the context.
NSError *error = nil;
if (![context save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}
}
}

```

## Initializing the Managed Context

Obviously, none of this can happen without initializing the managed context first. This is the role of the application delegate. In a Core Data-enabled application, the delegate must expose three properties.

```

@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

```

Note that they are all marked as read-only, which prevents any other component in the application from setting them directly. A closer look at `BasicApplicationAppDelegate.m` shows that all three properties have explicit getter methods.

First, the managed object model is derived from the data model (`BasicApplication.xcdatamodel`) and loaded.

```

- (NSManagedObjectModel *)managedObjectModel
{
    if (__managedObjectModel != nil)
    {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"BasicApplication"
withExtension:@"momd"];
    __managedObjectModel = [[NSManagedObjectModel alloc]
initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}

```

Then a persistent store is created to support the model. In this case, as well as in most Core Data scenarios, the persistent store is backed by a SQLite database. The managed object model is a logical representation of the data store, while the persistent store is the materialization of that data store.

```

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

```

```

NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"BasicApplication.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;
}

```

Finally, the managed object context is created.

```

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

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return __managedObjectContext;
}

```

The context is used throughout the application as the single interface with the Core Data framework and the persistent store, as Figure 1–8 demonstrates.



**Figure 1–8.** Core Data initialization sequence

Lastly, everything is put in motion when the application delegate's `application:didFinishLaunchingWithOptions:` method is called.

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.

    MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Calling the getter for the delegate's `managedObjectContext` starts a chain reaction in which

- `(NSManagedObjectContext *)managedObjectContext`, calls  
- `(NSPersistentStoreCoordinator *)persistentStoreCoordinator` and then in turns calls  
- `(NSManagedObjectContext *)managedObjectContext`. The call to `managedObjectContext` therefore initializes the entire Core Data stack and readies Core Data for use.

If you followed along with Xcode on your machine, you have a basic Core Data–based application, generated from Xcode's templates, that you can run to create, store, and retrieve event data. What if, however, you have an existing application to which you want to add the power of Core Data? The next section demonstrates how to add Core Data to an existing iOS application.

## Adding Core Data to an Existing Project

Creating a new application and selecting the Use Core Data check box, as shown in the previous section, isn't always possible. Frequently, developers start an application, write a lot of code, and only realize later that they need Core Data in their application. We've known developers who have refused to admit that they should just add Core Data by hand to an existing application. Instead, fueled by a desire to prove that they could write their own better persistence layer (rather than try to understand how to use the framework), they embarked on convoluted programming that led to less than adequate results. They confused persistence with obstinacy. In the spirit of making the jump easier, this section explains the steps involved with retrofitting an application in order to make it aware of and use Core Data.

Enabling an application to leverage Core Data is a three-step process.

1. Add the Core Data framework.
2. Create a data model.
3. Initialize the managed object context.

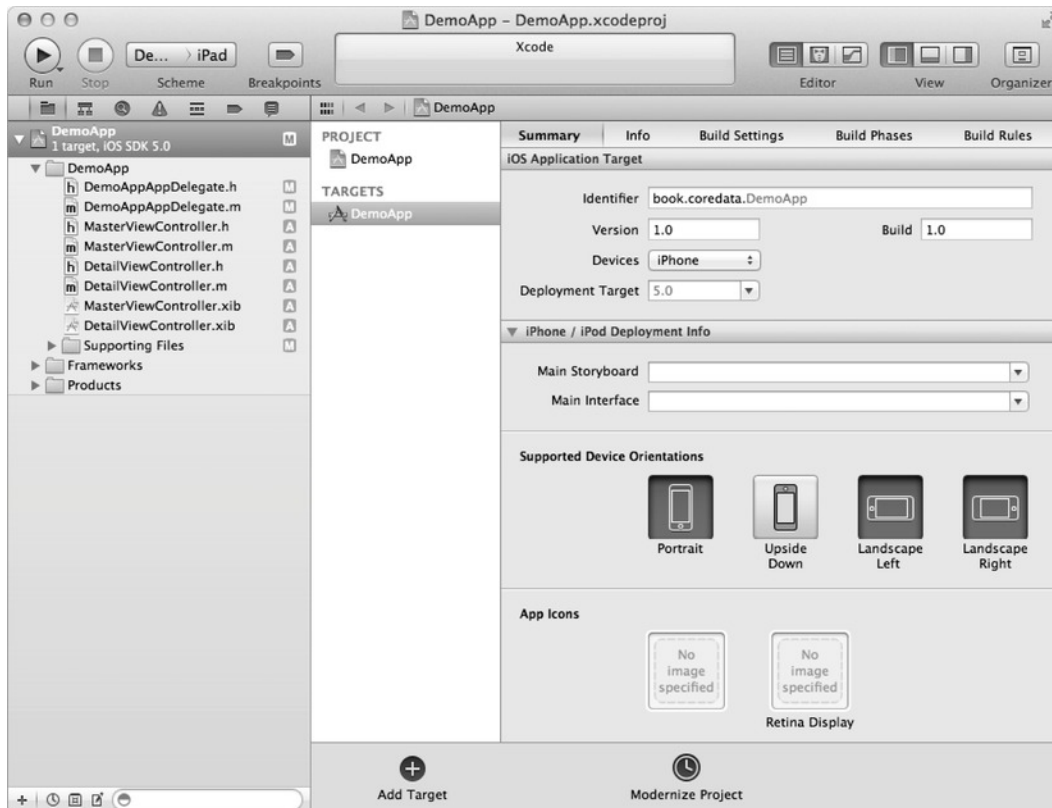
The next three sections walk you through these three steps so you can add Core Data support to any existing iOS application.

## Adding the Core Data Framework

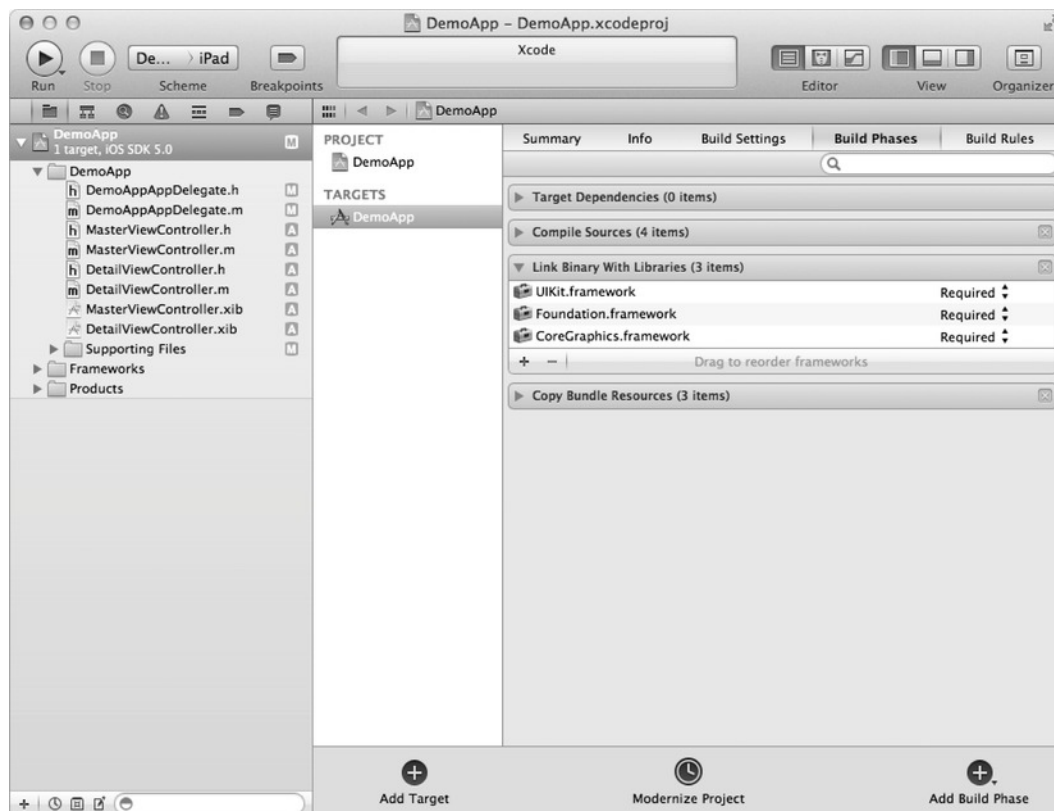
In the Objective-C world, libraries are referred to as *frameworks*. Expanding the Frameworks groups in the Xcode source tree shows that the project is aware of only a handful of frameworks. Typical iOS applications will at least have UIKit (the user interface framework for iOS), Foundation, and Core Graphics. The first step to add Core Data to an existing application consists of making the application aware of the Core Data framework by adding it to the project. To do this, perform the following steps:

1. Select the project on the left side of Xcode, which displays target settings on the right (see Figure 1–9).

2. Select the fourth tab, Build Phases, and open the section called Link Binary With Libraries (see Figure 1–10).
3. Click the + button below that section to see a list of frameworks you can add (see Figure 1–11).
4. Select CoreData.framework and click the Add button.
5. (Optional) Drag the CoreData.framework entry in the source list for your project to the Frameworks folder.



**Figure 1–9.** *Selecting the project to see target settings*



**Figure 1–10.** Viewing Link Binary With Libraries settings

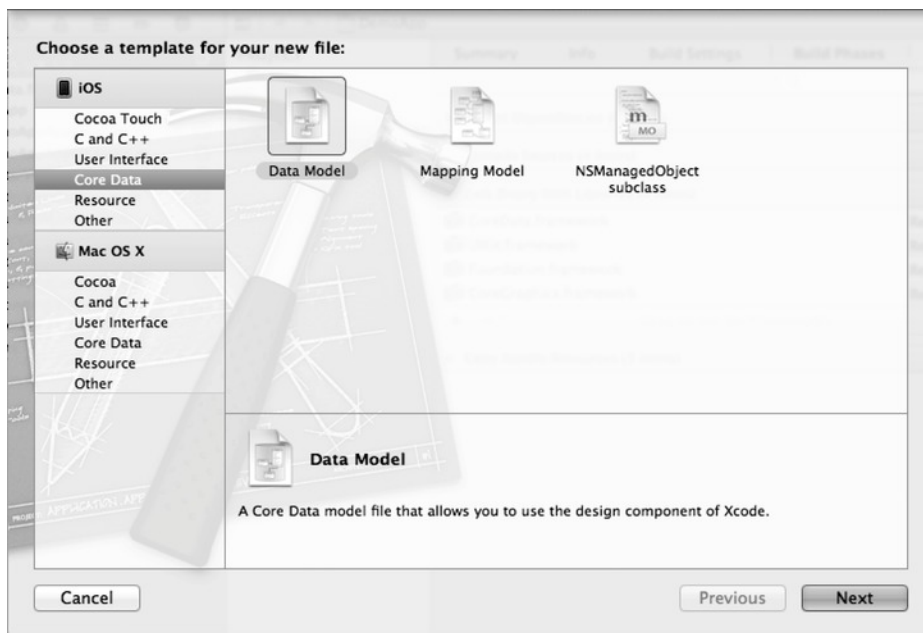


**Figure 1–11.** Selecting a Framework to link to

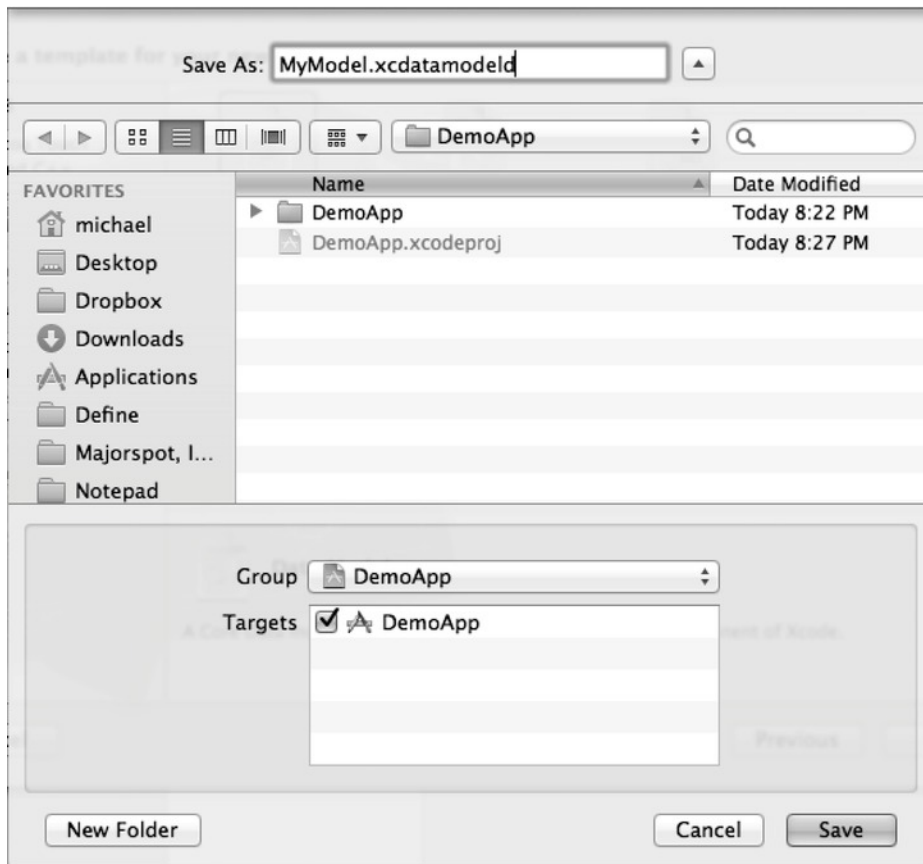
The CoreData.framework is now listed on the left side of Xcode. Now that the application is aware of the Core Data framework, the classes specific to that framework can be used without creating compilation errors.

## Creating the Data Model

No Core Data application is complete without a data model. The data model describes all the entities that will be managed by the framework. For the sake of simplicity, the model created in this section contains a single class with a single attribute. The data model can be created in Xcode by selecting **File > New > New File** in the menu and picking the type Data Model from the iOS Core Data templates, as shown in Figure 1–12. Click Next, name the data model **MyModel.xcdatamodeld**, as shown in Figure 1–13, and click Save. This generates your new data model and opens it in Xcode (see Figure 1–14).

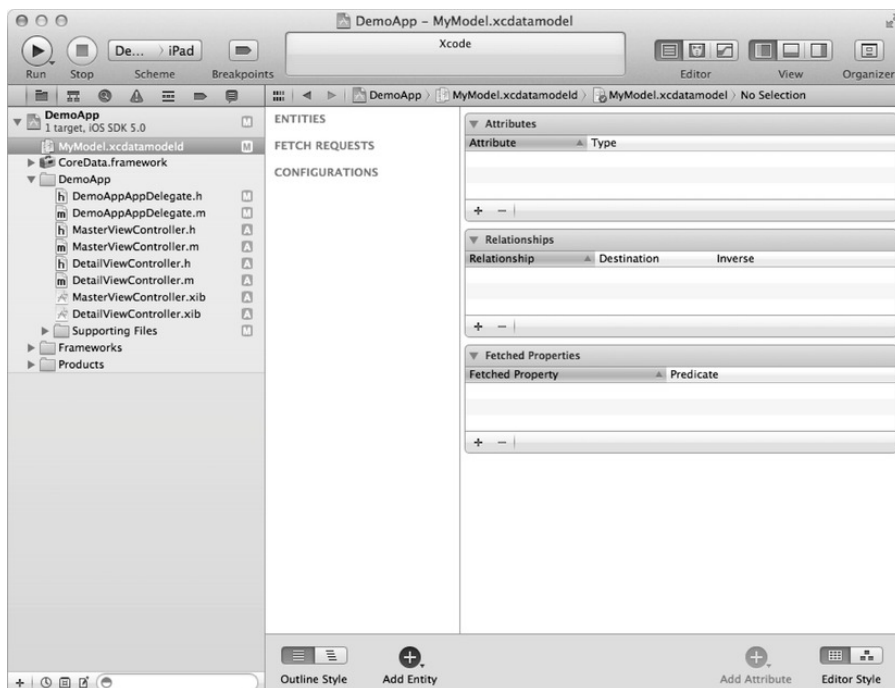


**Figure 1–12.** Adding a data model

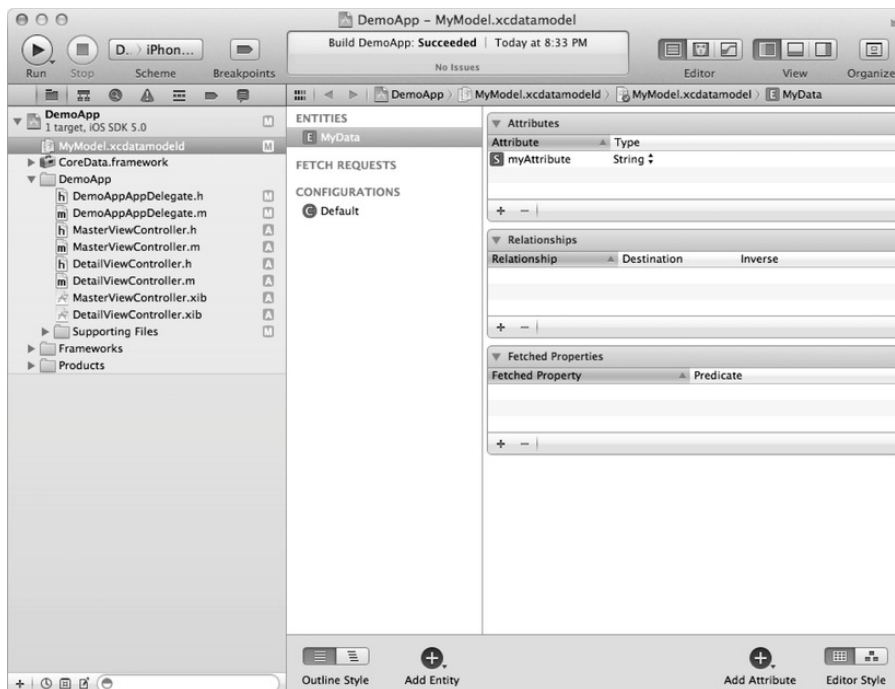


**Figure 1–13.** Naming your data model

Once the data model opens, you can add entities by clicking the Add Entity button at the bottom of the Xcode window. You can add properties to entities by selecting an entity and clicking and holding the Add Attribute button to the right of the Add Entity button, and selecting the type of property (attribute, relationship, or fetched property) from the drop-down menu. Xcode changes the label of this button to match the last property type you added. Simply clicking the button adds that type of property. For this example, create a single entity called `MyData` with a single attribute called `myAttribute` of type `String`, as Figure 1–15 shows.



**Figure 1-14.** Your new, empty data model



**Figure 1-15.** Adding an entity and attribute



## Initializing the Managed Object Context

The last step consists of initializing the managed object context, the persistent data store, and the object model. You can usually add the same code that Xcode would have generated for you if you had elected to use Core Data when you created your project: define these components as properties in the application delegate, declare a `saveContext` method, and declare a helper method for getting the application's document directory in which to store the Core Data database file. Your `DemoAppAppDelegate.h` file should look like Listing 1–3.

**Listing 1–3.** *The DemoAppAppDelegate.h file*

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

@interface DemoAppAppDelegate : UIResponder <UIApplicationDelegate> {

}

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) UINavigationController *navigationController;

@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```

Switch over to the implementation file (`DemoAppAppDelegate.m`). Start by adding `@synthesize` lines for your new properties, like this:

```
@synthesize managedObjectContext=__managedObjectContext;
@synthesize managedObjectModel=__managedObjectModel;
@synthesize persistentStoreCoordinator=__persistentStoreCoordinator;
```

The previous section showed that the context is created from a physical data store, which is in turn created from the data model. The initialization sequence remains the same and starts with loading the object model from the model you just defined.

```
- (NSManagedObjectModel *)managedObjectModel {
    if (__managedObjectModel != nil) {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"MyModel"
withExtension:@"momd"];
    __managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}
```

Now that you've loaded the object model, you can leverage it in order to create the persistent store handler. This example uses `NSSQLiteStoreType` in order to indicate that the storage mechanism should rely on a SQLite database, as shown here:

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

    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"DemoApp.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;
}
```

Notice that this code relies on the helper method `applicationDocumentsDirectory` to determine where to store the SQLite file; you'll define that method in a moment.

Next, initialize the context from the persistent store that you just defined.

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

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return __managedObjectContext;
}
```

To finish preparing Core Data for use in your application, you must implement the `applicationDocumentsDirectory` method and the `saveContext` method. Again, you can clone what Xcode generates, like this:

```
- (NSURL *)applicationDocumentsDirectory {
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
        inDomains:NSUserDomainMask] lastObject];
}

- (void)saveContext {
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil) {
        if ([managedObjectContext hasChanges] && ![managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
}
```

```
}
}
```

With Core Data in place, the application can now use the managed object context to store and retrieve entities. Let's use a simple example in which the application persists and displays the number of times it was launched in order to illustrate this process.

In the application delegate implementation file, `DemoAppAppDelegate.m`, edit the `didFinishLaunchingWithOptions:` method, and add code to retrieve the previous launches and add a new launch event.

**NOTE:** Although the managed object context, persistent store coordinator, and managed object model are typically members of the application delegate, code that stores and retrieves data usually goes in controllers corresponding to views of that data. Adding the code to store and retrieve entries directly to the application delegate is only done here for convenience and simplicity. In a real application, this kind of code would most likely belong to a controller.

The code to retrieve the previous launches grabs the context and executes a request to fetch entities of type `MyData`.

```
NSManagedObjectContext *context = [self managedObjectContext];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"MyData"
    inManagedObjectContext:context];
[request setEntity:entity];
NSArray *results = [context executeFetchRequest:request error:nil];
```

You can then iterate through the array of results in order to display the previous launches.

```
for (NSManagedObject *object in results) {
    NSLog(@"Found %@", [object valueForKey:@"myAttribute"]);
}
```

**NOTE:** One way to interact with the managed object's properties is to use the key/value pair generic accessor methods. `[object valueForKey:@"myAttribute"]` will retrieve the value of `myAttribute`, while `[object setValue:@"theValue" forKey:@"myAttribute"]` will set the value of `myAttribute`.

Lastly, add a new entry to the context before letting the application continue its normal execution, using the `saveContext:` method implemented earlier to save the new entry to the persistent store.

```
NSString *launchTitle = [NSString stringWithFormat:@"launch %d", [results count]];
NSManagedObject *object = [NSEntityDescription insertNewObjectForEntityForName:[entity
    name] inManagedObjectContext:context];
[object setValue:launchTitle forKey:@"myAttribute"];
[self saveContext];
NSLog(@"Added: %@", launchTitle);
```

Launching the application for the first time yields this output:

```
2011-02-25 05:13:23.783 DemoApp[2299:207] Added: launch 0
```

And launching it a second time displays the previous launch:

```
2011-02-25 05:15:48.883 DemoApp[2372:207] Found launch 0
```

```
2011-02-25 05:15:48.889 DemoApp[2372:207] Added: launch 1
```

The complete method from the application delegate implementation file is shown in Listing 1–4.

**Listing 1–4.** *The Complete Method from the Application Delegate Implementation File*

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Fetch the launches from Core Data
    NSManagedObjectContext *context = [self managedObjectContext];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"MyData"
inManagedObjectContext:context];
    [request setEntity:entity];
    NSArray *results = [context executeFetchRequest:request error:nil];

    // Iterate through the results and log them
    for (NSManagedObject *object in results) {
        NSLog(@"Found %@", [object valueForKey:@"myAttribute"]);
    }

    // Add a new entry for this launch
    NSString *launchTitle = [NSString stringWithFormat:@"launch %d", [results count]];
    NSManagedObject *object = [NSEntityDescription insertNewObjectForEntityForName:[entity
name] inManagedObjectContext:context];
    [object setValue:launchTitle forKey:@"myAttribute"];
    [self saveContext];
    NSLog(@"Added: %@", launchTitle);

    [self.window makeKeyAndVisible];
    return YES;
}
```

The existing application that used to be oblivious to Core Data has been outfitted with the powerful data storage management framework with a minimum amount of work. Follow these steps to add the power of Core Data to any of your existing applications.

## Summary

Whether building an iOS application from scratch or adding persistence to an existing iOS application, you should strongly consider turning to Apple's Core Data framework. Using Xcode's templates and code generation gives you a jump start down the Core Data path, or you can simply add Core Data by hand. Either way, you have in Core Data a persistence layer that abstracts most of the complexity of data storage and retrieval, and allows you to work with data reliably as an object graph.

This chapter gave you an overview of Core Data's classes and how they work together from the context to the managed objects to the persistent store and its coordinator. You caught a glimpse of fetch results controllers and how they work, but you saw only simple examples. The next chapter dives deeper into the Core Data framework, laying bare the classes and their interworkings so you know precisely how to use Core Data to persist your users' data.

# Understanding Core Data

Many developers, upon first seeing Core Data, deem Core Data and its classes a tangled mess of classes that impede, rather than enhance, data access. Perhaps they're Rails developers, used to making up method names to create dynamic finders and letting convention over configuration take care of the dirty work of data access. Maybe they're Java developers who have been annotating their Enterprise JavaBeans (EJBs) or hibernating their Plain Old Java Objects (POJOs). Whatever their backgrounds, many developers don't take naturally to the Core Data framework or its way of dealing with data, just as many developers squirm when first presented with Interface Builder and the live objects it creates when building user interfaces. We counsel you with patience and an open mind and assure you that the Core Data framework is no Rube Goldberg. The classes in the framework work together like Larry Bird's 1980s Boston Celtics in the half-court set, and when you understand them, you see their beauty and precision.

This chapter explains the classes in the Core Data framework, both individually and how they work together. Take the time to read about each class, to trace how they work together, and to type in and understand the examples.

## Core Data Framework Classes

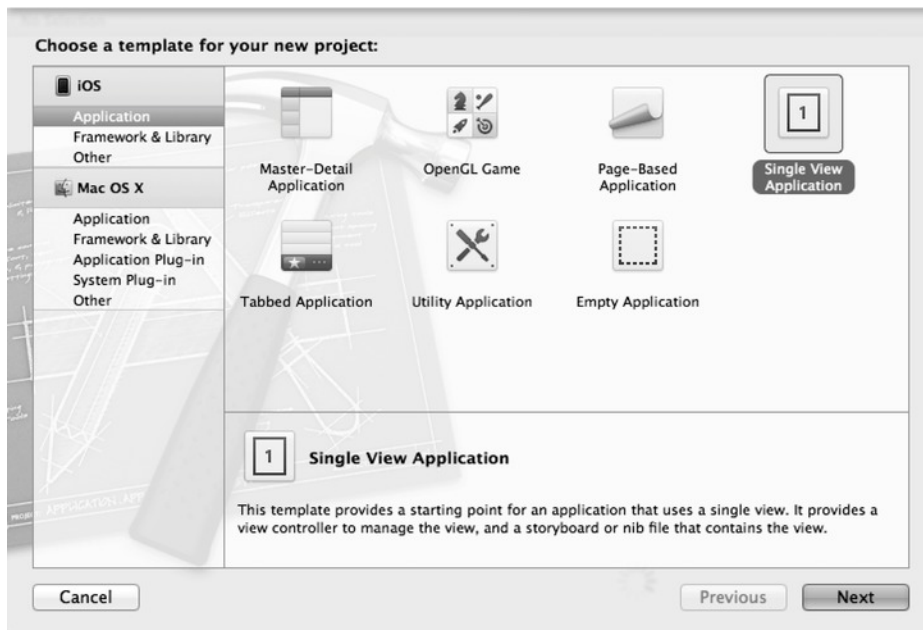
Chapter 1 guided you through the simple steps to get an application outfitted with Core Data. You saw bits of code, learned which classes to use, and discovered which methods and parameters to call and pass to make Core Data work. You faithfully, and perhaps somewhat blindly, followed Chapter 1's advice while perhaps wondering what was behind all the code, classes, methods, and parameters. Most developers reading the previous chapter probably wondered what would happen if they substituted a parameter value for another. A few of them probably even tried it. A small percentage of those who tried got something other than an explosion, and a percentage of them actually got what they thought they would get.

Edward Dijkstra, renowned computer scientist and recipient of the 1972 Turing Award for his work developing programming languages, spoke of elegance as a quality that decides between success and failure instead of being a dispensable luxury. Core Data not only solves the problem of object persistence but solves it elegantly. To achieve

elegance in your code, you should understand Core Data and not just guess at how it works. After reading this chapter, you will understand in detail not only the structure of the Core Data framework but also that the framework solves a complicated problem with only a small set of classes, making the solution simple, clear, and elegant.

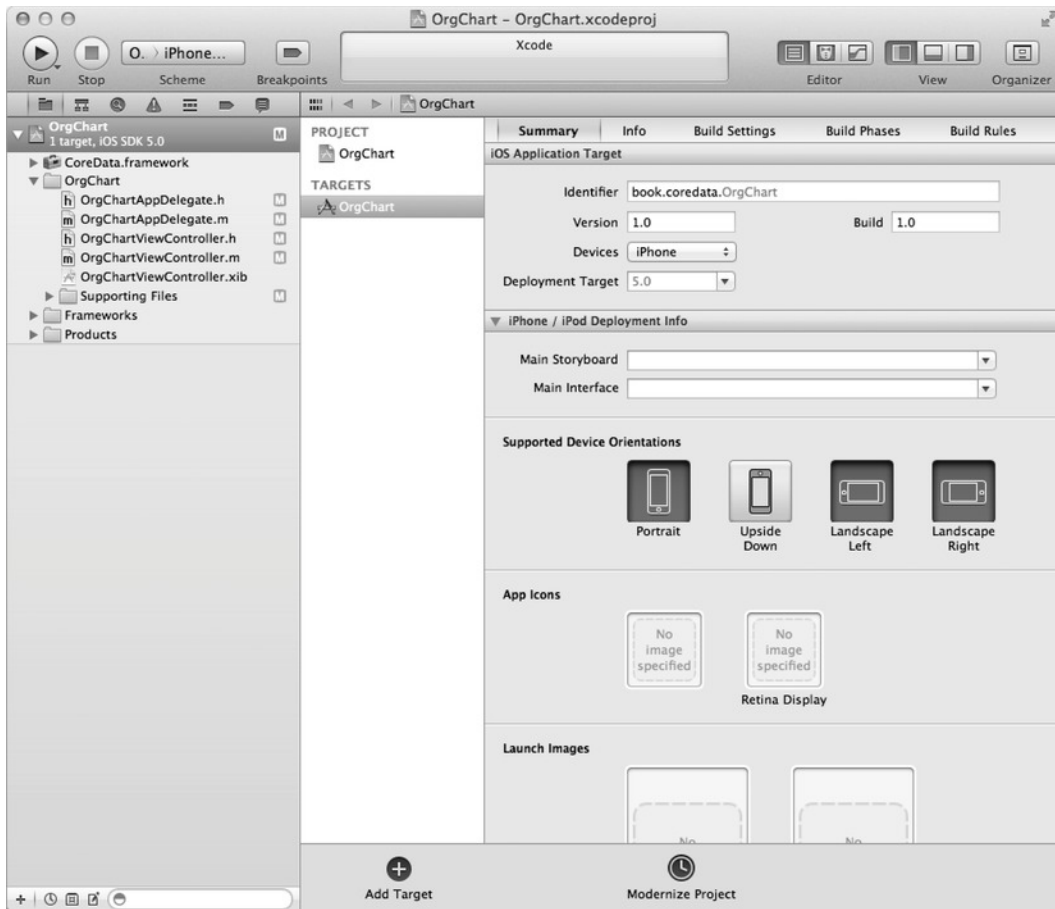
Throughout the chapter, you will build the class diagram that shows the classes involved in the framework and how they interact. You will also see that some classes belong to the Core Data framework and others are imported from other Cocoa frameworks such as the Foundation framework. In parallel with building the class diagram, you will build a small application in Xcode that deals with a fictitious company's organizational chart.

To follow along with this chapter, set up a blank iPhone application using the Single View Application template, as Figure 2–1 shows, and call it **OrgChart**.



**Figure 2–1.** Create a new application in Xcode.

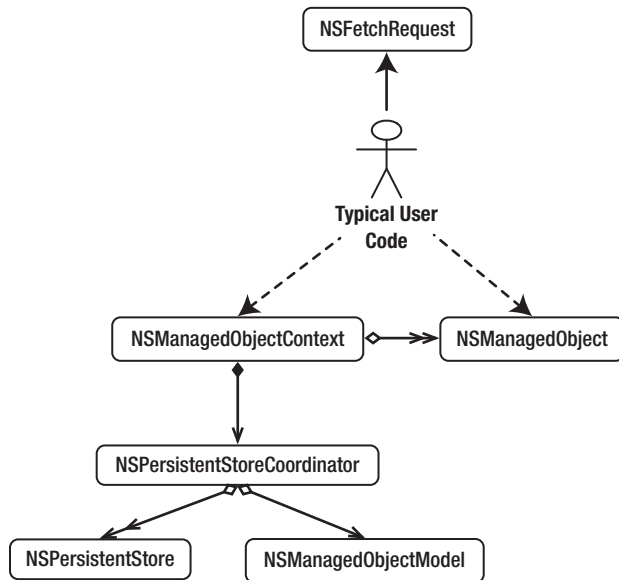
Add the Core Data framework to the project, as described in the “Adding Core Data to an Existing Project” section of Chapter 1. Your Xcode project should look like Figure 2–2. Launch the app and make sure it starts without crashing. It doesn't do anything other than display a gray screen, but that blank screen means you are set to continue building the rest of the Core Data–based application.



**Figure 2–2.** Xcode with a blank project, with Core Data added

Figure 2–3 depicts the classes of Core Data that you typically interact with. Chapter 1 talked about the managed object context, which is materialized by the `NSManagedObjectContext` class and contains references to managed objects of type `NSManagedObject`.





**Figure 2–3.** High-level overview

Your code stores data by adding managed objects to the context and retrieves data by using fetch requests implemented by the `NSFetchRequest` class. As shown in Chapter 1, the context is initialized using the persistent store coordinator, implemented by the `NSPersistentStoreCoordinator` class, and defined by the data model, implemented by the `NSManagedObjectModel` class. The remainder of this chapter deals with how these classes are created, how they interact, and how to use them.

## The Model Definition Classes

As explained in Chapter 1, all Core Data applications require an object model. The model defines the entities to be persisted and their properties. Entities have three kinds of properties:

- Attributes
- Relationships
- Fetched properties

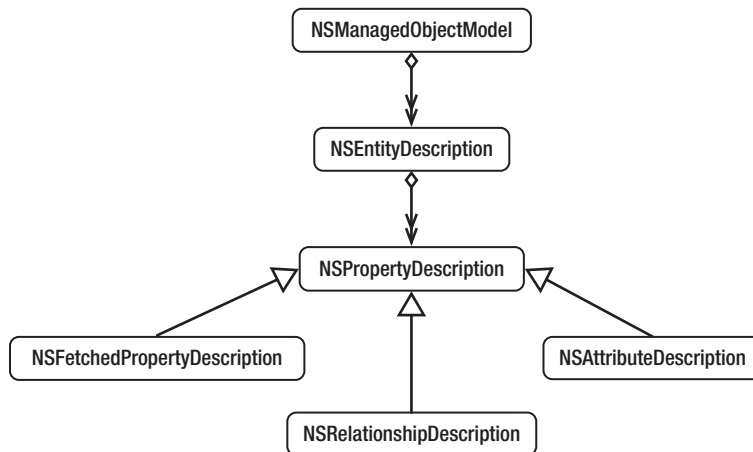
Table 2–1 shows the different classes and brief descriptions of their roles. Enumerating through the classes to better understand how the mechanics behind model instantiation work is an interesting exercise, but in practice creating the model in Xcode is typically done graphically without having to type a single line of code.

**Table 2–1.** *The Classes Involved in Defining a Model*

| Class Name                                | Role   |
|---|--|
| <code>NSManagedObjectModel</code>         | The data model itself  |
| <code>NSEntityDescription</code>          | An entity in the model   |
| <code>NSPropertyDescription</code>        | An abstract definition of an entity's property                     |
| <code>NSAttributeDescription</code>       | An attribute of an entity  |
| <code>NSRelationshipDescription</code>    | A reference from an entity to another entity                       |
| <code>NSFetchedPropertyDescription</code> | The definition of a subset of entity instances based on a criteria |

Figure 2–4 shows the relationships among the classes involved in defining a model. `NSManagedObjectModel` has references to zero or more `NSEntityDescription` objects. Each `NSEntityDescription` has references to zero or more `NSPropertyDescription` objects. `NSPropertyDescription` is an abstract class with three concrete implementations:

- `NSAttributeDescription`
- `NSRelationshipDescription`
- `NSFetchedPropertyDescription`

**Figure 2–4.** *The model definition classes*

This small set of classes is enough to define any object model you will use when working on your Core Data projects. As explained in more detail in Chapter 1, you create the data model in Xcode by selecting **File > New > New File** in the menu and picking the

type Data Model from the iOS Core Data templates. In this section, you will create a model that represents a company's organizational chart. In Xcode, create your model and call it **OrgChart**. In this data model, an organization has one person as the leader (the chief executive officer, or CEO). That leader has direct reports, which may or may not have direct reports of their own. For simplicity, say that a person has two attributes: a unique employee identifier and a name. You are finally ready to start defining the data model in Xcode.

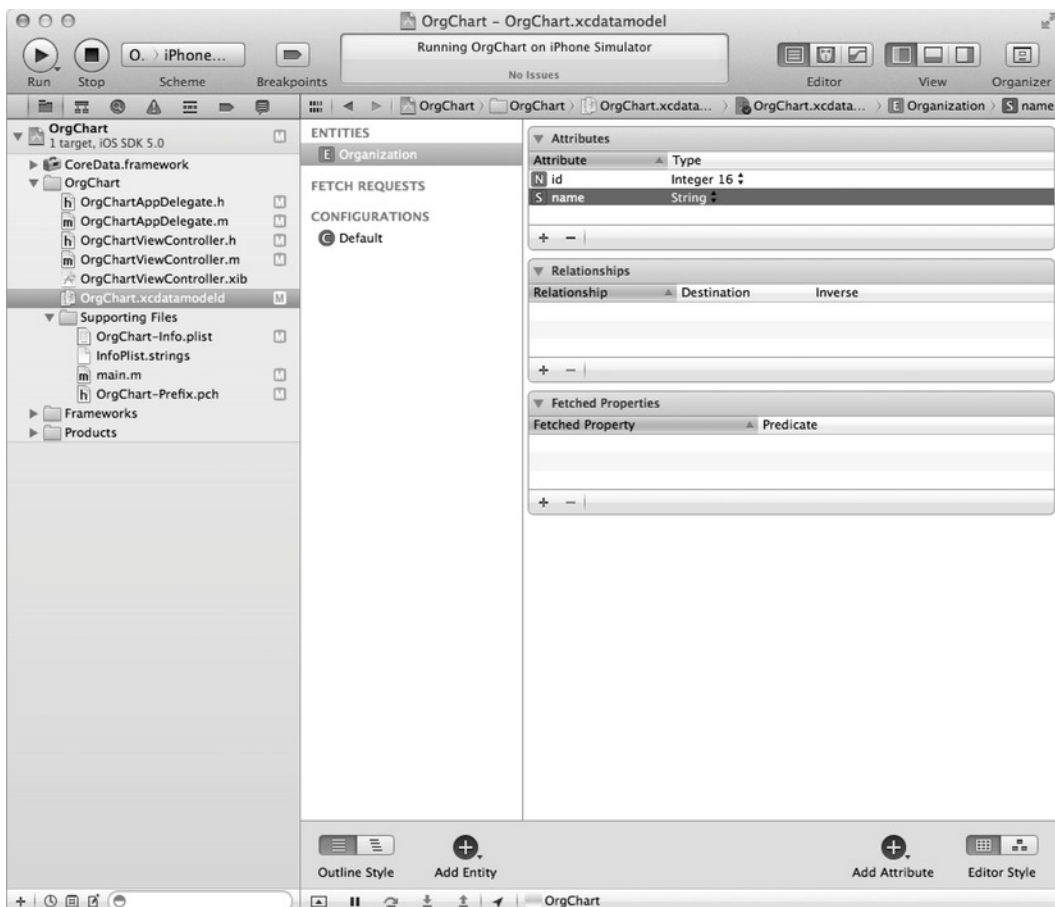
Open the data model file, and add a new *Organization* entity. Like a person, an organization is defined by a unique identifier and a name, so add two attributes to the *Organization* entity. Attributes are scalar properties of an entity, which means that they are simple data holders that can contain a single value. The attribute types are defined in the *NSAttributeType* structure, and each type enforces certain data constraints over the entity. For instance, if the type of the attribute is integer, an error will occur if you try to put an alphanumeric value in that field. Table 2-2 lists the different types and their meanings.

**Table 2-2.** *Attribute Types*

| Xcode Attribute Type | Objective-C Attribute Type          | Objective-C Data       | Description                                |
|----------------------|-------------------------------------|------------------------|--|
| Integer 16           | <i>NSInteger16AttributeType</i>     | <i>NSNumber</i>        | A 16-bit integer                           |
| Integer 32           | <i>NSInteger32AttributeType</i>     | <i>NSNumber</i>        | A 32-bit integer                           |
| Integer 64           | <i>NSInteger64AttributeType</i>     | <i>NSNumber</i>        | A 64-bit integer                           |
| Decimal              | <i>NSDecimalAttributeType</i>       | <i>NSDecimalNumber</i> | A base-10 subclass of <i>NSNumber</i>      |
| Double               | <i>NSDoubleAttributeType</i>        | <i>NSNumber</i>        | An object wrapper for double               |
| Float                | <i>NSFloatAttributeType</i>         | <i>NSNumber</i>        | An object wrapper for float                |
| String               | <i>NSStringAttributeType</i>        | <i>NSString</i>        | A character string                         |
| Boolean              | <i>NSBooleanAttributeType</i>       | <i>BOOL</i>            | An object wrapper for a Boolean value      |
| Date                 | <i>NSDateAttributeType</i>          | <i>NSDate</i>          | A date and time value                      |
| Binary data          | <i>NSBinaryDataAttributeType</i>    | <i>NSData</i>          | Unstructured binary data                   |
| Transformable        | <i>NSTransformableAttributeType</i> | Any nonstandard type   | Any type transformed into a supported type |

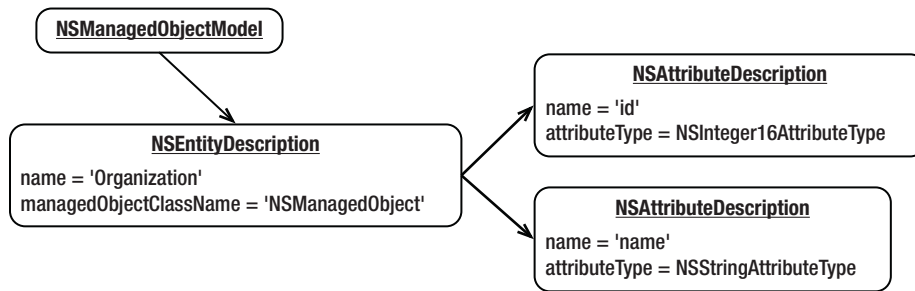
**NOTE:** Chapter 5 expands on the Transformable type and how to use it. Transformable attributes are a way to tell Core Data that you want to use a nonsupported data type in your managed object and that you will help Core Data by providing code to transform the attribute data at persist time into a supported type.

Name the first attribute `id` and the second name. By default, the type of new attributes is set to Undefined, which purposely prevents the code from compiling in order to force you to set the type for each attribute. Organization identifiers are always going to be simple numbers, so use the type `Integer 16` for the `id` attribute. Use the type `String` for the `name` attribute. At this point, your project should look like Figure 2–5.



**Figure 2–5.** The project with the Organization entity

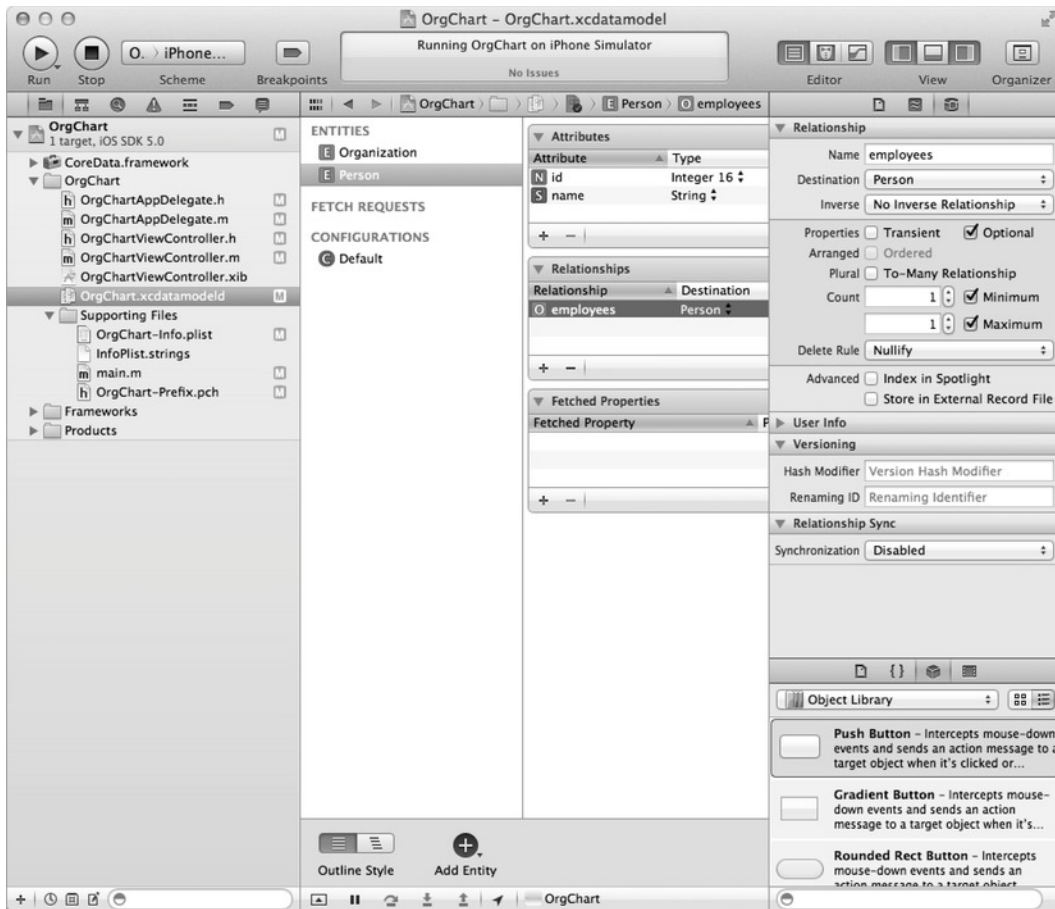
At this point, if the data model were to be loaded by a running program, it would be represented by the object graph shown in Figure 2–6.



**Figure 2–6.** *The organization model as objects*

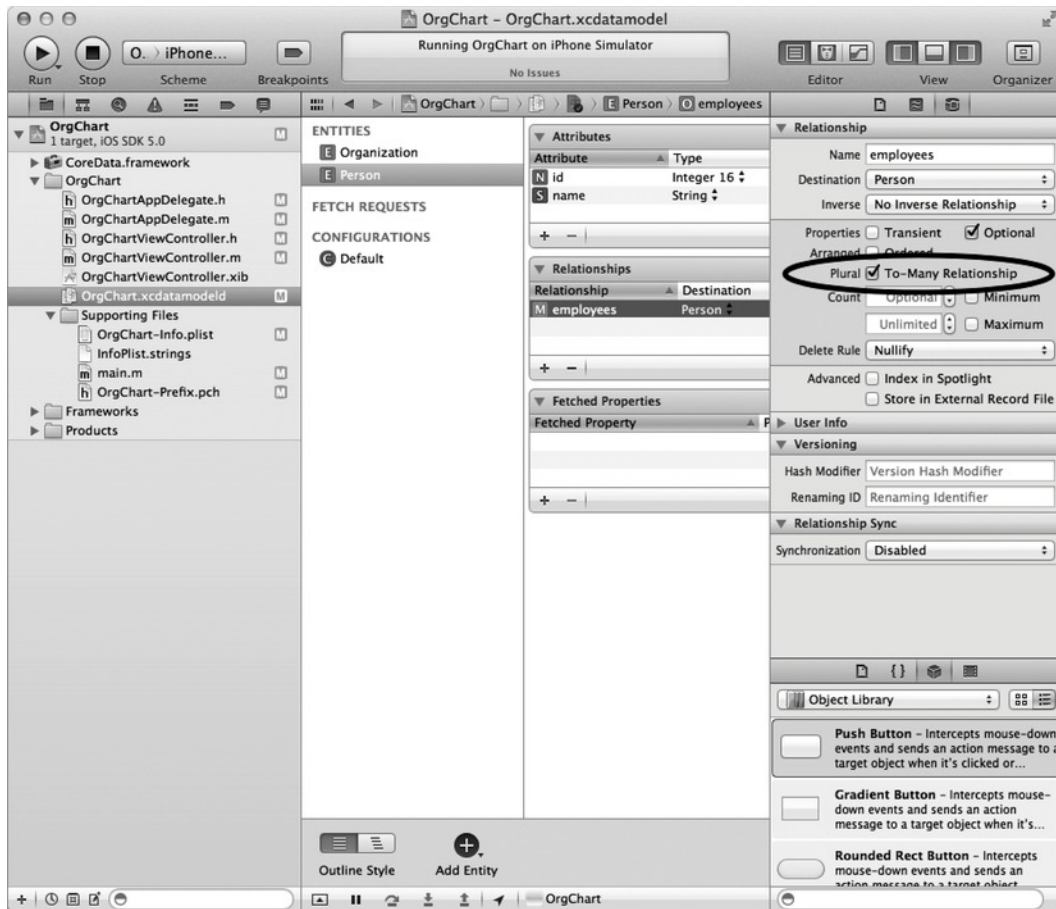
The graph shows that the managed object model, which is of type `NSManagedObjectModel`, points to an entity description, represented by an `NSEntityDescription` instance named `Organization` that uses an `NSManagedObject` for its `managedObjectClassName` property. This entity description has two attribute descriptions (type `NSAttributeDescription`). Each attribute description has two properties: `name` and `attributeType`. The first attribute description has the values `id` and `NSInteger16AttributeType` for its `name` and `attributeType` properties, respectively, while the second has the values `name` and `NSSStringAttributeType`.

In the same manner that you created the `Organization` entity, create another entity named `Person` with two attributes: `id` and `name`. You can link the organization to its leader by creating a relationship from the `Organization` entity to the `Person` entity and call it `leader`. Once the two entities exist in Xcode, creating a relationship is as simple as selecting the source entity (`Organization`), clicking and holding the `Add Attribute` button until the drop-down menu appears, and selecting `Add Relationship`. Name the new relationship `leader`, and set `Person` as its destination. Now, add a relationship from `Person` to `Person` (yes, a relationship from the `Person` entity back to itself), and call it `employees`. This defines the relationship from a person to the person's subordinates. By default, Xcode creates one-to-one relationships, which means that a relationship left at the default links one source to one destination. Since one leader can manage many subordinates, the relationship between `Person` and `Person` should be a one-to-many relationship. To correct the relationship in Xcode, first show the attributes for the relationship by displaying Xcode's utilities panel by clicking the rightmost button above `View` in the toolbar of Xcode. Then, click to show the `Data Model Inspector` by clicking the rightmost button at the top of the utilities panel (see Figure 2–7).



**Figure 2-7.** Showing the properties of a relationship

With the properties displayed, you can now change the employees relationship to one-to-many by checking the box beside To-Many Relationship, as shown in Figure 2-8.

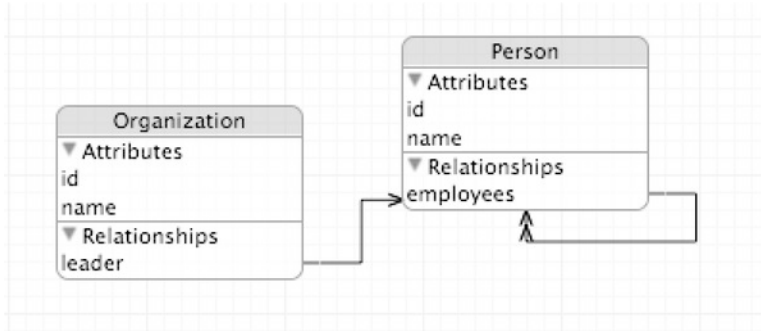


**Figure 2–8.** One-to-many relationship

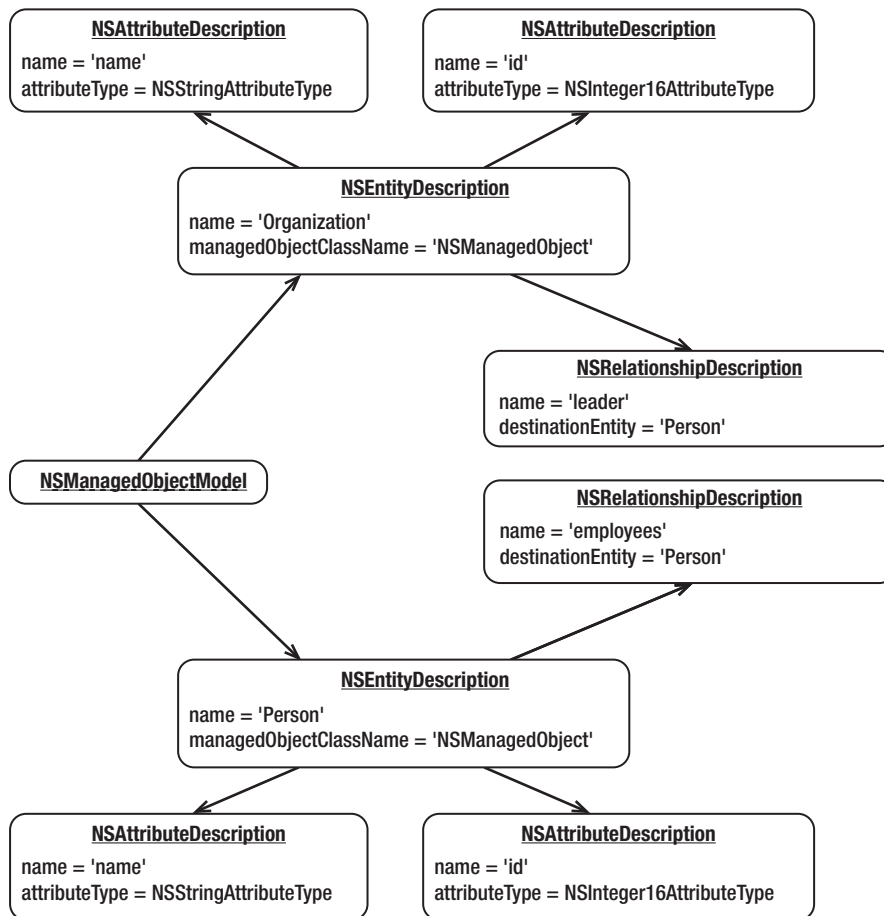
**NOTE:** In this chapter, we want to show you how a data model designed using the model editor is interpreted into Core Data classes. We have purposely taken shortcuts when creating the model. For more detailed information on how to create object models, please refer to Chapter 4.

**NOTE:** You may notice that Xcode complains about the two relationships you've created, warning you of a consistency error and a misconfigured property. Xcode complains about Core Data relationships without inverses, but this shouldn't be an issue in this application. The only impact the lack of the inverse relationships has is that you can't navigate the inverse relationships. You can safely ignore these warnings for the time being.

The current data model is illustrated in Figure 2–9, while Figure 2–10 shows how the data model definition would be loaded into live objects by Core Data.



**Figure 2–9.** *The data model*



**Figure 2–10.** *The organization model with Person*



Note that the object graph depicted in Figure 2–10 is not showing the object graph that Core Data stores. Instead, it is an illustration of how Core Data interprets the data model you created graphically as objects it can use to create the data store schema. The typical way of making Core Data load and interpret a data model into an `NSManagedObjectModel` is done by creating a URL to the data model using code like this:

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

Then, you can pass that URL to the managed object model's initializer to create the model, which loads the object model description into memory. That code looks like this:

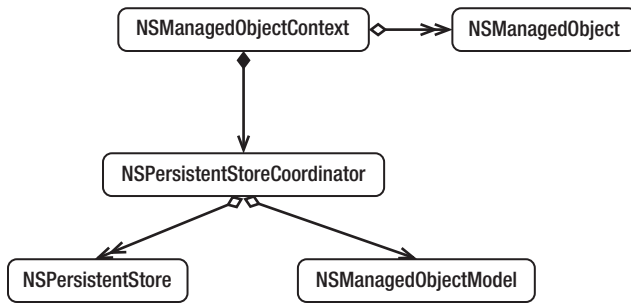
```
__managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];
```

Knowing how models are represented as objects inside Core Data is generally not very useful unless you are interested in creating custom data stores or want to generate the data model programmatically at runtime. This would be analogous to programmatically creating `UIView`s rather than using Xcode's built-in user interface editor, Interface Builder. Having a deep understanding of how Core Data works, however, allows you to anticipate and avoid complex issues, troubleshoot bugs, and solve problems creatively and elegantly.

You have seen how the data model is represented as Core Data classes, but you have also seen that unless you get into really advanced uses of the framework, you will hardly ever need to interact directly with these classes. All the classes discussed so far in this chapter deal with describing the model. The remainder of the chapter deals exclusively with classes that represent either the data itself or the accessors to the data.

## The Data Access Classes

You learned in Chapter 1 that the initialization of Core Data starts with loading the data model into the `NSManagedObjectModel` object. The previous section ran through an example of how `NSManagedObjectModel` represents that model as `NSEntityDescription` and `NSPropertyDescription` objects. The second step of the initialization sequence is to create and bind to the persistent store through the `NSPersistentStoreCoordinator`. Finally, the third step in the initialization sequence creates the `NSManagedObjectContext` that your code interacts with in order to store and retrieve data. To make things a bit clearer, Figure 2–11 shows the class diagram involved in representing the context and the underlying persistent store.



**Figure 2-11.** *The managed object context object graph*

Notice where the `NSManagedObjectModel` object sits in the class diagram in Figure 2-11. It is loaded and given to the persistent store coordinator (`NSPersistentStoreCoordinator`) so that the persistent store coordinator can figure out how to represent the data in the persistent stores. The persistent store coordinator acts as a mediator between the managed object context and the actual persistent stores where the data is written. Among other tasks, it manages the data migrations from one store to the other, through the `migratePersistentStore:` method.

The `NSPersistentStoreCoordinator` is initialized using the `NSManagedObjectModel` class. Once the coordinator object is allocated, it can load and register all the persistent stores. The following code demonstrates the initialization of the persistent store coordinator. It receives the managed object model in its `initWithManagedObjectModel:` method and then registers the persistent store by calling its `addPersistentStoreWithType:` method.

```

NSURL *storeURL = [[self applicationDocumentsDirectory]
    URLByAppendingPathComponent:@"OrgChart.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();
}

```

iOS offers three types of persistent stores by default, as shown in Table 2-3.

**Table 2-3.** *The Default Persistent Store Types on iOS*

| Store Type                       | Description       |
|----------------------------------|-------------------|
| <code>NSSQLiteStoreType</code>   | SQLite database   |
| <code>NSBinaryStoreType</code>   | Binary file       |
| <code>NSInMemoryStoreType</code> | In-memory storage |

**NOTE:** Core Data on Mac OS X offers a fourth type (`NSXMLStoreType`) that uses XML files as the storage mechanism. This fourth type isn't available on iOS, presumably because of iDevices' slower processors and the processor-intensive nature of parsing XML. Parsing XML is possible on iDevices, however, as the existence of several iOS XML parsing libraries proves. Apple seems to have issued no official explanation for the absence of the XML store type on iOS.

Typical users will find themselves using the SQLite type, which stores the data into a SQLite database running on your iDevice, most often or even exclusively. You should be aware of and understand the other types, however, because different circumstances may warrant using the other types. As useless as an in-memory “persistent” store may sound, a classic use of the in-memory storage is to cache information fetched from a remote server and create a local copy of the data to limit bandwidth usage and the associated latencies. The data remains persistent on the remote server and can always be fetched again.

Once the persistent store coordinator is created, the managed object context can be initialized as an `NSManagedObjectContext` instance. The managed object context is responsible for coordinating what goes in and what comes out of the persistent store. This may sound like a trivial task, but consider the following challenges:

- If two threads ask for the same object, they must obtain a pointer to the same instance.
- If the same object is asked for several times, the context must be intelligent enough to not go hit the persistent store but instead return the object from its cache.
- The context should be able to keep changes to the persistent store to itself until a commit operation is requested.

**NOTE:** Apple strongly, and rightfully, discourages subclassing the `NSManagedObjectContext` because of the complexity of what it does and the opportunities for putting the context and the objects it manages into an unpredictable state.

Table 2–4 lists some methods used to retrieve, create, or delete data objects. Updating data is done on the data object by changing its properties directly. The context keeps track of every object it pulls out of the data store and is aware of any change you make so that changes can be persisted to the backing store.

**Table 2–4.** *Some Useful Methods for Retrieving and Creating Data Objects*

| Method Name                              | Description  |
|--|--|
| <code>-executeFetchRequest:error:</code> | Executes a request to retrieve objects.                  |
| <code>-objectWithID:</code>              | Retrieves a specific object given its unique identifier. |
| <code>-insertObject:</code>              | Adds a new object to the context.                        |
| <code>-deleteObject:</code>              | Removes an object from the context.                      |

Keep in mind that `NSManagedObjectContext` orchestrates data transfers both into and out of the persistent store and guarantees the consistency of the data objects. You should think carefully about the impacts of deleting managed objects. When deleting a managed object that has a relationship to other managed objects, Core Data has to decide what to do with the related objects. One of the properties of a relationship is the “Delete rule,” which is one of the four types of delete rules explained in Table 2–5. Table 2–5 calls the object being deleted the *parent object* and the objects that are at the end of a relationship the *related objects*.

**Table 2–5.** *The Delete Rules*

| Rule      | Effect   |
|-----------|--|
| No action | Does nothing and lets the related objects think the parent object still exists.        |
| Nullify   | For each related object, sets the parent object property to null.                      |
| Cascade   | Deletes each related object.   |
| Deny      | Prevents the parent object from being deleted if there is at least one related object. |

In addition to controlling access to and from the persistent store, the managed object context allows undo and redo operations using the same paradigm used in user interface design. Table 2–6 lists the important methods used in dealing with undo and redo operations.

**Table 2–6.** *Life-Cycle Operations in NSManagedObjectContext*

| Method Name                   | Description   |
|-------------------------------|---|
| <code>-undoManager</code>     | Returns the <code>NSUndoManager</code> controlling undo in the context.   |
| <code>-setUndoManager:</code> | Specifies a new undo manager to use.  |
| <code>-undo</code>            | Sends an undo message to the <code>NSUndoManager</code> .   |
| <code>-redo</code>            | Sends a redo message to the <code>NSUndoManager</code> .  |
| <code>-reset</code>           | Forces the context to lose all references to the managed objects it retrieved.  |
| <code>-rollback</code>        | Sends undo messages to the <code>NSUndoManager</code> until there is nothing left to undo.  |
| <code>-save:</code>           | Sends all current changes in the context to the persistent store. Think of this as the “commit” action. Any changes in the context that have not been saved are lost if your application crashes. |
| <code>-hasChanges</code>      | Returns true if the context contains changes that have not yet been committed to the persistent store.  |

Chapter 5 discusses undoing and redoing Core Data operations.

The `NSManagedObjectContext` is the gateway into the persistent stores that all data objects must go through. To be able to keep track of the data, Core Data forces all data objects to inherit from the `NSManagedObject` class. You saw earlier in this chapter that `NSEntityDescription` instances define the data objects. `NSManagedObject` instances are the data objects. Core Data uses their entity descriptions to know what properties to access and what types to expect.

Managed objects support key-value coding, or name-value pairs, in order to give generic accessors to the data they contain. The simplest way of accessing data is through the `valueForKey:` method. Data can be put into the objects using the `setValue:forKey:` method. The key parameter is the name of the attribute from the data model.

```
- (id)valueForKey:(NSString *)key
- (void)setValue:(id)value forKey:(NSString *)key
```

Note that the `valueForKey:` method returns an instance of `id`, which is a generic type. The actual data type is dictated again by the data model and the type of attribute you specified. Refer to Table 2–2 for a list of these types.

`NSManagedObject` also provides several methods to help the `NSManagedObjectContext` determine whether anything has changed in the object. When the time comes to commit

any changes, the context can ask the managed objects it keeps track of if they have changed. Apple strongly discourages you from overriding these methods in order to prevent interference with the commit sequence.

**NOTE:** Each instance of `NSManagedObject` keeps a reference to the context to which it belongs. This can also be an inexpensive way of keeping a reference to the context without having to pass it around all the time. As long as you have one of the managed objects, you can get back to the context.

To create a new `NSManagedObject` instance, you first create an appropriate `NSEntityDescription` instance from the entity name and the managed object context that knows about that entity. The `NSEntityDescription` instance provides the definition, or description, for the new `NSManagedObject` instance. You then send a message to the `NSEntityDescription` instance you just created to insert a new managed object into the context and return it to you. The code looks like this:

```
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Organization"
inManagedObjectContext:managedObjectContext];
NSManagedObject *org = [NSEntityDescription insertNewObjectForEntityForName:[entity
name] inManagedObjectContext:managedObjectContext];
```

Once you create the managed object in the context, you can set its values. To set the name property of the `NSManagedObject` instance called `org` that the previous code creates, invoke the following code:

```
[org setValue:@"MyCompany, Inc." forKey:@"name"];
```

## Key-Value Observing

You can take responsibility for discovering any changes made to managed objects by querying them. This approach is useful in cases when some event prompts a controller to do something with the managed objects. In many cases, however, having the managed objects themselves take responsibility to notify you when they change allows more efficiency and elegance in your design. To provide this notification service, the `NSManagedObject` class supports key-value observation to send notifications immediately before and after a value changes for a key. Key-value observation isn't a pattern specific to Core Data but is used throughout the Cocoa frameworks.

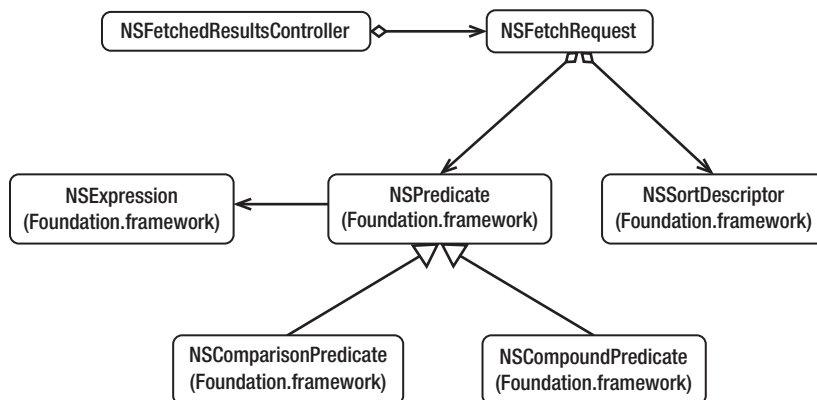
The two main methods involved in the notifications related to key-value observation are `willChangeValueForKey:` and `didChangeValueForKey:`. The first method is invoked immediately before the change and the latter immediately after the change. For an object observer object to receive notifications from changes to the value of property theProperty occurring in a managed object managedObject, however, the observer object must first register with that object using code like this:

```
[managedObject addObserver:observerObject forKeyPath:@"theProperty"
options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld) context:nil];
```

After registering, the observer will start receiving instant change notifications to the properties to which it is listening. This is an extremely useful mechanism to use when trying to keep a user interface in sync with the data it displays.

## The Query Classes

So far in this chapter, you have seen how to initialize the Core Data infrastructure and how to interact with the objects it creates. This section deals with the basics of retrieving data by sending fetch requests. Fetch requests are, not surprisingly, instances of `NSFetchRequest`. What might be a little more surprising to you is that it is almost the only class in the class structure related to retrieving data that is a member of the Core Data framework. Most of the other classes involved in formulating a request to fetch data belong to the Foundation Cocoa framework, also used by all other Cocoa frameworks. Figure 2–12 shows the class diagram derived from `NSFetchRequest`.



**Figure 2–12.** The *NSFetchRequest* class diagram

Fetch requests are composed of two main elements: an `NSPredicate` instance and an `NSSortDescriptor` instance. The `NSPredicate` helps filter the data by specifying constraints, and the `NSSortDescriptor` arranges the result set in a specific order. Both elements are optional, and if you don't specify them, your results aren't constrained if an `NSPredicate` isn't specified or aren't sorted if an `NSSortDescriptor` isn't specified.

Creating a simple request to retrieve all the organizations in your persistent store—unconstrained and unsorted—looks like this:

```

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Organization"
    inManagedObjectContext:managedObjectContext];
[fetchRequest setEntity:entity];
NSArray* organizations = [managedObjectContext executeFetchRequest:fetchRequest
    error:nil];
  
```

You first reserve the memory space for the request object. You then retrieve the entity description of the objects to retrieve from the context and set it into the request object.

Finally, you execute the query in the managed object context by calling `executeFetchRequest:`.

Because you don't constrain the request with an `NSPredicate` instance, the `organizations` array contains instances of `NSManagedObject` that represent each of the organizations found in the persistent store. To filter the results and return only some of the organizations, use the `NSPredicate` class. To create a simple predicate that limits the results to organizations whose names contain the string "Inc.", you call `NSPredicate`'s `predicateWithFormat:` method, passing the format string and the data that gets plugged into the format. You then add the predicate to the fetch request before executing the request in the managed object context. The code looks like this:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name contains %@", @"Inc."];  
[fetchRequest setPredicate:predicate];
```

To sort the result set alphabetically by the name of the organization, add a sort descriptor to the request prior to executing it.

```
NSSortDescriptor *sortByName = [[NSSortDescriptor alloc] initWithKey:@"name" ←  
ascending:YES];  
[fetchRequest setSortDescriptors:[NSArray arrayWithObject:sortByName]];
```

Chapter 6 goes much more in depth on how to use predicates for simple and complex queries and sort descriptors for sorting.

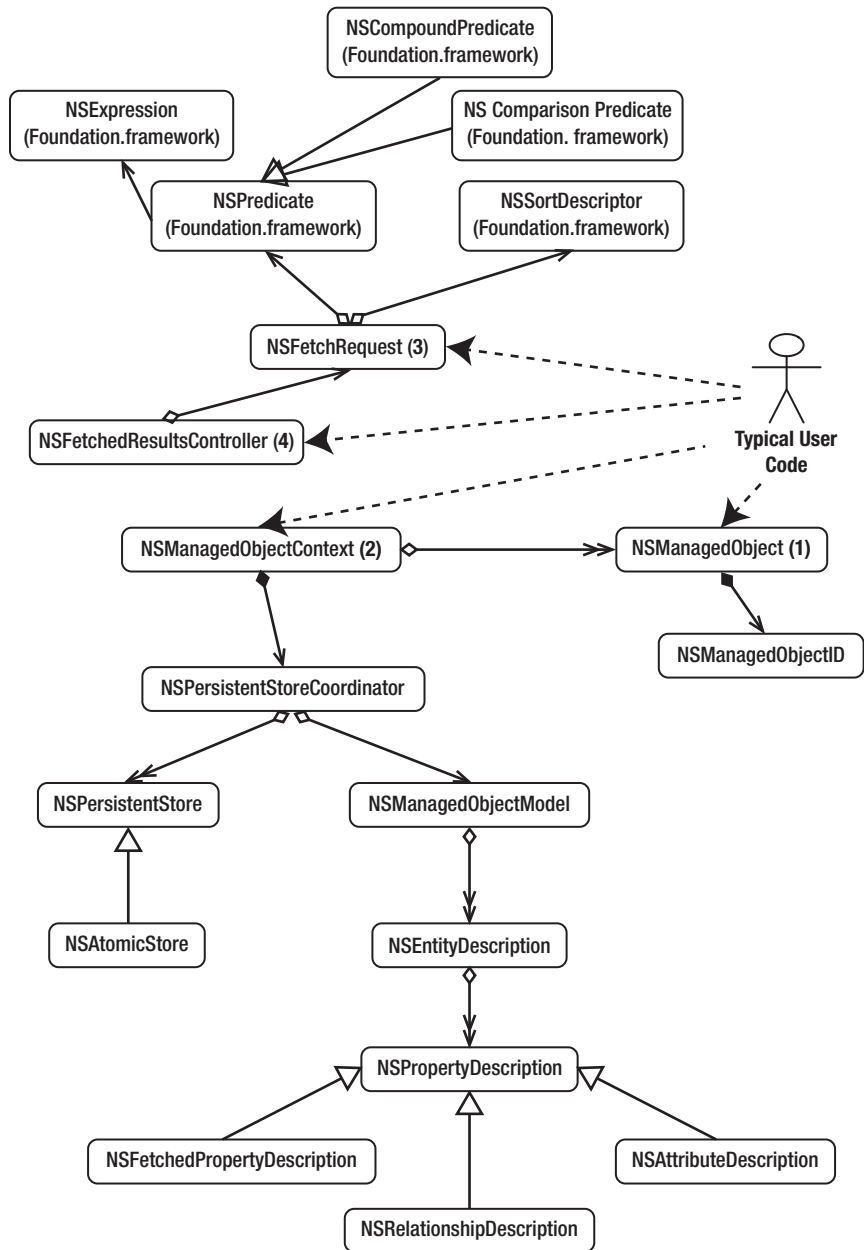
Figure 2–13 puts all the Core Data class diagrams you've seen in this chapter together into a single class diagram that shows you how all these classes fit in the framework.

Your user code, represented in the diagram by a stick figure, interacts chiefly with four classes:

- `NSManagedObject` (1)
- `NSManagedObjectContext` (2)
- `NSFetchRequest` (3)
- `NSFetchedResultsController` (4)

Find these four classes, numbered 1–4, in the diagram in Figure 2–13 and use them as anchor points, and you should recognize the individual class diagrams you've worked through in this chapter. The one exception is `NSFetchedResultsController` (4), which works closely with iOS table views. We cover `NSFetchedResultsController` in Chapter 9.



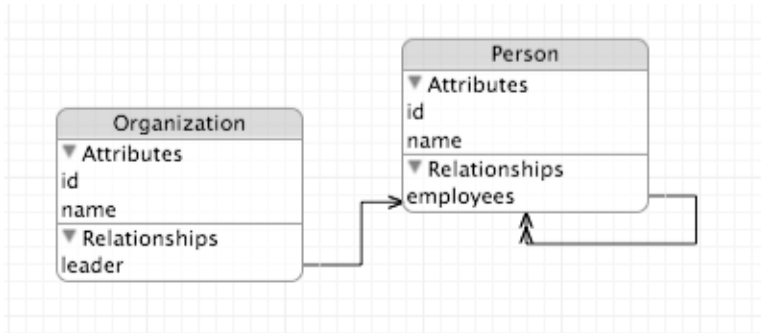


**Figure 2-13.** The main Core Data classes

## How the Classes Interact

You should now have a good understanding of the classes involved in interacting with Core Data and what is happening under the hood. This section expands on the OrgChart application created in the previous section to show some examples of interactions among Core Data classes. Like the previous section, this chapter maintains its focus at a relatively basic level. The goal for this chapter is to show how the classes play together. The rest of the book consists of chapters that take each concept (data manipulation, custom persistent store, performance enhancements, and so on) to much deeper levels.

The data model created in the previous section should look like the one in Figure 2-14.



**Figure 2-14.** *The Organization data model*

Both the Organization and Person entities use an attribute of type Integer 16 to serve as unique identifiers for each entity. Most programmers reading this have probably already thought about autoincrement and have been clicking around the Xcode user interface to find out by themselves how to enable autoincrement for the identifier attributes. If you're one of these programmers and you're back to reading this chapter, you probably became frustrated and have given up looking. The reason you couldn't find autoincrement is because it's not there. Core Data manages an object graph using real object references. It has no need for a unique autoincrement identifier that can be used as a primary key in the same sense it would be used in a relational database. If the persistent store happens to be a relational database like SQLite, the framework will probably use some autoincrement identifiers as primary keys. Regardless, you, as a Core Data user, should not have to worry about that; it's an implementation detail. We have purposely introduced the notion of identifiers in this example for two reasons:

- To raise the question of autoincrement.
- To show examples of how to manage numeric values with Core Data.

The person identifier is similar in meaning to a Social Security number. It isn't meant to autoincrement because it is an identifier computed outside the data store. In the Organization example, you simply derive the ID from an object hash. Although this

doesn't guarantee uniqueness, it serves this application's purpose and is simple to implement.

**NOTE:** As a programmer, you should be careful not to focus too much on what you already know about databases. Core Data isn't a database; it's an object graph persistence framework. Its behavior is closer to an object-oriented database than a traditional relational database.

Listing 2-1 shows the header file for the application delegate class, `OrgChartAppDelegate.h`. You can see the three Core Data–related properties:

- `NSManagedObjectContext *managedObjectContext`
- `NSManagedObjectModel *managedObjectModel`
- `NSPersistentStoreCoordinator *persistentStoreCoordinator`

You also see the declaration for a method to return the application's document directory, which is where the persistent store will live, and another declaration for a method, `saveContext:`, to save the managed object context. The implementation file, `OrgChartAppDelegate.m`, will define methods for each of these.

**Listing 2-1.** *OrgChartAppDelegate.h*

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

@class OrgChartViewController;

@interface OrgChartAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) OrgChartViewController *viewController;
@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
    *persistentStoreCoordinator;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```

In `OrgChartAppDelegate.m`, add `@synthesize` lines for the three Core Data–related properties from `OrgChartAppDelegate.h`, like this:

```
@synthesize managedObjectContext = __managedObjectContext;
@synthesize managedObjectModel = __managedObjectModel;
@synthesize persistentStoreCoordinator = __persistentStoreCoordinator;
```

You then write code for the accessors for the three Core Data–related properties. Each accessor first determines whether the object it's responsible for has been created. If it has, the accessor returns the object. If not, the accessor creates the object using the

appropriate Core Data formula, creating any of the other member objects it requires, and returns it. The code in Listing 2–2 to add to `OrgChartAppDelegate.m` should look familiar.

**Listing 2–2.** *Adding the Accessors for the Three Core Data-Related Properties*

```
- (void)saveContext
{
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil)
    {
        if ([managedObjectContext hasChanges] && ![managedObjectContext save:&error])
        {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
}

#pragma mark - Core Data stack

/**
 Returns the managed object context for the application.
 If the context doesn't already exist, it is created and bound to the persistent store
 coordinator for the application.
 */
- (NSManagedObjectContext *)managedObjectContext
{
    if (__managedObjectContext != nil)
    {
        return __managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil)
    {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return __managedObjectContext;
}

/**
 Returns the managed object model for the application.
 If the model doesn't already exist, it is created from the application's model.
 */
- (NSManagedObjectModel *)managedObjectModel
{
    if (__managedObjectModel != nil)
    {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"OrgChart"
withExtension:@"momd"];
}
```

```

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

/**
 Returns the persistent store coordinator for the application.
 If the coordinator doesn't already exist, it is created and the application's store
 added to it.
 */
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    if (__persistentStoreCoordinator != nil)
    {
        return __persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]~
    URLByAppendingPathComponent:@"OrgChart.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;
}

#pragma mark - Application's Documents directory

/**
 Returns the URL to the application's Documents directory.
 */
- (NSURL *)applicationDocumentsDirectory
{
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory~
    inDomains:NSUserDomainMask] lastObject];
}

```

Since this chapter focuses on how Core Data works, not on user interface design and development, the OrgChart application can remain content with a blank, gray screen. All its work will happen in the `didFinishLaunchingWithOptions:` method, and we show you how to use external tools to verify that Core Data persisted the objects appropriately.

Before you can read any data from the persistent store, you have to put it there. Change the `didFinishLaunchingWithOptions:` method to call a new method called `createData:`, like this:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self createData];
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[OrgChartViewController alloc]
initWithNibName:@"OrgChartViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

In `OrgChartAppDelegate.h`, declare the `createData:` method like this:

```
- (void)createData;
```

Now define the `createData:` method in `OrgChartAppDelegate.m` like this:

```

- (void)createData
{
}

```

Inside the `createData:` implementation, you create an organization and three employees. Start with the organization. Remember that the managed objects, as far as you're concerned, live in the managed object context. Don't worry about the persistent store; the managed object context takes care of managing the persistent store for you. You just create the managed object representing the organization and insert it into the managed object context using this code:

```

NSManagedObject *organization = [NSEntityDescription
insertNewObjectForEntityForName:@"Organization"
inManagedObjectContext:self.managedObjectContext];

```

You've now created an organization, but you've given it no name and no identifier. The `name` attribute is of type `String`, so set the name for the organization like this:

```
[organization setValue:@"MyCompany, Inc." forKey:@"name"];
```

The `id` attribute is of type `Integer 16`, which is a class. All Core Data attributes must have a class type, not a primitive type. To create the `id` attribute for the organization, create a primitive `int` from the hash of the name of the organization and then convert the primitive `int` into an `NSNumber` object, which can be used in the `setValue:forKey:` method. The code looks like this:

```
[organization setValue:[NSNumber numberWithInt:[@"MyCompany, Inc." hash]] forKey:@"id"];
```

**NOTE:** Core Data uses objects for attributes in order to comply with key-value coding requirements. Any primitive type should be converted into `NSNumber` to use the `setValue:forKey:` method.

The organization now has a name and an identifier, although the changes haven't yet been committed to the persistent store. The framework knows that you've altered an object that it's tracking and makes the necessary adjustments to the persistent store when the `save:` method is called to commit the context changes.

An organization without people accomplishes little, so create three people named John, Jane, and Bill using the same approach you used to create the organization: create a new managed object in the managed object context using the appropriately named entity description, and set the name and identifier values. The code looks like this:

```
NSManagedObject *john = [NSEntityDescription insertNewObjectForEntityForName:@"Person"
inManagedObjectContext:self.managedObjectContext];
[john setValue:@"John" forKey:@"name"];
[john setValue:[NSNumber numberWithInt:[@"John" hash]] forKey:@"id"];
```

```
NSManagedObject *jane = [NSEntityDescription insertNewObjectForEntityForName:@"Person"
inManagedObjectContext:self.managedObjectContext];
[jane setValue:@"Jane" forKey:@"name"];
[jane setValue:[NSNumber numberWithInt:[@"Jane" hash]] forKey:@"id"];
```

```
NSManagedObject *bill = [NSEntityDescription insertNewObjectForEntityForName:@"Person"
inManagedObjectContext:self.managedObjectContext];
[bill setValue:@"Bill" forKey:@"name"];
[bill setValue:[NSNumber numberWithInt:[@"Bill" hash]] forKey:@"id"];
```

You now have one organization and three unrelated people. The organizational chart should have John as the leader of the company, while Jane and Bill should report to John. The OrgChart application's data model has two types of relationships. The leader is a one-to-one relationship from Organization to Person, while the employees are set through a one-to-many relationship between Person and itself. Setting the value of the one-to-one relationship is surprisingly simple.

```
[organization setValue:john forKey:@"leader"];
```

Core Data knows that you are assigning a relationship value because it knows the data model and knows that `leader` represents a one-to-one relationship. Your code would not run if you were to pass a managed object with the wrong entity type as the value.

To assign Jane and Bill as John's subordinates, you have to assign values to the employees one-to-many relationship. Core Data returns the many side of a one-to-many relationship as a set (`NSSet`), which is an unordered collection of unique objects. For John's employees relationship, Core Data returns the set of the existing employees working for John if you call the `valueForKey:` method. Since you want to add objects to the set, however, call the `mutableSetValueForKey:` method, because it returns an `NSMutableSet`, which you can add to and delete from, instead of an immutable `NSSet`. Adding a new Person managed object to the returned `NSMutableSet` adds a new employee to the relationship. The code looks like this:

```
NSMutableSet *johnsEmployees = [john mutableSetValueForKey:@"employees"];
[johnsEmployees addObject:jane];
[johnsEmployees addObject:bill];
```

Once again, since you've modified the object graph of objects tracked by Core Data, you don't have to do anything else in order to help it manage the dependencies between the different pieces of data. Everything behaves just like you would expect from regular objects. Once again, though, the changes aren't committed until you save the managed object context, which you do using the `saveContext:` method you created.

```
[self saveContext];
```

This ends the contents of the `createData:` method. Run the application to create the data in your SQLite database. Again, you won't see anything in the iPhone simulator but a blank, gray screen. In the next section, we use other tools to verify that the OrgChart application added your data to the database.

## SQLite Primer

Since the OrgChart application uses SQLite for its persistent store and you just created the store and put some data into it, you can use any SQLite viewing tool such as Base (<http://menial.co.uk/software/base/>), SQLite Database Browser (<http://sqlitebrowser.sourceforge.net/>) or SQLite Manager (<http://code.google.com/p/sqlite-manager/>) to view it. Be aware, however, that poking around a SQLite database that Core Data owns can create problems if you change the database from an external tool. The way Core Data manages a SQLite database is an implementation detail that could change. Further, you can mangle the data in ways that prevent Core Data from reading the data or properly reconstructing the object graph. Consider yourself warned. While developing applications, however, you can recover from a damaged database by deleting it and allowing your application to re-create it, so don't hesitate to jump into the database using a tool to debug your applications.

Despite the availability of graphical browsing tools, we recommend using the adequate command-line tool that's already installed on your Mac: `sqlite3`. On your Mac, open `Terminal.app` to get to the command prompt and change to the iPhone Simulator's deployment directory, like this:

```
cd ~/Library/Application\ Support/iPhone\ Simulator
```

To find the database file, named `OrgChart.sqlite`, use the `find` command:

```
find . -name "OrgChart.sqlite" -print
```

You should see output that looks something like this:

```
./5.0/Applications/E8654A34-8EC4-4EAF-B531-00A032DD5977/Documents/OrgChart.sqlite
```

The generated ID will differ from the previous example, and Apple has moved the relative path in the past and may do so again, so your path will vary from this. This is the relative path to your database file, so to open the database, pass that relative path to the `sqlite3` executable, like this:

```
sqlite3 ./5.0/Applications/E8654A34-8EC4-4EAF-B531-00A032DD5977/Documents/OrgChart.sqlite
```



Running this command will put you inside the SQLite shell. You can run SQL commands from this point until you exit the shell.

```
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

One interesting command is the `.schema` command, which will display the SQL schema Core Data created to support your object model.

```
sqlite> .schema
CREATE TABLE ZORGANIZATION ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZID
INTEGER, ZLEADER INTEGER, ZNAME VARCHAR );
CREATE TABLE ZPERSON ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZID
INTEGER, Z2EMPLOYEES INTEGER, ZNAME VARCHAR );
CREATE TABLE Z_METADATA (Z_VERSION INTEGER PRIMARY KEY, Z_UUID VARCHAR(255), Z_PLIST
BLOB);
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY, Z_NAME VARCHAR, Z_SUPER INTEGER,
Z_MAX INTEGER);
CREATE INDEX ZORGANIZATION_ZLEADER_INDEX ON ZORGANIZATION (ZLEADER);
CREATE INDEX ZPERSON_Z2EMPLOYEES_INDEX ON ZPERSON (Z2EMPLOYEES);
sqlite>
```

Note that the tables are not named exactly like your entities, the tables have more columns than your entities have attributes, and the database has extra tables not found in the Core Data data model. You can also see that Core Data took care of creating integer columns to be used as primary keys, and it manages their uniqueness. If you know SQLite well, you know that any column defined as `INTEGER PRIMARY KEY` will autoincrement, but the point is that you don't have to know that or even care. Core Data handles the uniqueness.

You should be able to decipher Core Data's code for mapping entities to tables and recognize the two tables that support the entities: `ZORGANIZATION` and `ZPERSON`. You can query the tables after running the OrgChart application and validate that the data is in fact stored in the database.

```
sqlite> select Z_PK, ZID, ZLEADER, ZNAME from ZORGANIZATION;
1|-19904|2|MyCompany, Inc.
```

```
sqlite> select Z_PK, ZID, Z2EMPLOYEES, ZNAME from ZPERSON;
1|6050|2|Jane
2|-28989||John
3|28151|2|Bill
sqlite>
```

The rows are fairly easy to read. You see the single organization you created, and its leader (`ZLEADER`) is the person where `Z_PK=2` (John). John has no boss, but Jane and Bill have the same boss (`Z2EMPLOYEES`), and his `Z_PK` is 2 (John again, as expected).

To exit the SQLite shell, type `.quit` and press Enter.

```
sqlite> .quit
```

## Reading the Data Using Core Data

In this chapter, you wrote data to a SQLite database using Core Data, and you used the `sqlite3` command-line tool to read the data and confirm that it had been correctly stored in the persistent store. You can also use Core Data to read the data from the persistent store. Go back to Xcode, and open the `OrgChartAppDelegate.m` file to add methods for reading the data using Core Data.

Since the employees relationship is recursive (that is, the same source and destination entity), use a recursive method to display a person and their subordinates. The recursive `displayPerson:` method shown next accepts two parameters (the person to display and an indentation level) and recurses through a person's employees to print them at appropriate indentation levels. The code looks like this:

```
- (void)displayPerson:(NSManagedObject*)person withIndentation:(NSString*)indentation
{
    NSLog(@"%@Name: %@", indentation, [person valueForKey:@"name"]);

    // Increase the indentation for sub-levels
    indentation = [NSString stringWithFormat:@"%@" " ", indentation];

    NSSet *employees = [person valueForKey:@"employees"];
    id employee;
    NSEnumerator *it = [employees objectEnumerator];
    while ((employee = [it nextObject]) != nil)
    {
        [self displayPerson:employee withIndentation:indentation];
    }
}
```

Now write a method that retrieves all the organizations from the context and displays them along with their leaders and the leaders' employees. The code is in Listing 2–3.

**Listing 2–3.** *A Method for Reading the Data from the Database*

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *orgEntity = [NSEntityDescription entityForName:@"Organization"
inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:orgEntity];

    NSArray *organizations = [context executeFetchRequest:fetchRequest error:nil];

    id organization;
    NSEnumerator *it = [organizations objectEnumerator];
    while ((organization = [it nextObject]) != nil)
    {
        NSLog(@"Organization: %@", [organization valueForKey:@"name"]);
    }
}
```

```

        NSManagedObject *leader = [organization valueForKey:@"leader"];
        [self displayPerson:leader withIndentation:@"  "];
    }
}

```

Add the method declarations to `OrgChartAppDelegate.h` to silence the compiler warnings.

```

- (void)readData;
- (void)displayPerson:(NSManagedObject *)person withIndentation:(NSString *)indentation;

```

Finally, alter the `didFinishLaunchingWithOptions:` method to call `[self readData]` instead of `[self createData]`.

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    //[self createData];
    [self readData];
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[OrgChartViewController alloc]
initWithNibName:@"OrgChartViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Build and launch the application, and then look in the Debug log to see the following output, which shows the organization you created with John as its leader and also shows John's two subordinates:

```

2011-07-30 22:18:37.422 OrgChart[4460:f203] Organization: MyCompany, Inc.
2011-07-30 22:18:37.424 OrgChart[4460:f203]   Name: John
2011-07-30 22:18:37.425 OrgChart[4460:f203]   Name: Bill
2011-07-30 22:18:37.426 OrgChart[4460:f203]   Name: Jane

```

Feel free to experiment with the OrgChart application, altering the managed objects created in the `createData:` method and calling `createData:` to store your changes and `readData:` to verify your changes were written. You can delete the SQLite database file between runs and let the OrgChart application re-create it, or you can let multiple runs accumulate objects in the database. If you, for example, add a new person (Jack) as Jane's subordinate, then the OrgChart application's output would look like the following:

```

2011-07-30 22:22:21.858 OrgChart[4517:f203] Organization: MyCompany, Inc.
2011-07-30 22:22:21.860 OrgChart[4517:f203]   Name: John
2011-07-30 22:22:21.861 OrgChart[4517:f203]   Name: Bill
2011-07-30 22:22:21.861 OrgChart[4517:f203]   Name: Jane
2011-07-30 22:22:21.862 OrgChart[4517:f203]   Name: Jack

```

Jack works for Jane. Both Jane and Bill work for John. John is the company's leader.

## Summary

By now the Core Data framework should look a lot less overwhelming to you. In this chapter, you explored the main classes involved in efficiently managing the persistence of entire object graphs without having to expose the user to complex graph algorithms and grueling C APIs in order to interact with certain databases. Using Core Data is comparable to driving a car without having to be a mechanic. The framework is able to represent object models and manage data object graphs with just about a dozen classes. The simplicity of the class structure is misleading. You have seen the power of Core Data—and this is only a beginning. As you dig deeper into the many configuration options, UI bindings, alternate persistent stores, and custom managed object classes, you will also see the elegance of Core Data.

# Storing Data: SQLite and Other Options

Chapter 2 explains the classes that comprise the Core Data framework and the interdependencies and precision with which they work together. The following chapters explain and demonstrate the flexibility of the framework to get the desired results, but the framework flexes only so far. The framework imposes a structure, order, and rigidity for putting data into and taking data out of a persistent store. You must work within that order and structure, doing things the Core Data way, to store and retrieve data reliably. Core Data's shackles fall off, however, when defining what the persistent store looks like or how it works. Though people tend to think that data "belongs" in a database and Apple both provides and defaults to a SQLite database for Core Data's persistent store, you have other options; your data can rest in whatever form you want. For most cases, you'll probably be happy with the default SQLite database, but this chapter discusses other options and where they may be useful.

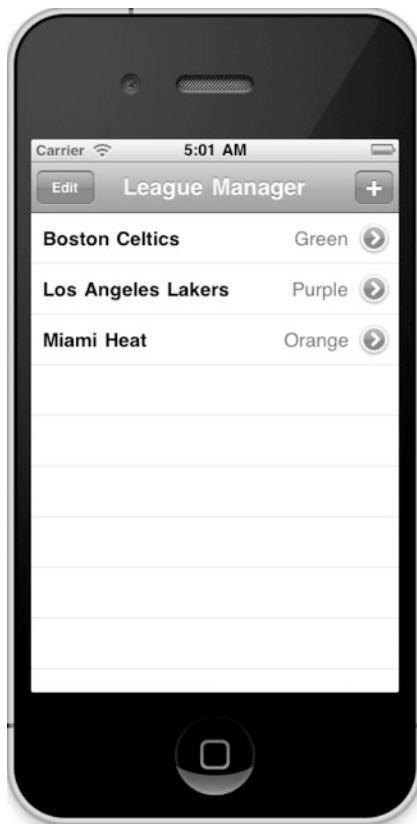
In this chapter, you will build a simple application with two tables and a one-to-many relationship between them. Imagine you've volunteered to run a youth soccer league, and you're trying to keep track of the teams. In your data model, you have a `Team` table that tracks the team name and the uniform color and a `Player` table that tracks the first name, last name, and e-mail address for each player. One team has many players, and each player belongs to only one team. You build this application first for SQLite and then port it to each of the other options for your persistent store: in-memory and atomic (or custom) stores. Follow along, and feel free to store your data in SQLite or any other persistent store you can imagine.

## Visualizing the User Interface

The League Manager application contains four screens:

- Team List
- Add/Edit Team
- Player List
- Add/Edit Player

The Team List screen lists all the teams stored in the application along with their jersey colors. It has a + button to add a team and an Edit button to delete a team. The screen looks like Figure 3–1.



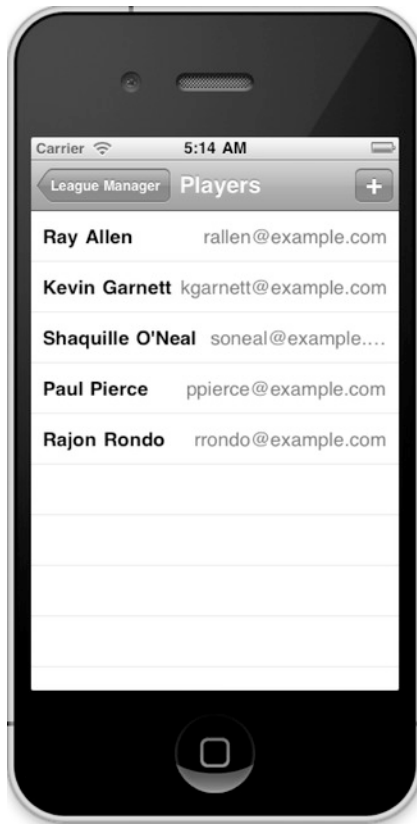
**Figure 3–1.** *The Team List screen*

The Add/Edit Team screen offers fields to type a team name and a jersey color. It looks like Figure 3–2. You display it by clicking the + button from the Team List screen to add a new team or by tapping on an existing team to edit the team name and jersey color for an existing team.



**Figure 3–2.** *The Add/Edit Team screen*

The Player List screen lists all the players for a particular team. You get to it by tapping the detail disclosure button beside any team. It has a + button for adding a new player and an Edit button for deleting a player, as you can see in Figure 3–3.



**Figure 3–3.** *The Player List screen*

Like the Add/Edit Team screen, the Add/Edit Player screen lets users add or edit players, offering the appropriate fields: first name, last name, and e-mail address. It looks like Figure 3–4. You display it by clicking the + button from the Player List screen to add a new player or by tapping on an existing player to edit that player's data.





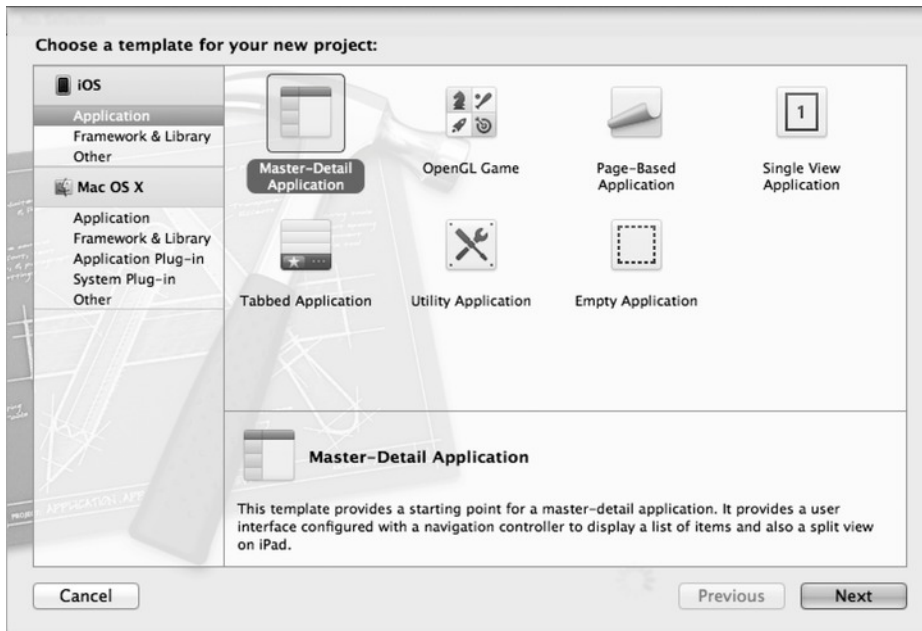
**Figure 3–4.** *The Add/Edit Player screen*

Now that you know what you’re building, let’s get started.

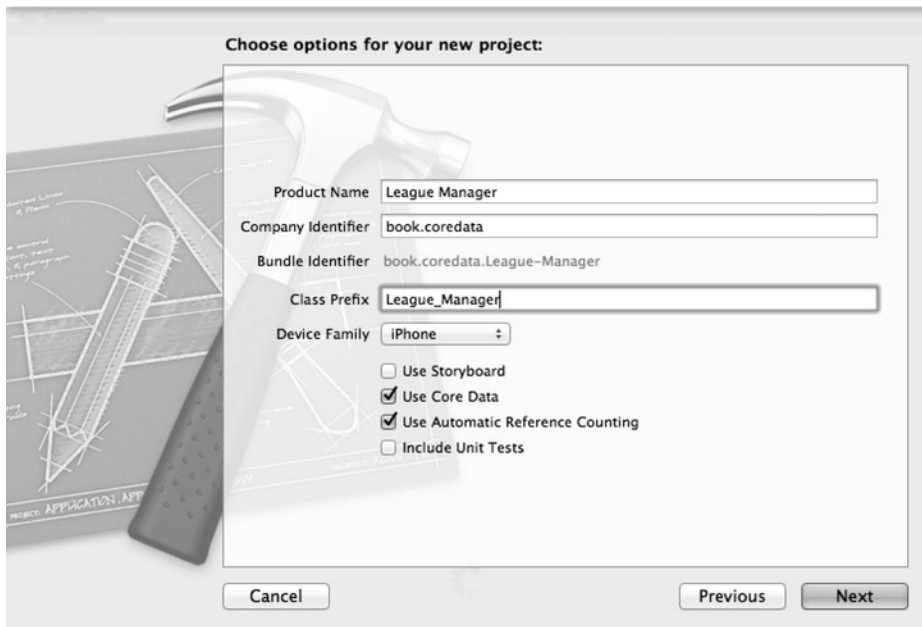
## Using SQLite as the Persistent Store

In this section, you will build the Core Data–based application that you’ll use throughout this chapter to demonstrate different persistent store options. You start from Xcode’s Core Data template and build upon the generated code to create an application that stores a bunch of teams and a bunch of players and allows users to maintain them. Don’t get caught up in the application code or the user interface; instead, focus on the backing persistent store and its many forms.

To begin, launch Xcode, and choose **File > New > New Project**. Select **Application** under **iOS** on the left of the ensuing dialog box, and select **Master-Detail Application** on the right, as shown in Figure 3–5. Click the **Next** button. For the project options, type **League Manager** for the **Product Name** and **book.coredata** for the **Company Identifier**, and check the **Use Core Data** box, as shown in Figure 3–6.



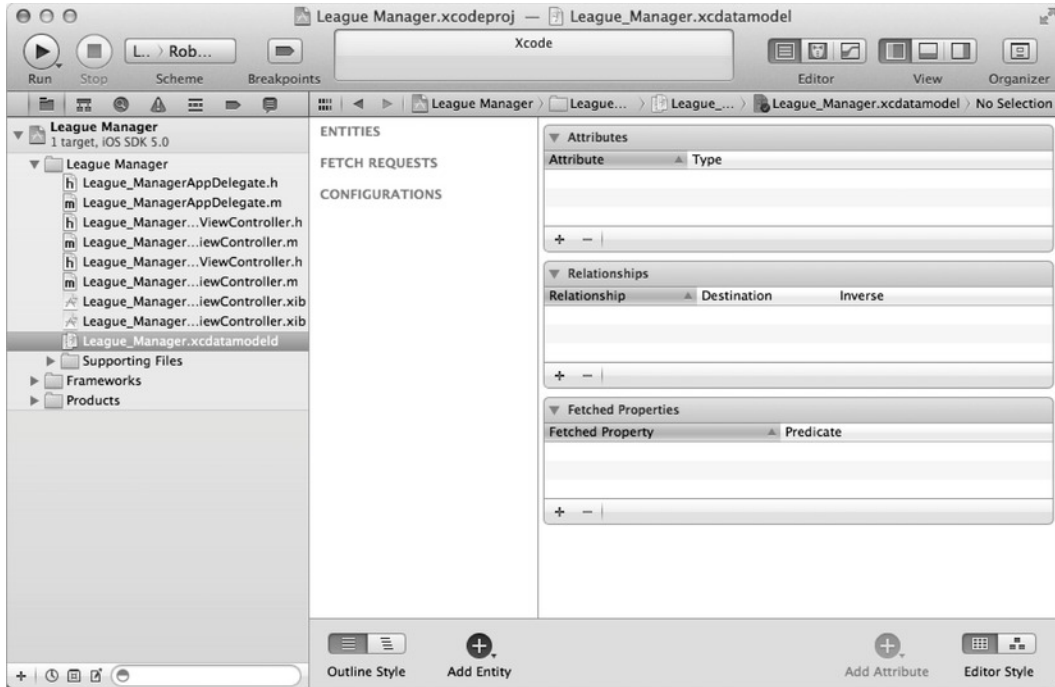
**Figure 3–5.** *Selecting the Master-Detail Application template*



**Figure 3–6.** *Naming the project and selecting to use Core Data*

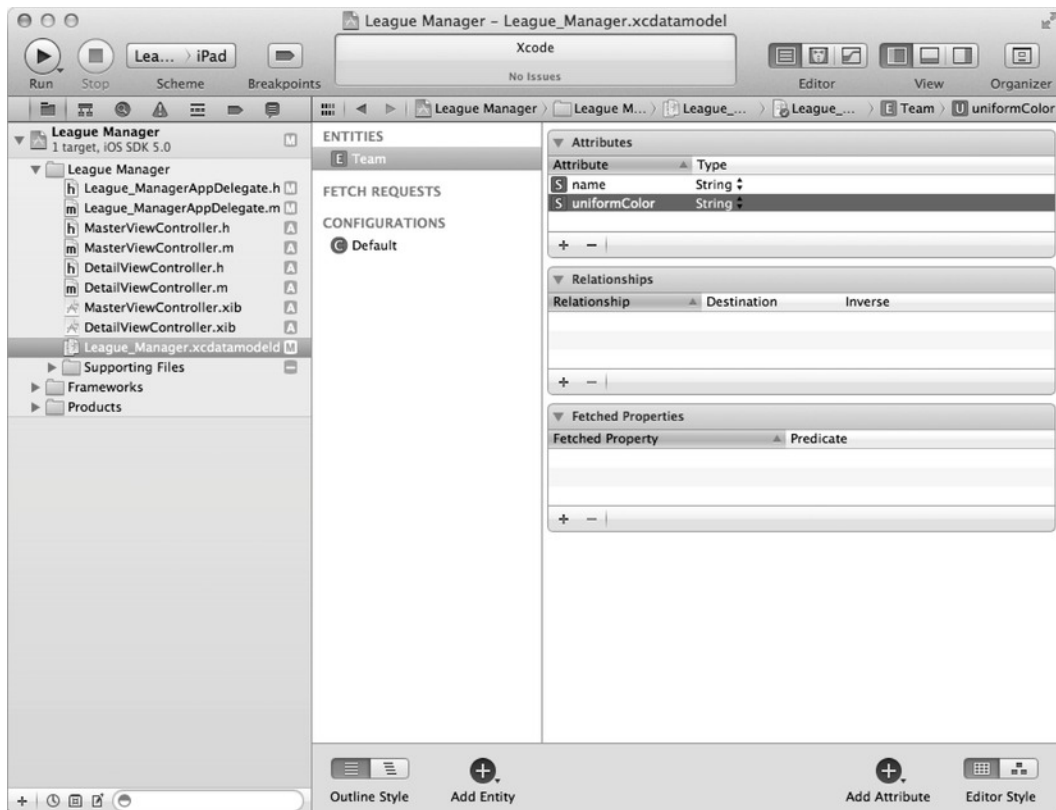
Save the project where you normally save projects.

The most interesting part of the League Manager application, at least for current purposes, is the data model. Understanding persistent stores depends on creating and understanding the proper data model, so step through this part carefully and precisely. Open the data model in Xcode by selecting `League_Manager.xcdatamodeld`. The right side of Xcode displays the generated data model, which has a single entity called `Event` with a single attribute called `timeStamp`. Delete the `Event` entity by selecting it and pressing your Delete key, which should give you an empty data model, as in Figure 3–7.



**Figure 3–7.** Empty data model

The League Manager data model calls for two entities: one to hold teams and one to hold players. For teams, you track two attributes: name and uniform color. For players, you track three: first name, last name, and e-mail address (so you can contact players about practices, games, and the schedule for orange-slice responsibilities). You also track which players play for a given team, as well as which team a given player plays for. Start by creating the Team entity: click the Add Entity button shown at the bottom of Xcode. (It may be labeled Add Fetch Request or Add Configuration; if so, click and hold to select Add Entity from the menu.) Xcode creates a new entity called `Entity` with the name conveniently highlighted so you can type **Team** and press Enter. Now click the Add Attribute button (it may be labeled Add Property or Add Fetched Property; if so, click and hold to select Add Attribute from the menu) and type **name** to name the new attribute. Note that you can also click the + button at the bottom of the Attributes section to add the attribute. Select String from the Type drop-down. Now create the second attribute, following the same steps, but name the attribute **uniformColor**, and select the String type. Your data model should look like Figure 3–8.



**Figure 3–8.** *Team data model*

Before you can create the one-to-many relationship between teams and players, you must have players for teams to relate to. Create another entity called *Player* with three attributes, all of type *String*: *firstName*, *lastName*, and *email*. Your data model should now look like Figure 3–9.

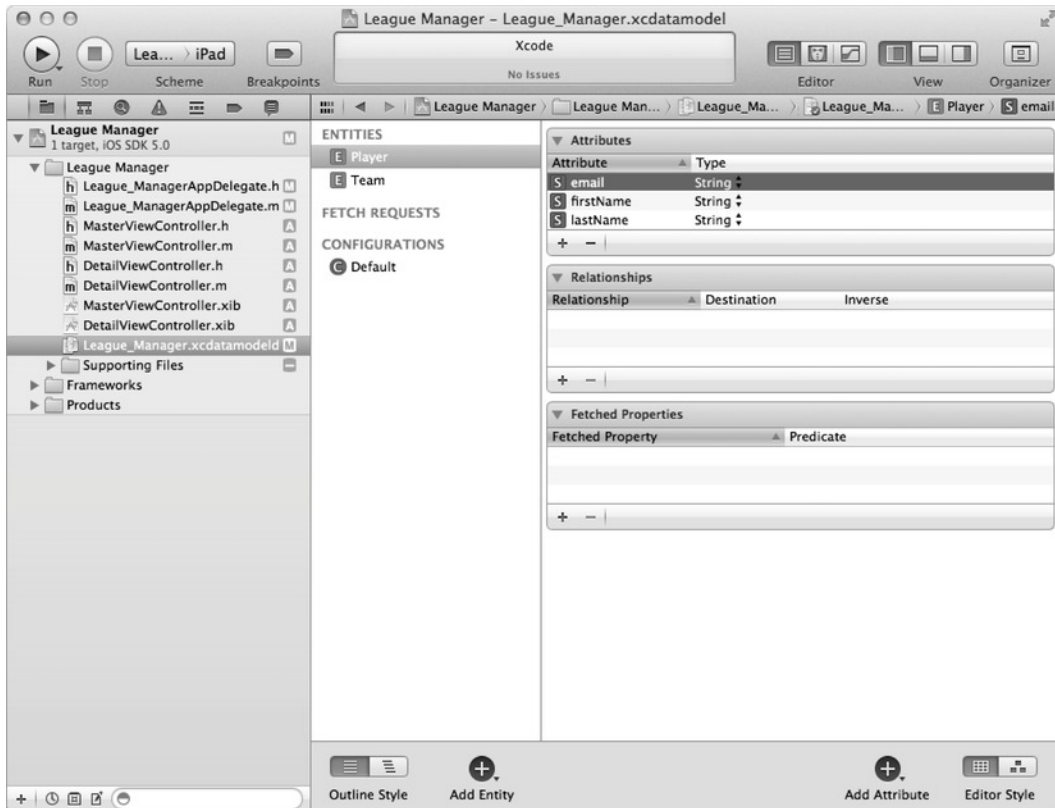
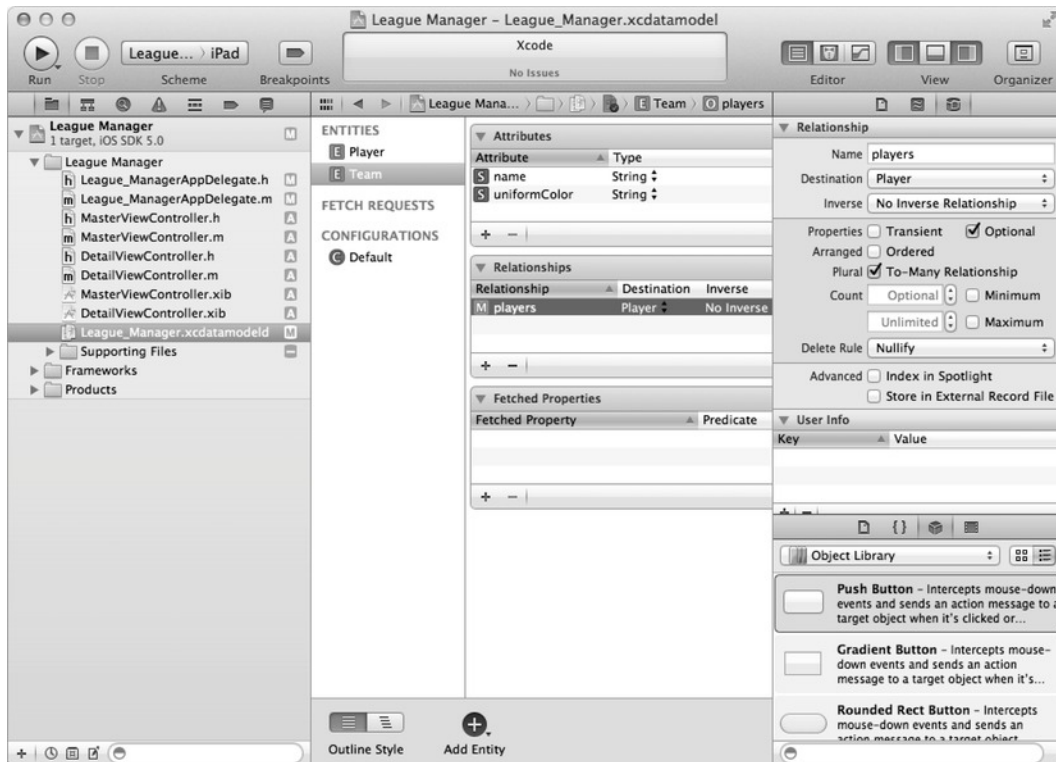


Figure 3–9. *Team and Player data model*

## Configuring the One-to-Many Relationship

To create the one-to-many relationship between the Team entity and the Player entity, select the Team entity, and click the + button below the Relationships section. Name this relationship **players**, and select Player from the Destination drop-down. In the Data Model Inspector view, leave the Optional check box selected. Select the To-Many Relationship check box; one team can have many players. For Delete Rule, select Cascade from the drop-down so that deleting a team deletes all its players. You cannot yet select the inverse relationship in the Inverse drop-down because you haven't created the inverse relationship yet. The Relationship information section should look like Figure 3–10.



**Figure 3-10.** *Players relationship options*

Next, create the relationship back from the `Player` entity to the `Team` entity by selecting the `Player` entity, adding a relationship, and calling it **team**. Select `Team` from the Destination column and “players” from the Inverse column. Leave the relationship options as the default. You should now be able to select the `Team` entity and verify that the “players” relationship has an inverse: “team.” Your Xcode window should look like Figure 3-11.

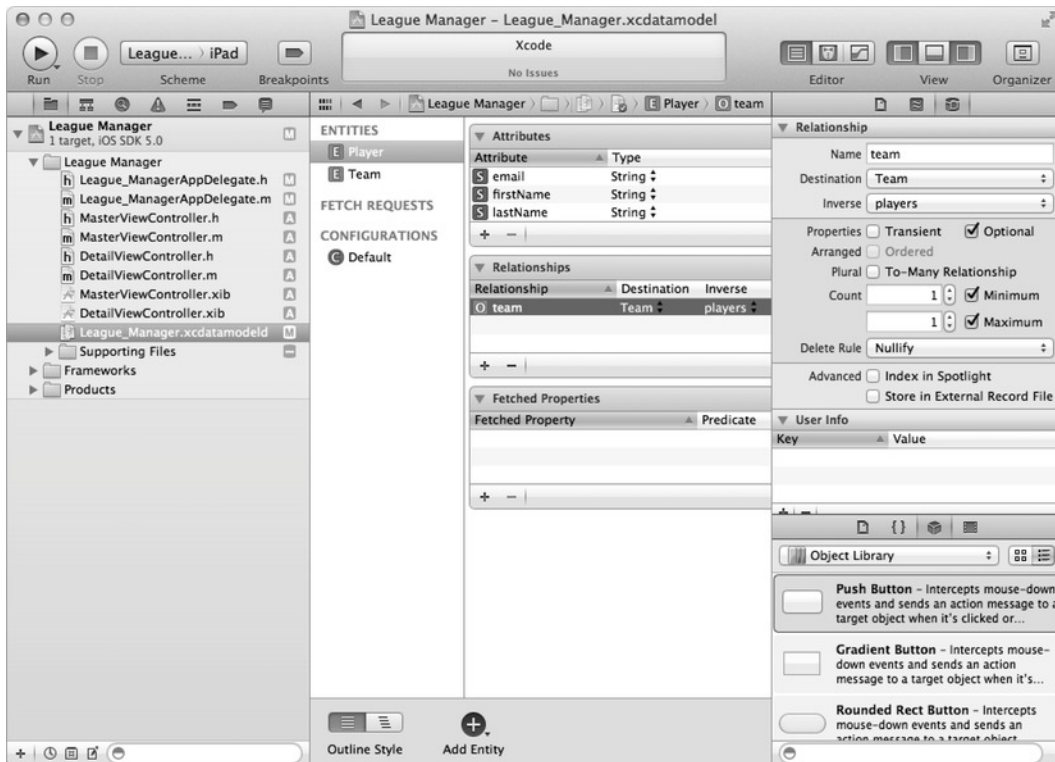


Figure 3-11. The complete League Manager data model

## Building the User Interface

With the data model complete, League Manager now requires code to display the data. Xcode generated most of the code necessary to display the list of teams; the next task is to tweak the code, since it's prepared to show Event entities and the Event entity no longer exists in the data model. This code resides in `MasterViewController.h` and `MasterViewController.m`, so start by opening the `MasterViewController.h` file. Notice that it has two members: an `NSManagedObjectContext` instance and an `NSFetchedResultsController` instance. You could move the `NSManagedObjectContext` instance, which you recognize as the object context for your application, to your application's delegate (`League_ManagerAppDelegate`), but you'll leave it here for this simple application. The `NSFetchedResultsController` instance works with the table view to show your teams.

You need a method to add a team, so add one called `insertTeamWithName:` and declare it in `MasterViewController.h`. Also, the generated code saves the context in a couple of places, so adhere to the Don't Repeat Yourself (DRY) principle and move it all to one method called `saveContext:`. The code for `MasterViewController.h` now looks like this, with the new method declarations in bold:

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

@interface MasterViewController : UITableViewController
<NSFetchResultsControllerDelegate>

@property (strong, nonatomic) NSFetchResultsController *fetchResultsController;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;

- (void)insertTeamWithName:(NSString *)name uniformColor:(NSString *)uniformColor;
- (void)saveContext;

@end
```

Open the MasterViewController.m file, and adjust the view title in the initWithNibName:bundle: method that sets the title for the application. That line looks like this:

```
self.title = NSLocalizedString(@"League Manager", @"League Manager");
```

Now define the insertTeamWithName: method in MasterViewController.m. That method looks like this:

```
- (void)insertTeamWithName:(NSString *)name uniformColor:(NSString *)uniformColor {
    // Create a new instance of the entity managed by the fetched results controller.
    NSManagedObjectContext *context = [self.fetchResultsController
managedObjectContext];
    NSEntityDescription *entity = [[self.fetchResultsController fetchRequest] entity];
    NSManagedObject *newManagedObject = [NSEntityDescription
insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];

    // Configure the new team
    [newManagedObject setValue:name forKey:@"name"];
    [newManagedObject setValue:uniformColor forKey:@"uniformColor"];

    // Save the context
    [self saveContext];
}
```

This code gets the managed object context from the application's fetchResultsController and then inserts a new NSManagedObject instance into that managed object context. Notice that it doesn't specify that the new entity you're trying to insert is named Team. The name gets defined in the accessor for fetchResultsController. The generated code used the name Event, so find the line in fetchResultsController that looks like this

```
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
inManagedObjectContext:self.managedObjectContext];
```

and change it to this

```
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Team"
inManagedObjectContext:self.managedObjectContext];
```



Also, you'll notice another vestige of the generated model in the `fetchResultsController` method: a reference to the attribute called `timeStamp`, used for sorting the fetched results (Chapter 6 discusses sorting and how it works). Change to sort on the team name, ascending, so that this line

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"timeStamp"
ascending:NO];
```

now looks like this

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

After creating the managed object representing the new team, the code in the `insertTeamWithName:` method sets its name and uniform color using the parameters passed:

```
[newManagedObject setValue:name forKey:@"name"];
[newManagedObject setValue:uniformColor forKey:@"uniformColor"];
```

Finally, the `insertTeamWithName:` method saves the object graph, including the new team, by calling the `saveContext:` method that you declared but haven't yet defined. You define it by cutting and pasting that bit of code from the now-superfluous `insertNewObject:` method that Xcode generated. After snipping that bit, delete the `insertNewObject:` method and define `saveContext:` like this:

```
- (void)saveContext {
    NSManagedObjectContext *context = [self.fetchResultsController
managedObjectContext];
    NSError *error = nil;
    if (![context save:&error]) {
        /*
            Replace this implementation with code to handle the error appropriately.

            abort() causes the application to generate a crash log and terminate. You should
            not use this function in a shipping application, although it may be useful during
            development. If it is not possible to recover from the error, display an alert panel
            that instructs the user to quit the application by pressing the Home button.
        */
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }
}
```

Leave the generated comment there to remind you that calling `abort` represents a decidedly un-user-friendly way to handle errors. Chapter 10 talks about appropriate error handling. Since you now have a method that you can reuse to save the context, replace the other instance of that code found in the `commitEditingStyle:` method, with a call to the new `saveContext:` method.

## Configuring the Table

The table cells are still configured to show Event entities instead of Team entities. You want to show two pieces of information for a team in each table cell: the team's name and its uniform color. To accomplish this, first change the style of the cells created, as well as the CellIdentifier used, in the cellForRowAtIndexPath: method. Change this line

```
static NSString *CellIdentifier = @"Cell";
```

to this

```
static NSString *CellIdentifier = @"TeamCell";
```

and change the created table cells from style UITableViewCellStyleDefault to style UITableViewCellStyleValue1 so that this line

```
cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault↵
reuseIdentifier:CellIdentifier];
```

looks like this

```
cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1↵
reuseIdentifier:CellIdentifier];
```

The generated code has created a method called configureCell:atIndexPath: that's responsible for, well, configuring the cell. Change that method from configuring for Event entities to configuring for Team entities. You also want to be able to drill down from the team to see its players, so you add a detail disclosure button to each cell. The generated method looks like this:

```
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *managedObject = [self.fetchedResultsController↵
objectAtIndexPath:indexPath];
    cell.textLabel.text = [[managedObject valueForKey:@"timeStamp"] description];
}
```

Change it to look like this:

```
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *managedObject = [self.fetchedResultsController↵
objectAtIndexPath:indexPath];
    cell.textLabel.text = [[managedObject valueForKey:@"name"] description];
    cell.detailTextLabel.text = [[managedObject valueForKey:@"uniformColor"] description];
    cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;
}
```

## Creating a Team

The application doesn't do much yet. For example, you can't add a team or any players. This is a good time, though, to compile and run the application to make sure you're on track. If the application doesn't build or run at this point, go back and review your data model and your code to make sure it matches the code shown previously before proceeding.

The Master-Detail application template also creates a controller called `DetailViewController`. You could reuse this class, but since we want to display both team and players details, it is best to get rid of it and name the controllers appropriately. Remove the `#import "DetailViewController.h"` line at the top of `MasterViewController.m`. Find the `tableView:didSelectRowAtIndexPath:` method and empty it as shown below:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
}
```

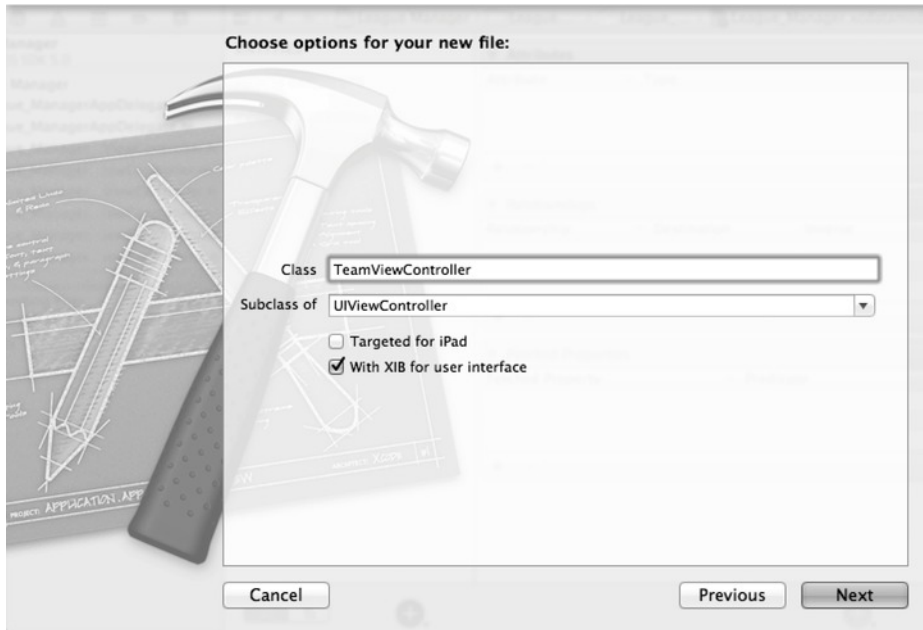
You can then safely delete the three `DetailViewController` files (.h, .m and .xib).

If you run the application and tap the + button to add a team, you notice that the application crashes. The + button is still wired to the `insertNewObject:` method that you deleted. You need to wire it to a method that will allow you to create a new team. The design for creating a new team calls for a modal window that allows you to enter the name and uniform color for the team. You reuse this modal window for editing an existing team as well, which users can do by tapping the team's cell. Create this modal window by selecting **File > New > New File** from the Xcode menu, and select **Cocoa Touch Class** under **iOS** on the left and **UIViewController subclass** on the right, as Figure 3-12 shows, and click the **Next** button.



**Figure 3-12.** Adding a view controller for the new team

Name the class `TeamViewController`, make it a subclass of `UIViewController`, and check the "With XIB for user interface" box as Figure 3-13 shows, then click **Next**.



**Figure 3–13.** Choosing options for the view controller for the new team

Open the new header file (`TeamViewController.h`). In League Manager, the `MasterViewController` class controls the managed object context, so `TeamViewController` needs a reference to it and an accompanying initializer. Since you can use this controller to edit a team as well as create a new one, you allow calling code to pass a team object to edit, and you store a property for that and add it to the initializer. The user interface has two text fields, one for the team name and one for the uniform color, so `TeamViewController` needs properties for those fields. The user interface also has two buttons, Save and Cancel, so `TeamViewController` must have methods to wire to those buttons. Add all that up, and you get the `TeamViewController.h` shown in Listing 3–1.

**Listing 3–1.** *TeamViewController.h*

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

@class MasterViewController;

@interface TeamViewController : UIViewController {
    IBOutlet UITextField *name;
    IBOutlet UITextField *uniformColor;
    NSManagedObject *team;
    MasterViewController *masterController;
}
@property (nonatomic, retain) UITextField *name;
@property (nonatomic, retain) UITextField *uniformColor;
@property (nonatomic, retain) NSManagedObject *team;
@property (nonatomic, retain) MasterViewController *masterController;
```

```
- (IBAction)save:(id)sender;
- (IBAction)cancel:(id)sender;
- (id)initWithMasterController:(MasterViewController *)aMasterController team: ↵
(NSManagedObject *)aTeam;
```

@end

Now open TeamViewController.m; import MasterViewController.h; delete the initWithNibName: method; add @synthesize lines for name, uniformColor, team, and masterController. Next, add a definition for the initWithMasterController: method that looks like this:

```
- (id)initWithMasterController:(MasterViewController *)aMasterController team: ↵
(NSManagedObject *)aTeam {
    if ((self = [super init])) {
        self.masterController = aMasterController;
        self.team = aTeam;
    }
    return self;
}
```

In the case in which users add a new team, the aTeam parameter will be nil, and the TeamViewController.m will own the responsibility to create it. In the case in which users edit an existing team, however, you must take the existing team's attribute values and put them into the appropriate text fields. Do that in the viewDidLoad: method, like this:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    if (team != nil) {
        name.text = [team valueForKey:@"name"];
        uniformColor.text = [team valueForKey:@"uniformColor"];
    }
}
```

Finally, implement the save: and cancel: methods so that this controller can respond appropriately to when the user taps the Save or Cancel buttons. The save: method checks for a non-nil masterController instance and then determines whether to create a new team or edit an existing team by checking whether its team member is nil. If it's not nil, it updates the values for the existing team and asks the masterController member to save its context. If it is nil, it asks the MasterViewController instance to create a new team in the managed object context, passing the user-entered values for team name and uniform color. Finally, it dismisses itself. The method implementation looks like this:

```
- (IBAction)save:(id)sender {
    if (masterController != nil) {
        if (team != nil) {
            [team setValue:name.text forKey:@"name"];
            [team setValue:uniformColor.text forKey:@"uniformColor"];
            [masterController saveContext];
        } else {
            [masterController insertTeamWithName:name.text uniformColor:uniformColor.text];
        }
    }
}
```

```
    [self dismissModalViewControllerAnimated:YES];
}
```

The `cancel:` method simply dismisses itself. The entire file should look like Listing 3–2.

**Listing 3–2.** *TeamViewController.m*

```
#import "TeamViewController.h"
#import "MasterViewController.h"

@implementation TeamViewController

@synthesize name;
@synthesize uniformColor;
@synthesize team;
@synthesize masterController;

- (id)initWithMasterController:(MasterViewController *)aMasterController team: ~
(NSManagedObject *)aTeam {
    if ((self = [super init])) {
        self.masterController = aMasterController;
        self.team = aTeam;
    }
    return self;
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

#pragma mark - View lifecycle

- (void)viewDidLoad {
    [super viewDidLoad];
    if (team != nil) {
        name.text = [team valueForKey:@"name"];
        uniformColor.text = [team valueForKey:@"uniformColor"];
    }
}

- (void)viewDidUnload {
    [super viewDidUnload];
}

-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientatio
n {
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

#pragma mark - Button handlers

- (IBAction)save:(id)sender {
    if (masterController != nil) {
        if (team != nil) {
            [team setValue:name.text forKey:@"name"];
        }
    }
}
```

```

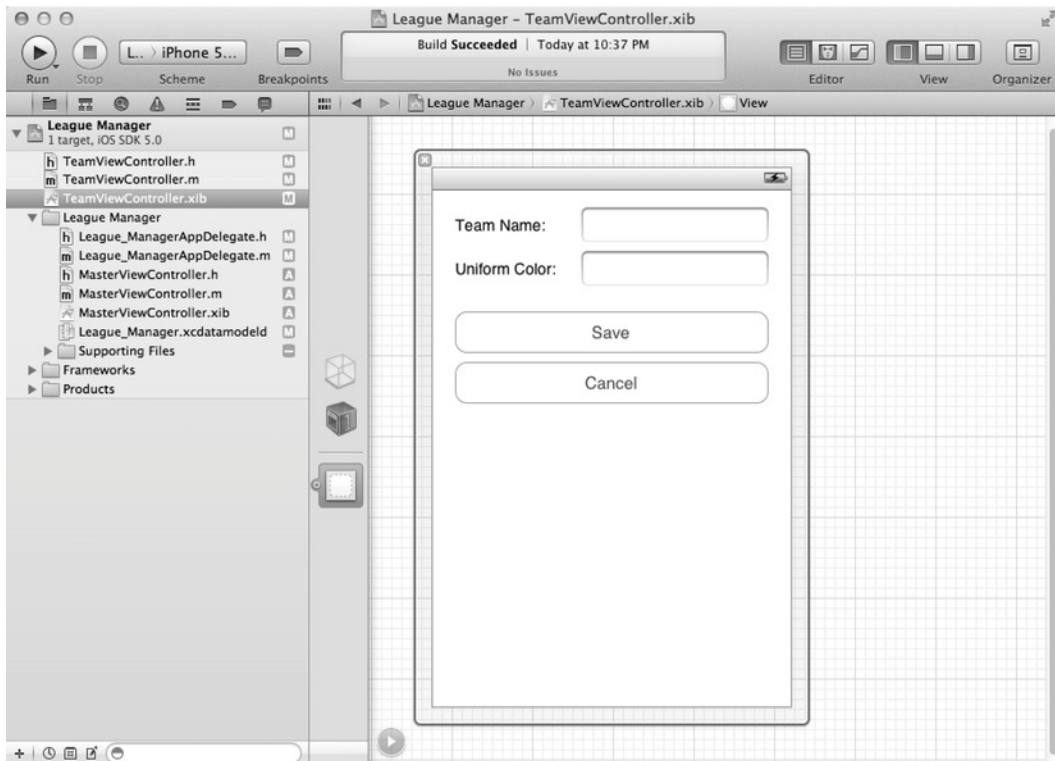
        [team setValue:uniformColor.text forKey:@"uniformColor"];
        [masterController saveContext];
    } else {
        [masterController insertTeamWithName:name.text uniformColor:uniformColor.text];
    }
}
[self dismissModalViewControllerAnimated:YES];
}

- (IBAction)cancel:(id)sender {
    [self dismissModalViewControllerAnimated:YES];
}

@end

```

With the code written to support the user interface, you're ready to build the labels, text fields, and buttons the users will interact with to create teams. Open `TeamViewController.xib` in Xcode. You should see a blank view. Drag two `UILabel` instances onto the view, and change them to read **Team Name:** and **Uniform Color:**. Drag two `UITextField` instances onto the view, and align them to the right of the Labels. Drag two `UIButton` instances below the Label and Text Field controls, and change the labels on them to **Save** and **Cancel**. Your view should look like Figure 3-14.



**Figure 3-14.** Updated team view

Bind the Text Field instances to the appropriate TeamViewController members, name and uniformColor, by Ctrl+dragging from the File's Owner icon to the respective Text Field instances and selecting name or uniformColor from the pop-up menu as appropriate. Wire the buttons to the save: and cancel: methods by Ctrl+dragging from each button, in turn, to the File's Owner icon and selecting the appropriate method from the pop-up menu.

Before building and running the application, you must go back to MasterViewController and include code to display the team interface you just built. You display it in two scenarios: when users tap the + button to create a new team and when they tap the team in the table to edit it. Start by creating the method to respond to the + button tap. Declare a method called showTeamView: in MasterViewController.h:, like so:

```
- (void)showTeamView;
```

Go to MasterViewController.m, import TeamViewController.h, and add the definition for the showTeamView: method. This method creates a TeamViewController instance, initializing it with the MasterViewController instance and a nil team, so that the TeamViewController knows to create a new team if the user taps Save. The method should look like this:

```
- (void)showTeamView {
    TeamViewController *teamViewController = [[TeamViewController alloc]
initWithMasterController:self team:nil];
    [self presentViewController:teamViewController animated:YES];
}
```

Now you need to wire the + button to call this method. Go to the viewDidLoad: method, and change this line

```
UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(insertNewObject)];
```

to this

```
UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(showTeamView)];
```

The application should now be able to add teams, but before testing that, add the code to edit teams. This code should determine the tapped team by asking the fetchedResultsController which team was tapped, which it will determine using the indexPath passed to the didSelectRowAtIndexPath: method. This code then creates a TeamViewController instance and initializes it with the MasterViewController and the tapped team. Find the didSelectRowAtIndexPath: method, and add the code to edit the tapped team, like this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
    NSManagedObject *team = [[self fetchedResultsController] objectAtIndexPath:indexPath];
    TeamViewController *teamViewController = [[TeamViewController alloc]
initWithMasterController:self team:team];
    [self presentViewController:teamViewController animated:YES];
}
```



Build the application and run it. The application looks like it did the last time you ran it, as Figure 3–15 shows.



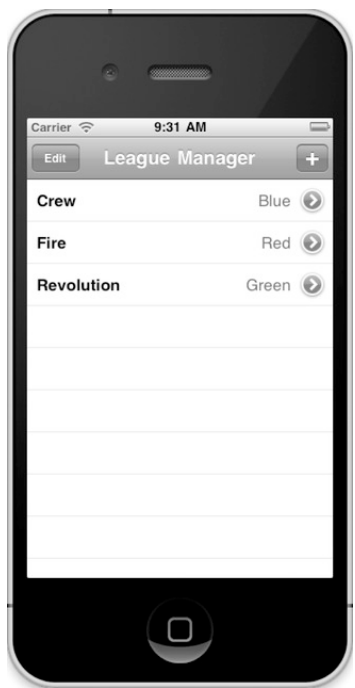
**Figure 3–15.** *League Manager without any teams*

Now, however, if you tap the + button, you see the screen to create a new team, as Figure 3–16 shows.

Go ahead and add a few teams, edit some teams, and delete some teams. You can close and relaunch the application, and you'll find the teams as they were when you quit the application—they're all being added to your SQLite data store. You will notice that the teams are sorted alphabetically by team name. Figure 3–17 shows some sample teams, with name and uniform color.



**Figure 3–16.** Adding a new team to *League Manager*



**Figure 3–17.** *League Manager* with teams

You might notice that you can create teams with blank names and uniform colors. If this were a real application, you would take steps to prevent users from creating teams with no name and perhaps with no uniform color. This chapter remains focused on the different persistent store options, however, so you've purposely left out any validation code. You'll notice that the player user interface created later in this chapter has no validation code either, so you can create blank players. Chapter 5 talks about how to validate data.

Quit the application and enjoy what you've accomplished thus far... and then realize that you've covered only the Team entity. You must still implement the Player user interface and entity to call the League Manager application complete!

## The Player User Interface

To implement the Player user interface, you must have two views and their accompanying controllers: one to list the players for a team and one to add a new or edit an existing player. These controllers largely mirror the ones for Team, although they don't contain an `NSFetchedResultsController` or the rest of the Core Data classes that `MasterViewController` does. Instead, they delegate the Core Data interaction to `MasterViewController`.

Create the controller and view to list players for a team first. Add a new `UIViewController` subclass, making sure to select `UITableViewController` in the "Subclass of" drop-down and deselect "With XIB for user interface." Call it **PlayerListViewController**, and then turn your attention to the file `PlayerListViewController.h`. This class lists players for a team, so it needs a reference to Team entity for which it manages players. Also, since it defers Core Data interaction to the `MasterViewController` class, it requires a reference to the `MasterViewController`. This controller will have a + button for adding a new player, so declare a method to respond to taps on that button called `showPlayerView:`. Finally, since it doesn't use an `NSFetchedResultsController` instance to sort the players, it must implement sorting for the players. To accomplish all this, you end up with a `PlayerListViewController.h` file that looks like Listing 3–3.

**Listing 3–3.** *PlayerListViewController.h*

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

@class MasterViewController;

@interface PlayerListViewController : UITableViewController {
    NSManagedObject *team;
    MasterViewController *masterController;
}
@property (nonatomic, retain) NSManagedObject *team;
@property (nonatomic, retain) MasterViewController *masterController;

- (id)initWithMasterController:(MasterViewController *)aMasterController team:(NSManagedObject *)aTeam;
- (void)showPlayerView;
```

```
- (NSArray *)sortPlayers;
```

```
@end
```

You'll recognize pieces of `MasterViewController.m` in the `PlayerListViewController.m` file. Open the file, add an import for `MasterViewController.h`, and @synthesize lines for the `team` and `masterController` properties. Change the generated `initWithStyle:` method to an `initWithMasterController:` method that accepts those two properties and stores them that looks like this:

```
- (id)initWithMasterController:(MasterViewController *)aMasterController team:~
(NSManagedObject *)aTeam {
    if ((self = [super init])) {
        self.masterController = aMasterController;
        self.team = aTeam;
    }
    return self;
}
```

Xcode generated a `viewDidLoad:` method for you. Add the following code to update the view title appropriately and to display a + button to add a player to the team. Since you haven't yet begun building the user interface for adding or editing a player, wire the + to a new method called `showPlayerView:` that you leave blank for now. Those two methods look like this:

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

    self.title = @"Players";

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]~
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self~
action:@selector(showPlayerView)];
    self.navigationItem.rightBarButtonItem = addButton;
}

- (void)showPlayerView {
}
```

In the `viewWillAppear:` method, instruct the controller's table view to reload its data, like this:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

If you want the table view to reload its data, you must tell the table view what data to load and display. The player list will display all the players for a team, sorted alphabetically, in a single section. To get the players for a team, call the team's `valueForKey:@"players"` method, which uses the "players" relationship from the data model to pull all the `Player` entities from the SQLite persistent store and returns them as an `NSSet`. The code to set up the single section and the number of rows for the table looks like this:

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [(NSSet *)[team valueForKey:@"players"] count];
}

```

For the table cells, again use the `UITableViewCellStyleValue1` to display text on the left (the first and last names of the player) and text on the right (the e-mail address). Change the generated `cellForRowAtIndexPath:` to look like this:

```

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"PlayerCell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1
reuseIdentifier:CellIdentifier];
    }

    NSManagedObject *player = [[self sortPlayers] objectAtIndex:indexPath.row];
    cell.textLabel.text = [NSString stringWithFormat:@"%s %@",
valueForKey:@"firstName" description], [[player valueForKey:@"lastName" description]];
    cell.detailTextLabel.text = [[player valueForKey:@"email" description]];
    return cell;
}

```

Here you see a call to `sortPlayers:`. Recall that the “players” relationship on the “team” instance returns an `NSSet`, which not only has no sorting but also isn’t indexable, because it has no deterministic order. The `cellForRowAtIndexPath:` method demands a cell for a specific index into the backing data, so no `NSSet` method can perform the task you need here: to return the appropriate cell for the table at this index path. Instead, you convert the `NSSet` to an `NSArray` sorted by players’ last names using this `sortPlayers:` method, like so:

```

- (NSArray *)sortPlayers {
    NSSortDescriptor *sortLastNameDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"lastName" ascending:YES];
    NSArray *sortDescriptors = [NSArray arrayWithObjects:sortLastNameDescriptor, nil];
    return [(NSSet *)[team valueForKey:@"players"] allObjects]
sortedArrayUsingDescriptors:sortDescriptors;
}

```

To display the player list view for a team, go back to `MasterViewController.m`, and add a method to respond to taps on the detail disclosure buttons for teams. The method to implement is called `accessoryButtonTappedForRowWithIndexPath:`, and in this method you retrieve the tapped team from the fetched results controller, create a `PlayerListViewController` instance, initialize it with the master view controller and the tapped team, and show the controller. The code looks like this:

```

- (void)tableView:(UITableView *)tableView
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {

```

```
NSManagedObject *team = [self.fetchedResultsController objectAtIndex:indexPath:indexPath];
PlayerListViewController *playerListViewController = [[PlayerListViewController
alloc] initWithMasterController:self team:team];
[self.navigationController pushViewController:playerListViewController animated:YES];
}
```

Add an import for `PlayerListViewController.h` to the top of `MasterViewController.m`, and you're ready to build and launch League Manager anew. You should see the teams you created before, but now when you tap the detail disclosure button for a team, you move to the Players list view, as in Figure 3–18. Since you as yet have no way to add players, the list is blank, and the + button does nothing.



**Figure 3–18.** Team with no players

## Adding, Editing, and Deleting Players

The League Manager application is nearly complete; it lacks means only for adding, editing, and deleting players. To accomplish these tasks, create a new `UIViewController` subclass, select `UIViewController` in the “Subclass of” dropdown, and select “With XIB for user interface.” Call this controller **PlayerViewController**. It looks similar to the `TeamViewController` class and interface but has three fields:

firstName, lastName, and email. It also has a reference to the MasterViewController instance so it can defer all Core Data storage and retrieval to that class. It has a member for the team this player belongs to, and it also has a reference to the player. If the player is nil, PlayerViewController knows to create a new player. Otherwise, it knows to edit the existing player object. The user interface has three buttons: one to save the player, one to cancel the operation (add or edit), and one to delete the player. Because you want to display a confirmation action sheet before actually deleting a player, you make PlayerViewController implement the `UIActionSheetDelegate` protocol. See Listing 3–4 for what `PlayerViewController.h` should look like.

**Listing 3–4. *PlayerViewController.h***

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

@class MasterViewController;

@interface PlayerViewController : UIViewController <UIActionSheetDelegate> {
    IBOutlet UITextField *firstName;
    IBOutlet UITextField *lastName;
    IBOutlet UITextField *email;
    NSManagedObject *team;
    NSManagedObject *player;
    MasterViewController *masterController;
}
@property (nonatomic, retain) UITextField *firstName;
@property (nonatomic, retain) UITextField *lastName;
@property (nonatomic, retain) UITextField *email;
@property (nonatomic, retain) NSManagedObject *team;
@property (nonatomic, retain) NSManagedObject *player;
@property (nonatomic, retain) MasterViewController *masterController;

- (IBAction)save:(id)sender;
- (IBAction)cancel:(id)sender;
- (IBAction)confirmDelete:(id)sender;
- (id)initWithMasterController:(MasterViewController *)aMasterController team:~
(NSManagedObject *)aTeam player:(NSManagedObject *)aPlayer;

@end
```

Now, open the `PlayerViewController.m` file, import `MasterViewController.h`, and add `@synthesize` lines for the various properties. Add the `initWithMasterController:` method declared in `PlayerViewController.h` to initialize this view controller with a `MasterViewController` instance, a team, and a possibly-nil player. That method looks like this:

```
- (id)initWithMasterController:(MasterViewController *)aMasterController team:~
(NSManagedObject *)aTeam player:(NSManagedObject *)aPlayer {
    if ((self = [super init])) {
        self.masterController = aMasterController;
        self.team = aTeam;
        self.player = aPlayer;
    }
    return self;
}
```

Use the `viewDidLoad:` method to take the values from the player managed object, if non-nil, and put them in the `firstName`, `lastName`, and `email` fields. That method looks like this:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    if (player != nil) {
        firstName.text = [player valueForKey:@"firstName"];
        lastName.text = [player valueForKey:@"lastName"];
        email.text = [player valueForKey:@"email"];
    }
}
```

Your next step is to add methods to respond to the three buttons. The `save:` and `cancel:` methods mirror the ones created for the `TeamViewController` class; they look like this:

```
- (IBAction)save:(id)sender {
    if (masterController != nil) {
        if (player != nil) {
            [player setValue:firstName.text forKey:@"firstName"];
            [player setValue:lastName.text forKey:@"lastName"];
            [player setValue:email.text forKey:@"email"];
            [masterController saveContext];
        } else {
            [masterController insertPlayerWithTeam:team firstName:firstName.text↵
lastName:lastName.text email:email.text];
        }
    }
    [self dismissModalViewControllerAnimated:YES];
}

- (IBAction)cancel:(id)sender {
    [self dismissModalViewControllerAnimated:YES];
}
```

The `insertPlayerWithTeam:` method in `MasterViewController` doesn't yet exist, so you'll create that in a moment. First, though, implement the `confirmDelete:` method for the Delete button in the user interface to call. This method doesn't delete the player right away but instead presents an action sheet requesting users to confirm their intentions. The implementation here first checks whether the player is not nil. In other words, you can delete only existing players. You really should show the Delete button only when editing a player, but in the interest of maintaining focus on Core Data, keep things simple and ignore Delete button presses when adding a player. The `confirmDelete:` method looks like this:

```
- (IBAction)confirmDelete:(id)sender {
    if (player != nil) {
        UIAlertController *confirm = [[UIAlertSheet alloc] initWithTitle:nil delegate:self↵
cancelButtonTitle:@"Cancel" destructiveButtonTitle:@"Delete Player"↵
otherButtonTitles:nil];
        confirm.actionSheetStyle = UIAlertControllerStyleBlackTranslucent;
        [confirm showInView:self.view];
    }
}
```



Note that you pass `self` as the delegate to the `UIActionSheet`'s initialization method. The Cocoa framework will call the `clickedButtonAtIndex:` method of the delegate you pass, so implement that method. It checks to see whether the clicked button was the Delete button and then asks the master view controller to delete the player using a method, `deletePlayer:`, that you must create in `MasterViewController`. The `clickedButtonAtIndex:` method looks like this:

```
- (void)actionSheet:(UIActionSheet *)actionSheet↵
clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (buttonIndex == 0 && masterController != nil) {
        // The Delete button was clicked
        [masterController deletePlayer:player];
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

Now, move back to `MasterViewController.h` and declare the two methods, `insertPlayerWithTeam:` and `deletePlayer:`, that `PlayerViewController` calls. Those declarations look like this:

```
- (void)insertPlayerWithTeam:(NSManagedObject *)team firstName:(NSString *)firstName↵
lastName:(NSString *)lastName email:(NSString *)email;
- (void)deletePlayer:(NSManagedObject *)player;
```

Open `MasterViewController.m`, and define those two methods. The `insertPlayerWithTeam:` method looks similar to the `insertTeamWithName:` method, with some important differences. The `insertTeamWithName:` method takes advantage of the fetched results controller and its tie to `Team` entities, while `Player` entities have no tie to the fetched results controller. The `insertPlayerWithTeam:` method, then, creates a `Player` entity by explicitly passing the `Player` name to the `insertNewObjectForEntityForName:` method. It also must create the relationship to the appropriate `Team` entity by setting it as the value for the “team” key, which is what the relationship is called in the data model. The `insertPlayerWithTeam:` method looks like this:

```
- (void)insertPlayerWithTeam:(NSManagedObject *)team firstName:(NSString *)firstName
lastName:(NSString *)lastName email:(NSString *)email {
    // Create the player
    NSManagedObjectContext *context = [self.fetchedResultsController↵
managedObjectContext];
    NSManagedObject *player = [NSEntityDescription↵
insertNewObjectForEntityForName:@"Player" inManagedObjectContext:context];
    [player setValue:firstName forKey:@"firstName"];
    [player setValue:lastName forKey:@"lastName"];
    [player setValue:email forKey:@"email"];
    [player setValue:team forKey:@"team"];

    // Save the context.
    [self saveContext];
}
```

The `deletePlayer:` method simply retrieves the managed object context, calls its `deleteObject:` method, passing in the `Player` managed object, and saves the managed object context. It looks like this:

```
- (void)deletePlayer:(NSManagedObject *)player {
    NSManagedObjectContext *context = [self.fetchedResultsController
managedObjectContext];
    [context deleteObject:player];
    [self saveContext];
}
```

The final step is to create the user interface and display it when users want to add or edit a player. Select `PlayerViewController.xib`, select the View icon, and drag three Label instances and three Text Field instances onto the view. Call the labels **First Name:**, **Last Name:**, and **E-mail:**. Connect the text fields to the appropriate properties. Drag three Round Rect Button instances to the view and call them **Save**, **Cancel**, and **Delete**, and wire them to the appropriate methods. Your view should look the one in Figure 3–19.

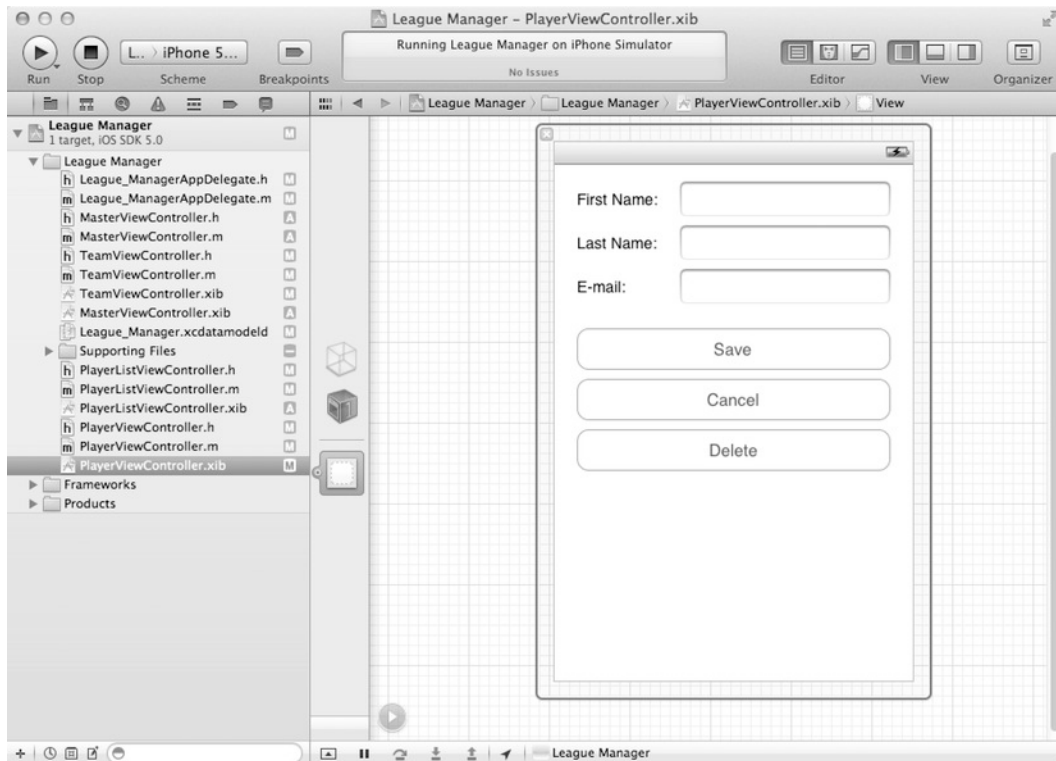


Figure 3–19. Player view

To display the Player view when users summon it, go to `PlayerListViewController.m`, import `PlayerViewController.h`, and find the empty `showPlayerView:` method you created earlier. In that method, you create a `PlayerViewController` instance; initialize it

with the master view controller, the team, and a nil player so that the application will create a new player; and then show the view as a modal window. The code looks like this:

```
- (void)showPlayerView {
    PlayerViewController *playerViewController = [[PlayerViewController alloc]
initWithMasterController:masterController team:team player:nil];
    [self presentViewController:playerViewController animated:YES];
}
```

You also must make the application respond to taps on a player's cell so that users can edit or delete the selected player. Find the generated `didSelectRowAtIndexPath:` method, still in `PlayerListViewController.m`, and gut it. Replace its contents with code to get the tapped player from the sorted players array, create the player view controller, initialize it as before but this time with the selected player, and show the view. The method now looks like this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
    NSManagedObject *player = [[self sortPlayers] objectAtIndex:indexPath.row];
    PlayerViewController *playerViewController = [[PlayerViewController alloc]
initWithMasterController:masterController team:team player:player];
    [self presentViewController:playerViewController animated:YES];
}
```

That finishes the League Manager application. Build and run it. Any teams you've added should still be shown, thanks to the SQLite persistent store. Drill down into the teams and add some players, delete some players, and edit some players. Try deleting teams as well, and watch the players disappear.

## Seeing the Data in the Persistent Store

Chapter 2 shows how to use the `sqlite3` command-line tool to browse the data in the SQLite Core Data persistent store. To finish the section on SQLite persistent stores, find your SQLite database (`League_Manager.sqlite3`), and launch `sqlite3`, passing the database, with a command like this:

```
sqlite3 ./5.0/Applications/CE79C20B-4CBF-47C3-9E7C-
9EC24FA22488/Documents/League_Manager.sqlite
```

Keep the League Manager application running in the iPhone Simulator so that you can bounce between the application and `sqlite3` tool to see the effects on the database.

Start by showing the tables using the `.tables` command. Your output should look like this:

```
sqlite> .tables
ZPLAYER      ZTEAM        Z_METADATA   Z_PRIMARYKEY
```

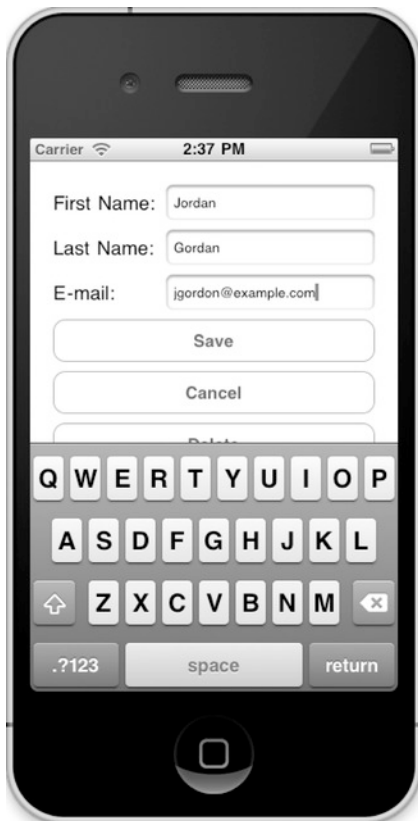
The `ZPLAYER` table holds the Player entities, and the `ZTEAM` table holds the Team entities. Create the three teams: Crew, with Blue uniforms; Fire, with Red uniforms, and Revolution, with Green uniforms. In the SQLite database, they look something like this, depending on how many teams you've created and deleted:

```
sqlite> select * from ZTEAM;  
1|2|3|Crew|Blue  
2|2|1|Fire|Red  
3|2|1|Revolution|Green
```

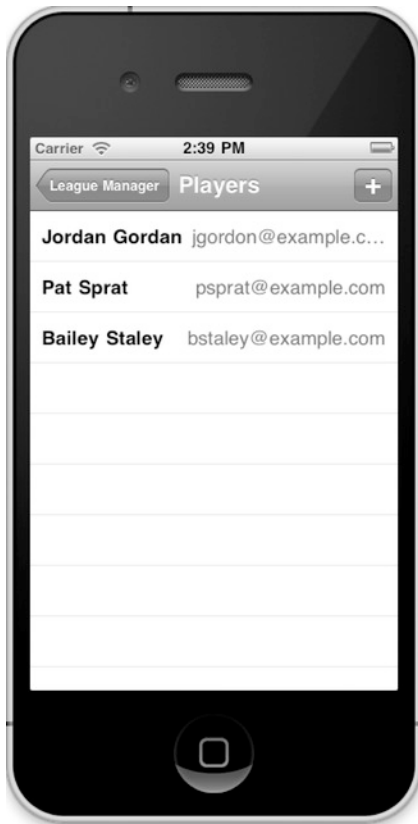
The League Manager application has no players, as a quick check in the database shows:

```
sqlite> select * from ZPLAYER;
```

Drill into the Crew team and add three players: Jordan Gordon, Pat Sprat, and Bailey Staley. Refer to Figure 3–20 to see how to enter a player. After adding the three players, you should see them all in a list on the Players screen, as in Figure 3–21.



**Figure 3–20** Adding a player



**Figure 3–21.** *Players*

Rerun the select command on the ZPLAYER table. The output should look something like this:

```
sqlite> select * from ZPLAYER;
1|1|1|1|Jordan|Gordan|jgordon@example.com
2|1|1|1|Pat|Sprat|psprat@example.com
3|1|1|1|Bailey|Staley|bstaley@example.com
```

Now add another player, but this time to the Fire team. Call this player Terry Gary. Now, run a command to show each team with the players on it, like this:

```
sqlite> select ZTEAM.ZNAME, ZPLAYER.ZFIRSTNAME, ZPLAYER.ZLASTNAME from ZTEAM, ZPLAYER
where ZTEAM.Z_PK = ZPLAYER.ZTEAM;
Crew|Jordan|Gordan
Crew|Pat|Sprat
Crew|Bailey|Staley
Fire|Terry|Gary
```

Now, delete Pat Sprat and rerun the same SQLite command. You should see output like this:

```
sqlite> select ZTEAM.ZNAME, ZPLAYER.ZFIRSTNAME, ZPLAYER.ZLASTNAME from ZTEAM, ZPLAYER
where ZTEAM.Z_PK = ZPLAYER.ZTEAM;
```

```
Crew|Jordan|Gordon
Crew|Bailey|Staley
Fire|Terry|Gary
```

Finally, delete the Fire team, and verify that not only has the Fire team been deleted, but also its only player, Terry Gary:

```
sqlite> select ZTEAM.ZNAME, ZPLAYER.ZFIRSTNAME, ZPLAYER.ZLASTNAME from ZTEAM, ZPLAYER
where ZTEAM.Z_PK = ZPLAYER.ZTEAM;
Crew|Jordan|Gordon
Crew|Bailey|Staley
```

The SQLite database proves to work in ways you understand. Feel free, as you're developing iOS applications, to peek into the SQLite database to gain a better understanding and appreciation for how Core Data works. Most of your data-backed applications will likely use SQLite databases, so understanding how they work with Core Data can help with troubleshooting issues or optimizing performance.

## Using an In-Memory Persistent Store

In the previous section, you built a Core Data–based application that uses the default SQLite persistent store type. This section deals with an alternate type: the in-memory persistent store. Let's take a look at how to switch the store type before elaborating on why you would ever want to use this type of store.

Changing the store type Core Data uses for your application is as simple as specifying the new type when creating the persistent store coordinator in the application delegate. The code for the `persistentStoreCoordinator:` method in `League_ManagerAppDelegate.m` now looks like that in Listing 3–5, with the updated code in bold.

**Listing 3–5.** *The `persistentStoreCoordinator:` Method in `League_ManagerAppDelegate.m`*

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

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

    NSError *error = nil;
    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
    if (![__persistentStoreCoordinator addPersistentStoreWithType:NSInMemoryStoreType
configuration:nil URL:nil options:nil error:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return __persistentStoreCoordinator;
}
```

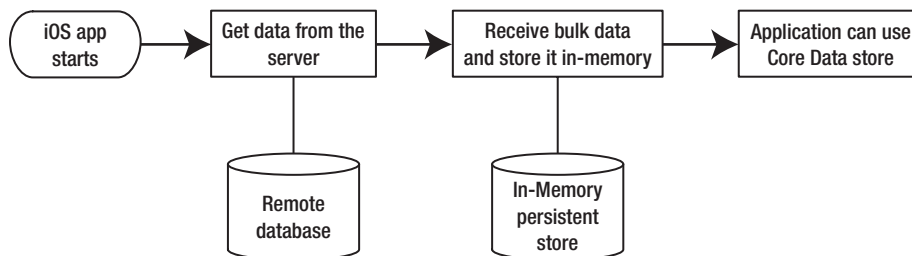
The data store has been switched to in-memory. The first thing to notice after launching the application again is that any data you previously had in your data store is gone. This happened because you switched the data store and didn't try to migrate the data from the old store to the new one. Chapter 8 explains how to migrate data between two persistent stores.

The life cycle of the in-memory data store starts when the Core Data stack is initialized and ends when the application stops.

**NOTE:** Since iOS4 and the introduction of multitasking to the iDevices, switching to another application does not necessarily terminate the currently running application. Instead, it goes into the background. The in-memory persistent store survives when an application is sent to the background so that the data is still around when the application comes back to the foreground.

When working on a data management framework and thinking about the different types of persistent stores that it should provide by default, an in-memory store isn't the first idea that comes to mind. Trying to come up with a good reason to use an in-memory store can be a challenge, but some legitimate reasons exist. For example, local caching of remote data can benefit from in-memory persistent stores. Consider a case in which your application is fed data from a remote server. If your application executes a lot of queries, good software engineering practices would prescribe the use of efficient data transfer. The remote server may transfer the data in compressed packages to your client application, which can then uncompress that data and store it in an in-memory store so that it can be efficiently queried. In this situation, you would want the data to be refreshed every time the application starts, or even periodically while the application runs, so losing the in-memory data store would be acceptable.

Figure 3-22 illustrates the start-up sequence of an application that caches remote information locally in an in-memory store.



**Figure 3-22.** *Caching remote information locally*

As you develop your Core Data-backed iOS applications, consider using in-memory data stores when applications don't require data persistence across invocations. Traditional applications, however, that require that users' data doesn't disappear simply because the application stopped running can't use this persistent store type.

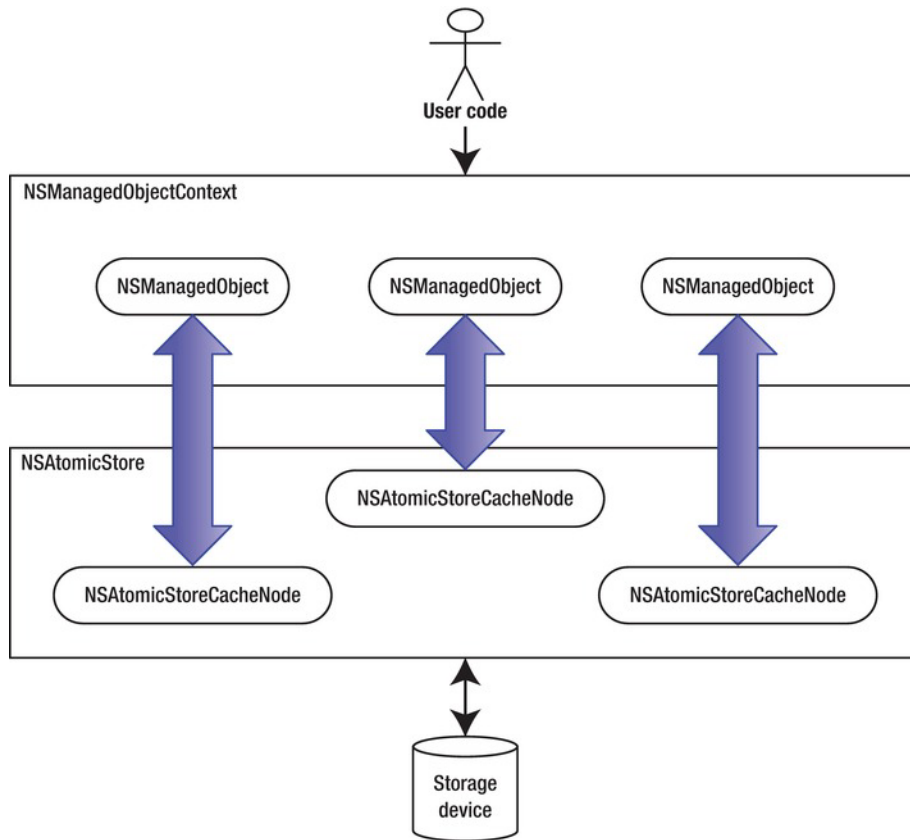
## Creating Your Own Custom Persistent Store

The principle of abstracting the persistent store implementation from the user forms the basis for the Core Data framework. This abstraction makes it possible to change the persistent store type among the different default types (NSSQLiteStoreType, NSInMemoryStoreType, NSBinaryStoreType) without changing more than a line of your code. In some cases, the default store types don't best accomplish what you are trying to achieve. The Core Data framework offers a hook for creating custom store types for these special cases. In this section, you create a new store type and use it with the League Manager application.

Before getting into the implementation itself, you should be aware that Core Data only allows you to create atomic store types. An atomic store is a store that writes its entire content all at once every time a save operation is executed. This effectively excludes the ability to create SQL-based store types that could be backed by a database other than SQLite where only the rows of data that are modified are affected in the database. In this section, you build a file-based custom store that will store its data in a comma-separated values (CSV) file except that you will use the pipe (|) symbol to separate values.

Custom data stores must extend the `NSAtomicStore` class (a subclass of `NSPersistentStore`) that provides the infrastructure necessary to hold the data. To get a better idea of how this works, picture two internal layers inside the Core Data framework, as shown in Figure 3–23. Users interact with the layer that contains `NSManagedObjects` and `NSManagedObjectContext`. The other layer performs the actual persistence and contains the persistent store coordinator and the persistent stores. In the case of custom stores, the persistence layer also contains `NSAtomicStoreCacheNode`, which contains objects that hold the data within this layer. The `NSAtomicStoreCacheNode` object is to the `NSAtomicStore` what the `NSManagedObject` is to the `NSManagedObjectContext`.





**Figure 3–23.** Two layers inside Core Data

## Initializing the Custom Store

A new custom store is responsible for transferring data between the storage device and the `NSAtomicStoreCacheNodes` as well as transferring data between the `NSManagedObjects` and the `NSAtomicStoreCacheNodes`.

The first step to create a custom store is to add a class (or classes) to implement it. The custom store this section builds lives in one class called `CustomStore`. Use Xcode to add a new Objective-C class called `CustomStore` to the League Manager application, specifying that it's a subclass of `NSAtomicStore`. `CustomStore.h` starts out trivial: it extends `NSAtomicStore` as expected, as Listing 3–6 shows.

### Listing 3–6. *CustomStore.h*

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

@interface CustomStore : NSAtomicStore {
}

@end
```

The implementation class, which starts out blank, must implement a few methods. It has accessors for its type and identifier, explained later in this section. It has an initializer that takes a persistent store coordinator and some other parameters. It also has a few other methods stubbed out that you'll implement as this section unfolds. See Listing 3–7.

**Listing 3–7. CustomStore.m**

```
#import "CustomStore.h"

@implementation CustomStore

#pragma mark - NSPersistentStore

- (NSString *)type {
    return [[self metadata] objectForKey:NSStoreTypeKey];
}

- (NSString *)identifier {
    return [[self metadata] objectForKey:NSStoreUUIDKey];
}

- (id)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator↵
configurationName:(NSString *)configurationName URL:(NSURL *)url options:(NSDictionary↵
*)options {
    self = [super initWithPersistentStoreCoordinator:coordinator↵
configurationName:configurationName URL:url options:options];

    return self;
}

+ (NSDictionary *)metadataForPersistentStoreWithURL:(NSURL *)url error:(NSError **)error
{
    return nil;
}

#pragma mark - NSAtomicStore

- (BOOL)load:(NSError **)error {
    return YES;
}

- (id)newReferenceObjectForManagedObject:(NSManagedObject *)managedObject {
    return nil;
}

- (NSAtomicStoreCacheNode *)newCacheNodeForManagedObject:(NSManagedObject↵
*)managedObject {
    return nil;
}

- (BOOL)save:(NSError **)error {
    return YES;
}
```

```

-(void)updateCacheNode:(NSAtomicStoreCacheNode *)node fromManagedObject: ←
(NSManagedObject *)managedObject {
}

```

```
@end
```

All Core Data stores have supporting metadata that help the persistent store coordinator manage the different stores. The metadata is materialized in the `NSPersistentStore` class as an `NSDictionary`. Two data elements are of particular interest to a new data store: `NSStoreTypeKey` and `NSStoreUUIDKey`. The `NSStoreTypeKey` value must be a string that uniquely identifies the data store type, while the `NSStoreUUIDKey` must be a string that uniquely identifies the data store itself. To create unique identifiers, add a static utility method that creates and returns universally unique identifiers (UUIDs):

```

+ (NSString *)makeUUID {
    CFUUIDRef uuidRef = CFUUIDCreate(NULL);
    CFStringRef uuidStringRef = CFUUIDCreateString(NULL, uuidRef);
    CFRelease(uuidRef);
    NSString* uuid = [NSString stringWithString:(__bridge NSString *)uuidStringRef];
    CFRelease(uuidStringRef);
    return uuid;
}

```

In this chapter's example, two data files support the custom store. The first file, which has a `.txt` extension, contains the data itself, and the second file, which has a `.plist` extension, contains the metadata. For the problem of loading and saving the metadata, you will add a method to save the metadata and complete the implementation of `metadataForPersistentStoreWithURL:error:` to load the metadata.

Data stores are initialized relative to a base URL. In the `CustomStore` example, the URL points to the data file (the `.txt` file), and the metadata file URL is derived from the base URL by swapping the `.txt` extension for a `.plist` extension.

The `writeMetadata:toURL:` method takes the metadata `NSDictionary` and writes it to a file, like so:

```

+ (void)writeMetadata:(NSDictionary*)metadata toURL:(NSURL*)url {
    NSString *path = [[url relativePath] stringByAppendingString:@".plist"];
    [metadata writeToFile:path atomically:YES];
}

```

Loading the metadata is slightly more complicated because if the data store is new and the metadata file does not exist, the metadata file must be created along with an empty data file. Core Data expects a store type, and a store UUID from the metadata helps the persistent store coordinator deal with the custom store, so set those values for the `NSStoreTypeKey` and `NSStoreUUIDKey`. Find the `metadataForPersistentStoreWithURL:error:` method and change its contents to check for the existence of a metadata file and, if not found, to write one with the store type key and the UUID key, and also to write a blank data file, like in Listing 3–8.

**Listing 3–8. The `metadataForPersistentStoreWithURL:error:` Method**

```
+ (NSDictionary *)metadataForPersistentStoreWithURL:(NSURL *)url error:(NSError **)error
{
    // Determine the filename for the metadata file
    NSString *path = [[url relativePath] stringByAppendingString:@"plist"];

    // If the metadata file doesn't exist, create it
    if(![[NSFileManager defaultManager] fileExistsAtPath:path]) {
        // Create a dictionary and store the store type key (CustomStore)
        // and the UUID key
        NSMutableDictionary *metadata = [NSMutableDictionary dictionary];
        [metadata setValue:@"CustomStore" forKey:NSStoreTypeKey];
        [metadata setValue:[CustomStore makeUUID] forKey:NSStoreUUIDKey];

        // Write the metadata to the .plist file
        [CustomStore writeMetadata:metadata toURL:url];

        // Write an empty data file
        [@" writeURL:url atomically:YES encoding:[NSString defaultCStringEncoding]
error:nil];

        NSLog(@"Created new store at %@", path);
    }
    return [NSDictionary dictionaryWithContentsOfFile:path];
}
```

Armed with methods to retrieve the metadata and create a blank store, you can complete the initialization method, like so:

```
- (id)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
configurationName:(NSString *)configurationName URL:(NSURL *)url options:(NSDictionary
*)options {
    self = [super initWithPersistentStoreCoordinator:coordinator
configurationName:configurationName URL:url options:options];

    NSDictionary *metadata = [CustomStore metadataForPersistentStoreWithURL:[self URL]
error:nil];
    [self setMetadata:metadata];

    return self;
}
```

## Mapping Between `NSManagedObject` and `NSAtomicStoreCacheNode`

To make the custom store function properly, you must provide implementations for three additional utility methods. The first one creates a new reference object for a given managed object. Reference objects represent unique identifiers for each `NSAtomicStoreCacheNode` (similar to a database primary key). A Reference object is to an `NSAtomicStoreCacheNode` what an `NSObjectID` is to an `NSManagedObject`. Since the custom data store has to manage data transfer between `NSManagedObjects` and

NSAtomicCacheNodes, it must be able to create a reference object for a newly created managed object. For this, you use the UUID again.

```
- (id)newReferenceObjectForManagedObject:(NSManagedObject *)managedObject {
    NSString *uuid = [CustomStore makeUUID];
    return uuid;
}
```

The second method needed creates a new NSAtomicStoreCacheNode instance to match a newly created NSManagedObject. When a new NSManagedObject is added and needs to be persisted, the framework first gets a reference object using the newReferenceObjectForManagedObject: method. NSAtomicCache keeps track of the mapping between NSObjectIDs and reference objects. When Core Data persists a managed object into the persistent store, it calls the newCacheNodeForManagedObject: method, which, like its name indicates, creates a new NSAtomicStoreCacheNode that will serve as a peer to the NSManagedObject.

```
- (NSAtomicStoreCacheNode *)newCacheNodeForManagedObject:(NSManagedObject *)managedObject {
    NSManagedObjectID *oid = [managedObject objectID];
    id referenceID = [self referenceObjectForObjectID:oid];

    NSAtomicStoreCacheNode* node = [self nodeForReferenceObject:referenceID
andObjectID:oid];
    [self updateCacheNode:node fromManagedObject:managedObject];
    return node;
}
```

The newCacheNodeForManagedObject: implementation looks up the reference object that was created for the managed object and creates a new cache node linked to that reference ID. Finally, the method copies the managed object's data into the node using the updateCacheNode:fromManagedObject: method. Your custom store also needs to provide an implementation for this third method, shown in Listing 3-9.

**Listing 3-9.** *The updateCacheNode:fromManagedObject: Method*

```
- (void)updateCacheNode:(NSAtomicStoreCacheNode *)node
fromManagedObject:(NSManagedObject *)managedObject {
    // Determine the entity for the managed object
    NSEntityDescription *entity = managedObject.entity;

    // Walk through all the attributes in the entity
    NSDictionary *attributes = [entity attributesByName];
    for (NSString *name in [attributes allKeys]) {
        // For each attribute, set the managed object's value into the node
        [node setValue:[managedObject valueForKey:name] forKey:name];
    }

    // Walk through all the relationships in the entity
    NSDictionary *relationships = [entity relationshipsByName];
    for (NSString *name in [relationships allKeys]) {
        id value = [managedObject valueForKey:name];
        // If this is a to-many relationship . . .
        if ([[relationships objectForKey:name] isToMany]) {
```

```

        // . . . get all the destination objects
        NSMutableSet *set = (NSMutableSet*)value;
        NSMutableSet *data = [NSMutableSet set];
        for (NSManagedObject *managedObject in set) {
            // For each destination object in the relationship,
            // add the cache node to the set
            NSManagedObjectID *oid = [managedObject objectID];
            id referenceID = [self referenceObjectForObjectID:oid];
            NSAtomicStoreCacheNode* n = [self nodeForReferenceObject:referenceID
andObjectID:oid];
            [data addObject:n];
        }
        [node setValue:data forKey:name];
    } else {
        // This is a to-one relationship, so just get the single
        // destination node for the relationship
        NSManagedObject *managedObject = (NSManagedObject*)value;
        NSManagedObjectID *oid = [managedObject objectID];
        id referenceID = [self referenceObjectForObjectID:oid];
        NSAtomicStoreCacheNode* n = [self nodeForReferenceObject:referenceID
andObjectID:oid];
        [node setValue:n forKey:name];
    }
}
}
}

```

The implementation finds the entity description for the given managed object and uses it to iterate through attributes and relationships in order to copy their values into the node.

To keep track of cache nodes, create a utility method that, given a reference object, returns the matching `NSAtomicStoreCacheNode` if it exists or creates a new one.

```

- (NSAtomicStoreCacheNode *)nodeForReferenceObject:(id)referenceObject
andObjectID:(NSManagedObjectID *)oid {
    NSAtomicStoreCacheNode *node = [nodeCacheRef objectForKey:reference];
    if (node == nil) {
        node = [[NSAtomicStoreCacheNode alloc] initWithObjectID:oid];
        [nodeCacheRef setObject:node forKey:reference];
    }
    return node;
}

```

The implementation of `nodeForReferenceObject:andObjectID:` uses a dictionary called `nodeCacheRef`, so declare it in the `CustomStore.h` header file, as Listing 3–10 shows.

**Listing 3–10.** *CustomStore.h*

```

#import <Foundation/Foundation.h>

@interface CustomStore : NSAtomicStore {
    NSMutableDictionary *nodeCacheRef;
}

@end

```

Initialize nodeCacheRef in the

initWithPersistentStoreCoordinator:configurationName:URL:options: method.

```
- (id)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
configurationName:(NSString *)configurationName URL:(NSURL *)url options:(NSDictionary
*)options {
    self = [super initWithPersistentStoreCoordinator:coordinator
configurationName:configurationName URL:url options:options];
    NSDictionary *metadata = [CustomStore metadataForPersistentStoreWithURL:[self URL]
error:nil];
    [self setMetadata:metadata];
    nodeCacheRef = [NSMutableDictionary dictionary];
    return self;
}
```

## Serializing the Data

So far, all you've done is implement utility methods to deal with the metadata, initialize the data store, and perform the data transfer between `NSManagedObject` instances and `NSAtomicStoreCacheNode` instances. Until you implement the methods that read and write to the storage device, however, the custom store has no use. When extending `NSAtomicStore`, you are required to provide implementations for the `load:` and `save:` methods, which serve as the meat of the custom store implementation. In this example, start with the `save:` method. The code for the following `save:` method might seem overwhelming at first, but if you take time to follow the code, you will realize that it simply iterates through the cache nodes and writes attribute values into the file, followed by relationship values. Attribute values are converted into `NSString`s and appended to the pipe-delimited file as key-value pairs in the form *attributeName=value*. Relationships work in a similar way except that the value written is not the destination node itself but its reference object as created by the `newReferenceObjectForManagedObject:` method. For one-to-many relationships, the code writes a comma-delimited list of reference objects. Listing 3–11 shows the `save:` method.

**Listing 3–11.** *The save: Method*

```
- (BOOL)save:(NSError **)error {
    NSURL *url = [self URL];

    // First update the metadata
    [CustomStore writeMetadata:[self metadata] toURL:url];

    NSString* dataFile = @"";
    // Then write the actual data
    NSSet *nodes = [self cacheNodes];
    NSAtomicStoreCacheNode *node;
   NSEnumerator *enumerator = [nodes objectEnumerator];

    // Enumerate through all the nodes
    while ((node = [enumerator nextObject]) != nil) {
        // Get the object ID and reference ID for each node
        NSManagedObjectID *oid = [node objectID];
        id referenceID = [self referenceObjectForObjectID:oid];
```

```

// Write the entity name and reference ID as the first two
// values in the row
NSEntityDescription *entity = [oid entity];
dataFile = [dataFile stringByAppendingFormat:@"%s|%@", entity.name, referenceID];

{
    // Write all the attributes
    NSDictionary *attributes = [entity attributesByName];
    NSAttributedString *key = nil;
    NSEnumerator *enumerator = [attributes objectEnumerator];
    while ((key = [enumerator nextObject]) != nil) {
        NSString *value = [node valueForKey:key.name];
        if (value == nil) value = @"(null)";
        dataFile = [dataFile stringByAppendingFormat:@"%s|%@", key.name, value];
    }
}

{
    // Write all the relationships
    NSDictionary *relationships = [entity relationshipsByName];
    NSRelationshipDescription *key = nil;
    NSEnumerator *enumerator = [relationships objectEnumerator];
    while ((key = [enumerator nextObject]) != nil) {
        id value = [node valueForKey:key.name];
        if (value == nil) {
            dataFile = [dataFile stringByAppendingFormat:@"%s|%@", key.name, @"(null)"];
        } else if (![key isToMany]) { // One-to-One
            NSManagedObjectID *oid = [(NSAtomicStoreCacheNode*)value objectID];
            id referenceID = [self referenceObjectForObjectID:oid];
            dataFile = [dataFile stringByAppendingFormat:@"%s|%@", key.name,
referenceID];
        } else { // One-to-Many
            NSMutableSet *set = (NSMutableSet*)value;
            if ([set count] == 0) {
                dataFile = [dataFile stringByAppendingFormat:@"%s|%@", key.name,
@"(null)"];
            } else {
                NSString *list = @"";
                for (NSAtomicStoreCacheNode *item in set) {
                    id referenceID = [self referenceObjectForObjectID:[item objectID]];
                    list = [list stringByAppendingFormat:@"%s,", referenceID];
                }
                list = [list substringToIndex:[list length]-1];
                dataFile = [dataFile stringByAppendingFormat:@"%s|%@", key.name, list];
            }
        }
    }
}

// Add a new line to go to the next row for the next node
dataFile = [dataFile stringByAppendingString:@"\n"];
}

// Write the file
NSString *path = [url relativePath];
[dataFile writeToFile:path atomically:YES encoding:[NSString defaultCStringEncoding]
error:error];

```



```
    return YES;
}
```

Each data record in the text file, represented in code by an `NSAtomicStoreCacheNode` instance, follows this format:

```
Entity Name|Reference
Object|attribute1=value1|attribute2=value2|...|relationship1=ref1,ref2,ref3|relationship
2=ref4|...
```

The `load:` method follows the same steps as the `save:` method but in reverse. It reads the data file line by line and, for each line, uses the first element to find the entity description, uses the second element as the node's reference object, and then iterates through the remaining elements to load the attributes and relationships. It uses these elements to reconstruct the `NSAtomicStoreCacheNode` instances; see Listing 3–12.

**Listing 3–12. The `load:` Method**

```
- (BOOL)load:(NSError **)error {
    // Find the file to load from
    NSURL* url = [self URL];
    NSMutableSet *nodes = [NSMutableSet set];
    NSString *path = [url relativePath];
    if(![[NSFileManager defaultManager] fileExistsAtPath:path]) {
        // File doesn't exist, so add an empty set and bail
        [self addCacheNodes:nodes];
        return YES;
    }
    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];

    // Load the entire file into an array, with each element
    // containing one line from the file. Each element is then one
    // node
    NSString *fileString = [NSString stringWithContentsOfFile:path encoding:[NSString
defaultCStringEncoding] error:error];
    NSArray *lines = [fileString componentsSeparatedByString:@"\n"];
    NSString *line;

    // Enumerate through each line from the file
    NSEnumerator *enumerator = [lines objectEnumerator];
    while ((line = [enumerator nextObject]) != nil) {
        // Split the fields into an array
        NSArray *components = [line componentsSeparatedByString:@"|"];

        // If you don't have at least an entity name and a reference ID,
        // ignore this line
        if ([components count] < 2) continue;
        NSString *entityName = [components objectAtIndex:0];
        NSString *pkey = [components objectAtIndex:1];

        // Make the node
        NSEntityDescription *entity = [[[coordinator managedObjectModel] entitiesByName]
valueForKeyPath:entityName];
        if (entity != nil) {
            NSManagedObjectID *oid = [self objectIDForEntity:entity referenceObject:pkey];
```

```

NSAtomicStoreCacheNode *node = [self nodeForReferenceObject:pkey andObjectID:oid];

// Get the attributes and relationships from the model
NSDictionary *attributes = [entity attributesByName];
NSDictionary *relationships = [entity relationshipsByName];

// Go through the rest of the fields
for (int i = 2; i < [components count]; i++) {
    // Each field is a name/value pair, separated by an equals
    // sign. Parse name and value
    NSArray *entry = [[components objectAtIndex:i]
componentsSeparatedByString:@"="];
    NSString *key = [entry objectAtIndex:0];
    if([attributes objectForKey:key] != nil) {
        // Get the type of the attribute from the model
        NSAttributedString *attributeDescription = [attributes objectForKey:key];
        NSAttributedString type = [attributeDescription attributeType];

        // Default value to type string
        id dataValue = [entry objectAtIndex:1];
        if ([([NSString*]dataValue compare:@"(null)"] == NSOrderedSame) {
            continue;
        }
        // Convert the value to the proper data type
        if ((type == NSInteger16AttributeType) || (type == NSInteger32AttributeType)
|| (type == NSInteger64AttributeType)) {
            dataValue = [NSNumber numberWithInt:[dataValue integerValue]];
        } else if (type == NSDecimalAttributeType) {
            dataValue = [NSDecimalNumber decimalNumberWithString:dataValue];
        } else if (type == NSDoubleAttributeType) {
            dataValue = [NSNumber numberWithDouble:[dataValue doubleValue]];
        } else if (type == NSFloatAttributeType) {
            dataValue = [NSNumber numberWithFloat:[dataValue floatValue]];
        } else if (type == NSBooleanAttributeType) {
            dataValue = [NSNumber numberWithBool:[dataValue intValue]];
        } else if (type == NSDateAttributeType) {
            NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
            [formatter setDateFormat:@"yyyy-MM-dd HH:mm:ss ZZZ"];
            dataValue = [formatter dateFromString:dataValue];
        } else if (type == NSBinaryDataAttributeType) {
            // This implementation doesn't support binary data
            // You could enhance this code to base64 encode and
            // decode binary data to be able to support binary
            // types.
            NSLog(@"Binary type not supported");
        }
        // Set the converted value into the node
        [node setValue:dataValue forKey:key];
    } else if ([relationships objectForKey:key] != nil) { // See if it's a
relationship
        // Destination objects are comma-separated
        NSArray *ids = [[entry objectAtIndex:1] componentsSeparatedByString:@","];
        NSRelationshipDescription *relationship = [relationships objectForKey:key];
        // If it's a to-many relationship . . .

```

```

    if ([relationship isToMany]) {
        // . . . get the set of destination objects and
        // iterate through them
        NSMutableSet* set = [NSMutableSet set];
        for (NSString *fKey in ids) {
            if (fKey != nil && [fKey compare:@"(null)" != NSOrderedSame) {
                // Create the node for the destination object
                NSManagedObject *oid = [self objectIDForEntity:[relationship
destinationEntity] referenceObject:fKey];
                NSAtomicStoreCacheNode *destinationNode = [self
nodeForReferenceObject:fKey andObjectID:oid];
                [set addObject:destinationNode];
            }
        }
        // Store the set into the node
        [node setValue:set forKey:key];
    } else {
        // This is a to-one relationship; check whether there's
        // a destination object
        NSString* fKey = [ids count] > 0 ? [ids objectAtIndex:0] : nil;
        if (fKey != nil && [fKey compare:@"(null)" != NSOrderedSame) {
            // Set the destination into the node
            NSManagedObject *oid = [self objectIDForEntity:[relationship
destinationEntity] referenceObject:fKey];
            NSAtomicStoreCacheNode *destinationNode = [self
nodeForReferenceObject:fKey andObjectID:oid];
            [node setValue:destinationNode forKey:key];
        }
    }
}
}
}
// Remember this node
[nodes addObject:node];
}
}
// Register all the nodes
[self addCacheNodes:nodes];
return YES;
}

```

Remember that although the text file stores the data as plain text, Core Data deals with objects. The code must check the data type of each attribute using the entity description and create the appropriate data object instance. For relationships, the code must use the store reference objects in order to either reuse existing nodes or create new ones if needed. To reuse existing nodes or create new ones, the `load:` method uses the previously implemented `nodeForReferenceObject:andObjectID:` method.

Before moving on from `CustomStore.m`, add declarations for the three utility methods you've created so the compiler won't complain. Because these methods aren't used outside the `CustomStore` class, you don't add them to the header file. The declarations look like this:

```
@interface CustomStore (private)

+ (void)writeMetadata:(NSDictionary*)metadata toURL:(NSURL*)url;
+ (NSString *)makeUUID;
- (NSAtomicStoreCacheNode*)nodeForReferenceObject:(id)referenceâ
andObjectID:(NSManagedObjectID*)oid;
@end
```

Add this code to the top of CustomStore.m, after the import for CustomStore.h and before the @implementation CustomStore line.

## Using the Custom Store

The last step required to use the custom store with the League Manager application is to register it and use it when initializing the persistent store coordinator. This is all done in the application delegate (League\_ManagerAppDelegate.m). First, add an import at the top of the class so that the implementation is aware of the CustomStore class.

```
#import "CustomStore.h"
```

Modify the application:didFinishLaunchingWithOptions: method to register the custom store when the application awakes.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [NSPersistentStoreCoordinator registerStoreClass:[CustomStore class]~
forStoreType:@"CustomStore"];
    NSLog(@"Registered types: %@", [NSPersistentStoreCoordinator registeredStoreTypes]);

    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.

    MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

Finally, alter the persistentStoreCoordinator: accessor to use the new custom store.

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

    NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"League_Manager.txt"];

    NSError *error = nil;
```

```

__persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
if (![__persistentStoreCoordinator addPersistentStoreWithType:@"CustomStore"
configuration:nil URL:storeURL options:nil error:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

return __persistentStoreCoordinator;
}

```

At this point, you should be able to start the application, and it will use the new custom store. A file called `League_Manager.txt` will be created in the same directory where the `League_Manager.sqlite` database was when you first implemented it earlier in this chapter. Take the application for a spin, add a few teams and players, and then go check the `League_Manager.txt` data file using any text editor. Depending on the teams and players you create, the file will look something like this:

```

Player|5247F86F-0397-4814-9AF7-
5F6946BE20C4|email=rrondo@example.com|firstName=Rajon|lastName=Rondo|team=9F2662B9-3006-
4B8D-BC43-EA68DB7AD037
Player|20C2D39C-D831-4AD7-9FE7-
E1CE22D765E0|email=twarner@example.com|firstName=Tyson|lastName=Warner|team=3BBC1923-
0EA7-485C-9516-3B993D30DOCE
Player|3E817374-2BE2-495F-8029-
1698C34DE07F|email=ljames@example.com|firstName=LeBron|lastName=James|team=3BBC1923-
0EA7-485C-9516-3B993D30DOCE
Player|7ABACD30-2586-4522-8369-
0D104FE248AF|email=cbosh@example.com|firstName=Chris|lastName=Bosh|team=3BBC1923-0EA7-
485C-9516-3B993D30DOCE
Player|D14E8D95-FC94-48AA-BE9B-
E20FC36B70EB|email=dwade@example.com|firstName=Dwyane|lastName=Wade|team=3BBC1923-0EA7-
485C-9516-3B993D30DOCE
Team|3BBC1923-0EA7-485C-9516-3B993D30DOCE|name=Miami
Heat|uniformColor=Red|players=3E817374-2BE2-495F-8029-1698C34DE07F,7ABACD30-2586-4522-
8369-0D104FE248AF,D14E8D95-FC94-48AA-BE9B-E20FC36B70EB,20C2D39C-D831-4AD7-9FE7-
E1CE22D765E0
Team|9F2662B9-3006-4B8D-BC43-EA68DB7AD037|name=Boston
Celtics|uniformColor=Green|players=5247F86F-0397-4814-9AF7-5F6946BE20C4

```

In the data file, you can see two teams, Boston Celtics and Miami Heat, in the last two lines of the file. The entity name, “Team,” is the first field on each of those two rows. You see that the Boston Celtics has one player, as it has only one reference ID listed for the players relationship in the last field of its row. That reference ID listed for the players relationship matches the reference ID for Rajon Rondo, which is the second field in the first row of the file. You can also see the four players that represent the Miami Heat’s future: LeBron James, Dwyane Wade, Chris Bosh, and Tyson Warner.

## What About XML Persistent Stores?

Core Data offers another persistent store type, XML, on Mac OS X that you specify by passing `NSXMLStoreType` to your persistent store coordinator’s `addPersistentStoreWithType:` method. Passing `NSXMLStoreType` when compiling for iOS,

however, gives you a compiler error: 'NSXMLStoreType' undeclared. If you look in the Core Data header files for iOS, you'll find the following in `NSPersistentStoreCoordinator.h`:

```
// Persistent store types supported by Core Data:
COREDATA_EXTERN NSString * const NSSQLiteStoreType __OSX_AVAILABLE_STARTING(__MAC_10_4,
__IPHONE_3_0);
COREDATA_EXTERN NSString * const NSBinaryStoreType __OSX_AVAILABLE_STARTING(__MAC_10_4,
__IPHONE_3_0);
COREDATA_EXTERN NSString * const NSInMemoryStoreType
__OSX_AVAILABLE_STARTING(__MAC_10_4, __IPHONE_3_0);
```

Indeed, `NSXMLStoreType` remains undeclared, though it's emphatically available for Mac OS X. To understand why, turn to Apple's developer documentation on Core Data Persistent Store Features, found at <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/Articles/cdPersistentStores.html>, to read the following explanation for XML's exclusion on iOS:

iOS: The XML store is not available on iOS.

That's all you get, keeping Apple's reputation for secrecy intact. You're left, then, to speculate. Apple probably left XML off iOS's version of Core Data for a few reasons:

- Parsing XML can consume a lot of processor cycles, making an iPhone, with its slower processor relative to an iMac, Mac Pro, and so on, slower.
- Because parsing XML uses more processing power, it can consume more battery life.
- XML, with its characters, brackets, and metadata, generally consumes more storage space than binary file types.
- Apple likes to steer its developers toward solutions it deems better, rather than providing as many options as possible. XML is superfluous for Core Data persistence.

If you miss XML and want it available for your applications, you can write your own XML custom store. If you imagine a nice, nested XML document issuing from your complex data models, however, you will probably become frustrated in your attempts to create a custom XML persistent store type. Core Data relationships have their inverses, meaning that you really can't arbitrate parenthood among entities. In the League Manager data model, for example, should `Team` entity tags contain `Player` entity tags, because the team "owns" its players, or should it be the reverse? Do players own the teams they play for (as many superstar professional athletes have demonstrated)? If you pursue an XML custom data store, you'll find that you don't produce readable XML documents that make relationships clear, but rather XML documents with lots of peers that are fit only for Core Data's consumption. For example, if you port the same League Manager data model to a Mac OS X Core Data application that uses an XML persistent store and enter the same data for teams and players, Core Data produces the XML document shown in Listing 3-13.

**Listing 3–13.** XML from Porting the League Manager Data Model to a Mac OS X Core Data Application

```

<?xml version="1.0"?>
<!DOCTYPE database SYSTEM "file:///System/Library/DTDs/CoreData.dtd">

<database>
  <databaseInfo>
    <version>134481920</version>
    <UUID>45AD66DE-CC52-4B2B-931C-6ACA69BB5507</UUID>
    <nextObjectID>108</nextObjectID>
    <metadata>
      <plist version="1.0">
        <dict>
          <key>NSPersistenceFrameworkVersion</key>
          <integer>251</integer>
          <key>NSStoreModelVersionHashes</key>
          <dict>
            <key>Player</key>
            <data>
              QRI+8jf50XSA5dkydbK20isvHVrWhCAAttY9Yh4oUSQ=
            </data>
            <key>Team</key>
            <data>
              V/p0fHFfixiAQ1Nb7Xlg2Xu4laNYWtrsg5Br1qtI9JMY=
            </data>
          </dict>
          <key>NSStoreModelVersionHashesVersion</key>
          <integer>3</integer>
          <key>NSStoreModelVersionIdentifiers</key>
          <array></array>
        </dict>
      </plist>
    </metadata>
  </databaseInfo>
  <object type="TEAM" id="z102">
    <attribute name="uniformcolor" type="string">Red</attribute>
    <attribute name="name" type="string">Fire</attribute>
    <relationship name="players" type="0/0" destination="PLAYER"
idrefs="z103"></relationship>
  </object>
  <object type="PLAYER" id="z103">
    <attribute name="lastname" type="string">Gary</attribute>
    <attribute name="firstname" type="string">Terry</attribute>
    <attribute name="email" type="string">tgary@example.com</attribute>
    <relationship name="team" type="1/1" destination="TEAM"
idrefs="z102"></relationship>
  </object>
  <object type="PLAYER" id="z104">
    <attribute name="lastname" type="string">Sprat</attribute>
    <attribute name="firstname" type="string">Pat</attribute>
    <attribute name="email" type="string">psprat@example.com</attribute>
    <relationship name="team" type="1/1" destination="TEAM"
idrefs="z105"></relationship>
  </object>
  <object type="TEAM" id="z105">
    <attribute name="uniformcolor" type="string">Blue</attribute>
    <attribute name="name" type="string">Crew</attribute>

```

```

    <relationship name="players" type="0/0" destination="PLAYER" idrefs="z107 z104
z106"></relationship>
  </object>
  <object type="PLAYER" id="z106">
    <attribute name="lastname" type="string">Staley</attribute>
    <attribute name="firstname" type="string">Bailey</attribute>
    <attribute name="email" type="string">bstaley@example.com</attribute>
    <relationship name="team" type="1/1" destination="TEAM"
idrefs="z105"></relationship>
  </object>
  <object type="PLAYER" id="z107">
    <attribute name="lastname" type="string">Gordon</attribute>
    <attribute name="firstname" type="string">Jordan</attribute>
    <attribute name="email" type="string">jgordon@example.com</attribute>
    <relationship name="team" type="1/1" destination="TEAM"
idrefs="z105"></relationship>
  </object>
  <object type="TEAM" id="z108">
    <attribute name="uniformcolor" type="string">Green</attribute>
    <attribute name="name" type="string">Revolution</attribute>
    <relationship name="players" type="0/0" destination="PLAYER"></relationship>
  </object>
</database>

```

Core Data makes no attempt to determine parentage between teams and players, nor does it create tags for the different entity types. It relies instead on tags called object with entity names specified as type attributes; moreover, it makes players and teams peers, which sounds like a recipe for lockouts and season cancellations.

## Summary

SQLite claims on its home page, [www.sqlite.org](http://www.sqlite.org), that it “is the most widely deployed SQL database engine in the world.” It enjoys the backing of technology titans like Oracle, Mozilla, and Adobe. If you’ve spent much time in your career in typical corporate development roles, you probably feel like data belongs in a database. SQLite stores your data efficiently and compactly. It doesn’t require atomically rewriting the persistent store every time you change it. It requires no custom coding to use it. Xcode generates all you need to get started with a SQLite-backed persistent store. Why wouldn’t you use a SQLite database for all your Core Data persistent stores?

Well, you might. SQLite is probably the right choice for data storage for most, if not all, of your applications’ data storage needs. You’ve learned in this chapter, however, that other options exist and that you can create your own store types optimized for your particular applications and data needs. Whether you’re working with remote data that you should store only in memory, wanting to persist data into text files, or imagining some other scenario best served by some other custom data store type, understand that your data can live in places other than a SQLite database. Remember that you’re free to store your data however you want, and Core Data will manage it for you appropriately.



## Creating a Data Model

You can create applications with the most intuitive user interfaces that perform tasks users can't live without, but if you don't model your data correctly, your applications will become difficult to maintain, will underperform, and might even become unusable. This chapter explains how to model your data to support, not undermine, your applications.

### Designing Your Database

The American philosopher Ralph Waldo Emerson once said, “A foolish consistency is the hobgoblin of little minds.” People often wield that quote to defend carelessness or inattention to detail. We hope we're not falling into that trap as we flip-flop inconsistently between claiming Core Data is not a database and treating it as if it were. This section discusses how to design your data structures, and likening this process to modeling a relational database greatly helps not only the discussion but also the resulting design. The analogy breaks down in spots, however, as all analogies do, and we point out those spots through this discussion.

Core Data entity descriptions look, act, smell, and taste an awful lot like tables in a relational database. Attributes seem like columns in the tables, relationships feel like joins on primary and foreign keys, and entities appear like rows. If your model sits on a SQLite persistent store, Core Data actually realizes the entity descriptions, attributes, relationships, and entities as the database structures you'd expect, as Chapters 2 and 3 demonstrate. Remember, though, that your data can live in memory only or in some flat-file atomic store that has no tables or rows or columns or primary and foreign keys. Core Data abstracts the data structure from a relational database structure, simplifying both how you model and interact with your data. Allow for that abstraction in your mental model of how Core Data modeling works, or you will fill your Core Data model with craft, work against Core Data, and produce suboptimal data structures.

Newcomers to Core Data who have data modeling experience decry the lack of an autoincrement field type, believing that they have the responsibility, as they do in traditional data modeling, to define a primary key for each table. Core Data has no autoincrement type because you don't define any primary keys. Core Data assumes the responsibility to establish and maintain the uniqueness of each managed object, or row.

No primary keys means no foreign keys, either; Core Data manages the relationships between entities, or tables, and performs any necessary joins for you. Get over the discomfort of not defining primary and foreign keys quickly, because agonizing over this implementation detail isn't worth your effort or angst.

Another place where too much database modeling knowledge can run you against Core Data's grain involves many-to-many relationships. Consider, for example, if players in the League Manager application could belong to more than one team. If this were true, each team could have many players, and each player could play for many teams. If you were creating a traditional relational data model, you'd create three tables: one for teams, one for players, and one to track each team-player relationship. In Core Data, you'd still have just two entities, Team and Player, and you'd change the relationship from Player to Team to a to-many relationship. Core Data would still implement this in a SQLite database as three tables, ZTEAM, ZPLAYER, and Z1TEAMS, but that's an implementation detail. You wouldn't need to know about the third table, because Core Data takes care of it for you.

**TIP:** Here's a rule of thumb to remember as you create your Core Data models: worry about data, not data storage mechanisms.

## Relational Database Normalization

Relational database theory espouses a process for designing a data model called *normalization*, which aims to reduce or eliminate redundancy and provide efficient access to the data inside the database. The normalization process divides into five levels, or *forms*, and work continues on a sixth form. The five normal forms carry definitions only a logician can love, or perhaps even understand; they read something like this:

*"A relation R is in fourth normal form (4NF) if and only if, wherever there exists an MVD in R, say  $A \twoheadrightarrow B$ , then all attributes of R are also functionally dependent on A. In other words, the only dependencies (FDs or MVDs) in R are of the form  $K \rightarrow X$  (i.e. a functional dependency from a candidate key K to some other attribute X). Equivalently: R is in 4NF if it is in BCNF and all MVD's in R are in fact FDs."*

([www.databasedesign-resource.com/normal-forms.html](http://www.databasedesign-resource.com/normal-forms.html))

We have neither sufficient pages in this book nor the inclination to walk through formal definitions for each of the normal forms and then explain what they mean. Instead, this section describes each of the normal forms in relation to Core Data and provides data modeling advice. Note that adherence to a given normal form requires adherence to all the normal forms that precede it.

A database that conforms to first normal form (1NF) is considered normalized. To conform to this level of normalization, each row in the database must have the same number of fields. For a Core Data model, this means that each managed object should have the same number of attributes. Since Core Data doesn't allow you to create variable numbers of attributes in your entities, your Core Data models are automatically normalized.

Second normal form (2NF) and third normal form (3NF) deal with the relationship between nonkey fields and key fields, dictating that nonkey fields should be facts about the entire key field to which they belong. Since your Core Data model has no key fields, you shouldn't run afoul of these concerns. You should, however, make sure that all the attributes for a given entity describe that entity, and not something else. For example, the `Player` entity should not have a `uniformColor` field since the uniform color describes the team rather than an independent player.

The next two normal forms, fourth normal form (4NF) and fifth normal form (5NF), can be considered the same form in the Core Data world. They deal with reducing or eliminating data redundancy, pushing you to move multiple-value attributes into separate entities and create to-many relationships between the entities. In Core Data terms, 4NF says that entities shouldn't have attributes that can have multiple values. Instead, you should move these attributes to a separate entity and create a to-many relationship between the entities. Consider, for example, the teams and players in the model in the League Manager application from the previous chapter. A model that would violate 4NF and 5NF would have a single entity, `Team`, with an additional attribute: `player`. We would then create a new `Team` managed object for each player, so that we'd have several redundant `Team` objects. Instead, the League Manager data model moves players to a `Player` entity and creates a to-many relationship, `players`, between each team and its players.

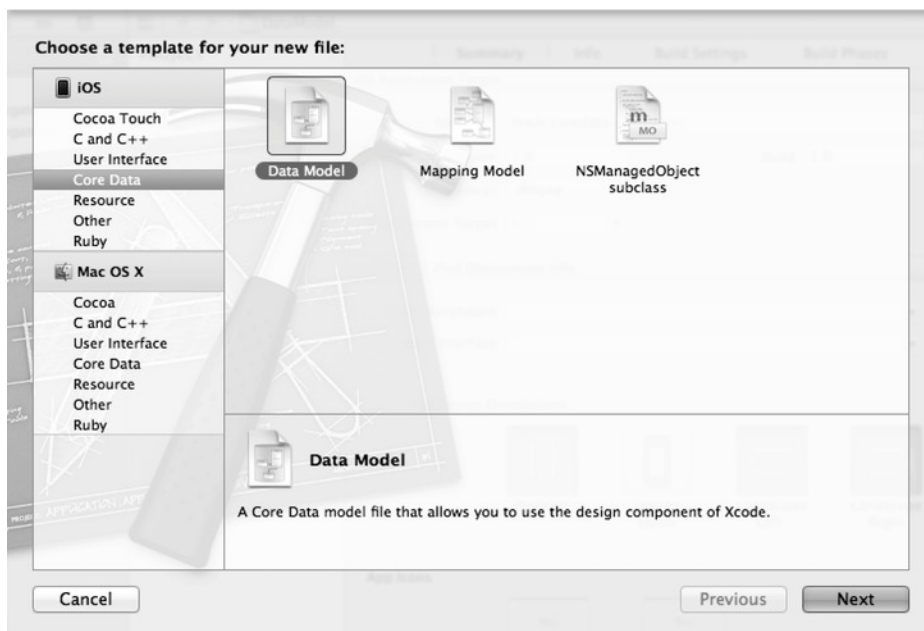
As you create your models in Core Data applications, keep database modeling in mind and consider removing redundancy by moving multiple-value attributes to separate entities. For example, in the League Manager data model, the `uniformColor` attribute of `Team` presents a normalization opportunity. Named colors represent a finite set, despite what Crayola may claim, so you could create an entity called `Color` that has an attribute, `name`, that relates to teams in a to-many relationship.

## Using the Xcode Data Modeler

Some developers simply accept the tools given to them. Others build tools only when they believe the existing tools are inferior. A few, however, stubbornly insist on building all their own tools. We tend to fall into this last category of developers, and our colleagues know us as programmers who use only tools we build ourselves. We take pride in building tools and even hold celebrations after broadcasting the amount of time the tools save. We never tally the time it takes to build the tools, filing that time under "play time" because of our enjoyment for building tools. Surprisingly, however, writing a data modeler tool for Core Data never crossed our minds, probably because Xcode does a reasonable job of allowing you to model data. It's by no means flawless, but it does work, and it's perfectly integrated with the development environment, an attribute

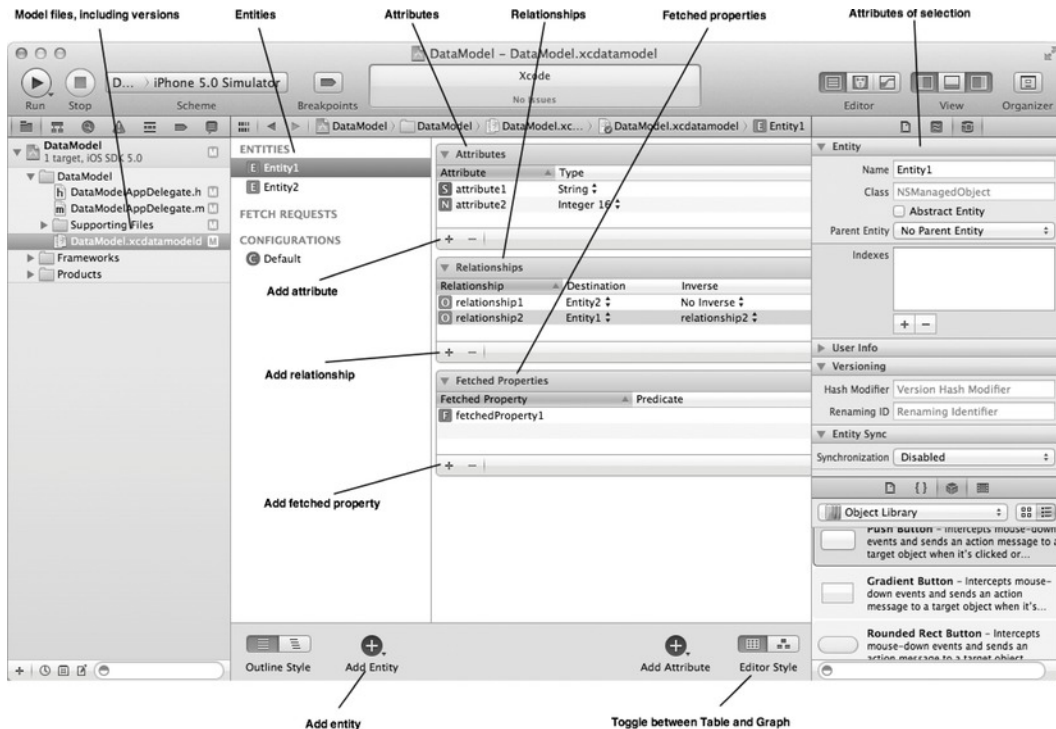
that even we would never be able to surpass with a tool we built. In fact, Xcode ships with a built-in data modeling tool that makes it easy to create data models visually rather than programming the `NSManagedObjectModel` by hand. You've gotten glimpses at this tool several times in the prior chapters. In this section, we spend less time dealing with how Core Data works and a little bit more time looking at the tool and how to use it.

In this section, you won't produce any runnable code. Instead, you will focus on the data modeler's user interface. To add a data model to a project in Xcode, select **File > New > New File** from the menu. In the iOS section on the left, select the Core Data subsection. It reveals three file types: Data Model, Mapping Model, and `NSManagedObject` subclass. Mapping models assist with migrating data across data model versions, which is covered in Chapter 8, and `NSManagedObject` subclasses generate classes from the entities in your models, which is covered in Chapter 5. Select Data Model, as shown in Figure 4–1, and click Next. Pick a name for the data model, and click Save. Xcode creates the data model for you.



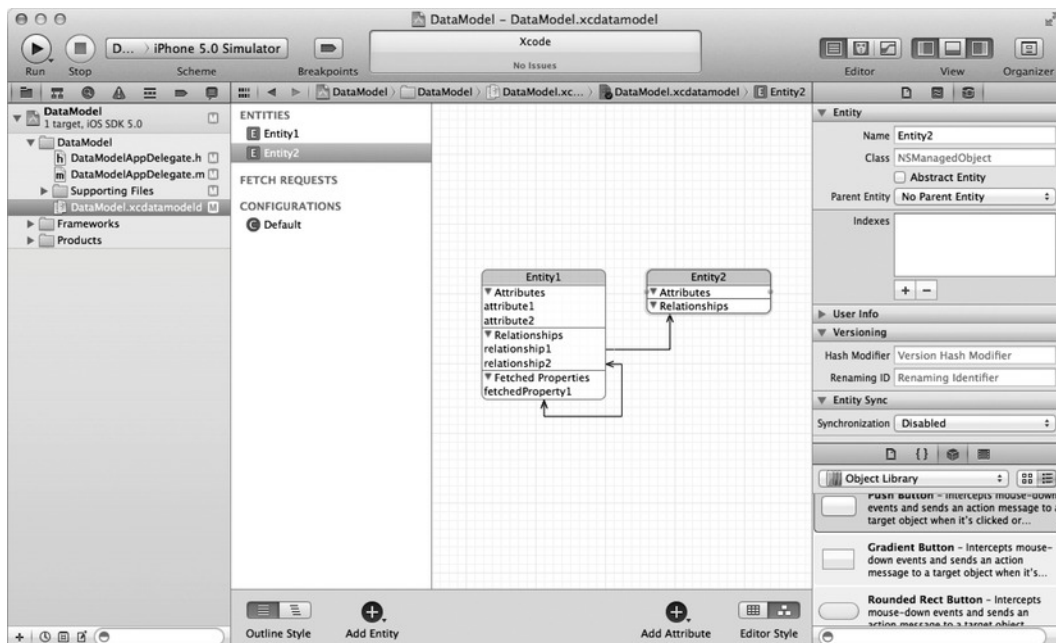
**Figure 4–1.** *New data model file*

Opening your new data model file in Xcode opens it in the data modeler interface. Rich in buttons and options, the Xcode data modeler interface can be confusing. Refer to Figure 4–2, which annotates the user interface to summarize the buttons relevant to data modeling, to understand how to work with the modeler to model your data.



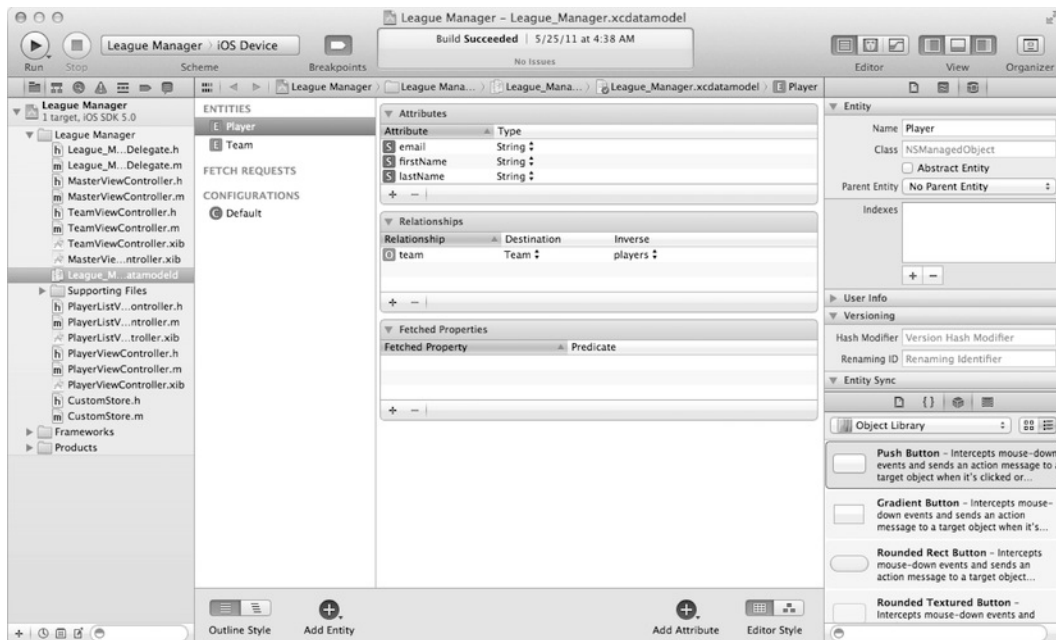
**Figure 4–2.** *The data modeler interface*

The modeler allows you to navigate through the entities, attributes, relationships, fetch requests, and configurations of the data model. You get more details about the chosen artifacts as you select them. You can also toggle between the Table view, shown in Figure 4–2, or an Entity-Relationship-like view that Xcode calls “Graph” view, shown in Figure 4–3, by pressing the corresponding Editor Style button at the bottom of the Xcode window.



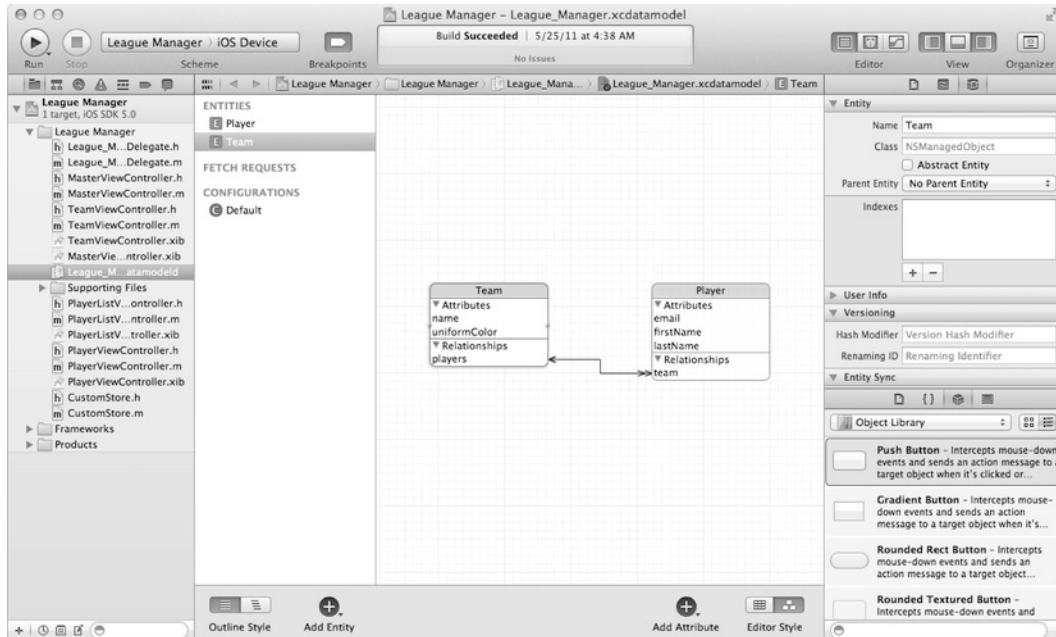
**Figure 4–3.** *The Graph view*

To make the model more interesting to talk about, open the League Manager data model from Chapter 3. The Xcode interface should look similar to Figure 4–4.



**Figure 4–4.** *The League Manager model in the Xcode data modeler*

You can see that Xcode displays the entities and properties in table form by default. You may find working with the graph form, which you get by clicking the Graph View button above Editor Style, easier. The League Manager in Graph View looks like Figure 4–5.



**Figure 4–5.** *The League Manager model in Graph View*

The graph view resembles a traditional entity-relationship diagram familiar to data modelers. In the graph view, you can easily see how the entities relate to each other. You may notice that the relationship from Team to Player is double-headed. That is because it is a one-to-many relationship. A team may have several players. The inverse relationship from Player to Team is single-headed because it is a one-to-one relationship. A player has only one team at most. We tend to find the Table View easier for creating and changing the model and the Graph View better for visualizing the model, but you can develop your own approach.

You can get more information about any entity or property by selecting it. Figure 4–4, for example, shows the Player entity selected. In the upper right, you can see details about the Player entity: it's named Player, it's of class `NSManagedObject`, it has no parent entity, and it isn't abstract. If you select a property, that panel changes to display details for that property, appropriate to its type. If you select, for example, the email attribute for the Player entity, the panel changes to look like Figure 4–6. You'll see the following:

- It's named email.
- It's optional.
- It's not transient or indexed.
- It's of type String.



- It has no minimum or maximum length.
- It has no default value.
- It has no attached regular expression.
- It won't be indexed in Spotlight.
- It won't be stored in an external record.

The screenshot shows a dialog box titled 'Attribute' with a dropdown arrow on the left. Inside, the 'Name' field contains 'email'. Under 'Properties', there are three checkboxes: 'Transient' (unchecked), 'Optional' (checked), and 'Indexed' (unchecked). The 'Attribute Type' is set to 'String' in a dropdown menu. The 'Validation' section has two rows, each with a 'No Value' dropdown and a checkbox for 'Min Length' and 'Max Length' respectively, all of which are unchecked. The 'Default Value' field contains 'Default Value'. The 'Reg. Ex.' field contains 'Regular Expression'. At the bottom, under 'Advanced', there are two checkboxes: 'Index in Spotlight' (unchecked) and 'Store in External Record File' (unchecked).

**Figure 4–6.** Details for the email attribute

Switching the selection to the team relationship of Player changes the panel to show relationship-oriented detail, as shown in Figure 4–7. From this view, you learn the following about the relationship:

- It's named team.
- It has a destination of Team.
- It has the inverse relationship named players.
- It's optional but not transient.
- It isn't a to-many relationship.
- It has a minimum count of 1 and a maximum count of 1.
- It uses the delete rule called Nullify.
- It won't be indexed in Spotlight.
- It won't be stored in an external record.



**Relationship**

Name: team

Destination: Team

Inverse: players

Properties: ☐ Transient ☒ Optional

Arranged: ☐ Ordered

Plural: ☐ To-Many Relationship

Count: 1 ☒ Minimum 1 ☒ Maximum

Delete Rule: Nullify

Advanced: ☐ Index in Spotlight ☐ Store in External Record File

**Figure 4-7.** Details for the team relationship

Let's look at the details for the Player entity shown in Figure 4-8. Entities are defined by their names. Core Data also allows you to subclass `NSManagedObject` to provide an alternate implementation class for your managed objects. If you create a subclass of `NSManagedObject` and want an entity mapped to that class, then you can specify the class name in the entity details pane.

**Entity**

Name: Player

Class: NSManagedObject

☐ Abstract Entity

Parent Entity: No Parent Entity

**Figure 4-8.** Entity details pane

## Viewing and Editing Attribute Details

Select an attribute from the Attributes section in the middle column of Xcode's data modeling tool in order to switch the detail pane to display attribute details.

Table 4-1 explains the fields in the Attribute pane. The first column, Name, refers to the name of the field, while the Description column explains what this field means and how it's used.

Table 4–1. What the Different Properties Mean

| Name           | Description  |
|----------------|--|
| Name           | The name of the attribute.   |
| Transient      | Tells Core Data to not persist this attribute. This is useful in conjunction with a second attribute to support nonstandard or custom types. Custom types are discussed later in this chapter.                               |
| Optional       | An optional attribute may have a nil value. Nonoptional attributes must have a non-nil value.  |
| Indexed        | Indexed attributes are faster to search but take up more space in the persistent store.  |
| Attribute Type | The type of the attribute. The value you select in this column can change which validation fields display.   |
| Validation     | Used with the Min Length/Max Length fields for an attribute of type String, or the Minimum/Maximum fields for an attribute of a numerical type. The values for the minimum and maximum length (String) or value (numerical). |
| Min Length     | Check box to turn on the Min Length validation.  |
| Max Length     | Check box to turn on the Max Length validation.  |
| Minimum        | Check box to turn on the Minimum validation.   |
| Maximum        | Check box to turn on the Maximum validation.   |
| Default Value  | A default value for this attribute when you provide no value for it.   |
| Reg. Ex.       | Regular expression used for validating data.   |

## Viewing and Editing Relationship Details

In a similar manner, you can click a relationship to view its details.

Relationships have several properties. First, they have a name and a destination entity. The destination describes the type of objects the relationship points to. For a Player entity, the “team” relationship points, as expected, to a Team entity. The Core Data architects strongly recommend that every relationship have an *inverse relationship*—a relationship that goes in the opposite direction. For example, Player has a relationship to Team, so it is recommended, as we have implemented, that Team have a relationship to Player as well. In fact, if you don’t specify an inverse relationship, the compiler will generate a warning message.

Two additional properties are available for attributes: Transient and Optional. A transient relationship does not get stored to the persistent store during the save: operation. If the team relationship were transient, then the information of which team a player belongs to would not be saved and would be lost when the application exits. The Player object itself would still be persisted, but the relationship with a team would not. Transient relationships can be useful if you want to set values at runtime, such as a password or something derived with code from other information, but don’t want to

store the relationship fact using Core Data. When a relationship is optional, it simply means that you may give it a `nil` value. For example, a player may not have a team.

The plurality of a relationship defines how many destination objects this source object can relate to: one or many. By default, a relationship is to-one, which means that the source object can relate to at most one destination object. This is the case for the “team” relationship of the `Player` entity: a player can have at most only one team. But a team can have multiple players; therefore, the “player” relationship of the `Team` entity is a to-many relationship. Its value is represented by a set of entities. The plurality of a relationship can also be further specified using a minimum and maximum value, which represents the cardinality of the destination entities for that relationship.

Finally, Core Data uses the delete rule to understand what to do with destination objects when a source object is deleted. Core Data can do different things to the players of a team if the team is deleted. The options are No Action, Nullify, Cascade, and Deny. The “Setting Rules” section in this chapter discusses the delete rules and what they mean.

## Using Fetched Properties

So far in this book, we have mentioned attributes and relationships on several occasions. A third property can be attached to an entity: a fetched property. Fetched properties are comparable to relationships in that they allow an object to have references to other objects. While relationships directly refer to destination objects, however, fetched properties identify the destination objects by using a predicate. Fetched properties behave like the smart playlists in iTunes in which the user specifies a source list (usually the music library) and then specifies some criteria to filter the results. Fetched properties behave differently from smart playlists, however, in that they’re evaluated the first time they are called, and their results are cached until you tell them to reevaluate themselves using the `refreshObject:mergeChanges:` method.

In the League Manager application, you could, for example, create a fetched property in the `Team` entity to return all the players whose last name begins with the letter *W*. Select the `Team` entity, click the + button below the Fetched Properties section, and call the new fetched property **wPlayers**. Select `Player` as the destination entity (that is, the Entity type the fetched property will retrieve).

In the Predicate field, specify the criteria for filtering the players using the standard `NSPredicate` format (see Chapter 6 for more details on `NSPredicate`). In this example, the predicate would be as follows:

```
lastName BEGINSWITH "W"
```

This fetched property, then, becomes a named property on the `Player` entity, just as all the attributes and relationships are, and you access them the same way. You can test this by adding code to the application delegate in League Manager’s `application:didFinishLaunchingWithOptions:` method that uses the `wPlayers` fetched property. The code in Listing 4–1 fetches all the teams, grabs the first team (if one exists), and then grabs all the players in the `wPlayers` fetched property. It then iterates through those players and prints their first and last names.

**Listing 4–1. Using the *wPlayers* Fetched Property**

```

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:[NSEntityDescription entityForName:@"Team"
    inManagedObjectContext:self.managedObjectContext]];
NSArray *teams = [self.managedObjectContext executeFetchRequest:fetchRequest error:nil];

if ([teams count] > 0)
{
    NSManagedObject *team = [teams objectAtIndex:0];
    NSSet *wPlayers = [team valueForKey:@"wPlayers"];
    for (NSManagedObject *wPlayer in wPlayers)
    {
        NSLog(@"%@ %@", [wPlayer valueForKey:@"firstName"], [wPlayer
valueForKey:@"lastName"]);
    }
}

```

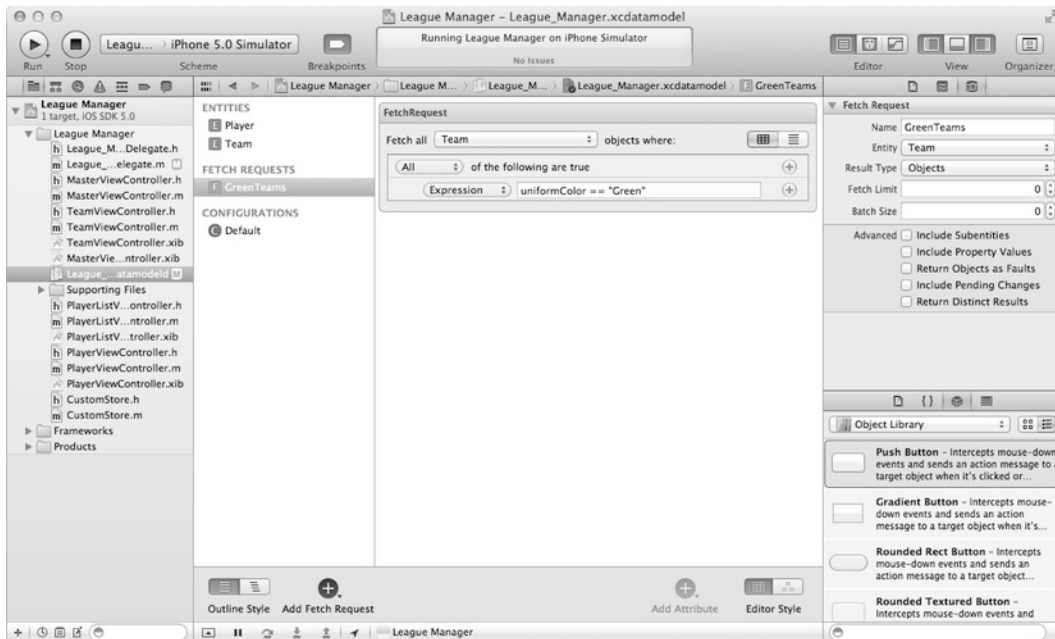
If you run this code, depending on what data you've entered in League Manager, you should see something like the following in the console:

```

2011-07-31 19:08:23.005 League Manager[8039:f203] Dwyane Wade
2011-07-31 19:08:23.006 League Manager[8039:f203] Tyson Warner

```

The last important aspect of configuring a data model with Xcode is creating fetch requests. You use fetch requests to retrieve managed objects from the data model, just like you would use `NSFetchRequest` in code, except you can predefine them here. To create a fetch request that will retrieve all the teams with a green uniform, for example, select **Team** in the Entity section, click and hold the **Add Entity** button until the drop-down menu appears, and select **Add Fetch Request**. Call the fetch request **GreenTeams**. In the middle column of Xcode you should see a + button to add criteria to the fetch request. Click that button and type **uniformColor == "Green"**. It should look like Figure 4–9.



**Figure 4–9.** Setting up the predicate for a fetch request

Fetch requests created this way are stored in the data model and can be retrieved using the `fetchRequestTemplateName:` method of the `ManagedObjectModel` class. You can add code to the `application:didFinishLaunchingWithOptions:` method of League Manager's application delegate to test this fetch request. Listing 4–2 shows code to do that.

**Listing 4–2.** Using the Fetch Request Created in the Model

```
NSFetchRequest *fetchRequest = [self.managedObjectModel
fetchRequestTemplateName:@"GreenTeams"];
NSArray *teams = [self.managedObjectContext executeFetchRequest:fetchRequest error:nil];
for (NSManagedObject *team in teams)
{
    NSLog(@"%@", [team valueForKey:@"name"]);
}
```

Again, depending on what data you've entered into League Manager, your console output should look like this:

```
2011-07-31 19:17:25.598 League Manager[8184:f203] Boston Celtics
```

## Creating Entities

We have been talking extensively about entities up to now. Entities describe the attributes and relationships of a managed object. To add an entity in Xcode, you simply click the Add Entity button below the Entity section in Xcode, and give your entity a name.

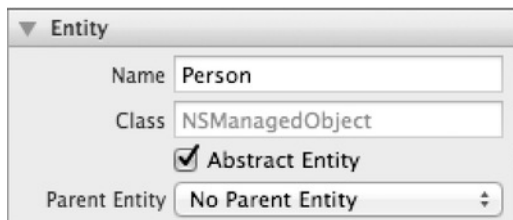
By default, instances of each entity are represented by objects of type `NSManagedObject` at runtime, but as your project evolves and matures, you typically opt for creating your own classes to represent your managed objects. Your custom managed object must extend `NSManagedObject`, and you must specify the custom object class name in the Class field of the Entity details pane, as shown in Figure 4–10.



The screenshot shows the 'Entity' details pane in Xcode. The 'Name' field is set to 'Player'. The 'Class' field is set to 'MyPlayer'. The 'Abstract Entity' checkbox is unchecked. The 'Parent Entity' dropdown menu is set to 'No Parent Entity'.

**Figure 4–10.** *A custom managed object class*

One topic we have not yet addressed is entity inheritance. Very similar to class inheritance in the object-oriented programming paradigm, entity inheritance allows you to create entities that inherit their properties from another entity and can add some of their own. For example, you could create a `Person` entity, and the `Player` entity could inherit from it. If you had other representations of people—`Coach`, for example—you could inherit from the `Person` entity. If you wanted to prevent the parent entity from being instantiated directly, preventing the user from creating a `Person` object as a stand-alone object, you could make the `Person` entity abstract. For example, you could create a `Person` entity that has a `dateOfBirth` attribute. Figure 4–11 shows how a `Person` entity could be configured. Note how it is made abstract to prevent direct instantiation.



The screenshot shows the 'Entity' details pane in Xcode. The 'Name' field is set to 'Person'. The 'Class' field is set to 'NSManagedObject'. The 'Abstract Entity' checkbox is checked. The 'Parent Entity' dropdown menu is set to 'No Parent Entity'.

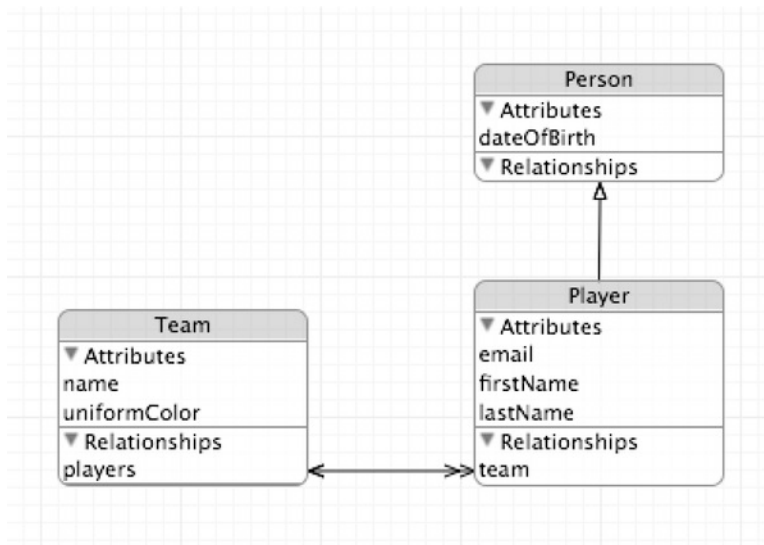
**Figure 4–11.** *The Person entity as an abstract entity*

The next step is to change the `Player` entity configuration and set its parent entity to `Person`. Select the `Player` entity, and change the parent field to `Person`, as in Figure 4–12. The `Player` entity now inherits from the `Person` entity, as illustrated in the graph view in Figure 4–13. This means that a player managed object also has a `dateOfBirth` attribute that can be set and read.

Entity configuration panel showing:

- Name: Player
- Class: NSObject
- ☐ Abstract Entity
- Parent Entity: Person

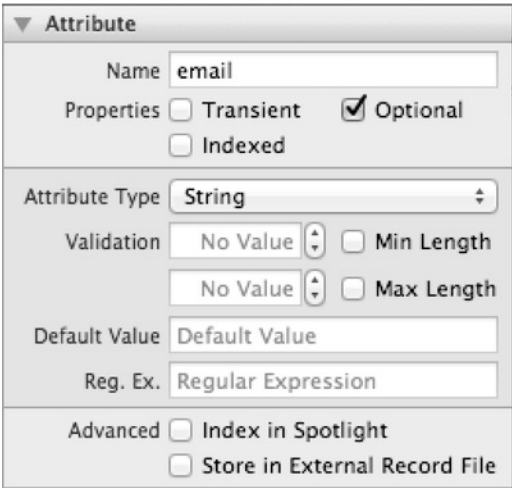
**Figure 4–12.** The *Player* entity configured to inherit from the *Person* entity



**Figure 4–13.** Graph view of the *Player* entity configured to inherit from the *Person* entity

## Creating Attributes

Attributes describe an entity. Similar to the attributes of an object in object-oriented programming, entity attributes define the current state of an entity and may be represented by various data types. Select an entity and click the + button under the Attributes heading to create a new attribute for the selected entity. Figure 4–14 shows the different options for configuring an attribute.



**Figure 4–14.** Attribute details pane

Earlier in this chapter, we talked about the meaning of the different properties of attributes. See Table 4–1 to review the properties. Arguably the most critical property of an attribute is its type. Several types are available by default in Core Data, as shown in Table 4–2.

**Table 4–2.** Available Attribute Types

| Xcode Attribute Type | Objective-C Attribute Type | Objective-C Data Type | Description                    |
|----------------------|----------------------------|-----------------------|--------------------------------|
| Integer 16           | NSInteger16AttributeType   | NSNumber              | A 16-bit integer               |
| Integer 32           | NSInteger32AttributeType   | NSNumber              | A 32-bit integer               |
| Integer 64           | NSInteger64AttributeType   | NSNumber              | A 64-bit integer               |
| Decimal              | NSDecimalAttributeType     | NSDecimalNumber       | A base-10 subclass of NSNumber |
| Double               | NSDoubleAttributeType      | NSNumber              | An object wrapper for double   |
| Float                | NSFloatAttributeType       | NSNumber              | An object wrapper for float    |
| String               | NSStringAttributeType      | NSString              | A character string             |



| Xcode Attribute Type | Objective-C Attribute Type   | Objective-C Data Type | Description                                |
|----------------------|------------------------------|-----------------------|--|
| Boolean              | NSBooleanAttributeType       | NSNumber              | An object wrapper for a Boolean value      |
| Date                 | NSDateAttributeType          | NSDate                | A date object                              |
| Binary Data          | NSBinaryDataAttributeType    | NSData                | Unstructured binary data                   |
| Transformable        | NSTransformableAttributeType | Any nonstandard type  | Any type transformed into a supported type |

Most of these types are straightforward because they map directly to an Objective-C data type. Using them is just a matter of leveraging the key/value accessors of the `NSManagedObject` instances. The Transformable type is the only type that doesn't have an Objective-C counterpart. The Transformable type is used for custom types, which are types that aren't supported by Core Data. Use Transformable as the type when the attribute you're trying to persist doesn't fit neatly into one of the supported data types. For example, if your managed object needs to represent a color and needs to persist it, it would make sense to use the `CGColorRef` type. In this case, you need to set the Core Data type to Transient and Transformable and provide Core Data with the mechanism for transforming the attribute into supported data types. Use custom attributes when you create your own managed objects that extend from `NSManagedObject`; they are covered in more detail in the next chapter.

## Creating Relationships

As you normalize your data model, you'll likely create several entities, depending on the complexity of the data you're modeling. Relationships allow you to tie entities together, as the League Manager application does with teams and players. Core Data allows you to tune the relationships you create to accurately reflect how your data relates to each other.

In Xcode, when you create a Core Data relationship or select an existing one, you see a set of options that you can modify to change the nature of the relationship. For example, if you select the "team" relationship from the Player entity in the League Manager data model, you see something that looks like Figure 4-15. The options in that panel allow you to configure fields of the relationship. These fields are as follows:

- Name
- Destination
- Inverse

- Transient
- Optional
- To-Many Relationship
- Count (Minimum and Maximum)
- Delete Rule

This next sections walk through all these fields, explaining what they mean and how they impact the relationship so that you can set up your relationships correctly in your data models.

The screenshot shows a 'Relationship' panel with the following settings:

- Name:** team
- Destination:** Team
- Inverse:** players
- Properties:**
  - ☐ Transient
  - ☒ Optional
- Arranged:**
  - ☐ Ordered
- Plural:**
  - ☐ To-Many Relationship
- Count:**
  - Minimum: 1 (checked)
  - Maximum: 1 (checked)
- Delete Rule:** Nullify
- Advanced:**
  - ☐ Index in Spotlight
  - ☐ Store in External Record File

**Figure 4–15.** *Relationship panel*

## Name

The first field, Name, becomes the name of the key `NSManagedObject` uses to reference the relationship. By convention, it's the name of the referenced entity in lowercase. In the case of a to-many relationship, the Name field conventionally is pluralized. Note that these guidelines are conventions only, but you will make your data models more understandable, maintainable, and easier to work with if you follow them. You can see that the League Manager data model follows these conventions—in the `Player` entity, the relationship to the `Team` entity is called “team.” In the `Team` entity, the relationship to the `Player` entity, a to-many relationship, is called “players.”

## Destination and Inverse

The next field, Destination, specifies the entity at the other end of the relationship. The Inverse field allows you to select the same relationship, but from the destination entity's perspective. To set this properly, create the relationship in both related entities, set the names and destinations in each entity, and then select the proper inverse relationship in one entity. Core Data figures out the inverse relationship for the other entity and sets it for you.

Leaving the Inverse field with the value No Inverse Relationship gives you two compiler warnings: consistency error and misconfigured property.

You can ignore these warnings and run your application without specifying an inverse relationship—after all, these are compiler warnings, not compiler errors—but you could face unexpected consequences. Core Data uses bidirectional relationship information to maintain the consistency of the object graph (hence the Consistency Error) and manage undo and redo information. If you leave a relationship without an inverse, you imply that you will take care of the object graph consistency and undo/redo management. The Apple documentation strongly discourages this, however, especially in the case of a to-many relationship. When you don't specify an inverse relationship, the managed object at the other end of the relationship isn't marked as changed when the managed object at this end of the relationship changes. Consider the example of a `Player` object being deleted and having no inverse relationship back to its team. Any optionality rules or delete rules aren't enforced when the object graph is saved.

## Transient

The Transient check box allows you to specify that a relationship should not be saved to the persistent store. A transient relationship still has support for undo and redo operations but disappears when the application exits, giving the relationship the life span of a Hollywood marriage. Here are some possible uses for transient relationships:

- Relationships between entities that are temporal and shouldn't survive beyond the current run of the application.
- Relationship information that can be derived from some external source of data, whether another Core Data persistent store or some other source of information.

In most cases, you'll want your relationships to be persistent, and you'll leave the Transient check box unchecked.

## Optional

The next field, Optional, is a check box that specifies whether this relationship requires a non-nil value. Think of this like a nullable versus non-nullable column in a database. If you select the Optional check box, a managed object of this entity type can be saved to

the persistent store without having anything specified for this relationship. If not selected, however, saving the context will fail. If the `Player` entity in the data model for League Manager, for example, left this check box unchecked, every player entity would have to belong to a team. Setting the “team” value for a player to `nil` (or never setting it at all) would cause all calls to `save:` to fail with the error description “team is a required value.”

## To-Many Relationship

To-Many Relationship is another check box. Leaving this check box unchecked makes this relationship to-one, which means this relationship has exactly one managed object (or zero, depending on optionality settings) on each end of the relationship. For the League Manager application, unchecking the To-Many Relationship check box for the “players” relationship on the Team entity would mean that only one player could belong on a team, which might work for golf but not soccer. Selecting this check box means the destination entity can have many managed objects that relate to each managed object instance of this entity type.

## Count (Minimum and Maximum)

The next fields, Count, Minimum, and Maximum, set limits on the number of managed objects of the destination entity type that can relate to each managed object of this entity type and take effect only if the To-Many Relationship check box is selected. Unchecking that box to make the relationship to-one resets both the Minimum and Maximum values to 1.

In a to-many relationship, you can use the Count fields to set limits on the number of managed objects to which this managed object can relate. Exceeding the limits causes the `save:` operation to fail with a “Too many items” or a “Too few items” error message. Note that the Optional setting overrides the Minimum Count setting; if Optional is checked and Min Count is 1, saving the context when the relationship count is zero succeeds. Note also that Core Data doesn’t require that you set these values intelligently. You can, for example, set Minimum Count higher than Maximum Count. If you do this, calls to `save:` will always fail because you can never meet the Count criteria set. You’ve probably been in relationships yourself that seem this way.

## Delete Rule

The Delete Rule setting allows you to specify what happens if you try to delete a managed object of this entity type. Table 4–3 lists the four possibilities and what they mean.

**Table 4–3. Delete Rule Options**

| Rule      | Description  |
|-----------|--|
| Cascade   | The source object is deleted, and any related destination objects are also deleted.  |
| Deny      | If the source object is related to any destination objects, the deletion request fails, and nothing is deleted.              |
| Nullify   | The source object is deleted, and any related destination objects have their inverse relationships set to <code>nil</code> . |
| No Action | The source object is deleted, and nothing changes in any related destination objects.  |

The League Manager application, for example, sets the Delete Rule for the “players” relationship in the Team entity to Cascade so that deleting a team would delete all the players related to it. If you set that rule to Deny and a team had any players, trying to delete the team would result in an error on any attempts to save the context with the message “team is not valid.” Setting the Delete Rule to Nullify would preserve the players in the persistent store, though they would no longer belong to any team.

The No Action option represents another way, such as not specifying an inverse relationship, that Core Data allows you to accept responsibility for managing object graph consistency. If you specify this value for the Delete Rule, Core Data allows you to delete the source object but pretends to the destination objects that the source object still exists. For the League Manager application, this would mean disbanding a team but still telling the players to show up for practices and games. You won’t find many compelling reasons to use the No Action setting, except perhaps for performance reasons when you have many destination objects. Chapter 7 discusses performance tuning with Core Data.

## Summary

For applications that depend on data, which means most applications, the importance of designing your data model correctly cannot be overstated. The Xcode data modeler provides a solid interface for designing and defining your data models, and in this chapter, you learned to navigate the many options available as you create entities, attributes, and relationships. Make sure to understand the implications of how you’ve configured your data model so that your applications can run efficiently and correctly.

In most cases, you should avoid options in your model that wrest control from Core Data and make you manage the consistency of the object graph yourself. Core Data will almost always do a better job of managing object graph consistency than you will. When you must take control, make sure to debug your solutions well to prevent application crashes or data corruption.

One feature of Xcode that you may find helpful is the ability to print your data model for review. You'll find that Xcode can print your data model to any printer that Mac OS X supports.

In the next chapter, you'll learn how to use the data models you create by actually creating, retrieving, updating, and deleting data in them.

## Working with Data Objects

The ability to easily interact with data in a database ranks high among the features that drive the success of most new programming languages and frameworks. This chapter talks about working with the data objects you create and store in your Core Data models. You'll find Core Data provides an abstraction from the complexities of Structured Query Language (SQL) and database programming that allows you to quickly and efficiently read and write data to your persistent stores.

### Understanding CRUD

Whether you claim the *R* stands for Read or Retrieve and whether you side with Destroy over Delete for the *D*, the acronym CRUD describes the four operations you perform on a persistent data store:

- Create
- Retrieve (Read)
- Update
- Delete (Destroy)

These four operations apply to all persistent store interaction, whether you're working with Core Data in an iOS application, an Oracle database in a Java Enterprise Web application, a Virtual Storage Access Method (VSAM) file in a COBOL program, or any other situation in which you have a program that works with data. In fact, the CRUD concept has been extended to describe other situations in which you create, retrieve, update, and delete data. For example, the latest approach to providing services over the Web, Representational State Transfer (REST), has been called CRUD for the Web; the HTTP verbs map to the CRUD operations like this:

- POST = Create
- GET = Retrieve
- PUT = Update
- DELETE = Delete

In this chapter, you'll build one application twice: once working directly with `NSManagedObject` instances and the second time with custom classes that extend `NSManagedObject`. This section builds the raw `NSManagedObject` version. With a nod to Scott Hanselman's Baby Smash! ([www.hanselman.com/babysmash/](http://www.hanselman.com/babysmash/)), this application, called *Shapes*, has the following requirements and characteristics:

- Each time someone taps the screen, a random shape appears where the screen was tapped.
- The shape is randomly a circle or a polygon.
- If a polygon, the shape has a random number of sides and can be concave, convex, or mixed.
- If a circle, the shape has a random radius.
- Shapes appear in random colors.
- Shaking the device deletes all the shapes.
- Rotating the device updates all the shapes to random colors.
- The screen splits down the middle, and each shape appears twice, once on each screen half.
- One of the screen halves is zoomed at 2x magnification (meaning some shapes won't appear, because they'll fall outside the screen's boundaries).
- *Shapes* is built for the iPad to take advantage of the extra screen space.

From a Core Data perspective, *Shapes* illustrates the following:

- *Create*: Each time you tap the screen, *Shapes* creates a shape object in the persistent store.
- *Retrieve*: Each time the screen draws the shapes, *Shapes* retrieves the shapes from the persistent store.
- *Update*: Each time you rotate the device, *Shapes* updates all the shapes in the persistent store with different random colors.
- *Delete*: Each time you shake the device, *Shapes* deletes all the shapes from the persistent store.
- *Inheritance*: The *Polygon* and *Circle* entities inherit from a common parent, *Shape*.



- *One-to-many*: One Polygon instance relates to many Vertex instances.
- *Many-to-many*: Many Shape instances relate to many Canvas instances.
- *One-to-one*: One Canvas instance relates to one Transform instance, which controls scaling.

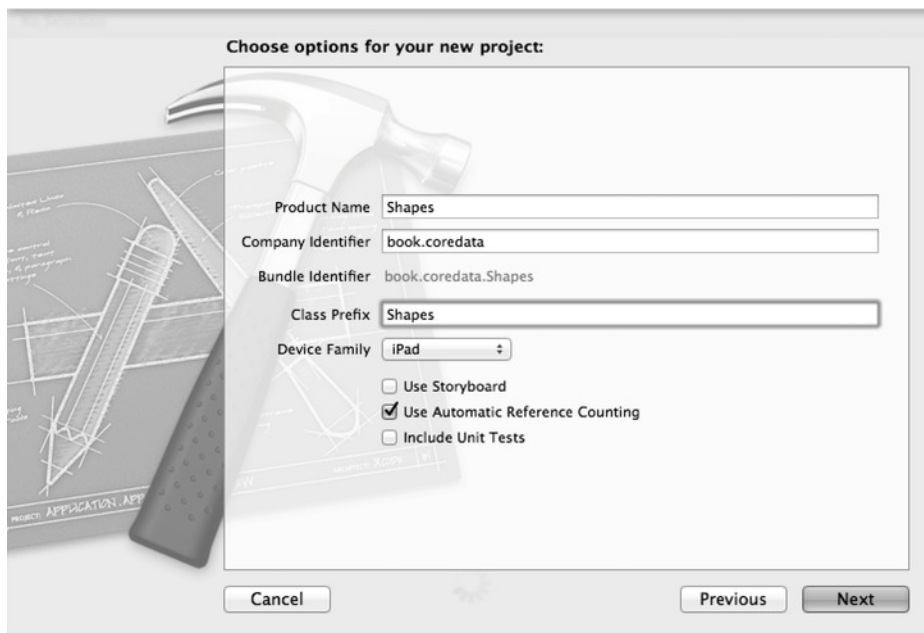
The finished application will look like Figure 5–1.



**Figure 5–1.** *The finished Shapes application*

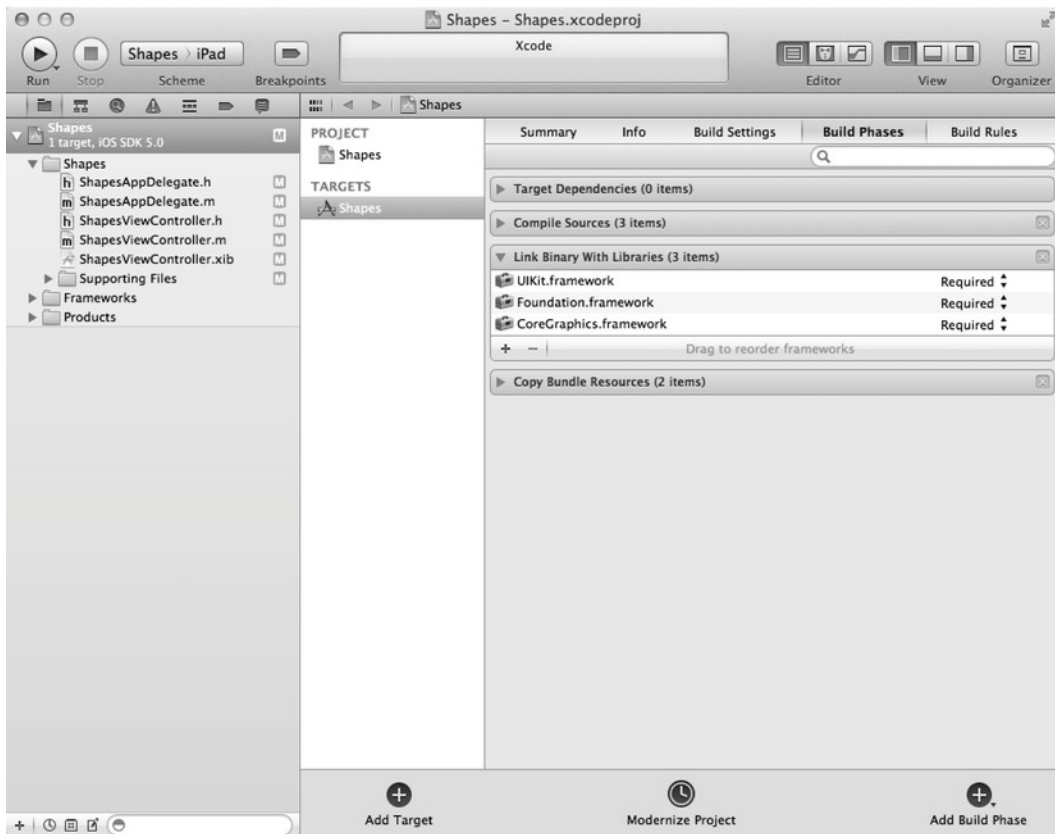
Shapes probably can't compete with the fun of BabySmash!, but we don't want to encourage smashing iPads anyway. Shapes illustrates a wide range of Core Data fundamentals. You'll notice, however, that Shapes doesn't filter or sort results, which Chapter 6 covers.

In Xcode, create a new project and select a Single View Application under iOS. Call it Shapes, enter book.coredata for the Company Identifier, and set Device Family to iPad (as Figure 5–2 shows). Save the project to open it in Xcode.



**Figure 5–2.** New iPad project called *Shapes*

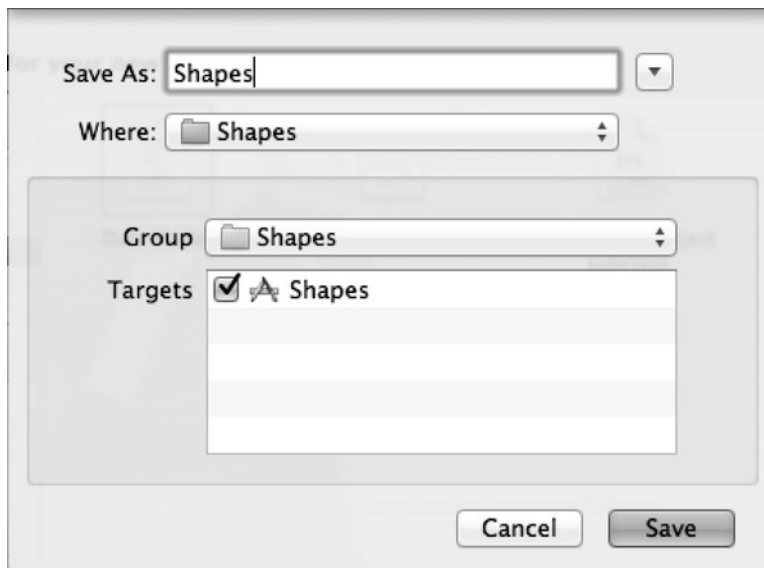
Notice that you had no opportunity in the Single View Application template to add Core Data to your application, so add Core Data now. You learned how to do that in Chapter 1, but to recap, select the Shapes project on the left, go to the Build Phases tab, and expand the Link Binary with Libraries section, as Figure 5–3 shows. Click the + button under that section, select CoreData.framework, and click Add. CoreData.framework appears in your project's file tree on the left side of Xcode. Drag it to the Frameworks folder.



**Figure 5–3.** Preparing to add Core Data to the Shapes application

## Creating the Shape Application Data Model

Before writing the code for the application, create your data model. Select **File** ► **New** ► **New File**, select **Core Data** under **iOS** on the left and **Data Model** on the right, and click **Next**. Call the model file `Shapes.xcdatamodel1` and save it to the **Shapes** folder in the **Shapes** group, as shown in Figure 5–4.



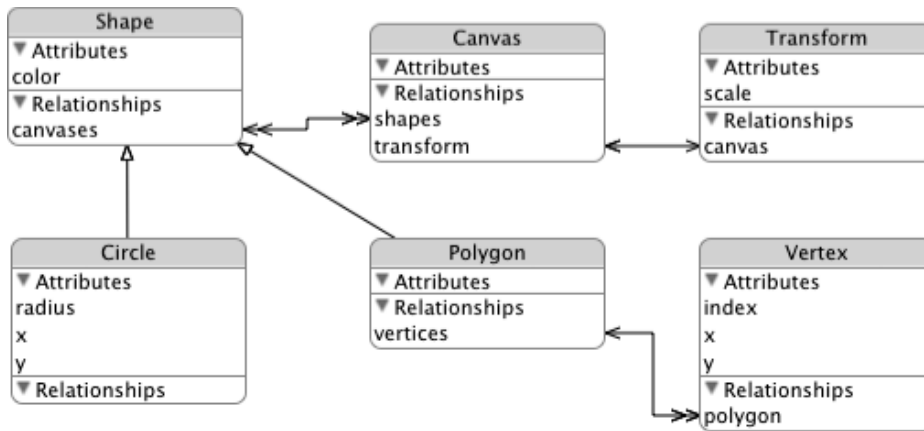
**Figure 5–4.** *Creating your Shapes data model*

Select `Shapes.xcdatamodeld` to edit your data model, and follow these steps to finalize the model:

1. Add an entity called `Shape`, and give it a nonoptional attribute called `color` of type `String`.
2. Add an entity called `Circle`, and give it three nonoptional attributes, all of type `Float`: `radius`, `x`, and `y`. Set its Parent Entity to `Shape`.
3. Add an entity called `Polygon` with no attributes, and set its Parent Entity to `Shape`.
4. Add an entity called `Vertex` with a nonoptional attribute called `index` of type `Integer 16`. Give it two nonoptional `Float` attributes called `x` and `y`. Add a relationship called `polygon`, and set Destination to `Polygon`. Make it optional, and leave the Delete Rule as `Nullify`.
5. Add a relationship to the `Polygon` entity called `vertices`, and set its Destination to `Vertex` and Inverse to `polygon`. Uncheck the `Optional` check box, check the `To-Many Relationship` check box, set `Min Count` to 3, and set Delete Rule to `Cascade`.
6. Add an entity called `Transform`, and give a nonoptional attribute called `scale` of type `Float`.

7. Add an entity called Canvas, and give it a relationship called transform. Set Destination to Transform, uncheck the Optional check box, and set Delete Rule to Cascade. Add another relationship called shapes. Set Destination to Shape, leave Optional checked, and check the To-Many Relationship check box. Leave Delete Rule as Nullify.
8. Select the Shape entity, and add a relationship called canvases. Set Destination to Canvas and Inverse to shapes. Check Optional and To-Many Relationship, and set Delete Rule to Nullify.
9. Select the Transform entity, and add a relationship called canvas. Set Destination to Canvas and Inverse to transform. Uncheck Optional, and set Delete Rule to Deny.

When you finish, your data model layout should look like Figure 5–5.



**Figure 5–5.** *The Shapes application data model*

With your data model in place, you're ready to add Core Data support to your application delegate. Open `ShapesAppDelegate.h`, import the Core Data headers, and add properties for your managed object context, managed object model, and persistent store coordinator. Also, add declarations for two methods: one to save the managed object context, and one to return your application's document directory. When you're done, your file should look like the following, with the added lines in bold:

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

@class ShapesViewController;

@interface ShapesAppDelegate : NSObject <UIApplicationDelegate>
@property (strong, nonatomic) IBOutlet UIWindow *window;
@property (strong, nonatomic) IBOutlet ShapesViewController *viewController;
@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
```

```
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;
```

```
- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;
```

```
@end
```

Next, open `ShapesAppDelegate.m`, and add `@synthesize` lines for the three Core Data-related properties you just added. Also add accessors for the same three Core Data-related properties. Finally, add implementations for your methods to save the managed object context and to return the application's document directory. Your `ShapesAppDelegate.m` file should match Listing 5-1.

**Listing 5-1.** *ShapesAppDelegate.m*

```
#import "ShapesAppDelegate.h"
#import "ShapesViewController.h"

@implementation ShapesAppDelegate

@synthesize window=_window;
@synthesize viewController=_viewController;
@synthesize managedObjectContext=__managedObjectContext;
@synthesize managedObjectModel=__managedObjectModel;
@synthesize persistentStoreCoordinator=__persistentStoreCoordinator;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[ShapesViewController alloc]
initWithNibName:@"ShapesViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

#pragma mark - Core Data stack

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

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return __managedObjectContext;
}
```

```

- (NSManagedObjectModel *)managedObjectModel {
    if (__managedObjectModel != nil) {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"Shapes"
withExtension:@"momd"];
    __managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}

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

    NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Shapes.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;
}

- (void)saveContext {
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil) {
        if ([managedObjectContext hasChanges] && ![managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
}

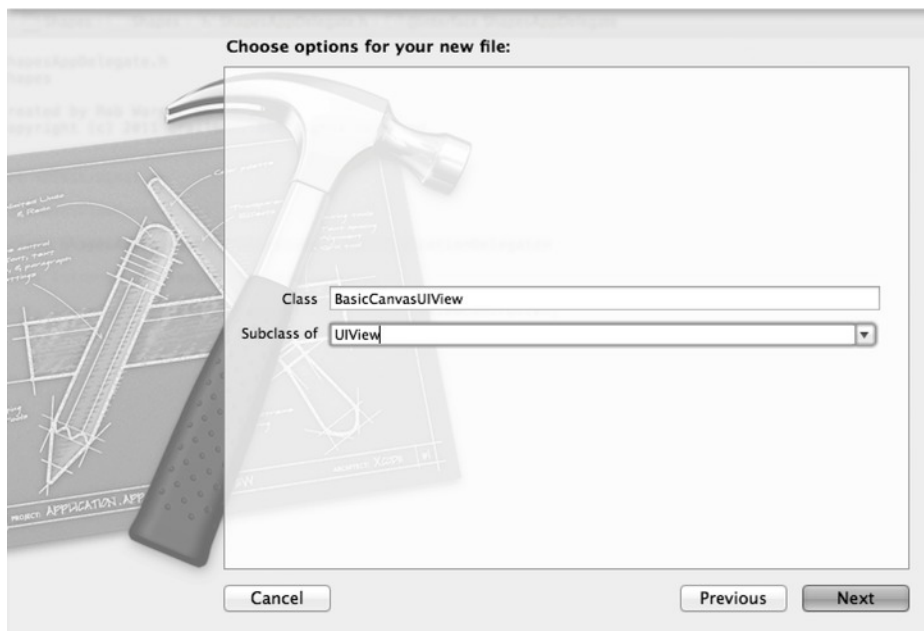
#pragma mark - Application's Documents directory

- (NSURL *)applicationDocumentsDirectory {
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];
}

@end

```

According to the requirements, the Shapes application divides the screen in half and shows the same shapes on both halves, with the shapes doubled in size on one half. The Canvas entity represents these screen halves in the Core Data data model, but you need corresponding user interface elements to actually draw the shapes on the screen. To accomplish this, you'll create a class derived from UIView and then add two of these views to your existing XIB file. Start by creating the class: select **File > New > New File**, select Cocoa Touch Class on the left and Objective-C class on the right, and click **Next**. Call the class `BasicCanvasUIView` and make it a subclass of `UIView`, as shown in Figure 5–6. You should see the two files for the class, `BasicCanvasUIView.h` and `BasicCanvasUIView.m`, listed in the files for your project. Open the `BasicCanvasUIView.h` file.



**Figure 5–6.** Deriving the `BasicCanvasUIView` class from `UIView`

In `BasicCanvasUIView`, you want a reference to the corresponding Canvas entity that this view will display, as well as a way to scale the view (one instance will have a 1x scale, while the other will have a 2x). Change `BasicCanvasUIView.h` to look like the following:

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

@interface BasicCanvasUIView : UIView {
    NSManagedObject *canvas;
}
@property (nonatomic, retain) NSManagedObject *canvas;

-(float)scale;

@end
```



In the implementation file for your class, `BasicCanvasUIView.m`, you need an accessor and a mutator for the canvas property, so add this line:

```
@synthesize canvas;
```

The `scale:` method returns the value stored for the `scale` attribute in the `Transform` entity that relates to the `Canvas` entity that this view represents. Remember that `Canvas` has a one-to-one relationship with `Transform`; each `Canvas` instance has a corresponding `Transform` instance. To get the value for `scale`, you use the `Retrieve` action of `CRUD`. You first retrieve the `Transform` managed object from the `Canvas` managed object that this `BasicCanvasUIView` class points to using this code:

```
NSManagedObject *transform = [canvas valueForKey:@"transform"];
```

Notice the absence of any Structured Query Language (SQL) code. Notice, also, that you retrieve the value for a relationship using the same syntax you'd use to retrieve the value of an attribute. Core Data doesn't differentiate among the types of properties in an entity.

Once you have a reference to the `Transform` managed object, you can retrieve its value for `scale` using code like this:

```
[transform valueForKey:@"scale"]
```

The `valueForKey:` method returns an object, so use the `floatValue` method to get the stored value as a float type. The `scale:` method, in its entirety, looks like this:

```
- (float)scale {
    NSManagedObject *transform = [canvas valueForKey:@"transform"];
    return [[transform valueForKey:@"scale"] floatValue];
}
```

The method that does the actual drawing of shapes looks a little more complex, and drawing and graphics lie outside the scope of this book. Read through the code carefully, however, and you'll find it's not as bad as it looks. Since this book covers Core Data, though, you can just type this in without understanding it. The parts relevant to Core Data are the `Retrieve` operations: the method retrieves all the shapes to draw using the relationship between the `Canvas` entity and the `Shape` entity and then retrieves the relevant properties for the shape to draw. In the case of `Circle` instances, the code retrieves the `x`, `y`, and `radius` attributes to construct the circle on the screen. In the case of `Polygon` instances, the code retrieves all the related `Vertex` instances using `Polygon`'s `vertices` relationship, sorts them on the `index` attribute, and draws a polygon that matches the vertices. Listing 5-2 shows the entire `drawRect:` method.

**Listing 5-2. The `drawRect:` Method**

```
- (void)drawRect:(CGRect)rect {
    // Check to make sure we have data
    if (canvas == nil) {
        return;
    }

    // Get the current graphics context for drawing
    CGContextRef context = UIGraphicsGetCurrentContext();
```

```

// Store the scale in a local variable so we don't hit the data store twice
float scale = self.scale;

// Scale the context according to the stored value
CGContextScaleCTM(context, scale, scale);

// Retrieve all the shapes that relate to this canvas and iterate through them
NSSet* shapes = [canvas valueForKey:@"shapes"];
for (NSManagedObject *shape in shapes) {
    // Get the entity name to determine whether this is a Circle or a Polygon
    NSString *entityName = [[shape entity] name];

    // Get the color, stored as RGB values in a comma-separated string, and set it into
    the context
    NSString *colorCode = [shape valueForKey:@"color"];
    NSArray *colorCodes = [colorCode componentsSeparatedByString:@","];
    CGContextSetRGBFillColor(context, [[colorCodes objectAtIndex:0] floatValue] / 255,
                                [[colorCodes objectAtIndex:1] floatValue] / 255,
                                [[colorCodes objectAtIndex:2] floatValue] / 255, 1.0);

    // If this shape is a circle . . .
    if ([entityName compare:@"Circle"] == NSOrderedSame) {
        // Get the x, y, and radius from the data store and draw the circle
        float x = [[shape valueForKey:@"x"] floatValue];
        float y = [[shape valueForKey:@"y"] floatValue];
        float radius = [[shape valueForKey:@"radius"] floatValue];
        CGContextFillEllipseInRect(context, CGRectMake(x-radius, y-radius, 2*radius,
2*radius));
    } else if ([entityName compare:@"Polygon"] == NSOrderedSame) {
        // This is a polygon
        // Use a sort descriptor to order the vertices using the index value
        NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"
@index" ascending:YES];
        NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
        NSArray* vertices = [[[shape mutableSetValueForKey:@"vertices"] allObjects]
sortedArrayUsingDescriptors:sortDescriptors];

        // Begin drawing the polygon
        CGContextBeginPath(context);

        // Place the current graphic context point on the last vertex
        NSManagedObject *lastVertex = [vertices lastObject];
        CGContextMoveToPoint(context, [[lastVertex valueForKey:@"x"] floatValue],
[[lastVertex valueForKey:@"y"] floatValue]);

        // Iterate through the vertices and link them together
        for (NSManagedObject *vertex in vertices) {
            CGContextAddLineToPoint(context, [[vertex valueForKey:@"x"] floatValue],
[[vertex valueForKey:@"y"] floatValue]);
        }
        // Fill the polygon
        CGContextFillPath(context);
    }
}

```

```
}
```

That completes the BasicCanvasUIView class. You still must add two instances of this class to the main view in your application, however, which is controlled by the ShapesViewController class. Open ShapesViewController.h, add an import for BasicCanvasUIView.h, and add two BasicCanvasUIView instances to the interface (one to represent the top half of the screen and the other to represent the bottom half), like so:

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

@interface ShapesViewController : UIViewController {
    IBOutlet BasicCanvasUIView *topView;
    IBOutlet BasicCanvasUIView *bottomView;
}
@property (nonatomic, retain) BasicCanvasUIView *topView;
@property (nonatomic, retain) BasicCanvasUIView *bottomView;

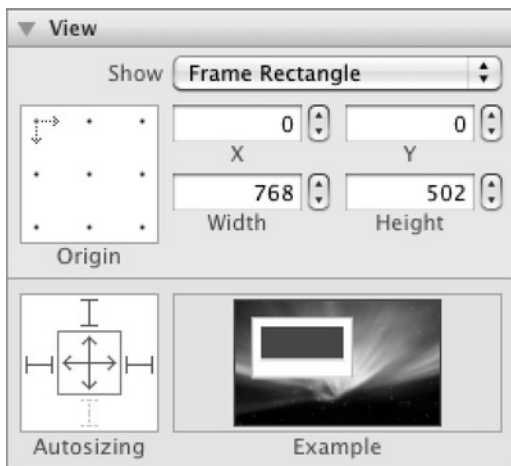
@end
```

Add two @synthesize directives to ShapesViewController.m, like this:

```
@synthesize topView;
@synthesize bottomView;
```

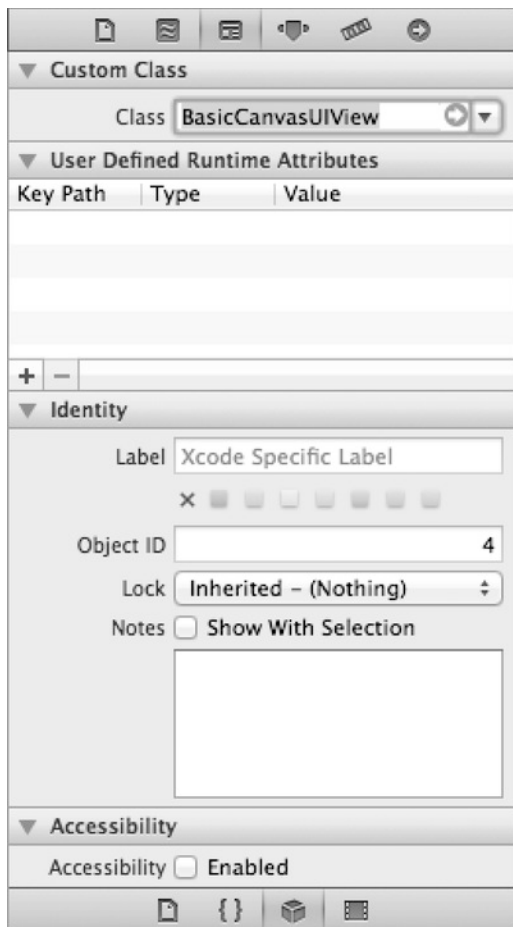
## Building the Shape Application User Interface

Now you are going to create the top and bottom views inside your shapes view. Select ShapesViewController.xib to open it in Interface Builder, which should also open the view associated with the ShapesViewController class. Drag a View object onto the view, and go to the Size tab to place and size it. Set X to 0, Y to 0, Width to 768, and Height to 502. In the Autosizing section, select all but the bottom band. See Figure 5–7 for a guide on what you’re trying to accomplish.



**Figure 5–7.** Top view sized and positioned

Select the view you just created and resized to the top half of the screen, and select the Identity Inspector tab. Here, you can change the class from `UIView` to `BasicCanvasUIView` by selecting it from the Class drop-down in the Class section, as Figure 5–8 shows.

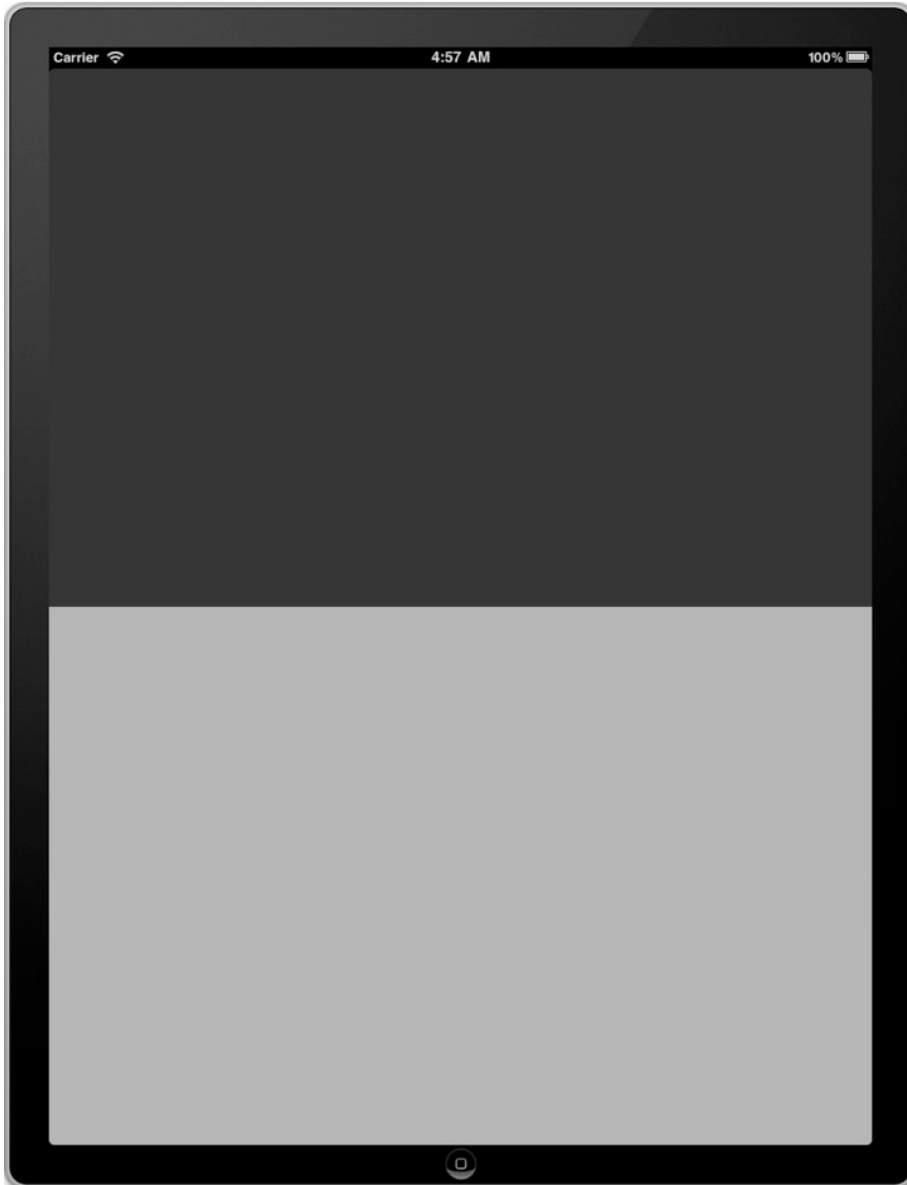


**Figure 5–8.** Top view changed to `BasicCanvasUIView`

Now you can Ctrl+click the File's Owner icon and drag your pointer into the view you created. When you release the mouse button, you should see a pop-up displaying the available outlets you can bind to. Select `topView`. Go to the Attributes inspector, and click the background to select a different color for the background for this view. Feel free to choose your own; we chose Blueberry.

Now create the bottom view by dragging another View object onto the screen and resizing it using these values: X = 0, Y = 502, Width = 768, and Height = 502. In the Autosizing section, select all but the top band. Go to the Identity inspector, and change Class to `BasicCanvasUIView`. Ctrl+click and drag from the File's Owner icon to this new view and bind it to the `bottomView` outlet. Change the color; we selected Tangerine.

Even though the application doesn't draw any shapes yet because you've as yet provided no mechanism to create them in the persistent store, now is a good time to build and run the application to verify that it compiles and runs. You should be able to run the application and see the views on the screen as in Figure 5–9. Rotating the device should cause the views to resize and remain on the top and bottom halves of the screen, as in Figure 5–10. If the application compiles and runs but the views don't properly display when the screen rotates, check the Autosizing settings for the views.



**Figure 5–9.** *Shapes without shapes in portrait mode*



**Figure 5-10.** *Shapes without shapes in landscape mode*

The next step is to make the `ShapesViewController` class Core Data–aware and then add the user interfaces to create, update, and delete shapes. Go back to the `ShapesViewController.h` file and add a property to point to the managed object context. `ShapesViewController.h` should look like the following, with the added lines in bold:

```
#import <UIKit/UIKit.h>
#import <CoreData/CoreData.h>
#import "BasicCanvasUIView.h"

@interface ShapesViewController : UIViewController {
    NSManagedObjectContext *managedObjectContext;
    IBOutlet BasicCanvasUIView *topView;
    IBOutlet BasicCanvasUIView *bottomView;
}
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain) BasicCanvasUIView *topView;
@property (nonatomic, retain) BasicCanvasUIView *bottomView;

@end
```

Now open `ShapesViewController.m`, where you have several tasks remaining to finish the Shapes application. Start with adding a synthesize line for the managed object context.

```
@synthesize managedObjectContext;
```

Next, create methods to create a shape, update all shapes, and delete all shapes. These methods are part of `ShapeViewController`'s private interface, which you declare in `ShapeViewController.m`. The other method to add to the private interface is a helper method that returns a random color. The lines to add to `ShapeViewController.m` above where the implementation begins look like this:

```
@interface ShapesViewController (private)
- (void)createShapeAt:(CGPoint)point;
- (void)updateAllShapes;
- (void)deleteAllShapes;
- (NSString *)makeRandomColor;
@end
```

```
@implementation ShapesViewController
...
```

The implementation for the method that creates a shape receives a parameter that describes where on the screen to create the shape. The method creates an `NSManagedObject` of entity type `Shape` and then randomly creates either a `Circle` type or a `Polygon` type. If a circle, it generates a random value for the radius attribute and uses the x and y values from the passed `CGPoint` instance. If a polygon, it creates a random number of vertices arranged around the passed `CGPoint` and stores them as `Vertex` types, creating relationships back to the `Polygon` instance just created. The implementation looks like that in Listing 5–3.

**Listing 5–3. Creating a Shape**

```
- (void)createShapeAt:(CGPoint)point {
    // Create a managed object to store the shape
    NSManagedObject *shape = nil;

    // Randomly choose a Circle or a Polygon
    int type = arc4random() % 2;
    if (type == 0) { // Circle
        // Create the Circle managed object
        NSEntityDescription *entity = [NSEntityDescription entityForName:@"Circle"
inManagedObjectContext:self.managedObjectContext];
        NSManagedObject *circle = [NSEntityDescription
insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];
        shape = circle;

        // Randomly create a radius and set the attributes of the circle
        float radius = 10 + (arc4random() % 90);
        [circle setValue:[NSNumber numberWithFloat:point.x] forKey:@"x"];
        [circle setValue:[NSNumber numberWithFloat:point.y] forKey:@"y"];
        [circle setValue:[NSNumber numberWithFloat:radius] forKey:@"radius"];
```

```

    NSLog(@"Made a new circle at (%f,%f) with radius %f", point.x, point.y, radius);
} else { // Polygon
    // Create the Polygon managed object
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Polygon"
inManagedObjectContext:self.managedObjectContext];
    NSManagedObject *polygon = [NSEntityDescription
insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];
    shape = polygon;

    // Get the vertices. At this point, no Vertex objects for this Shape exist.
    // Anything you add to the set, however, will be added to the Vertex entity.
    NSMutableSet *vertices = [polygon mutableSetValueForKey:@"vertices"];

    // Create a random number of vertices
    int nVertices = 3 + (arc4random() % 20);
    float angleIncrement = (2 * M_PI) / nVertices;
    int index = 0;
    for (float i = 0; i < nVertices; i++) {
        // Generate random values for each vertex
        float a = i * angleIncrement;
        float radius = 10 + (arc4random() % 90);
        float x = point.x + (radius * cos(a));
        float y = point.y + (radius * sin(a));

        // Create the Vertex managed object
        NSEntityDescription *vertexEntity = [NSEntityDescription entityForName:@"Vertex"
inManagedObjectContext:self.managedObjectContext];
        NSManagedObject *vertex = [NSEntityDescription
insertNewObjectForEntityForName:[vertexEntity name]
inManagedObjectContext:self.managedObjectContext];

        // Set the values for the vertex
        [vertex setValue:[NSNumber numberWithFloat:x] forKey:@"x"];
        [vertex setValue:[NSNumber numberWithFloat:y] forKey:@"y"];
        [vertex setValue:[NSNumber numberWithInt:index++] forKey:@"index"];

        // Add the Vertex object to the relationship
        [vertices addObject:vertex];
    }
    NSLog(@"Made a new polygon with %d vertices", nVertices);
}
// Set the shape's color
[shape setValue:[self makeRandomColor] forKey:@"color"];

// Add the same shape to both canvases
[[topView.canvas mutableSetValueForKey:@"shapes"] addObject:shape];
[[bottomView.canvas mutableSetValueForKey:@"shapes"] addObject:shape];

// Save the context
NSError *error = nil;
if (![self.managedObjectContext save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

```



```

    // Tell the views to repaint themselves
    [topView setNeedsDisplay];
    [bottomView setNeedsDisplay];
}

```

Though the method is long, read through it; between the codes and the comments, you should be able to see what it's doing. It uses some methods from the C standard library, so include the required headers, like so:

```
#include <stdlib.h>
```

It also calls the `makeRandomColor:` method to create a random color for this shape, whether circle or polygon, so implement that method. It creates three random color values, one each for red, green, and blue, and formats them as a comma-separated string, like this:

```

- (NSString *)makeRandomColor {
    // Generate three color values
    int red = arc4random() % 256;
    int green = arc4random() % 256;
    int blue = arc4random() % 256;

    // Put them in a comma-separated string
    return [NSString stringWithFormat:@"%d,%d,%d", red, green, blue];
}

```

The method that updates all the shapes to new random colors uses this method as well. It retrieves all the shapes from the persistent store and updates them with new random colors. The method looks like this:

```

- (void)updateAllShapes {
    // Retrieve all the shapes
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Shape"
inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];
    NSArray *shapes = [managedObjectContext executeFetchRequest:fetchRequest error:nil];

    // Go through all the shapes and update their colors randomly
    for (NSManagedObject *shape in shapes) {
        [shape setValue:[self makeRandomColor] forKey:@"color"];
    }

    // Save the context
    NSError *error = nil;
    if (![self.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    // Tell the views to repaint themselves
    [topView setNeedsDisplay];
    [bottomView setNeedsDisplay];
}

```

The method that deletes all the shapes looks eerily similar. In fact, this would be a good opportunity to use the new block support in Objective-C and iOS 4, but for simplicity's sake, it again retrieves all the shapes from the persistent store, but this time, instead of updating them with new colors, it deletes them. Note that the Delete Rules set in the model handles what to do with any related entities. Here is the `deleteAllShapes` method:

```
- (void)deleteAllShapes {
    // Retrieve all the shapes
    NSFetchedRequest *fetchRequest = [[NSFetchedRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Shape"
inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];
    NSArray *shapes = [managedObjectContext executeFetchRequest:fetchRequest error:nil];

    // Delete each shape.
    for (NSManagedObject *shape in shapes) {
        [managedObjectContext deleteObject:shape];
    }

    // Save the context
    NSError *error = nil;
    if (![self.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    // Tell the views to repaint themselves
    [topView setNeedsDisplay];
    [bottomView setNeedsDisplay];
}
```

You might notice that the `ShapesViewController` class has a member for the managed object context, but you haven't yet set any value into that member. Open `ShapesAppDelegate.m` and add code to set that value. The method definition, with the added line in bold, looks like this:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[ShapesViewController alloc]
initWithNibName:@"ShapesViewController" bundle:nil];
    self.viewController.managedObjectContext = self.managedObjectContext;
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

One final Core Data–related piece remains to build into the Shapes application: the code to create the Canvas instances and tie them to the views and to create the Transform instances, one for each Canvas. This code should go into the `viewDidLoad` method of `ShapesViewController.m`.

```

- (void)viewDidLoad {
    // Create the Canvas entities
    NSManagedObject *canvas1 = nil;
    NSManagedObject *canvas2 = nil;

    // Load the canvases
    NSFetchedRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Canvas"
inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];
    NSArray *canvases = [managedObjectContext executeFetchRequest:fetchRequest error:nil];

    // If the canvases already exist in the persistent store, load them
    if([canvases count] >= 2) {
        NSLog(@"Loading existing canvases");
        canvas1 = [canvases objectAtIndex:0];
        canvas2 = [canvases objectAtIndex:1];
    } else { // No canvases exist in the persistent store, so create them
        NSLog(@"Making new canvases");
        canvas1 = [NSEntityDescription insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];
        canvas2 = [NSEntityDescription insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];

        // Create the Transform instance for each canvas. The first has a scale of 1
        NSManagedObject *transform1 = [NSEntityDescription
insertNewObjectForEntityForName:@"Transform"
inManagedObjectContext:self.managedObjectContext];
        [transform1 setValue:[NSNumber numberWithFloat:1] forKey:@"scale"];
        [canvas1 setValue:transform1 forKey:@"transform"];

        // The second Transform for the second Canvas has a scale of 0.5
        NSManagedObject *transform2 = [NSEntityDescription
insertNewObjectForEntityForName:@"Transform"
inManagedObjectContext:self.managedObjectContext];
        [transform2 setValue:[NSNumber numberWithFloat:0.5] forKey:@"scale"];
        [canvas2 setValue:transform2 forKey:@"transform"];

        // Save the context
        NSError *error = nil;
        if (![self.managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
    // Set the Canvas instances into the views
    topView.canvas = canvas1;
    bottomView.canvas = canvas2;
}

```

## Enabling User Interactions with the Shapes Application

That completes the Core Data parts of the application. You could build and run it, but nothing visible has changed. You need to add user interface elements to create, update, and delete shapes. Start with creation. The user interface for creating a shape is to tap the screen, so add a method to `ShapesViewController.m` to capture any screen taps, determine the location of the tap, scale the location appropriately according to the canvas, and call the `createShapeAt:` method, passing in the tapped point.

**#pragma mark - Touch events handling**

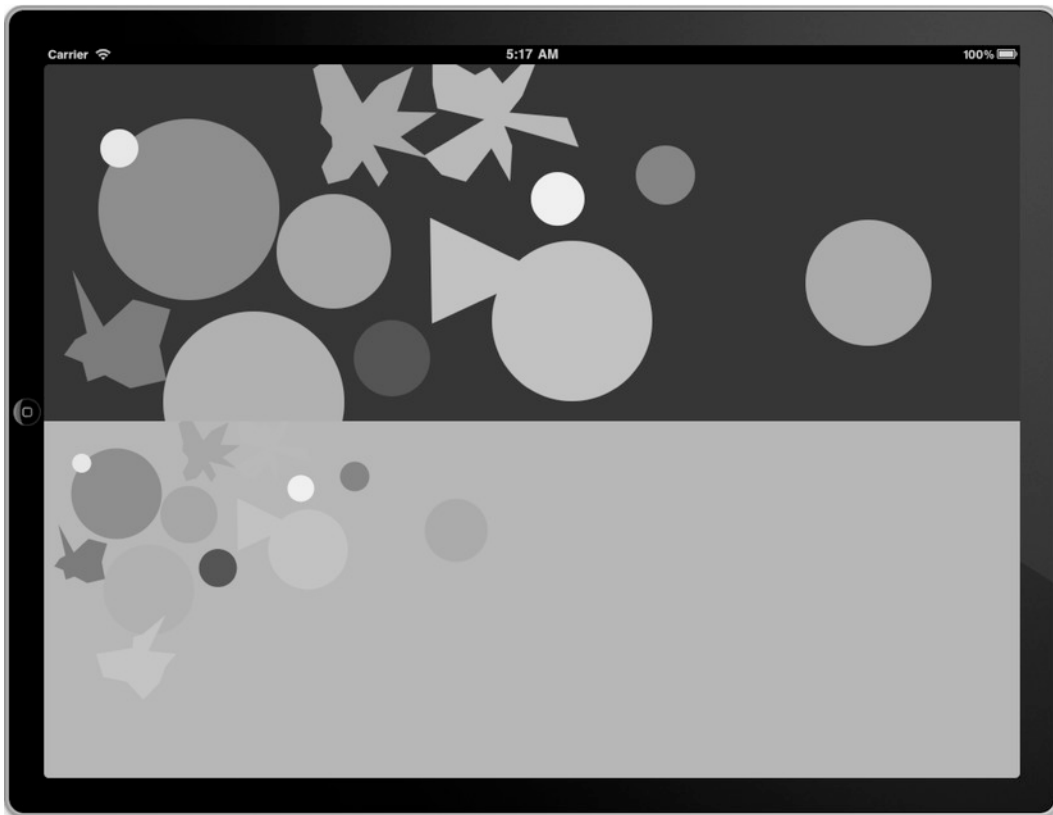
```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];

    // Where did the view get touched?
    CGPoint location = [touch locationInView: touch.view];

    // Scale according to the canvas' transform
    float scale = [(BasicCanvasUIView *)touch.view scale];
    location = CGPointMake(location.x / scale, location.y / scale);

    // Create the shape
    [self createShapeAt:location];
}
```

Now the application merits building and running. You haven't finished yet—you can't update or delete shapes—but you deserve some gratification for having worked this long. Build and run the application, and tap the screen a few times. Notice that you can tap either screen half, and the shapes appear on both, with the shapes twice as big on the top half. Rotate the screen, and see that the shapes still display. See Figure 5-11 for an example of what your screen should look like.



**Figure 5–11.** *Some shapes on your screen*

Stop the running application, and add the user interface elements to update and delete the shapes. The user interface for updating the shapes is rotating the device, so add a method to detect any device rotation, and call the `updateAllShapes` method from within it.

```
- (void) willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
duration:(NSTimeInterval)duration {
    [self updateAllShapes];
}
```

Adding support for detecting whether the device is shaken, so the application can delete all the shapes, is slightly more complicated. `ShapeViewController` must become the first responder when its view displays, so add the method that makes it eligible to be a first responder, and then make it become the first responder in the `viewDidAppear` method.

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}
```

Now, add the method to detect any shakes, and call the `deleteAllObjects` method from within it.

**#pragma mark - Shake events handling**

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    if (event.subtype == UIEventSubtypeMotionShake) {
        [self deleteAllShapes];
    }
}
```

This completes the Shapes application. Build it, run it, tap it, rotate it, shake it. You should see shapes appear, change colors, and disappear. Through the easy Core Data interface, you can create, retrieve, update, and delete data from your persistent store.

If you have any problems building or running the Shapes application, check the accuracy of your data model and also of your code. The complete source code of the Shapes application is available online on this book's page at <http://apress.com/>.

The next section enhances the Shapes application by adding custom data objects instead of using `NSManagedObject` instances. In the typical life cycle of a Core Data-enabled application, you will likely start by using `NSManagedObject`. As your application evolves, you will eventually feel the need to introduce custom data objects for code clarity and maintainability.

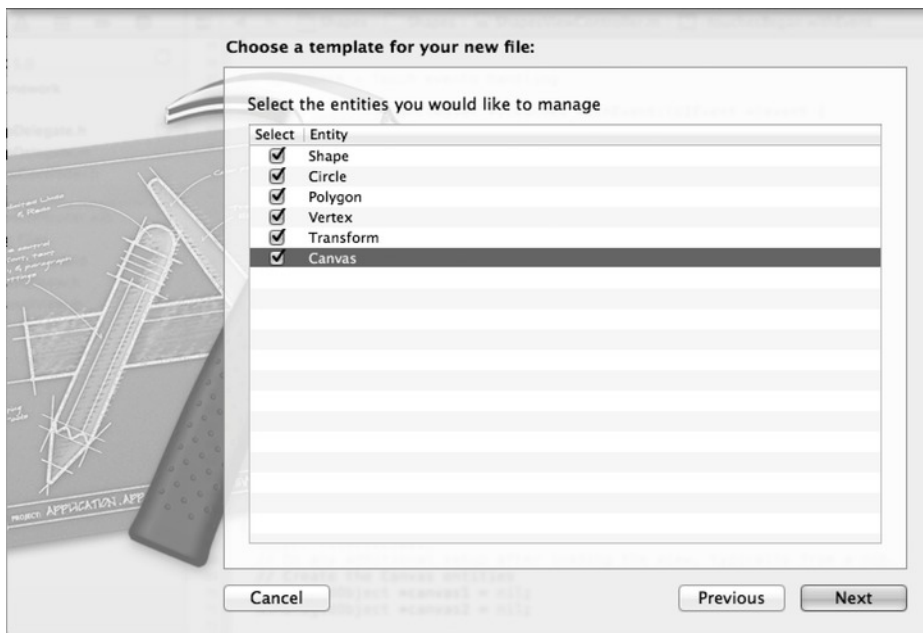
## Generating Classes

This is the first time in this book that you are on the verge of enhancing your managed objects to a level that makes them truly blend with the rest of your code. Core Data gives you so much flexibility to implement your own managed objects that you can almost completely hide the fact that they are backed by the framework. In this section, you rework the Shapes application in order to use custom managed objects. You can make the changes either to your existing Shapes project, or make a copy of the project and make changes to that. In the downloadable source code for this book, we made a copy of the Shapes project in a directory called Shapes2 and made the changes there.

The first step in enhancing the Shapes application is to create the data object classes themselves. You can create them manually or let Xcode generate them for you. Generating classes for your model's entities in Xcode 4 dispenses with the trickery that Xcode 3 required (open your model file, click somewhere in the background, select **File** ► **New File**, and only then do you have the option to create classes to represent your entities).

To generate classes for your model's entities, select **File** ► **New** ► **New File**, and select Core Data under iOS on the left and `NSManagedObject` subclass on the right, as shown in Figure 5-12. Click Next, and in the ensuing dialog, select the Shapes data model, as shown in Figure 5-13, and click Next. The next dialog lets you select the entities in your data model for which you'd like to generate classes. Select all of them, as shown in Figure 5-14, and click Next. Click the Create button in the next dialog to save your generated classes.

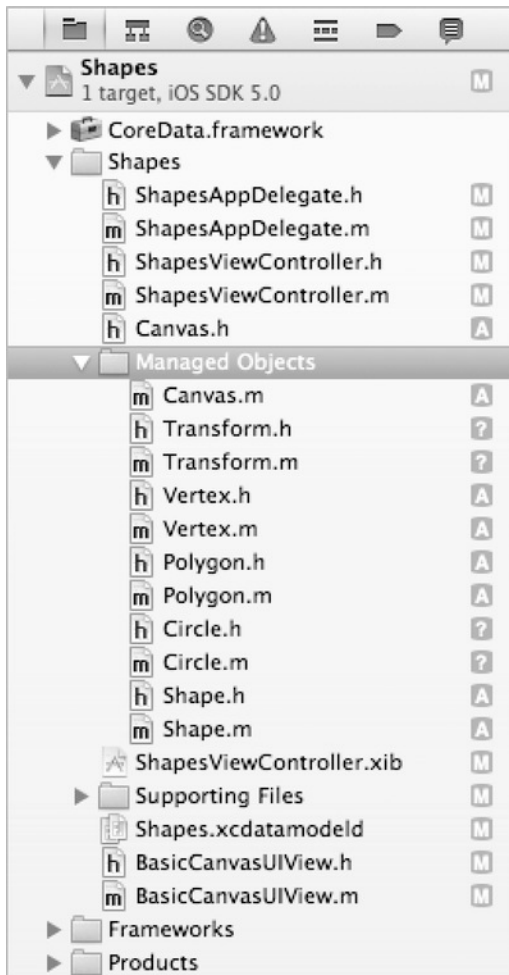




**Figure 5–14.** *Selecting which entities need classes generated*

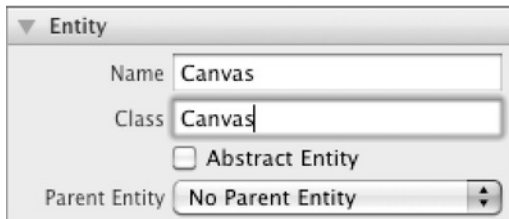
To keep the code organized, a good practice is to create a group of classes where you will store your data objects. In Xcode, select all the generated classes, Ctrl+click, and select New Group from Selection. Rename the new group to Managed Objects. Figure 5–15 shows what your source tree should look like.





**Figure 5–15.** Source tree with the Managed Object classes in a group

Xcode updates your model to use the generated classes for your entities. To verify, display the attributes of any of your entities in the Data Model inspector. For the Class field, the name of your generated class should display instead of `NSManagedObject`. Figure 5–16 shows an example using the Canvas entity; note that Xcode has changed the entry for Class from `NSManagedObject` to `Canvas`, the new class you generated.



**Figure 5–16.** The Canvas entity updated to use the Canvas class

You don't have to use Xcode's generator to create model classes, however. To perform the same task manually, without having Xcode generate your classes, you would do two things:

1. Create the code for the class (open any of the new files to see examples).
2. Update the Class field for the entity in your data model to point to the new class.

That's good information to know, in case you ever have to create those classes manually, but it's nice to have Xcode do the work for you.

Now, let's look at some of these classes. Open `Vertex.h`. You can see that it imports the Foundation and Core Data framework header files, that it derives from `NSManagedObject`, and that it exposes the properties (both attributes and relationships) of the Vertex entity as Objective-C properties, like so:

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

@class Polygon;

@interface Vertex : NSManagedObject

@property (nonatomic, retain) NSNumber * y;
@property (nonatomic, retain) NSNumber * x;
@property (nonatomic, retain) NSNumber * index;
@property (nonatomic, retain) Polygon * polygon;

@end
```

Check your classes, though; you may notice that some generated classes use `NSManagedObject` instances, instead of your generated classes, for relationships. Your generated `Transform` class, for example, uses an `NSManagedObject` instance instead of a Canvas image to represent its canvas relationship, as seen here:

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

@interface Transform : NSManagedObject

@property (nonatomic, retain) NSNumber * scale;
@property (nonatomic, retain) NSManagedObject * canvas;

@end
```

If you find anomalies like this in your generated classes, update the files to use the generated classes. The following code shows `Transform.h` updated to use a Canvas instance instead of an `NSManagedObject` instance:

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

@class Canvas;
```

```
@interface Transform : NSObject
@property (nonatomic, retain) NSNumber * scale;
@property (nonatomic, retain) Canvas * canvas;

@end
```

For any header files that you update, be sure to add the appropriate header to the corresponding implementation file. For the previous example, you add this line to `Transform.m`:

```
#import "Canvas.h"
```

Next, let's look at how these generated classes are implemented. The following code shows the implementation file for the `Vertex` class, `Vertex.m`. You probably can't help but notice how light this implementation is. The magic happens when using the `@dynamic` directive in the code. You already know the more common `@synthesize` directive, which tells the compiler to generate basic stubs for property accessors in order to fulfill the API contract with `@property` directives in the header file. The `@dynamic` directive is used to tell the compiler that even though it can't find the methods that fulfill the contract, they will be there by the time the runtime needs to invoke them. In essence, you are telling the compiler to trust you. Of course, if you break your promise and do not provide the accessors at runtime, you will be punished with an application crash. Because your class extends `NSObject`, the accessors are generated for you so you don't have to worry about any punishment. It is safe to use the `@dynamic` directive for entity properties.

```
#import "Vertex.h"
#import "Polygon.h"

@implementation Vertex
@dynamic y;
@dynamic x;
@dynamic index;
@dynamic polygon;

@end
```

Again, you could have written this code yourself, but it's nice to have Xcode do the work for you.

You can start the application, and it still runs even though the rest of the code doesn't use the newly generated classes. It still, for example, accesses the properties of any vertices using the `valueForKey:` method. Since `Vertex` is an instance of `NSObject`, accessing its properties through `valueForKey:` is still valid.

Next, open the header file for `Circle`. Because you made the `Circle` entity a subentity of `Shape` in the data model, the `Circle` class extends `Shape`. `Circle` is still an `NSObject` by inheritance since `Shape` extends `NSObject`.

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
#import "Shape.h"
```

```
@interface Circle : Shape

@property (nonatomic, retain) NSNumber * x;
@property (nonatomic, retain) NSNumber * y;
@property (nonatomic, retain) NSNumber * radius;

@end
```

Next in line to review is Polygon. Polygons are slightly different from the other objects created so far because they have a to-many relationship to another class.

Because a Polygon is a subclass of `NSManagedObject`, it benefits from its Key-Value Coding (KVC) compliance. Core Data provides accessor methods for attributes and relationships you modeled in your data model. For attributes, you have seen that it provides simple get/set methods. For to-many relationships, it automatically creates KVC mutable proxy methods for the `NSSet`. The methods are named after the key:

- (void)add<Key>Object:(id)value;
- (void)remove<Key>Object:(id)value;
- (void)add<Key>:(NSSet \*)value;
- (void)remove<Key>:(NSSet \*)value;

This feature means that it is possible to call `addVerticesObject:(Vertex*)` where before you called `[[managedObject mutableSetValueForKey:key] addObject:value]`.

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
#import "Shape.h"

@interface Polygon : Shape

@property (nonatomic, retain) NSSet *vertices;
@end

@interface Polygon (CoreDataGeneratedAccessors)

- (void)addVerticesObject:(NSManagedObject *)value;
- (void)removeVerticesObject:(NSManagedObject *)value;
- (void)addVertices:(NSSet *)values;
- (void)removeVertices:(NSSet *)values;

@end
```

You don't need implementations for these methods in `Polygon.m`, however, because they're automatically provided by the KVC mechanism. If Xcode generated these methods, you can leave them or delete them.

Feel free to peruse the rest of the code for the generated classes. Armed with your knowledge of Core Data and the previous information, you should understand what's happening with this code.

Now that you have your custom objects, you can go back to the application code to use their properties directly, rather than going through `valueForKey:`. Open `ShapesViewController.m`, and add import statements for all the new objects you've created, as shown here:

```
#import "Polygon.h"
#import "Circle.h"
#import "Shape.h"
#import "Canvas.h"
#import "Vertex.h"
#import "Transform.h"
```

Next, update the `createShapeAt:` method to use your new classes. The revised implementation of `createShapeAt:` is slightly cleaner; see Listing 5–4. In this first wave of cleanup, you’ve removed all references to `NSManagedObject` and put the actual type instead. This allows you to then set the properties directly instead of calling `setValue:forKey:`. Later in this chapter, you’ll run a second wave of cleanup that illustrates how to get rid of all references to Core Data in the code to make the framework even more seamless.

**Listing 5–4. Revised Implementation of `createShapeAt:`**

```
- (void)createShapeAt:(CGPoint)point {
    // Create a managed object to store the shape
    Shape *shape = nil;

    // Randomly choose a Circle or a Polygon
    int type = arc4random() % 2;
    if (type == 0) { // Circle
        // Create the Circle managed object
        NSEntityDescription *entity = [NSEntityDescription entityForName:@"Circle"
inManagedObjectContext:self.managedObjectContext];
        Circle *circle = [NSEntityDescription insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];
        shape = circle;

        // Randomly create a radius and set the attributes of the circle
        float radius = 10 + (arc4random() % 90);
        circle.x = [NSNumber numberWithFloat:point.x];
        circle.y = [NSNumber numberWithFloat:point.y];
        circle.radius = [NSNumber numberWithFloat:radius];

        NSLog(@"Made a new circle at (%f,%f) with radius %f", point.x, point.y, radius);
    } else { // Polygon
        // Create the Polygon managed object
        NSEntityDescription *entity = [NSEntityDescription entityForName:@"Polygon"
inManagedObjectContext:self.managedObjectContext];
        Polygon *polygon = [NSEntityDescription insertNewObjectForEntityForName:[entity
name] inManagedObjectContext:self.managedObjectContext];
        shape = polygon;

        // Create a random number of vertices
        int nVertices = 3 + (arc4random() % 20);
        float angleIncrement = (2 * M_PI) / nVertices;
        int index = 0;
        for (float i = 0; i < nVertices; i++) {
            // Generate random values for each vertex
            float a = i * angleIncrement;
            float radius = 10 + (arc4random() % 90);
            float x = point.x + (radius * cos(a));
            float y = point.y + (radius * sin(a));
```

```

        // Create the Vertex managed object
        NSEntityDescription *vertexEntity = [NSEntityDescription entityForName:@"Vertex"
inManagedObjectContext:self.managedObjectContext];
        Vertex *vertex = [NSEntityDescription
insertNewObjectForEntityForName:[vertexEntity name]
inManagedObjectContext:self.managedObjectContext];

        // Set the values for the vertex
        vertex.x = [NSNumber numberWithFloat:x];
        vertex.y = [NSNumber numberWithFloat:y];
        vertex.index = [NSNumber numberWithInt:index++];

        // Add the Vertex object to the relationship
        [polygon addVerticesObject:vertex];
    }
    NSLog(@"Made a new polygon with %d vertices", nVertices);
}
// Set the shape's color
shape.color = [self makeRandomColor];

// Add the same shape to both canvases
[[topView.canvas mutableSetValueForKey:@"shapes"] addObject:shape];
[[bottomView.canvas mutableSetValueForKey:@"shapes"] addObject:shape];

// Save the context
NSError *error = nil;
if (![self.managedObjectContext save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

// Tell the views to repaint themselves
[topView setNeedsDisplay];
[bottomView setNeedsDisplay];
}

```

Note how each vertex is added to the polygon by calling `addVerticesObject:`.

There still isn't anything groundbreaking at this point because you have simply removed references to `NSManagedObject` and calls to the key/value store. The resulting code is still slightly better because it enforces stronger typing with the managed objects and their properties. The real value of custom managed objects comes from enhancing them with more specialized methods. The next section explains how to do that.

## Modifying Generated Classes

Now that the application is outfitted with custom objects, it's time to exhibit the real power of custom managed objects. Having to constantly refer to the Core Data framework or key/value store for everything that relates to managed objects gets distracting very quickly when building an application. The framework should assist, not hinder, the developers in building data store-driven applications. With custom objects, you've gone halfway to a better place and have created an anemic object model and a

set of objects that have a state but no behavior. Martin Fowler, chief scientist at ThoughtWorks and self-labeled general loudmouth on software development, first identified the anemic domain model antipattern as well as the antidote. The injection of business logic into objects has been proven to rapidly cure anemia. The next step in refactoring the Core Data managed objects is to add the business logic that will make your objects useful and easier to use.

Adding a static managed object initializer method is a usual way to further detach Core Data from the rest of the code. In `ShapesViewController`, you create a canvas and its transform by doing the following:

```
canvas1 = [NSEntityDescription insertNewObjectForEntityForName:[entity name]
inManagedObjectContext:self.managedObjectContext];
...
NSManagedObject *transform1 = [NSEntityDescription
insertNewObjectForEntityForName:@"Transform"
inManagedObjectContext:self.managedObjectContext];
[transform1 setValue:[NSNumber numberWithInt:1] forKey:@"scale"];
[canvas1 setValue:transform1 forKey:@"transform"];
```

The application code still reeks of Core Data. A better way to create a Canvas and its Transform is to define methods for creating transforms and canvases inside the appropriate objects: Transform and Canvas. You first declare an initializer in `Transform.h`, shown in bold here:

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

@class Canvas;

@interface Transform : NSManagedObject

@property (nonatomic, retain) NSNumber * scale;
@property (nonatomic, retain) Canvas * canvas;

+ (Transform *)initWithScale:(float)scale inContext:(NSManagedObjectContext *)context;

@end
```

The implementation for the new initializer goes in `Transform.m`, shown in bold here:

```
#import "Transform.h"
#import "Canvas.h"

@implementation Transform
@dynamic scale;
@dynamic canvas;

+ (Transform *)initWithScale:(float)scale inContext:(NSManagedObjectContext *)context {
    Transform *transform = [NSEntityDescription
insertNewObjectForEntityForName:@"Transform" inManagedObjectContext:context];
    transform.scale = [NSNumber numberWithInt:scale];
    return transform;
}
```

```
}
```

```
@end
```

You do the same for Canvas, which can be initialized with its Transform instead of setting it later. This is a better design since the transform attribute of the Canvas entity is required. Canvas.h can be modified to look this:

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

@class Transform;

@interface Canvas : NSManagedObject

@property (nonatomic, retain) NSSet* shapes;
@property (nonatomic, retain) Transform * transform;

+ (Canvas *)initWithTransform:(Transform *)transform inContext:(NSManagedObjectContext*)context;

@end

@interface Canvas (CoreDataGeneratedAccessors)
- (void)addShapesObject:(NSManagedObject *)value;
- (void)removeShapesObject:(NSManagedObject *)value;
- (void)addShapes:(NSSet *)value;
- (void)removeShapes:(NSSet *)value;

@end
```

And you can then add the code in the implementation file, Canvas.m, shown here:

```
#import "Canvas.h"
#import "Transform.h"

@implementation Canvas
@dynamic shapes;
@dynamic transform;

+ (Canvas *)initWithTransform:(Transform *)transform inContext:(NSManagedObjectContext*)context {
    Canvas *canvas = [NSEntityDescription insertNewObjectForEntityForName:@"Canvas"
inManagedObjectContext:context];
    canvas.transform = transform;
    return canvas;
}

@end
```



Finally, you are ready to alter the `viewDidLoad:` method in the `ShapesViewController` implementation, like so:

```
- (void)viewDidLoad {
    // Create the Canvas entities
    Canvas *canvas1 = nil;
    Canvas *canvas2 = nil;

    // Load the canvases
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Canvas"
inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];
    NSArray *canvases = [managedObjectContext executeFetchRequest:fetchRequest error:nil];
    [fetchRequest release];

    // If the canvases already exist in the persistent store, load them
    if([canvases count] >= 2) {
        NSLog(@"Loading existing canvases");
        canvas1 = [canvases objectAtIndex:0];
        canvas2 = [canvases objectAtIndex:1];
    } else { // No canvases exist in the persistent store, so create them
        NSLog(@"Making new canvases");
        Transform *transform1 = [Transform initWithScale:1
inContext:self.managedObjectContext];
        canvas1 = [Canvas initWithTransform:transform1 inContext:self.managedObjectContext];

        Transform *transform2 = [Transform initWithScale:0.5
inContext:self.managedObjectContext];
        canvas2 = [Canvas initWithTransform:transform2 inContext:self.managedObjectContext];

        // Save the context
        NSError *error = nil;
        if (![self.managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
    // Set the Canvas instances into the views
    topView.canvas = canvas1;
    bottomView.canvas = canvas2;
}
```

The code is getting a lot cleaner, and using the managed objects is becoming more and more like using regular objects because the ties to Core Data get encapsulated in the classes themselves.

Turn your attention now to the creation of the shapes. The shapes view controller has code that will create a random circle. In the context of your application, this is business logic that could be encapsulated in the `Circle` class. Instead of building a circle yourself, you could ask the `Circle` class to give you a random instance of itself based on a given origin. You could then create a circle by calling `[Circle randomInstance:origin inContext:managedObjectContext]`.

To do this, you add a static initializer method in the `Circle` class

```
+ (Circle *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context;
and provide an implementation in Circle.m
```

```
+ (Circle *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context {
    Circle *circle = [NSEntityDescription insertNewObjectForEntityForName:@"Circle"↵
inManagedObjectContext:context];

    float radius = 10 + (arc4random() % 90);
    circle.x = [NSNumber numberWithFloat:origin.x];
    circle.y = [NSNumber numberWithFloat:origin.y];
    circle.radius = [NSNumber numberWithFloat:radius];

    return circle;
}
```

Do the same for Polygon by adding a similar method to Polygon.h

```
+ (Polygon *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context;
and provide an implementation in Polygon.m:
```

```
+ (Polygon *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context
{
    Polygon *polygon = [NSEntityDescription insertNewObjectForEntityForName:@"Polygon"↵
inManagedObjectContext:context];

    // Set the vertices
    int nVertices = 3 + (arc4random() % 20);
    float angleIncrement = (2 * M_PI) / nVertices;
    int index = 0;
    for (float i = 0; i < nVertices; i++) {
        float a = i * angleIncrement;
        float radius = 10 + (arc4random() % 90);
        float x = origin.x + (radius * cos(a));
        float y = origin.y + (radius * sin(a));

        Vertex *vertex = [NSEntityDescription insertNewObjectForEntityForName:@"Vertex"↵
inManagedObjectContext:context];
        vertex.x = [NSNumber numberWithFloat:x];
        vertex.y = [NSNumber numberWithFloat:y];
        vertex.index = [NSNumber numberWithFloat:index++];

        [polygon addVerticesObject:vertex];
    }
    return polygon;
}
```

You can now modify the implementation of createShapeAt: in ShapesViewController.m as follows:

```
- (void)createShapeAt:(CGPoint)point {
    // Create a managed object to store the shape
    Shape *shape = nil;

    // Randomly choose a Circle or a Polygon
    int type = arc4random() % 2;
```

```

if (type == 0) { // Circle
    shape = [Circle sharedInstance:point inContext:self.managedObjectContext];
} else { // Polygon
    shape = [Polygon sharedInstance:point inContext:self.managedObjectContext];
}
// Set the shape's color
shape.color = [self makeRandomColor];

// Add the same shape to both canvases
[(Canvas *)topView.canvas addShapesObject:shape];
[(Canvas *)bottomView.canvas addShapesObject:shape];

// Save the context
NSError *error = nil;
if (![self.managedObjectContext save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

// Tell the views to repaint themselves
[topView setNeedsDisplay];
[bottomView setNeedsDisplay];
}

```

The tidiness of the application code speaks for itself at this point. Most of the logic has been encapsulated into the custom managed objects.

## Using the Transformable Type

Despite all the efforts to hide Core Data, it is likely that some properties of the custom objects just feel like they are using the “wrong” type simply to satisfy Core Data. For this type of situation, the framework allows you to use the Transformable type (`NSTransformableAttributeType`). In your Shapes application, the `color` property of the Shape object gives this sensation of type misuse. In the current implementation, the `color` is represented by a comma-delimited string so that you could use the `NSStringAttributeType` in the data model. It works well, but it makes using the object more complicated because you find yourself having to decode the value back and forth between a string and a color. The first step is to change the data model. Select the Shape entity, pick the `color` attribute, and change its type to Transformable. In the Transformer field, put the name of the custom transformer you’ll be writing: `UIColorTransformer`.

**NOTE:** Value transformers are part of the Foundation framework in Cocoa. As the name indicates, they facilitate the transformation of data from one format to another. In Core Data, it is typical to use `NSKeyedUnarchiveFromDataTransformerName`, which converts objects into `NSData`. Alternatively, you may extend `NSValueTransformer` in order to provide a custom transformer, as you do in this chapter.

In the code, you now edit `Shape.h` to change the property type from `NSString` to `UIColor`.

```
@property (nonatomic, retain) UIColor *color;
```

Naturally, using the `Transformable` type isn't all good news. Because the transformer creates an `NSData` representation of the color property, the data store will use a binary field. In the case of `SQLite`, for example, a `BLOB` column will be created to store your data. This means that you can no longer use the data in the query. For instance, you would no longer be able to query for all red shapes. Note that with the string encoding you used before, it wasn't trivial either, but it was at least possible.

Since you changed the attribute type, you need to go back to the application code to make it use the new type. First, alter `ShapesViewController.m` to change the implementation of the `makeRandomColor:` method, like so:

```
- (UIColor *)makeRandomColor {
    // Set the shape's color
    float red = (arc4random() % 256) / 255.0;
    float green = (arc4random() % 256) / 255.0;
    float blue = (arc4random() % 256) / 255.0;

    return [UIColor colorWithRed:red green:green blue:blue alpha:1.0];
}
```

Don't forget to change the return type in the method definition in the category at the top of the file, like so:

```
@interface ShapesViewController (private)
- (void)createShapeAt:(CGPoint)point;
- (void)updateAllShapes;
- (void)deleteAllShapes;
- (UIColor *)makeRandomColor;
@end
```

The next place to modify is the implementation of the `drawRect:` method in `BasicCanvasUIView.m`. Be sure to import `Shape.h`, `Circle.h`, `Polygon.h`, and `Vertex.h` in `BasicCanvasUIView.m` and then update the `drawRect:` method as shown in Listing 5-5.

**Listing 5-5.** *Modifying the `drawRect:` Method in `BasicCanvasUIView.m`.*

```
- (void)drawRect:(CGRect)rect {
    // Check to make sure we have data
    if (canvas == nil) {
        return;
    }

    // Get the current graphics context for drawing
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Store the scale in a local variable so we don't hit the data store twice
    float scale = self.scale;

    // Scale the context according to the stored value
    CGContextScaleCTM(context, scale, scale);
```

```

// Retrieve all the shapes that relate to this canvas and iterate through them
NSSet* shapes = [canvas valueForKey:@"shapes"];
for (Shape *shape in shapes) {
    // Get the entity name to determine whether this is a Circle or a Polygon
    NSString *entityName = [[shape entity] name];

    // Get the color
    const CGFloat *rgb = CGColorGetComponents(shape.color.CGColor);
    CGContextSetRGBFillColor(context, rgb[0], rgb[1], rgb[2], 1.0);

    // If this shape is a circle . . .
    if ([entityName compare:@"Circle"] == NSOrderedSame) {
        // Get the x, y, and radius from the data store and draw the circle
        Circle *circle = (Circle *)shape;
        float x = [circle.x floatValue];
        float y = [circle.y floatValue];
        float radius = [circle.radius floatValue];
        CGContextFillEllipseInRect(context, CGRectMake(x-radius, y-radius, 2*radius,
2*radius));
    } else if ([entityName compare:@"Polygon"] == NSOrderedSame) {
        // This is a polygon
        Polygon *polygon = (Polygon *)shape;

        // Use a sort descriptor to order the vertices using the index value
        NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:
@"index" ascending:YES];
        NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
        NSArray* vertices = [polygon.vertices
sortedArrayUsingDescriptors:sortDescriptors];

        // Begin drawing the polygon
        CGContextBeginPath(context);

        // Place the current graphic context point on the last vertex
        Vertex *lastVertex = [vertices lastObject];
        CGContextMoveToPoint(context, [lastVertex.x floatValue], [lastVertex.y
floatValue]);

        // Iterate through the vertices and link them together
        for (Vertex *vertex in vertices) {
            CGContextAddLineToPoint(context, [vertex.x floatValue], [vertex.y floatValue]);
        }
        // Fill the polygon
        CGContextFillPath(context);

        // Clean up
        [sortDescriptors release];
        [sortDescriptor release];
    }
}
}

```

We've noted that it is possible to use an alternate value transformer. You must use an alternate value transformer for objects that can't be naturally serialized to NSData, and you can use one for any of your attributes.

**TIP:** If a class conforms to the `NSCoding` protocol, then `NSKeyedUnarchiveFromDataTransformerName` can be used.

You are now implementing a custom value transformer for your `UIColor` object. In Xcode, add a new class called `UIColorTransformer` that extends `NSValueTransformer`, as shown:

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

@interface UIColorTransformer : NSValueTransformer

@end
```

`NSValueTransformer` has several methods that must be implemented in `UIColorTransformer.m` in order to provide the functionality that Core Data requires. You first must make sure the transformer is reversible, which means it can transform a `UIColor` object into an `NSData` as well as transform the `NSData` back into a `UIColor`. This is critical in order to be able to store and retrieve the color attribute. For this, open the `UIColorTransformer.m` file, and override the `allowsReverseTransformation:` method, like so:

```
+ (BOOL)allowsReverseTransformation {
    return YES;
}
```

You also indicate that the transformed object will be represented as an `NSData`.

```
+ (Class)transformedValueClass {
    return [NSData class];
}
```

Finally, you provide the two transformation methods. One will transform from `UIColor` to `NSData`, and the reverse method does the opposite. Note that this is a simplified implementation for color transformation that assumes that the `UIColor` object was created in the RGB color space, which is the case in the Shapes application. Color spaces fall outside the scope of this book, but we encourage you to read further about colors and transformers if you want to implement a more complete color value transformer.

```
- (id)transformedValue:(id)value {
    UIColor* color = (UIColor *)value;
    const CGFloat *components = CGColorGetComponents(color.CGColor);

    NSString* result = [NSString stringWithFormat:@"%f,%f,%f", components[0],
components[1], components[2]];
    return [result dataUsingEncoding:[NSString defaultCStringEncoding]];
}

- (id)reverseTransformedValue:(id)value {
    NSString *string = [[NSString alloc] initWithData:value
encoding:[NSString defaultCStringEncoding]];
}
```

```

NSArray *components = [string componentsSeparatedByString:@","];
CGFloat red = [[components objectAtIndex:0] floatValue];
CGFloat green = [[components objectAtIndex:1] floatValue];
CGFloat blue = [[components objectAtIndex:2] floatValue];

return [UIColor colorWithRed:red green:green blue:blue alpha:1.0];
}

```

The last step for using your custom attribute transformer is to register the transformer in `ShapesAppDelegate.m`. This can be done simply by importing `UIColorTransformer.h` and adding the following two lines at the beginning of the `application:didFinishLaunchingWithOptions:` method so that they are called before the Core Data stack is initialized:

```

UIColorTransformer* transformer = [[UIColorTransformer alloc] init];
[UIColorTransformer setValueTransformer:transformer forName:(NSString
*)@"UIColorTransformerName"];

```

You'll have to delete your existing Shapes database, but then you can build and run the Shapes application to see your color transformer being used. You can also open the shapes SQLite database and look at the ZCOLOR column in the ZSHAPE table, which is now of type BLOB. If you look at any of the values there, you see that they are in the format created by the `transformedValue:` method of the `UIColorTransformer` class you created: three comma-separated float values that represent the components of the `CGColor`.

In this section, you have seen how to take advantage of your custom objects to hide Core Data from your application code as much as possible in order to make the persistence layer as transparent as possible.

## Validating Data

Core Data won't allow you to stuff data that doesn't fit into any of the attributes in your model. Try, for example, to put a string like "Books for Professionals by Professionals," the Apress tag line, into an attribute of type `Integer 16`, and you'll raise an `NSInvalidArgumentException` that looks something like this:

```

'NSInvalidArgumentException', reason: 'Unacceptable type of value for attribute:
property = "x"; desired type = NSNumber; given type = NSString; value = Books for
Professionals by Professionals.'

```

Core Data enforces data integrity, but sometimes you want more than that. Sometimes you want to enforce what corporate-speak terms *business rules*, though they might or might not have anything to do with business. Take, for example, the `Polygon` instances in the Shapes application and their relationships with `Vertex` instances. A real-world polygon with no vertices doesn't exist, so you set the "vertices" relationship to nonoptional. A polygon with one vertex isn't a polygon, either. It's a point. And a two-vertex polygon is a line. A polygon must have at least three vertices to make the club. For that reason, you set the value for Min Count in the "vertices" relationship to three. Relating a `Polygon` instance to only one or two `Vertex` instances prevents the managed

object context from saving successfully. Instead, you get an error that looks like this: “Operation could not be completed. (Cocoa error 1580.)” The error codes come from the header file `CoreDataErrors.h` and are listed, along with descriptions, in Table 5–1. Note that the descriptions are lifted directly from the header file. Checking this table reveals that error code 1580 means “to-many relationship with too few destination objects,” which describes exactly what you tried to do.

**Table 5–1.** *Core Data Validation Errors*

| Constant  | Code | Description  |
|---|------|--|
| <code>NSManagedObjectValidationError</code>                   | 1550 | Generic validation error   |
| <code>NSValidationMultipleErrorsError</code>                  | 1560 | Generic message for error containing multiple validation errors  |
| <code>NSValidationMissingMandatoryPropertyError</code>        | 1570 | Nonoptional property with a nil value                            |
| <code>NSValidationRelationshipLacksMinimumCountError</code>   | 1580 | To-many relationship with too few destination objects            |
| <code>NSValidationRelationshipExceedsMaximumCountError</code> | 1590 | Bounded, to-many relationship with too many destination objects  |
| <code>NSValidationRelationshipDeniedDeleteError</code>        | 1600 | Some relationship with <code>NSDeleteRuleDeny</code> is nonempty |
| <code>NSValidationNumberTooLargeError</code>                  | 1610 | Some numerical value is too large                                |
| <code>NSValidationNumberTooSmallError</code>                  | 1620 | Some numerical value is too small                                |
| <code>NSValidationDateTooLateError</code>                     | 1630 | Some date value is too late                                      |
| <code>NSValidationDateTooSoonError</code>                     | 1640 | Some date value is too soon                                      |
| <code>NSValidationInvalidDateError</code>                     | 1650 | Some date value fails to match date pattern                      |
| <code>NSValidationStringTooLongError</code>                   | 1660 | Some string value is too long                                    |
| <code>NSValidationStringTooShortError</code>                  | 1670 | Some string value is too short                                   |
| <code>NSValidationStringPatternMatchingError</code>           | 1680 | Some string value fails to match some pattern                    |



The table of error codes gives you some clues about what Core Data will validate for you. The Core Data modeling tool for defining attributes and relationships gives you the same clues. For relationships, you can validate that to-many relationships have a valid number of destination objects—not too many and not too few. For attributes, however, you have a larger number of parameters you can validate, depending on the attribute type. For all the number types, you can specify a Min value and a Max value:

- Integer 16
- Integer 32
- Integer 64
- Decimal
- Double
- Float

Violating the acceptable range gives you an `NSValidationNumberTooSmallError` or an `NSValidationNumberTooLargeError`, depending on which is appropriate. If, for example, you set the Min value to 7.0 and the Max value to 10.0 for the radius attribute of the `Circle` entity, random circles that don't fall within that radius range would not save to the persistent store.

For the `String` type, you can specify values for the Min Length and Max Length of the string, as well as provide a regular expression to which the value must conform. The `Date` attribute is a little more tricky, however, because the Core Data modeling tool provides only text entry fields to enter Min and Max values, with no hint for how to format the values or whether any “magic” values like Today, Yesterday, 3 Days Ago, or 1 Year From Now are available. Scouring Apple's documentation yields nothing, but a little Google work uncovers blog posts by Shane Crawford (<http://shanecrawford.org/2008/57/coredatas-default-date-value/>) and Jeff LaMarche (<http://iphonedevdevelopment.blogspot.com/2009/07/core-data-default-dates-in-data-model.html>) that make Date Min and Max values (and Default values; see the “Default Values” section) sound really cool: you can specify natural-language strings like “now,” “today,” or “last Saturday.” Just as the thrill factor hits, you read in the posts that the natural-language strings are interpreted at compile time, not runtime, so they're not updated to reflect the current date, ever. This renders them useless.

You can put specific dates in these fields, however, like 1/1/1900, but it's difficult to find much use for that. The next section, “Custom Validation,” covers how to validate relative dates.

## Custom Validation

If you've generated your own `NSManagedObject`-derived classes for your data model, you can easily create your own validation routines that Core Data automatically invokes when customizing your objects. One option is to edit your `NSManagedObject`

implementation (\*.m) file and override the `validateValue:forKey:error` method, which has this signature:

```
- (BOOL)validateValue:(id *)ioValue forKey:(NSString *)key error:(NSError **)outError
```

It returns YES for valid and NO for not valid. This method gets called for each of the properties of your object, with the key value holding the property's name for each invocation. You are responsible for validating all the properties of your object by determining which key has been passed and doing appropriate validation for the property with that key.

Core Data offers a second, cleaner option that allows you to write individual validation routines for any or all of your properties and allows the default validation to process for any properties you haven't explicitly written validation routines for. To use this validation option, you write methods in your `NSManagedObject`-derived class's implementation file that take this form:

```
- (BOOL)validate<AttributeName>:(id *)ioValue error:(NSError **)outError
```

Substitute the name of the attribute you're trying to validate for `<AttributeName>`. For example, remove any validation you have on the radius attribute of the Circle entity in the Core Data modeler, and create a method in `Circle.m` that looks like this:

```
- (BOOL)validateRadius:(id *)ioValue error:(NSError **)outError {
    NSLog(@"Validating radius using custom method");

    if ([*ioValue floatValue] < 7.0 || [*ioValue floatValue] > 10.0) {
        // Fill out the error object
        if (outError != NULL) {
            NSString *msg = @"Radius must be between 7.0 and 10.0";
            NSDictionary *dict = [NSDictionary dictionaryWithObject:msg
                                                                    forKey:NSLocalizedDescriptionKey];
            NSError *error = [[NSError alloc] initWithDomain:@"Shapes" code:10 userInfo:dict];
            *outError = error;
        }
        return NO;
    }
    return YES;
}
```

This code retrieves the value passed in the `ioValue` parameter as a float and then compares it to the acceptable boundaries: 7.0 and 10.0. If the comparison fails, the code fills out an `NSError` object, if one was passed, with an arbitrary error code of 10. The code then returns NO to fail the validation. If the comparison passes, the code returns YES.

If you run this code and create a few shapes that include some circles, eventually the application will crash. You'll see messages in the log that look something like this:

```
2011-03-24 04:57:59.948 Shapes[65076:207] Validating radius using custom method
```

If any shapes fail validation, you'll see log messages that look like this:

```
2011-03-24 04:57:59.949 Shapes[65076:207] Unresolved error Error Domain=Shapes Code=10
"Radius must be between 7.0 and 10.0" UserInfo=0x590f5a0
```

Use this same mechanism to create date validations that validate date ranges relative to the current date. The following example validates that a date called `myDate` in the Core Data model falls between a week ago and a week from now:

```
- (BOOL)validateMyDate:(id *)ioValue error:(NSError **)outError {
    // 1 week = 60 sec/min * 60 min/hr * 24 hr/day * 7 day/week
    static int SECONDS_IN_A_WEEK = 60 * 60 * 24 * 7;

    NSLog(@"Validating myDate using custom method");

    // Get the passed date and calculate the valid dates
    NSDate *myDate = (NSDate *)(*ioValue);
    NSDate *minDate = [NSDate dateWithTimeIntervalSinceNow:-SECONDS_IN_A_WEEK];
    NSDate *maxDate = [NSDate dateWithTimeIntervalSinceNow:SECONDS_IN_A_WEEK];

    // Check if it's valid
    if ([myDate earlierDate:minDate] == myDate || [myDate laterDate:maxDate] == myDate) {
        // The date isn't valid, so construct an NSError if one was passed and return NO
        if (outError != NULL) {
            NSString *msg = @"myDate must fall between a week ago and a week from now";
            NSDictionary *dict = [NSDictionary dictionaryWithObject:msg
                                                                    forKey:NSLocalizedStringKey];
            NSError *error = [[NSError alloc] initWithDomain:@"Shapes" code:20 userInfo:dict];
            *outError = error;
        }
        return NO;
    }
    return YES;
}
```

Although `ioValue` is a pointer to an object reference, which would allow you to change the input value in the object graph, Apple's documentation strongly discourages you from doing this because it could create memory management issues.

The first method of validation, `validateValue:forKey:error`, trumps the `validate<AttributeName>` way. If you've provided a `validateValue` implementation, none of your `validate<AttributeName>` methods get called, and Core Data uses the answer that `validateValue` returns to determine the validity of an object.

You can also override the three other validation methods offered by `NSManagedObject`:

- `validateForInsert`
- `validateForUpdate`
- `validateForDelete`

Core Data calls these methods to invoke validation before inserting, updating, or deleting an object, respectively. These are the methods that call the `validateValue` methods, whether yours or the default `NSManagedObject` implementation. You can override these to do your own custom validation, but be sure to call the superclass

implementation first so that you get all the appropriate validation. One tricky piece to doing this is that if your code detects an error, it shouldn't override any existing errors from the validation in the superclass. Instead, it should combine any errors. Apple's documentation offers a method, `errorFromOriginalError`, that you can write in your own classes to combine errors. You can use this method, listed here, or you can write your own. Either way, be sure to combine errors and not overwrite them.

```
- (NSError *)errorFromOriginalError:(NSError *)originalError error:(NSError *)secondError
{
    NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
    NSMutableArray *errors = [NSMutableArray arrayWithObject:secondError];

    if ([originalError code] == NSValidationMultipleErrorsError) {
        [userInfo addEntriesFromDictionary:[originalError userInfo]];
        [errors addObjectsFromArray:[userInfo objectForKey:NSDetailedErrorsKey]];
    } else {
        [errors addObject:originalError];
    }

    [userInfo setObject:errors forKey:NSDetailedErrorsKey];

    return [NSError errorWithDomain:NSCocoaErrorDomain
                            code:NSValidationMultipleErrorsError
                            userInfo:userInfo];
}
```

You typically override one of the `validateFor...` methods to perform validation that depends on multiple properties. Each property might be valid on its own, but some combinations may not be valid. Consider, for instance, that you want to make a circle invalid for insertion if the `x` value is more than twice the `y` value. You create a method in `Circle.m` that first calls the superclass implementation of `validateForInsert` and then does its validation to make sure `x` isn't more than twice `y`. It could look like this:

```
- (BOOL)validateForInsert:(NSError **)outError {
    BOOL valid = [super validateForInsert:outError];

    // x can't be more than twice as much as y
    float fx = [self.x floatValue], fy = [self.y floatValue];
    if (fx >= (2 * fy)) {
        // Create the error if one was passed
        if (outError != NULL) {
            NSString *msg = @"x can't be more than twice as much as y";
            NSDictionary *dict = [NSDictionary dictionaryWithObject:msg
                                                                    forKey:NSLocalizedDescriptionKey];
            NSError *error = [[NSError alloc] initWithDomain:@"Shapes" code:30 userInfo:dict];
            *outError = error;
        }
    }
    // Combine this error with any existing ones
}
```

```

        *outError = [self errorFromOriginalError:*outError error:error];
    }
    }
    valid = NO;
}
return valid;
}

```

Note that it calls the `errorFromOriginalError` method listed earlier to combine any existing errors.

Test this method by running `Shapes` and clicking in the upper right quadrant of the top view. As soon as `Shapes` generates a circle in that upper right quadrant, where `x` is more than twice the value of `y`, the application crashes when it tries to save that circle.

## Invoking Validation

Core Data allows you to create objects with invalid attributes and invokes the validation routines when you save the managed object context. Objects with invalid values in their attributes or incorrect numbers of relationships can live comfortably in the managed object context. The validation doesn't occur until you try to save the managed object, at which time the save fails with the appropriate error codes.

If you don't want to wait for the `save:` method to be invoked, however, you can invoke validation manually. To do so, simply call any of the validation methods that you just learned about.

## Default Values

Core Data allows you to set default values for each of your attributes. You might have noticed that setting validation rules in your attributes without setting default values that pass those rules nets you compiler warnings that look something like this:

Misconfigured Property. Circle.radius has default value smaller than minimum value.

The compiler tries to protect you from creating attributes that violate your validation rules. You can safely ignore them and make sure to set valid values yourself, or you can take advantage of Core Data's offer to help.

Setting default values works exactly as you'd suppose: you type a valid default value into the Default field, and any new instance of that entity type will begin life with that value for that attribute. Try setting the default value for `radius` to 8.0, for example, and comment out the code that sets a random radius. You'll find that all created circles have a radius of 8.0.

The one oddity for default values is an attribute of type `Date`, as noted earlier. You can use an explicit date or a natural-language string like "today," but that string evaluates at compile time, not runtime, so doesn't do what you probably want (unless you want your dates to all commemorate your application's ship day by default). To create a default date that's based on the runtime date, add a method to your `NSManagedObject`-derived class

that overrides `NSObject`'s `awakeFromInsert:` method. In this method, called when the object is inserted into the managed object context, you can provide your own default value for any fields. The following implementation inserts today's date into the attribute `myDate` so that any new instances of this object automatically have today's date:

```
- (void)awakeFromInsert {
    [super awakeFromInsert];
    [self setValue:[NSDate date] forKey:@"myDate"];
}
```

## Undoing and Redoing

Golfers call it a mulligan. School-yard children call it a do-over. Computer users call it **Edit ► Undo**. Whatever you call it, you've realized that you've blundered and want to take back your last action. Not all scenarios afford you that opportunity, to which many broken-hearted lovers will attest, but Core Data forgives and allows you to undo what you've done using the standard Cocoa `NSUndoManager` mechanism. This section instructs you how to use it to allow your users to undo their Core Data changes.

The Core Data undo manager, an object of type `NSUndoManager`, lives in your managed object context, and `NSManagedObjectContext` provides a getter and a setter for the undo manager. Unlike Core Data on Mac OS X, however, the managed object context in iOS's Core Data doesn't provide an undo manager by default for performance reasons. If you want undo capabilities for your Core Data objects, you must set the undo manager in your managed object context yourself.

**NOTE:** Core Data on iOS doesn't provide an undo manager by default. You must set it yourself.

If you want to support undoing actions in your iOS application, you typically create the undo manager when you set up your managed object context, which usually happens in the getter for the managed object context. The Shapes application, for example, sets up the managed object context in the application delegate, as shown here. Note the code in bold, which adds an undo manager to the managed object context.

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

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];

        NSUndoManager *undoManager = [[NSUndoManager alloc] init];
        [__managedObjectContext setUndoManager:undoManager];
    }
    return __managedObjectContext;
}
```

As you can see, the code allocates and initializes the undo manager and sets it into the managed object.

Once the undo manager is set into the managed object context, it tracks any changes in the managed object context and adds them to the undo stack. You can undo those changes by calling the `undo:` method of `NSUndoManager`, and each change (actually, each undo group, as explained in the section “Undo Groups”) is rolled back from the managed object context. You can also replay changes that have been undone by calling `NSUndoManager`’s `redo:` method.

The `undo:` and `redo:` methods perform their magic only if the managed object context has any change to undo or redo, so calling them when no changes can be undone or redone does nothing. You can check, however, if the undo manager can undo or redo any changes by calling the `canUndo:` and `canRedo:` methods, respectively.

## Undo Groups

By default, the undo manager groups all changes that happen during a single pass through the application’s run loop into a single change that can be undone or redone as a unit. This means, for example, that in the Shapes application, each shape creation can be undone or redone individually, because each shape is created in response to a touch event. When you shake the device, however, a single shake event occurs, and all the shapes are deleted in the method you call in response to that event. You can undo the deletion of all the shapes or call `redo:` to undo the deletion of all the shapes, but by default you can’t undo the deletion or redo the creation of single shapes.

You alter this behavior by turning off automatic grouping completely and managing the undo groups yourself. To accomplish this, pass `NO` to `setGroupsByEvent`. You become responsible, then, for creating all undo groups, because the undo manager will no longer create them for you. You create the undo group by calling `beginUndoGrouping:` to start creating the group and `endUndoGrouping:` to complete the undo group. These calls must be matched, or an exception of type `NSInternalInconsistencyException` is raised. You could, for example, create an undo group for each shape deletion in Shapes so that you can undo the deletion one shape at a time. You can also span undo groups across events, so you could, for example, group three shape creations and undo the three shapes with a single call to `undo`.

## Limiting the Undo Stack

By default, the undo manager tracks an unlimited number of changes for you to undo and redo. This can cause memory issues, especially on iOS devices. You can limit the size of the undo stack by calling `NSUndoManager`’s `setLevelsOfUndo:` method, passing an unsigned integer that represents the number of undo groups to retain on the undo stack. You can inspect the current undo stack size, measured in the number of undo groups, by calling `levelsOfUndo:`, which returns an unsigned integer. A value of 0 represents no limit. If you’ve imposed a limit on the size of the undo stack, the oldest undo groups roll off the stack to accommodate the newer groups.

## Disabling Undo Tracking

Once you create an undo manager and set it into the managed object context, any changes you make to the managed object context are tracked and can be undone. You can disable undo tracking, however, by calling `NSUndoManager's disableUndoRegistration:` method. To reenable undo tracking, call `NSUndoManager's enableUndoRegistration:` method. Disabling and enabling undo tracking uses a reference counting mechanism, so multiple calls to `disableUndoRegistration:` require an equal number of calls to `enableUndoRegistration:` before undo tracking becomes enabled again.

Calling `enableUndoRegistration:` when undo tracking is already enabled raises an exception of type `NSInternalInconsistencyException`, which will likely crash your application. To avoid this embarrassment, you can call `NSUndoManager's isUndoRegistrationEnabled:`, which returns a `BOOL`, before calling `enableUndoRegistration:`. For example, the following code checks whether undo tracking is enabled before enabling it:

```
if (![undoManager isUndoRegistrationEnabled]) {
    [undoManager enableUndoRegistration];
}
```

You can clear the undo stack entirely by calling the `removeAllActions:` method. This method has the side effect of reenabling undo tracking.

## Adding Undo to Shapes

The rest of this section puts into practice the concepts explained about undo managers by adding undo and redo support to the Shapes application. To begin, change the accessor for the managed object context in the application delegate to create an undo manager for the context. Set the undo stack to an arbitrary size of 10, as this code shows:

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

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];

        // Set up the undo manager
        NSUndoManager *undoManager = [[NSUndoManager alloc] init];
        [undoManager setLevelsOfUndo:10];
        [__managedObjectContext setUndoManager:undoManager];
    }
    return managedObjectContext_;
}
```



The typical interface for undoing actions on an iOS device is to shake the device, but Shapes already uses that to delete all the shapes from the persistent store. Instead, Shapes will provide two buttons, one to undo the last change and one to redo it. Open the `ShapesViewController.h` file, and add members for the two buttons, methods to invoke when the buttons are pressed, and a method that hides the buttons when the managed object context has no changes to undo or redo. The code should look like this, with the added lines in bold:

```
#import <UIKit/UIKit.h>
#import <CoreData/CoreData.h>
#import "BasicCanvasUIView.h"

@interface ShapesViewController : UIViewController {
    NSManagedObjectContext *managedObjectContext;
    IBOutlet BasicCanvasUIView *topView;
    IBOutlet BasicCanvasUIView *bottomView;
    IBOutlet UIButton *undoButton;
    IBOutlet UIButton *redoButton;
}
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain) BasicCanvasUIView *topView;
@property (nonatomic, retain) BasicCanvasUIView *bottomView;
@property (nonatomic, retain) UIButton *undoButton;
@property (nonatomic, retain) UIButton *redoButton;

- (IBAction)undo:(id)sender;
- (IBAction)redo:(id)sender;
- (void)updateUndoAndRedoButtons:(NSNotification *)notification;

@end
```

Add `@synthesize` directives to `ShapesViewController.m` for `undoButton` and `redoButton`. Add implementations for the `undo:`, `redo:`, and `updateUndoAndRedoButtons:` methods. The `undo` method should get the undo manager from the managed object context and call the `undo:` method and then tell the views to repaint themselves, like this:

```
- (IBAction)undo:(id)sender {
    [[self.managedObjectContext undoManager] undo];
    [topView setNeedsDisplay];
    [bottomView setNeedsDisplay];
}
```

The `redo:` method should do the same thing but call the undo manager's `redo:` method instead of its `undo:` method:

```
- (IBAction)redo:(id)sender {
    [[self.managedObjectContext undoManager] redo];
    [topView setNeedsDisplay];
    [bottomView setNeedsDisplay];
}
```

To show the Undo button only when the user can undo a change and the Redo button only when the user can redo a change, you could insert code everywhere you make data

changes to update the buttons. Take advantage of Cocoa Touch's notification mechanism, however, and have the managed object context notify you whenever its managed data changes. In the `viewDidLoad:` method, add code to call your `updateUndoAndRedoButtons:` method any time the data changes. You also want to call the `updateUndoAndRedoButtons:` method when the view first loads, so tack on a call to that method. This is the code you should add to `viewDidLoad:`, after setting the canvas instances into the views:

```
...
// Set the Canvas instances into the views
topView.canvas = canvas1;
bottomView.canvas = canvas2;

// Register for changes to the managed object context
NSNotificationCenter *notificationCenter = [NSNotificationCenter defaultCenter];
[notificationCenter addObserver:self selector:@selector(updateUndoAndRedoButtons:)
name:NSManagedObjectContextObjectsDidChangeNotification object:nil];
[self updateUndoAndRedoButtons:nil];
```

In the `viewDidUnload:` method, undo the notifications, like so:

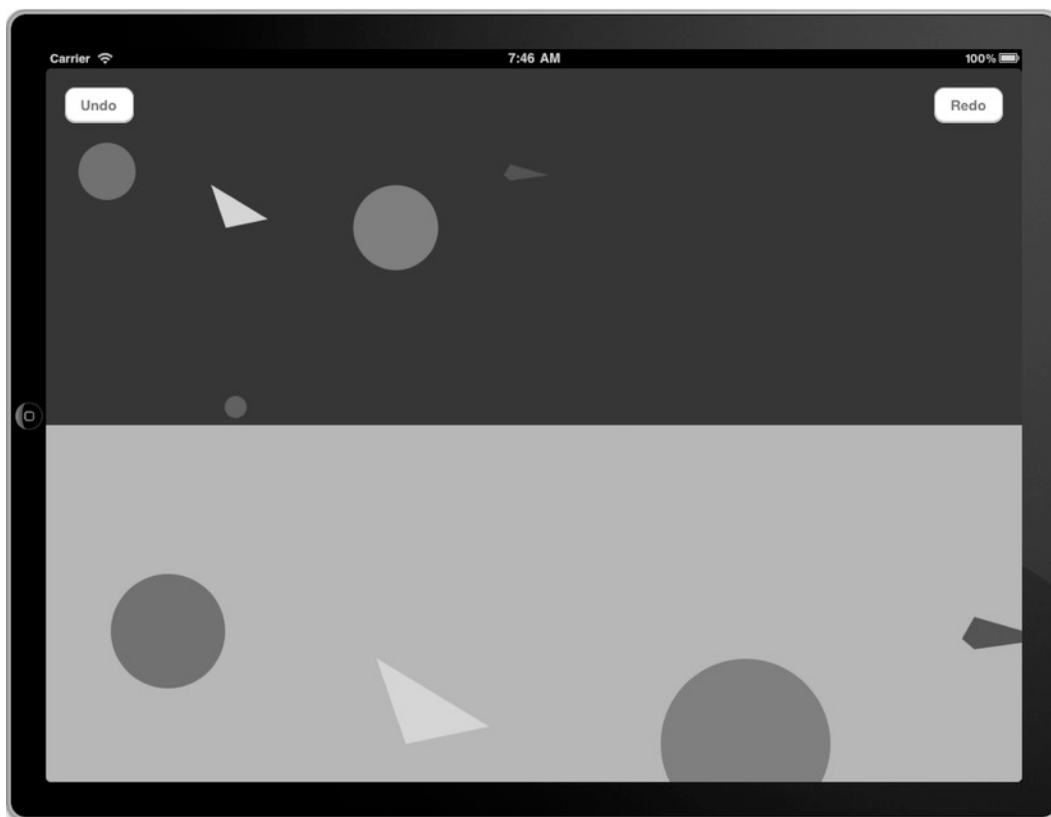
```
- (void)viewDidUnload {
    [super viewDidUnload];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

The `updateUndoAndRedoButtons:` method hides the buttons appropriately when the application has nothing to undo or redo. It looks like this:

```
- (void)updateUndoAndRedoButtons:(NSNotification *)notification {
    NSUndoManager *undoManager = [self.managedObjectContext undoManager];
    undoButton.hidden = ![undoManager canUndo];
    redoButton.hidden = ![undoManager canRedo];
}
```

Next, open `ShapesViewController.xib`, and drag two Round Rect Button instances to the top corners of the screen—one on the left and one on the right. Change the label on the one on the left to Undo and the one on the right to Redo. Set the Autosizing appropriately for each button. The Undo button should have the bands on the top and left only, and the Redo button should have the bands on the top and right only. Ctrl+drag from the File's Owner icon to the Undo button, and wire it to `undoButton`; repeat the process to wire the Redo button to `redoButton`. Wire the Touch Up Inside events for each button to the methods you created—the Undo button to the `undo:` method and the Redo button to the `redo:` method.

After making these changes, build and run the Shapes application. As you add shapes and undo changes, you should see the Undo and Redo buttons as in Figure 5-17. Add shapes, change the colors of shapes by rotating the device, and delete the shapes. Click Undo and Redo and see the data changes undo and redo themselves.



**Figure 5–17.** *The Undo and Redo buttons*

For a small amount of work, you can support undoing and redoing actions in your Core Data applications. Users expect mulligans, so make the small effort to give them to your users.

## Summary

This chapter covered the heart of the purpose of Core Data: to create, retrieve, update, and delete data in a persistent store. Like Ruby on Rails, Core Data provides simple yet powerful mechanisms to carry out these CRUD operations. Whether talking directly to your `NSManagedObject` instances or using custom classes to represent your data objects, you can perform CRUD operations through the interfaces Core Data provides you without getting into the messy details of SQL or other data storage details.

One area where Core Data trumps Ruby on Rails is its native support for undoing data operations. Although efforts continue to add Undo/Redo support for Rails, no solution built into Rails provides an easy way to undo data-related operation like Core Data does.

Although now you know almost all you need for writing usable Core Data applications, don't stop here. The next chapter talks about how to refine result sets so that you can filter, sort, and aggregate the data you retrieve from your persistent store.

## Refining Result Sets

Billing itself as “the ultimate automotive marketplace,” AutoTrader.com allows consumers in the United States to buy and sell cars. According to census data, the United States has around 250 million registered motor vehicles, so you can imagine that finding the perfect car can daunt even the most intrepid prospective car buyer. To help car buyers cut through the myriad cars they don’t want to find the one they do, AutoTrader.com provides tools on its web site to filter available cars by criteria such as body style, make, model, year, price, and location.

When dealing with large sets of data, refining result sets is essential for analysts to extract any meaning from the data. Computers, including iDevices, excel at narrowing data sets to make it easier for computer users to understand the data with which they’re working. Imagine, for example, how much more difficult calling someone from your iPhone would be if the Contacts application didn’t sort your contacts alphabetically or allow you to search for a contact.

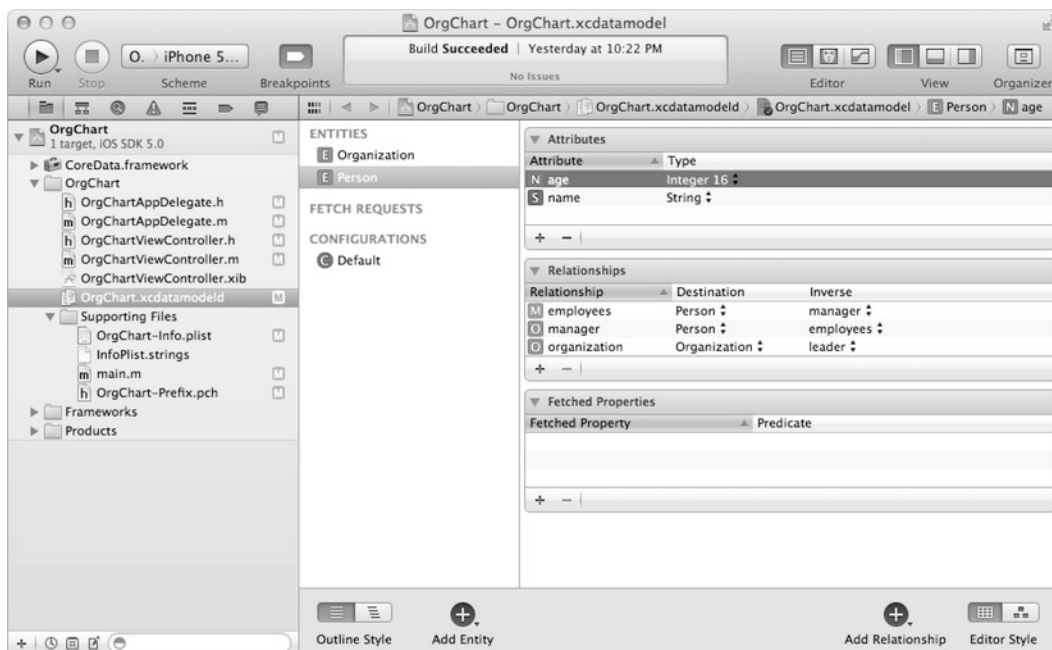
This chapter demonstrates how to sort, filter, and aggregate data using Core Data’s mechanisms so you can help your application’s users find the data they need.

### Building the Test Application

You can get data out of a Core Data persistent store in two ways. If you already have handles to existing managed objects, then you can follow the object graph and relationships to pull more objects out. If you don’t have any objects yet, such as when the application first starts, then you must go get them directly out of the persistent store. In this case, you typically use an instance of `NSFetchRequest` to go get objects. You initialize an `NSFetchRequest` with an `NSEntityDescription`, which helps narrow the result set by constraining the type of managed objects to retrieve. This chapter shows you how to narrow your fetch requests in various ways using the `NSFetchRequest` class.

To support the examples in this chapter, you’ll reuse the `OrgChart` application from Chapter 2. We suggest you make a copy of the application because you will make some changes to Chapter 2’s data model in this chapter. Since Chapter 2, you’ve learned that Core Data can better deal with managing the object graph if all the relationships have an

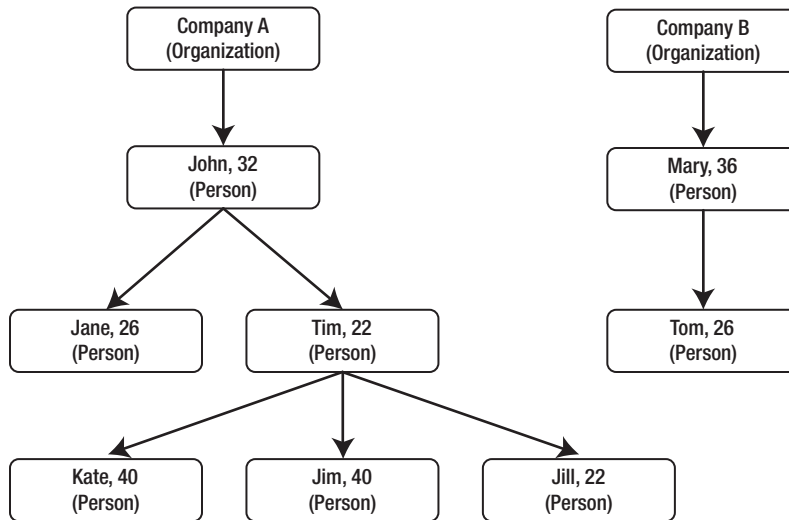
inverse, so go back to the OrgChart application and add reverse relationships for leader and employees. To accomplish this, open the `OrgChart.xcdatamodeld` data model, and select the `Person` entity. Since `Organization` already has the `leader` relationship to the `Person` entity, add a relationship called `organization` in `Person`, set the destination to `Organization`, and select `leader` as the inverse relationship. `Person` also has a self-referencing relationship called `employees`, so create a relationship from `Person` to `Person`, call it `manager`, and select `employees` as the inverse relationship. Also, rename the `id` attribute of `Person` to `age`. Your `Person` entity should look like Figure 6–1.



**Figure 6–1.** *The Person entity with inverse relationships*

## Creating the Org Chart Data

The `OrgChart` application is a very simple application with no fancy user interface—just a blank, gray window. For the purposes of this chapter, you will use the `OrgChart` application to retrieve the data from the persistent store. In the `OrgChartAppDelegate.m` file, the `createData` method populates the data store. The data it uses to populate the data store is represented in the org chart shown in Figure 6–2.



**Figure 6-2.** The sample org chart data set

Listing 6-1 shows the `createData` method that populates the data according to Figure 6-2. Open `OrgChartAppDelegate.m`, and replace the `createData` method from Chapter 2 with Listing 6-1's code.

**Listing 6-1.** The `createData` Method That Populates the Org Chart

```

- (void)createData
{
    NSManagedObjectContext *context = [self managedObjectContext];

    NSEntityDescription *orgEntity = [NSEntityDescription entityForName:@"Organization"
inManagedObjectContext:context];
    NSEntityDescription *personEntity = [NSEntityDescription entityForName:@"Person"
inManagedObjectContext:context];

    { // Company A
        NSManagedObject *organization = [NSEntityDescription
insertNewObjectForEntityForName:[orgEntity name] inManagedObjectContext:context];

        [organization setValue:@"Company A" forKey:@"name"];
        int orgId = [organization hash];
        [organization setValue:[NSNumber numberWithInt:orgId] forKey:@"id"];

        NSManagedObject *john = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
        [john setValue:@"John" forKey:@"name"];
        [john setValue:[NSNumber numberWithInt:32] forKey:@"age"];

        NSManagedObject *jane = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
        [jane setValue:@"Jane" forKey:@"name"];
        [jane setValue:[NSNumber numberWithInt:26] forKey:@"age"];
    }
}

```

```

    NSManagedObject *tim = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [tim setValue:@"Tim" forKey:@"name"];
    [tim setValue:[NSNumber numberWithInt:22] forKey:@"age"];

    NSManagedObject *jim = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [jim setValue:@"Jim" forKey:@"name"];
    [jim setValue:[NSNumber numberWithInt:40] forKey:@"age"];

    NSManagedObject *kate = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [kate setValue:@"Kate" forKey:@"name"];
    [kate setValue:[NSNumber numberWithInt:40] forKey:@"age"];

    NSManagedObject *jill = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [jill setValue:@"Jill" forKey:@"name"];
    [jill setValue:[NSNumber numberWithInt:22] forKey:@"age"];

    NSMutableSet *johnsEmployees = [john mutableSetValueForKey:@"employees"];
    [johnsEmployees addObject:jane];
    [johnsEmployees addObject:tim];

    NSMutableSet *timsEmployees = [tim mutableSetValueForKey:@"employees"];

    [timsEmployees addObject:jim];
    [timsEmployees addObject:kate];
    [timsEmployees addObject:jill];

    [organization setValue:john forKey:@"leader"];
}

{ // Company B
    NSManagedObject *organization = [NSEntityDescription
insertNewObjectForEntityForName:[orgEntity name] inManagedObjectContext:context];

    [organization setValue:@"Company B" forKey:@"name"];
    int orgId = [organization hash];
    [organization setValue:[NSNumber numberWithInt:orgId] forKey:@"id"];

    NSManagedObject *mary = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [mary setValue:@"Mary" forKey:@"name"];
    [mary setValue:[NSNumber numberWithInt:36] forKey:@"age"];

    NSManagedObject *tom = [NSEntityDescription
insertNewObjectForEntityForName:[personEntity name] inManagedObjectContext:context];
    [tom setValue:@"Tom" forKey:@"name"];
    [tom setValue:[NSNumber numberWithInt:26] forKey:@"age"];

    [[mary mutableSetValueForKey:@"employees"] addObject:tom];

    [organization setValue:mary forKey:@"leader"];
}

```



```

    }

    [self saveContext];
}

```

The next step is to edit the `application:didFinishLaunchingWithOptions:` method to call `createData`, like so:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self createData];

    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[OrgChartViewController alloc]
initWithNibName:@"OrgChartViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Delete the existing data store from when you ran `OrgChart` in Chapter 2 and launch the application. It should start, display the gray screen, and have no output. The application has created the persistent store and populated it with the org chart data that you use throughout the rest of the chapter for sorting, filtering, and aggregating result sets. You can verify that the application created the persistent store using the techniques outlined in earlier chapters to find the SQLite data store and run queries against it using the `sqlite3` command-line tool. See Chapter 2 for more information.

## Reading and Outputting the Data

Now that you've created your persistent store and inserted all the org chart test data, you don't need to call the `createData:` method anymore. Stop the application and comment out the `createData:` call in the `application:didFinishLaunchingWithOptions:` method. Right below it, add a call to the `readData:` method so that the application reads the data and outputs it to the Debugger Console when the application launches. Through this chapter, you'll replace the `readData:` method multiple times to read and output different data from your org chart data model. The `application:didFinishLaunchingWithOptions:` method should now look like this:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    //[self createData];
    [self readData];

    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[OrgChartViewController alloc]
initWithNibName:@"OrgChartViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
}

```

```
    return YES;
}
```

Next, edit the `displayPerson:withIndentation:` method to display the person's age since you just added this new field, like so:

```
- (void)displayPerson:(NSManagedObject*)person withIndentation:(NSString*)indentation
{
    NSLog(@"%@Name: %@ (%@)", indentation, [person valueForKey:@"name"], [person
valueForKey:@"age"]);

    // Increase the indentation for sub-levels
    indentation = [NSString stringWithFormat:@"%@" " ", indentation];

    NSSet *employees = [person valueForKey:@"employees"];
    id employee;
    NSEnumerator *it = [employees objectEnumerator];
    while((employee = [it nextObject]) != nil)
    {
        [self displayPerson:employee withIndentation:indentation];
    }
}
```

Launch the application again, and the log output will look similar to the following example, showing that your two organizations are there with the proper org charts. (The ordering in your output may differ—for example, Company B could come before Company A—but that's not important.)

```
2011-07-31 20:59:09.057 OrgChart[9928:f203] Organization: Company A
2011-07-31 20:59:09.059 OrgChart[9928:f203]     Name: John (32)
2011-07-31 20:59:09.059 OrgChart[9928:f203]     Name: Tim (22)
2011-07-31 20:59:09.060 OrgChart[9928:f203]     Name: Kate (40)
2011-07-31 20:59:09.060 OrgChart[9928:f203]     Name: Jim (40)
2011-07-31 20:59:09.061 OrgChart[9928:f203]     Name: Jill (22)
2011-07-31 20:59:09.061 OrgChart[9928:f203]     Name: Jane (26)
2011-07-31 20:59:09.062 OrgChart[9928:f203] Organization: Company B
2011-07-31 20:59:09.063 OrgChart[9928:f203]     Name: Mary (36)
2011-07-31 20:59:09.063 OrgChart[9928:f203]     Name: Tom (26)
```

Your data set is now ready and verified. The remainder of this chapter deals with extracting only certain objects and rearranging them based on constraints you specify.

## Filtering

Core Data uses the `NSPredicate` class from the Foundation framework to specify how to select the objects that should be part of the result set when executing a fetch request. Predicates can be built in two ways: either you can build the object and its graph manually or you can use the query language `NSPredicate` implements. You will find that most of the time you will prefer the convenience and readability of the latter method.

This section looks at both ways of building predicates for each example in order to help draw a parallel between the query language and the `NSPredicate` object graph.

Understanding both ways will help you determine your preferred approach. You can also mix and match the approaches depending on your applications and data scenarios.

The predicate query language builds an `NSPredicate` object graph from a string representation that has similarities with a SQL `WHERE` clause. The method that builds a predicate from the language is `+predicateWithFormat:(NSString*)`. For instance, if you want to get the managed object that represents Jane, you build the predicate by calling `[NSPredicate predicateWithFormat:@"name = 'Jane']`.

To test this, modify the `readData:` method as follows:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
    inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name='Jane'"];
    [fetchRequest setPredicate:predicate];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}
```

Running the application again yields the following expected output:

```
name=Jane age=26
```

In its simplest form, a predicate executes a test on an object. If the test succeeds, then the object is part of the result set. If the test is negative, then the object isn't included in the result set. In the previous example, `name='Jane'` is an example of a simple predicate. The key path `name` and the constant string value `'Jane'` are called *expressions*. The operator `=` is called a *comparator*.

## Expressions for a Single Value

Expressions, represented by the `NSExpression` class, are the basic building blocks of a predicate. They represent values that can be compared with each other or simply evaluated by themselves. The simplest and most common type of expression is the expression that represents a single value (as opposed to representing a collection of values). `NSPredicate` supports four types of single-value expressions, as shown in Table 6-1.

**Table 6–1.** *The Four Types of Single-Value Expressions*

| Type             | Static Initializer Method     | Description  |
|------------------|-------------------------------|--|
| Constant value   | +expressionForConstantValue:  | Creates an expression to represent the given object.   |
| Evaluated object | +expressionForEvaluatedObject | When the predicate is run in order to decide whether an object should be part of the result, the evaluated object is the object in question. This method creates an expression that represents the object being evaluated for inclusion in the result set. |
| Key path         | +expressionForKeyPath:        | Managed objects are Key-Value Coding (KVO) compliant. This method creates an expression that represents the value for the given key in the evaluated object.   |
| Variable         | +expressionForVariable:       | A value from the variable bindings dictionary associated with the evaluated object.  |

The previous example using the query language had two expressions: name and Jane. The name expression is a key path into the managed object, while the Jane expression is a constant value. Creating the same expressions in code looks like this:

```
NSExpression *exprName = [NSExpression expressionForKeyPath:@"name"];
NSExpression *exprJane = [NSExpression expressionForConstantValue:@"Jane"];
```

## Expressions for a Collection

Some comparisons require more than one right-side expression. This is the case, for example, when you want to compare a value against a range of values. In this situation, you need to provide a lower bound value and an upper bound value. Predicates use arrays of expressions to represent multiple values. The following example shows how to build a range of values. First, you create the expressions representing the single values; you then put them into an expression representing the array using the `expressionForAggregate:` method.

```
NSExpression *lower = [NSExpression expressionForConstantValue:[NSNumber numberWithInt:20]];
NSExpression *upper = [NSExpression expressionForConstantValue:[NSNumber numberWithInt:35]];
NSExpression *exprRange = [NSExpression expressionForAggregate:[NSArray arrayWithObjects:lower, upper, nil]];
```

**NOTE:** Although the `NSExpression` class provides a way to build an expression from a selector using the `expressionForFunction:arguments:` method, this mechanism is not supported by Core Data.

## Comparison Predicates

Expressions aren't predicates by themselves. The other element required is a comparator to evaluate the predicate. Two expressions are compared with each other using the `NSComparisonPredicate` class, which is a subclass of `NSPredicate`. The comparator is a binary operator between a left expression and a right expression. Table 6–2 lists them and describes how `NSComparisonPredicate` uses the left (L) and the right (R) expressions to determine the outcome of the comparison based on the selected type.

**Table 6–2.** *The Predefined Comparators*

| Type   | Query Language              | Logical Description                                 |
|--|-----------------------------|---|
| <code>NSLessThanPredicateOperatorType</code>             | <code>L &lt; R</code>       | L less than R.                                      |
| <code>NSLessThanOrEqualToPredicateOperatorType</code>    | <code>L &lt;= R</code>      | L less than or equal to R.                          |
| <code>NSGreaterThanPredicateOperatorType</code>          | <code>L &gt; R</code>       | L greater than R.                                   |
| <code>NSGreaterThanOrEqualToPredicateOperatorType</code> | <code>L &gt;= R</code>      | L greater than or equal to R.                       |
| <code>NSEqualToPredicateOperatorType</code>              | <code>L = R</code>          | L equals R.   |
| <code>NSNotEqualToPredicateOperatorType</code>           | <code>L != R</code>         | L different from R.                                 |
| <code>NSMatchesPredicateOperatorType</code>              | <code>L MATCHES R</code>    | L matches the R regular expression.                 |
| <code>NSLikePredicateOperatorType</code>                 | <code>L LIKE R</code>       | <code>L = R</code> where R can contain * wildcards. |
| <code>NSBeginsWithPredicateOperatorType</code>           | <code>L BEGINSWITH R</code> | L starts with R.                                    |
| <code>NSEndsWithPredicateOperatorType</code>             | <code>L ENDSWITH R</code>   | L ends with R.                                      |
| <code>NSInPredicateOperatorType</code>                   | <code>L IN R</code>         | L belongs to the collection R.                      |
| <code>NSContainsPredicateOperatorType</code>             | <code>L CONTAINS R</code>   | L is a collection that contains item R.             |
| <code>NSBetweenPredicateOperatorType</code>              | <code>L BETWEEN R</code>    | L is a value between the two values of the array R. |

**NOTE:** `NSLikePredicateOperatorType` is a simplified version of `NSMatchesPredicateOperatorType`. While `NSMatchesPredicateOperatorType` can use complex regular expressions, the `NSLikePredicateOperatorType` type uses simple wildcard replacement (\*) characters. So, the expression `name like 'J*n*'` would match Jane and John but not Jim.

The `name='Jane'` predicate could be written in code without going through the query language by using `NSExpression` and `NSComparisonPredicate`, like so:

```
NSExpression *exprName = [NSExpression expressionForKeyPath:@"name"];
NSExpression *exprJane = [NSExpression expressionForConstantValue:@"Jane"];
NSPredicate *predicate = [NSComparisonPredicate predicateWithLeftExpression:exprName
rightExpression:exprJane modifier:NSDirectPredicateModifier
type:NSEqualToPredicateOperatorType options:0];
```

You could rewrite the `readData:` method using the predicate object graph rather than the query language, and the output would still be the same. Obviously, most programmers prefer using the query language, but it's a good exercise to see how the query language translates into predicate objects behind the scenes. The rewrite looks like this:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSExpression *exprName = [NSExpression expressionForKeyPath:@"name"];
    NSExpression *exprJane = [NSExpression expressionForConstantValue:@"Jane"];
    NSPredicate *predicate = [NSComparisonPredicate predicateWithLeftExpression:exprName
rightExpression:exprJane modifier:NSDirectPredicateModifier
type:NSEqualToPredicateOperatorType options:0];

    [fetchRequest setPredicate:predicate];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}
```

Running the application with this version of `readData:` reveals the same output as before.

```
name=Jane age=26
```

You might have noticed that the fifth argument of the static `init` method you just used is options. Options are used to modify the behavior of the comparator. Options are not always applicable but are used with the string comparison comparators only. They can also be specified in the query language by appending the option code between square brackets right after the operator. Table 6–3 lists the available options as well as how to use them in the query language.

**Table 6–3.** *The String Comparison Options*

| Option   | Code           | Description   | Query Language Example                   |
|--|----------------|---|--|
| <code>NSCaseInsensitivePredicateOption</code>      | <code>c</code> | The comparison should not be case sensitive.  | <code>"X" =[c] "x"</code>                |
| <code>NSDiacriticInsensitivePredicateOption</code> | <code>d</code> | The comparison should overlook accents.   | <code>"é" =[d] "e"</code>                |
| <code>NSNormalizedPredicateOption</code>           | <code>n</code> | Indicates that the operands have been preprocessed (that is, made all the same case, accents removed, and so on), so <code>NSCaseInsensitivePredicateOption</code> and <code>NSDiacriticInsensitivePredicateOption</code> options can be overlooked if specified. | <code>"abc" =[n]<br/>"abc"</code>        |
| <code>NSLocaleSensitivePredicateOption</code>      | <code>l</code> | Indicates that the comparison should take the current locale into consideration.  | <code>"straße" =[l]<br/>"strasse"</code> |

To specify multiple options, you simply use the binary OR operator (`|`) between them. In the query language, you append the codes together, as shown in the following example:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name =[cd] 'jàne'"];
```

Note that `predicateWithFormat:` accepts string formats. So, the previous example could also be written like so:

```
[NSPredicate predicateWithFormat:@"name =[cd] %@", @"jàne"];
```

If the right expression is a collection rather than a single object, the modifier can be used to indicate what the right expression is referring to; the modifiers are explained in Table 6–4.

Table 6–4. The Comparison Predicate Modifiers

| Modifier                  | Query Language Example | Description   |
|---------------------------|------------------------|---|
| NSDirectPredicateModifier | X                      | The right expression refers to the given expression itself.   |
| NSAllPredicateModifier    | ALL X                  | The comparator returns true only if the comparison returns TRUE for every object in the collection.                       |
| NSAnyPredicateModifier    | ANY X                  | The comparator returns true only if there is at least one object in the collection for which the comparison returns TRUE. |

## Compound Predicates

You just looked at a simple example for a predicate that used only one criterion (name = 'Jane'). Predicates are often formed by a logical combination of multiple subpredicates. If you want to find all the people with a first name that starts with the letter *J* and with an age between 20 and 35, the system should find John, Jane, and Jill, who are 32, 26, and 22 years old, respectively, but not Jim, who is 40 years old. Core Data allows you to build compound predicates using the `NSCompoundPredicate` class. The compounding operator is one of the three main logic operators: OR, AND, and NOT. Compound predicates can be built from one the three following methods, as shown in Table 6–5.

Table 6–5. Methods for Creating Compound Predicates

| Method Name                                 | Query Language Example | Description  |
|---|------------------------|--|
| <code>andPredicateWithSubpredicates:</code> | P1 AND P2 AND P3       | Returns TRUE only if all the predicates in the given collection of predicates are TRUE as well.    |
| <code>orPredicateWithSubpredicates:</code>  | P1 OR P2 OR P3         | Returns TRUE only if at least one of the predicates in the given collection of predicates is TRUE. |
| <code>notPredicateWithSubpredicate:</code>  | NOT P                  | Returns TRUE only if the given predicate is FALSE.   |



For this example, you build the predicate manually as follows:

```

NSExpression *exprName = [NSExpression expressionForKeyPath:@"name"];
NSExpression *exprJ = [NSExpression expressionForConstantValue:@"J"];

NSPredicate *p1 = [NSComparisonPredicate predicateWithLeftExpression:exprName
rightExpression:exprJ modifier:NSDirectPredicateModifier
type:NSBeginsWithPredicateOperatorType options:0];

NSExpression *exprAge = [NSExpression expressionForKeyPath:@"age"];

NSExpression *lower = [NSExpression expressionForConstantValue:[NSNumber
 numberWithInt:20]];
NSExpression *upper = [NSExpression expressionForConstantValue:[NSNumber
 numberWithInt:35]];
NSExpression *exprRange = [NSExpression expressionForAggregate:[NSArray
 arrayWithObjects:lower, upper, nil]];

NSPredicate *p2 = [NSComparisonPredicate predicateWithLeftExpression:exprAge
rightExpression:exprRange modifier:NSDirectPredicateModifier
type:NSBetweenPredicateOperatorType options:0];

NSPredicate *predicate = [NSCompoundPredicate andPredicateWithSubpredicates:[NSArray
 arrayWithObjects:p1, p2, nil]];

```

After updating the `readData:` method with this predicate, running the application yields the following expected output.

```

name=John age=32
name=Jane age=26
name=Jill age=22

```

As shown in Table 6–5, the query language also supports compound predicates using the OR, AND, and NOT operators. The same example could be rewritten more simply using the query language, like so:

```

NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name BEGINSWITH %@ AND age
BETWEEN {%d, %d}", @"J", 20, 35];

```

**NOTE:** If you can't figure out the syntax of the query language, You can always build the predicate object graph and then print it to the console using `NSLog(@"%@", myPredicate)`. It will give you the exact syntax to use.

The result of rewriting the `readData` method with the string-based predicated is shown here. Running it yields the same result, as expected.

```

- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
    inManagedObjectContext:context];

```

```

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name BEGINSWITH %@ AND age BETWEEN {%d, %d}", @"J", 20, 35];
    [fetchRequest setPredicate:predicate];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}

```

## Subqueries

In some cases, you want to find objects based on criteria that pertain to other related objects. For example, you might want to find all the managers who have at least one direct report whose age is 26. According to the test data set, this should return Mary and John, since both have 26-year-old direct reports. Without subqueries, you can't fetch this result set directly. Instead, you would need to perform two steps: first, fetch all the persons of the right age, and then iterate through each of them to find their managers. This solution is shown here:

```

- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
    inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"age=26"];
    [fetchRequest setPredicate:predicate];

    NSArray *matches = [context executeFetchRequest:fetchRequest error:nil];

    NSMutableArray *persons = [[NSMutableArray alloc] init];
    for (NSManagedObject *obj in matches)
    {
        [persons addObject:[obj valueForKey:@"manager"]];
    }

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}

```

Subquery expressions provide a way to encapsulate the solution into a single query, eliminating the need to manually post-process the data after retrieving it. Your entire predicate and logic can be replaced with "SUBQUERY(employees, \$x, \$x.age == 26)".

@count > 0", which returns true for any person with an employees relationship that has at least one employee who is 26 years old. A subquery iterates through a collection expression (employees in this case) and creates a temporary variable (\$x in this example, but name it however you'd like), which takes the value of each item in the collection to evaluate the predicate. The syntax @count returns a count of the items in the collection. (You will learn more about aggregators later in this chapter.) Using this predicate, you can clean up the readData: method, like so:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
    inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"(SUBQUERY(employees,
    $x, $x.age == %d).@count > 0)", 26];
    [fetchRequest setPredicate:predicate];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}
```

Another way to return the same result would be to use the expression modifier and apply the predicate to any of the evaluated object's employees. The predicate would then look like this:

```
[NSPredicate predicateWithFormat:@"ANY employees.age == %d", 26].
```

Unfortunately, this kind of logic is risky because it works only with simple predicates. If you had a compound predicate, the logic would be flawed. Imagine, for example, that you wanted to return all the managers who not only have a direct report whose age is 26 years old but also have a name that starts with the letter *T*. According to your test data, this would only match Mary because she has an employee named Tom who is 26 years old. It would no longer match John because although Jane is 26 years old, her name does not start with *T*. If you tried to use the expression modifier ANY, you would write your predicate like this:

```
[NSPredicate predicateWithFormat:@"ANY employees.age == %d AND ANY employees.name BEGINS
WITH %@", 26, @"T"];
```

The resulting readData: method is as follows:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
    inManagedObjectContext:context];
```

```

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:entity];

NSPredicate *predicate = [NSPredicate predicateWithFormat:@"ANY employees.age == %d
AND ANY employees.name BEGINSWITH %@", 26, @"T"];
[fetchRequest setPredicate:predicate];

NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

for (NSManagedObject *person in persons)
{
    NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
}

```

Running the test yields the following output, which is wrong:

```

name=John age=32
name=Mary age=36

```

The problem you run into is that John has two employees: Jane, age 26, and Tim, age 22. Jane matches the criterion for age, and Tim matches the criterion for the first letter in the name. The problem is that you want the ANY modifier to encapsulate both conditions instead of ANY one and ANY the other. This negative example illustrates why subqueries are a more adapted tool for this problem. The following implementation of `readData:` shows how to use the subquery with the compounded predicate:

```

- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"(SUBQUERY(employees,
$x, $x.age == %d and $x.name BEGINSWITH %@).count > 0)", 26, @"T"];
    [fetchRequest setPredicate:predicate];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

    for (NSManagedObject *person in persons)
    {
        NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
    }
}

```

This time, it yields the correct output.

```

name=Mary age=36

```

## Aggregating

When we think about aggregating data, we think about gathering a global statistic on a collection. This was the case in the previous example where you wanted to know how many items there were in the collection returned by the subquery. Collection operators are part of the Key-Value Coding paradigm. They can be embedded in a key path to signify that an aggregation operation should be executed. Figure 6–3 shows the general syntax for collection operators.

key path to collection (if any) .@ operator . key path to argument property

**Figure 6–3.** *The syntax of collection operators*

You currently can't define your own custom operators, but the most common are already available by default, as shown in Table 6–6.

**Table 6–6.** *The Comparison Predicate Modifiers*

| Operator | Description  |
|----------|--|
| avg      | Computes the average of the argument property in the collection.                     |
| count    | Computes the number of items in the collection (does not need an argument property). |
| min      | Computes the minimum value of the argument property in the collection.               |
| max      | Computes the maximum value of the argument property in the collection.               |
| sum      | Computes the sum value of the argument property in the collection.                   |

Let's play around with these operators. Go back to your test data; this time you want to return any managers whose direct reports' average age is 24 years old. The result from your query should contain only John, since Jane is 26 and Tim is 22. Using the avg operator, the predicate is surprisingly simple.

```
"employees.@avg.age = 24"
```

Replace the `readData:` method with the following implementation:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityWithName:@"Person"
    inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"employees.@avg.age ="]
```

```
%d", 24];
[fetchRequest setPredicate:predicate];

NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

for (NSManagedObject *person in persons)
{
    NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
}
}
```

Running this code returns John, as expected.

The other operators work the same way. To pull, for example, all managers whose youngest direct report is 22 years old, use the `min` operator and construct a predicate like this:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"employees.@min.age = %d" ←
, 22];
```

This predicate returns John, who has Tim (age 22) working for him, and Tim, who has Jill (age 22) working for him.

```
name=John age=32
name=Tim age=22
```

The one operator that behaves somewhat differently is the `count` operator, which doesn't take an argument property. To pull all managers, for example, that have exactly one direct report, use this predicate:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"employees.@count = %d", 1];
```

This pulls only Mary, who has only one employee, Tom, reporting to her, as the Console shows.

```
name=Mary age=36
```

## Sorting

Sorting is the operation of ordering the resulting data set based on some criteria. Say you want to find all the people whose managers' names contain the letter *M*, but you want to sort the result set by age.

## Returning Unsorted Data

Using the data set you just set up in the previous section, your query should return Mary's employee (Tom) and Tim's employees (Kate, Jim, and Jill). For such an example, the `readData:` method would look as follows:

```
- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityWithName:@"Person" ←
inManagedObjectContext:context];
```

```

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:entity];

NSPredicate *predicate = [NSPredicate predicateWithFormat:@"manager.name↵
CONTAINS[c] %@", @"m"];
[fetchRequest setPredicate:predicate];

NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

for (NSManagedObject *person in persons)
{
    NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
}
}

```

Replace the existing `readData:` method with the code listed earlier, and rerun the application. In Xcode's Console, you should see the following output (with each line preceded by date, time, and application information, deleted here), with the objects in no particular order:

```

name=Jill age=22
name=Jim age=40
name=Kate age=40
name=Tom age=26

```

## Sorting Data on One Criterion

In many cases, especially when dealing with user interfaces, sorting the data provides value, whether to make it more visually appealing or to make data easier to find. This is where `NSSortDescriptor` comes into play. Using this class, you can specify sort orders in a manner similar to what you would do in SQL using the `ORDER BY` keyword. If you wanted your result sorted by age, alter the `readData:` method to include a sort descriptor, like so:

```

- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"↵
inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"manager.name↵
CONTAINS[c] %@", @"m"];
    [fetchRequest setPredicate:predicate];

    NSSortDescriptor *sortDescriptorByAge = [[NSSortDescriptor alloc] initWithKey:@"age"↵
ascending:YES];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptorByAge, nil];
    [fetchRequest setSortDescriptors:sortDescriptors];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];

```

```

for (NSManagedObject *person in persons)
{
    NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
}
}

```

Again, replace the `readData:` method in `OrgChartAppDelegate.m` with the code listed earlier and run the application. This time, the output comes back sorted by age. Note that Jim and Kate have the same age, 40, so your output could have them reversed. Either way, your output should look similar to this:

```

name=Jill age=22
name=Tom age=26
name=Kate age=40
name=Jim age=40

```

Just like some expressions in a predicate, sort descriptors also rely on key paths to specify the property to use when sorting. The second attribute of the `initWithKey:ascending` method specifies whether the order should be ascending or descending. Changing this value has the effect of reversing the order.

## Sorting on Multiple Criteria

You can also add additional sorting criteria to obtain a multilevel sort. For example, you might want to sort the data by age, and if multiple ages are the same, use a complementary sort order such as alphabetically arranging the names. This would remove the arbitrary return order of Jim and Kate in the previous example. You may have noticed that the `setSortDescriptors:` expects an array of sort descriptors. Adding a secondary sort level is simply a matter of adding another sort descriptor to the array, like so:

```

- (void)readData
{
    NSManagedObjectContext *context = [self managedObjectContext];
    NSEntityDescription *entity = [NSEntityDescription entityWithName:@"Person"
    inManagedObjectContext:context];

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"manager.name
    CONTAINS[c] %@", @"m"];
    [fetchRequest setPredicate:predicate];

    NSSortDescriptor *sortDescriptorByAge = [[NSSortDescriptor alloc] initWithKey:@"age"
    ascending:YES];
    NSSortDescriptor *sortDescriptorByName = [[NSSortDescriptor alloc] initWithKey:
    @"name" ascending:YES];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptorByAge,
    sortDescriptorByName, nil];
    [fetchRequest setSortDescriptors:sortDescriptors];

    NSArray *persons = [context executeFetchRequest:fetchRequest error:nil];
}

```



```

for (NSManagedObject *person in persons)
{
    NSLog(@"name=%@ age=%@", [person valueForKey:@"name"], [person valueForKey:@"age"]);
}
}

```

Since Kate and Jim have the same age but Kate comes after Jim in alphabetical order, the output changes from an arbitrary order to a deterministic one.

```

name=Jill age=22
name=Tom age=26
name=Jim age=40
name=Kate age=40

```

You can verify that you control the order of Jim's and Kate's names by changing the `sortDescriptorByName` instance to sort descending by passing `NO` to the ascending parameter, like this:

```

NSSortDescriptor *sortDescriptorByName = [[NSSortDescriptor alloc] initWithKey:@"name"
ascending:NO];

```

The resulting output confirms that Kate now comes before Jim:

```

name=Jill age=22
name=Tom age=26
name=Kate age=40
name=Jim age=40

```

## Summary

Dumping an entire data set on someone overwhelms them and prevents them from finding what they're looking for. AutoTrader.com recognizes this and uses its ability to filter results as a marketing ploy to attract customers. You should recognize this, too, as you build Core Data applications. Use the information presented in this chapter to sort, filter, and aggregate results so that you can narrow your data into what users seek.

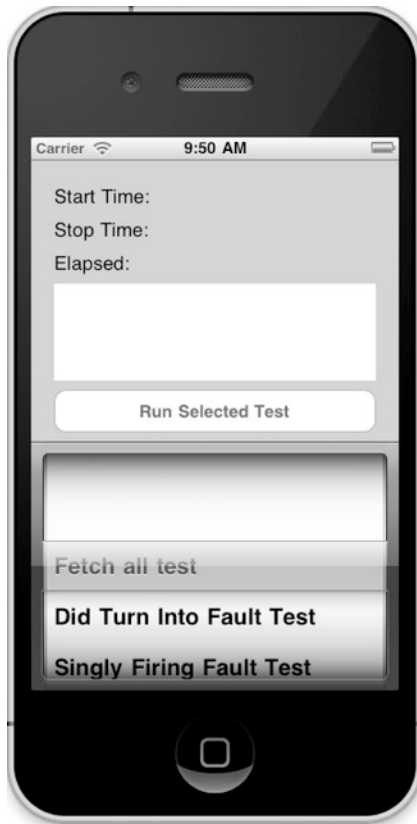
# Tuning Performance and Memory Usage

Lately, people have grown to hate hourglasses, spinning beach balls, and “Please Wait” pop-ups. Although giving a visual clue that the computer is working beats giving no such clue and letting customers think their computers have locked up, better still would be to never have episodes of slowness that make users wait. You may not always be able to achieve that goal, but you should always try.

Fetching and storing data in a persistent store can take a long time, especially when mounds of data are involved. This chapter assures that you understand how Core Data helps you—and how you must help yourself—to make sure you don’t pepper your users with “Please Wait” spinners or, worse, an unresponsive application that appears locked up while it retrieves or saves large object graphs. You will learn how to utilize the various tools and strategies, such as caching, faulting, and the Instruments application that comes with Xcode.

## Building the Application for Testing

You need a way to complete performance testing so that you can verify results. This section walks you through building an application that will allow you to run various tests and see the results. The application, shown in Figure 7–1, presents a list of tests you can run in a standard picker view. To run a test, select it in the picker, click the Run Selected Test button, and wait for the results. After the test runs, you’ll see the start time, the stop time, the number of elapsed seconds for the test, and some text describing the results of the test.



**Figure 7-1.** *The performance-tuning application*

## Creating the Core Data Project

Start by creating a new single view application with the Product option set to iPhone. Call it PerformanceTuning and add the Core Data framework. In your application delegate (PerformanceTuningAppDelegate), add a managed object context, a managed object model, and a persistent store coordinator, as well as a method for returning the application's document directory and a method for saving the managed object context. The following is the code for PerformanceTuningAppDelegate.h:

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

@class PerformanceTuningViewController;

@interface PerformanceTuningAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) IBOutlet UIWindow *window;
@property (strong, nonatomic) IBOutlet PerformanceTuningViewController *viewController;
@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
```

```
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator ←
*persistentStoreCoordinator;
```

```
- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;
```

```
@end
```

Listing 7–1 shows the code for `PerformanceTuningAppDelegate.m`.

**Listing 7–1. *PerformanceTuningAppDelegate.m***

```
#import "PerformanceTuningAppDelegate.h"
#import "PerformanceTuningViewController.h"

@implementation PerformanceTuningAppDelegate

@synthesize window=_window;
@synthesize viewController=_viewController;
@synthesize managedObjectContext=__managedObjectContext;
@synthesize managedObjectModel=__managedObjectModel;
@synthesize persistentStoreCoordinator=__persistentStoreCoordinator;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[PerformanceTuningViewController alloc]
initWithNibName:@"PerformanceTuningViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application {
}

- (void)applicationDidEnterBackground:(UIApplication *)application {
}

- (void)applicationWillEnterForeground:(UIApplication *)application {
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
}

- (void)applicationWillTerminate:(UIApplication *)application {
}

- (void)saveContext {
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil) {
        if ([managedObjectContext hasChanges] && ![managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        }
    }
}
```

```

        abort();
    }
}

#pragma mark - Core Data stack

/**
 Returns the managed object context for the application.
 If the context doesn't already exist, it is created and bound to the persistent store
 coordinator for the application.
 */
- (NSManagedObjectContext *)managedObjectContext {
    if (__managedObjectContext != nil) {
        return __managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        __managedObjectContext = [[NSManagedObjectContext alloc] init];
        [__managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return __managedObjectContext;
}

/**
 Returns the managed object model for the application.
 If the model doesn't already exist, it is created from the application's model.
 */
- (NSManagedObjectModel *)managedObjectModel {
    if (__managedObjectModel != nil) {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"PerformanceTuning"
        withExtension:@"momd"];
    __managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}

/**
 Returns the persistent store coordinator for the application.
 If the coordinator doesn't already exist, it is created and the application's store
 added to it.
 */
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (__persistentStoreCoordinator != nil) {
        return __persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"PerformanceTuning.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;
}

#pragma mark - Application's Documents directory

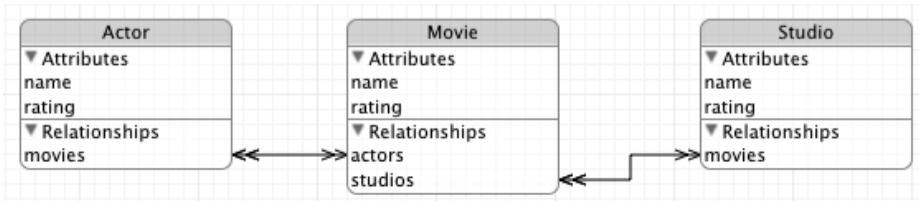
/**
 Returns the URL to the application's Documents directory.
 */
- (NSURL *)applicationDocumentsDirectory {
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];
}

@end

```

## Creating the Data Model and Data

The data model for this performance-testing application consists of three entities: Actor, Movie, and Studio. Actor and Movie have a many-to-many relationship, and Movie and Studio have a many-to-many relationship. Each entity has two attributes: name, of type String, and rating, of type Integer 16. Create your data model and call it `PerformanceTuning.xcdatamodeld`. Create entities for Actor, Movie, and Studio; give them each attributes called name of type String and rating of type Integer 16; and create optional to-many relationships between Actor and Movie and between Movie and Studio. Your completed data model should look like Figure 7-2.



**Figure 7-2.** The performance-tuning data model

Now that you've completed your data model, it's time to stuff data into it. You want enough data to be able to do performance testing and differentiate between results, so the plan is to create 200 actors, 200 movies, and 200 studios, and then to relate each actor to each movie and vice versa and relate each movie to each studio and vice versa. When you're done, you'll have three tables with 200 rows each (Actor, Movie, and Studio) and two join tables with 40,000 rows each (Actor-to-Movie and Movie-to-Studio).

To insert the data, open the `PerformanceTuningAppDelegate.h` file, and declare two methods: one that loads the data into the persistent data store and the other that acts as a helper method to create an entity. The helper method takes two strings: the name of the entity to create and the value for the name attribute. For the rating attribute, the helper will insert a random number between one and ten. The two method declarations should look like this:

```
- (void)loadData;
- (NSManagedObject *)insertObjectForName:(NSString *)entityName withName:(NSString *)name;
```

Now, open `PerformanceTuningAppDelegate.m`, and define those methods. The helper method creates the object within the specified entity type and then sets the value for the name attribute. It looks like this:

```
- (NSManagedObject *)insertObjectForName:(NSString *)entityName withName:(NSString *)name {
    NSManagedObjectContext *context = [self managedObjectContext];
    NSManagedObject *object = [NSEntityDescription
insertNewObjectForEntityForName:entityName inManagedObjectContext:context];
    [object setValue:name forKey:@"name"];
    [object setValue:[NSNumber numberWithInt:((arc4random() % 10) + 1)]
forKey:@"rating"];
    return object;
}
```

The method to load the data, `loadData:`, first checks the persistent store to determine whether the data has already been loaded so that subsequent runs of the program don't compound the data store. If the data has not been created, the `loadData:` method creates 200 actors with names like Actor 1, Actor 2, and so on; 200 movies with names like Movie 1; and 200 studios with names like Studio 1. After creating all the objects, the code loops through all the movies and adds relationships to all the actors and to all the studios. Finally, `loadData:` saves the object graph to the persistent data store. The following is the complete implementation of the `loadData` method:

```
- (void)loadData {
    // Pull the movies. If we have 200, assume our db is set up.
    NSManagedObjectContext *context = [self managedObjectContext];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
    NSArray *results = [context executeFetchRequest:request error:nil];
    if ([results count] != 200) {
        // Add 200 actors, movies, and studios
        for (int i = 1; i <= 200; i++) {
            [self insertObjectForName:@"Actor" withName:[NSString stringWithFormat:
@"Actor %d", i]];
            [self insertObjectForName:@"Movie" withName:[NSString stringWithFormat:
@"Movie %d", i]];
            [self insertObjectForName:@"Studio" withName:[NSString stringWithFormat:
@"Studio %d", i]];
        }
    }
}
```

```

// Relate all the actors and all the studios to all the movies
NSManagedObjectContext *context = [self managedObjectContext];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:[NSEntityDescription entityForName:@"Movie"↵
inManagedObjectContext:context]];
NSArray *results = [context executeFetchRequest:request error:nil];
for (NSManagedObject *movie in results) {
    [request setEntity:[NSEntityDescription entityForName:@"Actor"↵
inManagedObjectContext:context]];
    NSArray *actors = [context executeFetchRequest:request error:nil];
    NSMutableSet *set = [movie mutableSetValueForKey:@"actors"];
    [set addObjectsFromArray:actors];

    [request setEntity:[NSEntityDescription entityForName:@"Studio"↵
inManagedObjectContext:context]];
    NSArray *studios = [context executeFetchRequest:request error:nil];
    set = [movie mutableSetValueForKey:@"studios"];
    [set addObjectsFromArray:studios];
}
}
[self saveContext];
}

```

Go to the `application:didFinishLaunchingWithOptions:` method, and add a call to your `loadData:` method before it displays the window and view so it looks like this:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self loadData];
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[PerformanceTuningViewController alloc]
initWithNibName:@"PerformanceTuningViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Build and run the program to create your persistent data store.

## Creating the Testing View

The application you're building will present a picker with a list of tests you can run and a button to launch the selected test. It will list the start time, the stop time, and the elapsed time for the test, as well as a string describing the results of the test. Start creating this user interface by opening the `PerformanceTuningViewController.h` file, declaring that it implements the protocols for your picker view, declaring fields for your user interface elements, and declaring a method to be called when someone clicks the Run Selected Test button. The following code contains the contents of `PerformanceTuningViewController.h`:

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



```

@interface PerformanceTuningViewController : UIViewController <UIPickerViewDataSource,
UIPickerViewDelegate> {
    IBOutlet UILabel *startTime;
    IBOutlet UILabel *stopTime;
    IBOutlet UILabel *elapsedTime;
    IBOutlet UITextView *results;
    IBOutlet UIPickerView *testPicker;
}
@property (nonatomic, retain) UILabel *startTime;
@property (nonatomic, retain) UILabel *stopTime;
@property (nonatomic, retain) UILabel *elapsedTime;
@property (nonatomic, retain) UITextView *results;
@property (nonatomic, retain) UIPickerView *testPicker;

- (IBAction)runTest:(id)sender;

@end

```

In `PerformanceTuningViewController.m`, add `@synthesize` directives for your fields, stub implementations for the picker view protocols you promised to implement, and a stub implementation for `runTest`, like so:

```

@synthesize startTime, stopTime, elapsedTime, results, testPicker;

#pragma mark - UIPickerViewDataSource methods

- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component {
    return 1;
}

#pragma mark - UIPickerViewDelegate methods

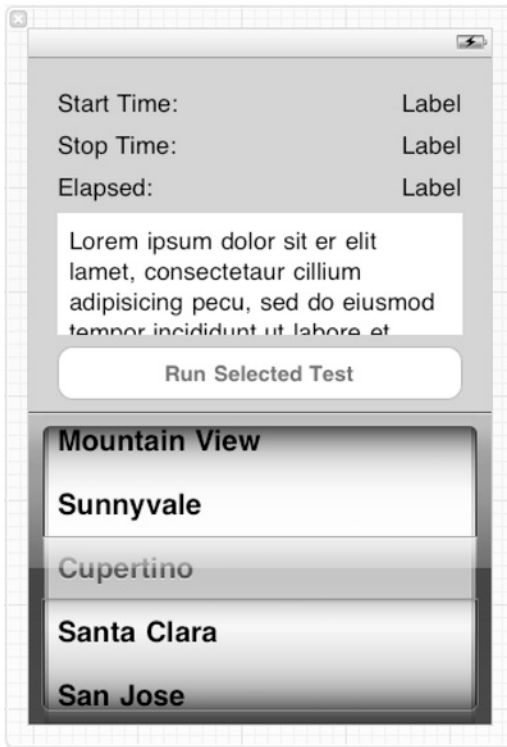
- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)row
forComponent:(NSInteger)component {
    return @"Test";
}

#pragma mark - Run the test

- (IBAction)runTest:(id)sender {

```

Open the `PerformanceTuningViewController.xib` file, and start building the view. You want labels for Start Time, Stop Time, and Elapsed, as well as right-aligned labels to store the actual start time, stop time, and elapsed time; a Text View to store the results of any test that runs; a button for launching a test; and a picker view to show the available tests. Drag all those items to the screen, and arrange them so they look like Figure 7-3.



**Figure 7–3.** Building the view for the performance-tuning application

The next step is to wire all the controls to the fields you created in the `PerformanceTuningViewController` class. Ctrl+drag from the File's Owner icon to each of the labels for start time, stop time, and elapsed time (the right-aligned ones for displaying the actual times), and pick the appropriate outlet for each. Ctrl+drag from File's Owner to the Text View, and select results. Ctrl+drag from File's Owner to the picker view, select `testPicker`, and then Ctrl+drag twice from the picker view back to File's Owner: the first time select `dataSource`, and the second time select `delegate`. You'll know if you forget to do this, because the picker view won't appear when you run your application. Finally, Ctrl+drag from the Run Selected Test button to File's Owner, and select `runTest:` from the pop-up. Save everything, build your application, and run it. You should see something that looks like Figure 7–4.



**Figure 7–4.** *The first run of the performance-tuning application*

## Building the Testing Framework

The approach for adding tests for the performance testing application to run is to create an Objective-C protocol called `PerformanceTest` and then create an array of tests that all implement that protocol. The application will display the names of the tests in the picker view and execute the selected test when the user taps the `Run Selected Test` button. The protocol, then, requires two methods.

- One to return the name of the test for display in the picker view.
- One to execute the selected test in the application's managed object context.

To keep the project organized, create a new group under the `PerformanceTuning` folder (Ctrl+click `PerformanceTuning` and select `New Group` from the pop-up menu), and call it `Tests`. The tests you add throughout this chapter will go in this group. To create the protocol, select **File** > **New** > **New File**, select **Cocoa Touch** on the left and **Objective-C protocol** on the right, and click `Next`. Call the protocol `PerformanceTest.h`, and put it in the `Tests` group. Open `PerformanceTest.h`, and modify it to match the following code:

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

@protocol PerformanceTest <NSObject>

- (NSString *)name;
- (NSString *)runWithContext:(NSManagedObjectContext *)context;

@end
```

Before integrating the testing framework into the application, create a test (a “test” test, if you will) so that you have something to both see in the picker view and to run. The first test you create will retrieve all the movies, actors, and studios from the persistent store. Create a new Objective-C class called `FetchAllMoviesActorsAndStudiosTest` as a subclass of `NSObject`, and add it to the Tests group you just created. The following code is for the header file:

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

@interface FetchAllMoviesActorsAndStudiosTest : NSObject <PerformanceTest>

@end
```

Here is the code for the implementation file. Modify your file to look like this:

```
#import <CoreData/CoreData.h>
#import "FetchAllMoviesActorsAndStudiosTest.h"

@implementation FetchAllMoviesActorsAndStudiosTest

- (NSString *)name {
    return @"Fetch all test";
}

- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
        inManagedObjectContext:context]];
    NSArray *results = [context executeFetchRequest:request error:nil];

    int actorsRead = 0, studiosRead = 0;
    for (NSManagedObject *movie in results) {
        actorsRead += [[movie valueForKey:@"actors"] count];
        studiosRead += [[movie valueForKey:@"studios"] count];
        [context refreshObject:movie mergeChanges:NO];
    }
    return [NSString stringWithFormat:@"Fetched %d actors and %d studios", actorsRead,
        studiosRead];
}

@end
```

## Adding the Testing Framework to the Application

Now you must modify the view controller to use the testing framework. Open `PerformanceTuningViewController.h` and add an instance of `NSArray` to hold the tests. The following code is the updated version of the file with the lines to add in bold. Save the file, and move on to `PerformanceTuningViewController.m`.

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

@interface PerformanceTuningViewController : UIViewController <UIPickerViewDataSource,
UIPickerViewDelegate> {
    IBOutlet UILabel *startTime;
    IBOutlet UILabel *stopTime;
    IBOutlet UILabel *elapsedTime;
    IBOutlet UITextView *results;
    IBOutlet UIPickerView *testPicker;
    NSArray *tests;
}
@property (nonatomic, retain) UILabel *startTime;
@property (nonatomic, retain) UILabel *stopTime;
@property (nonatomic, retain) UILabel *elapsedTime;
@property (nonatomic, retain) UITextView *results;
@property (nonatomic, retain) UIPickerView *testPicker;
@property (nonatomic, retain) NSArray *tests;

- (IBAction)runTest:(id)sender;

@end
```

In `PerformanceTuningViewController.m`, add the new `tests` member to your `@synthesize` line, which now looks like this:

```
@synthesize startTime, stopTime, elapsedTime, results, testPicker, tests;
```

Add the following import statements to support the changes you're going to make to allow your new test, and the subsequent tests you build in this chapter, to run:

```
#import "PerformanceTuningAppDelegate.h"
#import "PerformanceTest.h"
#import "FetchAllMoviesActorsAndStudiosTest.h"
```

Now, create a `viewDidLoad:` method that blanks out the fields that tests will fill in after they run and adds the new test to the `tests` array. Your implementation should match this:

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

    startTime.text = @"";
    stopTime.text = @"";
    elapsedTime.text = @"";
    results.text = @"";

    FetchAllMoviesActorsAndStudiosTest *famaasTest = [[FetchAllMoviesActorsAndStudiosTest
alloc] init];
```

```

    self.tests = [[NSArray alloc] initWithObjects:famaasTest, nil];
}

```

Since you still have only one component to show in the picker view, you can leave the `numberOfComponentsInPickerView:` method alone and continue to allow it to return 1. Your `pickerView:numberOfRowsInComponent:` method, however, should return the number of tests in your tests array.

```

- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component {
    return [self.tests count];
}

```

The `pickerView:titleForRow:forComponent:` method allows you to use your new `PerformanceTest` protocol. This method should retrieve the test for the corresponding row from the tests array and return the name for the test so the picker view displays this name. The method should now look like this:

```

- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)row
forComponent:(NSInteger)component {
    id <PerformanceTest> test = [self.tests objectAtIndex:row];
    return [test name];
}

```

Finally, you must instruct the `runTest:` method to do the following:

1. Get the managed object context from the application delegate.
2. Determine which test is selected.
3. Get the start time.
4. Run the selected test.
5. Get the stop time.
6. Update the fields with the times and results.

That implementation looks like this:

```

- (IBAction)runTest:(id)sender {
    PerformanceTuningAppDelegate *delegate = (PerformanceTuningAppDelegate *)
    [[UIApplication sharedApplication] delegate];
    NSManagedObjectContext *context = [delegate managedObjectContext];
    id <PerformanceTest> test = [self.tests objectAtIndex:[testPicker
    selectedRowInComponent:0]];

    NSDate *start = [NSDate date];
    results.text = [test runWithContext:context];
    NSDate *stop = [NSDate date];

    startTime.text = [start description];
    stopTime.text = [stop description];
    elapsedTime.text = [NSString stringWithFormat:@"%f seconds", [stop
    timeIntervalSinceDate:start]];
}

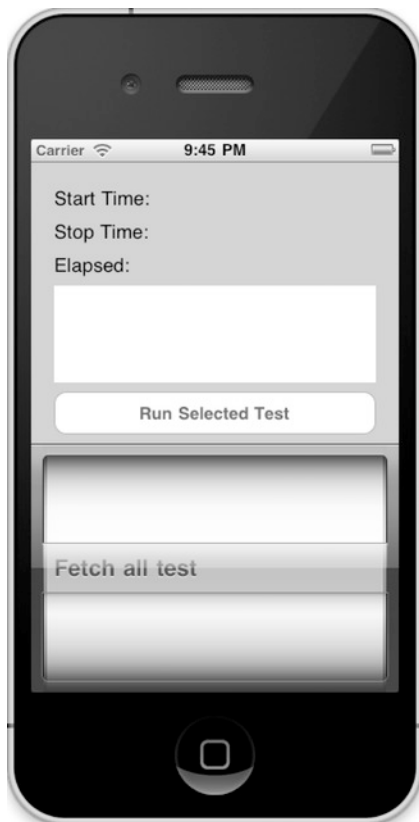
```

You have completed the performance-tuning application—for now. As you work through this chapter, you will add tests to cover the scenarios you read about.

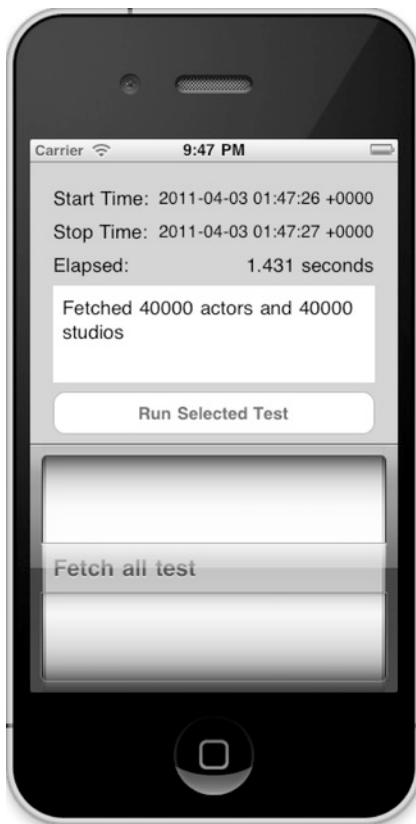
## Running Your First Test

Build and run the application. One important thing to note is that you're loading a lot of data into the managed object context and stressing the memory handling of iOS. You will likely see some instability in the application throughout the chapter if you run a few tests in a row. When the application crashes, simply restart it and carry on.

When you run the application, you should see the iPhone Simulator with the screen shown in Figure 7–5. The only test you've created so far, “Fetch all test,” stands alone in the picker view, selected and ready for you to run. Click the Run Selected Test button, and wait. Be patient. You didn't add any spinners, progress indicators, or other feedback mechanisms to the user interface while it runs, so all you'll see for a few moments is that the button turns blue and stays that way, but the test will eventually complete, and you'll see something like Figure 7–6. You can see in Figure 7–6 that the test took 1.431 seconds to run on our machine.



**Figure 7–5.** *The performance-tuning application with a single test available*



**Figure 7–6.** *The performance-tuning application after running a test*

The rest of this chapter explains the various performance considerations imposed by Core Data on iOS and the devices it runs on. As you learn these performance considerations, you'll add new tests to the application. Feel free to add your own tests as well to give the users of your applications the best possible experience.

## Faulting

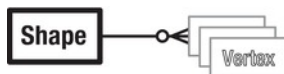
When you run SQL queries against a database, you know the depth and range of the data you're pulling. You know if you're joining tables, you know the columns you're pulling back, and, if you're careful, you can limit the number of rows you yank from the database into working memory. You may have to work harder to get data than you do with Core Data, but you have more control. You know about how much memory you're using to hold the data you fetch.

With Core Data, however, you've given up some amount of control in exchange for ease of use. With the Shapes application from Chapter 5, for example, each Polygon instance had a collection of vertices, which you accessed in an object-oriented, not a data-oriented, way. You didn't have to care whether Core Data pulled the vertex data from



the database when the application started, when the polygon was loaded, or when you first accessed the vertices. Core Data cares, however, and attempts to delay loading data from the persistent store into the object graph until necessary, reducing both fetch time and memory usage. It does this using what's termed *faults*.

Think of a fault like a symlink on a Unix file system, which is not the actual file it points to nor does it contain the file's data. The symlink represents the file, though, and when called upon can get to the file and its data just as if it were the file itself. Like a symlink, a fault is a placeholder that can get to the persistent store's data. A fault can represent either a managed object or, when it represents a to-many relationship, a collection of managed objects. As a link to or a shadow of the data, a fault occupies much less memory than the actual data does. See Figure 7-7 for a depiction of a managed object and a fault. In this case, the managed object is an instance of *Shape*, and the fault points to the related collection of *Vertex* objects. The *Shape* is boldly depicted, representing the fact that it has been fetched from the persistent store and resides in memory. The vertices, however, are faint and grayed out, because they haven't been fetched from the persistent store and do not reside in memory.



**Figure 7-7.** A shape and its faulted vertices

## Firing Faults

Core Data uses the term *firing faults* when it must pull the data from the persistent store that a fault points to and then put it into memory. Core Data has “fired” off a request to fetch data from the data store, and the fault has been “fired” from its job to represent the actual data. You cause a fault to fire any time you request a managed object's persistent data, whether through `valueForKey:` or through methods of a custom class that either return or access the object's persistent data. Methods that access a managed object's metadata and not any of the data stored in the persistent store don't fire a fault. This means you can query a managed object's class, hash, description, entity, and object ID, among other things, and not cause a fault to fire. For the complete list of which methods don't fire a fault, see Apple's documentation at <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/Articles/cdPerformance.html> under the section “Faulting Behavior.”

Explicitly asking a managed object for its persistent data causes a fault to fire, but so does calling any methods or constructs that access persistent data from a managed object. You can, for example, access a collection through a managed object's relationships without causing a fault to fire. Sorting the collection, however, will fire a fault because any logical sorting will sort by some of the object's persistent data. The test you build later in this section demonstrates this behavior.

## Faulting and Caching

We fibbed a bit when we said that firing faults fetches data from the persistent store. That's often true, but before Core Data treks all the way to the persistent store to retrieve data, it first checks its cache for the data it seeks. If Core Data finds its intended data in the cache, it pulls the data from cache and skips the longer trip to the persistent store. The “Caching” and “Expiring” sections discuss caching in more detail.

## Refaulting

One way to take control of your application's persistent data and memory use is to turn managed objects back into faults, thereby relinquishing the memory its data was occupying. You turn objects back into faults by calling the managed object context's `refreshObject:mergeChanges:` method, passing the object you want to fault and `NO` for the `mergeChanges:` parameter. If, for example, you had a managed object called `foo` that you wanted to turn back into a fault, you would use code similar to this:

```
[context refreshObject:foo mergeChanges:NO];
```

After this call, `foo` is now a fault, and `[foo isFault]` returns `YES`. If you look at the code for the previous test, you'll see a call within the loop that iterates over all movie instances to turn each movie instance back into a fault.

```
[context refreshObject:movie mergeChanges:NO];
```

Understanding the `mergeChanges:` parameter is important. Passing `NO`, as the previous code does, throws away any changed data that has not yet been saved to the persistent store. Take care when doing this because you lose all data changes in this object you're faulting, and all the objects to which the faulted object relates are released. If any of the relationships have changed and the context is then saved, your faulted object is out of sync with its relationships and you have created data integrity issues in your persistent store.

**NOTE:** Because faulting an object by calling `refreshObject:mergeChanges:NO` releases relationships, you can use this to prune your object graph in memory. Faulting a movie in the PerformanceTuning application, for example, would release its 200 related actors and 200 related studios.

Passing `YES` to `mergeChanges:` doesn't fault the object. Instead, it reloads all the object's persistent data from the persistent store (or the last cached state) and then reapplies any changes that existed before the call to `refreshObject:` that have not yet been saved.

When you turn a managed object into a fault, Core Data sends the following two messages to the managed object:

- `willTurnIntoFault:` before the object faults
- `didTurnIntoFault:` after the object faults

If you have implemented custom classes for your managed objects, you can implement either or both of these methods in your classes to perform some action on the object. Suppose, for example, that your custom managed object performs an expensive calculation on some persistent values and caches the result, and you want to nullify that cached result if the values it depends on aren't present. You could nullify the calculation in `didTurnIntoFault:`.

## Building the Faulting Test

To test what you've learned about faulting in this section, build a test that will do the following:

1. Retrieve the first movie from the persistent store.
2. Grab an actor from that movie.
3. Check whether the actor is a fault.
4. Get the name of the actor.
5. Check whether the actor is a fault.
6. Turn the actor back into a fault.
7. Check whether the actor is a fault.

To start, generate an `Actor` class from your data model that will represent the `Actor` entity. You can, if you want, create it manually. The following code is for the header file, `Actor.h`:

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

@interface Actor : NSManagedObject

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSNumber * rating;
@property (nonatomic, retain) NSSet* movies;

@end

@interface Actor (CoreDataGeneratedAccessors)

- (void)addMoviesObject:(NSManagedObject *)value;
- (void)removeMoviesObject:(NSManagedObject *)value;
- (void)addMovies:(NSSet *)value;
- (void)removeMovies:(NSSet *)value;

@end
```

The following code shows the implementation file, `Actor.m`. In the implementation file, implement the methods `willTurnIntoFault:` and `didTurnIntoFault:`, so that you can verify that Core Data indeed does call these methods when you turn the object back into

a fault. You don't do anything special in these methods but simply log that the events happened.

```
#import "Actor.h"

@implementation Actor
@dynamic name;
@dynamic rating;
@dynamic movies;

- (void)willTurnIntoFault {
    NSLog(@"Actor named %@ will turn into fault", self.name);
}

- (void)didTurnIntoFault {
    NSLog(@"Actor named %@ did turn into fault", self.name);
}

@end
```

Now, create a class called `DidTurnIntoFaultTest` that implements the `PerformanceTest` protocol. The following code is for `DidTurnIntoFaultTest.h`:

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

@interface DidTurnIntoFaultTest : NSObject <PerformanceTest>

@end
```

The following code is for `DidTurnIntoFaultTest.m`:

```
#import <CoreData/CoreData.h>
#import "DidTurnIntoFaultTest.h"
#import "Actor.h"

@implementation DidTurnIntoFaultTest

- (NSString *)name {
    return @"Did Turn Into Fault Test";
}

// Pull all the movies and verify that each of their actor objects are pointing to the
// same actors
- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSString *result = nil;

    // Fetch the first movie
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name = %@", @"Movie 1"];
    [request setPredicate:predicate];
    NSArray *results = [context executeFetchRequest:request error:nil];

    if ([results count] == 1) {
```

```

NSMutableDictionary *movie = (NSMutableDictionary *)[results objectAtIndex:0];
NSSet *actors = [movie valueForKey:@"actors"];
if ([actors count] != 200) {
    result = @"Failed to find the 200 actors for the first movie";
} else {
    // Get an actor
    Actor *actor = (Actor *)[actors anyObject];

    // Check if it's a fault
    result = [actor isFault] ? @"Actor is a fault.\n" : @"Actor is NOT a fault.\n";

    // Get its name and rating
    result = [result stringByAppendingFormat:@"Actor is named %@\n", actor.name];
    result = [result stringByAppendingFormat:@"Actor has rating %d\n", [actor.rating
integerValue]];

    // Check if it's a fault
    result = [result stringByAppendingString:[actor isFault] ? ~
@"Actor is a fault.\n" : @"Actor is NOT a fault.\n"];

    // Turn actor back into a fault
    result = [result stringByAppendingString:@"Turning actor back into a fault.\n"];
    [context refreshObject:actor mergeChanges:NO];

    // Check if it's a fault
    result = [result stringByAppendingString:[actor isFault] ? ~
@"Actor is a fault.\n" : @"Actor is NOT a fault.\n"];
}
} else {
    result = @"Failed to fetch the first movie";
}
return result;
}

@end

```

Turn your attention to the `runWithContext:` method, which fetches the first movie from the persistent store. It then grabs any one of the actors related to that movie and checks to see whether this actor is a fault. Note that, although you have done nothing to this point in the code to cause a fault to fire for the actor, the actor may not be a fault at this point, depending on what you've been doing in the application before this point. The code then accesses the actor's name and checks again whether the actor is a fault. It won't be. The access to the name fired a fault, if it hadn't already been fired elsewhere. The code then turns the actor into a fault and verifies that the actor, indeed, is a fault.

To add your new test to the application, open `PerformanceTuningViewController.m`, make sure you import `DidTurnIntoFaultTest.h`, find the line in `viewDidLoad:` that adds the tests to the tests array, and add the new test, like this:

```

FetchAllMoviesActorsAndStudiosTest *famaasTest = [[FetchAllMoviesActorsAndStudiosTest
alloc] init];
DidTurnIntoFaultTest *dtifTest = [[DidTurnIntoFaultTest alloc] init];
self.tests = [[NSArray alloc] initWithObjects:famaasTest, dtifTest, nil];

```

Build the application and run the test. You should see output similar to this:

```
Actor is a fault.
Actor is named Actor 42
Actor has rating 8
Actor is NOT a fault.
Turning actor back into a fault.
Actor is a fault.
```

Go back to `Actor.m` and comment out the `willTurnIntoFault:` and `didTurnIntoFault:` methods, as they and their calls to `NSLog()` can skew timing results for any other tests that use the `Actor` class.

## Taking Control: Firing Faults on Purpose

This section on faulting began by explaining that you had more control over memory usage when you ran SQL queries yourself than you do by allowing Core Data to manage data fetches. Although true, you can exert some amount of control over Core Data's fault management by firing faults yourself. By firing faults yourself, you can avoid inefficient scenarios in which Core Data must fire several small faults to fetch your data, incurring several trips to the persistent store.

Core Data provides two means for optimizing the firing of faults:

- Batch faulting
- Prefetching

As a control group, create a test called `SinglyFiringFaultTest`. This test should fetch all the movies, loop through them one by one, and do the following:

- Access the name attribute so that a fault fires for this movie only.
- Loop through all the related actors and access their name attributes, one at a time, so each access fires a fault.
- Reset each actor so the next movie will have to fire faults for each actor.
- Do the same for all the related studios.

The following code is for `SinglyFiringFaultTest.h`:

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

@interface SinglyFiringFaultTest : NSObject <PerformanceTest>

@end
```

The following code is for `SinglyFiringFaultTest.m`:

```
#import <CoreData/CoreData.h>
#import "SinglyFiringFaultTest.h"
```

```

@implementation SinglyFiringFaultTest

- (NSString *)name {
    return @"Singly Firing Fault Test";
}

- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSString *result = @"Singly Firing Fault Test Complete!";

    // Fetch all the movies
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
    NSArray *results = [context executeFetchRequest:request error:nil];

    // Loop through all the movies
    for (NSManagedObject *movie in results) {
        // Fire a fault just for this movie
        [movie valueForKey:@"name"];

        // Loop through all the actors for this movie
        for (NSManagedObject *actor in [movie valueForKey:@"actors"]) {
            // Fire a fault just for this actor
            [actor valueForKey:@"name"];

            // Put this actor back in fault so the next movie
            // will have to fire a fault
            [context refreshObject:actor mergeChanges:NO];
        }

        // Loop through all the studios for this movie
        for (NSManagedObject *studio in [movie valueForKey:@"studios"]) {
            // Fire a fault just for this studio
            [studio valueForKey:@"name"];

            // Put this studio back in fault so the next movie
            // will have to fire a fault
            [context refreshObject:studio mergeChanges:NO];
        }
    }
    return result;
}

@end

```

Create these files, and add an instance of `SinglyFiringFaultTest` to your tests array. Don't forget to import `SinglyFiringFaultTest.h`. Build the application, launch the simulator, and run the test. Running this test on our machine took about 2.2 seconds. By using prefetching, you should be able to improve on these results!

## Prefetching

Similar to batch faulting, prefetching minimizes the number of times that Core Data has to fire faults and go fetch data. With prefetching, though, you tell Core Data when you perform a fetch to also fetch the related objects you specify. For example, using this chapter's data model, when you fetch the movies, you can tell Core Data to prefetch the related actors, studios, or both.

To prefetch related objects, call `NSFetchRequest`'s `setRelationshipKeyPathsForPrefetching:` method, passing an array that contains the names of the relationships that you want Core Data to prefetch. To prefetch the related actors and studios when you fetch the movies, for example, use this code:

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];

[request setRelationshipKeyPathsForPrefetching:[NSArray arrayWithObjects:@"actors",
@"studios", nil]];

NSArray *results = [context executeFetchRequest:request error:nil];
```

The line in bold instructs Core Data to prefetch all the related actors and studios when it fetches the movies.

Before testing this, recognize that your baseline numbers depend on turning each actor and studio back into a fault after triggering the fault to load the data for a given movie. You did this because each movie relates to the same 200 actors and the same 200 studios, so the only time the actors and movies would normally be faults would be for the first movie, not the remaining 199. Turning each actor and each studio back into a fault for each movie allows you to measure firing faults for each actor and studio for each movie.

Turning each actor and each studio back into a fault negates the performance gains offered by prefetching them, however, so you're not going to refault them. To get good comparison numbers, then, you must change the code for `SinglyFiringFaultTest` to not reset the Actor and Studio objects. Open `SinglyFiringFaultTest.m`, go to the `runWithContext:` method, and comment these two lines:

```
[context refreshObject:actor mergeChanges:NO];
[context refreshObject:studio mergeChanges:NO];
```

Launch the app, and rerun the test. On our machine, it takes about 1.9 seconds.

Now, create a new class called `PreFetchFaultingTest`, and implement the `PerformanceTest` protocol. This test will look similar to `SinglyFiringFaultTest` with two important differences:

- The fetch prefetches the actors and studios.
- The actors and studios aren't turned back into faults.



The following code is the header file:

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

@interface PreFetchFaultingTest : NSObject <PerformanceTest>

@end
```

The following is the implementation file:

```
#import <CoreData/CoreData.h>
#import "PreFetchFaultingTest.h"

@implementation PreFetchFaultingTest

- (NSString *)name {
    return @"Pre-fetch Faulting Test";
}

- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSString *result = @"Pre-fetch Fault Test Complete!";

    // Fetch all the movies
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];

    // Pre-fetch the actors and studios
    [request setRelationshipKeyPathsForPrefetching:[NSArray arrayWithObjects:@"actors",
@"studios", nil]];
    NSArray *results = [context executeFetchRequest:request error:nil];

    // Loop through all the movies
    for (NSManagedObject *movie in results) {
        // Fire a fault just for this movie
        [movie valueForKey:@"name"];

        // Loop through all the actors for this movie
        for (NSManagedObject *actor in [movie valueForKey:@"actors"]) {
            // Get the name for this actor
            [actor valueForKey:@"name"];
        }

        // Loop through all the studios for this movie
        for (NSManagedObject *studio in [movie valueForKey:@"studios"]) {
            // Get the name for this studio
            [studio valueForKey:@"name"];
        }
    }
    return result;
}

@end
```

This line of code sets up the prefetching for actors and studios:

```
[request setRelationshipKeyPathsForPrefetching:[NSArray arrayWithObjects:@"actors",  
@"studios", nil]];
```

Add this test to the tests array in the viewDidLoad: method of PerformanceTuningViewController, as you did the other tests. Build and run this test. Our results were about 0.6 seconds, which is about 70 percent faster than the 1.9 seconds the same test took without prefetching.

## Caching

Regardless of target language or platform, most data persistence frameworks and libraries have an internal caching mechanism. Properly implemented caches provide opportunities for significant performance gains, especially for applications that need to retrieve the same data repeatedly. Core Data is no exception. The `NSManagedObjectContext` class serves as a built-in cache for the Core Data framework. When you retrieve an object from the backing persistent store, the context keeps a reference to it to track its changes. If you retrieve the object again, the context can give the caller the same object reference as it did in the first invocation.

The obvious trade-off that results from the use of caching is that, while improving performance, caching uses more memory. If no cache management scheme were in place to limit the memory usage of the cache, the cache could fill up with objects until the whole system collapses from lack of memory. To manage memory, the Core Data context has weak references to the managed objects it pulls out of the persistent store. This means that if the retain count of a managed object reaches zero because no other object has a reference to it, the managed object will be discarded. The exception to this rule is if the object has been modified in any way. In this case, the context keeps a strong reference (that is, sends a retain signal to the managed object) and keeps it until the context is either committed or rolled back, at which point it becomes a weak reference again.

**NOTE:** The default retain behavior can be changed by passing YES to the `setRetainsRegisteredObjects:` of `NSManagedObjectContext`. By passing YES, the context will retain all registered objects. The default behavior retains registered objects only when they are inserted, updated, deleted, or locked.

In this section, you will examine the difference between fetching objects from the persistent store or from the cache. You will build a test that does the following:

1. Resets the managed object context to flush the cache.
2. Retrieves all movies.
3. Retrieves all actors for each movie.
4. Displays the time it took to perform both retrievals.

5. Retrieves all movies (this time the objects will be cached).
6. Retrieves all actors for each movie.
7. Displays the time it took to perform both retrievals.

To start, create a class called `CacheTest`, and add it to the Tests group in Xcode. Make sure that `CacheTest` implements the `PerformanceTest` protocol. The following code is for `CacheTest.h`:

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

@interface CacheTest : NSObject <PerformanceTest>

@end
```

The following code is for `CacheTest.m`:

```
#import "CacheTest.h"
#import "Actor.h"

@implementation CacheTest

- (NSString *)name {
    return @"Cache Test";
}

- (void)loadDataFromContext:(NSManagedObjectContext *)context {
    // Fetch all the movies and all actors
    NSFetchedRequest *request = [[NSFetchedRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
    NSArray *results = [context executeFetchRequest:request error:nil];

    // Fetch all the actors
    for (NSManagedObject *movie in results) {
        NSSet *actors = [movie valueForKey:@"actors"];
        for (Actor *actor in actors) {
            [actor valueForKey:@"name"];
        }
    }
}

// Pull all the movies and verify that each of their actor objects are pointing to the
same actors
- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSMutableString *result = [NSMutableString string];

    [context reset]; // Clear all potentially cached objects

    NSDate *startTest1 = [NSDate date];
    [self loadDataFromContext:context];
    NSDate *endTest1 = [NSDate date];
```

```

NSTimeInterval test1 = [endTest1 timeIntervalSinceDate:startTest1];
[result appendFormat:@"Without cache: %.2f s\n", test1];

NSDate *startTest2 = [NSDate date];
[self loadDataFromContext:context];
NSDate *endTest2 = [NSDate date];

NSTimeInterval test2 = [endTest2 timeIntervalSinceDate:startTest2];
[result appendFormat:@"With cache: %.2f s\n", test2];

return result;
}

@end

```

Let's examine the `runWithContext:` method. It does the same thing twice: fetch all the movies and all the actors. The first time, the cache is explicitly cleared using `[context reset]`. The second time, all the objects will be in the cache, and therefore the data will come back much faster. The `loadDataFromContext:` method is invoked to retrieve all the movies and then pulls every actor for each movie. You explicitly get the name property of the actors in order to force Core Data to load the property, firing a fault if necessary.

Add an instance of `CacheTest` to the tests array, build the application, and run the Cache Test test. The output of the application will look similar to Figure 7–8, with the cached version running significantly faster than the version without a cache.



**Figure 7–8.** *Measuring the difference between loading cached and noncached objects*

If you reset the cache between the first and second tests using `[context reset]`, then the time to execute the second test would be about the same as the first test. This is because the objects would all have to be retrieved from the persistent store again.

## Expiring

Any time an application uses a cache, the question of cache expiration arises: when should objects in the cache expire and be reloaded from the persistent store? The difficulty in determining the expiration interval comes from juggling the performance gain obtained by caching objects for long intervals versus the extra memory consumption that entails and the potential staleness of the cached data. This section examines the trade-offs of the two possibilities for expiring the cache and freeing some memory.

## Memory Consumption

As more and more objects are put into the cache, the memory usage increases. Even if a managed object is entirely faulted, a quick call to `NSLog(@"%d", class_getInstanceSize(NSManagedObject.class))` will show that the allocated size of an unpopulated managed object is 48 bytes. This is because it holds references (that is, 4-byte pointers) to other objects such as the entity description, the context, and the object ID. Even without any actual data, a managed object occupies a minimum of 48 bytes. This is a best-case scenario because this approximation does not include the memory occupied by the unique object ID, which is populated. This means that if you have 100,000 managed objects in the cache, even faulted, you are using at least 5MB of memory for things other than your data. If you start fetching data without faulting, you can run into memory issues quickly.

The trick to this balancing act is to remove data from the cache when it's no longer needed or if you can afford to pay the price of retrieving the objects from the persistent store when you need them again.

## Brute-Force Cache Expiration

If you don't care about losing all the managed objects, you can reset the context entirely. This is rarely the option you want to choose, but it is extremely efficient. `NSManagedObjectContext` has a `reset` method that will wipe the cache out in one swoop. Once you call `[managedObjectContext reset]`, your memory footprint will be dramatically smaller, but you will have to pay the price of going to the persistent store if you want to retrieve any objects again. Please also understand that, as with any other kind of mass destruction mechanism, resetting the cache in the middle of a running application has serious side effects and collateral damage. For example, any managed object that you were using prior to the reset is now invalid. If you try to do anything with them, your efforts will be met with runtime errors.

## Expiring the Cache Through Faulting

Faulting is a more subtle option, as the section on faulting explains. You can fault any managed object by calling `[context refreshObject:managedObject mergeChanges:NO]`. After this method call, the object is faulted, and therefore the memory it occupies in the cache is minimized, although not zero. A non-negligible advantage of this strategy, however, is that when the managed object is turned into a fault, any managed object it has a reference to (through relationships) is released. If those related managed objects have no other references to them, then they will be removed from the cache, further reducing the memory footprint. Faulting managed objects in this manner helps prune the entire object graph.

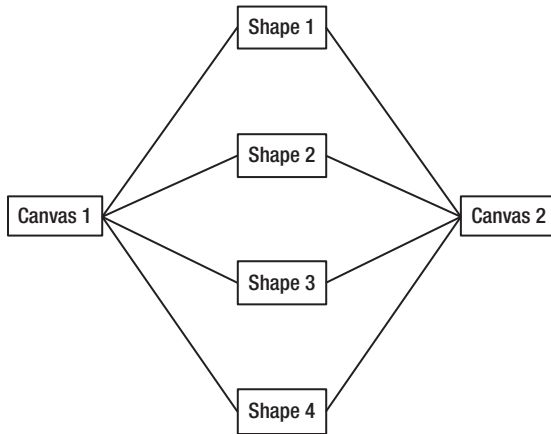
## Uniquing

Both business and technology like to turn nouns into verbs, and the pseudoword *uniquing* testifies to this weakness. It attempts to define the action of making something unique or ensuring uniqueness. Usage suggests not only that Apple didn't invent the term but also that it predates Core Data. Apple embraces the term in its Core Data documentation, however, raising fears that one day Apple will call the action of listening to music *iPodding*.

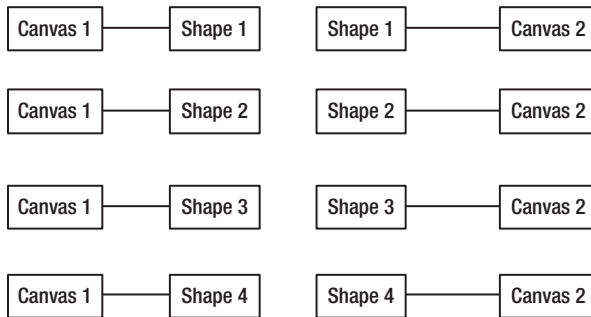
The technology industry uses the term *uniquing* in conjunction with memory objects versus their representation in a data store. In “Core Java Data Objects” (Prentice Hall, 2003), it says that uniquing “ensures that no matter how many times a persistent object is found, it has only one in-memory representation. All references to the same persistent object within the scope of the same `PersistenceManager` instance reference the same in-memory object.”

Martin Fowler, in “Patterns of Enterprise Application Architecture” (Addison-Wesley Professional, 2005), gives it a less colorful, more descriptive, and more English-compliant name: *identity map*. He explains that an identity map “ensures that each object gets loaded only once by keeping every loaded object in a map.” Whatever you call it or however you describe it, *uniquing* means that Core Data conserves memory use by ensuring the uniqueness of each object in memory and that no two memory instances of an object point to the same instance in the persistent store.

Consider, for example, the Shapes application from Chapter 5. Each Shape instance has a relationship with two Canvas instances. When you run the Shapes application and reference the two Canvas instances through any Shape instances, you always get the same two Canvas instances, as Figure 7–9 depicts. If Core Data didn't use uniquing, you could find yourself in the scenario shown in Figure 7–10, where each Canvas instance is represented in memory several times, once for each Shape, and each Shape instance is represented in memory several times, once for each Canvas.



**Figure 7-9.** *Uniquing shapes and canvases*



**Figure 7-10.** *Nonuniquing shapes and canvases*

Uniquing not only conserves memory but also eliminates data inconsistency issues. Think what could happen, for example, if Core Data didn't employ uniquing and the Shapes application had two memory instances of each shape (one for each canvas). Suppose that Canvas 1 changed the color of a shape to mauve and Canvas 2 changed the color of the shape to lime green. What color should the shape display? When the application stores the shape in the persistent store, what color should it save? You can imagine the data inconsistency bugs you'd have to track down if Core Data maintained more than one instance of each data object in memory.

Note that uniquing occurs within a single managed object context only, not across managed object contexts. The good news, however, is that Core Data's default behavior, which you can't change, is to unique. Uniquing comes free with Core Data.

To test uniquing, you'll create a test that fetches all the movies from the persistent store and then compares each of their related actors to each other movie's related actors. For the test to pass, the code must verify that only 200 Actor instances live in memory and that each movie points to the same 200 actors. To begin, create a new class called `UniquingTest`, which should implement the `PerformanceTest` protocol, like so:

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

```
@interface UniquingTest : NSObject <PerformanceTest>
```

```
@end
```

The implementation file, `UniquingTest.m`, fetches all the movies and iterates through them, pulling all the actors that relate to each one. The code sorts the actors so they're in a determined order so you can predictably compare instances. The first time through the loop (for the first movie), the code stores each of the actors in a reference array so that all subsequent loops have something to compare the actors against. Each subsequent loop, the code pulls the actors for the movie and compares them to the reference array. If just one actor doesn't match, the test fails. The code looks like this:

```
#import <CoreData/CoreData.h>
#import "UniquingTest.h"
```

```
@implementation UniquingTest
```

```
- (NSString *)name {
    return @"Uniquing test";
}
```

```
// Pull all the movies and verify that each of their actor objects are pointing to the
same actors
```

```
- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSString *result = @"Uniquing test passed";
```

```
    // Array to hold the actors for comparison purposes
    NSMutableArray *referenceActors = nil;
```

```
    // Sorting for the actors
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"name"↵
ascending:YES];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
```

```
    // Fetch all the movies
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"↵
inManagedObjectContext:context]];
    NSArray *results = [context executeFetchRequest:request error:nil];
```

```
    // Go through each movie
    for (NSManagedObject *movie in results) {
        // Get the actors
        NSSet *actors = [movie mutableSetValueForKey:@"actors"];
        NSArray *sortedActors = [actors sortedArrayUsingDescriptors:sortDescriptors];
```

```
        if (referenceActors == nil) {
            // First time through; store the references
            referenceActors = [[NSMutableArray alloc] initWithArray:sortedActors];
        } else {
            for (int i = 0, n = [sortedActors count]; i < n; i++) {
```



```

        if ([sortedActors objectAtIndex:i] != [referenceActors objectAtIndex:i]) {
            result = [NSString stringWithFormat:@"Uniquing test failed; %@ != %@",
        [sortedActors objectAtIndex:i], [referenceActors objectAtIndex:i]];
            break;
        }
    }
}
return result;
}

@end

```

To run this test, open `PerformanceTuningViewController.m`, and add an instance of `UniquingTest` to the tests array and an import for `UniquingTest.h`. Running the test should show that each actor exists only once in memory and that Core Data has used uniquing to reduce your application's memory footprint (see Figure 7–11).



**Figure 7–11.** *The results of the uniquing test*

## Improve Performance with Better Predicates

Despite the year in the movie title *2001: A Space Odyssey*, we are still far away from our machines having the intelligence to respond with things like “I’m sorry, Dave. I’m afraid I can’t do that.” As of the time of writing this book, programmers still have to do a lot of hand-holding to walk their machines through the process of doing what they are asked to do. This means that how you write your code determines how efficient it will be. There are often multiple ways to retrieve the same data, but the solution you use can significantly alter the performance of your application.

### Using Faster Comparators

Generally speaking, string comparators perform slower than primitive comparators. When predicates are compounded using an OR operator, it is always more efficient to put the primitive comparators first because if they resolve to TRUE, then the rest of the comparators don’t need to be evaluated. This is because “TRUE OR anything” is always true. A similar strategy can be used with AND-compounded predicates. In this case, if the first predicate fails, then the second will not be evaluated, because “FALSE AND anything” is always false.

To validate this, add a new test to your performance test application. You call the new class `PredicatePerformanceTest`. The following code is the header file:

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

@interface PredicatePerformanceTest : NSObject <PerformanceTest>

@end
```

This test consists of two fetch requests, which retrieve the same objects using an OR-compounded predicate. In the first test, the string comparator `LIKE` is used before the primitive comparator. In the second test, the comparators are permuted. Run each test 1,000 times in order to get significant timings in seconds; each time through the loop, the cache is reset to keep a clean context. The following code shows the implementation:

```
#import <CoreData/CoreData.h>
#import "PredicatePerformanceTest.h"

@implementation PredicatePerformanceTest

- (NSString *)name {
    return @"Predicate Performance Test";
}

- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSMutableString *result = [NSMutableString string];
    NSFetchRequest *request;
    NSDate *startTest1 = [NSDate date];
```

```

    for (int i=0; i<1000; i++) {
        [context reset];
        request = [[NSFetchRequest alloc] init];
        [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
        [request setPredicate:[NSPredicate predicateWithFormat:@"(name LIKE %@
OR (rating < %d)", @"*c*or*", 5)];
        [context executeFetchRequest:request error:nil];
    }
    NSDate *endTest1 = [NSDate date];

    NSTimeInterval test1 = [endTest1 timeIntervalSinceDate:startTest1];
    [result appendFormat:@"Slow predicate: %.2f s\n", test1];

    NSDate *startTest2 = [NSDate date];
    for (int i=0; i<1000; i++) {
        [context reset];
        request = [[NSFetchRequest alloc] init];
        [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
        [request setPredicate:[NSPredicate predicateWithFormat:@"(rating < %d) OR (name
like %@)", 5, @"*c*or*"]];
        [context executeFetchRequest:request error:nil];
    }
    NSDate *endTest2 = [NSDate date];

    NSTimeInterval test2 = [endTest2 timeIntervalSinceDate:startTest2];
    [result appendFormat:@"Fast predicate: %.2f s\n", test2];

    return result;
}

@end

```

To see the test in the user interface, open `PerformanceTuningViewController.m`, and add an instance of `PredicatePerformanceTest` to the tests array and an import for `PredicatePerformanceTest.h`.

Running the test will consistently show a significant timing difference between the two tests. Of course, the timings will differ depending on the performance of your machine, but here's an example run, with the fast predicate outperforming the slow predicate by over 25%:

```

Slow predicate: 1.21s
Fast predicate: 0.89s

```

## Using Subqueries

You saw in the previous chapter how to use subqueries to help simplify the code. In this section, you add a test to show the difference between using subqueries and retrieving related data manually. Consider an example in which you want to find all actors from a movie that match certain criteria. To do this without using a subquery, you have to first

fetch all the movies that match the criteria. You then have to iterate through each movie and extract the actors. You then go through the actors and add them to your result set, making sure you don't duplicate actors if they play in multiple movies. The manually subquerying test is shown here:

```
NSMutableDictionary *actorsMap = [NSMutableDictionary dictionary];
request = [[NSFetchRequest alloc] init];
[request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
[request setPredicate:[NSPredicate predicateWithFormat:@"(rating < %d) OR (name LIKE
%@", 5, @"*c*or*")]];
NSArray *movies = [context executeFetchRequest:request error:nil];

for (NSManagedObject *movie in movies) {
    NSSet *actorSet = [movie valueForKey:@"actors"];
    for (NSManagedObject *actor in actorSet) {
        [actorsMap setValue:actor forKey:[[[actor objectID] URIRepresentation]
description]];
    }
}
```

In this implementation, the actorsMap dictionary contains all the actors. You keyed them by objectID in order to eliminate duplicates. The alternative to doing this is to use subqueries (please refer to Chapter 6 for more information on building subqueries). In this case, the subquery looks like this:

```
request = [[NSFetchRequest alloc] init];
[request setEntity:[NSEntityDescription entityForName:@"Actor"
inManagedObjectContext:context]];
[request setPredicate:[NSPredicate predicateWithFormat:@"(SUBQUERY(movies, $x,
($x.rating < %d) OR ($x.name LIKE %@)).@count > 0)", 5, @"*c*or*")]];

NSArray *actors = [context executeFetchRequest:request error:nil];
```

One of the major differences here is that you let the persistent store do all the work of retrieving the matching actors, which means that most of the results don't have to make it back up to the context layer of Core Data for you to post-process. With the subquery, you don't actually retrieve the movies, and the fetched request is set up to fetch actors directly. The manual option, on the other hand, has to fire faults to retrieve the actors after retrieving the movies.

To demonstrate this in a test, create a new class in your project called SubqueryTest. The header file is shown here:

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

@interface SubqueryTest : NSObject <PerformanceTest>

@end
```

The tests are implemented in `SubqueryTest.m`, like so:

```
#import <CoreData/CoreData.h>
#import "SubqueryTest.h"

@implementation SubqueryTest

- (NSString *)name {
    return @"Subquery Performance Test";
}

- (NSString *)runWithContext:(NSManagedObjectContext *)context {
    NSMutableString *result = [NSMutableString string];

    NSFetchRequest *request;

    int count = 0;

    [context reset];
    NSDate *startTest1 = [NSDate date];
    NSMutableDictionary *actorsMap = [NSMutableDictionary dictionary];
    request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Movie"
inManagedObjectContext:context]];
    [request setPredicate:[NSPredicate predicateWithFormat:@"(rating < %d) OR (name LIKE
%@", 5, @"*c*or*)"]];
    NSArray *movies = [context executeFetchRequest:request error:nil];

    for (NSManagedObject *movie in movies) {
        NSSet *actorSet = [movie valueForKey:@"actors"];
        for (NSManagedObject *actor in actorSet) {
            [actorsMap setValue:actor forKey:[[[actor objectID] URIRepresentation]
description]];
        }
    }

    count = [actorsMap count];

    NSDate *endTest1 = [NSDate date];

    NSTimeInterval test1 = [endTest1 timeIntervalSinceDate:startTest1];
    [result appendFormat:@"No subquery: %.2f s\n", test1];
    [result appendFormat:@"Actors retrieved: %d\n", count];

    [context reset];
    NSDate *startTest2 = [NSDate date];
    request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Actor"
inManagedObjectContext:context]];
    [request setPredicate:[NSPredicate predicateWithFormat:@"(SUBQUERY(movies, $x,
($x.rating < %d) OR ($x.name LIKE %@)).@count > 0)", 5, @"*c*or*)"]];

    NSArray *actors = [context executeFetchRequest:request error:nil];
    count = [actors count];
```

```

NSDate *endTest2 = [NSDate date];

NSTimeInterval test2 = [endTest2 timeIntervalSinceDate:startTest2];
[result appendFormat:@"Subquery: %.2f s\n", test2];
[result appendFormat:@"Actors retrieved: %d\n", count];

return result;
}

@end

```

Open `PerformanceTuningViewController.m`, and add an instance of `SubqueryTest` to the `tests` array in order to be able to run the test. The test results—over 80% faster—are unequivocal.

```

No subquery: 0.74 s
Actors retrieved: 200
Subquery: 0.13 s
Actors retrieved: 200

```

How you write your predicates can have a profound effect on the performance of your application. You should always be mindful of what you are asking Core Data to do and how you can accomplish the same results with more efficient predicates.

## Analyzing Performance

Although thinking things through before writing code and understanding the implications of things such as faulting, prefetching, and memory usage usually nets you solid code that performs well, you often need a nonbiased, objective opinion on how your application is performing. The least biased and most objective opinion on your application's performance comes from the computer it's running on, so asking your computer to measure the results of your application's Core Data interaction provides essential insight for optimizing performance.

Apple provides a tool called Instruments that allows you to measure several facets of an application, including Core Data–related items. This section shows how to use Instruments to measure the Core Data aspects of your application. We encourage you to explore the other measurements Instruments offers.

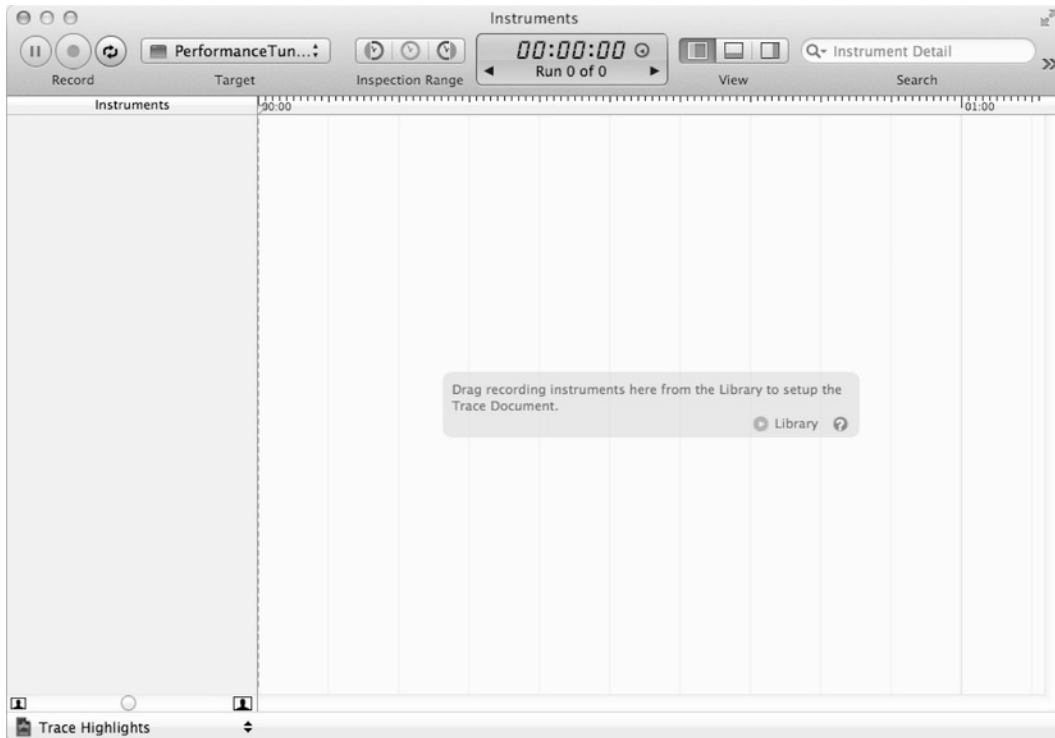
## Launching Instruments

Although you can launch Instruments as you would any other application, Xcode provides a simpler way to launch it to run your application. From the Xcode menu, select **Product ► Profile**. This launches Instruments and displays a dialog asking you what you want to profile, as shown in Figure 7–12.



**Figure 7-12.** Instruments asking you to choose a template

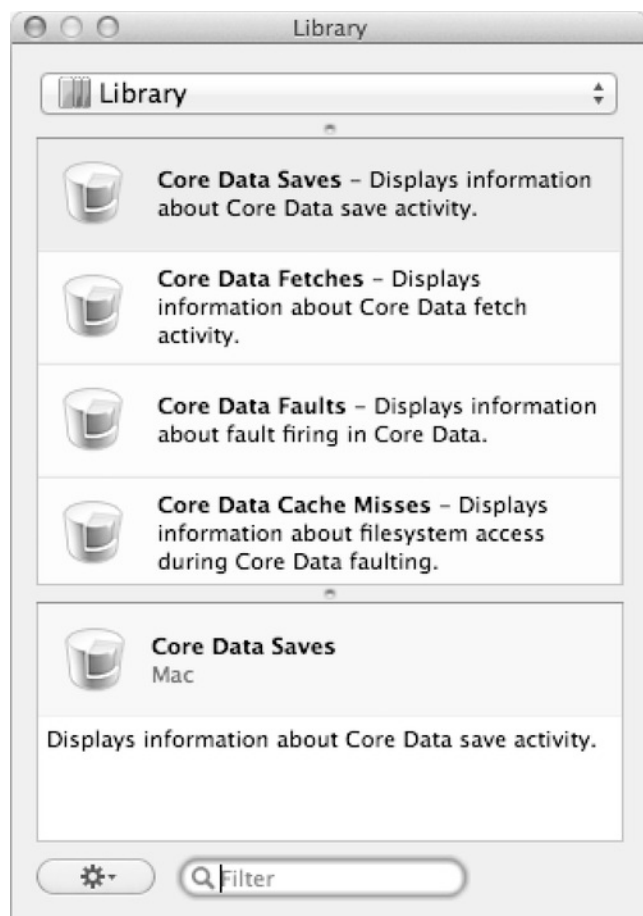
Since Instruments offers no Core Data template for iOS, select the Blank template and click the Profile button. That opens the main Instruments window with a message asking you to drag recording instruments here from the Library, as Figure 7-13 shows.



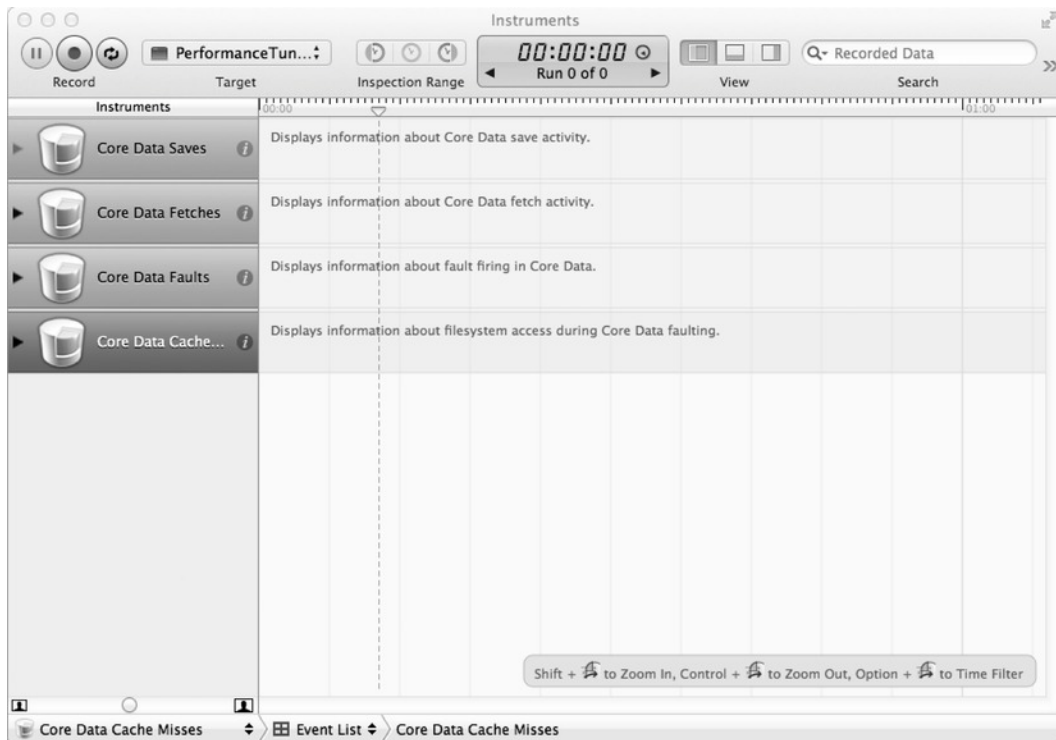
**Figure 7-13.** *The main Instruments window*

Click the arrow beside Library to display the library. You'll see four Core Data-related instruments among the palette of recording instruments, as shown in Figure 7-14. Drag all four of these to the left sidebar of the Instruments window so that it resembles Figure 7-15.





**Figure 7-14.** *The Core Data recording instruments in the instrument library*



**Figure 7–15.** The main Instruments window with the Core Data recording instruments

Now that you’ve configured Instruments to record Core Data interactions, click the Record button at the top left of the Instruments window to launch the iOS Simulator and the PerformanceTuning application.

## Understanding the Results

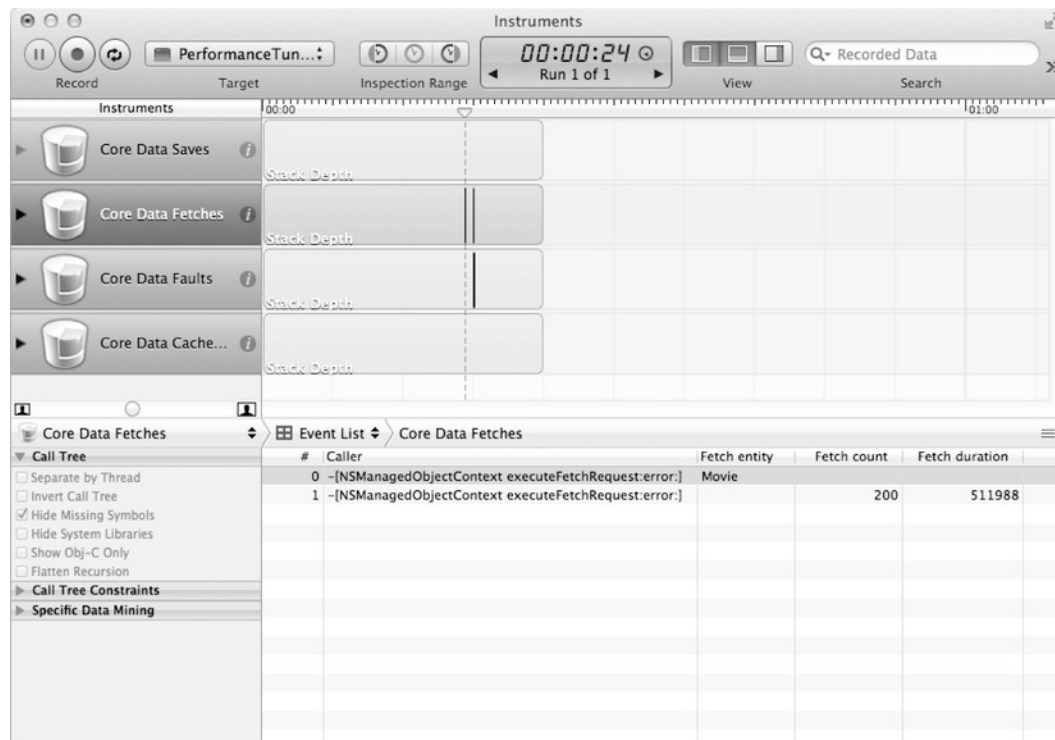
The Instruments window shows the Core Data measurements it’s tracking, which include the following:

- Core Data saves
- Core Data fetches
- Core Data faults
- Core Data cache misses

Run any of the tests in the PerformanceTuning application—say, the Pre-fetch Faulting Test—and wait for the test to complete. When the test finishes, click the Stop button in the upper left of Instruments to stop the application and stop recording Core Data measurements. You can then review the results from your test to see information about saves, fetches, faults, and cache misses. You can save the results to the file system for

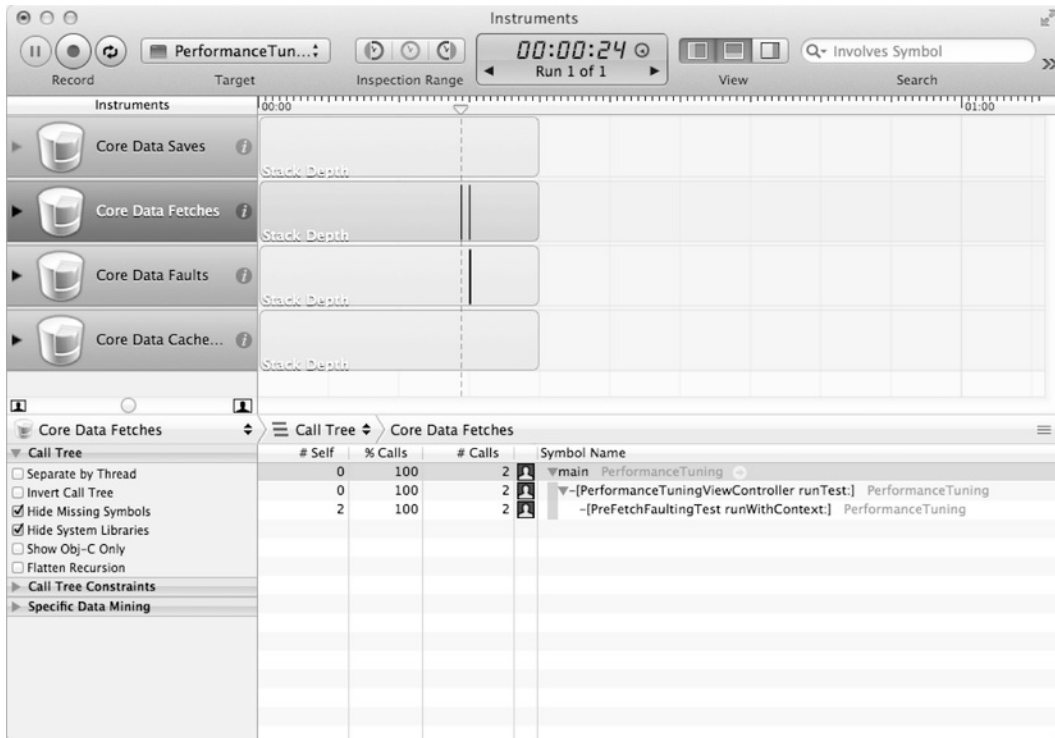
further review by selecting **File ► Save As** from the menu. You reopen them in Instruments using the standard **File ► Open** menu item.

Figure 7-16 shows the Instruments window after running the Pre-fetch Faulting Test. You can see the fetch counts and fetch durations, which are in microseconds, for the Core Data fetches (you may have to jiggle the headers a bit to get everything to display). The code ran one fetch request against the *Movie* entity (Instruments lists the call for a fetch twice: once when the call starts and once when it finishes). The request, which took 450,700 microseconds, fetched 200 movies, which you can tell from the fetch entity, *Movie*, and the fetch count, 200.



**Figure 7-16.** Results from the Pre-fetch Faulting Test

You can see the call tree for the fetch request by changing the drop-down in the middle of the window from Event List to Call Tree. You can reduce the navigation depth required to see the calls in your application's code by checking the box next to Hide System Libraries on the left of the window. Figure 7-17 shows the call tree for the fetch request. Using the call tree, you can determine which parts of your code are fetching data from your Core Data store.



**Figure 7-17.** The call tree for the fetch request

This is just a taste of what Instruments can do for you to help you determine how your application is using Core Data and how it can lead you to places where Core Data performance is slow enough to warrant optimization efforts.

## Summary

From uniquing to faulting to caching managed objects, Core Data performs a significant amount of data access performance optimization for you. These optimizations come free, without any extra effort on your part. You should be aware of the optimizations that Core Data provides, however, so that you make sure to work with, not against, them.

Not all Core Data performance gains come automatically, however. In this chapter, you learned how to use techniques such as prefetching and predicate optimization to squeeze all the performance from Core Data that you can for your applications. You also learned how to analyze your Core Data application using the Instruments application, so you can understand how your application is using Core Data and where the trouble spots are.

Other iOS programming books might do an excellent job showing you how to display a spinner and a “Please Wait” message when running long queries against your persistent store. This book shows you how to avoid the need for the spinner and “Please Wait” message entirely.

## Versioning and Migrating Data

As you develop Core Data–based applications, you usually don’t get your data model exactly right the first time. You start by creating a data model that seems to meet your application’s data needs, but as you progress through the development of the application, you’ll often find that your data model needs to change to serve your growing vision of what your application should do. During this stage of your application’s life, changing your data model to match your new understanding of the application’s data poses little cost: your application will no longer launch, crashing on startup with the message:

The model used to open the store is incompatible with the one used to create the store.

You resolve this issue either by finding the database file on the file system and deleting it or by deleting your fledgling application from the iPhone Simulator or your device. Either way, the database file that uses your outdated schema disappears, along with data in the persistent store, and your application re-creates the database file the next time it launches. You’ll probably do this several times during the development of your application.

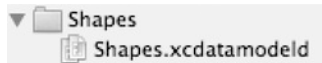
Once you release your application, however, and people start using it, they will accumulate data they deem important in the persistent stores on their devices. Asking them to delete their data stores any time you want to release a new version of the application with a changed data model will drop your app instantly to one-star status, and the people commenting will decry Apple’s rating system for not allowing ratings of zero or even negative stars.

Does that mean releasing your application freezes its data model, that the data model in the 1.0 version of your application is permanent? That you’d better get the first public version of the data model perfect, because you’ll never be able to change it? Thankfully, no. Apple anticipated the need for improving Core Data models over the life of applications and built mechanisms for you to change data models and then migrate users’ data to the new models, all without their intervention or even awareness. This

chapter goes through the process of versioning your data models and migrating data across those versions, however complex the changes you've made to the model.

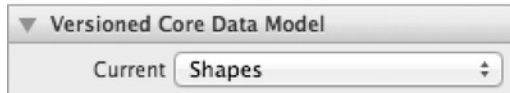
## Versioning

To take advantage of Core Data's support for versioning your data model and migrating data from version to version, you start by explicitly creating a new version of the data model. To illustrate how this works, let's bring back the Shapes application from Chapter 5. Make a copy of it, because you'll be changing its data model several times in this chapter. In the original version of Shapes, you created a data model, which makes the model appear as shown in Figure 8–1.

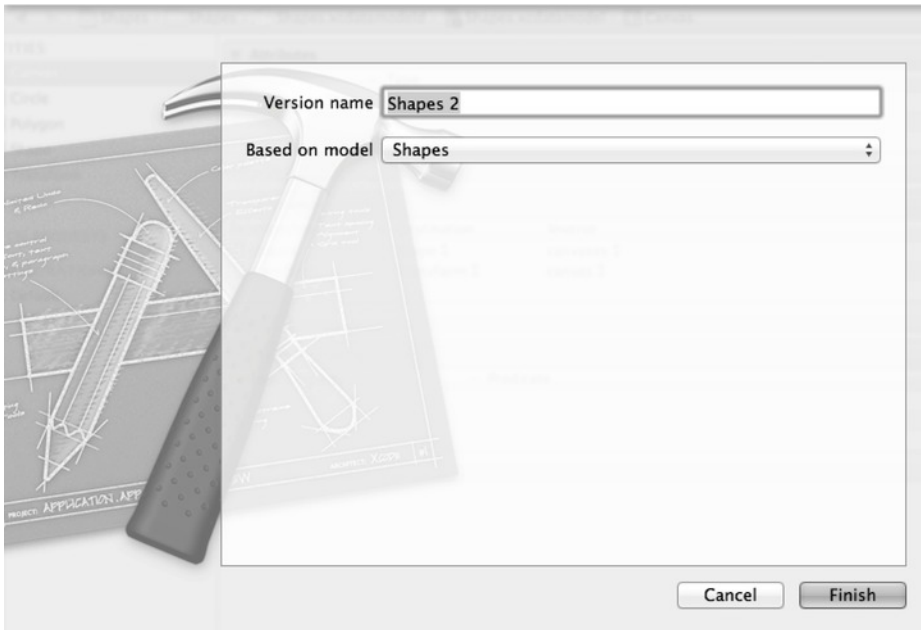


**Figure 8–1.** *The single-version data model*

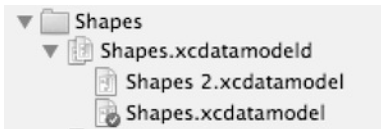
Shapes.xcdatamodeld contains the model and is actually a directory on your file system. It contains the files necessary to create the Core Data storage when your application runs. Select Shapes.xcdatamodeld and take a look at the File inspector in the Utilities view in Xcode, as shown in Figure 8–2. It shows the name of the current model version, Shapes. Since there is only one version of the object model, the current version is automatically set. To add a new version, from the Xcode menu, select **Editor > Add Model Version**. A panel displays that allows you to enter a version name and select the model that the new version is based on, as shown in Figure 8–3. Accept the defaults and select Finish. Xcode creates a new directory inside Shapes.xcdatamodeld called Shapes 2.xcdatamodel1, as shown in Figure 8–4. Each listing below Shapes.xcdatamodeld represents a version of your data model; the green check mark denotes the version your application is currently using.



**Figure 8–2.** *The current version of the data model*



**Figure 8-3.** *Selecting the version name and model basis*



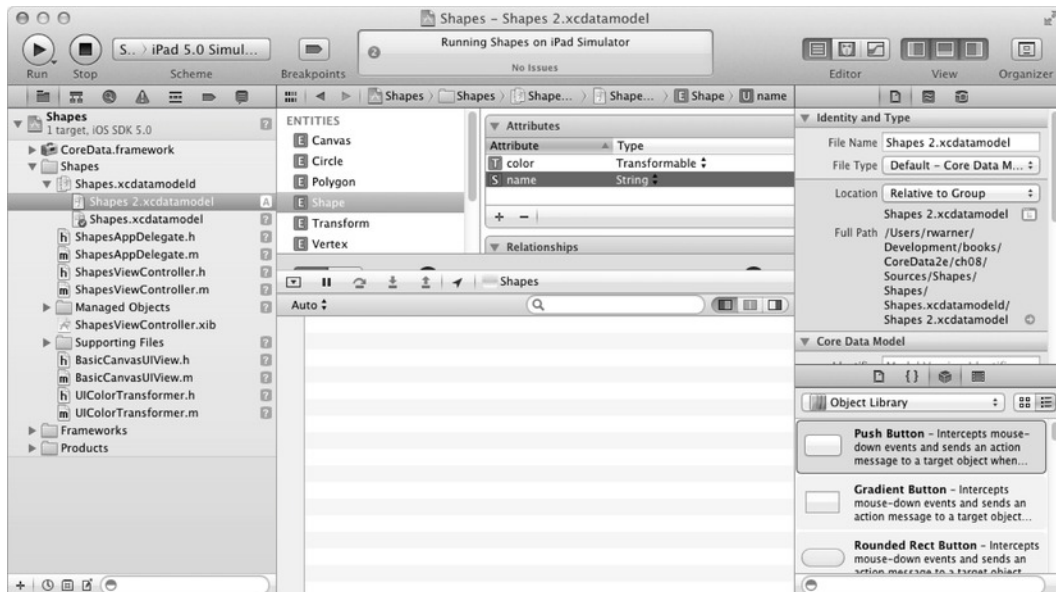
**Figure 8-4.** *The data model with two versions*

You can see that your original data model, `Shapes.xcdatamodel`, is the current version. Before changing the current version from `Shapes.xcdatamodel` to `Shapes 2.xcdatamodel`, run the Shapes application and create a few shapes so you have data to migrate across versions. You need data in your database to test that the data migrations through this chapter work properly.

Now, suppose you want to add a new name attribute to the Shape entity. If you edit your current model and add a name attribute to the Shape entity, you will have the unpleasant surprise of seeing your application crash on startup. That's because the data stored in the data store does not match the Core Data data model. One option to alleviate this problem is to delete your existing data store. This is an acceptable option while you are developing your app, but if you already have customers using this app, this will trigger their wrath. For a smoother experience, we strongly recommend versioning your model and using Core Data's migrations to preserve your users' data by migrating it from the old model to the new. Any changes you make to your data model go into the new version of the model, not any of the old ones. For you, this means that you'll add the name attribute to the Shape entity in the `Shapes 2` data model.



To make this change, select the Shapes 2 data model, and then add a name attribute of type String to the Shape entity in the normal way. Figure 8–5 shows the Xcode window with the new version of the Core Data model and the name attribute added to the Shape entity.

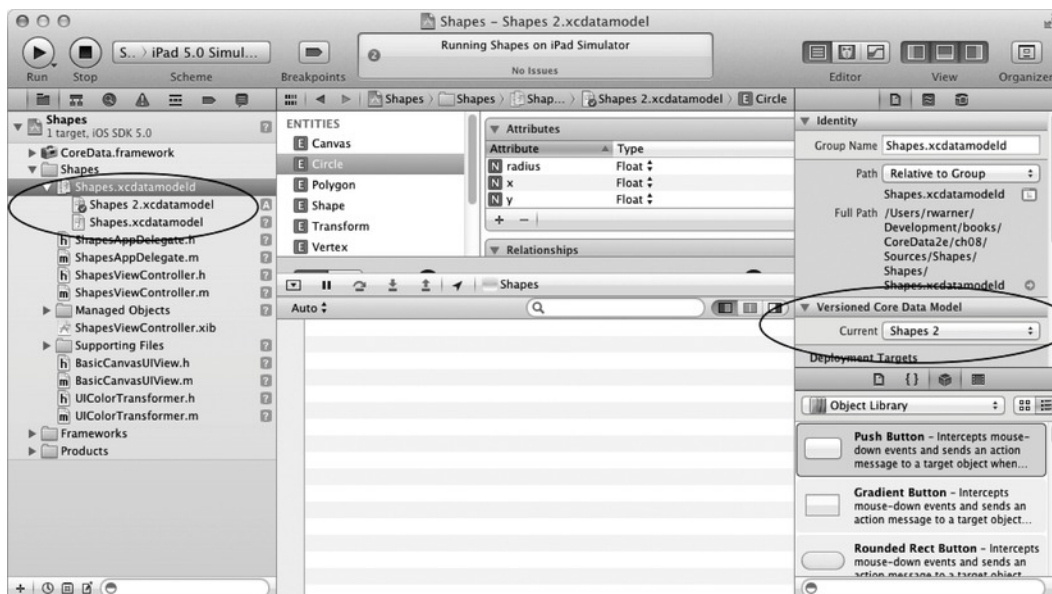


**Figure 8–5.** The Xcode window with a second version of the model

In addition to adding new attributes, you can make nonoptional attributes become optional and Core Data can deal with it. Change the following attributes to optional:

- The color attribute in the Shape entity
- The scale attribute in the Transform entity
- The index, x and y attributes in the Vertex entity

To change the current version of the model from Shapes to Shapes 2, select Shapes.xcdatamodel1 and set the current version in the Utility panel to Shapes 2, as shown in Figure 8–6.



**Figure 8–6.** Updating the current data model version to Shapes 2

At this point, your application has a new current version of the data model, but it's not ready to run yet. You need to define a policy for migrating the data from the version of the model called Shapes to the one called Shapes 2. If you tried running the application now, you would get a crash because, just as if you had changed the first model, the data store does not match the current model. The application needs to be told how you want it to handle switching from the old model to the new one. The rest of this chapter discusses the various ways to do that.

## Lightweight Migrations

Once you have a versioned data model, you can take advantage of Core Data's support for migrations as you evolve your data model. Each time you create a new data model version, users' application data must migrate from the old data model to the new. For the migration to occur, Core Data must have rules to follow to know how to properly migrate the data. You create these rules using what's called a *mapping model*. Support for creating these is built in to Xcode. For certain straightforward cases, however, Core Data has enough smarts to figure out the mapping rules on its own without requiring you to create a mapping model. Called *lightweight migrations*, these cases represent the least work for you as a developer. Core Data does all the work and migrates the data. This section details the lightweight migration process and walks you through the list of changes that are supported and how to implement them.

For your migration to qualify as a lightweight migration, your changes must be confined to this narrow band:

- Add or remove a property (attribute or relationship).
- Make a nonoptional property optional.
- Make an optional attribute nonoptional, as long as you provide a default value.
- Add or remove an entity.
- Rename a property.
- Rename an entity.

In addition to requiring less work from you, a lightweight migration using a SQLite data store runs faster and uses less space than other migrations. Because Core Data can issue SQL statements to perform these migrations, it doesn't have to load all the data into memory in order to migrate it, and it doesn't have to move the data from one store to the other. Core Data simply uses SQL statements to alter the SQLite database in place. If feasible, you should aggressively try to confine your data model changes to those that lightweight migrations support. If not, the other sections in this chapter walk you through more complicated migrations.

## Migrating a Simple Change

In the previous section called “Versioning” you created a new model version in the Shapes application called Shapes 2, and you added a new attribute called name to the Shape entity in the Shapes 2 model. The final step to performing a lightweight migration with that change is to tell the persistent store coordinator two things:

- It should migrate the model automatically.
- It should infer the mapping model.

You do that by passing those instructions in the options parameter of the persistent store coordinator's `addPersistentStoreWithType:` method. You first set up options, an `NSDictionary`, like this:

```
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:[NSNumber↵  
    numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption, [NSNumber↵  
    numberWithBool:YES], NSInferMappingModelAutomaticallyOption, nil];
```

This code creates an instance of `NSDictionary` with two entries.

- One with the key of `NSMigratePersistentStoresAutomaticallyOption`, value of YES, which tells the persistent store coordinator to automatically migrate the data.
- One with the key of `NSInferMappingModelAutomaticallyOption`, value of YES, which tells the persistent store coordinator to infer the mapping model.

You then pass this options object, instead of nil, for the options parameter in the call to `addPersistentStoreWithType:`. Open the `ShapesAppDelegate.m` file and change the `persistentStoreCoordinator` method to look like this:

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

    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"Shapes.sqlite"];
    NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:[NSNumber
        numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption, [NSNumber
        numberWithBool:YES], NSInferMappingModelAutomaticallyOption, nil];

    NSError *error = nil;
    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self managedObjectModel]];
    if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil URL:storeURL options:options error:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }
    return __persistentStoreCoordinator;
}
```

That's all you have to do to migrate the data. Build and run the application, which no longer crashes but instead shows all the shapes you've created. Open a Terminal and navigate to the directory that contains the SQLite file for the Shapes application. Unlike older versions of iOS, which kept both the pre-migration and the post-migration versions of the database file, iOS 5 keeps only the post-migration version. You'll find the post-migration file, `Shapes.sqlite`. Open it using the `sqlite3` application and run the `.schema` command to see the definition for the `ZSHAPE` table. You can see that the `Shapes.sqlite` file has added a column for the name attribute: `ZNAME VARCHAR`, as shown below.

```
sqlite> .schema
...
CREATE TABLE ZSHAPE ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZRADIUS
FLOAT, ZX FLOAT, ZY FLOAT, ZNAME VARCHAR, ZCOLOR BLOB );
...
sqlite>
```

## Migrating More Complex Changes

As claimed earlier, lightweight migrations handle a few more cases other than adding an attribute. Create another new version of the model as you did in the "Versioning" section. Xcode will call this version Shapes 3. Do the following to the model:

1. Add an entity called `Ellipse` that has four Float values: `x`, `y`, `width`, and `height`.
2. Make the transform relationship of the `Canvas` entity optional.

In Xcode, change the current model to use the new one you just created. Once again, build and run the Shapes application, and everything will start fine: the new data model

will be applied to the SQLite data store, and the data will migrate appropriately. Of course, ellipses haven't magically been added to the random shape mix because you haven't added any code to create them, and you won't bother doing that because it doesn't further your understanding of model versioning and data migration, but if you look at the schema for your database, you'll see the table for ellipses, like so:

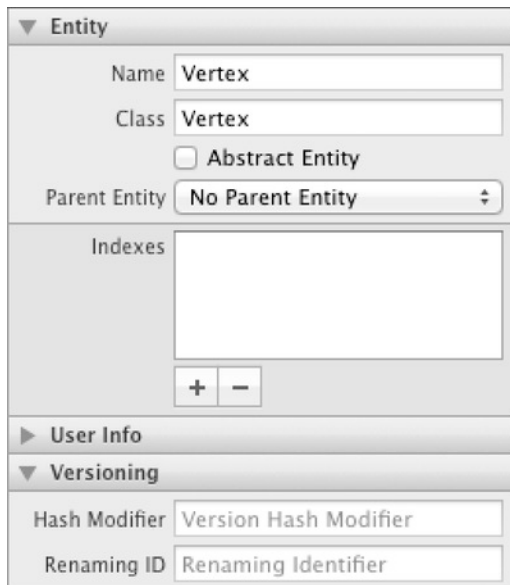
```
CREATE TABLE ZELLIPSE ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZHEIGHT
FLOAT, ZWIDTH FLOAT, ZX FLOAT, ZY FLOAT );
```

## Renaming Entities and Properties

Lightweight migrations also support renaming entities and properties but require a little more effort from you. In addition to changing your model, you must specify the old name for the item whose name you changed. You can do this in one of two ways:

- In the Xcode data modeler
- In code

The Xcode data modeler is the simpler option. When you select an entity or property in the Xcode data modeler, general information about that entity or property shows in the Utilities panel to the right. With the Utilities panel displayed, open the Data Model inspector, and then the Versioning section. In the Renaming ID field, enter the old name of whatever you've changed (see Figure 8–7).



**Figure 8–7.** The Versioning section with the Renaming Identifier field

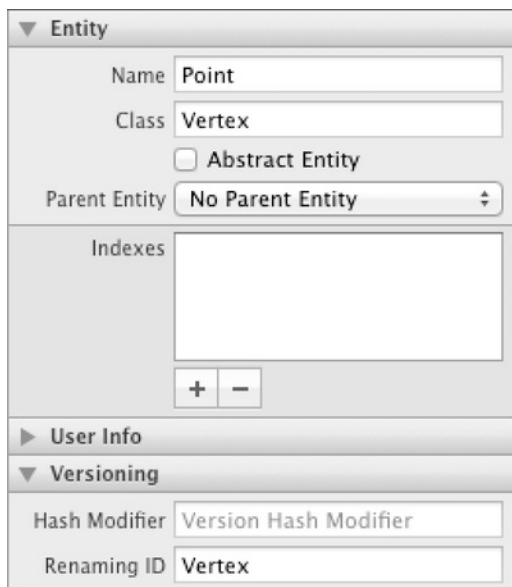
If you insist on specifying the old name in code (or if you're still running under Leopard and have no field for Renaming ID), you'd make a call to the `setRenamingIdentifier:` method of `NSEntityDescription` or `NSPropertyDescription`, depending on what you've

renamed, passing the old name for the entity or property. You do this after the model has loaded but before the call to open the persistent store (`addPersistentStoreWithType:`). If, for example, you want to change the name of the Vertex entity to Point, you'd add this code before the call to `addPersistentStoreWithType:`:

```
NSEntityDescription *point = [[managedObjectModel entitiesByName]
objectForKey:@"Point"];
[point setRenamingIdentifier:@"Vertex"];
```

Core Data takes care of migrating the data, but you still have the responsibility to update any code in your application that depends on the old name.

To see this in practice, create a new version of your data model and set it as the current version. You should now be on version 4 (Shapes 4). In this version of the model, you'll rename the Vertex entity to Point. Go to your new model file, `Shapes 4.xcdatamodel`, and rename the Vertex entity to Point. Then, go to the Versioning section in the Data Model inspector tab, and type the old name, Vertex, into the Renaming ID field so that Core Data will know how to migrate the existing data (as Figure 8–8 shows), and save this model.



**Figure 8–8.** Renaming Vertex to Point and specifying the Renaming ID

Wait! Before running the application, remember that you're responsible for changing any code that relies on the old name, Vertex. You could change the custom managed object class name from Vertex to Point, change the relationship name in the Polygon entity from vertices to points, and change any variable names appropriately. Let's keep it simple here and change the one place the Shapes application refers to the Vertex entity name. You find it in `Polygon.m` in the call to `[NSEntityDescription insertNewObjectForEntityForName:]`. It currently reads as follows:

```
Vertex *vertex = [NSEntityDescription insertNewObjectForEntityForName:@"Vertex"↵  
inManagedObjectContext:context];
```

Change the entity name passed to the method from Vertex to Point so the line reads like this:

```
Vertex *vertex = [NSEntityDescription insertNewObjectForEntityForName:@"Point"↵  
inManagedObjectContext:context];
```

You can now build and run the Shapes application; all existing shapes, including polygons, should appear. You can open the SQLite database and confirm that the schema now has a ZPOINT table

```
CREATE TABLE ZPOINT ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZINDEX  
INTEGER, ZPOLYGON INTEGER, ZX FLOAT, ZY FLOAT );
```

and no ZVERTEX table.

As you've seen, lightweight migrations are nearly effortless, at least for you. Core Data handles the difficult work of figuring out how to map and migrate the data from the old model to new. However, if your model changes don't fit within what lightweight migrations can handle, you must specify your own mapping model.

## Creating a Mapping Model

When your data model changes exceed Core Data's ability to infer how to map data from the old model to the new, you can't use a lightweight migration to automatically migrate your data. Instead, you have to create what's called a *mapping model* to tell Core Data how to execute the migration. This section walks you through the steps involved in creating a mapping model for your migration. A mapping model is roughly analogous to a data model—whereas a data model contains entities and properties, a mapping model, which is of type `NSMappingModel`, has entity mappings (of type `NSEntityMapping`) and property mappings (of type `NSPropertyMapping`). The entity mappings do just what you'd expect them to do: they map a source entity to a target entity. The property mappings, as well, do what you'd think: map source properties to target properties. The mapping model uses these mappings, along with their associated types and policies, to perform the migration.

**NOTE:** Apple's documentation and code use the terms *destination* and *target* interchangeably to refer to the new data model. In this chapter, we follow suit and use both *destination* and *target* interchangeably.

In a typical data migration, most entities and properties haven't changed from the old version to the new. For these cases, the entity mappings simply copy each entity from the source model to the target model. When you create a mapping model, you'll notice that Xcode generates these entity mappings, along with anything else it can infer from your model changes, and stores these mappings in the mapping model. These mappings represent how Core Data would have migrated your data in a lightweight

migration. You have to create new mappings, or adjust the mappings Xcode generates, to change how your data migrates.

## Understanding Entity Mappings

Each entity mapping, represented by an `NSEntityMapping` instance, contains three things:

- A source entity
- A destination entity
- A mapping type

When Core Data performs the migration, it uses the entity mapping to move the source to the destination, using the mapping type to determine how to do that. Table 8–1 lists the mapping types, their corresponding Core Data constants, and what they mean.

**Table 8–1.** *The Entity Mapping Types*

| Type      | Core Data Constant                        | Meaning  |
|-----------|---|--|
| Add       | <code>NSAddEntityMappingType</code>       | The entity is new in the destination model—it doesn't exist in the source model—and should be added.   |
| Remove    | <code>NSRemoveEntityMappingType</code>    | The entity doesn't exist in the destination model and should be removed.   |
| Copy      | <code>NSCopyEntityMappingType</code>      | The entity exists in both the source and destination models unchanged and should be copied as is.  |
| Transform | <code>NSTransformEntityMappingType</code> | The entity exists in both the source and destination models but with changes. The mapping tells Core Data how to migrate each source instance to a destination instance. |
| Custom    | <code>NSCustomEntityMappingType</code>    | The entity exists in both the source and destination models but with changes. The mapping tells Core Data how to migrate each source instance to a destination instance. |

The Add, Remove, and Copy types don't generate much interest because lightweight migrations handle these types of entity mappings. The Transform and Custom types, however, are what make this section of the book necessary. They tell Core Data that each source entity instance must be transformed, according to any specified rules, into an instance of the destination entity. You'll see an example of both a Transform and a Custom entity mapping type in this chapter. If you specify a value expression for one of the entity's properties, the entity mapping is of type Transform. If you specify a custom



migration policy for the entity mapping, the entity mapping becomes a Custom type. As you work through this chapter, pay attention to the types the Core Data mapping modeler makes to your mapping model in response to changes you make.

To specify the rules for a Custom entity mapping type, you create a migration policy, which is a class you write that derives from `NSEntityMigrationPolicy`. You then set the class you create as the custom policy for the entity mapping in the Xcode mapping modeler.

Core Data runs your migration in three stages.

1. It creates the objects in the destination model, including their attributes, based on the objects in the source model.
2. It creates the relationships among the objects in the destination model.
3. It validates the data in the destination model and saves it.

You can customize how Core Data performs these three steps through the Custom Policy. The `NSEntityMigrationPolicy` class has seven methods you can override to customize how Core Data will migrate data from the source entity to the target entity, though you'll rarely override all of them. These methods, listed in Apple's documentation for the `NSEntityMigrationPolicy` class, provide various places during the migration that you can override and change Core Data's migration behavior. You can override as few or as many of these methods as you'd like, though a custom migration policy that overrides none of the methods is pointless. Typically, you'll override `createDestinationInstancesForSourceInstance:` if you want to change how destination instances are created or how their attributes are populated with data. You'll override `createRelationshipsForDestinationInstance:` if you want to customize how relationships between the destination entity and other entities are created. Finally, you'll override `performCustomValidationForEntityMapping:` if you want to perform any custom validations during your migration.

The `createDestinationInstancesForSourceInstance:` method carries with it a caveat: if you don't call the superclass's implementation, which you probably won't because you're overriding this method to change the default behavior, you must call the migration manager's `associateSourceInstance:withDestinationInstance:forEntityMapping:` method to associate the source instance with the destination instance. Forgetting to do this will cause problems with your migration. You'll learn the proper way to call this method later in this chapter, with the Shapes application.

## Understanding Property Mappings

A property mapping, like an entity mapping, tells Core Data how to migrate source to destination. A property mapping is an instance of `NSPropertyMapping` and contains the following three things:

- The name of the property in the source entity.
- The name of the property in the destination entity.
- A value expression that tells Core Data how to get from source to destination.

To change the way a property mapping migrates data, you provide the value expression for the property mapping to use. Value expressions follow the same syntax as predicates and use the following six predefined keys to assist with retrieving values:

- `$manager`, which represents the migration manager.
- `$source`, which represents the source entity.
- `$destination`, which represents the destination entity.
- `$entityMapping`, which represents the entity mapping.
- `$propertyMapping`, which represents this property mapping.
- `$entityPolicy`, which represents the entity migration policy.

As you can see, these keys have names that make deducing their purposes easy. When you create a mapping model, you can explore the property mappings Xcode infers from your source and target data models to better understand what these keys mean and how they're used.

The simplest value expressions copy an attribute from the source entity to the destination entity. If you have a `Person` entity, for example, that has a `name` attribute and the `Person` entity hasn't changed between your old and your new model versions, the value expression for the `name` attribute in your `PersonToPerson` entity mapping would be as follows:

```
$source.name
```

You can also perform manipulations of source data using value expressions. Suppose, for example, that the same `Person` entity had an attribute called `salary` that stores each person's salary. In the new model, you want to give everyone four percent raises. Your value expression would look like this:

```
$source.salary*1.04
```

Since properties represent both attributes and relationships, property mappings represent mappings for both attributes and relationships. The typical value expression for a relationship calls a function, passing the migration manager, the destination instances, the entity mapping, and the name of the source relationship. For example, if your old data model had a relationship called `staff` in the `Person` entity that represented everyone who reported to this person and your new model has renamed this relationship to `reports`, the value expression for the `reports` property would look like this:

```
FUNCTION($manager, "destinationInstancesForEntityMappingNamed:sourceInstances:",  
"PersonToPerson", $source.staff)
```

Note that you use the `$manager` key to pass the migration manager, that you get the destination instances from the entity mapping for the source instances, that you pass the appropriate entity mapping (`PersonToPerson`), and that you pass the old relationship that you get from the `$source` key.

## Creating a New Model Version That Requires a Mapping Model

Earlier in this chapter, you created an entity called `Ellipse` that had four attributes: `x`, `y`, `width`, and `height`. You didn't do anything with that entity, and it wasn't incorporated into the application. The time for `Ellipse` to shine has come, however, and you're going to incorporate it into the application. Acting on the idea that `Circles` are just `Ellipses` that have the same `width` and `height` (which may make mathematicians shudder but fits nicely with the `Shape` application's code), you'll migrate all the `Circle` instances to the `Ellipse` entity and delete the `Circle` entity entirely.

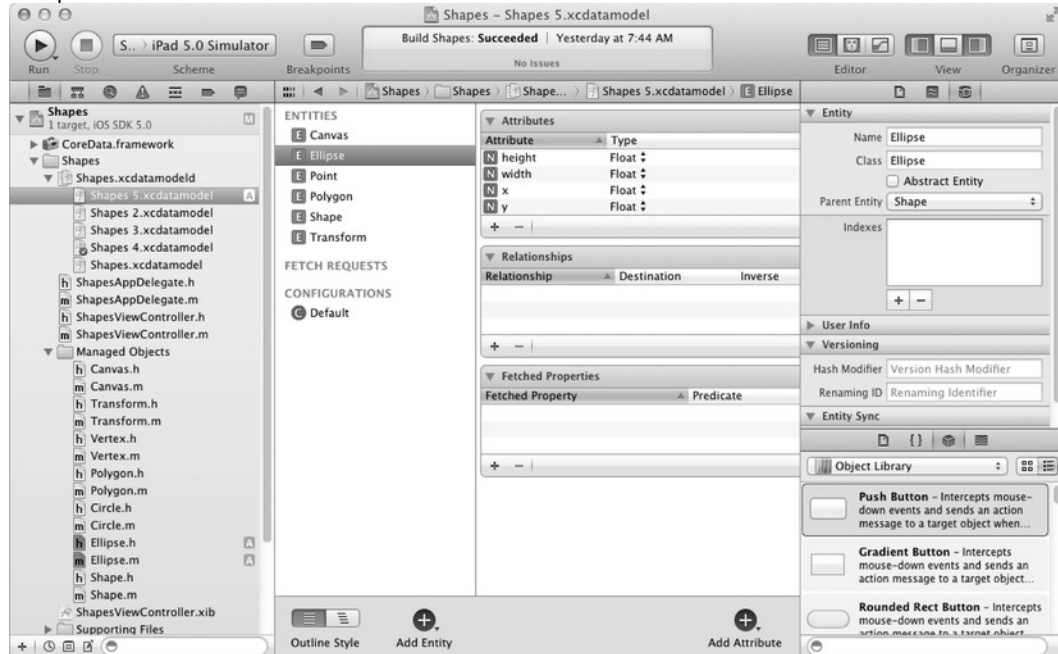
You've also decided that you'd rather call the `scale` attribute of the `Transform` entity `scalarValue`. What's more, you've decided that the shapes that the application draws are too big and that the scales should be reduced 20 percent to 0.4 and 0.8, instead of the existing 0.5 and 1.0.

To perform this migration, you'll need to create a mapping model and a custom policy that will migrate each `Circle` instance to a corresponding `Ellipse` instance. The `x` and `y` values will copy across as is, and the `radius` value from the `Circle` will be copied to the `width` and `height` attributes of the `Ellipse`. This custom policy will attach to an entity mapping that's responsible for mapping `Circles` to `Ellipses`.

You'll also need to modify the property mapping for `scalarValue` to multiply each `scale` value by 0.8 as the data migrates.

To get started, create a new version of your data model; you should be up to version 5. In the new data model, delete the `Circle` entity. Next, go to the `Transform` entity, and change the name for the `scale` attribute to `scalarValue`. Change the parent entity for the `Ellipse` entity to `Shape`. Create or generate a custom class for the `Ellipse` entity, put it in the `Managed Objects` group, and then set the `Ellipse` entity in the data model (if necessary) to this class, as shown in Figure 8–9.

## Ellipse



**Figure 8–9.** The entity set to the *Ellipse* class

Now that the entity is defined, modify the *Ellipse* code according to Listings 8–1 and 8–2 so that you can use it from the controller.

**Listing 8–1.** *Ellipse.h*

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
#import "Shape.h"

@interface Ellipse : Shape

@property (nonatomic, retain) NSNumber * height;
@property (nonatomic, retain) NSNumber * width;
@property (nonatomic, retain) NSNumber * x;
@property (nonatomic, retain) NSNumber * y;

+ (Ellipse *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context;

@end
```

**Listing 8–2.** *Ellipse.m*

```
#import "Ellipse.h"

@implementation Ellipse

@dynamic height;
@dynamic width;
@dynamic x;
```

```

@dynamic y;

+ (Ellipse *)randomInstance:(CGPoint)origin inContext:(NSManagedObjectContext *)context
{
    Ellipse *ellipse = [NSEntityDescription insertNewObjectForEntityForName:@"Ellipse"
inManagedObjectContext:context];

    float width = 10 + (arc4random() % 90);
    float height = 10 + (arc4random() % 90);
    ellipse.x = [NSNumber numberWithFloat:origin.x];
    ellipse.y = [NSNumber numberWithFloat:origin.y];
    ellipse.width = [NSNumber numberWithFloat:width];
    ellipse.height = [NSNumber numberWithFloat:height];

    return ellipse;
}

@end

```

The Ellipse class looks suspiciously like the Circle class but with width and height properties instead of a single radius property. You now must update the Shapes application code in a few places to use Ellipse instead of Circle. In `ShapesViewController.m`, change the line

```
#import "Circle.h"
```

to this

```
#import "Ellipse.h"
```

Then, in the `createShapeAt:` method, change the code that creates the shapes to create an Ellipse instead of a Circle for the case where the random value for type is 0:

```

if (type == 0) { // Ellipse
    shape = [Ellipse randomInstance:point inContext:self.managedObjectContext];
}
else { // Polygon
    shape = [Polygon randomInstance:point inContext:self.managedObjectContext];
}

```

In `BasicCanvasUIView.m`, change the `Circle.h` import to instead import `Ellipse.h`, then find the line in the `drawRect:` method that looks like this:

```
if ([entityName compare:@"Circle"] == NSOrderedSame) {
```

The code inside the if condition draws a Circle, and you must change it to draw an Ellipse. The updated code looks like this:

```

if ([entityName compare:@"Ellipse"] == NSOrderedSame) {
    Ellipse *ellipse = (Ellipse *)shape;
    float x = [ellipse.x floatValue];
    float y = [ellipse.y floatValue];
    float width = [ellipse.width floatValue];
    float height = [ellipse.height floatValue];

    CGContextFillEllipseInRect(context, CGRectMake(x - (width / 2), y - (height / 2),
width, height));
}

```

```
}
```

This completes the code changes to use Ellipses instead of Circles. You now must make code changes to use the new name for the scale attribute of the Transform entity, `scalarValue`. In `Transform.h`, change the property name from `scale` to **`scalarValue`**. In `Transform.m`, change this line

```
@dynamic scale;
```

to this

```
@dynamic scalarValue;
```

and change this line

```
transform.scale = [NSNumber numberWithFloat:scale];
```

to this

```
transform.scalarValue = [NSNumber numberWithFloat:scale];
```

Open `BasicCanvasUIView.m`, and change the `scale:` method to look like this:

```
-(float)scale {
    NSManagedObject *transform = [canvas valueForKey:@"transform"];
    return [[transform valueForKey:@"scalarValue"] floatValue];
}
```

You must also change the code that initially creates the two Transform instances to use 0.4 and 0.8 for the scale instead of 0.5 and 1.0. This code is in `ShapesViewController.m`, in the `viewDidLoad:` method, and looks like this:

```
// If there aren't any canvases, then we create them
Transform *transform1 = [Transform initWithScale:1 inContext:self.managedObjectContext];
canvas1 = [Canvas initWithTransform:transform1 inContext:self.managedObjectContext];
```

```
Transform *transform2 = [Transform initWithScale:0.5
inContext:self.managedObjectContext];
canvas2 = [Canvas initWithTransform:transform2 inContext:self.managedObjectContext];
```

Change it to this:

```
// If there aren't any canvases, then we create them
Transform *transform1 = [Transform initWithScale:0.8
inContext:self.managedObjectContext];
canvas1 = [Canvas initWithTransform:transform1 inContext:self.managedObjectContext];
```

```
Transform *transform2 = [Transform initWithScale:0.4
inContext:self.managedObjectContext];
canvas2 = [Canvas initWithTransform:transform2 inContext:self.managedObjectContext];
```

You have your new model, and you've updated the code to use it. You're ready to create a mapping model that will transform your Circles into Ellipses and reduce the scales by 20 percent.

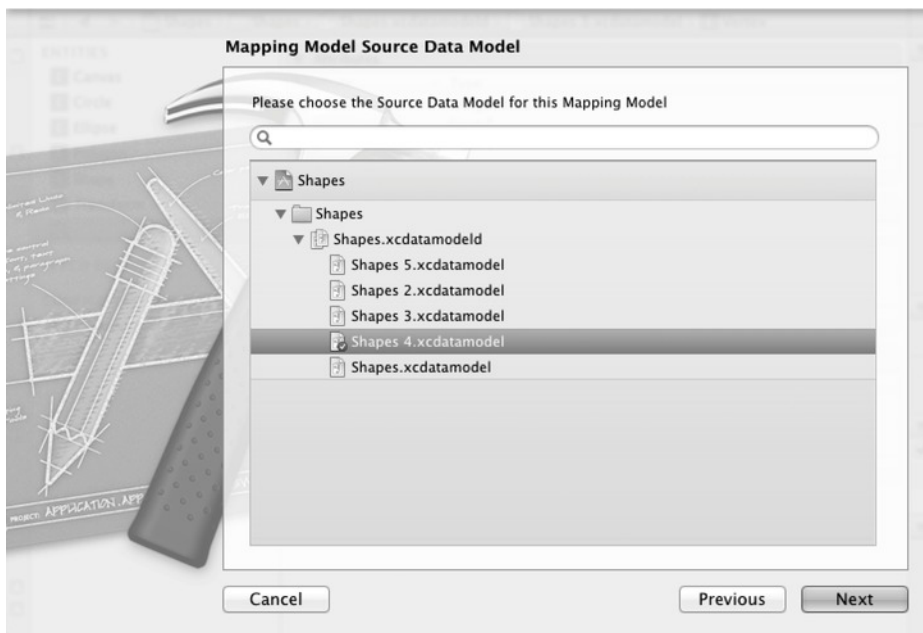
## Creating a Mapping Model

To create a mapping model, go to Xcode's menu, and select **File > New > New File**. In the ensuing dialog box, select Core Data under iOS on the left, and select Mapping Model on the right, as shown in Figure 8–10. Click Next.

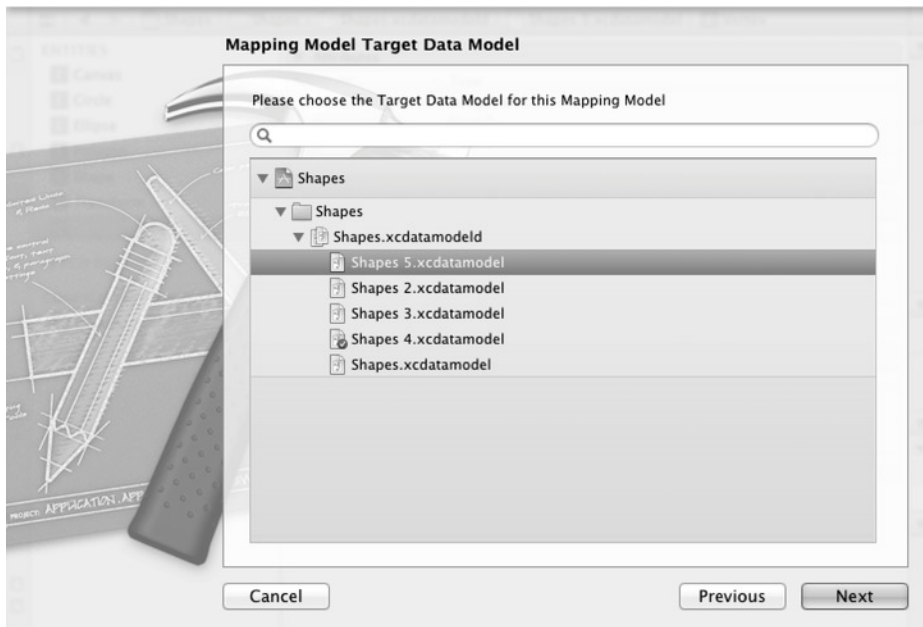


**Figure 8–10.** *Creating a new mapping model*

The next step is to select the source model. Select `Shapes 4.xcdatamodel`, as shown in Figure 8–11, and click Next. You are then asked to select the target model. Select `Shapes 5.xcdatamodel` for the destination, as shown in Figure 8–12, and click Next. The last step is to name the mapping model. Name it `Model4to5.xcmappingmodel` and click Create.



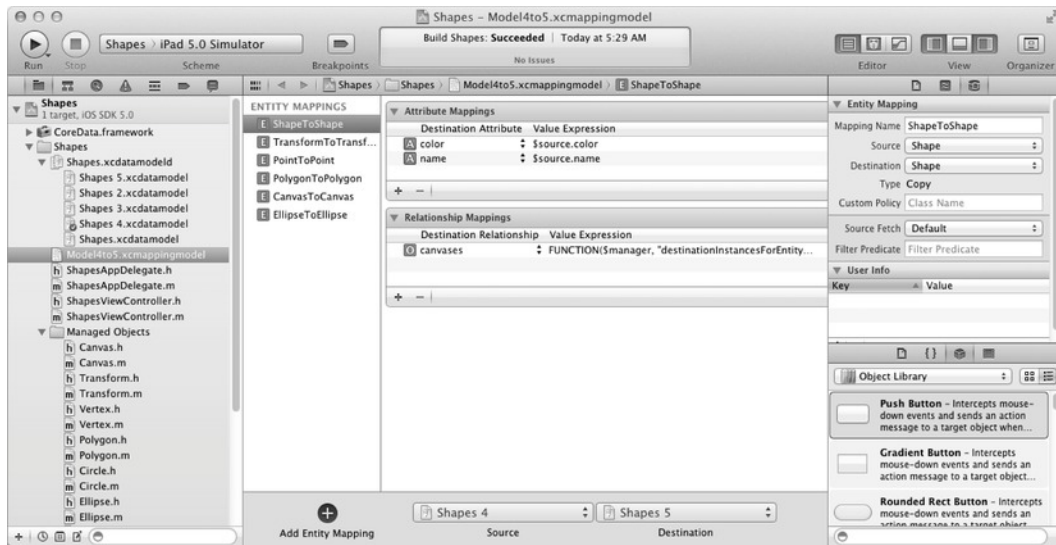
**Figure 8–11.** *Selecting the source data model*



**Figure 8–12.** *Selecting the target data model*

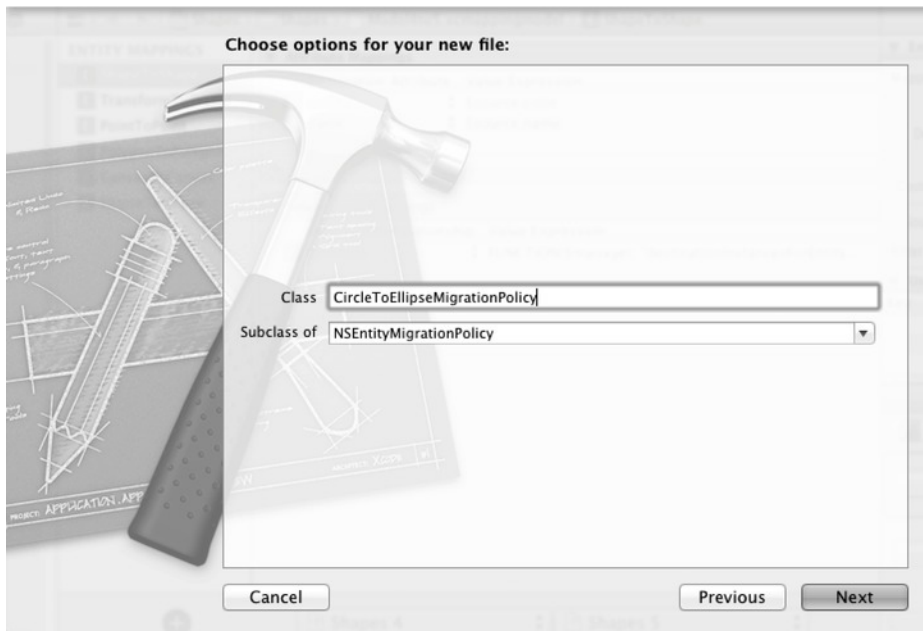
You should now see Xcode with your mapping model created, including all the entity mappings and property mappings that Core Data could infer, as shown in Figure 8–13.





**Figure 8–13.** *Your new mapping model*

Creating a mapping model puts you in the same place you'd be with a lightweight migration, with mappings that Core Data can infer from your source and target data models. The next step is to customize the mapping model to migrate Circles to Ellipses. Start by creating the custom policy you'll use, which is a subclass of `NSEntityMigrationPolicy`. Create a new Objective-C class (under the Cocoa Touch templates) called `CircleToEllipseMigrationPolicy` that subclasses `NSEntityMigrationPolicy`, as shown in Figure 8–14. Once it's created, create a new group for it called Migration Mappings.



**Figure 8–14.** *Creating a new migration policy*

The header file for your new migration policy class is simple, as you can see:

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

```
@interface CircleToEllipseMigrationPolicy : NSEntityMigrationPolicy
```

```
@end
```

In the implementation file, `CircleToEllipseMigrationPolicy.m`, you want to override the `createDestinationInstancesForSourceInstance:` method. While performing the migration, Core Data will call this method each time it goes to create an `Ellipse` instance from a `Circle` instance. Keep in mind that both `Circle` and `Ellipse` inherit from the `Shape` entity, so you're responsible for handling any of `Shape`'s properties as well. In your implementation, you create an `Ellipse` instance; copy over the `x`, `y`, and `color` values from the `Circle` instance passed to this method; and then copy the `Circle`'s `radius` value to the `Ellipse`'s `width` and `height` values. Finally, you tell the migration manager about the relationship between the source's `Circle` instance and the target's new `Ellipse` instance, which is an important step for the migration to occur properly. Listing 8–3 shows the implementation file.

**Listing 8–3.** *CircleToEllipseMigrationPolicy.m*

```
#import "CircleToEllipseMigrationPolicy.h"
```

```
@implementation CircleToEllipseMigrationPolicy
```

```

- (BOOL)createDestinationInstancesForSourceInstance:(NSManagedObject *)sInstance↵
entityMapping:(NSEntityMapping *)mapping manager:(NSMigrationManager *)manager↵
error:(NSError **)error {
    // Create the ellipse managed object
    NSManagedObject *ellipse = [NSEntityDescription↵
insertNewObjectForEntityForName:[mapping destinationEntityName]↵
inManagedObjectContext:[manager destinationContext]];

    // Copy the x, y, and color values from the Circle to the Ellipse
    [ellipse setValue:[sInstance valueForKey:@"x"] forKey:@"x"];
    [ellipse setValue:[sInstance valueForKey:@"y"] forKey:@"y"];
    [ellipse setValue:[sInstance valueForKey:@"color"] forKey:@"color"];

    // Copy the radius value from the Circle to the width and height of the Ellipse
    [ellipse setValue:[sInstance valueForKey:@"radius"] forKey:@"width"];
    [ellipse setValue:[sInstance valueForKey:@"radius"] forKey:@"height"];

    // Set up the association between the Circle and the Ellipse for the migration manager
    [manager associateSourceInstance:sInstance withDestinationInstance:ellipse↵
forEntityMapping:mapping];

    return YES;
}

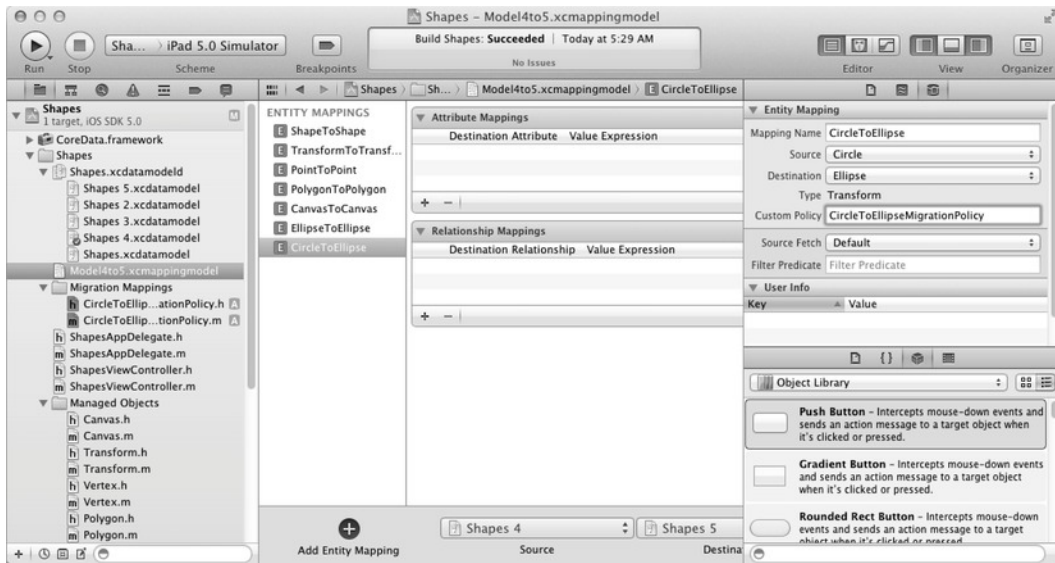
@end

```

You might notice that we've ignored the name attribute of Shape. Normally, you would copy this property from the source to the target as well, but we've done nothing with the name attribute in the Shapes application. We never added it to the Shape custom class, and we haven't added any code that would fill that value, so we left it out here to illustrate a point: if you leave an attribute out of a mapping policy, that attribute won't be copied over and will be blank in the target entity.

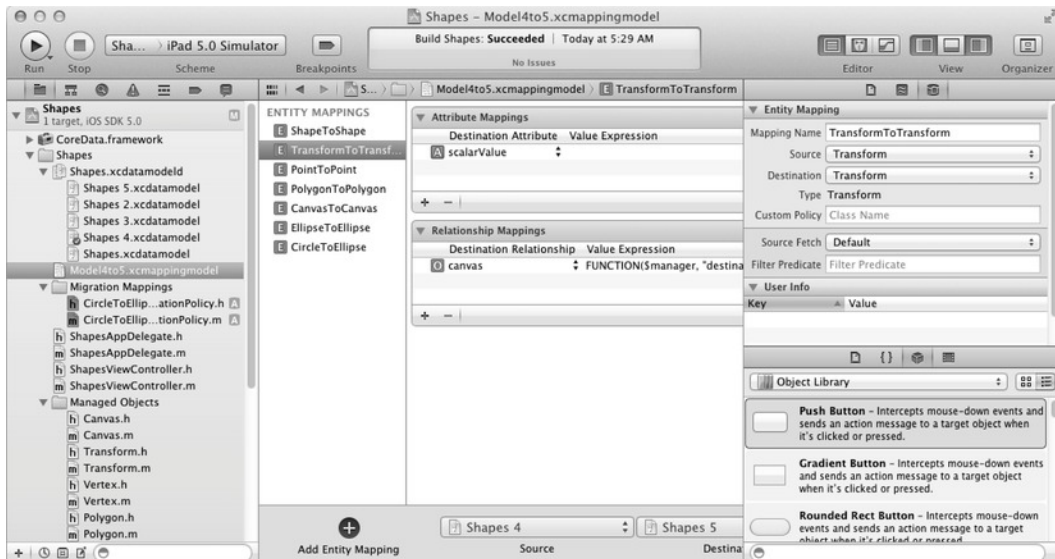
Notice also that we didn't copy over the canvases relationships. The relationships are copied over by default in `NSEntityMigrationPolicy's createRelationshipsForDestinationInstance:` method. By not overriding that method, we get Core Data to copy those over for us.

Now, you're ready to create the entity mapping between Circle and Ellipse. With your mapping model (`Model4to5.xcmappingmodel`) selected, click the Add Entity Mapping button. Name the new entity mapping `CircleToEllipse` and set its Source to Circle, its Destination to Ellipse, and its Custom Policy to `CircleToEllipseMigrationPolicy`, as shown in Figure 8–15.



**Figure 8-15.** Adding and configuring the *CircleToEllipse* entity mapping

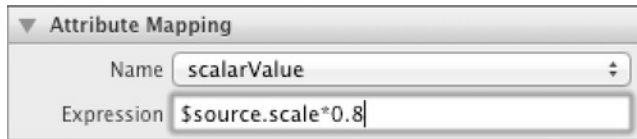
This takes care of mapping Circles to Ellipses. Next, you need to set up the property mapping for migrating the scale values of Transform to scalarValues at 80 percent of their original values. Select the TransformToTransform entity mapping in the mapping model, which displays its information, including its Property Mappings, as shown in Figure 8-16.



**Figure 8-16.** The *TransformToTransform* mapping

Select the `scalarValue` property mapping, and under Attribute Mapping enter the following for the Expression setting (as shown in Figure 8–17):

```
$source.scale*0.8
```



**Figure 8–17.** Setting the Expression to map `scale` to `scalarValue`

This tells Core Data to take the value from the `scale` attribute from the source data model, multiply it by 0.8, and store the value in the `scalarValue` attribute of the target.

Your mapping model is ready to perform the migration. Just as with lightweight migrations, however, you need to add some code to the application delegate to tell Core Data to execute a migration.

## Migrating Data

Creating the mapping model is essential to performing a migration that lightweight migrations can't handle, but you still must perform the actual migration. This section explains how to do this and then update the code for the Shapes application to actually run the migration. By the end of this section, you will have a Shapes application that includes all the shapes it had before, including the circles, but now the circles are ellipses with the same width and height. As you tap the screen, you'll see random ellipses being created. You also may notice that everything is 80 percent of the size it was before.

Telling Core Data to migrate from the old model to the new one using a mapping model you create is similar to how you tell Core Data to use a lightweight migration. As you learned earlier, the way to tell Core Data to perform a lightweight migration is to pass an options dictionary that contains YES for two keys:

- `NSMigratePersistentStoresAutomaticallyOption`
- `NSInferMappingModelAutomaticallyOption`

The first key tells Core Data to automatically migrate the data, and the second tells Core Data to infer the mapping model.

For a migration using a mapping model you created, you clearly want Core Data to still automatically perform the migration, so you still set the `NSMigratePersistentStoresAutomaticallyOption` to YES. Since you've created the mapping model, however, you don't want Core Data to infer anything; you want it to use your mapping model. Therefore, you set `NSInferMappingModelAutomaticallyOption` to NO, or you leave it out of the options dictionary entirely.

How, then, do you specify the mapping model for Core Data to use to perform the migration? Do you set it in the options dictionary? Do you pass it somehow to the

persistent store coordinator's `addPersistentStoreWithType:` method? Do you set it into the persistent store coordinator? Into the managed object model? Into the context?

The answer, which may seem a little shocking, is that you do nothing. Core Data will figure it out. It searches through your mapping models, finds one that's appropriate for migrating from the source to the destination, and uses it. This makes migrations almost criminally easy.

## Running Your Migration

To run the migration of your Shapes data store, open the `ShapesAppDelegate.m` file, and remove the `NSInferMappingModelAutomaticallyOption` key from the options dictionary. This means you change this line

```
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:[NSNumber
numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption, [NSNumber
numberWithBool:YES], NSInferMappingModelAutomaticallyOption, nil];
```

to this

```
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:[NSNumber
numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption, nil];
```

This is all you have to do to get Core Data to migrate your data using your mapping model, `Model4to5.xcmappingmodel`. Set your current model version to Shapes 5, and then build and run the Shapes application; you should see all the shapes you had in the database before you ran the migration, albeit shown at 80 percent of the size they were before. Go look at the SQLite database to confirm that the migration ran. You can see that the `ZSHAPE` table now has columns for `ZWIDTH` and `ZHEIGHT`, but none for `ZRADIUS`.

```
CREATE TABLE ZSHAPE ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZHEIGHT
FLOAT, ZWIDTH FLOAT, ZX FLOAT, ZY FLOAT, ZNAME VARCHAR, ZCOLOR BLOB );
```

You can also fetch the values for `ZSCALARVALUE` (not `ZSCALE`) from the `ZTRANSFORM` table to verify that they're 20 percent smaller than the 0.5 and 1.0 that they used to be.

```
sqlite> select distinct zscalarvalue from ztransform;
0.4
0.8
```

Tap the screen of the Shapes application a few times to see some ellipses created, as shown in Figure 8–18.



**Figure 8–18.** *The Shapes application with ellipses*

## Custom Migrations

In most data migration cases, using the default three-step migration process (create the destination objects, create the relationships, and validate and save) is sufficient. Data objects are created in the new data store from the previous version, then all

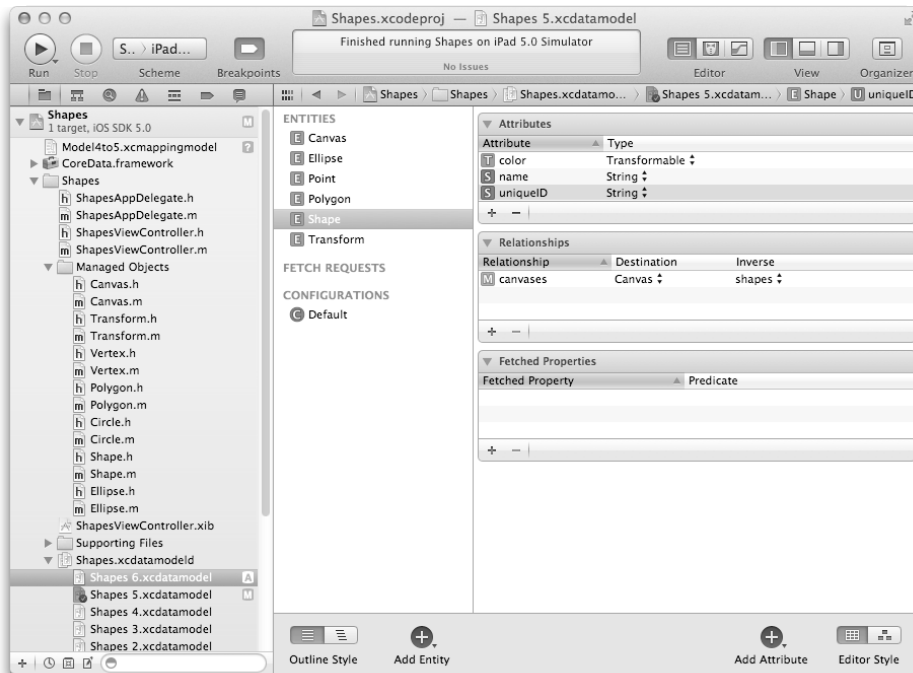
relationships are created, and finally the data is validated and persisted. There are, however, some cases where you want to intervene in the middle of the migration. This is where you need custom migrations.

Let's imagine a case where, as part of the migration, a new attribute in the destination object has to be prepopulated before the application can function with the new model. Taking control of the migration process initialization means that you need to perform all the work that Core Data usually does for you. This means that you need to do the following:

- Make sure migration is actually needed.
- Set up the migration manager.
- Run your migration by splitting the model into independent parts.

This section goes through a custom migration example step by step to show you how to make a non-trivial migration work.

To illustrate this example, start by creating a new version of the Shapes model. You should now be on version 6 (Shapes 6). Make sure the current model version is set to the new version of the model you just created. In the new model, edit the Shape entity and add an attribute called 'uniqueID' of type String, as shown in Figure 8–19. You will use the migration process to populate this field with a Universally Unique Identifier (UUID) so that each shape in your application has a unique serial number.



**Figure 8–19.** The shape entity with a new field



## Making Sure Migration Is Needed

To validate that a model is compatible with the persistent store, use the `isConfiguration:compatibleWithStoreMetadata:` method of the `NSManagedObjectModel` class before adding the persistent store to the coordinator. The data store metadata can be retrieved by querying the coordinator using the `metadataForPersistentStoreOfType:URL:error:` method.

Open `ShapesAppDelegate.m` and edit the `persistentStoreCoordinator` getter as shown in Listing 8–4.

**Listing 8–4.** *persistentStoreCoordinator Detecting if Migration Is Needed*

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

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

    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self managedObjectModel]];

    NSError *error = nil;
    NSDictionary *sourceMetadata = [NSPersistentStoreCoordinator
        metadataForPersistentStoreOfType:NSSQLiteStoreType URL:storeURL error:&error];

    NSManagedObjectModel *destinationModel = [__persistentStoreCoordinator
        managedObjectModel];

    BOOL pscCompatible = [destinationModel isConfiguration:nil
        compatibleWithStoreMetadata:sourceMetadata];

    if(!pscCompatible) {
        // Migration is needed

        ... perform the migration
    }

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

If migration is not needed, then the persistent store can be registered with the coordinator as usual.

## Setting Up the Migration Manager

Once it has been determined that migration is indeed needed, the next step is to set up the migration manager, which is a subclass of `NSMigrationManager`. To do this, you need to get a hold of the model that matches the current persistent store: the previous version of the model. This is the model you use as the source model of your migration process.

The source model can be obtained using the

`mergedModelFromBundles:forStoreMetadata:` method, which looks for a model in the application main bundle that matches the given metadata.

Start creating your custom migration manager by creating a new class that extends `NSMigrationManager`. In this example, you'll call it `MyMigrationManager` (see Listing 8–5).

### Listing 8–5. *MyMigrationManager.h*

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

@interface MyMigrationManager : NSMigrationManager
@end
```

The implementation of the migration creates a new UUID for each `Shape` instance that is being migrated and assigns the `uniqueID` attribute during the migration (see Listing 8–6).

### Listing 8–6. *MyMigrationManager.m*

```
#import "MyMigrationManager.h"

@implementation MyMigrationManager

-(void)associateSourceInstance:(NSManagedObject*)sourceInstance
withDestinationInstance:(NSManagedObject *)destinationInstance
forEntityMapping:(NSEntityMapping *)entityMapping {

    [super associateSourceInstance:sourceInstance
withDestinationInstance:destinationInstance forEntityMapping:entityMapping];

    NSString *name = [entityMapping destinationEntityName];

    if([name compare:@"Ellipse"] == NSOrderedSame || [name compare:@"Polygon"] ==
NSOrderedSame) {
        // Generate UUID
        CFUUIDRef theUUID = CFUUIDCreate(NULL);
        CFStringRef string = CFUUIDCreateString(NULL, theUUID);
        CFRelease(theUUID);
        NSString *uuid = (__bridge NSString *)string;

        [destinationInstance setValue:uuid forKey:@"uniqueID"];
    }
}

@end
```

## Running the Migration

In this last step, you need to set your own class as the manager in charge of performing this migration. Add an import for `MyMigrationManager.h` at the top of the `ShapesAppDelegate.m` file and edit the `persistentStoreCoordinator` method again, as shown in Listing 8–7.

**Listing 8–7.** *Launching the Custom Migration*

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

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

    NSURL *newStoreURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"Shapes-Temp.sqlite"];

    NSError *error = nil;
    NSDictionary *sourceMetadata = [NSPersistentStoreCoordinator
        metadataForPersistentStoreOfType:NSSQLiteStoreType URL:storeURL error:&error];

    if(sourceMetadata != nil) {
    }

    NSManagedObjectModel *destinationModel = [self managedObjectModel];

    BOOL pscCompatible = [destinationModel isConfiguration:nil
        compatibleWithStoreMetadata:sourceMetadata];

    if(!pscCompatible) {
        NSLog(@"Migration is needed");
        // Migration is needed

        NSManagedObjectModel *sourceModel = [NSManagedObjectModel mergedModelFromBundles:nil
            forStoreMetadata:sourceMetadata];

        MyMigrationManager *migrationManager = [[MyMigrationManager alloc]
            initWithSourceModel:sourceModel destinationModel:destinationModel];

        NSMappingModel *mappingModel = [NSMappingModel
            inferredMappingModelForSourceModel:sourceModel destinationModel:destinationModel
            error:&error];

        if(mappingModel == nil) {
            NSLog(@"Could not find a mapping model");
            abort();
        }

        if(![migrationManager migrateStoreFromURL:storeURL
            type:NSSQLiteStoreType
```

```

        options:nil
        withMappingModel:mappingModel
        toDestinationURL:newStoreURL
        destinationType:NSSQLiteStoreType
        destinationOptions:nil
        error:&error]) {
    // Deal with error
    NSLog(@"Error migrating %@, %@", error, [error userInfo]);
    abort();
}

// Replace the old store with the new one now that it's migrated
NSFileManager *fm = [NSFileManager defaultManager];
NSURL *backupURL = [[self applicationDocumentsDirectory]
    URLByAppendingPathComponent:@"Shapes-old.sqlite"];

if(![fm moveItemAtURL:storeURL toURL:backupURL error:&error]) {
    NSLog(@"Error backing up the store");
    abort();
}
else if(![fm moveItemAtURL:newStoreURL toURL:storeURL error:&error]) {
    NSLog(@"Error putting the new store in place");
    abort();
}
else if(![fm removeItemAtURL:backupURL error:&error]) {
    NSLog(@"Error getting rid of the old store");
}
}

__persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:destinationModel];

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

return __persistentStoreCoordinator;
}

```

Note that the source and destination store URLs must be different so that the migration can run. Once it is successful, it is up to you to get rid of the old data store and clean up. This is done by moving the SQLite files directly using the `NSFileManager` class as shown in the implementation above. Upon returning from the method, the new and properly populated data store is in place for the application to use.

Run the application to execute the custom migration and verify that the data store contains the right data by using the `sqlite3` command line as usual.

## Summary

Although changing data models can be painful in other programming environments, Core Data makes changing your data models nearly painless. With its support for model versions, lightweight migrations, and mapping models for migrations that aren't lightweight, you can change your data models with impunity and let Core Data make sure your users' data stays intact.

As with any part of your application, however, make sure you test your migrations. Test them extensively. Test them even more than you test your application code. Users can handle an occasional application crash, but if they upgrade to the latest version of your application and lose their data in the process, your application and reputation may never recover.

# Managing Table Views Using a Fetched Results Controller

A fetched results controller, implemented by the `NSFetchedResultsController` class, blends Core Data results with a table view. Launch any nongame app on your iPhone you'll probably find at least one table view. Apple made the table view an essential part of the iPhone interface and uses it to great advantage in its own applications. Many, if not all, of the applications you develop will have table views, so anything you can learn to make using table views easier will benefit you.

In line with its legendary attention to detail, Apple includes a class with the Core Data framework that's designed to make Core Data work better with table views. Called `NSFetchedResultsController`, it eases the development tasks required to display data from a Core Data persistent store in a table view. You saw the `NSFetchedResultsController` class in the League Manager application from Chapter 3 in which the `MasterViewController` class uses an instance of `NSFetchedResultsController` to display the teams. This chapter dives deeper into how to use an `NSFetchedResultsController` and implements a new All-Player view using `NSFetchedResultsController` in League Manager.

## Understanding NSFetchedResultsController

The `NSFetchedResultsController` class works closely with `UITableView` instances to display data from a Core Data data model in a table view. It pulls managed objects from the persistent store, from the entity you specify, caching them to improve performance, and gives the data to the table view as necessary so that the table can show it. It also manages adding, removing, and moving rows in the table in response to data changes.

You create a fetched results controller with the following four parameters:

- A fetch request (NSFetchRequest instance)
- A managed object context
- A section name key path
- A cache name

These parameters are explained in the following sections.

## The Fetch Request

The fetch request is almost the same as any fetch request you've used throughout this book and in any of your Core Data development. It works with the entity in your data model that you specify and can optionally have a predicate (NSPredicate) to filter what it fetches. The one difference in this fetch request is that it must have at least one sort descriptor or your application will crash with this message:

```
'NSInvalidArgumentException', reason: 'An instance of NSFetchedResultsController requires a fetch request with sort descriptors'
```

This is because the fetched results controller works within the constraints of a table, which displays cells in a predictable order, so the fetched results controller must also have the data in a predictable order. A sort descriptor provides the mechanism required to help sort the data.

## The Managed Object Context

This is a normal managed object context that holds the managed objects. Saving the context saves all the objects in it. You typically use your application's managed object context for this parameter.

## The Section Name Key Path

Table views on iDevices are divided into sections, with some number of rows in each section. This structure is fundamental to the operation of table views. A fetched results controller is optimized to work in that environment and can divide its data into sections that correspond to the table sections. The `sectionNameKeyPath` parameter specifies a key path into your Core Data model that divides the managed objects for the fetch request's entity into these sections.

Typically, you'd make this `sectionNameKeyPath` parameter point to one of the properties of the entity this table displays. Note that if you specify a value for `sectionNameKeyPath` parameter, you also must sort your fetch by that value or your application will crash with an error. Suppose, for example, that the League Manager application had a table view that listed all the players for all the teams, grouped by team. Each team, then, would be a section in the table, and each player would occupy a row in their team's section. The

entity for the fetch request in this case would be the `Player` entity, and the section name key path would be the `team.name` property, which is the name attribute of the `Team` entity that's stored in the `Player`'s team relationship. You'd also add a `SortDescriptor` instance to the `NSFetchRequest` that points to `team.name`. You'll add such a view to the `League Manager` application in this chapter.

If your table view contains only one section (in other words, if your data doesn't divide into sections, so you want to display just a single list), you can pass `nil` for the `sectionNameKeyPath` parameter.

## The Cache Name

The cache name parameter specifies a name for the cache that the fetched results controller uses to cache the managed objects it fetches and feeds to the table. In the 3.x versions of iOS, you were encouraged to make this cache name unique across fetch results controllers, but your application would still work if you shared the cache name with other fetch results controllers. As of iOS 4.0, however, your application will not work correctly if you share cache names across fetched results controllers. Make sure your cache names are unique.

## Understanding NSFetchedResultsController Delegates

`NSFetchedResultsController` follows Apple's design patterns, using a delegate called `NSFetchedResultsControllerDelegate` to implement key methods to make the table view and the fetched results controller work together. You typically make your table view's view controller the delegate for its fetched results controller. This delegate has a method for customizing the index names if you show an index along the right side of your table view called `controller:sectionIndexTitleForSectionName:`. The delegate also has the following four methods that deal with changing the data displayed in the table:

- `controllerWillChangeContent:` is called when some content is about to change. This is a good place to tell your table view that you're about to make some changes so that it can initiate a sequence of changes through its `beginUpdates:` method. Table views use the `beginUpdates:` method to group changes that should be animated simultaneously.
- `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:` is called when a managed object changes (is added, deleted, edited, or moved). In this method, you figure out what happened, and you add, move, delete, or refresh the content of the affected table rows accordingly.
- `controller:didChangeSection:atIndex:forChangeType:` is called when a section changes (is added or deleted). In this method, you figure out whether a section was added or deleted and either insert or delete a section in the table.



- `controllerDidChangeContent:` is called when the changes are done. This is a good place to tell the table view that the changes initiated by the `beginUpdates:` method are complete by calling the `endUpdates:` method.

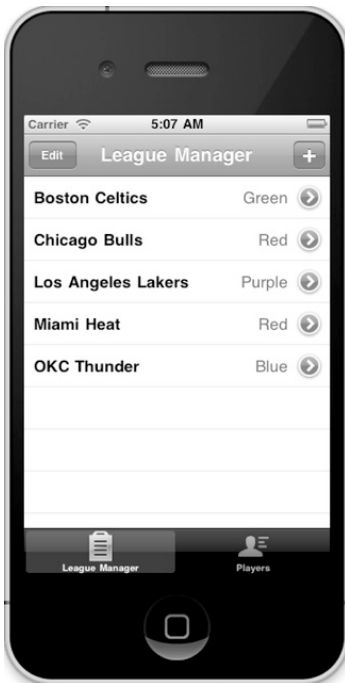
Implement these methods in your view controller, and when you create your fetched results controller, set its delegate to the view controller.

## Using NSFetchedResultsController

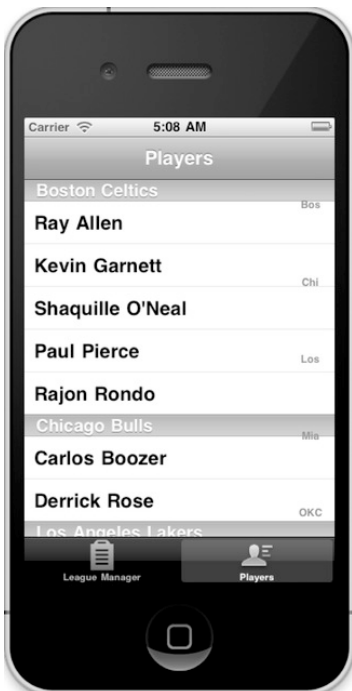
During the execution of your Core-Data–based, table-view–based application, your fetched results controller becomes the liaison between your data and the table. The fetched results controller contains methods that return the data the table needs, according to section and row. This means that whether the table needs to display a cell, figure out which cell was tapped, determine how many rows are in a section, or any of the other pieces of data a table needs to display and function, the fetched results controller has the answers.

## Implementing NSFetchedResultsController in League Manager

League Manager already uses an `NSFetchedResultsController` instance to control data display in its `MasterViewController` instance, which displays the teams. That instance of `NSFetchedResultsController` was generated for you, however, and you didn't really dive into it. To better understand `NSFetchedResultsController`, in this section you will build a new all-player view called `Players` and add it as a tab in the League Manager application. You also put the existing League Manager view in a tab. The resulting application looks like Figure 9–1, which shows the existing team view in a tab, and Figure 9–2, which shows the new all-player view in another tab.



**Figure 9-1.** *The League Manager view in a tab*



**Figure 9-2.** *The Players view in a tab*

To get started, make a copy of the League Manager application. When last you left League Manager, it was using a custom store for its persistent store. You can change it back to a SQLite store, if you'd like, but remember that Core Data abstracts the way data is stored from the rest of the application. The downloaded source code for this book leaves League Manager using the custom store.

Next, make sure League Manager builds and runs, and then change the view to use tabs. To do this, open the `League_ManagerAppDelegate.m` file and change the `application:didFinishLaunchingWithOptions:` method to create an instance of `UITabBarController`, add the existing team view controller to it (stored in the `navigationController` member), and set the application window's root view controller to your new `UITabBarController` instance, like this:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];

    MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];

    UITabBarController *tabBarController = [[UITabBarController alloc] init];
    [tabBarController setViewControllers:[NSArray
arrayWithObjects:self.navigationController, nil]];
    self.window.rootViewController = tabBarController;

    [self.window makeKeyAndVisible];
    return YES;
}
```

An image for your tab, though not required, makes your application more attractive and helps differentiate among the tabs in your application. You should have two images for each tab: one for iDevices with retina views and one for those without. The image for the retina view should be about 60 by 60 pixels, and the other should be about 30 x 30 pixels. We've selected an image that looks like a clipboard with a list, representing the teams. You can use this image, available in the downloaded source code, or create your own. In either case, add your image files as `teams.png` and `teams@2x.png` into your project in a new group called Images. To add the image to the tab, go back to the `application:didFinishLaunchingWithOptions:` method that you just edited and add this code to set the image:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];

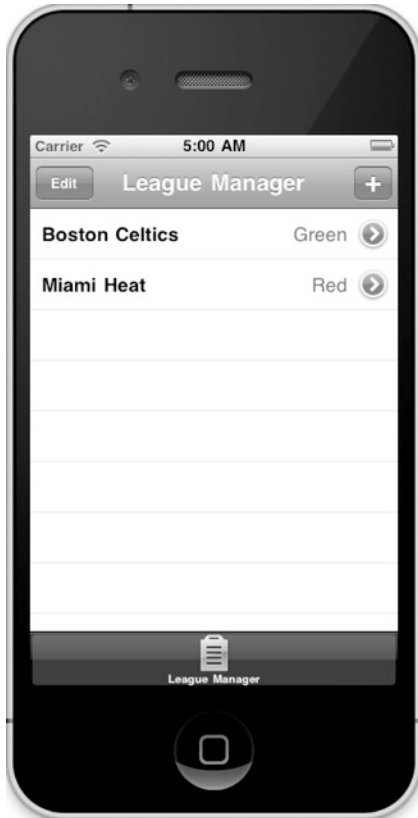
    MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];

    self.navigationController.tabBarItem.image = [UIImage imageNamed:@"teams.png"];
```

```
UITabBarController *tabBarController = [[UITabBarController alloc] init];
[tabBarController setViewControllers:[NSArray
arrayWithObjects:self.navigationController, nil]];
self.window.rootViewController = tabBarController;

[self.window makeKeyAndVisible];
return YES;
}
```

Build and run the application. Though your data may differ, you should see the single tab shown in Figure 9–3.



**Figure 9–3.** *League Manager with the teams in a tab*

Now you're ready to add the all-players view. Start by creating a new `UITableViewController` subclass without a XIB called `AllPlayerViewController`. Add images for the tab called `players.png` and `players@2x.png` (we chose an image that looks like the silhouette of a person with a list beside it, representing the list of players) to your project. Edit the `initWithStyle:` method in `AllPlayerViewController.m` to set the tab's title and image, like this:

```
- (id)initWithStyle:(UITableViewStyle)style {
    self = [super initWithStyle:style];
    if (self) {
```

```

        self.title = @"Players";
        self.tabBarItem.image = [UIImage imageNamed:@"players.png"];
    }
    return self;
}

```

Add the tab to your application by editing your `application:didFinishLaunchingWithOptions:` method. In that method, create an instance of `AllPlayerViewController` (be sure to import its header file), wrap it in an instance of `UINavigationController`, and add it to the tab bar controller's array of view controllers, like this:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];

    MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];

    self.navigationController.tabBarItem.image = [UIImage imageNamed:@"teams.png"];

    AllPlayerViewController *playersController = [[AllPlayerViewController alloc]
initWithStyle:UITableViewStylePlain];
    UINavigationController *navPlayersController = [[UINavigationController alloc]
initWithRootViewController:playersController];

    UITabBarController *tabBarController = [[UITabBarController alloc] init];
    [tabBarController setViewControllers:[NSArray
arrayWithObjects:self.navigationController, navPlayersController, nil]];

    self.window.rootViewController = tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Build and run the application. You should see a tab called `Players` that you can select to see an empty list, as shown in Figure 9–4.

With the tab and table view for the players set up, you're ready to implement the `NSFetchedResultsController` instance to display the players.



**Figure 9–4.** *League Manager with the Players tab*

## Implementing the NSFetchedResultsController

The NSFetchedResultsController's job is to fetch data from the persistent store and provide this data to the table view controller as necessary. To get started, import the Core Data headers in AllPlayerViewController.h, like so:

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

Create a property in AllPlayerViewController.h to store the NSFetchedResultsController instance. Also, create a property to store the NSManagedObjectContext from which your NSFetchedResultsController will fetch its data. These properties look like this:

```
@property (nonatomic, retain) NSFetchedResultsController *fetchedResultsController;  
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;
```

Add code to AllPlayerViewController.m to synthesize getters and setters for these properties.

```
@synthesize fetchedResultsController=__fetchedResultsController;  
@synthesize managedObjectContext=__managedObjectContext;
```

Override the accessor for `fetchedResultsController` to create and configure it. In the accessor, do the following:

1. Check to see whether it's been created and return it if it has; although the data it fetches may change, the controller itself never has to change once created and configured.
2. Create the fetch request that the controller will use and set its entity.
3. Set the batch size for the fetch request. This step isn't necessary, but can improve performance if your `NSFetchRequest` is fetching a large number of data rows. Rows will be transparently fetched in batches of the size you specify, lazy loading rows as your table and `NSFetchedResultsController` need them.
4. Set up the sorting to sort by team name, then player last name, then player first name.
5. Create the controller, initializing it with the fetch request, the managed object context, the key path for the sections, and the cache name.
6. Set the delegate for the controller. For now, you'll set it to `nil`; you'll revisit it later.
7. Fetch the data.
8. Return the newly-created controller.

The following code, which you should add to `AllPlayerViewController.m`, performs all of these steps:

```
#pragma mark - Fetched results controller

- (NSFetchedResultsController *)fetchedResultsController {
    if (__fetchedResultsController != nil) {
        return __fetchedResultsController;
    }

    // Set the entity to retrieve Players
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:[NSEntityDescription entityForName:@"Player"
        inManagedObjectContext:self.managedObjectContext]];

    // Set the batch size to a suitable number.
    [fetchRequest setFetchBatchSize:20];

    // Set up the sorting: team name, then player last name, then player first name
    NSSortDescriptor *sdTeam = [[NSSortDescriptor alloc] initWithKey:@"team.name"
        ascending:YES];
    NSSortDescriptor *sdLastName = [[NSSortDescriptor alloc] initWithKey:@"lastName"
        ascending:YES];
    NSSortDescriptor *sdFirstName = [[NSSortDescriptor alloc] initWithKey:@"firstName"
        ascending:YES];
```

```

NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sdTeam, sdLastName,
sdFirstName, nil];
[fetchRequest setSortDescriptors:sortDescriptors];

// Create the fetched results controller
NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
alloc] initWithFetchRequest:fetchRequest managedObjectContext:self.managedObjectContext
sectionNameKeyPath:@"team.name" cacheName:@"AllPlayer"];
aFetchedResultsController.delegate = nil;
self.fetchedResultsController = aFetchedResultsController;

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

```

Your NSFetchedResultsController will now be created and configured the first time it's accessed. You might wonder, though, what accesses it so that it's created, configured, and used. The answer: the table view data source delegate methods, which talk directly to the NSFetchedResultsController instance to determine what information to draw in the table, which are all in AllPlayerViewController.m (since it's the data source delegate for this table view). Let's start with the method to obtain the number of sections in the table, which should look like this:

```

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

```

The sections: method of NSFetchedResultsController returns an NSArray of objects that contain information about each section that the NSFetchedResultsController has fetched. For this method, you return the count of objects in the array.

The objects contained in the array returned by the sections: method adhere to the NSFetchedResultsSectionInfo protocol, which provides methods for extracting data necessary to properly display information in the table. You see one of those methods, name:, in the method to get the section titles.

```

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

```

This code grabs the object from the same array returned by sections: that corresponds to the section for which the title is needed, and then returns the result of the name: method. This result contains the value corresponding to the sectionNameKeyPath parameter you passed to the NSFetchedResultsController initialize—in your case, the



value for `team.name`, which should be the name of the team that the players in this section are on.

At this point, you've provided the number of sections and the title for each section to the table view, but you haven't yet provided the number of rows to display for a given section. You do this with the following method:

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

Like the previous method, this method grabs the appropriate `NSFetchedResultsController` object from the array returned by `sections:`. This time, however, it calls its `numberOfObjects:` method, which returns the number of objects this section contains. This corresponds to the number of rows in the table, since you display each object in its own table row.

The last piece of information you must provide to the table is the cell to display, which you do in the `tableView:cellForRowAtIndexPath:` method. In this method, you create or retrieve an available cell of the appropriate cell type, but you then pull the actual configuration of the cell to its own method for reasons that will be explained later in this chapter. Here's the `tableView:cellForRowAtIndexPath:` method:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"AllPlayerCell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    }
    [self configureCell:cell forIndexPath:indexPath];
    return cell;
}
```

The implementation for `configureCell:indexPath:` looks like this:

```
- (void)configureCell:(UITableViewCell *)cell forIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *managedObject = [self.fetchedResultsController
objectAtIndexPath:indexPath];
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@", [[managedObject
valueForKey:@"firstName"] description], [[managedObject valueForKey:@"lastName"]
description]];
}
```

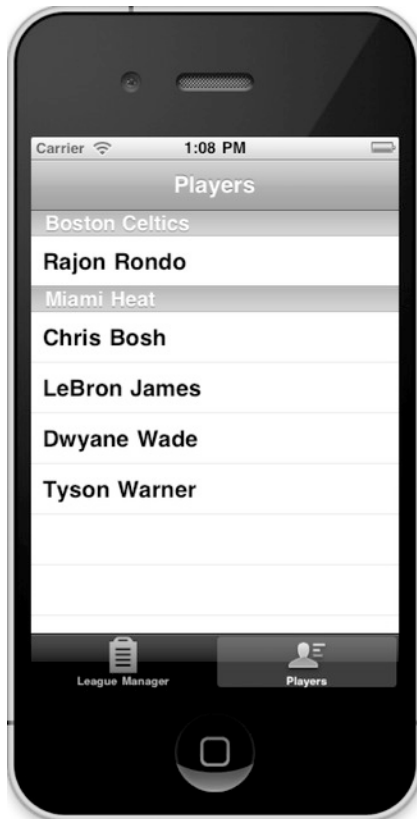
Add a category for the `configureCell:indexPath:` method at the top of `AllPlayerViewController.m`, like so;

```
@interface AllPlayerViewController ()
- (void)configureCell:(UITableViewCell *)cell forIndexPath:(NSIndexPath *)indexPath;
@end
```

Before you can run the application, however, you must set the managed object context for the `NSFetchedResultsController` instance to use. Connect the `managedObjectContext` by editing the `initWithStyle:` method:

```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        self.title = @"Players";
        self.tabBarItem.image = [UIImage imageNamed:@"players.png"];
        id delegate = [[UIApplication sharedApplication] delegate];
        self.managedObjectContext = [delegate managedObjectContext];
    }
    return self;
}
```

Now, you can build and run the application. If you tap on the Players tab, you should see all the players in your persistent store, grouped into sections by team name (see Figure 9–5).



**Figure 9–5.** *League Manager with the players listed*

## Implementing the NSFetchedResultsControllerDelegate Protocol

The NSFetchedResultsControllerDelegate protocol declares five optional methods that allow you to modify behavior or respond to change with respect to your NSFetchedResultsController instance. Four of these methods deal with changes to the underlying data, while one deals with the strings used for indices in an indexed table. Make the AllPlayerViewController your delegate for its NSFetchedResultsController, first by modifying AllPlayerViewController.h to declare that AllPlayerViewController implements the NSFetchedResultsControllerDelegate protocol, like this:

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

Then, go back to the fetchedResultsController: accessor and, instead of setting the delegate for the fetchedResultsController to nil, set it to self:

```
NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
alloc] initWithFetchRequest:fetchRequest managedObjectContext:self.managedObjectContext
sectionNameKeyPath:@"team.name" cacheName:@"AllPlayer"];
aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;
```

The AllPlayerViewController instance is now the delegate for the fetchedResultsController member, and you can implement the five delegate methods in the AllPlayerViewController class to change some behaviors. Let's look at the method that deals with indices first.

## Indexing Your Table

An iOS table view can display an index along the right side of the table. Tapping on the index allows you to jump through the table without having to scroll through all the table rows. See, for example, the Contacts app, which displays the alphabet (plus a search icon and an octothorpe) along the right side of the All Contacts view. Tap any letter in that list to jump directly to the section of contacts whose last names begin with that letter. iOS calls that list of letters an *index*, similar to an index in the back of a book that lets you look up and jump to specific pages in the book that refer to what you seek.

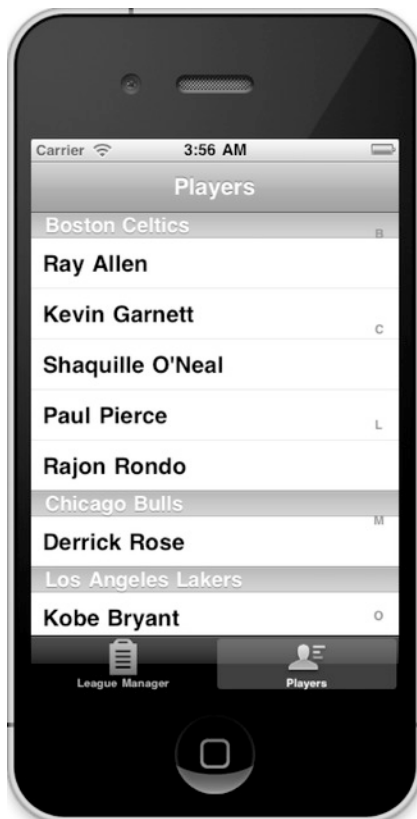
To make your table view display an index, you implement two methods from the UITableViewDataSource protocol. The first, sectionIndexTitlesForTableView:, returns an NSArray containing all the index titles that the table should display. In the case of the Contacts app, this is the letters A-Z. The second, sectionForSectionIndexTitle:atIndex:, returns the section number of the section that corresponds to the specified section index; you get both the index's title and its index to help you determine which section to return. This second method is called when the user taps the index on the right side of the table, so the app can determine where to jump to in the table.

NSFetchedResultsController provides methods to help you implement these two methods: one that returns all the section index titles, called `sectionIndexTitles:`, and one that returns the section number for the specified index title and index, called `sectionForSectionIndexTitle:atIndex:`. You'll use these methods to implement an index in the all player view. Add the following code to `AllPlayerViewController.m`:

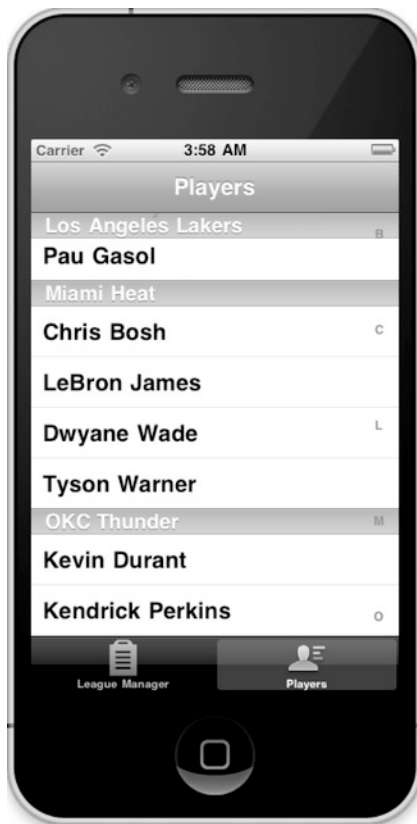
```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    return [self.fetchedResultsController sectionIndexTitles];
}

- (NSInteger)tableView:(UITableView *)tableView sectionForSectionIndexTitle:(NSString *)title atIndex:(NSInteger)index {
    return [self.fetchedResultsController sectionForSectionIndexTitle:title atIndex:index];
}
```

Run the app again to view the index along the right side of the table, as shown in Figure 9-6. The letters B-C-L-M-O along the right edge represent the index. Tap the O at the bottom to scroll directly to the “O” section, as shown in Figure 9-7.



**Figure 9-6.** The all-player view with an index



**Figure 9–7.** The all-player view after tapping on the O in the index

An `NSFetchedResultsController` instance defaults to supplying the initial letter of the section name only for the index titles, as you’ve just seen. What if, instead of showing only the first letter, you wanted to show the first three letters of a section’s name? The `NSFetchedResultsControllerDelegate` protocol has a method, added in iOS 4.0, called `controller:sectionIndexTitleForSectionName:`, that allows you to customize what shows in the index for a section. Prior to iOS 4.0, to customize what displays in the table index, you had to create your own `NSFetchedResultsController` subclass and override that method. Using the delegate method is a friendlier approach.

When the `controller:sectionIndexTitleForSectionName:` method is called, it’s passed the section name as an `NSString`, and it returns the index you want to be displayed as an `NSString`. Add the following code to `AllPlayerViewController` to use up to the first three letters of the section name as the index title:

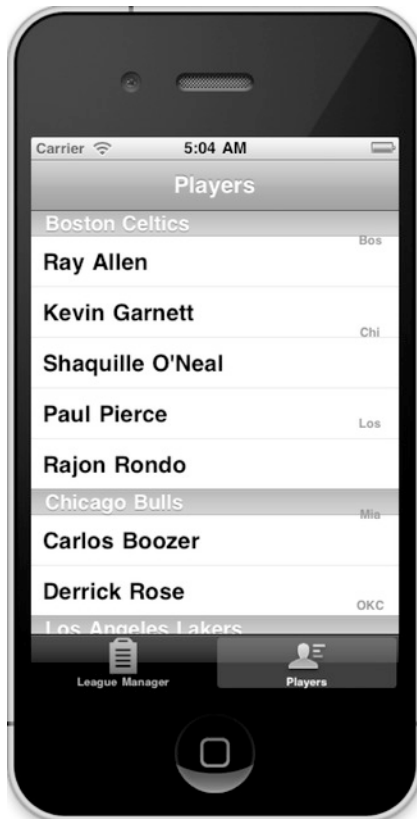
```
- (NSString *)controller:(NSFetchedResultsController *)controller
sectionIndexTitleForSectionName:(NSString *)sectionName {
    return [sectionName substringToIndex:MIN([sectionName length], 3)];
}
```

iOS seems to cache index names aggressively, however, even across invocations of your application. If you build and run your application, you'll probably still see single-letter index names, as your index names are read from cache and the method you just wrote isn't being called. To purge the cache and fix this problem, call a static method on `NSFetchedResultsController` called `deleteCacheWithName:`, passing the name of your cache. Add this call right before you fetch the results in your `fetchResultsController: accessor`, like this:

```
[NSFetchedResultsController deleteCacheWithName:@"AllPlayer"];

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

Build and run your application, and you should see the names of your indices updated to show up to the first three letters of the team names, as shown in Figure 9–8.



**Figure 9–8.** *The all-player view with three-letter index names*

## Responding to Data Change

You might notice that the all-player view doesn't respond well to change. If you add, edit, or delete players or teams in the League Manager tab, then switch over to the Players tab, you don't see any of your changes reflected. You could solve this by making the table reload its data any time the view appears, like so:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

The brute force nature of this approach contrasts, however, with the elegance of `NSFetchedResultsController`. Instead, you can implement the four methods of the `NSFetchedResultsControllerDelegate` protocol that respond to changes in the underlying model to gracefully update your views in response to data change. The first two methods bookend any data changes. The `controllerWillChangeContent:` method is called when data changes and signals that the `NSFetchedResultsController` instance is about to respond to those changes. In this method, you typically call `beginUpdates:` on your table view, so that it can perform any animations simultaneously. Then, in `controllerDidChangeContent:` (called after the `NSFetchedResultsController` has finished responding to changes), you call `endUpdates:` on your table view to signal the completion of any simultaneous animations. Add this code to `AllPlayerViewController.m`:

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

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

The other two methods in the protocol, `controller:didChangeSection:atIndex:forChangeType:` and `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`, allow you to respond to changes in sections and in rows, respectively. At the point these methods are called, your `NSFetchedResultsController` has already updated itself with the most current data and is giving you the opportunity to update your table view in response to the changed data. In your implementations, you should figure out what data changed and then instruct your table view to update itself accordingly.

When a section changes, the `controller:didChangeSection:atIndex:forChangeType:` method is called, passing in the `NSFetchedResultsSectionInfo` object representing the object that changed, the index of the section that changed, and the type of the change that occurred, represented by an `NSFetchedResultsControllerChangeType` instance. The `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:` method also uses the `NSFetchedResultsControllerChangeType` values to denote what change occurred. Table 9-1 lists the possible values for `NSFetchedResultsControllerChangeType` along with their meanings.

**Table 9–1.** *The Four NSFetchedResultsControllerChangeType Values*

| Value                                  | Description  |
|--|--|
| NSFetchedResultsControllerChangeInsert | Data has been inserted.  |
| NSFetchedResultsControllerChangeDelete | Data has been deleted.   |
| NSFetchedResultsControllerChangeMove   | Data has been moved (not valid for section changes).                 |
| NSFetchedResultsControllerChangeUpdate | Data for a section has been updated (not valid for section changes). |

Notice from Table 9–1 that only two `NSFetchedResultsControllerChangeType` values, `NSFetchedResultsControllerChangeInsert` and `NSFetchedResultsControllerChangeDelete`, are ever passed for section changes. All four values can be passed for object (or row) changes.

In your implementation, you typically determine what type of change occurred, and then you tell your table view what to do. Be aware, however, of what data your `NSFetchedResultsController` instance is fetching and what data you're using for sections. In the League Manager application, for example, the `NSFetchedResultsController` instance is fetching data from the `Player` entity and using the players' team names for the sections. The `NSFetchedResultsController` isn't watching the `Team` entity at all. This means, for example, that if you go to the League Manager tab and add a new team, which adds a new `Team` entity to the data store, the `didChangeSection:` method isn't called with an `NSFetchedResultsControllerChangeInsert` change type. In fact, it isn't called at all, as nothing related to the `NSFetchedResultsController` instance (that is, nothing related to the `Player` entity) has changed. If you add a new player to the new team, however, then the `didChangeSection:` method is called with `NSFetchedResultsControllerChangeInsert` to tell you that a new section, or team, has been added.

Likewise, if you edit a team on the League Manager tab (say, for example, you rename it), the `didChangeSection:` method isn't called. If you do anything to the players on that team, however (add one, delete one, edit one), then the `didChangeSection:` method is called twice: once with `NSFetchedResultsControllerChangeInsert` and once with `NSFetchedResultsControllerChangeDelete`, producing the net effect of updating the section with the new team name. This means that you can rename a team in the League Manager tab, then switch to the Players tab, and the section name will still display the old team name. If you go back and add, edit, or delete a player, however, the team name will automatically update. Make sure you understand these limitations and their consequences to avoid frustrations.

To respond to section changes, implement the `controller:didChangeSection:atIndex:forChangeType:` method, checking for the two valid values for the change type, and instruct your table view to either insert or delete sections as appropriate, like this:



```

- (void)controller:(NSFetchedResultsController *)controller didChangeSection:(id
<NSFetchedResultsSectionInfo>)sectionInfo atIndex:(NSUInteger)sectionIndex
forChangeType:(NSFetchedResultsChangeType)type {
    switch(type) {
        case NSFetchedResultsChangeInsert:
            [self.tableView insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeDelete:
            [self.tableView deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

```

The remaining method to implement that responds to change in your NSFetchedResultsController's underlying data is `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`, which is called when any object in your result set changes. As parameters, you're passed the object that changed, the original index path (in the table view) of the object before the change, the change type, and the new index path (in the table view). The possible values for the change type are the values listed in Table 9-1. Depending on which change type you're passed, you should update the table accordingly. The following implementation does just that:

```

- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(id)anObject
atIndexPath:(NSIndexPath *)indexPath forChangeType:(NSFetchedResultsChangeType)type
newIndexPath:(NSIndexPath *)newIndexPath {
    switch(type) {
        // Data was inserted -- insert the data into the table view
        case NSFetchedResultsChangeInsert:
            [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
withRowAnimation:UITableViewRowAnimationFade];
            break;

        // Data was deleted -- delete the data from the table view
        case NSFetchedResultsChangeDelete:
            [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
            break;

        // Data was updated (changed) -- reconfigure the cell for the data
        case NSFetchedResultsChangeUpdate:
            [self configureCell:[self.tableView cellForRowAtIndexPath:indexPath]
atIndexPath:indexPath];
            break;

        // Data was moved -- delete the data from the old location and insert the data into
        the new location
        case NSFetchedResultsChangeMove:
            [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];

```

```
        [self.tableView insertRowsAtIndexPaths:[NSArray  
arrayWithObject:newIndexPath]withRowAnimation:UITableViewRowAnimationFade];  
        break;  
    }  
}
```

Like the code for responding to section changes, this code simply updates the table view according to what happened to the underlying data. As you read through this code, you should understand why you moved table cell configuration to its own method of `configureCell:atIndexPath:`. If any of the rows in the data are updated, you call this method to update the table view cell.

Build and run the application. You'll see that as you change the data in the League Manager tab, the changes are immediately reflected in the Players tab.

## Summary

As long as the Core-Data-backed table views in your applications fit the paradigm covered by the `NSFetchedResultsController` class, using `NSFetchedResultsController` will simplify your development efforts when working with tables. Most of the code you write when working with `NSFetchedResultsController` instances doesn't change from application to application, so you can get new applications running quickly.

The batching and caching of `NSFetchedResultsController` also lets you more efficiently show the results in your table view. Rows will be loaded as your table view needs them for display, allowing your applications to consume less memory and perform better.

## Using Core Data in Advanced Applications

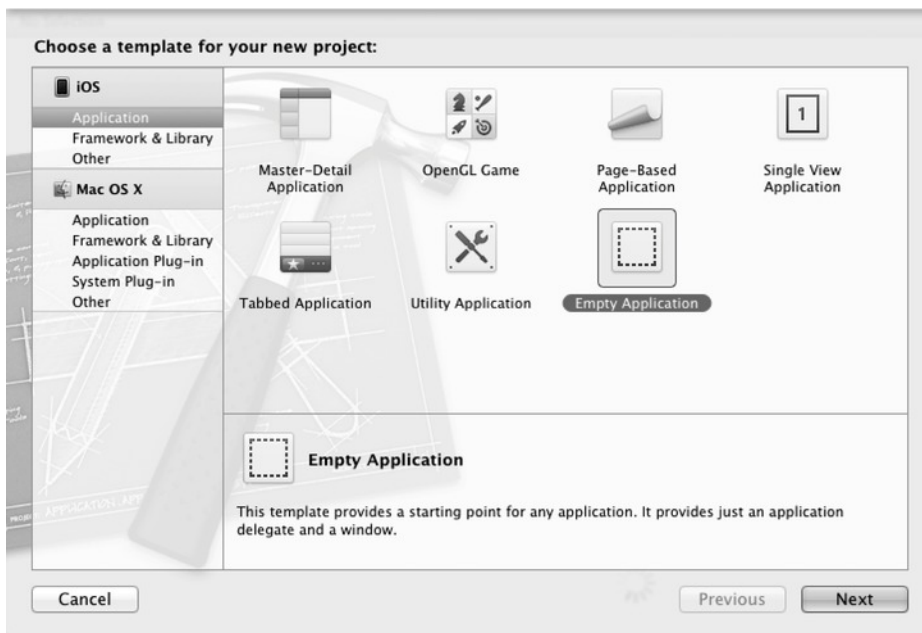
You can build a lot of cool apps without using the material in this chapter. Before you slam this book closed, however, we recommend you keep reading. This chapter contains material that addresses some advanced Core Data topics that your particular applications may require, especially if securing user data is important to your application. Follow along with this chapter to put some advanced Core Data tools in your toolbox.

In this chapter, we walk you through building an application that illustrates each of the topics we cover. You'll build the application incrementally and will be able to run the application at the end of each section so you can see that section's advanced feature working. Because each section builds on the previous ones, you shouldn't try to skip any sections.

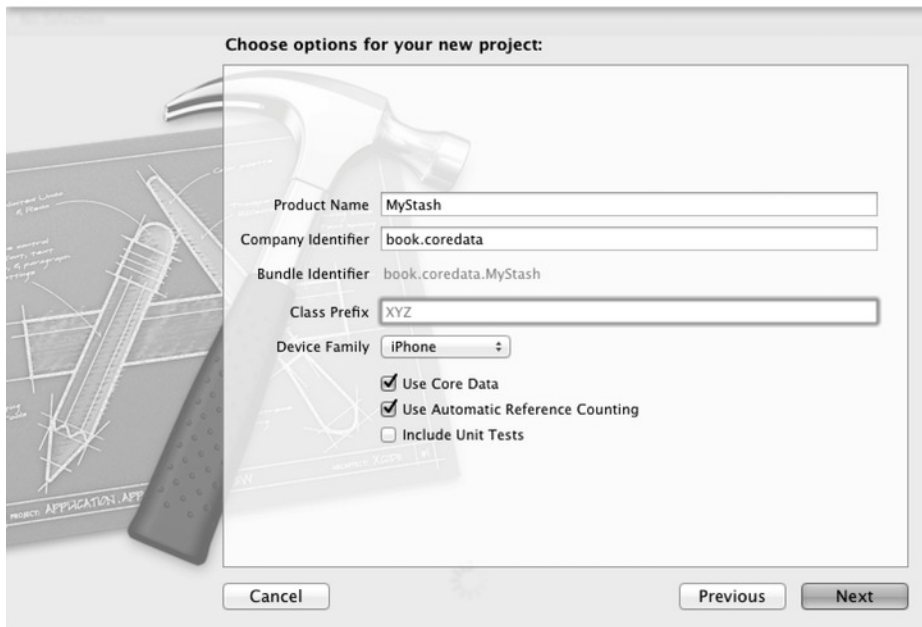
The application you build in this chapter, called MyStash, stores both passwords and notes. By the time you're done building it, the passwords will all be safely encrypted. The notes, by default, aren't encrypted, but you can choose to encrypt individual notes. The passwords and notes live in separate databases, though they share the same data model. When you're done, you can make a few tweaks to the code, add an enticing icon, and who knows? You may have an App Store hit on your hands!

### Creating an Application for Note and Password Storage and Encryption

To begin creating the application that you'll work with throughout this chapter, create a new Xcode project called MyStash. Choose the Empty Application template, as shown in Figure 10–1, and click Next. Type **MyStash** for Product Name, check Use Core Data, and set Device Family to iPhone (see Figure 10–2). Click Next, select the directory to save your project in, and click Create.



**Figure 10–1.** Selecting the *Empty Application* template



**Figure 10–2.** Setting up the *MyStash* project

You can build and run your application now to view a blank, white screen. You have neither data nor a means to view that data. You'll start by adding data to the application.

## Setting Up the Data Model

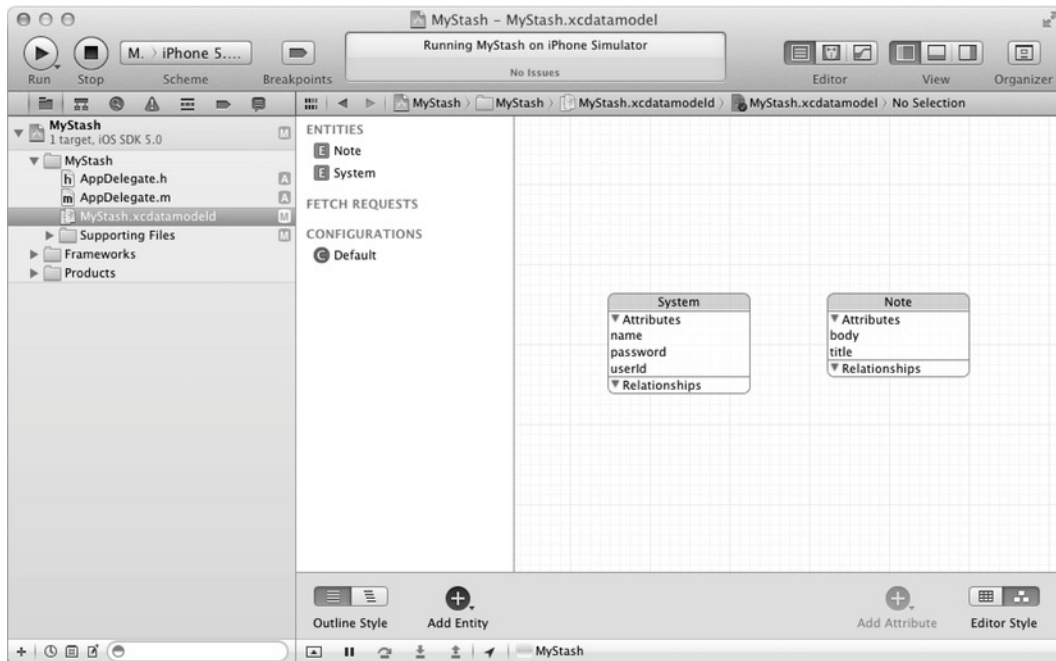
The MyStash data model will have two entities: one to hold notes and one to hold passwords for systems (such as web sites). Open the data model, and create an entity called Note and another called System. Add two attributes to the Note entity:

- title (type String)
- body (type String)

Add three attributes to the System entity:

- name (type String)
- userId (type String)
- password (type String)

Your data model should look like Figure 10–3.



**Figure 10–3.** *The MyStash data model*

Once again, you can build and run the application, but all you'll see is a blank screen. In the next section, you add a rudimentary interface. Actually adding, editing, and deleting data, however, won't happen until you piece the user interface together in the next sections in this chapter.

## Setting Up the Tab Bar Controller

The MyStash application uses two tabs to swap between Notes view and Passwords view. This means you need to add a tab bar controller to the main window. Open AppDelegate.h, and add a UITabBarController property.

```
@property (strong, nonatomic) UITabBarController *tabBarController;
```

Now, open the file AppDelegate.m, and add a @synthesize line for the tab bar controller.

```
@synthesize tabBarController = _tabBarController;
```

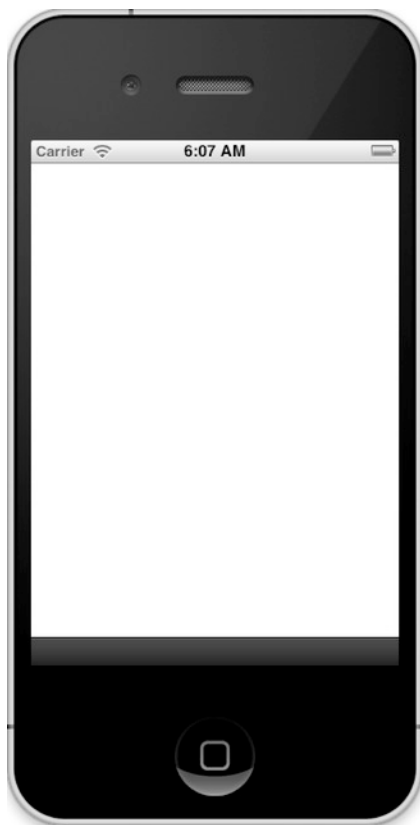
In the application:didFinishLaunchingWithOptions: method, add the tab bar controller to the main window so that method now looks like this:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];

    self.tabBarController = [[UITabBarController alloc] init];
    [self.window addSubview:self.tabBarController.view];

    [self.window makeKeyAndVisible];
    return YES;
}
```

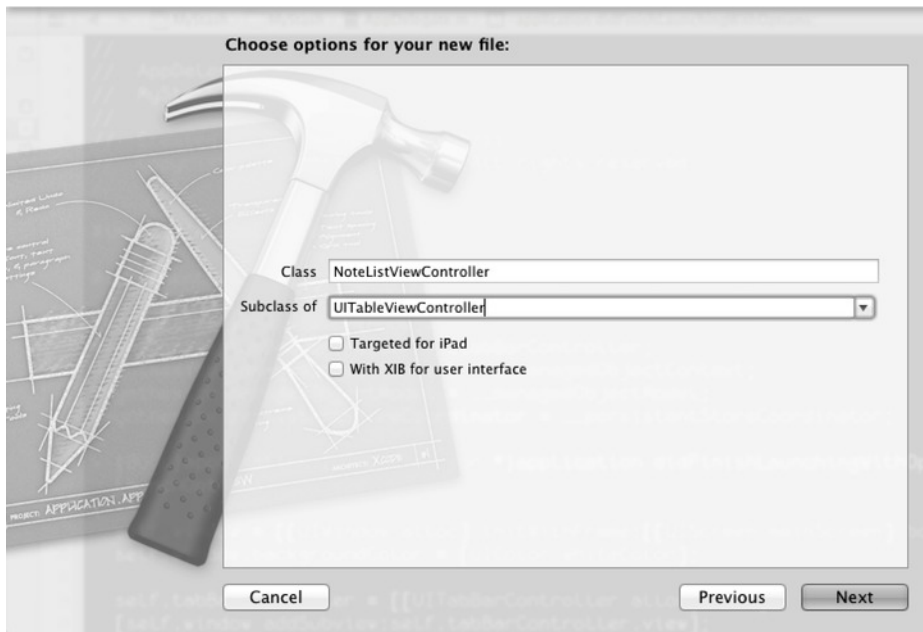
Build and run the app to see the tab bar controller, albeit small, added at the bottom of the screen, as shown in Figure 10–4.



**Figure 10–4.** *MyStash* with a tab bar

## Adding the Tab

To add the tab for the Notes view, create a new UIViewController subclass called `NoteListViewController` (see Figure 10–6). On the next screen, be sure to make it a subclass of `UITableViewController` and uncheck `With XIB for user interface`, as shown in Figure 10–5.



**Figure 10–5.** Adding the view class for the Notes view

Open `NoteListViewController.m`, find the `initWithStyle:` method, and in it, add a title and an image for the tab, like this:

```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        self.title = @"Notes";
        self.tabBarItem.image = [UIImage imageNamed:@"note"];
    }
    return self;
}
```

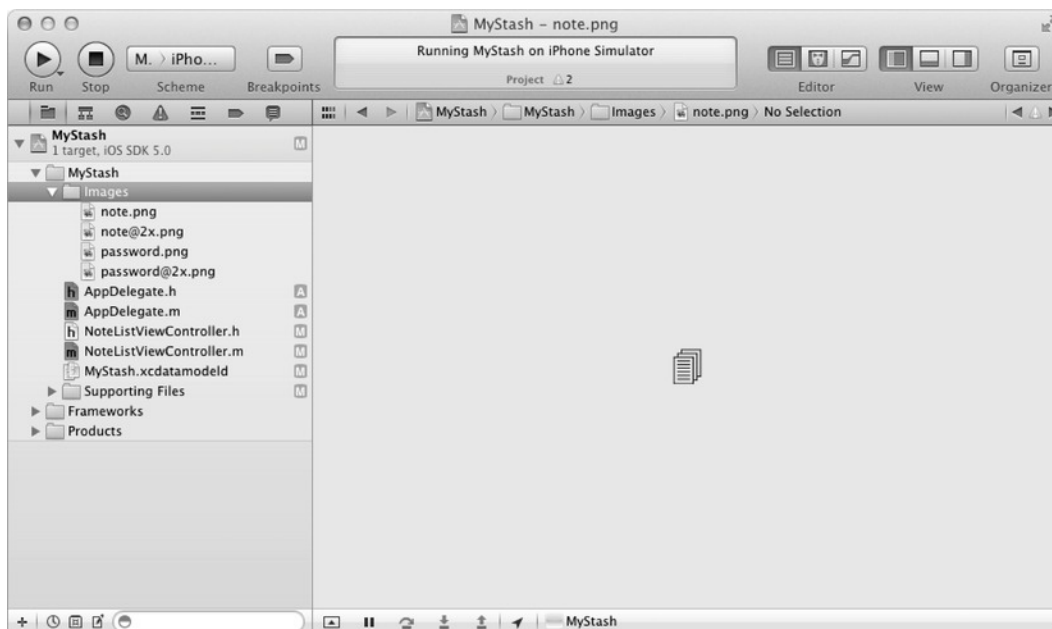
For the tab images, grab four files from the downloaded source code for this book:

- `note.png`
- `password.png`
- `note@2x.png`
- `password@2x.png`

Alternatively, you can create your own. The iPhone 4 versions (`*@2x.png`) should be about 60 pixels by 60 pixels, and the regular versions should be about 30 pixels by 30 pixels. Make them have a transparent background with a black image.

However you get the four images, add them to your project in a new group called `Images`, as in Figure 10–6.





**Figure 10–6.** The tab images added to the project

Now, add the Notes tab to the tab bar controller in the application delegate. Open `AppDelegate.m`, and add an import for the Note List view.

```
#import "NoteListViewController.h"
```

Then, in the `application:didFinishLaunchingWithOptions:` method, create an instance of `NoteListViewController`, wrap it in an instance of `UINavigationController`, and add it to the tab bar controller. The updated method looks like this:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];

    self.tabBarController = [[UITabBarController alloc] init];
    [self.window addSubview:self.tabBarController.view];

    // Add the note list tab
    NoteListViewController *noteListViewController = [[NoteListViewController alloc]
initWithStyle:UITableViewStylePlain];
    UINavigationController *navNoteList = [[UINavigationController alloc]
initWithRootViewController:noteListViewController];
    [self.tabBarController setViewControllers:[NSArray arrayWithObjects:navNoteList,
nil]];

    [self.window makeKeyAndVisible];
    return YES;
}
```

Now you can build and run the application, and you see the view has a single tab (the one for Notes), as shown in Figure 10–7.



**Figure 10–7.** *The MyStash application with a single tab*

You still need the tab for passwords, so make another view controller, patterned after the one for notes. Once again, select “UIViewController subclass,” further specialize it as a “UITableViewController subclass,” and call it `PasswordListViewController`. Open the `PasswordListViewController.m` file, find the `initWithStyle:` method, and use it to add a title and an icon for the tab.

```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        self.title = @"Passwords";
        self.tabBarItem.image = [UIImage imageNamed:@"password"];
    }
    return self;
}
```

Go back to the `AppDelegate.m` file and add an import for the password list view controller.

```
#import "PasswordListViewController.h"
```

In that same file, update the `application:didFinishLaunchingWithOptions:` method to add the Passwords tab to the tab bar controller.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];

    self.tabBarController = [[UITabBarController alloc] init];
    [self.window addSubview:self.tabBarController.view];

    // Add the note list tab
    NoteListViewController *noteListViewController = [[NoteListViewController alloc]
initWithStyle:UITableViewStylePlain];
    UINavigationController *navNoteList = [[UINavigationController alloc]
initWithRootViewController:noteListViewController];

    // Add the password list tab
    PasswordListViewController *passwordListViewController =
[[PasswordListViewController alloc] initWithStyle:UITableViewStylePlain];
    UINavigationController *navPasswordList = [[UINavigationController alloc]
initWithRootViewController:passwordListViewController];

    [self.tabBarController setViewControllers:[NSArray arrayWithObjects:navNoteList,
navPasswordList, nil]];

    [self.window makeKeyAndVisible];
    return YES;
}
```

Now, when you build and run the MyStash application, you should see two tabs: one for notes and one for passwords, as Figure 10–8 shows. You now have the bare-bones MyStash application built; you'll add the advanced features in the rest of the chapter!



**Figure 10–8.** *The MyStash application with two tabs*

## Incorporating NSFetchedResultsController into MyStash

If you run the MyStash application now, you see the expected tabs for notes and passwords. You can't add any notes or passwords, however, and you couldn't see them if you somehow added them. In the previous chapter, you've learned how to use the `NSFetchedResultsController` class to display data in a table. In this section, you apply the same technique to display notes and passwords using two instances of `NSFetchedResultsController` (one for notes and one for passwords).

### Creating the Fetched Results Controller

To create fetched results controllers to display notes and passwords, start by declaring a fetched results controller and a managed object context in `NoteListViewController.h`, shown in Listing 10–1, and in `PasswordListViewController.h`, shown in Listing 10–2. You'll notice that you also declare that both controllers implement the `NSFetchedResultsControllerDelegate` protocol.

**Listing 10–1.** *NoteListViewController.h*

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

@interface NoteListViewController : UITableViewController
<NSFetchedResultsControllerDelegate>

@property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;

@end
```

**Listing 10–2.** *PasswordListViewController.h*

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

@interface PasswordListViewController : UITableViewController
<NSFetchedResultsControllerDelegate>

@property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;

@end
```

The next step is to initialize the `NSFetchedResultsController` instances. Once again, you initialize the instances in the accessors for the `fetchedResultsController` members. Start with `NoteListViewController.m` and add `@synthesize` lines for your two new members, like this:

```
@synthesize fetchedResultsController;
@synthesize managedObjectContext;
```

Next, add the `fetchedResultsController:` accessor method shown in Listing 10–3.

**Listing 10–3.** *The fetchedResultsController: accessor*

```
#pragma mark -
#pragma mark Fetched results controller

- (NSFetchedResultsController *)fetchedResultsController
{
    if (fetchedResultsController != nil)
    {
        return fetchedResultsController;
    }

    // Create the fetch request for the entity.
    NSFetchedRequest *fetchRequest = [[NSFetchedRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Note"
    inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];

    // Set the batch size
    [fetchRequest setFetchBatchSize:20];

    // Sort by note title, ascending
```

```

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"title"
ascending:YES];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
[fetchRequest setSortDescriptors:sortDescriptors];

// Create the fetched results controller using the
// fetch request we just created, and with the managed
// object context member, and set this controller to
// be the delegate
NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
alloc] initWithFetchRequest:fetchRequest managedObjectContext:managedObjectContext
sectionNameKeyPath:nil cacheName:@"Note"];
aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;

// Fetch the results into the fetched results controller
NSError *error = nil;
if (![self fetchedResultsController] performFetch:&error)
{
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}
return fetchedResultsController;
}

```

Add the same @synthesize lines and the same method to PasswordListViewController.m, changing both the entity name and the cache name from Note to System to match the data model. Also, change the attribute used for the sort descriptor from title to name.

Your controllers can now create their fetched results controllers. The next step is to implement the NSFetchedResultsControllerDelegate protocol. Add the code in Listing 10–4 to both NoteListViewController.m and PasswordListViewController.m.

**Listing 10–4.** *The NSFetchedResultsControllerDelegate protocol methods*

```

#pragma mark -
#pragma mark Fetched results controller delegate

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

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

- (void)controller:(NSFetchedResultsController *)controller didChangeSection:(id
<NSFetchedResultsSectionInfo>)sectionInfo atIndex:(NSUInteger)sectionIndex
forChangeType:(NSFetchedResultsChangeType)type
{

```

```

switch(type)
{
    case NSFetchedResultsControllerChangeInsert:
        [self.tableView insertSections:[NSIndexSet indexSetWithIndex:sectionIndex] ↵
withRowAnimation:UITableViewRowAnimationFade];
        break;
    case NSFetchedResultsControllerChangeDelete:
        [self.tableView deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex] ↵
withRowAnimation:UITableViewRowAnimationFade];
        break;
}
}
}

```

## Incorporating the Fetched Results Controllers into the Tables

The last step is to delegate calls from the UITableViewController to the FetchedResultsController. Edit the following methods in both `NoteListViewController.m` and `PasswordListViewController.m` to match the content shown in Listing 10–5.

### Listing 10–5. Delegating calls to the FetchedResultsController

```

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

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    id <NSFetchedResultsSectionInfo> sectionInfo = [[fetchedResultsController sections]↵
objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}

- (UITableViewCell *)tableView:(UITableView *)tableView↵
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"NoteCell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault↵
reuseIdentifier:CellIdentifier];
    }
    [self configureCell:cell forIndexPath:indexPath];
    return cell;
}

- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(id)↵
anObject atIndexPath:(NSIndexPath *)indexPath↵
forChangeType:(NSFetchedResultsControllerChangeType)type newIndexPath:(NSIndexPath *)newIndexPath
{
    switch(type)

```

```

{
    case NSFetchedResultsControllerChangeInsert:
        [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath] ↵
        withRowAnimation:UITableViewRowAnimationFade];
        break;
    case NSFetchedResultsControllerChangeDelete:
        [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] ↵
        withRowAnimation:UITableViewRowAnimationFade];
        break;
    case NSFetchedResultsControllerChangeUpdate:
        [self configureCell:[self.tableView cellForRowAtIndexPath:indexPath] ↵
        atIndexPath:indexPath];
        break;
    case NSFetchedResultsControllerChangeMove:
        [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] ↵
        withRowAnimation:UITableViewRowAnimationFade];
        [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath] ↵
        withRowAnimation:UITableViewRowAnimationFade];
        break;
}
}

```

For PasswordListViewController.m, change the value for CellIdentifier to PasswordCell.

```
static NSString *CellIdentifier = @"PasswordCell";
```

It's time to finally implement the configureCell: method for each controller. Start with NoteViewController.m, and add at the top, before the line that says @implementation NoteListViewController, a private category that declares a configureCell: method on the NoteListViewController interface. That code looks like this:

```

@interface NoteListViewController()
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath;
@end

```

Inside the implementation, add an implementation of this method that configures the cell to display the note's text attribute. The interesting thing to note in this method is that you don't have to parse out the section and row from the passed index path. The fetched results controller knows how to work with an index path and uses the section and row indices it contains to return the correct managed object. Here is the method:

```

- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath
{
    NSManagedObject *note = [self.fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text = [note valueForKey:@"title"];
}

```

These lines of code are similar but different for PasswordListViewController.m. They obviously declare the configureCell: method on the PasswordListViewController interface, not the NoteListViewController interface, and the configureCell: implementation displays the name attribute of the managed object. The lines look like this:



```

@interface PasswordListViewController(private)
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath;
@end

- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath
{
    NSManagedObject *system = [self.fetchedResultsController objectAtIndex:indexPath];
    cell.textLabel.text = [system valueForKey:@"name"];
}

```

The last method in the table view data source methods that you need to update is the `tableView:CommitEditingStyle:forRowAtIndexPath:` method, which is called when a user moves or deletes a row. In the MyStash application, the users can't move either notes or passwords, but they can delete them. Add the methods in Listing 10–6 to both controllers' implementation files and declare `saveContext:` in both header files.

**Listing 10–6.** *Methods for deleting rows*

```

- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [[self.fetchedResultsController managedObjectContext]
deleteObject:[self.fetchedResultsController objectAtIndex:indexPath]];
        [self saveContext];
    }
}

- (void)saveContext
{
    NSError *error = nil;
    if(![[self.fetchedResultsController managedObjectContext] save:&error])
    {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }
}

```

Be sure to add the `saveContext:` declaration to both `NoteListViewController.h` and `PasswordListViewController.h`:

```
-(void)saveContext;
```

To finish incorporating the fetched results controllers into the MyStash application, you must provide the managed object contexts for the two view controllers. Open `AppDelegate.m`, find the `application:didFinishLaunchingWithOptions:` method, and set each controller's managed object context member right after initializing it. The two lines to add are as follows:

```

noteListViewController.managedObjectContext = self.managedObjectContext;
...
passwordListViewController.managedObjectContext = self.managedObjectContext;

```

## Creating the Interface for Adding and Editing Notes and Passwords

The MyStash application can now display and delete notes and passwords, but you can't yet add or edit any notes or passwords. This next part walks you through adding interfaces to add and edit notes and passwords. You'll create one modal for notes and a different modal for passwords, and MyStash will use the same modal for both adding and editing.

Start with the interface for adding and editing notes. Create a `UIViewController` subclass called `NoteViewController`, and make sure only the "With XIB for user interface" option is selected. This creates three files:

- `NoteViewController.h`
- `NoteViewController.m`
- `NoteViewController.xib`

Open `NoteViewController.h`. You'll add the following:

- A text field to allow users to enter the note's title.
- A Text view to hold the body of the note.
- A pointer to the parent controller so you can ask the parent to save any new note.
- A managed object that represents the current note, so the user can edit an existing note. If the current note is `nil`, the code will create a new one.
- An initializer that receives the parent controller and the note object.
- A method that responds to the user dismissing the modal with the Save button.
- A method that responds to the user dismissing the modal with the Cancel button.

`NoteListViewController.h` should match Listing 10–7.

**Listing 10–7.** *NoteListViewController.h*

```
@class NoteListViewController;

@interface NoteViewController : UIViewController
{
    UITextField *titleField;
    UITextView *body;
    NoteListViewController *parentController;
    NSManagedObject *note;
}
@property (strong, nonatomic) IBOutlet UITextField *titleField;
```

```

@property (strong, nonatomic) IBOutlet UITextView *body;
@property (strong, nonatomic) NoteListViewController *parentController;
@property (strong, nonatomic) NSManagedObject *note;

- (id)initWithParentController:(NoteListViewController *)parentController
note:(NSManagedObject *)note;
- (IBAction)save:(id)sender;
- (IBAction)cancel:(id)sender;

@end

```

The implementation file for the Note view, `NoteViewController.m`, is shown next, in Listing 10–8. Note the following:

- The initialization method stores the parent `NoteListViewController` instance and the specified `NSManagedObject` instance.
- When the view loads and the `viewDidLoad:` method is called, the code checks whether the note managed object is `nil`. If it is, the code sets the body text to “Type text here . . .” so that users know what to do. If it’s not `nil`, the code takes the values from the managed object and puts them into the user interface fields.
- The `save:` method checks again if it’s editing an existing note or adding a new one. If it’s editing an existing note, it updates the note managed object with the values the user entered and asks the parent controller to save the context. If it’s creating a new note, it asks the parent controller to insert a new note object, passing the user-entered values. Either way, it then dismisses the modal.
- The `cancel:` method simply dismisses the modal, discarding any user input.

**Listing 10–8.** *NoteViewController.m*

```

#import "NoteViewController.h"
#import "NoteListViewController.h"

@implementation NoteViewController

@synthesize titleField, body, parentController, note;

- (id)initWithParentController:(NoteListViewController *)parentController_
note:(NSManagedObject *)note_
{
    if ((self = [super init]))
    {
        self.parentController = parentController_;
        self.note = note_;
    }
    return self;
}

- (void)viewDidLoad

```

```

{
    [super viewDidLoad];
    if (note != nil)
    {
        titleField.text = [note valueForKey:@"title"];
        body.text = [note valueForKey:@"body"];
    }
    else
    {
        body.text = @"Type text here . . .";
    }
}

- (void)viewDidLoad
{
    self.titleField = nil;
    self.body = nil;
    [super viewDidLoad];
}

- (IBAction)save:(id)sender
{
    if (parentController != nil)
    {
        if (note != nil)
        {
            [note setValue:titleField.text forKey:@"title"];
            [note setValue:body.text forKey:@"body"];
            [parentController saveContext];
        }
        else
        {
            [parentController insertNoteWithTitle:titleField.text body:body.text];
        }
    }
    [self dismissModalViewControllerAnimated:YES];
}

- (IBAction)cancel:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
}

@end

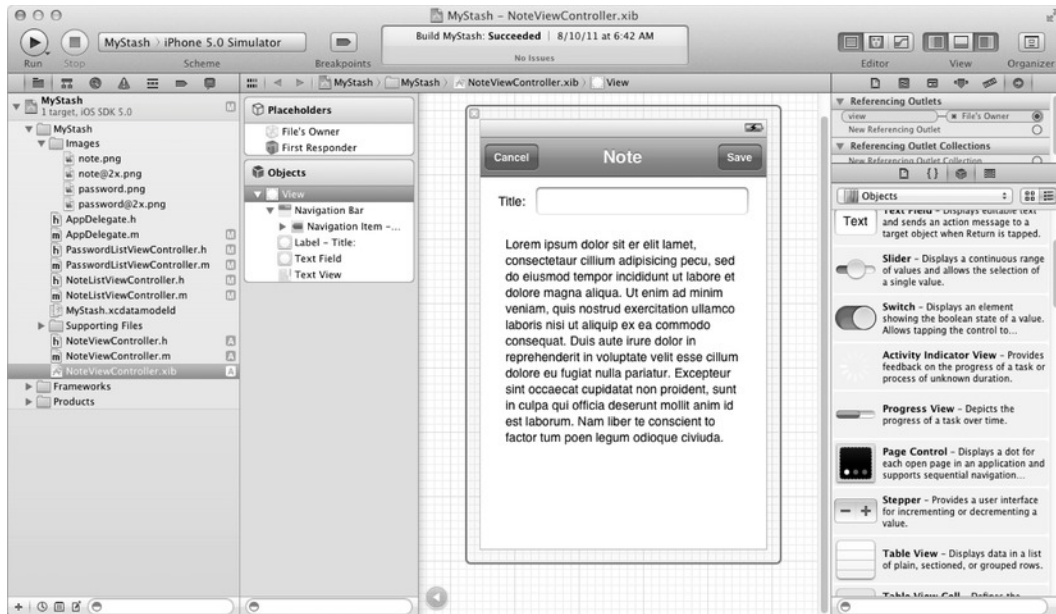
```

Note that you haven't yet implemented the `insertNoteWithTitle:body:` method in `NoteListViewController`, so this file won't yet compile.

Next, you must create the actual user interface that the `NoteViewController` instance controls. Select the `NoteViewController.xib` file to open it in Interface Builder. Perform the following steps:

1. Drag a Navigation Bar onto the view, and align it with the top edge of the view. Change its title to `Note`.
2. Drag two Bar Button Items onto the navigation bar—one on the left and one on the right.
3. Change the labels on the two Bar Button items you just added—the one on the left to `Cancel` and the one on the right to `Save`.
4. Ctrl-drag from the `Cancel` Bar Button Item to the File's Owner icon and select `cancel:` from the pop-up menu.
5. Ctrl-drag from the `Save` Bar Button Item to the File's Owner icon and select `save:` from the pop-up menu.
6. Drag a Label onto the top left of the view, below the navigation bar, and change its text to `Title:`.
7. Drag a Text Field onto the view, to the right of the `Title:` label.
8. Drag a Text View onto the view, below the `Title:` label, and make it fill the rest of the view.
9. Ctrl-drag from the File's Owner icon to the text field, and select `textField` from the pop-up menu.
10. Ctrl-drag from the File's Owner icon to the Text view, and select `body` from the pop-up menu.

Figure 10–9 shows what your view should look like when you're done. Make sure you follow all the previous steps, including wiring the buttons to the action methods you created (`save:` and `cancel:`) and the user interface components to the appropriate variables.



**Figure 10–9.** Setting up the single note view

All that's left for the MyStash application to be able to add and edit notes is to display this modal in response to two user actions from the Note List view.

1. Tap the + button to add a new note.
2. Tap the note's title in the list to edit that note.

Open `NoteListViewController.h`, and add the following two method declarations: one to show the modal when the user clicks the + button and one to insert a new note:

```
- (void)showNoteView;
- (void)insertNoteWithTitle:(NSString *)title body:(NSString *)body;
```

Open `NoteListViewController.m`, import `NoteViewController.h`, and add definitions for those two methods. The first, `showNoteView:`, allocates an instance of `NoteViewController` and initializes it with the parent view controller and a nil note so that a new note will be created. The second, `insertNoteWithTitle:`, is the method that `NoteViewController` calls when the user saves a new note. Listing 10–9 shows these two methods.

**Listing 10–9.** Methods for creating a note

```
- (void)showNoteView
{
    NoteViewController *noteViewController = [[NoteViewController alloc] ←
initWithParentController:self note:nil];
    [self presentViewController:noteViewController animated:YES];
}

- (void)insertNoteWithTitle:(NSString *)title body:(NSString *)body
```

```

{
    NSManagedObjectContext *context = [fetchResultsController managedObjectContext];
    NSEntityDescription *entity = [[fetchResultsController fetchRequest] entity];
    NSManagedObject *newNote = [NSEntityDescription
insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];

    [newNote setValue:title forKey:@"title"];
    [newNote setValue:body forKey:@"body"];

    [self saveContext];
}

```

Now, you need to add the + button and the Edit button to the note list interface. Go to the `viewDidLoad:` method, and add the Edit button on the left and the + button on the right. Wire the + button to the `showNoteView` action. Listing 10–10 shows the code.

**Listing 10–10.** *Adding buttons in `viewDidLoad`:*

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Set up the edit and add buttons.
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(showNoteView)];
    self.navigationItem.rightBarButtonItem = addButton;
}

```

All that remains is to add support for editing a note. When the user taps an existing note, the `tableView:didSelectRowAtIndexPath:` method is called. In that method, grab the note that corresponds to the `indexPath` parameter, and then allocate and initialize a `NoteViewController` instance. This time, you pass the note managed object so that the existing note is edited. The method is shown in Listing 10–11.

**Listing 10–11.** *Displaying a note when tapped*

```

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath*)indexPath
{
    NSManagedObject *note = [[self fetchResultsController] objectAtIndexPath:indexPath];

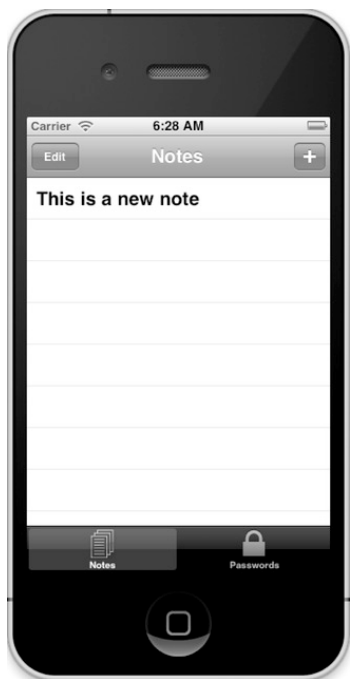
    NoteViewController *noteViewController = [[NoteViewController alloc]
initWithParentController:self note:note];
    [self presentViewController:noteViewController animated:YES];
}

```

Build and run the application. You will see a blank screen, as before, but now you can click the + button to add a note. If you do, you'll see the screen shown in Figure 10–10. Type in a note, and click Save. Your new note should appear in the List view, as Figure 10–11 shows.



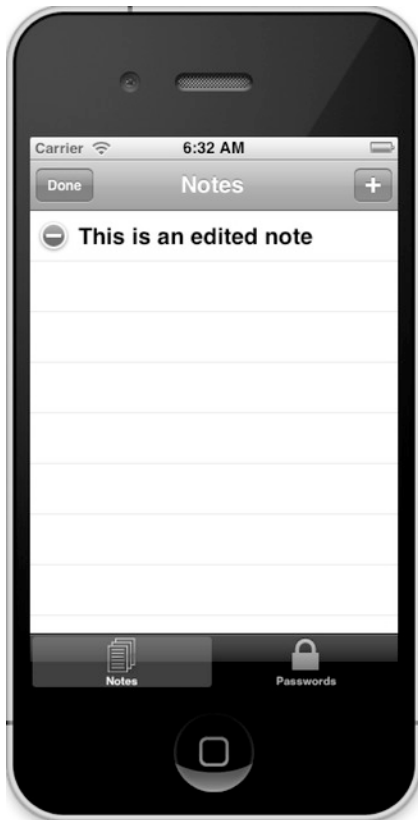
**Figure 10-10.** *A new blank note*



**Figure 10-11.** *The Note List view with a note added*



You can tap the note, and it will appear for you to edit. You can edit it and save the changes, or you can cancel and no changes will be saved. From the Notes List view, you can tap the Edit button to delete any notes, as Figure 10–12 shows. Tap the Done button to leave edit mode.



**Figure 10–12.** *The Notes List View in edit mode*

To complete this phase of the MyStash application, you must create the interface for adding and editing passwords. Again, create a new `UIViewController` subclass with an XIB for the user interface and call it `PasswordViewController`. The code looks very much like the code for the `NoteViewController`, but with different fields that are appropriate to the `System` entity in the data model. Listing 10–12 shows the code for `PasswordViewController.h`:

**Listing 10–12.** *PasswordViewController.h*

```
@class PasswordListViewController;

@interface PasswordViewController : UIViewController
{
    UITextField *name;
    UITextField *userId;
    UITextField *password;
}
```

```

    PasswordListViewController *parentController;
    NSManagedObject *system;
}
@property (strong, nonatomic) IBOutlet UITextField *name;
@property (strong, nonatomic) IBOutlet UITextField *userId;
@property (strong, nonatomic) IBOutlet UITextField *password;
@property (strong, nonatomic) PasswordListViewController *parentController;
@property (strong, nonatomic) NSManagedObject *system;

- (id)initWithParentController:(PasswordListViewController *)parentController
system:(NSManagedObject *)system;
- (IBAction)save:(id)sender;
- (IBAction)cancel:(id)sender;

@end

```

Listing 10–13 contains the code for PasswordViewController.m.

**Listing 10–13.** *PasswordViewController.m*

```

#import "PasswordViewController.h"
#import "PasswordListViewController.h"

@implementation PasswordViewController

@synthesize name, userId, password, parentController, system;

- (id)initWithParentController:(PasswordListViewController *)parentController_
system:(NSManagedObject *)system_ {
    if ((self = [super init]))
    {
        self.parentController = parentController_;
        self.system = system_;
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    if (system != nil)
    {
        name.text = [system valueForKey:@"name"];
        userId.text = [system valueForKey:@"userId"];
        password.text = [system valueForKey:@"password"];
    }
}

- (void)viewDidUnload
{
    self.name = nil;
    self.userId = nil;
    self.password = nil;
    [super viewDidUnload];
}

```

```

- (IBAction)save:(id)sender
{
    if (parentController != nil)
    {
        if (system != nil)
        {
            [system setValue:name.text forKey:@"name"];
            [system setValue:userId.text forKey:@"userId"];
            [system setValue:password.text forKey:@"password"];
            [parentController saveContext];
        }
        else
        {
            [parentController insertPasswordWithName:name.text userId:userId.text
password:password.text];
        }
    }
    [self dismissModalViewControllerAnimated:YES];
}

- (IBAction)cancel:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
}

@end

```

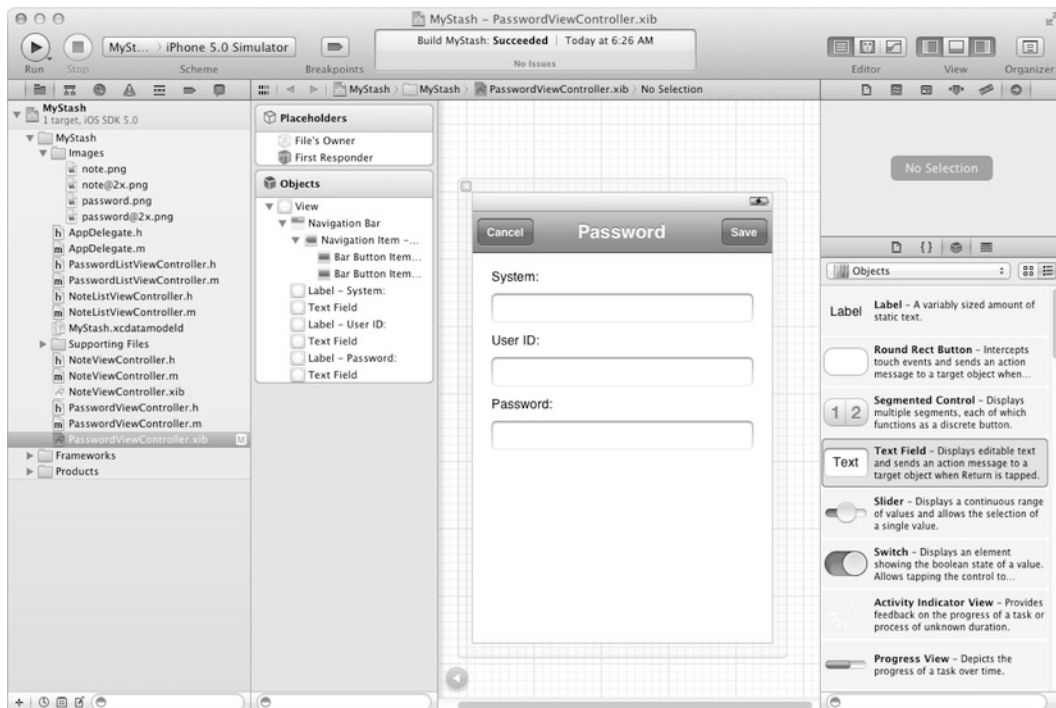
Again, this won't compile, because you haven't implemented the `insertPasswordWithName:userId:password:` method in `PasswordListViewController` yet.

The user interface for the Password view looks similar to, but different from, the Note view. The steps to follow, then, are similar to the ones you followed for the Note view. Open `PasswordViewController.xib` and follow these steps:

1. Drag a Navigation Bar onto the view, and align it with the top edge of the view. Change its title to Password.
2. Drag two Bar Button Items onto the navigation bar—one on the left and one on the right.
3. Change the labels on the two Bar Button items you just added—the one on the left to Cancel and the one on the right to Save.
4. Ctrl+drag from the Cancel Bar Button Item to the File's Owner icon and select `cancel:` from the pop-up menu.
5. Ctrl+drag from the Save Bar Button Item to the File's Owner icon, and select `save:` from the pop-up menu.
6. Drag a Label onto the top left of the view, below the navigation bar, and change its text to System:.

7. Drag a Text Field onto the view, below the System: label, and make it span the width of the view.
8. Drag another Label onto the view, below the text field you just added, and change its text to User ID:.
9. Drag a Text Field onto the view, below the User ID: label, and make it span the width of the view.
10. Drag another Label onto the view, below the text field you just added, and change its text to Password:.
11. Drag a Text Field onto the view, below the Password: label, and make it span the width of the view.
12. Ctrl+drag from the File's Owner icon to the text field below System: , and select name from the pop-up menu.
13. Ctrl+drag from the File's Owner icon to the text field below User ID:, and select userId from the pop-up menu.
14. Ctrl+drag from the File's Owner icon to the text field below Password:, and select password from the pop-up menu.

When you finish, the view should look like Figure 10–13.



**Figure 10–13.** The configured Password view

The remaining steps are to update the `PasswordListViewController` class, as you did `NoteListViewController`, to show the password modal when adding or editing a password. Open `PasswordListViewController.h`, and add these two method declarations:

```
- (void)showPasswordView;
- (void)insertPasswordWithName:(NSString *)name userId:(NSString *)userId
password:(NSString *)password;
```

In `PasswordListViewController.m`, import `PasswordViewController.h`, and add the code in Listing 10–14.

**Listing 10–14.** *Methods for adding and editing a password*

```
- (void)showPasswordView
{
    PasswordViewController *passwordViewController = [[PasswordViewController alloc]
initWithParentController:self system:nil];
    [self presentViewController:passwordViewController animated:YES];
}

- (void)insertPasswordWithName:(NSString *)name userId:(NSString *)userId
password:(NSString *)password
{
    NSManagedObjectContext *context = [fetchResultsController managedObjectContext];
    NSEntityDescription *entity = [[fetchResultsController fetchRequest] entity];
    NSManagedObject *newPassword = [NSEntityDescription
insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];

    [newPassword setValue:name forKey:@"name"];
    [newPassword setValue:userId forKey:@"userId"];
    [newPassword setValue:password forKey:@"password"];

    [self saveContext];
}
```

Change the `viewDidLoad:` method to match Listing 10–15.

**Listing 10–15.** *The updated `viewDidLoad:` method*

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

    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(showPasswordView)];
    self.navigationItem.rightBarButtonItem = addButton;
}
```

Finally, update the `tableView:didSelectRowAtIndexPath:` method to match the code in Listing 10–16.

**Listing 10–16.** *The updated tableView:didSelectRowAtIndexPath: method*

```

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSManagedObject *system = [[self fetchedResultsController] ←
objectAtIndexPath:indexPath];

    PasswordViewController *passwordViewController = [[PasswordViewController alloc] ←
initWithParentController:self system:system];
    [self presentViewController:passwordViewController animated:YES];
}

```

This phase of the MyStash application is now complete. You can add, edit, and delete notes and passwords. Build and run the application now and add a new password. When you tap the + button on the Passwords tab, you should see the modal for adding a new password, as Figure 10–14 shows. Add a few passwords, and you should see them appear in the List view, as depicted in Figure 10–15.



**Figure 10–14.** *Entering a new password*



**Figure 10–15.** *The Password List view with three entries*

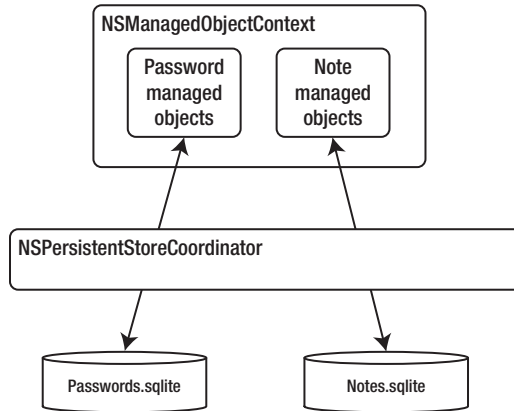
All the notes and passwords you create in the MyStash application live in the same persistent store. Read the next section to understand how to spread this data across multiple persistent stores.

## Splitting Data Across Multiple Persistent Stores

Up to this point in the book, you have always associated a data model with a single persistent store. You may have cases, however, that would benefit from splitting the model into multiple stores. When multiple persistent stores are involved, Core Data is able to dispatch the entities to persist to the proper store and rebuild object graphs spawning across multiple stores. The role of the `NSPersistentStoreCoordinator` is to manage the multiple stores and make them appear to the Core Data API user as if they were a single store.

The MyStash data model contains both notes and passwords. When you run the MyStash application, all notes and passwords are stored in the `MyStash.sqlite` persistent store. Suppose, though, that you want to split the persistent store into two so that notes are stored in the `Notes.sqlite` persistent store and passwords go into the

Passwords.sqlite persistent store. Figure 10–16 illustrates the split, showing where each type of managed object is persisted.



**Figure 10–16.** *Persistent store split*

## Using Model Configurations

A Core Data model defines entities and their relationships in order to specify how data should be laid out, but it can also stipulate where the data should be written. To specify where data should be written, Core Data uses what’s called *configurations*. Each entity can be associated with one or more configurations, and when a persistent store is created for a configuration, only the entities linked to that configuration are included in the persistence operation. In your example, you’ll create two configurations. The first one will contain notes, and the second will store passwords.

In Xcode, open the `MyStash.xcdatamodel` model. Select the default configuration (below the list of Entities). It displays all the entities associated with that configuration, as shown in Figure 10–17. Initially, only the default configuration exists, which contains both the Note and the System entities. Click and hold the Add Entity button at the bottom of the Xcode window, select Add Configuration from the menu, and add a configuration called Notes. Repeat the process to create another configuration called Passwords.

The Notes configuration currently has no associated entities. To add the Note entity to the associated list of entities, select the Notes entity, and then drag it to the Entities list for that configuration, as shown in Figure 10–18. To add the System entity to the Passwords configuration, drag the System entity to the Passwords configuration, as shown in Figure 10–19.





Figure 10-17. The default configuration



Figure 10-18. Adding the Note entity to the Notes configuration



**Figure 10–19.** The *System* entity added to the *Passwords* configuration

You have now prepared your model for the persistent store split. All that is left to do is to create the persistent stores using the new configurations. For this, open `AppDelegate.m`, and change the `persistentStoreCoordinator:` method to match Listing 10–17.

**Listing 10–17.** The updated `persistentStoreCoordinator:` accessor

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

    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];

    {
        // Create the Notes persistent store
        NSURL *noteStoreURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Notes.sqlite"];
        NSError *error = nil;
        if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:@"Notes" URL:noteStoreURL options:nil error:&error])
        {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }

    {
        // Create the Passwords persistent store
```

```

        NSURL *passwordStoreURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Passwords.sqlite"];
        NSError *error = nil;
        if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:@"Passwords" URL:passwordStoreURL options:nil error:&error])
        {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }

    return __persistentStoreCoordinator;
}

```

Instead of creating a single store, you created two persistent stores in an almost identical manner. The only difference is that you explicitly named the configuration for the store to use, instead of passing `nil` for the configuration parameter. For `Passwords.sqlite`, you use the `Passwords` configuration, while the `Notes.sqlite` persistent store uses the `Notes` configuration.

This is a good time to take a deserved break and launch the application. Add a new note as well as a new password entry to verify that it works. To your dismay, you don't notice any difference. But rejoice because this is exactly what you were aiming for: a change in the underlying persistent store structure shouldn't affect the rest of the MyStash application. Be content that the app still works even though you've completely changed how data is stored. To see the difference, you need to pop the hood open and take a look at the SQLite databases. Open a Terminal window, and go find your data stores using the following command:

```
find ~/Library/Application\ Support/iPhone\ Simulator/ -name "*.sqlite"
```

You will see your two new databases listed there, `Notes.sqlite` and `Passwords.sqlite`. If you open the `Notes.sqlite` database and run the `.schema` command, you will see the entire model, but although you can see that the note has been saved correctly, the `System` table does not contain any data.

```

sqlite> .schema
CREATE TABLE ZNOTE ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZBODY
VARCHAR, ZTITLE VARCHAR );
CREATE TABLE ZSYSTEM ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZNAME
VARCHAR, ZPASSWORD VARCHAR, ZUSERID VARCHAR );
CREATE TABLE Z_METADATA ( Z_VERSION INTEGER PRIMARY KEY, Z_UUID VARCHAR(255), Z_PLIST
BLOB);
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY, Z_NAME VARCHAR, Z_SUPER INTEGER,
Z_MAX INTEGER);
sqlite> select * from ZNOTE;
1|1|1|This is the note's body|This is a note
sqlite> select * from ZSYSTEM;
sqlite>

```

Now do the same procedure on the `Passwords.sqlite` database. The opposite is true: the table for the `System` entity is populated but not the `Note`.

```
sqlite> .schema
```

```
CREATE TABLE ZNOTE ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZBODY
VARCHAR, ZTITLE VARCHAR );
CREATE TABLE ZSYSTEM ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZNAME
VARCHAR, ZPASSWORD VARCHAR, ZUSERID VARCHAR );
CREATE TABLE Z_METADATA (Z_VERSION INTEGER PRIMARY KEY, Z_UUID VARCHAR(255), Z_PLIST
BLOB);
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY, Z_NAME VARCHAR, Z_SUPER INTEGER,
Z_MAX INTEGER);
sqlite> select * from ZNOTE;
sqlite> select * from ZSYSTEM;
1|2|1|My Secure System|mypassword|myuserid
sqlite>
```

You have successfully split your data model into two persistent stores. The next section explains a situation in which this might be useful.

## Adding Encryption

Data encryption is a big deal for any application that requires storage of sensitive information. Core Data does not do much to help you with this task, but it doesn't stand in your way, either. Securing data follows two schools of thought, and you may choose to adhere to either one based on your own needs and requirements. These two schools are either to encrypt the entire database file or to just encrypt selected fields within your data store. The next sections explain both options.

## Persistent Store Encryption Using Data Protection

One of the security options that was deployed with the iPhone 3GS is hardware-level disk encryption. Apple calls it *data protection*. When enabled, it encrypts a portion of your disk when the device is locked and is automatically decrypted when the device is unlocked.

For data protection to work, it must be manually enabled. Keep in mind that this is hardware-level disk encryption and therefore it works only on the device itself, not on the iPhone Simulator. To enable it, open the iPhone settings, and navigate to **Settings** ► **General** ► **Passcode Lock**. Enable the passcode if not enabled yet. The bottom of the page will display “Data protection is enabled,” as illustrated in Figure 10–20, if everything is properly set up.



**Figure 10–20.** Enabling data protection on the device

From a programming standpoint, the work involved in encrypting your database is surprisingly simple. When creating the persistent store, you simply need to set the right attribute on the database file and let iOS do the rest. Open `AppDelegate.m`, and change the implementation of the `persistentStoreCoordinator:` to add the proper file attribute to encrypt the password database, as shown in Listing 10–18.

**Listing 10–18.** Encrypting the password database

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    ...
    {
        // Create the Passwords persistent store
        NSURL *passwordStoreURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Passwords.sqlite"];
        NSError *error = nil;
        if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:@"Passwords" URL:passwordStoreURL options:nil error:&error])
        {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }

        // Encrypt the password database
        NSDictionary *fileAttributes = [NSDictionary
dictionaryWithObject:NSFileProtectionComplete forKey:NSFileProtectionKey];
        if (![NSFileManager defaultManager] setAttributes:fileAttributes
ofItemAtPath:[passwordStoreURL path] error:&error])
    }
}
```

```
{
    NSLog(@"Unresolved error with password store encryption %@, %@", error, [error↵
userInfo]);
    abort();
}
}
...
}
```

When you run your application on a device that has data protection enabled, the `Passwords.sqlite` database file will be automatically encrypted when the device is locked. This method is cheap to implement, but understand that once the device is unlocked, the file is automatically decrypted so the database is as secure as the phone. If someone can guess the passcode, the database will be accessible, especially if the phone is jailbroken (that is, root access is available). Another inconvenience with this method is that for very large databases, the time it takes to unlock and start your application might be rather large since a huge file will need to be decrypted before it can be used.

## Data Encryption

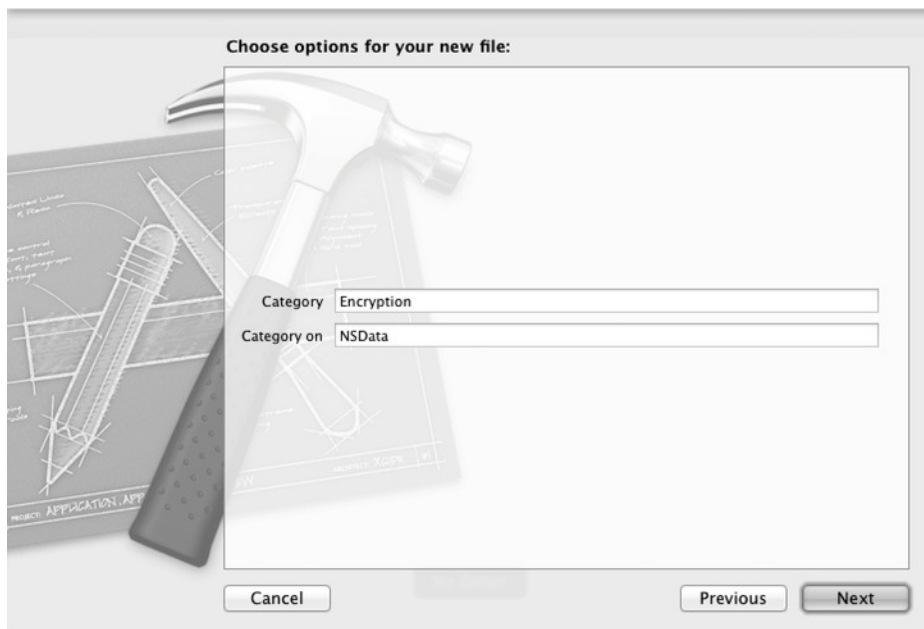
The other alternative to data security is to encrypt the data as you put it in the database. This is a case when you encrypt only what you want to secure. Let's assume that you want to be able to secure notes in the database.

## Using Encryption

iOS comes with an encryption library. To make encryption easier, you add a new category to the `NSData` class. In Xcode, create a new Objective-C category, as shown in Figure 10–21. Enter **Encryption** for Category and **NSData** for Category on, as shown in Figure 10–22. This creates a category called `NSData+Encryption` and adds it to your project. Since encryption algorithms lie outside the scope of this book, we don't dig into how this category works, but simply give you the source in Listing 10–18 (`NSData+Encryption.h`) and Listing 10–19 (`NSData+Encryption.m`).



**Figure 10–21.** Creating an Objective-C category



**Figure 10–22.** Creating the NSData+Encryption category

**Listing 10–18.** *NSData+Encryption.h*

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

@interface NSData (Encryption)
- (NSData *)encryptWithKey:(NSString *)key;
- (NSData *)decryptWithKey:(NSString *)key;
@end
```

**Listing 10–19.** *NSData+Encryption.m*

```
#import <CommonCrypto/CommonCryptor.h>
#import "NSData+Encryption.h"

@implementation NSData (Encryption)

- (NSData *)transpose:(NSString *)_key forOperation:(int)operation
{
    // Make sure the key is big enough or else add zeros
    char key[kCCKeySizeAES256+1];
    bzero(key, sizeof(key));

    // Populate the key into the character array
    [_key getCString:key maxLength:sizeof(key) encoding:NSUTF8StringEncoding];

    size_t allocatedSize = self.length + kCCBlockSizeAES128;
    void *output = malloc(allocatedSize);

    size_t actualSize = 0;
    CCCryptorStatus resultCode = CCCrypt(operation, kCCAlgorithmAES128,
    kCCOptionPKCS7Padding, key, kCCKeySizeAES256, nil, self.bytes, self.length, output,
    allocatedSize, &actualSize);
    if (resultCode != kCCSuccess)
    {
        // Free the output buffer
        free(output);
        return nil;
    }

    return [NSData dataWithBytesNoCopy:output length:actualSize];
}

- (NSData *)encryptWithKey:(NSString *)key
{
    return [self transpose:key forOperation:kCCEncrypt];
}

- (NSData *)decryptWithKey:(NSString *)key
{
    return [self transpose:key forOperation:kCCDecrypt];
}

@end
```

This category adds two methods to the `NSData` class that will allow you to obtain an encrypted and decrypted version of its binary content.



## Automatically Encrypting Fields

To support encrypted data, you modify the data model to use Transformable instead of String as the data type for the body attribute of the Note entity. Set the transformer name to EncryptedStringTransformer, which is a class you'll write next, as illustrated in Figure 10–23.



**Figure 10–23.** The body attribute with the Transformable data type

The role of the transformer is to encrypt/decrypt data as it goes into and comes out of the persistent store so that the rest of the application does not need to worry or know about encryption.

Create a new Objective-C class called EncryptedStringTransformer and make it a subclass of NSValueTransformer. Listing 10–20 shows the code for EncryptedStringTransformer.h, which is just as Xcode generated it.

### Listing 10–20. EncryptedStringTransformer.h

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

@interface EncryptedStringTransformer : NSValueTransformer

@end
```

Implement the transformer as shown in Listing 10–21. This class uses the NSData+Encryption category to do all the encryption and decryption, so the implementation is simple and consists of transforming the clear text to encrypted data and vice versa.

**Listing 10–21.** *EncryptedStringTransformer.m*

```
#import "EncryptedStringTransformer.h"
#import "NSData+Encryption.h"

@implementation EncryptedStringTransformer

+ (Class)transformedValueClass
{
    return [NSString class];
}

+ (BOOL)allowsReverseTransformation
{
    return YES;
}

- (id)transformedValue:(id)value
{
    if (value == nil)
        return nil;

    NSData *clearData = [value dataUsingEncoding:NSUTF8StringEncoding];
    return [clearData encryptWithKey:@"secret"];
}

- (id)reverseTransformedValue:(id)value
{
    if (value == nil)
        return nil;

    NSData *data = [value decryptWithKey:@"secret"];

    return [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
}

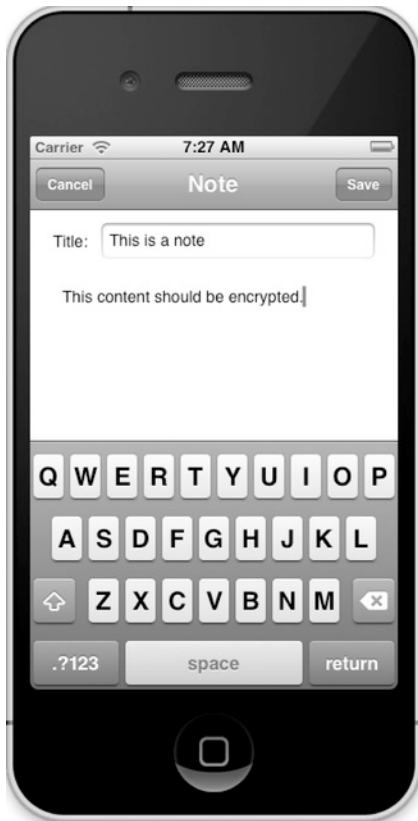
@end
```

**NOTE:** For the sake of simplicity, we took the liberty of using the same string (“secret”) as the encryption password in the transformer. Obviously, this defies the notion of secured encryption since anyone can have access to the password. We leave it up to you to change that password and/or make it a user-configurable variable.

## Testing the Encryption

Unlike the previous section where you couldn’t see the hardware-level encryption since it occurred only when the device was locked, with data encryption you can enjoy the peace of mind of verifying that your data is encrypted. Since your model isn’t versioned, you will need to reset the simulator before launching.

Launch the application and add a new note, as shown in Figure 10–24.



**Figure 10–24.** Adding a new encrypted note

We know you really enjoy running the MyStash app, but we need to ask you to put the toy down and take a look at the `Notes.sqlite` data store. Displaying the data in the `ZNOTE` table shows the following encrypted data:

```
sqlite> select * from ZNOTE;
=??X?C^?????
```

Of course, if you relaunch the application, the data will be decrypted properly before being displayed on the user interface. With this encryption strategy, you are in complete control of the data security. Regardless of whether the user has enabled data protection or not, the data is encrypted. If the device is stolen and unlocked, the data will still be encrypted.

## Sending Notifications When Data Changes

On Mac OS X, Core Data provides UI bindings that allow you to link your data directly to your user interface. With iOS, however, Core Data currently offers no UI bindings; the

closest thing you have is the `NSFetchedResultsController`. When using that controller is not an option, you have to rely on your own coding, with some help from the Key-Value Observing (KVO) mechanism. Since `NSManagedObject` is Key-Value Coding (KVC) compliant, all the usual principles of KVO can be applied, and notifications can be fired on data modifications. The KVO model is very simple and decentralized, with only two objects at play: the observer and the observed. The observer must be registered with the observed object and the observed object is responsible for firing notifications when its data changes. Thanks to the default implementation of `NSManagedObject`, firing notifications is automatically handled for you.

To illustrate how to use notifications, you'll create an example in which an observer outputs an acknowledgment that it has received a change notification when the `name` attribute of the `System` entity is modified. For simplicity, you output directly to the application console. In a real application, the KVO mechanism could be used to automatically update user interface elements.

## Registering an Observer

To receive change notifications, the observer must be registered with the observed object using the `addObserver:forKeyPath:options:context:` method. The registration is valid for a given key path only, which means it is valid only for the named attribute or relationship. The advantage is that it gives you a very fine level of control over what notifications fire.

The options give you an even finer level of granularity by specifying what information you want to receive in your notifications. Table 10–1 shows the options, which can be combined with a bitwise OR.

**Table 10–1.** *The Notification Options*

| Option                                    | Description   |
|---|---|
| <code>NSKeyValueObservingOptionOld</code> | The change dictionary will contain the old value under the lookup key <code>NSKeyValueChangeOldKey</code> . |
| <code>NSKeyValueObservingOptionNew</code> | The change dictionary will contain the new value under the lookup key <code>NSKeyValueChangeNewKey</code> . |

In `MyStash`, you will add an observer for the `System` entities. Open `PasswordListViewController.m`, and edit the `configureCell:atIndexPath:` method to add the observer, as shown in Listing 10–22.

**Listing 10–22.** *Adding an observer for changes to name*

```
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath
{
    NSManagedObject *system = [self.fetchedResultsController objectAtIndex:indexPath.index];
    cell.textLabel.text = [system valueForKey:@"name"];

    // Register this object for KVO
}
```

```
[system addObserver:self forKeyPath:@"name" options:NSKeyValueObservingOptionOld |←
NSKeyValueObservingOptionNew context:nil];
}
```

In this case, you are registering the `PasswordListViewController` as an observer of the `name` attribute that wants to receive the old and new values when the attribute for this managed object is modified.

## Receiving the Notifications

To receive notifications, the observer must provide an implementation for the `observeValueForKeyPath:ofObject:change:context:` method. This is the method that is called when a registered notification is fired. Since `PasswordListViewController` is the observer, you add the method in `PasswordListViewController.m`, as shown in Listing 10–23.

**Listing 10–23.** *Observing a change notification*

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
    change:(NSDictionary *)change context:(void *)context
{
    NSLog(@"Changed value for %@: %@ -> %@", keyPath,
        [change objectForKey:NSKeyValueChangeOldKey],
        [change objectForKey:NSKeyValueChangeNewKey]);
}
```

The change dictionary contains the modifications. Since you asked for both the old and new values when you registered the observer, you can extract both values from the dictionary.

Launch the application, and open the output console. Edit an existing password by changing the system name, and then save the change. You will see output in the console to show that the notification was received, like so:

```
2011-08-20 07:46:35.047 MyStash[11493:f503] Changed value for name: TouchPad -> iPad
```

Note that the change doesn't happen until you save the change and the context is saved.

The KVO mechanism provides you a simple way to respond to changes in your application's data.

## Seeding Data

A question we hear often concerns how to distribute a persistent store that already contains some data. Many applications, for example, include a list of values for the user to pick from to categorize the data they enter, and developers need to populate the data store with those values. You can approach this problem in several ways, including creating a SQLite database yourself and distributing that with your application instead of allowing Core Data to create the database itself, but this approach has its drawbacks, especially if a future version of your application augments this list of values. You really

shouldn't overwrite the user's database file with a new, otherwise blank one that has more values in its list.

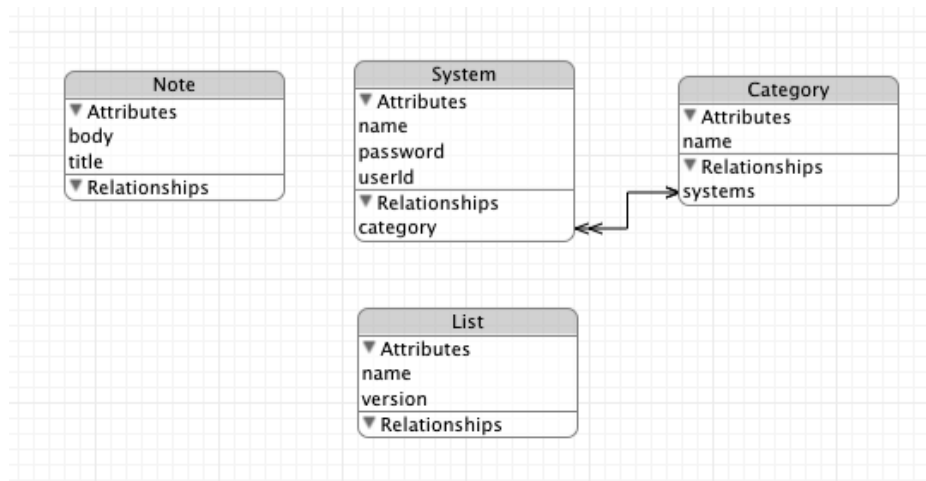
What, then, is a simple approach that allows for future change? If you've been following along with the examples in this book, you've already preloaded data into your data stores many times. Think about all the applications in this book that don't have user interfaces and how you got data into those. That's right: you created managed objects in code and then saved the managed object context. If you extend this approach a bit with a version to identify the data that's been loaded, you can easily determine in code whether to add more values in the future. In this section, you learn how to seed data into a Core Data–created persistent store in a way that allows you to change the preseeded data in future versions of your application.

## Adding Categories to Passwords

To demonstrate seeding a list of values in the data store, you'll add a category to the passwords in the MyStash application. Open the data model for MyStash, and add an entity named *Category* with a nonoptional attribute of type *String* called *name*. Add this new entity to the *Passwords* configuration. Add a to-many relationship called *systems* that has the *System* entity as its destination, and add an inverse relationship called *category* to the *System* entity.

To support versioning, add another entity called *List* with two non-optional attributes: a *String* called *name* and an *Integer 16* called *version*. Add it to the *Passwords* configuration.

Your data model should look like Figure 10–25.



**Figure 10–25.** The updated MyStash data model for seeding data

Add the following three method declarations to AppDelegate.h: one to load the data, one to save the context, and a helper to create a Category managed object:

```
- (void)loadData;
- (void)saveContext;
- (NSManagedObject *)addCategoryWithName:(NSString *)name;
```

Add a call to the loadData: method in AppDelegate.m, in the application:didFinishLaunchingWithOptions: method, as the first line in that method.

Then, implement the method. It should do the following:

1. Fetch the version number for the category entry in the List entity, creating the entry if it doesn't exist.
2. Based on the version number, determine which category managed objects to create and create them.
3. Update the version number.
4. Save the managed object context.

The code for the loadData: is shown in Listing 10–24.

**Listing 10–24.** *The loadData: implementation*

#pragma mark - Seed Data

```
- (void)loadData
{
    // Get the version object for "category"
    NSManagedObjectContext *context = [self managedObjectContext];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"List"↵
inManagedObjectContext:context]];
    [request setPredicate:[NSPredicate predicateWithFormat:@"name = 'category'"]];
    NSArray *results = [context executeFetchRequest:request error:nil];

    // Get the version number. If it doesn't exist, create it and set version to 0
    NSManagedObject *categoryVersion = nil;
    NSInteger version = 0;
    if ([results count] > 0)
    {
        categoryVersion = (NSManagedObject *)[results objectAtIndex:0];
        version = [(NSNumber *)[categoryVersion valueForKey:@"version"] intValue];
    }
    else
    {
        categoryVersion = [NSEntityDescription insertNewObjectForEntityForName:@"List"↵
inManagedObjectContext:context];
        [categoryVersion setValue:@"category" forKey:@"name"];
        [categoryVersion setValue:[NSNumber numberWithInt:0] forKey:@"version"];
    }

    // Create the categories to get to the latest version
```

```

if (version < 1)
{
    [self addCategoryWithName:@"Web Site"];
    [self addCategoryWithName:@"Desktop Software"];
}

// Update the version number and save the context
[categoryVersion setValue:[NSNumber numberWithInt:1] forKey:@"version"];
[self saveContext];
}

```

You can see that the code fetches the entry for category in the List entity and creates it if it doesn't exist. It pulls the version number for that entry to determine which version of the category list has been loaded. The first time you run this, the version will be set to zero. The code then adds the categories it should based on the version number and updates the version number to the latest so that subsequent runs won't append data that has already been loaded.

**NOTE:** For the purpose of demonstrating the mechanism, we coded our seed data by hand. In a real application, you should consider pulling the lists from some text file or CSV file.

Listing 10–25 shows the helper method for adding categories, `addCategoryWithName:`, which simply inserts a Category entity with the specified name into the managed object context.

**Listing 10–25.** *Method to add a category*

```

- (ManagedObject *)addCategoryWithName:(NSString *)name
{
    ManagedObject *category = [NSEntityDescription
insertNewObjectForEntityForName:@"Category" inManagedObjectContext:[self←
managedObjectContext]];
    [category setValue:name forKey:@"name"];
    return category;
}

```

Build and run the application, and then open `Passwords.sqlite` using `sqlite3`. If you run the `.schema` command, you'll see that the new tables for Category and List are present.

```

CREATE TABLE ZCATEGORY ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZNAME
VARCHAR );
CREATE TABLE ZLIST ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZVERSION
INTEGER, ZNAME VARCHAR );

```

The version number for the category list has been created.

```

sqlite> select zversion from zlist where zname='category';
1

```

Finally, the expected categories have been loaded.

```

sqlite> select * from zcategory;
1|1|1|Web Site
2|1|1|Desktop Software

```



You have successfully seeded your data store with category values. You can quit and rerun the application, and you'll notice that your versioning prevents these values from reloading and duplicating what's already been loaded.

## Creating a New Version of Seeded Data

Suppose you want to add credit card PINs as a category. Adding a version to this code is simple. You simply add another `if` statement, guarded by the new version number, and change the version number that's written to the database. You want to add the entries cumulatively so that if you're running the application for the first time, you get the categories for both versions 1 and 2, but if you're already at version 1, the code loads only the values for version 2. Finally, you update the version number in the data store to reflect the version currently loaded. Listing 10–26 shows the updated code in the `loadData:` method.

**Listing 10–26.** *Adding version 2 of the seeded data*

```
- (void)loadData
{
    ...
    // Create the categories to get to the latest version
    if (version < 1)
    {
        [self addCategoryWithName:@"Web Site"];
        [self addCategoryWithName:@"Desktop Software"];
    }
    if (version < 2)
    {
        [self addCategoryWithName:@"Credit Card PIN"];
    }

    // Update the version number and save the context
    [categoryVersion setValue:[NSNumber numberWithInt:2] forKey:@"version"];
    [self saveContext];
}
```

We haven't updated the `MyStash` user interface to use the categories—we leave that as an exercise for you.

## Error Handling

When you ask Xcode to generate a Core Data application for you, it creates boilerplate code for every vital aspect of talking to your Core Data persistent store. This code is more than adequate for setting up your persistent store coordinator, your managed object context, and your managed object model. In fact, if you go back to a non-Core Data project and add Core Data support, you'll do well to drop in the same code, just as Xcode generates it, to manage Core Data interaction. The code is production-ready.

That is, it is production-ready except in one aspect: error handling.

The Xcode-generated code alerts you to this shortcoming with a comment that says this:

```
/*
Replace this implementation with code to handle the error appropriately.

abort() causes the application to generate a crash log and terminate. You should not use
this function in a shipping application, although it may be useful during development.
If it is not possible to recover from the error, display an alert panel that instructs
the user to quit the application by pressing the Home button.
```

Typical reasons for an error here include:

- \* The persistent store is not accessible;
  - \* The schema for the persistent store is incompatible with current managed object model.
- Check the error message to determine what the actual problem was.

If the persistent store is not accessible, there is typically something wrong with the file path. Often, a file URL is pointing into the application's resources directory instead of a writeable directory.

If you encounter schema incompatibility errors during development, you can reduce their frequency by:

- \* Simply deleting the existing store:

```
[[NSFileManager defaultManager] removeItemAtURL:storeURL error:nil]
```

- \* Performing automatic lightweight migration by passing the following dictionary as the options parameter:

```
[NSDictionary dictionaryWithObjectsAndKeys:[NSNumber numberWithInt:YES],
NSMigratePersistentStoresAutomaticallyOption, [NSNumber numberWithInt:YES],
NSInferMappingModelAutomaticallyOption, nil];
```

Lightweight migration will only work for a limited set of schema changes; consult "Core Data Model Versioning and Data Migration Programming Guide" for details.

```
*/
```

The Xcode-generated implementation logs the error and aborts the application, which is a decidedly unfriendly approach. All users see when this happens is your application abruptly disappearing, without any clue as to why. If this happens in a shipping application, you'll get poor reviews and low sales.

Happily, however, you don't have to fall into this trap of logging and crashing. In this section, you will explore strategies for handling errors in Core Data. No one strategy is the best, but this section should spark ideas for your specific applications and audiences and help you devise an error-handling strategy that makes sense.

Errors in Core Data can be divided into two major categories:

- Errors in normal Core Data operations
- Validation errors

The next sections will discuss strategies for handling both types of errors.

## Handling Core Data Operational Errors

All the examples in this book respond to any Core Data errors using the default, Xcode-generated error-handling code, dutifully outputting the error message to Xcode's console and aborting the application. This approach has two advantages.

- It helps diagnose issues during development and debugging.
- It's easy to implement.

These advantages help only you as developer, however, and do nothing good for the application's users. Before you publically release an application that uses Core Data, you should design and implement a better strategy for responding to errors. The good news is that the strategy needn't be large or difficult to implement, because your options for how to respond are limited. Although applications are all different, in most cases you won't be able to recover from a Core Data error and should probably follow Apple's advice, represented in the previous comment: display an alert and instruct the user to close the application. Doing this explains to users what happened and gives them control over when to terminate the app. It's not much control, but hey—it's better than just having the app disappear.

Virtually all Core Data operational errors should be caught during development, so careful testing of your app should prevent these scenarios. Dealing with them, however, is simple: just follow Apple's advice from the comment. To add error handling code to the MyStash application, declare a method in AppDelegate.h for showing the alert:

```
- (void)showCoreDataError;
```

Then, add the implementation to AppDelegate.m. You can quibble with the wording, but remember that error messages aren't a paean to the muses, and the longer the message, the less likely it will be read. Listing 10-27 contains an implementation with a short and simple message—with only an extraneous exclamation mark to plead with the user to read it:

**Listing 10-27.** *A method to show a Core Data error*

```
- (void)showCoreDataError
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error!" message:@"MyStash
can't continue.\nPress the Home button to close MyStash." delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles: nil];
    [alert show];
}
```

Now, change the persistentStoreCoordinator: accessor method to use this new method instead of logging and aborting. Listing 10-28 shows the updates to persistentStoreCoordinator:.

**Listing 10-28.** *The updated persistentStoreCoordinator: method*

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

```

    {
        return __persistentStoreCoordinator;
    }

    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
    {
        // Create the Notes persistent store
        NSURL *noteStoreURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Notes.sqlite"];
        NSError *error = nil;
        if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:@"Notes" URL:noteStoreURL options:nil error:&error])
        {
            [self showCoreDataError];
        }
    }

    {
        // Create the Passwords persistent store
        NSURL *passwordStoreURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Passwords.sqlite"];
        NSError *error = nil;
        if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:@"Passwords" URL:passwordStoreURL options:nil error:&error])
        {
            [self showCoreDataError];
        }

        // Encrypt the password database
        NSDictionary *fileAttributes = [NSDictionary
dictionaryWithObject:NSFileProtectionComplete forKey:NSFileProtectionKey];
        if (![NSFileManager defaultManager] setAttributes:fileAttributes
ofItemAtPath:[passwordStoreURL path] error:&error))
        {
            [self showCoreDataError];
        }
    }

    return __persistentStoreCoordinator;
}

```

To force this error to display, run the MyStash application, and then close it. Go to the data model, add an attribute called `foo` of type `String` to the `Note` entity, and then run the application again. The persistent store coordinator will be unable to open the data store because the model no longer matches, and your new error message will display, as shown in Figure 10–26.



**Figure 10–26.** *Handling a Core Data error*

That's probably the best you can do to handle unexpected errors. The next section, however, talks about handling expected errors: validation errors.

## Handling Validation Errors

If you've configured any properties in your Core Data model with any validation parameters and you allow users to input values that don't automatically meet those validation parameters, you can expect to have validation errors. As you learned in Chapter 4, you can stipulate validation parameters on the properties of the entities in your data model. Validations ensure the integrity of your data; Core Data won't store anything you've proclaimed invalid into the persistent store. Just because you've created the validation rules, however, doesn't mean that users are aware of them—or that they know that they can violate them. If you leave the Xcode-generated error handling in place, users won't know they've violated the validation rules even after they input invalid data. All that will happen is that your application will crash, logging a long stack trace that the users will never see. Bewildered users will be left with a crashing application, and they won't know why or how to prevent its occurrence. Instead of

crashing when users enter invalid data, you should instead alert users and give them an opportunity to correct the data.

Validation on the database side can be a controversial topic, and for good reason. You can protect your data's integrity at its source by putting your validation rules in the data model, but you've probably made your coding tasks more difficult. Validation rules in your data model are one of those things that sound good in concept but prove less desirable in practice. Can you imagine, for example, using Oracle to do field validation on a web application? Yes, you can do it, but other approaches are probably simpler, more user-friendly, and architecturally superior. Validating user-entered values in code, or even designing user interfaces that prevent invalid entry altogether, makes your job easier and users' experiences better.

Having said that, however, we'll go ahead and outline a possible strategy for handling validation errors. Don't say we didn't warn you, though.

Detecting that users have entered invalid data is simple: just inspect the `NSError` object that you pass to the managed object context's `save:` method if an error occurs. The `NSError` object contains the error code that caused the `save:` method to fail, and if that code matches one of the Core Data validation error codes shown in Table 10–2, you know that some part of the data you attempted to save was invalid. You can use `NSError`'s `userInfo` dictionary to look up more information about what caused the error. Note that if multiple errors occurred, the error code is 1560, `NSValidationMultipleErrorsError`, and the `userInfo` dictionary holds the rest of the error codes in the key called `NSDetailedErrorsKey`.

**Table 10–2.** *Core Data Validation Errors*

| Constant  | Code | Description  |
|---|------|--|
| <code>NSManagedObjectValidationError</code>                   | 1550 | Generic validation error   |
| <code>NSValidationMultipleErrorsError</code>                  | 1560 | Generic message for error containing multiple validation errors  |
| <code>NSValidationMissingMandatoryPropertyError</code>        | 1570 | Nonoptional property with a <code>nil</code> value               |
| <code>NSValidationRelationshipLacksMinimumCountError</code>   | 1580 | To-many relationship with too few destination objects            |
| <code>NSValidationRelationshipExceedsMaximumCountError</code> | 1590 | Bounded, to-many relationship with too many destination objects  |
| <code>NSValidationRelationshipDeniedDeleteError</code>        | 1600 | Some relationship with <code>NSDeleteRuleDeny</code> is nonempty |

| Constant  | Code | Description                                   |
|---|------|---|
| <code>NSValidationNumberTooLargeError</code>        | 1610 | Some numerical value is too large             |
| <code>NSValidationNumberTooSmallError</code>        | 1620 | Some numerical value is too small             |
| <code>NSValidationDateTooLateError</code>           | 1630 | Some date value is too late                   |
| <code>NSValidationDateTooSoonError</code>           | 1640 | Some date value is too soon                   |
| <code>NSValidationInvalidDateError</code>           | 1650 | Some date value fails to match date pattern   |
| <code>NSValidationStringTooLongError</code>         | 1660 | Some string value is too long                 |
| <code>NSValidationStringTooShortError</code>        | 1670 | Some string value is too short                |
| <code>NSValidationStringPatternMatchingError</code> | 1680 | Some string value fails to match some pattern |

You can choose to implement an error-handling routine that's familiar with your data model and thus checks only for certain errors, or you can write a generic error handling routine that will handle any of the validation errors that occur. Though a generic routine scales better and should continue to work no matter the changes to your data model, a more specific error-handling routine may allow you to be more helpful to your users in your messaging and responses. Neither is the correct answer—the choice is yours for how you want to approach validation error handling.

To write a truly generic validation error handling routine would be a lot of work. One thing to consider is that the `NSError` object contains a lot of information about the error that occurred, but not necessarily enough information to tell the user why the validation failed. Imagine, for example, that you have an entity `Foo` with an attribute `bar` that must be at least five characters long. If the user enters “abc” for `bar`, you'll get an `NSError` message that tells you the error code (1670), the entity (`Foo`), the attribute (`bar`), and the value (`abc`) that failed validation. The `NSError` object doesn't tell you why `abc` is too short—it contains no information that `bar` requires at least five characters. To arrive at that, you'd have to ask the `Foo` entity for the `NSPropertyDescription` for the `bar` attribute, get the validation predicates for that property description, and walk through the predicates to see what the minimum length is for `bar`. It's a noble goal but tedious and usually overkill. This is one place where violating DRY and letting your code know something about the data model might be a better answer.

One other strange thing to consider when using validations in your data model is that they aren't enforced when you create a managed object; they're enforced only when you try to save the managed object context that the managed object lives in. This makes sense if you think it through, since creating a managed object and populating its

properties happens in multiple steps. First, you create the object in the context, and then you set its attributes and relationships. So, for example, if you were creating the managed object for the Foo entity in the previous paragraph, you'd write code like this:

```
NSManagedObject *foo = [NSEntityDescription insertNewObjectForEntityForName:@"Foo"
inManagedObjectContext:[self managedObjectContext]]; // foo is invalid at this point;
bar has fewer than five characters
[foo setValue:@"abcde" forKey:@"bar"];
```

The managed object foo is created and lives in the managed object context in an invalid state, but the managed object context ignores that. The next line of code makes the foo managed object valid, but it won't be validated until the managed object context is saved.

## Handling Validation Errors in MyStash

In this section, you implement a validation error handling routine for the MyStash application. It's generic in that it doesn't have any knowledge of which attributes have validation rules, but specific in that it doesn't handle all the validation errors—just the ones that you know you set on the model. Before doing that, however, you need to add some validation rules to MyStash's data model. Make the following changes to the `userId` attribute of the `System` entity:

- Set Min Length to 3.
- Set Max Length to 10.
- Set the regular expression to allow only letters and numbers: `[A-Za-z0-9]*`.

The attribute fields should match Figure 10-27. Save the data model. Now you're ready to implement the validation error-handling routine.

The screenshot shows the 'Attribute' inspector in Xcode. The 'Name' field is 'userId'. Under 'Properties', 'Optional' is checked. 'Attribute Type' is 'String'. Under 'Validation', 'Min Length' is 3 and 'Max Length' is 10, both checked. 'Default Value' is 'Default Value'. 'Reg. Ex.' is '[A-Za-z0-9]\*'.

**Figure 10-27.** Setting validations on the `userId` attribute of `System`



## Implementing the Validation Error Handling Routine

The validation error handling routine you write should accept a pointer to an `NSError` object and return an `NSString` that contains the error messages, separated by line feeds. Open `PasswordListViewController.h`, and declare the routine.

```
- (NSString *)validationErrorText:(NSError *)error;
```

The routine itself, which goes in `PasswordListViewController.m`, creates a mutable string to hold all the error messages. It then creates an array that holds all the errors, which can be multiple errors or a single error. It then iterates through all the errors, gets the property name that was in error, and, depending on the error code, forms a proper error message. Notice that some knowledge of the model (the minimum and maximum lengths, 3 and 10) is hardcoded, because the error objects don't have that information. Listing 10–29 contains the `validationErrorText:` implementation.

**Listing 10–29.** *A method for producing error text from an `NSError` object*

```
#pragma mark - Validation Error Handling
```

```
- (NSString *)validationErrorText:(NSError *)error
{
    // Create a string to hold all the error messages
    NSMutableString *errorText = [NSMutableString stringWithCapacity:100];

    // Determine whether we're dealing with a single error or multiples, and put them all
    // in an array
    NSArray *errors = [error code] == NSValidationMultipleErrorsError ? [[error userInfo]
objectForKey:NSDetailedErrorsKey] : [NSArray arrayWithObject:error];

    // Iterate through the errors
    for (NSError *err in errors)
    {
        // Get the property that had a validation error
        NSString *propName = [[err userInfo] objectForKey:@"NSValidationErrorKey"];
        NSString *message;
        // Form an appropriate error message
        switch ([err code])
        {
            case NSValidationMissingMandatoryPropertyError:
                message = [NSString stringWithFormat:@"%@" required", propName];
                break;
            case NSValidationStringTooShortError:
                message = [NSString stringWithFormat:@"%@" must be at least %d characters",
propName, 3];
                break;
            case NSValidationStringTooLongError:
                message = [NSString stringWithFormat:@"%@" can't be longer than %d characters",
propName, 10];
                break;
            case NSValidationStringPatternMatchingError:
                message = [NSString stringWithFormat:@"%@" can contain only letters and
numbers", propName];
```

```

        break;
    default:
        message = @"Unknown error. Press Home button to halt.";
        break;
    }
    // Separate the error messages with line feeds
    if ([errorText length] > 0)
    {
        [errorText appendString:@"\n"];
    }
    [errorText appendString:message];
}
return errorText;
}

```

You have a fair amount of jiggling to do to the code to incorporate this routine into the application. Start in `PasswordListViewController.h`, and change the return types for two methods, for reasons that will become apparent in a few moments. Change `insertPasswordWithName:` to return an `NSManagedObject*`, and change `saveContext:` to return an `NSString*`, as shown in Listing 10–30.

**Listing 10–30.** *Updating return types in `PasswordListViewController.h`*

```

- (NSManagedObject *)insertPasswordWithName:(NSString *)name userId:(NSString *)userId
password:(NSString *)password;
- (NSString *)saveContext;

```

The reason `insertPasswordWithName:` must now return an `NSManagedObject*` is that you may have to delete the managed object this method creates. Consider the following sequence of events:

1. User taps + to create a new password.
2. User enters invalid data.
3. User taps Save. A new managed object is created in the context, but saving the context fails.
4. User dismisses the alert that complains of invalid data.
5. User taps Cancel.

If you don't have a handle to the newly created object so you can delete it, in this scenario the invalid object would still exist in the managed object context, and subsequent saves would fail without recourse. Listing 10–31 shows the updated `insertPasswordWithName:` method that returns the managed object. It also shows the updated `saveContext:` method, which will now return the error text if the save fails. Otherwise, it will return `nil`.

**Listing 10–31.** *The `initWithPasswordWithName:` and `saveContext:` methods*

```

- (NSManagedObject *)insertPasswordWithName:(NSString *)name userId:(NSString *)userId
password:(NSString *)password
{
    NSManagedObjectContext *context = [fetchResultsController managedObjectContext];

```

```

    NSEntityDescription *entity = [[fetchResultsController fetchRequest] entity];
    NSManagedObject *newPassword = [NSEntityDescription
insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];

    [newPassword setValue:name forKey:@"name"];
    [newPassword setValue:userId forKey:@"userId"];
    [newPassword setValue:password forKey:@"password"];

    return newPassword;
}

- (NSString *)saveContext
{
    NSString *errorText = nil;
    NSError *error = nil;
    if (![self.fetchResultsController managedObjectContext] save:&error))
    {
        errorText = [self validationErrorText:error];
    }
    return errorText;
}

```

The only thing that has to change in the PasswordViewController class is the implementation of the `save:` method. Remember how simple and clean it used to be? Well, it's simple no more. Users can now do things like edit an existing system entry, change some values so they're invalid, tap Save (which updates the values on the object), and then tap Cancel after dismissing the alert. You must change this method to undo any changes if anything doesn't pass validation, so you save state before you make any changes so you can restore the state if any errors occur. If any errors occur, show the alert and don't dismiss the modal. Listing 10–32 shows the new `save:` method.

**Listing 10–32.** *The updated `save:` method*

```

- (IBAction)save:(id)sender
{
    NSString *errorText = nil;

    // Create variables to store pre-change state, so we can back out if
    // validation errors occur
    NSManagedObject *tempSystem = nil;
    NSString *tempName = nil;
    NSString *tempUserId = nil;
    NSString *tempPassword = nil;

    if (parentController != nil)
    {
        if (system != nil)
        {
            // User is editing an existing system. Store its current values
            tempName = [NSString stringWithString:(NSString *)[system valueForKey:@"name"]];
            tempUserId = [NSString stringWithString:(NSString *)[system
valueForKey:@"userId"]];
            tempPassword = [NSString stringWithString:(NSString *)[system
valueForKey:@"password"]];

```

```

        // Update with the new values
        [system setValue:name.text forKey:@"name"];
        [system setValue:userId.text forKey:@"userId"];
        [system setValue:password.text forKey:@"password"];
    }
    else
    {
        // User is adding a new system. Create the new managed object but keep
        // a pointer to it
        tempSystem = [parentController insertPasswordWithName:name.text
userId:userId.text password:password.text];
    }
    // Save the context and gather any validation errors
    errorText = [parentController saveContext];
}
if (errorText != nil)
{
    // Validation error occurred. Show an alert.
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error!"
message:errorText delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alert show];

    // Because we had errors and the context didn't save, undo any changes
    // this method made
    if (tempSystem != nil)
    {
        // We added an object, so delete it
        [[parentController.fetchedResultsController managedObjectContext]
deleteObject:tempSystem];
    }
    else
    {
        // We edited an object, so restore it to how it was
        [system setValue:tempName forKey:@"name"];
        [system setValue:tempUserId forKey:@"userId"];
        [system setValue:tempPassword forKey:@"password"];
    }
}
else
{
    // Successful save! Dismiss the modal only on success
    [self dismissModalViewControllerAnimated:YES];
}
}
}

```

Whew! That was an awful lot of work to move validations to the data model. As you explore this way to do validations, you'll probably agree that designing better user interfaces (for example, enabling the Save button only when the `userId` field contains valid data) or validating values in code is a better approach.

If you run this application, tap to add a new password, and enter more than ten characters (including punctuation) in the User ID field, you'll see the fruits of your validation labor, as shown in Figure 10–28.



**Figure 10–28.** Setting validations on the *userId* attribute of *System*

## Summary

In this chapter, you explored several advanced topics relating to Core Data. Some you'll most likely use, like error handling and validation, in many of your applications. Some, like encryption, you might use only occasionally.

As you've worked through this book, you've learned the breadth and depth of Core Data. You learned how much work Core Data does for you, and you learned what work you must do to properly interact with the Core Data framework. We hope you've enjoyed reading this book as much as we enjoyed writing it, and we look forward to hearing from you as you use Core Data in your applications!

---

# Index

## A

- Actor class, [226](#)
- Actor entity, [213](#), [226](#)
- Actor.h file, [226](#)
- Actor.m file, [226](#)
- ActorsMap dictionary, [243](#)
- Actor table, [213](#)
- Actor-to-Movie table, [213](#)
- Add Fetch Request option, Xcode IDE, [122](#), [123](#)
- Add Model Version option, Xcode IDE, [254](#)
- Add type, [263](#)
- AddObserver forKeyPath options
  - context method, [348](#)
- AddPersistentStoreWithType method, [258](#), [261](#)
- AddVerticesObject method, [164](#)
- AllowsReverseTransformation method, [172](#)
- Application delegate, [9](#), [14](#), [24](#), [25](#)
- Application
  - didFinishLaunchingWithOptions method, [173](#), [191](#), [215](#)
- Application testing, [209](#)
  - adding framework to application, [220](#), [222](#)
  - building testing framework, [218](#), [219](#)
  - creating Core Data project, [210](#)
  - creating data and data model, [213](#), [215](#)
  - creating testing view, [215](#), [217](#)
  - Run Selected Test button, [209](#)
  - running initial test, [222](#), [223](#)
- Application(s)
  - managing table views using
    - NSFetchedResultsController class, [285](#)
    - cache name, [287](#)
    - delegates, [287–288](#)
    - fetch request, [286](#)
    - implementing, [293–297](#)
    - implementing in League Manager, [288](#), [292](#)
    - implementing
      - NSFetchedResultsController Delegate protocol (see NSFetchedResultsController Delegate protocol)
    - managed object context, [286](#)
    - section name key path, [286](#)
  - Shapes, enabling user interactions with, [156](#)
- AssociateSourceInstance
  - withDestinationInstance
    - forEntityMapping method, [264](#)
- Atomic store, [94](#)
- Attribute details pane, Xcode IDE, [126](#)
- Attributes
  - creating, [125–127](#)
  - viewing and editing details, [119](#)
    - default value property, [120](#)
    - Description column, [119](#)
    - detail pane, [119](#)
    - indexed property, [120](#)
    - max length property, [120](#)
    - max value property, [120](#)
    - min length property, [120](#)
    - min value property, [120](#)
    - Name column, [119](#)

- name property, [120](#)
- optional property, [120](#)
- reg. ex. property, [120](#)
- transient property, [120](#)
- type property, [120](#)
- validation, [120](#)

Autosizing section, [145](#), [146](#)

AwakeFromInsert method, [180](#)

## B

BasicApplication.xcdatamodeld group, [9](#)

BasicApplicationAppDelegate.m file, [14](#)

BasicCanvasUIView class, [145](#), [146](#)

BasicCanvasUIView.m file, [170](#)

BasicCanvasUIView.m file, [268](#), [269](#)

Batch faulting, [229](#), [231](#)

BeginUpdates method, [287](#)

Binary attribute type, [127](#)

Binary data attribute type, [32](#)

BLOB column, [170](#)

Bool attribute type, [127](#)

BOOL data type, [32](#)

Boolean attribute type, [32](#)

Brute-force cache expiration, [236](#)

## C

Cache expiration, [236](#)

- brute-force, [236](#)
- memory consumption, [236](#)
- through faulting, [237](#)

Cache name, [287](#)

CacheName parameter, [11](#)

CacheTest class, [234](#)

CacheTest.h file, [234](#)

CacheTest.m file, [234](#)

Caching, [233](#)

- cache expiration, [236](#)
- brute-force, [236](#)
- memory consumption, [236](#)
- through faulting, [237](#)
- cache management scheme, [233](#)
- cached vs. noncached objects
- loading, [235](#)

CacheTest class, [234](#)

CacheTest.h file, [234](#)

CacheTest.m file, [234](#)

and faulting, [225](#)

LoadDataFromContext method, [235](#)

NSManagedObjectContext class, [233](#)

PerformanceTest protocol, [234](#)

RunWithContext method, [235](#)

CanRedo method, [181](#)

CanUndo method, [181](#)

Canvas entity, [259](#)

Cascade rule, [41](#), [131](#)

Circle.h file, [268](#)

CircleToEllipse entity mapping, [275](#)

CircleToEllipseMigrationPolicy class, [272](#)

CircleToEllipseMigrationPolicy.h file, [273](#)

CircleToEllipseMigrationPolicy.m file, [273](#)

Class field, Xcode IDE, [124](#)

Comparators, [241](#)

ConfigureCell method, [320](#)

Context, [4](#)

Controller didChangeObject

- atIndexPath forChangeType
- newIndexPath method, [287](#)

Controller didChangeSection atIndexPath forChangeType method, [287](#)

Controller sectionIndexTitleForSection Name method, [287](#)

ControllerDidChangeContent method, [288](#)

ControllerWillChangeContent method, [287](#)

Copy type, [263](#)

Core Data

- adding to existing project, [16](#)
- add Core Data framework, [16–19](#)
- creating data model, [19–21](#)
- initializing managed object, [22–24](#)
- application creation, [3](#)
- components, [3–4](#)
- creating new project, [5–6](#)

- fetching results, [10–13](#)
- initializing managed context,  
[14–16](#)
- inserting new objects, [13–14](#)
- running new project, [6–8](#)
- understanding components, [9–10](#)
- data models (see Data models)
- faults (see Faulting)
- initialization sequence, [15](#)
- NSFetchedResultsController (see NSFetchedResultsController class)
- persistent store (see Persistent store)
- testing (see Application testing; Performance testing; Testing)
- Core Data framework classes, [27](#)
- create new application in Xcode, [28](#)
- data access classes, [38](#)
  - addPersistentStoreWithType method, [39](#)
- class diagram, [38](#)
- default persistent store types in iOS, [39](#)
- delete rules, [41](#)
- initialization sequence, [38](#)
- initWithManagedObjectModel method, [39](#)
- life-cycle operations in NSManagedObjectContext, [42](#)
- managed object context object graph, [39](#)
- methods for retrieving and creating data objects, [41](#)
- migratePersistentStore method, [39](#)
- NSManagedObject class, [42](#)
- NSManagedObjectContext, [40](#)
- NSManagedObjectModel class, [39](#)
- NSXMLStoreType, [40](#)
- redo operation, [41](#)
- reference to context, [43](#)
- setValue forKey method, [42](#)
- undo operation, [41](#)
- valueForKey method, [42](#)
- high-level overview, [30](#)
- interaction of classes, [47](#)
  - autoincrement identifier, [47](#)
  - createData method, [50–53](#)
  - didFinishLaunchingWithOptions method, [50](#)
  - id attribute, [51](#)
  - mutableSetValueForKey method, [52](#)
  - object graph, [47](#)
  - organization data model, [47](#)
  - Organization entity, [47](#)
  - OrgChart application, [47](#)
  - OrgChartAppDelegate.h file, [48–50](#)
  - OrgChartAppDelegate.m file, [48](#)
  - Person entity, [47](#)
  - reading data using Core Data, [55–56](#)
  - save method, [52](#)
  - setValue forKey method, [51](#)
  - SQLite primer, [53–54](#)
  - valueForKey method, [52](#)
- key-value observing, [43](#)
  - didChangeValueForKey method, [43](#)
  - NSManagedObject class, [43](#)
  - willChangeValueForKey method, [43](#)
- model definition classes, [30](#)
  - Add Relationship option, Xcode IDE, [34](#)
  - attributes, [30](#)
  - attributeType property, [34](#)
  - data model, [37](#)
  - fetchd properties, [30](#)
  - id attribute, [34](#)
  - inverse relationships, [36](#)
  - managedObjectClassName property, [34](#)
  - name attribute, [34](#)
  - name property, [34](#)
  - NSAttributeDescription class, [31, 34](#)
  - NSAttributeType structure, [32](#)
  - NSEntityDescription class, [31](#)



- NSFetchedPropertyDescription class, 31
- NSInteger16AttributeType property, 34
- NSManagedObjectModel class, 31
- NSPropertyDescription class, 31
- NSRelationshipDescription class, 31
- NSStringAttributeType property, 34
- object graph, 33, 38
- object model description, 38
- one-to-many relationship, 36
- Organization attribute, 34
- organization model, 34, 37
- Person entity, 34
- project with organization entity, 33
- properties of relationship, 35
- relationships, 30
- transformable attributes, 33
- NSManagedObjectContext class, 29
- NSManagedObjectModel class, 30
- NSPersistentStoreCoordinator class, 30
- OrgChart, 28
- query classes, 44
  - main classes, 46
  - NSFetchedResultsController class, 45
  - NSFetchRequest class, 45
  - NSFetchRequest class diagram, 44
  - NSManagedObject class, 45
  - NSManagedObjectContext class, 45
  - NSPredicate class, 45
  - NSPredicate instance, 44
  - NSSortDescriptor instance, 44
  - predicateWithFormat method, 45
- Core Data model configuration
  - MyStash.xcdatamodel model, 337
- Notes configuration, 336
  - adding Note entity, 337
  - Notes.sqlite database, 339
  - Passwords configuration, 336
    - adding System entity, 338
    - Passwords.sqlite database, 339
    - updated persistentStoreCoordinator method, 338
- Core Data project, 210
- Core Data recording instruments, 248
- Core Data template, Xcode IDE, 63
- Crawford, Shane, 175
- CreateData method, 56, 191
- CreateDestinationInstancesForSourceIn stance method, 264, 273
- CreateRelationshipsForDestination Instance method, 264, 274
- CreateShapeAt method, 154, 163, 168, 268
- Creating attributes, 125-127
- Creating entities, 123-125
- Creating relationships, 127
  - Delete Rule field, 130-131
  - Destination and Inverse field, 129
  - Min Count and Max Count fields, 130
  - Name field, 128
  - Optional check box, 129
  - To-Many Relationship check box, 130
  - Transient check box, 129
- CRUD (create, retrieve, update, delete), 133, 156
  - building Shape application user interfaces, 145-153
  - enabling user interactions with Shapes application, 154-156
  - Shape application data models, 137-145
- Custom data migration
  - launching, 282
  - need for, 280
  - setting up migration manager, 281
    - MyMigrationManager.h file, 281
    - MyMigrationManager.m file, 281
- Custom entity mapping type, 263-264
- Custom migration, 278
- Custom type, 263

## D

### Data encryption

- EncryptedStringTransformer
  - EncryptedStringTransformer.h, [345](#)
  - EncryptedStringTransformer.m, [346](#)
- NSData class, [342](#)
  - NSData+Encryption.h, [343](#)
  - NSData+Encryption.m, [344](#)
- Objective-C category, [342](#)
- testing, [346–347](#)

### Data migration, [253](#), [276](#)

- custom, [278](#)
  - launching, [282](#)
  - need for, [280](#)
  - setting up migration manager, [281](#)
- lightweight, [257](#)
  - complex changes, [259](#)
  - creating mapping model, [262](#)
  - mapping model, [257](#)
  - renaming entities and properties, [260](#)
  - simple changes, [258](#)
- running Shapes application, [277](#)

### Data models, [9](#), [10](#), [111](#)

- adding, [19](#)
- adding entity and attribute, [21](#)
- creating attributes, [125–127](#)
- creating entities, [123=125](#)
- creating mapping model, [262](#), [270](#)
  - creating new model versions, [266](#)
  - source and target selection, [271](#)
  - understanding entity mappings, [263](#)
  - understanding property mappings, [264](#)
- creating relationships, [127](#)
  - Delete Rule field, [130–132](#)
  - Destination and Inverse field, [129](#)
  - Min Count and Max Count fields, [130](#)
  - Name field, [128](#)
  - Optional check box, [129](#)

- To-Many Relationship check box, [130](#)
- Transient check box, [129](#)
- designing databases, [111](#)
- League Manager application, [112](#)
- Player entity, [112](#)
- relational database normalization, [112](#), [113](#)
- Team entity, [112](#)
- Z1TEAMS table, [112](#)
- ZPLAYER table, [112](#)
- ZTEAM table, [112](#)
- naming, [20](#)
- new empty, [21](#)
- performance tuning, [213](#)
- using Xcode data modeler, [113](#)
  - graph view, [116](#), [117](#)
  - League Manager model, [116](#), [117](#)
  - mapping models, [114](#)
  - new file, [114](#)
  - NSManagedObject class, [117](#)
  - Player entity, [117](#)
  - team relationship, [119](#)
  - user interface, [114](#), [115](#)
  - using fetched properties, [121](#)
  - viewing and editing attribute details, [119–120](#)
  - viewing and editing relationship details, [120–121](#)

### Data objects

- CRUD (create, retrieve, update, delete), [133](#), [156](#)
  - building Shape application user interfaces, [145–153](#)
  - creating Shape application data models, [145](#)
  - enabling user interactions with Shapes application, [154–156](#)
  - Shape application data models, [137](#)
- generating classes, [156–164](#)
- modifying generated classes, [164–169](#)
- shapes, [134](#), [136](#)
- undoing and redoing, [180–186](#)
  - adding undo to Shapes, [182–186](#)

- disabling undo tracking, 182
  - limiting undo stack, 181
  - undo groups, 181
- using Transformable type, 169–173
- validating data, 173
  - custom validation, 175–179
  - data validation errors, 174
  - default values, 179–180
  - invoking validation, 179
- Data store
  - managed object model, 14
  - persistent store, 14, 15
- Data validation, 173
  - custom validation, 175–179
  - default values, 179–180
  - invoking validation, 179
  - validation errors, 174, 175
- Databases
  - creating attributes, 125–127
  - creating entities, 123–125
  - creating relationships, 127
    - Delete Rule field, 130–132
    - Destination and Inverse field, 129
    - Min Count and Max Count fields, 130
    - Name field, 128
    - Optional check box, 129
    - To-Many Relationship check box, 130
    - Transient check box, 129
  - designing, 111
    - League Manager application, 112
    - Player entity, 112
    - relational database normalization, 112–113
    - Team entity, 112
    - Z1TEAMS table, 112
    - ZPLAYER table, 112
    - ZTEAM table, 112
- DataSource option, 217
- Date attribute type, 32, 127
- DateOfBirth attribute, Person entity, 124
- Decimal attribute type, 32, 126
- Default Value property, 120
- Delegate option, 217
- Delegates, 287–288
- Delete Rule field, 130–132
- DeleteAllObjects method, 156
- DeleteAllShapes method, 152
- DeleteObject method, 41
- DemoApp1AppDelegate.m file, 24
- DemoAppDelegate.h file, 22
- Deny rule, 41, 131
- Designing databases, 111
  - League Manager application, 112
  - Player entity, 112
  - relational database normalization, 112–113
  - Team entity, 112
  - Z1TEAMS table, 112
  - ZPLAYER table, 112
  - ZTEAM table, 112
- Destination field, 129
- Detail pane, Xcode IDE, 119
- DidFinishLaunchingWithOptions
  - method, 24, 56
- DidTurnIntoFault method, 226
- DidTurnIntoFaultTest class, 227
- DidTurnIntoFaultTest.h file, 227, 228
- DidTurnIntoFaultTest.m file, 227
- Dijkstra, Edward, 27
- DisableUndoRegistration method, 182
- DisplayPerson method, 55
- Double attribute type, 32, 126
- DrawRect method, 143, 170, 268

## E

- Ellipse class, 267, 268
- Ellipse entity, 259, 266, 267
- Ellipse.h file, 267
- Ellipse.m file, 267
- Email attribute, 66
- Email attribute, Player entity, 117, 118
- Emerson, Ralph Waldo, 111
- EnableUndoRegistration method, 182
- Encryption
  - data protection, 340
    - on the device, 341
    - persistent store encryption, 340
  - EncryptedStringTransformer, 345

- EncryptedStringTransformer.h, 345
  - EncryptedStringTransformer.m, 346
  - NSData+Encryption category, 343
  - objective-C category, 343
  - password database, 341
  - testing, 346, 347
  - Enterprise JavaBeans (EJBs), 27
  - Enterprise Objects Framework (EOF), 2
  - Entities, 123–125
  - Entity details pane, Xcode IDE, 119, 124
  - Entity mappings, 263
  - Entity section, Xcode IDE, 123
  - Error handling, MyStash, 353
    - core data operations, 355
    - initWithPasswordWithName methods, 362
  - NSError object, 361
  - PasswordListViewController.h, 362
  - production-ready code, 354
  - saveContext methods, 362
  - updated save method, 363
  - validation errors, 357
  - ErrorFromOriginalError method, 178, 179
  - Event entity, 9–11
  - ExecuteFetchRequest error method, 41
  - Expiring of cache, 236
    - brute-force cache expiration, 236
    - memory consumption, 236
    - through faulting, 237
  - ExpressionForAggregate method, 194
- F, G**
- Faulting, 223
    - building tests, 226–229
    - and caching, 225
    - firing faults, 224
    - firing faults on purpose, 229–230
      - batch faulting, 229
      - prefetching, 229
    - prefetching, 231
      - PreFetchFaultingTest class, 231
      - RunWithContext method, 231
      - setRelationshipKeyPathsForPrefetching method, 231
      - SinglyFiringFaultTest.m file, 231
    - refaulting, 225–226
  - Fetch request, 286
  - FetchAllMoviesActorsAndStudiosTest class, 219
  - Fetches properties, 121
  - FetchesObjects property, 13
  - FetchesResultsController
    - entire getter method, 12, 13
    - property, 11
  - Fifth normal form (5NF), 113
  - Firing faults, 224, 229–230
    - batch faulting, 229
    - prefetching, 229
  - First normal form (1NF), 113
  - FirstName attribute, 66
  - Float attribute type, 32, 126
  - FloatValue method, 143
  - Forms, 112
  - Fourth normal form (4NF), 113
  - Fowler, Martin, 165, 237
  - Frameworks, 16
- H**
- Hanselman, Scott, 134
  - HasChanges method, 42
  - Hide System Libraries check box, 250
- I, J**
- Identity map, 237
  - Indexed property, 120
  - InitWithStyle method, 314
  - In-memory persistent store
    - caching remote information, 93
    - League\_ManagerAppDelegate.m, 92
    - life cycle, 93
    - persistentStoreCoordinator method, 92
  - InsertNewObject method, 13
  - InsertObject method, 41
  - Instrument library, 248

- Instruments, launching, 245
  - library, 248
  - main window, 247
  - template selection, 246
- Int16 attribute type, 126
- Int32 attribute type, 126
- Int64 attribute type, 126
- Integer 16 attribute type, 32
- Integer 32 attribute type, 32
- Integer 64 attribute type, 32
- Inverse field, 129
- iOS
  - Core Data, application creation (see Core Data, application creation)
  - history of persistence in, 2, 3
- IsConfiguration
  - compatibleWithStoreMetadata method,
  - NSManagedObjectModel class, 280

## K

- Key-Value Observing (KVO) mechanism, 348
- KVC (Key-Value Coding), 162

## L

- LaMarche, Jeff, 175
- LastName attribute, 66
- Launching instruments, 245
  - library, 248
  - main window, 247
  - template selection, 246
- League Manager application
  - adding new team, 80
  - CustomStore class, 95, 106
  - CustomStore.h file, 95
  - CustomStore.m file, 96
  - implementing
    - NSFetchedResultsController in, 288, 292
  - League Manager view, 289
  - League\_Manager.sqlite database, 107

- League\_Manager.txt file, 107
- League\_ManagerAppDelegate.m file, 106
- metadataForPersistentStoreWithURL
  - error method, 98
- NSPersistentStore class, 97
- persistentStoreCoordinator method, 106
- Players view, 289
  - with teams, 80
  - without teams, 79
  - writeMetadata toURL method, 97
- LoadData method, 214, 215
- LoadDataFromContext method, 235

## M

- MakeRandomColor method, 151, 170
- Managed object context, 4, 286
- Managed Objects group, 266
- Mapping model, 114, 257
  - creating new model versions, 266
  - source and target selection, 271
  - understanding entity mappings, 263
  - understanding property mappings, 264
- MasterViewController, 10
- Max Count field, 130
- Max Length property, 120
- Max Value property, 120
- Memory consumption, 236
- Memory management, 233
- Memory usage
  - cache expiration, 236
    - brute-force, 236
    - memory consumption, 236
    - through faulting, 237
- caching, 233
  - cache management scheme, 233
  - cached vs. noncached objects
    - loading, 235
  - CacheTest class, 234
  - CacheTest.h file, 234
  - CacheTest.m file, 234
  - LoadDataFromContext method, 235

- NSManagedObjectContext class, 233
- PerformanceTest protocol, 234
- RunWithContext method, 235
- prefetching, 231
  - PreFetchFaultingTest class, 231
  - RunWithContext method, 231
  - setRelationshipKeyPathsForPrefetching method, 231
  - SinglyFiringFaultTest.m file, 231
- MergeChanges parameter, 225
- MergedModelFromBundles
  - forStoreMetadata method, 281
- MetadataForPersistentStoreOfType
  - URL error method, 280
- Migration Mappings group, 272
- Min Count field, 130
- Min Length property, 120
- Min Value property, 120
- Model versions, 266
- Model4to5.xcmappingmodel file, 270, 277
- Movie entity, 213, 250
- Movie table, 213
- Movie-to-Studio table, 213
- MyAttribute attribute, 20, 24
- MyMigrationManager.h file, 281
- MyMigrationManager.m file, 281
- MyStash project, 307
  - adding tab
    - Notes view, 311, 314
    - password list view, 314
  - Empty Application template, 307
  - Encryption (see Encryption)
  - error handling
    - core data operations, 355
    - production-ready code, 353
    - validation errors, 357, 361
  - incorporating
    - NSFetchedResultsController, 316
    - accessor method, 317
    - delegating calls, 319
    - methods for deleting rows, 321
    - NoteListViewController.h, 317
- NSFetchedResultsController
  - Delegate protocol methods, 318
- PasswordListViewController.h, 317
- interface for editing notes
  - methods, 326
  - NoteViewController.h, 322
  - NoteViewController.m, 323
  - NoteViewController.xib, 325
  - tableView
    - didSelectRowAtIndexPath method, 327
  - viewDidLoad, adding buttons, 327
- interface for editing passwords
  - configured password view, 332
  - PasswordViewController.m, 330
  - methods, 333
  - PasswordViewController.h, 329
  - PasswordViewController.xib, 331
  - updated tableView
    - didSelectRowAtIndexPath method, 334
  - updated viewDidLoad method, 333
- multiple persistent stores, 335 (see *also* Core Data model configuration)
- Note entity, data model, 309
- Note List view
  - in edit mode, 329
  - note added, 328
- notifications
  - KVO model, 348
  - NSKeyValueObservingOptionNew, 348
  - NSKeyValueObservingOptionOld, 348
  - receiving notifications, 349
  - registered observer, 348
- seeding data, 349
  - adding category to passwords, 350
  - creating new version, 353
- setting up, 308

System entity, data model, [309](#)  
 tab bar controller, [310](#), [311](#)

## N

- Name attribute, [213](#), [214](#), [229](#)
- Name field, [128](#)
- Name property, [120](#)
- NeXT technology, [2](#)
- No action rule, [41](#), [131](#)
- Normalization. (See Relational database normalization)
- NoteListViewController class, [311](#)
- Notes.sqlite database, [339](#)
- NoteViewController class
  - methods, [326](#)
  - NoteViewController.h, [322–324](#), [326](#)
  - NoteViewController.m, [323](#)
  - NoteViewController.xib, [325](#)
  - viewDidLoad, adding buttons, [327](#)
- NSAddEntityMappingType, [263](#)
- NSAtomicStoreCacheNode, [98](#)
  - CustomStore.h file, [100](#)
  - newCacheNodeForManagedObject method, [99](#)
  - nodeForReferenceObject
    - andObjectID method, [100](#)
  - updateCacheNode
    - fromManagedObject method, [99](#), [100](#)
- NSBinaryDataAttributeType, [32](#)
- NSBinaryDataAttributeType, [127](#)
- NSBinaryStoreType, [39](#)
- NSBooleanAttributeType, [32](#), [127](#)
- NSCopyEntityMappingType, [263](#)
- NSCustomEntityMappingType, [263](#)
- NSData attribute type, [127](#)
- NSData data type, [32](#)
- NSDate attribute type, [127](#)
- NSDate data type, [32](#)
- NSDateAttributeType, [32](#), [127](#)
- NSDecimalAttributeType, [32](#), [126](#)
- NSDecimalNumber attribute type, [126](#)
- NSDecimalNumber data type, [32](#)
- NSDoubleAttributeType, [126](#)
- NSDoubleAttributeType, [32](#)
- NSEntityMigrationPolicy class, [264](#)
- NSFetchedResultsControllerChangeType values
  - NSFetchedResultsControllerDelete, [303](#)
  - NSFetchedResultsControllerInsert, [303](#)
  - NSFetchedResultsControllerMove, [303](#)
  - NSFetchedResultsControllerUpdate, [303](#)
- NSFetchedResultsController class, [10](#), [11](#), [316](#)
  - accessor method, [317](#)
  - delegating calls, [319](#)
  - managing table views, [285](#)
    - cache name, [287](#)
    - delegates, [287–288](#)
    - fetch request, [286](#)
    - implementing, [293–297](#)
    - implementing in League Manager, [288](#), [292](#)
    - implementing
      - NSFetchedResultsControllerDelegate protocol (see NSFetchedResultsControllerDelegate protocol)
      - managed object context, [286](#)
      - section name key path, [286](#)
    - methods for deleting rows, [321](#)
    - NoteListViewController.h, [317](#)
    - NSFetchedResultsControllerDelegate protocol methods, [318](#)
    - PasswordListViewController.h, [317](#)
  - NSFetchedResultsControllerDelegate protocol, [287](#), [298](#)
    - indexing your table, [298–301](#)
    - responding to data change, [302–305](#)
- NSFetchRequest class, [187](#)
- NSFloatAttributeType, [32](#), [126](#)
- NSInferMappingModelAutomaticallyOption, [276](#)
- NSInferMappingModelAutomaticallyOption key, [277](#)
- NSInMemoryStoreType, [39](#)
- NSInteger16AttributeType, [32](#), [126](#)
- NSInteger32AttributeType, [32](#), [126](#)
- NSInteger64AttributeType, [32](#), [126](#)



[NSInternalInconsistencyException](#), [181](#), [182](#)  
[NSManagedObject](#), [98](#)  
     CustomStore.h file, [100](#)  
     newReferenceObjectForManaged  
         Object method, [99](#)  
     nodeForReferenceObject  
         andObjectID method, [100](#)  
     updateCacheNode  
         fromManagedObject method,  
         [99](#), [100](#)  
[NSManagedObjectContext](#) class, [233](#),  
     [236](#)  
[NSManagedObjectModel](#) class, [280](#)  
[NSMigratePersistentStoresAutomaticall](#)  
     yOption, [276](#)  
[NSMigrationManager](#) class, [281](#)  
[NSNumber](#) attribute type, [126](#)  
[NSNumber](#) data type, [32](#)  
[NSRemoveEntityMappingType](#), [263](#)  
[NSSQLiteStoreType](#), [39](#)  
[NSString](#) attribute type, [126](#)  
[NSString](#) data type, [32](#)  
[NSStringAttributeType](#), [32](#), [126](#)  
[NSTransformableAttributeType](#), [32](#), [127](#)  
[NSTransformEntityMappingType](#), [263](#)  
[NSUndoManager](#) mechanism, [180](#)  
 Nullify rule, [41](#), [131](#)  
[NumberOfComponentsInPickerView](#)  
     method, [221](#)

## O

Object graph persistence framework, [48](#)  
 Objective-C world libraries. (See  
     Frameworks)  
[ObjectWithID](#) method, [41](#)  
 One-to-many relationship, [67](#)  
     destination column, [68](#)  
     inverse column, [68](#)  
     League Manager data model, [69](#)  
     Optional check box, Xcode IDE, [67](#)  
     player entity, [67](#), [68](#)  
     players relationship options, [68](#)  
     relationship information section,  
         Xcode IDE, [67](#)

    team entity, [67](#)  
     To-Many Relationship check box,  
         Xcode IDE, [67](#)  
 Optional check box, [129](#)  
 Optional property, [120](#)  
[OrgChart](#) application, [28](#), [56](#)  
     createData method, [188–191](#)  
     OrgChart.xcdatamodel, [188](#)  
     reading and outputting data,  
         [191–192](#)  
     sqlite3 command-line tool, [191](#)  
[OrgChart.xcdatamodel](#), [188](#)  
[OrgChartAppDelegate.h](#) file, [56](#)  
[OrgChartAppDelegate.m](#) file, [49](#), [55](#)  
 Owner icon, [217](#)

## P, Q

[Passwords.sqlite](#) database, [339](#)  
[PasswordViewController](#)  
     configured Password view, [332](#)  
     interface for adding and editing  
         passwords  
         methods, [333](#)  
     PasswordViewController.h, [329](#),  
         [333](#)  
     PasswordViewController.m, [330](#),  
         [331](#)  
     PasswordViewController.xib, [331](#),  
         [332](#)  
     updated tableView  
         didSelectRowAtIndexPath  
         method, [334](#)  
     updated viewDidLoad method,  
         [333](#)  
 Performance analysis, [245](#)  
     launching instruments, [245](#)  
     library, [248](#)  
     main window, [247](#)  
     template selection, [246](#)  
     test results, [249](#)  
         call tree for fetch request, [251](#)  
         pre-fetch faulting test, [250](#)  
 Performance improvement  
     using faster comparators, [241](#)  
     using subqueries, [242](#)



- Performance testing, 209
  - after test run, 223
  - building view for, 217
  - data model, 213
  - improving performance with better
    - predicates, 241
    - using faster comparators, 241
    - using subqueries, 242
  - performance analysis, 245
    - launching instruments, 245
    - test results, 249
  - PerformanceTest protocol, 218, 221, 227, 231, 238
  - PerformanceTest.h file, 218
  - performance-tuning application, 210
  - PerformanceTuning.xcdatamodel file, 213
  - PerformanceTuningAppDelegate.h file, 214
  - PerformanceTuningAppDelegate.m file, 214
  - PerformanceTuningApplicationDelegate.m file, 211–213
  - PerformanceTuningViewController.h file, 215, 220
  - PerformanceTuningViewController.m file, 216, 220, 228, 240
  - PerformanceTuningViewController.xib file, 216
  - Run Selected Test button, 218
  - single test, 222
  - @synthesize directive, 216
- PerformanceTest protocol, 218, 221, 227, 231, 234, 238
- PerformanceTest.h file, 218
- Performance-tuning application, 210
  - after test run, 223
  - building view for, 217
  - data model, 213
  - improving performance with better
    - predicates, 241
    - using faster comparators, 241
    - using subqueries, 242
  - initial run, 218
  - performance analysis, 245
    - launching instruments, 245
    - test results, 249
  - PerformanceTest protocol, 218, 221, 227, 231, 238
  - PerformanceTest.h file, 218
  - PerformanceTuning.xcdatamodel file, 213
  - PerformanceTuningAppDelegate.h file, 214
  - PerformanceTuningAppDelegate.m file, 214
  - PerformanceTuningApplicationDelegate.m file, 211–213
  - PerformanceTuningViewController.h file, 215, 220
  - PerformanceTuningViewController.m file, 216, 220, 228, 240
  - PerformanceTuningViewController.xib file, 216
  - Run Selected Test button, 218
  - single test, 222
  - @synthesize directive, 216
- PerformanceTuning.xcdatamodel file, 213
- PerformanceTuningAppDelegate.h file, 214
- PerformanceTuningAppDelegate.m file, 214
- PerformanceTuningApplicationDelegate.m file, 211–213
- PerformanceTuningViewController class, 217
- PerformanceTuningViewController.h file, 215, 220
- PerformanceTuningViewController.m file, 216, 220, 228, 240, 242, 245
- PerformanceTuningViewController.xib file, 216
- PerformCustomValidationForEntityMapping method, 264
- PerformFetch method, 12
- Persistence, 1
- Persistence in iOS, 2, 3
- Persistent data store, 215

## Persistent store

- adding, editing, and deleting Players, 84
- cancel method, 86
- clickedButtonAtIndex method, 87
- confirmDelete method, 86
- Delete button, 86
- deletePlayer method, 87, 88
- didSelectRowAtIndexPath method, 89
- initWithMasterController method, 85
- insertNewObjectForEntityForName method, 87
- insertPlayerWithTeam method, 86, 87
- insertTeamWithName method, 87
- PlayerViewController class, 84
- PlayerViewController.h file, 85
- PlayerViewController.m file, 85
- PlayerViewController.xib file, 88
- save method, 86
- showPlayerView method, 88
- TeamViewController class, 84
- UIViewController subclass, 84
- View icon, 88
- viewDidLoad method, 86
- with XIB for user interface check box, Xcode IDE, 84
- building user interface, 69
  - commitEditingStyle method, 71
  - fetchResultsController method, 71
  - insertNewObject method, 71
  - insertTeamWithName method, 69–71
  - saveContext method, 69, 71
  - timestamp attribute, 71
- configuring one-to-many relationship, 67
  - destination column, 68
  - inverse column, 68
  - League Manager data model, 69
  - Optional check box, Xcode IDE, 67
  - player entity, 67, 68

- players relationship options, 68
- relationship information section, Xcode IDE, 67
- team entity, 67
- To-Many Relationship check box, Xcode IDE, 67
- configuring table, 72
  - cellForRowAtIndexPath method, 72
  - configureCell atIndexPath method, 72
- creating custom persistent store, 94
  - atomic store, 94
  - comma-separated values (CSV) file, 94
  - initializing custom store, 95–98
  - layers in Core Data, 95
  - mapping between
    - NSManagedObject and
    - NSAtomicStoreCacheNode, 98–101
  - NSAtomicStore class, 94
  - serializing data, 101–106
  - using custom store, 106–107
  - XML persistent stores, 107–110
- creating new team, 72
  - adding new team to League Manager, 80
  - adding view controller, 73
  - aTeam parameter, 75
  - cancel method, 75
  - didSelectRowAtIndexPath method, 78
  - initWithMasterController method, 75
  - insertNewObject method, 73
  - League Manager with teams, 80
  - League Manager without teams, 79
  - save method, 75
  - showTeamView method, 78
  - TeamViewController.h file, 74, 78
  - TeamViewController.m file, 75–77
  - TeamViewController.xib file, 77
  - updated team view, 77
  - viewDidLoad method, 75, 78

- empty data model, 65
- naming project and selecting Core Data, 64
- player user interface
  - accessoryButtonTappedForRowWithIndexPath method, 83
  - cellForRowAtIndexPath method, 83
  - initWithMasterController method, 82
  - Player entities, 82
  - PlayerListViewController class, 81
  - PlayerListViewController.h file, 81, 84
  - PlayerListViewController.m file, 82
  - showPlayerView method, 81, 82
  - sortPlayers method, 83
  - UITableViewController subclass option, Xcode IDE, 81
  - UIViewController subclass, 81
  - valueForKey@players method, 82
  - viewDidLoad method, 82
  - viewWillAppear method, 82
  - with XIB for user interface check box, Xcode IDE, 81
- seeing data in, 89
  - Player entity, 89
  - Player List screen, 90
  - ZPLAYER table, 89, 91
  - ZTEAM table, 89
- selecting navigation-based application template, 64
- team and player data model, 67
- team data model, 66
- using in-memory persistent store, 92
  - caching remote information in, 93
  - League\_ManagerAppDelegate.m file, 92
  - life cycle, 93
  - persistentStoreCoordinator method, 92
- PersistentStoreCoordinator method, 259, 338
- Person entity, 124, 125
- PickerView
  - numberOfRowsInComponent method, 221
- PickerView titleForRow forComponent method, 221
- Plain Old Java Objects (POJOs), 27
- Player entity, 124, 128, 130
- Player user interface
  - accessoryButtonTappedForRowWithIndexPath method, 83
  - cellForRowAtIndexPath method, 83
  - initWithMasterController method, 82
  - Player entities, 82
  - PlayerListViewController class, 81
  - PlayerListViewController.h file, 81, 84
  - PlayerListViewController.m file, 82
  - showPlayerView method, 81, 82
  - sortPlayers method, 83
  - UITableViewController subclass option, Xcode IDE, 81
  - UIViewController subclass, 81
  - valueForKey@players method, 82
  - viewDidLoad method, 82
  - viewWillAppear method, 82
  - with XIB for user interface check box, Xcode IDE, 81
- Point entity, 261
- Polygon.m file, 261
- PredicatePerformanceTest class, 241
- PredicatePerformanceTest.h file, 242
- Predicates, 241
  - using faster comparators, 241
  - using subqueries, 242
- Pre-fetch faulting test, 250
- PreFetchFaultingTest class, 231
- Prefetching
  - PreFetchFaultingTest class, 231
  - RunWithContext method, 231
  - setRelationshipKeyPathsForPrefetching method, 231
  - SinglyFiringFaultTest.m file, 231
- Property mappings, 264
- Property section, Xcode IDE, 121

## R

Rating attribute, [213](#), [214](#)  
 ReadData method, [191](#), [193](#), [196](#)  
 Redo method, [42](#), [183](#), [184](#)  
 Redoing data objects, [180](#), [186](#)  
     adding undo to Shapes, [182](#), [186](#)  
     disabling undo tracking, [182](#)  
     limiting undo stack, [181](#)  
     undo groups, [181](#)  
 Refaulting, [225–226](#)  
 Refining result sets, [187](#)  
     aggregating, [203](#)  
     building test application  
         NSFetchRequest class, [187](#)  
         OrgChart application (see  
         OrgChart application)  
     filtering, [192](#)  
         comparison predicates, [195](#)  
         compound predicates, [198](#)  
         expressionForAggregate method,  
         [194](#)  
         single-value expressions, [193](#)  
         subqueries, [200](#)  
     sorting, [204](#)  
         multilevel sort, [206](#)  
         NSSortDescriptor, [205](#)  
         returning unsorted data, [204](#)  
 RefreshObject mergeChanges method,  
     [121](#), [225](#)  
 RefreshObject method, [225](#)  
 Reg. Ex. property, [120](#)  
 Relation, [112](#)  
 Relational database normalization, [112](#)  
     fifth normal form (5NF), [113](#)  
     first normal form (1NF), [113](#)  
     forms, [112](#)  
     fourth normal form (4NF), [113](#)  
     player attribute, [113](#)  
     relation, [112](#)  
     second normal form (2NF), [113](#)  
     third normal form (3NF), [113](#)  
     uniformColor attribute, [113](#)  
 Relational database theory, [112](#)  
 Relationship(s)  
     creating, [127](#)  
         Delete Rule field, [130](#), [132](#)

        Destination and Inverse field, [129](#)  
         Min Count and Max Count fields,  
         [130](#)  
         Name field, [128](#)  
         Optional check box, [129](#)  
         To-Many Relationship check box,  
         [130](#)  
         Transient check box, [129](#)  
         viewing and editing details, [120–121](#)  
 Relationship panel, [128](#)  
 Remove type, [263](#)  
 RemoveAllActions method, [182](#)  
 Renaming Identifier field, Xcode IDE,  
     [260](#)  
 Reset method, [42](#)  
 REST (Representation State Transfer),  
     [133](#)  
 Results option, [217](#)  
 Rollback method, [42](#)  
 RootViewController class, [9](#), [13](#)  
 Run Selected Test button, [217–223](#)  
 RunTest method, [221](#)  
 RunWithContext method, [228](#), [231](#), [235](#)

## S

Save method, [13](#), [42](#)  
 ScalarValue attribute, [276](#)  
 ScalarValue property mapping, [276](#)  
 Scale method, [143](#), [269](#)  
 .Schema command, [259](#)  
 Second normal form (2NF), [113](#)  
 SectionNameKeyPath parameter, [286](#)  
 Serializing data  
     CustomStore class, [105](#)  
     CustomStore.m file, [105](#)  
     load method, [103](#), [105](#)  
     newReferenceObjectForManaged  
         Object method, [101](#)  
     nodeForReferenceObject  
         andObjectID method, [105](#)  
     save method, [101](#), [103](#)  
 SetLevelsOfUndo method, [181](#)  
 SetRelationshipKeyPathsForPrefetching  
     method, [231](#)  
 SetRenamingIdentifier method, [260](#)

- SetRetainsRegisteredObjects method, 233
- SetUndoManager method, 42
- SetValue forKey method, 163
- Shapes 2.xcdatamodel directory, 254, 255
- Shapes 4.xcdatamodel file, 261, 270
- Shapes 5.xcdatamodel file, 270
- Shapes application
  - adding undo to, 186
  - building user interfaces, 145, 153
  - creating data models, 137, 145
  - enabling user interactions with, 154, 156
- Shapes.sqlite file, 259
- Shapes.xcdatamodel file, 137
- Shapes.xcdatamodel file, 254, 255
- ShapesAppDelegate.h file, 139
- ShapesAppDelegate.m file, 140, 152, 173
- ShapesAppDelegate.m file, 259, 277
- ShapesViewController class, 145, 148, 152
- ShapesViewController.h file, 148, 183
- ShapesViewController.m file, 149, 152, 162, 170, 183, 268, 269
- ShapesViewController.xib file, 145
- ShapeViewController.m file, 149
- SinglyFiringFaultTest class, 229
- SinglyFiringFaultTest.h file, 229, 230
- SinglyFiringFaultTest.m file, 229, 231
- Sort descriptor, 11
- SQL (Structured Query Language), 133, 143
- SQLite database, 63
  - adding, editing, and deleting Players, 84
  - cancel method, 86
  - clickedButtonAtIndex method, 87
  - confirmDelete method, 86
  - Delete button, 86
  - deletePlayer method, 87, 88
  - didSelectRowAtIndexPath method, 89
  - initWithMasterController method, 85
  - insertNewObjectForEntityForName method, 87
  - insertPlayerWithTeam method, 86, 87
  - insertTeamWithName method, 87
  - PlayerViewController class, 84
  - PlayerViewController.h file, 85
  - PlayerViewController.m file, 85
  - PlayerViewController.xib file, 88
  - save method, 86
  - showPlayerView method, 88
  - TeamViewController class, 84
  - UIViewController subclass, 84
  - View icon, 88
  - viewDidLoad method, 86
  - with XIB for user interface check box, Xcode IDE, 84
- building user interface, 69
  - commitEditingStyle method, 71
  - fetchedResultsController method, 71
  - insertNewObject method, 71
  - insertTeamWithName method, 69–71
  - saveContext method, 69, 71
  - timestamp attribute, 71
- configuring one-to-many relationship, 67
  - destination column, 68
  - inverse column, 68
  - League Manager data model, 69
  - Optional check box, Xcode IDE, 67
  - player entity, 67, 68
  - players relationship options, 68
  - relationship information section, Xcode IDE, 67
  - team entity, 67
  - To-Many Relationship check box, Xcode IDE, 67
- configuring table, 72
  - cellForRowAtIndexPath method, 72
  - configureCell atIndexPath method, 72
- creating custom persistent store, 94

- atomic store, 94
- comma-separated values (CSV)
  - file, 94
- initializing custom store, 95–98
- layers in Core Data, 95
- mapping between
  - NSManagedObject and
  - NSAtomicStoreCacheNode, 98–101
- NSAtomicStore class, 94
- serializing data, 101–106
- using custom store, 106, 107
- XML persistent stores, 107–110
- creating new team, 72
  - adding new team to League Manager, 80
  - adding view controller, 73
  - aTeam parameter, 75
  - cancel method, 75
  - didSelectRowAtIndexPath method, 78
  - initWithMasterController method, 75
  - insertNewObject method, 73
  - League Manager with teams, 80
  - League Manager without teams, 79
  - save method, 75
  - showTeamView method, 78
  - TeamViewController.h file, 74, 78
  - TeamViewController.m file, 75–77
  - TeamViewController.xib file, 77
  - updated team view, 77
  - viewDidLoad method, 75, 78
- empty data model, 65
- naming project and selecting Core Data, 64
- player user interface, 81
  - accessoryButtonTappedForRowWithIndexPath method, 83
  - cellForRowAtIndexPath method, 83
  - initWithMasterController method, 82
  - Player entities, 82
  - PlayerListViewController class, 81
  - PlayerListViewController.h file, 81, 84
  - PlayerListViewController.m file, 82
  - showPlayerView method, 81, 82
  - sortPlayers method, 83
  - UITableViewController subclass
    - option, Xcode IDE, 81
  - UIViewController subclass, 81
  - valueForKey@players method, 82
  - viewDidLoad method, 82
  - viewWillAppear method, 82
  - with XIB for user interface check box, Xcode IDE, 81
- primer, 53
  - find command, 53
  - OrgChart.sqlite file, 53
  - quit command, 54
  - schema command, 54
  - SQLite shell, 54
  - Terminal.app file, 53
  - ZORGANIZATION entity, 54
  - ZPERSON entity, 54
- seeing data in, 89
  - Player entity, 89
  - Player List screen, 90
  - ZPLAYER table, 89, 91
  - ZTEAM table, 89
- selecting navigation-based application template, 64
- team and player data model, 67
- team data model, 66
- using in-memory persistent store, 92
  - caching remote information in, 93
  - League\_ManagerAppDelegate.m file, 92
  - life cycle, 93
  - persistentStoreCoordinator method, 92
- SQLite3 application, 259
- SQLite3 command-line tool, 191
- Storing data. (See SQLite database; Core Data:persistent store)
- String attribute type, 32, 126

Studio entity, [213](#)  
 Studio table, [213](#)  
 Subqueries, [242](#)  
 SubqueryTest class, [243](#)  
 SubqueryTest.m file, [244](#)  
 @synthesize directive, [216](#)

## T

Table, configuration, [72](#)  
 Table views,  
     NSFetchedResultsController  
       class, [285](#)  
     cache name, [287](#)  
     delegates, [287–288](#)  
     fetch request, [286](#)  
     implementing, [293–297](#)  
     implementing in League Manager,  
       [288](#), [292](#)  
     implementing  
       NSFetchedResultsControllerDel  
       egate protocol (see  
       NSFetchedResultsControllerDel  
       egate protocol)  
     managed object context, [286](#)  
     section name key path, [286](#)  
 tableView CommitEditingStyle  
     forRowAtIndexPath method,  
       [321](#)  
 tableView didSelectRowAtIndexPath  
     method, [327](#)  
 Team entity, [120](#), [128](#), [287](#)  
 Testing, [209](#)  
     building application for, [209](#)  
       adding framework to application,  
         [220–222](#)  
       creating Core Data project, [210](#)  
       creating data and data model,  
         [213–215](#)  
       creating testing view, [215–217](#)  
       framework, [218–219](#)  
       Run Selected Test button, [209](#)  
       running initial test, [222–223](#)  
     results, [249](#)  
       call tree for fetch request, [251](#)  
       pre-fetch faulting test, [250](#)

Testing framework  
     adding framework to application,  
       [220–222](#)  
     building, [218–219](#)  
 TestPicker option, [217](#)  
 Tests group, [218](#)  
 Third normal form (3NF), [113](#)  
 TimeStamp attribute, [9](#), [65](#)  
 To-Many Relationship check box, [130](#)  
 Transform entity, [263](#), [266](#)  
     scalarValue attribute, [266](#), [269](#)  
     scale attribute, [266](#), [269](#)  
 Transform type, [263](#)  
 Transform.h file, [269](#)  
 Transform.m file, [269](#)  
 Transformable attribute type, [32](#), [127](#)  
 TransformToTransform entity mapping  
     model, [275](#)  
 TransformToTransform mapping, [275](#)  
 Transient check box, [129](#)  
 Transient property, [120](#)  
 Type property, [120](#)

## U

UIColorTransformer class, [172](#)  
 UIColorTransformer.h file, [173](#)  
 UIColorTransformer.m file, [172](#)  
 Undo method, [42](#)  
 Undoing data objects, [180](#), [186](#)  
     adding undo to Shapes, [182](#), [186](#)  
     disabling undo tracking, [182](#)  
     limiting undo stack, [181](#)  
     undo groups, [181](#)  
 UndoManager method, [42](#)  
 UniformColor attribute, [65](#)  
 Uniquing, [237](#)  
     data inconsistency elimination, [238](#)  
     definition, [237](#)  
     memory conservation, [238](#)  
     PerformanceTest protocol, [238](#)  
     UniquingTest class, [238](#)  
     UniquingTest.h file, [238](#), [240](#)  
     UniquingTest.m file, [239](#)  
 UniquingTest class, [238](#)  
 UniquingTest.h file, [238](#), [240](#)

UniquingTest.m file, [239](#)  
 Universally unique identifiers (UUIDs), [97](#)  
 UpdateUndoAndRedoButtons method, [184](#)  
 Use Core Data for storage check box, [16](#)  
 User interface building, [69](#)  
     commitEditingStyle method, [71](#)  
     fetchedResultsController method, [71](#)  
     insertNewObject method, [71](#)  
     insertTeamWithName method, [69–71](#)  
     saveContext method, [69, 71](#)  
     timeStamp attribute, [71](#)  
 User interface visualization, League Manager application, [60](#)  
     Add/Edit Player screen, [62, 63](#)  
     Add/Edit Team screen, [60, 61](#)  
     Player List screen, [61, 62](#)  
     Team List screen, [60](#)  
 Using NSFetchedResultsController, [288](#)

## V

ValidateValue forKey error method, [176, 177](#)  
 Validating data  
     custom validation, [175–179](#)  
     default values, [179–180](#)  
     invoking validation, [179](#)  
 Validation, [120](#)  
 ValueForKey method, [143, 161, 162, 224](#)  
 Variable type, [303](#)  
 Versioning, [253, 254](#)  
     current version data model, [254](#)  
     data model with two versions, [255](#)  
     single-version data model, [254](#)  
     version name and model basis selection, [255](#)  
     Xcode window, [256](#)  
 Versioning section, Xcode IDE, [260, 261](#)  
 Vertex entity, [261](#)  
 View controller, [9](#)  
 ViewDidAppear method, [155](#)  
 ViewDidLoad method, [152, 167, 184, 220, 228, 269](#)  
 ViewDidUnload method, [184](#)  
 VSAM (Virtual Storage Access Method), [133](#)

## W

WillTurnIntoFault method, [226](#)

## X, Y

Xcode data modeler, [113](#)  
     graph view, [116, 117](#)  
     League Manager model, [116, 117](#)  
     mapping models, [114](#)  
     new file, [114](#)  
     NSManagedObject class, [117](#)  
     Player entity, [117](#)  
     renaming entities and properties, [260](#)  
     team relationship, [119](#)  
     user interface, [114, 115](#)  
     using fetched properties, [121](#)  
     viewing and editing attribute details, [119](#)  
         default value property, [120](#)  
         Description column, [119](#)  
         detail pane, [119](#)  
         indexed property, [120](#)  
         max length property, [120](#)  
         max value property, [120](#)  
         min length property, [120](#)  
         min value property, [120](#)  
         Name column, [119](#)  
         name property, [120](#)  
         optional property, [120](#)  
         reg. ex. property, [120](#)  
         transient property, [120](#)  
         type property, [120](#)  
         validation, [120](#)  
     viewing and editing relationship details, [120, 121](#)  
 Xcode IDE, [5](#)



XML persistent stores, 107  
    addPersistentStoreWithType  
        method, 107  
    iOS, 108  
    NSPersistentStoreCoordinator.h file,  
        108  
    NSXMLStoreType undeclared, 108  
    porting League Manager data model  
        to Mac OS X Core Data  
        application, 109, 110

## Z

ZPOINT table, 262  
ZSHAPE table, 259, 277  
    ZHEIGHT column, 277  
    ZWIDTH column, 277  
ZTRANSFORM table, 277  
ZVERTEX table, 262