

Core Data Programming Guide

Contents

Introduction to Core Data Programming Guide 11

Who Should Read This Document 11

Organization of This Document 11

See Also 13

Technology Overview 14

Core Data Features 14

Why Should You Use Core Data? 15

What Core Data Is Not 15

Core Data Basics 17

Basic Core Data Architecture 17

Managed Objects and Contexts 20

Fetch Requests 20

Persistent Store Coordinator 21

Persistent Stores 22

Persistent Documents 23

Managed Objects and the Managed Object Model 24

Managed Object Models 26

Features of a Managed Object Model 26

Entities 26

Properties 27

Fetch Request Templates 29

User Info Dictionaries 29

Configurations 29

Using a Managed Object Model 31

Creating and Loading a Managed Object Model 31

Compiling a Data Model 31

Loading a Data Model 31

Problems May Arise if Your Project Contains More Than One Model 32

Changing the Schema Makes a Model Incompatible With Old Stores 33

Accessing and Using a Managed Object Model at Runtime 33

Creating Fetch Request Templates Programmatically	34
Accessing Fetch Request Templates	34
Localizing a Managed Object Model	35
Strings File	36
Setting a Localization Dictionary Programmatically	36

Managed Objects 38

Basics	38
Properties and Data Storage	39
Non-Standard Attributes	39
Dates and Times	39
Custom Managed Object Classes	39
Overriding Methods	40
Modeled Properties	40
Object Life-Cycle—Initialization and Deallocation	41
Validation	42
Faulting	42

Managed Object Accessor Methods 44

Overview	44
Custom implementation	44
Key-value coding access pattern	45
Dynamically-Generated Accessor Methods	45
Declaration	46
Implementation	47
Inheritance	47
Custom Attribute and To-One Relationship Accessor Methods	48
Custom To-Many Relationship Accessor Methods	51
Custom Primitive Accessor Methods	54

Creating and Deleting Managed Objects 56

Creating, Initializing, and Saving a Managed Object	56
Behind the Scenes of Creating a Managed Object	57
The Managed Object Context	57
The Entity Description	58
Creating a Managed Object	58
Assigning an Object to a Store	59
Deleting a Managed Object	60
Relationships	60
Deleted status and notifications	60

Fetching Managed Objects 61

Fetching Managed Objects 61

Retrieving Specific Objects 62

Fetching Specific Values 63

Fetching and Entity Inheritance 66

Using Managed Objects 67

Accessing and Modifying Properties 67

Attributes and to-one relationships 67

To-many relationships 68

Saving Changes 70

Managed Object IDs and URLs 70

Copying and Copy and Paste 71

Copying Attributes 71

Copying Relationships 72

Drag and Drop 73

Validation 73

Undo Management 74

Faults 75

Ensuring Data Is Up-to-Date 76

Refreshing an object 76

Merging changes with transient properties 77

Object Lifetime Management 79

The Role of the Managed Object Context 79

Breaking Relationship Strong Reference Cycles 80

Change and Undo Management 80

Relationships and Fetched Properties 82

Relationship Definitions in the Model 82

Relationship Fundamentals 82

Inverse Relationships 83

Relationship Delete Rules 84

Manipulating Relationships and Object Graph Integrity 85

Many-to-Many Relationships 86

Unidirectional Relationships 88

Cross-Store Relationships 89

Fetched Properties 89

Non-Standard Persistent Attributes 91

Introduction	91
Transformable Attributes	91
Custom Code	93
Basic Approach	93
Scalar Value Constraints	93
The Persistent Attribute	94
An Object Attribute	94
Scalar Values	98
A Non-Object Attribute	99
Type-Checking	101
Managed Object Validation	103
Core Data Validation	103
Property-Level Validation	104
Inter-Property validation	105
Combining Validation Errors	108
Faulting and Uniquing	110
Faulting Limits the Size of the Object Graph	110
Firing Faults	111
Turning Objects into Faults	111
Faults and KVO Notifications	112
Uniquing Ensures a Single Managed Object per Record per Context	112
Using Persistent Stores	115
Creating and Accessing a Store	115
Changing a Store's Type and Location	116
Associate Metadata With a Store to Provide Additional Information and Support Spotlight Indexing	117
Getting the Metadata	118
Setting the Metadata	118
Core Data and Cocoa Bindings	120
Additions to Controllers	120
Automatically Prepares Content Flag	121
Entity Inheritance	121
Filter Predicate for a To-many Relationship	122
Change Management	124
Disjoint Edits	124
Conflict Detection and Optimistic Locking	125

Conflict Resolution	126
Snapshot Management	126
Communicating Changes Between Contexts	127
Persistent Store Features	130
Store Types and Behaviors	130
Store-specific behavior	131
Custom store types	131
Security	131
Fetch Predicates and Sort Descriptors	132
SQLite Store	132
File-systems supported by the SQLite store	132
File Size May Not Reduce After Deleting a Record	133
Configuring a SQLite Store's Save Behavior	133
Concurrency with Core Data	136
Use Thread Confinement to Support Concurrency	136
Track Changes in Other Threads Using Notifications	137
Fetch in the Background for UI Responsiveness	138
Saving in a Background Thread is Error-prone	138
If You Don't Use Thread Containment	139
Core Data Performance	140
Introduction	140
Fetching Managed Objects	141
Fetch Predicates	141
Fetch Limits	141
Faulting Behavior	142
Batch Faulting and Pre-fetching with the SQLite Store	142
Reducing Memory Overhead	145
Large Data Objects (BLOBs)	146
Analyzing Performance	147
Analyzing Fetch Behavior with SQLite	147
Instruments	147
Troubleshooting Core Data	149
Object Life-Cycle Problems	149
Merge errors	149
Assigning a managed object to a different store	149
Fault cannot be fulfilled	150

Managed object invalidated	151
Class is not key-value coding compliant	151
Entity class does not respond to invocations of custom methods	152
Custom accessor methods are not invoked, key dependencies are not obeyed	152
Problems with Fetching	152
SQLite store does not work with sorting	152
Problems with Saving	153
SQLite store takes a "long time" to save	153
Cannot save documents because entity is null	154
Exception generated in <code>retainedDataForObjectID:withContext.</code>	154
Debugging Fetching	155
Managed Object Models	156
My application generates the message "+entityForName: could not locate an NSManagedObjectModel"	156
Bindings Integration	157
Custom relationship set mutator methods are not invoked by an array controller	157
Cannot access contents of an object controller after a nib is loaded	157
Cannot create new objects with array controller	158
A table view bound to an array controller doesn't display the contents of a relationship	158
A new object is not added to the relationship of the object currently selected in a table view	158
Table view or outline view contents not kept up-to-date when bound to an NSArrayController or NSTreeController object	159
Efficiently Importing Data	160
Cocoa Fundamentals	160
Reducing Peak Memory Footprint	161
Importing in batches	161
Dealing with strong reference cycles	162
Implementing Find-or-Create Efficiently	162
Core Data FAQ	165
Where does a Managed Object Context Come From?	165
How do I initialize a store with default data?	165
How do I use my existing SQLite database with Core Data?	166
I have a to-many relationship from Entity A to Entity B. How do I fetch the instances of Entity B related to a given instance of Entity A?	166
How do I fetch objects in the same order I created them?	167
How do I copy a managed object from one context to another?	167
I have a key whose value is dependent on values of attributes in a related entity—how do I ensure it is kept up to date as the attribute values are changes and as the relationship is manipulated?	167

OS X v10.5 and later for a to-one relationship	167
To-many relationships	168
In Xcode's predicate builder, why don't I see any properties for a fetched property predicate?	170
How efficient is Core Data?	170
Core Data looks similar to EOF. What are the differences?	170
Features Supported Only by EOF	170
Features Supported Only by Core Data	170
Class Mapping	171
Change Management	171
Multi-Threading	171
OS X Desktop	171
How do I get the GUI to validate the data entered by the user?	171
When I remove objects from a detail table view managed by an array controller, why are they not removed from the object graph?	172
How do I get undo/redo for free in my non-document-architecture-based app?	172
Document Revision History	173
Glossary	176

Figures, Tables, and Listings

Core Data Basics 17

- Figure 1 Document management using the standard Cocoa document architecture 18
- Figure 2 Document management using Core Data 19
- Figure 3 An example fetch request 21
- Figure 4 Advanced persistence stack 22
- Figure 5 Managed object model with two entities 24
- Figure 6 Entity description with two attributes and a relationship 24

Managed Object Models 26

- Figure 1 Selecting a parent entity in Xcode 27
- Figure 2 Xcode predicate builder 29

Using a Managed Object Model 31

- Table 1 Keys and values in a localization dictionary for a managed object model 35
- Listing 1 Creating a fetch request template programmatically 34
- Listing 2 Using a fetch request template 35
- Listing 3 Creating a managed object model in code 37

Managed Object Accessor Methods 44

- Listing 1 Implementation of a custom managed object class illustrating attribute accessor methods 48
- Listing 2 Implementation of a custom managed object class illustrating copying setter 50
- Listing 3 Implementation of a custom managed object class illustrating a scalar attribute value 50
- Listing 4 A managed object class illustrating implementation of custom accessors for a to-many relationship 52

Fetching Managed Objects 61

- Listing 1 Example of creating and executing a fetch request 61

Relationships and Fetched Properties 82

- Figure 1 Transferring an employee to a new department 85
- Figure 2 Example of a reflexive many-to-many relationship 86
- Figure 3 A model illustrating a “friends” relationship using an intermediate entity 87

Managed Object Validation 103

-
- Listing 1 Inter-property validation for a Person entity 106
- Listing 2 A method for combining two errors into a single multiple errors error 108

Faulting and Uniquing 110

- Figure 1 A department represented by a fault 110
- Figure 2 Independent faults for a department object 113
- Figure 3 Uniqued fault for two employees working in the same department 113

Change Management 124

- Figure 1 Managed object contexts with mutually inconsistent data values 125

Introduction to Core Data Programming Guide

The Core Data framework provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence.

Who Should Read This Document

You should read this document to gain an understanding of the Core Data framework. *You must be familiar with the basics of Cocoa development, including the Objective-C language and memory management.*

Important: Although this document provides a thorough treatment of the fundamentals of the Core Data framework, simply reading from start to finish is not a good strategy for learning how to use the technology effectively. Instead, you should typically augment your understanding by following the related tutorials provided in the Reference Library. For a description of the recommended learning path, see *Core Data Starting Point*.

Organization of This Document

The following articles explain the problems the Core Data Framework addresses, the solutions it provides, its basic functionality, and common tasks you might perform:

- [“Technology Overview”](#) (page 14) describes what Core Data is and why you might choose to use it.
- [“Core Data Basics”](#) (page 17) describes the fundamental architecture of the technology.
- [“Managed Object Models”](#) (page 26) describes the features of a managed object model.
- [“Using a Managed Object Model”](#) (page 31) describes how you use a managed object model in your application.
- [“Managed Objects”](#) (page 38) describes the features of a managed object, the `NSManagedObject` class, and how and why you might implement a custom class to represent an entity.
- [“Managed Object Accessor Methods”](#) (page 44) describes how to write accessor methods for custom managed objects.
- [“Creating and Deleting Managed Objects”](#) (page 56) describes how to correctly instantiate and delete managed objects programmatically.

- [“Fetching Managed Objects”](#) (page 61) describes how to fetch managed objects, and some considerations to ensure that fetches are efficient.
- [“Using Managed Objects”](#) (page 67) describes issues related to manipulating managed objects in your application.
- [“Object Lifetime Management”](#) (page 79) describes aspects of memory management when using Core Data.
- [“Relationships and Fetched Properties”](#) (page 82) describes relationships, how to model them, and issues related to manipulating relationships between managed objects. It also describes fetched properties, which are like weak unidirectional relationships.
- [“Non-Standard Persistent Attributes”](#) (page 91) describes how to use attributes with non-standard value types (such as colors and C-structures).
- [“Managed Object Validation”](#) (page 103) describes types of validation, how to implement validation methods, and when to use validation.
- [“Faulting and Uniquing”](#) (page 110) describes how Core Data constrains the size of the object graph, and ensures that each managed object within a managed object context is unique.
- [“Using Persistent Stores”](#) (page 115) describes how you create a persistent store, how you can migrate a store from one type to another, and manage store metadata.
- [“Core Data and Cocoa Bindings”](#) (page 120) describes how Core Data integrates with and leverages Cocoa bindings.
- [“Change Management”](#) (page 124) describes the issues that may arise if you create multiple managed object contexts or multiple persistence stacks.
- [“Persistent Store Features”](#) (page 130) describes the features of the different types of store, and how you can configure the behavior of the SQLite store.
- [“Concurrency with Core Data”](#) (page 136) describes how to use concurrent programming in a Core Data application.
- [“Core Data Performance”](#) (page 140) describes techniques you can use to ensure a Core Data application is as efficient as possible.
- [“Troubleshooting Core Data”](#) (page 149) describes common errors developers make when using Core Data, and how to correct them.
- [“Efficiently Importing Data”](#) (page 160) describes how you can import data into a Core Data application.
- [“Core Data FAQ”](#) (page 165) provides answers to questions frequently asked about Core Data.
- [“Glossary”](#) (page 176) provides a glossary of terms used in Core Data.

See Also

You should also refer to:

- *Core Data Starting Point*
- *Core Data Tutorial for iOS*
- *Core Data Utility Tutorial*
- *Core Data Model Editor Help*
- *Core Data Snippets*

Technology Overview

This article describes the basic features provided by Core Data, and reasons why it might be appropriate for you to adopt the technology.

Core Data Features

The Core Data framework provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence. Its features include:

- Change tracking and undo support.

Core Data provides built-in management of undo and redo beyond basic text editing.

- Relationship maintenance.

Core Data manages change propagation, including maintaining the consistency of relationships among objects.

- Futures (faulting).

Core Data can reduce the memory overhead of your program by lazily loading objects. It also supports partially materialized futures, and copy-on-write data sharing.

- Automatic validation of property values.

Core Data's managed objects extend the standard key-value coding validation methods that ensure that individual values lie within acceptable ranges so that combinations of values make sense.

- Schema migration.

Dealing with a change to your application's schema can be difficult, in terms of both development effort and runtime resources. Core Data's schema migration tools simplify the task of coping with schema changes, and in some cases allow you to perform extremely efficient in-place schema migration.

- Optional integration with the application's controller layer to support user interface synchronization.

Core Data provides the `NSFetchedResultsController` object on iOS, and integrates with Cocoa Bindings on OS X.

- Full, automatic, support for key-value coding and key-value observing.

In addition to synthesizing key-value coding and key-value observing compliant accessor methods for attributes, Core Data synthesizes the appropriate collection accessors for to-many relationships.

- Grouping, filtering, and organizing data in memory and in the user interface.
- Automatic support for storing objects in external data repositories.
- Sophisticated query compilation.

Instead of writing SQL, you can create complex queries by associating an `NSPredicate` object with a fetch request. `NSPredicate` provides support for basic functions, correlated subqueries, and other advanced SQL. With Core Data, it also supports proper Unicode, locale-aware searching, sorting, and regular expressions.

- Merge policies.

Core Data provides built in version tracking and optimistic locking to support automatic multi-writer conflict resolution.

Why Should You Use Core Data?

There are a number of reasons why it may be appropriate for you to use Core Data. One of the simplest metrics is that, with Core Data, the amount of code you write to support the model layer of your application is typically 50% to 70% smaller as measured by lines of code. This is primarily due to the features listed above—the features Core Data provides are features you don't have to implement yourself. Moreover they're features you don't have to test yourself, and in particular you don't have to optimize yourself.

Core Data has a mature code base whose quality is maintained through unit tests, and is used daily by millions of customers in a wide variety of applications. The framework has been highly optimized over several releases. It takes advantage of information provided in the model and runtime features not typically employed in application-level code. Moreover, in addition to providing excellent security and error-handling, it offers best memory scalability of any competing solution. Put another way: you could spend a long time carefully crafting your own solution optimized for a particular problem domain, and not gain any performance advantage over what Core Data offers for free for any application.

In addition to the benefits of the framework itself, Core Data integrates well with the OS X tool chain. The model design tools allow you to create your schema graphically, quickly and easily. You can use templates in the Instruments application to measure Core Data's performance, and to debug various problems. On OS X desktop, Core Data also integrates with Interface Builder to allow you to create user interfaces from your model. These aspects help to further shorten your application design, implementation, and debugging cycles.

What Core Data Is Not

Having given an overview of what Core Data is and does, and why it may be useful, it is also useful to correct some common misperceptions and state what it is not.

- Core Data is not a relational database or a relational database management system (RDBMS).

Core Data provides an infrastructure for change management and for saving objects to and retrieving them from storage. It can use SQLite as one of its persistent store types. It is not, though, in and of itself a database. (To emphasize this point: you could for example use just an in-memory store in your application. You could use Core Data for change tracking and management, but never actually save any data in a file.)

- Core Data is not a silver bullet.

Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

- Core Data does not rely on Cocoa bindings.

Core Data integrates well with Cocoa bindings and leverages the same technologies—and used together they can significantly reduce the amount of code you have to write—but it is possible to use Core Data without bindings. You can readily create a Core Data application without a user interface (see *Core Data Utility Tutorial*).

Core Data Basics

This article describes the basic Core Data architecture, and the way you use the framework.

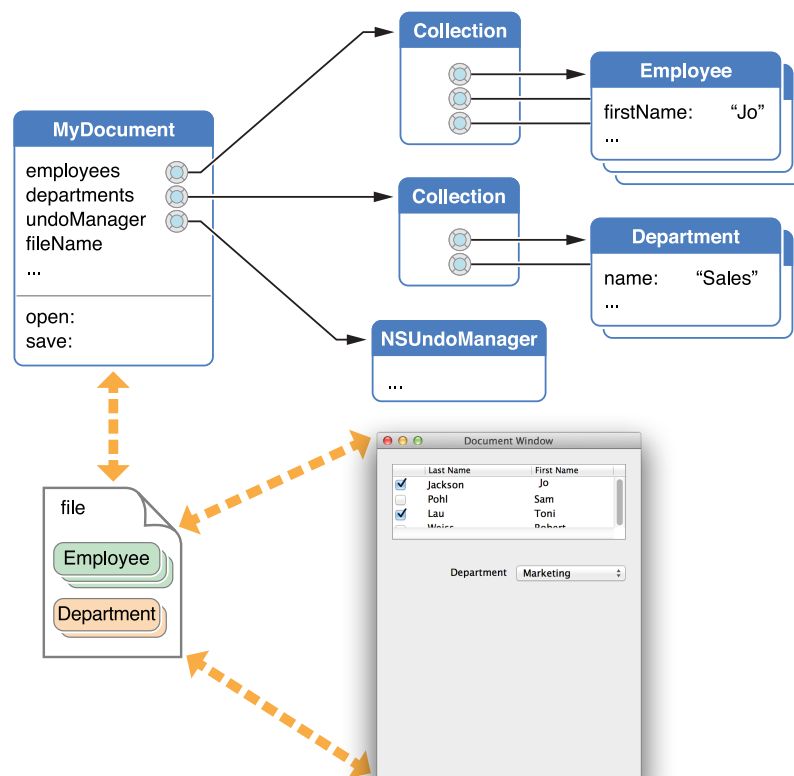
Basic Core Data Architecture

In most applications, you need a means to open a file containing an archive of objects, and a reference to at least one root object. You also need to be able to archive all the objects to a file and—if you want to support undo—to track changes to the objects. For example, in an employee management application, you need a means to open a file containing an archive of employee and department objects, and a reference to at least one root object—for example, the array of all employees—as illustrated in [Figure 1](#) (page 18). You also need to be able to archive to a file all the employees and all the departments.

Note : This document uses the employees example for reasons of expediency and clarity. It represents a rich but easily understood problem domain. The utility of the Core Data framework, however, is not restricted to database-style applications, nor is there an expectation of client-server behavior. The framework is equally useful as the basis of a vector graphics application such as Sketch or a presentation application such as Keynote.

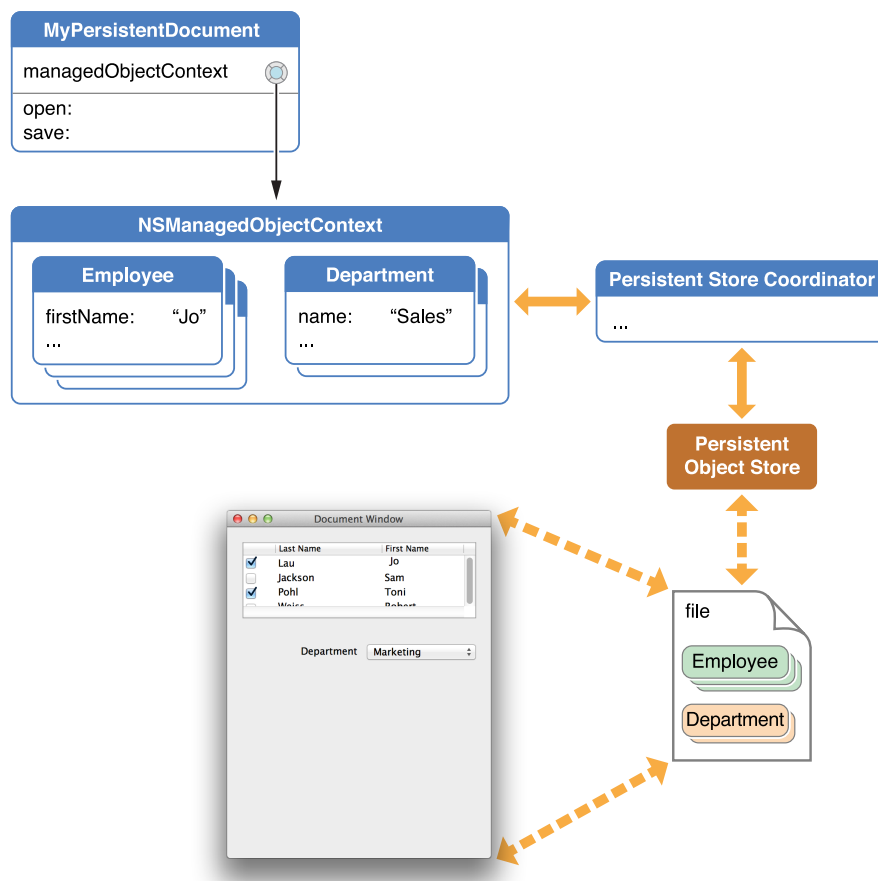
You are responsible for writing the code that manages these tasks either in whole or in part. For example, on OS X desktop, the Cocoa document architecture provides an application structure and functionality that helps to reduce the burden, but you still have to write methods to support archiving and unarchiving of data, to keep track of the model objects, and to interact with an undo manager to support undo.

Figure 1 Document management using the standard Cocoa document architecture



Using the Core Data framework, most of this functionality is provided for you automatically, primarily through an object known as a **managed object context** (or just “context”). The managed object context serves as your gateway to an underlying collection of framework objects—collectively known as the **persistence stack**—that mediate between the objects in your application and external data stores. At the bottom of the stack are **persistent object stores**, as illustrated in [Figure 2](#) (page 19).

Figure 2 Document management using Core Data



Core Data is not restricted to document-based applications—indeed it is possible to create a Core Data–based utility with no user interface at all (see *Core Data Utility Tutorial*). The same principles apply in other applications.

Managed Objects and Contexts

You can think of a managed object context as an intelligent scratch pad. When you fetch objects from a persistent store, you bring temporary copies onto the scratch pad where they form an object graph (or a collection of object graphs). You can then modify those objects however you like. Unless you actually save those changes, however, the persistent store remains unaltered.

Model objects that tie into in the Core Data framework are known as **managed objects**. All managed objects must be registered with a managed object context. You add objects to the graph and remove objects from the graph using the context. The context tracks the changes you make, both to individual objects' attributes and to the relationships between objects. By tracking changes, the context is able to provide undo and redo support for you. It also ensures that if you change relationships between objects, the integrity of the object graph is maintained.

If you choose to save the changes you've made, the context ensures that your objects are in a valid state. If they are, then the changes are written to the persistent store (or stores) and new records added for objects you created and records removed for objects you deleted.

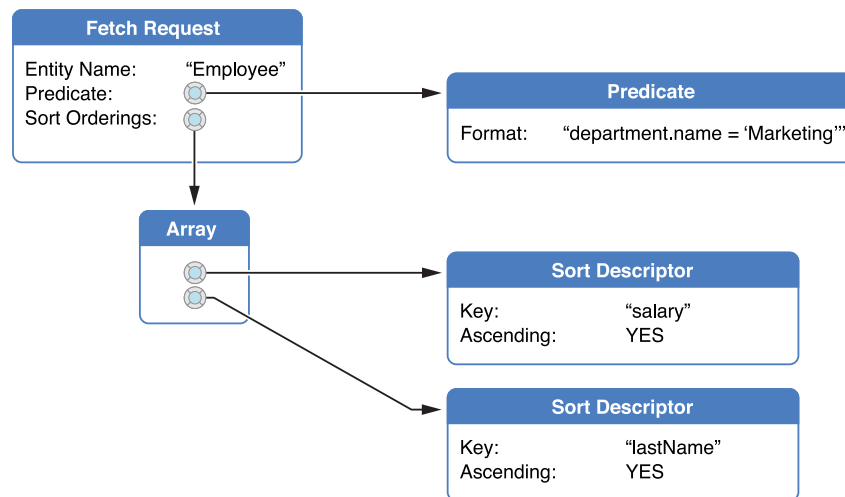
You may have more than one managed object context in your application. For every object in a persistent store there may be at most one corresponding managed object associated with a given context (for more details, see [“Faulting and Uniquing”](#) (page 110)). To consider this from a different perspective, a given object in a persistent store may be edited in more than one context simultaneously. Each context, however, has its own managed object that corresponds to the source object, and each managed object may be edited independently. This can lead to inconsistencies during a save—Core Data provides a number of ways to deal with this (see, for example, [“Using Managed Objects”](#) (page 67)).

Fetch Requests

To retrieve data using a managed object context, you create a **fetch request**. A fetch request is an object that specifies what data you want, for example, “all Employees,” or “all Employees in the Marketing department ordered by salary, highest to lowest.” A fetch request has three parts. Minimally it must specify the name of

an entity (by implication, you can only fetch one type of entity at a time). It may also contain a predicate object that specifies conditions that objects must match and an array of sort descriptor objects that specifies the order in which the objects should appear, as illustrated in [Figure 3](#) (page 21).

Figure 3 An example fetch request



You send a fetch request to a managed object context, which returns the objects that match your request (possibly none) from the data sources associated with its persistent stores. Since all managed objects must be registered with a managed object context, objects returned from a fetch are automatically registered with the context you used for fetching. Recall though that for every object in a persistent store there may be at most one corresponding managed object associated with a given context (see [“Faulting and Uniquing”](#) (page 110)). If a context already contains a managed object for an object returned from a fetch, then the existing managed object is returned in the fetch results.

The framework tries to be as efficient as possible. Core Data is demand driven, so you don't create more objects than you actually need. The graph does not have to represent all the objects in the persistent store. Simply specifying a persistent store does not bring any data objects into the managed object context. When you fetch a subset of the objects from the persistent store, you only get the objects you asked for. If you follow a relationship to an object that hasn't been fetched, it is fetched automatically for you. If you stop using an object, by default it will be deallocated. (This is of course not the same as removing it from the graph.)

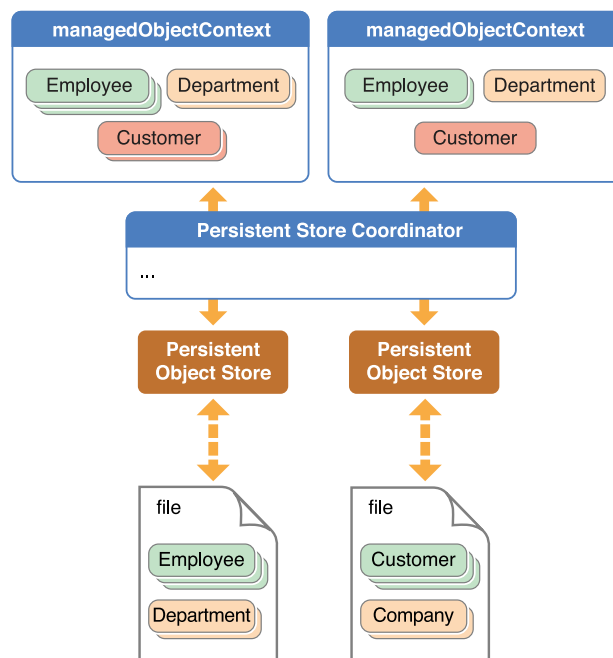
Persistent Store Coordinator

As noted earlier, the collection of framework objects that mediate between the objects in your application and external data stores is referred to collectively as the persistence stack. At the top of the stack are managed object contexts, at the bottom of the stack are persistent object stores. Between managed object contexts and persistent object stores there is a **persistent store coordinator**.

In effect, a persistent store coordinator defines a stack. The coordinator is designed to present a façade to the managed object contexts so that a group of persistent stores appears as a single aggregate store. A managed object context can then create an object graph based on the union of all the data stores the coordinator covers. A coordinator can only be associated with one managed object model. If you want to put different entities into different stores, you must partition your model entities by defining configurations within the managed object models (see “[Configurations](#)” (page 29)).

[Figure 4](#) (page 22) shows an example where employees and departments are stored in one file, and customers and companies in another. When you fetch objects, they are automatically retrieved from the appropriate file, and when you save, they are archived to the appropriate file.

Figure 4 Advanced persistence stack



Persistent Stores

A given persistent object store is associated with a single file or other external data store and is ultimately responsible for mapping between data in that store and corresponding objects in a managed object context. Normally, the only interaction you have with a persistent object store is when you specify the location of a new external data store to be associated with your application (for example, when the user opens or saves a document). Most other interactions with the Core Data framework are through the managed object context.

Your application code—and in particular the application logic associated with managed objects—should not make any assumptions about the persistent store in which data may reside. Core Data provides native support for several file formats. You can choose which to use depending on the needs of your application. If at some stage you decide to choose a different file format, your application architecture remains unchanged. Moreover, if your application is suitably abstracted, then you will be able to take advantage of later enhancements to the framework without any additional effort. For example—even if the initial implementation is able to fetch records only from the local file system—if an application makes no assumptions about where it gets its data from, then if at some later stage support is added for a new type of remote persistent store, it should be able to use this new type with no code revisions.

Important: Although Core Data supports SQLite as one of its persistent store types, Core Data cannot manage any arbitrary SQLite database. In order to use a SQLite database, Core Data must create and manage the database itself. For more about store types, see [“Persistent Store Features”](#) (page 130).

Persistent Documents

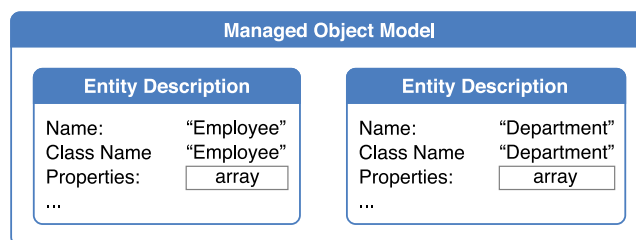
You can create and configure the persistence stack programmatically. In many cases, however, you simply want to create a document-based application able to read and write files. The `NSPersistentDocument` class is a subclass of `NSDocument` that is designed to let you easily take advantage of the Core Data framework. By default, an `NSPersistentDocument` instance creates its own ready-to-use persistence stack, including a managed object context and a single persistent object store. There is in this case a one-to-one mapping between a document and an external data store.

The `NSPersistentDocument` class provides methods to access the document’s managed object context and provides implementations of the standard `NSDocument` methods to read and write files that use the Core Data framework. By default you do not have to write any additional code to handle object persistence. A persistent document’s undo functionality is integrated with the managed object context.

Managed Objects and the Managed Object Model

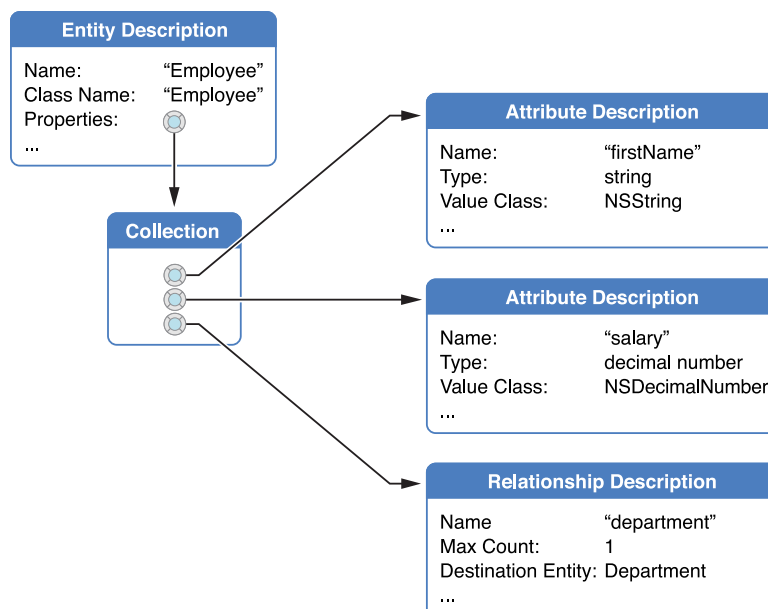
In order both to manage the object graph and to support object persistence, Core Data needs a rich description of the objects it operates on. A **managed object model** is a schema that provides a description of the managed objects, or entities, used by your application, as illustrated in [Figure 5](#) (page 24). You typically create the managed object model graphically using Xcode's Data Model Design tool. (If you wish you can construct the model programmatically at runtime.)

Figure 5 Managed object model with two entities



The model is composed of a collection of entity description objects that each provide metadata about an entity, including the entity's name, the name of the class that represents it in your application (this does not have to be the same as its name), and its attributes and relationships. The attributes and relationships in turn are represented by attribute and relationship description objects, as illustrated in [Figure 6](#) (page 24).

Figure 6 Entity description with two attributes and a relationship



Managed objects must be instances of either `NSManagedObject` or of a subclass of `NSManagedObject`. `NSManagedObject` is able to represent any entity. It uses a private internal store to maintain its properties and implements all the basic behavior required of a managed object. A managed object has a reference to the entity description for the entity of which it is an instance. It refers to the entity description to discover metadata about itself, including the name of the entity it represents and information about its attributes and relationships. You can also create subclasses of `NSManagedObject` to implement additional behavior.

Managed Object Models

Much of Core Data's functionality depends on the schema you create to describe your application's entities, their properties, and the relationships between them. The schema is represented by a managed object model—an instance of `NSManagedObjectModel`. In general, the richer the model, the better Core Data is able to support your application. This article describes the features of a managed object model, how you create one, and how you use it in your application.

Features of a Managed Object Model

A managed object model is an instance of the `NSManagedObjectModel` class. It describes a schema—a collection of entities—that you use in your application. (If you do not understand the term "entity"—or the related terms, "property," "attribute," and "relationship"—you should first read [“Core Data Basics”](#) (page 17) and [“Object Modeling”](#).)

Entities

A model contains `NSEntityDescription` objects that represent the model's entities. Two important features of an entity are its name, and the name of the class used to represent the entity at runtime. You should be careful to keep clear the differences between an entity, the class used to represent the entity, and the managed objects that are instances of that entity.

An `NSEntityDescription` object may have `NSAttributeDescription` and `NSRelationshipDescription` objects that represent the properties of the entity in the schema. An entity may also have fetched properties, represented by instances of `NSFetchedPropertyDescription`, and the model may have fetch request templates, represented by instances of `NSFetchRequest`.

In a model, entities may be arranged in an inheritance hierarchy, and entities may be specified as abstract.

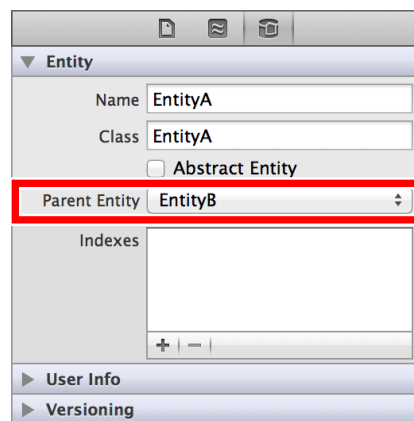
Entity Inheritance

Entity inheritance works in a similar way to class inheritance, and is useful for the same reasons. If you have a number of entities that are similar, you can factor the common properties into a super-entity. Rather than specifying the same properties in several entities, you can define them in one and the sub-entities inherit them. For example, you might define a `Person` entity with attributes `firstName` and `lastName`, and sub-entities `Employee` and `Customer` which inherit those attributes.

In many cases, you also implement a custom class to correspond to the entity from which classes representing the sub-entities also inherit. Rather than implementing business logic common to all the entities several times over, you implement them in one place and they are inherited by the subclasses.

If you create a model using the data modeling tool in Xcode, you specify an entity's parent by selecting the name of the entity from the Parent pop-up menu in the entity Info pane, as shown in [Figure 1](#) (page 27).

Figure 1 Selecting a parent entity in Xcode



If you want to create an entity inheritance hierarchy in code, you must build it top-down. You cannot set an entity's super-entity directly, you can only set an entity's sub-entities (using the method `setSubentities:`). To set a super-entity for a given entity, you must therefore set an array of sub-entities for that super entity and include the current entity in that array.

Abstract Entities

You can specify that an entity is abstract—that is, that you will not create any instances of that entity. You typically make an entity abstract if you have a number of entities that all represent specializations of (inherit from) a common entity which should not itself be instantiated. For example, in a drawing application you might have a `Graphic` entity that defines attributes for `x` and `y` coordinates, color, and drawing bounds. You never, though, instantiate a `Graphic`. Concrete sub-entities of `Graphic` might be `Circle`, `TextArea`, and `Line`.

Properties

An entity's properties are its attributes and relationships, including its fetched properties (if it has any). Amongst other features, each property has a name and a type. Attributes may also have a default value. A property name cannot be the same as any no-parameter method name of `NSObject` or `NSManagedObject`—for example, you cannot give a property the name "description" (see `NSPropertyDescription`).

Transient properties are properties that you define as part of the model, but which are not saved to the persistent store as part of an entity instance's data. Core Data does track changes you make to transient properties, so they are recorded for undo operations.

Note: If you undo a change to a transient property that uses non-modeled information, Core Data does not invoke your set accessor with the old value—it simply updates the snapshot information.

Attributes

Core Data natively supports a variety of attribute types, such as string, date, and integer (represented as instances of `NSString`, `NSDate`, and `NSNumber` respectively). If you want to use an attribute type that is not natively supported, you can use one of the techniques described in [“Non-Standard Persistent Attributes”](#) (page 91).

You can specify that an attribute is optional—that is, it is not required to have a value. In general, however, you are discouraged from doing so—especially for numeric values (typically you can get better results using a mandatory attribute with a default value—in the model—of 0). The reason for this is that SQL has special comparison behavior for `NULL` that is unlike Objective-C's `nil`. `NULL` in a database is not the same as 0, and searches for 0 will not match columns with `NULL`.

```
false == (NULL == 0)
false == (NULL != 0)
```

Moreover, `NULL` in a database is not equivalent to an empty string or empty data blob, either:

```
false == (NULL == @"")
false == (NULL != @"")
```

This has no bearing on relationships.

Relationships

Core Data supports to-one and to-many relationships, and **fetchd properties**. Fetchd properties represent weak, one-way relationships. In the employees and departments domain, a fetchd property of a department might be “recent hires” (employees do not have an inverse to the recent hires relationship).

You can specify the optionality and cardinality of a relationship, and its delete rule. You should typically model a relationship in both directions. A many-to-many relationship is one in which a relationship and its inverse are both to-many. Relationships are described in greater detail in [“Relationships and Fetched Properties”](#) (page 82).

Fetch Request Templates

You use the `NSFetchRequest` class to describe fetch requests to retrieve objects from a persistent store. It is often the case that you want to execute the same request on multiple occasions, or execute requests that follow a given pattern but which contain variable elements (typically supplied by the user). For example, you might want to be able to retrieve all publications written by a certain author, perhaps after a date specified by the user at runtime.

You can predefine fetch requests and store them in a managed object model as named templates. This allows you to pre-define queries that you can retrieve as necessary from the model. Typically, you define fetch request templates using the Xcode data modeling tool (see *Xcode Tools for Core Data*). The template may include variables, as shown in Figure 2.

Figure 2 Xcode predicate builder



For more about using fetch request templates, see [“Accessing and Using a Managed Object Model at Runtime”](#) (page 33).

User Info Dictionaries

Many of the elements in a managed object model—entities, attributes, and relationships—have an associated user info dictionary. You can put whatever information you want into a user info dictionary, as key-value pairs. Common information to put into the user info dictionary includes version details for an entity, and values used by the predicate for a fetched property.

Configurations

A configuration has a name and an associated set of entities. The sets may overlap—that is, a given entity may appear in more than one configuration. You establish configurations programmatically using `setEntities:forConfiguration:` or using the Xcode data modeling tool (see *Xcode Tools for Core Data*), and retrieve the entities for a given configuration name using `entitiesForConfiguration:`.

You typically use configurations if you want to store different entities in different stores. A persistent store coordinator can only have one managed object model, so by default each store associated with a given coordinator must contain the same entities. To work around this restriction, you can create a model that contains the union of all the entities you want to use. You then create configurations in the model for each of the subsets of entities that you want to use. You can then use this model when you create a coordinator. When you add stores, you specify the different store attributes by configuration. When you are creating your configurations, though, remember that you cannot create cross-store relationships.

Using a Managed Object Model

This article describes how you use a managed object model in your application.

Creating and Loading a Managed Object Model

You usually create a model in Xcode, as described in *Core Data Model Editor Help*. You can also create a model entirely in code, as show in [Listing 3](#) (page 37) and described in *Core Data Utility Tutorial*—typically, however, this is too long-winded to consider in anything but the most trivial application. (You are nevertheless encouraged to review the tutorial to gain an understanding of what the modeling tool does, and in particular to gain an appreciation that the model is simply a collection of objects.)

Compiling a Data Model

A data model is a deployment resource. In addition to details of the entities and properties in the model, a model you create in Xcode contains information about the diagram—its layout, colors of elements, and so on. This latter information is not needed at runtime. The model file is compiled using the model compiler, `momc`, to remove the extraneous information and make runtime loading of the resource as efficient as possible. An `xcdatamodeld` “source” directory is compiled into a `momd` deployment directory, and an `xcdatamodel` “source” file is compiled into a `mom` deployment file.

`momc` is located in `/Developer/usr/bin/`. If you want to use it in your own build scripts, its usage is `momc source destination`, where `source` is the path of the Core Data model to compile and `destination` is the path of the output.

Loading a Data Model

In some cases, you do not have to write any code to load a model. If you use a document-based application on OS X, `NSPersistentDocument` manages the task of finding and loading your application’s model for you. If you use Xcode to create a non-document application that uses Core Data (for OS X or for iOS), the application delegate includes code to retrieve the model. The name of a model—as represented by the filename used to store it on disk—is not relevant at runtime. Once the model is loaded by Core Data, the filename is meaningless and has no use, so you can name the model file whatever you like.

If you want to load a model yourself, there are two mechanisms you can use:

- You can load a single model from a specific URL, using the instance method `initWithContentsOfURL:`.

This is the generally-preferred technique. Typically an application has a single model, and using this method you ensure that you load only that model. You can also load individual models via URLs and then unify them using `modelByMergingModels:` before instantiating a coordinator with them.

In cases where you have more than one model—and particularly in cases where the models represent different versions of the same schema—knowing which model to load is essential (merging together models with the same entities at runtime into a single collection would cause naming collisions and errors). This method is also useful if you want to store the model outside of the bundle for your application, and so need to reference it via a file-system URL.

- You can create a merged model from a specific collection of bundles, using the class method `mergedModelFromBundles:`.

This method may be useful in cases where segregation of models is not important—for example, you may know your application and a framework it links to both have models you need or want to load. The class method allows you to easily load all of the models at once without having to consider what the names are, or put in specialized initialization code to ensure all of your models are found.

Problems May Arise if Your Project Contains More Than One Model

There are a few situations in which you may encounter problems when trying to load a model. Typically these are caused by the build products of your project being out of date combined with use of the class method `mergedModelFromBundles:`.

- If you simply rename your model file, Core Data attempts to merge the current and the old versions and you get an error similar to the following:

```
reason = "'Can't merge models with two different entities named  
'EntityName'";
```

- If you create a new model that contains different entities from those in your original model, then Core Data merges the old and new models. If you have an existing store, you get an error similar to the following when you attempt to open it:

```
reason = "The model used to open the store is incompatible with the one  
used to create the store";
```

There are two solutions:

- Make sure that you clean any old build products before running the application. If the application bundle itself contains old model files, you can delete the application.
- Instead of `mergedModelFromBundles:`, use `initWithContentsOfURL:` to initialize the model. The URL uniquely identifies a model so that Core Data will not merge the current model with any legacy models.

Changing the Schema Makes a Model Incompatible With Old Stores

Because a model describes the structure of the data in a persistent store, changing any parts of a model that alters the schema renders it incompatible with (and so unable to open) the stores it previously created. If you change your schema, you therefore need to migrate the data in existing stores to new version (see *Core Data Model Versioning and Data Migration Programming Guide*). For example, if you add a new entity or a new attribute to an existing entity, you will not be able to open old stores; if you add a validation constraint or set a new default value for an attribute, you will be able to open old stores.

Important: If you want to change the model but also retain the ability to open stores created using a previous version of the model, *you must keep the previous version of the model* (as a version in a versioned model). Core Data cannot open a store for which it has no compatible model. Thus, if you want to change the model but also retain the ability to open existing stores, you must:

1. Ensure that you have a versioned model—if you don't, make the current model into a versioned model.
2. *Before editing the schema*, create a new version of the current model.
3. Edit the new current version of the model, leaving the old version unaltered.

Accessing and Using a Managed Object Model at Runtime

You sometimes need to gain access to the model at runtime, typically to—for example—retrieve a fetch request template, a localized entity name, or perhaps the data type of an attribute. You may also want to programmatically modify the model (you can do this only before it is used at runtime, see `NSManagedObjectModel`). There are a number of ways you can access a managed object model at runtime. Through the persistence stack you ultimately get the model from the persistent store coordinator. Thus to get the model from a managed object context, you use the following code:

```
[[<#A managed object context#> persistentStoreCoordinator] managedObjectModel];
```

You can also retrieve the model from an entity description, so given a managed object you can retrieve its entity description and hence the model, as shown in the following example.

```
[[<#A managed object#> entity] managedObjectModel];
```

In some cases, you maintain a “direct” reference to the model—that is, a method that returns the model directly. `NSPersistentDocument` provides `managedObjectModel` that returns the model associated with the persistent store coordinator used by the document’s managed object context. If you use the Core Data Application template, the application delegate maintains a reference to the model.

Creating Fetch Request Templates Programmatically

You can create fetch request templates programmatically and associate them with a model using `setFetchRequestTemplate:forName:` as illustrated in Listing 1. Recall, though, that you can only modify the model before it has been used by a store coordinator.

Listing 1 Creating a fetch request template programmatically

```
NSManagedObjectModel *model = <#Get a model#>;
NSFetchRequest *requestTemplate = [[NSFetchRequest alloc] init];
NSEntityDescription *publicationEntity =
    [[model entitiesByName] objectForKey:@"Publication"];
[requestTemplate setEntity:publicationEntity];

NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:
    @"(mainAuthor.firstName like[cd] $FIRST_NAME) AND \
    (mainAuthor.lastName like[cd] $LAST_NAME) AND \
    (publicationDate > $DATE)"];
[requestTemplate setPredicate:predicateTemplate];

[model setFetchRequestTemplate:requestTemplate
    forName:@"PublicationsForAuthorSinceDate"];
```

Accessing Fetch Request Templates

You can retrieve and use a fetch request template as illustrated in the code fragment in “Accessing and Using a Managed Object Model at Runtime.” The substitution dictionary must contain keys for all the variables defined in the template; if you want to test for a null value, you must use an `NSNull` object—see “Using Predicates”.

Listing 2 Using a fetch request template

```
NSManagedObjectModel *model = <#Get a model#>;
NSError *error = nil;
NSDictionary *substitutionDictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Fiona", @"FIRST_NAME", @"Verde", @"LAST_NAME",
    [NSDate dateWithTimeIntervalSinceNow:-31356000], @"DATE", nil];
NSFetchRequest *fetchRequest =
    [model fetchRequestFromTemplateWithName:@"PublicationsForAuthorSinceDate"
        substitutionVariables:substitutionDictionary];
NSArray *results =
    [aManagedObjectContext executeFetchRequest:fetchRequest error:&error];
```

If the template does not have substitution variables, you must either:

1. Use `fetchRequestFromTemplateWithName:substitutionVariables:` and pass `nil` as the variables argument; or
2. Use `fetchRequestTemplateName:` and copy the result.

If you try to use the fetch request returned by `fetchRequestTemplateName:`, this generates an exception ("Can't modify a named fetch request in an immutable model").

Localizing a Managed Object Model

You can localize most aspects of a managed object model, including entity and property names and error messages. It is important to consider that localization also includes "localization into your own language." Even if you do not plan to provide foreign-language versions of your application, you can provide a better experience for your users if error messages show "natural language" names rather than "computer language" names (for example, "First Name is a required property" rather than "firstName is a required property").

You localize a model by providing a localization dictionary that follows the pattern shown in the table below.

Table 1 Keys and values in a localization dictionary for a managed object model

Key	Value	Note
"Entity/NonLocalizedEntityName"	"LocalizedEntityName"	
"Property/NonLocalizedPropertyName/Entity/EntityName"	"LocalizedPropertyName"	1
"Property/NonLocalizedPropertyName"	"LocalizedPropertyName"	

Key	Value	Note
"ErrorString/NonLocalizedErrorString"	"LocalizedErrorString"	

Note: (1) For properties in different entities with the same non-localized name but which should have different localized names.

You can access the localization dictionary using the method `localizationDictionary`. Note, however, that in the implementation in OS X version 10.4, `localizationDictionary` may return `nil` until Core Data lazily loads the dictionary for its own purposes (for example, reporting a localized error).

Strings File

The easiest way to localize a model is to create a corresponding strings file—the strings file name is the same as the model file name, but with a `.strings` rather than a `.xcdatamodel` extension (for example, for a model file named `MyDocument.xcdatamodel` the corresponding strings file is `MyDocumentModel.strings`—if your model file name already includes the suffix "Model", you must append a further "Model", so the strings file corresponding to `JimsModel.xcdatamodel` would be the rather unlikely-looking `JimsModelModel.strings`). The file format is similar to a standard strings file you use for localization (see “Localizing String Resources”) but the key and value pattern follows that shown in [Table 1](#) (page 35).

A strings file for a model that includes an employee entity might contain the following:

```
"Entity/Emp" = "Employee";  
"Property/firstName" = "First Name";  
"Property/lastName" = "Last Name";  
"Property/salary" = "Salary";
```

Setting a Localization Dictionary Programmatically

You can set a localization dictionary at runtime using the `NSManagedObjectModel` method `setLocalizationDictionary:`. You must create a dictionary with keys and values as shown in [Table 1](#) (page 35), and associate it with the model. You must ensure you do this before the model is used to fetch or create managed objects, because the model is read-only after doing so. The listing shown in [Listing 3](#) (page 37) illustrates the creation in code of a managed object model including a localization dictionary. The entity is named "Run" and is represented at runtime by the `Run` class. The entity has two attributes, "date" and "processID"—a date and an integer respectively. The process ID has a constraint that its value must not be less than zero.

Listing 3 Creating a managed object model in code

```
NSManagedObjectModel *mom = [[NSManagedObjectModel alloc] init];
NSEntityDescription *runEntity = [[NSEntityDescription alloc] init];
[runEntity setName:@"Run"];
[runEntity setManagedObjectClassName:@"Run"];
[mom setEntities:@[runEntity]];

NSMutableArray *runProperties = [NSMutableArray array];

NSAttributeDescription *dateAttribute = [[NSAttributeDescription alloc] init];
[runProperties addObject:dateAttribute];
[dateAttribute setName:@"date"];
[dateAttribute setAttributeType:NSDateAttributeType];
[dateAttribute setOptional:NO];

NSAttributeDescription *idAttribute= [[NSAttributeDescription alloc] init];
[runProperties addObject:idAttribute];
[idAttribute setName:@"processID"];
[idAttribute setAttributeType:NSInteger32AttributeType];
[idAttribute setOptional:NO];
[idAttribute setDefaultValue:@0];

NSPredicate *validationPredicate = [NSPredicate predicateWithFormat:@"SELF >= 0"];
NSString *validationWarning = @"Process ID < 0";
[idAttribute setValidationPredicates:@[validationPredicate]
    withValidationWarnings:@[validationWarning]];

[runEntity setProperties:runProperties];

NSDictionary *localizationDictionary = @{
    @"Property/processID/Entity/Run" : @"Process ID",
    @"Property/date/Entity/Run" : @"Date"
    @"ErrorString/Process ID < 0" : @"Process ID must not be less than 0" };
[mom setLocalizationDictionary:localizationDictionary];
```

Managed Objects

This article provides basic information about what is a managed object, how its data is stored, how you implement a custom managed object class, object life-cycle issues, and faulting. There are several other articles in the *Core Data Programming Guide* that describe other aspects of using managed objects:

- [“Creating and Deleting Managed Objects”](#) (page 56)
- [“Fetching Managed Objects”](#) (page 61)
- [“Using Managed Objects”](#) (page 67)

Basics

Managed objects are instances of the `NSManagedObject` class, or of a subclass of `NSManagedObject`, that represent instances of an entity. `NSManagedObject` is a generic class that implements all the basic behavior required of a managed object. You can create custom subclasses of `NSManagedObject`, although this is often not required. If you do not need any custom logic for a given entity, you do not need to create a custom class for that entity. You might implement a custom class, for example, to provide custom accessor or validation methods, to use non-standard attributes, to specify dependent keys, to calculate derived values, or to implement any other custom logic.

A managed object is associated with an entity description (an instance of `NSEntityDescription`) that provides metadata about the object (including the name of the entity that the object represents and the names of its attributes and relationships) and with a managed object context that tracks changes to the object graph.

A managed object is also associated with a managed object context (“context”). In a given context, a managed object provides a representation of a record in a persistent store. In a given context, for a given record in a persistent store, there can be only one corresponding managed object, but there may be multiple contexts each containing a separate managed object representing that record. Put another way, there is a to-one relationship between a managed object and the data record it represents, but a to-many relationship between the record and corresponding managed objects.

Properties and Data Storage

In some respects, an `NSManagedObject` acts like a dictionary—it is a generic container object that efficiently provides storage for the properties defined by its associated `NSEntityDescription` object. `NSManagedObject` provides support for a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). There is therefore commonly no need to define instance variables in subclasses. There are some performance considerations to bear in mind if you use large binary data objects—see “[Large Data Objects \(BLOBs\)](#)” (page 146).

Non-Standard Attributes

`NSManagedObject` provides support for a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). By default, `NSManagedObject` stores its properties as objects in an internal structure, and in general Core Data is more efficient working with storage under its own control rather than using custom instance variables.

Sometimes you want to use types that are not supported directly, such as colors and C structures. For example, in a graphics application you might want to define a `Rectangle` entity that has attributes `color` and `bounds` that are an instance of `NSColor` and an `NSRect` struct respectively. This may require you to create a subclass of `NSManagedObject`, and is described in “[Non-Standard Persistent Attributes](#)” (page 91).

Dates and Times

`NSManagedObject` represents date attributes using `NSDate` objects, and stores times internally as an `NSTimeInterval` value since the reference date (which has a time zone of GMT). Time zones are not explicitly stored—indeed you should always represent a Core Data date attribute in GMT, this way searches are normalized in the database. If you need to preserve the time zone information, you need to store a time zone attribute in your model. This may again require you to create a subclass of `NSManagedObject`.

Custom Managed Object Classes

In combination with the entity description in the managed object model, `NSManagedObject` provides a rich set of default behaviors including support for arbitrary properties and value validation. There are nevertheless many reasons why you might wish to subclass `NSManagedObject` to implement custom features. There are also, however, some things to avoid when subclassing. It’s also important to be aware that Core Data manages the life-cycle of modeled properties.

Overriding Methods

`NSManagedObject` itself customizes many features of `NSObject` so that managed objects can be properly integrated into the Core Data infrastructure. Core Data relies on `NSManagedObject`'s implementation of the following methods, which you should therefore not override: `primitiveValueForKey:`, `setPrimitiveValue:forKey:`, `isEqual:`, `hash`, `superclass`, `class`, `self`, `zone`, `isProxy`, `isKindOfClass:`, `isMemberOfClass:`, `conformsToProtocol:`, `respondsToSelector:`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, and `isFault`. You are discouraged from overriding `description`—if this method fires a fault during a debugging operation, the results may be unpredictable—and `initWithEntity:insertIntoManagedObjectContext:`. You should typically not override the key-value coding methods such as `valueForKey:` and `setValue:forKeyPath:`.

In addition to methods you should not override, there are others that if you do override you should invoke the superclass's implementation first, including `awakeFromInsert`, `awakeFromFetch`, and validation methods such as `validateForUpdate:`.

Modeled Properties

Core Data dynamically generates efficient public and primitive get and set attribute accessor methods and relationship accessor methods for properties that are defined in the entity of a managed object's corresponding managed object model. Typically, therefore, you don't need to write custom accessor methods for modeled properties.

In a managed object sub-class, you can declare the properties for modeled attributes in the interface file, but you don't declare instance variables:

```
@interface MyManagedObject : NSManagedObject

@property (nonatomic, strong) NSString *title;
@property (nonatomic, strong) NSDate *date;
@end
```

Notice that the properties are declared as `nonatomic`, and `strong`. For performance reasons, Core Data typically does not copy object values, even if the value class adopts the `NSCopying` protocol.

In the implementation file, you specify the properties as `dynamic`:

```
@implementation MyManagedObject

@dynamic title;
```



```
@dynamic date;  
@end
```

If you do need to implement custom accessor methods, there are several implementation patterns you must follow—see [“Managed Object Accessor Methods”](#) (page 44).

Object Life-Cycle—Initialization and Deallocation

Core Data “owns” the life-cycle of managed objects. With faulting and undo, you cannot make the same assumptions about the life-cycle of a managed object as you would of a standard Cocoa object—managed objects can be instantiated, destroyed, and resurrected by the framework as it requires.

When a managed object is created, it is initialized with the default values given for its entity in the managed object model. In many cases the default values set in the model may be sufficient. Sometimes, however, you may wish to perform additional initialization—perhaps using dynamic values (such as the current date and time) that cannot be represented in the model.

In a typical Cocoa class, you usually override the designated initializer (often the `init` method). In a subclass of `NSManagedObject`, there are three different ways you can customize initialization —by overriding `initWithEntity:insertIntoManagedObjectContext:`, `awakeFromInsert`, or `awakeFromFetch`. You should not override `init`. You are discouraged from overriding `initWithEntity:insertIntoManagedObjectContext:` as state changes made in this method may not be properly integrated with undo and redo. The two other methods, `awakeFromInsert` and `awakeFromFetch`, allow you to differentiate between two different situations:

- `awakeFromInsert` is invoked only once in the lifetime of an object—when it is first created.

`awakeFromInsert` is invoked immediately after you invoke

`initWithEntity:insertIntoManagedObjectContext:` or

`insertNewObjectForEntityForName:inManagedObjectContext:`. You can use `awakeFromInsert` to initialize special default property values, such as the creation date of an object, as illustrated in the following example.

```
- (void) awakeFromInsert  
{  
    [super awakeFromInsert];  
    [self setCreationDate:[NSDate date]];  
}
```

- `awakeFromFetch` is invoked when an object is re-initialized from a persistent store (during a fetch).
You can override `awakeFromFetch` to, for example, establish transient values and other caches. Change processing is explicitly disabled around `awakeFromFetch` so that you can conveniently use public set accessor methods without dirtying the object or its context. This does mean, however, that you should not manipulate relationships, as changes will not be properly propagated to the destination object or objects. Instead, you can override `awakeFromInsert` or employ any of the run loop related methods such as `performSelector:withObject:afterDelay:`.

You should typically not override `dealloc` to clear transient properties and other variables. Instead, you should override `didTurnIntoFault`. `didTurnIntoFault` is invoked automatically by Core Data when an object is turned into a fault and immediately prior to actual deallocation. You might turn a managed object into a fault specifically to reduce memory overhead (see [“Reducing Memory Overhead”](#) (page 145)), so it is important to ensure that you properly perform clean-up operations in `didTurnIntoFault`.

Validation

`NSManagedObject` provides consistent hooks for validating property and inter-property values. You typically should not override `validateValue:forKey:error:`, instead you should implement methods of the form `validate<Key>:error:`, as defined by the `NSKeyValueCoding` protocol. If you want to validate inter-property values, you can override `validateForUpdate:` and/or related validation methods.

You should not call `validateValue:forKey:error:` within custom property validation methods—if you do so you will create an infinite loop when `validateValue:forKey:error:` is invoked at runtime. If you do implement custom validation methods, you should typically not call them directly. Instead you should call `validateValue:forKey:error:` with the appropriate key. This ensures that any constraints defined in the managed object model are applied.

If you implement custom inter-property validation methods (such as `validateForUpdate:`), you should call the superclass’s implementation first. This ensures that individual property validation methods are also invoked. If there are multiple validation failures in one operation, you should collect them in an array and add the array—using the key `NSDetailedErrorsKey`—to the `userInfo` dictionary in the `NSError` object you return.

Faulting

Managed objects typically represent data held in a persistent store. In some situations a managed object may be a “fault”—an object whose property values have not yet been loaded from the external data store—see [“Faulting and Uniquing”](#) (page 110) for more details. When you access persistent property values, the fault

“fires” and the data is retrieved from the store automatically. This can be a comparatively expensive process (potentially requiring a round trip to the persistent store), and you may wish to avoid unnecessarily firing a fault (see [“Faulting Behavior”](#) (page 142)).

Although the `description` method does not cause a fault to fire, if you implement a custom `description` method that accesses the object’s persistent properties, this will cause a fault to fire. You are strongly discouraged from overriding `description` in this way.

There is no way to load individual attributes of a managed object on an as-needed basis. For patterns to deal with large attributes, see [“Large Data Objects \(BLOBs\)”](#) (page 146).

Managed Object Accessor Methods

This article explains why you might want to implement custom accessor methods for managed objects, and how to implement them for attributes and for relationships. It also illustrates how to implement primitive accessor methods.

Overview

In OS X v10.5, Core Data dynamically generates efficient public and primitive get and set attribute accessor methods and relationship accessor methods for managed object classes. Typically, therefore, there's no need for you to write accessor methods for properties that are defined in the entity of a managed object's corresponding managed object model—although you may use the Objective-C declared property feature to declare properties to suppress compiler warnings. To get the best performance—and to benefit from type-checking—you use the accessor methods directly, although they are also key-value coding (KVC) compliant so if necessary you can use standard key-value coding methods such as `valueForKey:`. You do need to write custom accessor methods if you use transient properties to support non-standard data types (see [“Non-Standard Persistent Attributes”](#) (page 91)) or if you use scalar instance variables to represent an attribute.

Custom implementation

The implementation of accessor methods you write for subclasses of `NSManagedObject` is typically different from those you write for other classes.

- If you do not provide custom instance variables, you retrieve property values from and save values into the internal store using primitive accessor methods.
- You must ensure that you invoke the relevant access and change notification methods (`willAccessValueForKey:`, `didAccessValueForKey:`, `willChangeValueForKey:`, `didChangeValueForKey:`, `willChangeValueForKey:withSetMutation:usingObjects:`, and `didChangeValueForKey:withSetMutation:usingObjects:`).

`NSManagedObject` disables automatic key-value observing (KVO) change notifications for modeled properties, and the primitive accessor methods do not invoke the access and change notification methods. For unmodeled properties, on OS X v10.4 Core Data also disables automatic KVO; on OS X v10.5 and later, Core Data adopts to `NSObject`'s behavior.

- In accessor methods for properties that are *not* defined in the entity model, you can either enable automatic change notifications or invoke the appropriate change notification methods.

You can use the Xcode data modeling tool to generate the code for accessor methods for any modeled property.

Key-value coding access pattern

The access pattern key-value coding uses for managed objects is largely the same as that used for subclasses of `NSObject`—see `valueForKey:`. The difference is that, if after checking the normal resolutions `valueForKey:` would throw an unbound key exception, the key-value coding mechanism for `NSManagedObject` checks whether the key is a modeled property. If the key matches an entity's property, the mechanism looks first for an accessor method of the form `primitiveKey`, and if that is not found then looks for a value for *key* in the managed object's internal storage. If these fail, `NSManagedObject` throws an unbound key exception (just like `valueForKey:`).

Dynamically-Generated Accessor Methods

By default, Core Data dynamically creates efficient public and primitive get and set accessor methods for modeled properties (attributes *and* relationships) of managed object classes. This includes the key-value coding mutable proxy methods such as `add<Key>Object:` and `remove<Key>s:`, as detailed in the documentation for `mutableSetValueForKey:`—managed objects are effectively mutable proxies for all their to-many relationships.

Note: If you choose to implement your own accessors, the dynamically-generated methods never replace your own code.

For example, given an entity with an attribute `firstName`, Core Data automatically generates `firstName`, `setFirstName:`, `primitiveFirstName`, and `setPrimitiveFirstName:`. *Core Data does this even for entities represented by `NSManagedObject`.* To suppress compiler warnings when you invoke these methods, you should use the Objective-C 2.0 declared properties feature, as described in “[Declaration](#)” (page 46).

The property accessor methods Core Data generates are by default (`nonatomic`, `strong`)—*this is the recommended configuration*. The methods are `nonatomic` because non-atomic accessors are more efficient than atomic accessors, and in general it is not possible to assure thread safety in a Core Data application at the level of accessor methods. (To understand how to use Core Data in a multi-threaded environment, see “[Concurrency with Core Data](#)” (page 136).)

In addition to always being `nonatomic`, dynamic properties only honor `strong` or `copy` attributes—`weak` is treated as `strong`. You should use `copy` sparingly as it increases overhead. You cannot use `copy` for relationships because `NSManagedObject` does not adopt the `NSCopying` protocol, and it's irrelevant to the behavior of to-many relationships.

Important: If you specify `copy` for a to-one relationship, you will generate a *run-time* error.

Declaration

You can use the Objective-C declared property feature to declare properties of managed object classes—you typically do this so that you can use the default accessors Core Data provides without generating compiler warnings.

You declare attributes and relationships as you would properties for any other object, as illustrated in the following example. When you declare a to-many relationship, the property type should be `NSSet *`. (The value returned from the get accessor is *not* a KVO-compliant mutable proxy—for more details, see [“To-many relationships”](#) (page 68).)

```
@interface Employee : NSManagedObject

@property (nonatomic) NSString *firstName, *lastName;
@property (nonatomic) Department *department;
@property (nonatomic) Employee *manager;
@property (nonatomic) NSSet *directReports;
@end
```

If you are not using a custom class, to suppress compiler warnings you can declare the properties in a category of `NSManagedObject`:

```
@interface NSManagedObject (EmployeeAccessors)

@property (nonatomic) NSString *firstName, *lastName;
@property (nonatomic) Department *department;
@property (nonatomic) Employee *manager;
@property (nonatomic) NSSet *directReports;
@end
```

You can use the same techniques to suppress compiler warnings for the automatically-generated to-many relationship mutator methods, for example:

```
@interface Employee (DirectReportsAccessors)

- (void)addDirectReportsObject:(Employee *)value;
- (void)removeDirectReportsObject:(Employee *)value;
- (void)addDirectReports:(NSSet *)value;
- (void)removeDirectReports:(NSSet *)value;

@end
```

To preserve encapsulation where the attribute class has a mutable subclass and it implements the `NSCopying` protocol you can also use `copy`, for example:

```
@property(n nonatomic, copy) NSString* firstName, lastName;
```

Implementation

You can specify an implementation using the `@dynamic` keyword, as shown in the following example:

```
@dynamic firstName, lastName;
@dynamic department, manager;
@dynamic directReports;
```

There should typically be no need for you to provide your own implementation of these methods, unless you want to support scalar values. The methods that Core Data generates at runtime are more efficient than those you can implement yourself.

Inheritance

If you have two subclasses of `NSManagedObject` where the parent class implements a dynamic property and its subclass (the grandchild of `NSManagedObject`) overrides the methods for the property, those overrides cannot call `super`.

```
@interface Parent : NSManagedObject
@property(n nonatomic, strong) NSString* parentString;
```

```
@end

@implementation Parent
@dynamic parentString;
@end

@interface Child : Parent
@end

@implementation Child
- (NSString *)parentString {

    // This throws a "selector not found" exception.
    return parentString.foo;
}
@end
```

Custom Attribute and To-One Relationship Accessor Methods

Important: You are strongly encouraged to use dynamic properties (that is, properties whose implementation you specify as `@dynamic`) instead of creating custom implementations for standard or primitive accessor methods.

If you want to implement your own attribute or to-one relationship accessor methods, you use the primitive accessor methods to get and set values from and to the managed object's private internal store. You must invoke the relevant access and change notification methods, as illustrated in [Listing 1](#) (page 48). `NSManagedObject`'s implementation of the primitive set accessor method handles memory management for you.

Listing 1 Implementation of a custom managed object class illustrating attribute accessor methods

```
@interface Department : NSManagedObject

@property(nonatomic, strong) NSString *name;
@end
```



```
@interface Department (PrimitiveAccessors)
- (NSString *)primitiveName;
- (void)setPrimitiveName:(NSString *)newName;
@end

@implementation Department

@dynamic name;

- (NSString *)name
{
    [self willAccessValueForKey:@"name"];
    NSString *myName = [self primitiveName];
    [self didAccessValueForKey:@"name"];
    return myName;
}

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    [self setPrimitiveName:newName];
    [self didChangeValueForKey:@"name"];
}
@end
```

The default implementation does not copy attribute values. If the attribute value may be mutable and implements the `NSCopying` protocol (as is the case with `NSString`, for example), you can copy the value in a custom accessor to help preserve encapsulation (for example, in the case where an instance of `NSMutableString` is passed as a value). This is illustrated in [Listing 2](#) (page 50). Notice also that (for the purposes of illustration) in this example the get accessor is not implemented—since it’s not implemented, Core Data will generate it automatically.

Listing 2 Implementation of a custom managed object class illustrating copying setter

```
@interface Department : NSManagedObject
{
}
@property(nonatomic, copy) NSString *name;
@end

@implementation Department

@dynamic name;

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    // NSString implements NSCopying, so copy the attribute value
    NSString *newNameCopy = [newName copy];
    [self setPrimitiveName:newNameCopy];
    [self didChangeValueForKey:@"name"];
}
@end
```

If you choose to represent an attribute using a scalar type (such as `NSInteger` or `CGFloat`), or as one of the structures supported by `NSKeyValueCoding` (`NSRect`, `NSPoint`, `NSSize`, `NSRange`), then you should implement accessor methods as illustrated in [Listing 3](#) (page 50). If you want to use any other attribute type, then you should use a different pattern, described in “[Non-Standard Persistent Attributes](#)” (page 91).

Listing 3 Implementation of a custom managed object class illustrating a scalar attribute value

```
@interface Circle : NSManagedObject
@property CGFloat radius;
@end

@implementation Circle
{
    CGFloat _radius;
}
```

```
}

- (CGFloat)radius
{
    [self willAccessValueForKey:@"radius"];
    float f = _radius;
    [self didAccessValueForKey:@"radius"];
    return f;
}

- (void)setRadius:(CGFloat)newRadius
{
    [self willChangeValueForKey:@"radius"];
    _radius = newRadius;
    [self didChangeValueForKey:@"radius"];
}

@end
```

Custom To-Many Relationship Accessor Methods

Important: You are strongly encouraged to use dynamic properties (that is, properties whose implementation you specify as `@dynamic`) instead of creating custom implementations for standard or primitive accessor methods.

You usually access to-many relationships using `mutableSetValueForKey:`, which returns a proxy object that both mutates the relationship and sends appropriate key-value observing notifications for you. There should typically be little reason to implement your own collection accessor methods for to-many relationships. If they are present, however, the framework calls the mutator methods (such as `add<Key>Object:` and `remove<Key>Object:`) when modifying a collection that represents a persistent relationship. (Fetched properties do not support the mutable collection accessor methods.) In order for this to work correctly, you must implement an `add<Key>Object:/remove<Key>Object:` pair, an `add<Key>:/remove<Key>:` pair, or both pairs. You may also implement other get accessors (such as `countOf<Key>:`, `enumeratorOf<Key>:`, and `memberOf<Key>:`) and use these in your own code, however these are not guaranteed to be called by the framework.

Important: For performance reasons, the proxy object returned by managed objects for `mutableSetValueForKey:` does not support `set<Key>:` style setters for relationships. For example, if you have a to-many relationship `employees` of a `Department` class and implement accessor methods `employees` and `setEmployees:`, then manipulate the relationship using the proxy object returned by `mutableSetValueForKey:@"employees"`, `setEmployees:` is *not* invoked. You should implement the other mutable proxy accessor overrides instead.

If you do implement collection accessors for model properties, they must invoke the relevant KVO notification methods. [Listing 4](#) (page 52) illustrates the implementation of accessor methods for a to-many relationship—`employees`—of a `Department` class. *The easiest way to generate the implementation is to select the relationship in the Xcode modeling tool and choose Design > Data Model > Copy Obj-C 2.0 Method {Declarations/Implementations} to Clipboard.*

Listing 4 A managed object class illustrating implementation of custom accessors for a to-many relationship

```
@interface Department : NSManagedObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSSet *employees;
@end

@interface Department (DirectReportsAccessors)

- (void)addEmployeesObject:(Employee *)value;
- (void)removeEmployeesObject:(Employee *)value;
- (void)addEmployees:(NSSet *)value;
- (void)removeEmployees:(NSSet *)value;

- (NSMutableSet*)primitiveEmployees;
- (void)setPrimitiveEmployees:(NSMutableSet*)value;

@end

@implementation Department
```

```
@dynamic name;
@dynamic employees;

- (void)addEmployeesObject:(Employee *)value {

    NSMutableSet *changedObjects = [NSMutableSet alloc] initWithObjects:&value count:1;

    [self willChangeValueForKey:@"employees"
         withSetMutation:NSMutableSetUnionSetMutation
         usingObjects:changedObjects];
    [[self primitiveEmployees] addObject:value];
    [self didChangeValueForKey:@"employees"
     withSetMutation:NSMutableSetUnionSetMutation
     usingObjects:changedObjects];
}

- (void)removeEmployeesObject:(Employee *)value
{
    NSMutableSet *changedObjects = [NSMutableSet alloc] initWithObjects:&value count:1;

    [self willChangeValueForKey:@"employees"
         withSetMutation:NSMutableSetMinusSetMutation
         usingObjects:changedObjects];
    [[self primitiveEmployees] removeObject:value];
    [self didChangeValueForKey:@"employees"
     withSetMutation:NSMutableSetMinusSetMutation
     usingObjects:changedObjects];
}

- (void)addEmployees:(NSMutableSet *)value
{
    [self willChangeValueForKey:@"employees"
         withSetMutation:NSMutableSetUnionSetMutation
         usingObjects:value];
}
```

```
[[self primitiveEmployees] unionSet:value];
[self didChangeValueForKey:@"employees"
 withSetMutation:NSKeyValueUnionSetMutation
 usingObjects:value];
}

- (void)removeEmployees:(NSSet *)value
{
    [self willChangeValueForKey:@"employees"
 withSetMutation:NSKeyValueMinusSetMutation
 usingObjects:value];
    [[self primitiveEmployees] minusSet:value];
    [self didChangeValueForKey:@"employees"
 withSetMutation:NSKeyValueMinusSetMutation
 usingObjects:value];
}
```

Custom Primitive Accessor Methods

Primitive accessor methods are similar to "normal" or public key-value coding compliant accessor methods, except that Core Data uses them as the most basic data methods to access data, consequently they do *not* issue key-value access or observing notifications. Put another way, they are to `primitiveValueForKey:` and `setPrimitiveValue:forKey:` what public accessor methods are to `valueForKey:` and `setValue:forKey:`.

Typically there should be little reason to implement primitive accessor methods. They are, however, useful if you want custom methods to provide direct access to instance variables for persistent Core Data properties. The example below contrasts public and primitive accessor methods for an attribute, `int16`, of type `Integer16`, stored in a custom instance variable, `nonCompliantKVCivar`.

```
// primitive get accessor
- (short)primitiveInt16 {
    return nonCompliantKVCivar;
}

// primitive set accessor
```

```
- (void)setPrimitiveInt16:(short)newInt16 {
    nonCompliantKVCivar = newInt16;
}

// public get accessor
- (short)int16 {
    short tmpValue;
    [self willAccessValueForKey: @"int16"];
    tmpValue = nonCompliantKVCivar;
    [self didAccessValueForKey: @"int16"];
    return tmpValue;
}

// public set accessor
- (void)setInt16:(short)int16 {
    [self willChangeValueForKey: @"int16"];
    nonCompliantKVCivar = int16;
    [self didChangeValueForKey:@"int16"];
}
```

Creating and Deleting Managed Objects

The Core Data Framework relieves you from the need to implement many of the mechanisms needed to manage data-bearing (model) objects. It does, though, impose the requirement that model objects are instances of, or instances of classes that inherit from, `NSManagedObject`, and that the model objects are properly integrated in to the Core Data infrastructure. This document first describes the basic pieces of the infrastructure you need to create a managed object, and how to easily instantiate an instance of a managed object and integrate it into that infrastructure. It then describes the processes that are abstracted by the convenience methods you typically use to create a managed object; how to assign an object to a particular store; and finally how to delete a managed object.

Creating, Initializing, and Saving a Managed Object

A managed object is an instance of an Objective-C class. From this perspective, it is no different from any other object you use—you can simply create an instance using `alloc`. A managed object differs from other objects in three main ways—a managed object:

- Must be an instance of `NSManagedObject` or of a class that inherits from `NSManagedObject`
- Exists in an environment defined by its managed object context
- Has an associated entity description that describes the properties of the object

In principle, there is therefore a lot of work to do to create a new managed object and properly integrate it into the Core Data infrastructure. In practice, however, this task is made easy by a convenience class method (`insertNewObjectForEntityForName:inManagedObjectContext:`) of `NSEntityDescription`. The following example shows the easiest way to create a new instance of an entity named “Employee”:

```
NSManagedObject *newEmployee = [NSEntityDescription
    insertNewObjectForEntityForName:@"Employee"
    inManagedObjectContext:context];
```

The method returns an instance of whatever class is defined in the managed object model to represent the entity, initialized with the default values given for its entity in the model.

In many cases the default values you set in the model may be sufficient. Sometimes, however, you may wish to perform additional initialization—perhaps using dynamic values (such as the current date and time) that cannot be represented in the model. In a typical Cocoa application you would override the class’s `init` method to provide this functionality. With `NSManagedObject`, you are discouraged from overriding `initWithEntity:insertIntoManagedObjectContext:`; instead, Core Data provides several other means of initializing values—these are described in [“Object Life-Cycle—Initialization and Deallocation”](#) (page 41).

Simply creating a managed object does not cause it to be saved to a persistent store. The managed object context acts as a scratchpad. You can create and register objects with it, make changes to the objects, and undo and redo changes as you wish. If you make changes to managed objects associated with a given context, those changes remain local to that context until you commit the changes by sending the context a `save:` message. At that point—provided that there are no validation errors—the changes are committed to the store.

See also [“Assigning an Object to a Store”](#) (page 59).

Behind the Scenes of Creating a Managed Object

Although `NSEntityDescription`’s convenience method makes it easy to create and configure a new managed object, it may be instructive to detail what is happening behind the scenes. If this is not of current interest, you may safely skip this section (go to [“Assigning an Object to a Store”](#) (page 59))—you are encouraged, however, to revisit this material to ensure that you fully understand the process.

In order to properly integrate a managed object into the Core Data infrastructure there are two elements you need:

- A managed object context
- An entity description

The Managed Object Context

The context is responsible for mediating between its managed objects and the rest of the Core Data infrastructure. The infrastructure is in turn responsible for, for example, translating changes to managed objects into undo actions maintained by the context, and also into operations that need to be performed on the persistent store with which the managed object is assigned.

The context is in effect also your gateway to the rest of the Core Data infrastructure. As such, it is expected that you either keep a reference to the context, or you have a means of easily retrieving it—for example, if you are developing a document-based application that uses `NSPersistentDocument`, you can use the document class’s `managedObjectContext` method.

The Entity Description

An entity description specifies (amongst other things) the name of an entity, the class used to represent the entity, and the entity's properties. The entity description is important since a given class may be used to represent more than one entity—by default all entities are represented by `NSManagedObject`. Core Data uses the entity description to determine what properties a managed object has, what needs to be saved to or retrieved from the persistent store, and what constraints there are on property values. Entity descriptions are properties of a managed object model.

Given a managed object context, you could retrieve the appropriate entity description through the persistent store coordinator as illustrated in the following example:

```
NSManagedObjectContext *context = <#Get a context#>;
NSManagedObjectModel *managedObjectModel =
    [[context persistentStoreCoordinator] managedObjectModel];
NSEntityDescription *employeeEntity =
    [[managedObjectModel entitiesByName] objectForKey:@"Employee"];
```

In practice, you would use the convenience method `entityForName:inManagedObjectContext:` of `NSEntityDescription` which does the same thing—as illustrated in the following example:

```
NSManagedObjectContext *context = /* assume this exists */;
NSEntityDescription *employeeEntity = [NSEntityDescription
    entityForName:@"Employee"
    inManagedObjectContext:context];
```

Creating a Managed Object

Fundamentally `NSManagedObject` is an Objective-C class like any other Objective-C class. Like various other classes, `NSManagedObject` imposes some constraints on instance creation. As described earlier, you must associate the new managed object instance with the entity object that defines its properties and with the managed object context that defines its environment. You cannot therefore initialize a managed object simply by sending an `init` message, you must use the designated initializer—`initWithEntity:insertIntoManagedObjectContext:`—which sets both the entity and context:

```
NSManagedObject *newEmployee = [[NSManagedObject alloc]
    initWithEntity:employeeEntity
```

```
insertIntoManagedObjectContext:context];
```

This is in effect what `NSEntityDescription`'s convenience method `insertNewObjectForEntityForName:inManagedObjectContext:` does for you—including the entity instance look-up described in “[The Entity Description](#)” (page 58). This is why you should typically use that method rather than `NSManagedObject`'s `initWithEntity:insertIntoManagedObjectContext:`.

An important additional point here is that `initWithEntity:insertIntoManagedObjectContext:` returns an instance of the class specified by the entity description to represent the entity. If you want to create a new `Employee` object and in the model you specified that the `Employee` entity should be represented by a custom class, say `Employee`, it returns an instance of `Employee`. If you specified that the `Employee` entity should be represented by `NSManagedObject`, it returns an instance of `NSManagedObject`.

Assigning an Object to a Store

Typically there is only one persistent store for a given entity, and Core Data automatically ensures that new objects are saved to this store when the object's managed object context is saved. Sometimes, however, you may have multiple writable stores for a given entity—for example you may store some data in a specific document and some in a common global repository (say, a store in the user's Application Support folder). In this situation you must specify the store in which the object is to reside.

You specify the store for an object using the `NSManagedObjectContext` method, `assignObject:toPersistentStore:`. This method takes as its second argument the identifier for a store. You obtain the store identifier from the persistent store coordinator, using for example `persistentStoreForURL:`. The following example illustrates the complete process of creating a new managed object and assigning it to a global store.

```
NSURL *storeURL = <#URL for path to global store#>;

id globalStore = [[context persistentStoreCoordinator]
    persistentStoreForURL:storeURL];

NSManagedObject *newEmployee = [NSEntityDescription
    insertNewObjectForEntityForName:@"Employee"
    inManagedObjectContext:context];

[context assignObject:newEmployee toPersistentStore:globalStore];
```

Of course, the object is not saved to the store until the managed object context is saved.

Deleting a Managed Object

Deleting a managed object is straightforward. You simply send its managed object context a `deleteObject:` message, passing the object you want to delete as the argument.

```
[aContext deleteObject:aManagedObject];
```

This removes the managed object from the object graph. Just as a new object is not saved to the store until the context is saved, a deleted object is not removed from the store until the context is saved.

Relationships

When you delete a managed object it is important to consider its relationships and in particular the delete rules specified for the relationships. If all of a managed object's relationship delete rules are Nullify, then for that object at least there is no additional work to do (you may have to consider other objects that were at the destination of the relationship—if the inverse relationship was either mandatory or had a lower limit on cardinality, then the destination object or objects might be in an invalid state). If a relationship delete rule is Cascade, then deleting one object may result in the deletion of others. If a rule is Deny, then before you delete an object you must remove the destination object or objects from the relationship, otherwise you will get a validation error when you save. If a delete rule is No Action, then you must ensure that you take whatever steps are necessary to ensure the integrity of the object graph. For more details, see [“Relationship Delete Rules”](#) (page 84).

Deleted status and notifications

You can find out if a managed object has been marked for deletion by sending it an `isDeleted` message. If the return value is YES, this means that the object will be deleted during the next save operation, or put another way, that the object is marked deleted for the current (pending) transaction. In addition, when you send a managed object context a `deleteObject:` message, the context posts a `NSManagedObjectContextObjectsDidChangeNotification` notification that includes the newly-deleted object in its list of deleted objects. Note, however, that an object being marked for deletion from a context is not the same as its being marked for deletion from a persistent store. If an object is created and deleted within the same transaction—that is, without an intervening save operation—it will not appear in the array returned by `NSManagedObjectContext's deletedObjects` method or in the set of deleted objects in a `NSManagedObjectContextDidSaveNotification` notification.

Fetching Managed Objects

This article describes how to fetch managed objects and discusses some considerations for ensuring that fetching is efficient. It also shows how you can use `NSEntityDescription` objects to retrieve particular values. For code snippets that you can use for various scenarios, see *Core Data Snippets*.

Fetching Managed Objects

You fetch managed objects by sending a fetch request to a managed object context. You first create a fetch request. As a minimum you must specify an entity for the request. You can get the entity from your managed object model using the `NSEntityDescription` method `entityForName:inManagedObjectContext:`. You may also set a predicate (for details about creating predicates, see *Predicate Programming Guide*), sort descriptors, and other attributes if necessary. You retrieve objects from the context using `executeFetchRequest:error:`, as illustrated in the example below.

Listing 1 Example of creating and executing a fetch request

```
NSManagedObjectContext *moc = [self managedObjectContext];
NSEntityDescription *entityDescription = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:moc];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:entityDescription];

// Set example predicate and sort orderings...
NSNumber *minimumSalary = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"(lastName LIKE[c] 'Worsley') AND (salary > %@)", minimumSalary];
[request setPredicate:predicate];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:@"firstName" ascending:YES];
[request setSortDescriptors:@[sortDescriptor]];
```

```
NSError *error;
NSArray *array = [moc executeFetchRequest:request error:&error];
if (array == nil)
{
    // Deal with error...
}
```

You cannot fetch using a predicate based on transient properties (although you can use transient properties to filter in memory yourself). Moreover, there are some interactions between fetching and the type of store—for details, see [“Store Types and Behaviors”](#) (page 130). To summarize, though, if you execute a fetch directly, you should typically not add *Objective-C-based* predicates or sort descriptors to the fetch request. Instead you should apply these to the results of the fetch. If you use an array controller, you may need to subclass `NSArrayController` so you can have it not pass the sort descriptors to the persistent store and instead do the sorting after your data has been fetched.

If you use multiple persistence stacks in your application, or if multiple applications might access (and modify) the same store simultaneously, you can perform fetches to ensure that data values are current—see [“Ensuring Data Is Up-to-Date”](#) (page 76).

Retrieving Specific Objects

If your application uses multiple contexts and you want to test whether an object has been deleted from a persistent store, you can create a fetch request with a predicate of the form `self == %@`. The object you pass in as the variable can be either a managed object or a managed object ID, as in the following example:

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entity =
    [NSEntityDescription entityForName:@"Employee"
     inManagedObjectContext:managedObjectContext];
[request setEntity:entity];

NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"self == %@", targetObject];
[request setPredicate:predicate];

NSError *error;
```

```
NSArray *array = [managedObjectContext executeFetchRequest:request error:&error];
if (array != nil) {
    NSUInteger count = [array count]; // May be 0 if the object has been deleted.
    //
}
else {
    // Deal with error.
}
```

The count of the array returned from the fetch will be 0 if the target object has been deleted. If you need to test for the existence of several objects, it is more efficient to use the `IN` operator than it is to execute multiple fetches for individual objects, for example:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"self IN %@",
                                                             arrayOfManagedObjectIDs];
```

Fetching Specific Values

Sometimes you don't want to fetch actual managed objects; instead, you just want to retrieve—for example—the largest or smallest value of a particular attribute. In OS X v10.6 and later and on iOS, you can use `NSExpressionDescription` to directly retrieve values that meet your criteria.

You create a fetch request object and set its entity, just as you would for a normal fetch, but:

- You specify that the fetch should return dictionaries.
You send the fetch request a `setResultType:` message with the argument `NSDictionaryResultType`.
- You create instances of `NSExpressionDescription` to specify the properties you want to retrieve.
If you just want a single value—such as the largest salary in an `Employee` table—then you just create a single expression description.

There are a number of steps to follow to create and use the expression description.

1. First you need to create expressions (instances of `NSExpression`) to represent the key-path for the value you're interested in, and to represent the function you want to apply (such as `max:` or `min:`):

```
NSExpression *keyPathExpression = [NSExpression
expressionForKeyPath:@"salary"];
```

```
NSEExpression *maxSalaryExpression = [NSEExpression  
expressionForFunction:@"max:"  
  
arguments:[NSArray  
arrayWithObject:keyPathExpression]]];
```

For a full list of supported functions, see `expressionForFunction:arguments:`.

2. You then create the expression description and set its name, expression, and result type.

The name is the key that will be used in the dictionary for the return value. If you want to retrieve multiple values—such as the largest and the smallest salaries in an `Employee` table—the name of each expression description must be unique for a given fetch request.

```
NSEExpressionDescription *expressionDescription = [[NSEExpressionDescription  
alloc] init];  
[expressionDescription setName:@"maxSalary"];  
[expressionDescription setExpression:maxSalaryExpression];  
[expressionDescription setExpressionResultType:NSDecimalAttributeType];
```

3. Finally, you set the request's properties to fetch just the property represented by the expression:

```
[request setPropertiesToFetch:[NSArray  
arrayWithObject:expressionDescription]];
```

You can then execute the fetch request just as you would any other (using `executeFetchRequest:error:`). The request returns, though, an array containing a dictionary whose keys and values correspond to the names of the expression descriptions and the values you requested.

The following example illustrates how you can get the minimum value of an attribute “creationDate” in an entity named “Event”.

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];  
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"  
inManagedObjectContext:context];  
[request setEntity:entity];  
  
// Specify that the request should return dictionaries.  
[request setResultType:NSDictionaryResultType];
```



```
// Create an expression for the key path.
NSEExpression *keyPathExpression = [NSEExpression
expressionForKeyPath:@"creationDate"];

// Create an expression to represent the minimum value at the key path 'creationDate'
NSEExpression *minExpression = [NSEExpression expressionForFunction:@"min:"
arguments:[NSArray arrayWithObject:keyPathExpression]];

// Create an expression description using the minExpression and returning a date.
NSEExpressionDescription *expressionDescription = [[NSEExpressionDescription alloc]
init];

// The name is the key that will be used in the dictionary for the return value.
[expressionDescription setName:@"minDate"];
[expressionDescription setExpression:minExpression];
[expressionDescription setExpressionResultType:NSDateAttributeType];

// Set the request's properties to fetch just the property represented by the
expressions.
[request setPropertiesToFetch:[NSArray arrayWithObject:expressionDescription]];

// Execute the fetch.
NSError *error = nil;
NSArray *objects = [managedObjectContext executeFetchRequest:request error:&error];
if (objects == nil) {
    // Handle the error.
}
else {
    if ([objects count] > 0) {
        NSLog(@"Minimum date: %@", [[objects objectAtIndex:0]
valueForKey:@"minDate"]);
    }
}
}
```

Fetching and Entity Inheritance

If you define an entity inheritance hierarchy (see [“Entity Inheritance”](#) (page 26)), when you specify a super-entity as the entity for a fetch request, the request returns all matching instances of the super-entity and of sub-entities. In some applications, you might specify a super-entity as being abstract (see [“Abstract Entities”](#) (page 27)). To fetch matching instances of all concrete sub-entities of the abstract entity, you set the entity for fetch specification to be the abstract entity. In the case of the domain described in [“Abstract Entities,”](#) if you specify a fetch request with the Graphic entity, the fetch returns matching instances of Circle, TextArea, and Line.

Using Managed Objects

This document describes issues related to using and manipulating managed objects in your application.

Accessing and Modifying Properties

Core Data automatically generates efficient public and primitive get and set accessor methods for modeled properties (attributes *and* relationships) of managed object classes (see [“Managed Object Accessor Methods”](#) (page 44)). When you access or modify properties of a managed object, you should use these methods directly.

Most relationships are inherently bidirectional. Any changes made to the relationships between objects should maintain the integrity of the object graph. Provided that you have correctly modeled a relationship in both directions and set the inverses, modifying one end of a relationship automatically updates the other end—see [“Manipulating Relationships and Object Graph Integrity”](#) (page 85).

Attributes and to-one relationships

You access attributes and to-one relationships of a managed object using standard accessor methods or using the Objective-C 2.0 dot syntax (see “Dot Syntax” in *The Objective-C 2.0 Programming Language*) as illustrated in the following code fragment:

```
NSString *firstName = [anEmployee firstName];  
Employee *manager = anEmployee.manager;
```

Similarly, you can use either standard accessor methods or the dot syntax to modify attributes; for example:

```
newEmployee.firstName = @"Stig";  
[newEmployee setManager:manager];
```

In the cases of both getters and setters, the dot syntax is exactly equivalent to standard method invocation. For example, the following two statements use identical code paths:

```
[[aDepartment manager] setSalary:[NSNumber numberWithInt:100000]];
```

```
aDepartment.manager.salary = [NSNumber numberWithInt:100000];
```

Note: If you get a compiler warning or error when trying to use custom accessors, you need to either declare a category of `NSManagedObject` that declares the relevant properties, or (typically better) implement a custom subclass of `NSManagedObject` for your entity that declares the properties (see [“Managed Object Accessor Methods”](#) (page 44)).

You can also use key-value coding (KVC) to get or set the value of a simple attribute as illustrated in the following code fragment. Using KVC, though, is considerably less efficient than using accessor methods, so you should only use KVC when necessary (for example when you are choosing the key or key path dynamically).

```
[newEmployee setValue:@"Stig" forKey:@"firstName"];  
[aDepartment setValue:[NSNumber numberWithInt:100000]  
forKeyPath:@"manager.salary"];
```

You must, however, change attribute values in a KVC-compliant fashion. For example, the following typically represents a programming error:

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Stig"];  
[newEmployee setFirstName:mutableString];  
[mutableString setString:@"Laura"];
```

For mutable values, you should either transfer ownership of the value to Core Data, or implement custom accessor methods to always perform a copy. The previous example may not represent an error if the class representing the `Employee` entity declared the `firstName` property (copy) (or implemented a custom `setFirstName:` method that copied the new value). In this case, after the invocation of `setString:` (in the third code line) the value of `firstName` would then still be “Stig” and not “Laura”.

There should typically be no reason to invoke the primitive accessor methods except within custom accessor methods (see [“Managed Object Accessor Methods”](#) (page 44)).

To-many relationships

To access a to-many relationship (whether the destination of a one-to-many relationship or a many-to-many relationship), you use the standard get accessor method. A to-many relationship is represented by a set, as illustrated in the following code fragment:

```
NSSet *managersPeers = [managersManager directReports];  
NSSet *departmentsEmployees = aDepartment.employees;
```

When you access the destination of a relationship, you may initially get a fault object (see [“Faulting and Uniquing”](#) (page 110))—the fault fires automatically if you make any changes to it. (There’s typically no need to know whether the relationship is a fault, however you can find out using `NSManagedObject’s` `hasFaultForRelationshipNamed:` method.)

You can in principle manipulate an entire to-many relationship in the same way you do a to-one relationship, using either a custom accessor method or (more likely) key-value coding, as in the following example.

```
NSSet *newEmployees = [NSSet setWithObjects:employee1, employee2, nil];  
[aDepartment setEmployees:newEmployees];  
  
NSSet *newDirectReports = [NSSet setWithObjects:employee3, employee4, nil];  
manager.directReports = newDirectReports;
```

Typically, however, you do not want to set an entire relationship, instead you want to add or remove a single element at a time. To do this, you should use `mutableSetValueForKey:` or one of the automatically-generated relationship mutator methods (see [“Dynamically-Generated Accessor Methods”](#) (page 45)):

```
NSMutableSet *employees = [aDepartment mutableSetValueForKey:@"employees"];  
[employees addObject:newEmployee];  
[employees removeObject:firedEmployee];  
  
// or  
[aDepartment addEmployeesObject:newEmployee];  
[aDepartment removeEmployeesObject:firedEmployee];
```

It is important to understand the difference between the values returned by the dot accessor and by `mutableSetValueForKey:`. `mutableSetValueForKey:` returns a mutable proxy object. If you mutate its contents, it will emit the appropriate key-value observing (KVO) change notifications for the relationship. The dot accessor simply returns a set. If you manipulate the set as shown in this code fragment:

```
[aDepartment.employees addObject:newEmployee]; // do not do this!
```

then KVO change notifications are *not* emitted and the inverse relationship is not updated correctly.

Recall that the dot simply invokes the accessor method, so for the same reasons:

```
[[aDepartment employees] addObject:newEmployee]; // do not do this, either!
```

Saving Changes

Simply modifying a managed object does not cause the changes to be saved to a store. The managed object context acts as a scratchpad. You can create and register managed objects with it, make changes to the objects, and undo and redo changes as you wish. If you make changes to managed objects associated with a given context, those changes remain local to that context until you commit the changes by sending the context a `save:` message. At that point—provided that there are no validation errors—the changes are committed to the store. As a corollary, simply creating a managed object does not cause it to be saved to a persistent store, and deleting a managed object does not cause the record to be removed from the store—you must save the context to commit the change.

See also [“Ensuring Data Is Up-to-Date”](#) (page 76).

Managed Object IDs and URIs

An `NSManagedObjectID` object is a universal identifier for a managed object, and provides basis for uniquing in the Core Data Framework. A managed object ID uniquely identifies the same managed object both between managed object contexts in a single application, and in multiple applications (as in distributed systems). Like the primary key in the database, an identifier contains the information needed to exactly describe an object in a persistent store, although the detailed information is not exposed. The framework completely encapsulates the “external” information and presents a clean object oriented interface.

```
NSManagedObjectID *moID = [managedObject objectID];
```

There are two forms of an object ID. When a managed object is first created, Core Data assigns it a temporary ID; only if it is saved to a persistent store does Core Data assign a managed object a permanent ID. You can readily discover whether an ID is temporary:

```
BOOL isTemporary = [[managedObject objectID] isTemporaryID];
```

You can also transform an object ID into a URI representation:

```
NSURL *moURI = [[managedObject objectID] URIRepresentation];
```

Given a managed object ID or a URI, you can retrieve the corresponding managed object using `managedObjectIDForURIRepresentation:` or `objectWithID:`.

An advantage of the URI representation is that you can archive it—although in many cases you should not archive a temporary ID since this is obviously subject to change. You could, for example, store archived URIs in your application’s user defaults to save the last selected group of objects in a table view. You can also use URIs to support copy and paste operations (see [“Copying and Copy and Paste”](#) (page 71)) and drag and drop operations (see [“Drag and Drop”](#) (page 73)).

You can use object IDs to define “weak” relationships across persistent stores (where no hard join is possible). For example, for a weak to-many relationship you store as archived URIs the IDs of the objects at the destination of the relationship, and maintain the relationship as a transient attribute derived from the object IDs.

You can sometimes benefit from creating your own unique ID (UUID) property which can be defined and set for newly inserted objects. This allows you to efficiently locate specific objects using predicates (though before a save operation new objects can be found only in their original context).

Copying and Copy and Paste

It is difficult to solve the problem of copying, or supporting copy and paste, in a generic way for managed objects. You need to determine on a case-by-case basis what properties of a managed object you actually want to copy.

Copying Attributes

If you just want to copy a managed object’s attributes, then in many cases the best strategy may be in the copy operation to create a dictionary (property list) representation of a managed object, then in the paste operation to create a new managed object and populate it using the dictionary. You can use the managed object’s ID (described in [“Managed Object IDs and URIs”](#) (page 70)) to support copy and paste. Note, however, that the technique needs to be adapted to allow for copying of new objects.

A new, unsaved, managed object has a temporary ID. If a user performs a copy operation and then a save operation, the managed object’s ID changes and the ID recorded in the copy will be invalid in a subsequent paste operation. To get around this, you use a “lazy write” (as described in [“Copy and Paste”](#)). In the copy operation, you declare your custom type but if the managed object’s ID is temporary you do not write the data—but you do keep a reference to the original managed object. In the `pasteboard:provideDataForType:` method you then write the current ID for the object.

As a further complication, it is possible that the ID is still temporary during the paste operation, yet you must still allow for the possibility of future paste operations after an intervening save operation. You must therefore re-declare the type on the pasteboard to set up lazy pasting again, otherwise the pasteboard will retain the temporary ID. You cannot invoke `addTypes:owner:` during `pasteboard:provideDataForType:`, so you must use a delayed perform—for example:

```
- (void)pasteboard:(NSPasteboard *)sender provideDataForType:(NSString *)type
{
    if ([type isEqualToString:MyM0IDType]) {
        // assume cachedManagedObject is object originally copied
        NSManagedObjectID *moID = [cachedManagedObject objectID];
        NSURL *moURI = [moID URIRepresentation];
        [sender setString:[moURI absoluteString] forType:MyM0IDType];
        if ([moID isTemporaryID]) {
            [self performSelector:@selector(clearM0IDInPasteboard:)
                withObject:sender afterDelay:0];
        }
    }
    // implementation continues...
}

- (void)clearM0IDInPasteboard:(NSPasteboard *)pb
{
    [pb addTypes:[NSArray arrayWithObject:MyM0IDType] owner:self];
}
```

Copying Relationships

If you want to copy relationships you also need to consider the objects related to those first tier of related objects—if you are not careful, it is possible that you will copy the whole object graph (which may not be what you want!). If you want to copy a to-one relationship, you need to decide whether the copy of the destination should be a new object or a reference. If it is a reference, what should happen to the inverse relationship to the original object—should making a copy redefine relationships between other objects? You need to make similar decisions for to-many relationships.

Drag and Drop

You can perform drag and drop operations with managed objects—such as, for example, transferring an object from one relationship to another—using a URI representation, as described in [“Managed Object IDs and URIs”](#) (page 70).

```
NSURL *moURI = [[managedObject objectID] URIRepresentation];
```

You can put the URI on a dragging pasteboard, from which you can later retrieve it and recreate a reference to the original managed object using the persistent store coordinator, as illustrated in the following code sample.

```
NSURL *moURL = // get it from the pasteboard ...
NSManagedObjectID *moID = [[managedObjectContext persistentStoreCoordinator]
    managedObjectIDForURIRepresentation:moURL];
// assume moID non-nil...
NSManagedObject *mo = [managedObjectContext objectWithID:moID];
```

This assumes that drag and drop is “within a single persistence stack”—that is, that if there is more than one managed object context involved that they use a shared persistent store coordinator—or that the object(s) being dragged and dropped are in a store referenced by the persistent store coordinators.

If you want to copy-and-paste via drag-and-drop then you must put a suitable representation of the managed object onto the pasteboard, get the representation during the drop method, and initialize a new managed object using the representation (see [“Copying and Copy and Paste”](#) (page 71)).

Validation

The Core Data framework provides a clean infrastructure for supporting validation, both through logic encapsulated in the object model and through custom code. In the managed object model, you can specify constraints on values that a property may have (for example, an Employee's salary cannot be negative, or that every employee must belong to a Department). There are two forms of custom validation methods—those that follow standard key-value coding conventions (see [“Key-Value Validation”](#)) to validate a value for a single attribute, and a special set (`validateForInsert:`, `validateForUpdate:`, and `validateForDelete:`) for validating the whole object at different stages of its life-cycle (insertion, update, and deletion). The latter may be particularly useful for validating combinations of values—for example, to ensure that an employee can be entered into a stock purchase plan only if their period of service exceeds a given length and their pay grade is at or above a certain level.

Model-based constraints are checked and validation methods are invoked automatically before changes are committed to the external store to prevent invalid data being saved. You can also invoke them programmatically whenever necessary. You validate individual values using `validateValue:forKey:error:`. The managed object compares the new value with the constraints specified in the model, and invokes any custom validation method (of the form `validate<Key>:error:`) you have implemented. Even if you implement custom validation methods, you should typically not call custom validation methods directly. This ensures that any constraints defined in the managed object model are applied.

Undo Management

The Core Data framework provides automatic support for undo and redo. Undo management even extends to transient properties (properties that are not saved to persistent store, but are specified in the managed object model).

Managed objects are associated with a managed object context. Each managed object context maintains an undo manager. The context uses key-value observing to keep track of modifications to its registered objects. You can make whatever changes you want to a managed object's properties using normal accessor methods, key-value coding, or through any custom key-value-observing compliant methods you define for custom classes, and the context registers appropriate events with its undo manager.

To undo an operation, you simply send the context an `undo` message and to redo it send the context a `redo` message. You can also roll back all changes made since the last save operation using `rollback` (this also clears the undo stack) and reset a context to its base state using `reset`.

You also can use other standard undo manager functionality, such as grouping undo events. Core Data, though, queues up the undo registrations and adds them in a batch (this allows the framework to coalesce changes, negate contradictory changes, and perform various other operations that work better with hindsight than immediacy). If you use methods other than `beginUndoGrouping` and `endUndoGrouping`, to ensure that any queued operations are properly flushed you must first therefore send the managed object context a `processPendingChanges` message.

For example, in some situations you want to alter—or, specifically, disable—undo behavior. This may be useful if you want to create a default set of objects when a new document is created (but want to ensure that the document is not shown as being dirty when it is displayed), or if you need to merge new state from another thread or process. In general, to perform operations without undo registration, you send an undo manager a `disableUndoRegistration` message, make the changes, and then send the undo manager an `enableUndoRegistration` message. Before each, you send the context a `processPendingChanges` message, as illustrated in the following code fragment:

```
NSManagedObjectContext *moc = ...;
[moc processPendingChanges]; // flush operations for which you want undos
[[moc undoManager] disableUndoRegistration];
// make changes for which undo operations are not to be recorded
[moc processPendingChanges]; // flush operations for which you do not want undos
[[moc undoManager] enableUndoRegistration];
```

Faults

Managed objects typically represent data held in a persistent store. In some situations a managed object may be a “fault”—an object whose property values have not yet been loaded from the external store. When you access persistent property values, a fault “fires” and its persistent data is retrieved automatically from the store. In some circumstances you may explicitly turn a managed object into a fault (typically to ensure that its values are up to date, using `NSManagedObjectContext`'s `refreshObject:mergeChanges:`). More commonly you encounter faults when traversing relationships.

When you fetch a managed object, Core Data does not automatically fetch data for other objects to which it has relationships (see [“Faulting Limits the Size of the Object Graph”](#) (page 110)). Initially, an object's relationships are represented by faults (unless the destination object has already been fetched—see [“Uniquing Ensures a Single Managed Object per Record per Context”](#) (page 112)). If, however, you access the relationship's destination object or objects, their data are retrieved automatically for you. For example, suppose you fetch a single `Employee` object from a persistent store when an application first launches, then (assuming these exist in the persistent store) its `manager` and `department` relationships are represented by faults. You can nevertheless ask for the employee's manager's last name as shown in the following code example:

```
NSString *managersName =
    [[anEmployee valueForKey:@"manager"] valueForKey:@"lastName"];
```

or more easily using key paths:

```
NSString *managersName =
    [anEmployee valueForKeyPath:@"manager.lastName"];
```

In this case, the data for destination `Employee` object (the manager) is retrieved for you automatically.

There is a subtle but important point here. Notice that, in order to traverse a relationship—in this example to find an employee’s manager—you do not have to explicitly *fetch* the related objects (that is, you do not create and execute a fetch request). You simply use key-value coding (or if you have implemented them, accessor methods) to retrieve the destination object (or objects) and they are created for you automatically by Core Data. For example, you could ask for an employee’s manager’s manager’s department’s name like this:

```
NSString *departmentName = [anEmployee  
    valueForKeyPath:@"manager.manager.department.name"];
```

(This assumes, of course, that the employee is at least two levels deep in the management hierarchy.) You can also use collection operator methods. You could find the salary overhead of an employee's department like this:

```
NSNumber *salaryOverhead = [anEmployee  
    valueForKeyPath:@"department.employees.@sum.salary"];
```

In many cases, your initial fetch retrieves a starting node in the object graph and thereafter you do not execute fetch requests, you simply follow relationships.

Ensuring Data Is Up-to-Date

If two applications are using the same data store, or a single application has multiple persistence stacks, it is possible for managed objects in one managed object context or persistent object store to get out of sync with the contents of the repository. If this occurs, you need to “refresh” the data in the managed objects, and in particular the persistent object store (the snapshots) to ensure that the data values are current.

Refreshing an object

Managed objects that have been realized (their property values have been populated from the persistent store) as well as pending updated, inserted, or deleted objects, are never changed by a fetch operation without developer intervention. For example, consider a scenario in which you fetch some objects and modify them in one editing context; meanwhile in another editing context you edit the same data and commit the changes. If in the first editing context you then execute a new fetch which returns the same objects, you do not see the newly-committed data values—you see the existing objects in their current in-memory state.

To refresh a managed object's property values, you use the managed object context method `refreshObject:mergeChanges:`. If the `mergeChanges` flag is YES, the method merges the object's property values with those of the object available in the persistent store coordinator; if the flag is NO, the method simply

turns an object back into a fault without merging (which also causes strong references to other related managed objects to be broken, so you can use this method to trim the portion of your object graph you want to hold in memory).

An object's staleness interval is the time that has to pass until the store re-fetches the snapshot. This therefore only affects firing faults—moreover it is only relevant for SQLite stores (the other stores never re-fetch because the entire data set is kept in memory).

Merging changes with transient properties

If you use `refreshObject:mergeChanges:` with the `mergeChanges` flag YES, then any transient properties are restored to their pre-refresh value after `awakeFromFetch` is invoked. This means that, if you have a transient property with a value that depends on a property that is refreshed, the transient value may become out of sync.

Consider an application in which you have a `Person` entity with attributes `firstName` and `lastName`, and a *cached* transient derived property, `fullName` (in practice it might be unlikely that a `fullName` attribute would be cached, but the example is easy to understand). Suppose also that `fullName` is calculated and cached in a custom `awakeFromFetch` method.

A `Person`, currently named "Sarit Smith" in the persistent store, is edited in two managed object contexts:

- In context one, the corresponding instance's `firstName` is changed to "Fiona" (which causes the cached `fullName` to be updated to "Fiona Smith") and the context saved.

In the persistent store, the person is now "Fiona Smith".

- In context two, corresponding instance's `lastName` is changed to "Jones", which causes the cached `fullName` to be updated to "Sarit Jones".

The object is then refreshed with the `mergeChanges` flag YES. The refresh fetches "Fiona Smith" from the store.

- `firstName` was *not* changed prior to the refresh; the refresh causes it to be updated to the new value from the persistent store, so it is now "Fiona".
- `lastName` was changed prior to the refresh; so, after the refresh, it is set back to its modified value—"Jones".
- The *transient* value, `fullName`, was also changed prior to the refresh. After the refresh, its value is restored to "Sarit Jones" (to be correct, it should be "Fiona Jones").

The example shows that, because pre-refresh values are applied *after* `awakeFromFetch`, you cannot use `awakeFromFetch` to ensure that a transient value is properly updated following a refresh (or if you do, the value will subsequently be overwritten). In these circumstances, the best solution is to use an additional instance variable to note that a refresh has occurred and that the transient value should be recalculated. For example,

in the `Person` class you could declare an instance variable `fullNameIsValid` of type `BOOL` and implement the `didTurnIntoFault` method to set the value to `NO`. You then implement a custom accessor for the `fullName` attribute that checks the value of `fullNameIsValid`—if it is `NO`, then the value is recalculated.

Object Lifetime Management

The life-cycle of the data a managed object represents is largely independent of the lifetime of individual managed object instances. In order to add a record to a persistent store, you must allocate and initialize a managed object—and then save the managed object context. When you remove a record from a persistent store, you should ensure its corresponding managed object is eventually deallocated. In between these events, however, you can create and destroy any number of instances of a managed object that represent the same record in a given persistent store.

The Role of the Managed Object Context

Managed objects know what managed object context they're associated with, and managed object contexts know what managed objects they contain. *By default*, though, the references between a managed object and its context are weak. This means that in general you cannot rely on a context to ensure the longevity of a managed object instance, and you cannot rely on the existence of a managed object to ensure the longevity of a context. Put another way, just because you fetched an object doesn't mean it will stay around.

The exception to this rule is that a managed object context maintains a strong reference to any changed (inserted, deleted, and updated) objects until the pending transaction is committed (with a `save:`) or discarded (with a `reset` or `rollback`). Note that the undo manager may also keep strong references to changed objects—see [“Change and Undo Management”](#) (page 80).

You can change a context's default behavior such that it does keep strong references to its managed objects by sending it a `setRetainsRegisteredObjects:` message (with the argument `YES`)—this makes the managed objects' lifetimes depend on the context's. This can be a convenience if you are caching smaller data sets in memory—for example if the context controls a temporary set of objects that may persist beyond a single event cycle, such as when editing in a sheet. It can also be useful if you are using multiple threads and passing data between them—for example if you are performing a background fetch and passing object IDs to the main thread. The background thread needs to keep strong references to the objects it pre-fetched for the main thread until it knows the main thread has actually used the object IDs to fault local instances into itself.

You should typically use a separate container to keep strong references only those managed objects you really need. You can use an array or dictionary, or an object controller (for example an `NSArrayController` instance) that has strong references the objects it manages. The managed objects you don't need will then be deallocated when possible (for example, when relationships are cleared).

If you have finished with a managed object context, or for some other reason you want to “disconnect” a context from its persistent store coordinator, you should *not* set the context’s coordinator to `nil`:

```
// This will raise an exception.  
[myManagedObjectContext setPersistentStoreCoordinator:nil];
```

Instead, you should simply relinquish ownership of the context and allow it to be deallocated normally.

Breaking Relationship Strong Reference Cycles

When you have relationships between managed objects, each object maintains a strong reference to the object or objects to which it is related. This can cause strong reference cycles. To ensure that reference cycles are broken, when you’re finished with an object you can use the managed object context method `refreshObject:mergeChanges:` to turn it into a fault.

You typically use `refreshObject:mergeChanges:` to refresh a managed object’s property values. If the `mergeChanges` flag is YES, the method merges the object’s property values with those of the object available in the persistent store coordinator. If the flag is NO, however, the method simply turns an object back into a fault without merging, which causes it to break strong references to related managed objects. This breaks the strong reference cycle between that managed object and the other managed objects.

Note that, of course, before a managed object can be deallocated there must be no strong references to it, including from outside of Core Data. See also “[Change and Undo Management](#)” (page 80).

Change and Undo Management

A context keeps strong references to managed objects that have pending changes (insertions, deletions, or updates) until the context is sent a `save:`, `reset`, `rollback`, or `dealloc` message, or the appropriate number of undos to undo the change.

The undo manager associated with a context keeps strong references to any changed managed objects. By default, in OS X the context’s undo manager keeps an unlimited undo/redo stack. To limit your application’s memory footprint, you should make sure that you scrub (using `removeAllActions`) the context’s undo stack as and when appropriate. Unless you keep a strong reference to a context’s undo manager, it is deallocated with its context.

If you do not intend to use Core Data's undo functionality, you can reduce your application's resource requirements by setting the context's undo manager to `nil`. This may be especially beneficial for background worker threads, as well as for large import or batch operations.

Relationships and Fetched Properties

There are a number of things you have to decide when you create a relationship. What is the destination entity? Is it a to-one or a to-many? Is it optional? If it's a to-many, are there maximum or minimum numbers of objects that can be in the relationship? What should happen when the source object is deleted? You can provide answers to all these in the model. One of the particularly interesting cases is a many-to-many relationship; there are two ways to model these, and which one you choose will depend on the semantics of your schema.

When you modify an object graph, it is important to maintain referential integrity. Core Data makes it easy for you to alter relationships between managed objects without causing referential integrity errors. Much of this behavior derives from the relationship descriptions specified in the managed object model.

Core Data does not let you create relationships that cross stores. If you need to create a relationship from objects in one store to objects in another, you should consider using fetched properties.

Relationship Definitions in the Model

Creating a relationship in a managed object model is straightforward, but there are a number of aspects of a relationship that you need to specify properly. The most immediately obvious features are the relationship's name, the destination entity, and the **cardinality** (is it a to-one relationship, or a to-many relationship). The most important features with respect to object graph integrity, however, are the inverse relationship and the delete rule. The validity of the graph is affected by the settings for optionality and for maximum and minimum count.

Relationship Fundamentals

A relationship specifies the entity, or the parent entity, of the objects at the destination. This can be the same as the entity at the source (a **reflexive** relationship). Relationships do not have to be homogeneous. If the Employee entity has two sub-entities, say Manager and Flunky, then a given department's employees may be made up of Employees (assuming Employee is not an abstract entity), Managers, Flunkies, or any combination thereof.

You can specify a relationship as being to-one or to-many. To-one relationships are represented by a reference to the destination object. To-many relationships are represented by mutable sets (although fetched properties are represented by arrays). Implicitly, “to-one” and “to-many” typically refer to “one-to-one” and “one-to-many”

relationships respectively. A many-to-many relationship is one where a relationship and its inverse are both to-many. These present some additional considerations, and are discussed in greater detail in [“Many-to-Many Relationships”](#) (page 86).

You can also put upper and lower limits on the number of objects at the destination of a to-many relationship. The lower limit does not have to be zero. You can if you want specify that the number of employees in a department must lie between 3 and 40. You also specify a relationship as either optional or not optional. If a relationship is not optional, then in order to be valid there must be an object or objects at the destination of the relationship.

Cardinality and optionality are orthogonal properties of a relationship. You can specify that a relationship is optional, even if you have specified upper and/or lower bounds. This means that there do not have to be any objects at the destination, but if there are then the number of objects must lie within the bounds specified.

It is important to note that simply defining a relationship does not cause a destination object to be created when a new source object is created. In this respect, defining a relationship is akin to declaring an instance variable in a standard Objective-C class. Consider the following example.

```
@interface Widget : NSObject
{
    Sprocket *sprocket;
}
```

If you create an instance of `Widget`, an instance of `Sprocket` is not created unless you write code to cause it to happen (for example, by overriding the `init` method). Similarly, if you define an `Address` entity, and a non-optional to-one relationship from `Employee` to `Address`, then simply creating an instance of `Employee` does not create a new `Address` instance. Likewise, if you define a non-optional to-many relationship from `Employee` to `Address` with a minimum count of 1, then simply creating an instance of `Employee` does not create a new `Address` instance.

Inverse Relationships

Most relationships are inherently bi-directional. If a `Department` has a to-many relationship to the `Employees` that work in a `Department`, there is an inverse relationship from an `Employee` to the `Department`. The major exception is a fetched property, which represents a weak one-way relationship—there is no relationship from the destination to the source (see [“Fetched Properties”](#) (page 89)).

You should typically model relationships in both directions, and specify the inverse relationships appropriately. Core Data uses this information to ensure the consistency of the object graph if a change is made (see [“Manipulating Relationships and Object Graph Integrity”](#) (page 85)). For a discussion of some of the reasons why you might not want to model a relationship in both directions, and some of the problems that might arise if you don’t, see [“Unidirectional Relationships”](#) (page 88).

Relationship Delete Rules

A relationship's delete rule specifies what should happen if an attempt is made to delete the source object. Note the phrasing in the previous sentence—“if an attempt is made...”. If a relationship's delete rule is set to Deny, it is possible that the source object will not be deleted. Consider again a department's employees relationship, and the effect that the different delete rules have.

Deny

If there is at least one object at the relationship destination, then the source object cannot be deleted.

For example, if you want to remove a department, you must ensure that all the employees in that department are first transferred elsewhere (or fired!) otherwise the department cannot be deleted.

Nullify

Set the inverse relationship for objects at the destination to null.

For example, if you delete a department, set the department for all the current members to null. This only makes sense if the department relationship for an employee is optional, or if you ensure that you set a new department for each of the employees before the next save operation.

Cascade

Delete the objects at the destination of the relationship.

For example, if you delete a department, fire all the employees in that department at the same time.

No Action

Do nothing to the object at the destination of the relationship.

For example, if you delete a department, leave all the employees as they are, even if they still believe they belong to that department.

It should be clear that the first three of these rules are useful in different circumstances. For any given relationship it is up to you to choose which is most appropriate, depending on the business logic. It is less obvious why the No Action rule might be of use, since if you use it you have the possibility of leaving the object graph in an inconsistent state (employees having a relationship to a deleted department).

If you use the No Action rule, it is up to you to ensure that the consistency of the object graph is maintained. You are responsible for setting any inverse relationship to a meaningful value. This may be of benefit in a situation where you have a to-many relationship and there may be a large number of objects at the destination.

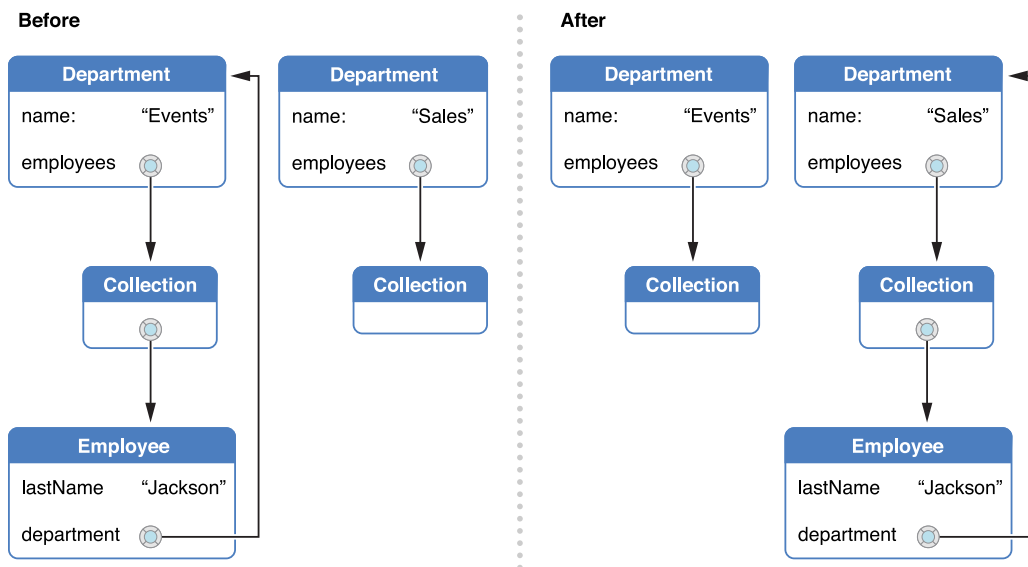
Manipulating Relationships and Object Graph Integrity

In general, programmatically manipulating relationships is straightforward. For examples of how to manipulate relationships programmatically, see [“Accessing and Modifying Properties”](#) (page 67)

Since Core Data takes care of the object graph consistency maintenance for you, you only need to change one end of a relationship and all other aspects are managed for you. This applies to to-one, to-many, and many-to-many relationships. Consider the following examples.

An employee’s relationship to a manager implies a reverse relationship between a manager and the manager’s employees. If a new employee is assigned to a particular manager, it is important that the manager be made aware of this responsibility. The new employee must be added to the manager’s list of reports. Similarly, if an employee is transferred from one department to another, a number of modifications must be made, as illustrated in [Figure 1](#) (page 85). The employee’s new department is set, the employee is removed from the previous department’s list of employees, and the employee is added to the new department’s list of employees.

Figure 1 Transferring an employee to a new department



Without the Core Data framework, you must write several lines of code to ensure that the consistency of the object graph is maintained. Moreover you must be familiar with the implementation of the Department class to know whether or not the inverse relationship should be set (this may change as the application evolves). Using the Core Data framework, all this can be accomplished with a single line of code:

```
anEmployee.department = newDepartment;
```

Alternatively, you can use:

```
[newDepartment addEmployeeObject:anEmployee];
```

(To understand the derivation of the second version, see [“Managed Object Accessor Methods”](#) (page 44).) Both of these have the same net effect: By referencing the managed object model, the framework automatically determines from the current state of the object graph which relationships must be established and which must be broken.

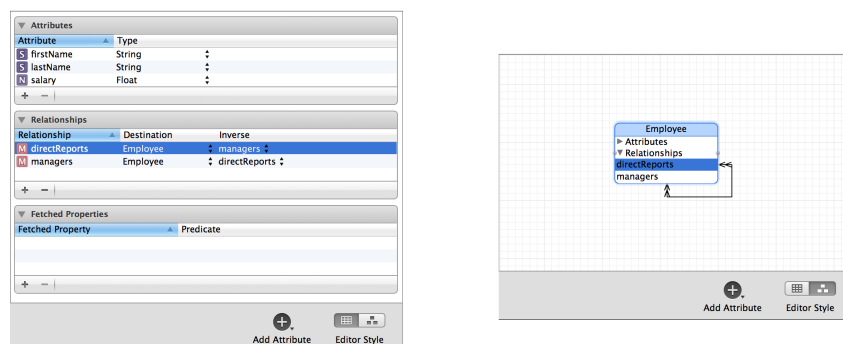
Many-to-Many Relationships

You define a many-to-many relationship using two to-many relationships. The first to-many relationship goes from the first entity to the second entity. The second to-many relationship goes from the second entity to the first entity. You then set each to be the inverse of the other. (If you have a background in database management and this causes you concern, don't worry: if you use a SQLite store, Core Data automatically creates the intermediate join table for you.)

Important: You must define many-to-many relationships in both directions—that is, you must specify two relationships, each being the inverse of the other. You can't just define a to-many relationship in one direction and try to use it as a many-to-many. If you do, you will end up with referential integrity problems.

This works even for relationships back to the same entity (often called “reflexive” relationships). For example, if an employee can have more than one manager (and a manager can have more than one direct report), then you can define a to-many relationship `directReports` from Employee to itself that is the inverse of another to-many relationship, `managers`, again from Employee to itself. This is illustrated in [Figure 2](#) (page 86).

Figure 2 Example of a reflexive many-to-many relationship

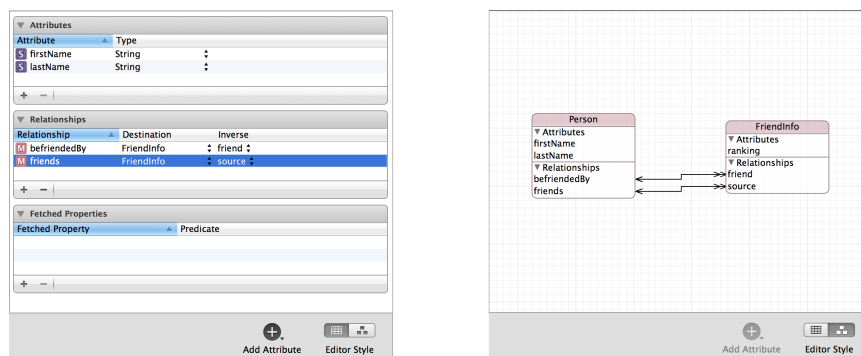


A relationship can also be the inverse of itself. For example, a Person entity may have a `cousins` relationship that is the inverse of itself.

Important: In OS X v10.4, many-to-many relationships do not work with SQLite stores if the relationship is an inverse of itself (such as is the case with cousins).

You should also consider, though, the semantics of the relationship and how it should be modeled. A common example of a relationship that is initially modeled as a many-to-many relationship that's the inverse of itself is "friends." Although it's the case that you are your cousin's cousin whether they like it or not, it's not necessarily the case that you are your friend's friend. For this sort of relationship, you should use an intermediate ("join") entity. An advantage of the intermediate entity is that you can also use it to add more information to the relationship—for example a "FriendInfo" entity might include some indication of the strength of the friendship with a "ranking" attribute. This is illustrated in [Figure 3](#) (page 87)

Figure 3 A model illustrating a "friends" relationship using an intermediate entity



In this example, Person has two to-many relationships to FriendInfo: `friends` represents the source person's friends, and `befriendedBy` represents those who count the source as their friend. FriendInfo represents information about one friendship, "in one direction." A given instance notes who the source is, and one person they consider to be their friend. If the feeling is mutual, then there will be a corresponding instance where `source` and `friend` are swapped. There are several other considerations when dealing with this sort of model:

- To establish a friendship from one person to another, you have to create an instance of FriendInfo. If both people like each other, you have to create two instances of FriendInfo.
- To break a friendship, you must delete the appropriate instance of FriendInfo.
- The delete rule from Person to FriendInfo should be cascade. If a person is removed from the store, then the FriendInfo instance becomes invalid, so must also be removed.

As a corollary, the relationships from FriendInfo to Person must not be optional—an instance of FriendInfo is invalid if the `source` or `friend` is null.

- To find out who one person's friends are, you have to aggregate all the `friend` destinations of the `friends` relationship, for example:

```
NSSet *personsFriends = [aPerson valueForKeyPath:@"friends.friend"];
```

Conversely, to find out who consider a given person to be their friends, you have to aggregate all the source destinations of the `befriendedBy` relationship, for example:

```
NSSet *befriendedByPerson = [aPerson  
valueForKeyPath:@"befriendedBy.source"];
```

Unidirectional Relationships

It is not *strictly* necessary to model a relationship in both directions. In some cases it may be useful not to, for example when a to-many relationship may have a very large number of destination objects and you are rarely likely to traverse the relationship (you may want to ensure that you do not unnecessarily fault in a large number of objects at the destination of a relationship). Not modeling a relationship in both directions, however, imposes on you a great number of responsibilities, to ensure the consistency of the object graph, for change tracking, and for undo management. For this reason, the practice is strongly discouraged. It typically only makes sense to model a to-one relationship in one direction.

If you create a model with unidirectional relationships (relationships where you have specified no inverse), your object graph may end up in an inconsistent state.

The following example illustrates a situation where only modeling a relationship in one directions might cause problems. Consider a model in which you have two entities, `Employee` and `Department`, with a to-one relationship, "department", from `Employee` to `Department`. The relationship is non-optional and has a "deny" delete rule. The relationship does not have an inverse. Now consider the following code sample:

```
Employee *employee;  
Department *department;  
// assume entity instances correctly instantiated  
[employee setDepartment:department];  
[managedObjectContext deleteObject:department];  
BOOL saved = [managedObjectContext save:&error];
```

The save succeeds (despite the fact that the relationship is non-optional) as long as `employee` is not changed in any other way. Because there is no inverse for the `Employee.department` relationship, `employee` is not marked as changed when `department` is deleted (and therefore `employee` is not validated for saving).

If you then add the following line of code:

```
id x = [employee department];
```

x will be a fault to "nowhere" rather than nil.

If, on the other hand, the "department" relationship has an inverse (and the delete rule is not No Action), everything behaves "as expected" since employee is marked as changed during delete propagation.

This illustrates why, in general, you should avoid using unidirectional relationships. Bidirectional relationships provide the framework with additional information with which to better maintain the object graph. If you do want to use unidirectional relationships, you need to do some of this maintenance yourself. In the case above, this would mean that after this line of code:

```
[managedObjectContext deleteObject:department];
```

you should write:

```
[employee setValue:nil forKey:@"department"]
```

The subsequent save will now (correctly) fail because of the non-optional rule for the relationship.

Cross-Store Relationships

You must be careful not to create relationships from instances in one persistent store to instances in another persistent store, as this is not supported by Core Data. If you need to create a relationship between entities in different stores, you typically use fetched properties (see "[Fetched Properties](#)" (page 89)).

Fetched Properties

Fetched properties represent weak, one-way relationships. In the employees and departments domain, a fetched property of a department might be "recent hires" (employees do not have an inverse to the recent hires relationship). In general, fetched properties are best suited to modeling cross-store relationships, "loosely coupled" relationships, and similar transient groupings.

A fetched property is like a relationship, but it differs in several important ways:

- Rather than being a "direct" relationship, a fetched property's value is calculated using a fetch request. (The fetch request typically uses a predicate to constrain the result.)
- A fetched property is represented by an array, not a set. The fetch request associated with the property can have a sort ordering, and thus the fetched property may be ordered.
- A fetched property is evaluated lazily, and is subsequently cached.

In some respects you can think of a fetched property as being similar to a smart playlist, but with the important constraint that it is not dynamic. If objects in the destination entity are changed, you must reevaluate the fetched property to ensure it is up-to-date. You use `refreshObject:mergeChanges:` to manually refresh the properties—this causes the fetch request associated with this property to be executed again when the object fault is next fired.

There are two special variables you can use in the predicate of a fetched property—`$FETCH_SOURCE` and `$FETCHED_PROPERTY`. The source refers to the specific managed object that has this property, and you can create key-paths that originate with this, for example `university.name LIKE [c] $FETCH_SOURCE.searchTerm`. The `$FETCHED_PROPERTY` is the entity's fetched property description. The property description has a `userInfo` dictionary that you can populate with whatever key-value pairs you want. You can therefore change some expressions within a fetched property's predicate or (via key-paths) any object to which that object is related.

To understand how the variables work, consider a fetched property with a destination entity `Author` and a predicate of the form, `(university.name LIKE [c] $FETCH_SOURCE.searchTerm) AND (favoriteColor LIKE [c] $FETCHED_PROPERTY.userInfo.color)`. If the source object had an attribute `searchTerm` equal to "Cambridge", and the fetched property had a user info dictionary with a key "color" and value "Green", then the resulting predicate would be `(university.name LIKE [c] "Cambridge") AND (favoriteColor LIKE [c] "Green")`. This would match any Authors at Cambridge whose favorite color is green. If you changed the value of `searchTerm` in the source object to, say, "Durham", then the predicate would be `(university.name LIKE [c] "Durham") AND (favoriteColor LIKE [c] "Green")`.

The most significant constraint is that you cannot use substitutions to change the structure of the predicate—for example you cannot change a `LIKE` predicate to a compound predicate, nor can you change the operator (in this example, `LIKE [c]`). Moreover, in OS X version 10.4, this only works with the XML and Binary stores as the SQLite store will not generate the appropriate SQL.

Non-Standard Persistent Attributes

Core Data supports a range of common types for values of persistent attributes, including string, date, and number. Sometimes, however, you want an attribute's value to be a type that is not supported directly. For example, in a graphics application you might want to define a `Rectangle` entity that has attributes `color` and `bounds` that are an instance of `NSColor` and an `NSRect` struct respectively. This article describes the two ways in which you can use non-standard attribute types: using transformable attributes, or by using a transient property to represent the non-standard attribute backed by a supported persistent property.

Introduction

Persistent attributes must be of a type recognized by the Core Data framework so that they can be properly stored to and retrieved from a persistent store. Core Data provides support for a range of common types for persistent attribute values, including string, date, and number (see `NSAttributeDescription` for full details). Sometimes, however, you want to use types that are not supported directly, such as colors and C structures.

You can use non-standard types for persistent attributes either by using transformable attributes or by using a *transient* property to represent the non-standard attribute backed by a supported persistent property. The principle behind the two approaches is the same: you present to consumers of your entity an attribute of the type you want, and “behind the scenes” it’s converted into a type that Core Data can manage. The difference between the approaches is that with transformable attributes you specify just one attribute and the conversion is handled automatically. In contrast, with transient properties you specify two attributes and you have to write code to perform the conversion.

Transformable Attributes

The idea behind transformable attributes is that you access an attribute as a non-standard type, but behind the scenes Core Data uses an instance of `NSValueTransformer` to convert the attribute to and from an instance of `NSData`. Core Data then stores the data instance to the persistent store.

By default, Core Data uses the `NSKeyedUnarchiveFromDataTransformerName` transformer, however you can specify your own transformer if you want. If you specify a custom transformer, it must transform an instance of the non-standard data type into an instance of `NSData` and support reverse transformation. You should not specify a name if you are using the default transformer.

Important: Although the default transformer is the transformer specified by `NSKeyedUnarchiveFromDataTransformerName`, this transformer is actually used in reverse. If you specify the default transformer explicitly, Core Data would use it “in the wrong direction.”

You specify that an attribute is transformable and the name of the transformer to use in the model editor in Xcode or programmatically:

- If you are using the model editor in Xcode, select Transformable in the attribute’s Type popup and type the name in the Value Transformer Name text field.
- If you are setting the type programmatically, use `setAttributeType:` and pass `NSTransformableAttributeType` as the parameter, then (if appropriate) use `setValueTransformerName:` to specify the name of the transformer.

In principle, you don’t have to do anything else. In practice, to suppress compiler warnings you should declare a property for the attribute as shown in the following example (notice `favoriteColor`):

```
@interface Person : NSManagedObject

@property (nonatomic) NSString *firstName;
@property (nonatomic) NSString *lastName;

@property (nonatomic) NSColor *favoriteColor;

@end
```

To suppress compiler warnings, you can also add an implementation directive:

```
@implementation Person

@dynamic firstName;
@dynamic lastName;
@dynamic favoriteColor;

@end
```

You can now use the attribute as you would any other standard attribute, as illustrated in the following code fragment:

```
Employee *newEmployee =  
    [NSEntityDescription insertNewObjectForEntityForName:@"Employee"  
        inManagedObjectContext:myManagedObjectContext];  
  
newEmployee.firstName = @"Captain";  
newEmployee.lastName = @"Scarlet";  
newEmployee.favoriteColor = [NSColor redColor];
```

Custom Code

The following sections illustrate implementations for object and scalar values. Both start, however, with a common task—you must specify a persistent attribute.

Note: The example for an object value uses an instance of `NSColor`; you should typically use a transformable attribute instead.

Basic Approach

To use non-supported types, in the managed object model you define two attributes. One is the attribute you actually want (its value is for example a color object or a rectangle struct). This attribute is transient. The other is a “shadow” representation of that attribute. This attribute is persistent.

You specify the type of the transient attribute as undefined (`NSUndefinedAttributeType`). Since Core Data does not need to store and retrieve transient properties, you can use any object type you want for the attribute in your implementation. Core Data does, though, track the state of transient properties so that they can participate in the object graph management (for example, for undo and redo).

The type of the shadow attribute must be one of the “concrete” supported types. You then implement a custom managed object class with suitable accessor methods for the transient attribute that retrieve the value from and store the value to the persistent attribute.

The basic approach for object and scalar values is the same—you must find a way to represent the unsupported data type as one of the supported data types—however there is a further constraint in the case of scalar values.

Scalar Value Constraints

A requirement of the accessor methods you write is that they must be key-value coding (and key-value observing) compliant.

If you want to use a scalar type or structure that is not one of those supported directly by Core Data and not one of the structures supported by key-value coding, you must store it in your managed object as an object—typically an `NSValue` instance, although you can also define your own custom class. You will then treat it as an object value as described later in this article. It is up to users of the object to extract the required structure from the `NSValue` (or custom) object when retrieving the value, and to transform a structure into an `NSValue` (or custom) object when setting the value.

The Persistent Attribute

For any non-standard attribute type you want to use, you must choose a supported attribute type that you will use to store the value. Which supported type you choose depends on the non-standard type and what means there are of transforming it into a supported type. In many cases you can easily transform a non-supported object into an `NSData` object using an archiver. For example, you can archive a color object as shown in the following code sample. The same technique can be used if you represent the attribute as an instance of `NSValue` or of a custom class (your custom class would, of course, need to adopt the `NSCoding` protocol or provide some other means of being transformed into a supported data type).

```
NSData *colorAsData = [NSKeyedArchiver archivedDataWithRootObject:aColor];
```

You are free to use whatever means you wish to effect the transformation. For example, you could transform an `NSRect` structure into a string object (strings can of course be used in a persistent store).

```
NSRect aRect; // Instance variable.  
NSString *rectAsString = NSStringFromRect(aRect);
```

You can transform the string back into a rectangle using `NSRectFromString`. You should bear in mind, however, that since the transformation process may happen frequently, you should ensure that it is as efficient as possible.

Typically you do not need to implement custom accessor methods for the persistent attribute. It is an implementation detail, the value should not be accessed other than by the entity itself. If you do modify this value directly, it is possible that the entity object will get into an inconsistent state.

An Object Attribute

If the non-supported attribute is an object, then in the managed object model you specify its type as undefined, and that it is transient. When you implement the entity's custom class, there is no need to add an instance variable for the attribute—you can use the managed object's private internal store. A point to note about the

implementations described below is that they cache the transient value. This makes accessing the value more efficient—it is also necessary for change management. If you define custom instance variables, you should clean up these variables in `didTurnIntoFault` rather than `dealloc`.

There are two strategies both for getting and for setting the transient value. You can retrieve the transient value either "lazily" (on demand—described in [“The On-demand Get Accessor”](#) (page 95)) or during `awakeFromFetch` (described in [“The Pre-calculated Get”](#) (page 95)). It may be preferable to retrieve it lazily if the value may be large (if for example it is a bitmap). For the persistent value, you can either update it every time the transient value is changed (described in [“The Immediate-Update Set Accessor”](#) (page 96)), or you can defer the update until the object is saved (described in [“The Delayed-Update Set Accessor”](#) (page 97)).

The On-demand Get Accessor

In the get accessor, you retrieve the attribute value from the managed object's private internal store. If the value is `nil`, then it is possible it has not yet been cached, so you retrieve the corresponding persistent value, then if that value is not `nil`, transform it into the appropriate type and cache it. (You don't need to invoke the key-value observing change notification methods for the set method because this doesn't represent a change in the value.) The following example illustrates the on-demand get accessor for a color attribute.

```
- (NSColor *)color {

    [self willAccessValueForKey:@"color"];
    NSColor *color = [self primitiveColor];
    [self didAccessValueForKey:@"color"];
    if (color == nil) {
        NSData *colorData = [self colorData];
        if (colorData != nil) {
            color = [NSKeyedUnarchiver unarchiveObjectWithData:colorData];
            [self setPrimitiveColor:color];
        }
    }
    return color;
}
```

The Pre-calculated Get

Using this approach, you retrieve and cache the persistent value in `awakeFromFetch`. (You don't need to invoke the key-value observing change notification methods for the set method because this doesn't represent a change in the value.)

```
- (void)awakeFromFetch {  
  
    [super awakeFromFetch];  
    NSData *colorData = [self colorData];  
    if (colorData != nil) {  
        NSColor *color = [NSKeyedUnarchiver unarchiveObjectWithData:colorData];  
        [self setPrimitiveColor:color];  
    }  
}
```

In the get accessor you then simply return the cached value.

```
- (NSColor *)color {  
  
    [self willAccessValueForKey:@"color"];  
    NSColor *color = [self primitiveColor];  
    [self didAccessValueForKey:@"color"];  
    return color;  
}
```

This technique is useful if you are likely to access the attribute frequently—you avoid the conditional statement in the get accessor.

The Immediate-Update Set Accessor

In this set accessor, you set the value for both the transient and the persistent attributes at the same time. You transform the unsupported type into the supported type to set as the persistent value. You must ensure that you invoke the key-value observing change notification methods, so that objects observing the managed object—including the managed object context—are notified of the modification. The following example illustrates the set accessor for a color attribute.

```
- (void)setColor:(NSColor *)aColor {  
  
    [self willChangeValueForKey:@"color"];  
    [self setPrimitiveValue:aColor forKey:@"color"];  
    [self didChangeValueForKey:@"color"];
```



```
[self setValue:[NSKeyedArchiver archivedDataWithRootObject:aColor]
              forKey:@"colorData"];
}
```

The main disadvantage with this approach is that the persistent value is recalculated each time the transient value is updated, which may be a performance issue.

The Delayed-Update Set Accessor

In this technique, in the set accessor you only set the value for the transient attribute. You implement a `willSave` method that updates the persistent value just before the object is saved. (You don't need to invoke the key-value observing change notification methods around the set method because this doesn't represent a change in the value.)

```
- (void)setColor:(NSColor *)aColor {

    [self willChangeValueForKey:@"color"];
    [self setPrimitiveValue:aColor forKey:@"color"];
    [self didChangeValueForKey:@"color"];
}

- (void)willSave {

    NSColor *color = [self primitiveValueForKey:@"color"];
    if (color != nil) {
        [self setPrimitiveValue:[NSKeyedArchiver archivedDataWithRootObject:color]
              forKey:@"colorData"];
    }
    else {
        [self setPrimitiveValue:nil forKey:@"colorData"];
    }
    [super willSave];
}
```

If you adopt this approach, you must take care when specifying your optionality rules. If `color` is a required attribute, then (unless you take other steps) you must specify the `color` attribute as not optional, and the `color data` attribute as optional. If you do not, then the first save operation may generate a validation error.

When the object is first created, the value of `colorData` is `nil`. When you update the color attribute, the `colorData` attribute is unaffected (that is, it remains `nil`). When you save, `validateForUpdate:` is invoked before `willSave`. In the validation stage, the value of `colorData` is still `nil`, and therefore validation fails.

Scalar Values

You can declare properties as scalar values, but for scalar values Core Data cannot dynamically generate accessor methods—you must provide your own implementations (see [“Managed Object Accessor Methods”](#) (page 44)). Core Data automatically synthesizes the primitive accessor methods (`primitiveLength` and `setPrimitiveLength:`), but you need to declare them to suppress compiler warnings.

For objects that will be used in either a Foundation collection or an AppKit view, you should typically allow Core Data to use its default storage instead of creating scalar instances to hold property values:

- There is CPU and memory overhead in creating and destroying `NSNumber` object wrappers for your scalars;
- Core Data optimizes at runtime any accessor methods you do not override—for example, it inlines the access and change notification method calls.

The advantages of allowing Core Data to manage its own storage usually outweigh any advantages of interacting directly with scalar values, although if you suspect that this is not true for your application you should use performance analysis tools to check.

You can declare properties as scalar values. Core Data cannot, though, dynamically generate accessor methods for scalar values—you must provide your own implementations. If you have an attribute `length` that is specified in the model as a `double` (`NSDoubleAttributeType`), in the interface file you declare `length` as:

```
@property double length;
```

In the implementation file, you implement accessors that invoke the relevant access and change notification methods, and the primitive accessors. Core Data automatically synthesizes the primitive accessor methods (`primitiveLength` and `setPrimitiveLength:`), but you need to declare them to suppress compiler warnings (you can declare them using a property).

```
@interface MyManagedObject (PrimitiveAccessors)
@property (nonatomic) NSNumber *primitiveLength;
@end

- (double)length {
```

```
[self willAccessValueForKey:@"length"];
NSNumber *tmpValue = [self primitiveLength];
[self didAccessValueForKey:@"length"];
return (tmpValue!=nil) ? [tmpValue doubleValue] : 0.0; // Or a suitable
representation for nil.
}

- (void)setLength:(double)value {

    NSNumber *temp = @(value);
    [self willChangeValueForKey:@"length"];
    [self setPrimitiveLength:temp];
    [self didChangeValueForKey:@"length"];
}
```

A Non-Object Attribute

If the non-supported attribute is supported by key-value coding, then in the managed object model you again specify its type as undefined, and that it is transient. When you implement the entity's custom class, you typically add an instance variable for the attribute. For example, given an attribute called `bounds` that you want to represent using an `NSRect` structure, your class interface might be like that shown in the following example.

```
@interface MyManagedObject : NSManagedObject
{
    NSRect bounds;
}
@property (nonatomic) NSRect bounds;
@end
```

If you use an instance variable to hold an attribute, you must also implement primitive get and set accessors (see [“Custom Primitive Accessor Methods”](#) (page 54)), as shown in the following example.

```
@interface MyManagedObject : NSManagedObject
{
    NSRect myBounds;
```

```
}  
@property (nonatomic, assign) NSRect bounds;  
@property (nonatomic, assign) NSRect primitiveBounds;  
@end
```

The primitive methods simply get and set the instance variable—they do not invoke key-value observing change or access notification methods—as shown in the following example.

```
- (NSRect)primitiveBounds  
{  
    return myBounds;  
}  
- (void)setPrimitiveBounds:(NSRect)aRect  
    myBounds = aRect;  
}
```

Whichever strategy you adopt, you then implement accessor methods mostly as described for the object value. For the get accessor you can adopt either the lazy or pre-calculated technique, and for the set accessor you can adopt either the immediate update or delayed update technique. The following sections illustrate only the former versions of each.

The Get Accessor

In the get accessor, you retrieve the attribute value from the managed object's private internal store. If the value has not yet been set, then it is possible it has not yet been cached, so you retrieve the corresponding persistent value, then if that value is not `nil`, transform it into the appropriate type and cache it. The following example illustrates the get accessor for a rectangle (this example makes a simplifying assumption that the bounds width cannot be 0, so if the value is 0 then the bounds has not yet been unarchived).

```
- (NSRect)bounds  
{  
    [self willAccessValueForKey:@"bounds"];  
    NSRect aRect = bounds;  
    [self didAccessValueForKey:@"bounds"];  
    if (aRect.size.width == 0)  
    {
```

```
NSString *boundsAsString = [self boundsAsString];
if (boundsAsString != nil)
{
    bounds = NSRectFromString(boundsAsString);
}
return bounds;
}
```

The Set Accessor

In the set accessor, you must set the value for both the transient and the persistent attributes. You transform the unsupported type into the supported type to set as the persistent value. You must ensure that you invoke the key-value observing change notification methods, so that objects observing the managed object—including the managed object context—are notified of the modification. The following example illustrates the set accessor for a rectangle.

```
- (void)setBounds:(NSRect)aRect
{
    [self willChangeValueForKey:@"bounds"];
    bounds = aRect;
    [self didChangeValueForKey:@"bounds"];
    NSString *rectAsString = NSStringFromRect(aRect);
    [self setValue:rectAsString forKey:@"boundsAsString"]; }
}
```

Type-Checking

If you define an attribute to use a non-standard type, you can also specify the name of the class used to represent the value, using `setAttributeValueClassName:`.

You can only set the value class name in code. The following example shows how you can modify the managed object model to include a value class name for a non-standard attribute (`favoriteColor`) represented in this case by an instance of a custom class, `MyColor`.

```
myManagedObjectModel = <#Get a managed object context#>;

NSEntityDescription *employeeEntity =
```

```
[[myManagedObjectModel entitiesByName] objectForKey:@"Employee"];
NSAttributeDescription *favoriteColorAttribute =
    [[employeeEntity attributesByName] objectForKey:@"favoriteColor"];

// Set the attribute value class to MyColor
[favoriteColorAttribute setAttributeValueClassName:@"MyColor"];
```

The attribute value class must actually exist at runtime. If you misspell the class name itself (for example, `MyColour` instead of `MyColor`), the check succeeds silently.

Core Data checks the class of any value set as the attribute value and throws an exception if it is an instance of the wrong class:

```
Employee *newEmployee =
    [NSEntityDescription insertNewObjectForEntityForName:@"Employee"
    inManagedObjectContext:aManagedObjectContext];
newEmployee.favoriteColor = [NSColor redColor]; // Exception thrown here.
```

Managed Object Validation

There are two types of validation—property-level and inter-property. You use property-level validation to ensure the correctness of individual values; you use inter-property validation to ensure the correctness of combinations of values.

Core Data Validation

Cocoa provides a basic infrastructure for model value validation. It requires you, though, to write code for all the constraints you want to apply. Core Data allows you to put validation logic into the managed object model. You can specify maximum and minimum values for numeric and date attributes; maximum and minimum lengths for string attributes, and a regular expression that a string attribute must match. You can also specify constraints on relationships, for example that they are mandatory or cannot exceed a certain number. You can therefore specify most common constraints on attribute values without writing any code.

If you do want to customize validation of individual properties, you use standard validation methods as defined by the `NSKeyValueCoding` protocol and described in [“Property-Level Validation”](#) (page 104)). Core Data also extends validation to validation of relationships and inter-property values. These are described in [“Inter-Property validation”](#) (page 105).

It is important to understand that *how* to validate is a model decision, *when* to validate is a user interface or controller-level decision (for example, a value binding for a text field might have its “validates immediately” option enabled). Moreover, at various times, inconsistencies are expected to arise in managed objects and object graphs.

There is nothing to disallow an in-memory object from becoming inconsistent on a temporary basis. The validation constraints are applied by Core Data only during a “save” operation or upon request (you can invoke the validation methods directly as and when you wish). Sometimes it may be useful to validate changes as soon as they are made and to report errors immediately. This can prevent the user being presented with a long list of errors when they finally come to save their work. If managed objects were required to be always in a valid state, it would amongst other things force a particular workflow on the end-user. This also underpins the idea of a managed object context representing a “scratch pad”—in general you can bring managed objects onto the scratch pad and edit them however you wish before ultimately either committing the changes or discarding them.

Property-Level Validation

The `NSKeyValueCoding` protocol specifies a method—`validateValue:forKey:error:`—that provides general support for validation methods in a similar way to that in which `valueForKey:` provides support for accessor methods.

If you want to implement logic in addition to the constraints you provide in the managed object model, you should not override `validateValue:forKey:error:`. Instead you should implement methods of the form `validate<Key>:error:`.

Important: If you do implement custom validation methods, you should typically not invoke them directly. Instead you should call `validateValue:forKey:error:` with the appropriate key. This ensures that any constraints defined in the managed object model are also applied.

In the method implementation, you check the proposed new value and if it does not fit your constraints you return `NO`. If the error parameter is not `NULL`, you also create an `NSError` object that describes the problem, as illustrated in this example.

```
-(BOOL)validateAge:(id *)ioValue error:(NSError **)outError {

    if (*ioValue == nil) {
        // trap this in setNilValueForKey? new NSNumber with value 0?
        return YES;
    }
    if ([*ioValue floatValue] <= 0.0) {
        if (outError != NULL) {
            NSString *errorStr = NSLocalizedStringFromTable(
                @"Age must greater than zero", @"Employee",
                @"validation: zero age error");
            NSDictionary *userInfoDict = @{ NSLocalizedDescriptionKey : errorStr
};
            NSError *error = [[NSError alloc] initWithDomain:EMPLOYEE_ERROR_DOMAIN
                code:PERSON_INVALID_AGE_CODE
                userInfo:userInfoDict];
            *outError = error;
        }
        return NO;
    }
}
```



```
else {  
    return YES;  
}  
// . . .
```

The input value is a pointer to object reference (an `id *`). This means that in principle you can change the input value. Doing so is, however, strongly discouraged, as there are potentially serious issues with memory management (see “Key-Value Validation” in *Key-Value Coding Programming Guide*). Moreover, you should not call `validateValue:forKey:error:` within custom property validation methods. If you do, you will create an infinite loop when `validateValue:forKey:error:` is invoked at runtime.

If you change the input value in a `validate<Key>:error:` method, you must ensure that you only change the value if it is invalid or uncoerced. The reason is that, since the object and context are now dirtied, Core Data may validate that key again later. If you keep performing a coercion in a validation method, this can therefore produce an infinite loop. Similarly, you should also be careful if you implement validation and `willSave` methods that produce mutations or side effects—Core Data will revalidate those changes until a stable state is reached.

Inter-Property validation

It is possible for the values of all the individual attributes of an object to be valid and yet for the combination of values to be invalid. Consider, for example, an application that stores information about people including their age and whether or not they have a driving license. For a `Person` object, 12 might be a valid value for an `age` attribute, and `YES` is a valid value for a `hasDrivingLicense` attribute, but (in most countries at least) this combination of values would be invalid.

`NSManagedObject` provides additional opportunities for validation—update, insertion, and deletion—through the `validateFor...` methods such as `validateForUpdate:`. If you implement custom inter-property validation methods, you call the superclass's implementation first to ensure that individual property validation methods are also invoked. If the superclass's implementation fails (that is, if there is an invalid attribute value), then you can:

1. Return `NO` and the error created by the superclass's implementation.
2. Continue to perform validation, looking for inconsistent combinations of values.

If you continue, you must make sure that any values you use in your logic are not themselves invalid in such a way that your code might itself cause errors (for example, if there is an attribute whose value is required to be greater than 0, which is actually 0 so fails validation but which you use as a divisor in a computation). Moreover, if you discover further validation errors, you must combine them with the existing error and return a “multiple errors error” as described in [“Combining Validation Errors”](#) (page 108).

The following example shows the implementation of an inter-property validation method for a Person entity that has two attributes, `birthday` and `hasDrivingLicense`. The constraint is that a person aged less than 16 years cannot have a driving license. This constraint is checked in both `validateForInsert:` and `validateForUpdate:`, so the validation logic itself is factored into a separate method.

Listing 1 Inter-property validation for a Person entity

```
- (BOOL)validateForInsert:(NSError **)error
{
    BOOL propertiesValid = [super validateForInsert:error];
    // could stop here if invalid
    BOOL consistencyValid = [self validateConsistency:error];
    return (propertiesValid && consistencyValid);
}

- (BOOL)validateForUpdate:(NSError **)error
{
    BOOL propertiesValid = [super validateForUpdate:error];
    // could stop here if invalid
    BOOL consistencyValid = [self validateConsistency:error];
    return (propertiesValid && consistencyValid);
}

- (BOOL)validateConsistency:(NSError **)error
{
    static    NSCalendar *gregorianCalendar;

    BOOL valid = YES;
    NSDate *myBirthday = [self birthday];
```

```
if ((myBirthday != nil) && ([[self hasDrivingLicense] boolValue] == YES)) {

    if (gregorianCalendar == nil) {
        gregorianCalendar = [[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
    }
    NSDateComponents *components = [gregorianCalendar
components:NSYearCalendarUnit

                                fromDate:myBirthday
                                toDate:[NSDate date]
                                options:0];

    int years = [components year];

    if (years < 16) {

        valid = NO;

        // don't create an error if none was requested
        if (error != NULL) {

            NSBundle *myBundle = [NSBundle bundleForClass:[self class]];
            NSString *drivingAgeErrorString = [myBundle
localizedStringForKey:@"TooYoungToDriveError"
                    value:@"Person is too young to have a driving
license."
                    table:@"PersonErrorStrings"];

            NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
            [userInfo setObject:drivingAgeErrorString
forKey:NSLocalizedFailureReasonErrorKey];
            [userInfo setObject:self forKey:NSValidationErrorKey];

            NSError *drivingAgeError = [NSError errorWithDomain:PERSON_DOMAIN
code:NSManagedObjectValidationError

                                userInfo:userInfo];
```

```
        // if there was no previous error, return the new error
        if (*error == nil) {
            *error = drivingAgeError;
        }
        // if there was a previous error, combine it with the existing one
        else {
            *error = [self errorFromOriginalError:*error
error:drivingAgeError];
        }
    }
}
return valid;
}
```

Combining Validation Errors

If there are multiple validation failures in a single operation, you create and return a "multiple errors error"—that is, an `NSError` object with the code `NSValidationMultipleErrorsError`. You add individual errors to an array and add the array—using the key `NSDetailedErrorsKey`—to the user info dictionary in the `NSError` object. This pattern also applies to errors returned by the superclass's validation method. Depending on how many tests you perform, it may be convenient to define a method that combines an existing `NSError` object (which may itself be a multiple errors error) with a new one and returns a new multiple errors error.

The following example shows the implementation of a simple method to combine two errors into a single multiple errors error. How the combination is made depends on whether or not the original error was itself a multiple errors error.

Listing 2 A method for combining two errors into a single multiple errors error

```
- (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];
}
```

Faulting and Uniquing

Faulting is a mechanism Core Data employs to reduce your application's memory usage. A related feature called uniquing ensures that, in a given managed object context, you never have more than one managed object to represent a given record.

Faulting Limits the Size of the Object Graph

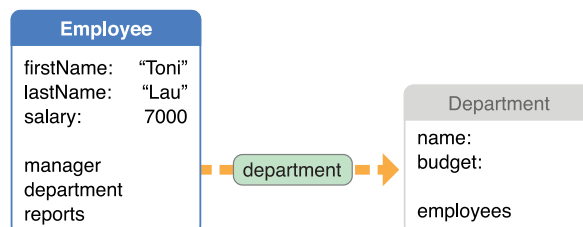
Faulting reduces the amount of memory your application consumes. A fault is a placeholder object that represents a managed object that has not yet been fully realized, or a collection object that represents a relationship:

- A managed object fault is an instance of the appropriate class, but its persistent variables are not yet initialized.
- A relationship fault is a subclass of the collection class that represents the relationship.

Faulting allows Core Data to put boundaries on the object graph. Because a fault is not realized, a managed object fault consumes less memory, and managed objects related to a fault are not required to be represented in memory at all.

To illustrate, consider an application that allows a user to fetch and edit details about a single employee. The employee has a relationship to a manager and to a department, and these objects in turn have other relationships. If you retrieve just a single Employee object from a persistent store, its manager, department, and reports relationships are initially represented by faults. Figure 1 shows an employee's department relationship represented by a fault.

Figure 1 A department represented by a fault



Although the fault is an instance of the `Department` class, it has not yet been realized—none of its persistent instance variables have yet been set. This means that not only does the department object consume less memory itself, but there's no need to populate its employees relationship. If it were a requirement that the object graph be complete, then to edit a single attribute of a single employee, it would ultimately be necessary to create objects to represent the whole corporate structure.

Fault handling is transparent—you do not have to execute a fetch to realize a fault. If at some stage a persistent property of a fault object is accessed, then Core Data automatically retrieves the data for the object and initializes the object (see *NSManagedObject Class Reference* for a list of methods that do not cause faults to fire). This process is commonly referred to as firing the fault. If you send the `Department` object a message to get, say, its name, then the fault fires—and in this situation Core Data executes a fetch for you to retrieve all the object's attributes.

Firing Faults

Core Data automatically fires faults when necessary (when a persistent property of a fault is accessed). However, firing faults individually can be inefficient, and there are better strategies for getting data from the persistent store (see [“Batch Faulting and Pre-fetching with the SQLite Store”](#) (page 142)). For more about how to efficiently deal with faults and relationships, see [“Fetching Managed Objects”](#) (page 141).

When a fault is fired, Core Data does not go back to the store *if* the data is available in its cache. With a cache hit, converting a fault into a realized managed object is very fast—it is basically the same as normal instantiation of a managed object. If the data is not available in the cache, Core Data automatically executes a fetch for the fault object; this results in a round trip to the persistent store to fetch the data, and again the data is cached in memory.

The corollary of this point is that whether an object is a fault is not the same as whether its data has been retrieved from the store. Whether or not an object is a fault simply means whether or not a given managed object has all its attributes populated and is ready to use. If you need to determine whether an object is a fault, you can send it an `isFault` message without firing the fault. If `isFault` returns `NO`, then the data must be in memory. However, if `isFault` returns `YES`, it does *not* imply that the data is *not* in memory. The data may be in memory, or it may not, depending on many factors influencing caching.

Turning Objects into Faults

Turning a realized object into a fault can be useful in pruning the object graph (see [“Reducing Memory Overhead”](#) (page 145)), as well as ensuring property values are current (see [“Ensuring Data Is Up-to-Date”](#) (page 76)). Turning a managed object into a fault releases unnecessary memory, sets its in-memory property values to `nil`, and breaks strong references to related objects.

You can turn a realized object into a fault with the `refreshObject:mergeChanges:` method. If you pass `NO` as the `mergeChanges` argument, you must be sure that there are no changes to that object's relationships. If there are, and you then save the context, you will introduce referential integrity problems to the persistent store.

When an object turns into a fault, it is sent a `didTurnIntoFault` message. You may implement a custom `didTurnIntoFault` method to perform various “housekeeping” functions (see, for example, [“Ensuring Data Is Up-to-Date”](#) (page 76)).

Note: Core Data avoids the term *unfaulting* because it is confusing. There's no “unfaulting” a virtual memory page fault. Page faults are triggered, caused, fired, or encountered. Of course, you can release memory back to the kernel in a variety of ways (using the functions `vm_deallocate`, `munmap`, or `sbrk`). Core Data describes this as “turning an object into a fault.”

Faults and KVO Notifications

When Core Data turns an object into a fault, key-value observing (KVO) change notifications (see *Key-Value Observing Programming Guide*) are sent for the object's properties. If you are observing properties of an object that is turned into a fault and the fault is subsequently realized, you receive change notifications for properties whose values have not in fact changed.

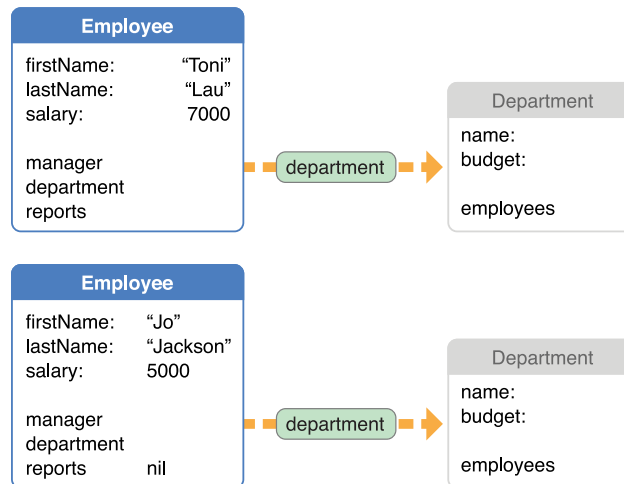
Although the values are not changing semantically from your perspective, the literal bytes in memory are changing as the object is materialized. The key-value observing mechanism requires Core Data to issue the notification whenever the values change as considered from the perspective of pointer comparison. KVO needs these notifications to track changes across key paths and dependent objects.

Uniquing Ensures a Single Managed Object per Record per Context

Core Data ensures that—in a given managed object context—an entry in a persistent store is associated with only one managed object. The technique is known as **uniquing**. Without uniquing, you might end up with a context maintaining more than one object to represent a given record.

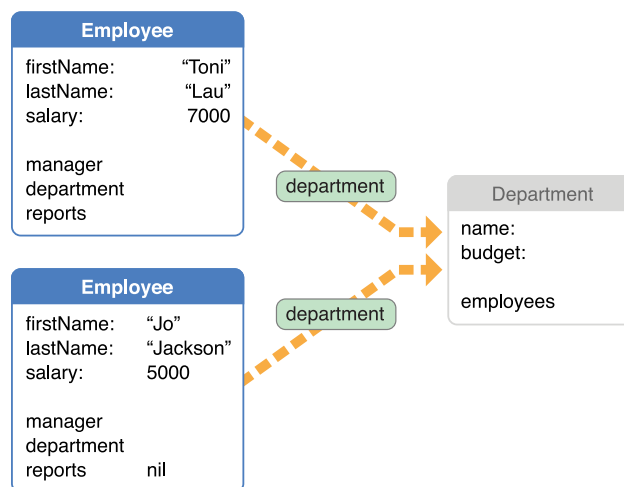
For example, consider the situation illustrated in Figure 2; two employees have been fetched into a *single managed object context*. Each has a relationship to a department, but the department is currently represented by a fault.

Figure 2 Independent faults for a department object



It would appear that each employee has a separate department, and if you asked each employee for their department in turn—turning the faults into regular objects—you would have two separate Department objects in memory. However, if both employees belong to the same department (for example, "Marketing"), then Core Data ensures that (in a given managed object context) only one object representing the Marketing department is ever created. If both employees belong to the same department, their department relationships would both therefore reference the same fault, as illustrated in Figure 3.

Figure 3 Uniqued fault for two employees working in the same department



If Core Data did not use uniquing, then if you fetched all the employees and asked each in turn for their department—thereby firing the corresponding faults—a new Department object would be created every time. This would result in a number of objects, each representing the same department, that could contain different and conflicting data. When the context was saved, it would be impossible to determine which is the correct data to commit to the store.

More generally, all the managed objects in a given context that refer to the Marketing Department object refer to the same instance—they have a single view of Marketing’s data—even if it is a fault.

Note: This discussion has focused on a single managed object context. Each managed object context represents a different view of the data. If the same employees are fetched into a second context, then they—and the corresponding Department object—are all represented by different objects in memory. The objects in different contexts *may* have different and conflicting data. It is precisely the role of the Core Data architecture to detect and resolve these conflicts at save time.

Using Persistent Stores

This article describes how you create a persistent store, and how you can migrate a store from one type to another, and manage store metadata. For more about persistent store types, the differences between them, and how you can configure aspects of their behavior, see [“Persistent Store Features”](#) (page 130).

Creating and Accessing a Store

Access to stores is mediated by an instance of `NSPersistentStoreCoordinator`. You should not need to directly access a file containing a store. From a persistent store coordinator, you can retrieve an object that represents a particular store on disk. Core Data provides an `NSPersistentStore` class to represent persistent stores.

To create a store, you use a persistent store coordinator. You must specify the type of the store to be created, optionally a configuration of managed object model associated with the coordinator, and its location if it is not an in-memory store. The following code fragment illustrates how you can create a read-only XML store:

```
NSManagedObjectContext *moc = <#Get a context#>;
NSPersistentStoreCoordinator *psc = [moc persistentStoreCoordinator];
NSError *error = nil;
NSDictionary *options =
    [NSDictionary dictionaryWithObject:[NSNumber numberWithInt:1]
                                forKey:NSReadOnlyPersistentStoreOption];

NSPersistentStore *roStore =
    [psc addPersistentStoreWithType:NSXMLStoreType
                        configuration:nil URL:url
                        options:options error:&error];
```

To retrieve a store object from a coordinator, you use the method `persistentStoreForURL:`. You can use a store to restrict a fetch request to a specific store, as shown in the following code fragment:

```
NSPersistentStoreCoordinator *psc = <#Get a coordinator#>;
```

```
NSURL *myURL = <#A URL identifying a store#>;
NSPersistentStore *myStore = [psc persistentStoreForURL:myURL];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setAffectedStores:[NSArray arrayWithObject:myStore]];
```

Changing a Store's Type and Location

You can migrate a store from one type or location to another (for example, for a Save As operation) using the `NSPersistentStoreCoordinator` method

`migratePersistentStore:toURL:options:withType:error:.` After invocation of this method, the original store is removed from the coordinator, thus store is therefore no longer a useful reference. The method is illustrated in the following code fragment, which shows how you can migrate a store from one location to another. If the old store type is XML, then the example also converts the store to SQLite

```
NSPersistentStoreCoordinator *psc = [aManagedObjectContext
persistentStoreCoordinator];
NSURL *oldURL = <#URL identifying the location of the current store#>;
NSURL *newURL = <#URL identifying the location of the new store#>;
NSError *error = nil;
NSPersistentStore *xmlStore = [psc persistentStoreForURL:oldURL];
NSPersistentStore *sqliteStore = [psc migratePersistentStore:xmlStore
    toURL:newURL
    options:nil
    withType:NSSQLiteStoreType
    error:&error];
```

Core Data follows the procedure below to migrate a store:

1. Create a temporary persistence stack
2. Mount the old and new stores
3. Load all objects from the old store
4. Migrate the objects to the new store

The objects are given temporary IDs, then assigned to the new store. The new store then saves the newly assigned objects (committing them to the external repository).

Core Data then informs other stacks that the object IDs have changed (from the old to the new stores), which is how things "keep running" after a migration.

5. Unmount old store
6. Return the new store

An error can occur if:

- You provide invalid parameters to the method
- Core Data cannot add the new store
- Core Data cannot remove the old store

In the latter two cases, you get the same errors you would if you called `addPersistentStore:` or `removePersistentStore:` directly. If an error occurs when adding or removing the store, you should treat this as an exception since the persistence stack is likely to be in an inconsistent state.

If something fails during the migration itself, instead of an error you get an exception. In these cases, Core Data unwinds cleanly and there should be no repair work necessary. You can examine the exception description to determine what went wrong—there is a wide variety of possible errors, ranging from "disk is full" and "permissions problems" to "The SQLite store became corrupted" and "Core Data does not support cross store relationships".

Associate Metadata With a Store to Provide Additional Information and Support Spotlight Indexing

A store's **metadata** provides additional information about the store that is not directly associated with any of the entities in the store.

The metadata is represented by a dictionary. Core Data automatically sets key-value pairs to indicate the store type and its UUID. You can add additional keys that may be either custom for your application, or one of the standard set of keys to support Spotlight indexing (if you also write a suitable importer) such as `kMDItemKeywords`.

You should be careful about what information you put into metadata. First, Spotlight imposes a limit to the size of metadata. Second, replicating an entire document in metadata is probably not useful. Note, though, that it is possible to create a URL to identify a particular object in a store (using `URIRepresentation`)—the URL may be useful to include as metadata.

Getting the Metadata

There are two ways to get the metadata for a store:

1. Given an instance of a persistent store, you can get its metadata using the `NSPersistentStoreCoordinator` instance method `metadataForPersistentStore:`.
2. You can retrieve metadata from a store without the overhead of creating a persistence stack using the `NSPersistentStoreCoordinator` class method, `metadataForPersistentStoreOfType:URL:error:`.

There is an important difference between these approaches. The instance method, `metadataForPersistentStore:`, returns the metadata as it currently is in your program, including any changes that may have been made since the store was last saved. The class method, `metadataForPersistentStoreOfType:URL:error:`, returns the metadata as it is currently represented in the store itself. If there are pending changes to the store, the returned value may therefore be out of sync.

Setting the Metadata

There are two ways you can set the metadata for a store:

1. Given an instance of a persistent store, you can set its metadata using the `NSPersistentStoreCoordinator` instance method, `setMetadata:forPersistentStore:`.
2. You can set the metadata without the overhead of creating a persistence stack using the `NSPersistentStoreCoordinator` class method, `setMetadata:forPersistentStoreOfType:URL:error:`.

There is again an important difference between these approaches. If you use `setMetadata:forPersistentStore:`, you must save the store (through a managed object context) before the new metadata is saved. If you use `setMetadata:forPersistentStoreOfType:URL:error:`, however, the metadata is updated immediately (and the last-modified date of the file changed). This difference has particular implications if you use `NSPersistentDocument` on OS X. If you update the metadata using `setMetadata:forPersistentStoreOfType:URL:error:` while you are actively working on the persistent store (that is, while there are unsaved changes), then when you save the document you will see a warning, “This document's file has been changed by another application since you opened or saved it.” To avoid this, you should instead use `setMetadata:forPersistentStore:`. To find the document's persistent store, you typically ask the persistent store coordinator for its persistent stores (`persistentStores`) and use the first item in the returned array.

Because Core Data manages the values for `NSStoreType` and `NSStoreUUID`, you should make a mutable copy of any existing metadata before setting your own keys and values, as illustrated in the following code fragment.

```
NSError *error;
NSURL *storeURL = <#URL identifying the location of the store#>;

NSDictionary *metadata =
    [NSPersistentStoreCoordinator metadataForPersistentStoreWithURL:storeURL
    error:&error]
if (metadata == nil) {
    /* Deal with the error. */
}
else {
    NSMutableDictionary *newMetadata = [metadata mutableCopy];
    newMetadata[(NSString *)kMDItemKeywords] = @[ @"MyKeyWord" ];
    // Set additional key-value pairs as appropriate.
    [NSPersistentStore setMetadata:newMetadata
                        forPersistentStoreWithURL:storeURL
                        error:&error];
}
```

Core Data and Cocoa Bindings

Changes made to objects' property values should be propagated to the user interface, and user interface elements displaying the same property should be kept synchronized. Cocoa bindings provides a control layer for Cocoa but, whereas the Core Data framework focuses on the model, Cocoa bindings focus on the user interface. In many situations, Cocoa bindings makes it easy to keep the user interface properly synchronized. The Core Data framework is designed to interoperate seamlessly with, and enhance the utility of, Cocoa bindings.

iOS: Note that Cocoa bindings are not available on iOS.

Cocoa bindings and Core Data are largely orthogonal. In general, Cocoa bindings work in exactly the same way with managed objects as with other Cocoa model objects. You can also use the same predicate objects and sort descriptors as you use to fetch objects from the persistent store to filter and sort objects in memory—for example to present in a table view. This gives you a consistent API set to use throughout your application. There, however, are a few (typically self-evident) differences in configuration and operation.

In addition to the issues described in this article, there are a few other areas where the interaction between Core Data and Cocoa Bindings may cause problems; these are described in [“Troubleshooting Core Data”](#) (page 149), in particular:

- [“Custom relationship set mutator methods are not invoked by an array controller”](#) (page 157)
- [“Cannot access contents of an object controller after a nib is loaded”](#) (page 157)
- [“Table view or outline view contents not kept up-to-date when bound to an NSArrayController or NSTreeController object”](#) (page 159)

Modulo these exceptions, everything that is discussed and described in *Cocoa Bindings Programming Topics* applies equally to Core Data-based applications and you should use the same techniques for configuring and debugging bindings when using Core Data as you would if you were not using Core Data.

Additions to Controllers

The main area where Core Data adds to Cocoa bindings is in the configuration of the controller objects such as `NSObjectController` and `NSArrayController`. Core Data adds the following features to those classes:

- A reference to a managed object context that is used for all fetches, insertions, and deletions.
If a controller's content is a managed object or collection of managed objects, you must either bind or set the managed object context for the controller.
- An entity name that is used instead of the content object class to create new objects
- A reference to a fetch predicate that constrains what is fetched to set the content if the content is not set directly
- A content binding option ("Deletes Objects On Remove") that—if the content is bound to a relationship—specifies whether objects removed from the controller are deleted in addition to being removed from the relationship

Automatically Prepares Content Flag

If the "automatically prepares content" flag (see, for example, `setAutomaticallyPreparesContent:`) is set for a controller, the controller's initial content is fetched from its managed object context using the controller's current fetch predicate. It is important to note that the controller's fetch is executed as a delayed operation performed after its managed object context is set (by nib loading)—this therefore happens after `awakeFromNib` and `windowControllerDidLoadNib:`. This can create a problem if you want to perform an operation with the contents of an object controller in either of these methods, since the controller's content is `nil`. You can work around this by executing the fetch "manually" with `fetchWithRequest:merge:error:`. You pass `nil` as the fetch request argument to use the default request, as illustrated in the following code fragment.

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

    NSError *error = nil;
    BOOL ok = [arrayController fetchWithRequest:nil merge:NO error:&error];
    // ...
}
```

Entity Inheritance

If you specify a super entity as the entity for a fetch request, the fetch returns matching instances of the entity and sub-entities (see [“Fetching and Entity Inheritance”](#) (page 66)). As a corollary, if you specify a super entity as the entity for a controller, it fetches matching instances of the entity and any sub-entities. If you specify an abstract super-entity, the controller fetches matching instances of concrete sub-entities.

Filter Predicate for a To-many Relationship

Sometimes you may want to set up a filter predicate for a search field that lets a user filter the contents of an array controller based on the destination of a to-many relationship. If you want to search a to-many relationship, you need to use an **ANY** or **ALL** in the predicate. For instance, if you want to fetch Departments in which at least one of the employees has the first name "Matthew", you use an **ANY** operator as shown in the following example:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"ANY employees.firstName like 'Matthew'"];
```

You use the same syntax in a search field's predicate binding:

```
ANY employees.firstName like $value
```

Note: You cannot use the contains operator (for example, `ANY employees.firstName contains 'Matthew'`) because the contains operator does not work with the **ANY** operator.

Things are more complex, however, if you want to match prefix and/or suffix—for instance, if you want to look for Departments in which at least one of the employees has the first name “Matt”, “Matthew”, “Mattie”, or any other name beginning with “Matt.” Fundamentally you simply need to add wildcard matching:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"ANY employees.firstName like 'Matt*'];
```

You *cannot*, though, use the same syntax within a search field's predicate binding:

```
// does not work
ANY employees.firstName like '$value*'
```

The reasons for this are described in *Predicate Programming Guide*—putting quotes in the predicate format prevents the variable substitution from happening. Instead, you must use substitute any wildcards first as illustrated in this example:

```
NSString *value = @"Matt";
NSString *wildcardedString = [NSString stringWithFormat:@"%s*", value];
```

```
[[NSPredicate predicateWithFormat:@"ANY employees.firstName like %@",  
wildcardedString];
```

By implication, therefore, you must write some code to support this behavior.

Note: You may find that search field predicate bindings filter results inconsistently with wildcard characters. This is due to a bug in `NSArrayController`. The workaround is to create a subclass of `NSArrayController` and override `arrangeObjects:` to simply invoke `super`'s implementation.

Change Management

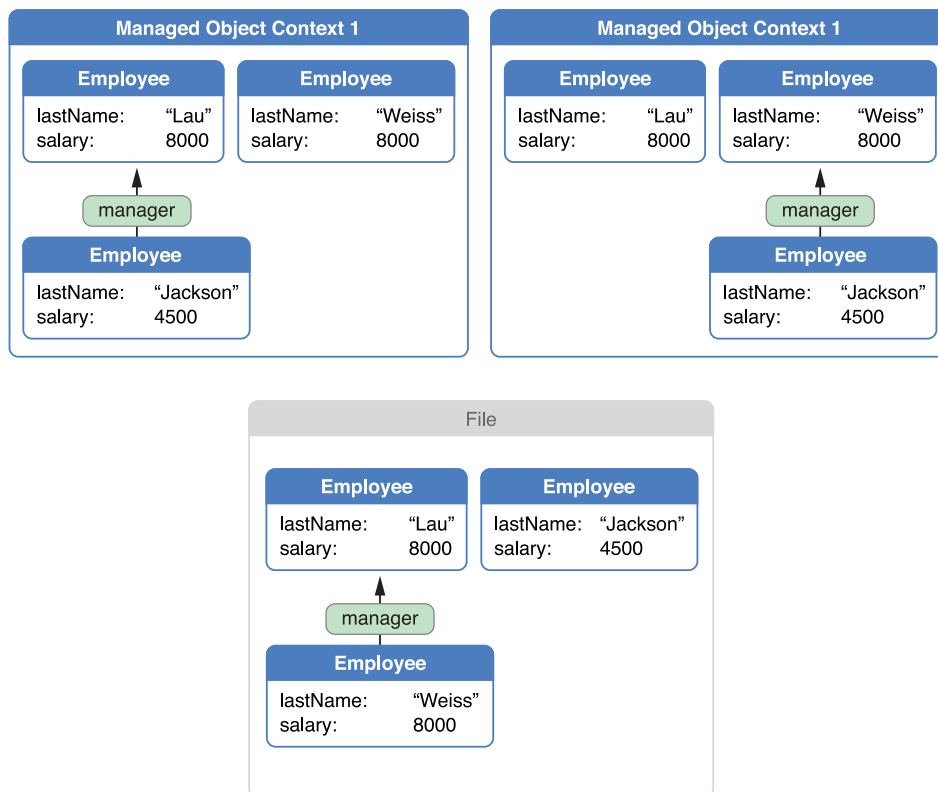
If your application contains more than one managed object context and you allow objects to be modified in more than context, then you need to be able to reconcile the changes.

Disjoint Edits

The object graph associated with any given managed object context must be internally consistent. If you have multiple managed object contexts in the same application, however, it is possible that each may contain objects that represent the same records in the persistent store, but whose characteristics are mutually inconsistent. In

an employee application, for example, you might have two separate windows that display the same set of employees, but distributed between different departments and with different managers, as shown in Figure 1.

Figure 1 Managed object contexts with mutually inconsistent data values



Ultimately though there can only be one “truth” and differences between these views must be detected and reconciled when data is saved. When one of the managed object contexts is saved, its changes are pushed through the persistent store coordinator to the persistent store. When the second managed object context is saved, conflicts are detected using a mechanism called **optimistic locking**; how the conflicts are resolved depends on how you have configured the context.

Conflict Detection and Optimistic Locking

When Core Data fetches an object from a persistent store, it takes a **snapshot** of its state. A snapshot is a dictionary of an object’s persistent properties—typically all its attributes and the global IDs of any objects to which it has a to-one relationship. Snapshots participate in optimistic locking. When the framework saves, it compares the values in each edited object’s snapshot with the then-current corresponding values in the persistent store.

- If the values are the same, then the store has not been changed since the object was fetched, so the save proceeds normally. As part of the save operation, the snapshots' values are updated to match the saved data.
- If the values differ, then the store has been changed since the object was fetched or last saved; this represents an optimistic locking failure.

Conflict Resolution

You can get an optimistic locking failure if more than one persistence stack references the same external data store (whether you have multiple persistence stacks in a single application or you have multiple applications). In this situation there is the possibility that the same conceptual managed object will be edited in two persistence stacks simultaneously. In many cases, you want to ensure that subsequent changes made by the second stack do not overwrite changes made by the first, but there are other behaviors that may be appropriate. You can choose the behavior by choosing for the managed object context a merge policy that is suitable for your situation.

The default behavior is defined by the `NSErrorMergePolicy`. This policy causes a save to fail if there are any merge conflicts. In the case of failure, the save method returns with an error with a `userInfo` dictionary that contains the key `@\"conflictList\"`; the corresponding value is an array of conflict records. You can use the array to tell the user what differences there are between the values they are trying to save and those current in the store. Before you can save you must either fix the conflicts (by re-fetching objects so that the snapshots are updated) or choose a different policy. The `NSErrorMergePolicy` is the only policy that generates an error. Other policies—`NSMergeByPropertyStoreTrumpMergePolicy`, `NSMergeByPropertyObjectTrumpMergePolicy`, and `NSOverwriteMergePolicy`—allow the save to proceed by merging the state of the edited objects with the state of the objects in the store in different ways. The `NSRollbackMergePolicy` discards in-memory state changes for objects in conflict and uses the persistent store's version of the objects' state.

Snapshot Management

An application that fetches hundreds of rows of data can build up a large cache of snapshots. Theoretically, if enough fetches are performed, a Core Data-based application can contain all the contents of a store in memory. Clearly, snapshots must be managed in order to prevent this situation.

Responsibility for cleaning up snapshots rests with a mechanism called **snapshot reference counting**. This mechanism keeps track of the managed objects that are associated with a particular snapshot—that is, managed objects that contain data from a particular snapshot. When there are no remaining managed object instances associated with a particular snapshot (which Core Data determines by maintaining a list of these references), the strong reference to the snapshot is broken.

Communicating Changes Between Contexts

If you use more than one managed object context in an application, Core Data does not automatically notify one context of changes made to objects in another. In general, this is because a context is intended to provide a scratch pad where you can make changes to objects in isolation, and if you wish you can discard the changes without affecting other contexts. If you do need to synchronize changes between contexts, how a change should be handled depends on the user visible semantics you want in the second context, and on the state of the objects in the second context.

Consider an application with two managed object contexts and a single persistent store coordinator. If a user deletes an object in the first context (`moc1`), you may need to inform the second context (`moc2`) that has been deleted. In all cases, `moc1` posts an `NSManagedObjectContextDidSave` notification that your application should register for and use as the trigger for whatever actions it needs to take. This notification contains information not only about deleted objects, but also about changed objects. You need to handle these changes since they may be the result of the delete (most of the ways this can happen involve transient relationships or fetched properties).

There are multiple axes you must consider when deciding how you want to handle your delete notification. The important ones are:

- What other changes exist in the second context?
- Does the instance of the object that was deleted have changes in the second context?
- Can the changes made in the second context be undone?

These are somewhat orthogonal, and what actions you take to synchronize the contexts depend on the semantics of your application. The following three strategies are presented in order of increasing complexity.

1. The simplest case is when the object itself has not changed in `moc2` and you do not have to worry about undo; in this situation, you can just delete the object. The next time `moc2` saves, the framework will notice that you are trying to re-delete an object, ignore the optimistic locking warning, and continue without error.
2. If you do not care about the contents of `moc2`, you can simply reset it (using `reset`) and refetch any data you need after the reset. This will reset the undo stack as well, and the deleted object is now gone. The only issue here is determining what data to refetch. You can do this by, before you reset, collecting the IDs (`objectID`) of the managed objects you still need and using those to reload once the reset has happened (you must exclude the deleted IDs, and it is best to create fetch requests with `IN` predicates to avoid problems will not being able to fulfill faults for deleted IDs).

3. If the object has changed in `moc2`, but you do not care about undo, your strategy depends on what it means for the semantics of your application. If the object that was deleted in `moc1` has changes in `moc2`, should it be deleted from `moc2` as well? Or should it be resurrected and the changes saved? What happens if the original deletion triggered a cascade delete for objects that have not been faulted into `moc2`? What if the object was deleted as part of a cascade delete?

There are two workable options (a third, unsatisfactory option is described later):

- a. The simplest strategy is to just discard the changes by deleting the object.
- b. Alternatively, if the object is standalone, you can set the merge policy on the context to `NSMergePolicyOverwrite`. This will cause the changes in the second context to overwrite the delete in the database.

Note that this will cause *all* changes in `moc2` to overwrite any changes made in `moc1`.

The preceding are the best solutions, and are least likely to leave your object graph in an unsustainable state as a result of something you missed. There are various other strategies, but all are likely to lead to inconsistencies and errors. They are listed here as examples so that you can recognize them and avoid them. *If you find yourself trying to adopt any of these strategies, you should redesign your application's architecture to follow one of the patterns described previously.*

1. If you have a situation like 3(b) above, but the object *not* standalone, and for some reason you want to save those changes, the best you're likely to be able to do is to resurrect the part of the graph that had been loaded into `moc2`, which may or may not make sense in the context of your application. Again you do this by setting the merge policy to `NSMergePolicyOverwrite`, but you also need some up-front application design, and some meddling with the objects in the 'deleted' object's relationships.

In order for the world to make some amount of sense later, you need to automatically fault in any relationships that might need to be resurrected when you fault in the object. Then, when you get a delete notification, you need to make the context think all the objects related to the deleted object have changed, so that they will be saved as well. This will bloat your application's memory use, since you'll end up with possibly irrelevant data as a precaution against something that may not happen, and if you're not careful, you can end up with your database in a hybrid state where it is neither what `moc1` tried to create, nor what `moc2` would expect (for example, if you missed a relationship somewhere and you now have partial relationships, or orphaned nodes).

2. The second worst of all worlds is when you have changes to other objects you can't blow away in the second MOC, the object itself has changes that you are willing to discard, and you care about undo. You can't reset the context, because that loses the changes. If you delete the object, the delete will get pushed onto the undo stack and will be undoable, so the user could undo, resave, and run into the semantic problems mentioned in 3 above, only worse because you have not planned for them.

The only real way to solve this is to—separately, in your application code—keep track of the objects which are changed as a result of the delete. You then need to track user undo events, and when the user undoes past a delete, you can then "rerun" the deletion. This is likely to be complex and inefficient if a significant number of changes are propagated.

3. The worst case is you have changes to other objects you cannot discard, the object has changes you want to keep, and you care about undo. There may be a way to deal with this, but it will require considerable effort and any solution is likely to be complicated and fragile.

Persistent Store Features

Core Data provides several types of persistent store. This article describes the features and benefits of each, and how you can migrate from one type of store to another.

Important: In OS X v10.4, there is no explicit class for persistent stores—you can only type a store instance as an `id`—consequently there is also no API for persistent store objects in OS X v10.4. The techniques described below generally also apply to OS X v10.4, but where a type is given as `NSPersistentStore` * you should use `id`.

Store Types and Behaviors

Core Data provides three sorts of disk-based persistent store—XML, atomic, and SQLite—and an in-memory store. (Core Data provides the binary store type—`NSBinaryStoreType`—as a built-in atomic store; you can also create your own atomic store types—see “[Custom store types](#)” (page 131).) From the application code perspective, in general you should not be concerned about implementation details for any particular store. You should interact with managed objects and the persistence stack. There are, however, some behavioral differences between the types of store that you should consider when deciding what type of store to use.

iOS: The XML store is not available on iOS.

	XML	Atomic	SQLite	In-Memory
<i>Speed</i>	Slow	Fast	Fast	Fast
<i>Object Graph</i>	Whole	Whole	Partial	Whole
<i>Other Factors</i>	Externally parseable			No backing required

Important: Although Core Data supports SQLite as a store type, the store format—like those of the other native Core Data stores—is private. You cannot create a SQLite database using native SQLite API and use it directly with Core Data (nor should you manipulate an existing Core Data SQLite store using native SQLite API). If you have an existing SQLite database, you need to import it into a Core Data store (see [“Efficiently Importing Data”](#) (page 160)).

Store-specific behavior

Given the abstraction that Core Data offers, there is typically no need to use the same store throughout the development process. It is common, for example, to use the XML store early in a project life-cycle, since it is fairly human-readable and you can inspect a file to determine whether or not it contains the data you expect. In a deployed application that uses a large data set, you typically use an SQLite store, since this offers high performance and does not require that the entire object graph reside in memory. You might use the binary store if you want store writes to be atomic. There are, however, some features and considerations that are specific to particular store types. These are described in following sections.

Custom store types

In OS X v10.5 and later you can create your own atomic store types. For details, see *Atomic Store Programming Topics*.

In OS X v10.4, you cannot write your own object store which interoperates transparently with the Core Data stack. You can, however, manage object persistence yourself by using an in-memory store. Before you load your data, you create an in-memory store. When you load your data, you create instances of the appropriate model classes and insert them into a managed object context, associate them with the in-memory store (see `insertObject:` and `assignObject:toPersistentStore:`). The managed objects are then fully integrated into the Core Data stack and benefit from features such as undo management. You are also responsible, however, for saving the data. You must register to receive `NSManagedObjectContextDidSaveNotification` notifications from the managed object context, and upon receipt of the notification save the managed objects to the persistent store.

Security

Core Data makes no guarantees regarding the security of persistent stores from untrusted sources and cannot detect whether files have been maliciously modified. The SQLite store offers slightly better security than the XML and binary stores, but it should not be considered inherently secure. Note that you should also consider the security of store metadata since it is possible for data archived in the metadata to be tampered with independently of the store data. If you want to ensure data security, you should use a technology such as an encrypted disk image.

Fetch Predicates and Sort Descriptors

There are some interactions between fetching and the type of store. In the XML, binary, and in-memory stores, evaluation of the predicate and sort descriptors is performed in Objective-C with access to all Cocoa's functionality, including the comparison methods on `NSString`. The SQL store, on the other hand, compiles the predicate and sort descriptors to SQL and evaluates the result in the database itself. This is done primarily for performance, but it means that evaluation happens in a non-Cocoa environment, and so sort descriptors (or predicates) that rely on Cocoa cannot work. The supported sort selectors are `compare:` and `caseInsensitiveCompare:`, `localizedCompare:`, `localizedCaseInsensitiveCompare:`, and `localizedStandardCompare:` (the latter is Finder-like sorting, and what most people should use most of the time). In addition you cannot sort on transient properties using the SQLite store.

There are additional constraints on the predicates you can use with the SQLite store:

- You cannot necessarily translate “arbitrary” SQL queries into predicates.
- Prior to OS X v10.6, Core Data's SQL store did not support the `MATCHES` operator (you could use the `MATCHES` operator to perform in-memory filtering of results returned from the store).
- You can only have one to-many element in a key path in a predicate.

For example, no `toOne.toMany.toMany`, or `toMany.toOne.toMany` type constructions (they evaluate to sets of sets). As a consequence, in any predicate sent to the SQL store, there may be only one operator (and one instance of that operator) from `ALL`, `ANY`, and `IN`.

- CoreData supports a `noindex:` (see `NSPredicate` documentation re: function expressions) that can be used to drop indices in queries passed to SQLite. This is done primarily for performance reasons: SQLite uses a limited number of indices per query, and `noindex:` allows the user to preferentially specify which indexes should not be used.

SQLite Store

File-systems supported by the SQLite store

The SQLite store supports reading data from a file that resides on any type of file-system. The SQLite store does not in general, however, support writing directly to file-systems which do not implement byte-range locking. For DOS filesystems and for some NFS file system implementations that do not support byte-range locking correctly, SQLite will use "`<dbfile>.lock`" locking, and for SMB file systems it uses flock-style locking.

To summarize: byte-range locking file systems have the best concurrent read/write support; these include HFS+, AFP, and NFS. File systems with simplistic file locking are also supported but do not allow for as much concurrent read/write access by multiple processes; these include SMB, and DOS. The SQLite store does *not* support writing to WebDAV file-systems (this includes iDisk).

File Size May Not Reduce After Deleting a Record

Simply deleting a record from a SQLite store does not necessarily result in a reduction in the size of the file. If enough items are removed to free up a page in the database file, SQLite's automatic database vacuuming will reduce the size of the file as it rearranges the data to remove that page. Similarly, the file size may be reduced if you remove an item that itself occupies multiple pages (such as a thumbnail image).

An SQLite file is arranged as a collection of pages. The data within those pages is managed via B-trees, not as simple fixed-length records. This is much more efficient for searching and for overall storage, since it allows SQLite to optimize how it stores both data and indexes in a single file, and is also the foundation of its data integrity (transaction and journaling) mechanism. However, the cost of this is that some delete operations may leave holes in the file. If you delete some data and add other data, the holes left by the deleted data may be filled by the added data, or the file may be vacuumed to compact its data, whichever SQLite considers most appropriate based on the operations you're performing.

Configuring a SQLite Store's Save Behavior

When Core Data saves a SQLite store, SQLite updates just part of the store file. Loss of that partial update would be catastrophic, so you may want to ensure that the file is written correctly before your application continues. Unfortunately, doing so means that in some situations saving even a small set of changes to an SQLite store can take considerably longer than saving to, say, an XML store. (For example, where saving to an XML file might take less than a hundredth of a second, saving to an SQLite store may take almost half a second. This is not an issue for XML or Binary stores—since they are atomic, there is a much lower likelihood of data loss that involves corruption of the file, especially since the writes are typically atomic and the old file is not deleted until the new has been successfully written.)

fsync in OS X: Since in OS X the `fsync` command does not make the guarantee that bytes are written, SQLite sends a `F_FULLFSYNC` request to the kernel to ensure that the bytes are actually written through to the drive platter. This causes the kernel to flush all buffers to the drives and causes the drives to flush their track caches. Without this, there is a significantly large window of time within which data will reside in volatile memory—and in the event of system failure you risk data corruption.

Core Data provides a way to control sync behavior in SQLite using two independent pragmas, giving you control over the tradeoff between performance and reliability:

- `synchronous` controls the frequency of disk-syncing
`PRAGMA synchronous FULL [2] / NORMAL [1] / OFF [0]`
- `fullfsync` controls the type of disk-sync operation performed
`PRAGMA fullfsync 1 / 0`
In OS X v10.5, the default is 0.

The pragmas are publicly documented at <http://sqlite.org/pragmas.html>.

You can set both pragmas using the key `NSSQLitePragmasOption` in the options dictionary when opening the store. The `NSSQLitePragmasOption` dictionary contains pragma names as keys and string values as objects, as illustrated in the following example:

```
NSPersistentStoreCoordinator *psc = <#Get a persistent store coordinator#>;

NSMutableDictionary *pragmaOptions = [NSMutableDictionary dictionary];
[pragmaOptions setObject:@"NORMAL" forKey:@"synchronous"];
[pragmaOptions setObject:@"1" forKey:@"fullfsync"];

NSDictionary *storeOptions =
    [NSDictionary dictionaryWithObject:pragmaOptions forKey:NSSQLitePragmasOption];
NSPersistentStore *store;
NSError *error = nil;
store = [psc addPersistentStoreWithType:NSSQLiteStoreType
    configuration:nil
    URL:url
    options:storeOptions
    error:&error];
```

OS X v10.4: OS X v10.4 uses `full_fsync` by default. Since the `fsync` command does not make the guarantee that bytes are written, SQLite sends a `F_FULLFSYNC` request to the kernel. This causes the kernel to flush all buffers to the drives and causes the drives to flush their track caches.

In OS X v10.4, there are only two settings to control the way in which data in a SQLite-based store is written to disk. If you want to trade risk of data corruption against the time taken to save a file, you can set the defaults key `com.apple.CoreData.SQLiteDebugSynchronous` to one of three values:

0: Disk syncing is switched off

1: Normal

2 (The default): Disk syncing is performed via the `fcntl FULL_FSYNC` command—a costly operation but one that guarantees data is written to disk

Important: The default behaviors in OS X v10.4 and 10.5 are different. In OS X v10.4, SQLite uses `FULL_FSYNC` by default; in OS X v10.5 it does not.

Concurrency with Core Data

There are several situations in which performing operations with Core Data on a background thread or queue is beneficial, in particular if you want to ensure that your application's user interface remains responsive while Core Data is undertaking a long-running task. If you do perform concurrent operations with Core Data, however, you need to take considerable care that object graphs do not get into an inconsistent state.

Note: You can use threads, serial operation queues, or dispatch queues for concurrency. For the sake of conciseness, this article uses “thread” throughout to refer to any of these.

If you choose to use concurrency with Core Data, you also need to consider the application environment. For the most part, AppKit and UIKit are not thread safe; in particular, on OS X Cocoa bindings and controllers are not thread safe—if you are using these technologies, multi-threading may be complex.

Use Thread Confinement to Support Concurrency

The pattern recommended for concurrent programming with Core Data is *thread confinement*: each thread must have its own entirely private managed object context.

There are two possible ways to adopt the pattern:

1. Create a separate managed object context for each thread and share a single persistent store coordinator. This is the typically-recommended approach.
2. Create a separate managed object context and persistent store coordinator for each thread.

This approach provides for greater concurrency at the expense of greater complexity (particularly if you need to communicate changes between different contexts) and increased memory usage.

You must create the managed context on the thread on which it will be used. If you use `NSOperation`, note that its `init` method is invoked on the same thread as the caller. You must not, therefore, create a managed object context for the queue in the queue's `init` method, otherwise it is associated with the caller's thread. Instead, you should create the context in `main` (for a serial queue) or `start` (for a concurrent queue).

Using thread confinement, you should not pass managed objects or managed object contexts between threads. To “pass” a managed object from one context another across thread boundaries, you either:

- Pass its object ID (`objectID`) and use `objectWithID:` or `existingObjectWithID:error:` on the receiving managed object context.

The corresponding managed objects must have been saved—you cannot pass the ID of a newly-inserted managed object to another context.

- Execute a fetch on the receiving context.

These create a local version of the managed object in the receiving context.

You can use the methods provided by `NSFetchRequest` to make working with data across threads easier and more efficient. For example, you can configure a fetch request to return just object IDs but also include the row data (and update the row cache)—this can be useful if you're just going to pass those object IDs from a background thread to another thread.

There is typically no need to use locks with managed objects or managed object contexts. However, if you use a single persistent store coordinator shared by multiple contexts and want to perform operations on it (for example, if you want to add a new store), or if you want to aggregate a number of operations in one context together as if a virtual single transaction, you should lock the persistent store coordinator.

Track Changes in Other Threads Using Notifications

Changes you make to a managed object in one context are not propagated to a corresponding managed object in a different context unless you either refetch or re-fault the object. If you need to track in one thread changes made to managed objects in another thread, there are two approaches you can take, both involving notifications. For the purposes of explanation, consider two threads, “A” and “B”, and suppose you want to propagate changes from B to A.

Typically, on thread A you register for the managed object context save notification, `NSManagedObjectContextDidSaveNotification`. When you receive the notification, its user info dictionary contains arrays with the managed objects that were inserted, deleted, and updated on thread B. Because the managed objects are associated with a different thread, however, you should not access them directly. Instead, you pass the notification as an argument to `mergeChangesFromContextDidSaveNotification:` (which you send to the context on thread A). Using this method, the context is able to safely merge the changes.

If you need finer-grained control, you can use the managed object context *change* notification, `NSManagedObjectContextObjectsDidChangeNotification`—the notification’s user info dictionary again contains arrays with the managed objects that were inserted, deleted, and updated. In this scenario, however, you register for the notification on thread B. When you receive the notification, the managed objects in the user info dictionary are associated with the same thread, so you can access their object IDs. You pass the object IDs to thread A by sending a suitable message to an object on thread A. Upon receipt, on thread A you can refetch the corresponding managed objects.

Note that the *change* notification is sent in `NSManagedObjectContext`'s `processPendingChanges` method. The *main* thread is tied into the event cycle for the application so that `processPendingChanges` is invoked automatically after every user event on contexts owned by the main thread. This is not the case for background threads—when the method is invoked depends on both the platform and the release version, so you should not rely on particular timing. If the secondary context is not on the main thread, you should call `processPendingChanges` yourself at appropriate junctures. (You need to establish your own notion of a work “cycle” for a background thread—for example, after every cluster of actions.)

Fetch in the Background for UI Responsiveness

The `executeFetchRequest:error:` method intrinsically scales its behavior appropriately for the hardware and work load. If necessary, the Core Data will create additional private threads to optimize fetching performance. You will not improve absolute fetching speed by creating background threads for the purpose. It may still be appropriate, however, to fetch in a background thread or queue to prevent your application's user interface from blocking. This means that if a fetch is complicated or returns a large amount of data, you can return control to the user and display results as they arrive.

Following the thread confinement pattern, you use two managed object contexts associated with a single persistent store coordinator. You fetch in one managed object context on a background thread, and pass the *object IDs* of the fetched objects to another thread. In the second thread (typically the application's main thread, so that you can then display the results), you use the second context to fault in objects with those object IDs (you use `objectWithID:` to instantiate the object). (This technique is only useful if you are using an SQLite store, since data from binary and XML stores is read into memory immediately on open.)

Saving in a Background Thread is Error-prone

Asynchronous queues and threads do not prevent an application from quitting. (Specifically, all `NSThread`-based threads are “detached”—see the documentation for `pthread` for complete details—and a process runs only until all not-detached threads have exited.) If you perform a save operation in a background thread, therefore, it may be killed before it is able to complete. If you need to save on a background thread, you must write additional code such that the main thread prevents the application from quitting until all the save operation is complete.

If You Don't Use Thread Containment

If you choose not to use the thread containment pattern—that is, if you try to pass managed objects or contexts between threads, and so on—you must be *extremely* careful about locking, and as a consequence you are likely to negate any benefit you may otherwise derive from multi-threading. You also need to consider that:

- Any time you manipulate or access managed objects, you use the associated managed object context. Core Data does not present a situation where reads are “safe” but changes are “dangerous”—*every* operation is “dangerous” because every operation has cache coherency effects and can trigger faulting.
- Managed objects themselves are not thread safe.

If you want to work with a managed object across different threads, you must lock its context (see `NSLocking`).

If you share a managed object context or a persistent store coordinator between threads, you must ensure that any method invocations are made from a thread-safe scope. For locking, you should use the `NSLocking` methods on managed object context and persistent store coordinator instead of implementing your own mutexes. These methods help provide contextual information to the framework about the application's intent—that is, in addition to providing a mutex, they help scope clusters of operations.

Typically you lock the context or coordinator using `tryLock` or `lock`. If you do this, the framework will ensure that what it does behind the scenes is also thread-safe. For example, if you create one context per thread, but all pointing to the same persistent store coordinator, Core Data takes care of accessing the coordinator in a thread-safe way (the `lock` and `unlock` methods of `NSManagedObjectContext` handle recursion).

If you lock (or successfully `tryLock`) a context, you must keep a strong reference to that context until you invoke `unlock`. If you don't, in a multi-threaded environment, you may cause a deadlock.

Core Data Performance

In general, Core Data is very efficient. For many applications, an implementation that uses Core Data may be more efficient than a *comparable* application that does not. It is possible, however, to use the framework in such a way that its efficiency is reduced. This article describes how to get the most out of Core Data.

Introduction

Core Data is a rich and sophisticated object graph management framework capable of dealing with large volumes of data. The SQLite store can scale to terabyte sized databases with billions of rows/tables/columns. Unless your entities themselves have very large attributes (although see [“Large Data Objects \(BLOBs\)”](#) (page 146)) or large numbers of properties, 10,000 objects is considered to be a fairly small size for a data set.

For a very simple application it is certainly the case that Core Data adds some overhead (compare a vanilla Cocoa document-based application with a Cocoa Core Data document-based application), however Core Data adds significant functionality. For a small overhead, even a simple Core Data-based application supports undo and redo, validation, object graph maintenance, and provides the ability to save objects to a persistent store. If you implemented this functionality yourself, it is quite likely that the overhead would exceed that imposed by Core Data. As the complexity of an application increases, so the proportionate overhead that Core Data imposes typically decreases while at the same time the benefit typically increases (supporting undo and redo in a large application, for example, is usually *hard*).

`NSManagedObject` uses an internal storage mechanism for data that is highly optimized. In particular, it leverages the information about the types of data that is available through introspecting the model. When you store and retrieve data in a manner that is key-value coding and key-value observing compliant, it is likely that using `NSManagedObject` will be faster than any other storage mechanism—including for the simple get/set cases. In a modern Cocoa application that leverages Cocoa Bindings, given that Cocoa Bindings is reliant upon key-value coding and key-value observing it would be difficult to build a raw data storage mechanism that provides the same level of efficiency as Core Data.

Like all technologies, however, Core Data can be abused. Using Core Data does not free you from the need to consider basic Cocoa patterns, such as memory management. You should also consider how you fetch data from a persistent store. If you find that your application is not performing as well as you would like, you should use profiling tools such as Shark to determine where the problem lies (see [Performance & Debugging](#)).

Fetching Managed Objects

Each round trip to the persistent store (each fetch) incurs an overhead, both in accessing the store and in merging the returned objects into the persistence stack. You should avoid executing multiple requests if you can instead combine them into a single request that will return all the objects you require. You can also minimize the number of objects you have in memory.

Fetch Predicates

How you use predicates can significantly affect the performance of your application. If a fetch request requires a compound predicate, you can make the fetch more efficient by ensuring that the most restrictive predicate is the first, especially if the predicate involves text matching (`contains`, `endsWith`, `like`, and `matches`) since correct Unicode searching is slow. If the predicate combines textual and non-textual comparisons, then it is likely to be more efficient to specify the non-textual predicates first, for example `(salary > 5000000) AND (lastName LIKE 'Quincey')` is better than `(lastName LIKE 'Quincey') AND (salary > 5000000)`. For more about creating predicates, see *Predicate Programming Guide*.

Fetch Limits

You can set a limit to the number of objects a fetch will return using the method `setFetchLimit:` as shown in the following example.

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];  
[request setFetchLimit:100];
```

If you are using the SQLite store, you can use a fetch limit to minimize the working set of managed objects in memory, and so improve the performance of your application.

If you do need to retrieve a large number of objects, you can make your application appear more responsive by executing two fetches. In the first fetch, you retrieve a comparatively small number of objects—for example, 100—and populate the user interface with these objects. You then execute a second fetch to retrieve the complete result set (that is, you execute a fetch without a fetch limit).

Prior to OS X v10.6, there is no way to “batch” fetches (or in database terms, to set a cursor). That is, you cannot fetch the “first” 100 objects, then the second 100, then the third, and so on. In OS X v10.6 and later and on iOS, you can use `fetchOffset` to manage a subrange of an arbitrary result set.

In general, however, you are encouraged to use predicates to ensure that you retrieve only those objects you require.

Faulting Behavior

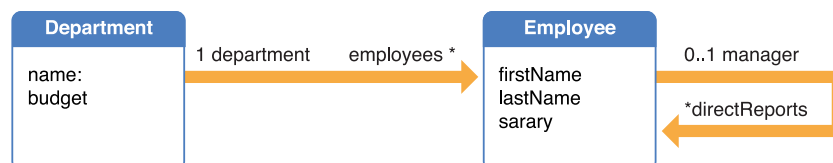
Firing faults can be a comparatively expensive process (potentially requiring a round trip to the persistent store), and you may wish to avoid unnecessarily firing a fault. You can safely invoke the following methods on a fault without causing it to fire: `isEqual:`, `hash`, `superclass`, `class`, `self`, `zone`, `isProxy`, `isKindOfClass:`, `isMemberOfClass:`, `conformsToProtocol:`, `respondsToSelector:`, `description`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, and `isFault`.

Since `isEqual` and `hash` do not cause a fault to fire, managed objects can typically be placed in collections without firing a fault. Note, however, that invoking key-value coding methods on the collection object might in turn result in an invocation of `valueForKey:` on a managed object, which would fire a fault. In addition, although the default implementation of `description` does not cause a fault to fire, if you implement a custom `description` method that accesses the object's persistent properties, this will cause a fault to fire.

Note that just because a managed object is a fault, it does not necessarily mean that the data for the object are not in memory—see the definition for `isFault`.

Batch Faulting and Pre-fetching with the SQLite Store

When you execute a fetch, Core Data fetches just instances of the entity you specify. In some situations (see [“Faulting Limits the Size of the Object Graph”](#) (page 110)), the destination of a relationship is represented by a fault. Core Data automatically resolves (fires) the fault when you access data in the fault. This lazy loading of the related objects is much better for memory use, and much faster for fetching objects related to rarely used (or very large) objects. It can also, however, lead to a situation where Core Data executes separate fetch requests for a number of individual objects, which incurs a comparatively high overhead. For example, given a model:



you might fetch a number of Employees and ask each in turn for their Department's name, as shown in the following code fragment.

```
NSFetchRequest * employeesFetch = <#A fetch request for Employees#>
// The request should include a predicate -- if you don't have a predicate here,
// you should probably just fetch all the Departments.
NSArray *fetchedEmployees = [moc executeFetchRequest:employeesFetch error:&error];
for (Employee *employee in fetchedEmployees)
```

```
{  
    NSLog(@"%@ -> %@ department", employee.name, employee.department.name);  
}
```

This might lead to the following behavior:

```
Jack -> Sales [fault fires]  
Jill -> Marketing [fault fires]  
Benjy -> Sales  
Gillian -> Sales  
Hector -> Engineering [fault fires]  
Michelle -> Marketing
```

Here, there are four round trips to the persistent store (one for the original fetch of Employees, and three for individual Departments) which represents a considerable overhead on top of the minimum (two—one for each entity).

There are two techniques you can use to mitigate this effect—**batch faulting** and **pre-fetching**.

Batch faulting

You can batch fault a collection of objects by executing a fetch request using a predicate with an `IN` operator, as illustrated by the following example. (In a predicate, `self` represents the object being evaluated—see “Predicate Format String Syntax”.)

```
NSArray *array = [NSArray arrayWithObjects:fault1, fault2, ..., nil];  
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"self IN %@", array];
```

In OS X v10.5 and later, when you create a fetch request you can use the `NSFetchRequest` method `setReturnsObjectsAsFaults:` to ensure that managed objects are not returned as faults.

Pre-fetching

Pre-fetching is in effect a special case of batch-faulting, performed immediately after another fetch. The idea behind pre-fetching is the anticipation of future needs. When you fetch some objects, sometimes you know that soon after you will also need related objects which may be represented by faults. To avoid the inefficiency of individual faults firing, you can pre-fetch the objects at the destination.

In OS X v10.5 and later, you can use the `NSFetchRequest` method `setRelationshipKeyPathsForPrefetching:` to specify an array of relationship key paths to prefetch along with the entity for the request. For example, given an `Employee` entity with a relationship to a `Department` entity: if you fetch all the employees then for each print out their name and the name of the department to which they belong, you can avoid the possibility of a fault being fired for each `Department` instance by prefetching the department relationship, as illustrated in the following code fragment:

```
NSManagedObjectContext *context = /* get the context */;
NSEntityDescription *employeeEntity = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:context];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:employeeEntity];
[request setRelationshipKeyPathsForPrefetching:
    [NSArray arrayWithObject:@"department"]];
```

In OS X v10.4, you create a fetch request to fetch just those instances of the destination entity that are related to the source objects you just retrieved, this reduces the number of fetches to two (the minimum). How (or whether) you implement the pre-fetch depends on the cardinality of the relationship.

- If the inverse relationship is a to-one, you can use a predicate with the format, `@"%K IN %@"` where the first argument is the key name for the inverse relationship, and the second argument an array of the original objects.
- If the inverse relationship is a to-many, you first collect the object IDs from the faults you care about (being careful not touch other attributes). You then create a predicate with the format, `@"SELF IN %@"`, where the argument is the array of object IDs.
- If the relationship is a many-to-many, pre-fetching is not recommended.

You could implement pre-fetching for the department relationship in the previous example as follows.

```
NSEntityDescription *deptEntity = [NSEntityDescription entityForName:@"Department"
    inManagedObjectContext:moc];
NSArray *deptIDs = [fetchedEmployees valueForKeyPath:@"department.objectID"];
NSPredicate *deptsPredicate = [NSPredicate predicateWithFormat:
    @"SELF in %@", deptIDs];
NSFetchRequest *deptFetch = [[NSFetchRequest alloc] init];
[deptFetch setEntity:deptEntity];
[deptFetch setPredicate:deptsPredicate];
```



```
// Execute the fetch.
```

If you know something about how the data will be accessed or presented, you can further refine the fetch predicate to reduce the number of objects fetched. Note, though, that this technique can be fragile—if the application changes and needs a different set of data, then you can end up pre-fetching the wrong objects.

For more about faulting, and in particular the meaning of the value returned from `isFault`, see [“Faulting and Uniquing”](#) (page 110).

Reducing Memory Overhead

It is sometimes the case that you want to use managed objects on a temporary basis, for example to calculate an average value for a particular attribute. This causes your object graph, and memory consumption, to grow. You can reduce the memory overhead by re-faulting individual managed objects that you no longer need, or you can reset a managed object context to clear an entire object graph. You can also use patterns that apply to Cocoa programming in general.

- You can re-fault an individual managed object using `NSManagedObjectContext`’s `refreshObject:mergeChanges:` method. This has the effect of clearing its in-memory property values thereby reducing its memory overhead. (Note that this is not the same as setting the property values to `nil`—the values will be retrieved on demand if the fault is fired—see [“Faulting and Uniquing”](#) (page 110).)
- When you create a fetch request you can set `includesPropertyValues` to `NO` to reduce memory overhead by avoiding creation of objects to represent the property values. You should typically only do so, however, if you are sure that either you will not need the actual property data or you already have the information in the row cache, otherwise you will incur multiple trips to the persistent store.
- You can use the `reset` method of `NSManagedObjectContext` to remove all managed objects associated with a context and “start over” as if you’d just created it. Note that any managed object associated with that context will be invalidated, and so you will need to discard any references to and re-fetch any objects associated with that context in which you are still interested.
- If you iterate over a lot of objects, you may need to use local autorelease pool blocks to ensure temporary objects are deallocated as soon as possible.
- If you do not intend to use Core Data’s undo functionality, you can reduce your application’s resource requirements by setting the context’s undo manager to `nil`. This may be especially beneficial for background worker threads, as well as for large import or batch operations.

- Finally, Core Data does not by default keep strong references to managed objects (unless they have unsaved changes). If you have lots of objects in memory, you should determine the owning references. Managed objects maintain strong references to each other through relationships, which can easily create strong reference cycles. You can break cycles by re-faulting objects (again by using the `refreshObject:mergeChanges:` method of `NSManagedObjectContext`).

Large Data Objects (BLOBs)

If your application uses large BLOBs ("Binary Large Objects" such as image and sound data), you need to take care to minimize overheads. The exact definition of "small", "modest", and "large" is fluid and depends on an application's usage. A loose rule of thumb is that objects in the order of kilobytes in size are of a "modest" sized and those in the order of megabytes in size are "large" sized. Some developers have achieved good performance with 10MB BLOBs in a database. On the other hand, if an application has millions of rows in a table, even 128 bytes might be a "modest" sized CLOB (Character Large Object) that needs to be normalized into a separate table.

In general, if you *need* to store BLOBs in a persistent store, you should use an SQLite store. The XML and binary stores require that the whole object graph reside in memory, and store writes are atomic (see "[Persistent Store Features](#)" (page 130)) which means that they do not efficiently deal with large data objects. SQLite can scale to handle extremely large databases. Properly used, SQLite provides good performance for databases up to 100GB, and a single row can hold up to 1GB (although of course reading 1GB of data into memory is an expensive operation no matter how efficient the repository).

A BLOB often represents an attribute of an entity—for example, a photograph might be an attribute of an Employee entity. For small to modest sized BLOBs (and CLOBs), you should create a separate entity for the data and create a to-one relationship in place of the attribute. For example, you might create Employee and Photograph entities with a one-to-one relationship between them, where the relationship from Employee to Photograph replaces the Employee's photograph attribute. This pattern maximizes the benefits of object faulting (see "[Faulting and Uniquing](#)" (page 110)). Any given photograph is only retrieved if it is actually needed (if the relationship is traversed).

It is better, however, if you are able to store BLOBs as resources on the filesystem, and to maintain links (such as URLs or paths) to those resources. You can then load a BLOB as and when necessary.

Analyzing Performance

Analyzing Fetch Behavior with SQLite

With OS X version 10.4.3 and later, you can use the user default `com.apple.CoreData.SQLiteDebug` to log to `stderr` the actual SQL sent to SQLite. (Note that user default names are case sensitive.) For example, you can pass the following as an argument to the application:

```
-com.apple.CoreData.SQLiteDebug 1
```

Higher levels of debug numbers produce more information, although this is likely to be of diminishing utility.

The information the output provides can be useful when debugging performance problems—in particular it may tell you when Core Data is performing a large number of small fetches (such as when firing faults individually). The output differentiates between fetches that you execute using a fetch request and fetches that are performed automatically to realize faults.

Instruments

With OS X version 10.5 and later, you can use the Instruments application (by default in `/Developer/Applications/`) to analyze the behavior of your application. There are several Instruments probes specific to Core Data:

- **Core Data Fetches**
Records invocations of `executeFetchRequest:error:`, providing information about the entity against which the request was made, the number of objects returned, and the time taken for the fetch.
- **Core Data Saves**
Records invocations of `save:` and the time taken to do the save.
- **Core Data Faults**
Records information about object and relationship fault firing. For object faults, records the object being faulted; for relationship faults, records the source object and the relationship being fired. In both cases, records the time taken to fire the fault.
- **Core Data Cache Misses**
Traces fault behavior that specifically results in filesystem activity—indicating that a fault was fired for which no data was available—and records the time taken to retrieve the data.

All the instruments provide a stack trace for each event so that you can see what caused it to happen.

When analyzing your application, you should of course also take into account factors not directly related to Core Data, such as overall memory footprint, object allocations, use and abuse of other API such as the key-value technologies and so on.

Troubleshooting Core Data

This article outlines some of the common issues encountered in applications that use Core Data and provides clues as to correcting the problem.

When troubleshooting Core Data-based applications, it is important to consider that Core Data provides functionality that builds on top of functionality provided by other parts of Cocoa. When attempting to diagnose a problem with an application that uses Core Data, you should take care to distinguish between issues that are specific to Core Data and those that arise because of an error with another framework or to an implementation or architectural pattern. Poor performance, for example, may not be due to Core Data per se, but instead are due to a failure to observe standard Cocoa techniques of memory management or resource conservation; or if a user interface does not update properly, this may be due to an error in how you have configured Cocoa bindings.

Object Life-Cycle Problems

Merge errors

Problem: You see the error message, "Could not merge changes".

Cause: Two different managed object contexts tried to change the same data. This is also known as an optimistic locking failure.

Remedy: Either set a merge policy on the context, or manually (programmatically) resolve the failure. You can retrieve the currently committed values for an object using `committedValuesForKeys:`, and you can re-fault the object (so that when it is next accessed its data values are retrieved from its persistent store) using `refreshObject:mergeChanges:`.

Assigning a managed object to a different store

Problem: You see an exception that looks similar to this example.

```
<NSInvalidArgumentException> [MyM0 0x3036b0>_assignObject:toPersistentStore:]:  
Can't reassign an object to a different store once it has been saved.
```

Cause: The object you are trying to assign to a store has already been assigned and saved to a different store.

Remedy: To move an object from one store to another, you must create a new instance, copy the information from the old object, save it to the appropriate store, and then delete the old instance.

Fault cannot be fulfilled

Problem: You see the error message, "Core Data could not fulfill a fault".

Cause: The corresponding object's underlying data has been deleted from the persistent store.

Remedy: You should discard this object.

This problem can occur in at least two situations:

First:

- Start with a strong reference to a managed object.
- Delete the managed object via the managed object context.
- Save changes on the object context.

At this point, the deleted object has been turned into a fault. It isn't destroyed because doing so would violate the rules of memory management.

- Try to retrieve an attribute or relationship from the previously retained reference.

Core Data will try to fault the faulted managed object but will fail to do so because the object has been deleted from the store. That is, there is no longer an object with the same global ID in the store.

Second:

- Delete an object from a managed object context.
- Fail to break all relationships from other objects to that object.
- Save changes.

At this point, if you try to fire the relationship from some other object to that object, it may fail (this depends on the details of the configuration of the relationship as that affects how the relationship is stored).

The delete rules for relationships affect relationships only from the source object to other objects (including inverses). Without potentially fetching large numbers of objects, possibly without reason, there is no way for Core Data to efficiently clean up the relationships to the object.

Keep in mind that a Core Data object graph is directional. That is, a relationship has a source and a destination. Following a source to a destination does not necessarily mean that there is an inverse relationship. So, in that sense, you need to ensure that you are properly maintaining the object graph across deletes.

In practice, a well-designed object graph does not require much manual post-deletion clean up. If you consider that most object graphs have "entry points" that in effect act as a root node for navigating the graph and that most insertion and deletion events are rooted at those nodes just like fetches, then delete rules take care of most of the work for you. Similarly, since smart groups and other "casual" relationships are generally best implemented with fetched properties, various ancillary collections of entry points into the object graph generally do not need to be maintained across deletes because fetched relationships have no notion of permanence when it comes to objects found via the fetched relationship.

Managed object invalidated

Problem: You see an exception that looks similar to this example:

```
<NSObjectInaccessibleException> [MyMO 0x3036b0>_assignObject:toPersistentStore:]:  
The NSManagedObject with ID:#### has been invalidated.
```

Cause: Either you have removed the store for the fault you are attempting to fire, or the managed object's context has been sent a reset message.

Remedy: You should discard this object. If you add the store again, you can try to fetch the object again.

Class is not key-value coding compliant

Problem: You see an exception that looks similar to the following example.

```
<NSUnknownKeyException> [MyMO 0x3036b0> valueForKey:]:  
this class is not key value coding-compliant for the key randomKey.
```

Cause: Either you used an incorrect key, or you initialized your managed object with `init` instead of `initWithEntity:inManagedObjectContext:`.

Remedy: Use a valid key (check the spelling and case carefully—also review the rules for key-value coding compliance in *Key-Value Coding Programming Guide*), or ensure that you use the designated initializer for `NSManagedObject` (see `initWithEntity:insertIntoManagedObjectContext:`).

Entity class does not respond to invocations of custom methods

Problem: You define an entity that uses a custom subclass of `NSManagedObject`, then in code you create an instance of the entity and invoke a custom method, as illustrated in this code fragment:

```
NSManagedObject *entityInstance =  
    [NSEntityDescription insertNewObjectForEntityForName:@"MyEntity"  
        inManagedObjectContext:managedObjectContext];  
[entityInstance setAttribute: newValue];
```

You get a runtime error like this:

```
"2005-05-05 15:44:51.233 MyApp[1234] ***  
-[NSManagedObject setNameOfEntity:]: selector not recognized [self = 0x30e340]"
```

Cause: In the model, you may have misspelled the name of the custom class for the entity.

Remedy: Ensure that the spelling of name of the custom class in the model matches the spelling of the custom class you implement.

Custom accessor methods are not invoked, key dependencies are not obeyed

Problem: You define a custom subclass of `NSManagedObject` for a particular entity and implement custom accessor methods (and perhaps dependent keys). At runtime, the accessor methods are not called and the dependent key is not updated.

Cause: In the model, you did not specify the custom class for the entity.

Remedy: Ensure that the model specifies of name of the custom class for the entity (that is, that it is not `NSManagedObject`).

Problems with Fetching

SQLite store does not work with sorting

Problem: You create a sort descriptor that uses a comparison method defined by `NSSString`, such as the following:


```
NSSortDescriptor *mySortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"lastName" ascending:YES
selector:@selector(localizedCaseInsensitiveCompare:)];
```

You then either use this descriptor with a fetch request or as one of an array controller's sort descriptors. At runtime, you might see an error message that looks similar to the following:

```
NSRunLoop ignoring exception 'unsupported NSSortDescriptor selector:
localizedCaseInsensitiveCompare:' that raised during posting of
delayed perform with target 3e2e42 and selector 'invokeWithTarget:'
```

Cause: Exactly how a fetch request is executed depends on the store—see [“Fetching Managed Objects”](#) (page 61).

Remedy: If you are executing the fetch directly, you should not use Cocoa-based sort operators—instead you should sort the returned array in memory. If you are using an array controller, you may need to subclass `NSArrayController` so you can have it not pass the sort descriptors to the database and instead do the sorting after your data has been fetched.

Problems with Saving

SQLite store takes a "long time" to save

Problem: You are using an SQLite store and notice that it takes longer to save to the SQLite store than it does to save the same data to an XML store.

Cause: This is probably expected behavior. The SQLite store ensures that all data is written correctly to disk—see [“Configuring a SQLite Store’s Save Behavior”](#) (page 133).

Remedy: First determine whether the time taken to save will be noticeable to the user. This is typically likely to be the case only if you configure your application to frequently save automatically—for example, after every edit that the user makes. First, consider changing the store's save behavior (switch off full sync). Then consider saving data only after a set period (for example, every 15 seconds) instead of after every edit. If necessary, consider choosing a different store—for example, the binary store.

Cannot save documents because entity is null

Problem: You have Core Data document-based application that is unable to save. When you try to save the document you get an exception:

```
Exception raised during posting of notification. Ignored. exception: Cannot perform operation since entity with name 'Wxyz' cannot be found
```

Cause: This error is emitted by an instance of `NSObjectController` (or one of its subclasses) that is set in Entity mode but can't access the entity description in the managed object model associated with the entity name specified in Interface Builder. In short, you have a controller in entity mode with an invalid entity name.

Remedy: Select in turn each of your controllers in Interface Builder, and press Command-1 to show the inspector. For each controller, ensure you have a valid entity name in the "Entity Name" field at the top.

Exception generated in `retainedDataForObjectID:withContext:`.

Problem: You add an object to a context. When you try to save the document you get an error that looks like this:

```
[date] My App[2529:4b03] cannot find data for a temporary oid: 0x60797a0  
<<coredata:///MyClass/8BB183A-045-4BE-840F-AF092549FA19><coredata:///MyClass/8BB183A-045-4BE-840F-AF092549FA19>
```

an exception in `-[NSSQLCore retainedDataForObjectID:withContext:]`, and the backtrace looks like:

```
#1  0x9599a6ac in -[NSSQLCore retainedDataForObjectID:withContext:]  
#2  0x95990238 in -[NSPersistentStoreCoordinator(_NSInternalMethods)  
_conflictsWithRowCacheForObject:andStore:]  
#3  0x95990548 in -[NSPersistentStoreCoordinator(_NSInternalMethods)  
_checkRequestForStore:originalRequest:andOptimisticLocking:]  
#4  0x9594e8f0 in -[NSPersistentStoreCoordinator(_NSInternalMethods)  
executeRequest:withContext:]  
#5  0x959617ec in -[NSManagedObjectContext save:]
```

The call to `_conflictsWithRowCacheForObject:` is comparing the object you're trying to save with its last cached version from the database. Basically, it's checking to see if any other code (thread, process, or just a different managed object context) changed this object out from underneath you.

Core Data does not do this check on newly inserted objects because they could not have existed in any other scope. They haven't been written to the database yet.

Cause: You may have forced a newly inserted object to "lose" its inserted status and then changed or deleted it. This could happen if you passed a temporary object ID to `objectWithID:`. You may have passed an inserted object to another managed object context.

Remedy: There are a number of possible remedies, depending on what was the root cause:

- Do not pass an inserted (not yet saved) object to another context. Only objects that have been saved can be passed between contexts.
- Do not invoke `refreshObject:` on a newly-inserted object.
- Do not make a relationship to an object that you never insert into the context.
- Ensure that you use the designated initializer for instances of `NSManagedObject`.

Before you save (frame #6 in the stack trace), the context's `updatedObjects` and `deletedObjects` sets should only have members whose object ID returns `NO` from `isTemporaryID`.

Debugging Fetching

With OS X version 10.4.3 and later, you can use the user default `com.apple.CoreData.SQLDebug` to log to `stderr` the actual SQL sent to SQLite. (Note that user default names are case sensitive.) For example, you can pass the following as an argument to the application:

```
-com.apple.CoreData.SQLDebug 1
```

Higher levels of debug numbers produce more information, although using higher numbers is likely to be of diminishing utility.

The information the output provides can be useful when debugging performance problems—in particular it may tell you when Core Data is performing a large number of small fetches (such as when firing faults individually). Like file I/O, executing many small fetches is expensive compared to executing a single large fetch. For examples of how to correct this situation, see [“Faulting Behavior”](#) (page 142).

Important: Using this information for reverse engineering to facilitate direct access to the SQLite file is *not* supported. It is exclusively a debugging tool.

As this is for debugging, the exact format of the logging is subject to change without notice. You should not, for example, pipe the output into an analysis tool with the expectation that it will work on all OS versions.

Managed Object Models

My application generates the message "+entityForName: could not locate an NSManagedObjectModel"

Problem: The error states clearly the issue—the entity description cannot find a managed object model from which to access the entity information.

Cause: The model may not be included in your application resources. You may be trying to access the model before it has been loaded. The reference to the context may be `nil`.

Remedy: Be sure that the model is included in your application resources and that the corresponding "project target" option in Xcode is selected.

The class method you invoked requires an entity name and context, and it is through the context that the entity gets the model. Basically, it looks like:

```
context ---> coordinator ---> model
```

In general, when working with Core Data and you have problems like these, you should ensure:

- That the managed object context is not `nil`
If you are setting the reference to the context in a nib file, make sure the appropriate outlet or binding is set correctly.
- If you are managing your own Core Data stack, that the managed object context has an associated coordinator (`setPersistentStoreCoordinator:` after allocating)
- That the persistent store coordinator has a valid model

If you are using `NSPersistentDocument`, then the managed object model is instantiated using the `mergedModelFromBundles:` method when the document is initialized.

The documentation also gives you enough information on how to debug and hooks for debugging; there are a handful of methods listed in the “Getting and setting the persistence objects” section of the API reference for `NSPersistentDocument` for either modifying or inspecting the Core Data objects your document is working with. Simply overriding the implementations, calling `super`, and inspecting the returned values would give you more information about what may (or may not) be occurring.

Bindings Integration

Many problems relating to bindings are not specific to Core Data, and are discussed in “Troubleshooting Cocoa Bindings”. This section describes some additional problems that could be caused by the interaction of Core Data and bindings.

Custom relationship set mutator methods are not invoked by an array controller

Problem: You have implemented set mutator methods for a relationship as described in “Custom To-Many Relationship Accessor Methods,” and have bound the `contentSet` binding of an `NSArrayController` instance to a relationship, but the set mutator methods are not invoked when you add objects to and remove objects from the array controller.

Cause: This is a bug.

Remedy: You can work around this by adding `self` to the `contentSet` binding's key path. For example, instead of binding to `[Department Object Controller].selection.employees`, you would bind to `[Department Object Controller].selection.self.employees`.

Cannot access contents of an object controller after a nib is loaded

Problem: You want to perform an operation with the contents of an object controller (an instance of `NSObjectController`, `NSArrayController`, or `NSTreeController`) after a nib file has been loaded, but the controller's content is `nil`.

Cause: The controller's `fetch` is executed as a delayed operation performed after its managed object context is set (by nib loading)—the fetch therefore happens after `awakeFromNib` and `windowControllerDidLoadNib:`.

Remedy: You can execute the `fetch` “manually” with `fetchWithRequest:merge:error:`—see “Core Data and Cocoa Bindings” (page 120).

Cannot create new objects with array controller

Problem: You cannot create new objects using an `NSArrayController`. For example, when you click the button assigned to the `add:` action, you get an error similar to the following:

```
2005-05-05 12:00:)) .000 MyApp[1234] *** NSRunLoop
ignoring exception 'Failed to create new object' that raised
during posting of delayed perform with target 123456
and selector 'invokeWithTarget:'
```

Cause: In your managed object model, you may have specified a custom class for the entity, but you have not implemented the class.

Remedy: Implement the custom class, or specify that the entity is represented by `NSManagedObject`.

A table view bound to an array controller doesn't display the contents of a relationship

Problem: You have a table view bound to an array controller that you want to display the contents of a relationship, but nothing is displayed and you get an error similar to the following:

```
2005-05-27 14:13:39.077 MyApp[1234] *** NSRunLoop ignoring exception
'Cannot create NSArray from object <_NSFaultingMutableSet: 0x3818f0> ()
of class _NSFaultingMutableSet - consider using contentSet
binding instead of contentArray binding' that raised during posting of
delayed perform with target 385350 and selector 'invokeWithTarget:'
```

Cause: You bound the controller's `contentArray` binding to a relationship. Relationships are represented by sets.

Remedy: Bind the controller's `contentSet` binding to the relationship.

A new object is not added to the relationship of the object currently selected in a table view

Problem: You have a table view that displays a collection of instances of an entity. The entity has a relationship to another entity, instances of which are displayed in a second table view. Each table view is managed by an array controller. When you add new instances of the second entity, they are not added to the relationship of the currently-selected instance of the first.

Cause: The two array controllers are not related. There is nothing to tell the second array controller about the first.

Remedy: Bind the second array controller's `contentSet` binding to the key path that specifies the relationship of the selection in the first array controller. For example, if the first array controller manages the Department entity, and the second the Employee entity, then the `contentSet` binding of the second array controller should be `[Department Controller].selection.employees`.

Table view or outline view contents not kept up-to-date when bound to an NSArrayController or NSTreeController object

Problem: You have a table view or outline view that displays a collection of instances of an entity. As new instances of the entity are added and removed, the table view is not kept in sync.

Cause: If the controller's content is an array that you manage yourself, then it is possible you are not modifying the array in a way that is key-value observing compliant.

If the controller's content is fetched automatically, then you have probably not set the controller to "Automatically prepare content."

Alternatively, the controller may not be properly configured.

Remedy: If the controller's content is a collection that you manage yourself, then ensure you modify the collection in a way that is key-value observing compliant—see "Troubleshooting Cocoa Bindings".

If the controller's content is fetched automatically, set the "Automatically prepares content" switch for the controller in the Attributes inspector in Interface Builder (see also `automaticallyPreparesContent`). Doing so means that the controller tracks inserts into and deletions from its managed object context for its entity.

If neither of these is a factor, check to see that the controller is properly configured (for example, that you have set the entity correctly).

Efficiently Importing Data

This article describes how you can efficiently import data into a Core Data application and turn the data into managed objects to save to a persistent store. It discusses some of the fundamental Cocoa patterns you should follow, and patterns that are specific to Core Data.

Cocoa Fundamentals

In common with many other situations, when you use Core Data to import a data file it is important to remember “normal rules” of Cocoa application development apply. If you import a data file that you have to parse in some way, it is likely you will create a large number of temporary objects. These can take up a lot of memory and lead to paging. Just as you would with a non-Core Data application, you can use local autorelease pool blocks to put a bound on how many additional objects reside in memory. For more about the interaction between Core Data and memory management, see [“Reducing Memory Overhead”](#) (page 145).

You should also avoid repeating work unnecessarily. One subtle case lies in creating a predicate containing a variable. If you create the predicate as shown in the following example, you are not only creating a predicate every time through your loop, you are *parsing* one.

```
// Loop over employeeIDs.
for (NSString *anID in employeeIDs) {
    NSString *predicateString = [NSString stringWithFormat:@"employeeID == %@",
anID];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];
}
```

To create a predicate from a formatted string, the framework must parse the string and create instances of predicate and expression objects. If you are using the same form of a predicate many times over but changing the value of one of the constant value expressions on each use, it is more efficient to create a predicate once and then use variable substitution (see “Creating Predicates”). This technique is illustrated in the following example.

```
NSString *predicateString = [NSString stringWithFormat @"employeeID ==
$EMPLOYEE_ID"];
NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];
```



```
NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];

for (NSString *anID in employeeIDs) {
    NSDictionary *variables = @{ @"EMPLOYEE_ID" : anID };
    NSPredicate *localPredicate = [predicate
    predicateWithSubstitutionVariables:variables];
```

Reducing Peak Memory Footprint

If you import a large amount of data into a Core Data application, you should make sure you keep your application's peak memory footprint low by importing the data in batches and purging the Core Data stack between batches. The relevant issues and techniques are discussed in [“Core Data Performance”](#) (page 140) (particularly [“Reducing Memory Overhead”](#) (page 145)) and [“Object Lifetime Management”](#) (page 79), but they're summarized here for convenience.

Importing in batches

First, you should typically create a separate managed object context for the import, and set its undo manager to `nil`. (Contexts are not particularly expensive to create, so if you cache your persistent store coordinator you can use different contexts for different working sets or distinct operations.)

```
NSManagedObjectContext *importContext = [[NSManagedObjectContext alloc] init];
NSPersistentStoreCoordinator *coordinator = <#Get the coordinator#>;
[importContext setPersistentStoreCoordinator:coordinator];
[importContext setUndoManager:nil];
```

(If you have an existing Core Data stack, you can get the persistent store coordinator from another managed object context.) Setting the undo manager to `nil` means that:

1. You don't waste effort recording undo actions for changes (such as insertions) that will not be undone;
2. The undo manager doesn't maintain strong references to changed objects and so prevent them from being deallocated (see [“Change and Undo Management”](#) (page 80)).

You should import data and create corresponding managed objects in batches (the optimum size of the batch will depend on how much data is associated with each record and how low you want to keep the memory footprint). You process batches within an autorelease pool block; at the end of each batch you need to save the managed object context (using `save:`). (Until you save, the context needs to keep strong references to all the pending changes you've made to the inserted objects.)

Dealing with strong reference cycles

Managed objects with relationships nearly always create unreclaimable strong reference cycles. If during the import you create relationships between objects, you need to break the cycles so that the objects can be deallocated when they're no longer needed. To do this, you can either turn the objects into faults, or reset the whole context. For a complete discussion, see ["Breaking Relationship Strong Reference Cycles"](#) (page 80).

Implementing Find-or-Create Efficiently

A common technique when importing data is to follow a "find-or-create" pattern, where you set up some data from which to create a managed object, determine whether the managed object already exists, and create it if it does not.

There are many situations where you may need to find existing objects (objects already saved in a store) for a set of discrete input values. A simple solution is to create a loop, then for each value in turn execute a fetch to determine whether there is a matching persisted object and so on. This pattern does not scale well. If you profile your application with this pattern, you typically find the fetch to be one of the more expensive operations in the loop (compared to just iterating over a collection of items). Even worse, this pattern turns an $O(n)$ problem into an $O(n^2)$ problem.

It is much more efficient—when possible—to create all the managed objects in a single pass, and then fix up any relationships in a second pass. For example, if you import data that you know does not contain any duplicates (say because your initial data set is empty), you can just create managed objects to represent your data and not do any searches at all. Or if you import "flat" data with no relationships, you can create managed objects for the entire set and weed out (delete) any duplicates before save using a single large `IN` predicate.

If you do need to follow a find-or-create pattern—say because you're importing heterogeneous data where relationship information is mixed in with attribute information—you can optimize how you find existing objects by reducing to a minimum the number of fetches you execute. How to accomplish this depends on the amount of reference data you have to work with. If you are importing 100 potential new objects, and only have 2000 in your database, fetching all of the existing and caching them may not represent a significant penalty (especially if you have to perform the operation more than once). However, if you have 100,000 items in your database, the memory pressure of keeping those cached may be prohibitive.

You can use a combination of an `IN` predicate and sorting to reduce your use of Core Data to a single fetch request. Suppose, for example, you want to take a list of employee IDs (as strings) and create `Employee` records for all those not already in the database. Consider this code, where `Employee` is an entity with a `name` attribute, and `listOfIDsAsString` is the list of IDs for which you want to add objects if they do not already exist in a store.

First, separate and sort the IDs (strings) of interest.

```
// get the names to parse in sorted order
NSArray *employeeIDs = [[listOfIDsAsString componentsSeparatedByString:@"\n"]
    sortedArrayUsingSelector: @selector(compare)];
```

Next, create a predicate using `IN` with the array of name strings, and a sort descriptor which ensures the results are returned with the same sorting as the array of name strings. (The `IN` is equivalent to an SQL `IN` operation, where the left-hand side must appear in the collection specified by the right-hand side.)

```
// Create the fetch request to get all Employees matching the IDs.
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:
    [NSEntityDescription entityForName:@"Employee" inManagedObjectContext:aMOC]];
[fetchRequest setPredicate: [NSPredicate predicateWithFormat:@"(employeeID IN %@)",
    employeeIDs]];

// make sure the results are sorted as well
[fetchRequest setSortDescriptors:
    @[[[NSSortDescriptor alloc] initWithKey:@"employeeID" ascending:YES]]];
```

Finally, execute the fetch.

```
NSError *error;
NSArray *employeesMatchingNames = [aMOC executeFetchRequest:fetchRequest
    error:&error];
```

You end up with two sorted arrays—one with the employee IDs passed into the fetch request, and one with the managed objects that matched them. To process them, you walk the sorted lists following these steps:

1. Get the next ID and `Employee`. If the ID doesn't match the `Employee` ID, create a new `Employee` for that ID.
2. Get the next `Employee`: if the IDs match, move to the next ID and `Employee`.

Regardless of how many IDs you pass in, you only execute a single fetch, and the rest is just walking the result set.

The listing below shows the complete code for the example in the previous section.

```
// Get the names to parse in sorted order.
NSArray *employeeIDs = [[listOfIDsAsString componentsSeparatedByString:@"\n"]
    sortedArrayUsingSelector: @selector(compare)];

// create the fetch request to get all Employees matching the IDs
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:
    [NSEntityDescription entityForName:@"Employee" inManagedObjectContext:aMOC]];
[fetchRequest setPredicate: [NSPredicate predicateWithFormat: @"(employeeID IN
%@", employeeIDs)];

// Make sure the results are sorted as well.
[fetchRequest setSortDescriptors:
    @[ [NSSortDescriptor alloc] initWithKey: @"employeeID" ascending:YES] ];
// Execute the fetch.
NSError *error;
NSArray *employeesMatchingNames = [aMOC executeFetchRequest:fetchRequest
    error:&error];
```

Core Data FAQ

This document provides answers to questions frequently asked about Core Data.

Where does a Managed Object Context Come From?

Where a managed object context comes from is entirely application-dependent. In a Cocoa document-based application using `NSPersistentDocument`, the persistent document typically creates the context, and gives you access to it through the `managedObjectContext` method.

In a single-window application, if you create your project using the standard project assistant, the application delegate (the instance of the `AppDelegate` class) again creates the context, and gives you access to it through the `managedObjectContext` method. In this case, however, the code to create the context (and the rest of the Core Data stack) is explicit. It is written for you automatically as part of the template.

Note that you should not use instances of subclasses of `NSController` directly to execute fetches (for example, you should not create an instance of `NSArrayController` specifically to execute a fetch). Controllers are for managing the interaction between your model objects and your human interface. At the model object level, you should just use a managed object context to perform the fetches directly.

How do I initialize a store with default data?

There are two issues here: creating the data, and ensuring the data is imported only once.

There are several ways to create the data.

- You can create a separate persistent store that contains the default data and include the store as an application resource. When you want to use it, you must either copy the whole store to a suitable location, or copy the objects from the defaults store to an existing store.
- For small datasets, you can create the managed objects directly in code.
- You can create a property list—or some other file-based representation—of the data, and store it as an application resource. When you want to use it, you must open the file and parse the representation to create managed objects.

You should not use this technique on iOS, and only if absolutely necessary on OS X. Parsing a file to create a store incurs unnecessary overhead. It is much better to create a Core Data store yourself offline and use it directly in your application.

There are also several ways to ensure that the defaults are imported only once:

- If you are using iOS or creating a non-document-based application for OS X, you can add a check on application launch to determine whether a file exists at the location you specify for the application's store. If it doesn't, you need to import the data. For an iOS-based example, see *CoreDataBooks*.
- If you are creating a document-based application using `NSPersistentDocument`, you initialize the defaults in `initWithType:error:`.

If there is a possibility that the store (hence file) might be created but the data not imported, then you can add a metadata flag to the store. You can check the metadata (using `metadataForPersistentStoreWithURL:error:`) more efficiently than executing a fetch (and it does not require you to hard code any default data values).

How do I use my existing SQLite database with Core Data?

You don't. Although Core Data supports SQLite as one of its persistent store types, the database format is private. You cannot create a SQLite database using native SQLite API and use it directly with Core Data (nor should you manipulate an existing Core Data SQLite store using native SQLite API). If you have an existing SQLite database, you need to import it into a Core Data store (see [“Efficiently Importing Data”](#) (page 160)).

I have a to-many relationship from Entity A to Entity B. How do I fetch the instances of Entity B related to a given instance of Entity A?

You don't. More specifically, there is no need to explicitly *fetch* the destination instances, you simply invoke the appropriate key-value coding or accessor method on the instance of Entity A. If the relationship is called “widgets”, then if you have implemented a custom class with a similarly named accessor method, you simply write:

```
NSSet *asWidgets = [instanceA widgets];
```

Otherwise you use key-value coding:

```
NSMutableDictionary *asWidgets = [instanceA mutableSetValueForKey:@"widgets"];
```

How do I fetch objects in the same order I created them?

Objects in a persistent store are unordered. Typically you should impose order at the controller or view layer, based on an attribute such as creation date. If there is order inherent in your data, you need to explicitly model that.

How do I copy a managed object from one context to another?

First, note that in a strict sense you are not copying the object. You are conceptually creating an additional reference to the same underlying data in the persistent store.

To copy a managed object from one context to another, you can use the object's object ID, as illustrated in the following example.

```
NSManagedObjectID *objectID = [managedObject objectID];  
NSManagedObject *copy = [context2 objectWithID:objectID];
```

I have a key whose value is dependent on values of attributes in a related entity—how do I ensure it is kept up to date as the attribute values are changes and as the relationship is manipulated?

There are many situations in which the value of one property depends on that of one or more other attributes in another entity. If the value of one attribute changes, then the value of the derived property should also be flagged for change. How you ensure that key-value observing notifications are posted for these dependent properties depends on which version of OS X you're using and the cardinality of the relationship.

OS X v10.5 and later for a to-one relationship

If there is a to-one relationship to the related entity, then to trigger notifications automatically you should either override `keyPathsForValuesAffectingValueForKey:` or implement a suitable method that follows the pattern it defines for registering dependent keys.

For example, you could override `keyPathsForValuesAffectingValueForKey:` as shown in the following example:

Core Data FAQ

I have a key whose value is dependent on values of attributes in a related entity—how do I ensure it is kept up to date as the attribute values are changes and as the relationship is manipulated?

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key {

    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];

    if ([key isEqualToString:@"fullNameAndDepartment"]) {

        NSSet *affectingKeys = [NSSet setWithObjects:@"lastName", @"firstName",
                                                    @"department.deptName", nil];

        keyPaths = [keyPaths setByAddingObjectsFromSet:affectingKeys];
    }
    return keyPaths;
}
```

Or, to achieve the same result, you could just implement

`keyPathsForValuesAffectingFullNameAndDepartment` as illustrated in the following example:

```
+ (NSSet *)keyPathsForValuesAffectingFullNameAndDepartment {

    return [NSSet setWithObjects:@"lastName", @"firstName",
                                @"department.deptName", nil];
}
```

To-many relationships

`keyPathsForValuesAffectingValueForKey:` does not allow key-paths that include a to-many relationship. For example, suppose you have an `Department` entity with a to-many relationship (`employees`) to a `Employee`, and `Employee` has a `salary` attribute. You might want the `Department` entity have a `totalSalary` attribute that is dependent upon the salaries of all the `Employees` in the relationship. You can *not* do this with, for example, `keyPathsForValuesAffectingTotalSalary` and returning `employees.salary` as a key.

There are two possible solutions in both situations:

1. You can use key-value observing to register the parent (in this example, `Department`) as an observer of the relevant attribute of all the children (`Employees` in this example). You must add and remove the parent as an observer as child objects are added to and removed from the relationship (see “Registering for Key-Value Observing”). In the `observeValueForKeyPath:ofObject:change:context:` method you update the dependent value in response to changes, as illustrated in the following code fragment:

Core Data FAQ

I have a key whose value is dependent on values of attributes in a related entity—how do I ensure it is kept up to date as the attribute values are changes and as the relationship is manipulated?

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary *)change context:(void *)context {

    if (context == totalSalaryContext) {
        [self updateTotalSalary];
    }
    else
        // Deal with other observations and/or invoke super...
}

- (void)updateTotalSalary {

    [self setTotalSalary:[self valueForKeyPath:@"employees.@sum.salary"]];
}

- (void)setTotalSalary:(NSNumber *)newTotalSalary
{
    if (totalSalary != newTotalSalary) {
        [self willChangeValueForKey:@"totalSalary"];
        totalSalary = newTotalSalary;
        [self didChangeValueForKey:@"totalSalary"];
    }
}

- (NSNumber *)totalSalary {

    return totalSalary;
}
```

2. You can register the parent with the application's notification center as an observer of its managed object context. The parent should respond to relevant change notifications posted by the children in a manner similar to that for key-value observing.

In Xcode's predicate builder, why don't I see any properties for a fetched property predicate?

If you want to create a predicate for a fetched property in the predicate builder in Xcode, but don't see any properties, you have probably not set the destination entity for the fetched property.

How efficient is Core Data?

Throughout the development of Core Data, the engineering team compared the runtime performance of a generic Core Data application with that of a similar application developed without using Core Data. In general, the Core Data implementation performed better. There may nevertheless be opportunities for further optimization, and the team continues to pursue performance aggressively. For a discussion of how you can ensure you use Core Data as efficiently as possible, see [“Core Data Performance”](#) (page 140).

Core Data looks similar to EOF. What are the differences?

Core Data and EOF (the Enterprise Objects Framework that ships with WebObjects) share a common heritage, but have different goals. EOF is a Java-based framework that connects as a client to a database server. Core Data is an Objective-C-based framework designed to support desktop application development. Core Data supports a number of features not supported by EOF, and vice-versa.

Features Supported Only by EOF

EOF allows you to use custom SQL, shared editing contexts, and nested editing contexts. Core Data does not provide the equivalent of an `EOModelGroup`—the `NSManagedObjectModel` class provides methods for merging models from existing models, and for retrieving merged models from bundles.

EOF supports pre-fetching and batch faulting of relationships, in OS X v10.4 Core Data does not. In OS X v10.5, when you create a fetch request, you can use `setRelationshipKeyPathsForPrefetching:` to specify key paths for relationships that should be fetched with the target entity.

Features Supported Only by Core Data

Core Data supports fetched properties; multiple configurations within a managed object model; local stores; store aggregation (the data for a given entity may be spread across multiple stores); customization and localization of property names and validation warnings; and the use of predicates for property validation.

Class Mapping

There are parallels between many of the classes in Core Data and EOF.

- `NSManagedObject` corresponds to `E0GenericRecord`.
- `NSManagedObjectContext` corresponds to `E0EditingContext`.
- `NSManagedObjectModel` corresponds to `E0Model`.
- `NSPersistentStoreCoordinator` corresponds to `E0ObjectStoreCoordinator`.
- `NSEntityDescription`, `NSPropertyDescription`, `NSRelationshipDescription`, and `NSAttributeDescription` correspond to `E0Entity`, `E0Property`, `E0Relationship`, and `E0Attribute` respectively.

Change Management

There is an important behavioral difference between EOF and Core Data with respect to change propagation. In Core Data, peer managed object contexts are *not* "kept in sync" in the same way as editing contexts in EOF. Given two managed object contexts connected to the same persistent store coordinator, and with the "same" managed object in both contexts, if you modify one of the managed objects then save, the other is *not* re-faulted (changes are not propagated from one context to another). If you modify then save the other managed object, then (at least if you use the default merge policy) you will get an optimistic locking failure.

Multi-Threading

The policy for locking a Core Data managed object context in a multithreaded environment is not the same policy as for an editing context in EOF.

OS X Desktop

These questions are only relevant to OS X/desktop.

How do I get the GUI to validate the data entered by the user?

Core Data validates all managed objects when a managed object context is sent a `save:` message. In a Core Data document-based application, this is when the user saves the document. You can have the GUI validate it as the data is being entered by selecting the "Validates Immediately" option for a value binding in the Interface Builder bindings inspector. If you establish the binding programmatically, you supply in the binding options dictionary a value of YES (as an `NSNumber` object) for the key `NSValidatesImmediatelyBindingOption` (see "Binding Options").

For details of how to write custom validation methods, see the subclassing notes for `NSManagedObject`.

When I remove objects from a detail table view managed by an array controller, why are they not removed from the object graph?

If an array controller manages the collection of objects at the destination of a relationship, then by default the remove method simply removes the current selection from the relationship. If you want removed objects to be deleted from the object graph, then you need to enable the “Deletes Objects On Remove” option for the `contentSet` binding.

How do I get undo/redo for free in my non-document-architecture-based app?

In a Core Data document-based application, the standard `NSDocument` undo manager is replaced by the document’s managed object context’s undo manager. In a non-document-based application for desktop OS X, your window’s delegate can supply the managed object context’s undo manager using the `windowWillReturnUndoManager:` delegate method. If your window delegate has an accessor method for the managed object context (as is the case if you use the Core Data Application template), your implementation of `windowWillReturnUndoManager:` might be as follows.

```
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)sender {  
    return [[self managedObjectContext] undoManager];  
}
```

Document Revision History

This table describes the changes to *Core Data Programming Guide*.

Date	Notes
2014-03-10	Corrected Scalar Value Constraints section in Non-Standard Persistent Attributes. Custom structures other than NSPoint, NSSize, NSRect, and NSRange may be key-value coding and key-value observing compliant, and therefore may be supported by Core Data.
2012-09-19	Removed reference to obsolete document.
2011-08-03	Removed references to a legacy document.
2010-11-15	Added a note re creating stores offline in the FAQ.
2009-11-17	Corrected example implementation of scalar value accessor methods; revised Faulting and Uniquing article.
2009-10-19	Incorporated editorial changes.
2009-08-25	Corrected typographical errors.
2009-06-04	Added discussion of using NSEntityDescription to retrieve specific values.
2009-02-26	First release of this document for iOS.
2008-11-19	Enhanced discussion of managing undo operations.
2008-02-08	Enhanced the discussions of legacy data importing and memory management. Added a discussion of many-to-many relationships in “Relationships and Fetched Properties” (page 82).

Document Revision History

Date	Notes
2007-12-11	Corrected typographical errors.
2007-10-31	Updated for OS X v10.5. Made several minor enhancements.
2007-08-23	Enhanced memory management article; noted that NSObject subclasses do not use all accessor methods with mutableSetValueForKey:.
2007-07-16	Enhanced discussion of threading options; added note about constraints of use of relationship accessor methods.
2007-03-15	Noted the file systems supported by the SQLite store.
2007-02-08	Clarified the behavior of entity inheritance in fetching; split "Managed Object Models" into two articles.
2007-01-08	Updated FAQ, "Memory Management Using Core Data", and "Core Data and Cocoa Bindings".
2006-12-05	Added a discussion of faulting and KVO notifications to "Faulting and Uniquing."
2006-11-09	Enhanced discussion of accessing and modifying properties and of creating and initializing managed objects.
2006-10-03	Enhanced the discussion of copying managed objects.
2006-09-05	Enhanced troubleshooting and multi-threading articles; incorporated validation article.
2006-07-24	Made minor revisions to "Persistent Stores."
2006-06-28	Corrected minor typographical errors.
2006-05-23	Added links to sample code and detail to the section on copy and paste.
2006-04-04	Added section on fetch request templates to Managed Object Models. Enhanced description of managed object lifecycle.

Document Revision History

Date	Notes
2006-03-08	Enhanced "Change Management" and "Faulting and Uniquing" articles; clarified meaning of SQLite debugging flag.
2006-02-07	Added notes about SQL logging to "Fetching Managed Objects" and about test-driven development to "Versioning."
2006-01-10	Added a new, preliminary article on threading. Added a new article, "Managed Objects," taken mainly from the NSManagedObject API reference.
2005-12-06	Augmented the articles "Faulting and Uniquing" and "Persistent Stores."
2005-11-09	Added article on importing legacy files.
2005-10-04	Corrected various minor typographical errors.
2005-09-08	Added new articles to describe managed object models and versioning.
2005-08-11	Added articles on memory management and fetching managed objects. Streamlined the introduction to "Managed Object Accessor Methods."
2005-07-07	Corrected various minor typographic errors, made several clarifications. Added article on Troubleshooting.
2005-06-04	Added article on managed object accessor methods. Corrected method listings in "Non-Standard Attributes" article; other minor enhancements.
2005-04-29	Update to include discussion of relationship manipulation, and enhancement to discussion of memory management. Updated for public release of OS X v10.4. Changed title from "Core Data." First public version.

Glossary

attribute A simple property of an entity that is typically not another entity (for example, an Employee object’s first name).

core data stack The ordered collection of objects from a managed object context, through a persistent object store coordinator, to a persistent store or collection of persistent stores. A stack is effectively defined by a persistent store coordinator (see [persistent store coordinator](#))—there is one and only one per stack. Creating a new persistent store coordinator implies creating a new stack.

entity An abstract description of a data-bearing object equivalent to “model” in the Model-View-Controller design pattern. The components of an entity are called attributes, and the references to other models are called relationships. Together, attributes and relationships are known as properties. Entities are to managed objects what Class is to instances of a class, or—using a database analogy—entities are to managed objects what tables are to rows.

fault A placeholder object that represents an object that has not yet been loaded from an external data store. A fault may represent a single object in the case of a to-one relationship, or a collection in the case of a to-many relationship.

faulting Transparent loading of objects on demand from an external data store.

fetch To retrieve data from a persistent store—akin to a database SELECT operation. The result of a fetch is the creation of a collection of managed objects that are registered with the managed object context used to issue the request.

fetch request An instance of `NSFetchRequest` that specifies an entity and optionally a set of constraints, represented by an `NSPredicate` object, and an array of sort descriptors (instances of `NSSortDescriptor`). These are akin to the table name, WHERE clause, and ORDER BY clauses of a database SELECT statement respectively. A fetch request is executed by being sent to a managed object context.

fetch property A property of an entity that is defined by a fetch request. Fetched properties allow a weak, unidirectional relationship. An example is a dynamic iTunes playlist, if expressed as a property of a containing object. Songs don’t “belong” to a particular playlist, especially when they’re on a remote server. The playlist may remain even after the songs have been deleted or the remote server has become inaccessible. (Consider also a Spotlight live query.)

inserting The process of adding a managed object to a managed object context so that the object becomes part of the object graph and will be committed to a persistent store. Typically “insertion” refers only to the initial creation of a managed object. Thereafter, managed objects retrieved from a persistent store (see [persistent store](#)) are considered as being fetched (see [fetch](#)). There is a

special method (`awakeFromInsert`) that is invoked only once during the lifetime of a managed object when it is first inserted into a managed object context (see [managed object context](#)). A managed object must be inserted into a managed object context before it is considered part of the object graph. A managed object context is responsible for observing changes to managed objects (for the purposes of undo support and maintaining the integrity of the object graph), and can only do so if new objects are inserted.

key-value coding A mechanism for accessing an object's properties indirectly.

managed object An object that is an instance of `NSManagedObject` or a subclass of `NSManagedObject`. After creation it should be registered with a managed object context.

managed object context An object that is an instance of `NSManagedObjectContext`. An `NSManagedObjectContext` object represents a single "object space" or scratch pad in an application. Its primary responsibility is to manage a collection of managed objects. These objects form a group of related model objects that represent an internally consistent view of one or more persistent stores. The context is a powerful object with a central role in the life-cycle of managed objects, with responsibilities from life-cycle management (including faulting) to validation, inverse relationship handling, and undo/redo.

managed object model An object that is an instance of `NSManagedObjectModel`. An `NSManagedObjectModel` object describes a schema, a collection of entities (data models) that you use in your application.

object graph A collection of interrelated objects. In Core Data, an object graph is associated with a managed object context. Moreover, when using Core Data, the object graph may be incomplete, with the edges represented by faults (see [fault](#)).

optimistic locking You can consider optimistic locking to be akin to specifying a `WHERE` clause in a database `UPDATE` statement... `WHERE` clause determined by constituents of snapshot(s) corresponding to object(s) being updated.

persistent store A repository in which objects may be stored. A repository is typically a file, which may be XML, binary, or a SQL database. The store format is transparent to the application. Core Data also provides an in-memory store that lasts no longer than the lifetime of a process.

persistent store coordinator An object that is an instance of `NSPersistentStoreCoordinator`. A coordinator associates persistent stores and a configuration of a managed object model and presents a facade to managed object contexts such that a group of persistent stores appears as a single aggregate store.

primitive accessor An accessor method that gets or sets a variable directly, without invoking access or change notification methods (such as `willAccessValueForKey:` and `didChangeValueForKey:`). Primitive accessors are typically used to initialize an object's variables when it is fetched from a persistent store. In this way, any side effects from any custom accessor methods are avoided.

property A component of an entity that is either an attribute or a relationship. Properties are to entities what instance variables are to classes.

refault Turn an object into a fault. The next time it is accessed, its variables may be re-fetched from the relevant persistent store, depending on the caching mechanism.

relationship In one entity, a reference to one instance of another entity (a to-one relationship) or to a collection of instances of another entity (a to-many relationship). For example, an Employee object's manager is an example of a to-one relationship.

snapshot A record of the state of an entry fetched from a persistent store at the time it is fetched. The information in a snapshot is used to support the framework's optimistic locking mechanism. In some persistent stores it is also used when changes are committed back to a data source to update only the attributes that were changed since the last fetch.

transient property A property of an entity that is not saved to a persistent data store, but which is recorded for undo and redo operations in memory.

uniquing Ensuring that an object graph does not have multiple objects representing the same entry in a persistent store. Core Data accomplishes unquing by using the information it maintains in the mapping of each managed object to its corresponding entry in a persistent store.

validation The process of ensuring that a property value is valid—for example, that it is of the correct type, and its value lies within a prescribed range. The Core Data framework provides an infrastructure to allow values to be tested for validity before they can be applied to an object. There are three aspects to validation: model-based validation, attribute validation using custom validation methods, inter-attribute validation (consistency checking) for update, insert, and delete.



Apple Inc.
Copyright © 2004, 2014 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Cocoa, Finder, Instruments, iTunes, Keynote, Mac, Objective-C, OS X, Spotlight, WebObjects, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

iDisk is a service mark of Apple Inc., registered in the U.S. and other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.