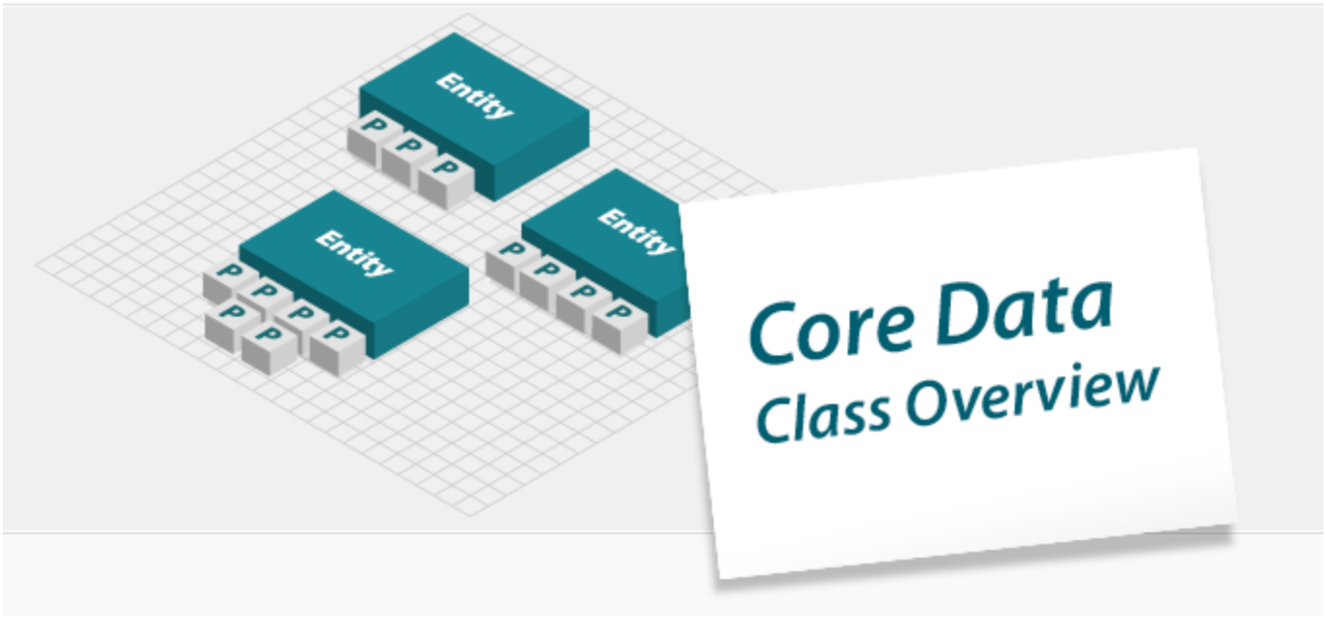


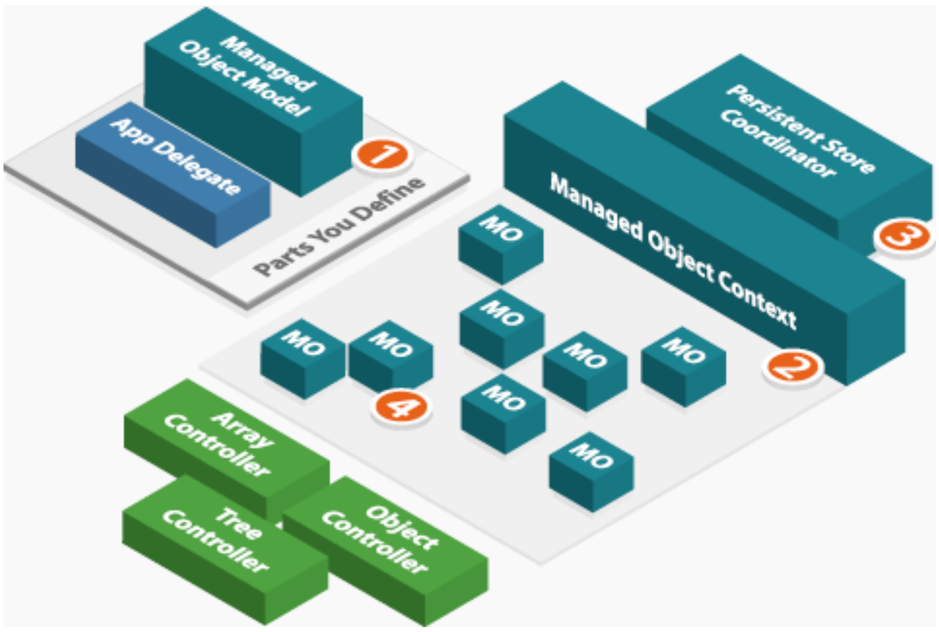
Core Data Class Overview

The Core Data framework provides a lot of new functionality to Cocoa developers, but manages to do so without creating an immense class hierarchy. There are approximately a dozen key classes, which are divided into Model, Runtime and Query classes in this document.

written / illustrated by Scott Stevenson

Core Data Classes in Use

1 of 15



1 The **Managed Object Model** contains a detailed description of an application's data types. The Model contains **Entities**, **Properties** and **Fetch Requests**.

2 The **Managed Object Context** is where the magic really happens. The Context stores and retrieves all user data transparently, provides undo and redo, as well as "revert to saved" functionality. When the Context notices data changes, any view which uses that data is updated via **Cocoa Bindings**.

3 The **Persistent Store Coordinator** handles the low-level aspects of reading and writing data files. It can project multiple files as one a storage location. Most applications do not need to directly interact with the Persistent Store Coordinator.

4 Standard data objects have been replaced by **Managed Objects**, which are connected to the Managed Object Context. You can either use NSManagedObject as is, or provide your own subclass of it for each type of data.

Model Classes

2 of 15

Conventional Cocoa applications usually have a top-down tree of data objects that make up the model. Core Data applications more closely resemble a *network* of objects.

You create a blueprint of this network using Core Data's model classes, which all revolve around the **Managed Object Model**.

Key Model Classes

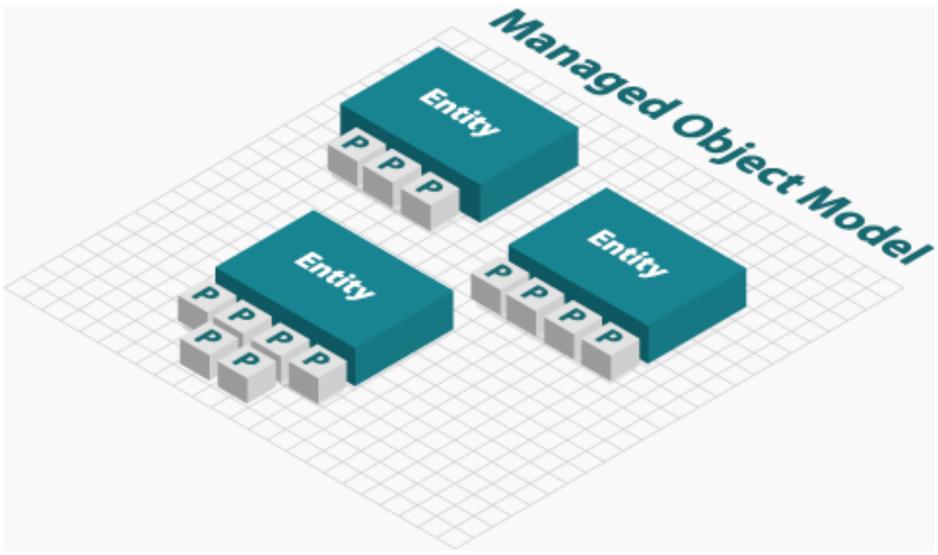
Managed Object Model	`NSManagedObjectModel`	data model
Entity	`NSEntityDescription`	abstract data type
Property	`NSPropertyDescription`	a quality of an Entity
Attribute	`NSAttributeDescription`	simple scalar value
Relationship	`NSRelationshipDescription`	reference to one or more objects
Fetched Property	`NSFetchedPropertyDescription`	criteria-based list of objects

Many of the model classes end in "Description". It's helpful to think of Entities and Properties as just that: descriptions of data types and their relationships to each other.

Managed Object Model

3 of 15

NSManagedObjectModel



The **Managed Object Model** is a detailed outline of an application's data types. A Core Data application has at least one Model, and multiple Models can be merged together at runtime. You'll usually want to use Xcode 2.0's visual modeling tool, but you can also create Models in code.

The Model is made up of **Entities**, which have **Properties**. Entities are connected to each other by **Relationships**. Entities are connected to tables and columns in a SQLite database, or fields in an XML file.

A Model also has rules for its data. For example, you might add a rule to a blogging application which says that every post has an author. Core Data can enforce this rule for you without custom code.

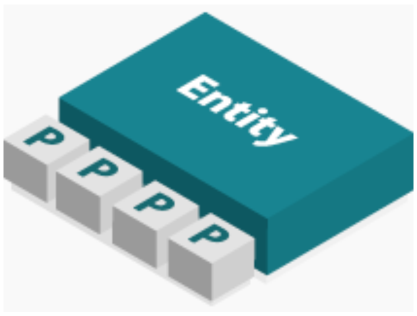
NSManagedObjectModel: Useful Methods	
`-entities`	returns an array of **Entities**
`-entitiesByName`	returns a dictionary of **Entities**, keyed by name
`-setFetchRequestTemplate: forName:`	adds a **Fetch Request** which often contains a **Predicate** with variables in the form of $AUTHORNAME
`-fetchRequestTemplateForName:`	retrieves a **Fetch Request** by name
`-fetchRequestFromTemplateWithName: substitutionVariables:`	retrieves a **Fetch Request** and replaces variables in the predicate with dictionary values

Entity

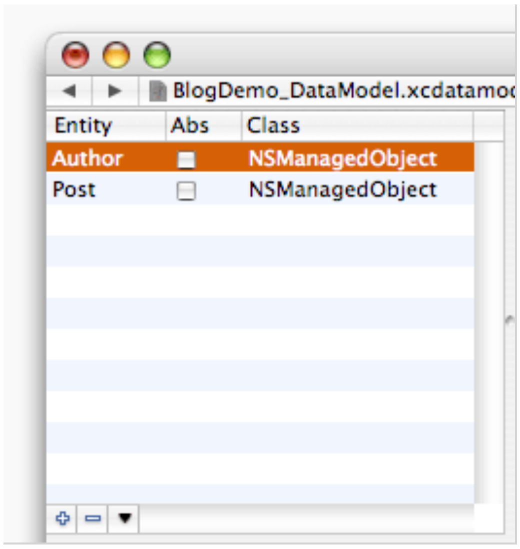
4 of 15

NSEntityDescription

The **Entity** is the most basic building block of a **Managed Object Model**. It's a description of something you want to store, such as an author or a mailbox. The Entity defines *what* will be managed by Core Data, not necessarily *how* it's managed.



If you're a Cocoa programmer, you're used to creating a custom class with instance variables for each data type. With Core Data, you create an Entity with Properties, and *map* it to a class.


BlogDemo_DataModel.xcdatamoc

Entity	Abs	Class
Author	☐	NSManagedObject
Post	☐	NSManagedObject

By default, each Entity is mapped to **NSManagedObject**. It uses the Entity to manage data without custom instances variables and accessors. Entities can be mapped to different classes, but all data classes must inherit from NSManagedObject.

Since the description of data is separate from a class implementation, you can assign multiple Entities to the same class, reducing the overall amount of code in a project. In a sense, **each Entity is a different role played by NSManagedObject**.

An Entity can be **abstract**, in which case it is never directly attached to a managed object. An Entity can also **inherit** Properties from a parent. For example, a SmartMailbox might be a *sub-Entity* of Mailbox.

NSEntityDescription: Useful Methods	
`+insertNewObjectForEntityForName: inManagedObjectContext:`	factory method which creates and returns a new **NSManagedObject** with the given Entity name
`-managedObjectClassName`	returns the class name this Entity is mapped to
`-attributesByName`	returns a dictionary of the Entity's **Attributes**, keyed by name
`-relationshipsByName`	returns a dictionary of the Entity's **Relationships**, keyed by name

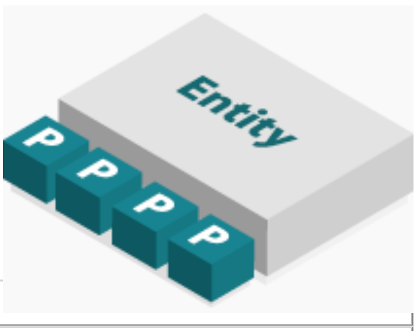
Property

5 of 15

NSPropertyDescription

A **Property** is a quality of an **Entity**. For example, an Author Entity might have **name, email** and **posts** Properties.

Each Property is created as a column in a SQLite store, or a field in an XML file.



Property names are used as KVC keys for Managed Objects. For example, you can set the value of an email Property like this:

Setting a Property Value

```
NSManagedObjectContext * context = [[NSApp delegate] managedObjectContext];
NSManagedObject        * author  = nil;

author = [NSEntityDescription insertNewObjectForEntityForName: @"Author"
                                       inManagedObjectContext: context];

[author setValue: @"nemo@pixar.com" forKey: @"email"];
NSLog (@"The Author's email is: %@", [author valueForKey:@"email"]);
```

A Property can be **optional**. If the user doesn't enter a value for a required Property, the application displays a customizable error. Keep in mind that this also applies if you create or change a Managed Object *in code*.

If a Property is **transient**, Core Data won't store it in the data file. This is useful for calculated values, or session-specific information.

Property Types

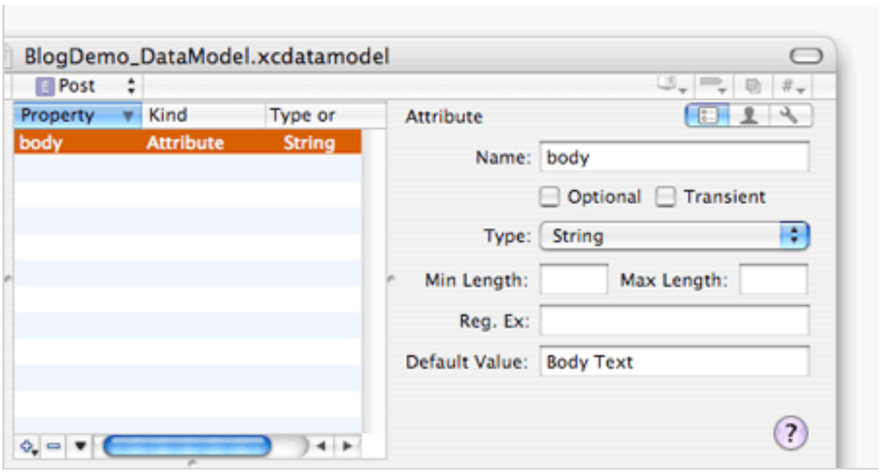
6 of 15

There are three kinds of Properties: **Attribute**, **Relationship**, and **Fetched Property**. Most Core Data applications will likely need to use each in some way.

Attribute

NSAttributeDescription

An **Attribute** stores a basic value, such as an NSString, NSNumber or NSDate. Attributes can have default values, and can be automatically validated using length limits and regular expressions. You have to choose a type for most Attributes. **Only transient attributes can have an "undefined" type**, which means you can use any object as the value.

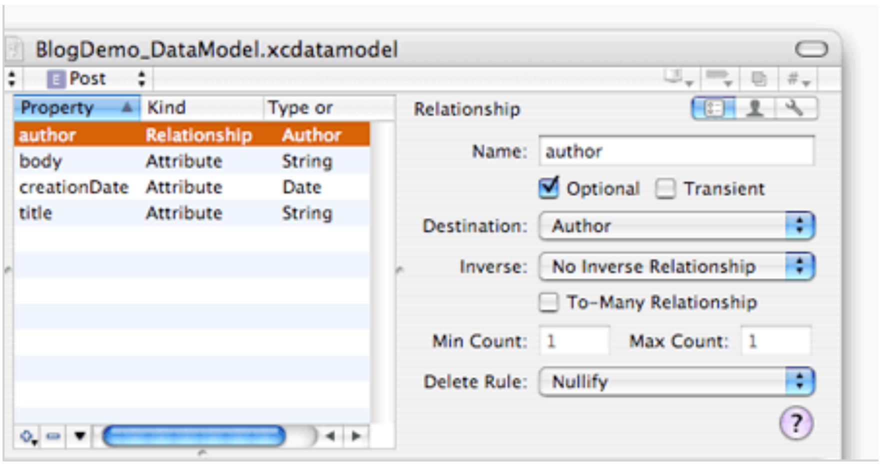


Relationship

NSRelationshipDescription

A **Relationship** is link to one or more specific objects. A **to-one relationship** links to a single object, whereas a **to-many Relationship** links to multiple objects. You can set upper and lower limits on the number of objects in a to-many Relationship.

Most Relationships have a mirrored **inverse relationship**. For example, a Post has a to-one Relationship to its Author, and each Author has an inverse to-many Relationship to all of its Posts.



Property	Kind	Type or
author	Relationship	Author
body	Attribute	String
creationDate	Attribute	Date
title	Attribute	String

Relationship

Name: author

[✓ Optional] Transient

Destination: Author

Inverse: No Inverse Relationship

To-Many Relationship

Min Count: 1 Max Count: 1

Delete Rule: Nullify

Fetched Property

NSFetchedPropertyDescription

Fetched Properties are similar to Relationships, but the objects returned from a Fetched Property are based on a search Predicate. For example, one Fetched Property on Author might be called "**unpublishedPosts**". The Predicate would be formatted as "**published == 0**".

Using iTunes as an example, a smart playlist would be equivalent to a Fetched Property, and a normal playlist would be a Relationship.

Since the list of objects is dynamic, you can't use -setValue:forKey: on a Fetched Property.

Runtime Classes

7 of 15

The runtime classes use the contents of the Managed Object Model to automate basic behavior for an application, like saving, loading and undo.

The Managed Object Context is the single most important runtime class, as it is responsible for handing tasks out to the others.

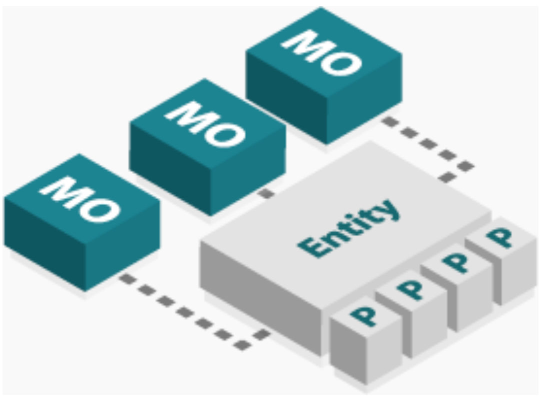
Classes Covered in This Section		
Managed Object	`NSManagedObject`	general-purpose data object
Managed Object Context	`NSManagedObjectContext`	runtime object graph controller
Persistent Store Coordinator	`NSPersistentStoreCoordinator`	data file manager
Persistent Document	`NSPersistentDocument`	subclass of NSDocument which works with Core Data

Managed Object

8 of 15

NSManagedObject

A **Managed Object** represents one record in a data file. Each Managed Object "hosts" an Entity which gives it the properties of a certain kind of data.



For example, an "Author object" might just be an instance of NSManagedObject with the Author Entity attached to it. The stock class is quite capable, so you can often use it as is.

Each Managed Object "belongs" to a Managed Object Context. Each object has a unique ID, represented by an instance of **NSManagedObjectID**. You can use a Managed Object ID to retrieve an object from a Context.

Getting and setting values on Managed Objects is a lot like setting values in an NSDictionary. You just use -valueForKey: and -setValue:forKey:, which fits nicely with Cocoa Bindings and the key-value protocols. Custom subclasses of NSManagedObject should use these KVC messages when possible.

Managed Objects have no actual order in the Context or in Relationships. In fact, the "native" collection class for Core Data is NSSet. You can order the results of Fetch Request using NSSortDescriptors, but the sorting is not saved to the data store.

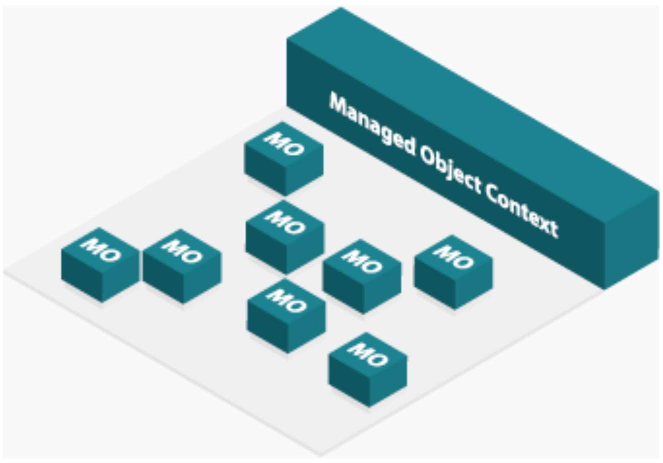
NSManagedObject: Useful Methods	
`-entity`	returns the object's **Entity**
`-objectID`	returns the **Managed Object ID**
`-valueForKey:`	returns the value for the given **Property** name
`-setValue: forKey:`	sets a value for a **Property**

Managed Object Context

9 of 15

NSManagedObjectContext

The **Managed Object Context** is the "director" of a Core Data application. The Context orchestrates or is involved in practically everything that happens with application data at runtime. In fact, the Context does a lot of work the "MyController" class would typically do.



When a new Managed Object is created, it's inserted into the Context. At that point, the Context starts tracking the object, observing any changes that take place. The Context can then perform undo operations, or talk to the **Persistent Store Coordinator** to save changes to a data file.

You usually bind controller classes like **NSArrayController** and **NSTreeController** to a Context. This enables the controllers to dynamically fetch and create Managed Objects, as well as add and remove objects from Relationships.

NSManagedObjectContext: Useful Methods	
`-save:`	writes changed data to the data file
`-objectWithID:`	return an object using the given **Managed Object ID** as a reference

`-deleteObject:`	marks a **Managed Object** for deletion, but object doesn't actually go away until the Context's changes are committed
`-undo`	reverses the last action. Mainly to support Undo menu item. Can also use –redo
`-lock`	controls locking. Useful for multiple threads or for creating transactions. –unlock and –tryLock are also available
`-rollback`	reverts to the contents of the data file
`-reset`	clears out any cached Managed Objects. This must be used when Persistent Stores are added or removed
`-undoManager`	returns the NSUndoManager instance in use by the Context
`-assignObject: toPersistantStore:`	a Context can manage objects from multiple data files at the same time. this links a **Managed Object** to a particular data file
`-executeFetchRequest: error:`	runs a **Fetch Request** and returns any matching objects

Persistent Store Coordinator

10 of 15

NSPersistentStoreCoordinator

Even single-document Core Data applications might store and load data from different locations on disk. The **Persistent Store Coordinator** handles management of actual data files on disk.

It can add and remove data files on the fly, and project multiple data files as a single storage location. In Core Data, individual data files are called *Persistent Stores*.

The Managed Object Context can handle data persistence transparently, so only more advanced applications need interact directly with the Coordinator.

NSPersistentStoreCoordinator: Useful Methods	
`-addPersistentStoreForURL: configuration: URL: options: error:`	brings a Persistent Store (data file) online. Unload stores with –removePersistentStore:error:
`-migratePersistentStore: toURL: options: withType: error:`	equivalent to "save as". The original store reference can't be used after sending this message

-managedObjectIDForURIRepresentation:	converts a URL representation of an object into a true ManagedObjectID instance. Won't work if the store holding the object isn't online
-persistentStoreForURL:	return the Persistent Store at the given path. -URLForPersistentStore: is also available

Persistent Document

11 of 15

NSPersistentDocument

A multi-document Core Data application uses **NSPersistentDocument**, which is a subclass of the traditional NSDocument class. In many cases, you don't need to customize this class at all for your application. The default implementation simply reads the Info.plist file to determine the store type.

NSPersistentDocument: Useful Methods

-managedObjectContext	returns the document's **Managed Object Context**. In a multi-document application, each document has its own context
-managedObjectModel	returns the document's **Managed Object Model**

Query Classes

12 of 15

Adopting Core Data for your application means thinking less about pointers to objects and more about relationships between things. For example, locating a particular object often involves forming a query of some sort.

Key Query Classes

Fetch Request	NSFetchRequest	defines query and result sorting
Predicate	NSPredicate	criteria for a fetch request
Comparison Predicate	NSComparisonPredicate	compares two expressions
Compound Predicate	NSCompoundPredicate	and/or/not criteria
Expression	NSExpression	predicate component

The query classes are inspired by relational databases, but they can used without any regard for the underlying data file format.

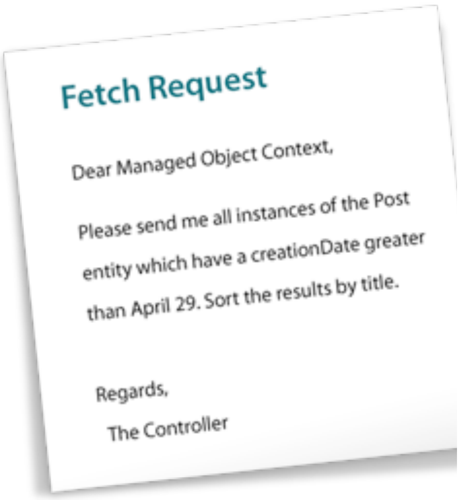
Fetch Requests

13 of 15

A **Fetch Request** is a self-contained query which is sent to the Managed Object Context. The Context evaluates the request and returns the results in the form of an NSArray.

The only thing a Fetch Request must have is an **Entity**. If you send a fetch request to the Context with only an Entity, you'll get an array of all known instances of that Entity.

You can supply a **Predicate** if you want to only return instances that match certain criteria.



Fetch Request

Dear Managed Object Context,

Please send me all instances of the Post entity which have a creationDate greater than April 29. Sort the results by title.

Regards,
The Controller

```
NSManagedObjectContext * context  = [[NSApp delegate] managedObjectContext];
NSManagedObjectModel   * model    = [[NSApp delegate] managedObjectModel];
NSDictionary           * entities = [model entitiesByName];
NSEntityDescription    * entity   = [entities valueForKey:@"Post"];

NSPredicate * predicate;
predicate = [NSPredicate predicateWithFormat:@"creationDate > %@", date];

NSSortDescriptor * sort = [[NSSortDescriptor alloc] initWithKey:@"title"];
NSArray * sortDescriptors = [NSArray arrayWithObject: sort];

NSFetchRequest * fetch = [[NSFetchRequest alloc] init];
[fetch setEntity: entity];
[fetch setPredicate: predicate];
[fetch setSortDescriptors: sortDescriptors];

NSArray * results = [context executeFetchRequest:fetch error:nil];
[sort release];
[fetch release];
```

Fetch Requests can be reused, so keep commonly-used requests in a dictionary instead of creating a new instance each time. This also makes it easier to tweak application behavior later.

NSFetchRequest: Useful Methods	
-setEntity:	specify the type of objects you'd like to fetch (*required*)
-setPredicate:	specify a **Predicate** to constrain the results to a certain set of critiera

-setFetchLimit:	set the maximum number of objects to fetch
-setSortDescriptors:	provide an array of NSSortDescriptors to specify the order the results should be return in
-setAffectedStores:	provide an array of Persistent Stores to retrieve objects from

Predicates

14 of 15

Predicate Classes

Predicate	NSPredicate
Compound	NSCompoundPredicate
Comparison	NSComparisonPredicate

Predicates are used to specify search criteria, and are made up of expressions and nested predicates.

Predicates are used throughout Core Data, but **NSPredicate** is actually defined in the **Foundation** framework, and can be used in a number of different ways. For example, Predicates are used to construct Spotlight queries and to sort arrays.

You can either create Predicates using a specialized query language, or you can assemble trees of NSExpressions and NSPredicate objects. Xcode 2.0's visual modeling tool also includes a Predicate editor.

Predicates understand the concept of substitution variables, so you can create a query at any point and easily fill in values at runtime.

```
// create Predicate from template string

NSString     * format    = @"ANY topic.title == $SEARCHSTRING";
NSPredicate  * predicate =  [NSPredicate predicateWithFormat: format];



// create new Predicate using a dictionary to fill in template values

NSDictionary * valueDict;
valueDict = [NSDictionary dictionaryWithObject: @"tiger"
                                        forKey: @"SEARCHSTRING"];

predicate  = [predicate predicateWithSubstitutionVariables: valueDict];
```

NSPredicate: Useful Methods

+predicateWithFormat:	create a new Predicate from a formatted string
-evaluateWithObject:	test a Predicate against a single object

Specialized Predicates

NSCompoundPredicate

A **Compound Predicate** is the equivalent of AND/OR statements in a SQL database. It effectively combines two nested Predicates into a single master Predicate. For example, a rules-based email filtering system could evaluate multiple rules by inserting multiple Predicates into a single Compound Predicate.

NSCompoundPredicate: Useful Methods	
`+andPredicateWithSubpredicates:`	create a new AND Compound Predicate with an array of sub-Predicates
`+orPredicateWithSubpredicates:`	create a new OR Compound Predicate with an array of sub-Predicates
`+notPredicateWithSubpredicate:`	create a new NOT Compound Predicate with a single sub-Predicate

NSComparisonPredicate

A **Comparison Predicate** evalatues two NSExpressions with an operator, and returns a result. For example, a Comparison Predicate could be used to find all Authors who posted after April 29.

NSComparisonPredicate: Useful Methods	
`+predicateWithFormat:`	create a new Predicate from a formatted string
`-evaluateWithObject:`	test a Predicate against a single object

Expressions

15 of 15

NSExpression

An **Expression** is a building block of a Predicate. It represents a value that is used when the parent Predicate is evaluated. Like NSPredicate, NSExpression is a part of the Foundation framework.

An Expression might contain a constant value like 1.68, or it could reference a predefined function (like avg or max), a variable name, or an object key path. **A Predicate formatted as "creationDate > YESTERDAY" contains two Expressions:** creationDate and YESTERDAY.

In-depth explanations of Predicates and Expressions are beyond the scope of this particular tutorial.

NSManagedObject: Useful Methods

`+expressionForConstantValue:`	returns a new Expression with a constant value
`+expressionForEvaluatedObject:`	returns an Expression with the literal object given
`+expressionForVariable:`	returns an Expression with a variable name like "$SEARCHSTRING"
`+expressionForKeyPath:`	returns an Expression with a KVC key path, such as "post.topic"
`+expressionForFunction:`	returns an Expression with avg, count, max, min or sum

Wrap Up

This tutorial introduced the key classes introduced in the Core Data framework. Take a look at the Build a Core Data Application tutorial if you'd like to put this into action.

As always, let us know what you think about the tutorial.

Further Reading

Build a Core Data Application	a step-by-step walkthrough of a Core Data app
Cocoa Dev Central blog	author's personal site
Core Data Reference	API listing for the Core Data classes
NSPredicate Reference	API listing for NSPredicate