Core Frameworks Series

Developing Data-Driven Applications for the iPad®, iPhone®, and iPod touch®

# Core Data for iOS

tim ISTED
tom HARRINGTON

Tim Isted
Tom Harrington

# Core Data for iOS

## Developing Data-Driven Applications for the iPad®, iPhone®, and iPod touch®

Addison-Wesley

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

*Library of Congress Cataloging-in-Publication Data is on file.*



ISBN-13: 978-0-321-67042-7

ISBN-10: 0-321-67042-6

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana

First printing June 2011

**Editor-in-Chief**
Mark Taub

**Senior Acquisitions Editor**
Chuck Toporek

**Development Editor**
Chuck Toporek

**Managing Editor**
Kristy Hart

**Project Editor**
Andy Beaster

**Indexer**
Larry Sweazy

**Proofreader**
Jennifer Gallant

**Technical Reviewers**
Jim Correia
Robert McGovern
Mike Swan

**Publishing Coordinator**
Olivia Basegio

**Interior Designer**
Gary Adair

**Cover Designer**
Chuti Prasertsith

**Compositor**
Gloria Schurick

# Table of Contents

# Preface

We live in a data-driven world. We consume social data, like email, Twitter, and Facebook, business data, like share prices, financial forecasts, and bank accounts, and occasionally we might have a little fun with the more recreational side to life, like brainteasers, or games involving squawking birds and mock air traffic control, where we expect to be able to track our progress and rejoice when we beat our previous high scores.

As mobile devices increase in performance, capacity, and capability, we place ever-increasing demands on our phones or tablet devices to consume, save, fetch, search and display our data. Consumers buy iPhones, iPod touches and iPads with storage capacities unheard of in handheld or even desktop devices only a few years ago, and they expect to fill those capacities either with media, or with applications and data.

It's increasingly difficult to imagine an application with any non-trivial functionality that doesn't maintain at least some kind of data store. Even if a Twitter client maintains only a temporary store of downloaded tweets, it will at least need to keep permanent track of one or more Twitter account usernames for timeline refresh; calculator applications have persistent memories for calculated values, or store a history of previous calculations; and, in order for us to feel a sense of achievement, games store a history of high scores, as well as a state of play so that we can return immediately to our gun-slinging 3D shoot-'em-up just as soon as we finish our FaceTime chat.

For us mobile app developers, the demands are high. Not only do users expect our apps to store data as efficiently as possible, they expect their applications to run quickly, smoothly, and without crashing. Given the relatively limited runtime memory capacities of these mobile devices, dealing with persistent data can quickly become a nightmare. Mobile devices also introduce the issue of power management, which is rarely a concern when writing software for desktop computers. An app that eats batteries will not please its users.

When the iPhone SDK first launched, developers were left to fend for themselves when it came to data access. Data persistence was possible only via basic file storage, or through direct access to a SQLite database. SQLite can certainly help out with the limited runtime memory problems, but generally requires developers to fashion their own persistence layer to interact with model objects and generate the underlying SQL commands necessary to save and restore plain data.

Core Data changes all that. From version 3.0 of the iPhone SDK (renamed to iOS as of version 4.0) onwards, Apple provides us with a ready-made data persistence layer. We define a schema for the model objects we want to store, then leave Core Data to figure out what to do to persist our model data to disk. We don't have to worry about low-level SQL commands, or the memory meltdowns involved when loading 1GB of data from a single file. Instead we work with a "data store," a term that is intentionally vague since details of managing data files are abstracted away. Developers are free to work with their objects and leave the file management to the framework.

If Core Animation is the sexy framework for views, we now have an equally sexy (though possibly less visually glamorous) framework to help us with our model. By taking away much of the drudgery of data persistence, we're left with more time to work on the features and functionality unique to our applications.

Core Data can be the perfect answer to many data-related prayers, but it comes with a steep initial learning curve. Because of its ability to work easily with a SQLite database for its storage, it's often mistaken for a database itself and, although this is inaccurate, it certainly helps to have a basic understanding of general database terms and techniques.

Although the number of Core Data classes is relatively small, it's necessary to make use of most of them before you can do anything at all with the framework. It's hard to understand terms like "managed object context" before you understand "managed object," or "persistent store," but in order to make use of a managed object, you need a managed object context, and a persistent store. Getting over the initial leap of faith in a basic Core Data stack can seem a sizeable obstacle in making use of the framework.

This book teaches Core Data from the ground up. You'll learn about these primary classes in the framework, seeing how they interact to provide amazing functionality with very little configuration and tweaking. You'll find out how to store and fetch data, look at best practices for providing data to the staple view of many data-driven apps, `UITableView`, and discover how easy it can be to perform data validation to ensure data integrity. Finally, you'll look at ways to troubleshoot your Core Data applications, or enhance data-related performance bottlenecks.

By the end of the book, you'll have a thorough understanding of the framework and its classes, and probably be left wondering how you ever managed without it.

## Audience for This Book

Aimed at intermediate to advanced iOS developers, the book assumes that you have a reasonable working knowledge of programming iOS applications. In particular, you should be comfortable working with Apple's basic developer tools (Xcode), the Objective-C language, and the Cocoa Touch framework.

It is not assumed that you have already worked with Core Data on the desktop, although the vast majority of the information included in this book applies to Core Data in general, both on iOS devices, and under Mac OS X (10.4 or later). Once you've mastered Core Data on iOS, you'll be able to use the same tools (Xcode's data modeler) and much of the same code (with the exception of `NSFetchedResultsController`, which is iOS-only) to build Core Data applications on the desktop.

## Who Should Read this Book

If you write iOS applications, you'll probably have data persistence needs. If you need to work with anything other than the most trivial data storage, you'll likely find it easier to work with Core Data than to create your own file-based or low-level SQLite-based persistence layer. If you need to work with Core Data, you should read this book.

If you've never used Core Data before, this book will teach you what you need to know to get started. Once you've mastered the fundamentals of the framework, including the iOS-specific `NSFetchedResultsController`, and walked through the construction of a complete Core Data-based application, you'll find performance tips and troubleshooting information.

If you've already been using Core Data for a while, and keep wondering why your app crashes when you work with large numbers of model objects, or can't figure out why you're suffering a performance hit in certain situations, this book will help clear up any mysteries with the fundamentals of the framework, and help you use Apple's developer tools to isolate the sources of those problems.

## Who Shouldn't Read This Book

Core Data is definitely not the easiest to understand of the Cocoa Touch frameworks. If you've never created an iOS application before, or struggle to remember the difference between a `UIView` and a `UIViewController`, you'll have problems working through this book.

Although the first few chapters aim to flatten the learning curve as much as possible, it's assumed that you have a solid understanding of both the Objective-C language, and the Cocoa Touch framework. If you don't, you'd be better looking at a suitable introductory iOS programming book.

It's also worth noting that Core Data is commonly mistaken for a database. Although Core Data can use SQLite, it's not by any means a SQLite wrapper nor is it designed for typical database usage. If what you really need is a database, Core Data may not be the right solution and this book might not be appropriate.

Finally, if you're looking for creative visual interface inspiration, or suggestions for stunning data representations, you might prefer to look elsewhere. The sample applications in this book are specifically designed with interfaces that are as simple as possible to help you learn the Core Data framework with minimal distractions. For this reason, all the sample projects are for iPhones or iPod touches, and don't include iPad-specific resources. Equally, if you're looking for tips to beautify table views or design jaw-dropping custom views, you'll likely be disappointed.

Do bear in mind, however, that just because many of the sample projects in this book make use of the more traditional data display controls (i.e., table views), you can use Core Data in any situation where you need easy and efficient access to data. Need to store a list of high scores to draw in an OpenGL view for a game? Core Data makes that easy (well, the storage part anyway). Need to store enough information to draw icons that represent the weather for the next 24 hours? Core Data can help with that too.

## What You Need to Know

The book also assumes that you're familiar and comfortable with Xcode and programming in Objective-C. You won't find any primers on how to define a method, or how to install and launch Xcode; there are plenty of entry-level books for newbies and converts from other platforms and programming environments, and if you're messing around with data models and such, we can assume that you've already got that grounding.

As with any iOS development, you'll need at least a free Apple developer account. To test your applications on real devices, or sell on the App Store, you'll need a paid iOS developer account ($99 per year at the time of writing). Go to **developer.apple.com/devcenter/ios** to register for access to all of the relevant updates for iOS, as well as Xcode, developer documentation, sample code, and even the session videos from Apple's annual World Wide Developer Conference. Without a paid account it's still possible to develop iOS code, but you'll only be able to run this code on the iPhone Simulator. While the Simulator is extremely useful, it's no substitute for getting your code on a real iOS device.

Core Data can support a number of different persistent store types, the inner workings of which it mostly hides from the developer. By far the most common type on iOS devices is the SQLite store, which saves persistent data into a SQLite database. You don't need to know anything about SQLite to read this book, but if you're already a database super-user, you'll probably know that there is some controversy about the pronunciation of SQLite. D. Richard Hipp, the creator of SQLite, pronounces it "like a mineral", pronouncing Ess-Queue-Ell-Ite as one might pronounce "pyrite" or "kryptonite".[1] Hipp does not insist on this pronunciation though, and in practice the vast majority of Mac and iOS developers we've encountered pronounce it Sequel-ite. For this reason we've chosen the latter pronunciation, so you'll find we talk about "a SQLite store" rather than "an SQLite store."

## How This Book is Organized

The book offers a comprehensive discussion of Apple's Core Data framework as it applies on iOS devices, building a firm grounding in the subject before covering more advanced and real-world examples of its use. Many of the chapters in the book are divided into two parts—you start by learning the relevant information, and then cement your understanding by putting the knowledge into practice with a sample project.

Chapter 10 walks you through the complete construction of a Core Data-based note taking application, from start to finish. If you want to jump straight in and find out what's possible with Core Data, you might like to begin with Chapter 10 to whet your appetite, then return to the beginning of the book to find out how it all works.

---

[1] *Hipp discussed "SQLite" pronunciation at the C4[2] conference in September 2008. His presentation can be viewed at* ***http://www.viddler.com/explore/rentzsch/videos/25/***

- **Part I: Introduction**

  Apple's Core Data framework presents a unified and powerful solution to storing an application's data. This book offers a comprehensive reference for the framework and its use in versions of iOS from iPhone SDK 3.0 onwards. As well as covering Core Data basics, this section discusses more general topics like object modeling and data persistence, and demonstrates how to build an object model using Xcode's data modeling tool.

  - **Chapter 1: An Overview of Core Data on iOS Devices**

    This first chapter introduces Core Data as a framework to fit into the MVC-pattern for development of applications for iOS. It gives a brief outline of its history as the Enterprise Objects Framework for web development, before discussing when, how and why Core Data is useful. It explains how there is little difference between working with Core Data on the desktop and on iOS, with the notable exception of the lack of support for Bindings on iOS. The overview finishes by showcasing a few real-world examples of Core Data use in publicly-available iOS applications, including MoneyWell for iPhone, Calcuccino and the Associated Press news application.

  - **Chapter 2: A Core Data Primer**

    Having given a high-level overview in Chapter 1, this chapter delves deeper and introduces the key features in Core Data, covering the interaction between Managed Object Contexts, Managed Objects and the underlying Persistent Stores. It also introduces the framework classes behind these and explains how impressive functionality can be achieved with very little code, often without the need to subclass the basic framework classes. The chapter continues by explaining the process of writing applications that use Core Data. It concludes by taking a look at what is happening behind-the-scenes in Apple's Xcode template projects for Core Data iOS applications.

  - **Chapter 3: Modeling Your Data**

    This chapter introduces general ideas behind data modeling. Having expressed clearly that Core Data is not in itself a database, the chapter does discuss basic relational database techniques and relevant best practices (for example, data normalization that applies to object model design). This chapter also explains how data stored in a relational database can be mapped into a relational object model for use in object-oriented programming languages and concludes with a demonstration of how an object model is defined for Core Data using the Data Model editor in Xcode.

- **Part II: Working with Core Data**

  The second part of the book focuses on a discussion of topics that apply to most applications wanting to make use of Core Data on iOS. Each facet of the framework or its related technologies is given a separate chapter so it is possible either to read this part of the book in order, building knowledge in incremental steps, or to pick

out the chapters that are of particular interest. Each of these chapters is divided into two sections: the first introduces the particular feature or functionality, discusses why and when it might be useful, then walks through the relevant classes and methods; the second section is written in a tutorial format that starts by adding a basic feature to a simple application, before building on more advanced functionality. The aim of these tutorial sections is to enable you to learn by doing, but in such a way that you relate the same techniques to your own applications.

- **Chapter 4: Basic Storing and Fetching**

  This chapter guides you through the process of building a simple iPhone application that uses a `UITableView` to display managed objects. It includes more information about managed object contexts, and how they relate to the underlying data—a good understanding of managed object contexts is absolutely fundamental to using Core Data effectively. This chapter explains what contexts are, how to use them, where a context 'comes from' and how they interact with each other and the data store. The project for this chapter features a simple Add button to add objects that are fetched and displayed in a table view; to keep it simple for now, each object is pre-populated with randomly-generated information.

- **Chapter 5: Using NSFetchedResultsController**

  This chapter demonstrates how to make use of the Fetched Results Controller, an object unique to iOS, to handle much of the functionality necessary to fetch and display objects in a table view. It explains why memory usage is so important on iOS, and how a fetched results controller works to keep to a minimum the number of objects held in memory at any one time.

- **Chapter 6: Working with Managed Objects**

  This chapter introduces the functionality provided by `NSManagedObject`, such as basic data validation. Although it frequently isn't necessary to subclass `NSManagedObject`, this chapter explains why, when and how to do so. You'll learn about the lifecycle of managed objects, and look at different types of modeled properties. The chapter covers features offered by Objective-C 2.0 to simplify accessor method code and finishes by looking at custom validation logic.

- **Chapter 7: Working with Predicates**

  This chapter begins with the basics of creating an NSPredicate, discussing simple predicate format strings. You'll learn how to use predicates to match against scalar values like numbers or dates, and also how to match objects, particularly across relationships, such as when fetching employees who work in a specific department. There's a whole section dedicated to working with strings, including information on case sensitivity, and you'll see how to examine the raw SQL that Core Data generates to query a SQLite store.

- **Chapter 8: Migration and Versioning**

  This chapter looks at how to use the provided versioning and migration functionality to maintain compatibility between old and new versions of an application's data model. By default, an application built around a newer model version won't be able to open an older version's model; through using automatic migration, the user can continue to work with their old data even after an application upgrade has occurred. You'll learn about both simple migration, where the Core Data framework itself works out how one data model version relates to another, as well as custom migration using mapping models and entity migration policies.

- **Chapter 9: Working with Multiple View Controllers and Undo**

  To keep the examples as simple as possible, and to minimize distractions, the previous projects up to this point have made use of only a single view controller. In this chapter, you'll see how to keep track of managed object contexts across multiple view controllers, and how to use editing view controllers to change values on existing managed objects. You'll learn how to work with multiple managed object contexts, and find out how to refer to managed objects across these multiple contexts, before finding out how simple it is to make use of the automatic Undo functionality provided by Core Data.

- **Part III: Building a Simple Core Data Application**

  The third part of the book takes the reader through building a complete application using Core Data.

  - **Chapter 10: Sample Application: Note Collector**

    This chapter puts your Core Data knowledge into context by walking through the creation of a more substantial application than you've worked with so far. You'll see how to work with abstract entities, entity inheritance and multiple view controllers to create a fully functional note-taking application that stores notes and organizes them in collections.

    You'll learn how to examine a raw SQLite file to peek at what Core Data is doing, and find out how to include a pre-populated data store so that users of the application see some sample data when they launch the application for the first time. You'll also look at one way to persist application state across launches, seeing how to archive the managed object information necessary to recreate a navigation-based stack of view controllers.

- **Part IV: Optimizing and Troubleshooting**

  The final part of the book looks at performance issues, optimization for the restricted memory requirements of iOS devices, and at debugging tools to aid in developing with Core Data on iOS.

- **Chapter 11: Optimizing for iOS Performance and Memory Requirements**

  This chapter is all about performance, optimization, and speed. You'll learn some simple tricks to help your application run faster and be more responsive for the user without consuming all available memory or running down the battery. This chapter assumes you already understand about retain counts and when objects are deallocated, which affects your memory usage but which are not directly related to Core Data.

- **Chapter 12: Troubleshooting Core Data**

  When things go amiss with Core Data the symptoms and error messages can seem obscure, even if you've been using it for a while. You can't very well fix your code if you don't understand what's wrong. In this chapter, you'll look at ways to help you diagnose and fix some of the most common Core Data problems. Keep in mind that Core Data can be affected by problems that are not specific to Core Data; for example, memory management errors can affect any Cocoa object, and managed objects are no exception. This chapter focuses on problems specifically related to Core Data.

Although the book is designed to be read in order, each chapter is mostly self-contained, so feel free to skip around to learn about specific topics. Some of the example projects in each chapter require code from a previous chapter as a starting point; if you need to grab a ready-made project from an earlier chapter, the sample code for the book is available online.

## About the Sample Code and Coding Style

All of the source code necessary to run the examples in this book is provided inline within chapters; in order to fit within the confines of a page, the code may have rather more newline characters than you might expect.

Because of the nature of the subject, the code includes a large number of accessor methods. As this book is likely to be read both by developers who prefer using full accessor methods and lots of nested square brackets, as well as those who have embraced Objective-C 2.0 dot syntax, we felt it important to include examples demonstrating both styles. The included code therefore uses a mixture of both traditional method calls and dot notation throughout the example listings. Feel free to substitute according to your own coding preferences.

The complete source code for the projects in this book is available as a downloadable disk image (.dmg), which you can get by clicking on the Resources tab on the book's catalog page:

*http://www.informit.com/title/9780321670427*

The disk image contains a README file along with folders containing the projects for each chapter.

Apple shipped Xcode 4 (with substantial changes over Xcode 3) just before this book went

to press. The screenshots in the book are taken from Xcode 4, but if you're still using Xcode 3, it should be relatively straightforward to work out any differences. We've added Xcode 3-specific instructions in the text anywhere that there might be confusion over major differences.

Although Mac OS X Lion had been announced, it hadn't yet shipped publicly when this book was published, so the screenshots are taken from Mac OS X Snow Leopard, which is the current required environment for iOS (i.e., iPhone, iPad, and iPod touch) development. For Core Data development for iOS, you don't need anything else: all the libraries, headers, and documentation are included with the Xcode tools and the iOS SDK.

# Acknowledgments from Tim Isted

Although writing a book notoriously takes longer than expected, I've certainly pushed the boundaries on this one. I have a vivid memory of the moment the words "Core Data" appeared on a slide at Apple's announcement of iPhone SDK 3.0. Half an hour later, Chuck and I were discussing the outline for a book dedicated to Core Data on iPhone. That was back in June 2009.

Since then, the iPhone OS has become iOS, the iPad was released, iPhone 4 appeared, multitasking was introduced, Xcode 4 went public, and the goal posts kept moving. It's hard to pick a time to publish a book on something that changes so frequently, but putting overall iOS changes aside, the Core Data framework (and certainly its API) has remained fairly stable, probably due to its earlier existence on the Mac. This book would never have made it were it not for the wonderfully patient and encouraging editorial guidance Chuck Toporek has given me, not to mention his personal friendship. Together we hope we've ensured it should remain useful across the inevitable series of major iOS version releases that will occur the moment the book hits the shelves.

After a few lengthy pauses for me to deal with various nasty bouts of illness among my close family, Tom Harrington agreed to come on board to help get the book out before iOS became obsolete. His work specifically on the two performance chapters, together with his contributions across the whole book, has taken it up so many notches.

The four anonymous (for the most part) technical reviewers have been fantastic. It's all too easy to become blinkered as an Indie developer and I thank the reviewers for saving me from too many of those "I've always done it like this" moments.

Finally, I can never thank enough the friends I have throughout the Mac/iOS developer community. It's a wonderful team to be a part of.

# Acknowledgments from Tom Harrington

I'd like to thank my wife Carey for encouraging me to embark on a career that appealed to me but seemed too risky to jump into. After the dot-com boom in 2001, Carey was the one who suggested there might be more interesting things to do than look for another day job. I never expected to run my own business and would not have done so without Carey's help. I've been independent ever since and have never looked back.

I'd also like to thank Tim Isted and Chuck Toporek for giving me the opportunity to work on this book, and to Marcus Zarra for introducing me to Tim in the first place. Also, this book would not have been possible without the technical reviewers who help make Tim and I look good.

# About the Authors

## Tim Isted

Tim Isted has been writing software for Macintosh computers since 1995. He also builds web applications using Rails, PHP, and .NET and has been known to develop for Windows machines too. Also a professional musician and singing teacher, he tries to divide his time fairly equally between conducting, accompanying, teaching, and writing software. Previous musings on Core Data for desktop development can be found on his blog at **www.timisted.net**, and he is also co-organizer of NSConference, a new Mac developer conference taking place in both Europe and the USA.

## Tom Harrington

Tom Harrington switched from writing software for embedded systems and Linux to Mac OS X in 2002 when he started Atomic Bird, LLC. After six years of developing highly regarded Mac software he moved to iPhone in 2008. He develops iOS software on a contract basis for a variety of clients. Tom also organizes iOS developer events in Colorado. When not writing software he can often be found on his mountain bike. His website is **www.atomicbird.com**.

CHAPTER 2

# A Core Data Primer

IN THIS CHAPTER

- Persisting Objects to Disk
- The Core Data Approach
- Examining the Xcode Core Data Templates

In order to get the most out of Core Data, it's extremely important to have a firm understanding of its fundamental operations. Over the course of this chapter, you'll learn the key terms and features of the different parts of a Core Data-based application.

Before looking at the Core Data terms, though, take a moment to think about how you might work with persisted data in an application *without* using Core Data.

## Persisting Objects to Disk

When you're working with data to be saved in an application, you typically have collections of objects, maybe held in arrays, sets or dictionaries, which need to be archived to disk. When it comes time to save the data, you might encode or serialize those objects ready to be saved into a binary file or, for small datasets, store them in a `.plist` file.

As an alternative to working with binary files, and before Core Data came to iOS, developers could also make direct use of SQLite, a simple and very lightweight database, available on iOS devices since the early versions of iPhone OS. When writing an application that made use of large collections of objects, it would make sense to store those items in a database, offering huge increases in speed when saving and fetching objects.

SQLite, as its name implies, is based around the *Structured Query Language*, or SQL. You talk to an SQL database by issuing commands to, for example, *insert* or *select* (fetch) data. If you only need a specific object from the database, you can issue a command to fetch just that object; you

don't need to worry about the efficiency and performance issues with loading an entire binary file from disk just to get hold of a particular object.

In order to work with SQLite, however, you need to make heavy use of procedural C APIs, writing lengthy portions of code to handle data access. To save an object into a SQLite database, for example, you would need to write out a string containing an SQL `INSERT` statement, populate that string with the values held by the object's instance variables, convert the string to a C-string, before finally passing it to a C function.

# The Core Data Approach

Core Data, on the other hand, combines all the speed and efficiency of database storage with the object-oriented goodness of object serialization.

## Entities and Managed Objects

When you create your model objects, instead of starting out by writing the `.h @interface` for the class, you typically begin by modeling your *entities*, using the Xcode Data Modeler. An entity corresponds to one *type* of object, such as a `Patient` or `Doctor` object, and sets out the *attributes* for that entity, such as `firstName` and `lastName`. You use the data modeler to set which attributes will be persisted to disk, along with various other features such as the type of data that an attribute will hold, data validation requirements, or whether an attribute is optional or required.

When you work with actual instances of model objects, such as a specific `Patient` object, you're dealing with an instance of a *managed* object. These objects will either be instances of the `NSManagedObject` class, or a custom subclass of `NSManagedObject`. If you don't specify a custom subclass in the modeler, you would typically access the attributes of the object through *Key Value Coding* (KVC), using code like that in Listing 2.1.

LISTING 2.1 Accessing the attributes of a managed object

```
NSManagedObject *aPatientObject; // Assuming this has already been fetched

NSString *firstName = [aPatientObject valueForKey:@"firstName"];
NSString *lastName = [aPatientObject valueForKey:@"lastName"];

[aPatientObject setValue:@"Pain killers" forKey:@"currentMedication"];
[aPatientObject setValue:@"Headache" forKey:@"currentIllness"];
```

If you choose to do so, you can also provide your own subclass of `NSManagedObject`, to expose *accessor methods* and/or *properties* for your managed object, so you could use the code shown in Listing 2.2. You'll look at this in more detail in Chapter 6, "Working with Managed Objects."

LISTING 2.2 Using a custom subclass of NSManagedObject

```
Patient *aPatientObject; // Assuming this has already been fetched

NSString *firstName = [aPatientObject firstName];
NSString *lastName = aPatientObject.lastName;

[aPatientObject setCurrentMedication:@"Pain killers"];
aPatientObject.currentIllness = @"Headache";
```

You could also still access the values of the object using `valueForKey:`, etc., if you wish.

## Relationships

The Data Modeler is also the place where you define the *relationships* between your entities. As an example, a `Patient` object would have a relationship to a `Doctor`, and the `Doctor` would have a relationship to the `Patient`, as shown in Figure 2.1.

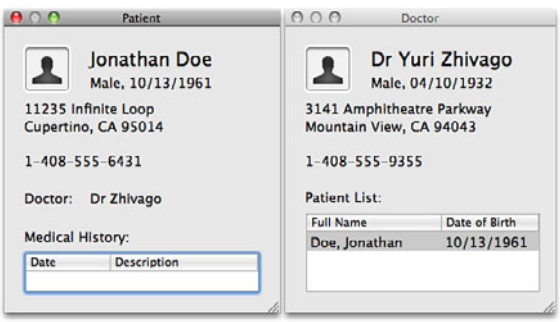


FIGURE 2.1 A Patient-Doctor relationship

When modeling relationships, you typically think in relational database terms, such as *one-to-one*, *one-to-many*, and *many-to-many*. In the example shown in Figure 2.1, a patient has only *one* doctor, but a doctor has *many* patients, so the doctor-patient relationship is one-to-many.

If the doctor-patient relationship is one-to-many, the *inverse* relationship (patient-doctor) is obviously *many-to-one*. When you model these relationships in the Data Modeler, you need to model them *both*, explicitly, and set one as the inverse of the other. By setting the inverse relationship explicitly, Core Data ensures the integrity of your data is automatically maintained; if you set a patient to have a particular doctor, the patient will also be added to the doctor's list of patients without you having to do it yourself.

You specify a *name* for each relationship, so that they are exposed in a similar way to an entity's attributes. Again, you can either work with KVC methods, or provide your own accessors and property declarations in a custom subclass, using code like that in Listing 2.3.

LISTING 2.3 Working with relationships

```
Patient *aPatientObject; // Assuming this has already been fetched
Doctor *aDoctorObject = [aPatientObject valueForKey:@"doctor"];

Patient *anotherPatientObject; // also already fetched
anotherPatientObject.doctor = aDoctorObject;
   // The inverse relationship is automatically set too

NSLog(@"Doctor's patients = %@", [aDoctorObject patients]);
/* Outputs:
      Doctor's patients = (
                              aPatientObject,
                              anotherPatientObject,
                              etc...
                          )
*/
```

It's important to note that Core Data doesn't maintain any *order* in collections of objects, including to-many relationships. You'll see later in the book how objects probably won't be returned to you in the order in which you input them. If order is important, you'll need to keep track of it yourself, perhaps using an ascending numerical index property for each object.

If you're used to working with databases such as MySQL, PostgreSQL, or MS SQL Server (maybe with web-based applications in Ruby/Rails, PHP, ASP.NET, etc.), you're probably used to every *record* in the database having a unique `id` of some sort. When you work with Core Data, you don't need to model any kind of unique identifier, nor do you have to deal with join tables between related records. Core Data handles this in the background; all you have to do is to define the relationships between objects, and the framework will decide how best to generate the underlying mechanisms, behind the scenes.

## Managed Object Contexts

So far, the code in this chapter has assumed that you've fetched an object "from somewhere." When you're working with managed objects and Core Data, you're working within a certain *context*, known as the *Managed Object Context*. This context keeps track of the persistent storage of your data on disk (which on iOS is probably a SQLite store) and acts as a kind of container for the objects that you work with.

Conceptually, it's a bit like working with a document object in a desktop application—the document represents the data stored on disk. It loads the data from disk when a document is opened, perhaps allowing you to display the contents in a window on screen. It keeps track of changes to the document, likely holding them in memory, and is then responsible for writing those changes to disk when it's time to save the data.

The Managed Object Context (MOC) works in a similar way. It is responsible for fetching the data from the store when needed, keeping track of the changes made to objects in memory, and then writing those changes back to disk when told to save. Unless you specifically tell the MOC to save, any changes you make to any managed objects in that context will be temporary, and won't affect the underlying data on disk.

Unlike a normal document object, however, you are able to work with more than one managed object context at a time, even though they all relate to the same underlying data. You might, for example, load the same patient object into two different contexts, and make changes to the patient in one of the contexts (as shown in Figure 2.2). The object in the other context would be unaffected by these changes, unless you chose to save the first context. At that point, a notification would be sent to inform you that another context had changed the data, and you could reload the second context if you wanted to.

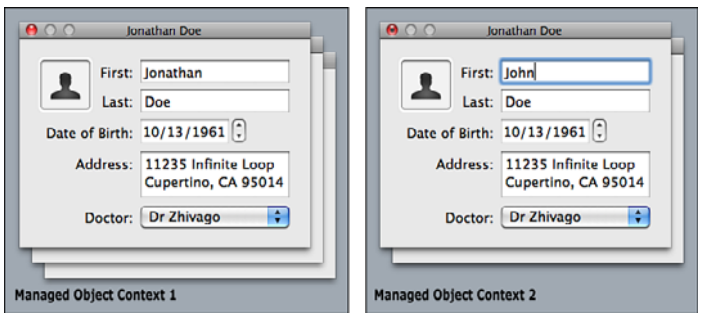


FIGURE 2.2 Managed Object Contexts and their Managed Objects

Although it's less common to work with multiple contexts on iOS than it is on the desktop, you typically use a separate context if you're working with objects in the background, such as pulling information from an online source and saving it into your local app's data. If you choose to use the automatic Undo handling offered by managed object contexts, you might set up a second context to work with an individual object, handling undo for any changes to individual attributes as separate actions. When it was time to save that object back into your primary context, the act of saving all those changes would count as one undo action in the primary context, allowing the user to undo all the changes in one go if they wanted to. You'll see examples of this in later chapters.

## Fetching Objects

The managed object context is also the medium through which you fetch objects from disk, using `NSFetchRequest` objects. A fetch request has at minimum the name of an entity; if you wanted to fetch all the patient records from the persistent store, you would create a fetch request object, specify the `Patient` entity to be retrieved, and tell the MOC to execute that fetch request. The MOC returns the results back to you as an array. Again, it's important to note that the order in that array probably won't be the same as the order in which you stored the objects, or the same as the next time you execute the fetch request, unless you request the results to be sorted in a particular order.

To fetch specific objects, or objects that match certain criteria, you can specify a *fetch predicate*; to sort the results in a certain order, you can provide an array of *sort descriptors*. You might choose to fetch all the patient records for a particular doctor, sorting them by last name. Or, if you had previously stored a numerical index on each patient as they were stored, you could ask for the results to be sorted by that index so that they would be returned to you in the same order each time.

## Faulting and Uniquing

Core Data also works hard to optimize performance and keep memory usage to a minimum, using a technique called *faulting*.

Consider what could happen if you loaded a `Patient` record into memory; in order that you have access to that patient's `Doctor` object, it might seem that you'd want to have the `Doctor` object loaded as well. And, since you might need to access the other patients related to that doctor, you should probably load all those `Patient` objects too. With this behavior, what you thought was a single-object fetch could turn into a fetch of thousands of objects—every related object would need to be fetched, possibly resulting in fetching your entire dataset.

To solve this problem, Core Data *doesn't* fetch all the relationships on an object. It simply returns you the managed object that you asked for, with the relationships set to *faults*. If you try and access one of those relationships, such as asking for the name of the patient's doctor, the "fault will fire" and Core Data will fetch the requested object for you. And, as before, the relationships on a newly fetched doctor object will also be set to faults, ready to fire when you need to access any of the related objects. All of this happens automatically, without you needing to worry about it.

A managed object context will also ensure that if an object has already been loaded, it will always return the existing instance in any subsequent fetches. Consider the code in Listing 2.4.

LISTING 2.4 Fetching unique objects

```
Patient *firstPatient; // From one fetch request
Doctor *firstPatientsDoctor = firstPatient.doctor;
```

```
Patient *secondPatient; // From a second fetch request
Doctor *secondPatientsDoctor = secondPatient.doctor;

/* If the two patients share the same doctor, then the doctor instance
   returned after each fault fires will be the same instance: */

if( firstPatientsDoctor == secondPatientsDoctor )
{
    NSLog(@"Patients share a doctor!");
}
```

This is known as *uniquing*—you will only ever be given one object instance in any managed object context for, say, a particular `Patient`.

### Persistent Stores and Persistent Store Coordinators

The underlying data is held on disk in a *persistent store*. On an iOS device, this is usually a SQLite store. You can also choose to use a binary store or even your own custom atomic store type, but these require the entire object graph to be loaded into memory, which can quickly become a problem on a device with limited resources.

You never need to communicate directly with a persistent store, or worry about how it is storing data. Instead, you rely on the relationship between the managed object context and a *persistent store coordinator*.

The persistent store coordinator acts as a mediator to the managed object contexts; it's also possible to have a coordinator talk to multiple persistent stores on disk, meaning that the coordinator would expose the union of those stores to be accessed by the managed object contexts.

You won't typically need to worry too much about persistent stores and coordinators unless you want to work with multiple stores or define your own store type. In the next section, you'll see the code from the Xcode template project that sets up the persistent store for you. Once this is dealt with, you'll spend most of your time concentrating on the managed objects, held within managed object contexts.

## Examining the Xcode Core Data Templates

Now that you have a better idea of Core Data terminology, let's take a look inside a standard Core Data template project for an iOS application to see how all of this works in practice.

## The Navigation-Based Project Template

The Xcode project templates for the Navigation-based, Split View-based, Utility and Window-based applications all offer the option to *Use Core Data*, as shown in Figure 2.3.

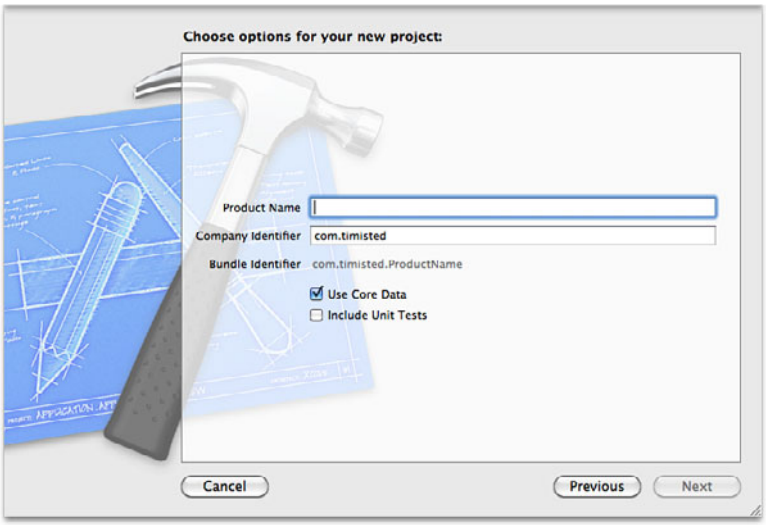


FIGURE 2.3 The New Project Window

To follow the rest of this chapter, launch Xcode and choose File > New > New Project (Shift-⌘-N), then select the Navigation-based Application template. Call the project `TemplateProject`, and tick the Use Core Data checkbox.

When the project window appears, you'll see that there are some extra items compared to a standard project. First, the project links to the `CoreData.framework`. Second, there is an item called `TemplateProject.xcdatamodeld`. This defines the structure of your data model, which you build visually using the Xcode Data Modeler. Click on the file to open it.

There are two ways to view a data model in Xcode 4—as a Table or a Graph, as shown in Figure 2.4.

**NOTE**

Xcode 3 has only one editor mode, which combines the Table and Graph into a single view.

## The Data Modeler

At the top-left of the editor, you'll see a list of *Entities*. In the template file, there is a single entity, called `Event`. If you select this entity for display using the Table editor style, you'll see a list of the entity's *attributes*, *relationships* and *fetched properties*.

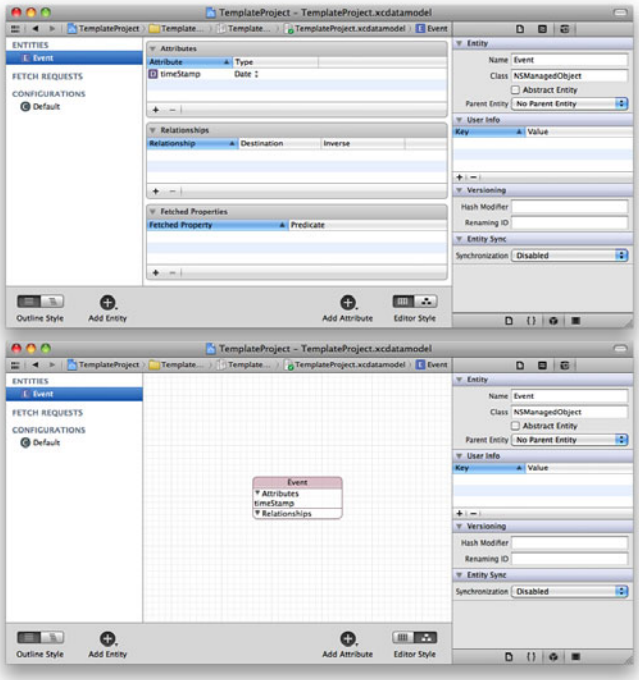


FIGURE 2.4 The two Editor Styles in the Xcode 4 Data Modeler

The `Event` entity has a single attribute listed, called `timeStamp`. If you click this attribute to select it, and look in Xcode 4's Data Model Inspector (Option-⌘-3), you'll see that its *Type* is set to `Date`.

This inspector offers a number of other options relating to that attribute. For example, this is where you can choose to validate the data that is stored, or mark an attribute as being optional; these options are covered in Chapter 3, "Modeling Your Data."

Xcode 4's Graph editor style offers a visual representation of the entities in your object model. At present, there is only a single entity in the model, but if there were more than one, you would see the relationships between the entities represented by lines and arrows connecting the two, as in Figure 2.5.

## Setting up the Core Data Stack

When working with data held in a persistent store, you will need to have built up a *stack* of objects; at the bottom is the actual persistent store on disk, then comes a persistent store controller to liaise between the store and the next level, the managed object context, as shown in Figure 2.6.

It's also possible to have more than one persistent store under the one coordinator, as well as multiple managed object contexts (as discussed earlier in the chapter).

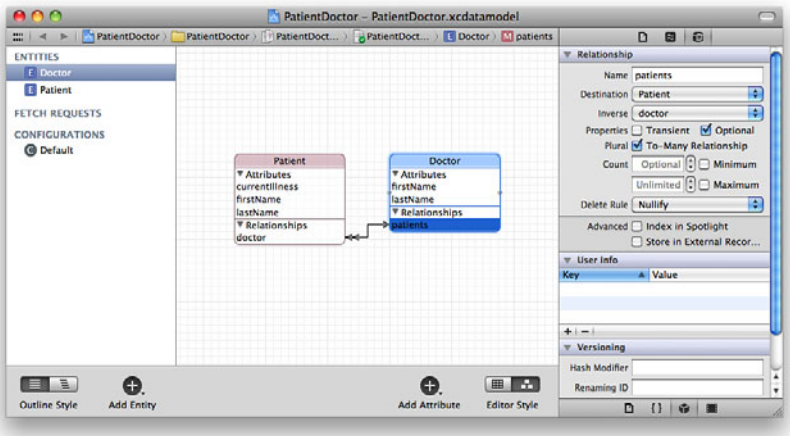


FIGURE 2.5 Relationships between objects in the Object Graph

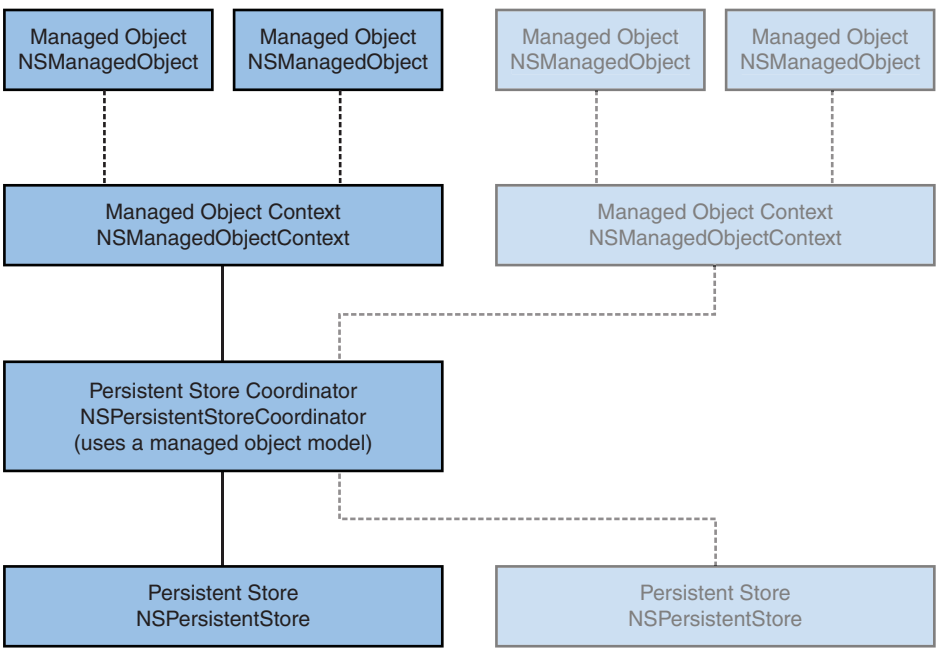


FIGURE 2.6 The Core Data Stack

Let's examine the template code provided to set up this Core Data stack. Open the `TemplateProjectAppDelegate.h` interface file, and you'll find that there are a number of property declarations defined for the app delegate, as shown in Listing 2.5 (the Xcode 4 templates make use of the modern runtime feature of synthesizing the corresponding instance variables).

LISTING 2.5 The app delegate interface

```
@interface TemplateProjectAppDelegate : NSObject <UIApplicationDelegate> {

}

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

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

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

@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;

@end
```

The app delegate keeps track of the managed object model, which is the information contained within the `TemplateProject.xcdatamodeld` file. It also has a reference to a persistent store coordinator, along with a managed object context. The `applicationDocumentsDirectory` is used to determine where the data will be held on disk.

Switch to the `TemplateProjectAppDelegate.m` interface file and scroll through the various methods that are provided. Near the bottom, you'll find the `applicationDocumentsDirectory` method. As the name implies, it simply returns the path to the application's documents directory.

Next find the `persistentStoreCoordinator` method. This method sets up a persistent store coordinator, to access a SQLite store file called `TemplateProject.sqlite`, located in the application's documents directory. The persistent store is initialized using the model provided by the `managedObjectModel` method, so jump to this method next.

The `managedObjectModel` method returns an `NSManagedObjectModel` object created from a file called `TemplateProject.momd`. When you compile the project, the `TemplateProject.xcdatamodeld` data model is compiled into this `.momd` resource and stored in the application's bundle.

It's also possible to create an `NSManagedObjectModel` by merging all the available model files using the class method `mergedModelFromBundles:`, or by merging selected models using `modelByMergingModels:`. Although there is only a single model file (it's actually a model bundle) in this template application, it is possible to split your model into multiple `.xcdatamodeld` files, if you wish.

At the top of the stack, you'll find the `managedObjectContext` method. This method sets up the context using the persistent store coordinator. If you look in the `awakeFromNib` method, towards the top of the file, you'll find that it sets a property on the `RootViewController` for a `managedObjectContext`. It is at this point that the chain is triggered to set up the managed object model, persistent store coordinator and finally the context.

**NOTE**

In earlier versions of Xcode, the template code may be slightly different; for example, setting up the managed object model by merging the models in the main bundle, and setting the `RootViewController`'s `managedObjectContext` property from within `application:didFinishLaunching:`.

Lastly, the `applicationWillTerminate:` method calls a `saveContext` method, which checks to see whether any changes have been made to objects in the managed object context, and tries to save those changes if so. This means that the persistent store will be updated with changes from the context when the application exits. Some versions of the project template also call `saveContext` from `applicationDidEnterBackground:`, so that the store will be updated if the user switches to a different application under iOS 4 multitasking.

You probably won't need to modify the code in these methods unless you need to work with multiple stores or custom store types. Once the managed object context has been set up, it's passed to the root view controller. This view controller has the code that actually performs the relevant fetches to display the data in a table view, and it's this sort of code that you will typically be writing most of the time when you're working with Core Data. In Chapter 4, "Basic Storing and Fetching," you'll start writing your own code to populate a table view with information from a Core Data store.

## Running the Application

To see the functionality you get from the basic project template, build and run the application. You'll find that you can add to a list of `Events`; the table view shows these, outputting the values of the events' `timeStamp` attributes. Note that you can remove items from the table view using the Edit button, or by swiping your finger across a row.

## A Quick Look at the RootViewController Code

To get an idea of how this all works, open up the `RootViewController.m` implementation file. The template application makes use of a *Fetched Results Controller* to simplify working with fetched objects and table views. This object is created lazily and told to fetch when it's needed by one of the table view data source methods.

**NOTE**

In earlier versions of Xcode, the template file creates the `FetchedResultsController` and executes a fetch at the end of the view controller's `viewDidLoad` method.

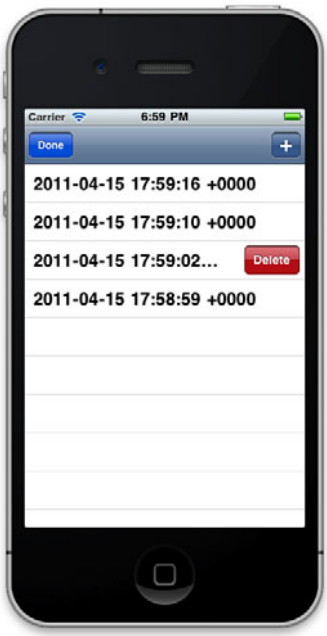


FIGURE 2.7 The Template Application in the Simulator

If you look at the code that generates the `fetchedResultsController` lazily, you'll see that it's set to use the `Event` entity and given sort descriptors to sort by the `timeStamp` attribute.

The methods to display the contents in the table view are the standard table view data source methods; for the `numberOfSections...` and `numberOfRows...` methods, the template code just queries the fetched results controller object. To display the information in the `cellForRowAtIndexPath:` method, a `configureCell:atIndexPath:` method is used. Notice how simple it is to get hold of the managed object at the selected index. The code to display the time stamp is shown in Listing 2.6.

LISTING 2.6 Displaying the timestamp in the cell

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

cell.textLabel.text = [[managedObject valueForKey:@"timeStamp"] description];
```

The fetched results controller returns the object at the specified index, and the code just queries that returned object for the description of its `timeStamp` key.

The `commitEditingStyle:forRowAtIndexPath:` method simply tells the managed object context to delete the object at the specified index path, and then asks the context to `save`.

This will mean that the object deleted from the table view will also be deleted from the persistent store.

Lastly, find the `insertNewObject` method that gets called when the user tries to add an object to the table view, and you'll see that the procedure is:

- Get a pointer to a managed object context.
- Decide which entity you need to use to create a new object.
- Insert a new object for that entity, into the managed object context.
- Set the relevant values on the new object.
- Tell the context to save.

When the context saves, the new object is written to the persistent store. It's as simple as that! Note that the template files are deliberately verbose—it's common to accomplish the above using only two or three lines of code.

## Summary

Now that you have an idea of how a Core Data application is constructed, it's time to start learning how to use the individual parts of the Core Data framework. In the next chapter, you'll see best practices for constructing managed object models from the point of view of memory and performance considerations.

Although the Apple-provided template application makes use of a fetched results controller, it's a good idea to see how objects are fetched *manually*. Chapter 4, "Basic Storing and Fetching," demonstrates how to work directly with managed object contexts and fetch requests to display objects in a table view. Chapter 5, "Using NSFetchedResultsController," then shows how to take advantage of the memory optimization and performance benefits offered by the fetch results controller.

# Index

## SYMBOLS



## A

## B



## C

## D



## E

## F

## G



## H



## I

## P

## Q



## R

## S

## T

## Y



## Z