
Core Data Framework Reference

Data Management



2009-03-10



Apple Inc.
© 2004, 2009 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.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Cocoa, iPhone, iTunes, Mac, Mac OS, Objective-C, Spotlight, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Numbers is a trademark of Apple Inc.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

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.

Contents

Introduction **Introduction to Core Data Reference Collection** 11

Part I **Classes** 13

Chapter 1 **NSAtomicStore Class Reference** 15

Overview 15
Tasks 16
Instance Methods 17

Chapter 2 **NSAtomicStoreCacheNode Class Reference** 25

Overview 25
Tasks 25
Instance Methods 26

Chapter 3 **NSAttributeDescription Class Reference** 29

Overview 29
Tasks 30
Instance Methods 31
Constants 34

Chapter 4 **NSEntityDescription Class Reference** 37

Overview 37
Tasks 38
Class Methods 40
Instance Methods 42

Chapter 5 **NSEntityMapping Class Reference** 55

Overview 55
Tasks 55
Instance Methods 57
Constants 66

Chapter 6 **NSEntityMigrationPolicy Class Reference** 69

Overview 69
Tasks 69

Instance Methods 70
Constants 75

Chapter 7 **NSExpressionDescription 77**

Overview 77
Tasks 77
Instance Methods 78

Chapter 8 **NSFetchedPropertyDescription Class Reference 81**

Overview 81
Tasks 82
Instance Methods 82

Chapter 9 **NSFetchedResultsController Class Reference 85**

Overview 85
Tasks 90
Properties 91
Class Methods 94
Instance Methods 94

Chapter 10 **NSFetchRequest Class Reference 99**

Overview 99
Tasks 100
Instance Methods 102
Constants 116

Chapter 11 **NSFetchRequestExpression Class Reference 117**

Overview 117
Tasks 117
Class Methods 118
Instance Methods 118
Constants 119

Chapter 12 **NSManagedObject Class Reference 121**

Overview 121
Tasks 124
Class Methods 127
Instance Methods 128
Constants 150

Chapter 13 **NSManagedObjectContext Class Reference** 153

Overview 153
Tasks 154
Instance Methods 157
Constants 176
Notifications 179

Chapter 14 **NSManagedObjectContext Class Reference** 181

Overview 181
Tasks 181
Instance Methods 182

Chapter 15 **NSManagedObjectContext Class Reference** 185

Overview 185
Tasks 187
Class Methods 188
Instance Methods 191

Chapter 16 **NSMappingModel Class Reference** 201

Overview 201
Tasks 201
Class Methods 202
Instance Methods 203

Chapter 17 **NSMigrationManager Class Reference** 207

Overview 207
Tasks 207
Instance Methods 208

Chapter 18 **NSPersistentStore Class Reference** 219

Overview 219
Tasks 220
Class Methods 221
Instance Methods 222

Chapter 19 **NSPersistentStoreCoordinator Class Reference** 229

Overview 229
Tasks 230
Class Methods 231

Instance Methods 233
Constants 241
Notifications 245

Chapter 20 [NSPropertyDescription Class Reference](#) 247

Overview 247
Tasks 248
Instance Methods 249

Chapter 21 [NSPropertyMapping Class Reference](#) 259

Overview 259
Tasks 259
Instance Methods 260

Chapter 22 [NSRelationshipDescription Class Reference](#) 263

Overview 263
Tasks 264
Instance Methods 265
Constants 270

Part II [Protocols](#) 271

Chapter 23 [NSFetchedResultsControllerDelegate Protocol Reference](#) 273

Overview 273
Tasks 275
Instance Methods 276
Constants 278

Chapter 24 [NSFetchedResultsControllerSectionInfo Protocol Reference](#) 281

Overview 281
Tasks 281
Properties 281

Part III [Constants](#) 283

Chapter 25 [Core Data Constants Reference](#) 285

Overview 285
Constants 285

Document Revision History 295

Tables

Chapter 15 **NSManagedObjectModel Class Reference** 185

Table 15-1 Key and value pattern for the localization dictionary. 198

Introduction to Core Data Reference Collection

Framework	/System/Library/Frameworks/CoreData.framework
Header file directories	/System/Library/Frameworks/CoreData.framework/Headers
Declared in	CoreDataDefines.h CoreDataErrors.h NSAtomicStore.h NSAtomicStoreCacheNode.h NSAttributeDescription.h NSEntityDescription.h NSEntityMapping.h NSEntityMigrationPolicy.h NSExpressionDescription.h NSFetchedRequest.h NSFetchedRequestExpression.h NSFetchedPropertyDescription.h NSFetchedResultsController.h NSManagedObject.h NSManagedObjectContext.h NSManagedObjectID.h NSManagedObjectModel.h NSMappingModel.h NSMigrationManager.h NSPersistentStore.h NSPersistentStoreCoordinator.h NSPropertyDescription.h NSPropertyMapping.h NSRelationshipDescription.h

This collection of documents provides the API reference for the Core Data framework. Core Data provides object graph management and persistence for Foundation and Cocoa applications. For more details, see [Core Data Basics](#).

INTRODUCTION

Introduction to Core Data Reference Collection

Classes

NSAtomicStore Class Reference

Inherits from	NSPersistentStore : NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSAtomicStore.h
Companion guides	Atomic Store Programming Topics Core Data Programming Guide

Overview

`NSAtomicStore` is an abstract superclass that you can subclass to create a Core Data atomic store. It provides default implementations of some utility methods. A custom atomic store allows you to define a custom file format that integrates with a Core Data application.

The atomic stores are all intended to handle data sets that can be expressed in memory. The atomic store API favors simplicity over performance.

Subclassing Notes

Methods to Override

In a subclass of `NSAtomicStore`, you must override the following methods to provide behavior appropriate for your store:

load: (page 19)	Loads the cache nodes for the receiver.
newReferenceObjectForManagedObject: (page 21)	Returns a new reference object for a given managed object.
save: (page 23)	Saves the cache nodes.
updateCacheNode:fromManagedObject: (page 24)	Updates the given cache node using the values in a given managed object.

Note that these are in addition to the methods you must override for a subclass of `NSPersistentStore`:

type (page 227)	Returns the type string of the receiver.
identifier (page 223)	Returns the unique identifier for the receiver.
setIdentifier: (page 226)	Sets the unique identifier for the receiver.
metadata (page 225)	Returns the metadata for the receiver.
metadataForPersistentStoreWithURL:error: (page 221)	Returns the metadata from the persistent store at the given URL.
setMetadata:forPersistentStoreWithURL:error: (page 222)	Sets the metadata for the store at a given URL.

Tasks

Initializing a Store

- [initWithPersistentStoreCoordinator:configurationName:URL:options:](#) (page 18)
Returns an atomic store, initialized with the given arguments.

Loading a Store

- [load:](#) (page 19)
Loads the cache nodes for the receiver.
- [objectIDForEntity:referenceObject:](#) (page 22)
Returns a managed object ID from the reference data for a specified entity.
- [addCacheNodes:](#) (page 17)
Registers a set of cache nodes with the receiver.

Updating Cache Nodes

- [newCacheNodeForManagedObject:](#) (page 20)
Returns a new cache node for a given managed object.
- [newReferenceObjectForManagedObject:](#) (page 21)
Returns a new reference object for a given managed object.
- [updateCacheNode:fromManagedObject:](#) (page 24)
Updates the given cache node using the values in a given managed object.
- [willRemoveCacheNodes:](#) (page 24)
Method invoked before the store removes the given collection of cache nodes.

Saving a Store

- [save:](#) (page 23)
Saves the cache nodes.

Utility Methods

- [cacheNodes](#) (page 18)
Returns the set of cache nodes registered with the receiver.
- [cacheNodeForObjectID:](#) (page 18)
Returns the cache node for a given managed object ID.
- [referenceObjectForObjectID:](#) (page 22)
Returns the reference object for a given managed object ID.

Managing Metadata

- [metadata](#) (page 20)
Returns the metadata for the receiver.
- [setMetadata:](#) (page 23)
Sets the metadata for the receiver.

Instance Methods

addCacheNodes:

Registers a set of cache nodes with the receiver.

```
- (void)addCacheNodes:(NSSet *)cacheNodes
```

Parameters

cacheNodes

A set of cache nodes.

Discussion

You should invoke this method in a subclass during the call to [load:](#) (page 19) to register the loaded information with the store.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

cacheNodeForObjectID:

Returns the cache node for a given managed object ID.

```
- (NSAtomicStoreCacheNode *)cacheNodeForObjectID:(NSManagedObjectID *)objectID
```

Parameters

objectID

A managed object ID.

Return Value

The cache node for *objectID*.

Discussion

This method is normally used by cache nodes to locate related cache nodes (by relationships).

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

cacheNodes

Returns the set of cache nodes registered with the receiver.

```
- (NSSet *)cacheNodes
```

Return Value

The set of cache nodes registered with the receiver.

Discussion

You should modify this collection using [addCacheNodes:](#) (page 17); and [willRemoveCacheNodes:](#) (page 24).

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

initWithPersistentStoreCoordinator:configurationName:URL:options:

Returns an atomic store, initialized with the given arguments.

```
- (id)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
    configurationName:(NSString *)configurationName
    URL:(NSURL *)url
    options:(NSDictionary *)options
```

Parameters

coordinator

A persistent store coordinator.

configurationName

The name of the managed object model configuration to use.

url

The URL of the store to load. This value must not be `nil`.

options

A dictionary containing configuration options.

Return Value

An atomic store, initialized with the given arguments, or `nil` if the store could not be initialized.

Discussion

You typically do not invoke this method yourself; it is invoked by the persistent store coordinator during [addPersistentStoreWithType:configuration:URL:options:error:](#) (page 233), both when a new store is created and when an existing store is opened.

In your implementation, you should check whether a file already exists at *url*; if it does not, then you should either create a file here or ensure that your [load:](#) (page 19) method does not fail if the file does not exist.

Any subclass of `NSAtomicStore` must be able to handle being initialized with a URL pointing to a zero-length file. This serves as an indicator that a new store is to be constructed at the specified location and allows you to securely create reservation files in known locations which can then be passed to Core Data to construct stores. You may choose to create zero-length reservation files during [initWithPersistentStoreCoordinator:configurationName:URL:options:](#) or [load:](#) (page 19). If you do so, you must remove the reservation file if the store is removed from the coordinator before it is saved.

You should ensure that you load metadata during initialization and set it using [setMetadata:](#) (page 23).

Special Considerations

You must invoke `super`'s implementation to ensure that the store is correctly initialized.

Availability

Available in iOS 3.0 and later.

See Also

- [load:](#) (page 19)
- [setMetadata:](#) (page 23)

Declared In

`NSAtomicStore.h`

load:

Loads the cache nodes for the receiver.

```
- (BOOL)load:(NSError **)error
```

Parameters

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the cache nodes were loaded correctly, otherwise NO.

Discussion

You override this method to to load the data from the URL specified in [initWithPersistentStoreCoordinator:configurationName:URL:options:](#) (page 18) and create cache nodes for the represented objects. You must respect the configuration specified for the store, as well as the options.

Any subclass of `NSAtomicStore` must be able to handle being initialized with a URL pointing to a zero-length file. This serves as an indicator that a new store is to be constructed at the specified location and allows you to securely create reservation files in known locations which can then be passed to Core Data to construct stores. You may choose to create zero-length reservation files during [initWithPersistentStoreCoordinator:configurationName:URL:options:](#) (page 18) or `load:`. If you do so, you must remove the reservation file if the store is removed from the coordinator before it is saved.

Special Considerations

You must override this method.

Availability

Available in iOS 3.0 and later.

See Also

- [addCacheNodes:](#) (page 17)

Declared In

`NSAtomicStore.h`

metadata

Returns the metadata for the receiver.

```
- (NSDictionary *)metadata
```

Return Value

The metadata for the receiver.

Discussion

`NSAtomicStore` provides a default dictionary of metadata. This dictionary contains the store type and identifier (`NSStoreTypeKey` and `NSStoreUUIDKey`) as well as store versioning information. Subclasses must ensure that the metadata is saved along with the store data.

See Also

- `metadata` (`NSPersistentStore`)

newCacheNodeForManagedObject:

Returns a new cache node for a given managed object.

```
- (NSAtomicStoreCacheNode *)newCacheNodeForManagedObject:(NSManagedObject *)managedObject
```

Parameters*managedObject*

A managed object.

Return ValueA new cache node for *managedObject*.

Following normal rules for Cocoa memory management (see Memory Management Rules), the returned object has a retain count of 1.

Discussion

This method is invoked by the framework after a save operation on a managed object content, once for each newly-inserted `NSManagedObject` instance.

`NSAtomicStore` provides a default implementation that returns a suitable cache node. You can override this method to take the information from the managed object and return a cache node with a retain count of 1 (the node will be registered by the framework).

Availability

Available in iOS 3.0 and later.

Declared In`NSAtomicStore.h`**newReferenceObjectForManagedObject:**

Returns a new reference object for a given managed object.

```
- (id)newReferenceObjectForManagedObject:(NSManagedObject *)managedObject
```

Parameters*managedObject*

A managed object. At the time this method is called, it has a temporary ID.

Return ValueA new reference object for *managedObject*.

Following normal rules for Cocoa memory management (see Memory Management Rules), the returned object has a retain count of 1.

Discussion

This method is invoked by the framework after a save operation on a managed object context, once for each newly-inserted managed object. The value returned is used to create a permanent ID for the object and must be unique for an instance within its entity's inheritance hierarchy (in this store), and must have a retain count of 1.

Special Considerations

You must override this method.

This method must return a stable (unchanging) value for a given object, otherwise Save As and migration will not work correctly. This means that you can use arbitrary numbers, UUIDs, or other random values only if they are persisted with the raw data. If you cannot save the originally-assigned reference object with the data, then the method must derive the reference object from the managed object's values. For more details, see *Atomic Store Programming Topics*.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

objectIDForEntity:referenceObject:

Returns a managed object ID from the reference data for a specified entity.

```
- (NSManagedObjectID *)objectIDForEntity:(NSEntityDescription *)entity  
referenceObject:(id)data
```

Parameters

entity

An entity description object.

data

Reference data for which the managed object ID is required.

Return Value

The managed object ID from the reference data for a specified entity

Discussion

You use this method to create managed object IDs which are then used to create cache nodes for information being loaded into the store.

Special Considerations

You should not override this method.

Availability

Available in iOS 3.0 and later.

See Also

- [addCacheNodes:](#) (page 17)

Declared In

NSAtomicStore.h

referenceObjectForObjectID:

Returns the reference object for a given managed object ID.

```
- (id)referenceObjectForObjectID:(NSManagedObjectID *)objectID
```

Parameters

objectID

A managed object ID.

Return Value

The reference object for *objectID*.

Discussion

Subclasses should invoke this method to extract the reference data from the object ID for each cache node if the data is to be made persistent.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

save:

Saves the cache nodes.

- (BOOL)save:(NSError **)error

Parameters

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Discussion

You override this method to make persistent the necessary information from the cache nodes to the URL specified for the receiver.

Special Considerations

You must override this method.

Availability

Available in iOS 3.0 and later.

See Also

- [newReferenceObjectForManagedObject:](#) (page 21)
- [updateCacheNode:fromManagedObject:](#) (page 24)
- [willRemoveCacheNodes:](#) (page 24)

Declared In

NSAtomicStore.h

setMetadata:

Sets the metadata for the receiver.

- (void)setMetadata:(NSDictionary *)storeMetadata

Parameters

storeMetadata

The metadata for the receiver.

See Also

- [metadata](#) (page 20)

updateCacheNode:fromManagedObject:

Updates the given cache node using the values in a given managed object.

```
- (void)updateCacheNode:(NSAtomicStoreCacheNode *)node  
    fromManagedObject:(NSManagedObject *)managedObject
```

Parameters

node

The cache node to update.

managedObject

The managed object with which to update *node*.

Discussion

This method is invoked by the framework after a save operation on a managed object context, once for each updated `NSManagedObject` instance.

You override this method in a subclass to take the information from *managedObject* and update *node*.

Special Considerations

You must override this method.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStore.h

willRemoveCacheNodes:

Method invoked before the store removes the given collection of cache nodes.

```
- (void)willRemoveCacheNodes:(NSSet *)cacheNodes
```

Parameters

cacheNodes

The set of cache nodes to remove.

Discussion

This method is invoked by the store before the call to [save:](#) (page 23) with the collection of cache nodes marked as deleted by a managed object context. You can override this method to track the nodes which will not be made persistent in the [save:](#) (page 23) method.

You should not invoke this method directly in a subclass.

Availability

Available in iOS 3.0 and later.

See Also

- [save:](#) (page 23)

Declared In

NSAtomicStore.h

NSAtomicStoreCacheNode Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSAtomicStoreCacheNode.h
Companion guide	Core Data Programming Guide

Overview

`NSAtomicStoreCacheNode` is a concrete class to represent basic nodes in a Core Data atomic store.

A node represents a single record in a persistent store.

You can subclass `NSAtomicStoreCacheNode` to provide custom behavior.

Tasks

Designated Initializer

- [initWithObjectID:](#) (page 26)
Returns a cache node for the given managed object ID.

Node Data

- [objectID](#) (page 26)
Returns the managed object ID for the receiver.
- [propertyCache](#) (page 26)
Returns the property cache dictionary for the receiver.
- [setPropertyCache:](#) (page 27)
Sets the property cache dictionary for the receiver.

- [valueForKey:](#) (page 28)
Returns the value for a given key.
- [setValue:forKey:](#) (page 27)
Sets the value for the given key.

Instance Methods

initWithObjectID:

Returns a cache node for the given managed object ID.

```
- (id)initWithObjectID:(NSManagedObjectID *)moid
```

Parameters

moid

A managed object ID.

Return Value

A cache node for the given managed object ID, or `nil` if the node could not be initialized.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStoreCacheNode.h

objectID

Returns the managed object ID for the receiver.

```
- (NSManagedObjectID *)objectID
```

Return Value

The managed object ID for the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

NSAtomicStoreCacheNode.h

propertyCache

Returns the property cache dictionary for the receiver.

```
- (NSMutableDictionary *)propertyCache
```

Return Value

The property cache dictionary for the receiver.

Discussion

This dictionary is used by `valueForKey:` (page 28) and `setValue:forKey:` (page 27) for property values. The default implementation returns `nil` unless the companion `-setPropertyCache:` method is invoked, or `setValue:forKey:` is invoked on the cache node with non-`nil` property values.

Availability

Available in iOS 3.0 and later.

Declared In

`NSAtomicStoreCacheNode.h`

setPropertyCache:

Sets the property cache dictionary for the receiver.

```
- (void)setPropertyCache:(NSMutableDictionary *)propertyCache
```

Parameters

propertyCache

The property cache dictionary for the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

`NSAtomicStoreCacheNode.h`

setValue:forKey:

Sets the value for the given key.

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

Parameters

value

The value for the property identified by *key*.

key

The name of a property.

Discussion

The default implementation forwards the request to the `propertyCache` (page 26) dictionary if *key* matches a property name of the entity for this cache node. If *key* does not represent a property, the standard `setValue:forKey:` implementation is used.

Availability

Available in iOS 3.0 and later.

Declared In

`NSAtomicStoreCacheNode.h`

valueForKey:

Returns the value for a given key.

```
- (id)valueForKey:(NSString *)key
```

Parameters

key

The name of a property.

Return Value

The value for the property named *key*. For an attribute, the return value is an instance of an attribute type supported by Core Data (see [NSAttributeDescription](#)); for a to-one relationship, the return value must be another cache node instance; for a to-many relationship, the return value must be an collection of the related cache nodes.

Discussion

The default implementation forwards the request to the [propertyCache](#) (page 26) dictionary if *key* matches a property name of the entity for the cache node. If *key* does not represent a property, the standard `valueForKey:` implementation is used.

Availability

Available in iOS 3.0 and later.

Declared In

`NSAtomicStoreCacheNode.h`

NSAttributeDescription Class Reference

Inherits from	NSPropertyDescription : NSObject
Conforms to	NSCoding (NSPropertyDescription) NSCopying (NSPropertyDescription) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSAttributeDescription.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial

Overview

The `NSAttributeDescription` class is used to describe attributes of an entity described by an instance of `NSEntityDescription`.

`NSAttributeDescription` inherits from `NSPropertyDescription`, which provides most of the basic behavior. Instances of `NSAttributeDescription` are used to describe attributes, as distinct from relationships. The class adds the ability to specify the attribute type, and to specify a default value. In a managed object model, you must specify the type of all attributes—you can only use the undefined attribute type (`NSUndefinedAttributeType`) for transient attributes.

Editing Attribute Descriptions

Attribute descriptions are editable until they are used by an object graph manager. This allows you to create or modify them dynamically. However, once a description is used (when the managed object model to which it belongs is associated with a persistent store coordinator), it *must not* (indeed cannot) be changed. This is enforced at runtime: any attempt to mutate a model or any of its sub-objects after the model is associated with a persistent store coordinator causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

Note: Default values set for attributes are retained by a managed object model, not copied. This means that attribute values do not have to implement the `NSCopying` protocol, however it also means that you should not modify any objects after they have been set as default values.

Tasks

Getting and Setting Type Information

- `attributeType` (page 31)
Returns the type of the receiver.
- `setAttributeType:` (page 32)
Sets the type of the receiver.
- `attributeValueClassName` (page 31)
Returns the name of the class used to represent the receiver.
- `setAttributeValueClassName:` (page 32)
Sets the name of the class used to represent the receiver.

Getting and Setting the Default Value

- `defaultValue` (page 31)
Returns the default value of the receiver.
- `setDefaultValue:` (page 33)
Sets the default value of the receiver.

Versioning Support

- `versionHash` (page 34)
Returns the version hash for the receiver.

Value Transformers

- `valueTransformerName` (page 34)
Returns the name of the transformer used to transform the attribute value.
- `setValueTransformerName:` (page 33)
Sets the name of the transformer to use to transform the attribute value.

Instance Methods

attributeType

Returns the type of the receiver.

- (NSAttributeType)attributeType

Return Value

The type of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [attributeValueClassName](#) (page 31)
- [setAttributeType:](#) (page 32)

Declared In

NSAttributeDescription.h

attributeValueClassName

Returns the name of the class used to represent the receiver.

- (NSString *)attributeValueClassName

Return Value

The name of the class used to represent the receiver, as a string.

Availability

Available in iOS 3.0 and later.

See Also

- [attributeType](#) (page 31)
- [setAttributeType:](#) (page 32)

Declared In

NSAttributeDescription.h

defaultValue

Returns the default value of the receiver.

- (id)defaultValue

Return Value

The default value of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setDefaultValue:](#) (page 33)

Declared In

NSAttributeDescription.h

setAttributeType:

Sets the type of the receiver.

```
- (void)setAttributeType:(NSAttributeType)type
```

Parameters

type

An `NSAttributeType` constant that specifies the type for the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [attributeType](#) (page 31)

- [attributeValueClassName](#) (page 31)

Declared In

NSAttributeDescription.h

setAttributeValueClassName:

Sets the name of the class used to represent the receiver.

```
- (void)setAttributeValueClassName:(NSString *)className
```

Parameters

className

The name of the class used to represent the receiver.

Discussion

If you set the value class name, Core Data can check the class of any instance set as the value of an attribute.

Availability

Available in iOS 3.0 and later.

See Also

- [attributeValueClassName](#) (page 31)

Declared In

NSAttributeDescription.h

setDefaultValue:

Sets the default value of the receiver.

```
- (void)setDefaultValue:(id)value
```

Parameters

value

The default value for the receiver.

Discussion

Default values are retained by a managed object model, not copied. This means that attribute values do not have to implement the `NSCopying` protocol, however it also means that you should not modify any objects after they have been set as default values.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [defaultValue](#) (page 31)

Declared In

NSAttributeDescription.h

setValueTransformerName:

Sets the name of the transformer to use to transform the attribute value.

```
- (void)setValueTransformerName:(NSString *)string
```

Parameters

string

The name of the transformer to use to transform the attribute value. The transformer must output an `NSData` object from `transformedValue:` and must allow reverse transformations.

Discussion

The receiver must be an attribute of type `NSTransformedAttributeType`.

If this value is not set, or is set to `nil`, Core Data will default to using a transformer which uses `NSCoding` to archive and unarchive the attribute value.

Availability

Available in iOS 3.0 and later.

See Also

- [valueTransformerName](#) (page 34)

Declared In

NSAttributeDescription.h

valueTransformerName

Returns the name of the transformer used to transform the attribute value.

- (NSString *)valueTransformerName

Return Value

The name of the transformer used to transform the attribute value.

Discussion

The receiver must be an attribute of type `NSTransformedAttributeType`.

Availability

Available in iOS 3.0 and later.

See Also

- [setValueTransformerName:](#) (page 33)

Declared In

NSAttributeDescription.h

versionHash

Returns the version hash for the receiver.

- (NSData *)versionHash

Return Value

The version hash for the receiver.

Discussion

The version hash is used to uniquely identify an attribute based on its configuration. This value includes the [versionHash](#) (page 256) information from `NSPropertyDescription` and the attribute type.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 256) (`NSPropertyDescription`)

Declared In

NSAttributeDescription.h

Constants

NSAttributeType

Defines the possible types of `NSAttributeType` properties. These explicitly distinguish between bit sizes to ensure data store independence.

```
typedef enum {
    NSUndefinedAttributeType = 0,
    NSInteger16AttributeType = 100,
    NSInteger32AttributeType = 200,
    NSInteger64AttributeType = 300,
    NSDecimalAttributeType = 400,
    NSDoubleAttributeType = 500,
    NSFloatAttributeType = 600,
    NSStringAttributeType = 700,
    NSBooleanAttributeType = 800,
    NSDateAttributeType = 900,
    NSBinaryDataAttributeType = 1000,
    NSTransformableAttributeType = 1800,
    NSObjectIDAttributeType = 2000
} NSAttributeType;
```

Constants

NSUndefinedAttributeType

Specifies an undefined attribute type.

NSUndefinedAttributeType is valid for *transient* properties—Core Data will still track the property as an id value and register undo/redo actions, and so on. NSUndefinedAttributeType is illegal for non-transient properties.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSInteger16AttributeType

Specifies a 16-bit signed integer attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSInteger32AttributeType

Specifies a 32-bit signed integer attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSInteger64AttributeType

Specifies a 64-bit signed integer attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSDecimalAttributeType

Specifies an NSDecimalNumber attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSDoubleAttributeType

Specifies a double attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSFloatAttributeType

Specifies a float attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSStringAttributeType

Specifies an NSString attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSBooleanAttributeType

Specifies a Boolean attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSDateAttributeType

Specifies an NSDate attribute.

Times are specified in GMT.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSDataAttributeType

Specifies an NSData attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSTransformableAttributeType

Specifies an attribute that uses a value transformer.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

NSObjectIDAttributeType

Specifies the object ID attribute.

Available in iOS 3.0 and later.

Declared in NSAttributeDescription.h.

Availability

Available in iOS 3.0 and later.

Declared In

NSAttributeDescription.h

NSEntityDescription Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSCopying NSFastEnumeration NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSEntityDescription.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial

Overview

An `NSEntityDescription` object describes an entity in Core Data. Entities are to managed objects what Class is to id, or—to use a database analogy—what tables are to rows. An instance specifies an entity's name, its properties (its attributes and relationships, expressed by instances of `NSAttributeDescription` and `NSRelationshipDescription`) and the class by which it is represented.

An `NSEntityDescription` object is associated with a specific class whose instances are used to represent entries in a persistent store in applications using the Core Data Framework. Minimally, an entity description should have:

- A name
 - The name of a managed object class
- (If an entity has no managed object class name, it defaults to `NSManagedObject`.)

You usually define entities in an `NSManagedObjectContext` using the data modeling tool in Xcode. `NSEntityDescription` objects are primarily used by the Core Data Framework for mapping entries in the persistent store to managed objects in the application. You are not likely to interact with them directly unless you are specifically working with models. Like the other major modeling classes, `NSEntityDescription` provides you with a user dictionary in which you can store any application-specific information related to the entity.

Editing Entity Descriptions

Entity descriptions are editable until they are used by an object graph manager. This allows you to create or modify them dynamically. However, once a description is used (when the managed object model to which it belongs is associated with a persistent store coordinator), it *must not* (indeed cannot) be changed. This is enforced at runtime: any attempt to mutate a model or any of its sub-objects after the model is associated with a persistent store coordinator causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

If you want to create an entity hierarchy, you need to consider the relevant API. You can only set an entity's sub-entities (see [setSubentities:](#) (page 49)), you cannot set an entity's super-entity directly. To set a super-entity for a given entity, you must therefore set an array of subentities on that super entity and include the current entity in that array. So, the entity hierarchy needs to be built top-down.

Using Entity Descriptions in Dictionaries

`NSEntityDescription`'s [copy](#) (page 43) method returns an entity such that

```
[[entity copy] isEqual: entity] == NO
```

Since `NSDictionary` copies its keys and requires that keys both conform to the `NSCopying` protocol and have the property that `copy` returns an object for which `[[object copy] isEqual:object]` is true, you should not use entities as keys in a dictionary. Instead, you should either use the entity's name as the key, or use a map table (`NSMutableDictionary`) with retain callbacks.

Fast Enumeration

In Mac OS v10.5 and later and on iOS, `NSEntityDescription` supports the `NSFastEnumeration` protocol. You can use this to enumerate over an entity's properties, as illustrated in the following example:

```
NSEntityDescription *anEntity = ...;
for (NSPropertyDescription *property in anEntity) {
    // property is each instance of NSPropertyDescription in anEntity in turn
}
```

Tasks

Information About an Entity Description

- [name](#) (page 44)
Returns the entity name of the receiver.
- [setName:](#) (page 48)
Sets the entity name of the receiver.
- [managedObjectModel](#) (page 44)
Returns the managed object model with which the receiver is associated.

- [managedObjectClassName](#) (page 44)
Returns the name of the class that represents the receiver's entity.
- [setManagedObjectClassName:](#) (page 47)
Sets the name of the class that represents the receiver's entity.
- [renamingIdentifier](#) (page 46)
Returns the renaming identifier for the receiver.
- [setRenamingIdentifier:](#) (page 49)
Sets the renaming identifier for the receiver.
- [isAbstract](#) (page 43)
Returns a Boolean value that indicates whether the receiver represents an abstract entity.
- [setAbstract:](#) (page 47)
Sets whether the receiver represents an abstract entity.
- [userInfo](#) (page 52)
Returns the user info dictionary of the receiver.
- [setUserInfo:](#) (page 50)
Sets the user info dictionary of the receiver.

Managing Inheritance

- [subentitiesByName](#) (page 51)
Returns the sub-entities of the receiver in a dictionary.
- [subentities](#) (page 50)
Returns an array containing the sub-entities of the receiver.
- [setSubentities:](#) (page 49)
Sets the subentities of the receiver.
- [superentity](#) (page 51)
Returns the super-entity of the receiver.
- [isKindOfEntity:](#) (page 43)
Returns a Boolean value that indicates whether the receiver is a sub-entity of another given entity.

Working with Properties

- [propertiesByName](#) (page 45)
Returns a dictionary containing the properties of the receiver.
- [properties](#) (page 45)
Returns an array containing the properties of the receiver.
- [setProperty:](#) (page 48)
Sets the properties array of the receiver.
- [attributesByName](#) (page 42)
Returns the attributes of the receiver in a dictionary, where the keys in the dictionary are the attribute names.
- [relationshipsByName](#) (page 46)
Returns the relationships of the receiver in a dictionary, where the keys in the dictionary are the relationship names.

- `relationshipsWithDestinationEntity:` (page 46)
Returns an array containing the relationships of the receiver where the entity description of the relationship is a given entity.

Retrieving an Entity with a Given Name

- + `entityForName:inManagedObjectContext:` (page 40)
Returns the entity with the specified name from the managed object model associated with the specified managed object context's persistent store coordinator.

Creating a New Managed Object

- + `insertNewObjectForEntityForName:inManagedObjectContext:` (page 41)
Creates, configures, and returns an instance of the class for the entity with a given name.

Supporting Versioning

- `versionHash` (page 52)
Returns the version hash for the receiver.
- `versionHashModifier` (page 52)
Returns the version hash modifier for the receiver.
- `setVersionHashModifier:` (page 50)
Sets the version hash modifier for the receiver.

Copying Entity Descriptions

- `copy` (page 43)
Returns a copy of the receiver

Class Methods

entityForName:inManagedObjectContext:

Returns the entity with the specified name from the managed object model associated with the specified managed object context's persistent store coordinator.

```
+ (NSEntityDescription *)entityForName:(NSString *)entityName
  inManagedObjectContext:(NSManagedObjectContext *)context
```

Parameters

entityName

The name of an entity.

context

The managed object context to use.

Return Value

The entity with the specified name from the managed object model associated with *context's* persistent store coordinator.

Discussion

This method is functionally equivalent to the following code example.

```
NSManagedObjectModel *managedObjectModel = [[context persistentStoreCoordinator]
    managedObjectModel];
NSEntityDescription *entity = [[managedObjectModel entitiesByName]
    objectForKey:entityName];
return entity;
```

Availability

Available in iOS 3.0 and later.

See Also

- [entitiesByName](#) (page 191)

Declared In

NSEntityDescription.h

insertNewObjectForEntityForName:inManagedObjectContext:

Creates, configures, and returns an instance of the class for the entity with a given name.

```
+ (id)insertNewObjectForEntityForName:(NSString *)entityName
    inManagedObjectContext:(NSManagedObjectContext *)context
```

Parameters

entityName

The name of an entity.

context

The managed object context to use.

Return Value

A new, autoreleased, fully configured instance of the class for the entity named *entityName*. The instance has its entity description set and is inserted it into *context*.

Discussion

This method makes it easy for you to create instances of a given entity without worrying about the details of managed object creation.

The method is particularly useful on Mac OS X v10.4, as you can use it to create a new managed object without having to know the class used to represent the entity. This is especially beneficial early in the development life-cycle when classes and class names are volatile. The method is conceptually similar to the following code example.

```
NSManagedObjectModel *managedObjectModel =
    [[context persistentStoreCoordinator] managedObjectModel];
NSEntityDescription *entity =
    [[managedObjectModel entitiesByName] objectForKey:entityName];
```

```
NSString *className = [entity managedObjectClassName];
Class entityClass = [[NSBundle mainBundle] classNamed:className];
id newObject = [[entityClass alloc]
                initWithEntity:entity insertIntoManagedObjectContext:context];
return [newObject autorelease];
```

On Mac OS X v10.5 and later and on iOS, you can instead use [initWithEntity:insertIntoManagedObjectContext:](#) (page 135) which returns an instance of the appropriate class for the entity. The equivalent code for Mac OS X v10.5 and on iOS is as follows:

```
NSManagedObjectModel *managedObjectModel =
    [[context persistentStoreCoordinator] managedObjectModel];
NSEntityDescription *entity =
    [[managedObjectModel entitiesByName] objectForKey:entityName];
NSManagedObject *newObject = [[NSManagedObject alloc]
    initWithEntity:entity insertIntoManagedObjectContext:context];
return [newObject autorelease];
```

Important: Despite the presence of the word “`new`” in the method name, in a reference counted environment you are not responsible for releasing the returned object. (“`new`” is not the *first* word in the method name—see Memory Management Rules).

Availability

Available in iOS 3.0 and later.

See Also

- [initWithEntity:insertIntoManagedObjectContext:](#) (page 135)

Declared In

NSEntityDescription.h

Instance Methods

attributesByName

Returns the attributes of the receiver in a dictionary, where the keys in the dictionary are the attribute names.

- (NSDictionary *)attributesByName

Return Value

The attributes of the receiver in a dictionary, where the keys in the dictionary are the attribute names and the values are instances of `NSAttributeDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [propertiesByName](#) (page 45)
- [relationshipsByName](#) (page 46)
- [relationshipsWithDestinationEntity:](#) (page 46)

Declared In

NSEntityDescription.h

copy

Returns a copy of the receiver

- (id)copy

Return Value

A copy of the receiver.

Special Considerations

NSEntityDescription's implementation of copy returns an entity such that:

[[entity copy] isEqual:entity] == NO

You should not, therefore, use an entity as a key in a dictionary (see [“Using Entity Descriptions in Dictionaries”](#) (page 38)).

isAbstract

Returns a Boolean value that indicates whether the receiver represents an abstract entity.

- (BOOL)isAbstract

Return Value

YES if the receiver represents an abstract entity, otherwise NO.

Discussion

An abstract entity might be Shape, with concrete sub-entities such as Rectangle, Triangle, and Circle.

Availability

Available in iOS 3.0 and later.

See Also- [setAbstract:](#) (page 47)**Declared In**

NSEntityDescription.h

isKindOfEntity:

Returns a Boolean value that indicates whether the receiver is a sub-entity of another given entity.

- (BOOL)isKindOfEntity:(NSEntityDescription *)entity

Parameters*entity*

An entity.

Return ValueYES if the receiver is a sub-entity of *entity*, otherwise NO.

Availability

Available in iOS 3.0 and later.

Declared In

NSEntityDescription.h

managedObjectClassName

Returns the name of the class that represents the receiver's entity.

- (NSString *)managedObjectClassName

Return Value

The name of the class that represents the receiver's entity.

Availability

Available in iOS 3.0 and later.

See Also

- [setManagedObjectClassName:](#) (page 47)

Declared In

NSEntityDescription.h

managedObjectModel

Returns the managed object model with which the receiver is associated.

- (NSManagedObjectModel *)managedObjectModel

Return Value

The managed object model with which the receiver is associated.

Availability

Available in iOS 3.0 and later.

See Also

[setEntities:](#) (page 196) (NSManagedObjectModel)

[setEntities:forConfiguration:](#) (page 196): (NSManagedObjectModel)

Declared In

NSEntityDescription.h

name

Returns the entity name of the receiver.

- (NSString *)name

Return Value

The entity name of receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setName:](#) (page 48)

Declared In

NSEntityDescription.h

properties

Returns an array containing the properties of the receiver.

- (NSArray *)properties

Return Value

An array containing the properties of the receiver. The elements in the array are instances of `NSAttributeDescription`, `NSRelationshipDescription`, and/or `NSFetchedPropertyDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [propertiesByName](#) (page 45)
- [setProperty:](#) (page 48)
- [attributesByName](#) (page 42)
- [relationshipsByName](#) (page 46)

Declared In

NSEntityDescription.h

propertiesByName

Returns a dictionary containing the properties of the receiver.

- (NSDictionary *)propertiesByName

Return Value

A dictionary containing the receiver's properties, where the keys in the dictionary are the property names and the values are instances of `NSAttributeDescription` and/or `NSRelationshipDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [attributesByName](#) (page 42)
- [relationshipsByName](#) (page 46)
- [relationshipsWithDestinationEntity:](#) (page 46)

Declared In

NSEntityDescription.h

relationshipsByName

Returns the relationships of the receiver in a dictionary, where the keys in the dictionary are the relationship names.

- (NSDictionary *)relationshipsByName

Return Value

The relationships of the receiver in a dictionary, where the keys in the dictionary are the relationship names and the values are instances of `NSRelationshipDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [attributesByName](#) (page 42)
- [propertiesByName](#) (page 45)
- [relationshipsWithDestinationEntity:](#) (page 46)

Declared In

`NSEntityDescription.h`

relationshipsWithDestinationEntity:

Returns an array containing the relationships of the receiver where the entity description of the relationship is a given entity.

- (NSArray *)relationshipsWithDestinationEntity:(NSEntityDescription *)entity

Parameters

entity

An entity description.

Return Value

An array containing the relationships of the receiver where the entity description of the relationship is *entity*. Elements in the array are instances of `NSRelationshipDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [attributesByName](#) (page 42)
- [propertiesByName](#) (page 45)
- [relationshipsByName](#) (page 46)

Declared In

`NSEntityDescription.h`

renamingIdentifier

Returns the renaming identifier for the receiver.

- (NSString *)renamingIdentifier

Return Value

The renaming identifier for the receiver.

Discussion

The renaming identifier is used to resolve naming conflicts between models. When creating a mapping model between two managed object models, a source entity and a destination entity that share the same identifier indicate that an entity mapping should be configured to migrate from the source to the destination.

If you do not set this value, the identifier will return the entity's name.

Availability

Available in iOS 3.0 and later.

See Also

- [setRenamingIdentifier:](#) (page 49)

Declared In

NSEntityDescription.h

setAbstract:

Sets whether the receiver represents an abstract entity.

```
- (void)setAbstract:(BOOL)flag
```

Parameters

flag

A Boolean value indicating whether the receiver is abstract (YES) or not (NO).

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [isAbstract](#) (page 43)

Declared In

NSEntityDescription.h

setManagedObjectClassName:

Sets the name of the class that represents the receiver's entity.

```
- (void)setManagedObjectClassName:(NSString *)name
```

Parameters

name

The name of the class that represents the receiver's entity.

Discussion

The class specified by *name* must either be, or inherit from, `NSManagedObject`.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [managedObjectClassName](#) (page 44)

Declared In

NSEntityDescription.h

setName:

Sets the entity name of the receiver.

```
- (void)setName:(NSString *)name
```

Parameters

name

The name of the entity the receiver describes.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [name](#) (page 44)

Declared In

NSEntityDescription.h

setProperty:

Sets the properties array of the receiver.

```
- (void)setProperties:(NSArray *)properties
```

Parameters

properties

An array of properties (instances of `NSAttributeDescription`, `NSRelationshipDescription`, and/or `NSFetchedPropertyDescription`).

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [properties](#) (page 45)

- [propertiesByName](#) (page 45)
- [attributesByName](#) (page 42)
- [relationshipsByName](#) (page 46)

Declared In

NSEntityDescription.h

setRenamingIdentifier:

Sets the renaming identifier for the receiver.

```
- (void)setRenamingIdentifier:(NSString *)value
```

Parameters

value

The renaming identifier for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [renamingIdentifier](#) (page 46)

Declared In

NSEntityDescription.h

setSubentities:

Sets the subentities of the receiver.

```
- (void)setSubentities:(NSArray *)array
```

Parameters

array

An array containing sub-entities for the receiver. Objects in the array must be instances of `NSEntityDescription`.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [subentities](#) (page 50)
- [subentitiesByName](#) (page 51)
- [superentity](#) (page 51)

Declared In

NSEntityDescription.h

setUserInfo:

Sets the user info dictionary of the receiver.

```
- (void)setUserInfo:(NSDictionary *)dictionary
```

Parameters

dictionary

A user info dictionary.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [userInfo](#) (page 52)

Declared In

NSEntityDescription.h

setVersionHashModifier:

Sets the version hash modifier for the receiver.

```
- (void)setVersionHashModifier:(NSString *)modifierString
```

Parameters

modifierString

The version hash modifier for the receiver.

Discussion

This value is included in the version hash for the entity. You use it to mark or denote an entity as being a different "version" than another even if all of the values which affect persistence are equal. (Such a difference is important in cases where, for example, the structure of an entity is unchanged but the format or content of data has changed.)

Availability

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 52)

- [versionHashModifier](#) (page 52)

Declared In

NSEntityDescription.h

subentities

Returns an array containing the sub-entities of the receiver.

```
- (NSArray *)subentities
```

Return Value

An array containing the receiver's sub-entities. The sub-entities are instances of `NSEntityDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [setSubentities:](#) (page 49)
- [subentitiesByName](#) (page 51)
- [superentity](#) (page 51)

Declared In

`NSEntityDescription.h`

subentitiesByName

Returns the sub-entities of the receiver in a dictionary.

- (`NSDictionary *`)`subentitiesByName`

Return Value

A dictionary containing the receiver's sub-entities. The keys in the dictionary are the sub-entity names, the corresponding values are instances of `NSEntityDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [setSubentities:](#) (page 49)
- [subentities](#) (page 50)
- [superentity](#) (page 51)

Declared In

`NSEntityDescription.h`

superentity

Returns the super-entity of the receiver.

- (`NSEntityDescription *`)`superentity`

Return Value

The receiver's super-entity. If the receiver has no super-entity, returns `nil`.

Availability

Available in iOS 3.0 and later.

See Also

- [setSubentities:](#) (page 49)
- [subentities](#) (page 50)
- [subentitiesByName](#) (page 51)

Declared In

NSEntityDescription.h

userInfo

Returns the user info dictionary of the receiver.

- (NSDictionary *)userInfo

Return Value

The receiver's user info dictionary.

Availability

Available in iOS 3.0 and later.

See Also

- [setUserInfo:](#) (page 50)

Declared In

NSEntityDescription.h

versionHash

Returns the version hash for the receiver.

- (NSData *)versionHash

Return Value

The version hash for the receiver.

Discussion

The version hash is used to uniquely identify an entity based on the collection and configuration of properties for the entity. The version hash uses only values which affect the persistence of data and the user-defined [versionHashModifier](#) (page 52) value. (The values which affect persistence are: the name of the entity, the version hash of the superentity (if present), if the entity is abstract, and all of the version hashes for the properties.) This value is stored as part of the version information in the metadata for stores which use this entity, as well as a definition of an entity involved in an `NSEntityMapping` object.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHashModifier](#) (page 52)

- [setVersionHashModifier:](#) (page 50)

Declared In

NSEntityDescription.h

versionHashModifier

Returns the version hash modifier for the receiver.

- (NSString *)versionHashModifier

Return Value

The version hash modifier for the receiver.

Discussion

This value is included in the version hash for the entity. See [setVersionHashModifier:](#) (page 50) for a full discussion.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 52)
- [setVersionHashModifier:](#) (page 50)

Declared In

NSEntityDescription.h

NSEntityMapping Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSEntityMapping.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSEntityMapping` specify how to map an entity from a source to a destination managed object model.

Tasks

Managing Source Information

- [sourceEntityName](#) (page 64)
Returns the source entity name for the receiver.
- [setSourceEntityName:](#) (page 62)
Sets the source entity name for the receiver.
- [sourceEntityVersionHash](#) (page 64)
Returns the version hash for the source entity for the receiver.
- [setSourceEntityVersionHash:](#) (page 63)
Sets the version hash for the source entity for the receiver.
- [sourceExpression](#) (page 65)
Returns the source expression for the receiver.
- [setSourceExpression:](#) (page 63)
Sets the source expression for the receiver.

Managing Destination Information

- [destinationEntityName](#) (page 57)
Returns the destination entity name for the receiver.
- [setDestinationEntityName:](#) (page 60)
Sets the destination entity name for the receiver.
- [destinationEntityVersionHash](#) (page 58)
Returns the version hash for the destination entity for the receiver.
- [setDestinationEntityVersionHash:](#) (page 60)
Sets the version hash for the destination entity for the receiver.

Managing Mapping Information

- [name](#) (page 59)
Returns the name of the receiver.
- [setName:](#) (page 62)
Sets the name of the receiver.
- [mappingType](#) (page 58)
Returns the mapping type for the receiver.
- [setMappingType:](#) (page 61)
Sets the mapping type for the receiver.
- [entityMigrationPolicyClassName](#) (page 58)
Returns the class name of the migration policy for the receiver.
- [setEntityMigrationPolicyClassName:](#) (page 61)
Sets the class name of the migration policy for the receiver.
- [attributeMappings](#) (page 57)
Returns the array of attribute mappings for the receiver.
- [setAttributeMappings:](#) (page 60)
Sets the array of attribute mappings for the receiver.
- [relationshipMappings](#) (page 59)
Returns the array of relationship mappings for the receiver.
- [setRelationshipMappings:](#) (page 62)
Sets the array of relationship mappings for the receiver.
- [userInfo](#) (page 65)
Returns the user info dictionary for the receiver.
- [setUserInfo:](#) (page 63)
Sets the user info dictionary for the receiver.

Instance Methods

attributeMappings

Returns the array of attribute mappings for the receiver.

- (NSArray *)attributeMappings

Return Value

The array of attribute mappings for the receiver.

Special Considerations

The order of mappings in the array specifies the order in which the mappings will be processed during a migration.

Availability

Available in iOS 3.0 and later.

See Also

- [setAttributeMappings:](#) (page 60)
- [relationshipMappings](#) (page 59)

Declared In

NSEntityMapping.h

destinationEntityName

Returns the destination entity name for the receiver.

- (NSString *)destinationEntityName

Return Value

The destination entity name for the receiver.

Discussion

Mappings are not directly bound to entity descriptions. You can use the migration manager's [destinationEntityForEntityMapping:](#) (page 210) method to retrieve the entity description for this entity name.

Availability

Available in iOS 3.0 and later.

See Also

- [setDestinationEntityName:](#) (page 60)
- [sourceEntityName](#) (page 64)

Declared In

NSEntityMapping.h

destinationEntityVersionHash

Returns the version hash for the destination entity for the receiver.

- (NSData *)destinationEntityVersionHash

Return Value

The version hash for the destination entity for the receiver.

Discussion

The version hash is calculated by Core Data based on the property values of the entity (see `NSEntityDescription`'s `versionHash` (page 52) method). The `destinationEntityVersionHash` must equal the version hash of the destination entity represented by the mapping.

Availability

Available in iOS 3.0 and later.

See Also

- [setDestinationEntityVersionHash:](#) (page 60)
- [sourceEntityVersionHash](#) (page 64)

Declared In

`NSEntityMapping.h`

entityMigrationPolicyClassName

Returns the class name of the migration policy for the receiver.

- (NSString *)entityMigrationPolicyClassName

Return Value

The class name of the migration policy for the receiver.

Discussion

If not specified, the default migration class name is `NSEntityMigrationPolicy`. You can specify a subclass to provide custom behavior.

Availability

Available in iOS 3.0 and later.

See Also

- [setEntityMigrationPolicyClassName:](#) (page 61)

Declared In

`NSEntityMapping.h`

mappingType

Returns the mapping type for the receiver.

- (NSEntityMappingType)mappingType

Return Value

The mapping type for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setMappingType:](#) (page 61)

Declared In

NSEntityMapping.h

name

Returns the name of the receiver.

```
- (NSString *)name
```

Return Value

The name of the receiver.

Discussion

The name is used only as a means of distinguishing mappings in a model. If not specified, the value defaults to *SOURCE->DESTINATION*.

Availability

Available in iOS 3.0 and later.

See Also

- [setName:](#) (page 62)

Declared In

NSEntityMapping.h

relationshipMappings

Returns the array of relationship mappings for the receiver.

```
- (NSArray *)relationshipMappings
```

Return Value

The array of relationship mappings for the receiver.

Special Considerations

The order of mappings in the array specifies the order in which the mappings will be processed during a migration.

Availability

Available in iOS 3.0 and later.

See Also

- [setRelationshipMappings:](#) (page 62)

- [attributeMappings](#) (page 57)

Declared In

NSEntityMapping.h

setAttributeMappings:

Sets the array of attribute mappings for the receiver.

```
- (void)setAttributeMappings:(NSArray *)mappings
```

Parameters

mappings

The array of attribute mappings for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [attributeMappings](#) (page 57)
- [setRelationshipMappings:](#) (page 62)

Declared In

NSEntityMapping.h

setDestinationEntityName:

Sets the destination entity name for the receiver.

```
- (void)setDestinationEntityName:(NSString *)name
```

Parameters

name

The destination entity name.

Availability

Available in iOS 3.0 and later.

See Also

- [destinationEntityName](#) (page 57)
- [setSourceEntityName:](#) (page 62)

Declared In

NSEntityMapping.h

setDestinationEntityVersionHash:

Sets the version hash for the destination entity for the receiver.

```
- (void)setDestinationEntityVersionHash:(NSData *)vhash
```

Parameters

vhash

The version hash for the destination entity.

Availability

Available in iOS 3.0 and later.

See Also

- [destinationEntityVersionHash](#) (page 58)
- [setSourceEntityVersionHash:](#) (page 63)

Declared In

NSEntityMapping.h

setEntityMigrationPolicyClassName:

Sets the class name of the migration policy for the receiver.

```
- (void)setEntityMigrationPolicyClassName:(NSString *)name
```

Parameters

name

The class name of the migration policy (either `NSEntityMigrationPolicy` or a subclass of `NSEntityMigrationPolicy`).

Availability

Available in iOS 3.0 and later.

See Also

- [entityMigrationPolicyClassName](#) (page 58)

Declared In

NSEntityMapping.h

setMappingType:

Sets the mapping type for the receiver.

```
- (void)setMappingType:(NSEntityMappingType)type
```

Parameters

type

The mapping type for the receiver.

Discussion

If you specify a custom entity mapping type, you must specify a value for the migration policy class name as well (see [setEntityMigrationPolicyClassName:](#) (page 61)).

Availability

Available in iOS 3.0 and later.

See Also

- [mappingType](#) (page 58)

Declared In

NSEntityMapping.h

setName:

Sets the name of the receiver.

```
- (void)setName:(NSString *)name
```

Parameters

name

The name of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [name](#) (page 59)

Declared In

NSEntityMapping.h

setRelationshipMappings:

Sets the array of relationship mappings for the receiver.

```
- (void)setRelationshipMappings:(NSArray *)mappings
```

Parameters

mappings

The array of relationship mappings for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [relationshipMappings](#) (page 59)
- [setAttributeMappings:](#) (page 60)

Declared In

NSEntityMapping.h

setSourceEntityName:

Sets the source entity name for the receiver.

```
- (void)setSourceEntityName:(NSString *)name
```

Parameters

name

The source entity name for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceEntityName](#) (page 64)

- [setDestinationEntityName:](#) (page 60)

Declared In

NSEntityMapping.h

setSourceEntityVersionHash:

Sets the version hash for the source entity for the receiver.

- (void)setSourceEntityVersionHash:(NSData *)*vhash*

Parameters

vhash

The version hash for the source entity for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceEntityVersionHash](#) (page 64)

- [setDestinationEntityVersionHash:](#) (page 60)

Declared In

NSEntityMapping.h

setSourceExpression:

Sets the source expression for the receiver.

- (void)setSourceExpression:(NSEExpression *)*source*

Parameters

source

The source expression for the receiver. The expression can be a fetch request expression, or any other expression which evaluates to a collection.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceExpression](#) (page 65)

Declared In

NSEntityMapping.h

setUserInfo:

Sets the user info dictionary for the receiver.

- (void)setUserInfo:(NSDictionary *)*dict*

Parameters*dict*

The user info dictionary for the receiver.

Discussion

You can set the contents of the dictionary using the appropriate inspector in the Xcode mapping model editor.

Availability

Available in iOS 3.0 and later.

See Also

- [userInfo](#) (page 65)

Declared In

NSEntityMapping.h

sourceEntityName

Returns the source entity name for the receiver.

- (NSString *)sourceEntityName

Return Value

The source entity name for the receiver.

Discussion

Mappings are not directly bound to entity descriptions; you can use the [sourceEntityForEntityMapping:](#) (page 215) method on the migration manager to retrieve the entity description for this entity name.

Availability

Available in iOS 3.0 and later.

See Also

- [setSourceEntityName:](#) (page 62)
- [destinationEntityName](#) (page 57)

Declared In

NSEntityMapping.h

sourceEntityVersionHash

Returns the version hash for the source entity for the receiver.

- (NSData *)sourceEntityVersionHash

Return Value

The version hash for the source entity for the receiver.

Discussion

The version hash is calculated by Core Data based on the property values of the entity (see NSEntityDescription's [versionHash](#) (page 52) method). The `sourceEntityVersionHash` must equal the version hash of the source entity represented by the mapping.

Availability

Available in iOS 3.0 and later.

See Also

- [setSourceEntityVersionHash:](#) (page 63)
- [destinationEntityVersionHash](#) (page 58)

Declared In

NSEntityMapping.h

sourceExpression

Returns the source expression for the receiver.

- (NSExpression *)sourceExpression

Return Value

The source expression. The expression can be a fetch request expression, or any other expression which evaluates to a collection.

Discussion

The source expression is used to obtain the collection of managed objects to process through the mapping.

Availability

Available in iOS 3.0 and later.

See Also

- [setSourceExpression:](#) (page 63)

Declared In

NSEntityMapping.h

userInfo

Returns the user info dictionary for the receiver.

- (NSDictionary *)userInfo

Return Value

The user info dictionary.

Discussion

You can use the info dictionary in any way that might be useful in your migration. You set the contents of the dictionary using [setUserInfo:](#) (page 63) or using the appropriate inspector in the Xcode mapping model editor.

Availability

Available in iOS 3.0 and later.

See Also

- [setUserInfo:](#) (page 63)

Declared In

NSEntityMapping.h

Constants

Entity Mapping Types

These constants specify the types of entity mapping.

```
enum {
    NSUndefinedEntityType      = 0x00,
    NSCustomEntityType         = 0x01,
    NSAddEntityType            = 0x02,
    NSRemoveEntityType         = 0x03,
    NSCopyEntityType           = 0x04,
    NSTransformEntityType      = 0x05
};
```

Constants

NSUndefinedEntityType

Specifies that the developer handles destination instance creation.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

NSCustomEntityType

Specifies a custom mapping.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

NSAddEntityType

Specifies that this is a new entity in the destination model.

Instances of the entity only exist in the destination.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

NSRemoveEntityType

Specifies that this entity is not present in the destination model.

Instances of the entity only exist in the source—source instances are not mapped to destination.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

NSCopyEntityType

Specifies that source instances are migrated as-is.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

NSTransformEntityType

Specifies that entity exists in source and destination and is mapped.

Available in iOS 3.0 and later.

Declared in NSEntityMapping.h.

Declared In

NSEntityMapping.h

NEntityType

Data type used for constants that specify types of entity mapping.

```
typedef NSUInteger NEntityType;
```

Discussion

For possible values, see [“Entity Mapping Types”](#) (page 66).

Availability

Available in iOS 3.0 and later.

Declared In

NSEntityMapping.h

NSEntityMigrationPolicy Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSEntityMigrationPolicy.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSEntityMigrationPolicy` customize the migration process for an entity mapping.

You set the policy for an entity mapping by passing the name of the migration policy class as the argument to `setEntityMigrationPolicyClassName:` (page 61) (typically you specify the name in the Xcode mapping model editor).

Tasks

Customizing Stages of the Mapping Life Cycle

- `beginEntityMapping:manager:error:` (page 70)
Invoked by the migration manager at the start of a given entity mapping.
- `createDestinationInstancesForSourceInstance:entityMapping:manager:error:` (page 70)
Creates the destination instance(s) for a given source instance.
- `endInstanceCreationForEntityMapping:manager:error:` (page 73)
Indicates the end of the creation stage for the specified entity mapping, and the precursor to the next migration stage.
- `createRelationshipsForDestinationInstance:entityMapping:manager:error:` (page 71)
Constructs the relationships between the newly-created destination instances.
- `endRelationshipCreationForEntityMapping:manager:error:` (page 73)
Indicates the end of the relationship creation stage for the specified entity mapping.

- [performCustomValidationForEntityMapping:manager:error:](#) (page 74)
Invoked during the validation stage of the entity migration policy, providing the option of performing custom validation on migrated objects.
- [endEntityMapping:manager:error:](#) (page 72)
Invoked by the migration manager at the end of a given entity mapping.

Instance Methods

beginEntityMapping:manager:error:

Invoked by the migration manager at the start of a given entity mapping.

- (BOOL)beginEntityMapping:(NSEntityMapping *)*mapping*
manager:(NSMigrationManager *)*manager*
error:(NSError **)*error*

Parameters

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an NSError object that describes the problem.

Return Value

YES if the method completes successfully, otherwise NO.

Discussion

This method is the precursor to the creation stage. In a custom class, you can implement this method to set up any state information that will be useful for the duration of the migration.

Availability

Available in iOS 3.0 and later.

See Also

- [createDestinationInstancesForSourceInstance:entityMapping:manager:error:](#) (page 70)
- [endEntityMapping:manager:error:](#) (page 72)

Declared In

NSEntityMigrationPolicy.h

createDestinationInstancesForSourceInstance:entityMapping:manager:error:

Creates the destination instance(s) for a given source instance.

- (BOOL)createDestinationInstancesForSourceInstance:(NSManagedObject *)*sInstance*
entityMapping:(NSEntityMapping *)*mapping*
manager:(NSMigrationManager *)*manager*
error:(NSError **)*error*

Parameters*sInstance*

The source instance for which to create destination instances.

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the method completes successfully, otherwise NO.

Discussion

This method is invoked by the migration manager on each source instance (as specified by the [sourceExpression](#) (page 65) in the mapping) to create the corresponding destination instance(s). It also associates the source and destination instances by calling `NSMigrationManager's associateSourceInstance:withDestinationInstance:forEntityMapping:` (page 208) method.

Special Considerations

If you override this method and do not invoke `super`, you must invoke `NSMigrationManager's associateSourceInstance:withDestinationInstance:forEntityMapping:` (page 208) to associate the source and destination instances as required. .

Availability

Available in iOS 3.0 and later.

See Also

- [beginEntityMapping:manager:error:](#) (page 70)
- [endInstanceCreationForEntityMapping:manager:error:](#) (page 73)

Declared In

`NSEntityMigrationPolicy.h`

createRelationshipsForDestinationInstance:entityMapping:manager:error:

Constructs the relationships between the newly-created destination instances.

```
- (BOOL)createRelationshipsForDestinationInstance:(NSManagedObject *)dInstance
    entityMapping:(NSEntityMapping *)mapping manager:(NSMigrationManager *)manager
    error:(NSError **)error
```

Parameters*dInstance*

The destination instance for which to create relationships.

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the relationships are constructed correctly, otherwise NO.

Discussion

You can use this stage to (re)create relationships between migrated objects—you use the association lookup methods on the `NSMigrationManager` instance to determine the appropriate relationship targets.

Availability

Available in iOS 3.0 and later.

See Also

- [endInstanceCreationForEntityMapping:manager:error:](#) (page 73)
- [endRelationshipCreationForEntityMapping:manager:error:](#) (page 73)

Declared In

`NSEntityMigrationPolicy.h`

endEntityMapping:manager:error:

Invoked by the migration manager at the end of a given entity mapping.

```
- (BOOL)endEntityMapping:(NSEntityMapping *)mapping
    manager:(NSMigrationManager *)manager
    error:(NSError **)error
```

Parameters

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the method completes correctly, otherwise NO.

Discussion

This is the end to the given entity mapping. You can implement this method to perform any clean-up at the end of the migration (from any of the three phases of the mapping).

Availability

Available in iOS 3.0 and later.

See Also

- [performCustomValidationForEntityMapping:manager:error:](#) (page 74)
- [beginEntityMapping:manager:error:](#) (page 70)

Declared In

`NSEntityMigrationPolicy.h`

endInstanceCreationForEntityMapping:manager:error:

Indicates the end of the creation stage for the specified entity mapping, and the precursor to the next migration stage.

```
- (BOOL)endInstanceCreationForEntityMapping:(NSEntityMapping *)mapping
    manager:(NSMigrationManager *)manager
    error:(NSError **)error
```

Parameters

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the relationships are constructed correctly, otherwise NO.

Discussion

You can override this method to clean up state from the creation of destination or to prepare state for the creation of relationships.

Availability

Available in iOS 3.0 and later.

See Also

- [createDestinationInstancesForSourceInstance:entityMapping:manager:error:](#) (page 70)
- [createRelationshipsForDestinationInstance:entityMapping:manager:error:](#) (page 71)

Declared In

`NSEntityMigrationPolicy.h`

endRelationshipCreationForEntityMapping:manager:error:

Indicates the end of the relationship creation stage for the specified entity mapping.

```
- (BOOL)endRelationshipCreationForEntityMapping:(NSEntityMapping *)mapping
    manager:(NSMigrationManager *)manager
    error:(NSError **)error
```

Parameters

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the method completes correctly, otherwise NO.

Discussion

This method is invoked after

[createRelationshipsForDestinationInstance:entityMapping:manager:error:](#) (page 71); you can override it to clean up state from the creation of relationships, or prepare state for custom validation in [performCustomValidationForEntityMapping:manager:error:](#) (page 74).

Availability

Available in iOS 3.0 and later.

See Also

- [createRelationshipsForDestinationInstance:entityMapping:manager:error:](#) (page 71)
- [performCustomValidationForEntityMapping:manager:error:](#) (page 74)

Declared In

NSEntityMigrationPolicy.h

performCustomValidationForEntityMapping:manager:error:

Invoked during the validation stage of the entity migration policy, providing the option of performing custom validation on migrated objects.

```
- (BOOL)performCustomValidationForEntityMapping:(NSEntityMapping *)mapping
      manager:(NSMigrationManager *)manager
      error:(NSError **)error
```

Parameters

mapping

The mapping object in use.

manager

The migration manager performing the migration.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the method completes correctly, otherwise NO.

Discussion

This method is called before the default save validation is performed by the framework.

If you implement this method, you must manually obtain the collection of objects you are interested in validating.

Availability

Available in iOS 3.0 and later.

See Also

- [endRelationshipCreationForEntityMapping:manager:error:](#) (page 73)
- [endEntityMapping:manager:error:](#) (page 72)

Declared In

NSEntityMigrationPolicy.h

Constants

Value Expression Keys

Keys used in value expression right hand sides.

```
NSString *NSMigrationManagerKey;  
NSString *NSMigrationSourceObjectKey;  
NSString *NSMigrationDestinationObjectKey;  
NSString *NSMigrationEntityMappingKey;  
NSString *NSMigrationPropertyMappingKey;  
NSString *NSMigrationEntityPolicyKey;
```

Constants

`NSMigrationManagerKey`

Key for the migration manager.

To access this key in a custom value expression string in the Xcode mapping model editor use `$manager`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationSourceObjectKey`

Key for the source object.

To access this key in a custom value expression string in the Xcode mapping model editor use `$source`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationDestinationObjectKey`

Key for the destination object.

To access this key in a custom value expression string in the Xcode mapping model editor use `$destination`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationEntityMappingKey`

Key for the entity mapping object.

To access this key in a custom value expression string in the Xcode mapping model editor use `$entityMapping`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationPropertyMappingKey`

Key for the property mapping object.

To access this key in a custom value expression string in the Xcode mapping model editor use `$propertyMapping`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationEntityPolicyKey`

Key for the entity migration policy object.

To access this key in a custom value expression string in the Xcode mapping model editor use `$entityPolicy`.

Available in iOS 3.0 and later.

Declared in `NSEntityMigrationPolicy.h`.

Discussion

You can use these keys in the right hand sides of a value expression.

NSExpressionDescription

Inherits from	NSPropertyDescription : NSObject
Conforms to	NSCoding (NSPropertyDescription) NSCopying (NSPropertyDescription) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSExpressionDescription.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSExpressionDescription` objects represent a special property description type intended for use with the `NSFetchRequest` `propertiesToFetch` (page 106) method.

An `NSExpressionDescription` describes a column to be returned from a fetch that may not appear directly as an attribute or relationship on an entity. Examples might include `upper(attribute)` or `max(attribute)`. You cannot set an `NSExpressionDescription` object as a property of an entity.

Tasks

Getting Information About an Expression Description

- `expression` (page 78)
Returns the expression for the receiver.
- `setExpression:` (page 78)
Sets the expression for the receiver.
- `expressionResultType` (page 78)
Returns the type of the receiver.
- `setExpressionResultType:` (page 79)
Sets the type of the receiver.

Instance Methods

expression

Returns the expression for the receiver.

- (NSExpression *)expression

Return Value

The expression for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setExpression:](#) (page 78)

Declared In

NSExpressionDescription.h

expressionResultType

Returns the type of the receiver.

- (NSAttributeType)expressionResultType

Return Value

The type of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [expressionResultType](#) (page 78)

Declared In

NSExpressionDescription.h

setExpression:

Sets the expression for the receiver.

- (void)setExpression:(NSExpression *)expression

Parameters

expression

The expression for the receiver.

Special Considerations

This method raises an exception if the receiver's `model` has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [expression](#) (page 78)

Declared In

NSExpressionDescription.h

setExpressionResultType:

Sets the type of the receiver.

- (void)setExpressionResultType:(NSAttributeType) *type*

Parameters

type

An `NSAttributeType` constant that specifies the type for the receiver.

Special Considerations

This method raises an exception if the receiverâs model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [setExpressionResultType:](#) (page 79)

Declared In

NSExpressionDescription.h

NSFetchedPropertyDescription Class Reference

Inherits from	NSPropertyDescription : NSObject
Conforms to	NSCoding (NSPropertyDescription) NSCopying (NSPropertyDescription) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSFetchedPropertyDescription.h
Companion guides	Core Data Programming Guide Predicate Programming Guide

Overview

The `NSFetchedPropertyDescription` class is used to define “fetched properties.” Fetched properties allow you to specify related objects through a weak, unidirectional relationship defined by a fetch request.

An example might be a iTunes playlist, if expressed as a property of a containing object. Songs don’t belong to a particular playlist, especially in the case that they’re on a remote server. The playlist may remain even after the songs have been deleted, or the remote server has become inaccessible. Note, however, that unlike a playlist a fetched property is static—it does not dynamically update itself as objects in the destination entity change.

The effect of a fetched property is similar to executing a fetch request yourself and placing the results in a transient attribute, although with the framework managing the details. In particular, a fetched property is not fetched until it is requested, and the results are then cached until the object is turned into a fault. You use `refreshObject:mergeChanges:` (page 167) (`NSManagedObjectContext`) 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.

Unlike other relationships, which are all sets, fetched properties are represented by an ordered `NSArray` object just as if you executed the fetch request yourself. The fetch request associated with the property can have a sort ordering. The value for a fetched property of a managed object does not support `mutableArrayValueForKey:`.

Fetch Request Variables

Fetch requests set on an fetched property have 2 special variable bindings you can use: `$FETCH_SOURCE` and `$FETCHED_PROPERTY`. The source refers to the specific managed object that has this property; the property refers to the `NSFetchedPropertyDescription` object itself (which may have a user info associated with it that you want to use).

Editing Fetched Property Descriptions

Fetched Property descriptions are editable until they are used by an object graph manager. This allows you to create or modify them dynamically. However, once a description is used (when the managed object model to which it belongs is associated with a persistent store coordinator), it *must not* (indeed cannot) be changed. This is enforced at runtime: any attempt to mutate a model or any of its subjects after the model is associated with a persistent store coordinator causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

Tasks

Getting and Setting the Fetch Request

- [fetchRequest](#) (page 82)
Returns the fetch request of the receiver.
- [setFetchRequest:](#) (page 83)
Sets the fetch request of the receiver.

Instance Methods

fetchRequest

Returns the fetch request of the receiver.

- (NSFetchRequest *)fetchRequest

Return Value

The fetch request of the receiver.

Availability

See Also

- [setFetchRequest:](#) (page 83)

Declared In

NSFetchedPropertyDescription.h

setFetchRequest:

Sets the fetch request of the receiver.

- (void)setFetchRequest:(NSFetchRequest *)*fetchRequest*

Parameters

fetchRequest

The fetch request of the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability**See Also**

- [fetchRequest](#) (page 82)

Declared In

NSFetchedPropertyDescription.h

NSFetchedResultsController Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSFetchedResultsController.h
Companion guide	Core Data Programming Guide

Overview

This class is intended to efficiently manage the results returned from a Core Data fetch request to provide data for a `UITableView` object.

You use this class to help populate cells in a `UITableView` object with objects fetched from Core Data. While table views can be used in several ways, this object is primarily intended to assist you with a master list view. `UITableView` expects its data source to provide cells as an array of sections made up of rows. You configure an instance of this class using a fetch request that specifies the entity, an array containing at least one sort ordering, and optionally a filter predicate. `NSFetchedResultsController` efficiently analyzes the result of the fetch request and computes all the information about sections in the result set, and for the index.

In addition, `NSFetchedResultsController` provides the following features:

- It optionally monitors changes to objects in its associated managed object context, and reports changes in the results set to its delegate (see [“The Controller’s Delegate”](#) (page 86)).
- It optionally caches the results of its computation so that if the same data is subsequently re-displayed, the work does not have to be repeated (see [“The Cache”](#) (page 87)).

A controller thus effectively has three modes of operation, determined by whether it has a delegate and whether the cache file name is set.

1. No tracking: the delegate is set to `nil`.

The controller simply provides access to the data as it was when the fetch was executed.

2. Memory-only tracking: the delegate is non-`nil` and the file cache name is set to `nil`.

The controller monitors objects in its result set and updates section and ordering information in response to relevant changes.

3. Full persistent tracking: the delegate and the file cache name are non-`nil`.

The controller monitors objects in its result set and updates section and ordering information in response to relevant changes. The controller maintains a persistent cache of the results of its computation.

Using NSFetchedResultsController

Creating the Fetched Results Controller

You typically create an instance of `NSFetchedResultsController` as an instance variable of a table view controller. You configure the controller with a fetch request, which must contain at least one sort descriptor to order the results. You can specify as the `sectionNameKeyPath` argument a key that the controller will use to split the results into sections, or pass `nil` to indicate that the controller should generate a single section. After creating an instance, you invoke `performFetch:` (page 96) to actually execute the fetch.

```
NSManagedObjectContext *context = <#Managed object context#>;
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
// Configure the request's entity, and optionally its predicate.
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"<#Sort
key#>" ascending:YES];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor,
nil];
[fetchRequest setSortDescriptors:sortDescriptors];
[sortDescriptors release];
[sortDescriptor release];

NSFetchedResultsController *controller = [[NSFetchedResultsController alloc]
initWithFetchRequest:fetchRequest
managedObjectContext:context
sectionNameKeyPath:nil
cacheName:@"<#Cache name#>"];
[fetchRequest release];

NSError *error;
BOOL success = [controller performFetch:&error];
```

Important: You must not modify the fetch request after you have initialized the controller. For example, you must not change the predicate or the sort orderings.

The Controller's Delegate

If you set a delegate for a fetched results controller, the controller registers to receive change notifications from its managed object context. Any change in the context that affects the result set or section information is processed and the results are updated accordingly. The controller notifies the delegate when result objects change location or when sections are modified (see `NSFetchedResultsControllerDelegate`). You typically use these methods to update the display of the table view.

Important: If you do not set a delegate, the controller does not monitor changes to objects in its associated managed object context. You may choose not to set a delegate if the results set will not change (if you are providing a read-only list, for example).

The Cache

Where possible, a controller uses a cache to avoid the need to repeat work performed in setting up any sections and ordering the contents. The cache is maintained across launches of your application.

When you initialize an instance of `NSFetchedResultsController`, you typically specify a cache name. (If you do not specify a cache name, the controller does not cache data.) When you create a controller, it looks for an existing cache with the given name:

- If the controller can't find an appropriate cache, it calculates the required sections and the order of objects within sections. It then writes this information to disk.
- If it finds a cache with the same name, the controller tests the cache to determine whether its contents are still valid. The controller compares the current entity name, entity version hash, sort descriptors, section key-path, and total object count with those stored in the cache, as well as the modification date of the cached information file and the persistent store file.

If the cache is consistent with the current information, the controller reuses the previously-computed information.

If the cache is not consistent with the current information, then the required information is recomputed, and the cache updated.

Any time the section and ordering information change, the cache is updated.

If you have multiple fetched results controllers with different configurations (different sort descriptors and so on), you must give each a different cache name.

You can purge a cache using `deleteCacheWithName:` (page 94).

Implementing the Table View Datasource Methods

You ask the object to provide relevant information in your implementation of the table view data source methods:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [[<#Fetched results controller#> sections] count];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    id <NSFetchedResultsSectionInfo> sectionInfo = [[<#Fetched results
controller#> sections] objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
        UITableViewCell *cell = <#Get the cell#>;
        NSManagedObject *managedObject = [[<#Fetched results controller#>
objectAtIndexPath:indexPath];
        // Configure the cell with data from the managed object.
        return cell;
    }

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    id <NSFetchedResultsControllerSectionInfo> sectionInfo = [[<#Fetched results
controller#> sections] objectAtIndex:section];
    return [sectionInfo name];
}

- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
}

- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title atIndex:(NSInteger)index {
    return [<#Fetched results controller#> sectionForSectionIndexTitle:title
atIndex:index];
}
```


Important: On iOS 3.0, if you configure a fetched results controller to not use sections (`sectionNameKeyPath` is set to `nil`), there is an incompatibility between the values returned by `NSFetchedResultsController` and the values expected by `UITableView`. You can work around this incompatibility as follows:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    NSInteger count = [[<#Fetched results controller#> sections] count];
    if (count == 0) {
        count = 1;
    }
    return count;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    NSArray *sections = [[<#Fetched results controller#> sections];
    NSInteger count = 0;
    if ([sections count]) {
        id <NSFetchedResultsSectionInfo> sectionInfo = [sections objectAtIndex:section];
        count = [sectionInfo numberOfObjects];
    }
    return count;
}
```

This workaround is not required on iOS 3.1 and later.

Responding to Changes

In general, `NSFetchedResultsController` is designed to respond to changes at the model layer, by informing its delegate when result objects change location or when sections are modified.

If you allow a user to reorder table rows, then your implementation of the delegate methods must take this into account—see `NSFetchedResultsControllerDelegate`.

Important: In versions of iOS prior to v4.0, `NSFetchedResultsController` does not support sections being deleted as a result of a UI-driven change.

Changes are not reflected until after the controller's managed object context has received a [processPendingChanges](#) (page 166) message. Therefore, if you change the value of a managed object's attribute so that its location in a fetched results controller's results set would change, its index as reported by the controller would typically not change until the end of the current event cycle (when [processPendingChanges](#) (page 166) is invoked). For example, the following code fragment would log "same":

```
NSFetchedResultsController *frc = <#A fetched results controller#>;
NSManagedObject *managedObject = <#A managed object in frc's fetchedObjects
array#>;
NSIndexPath *beforeIndexPath = [frc indexPathForObject:managedObject];
[managedObject setSortKeyAttribute:
    <#A new value that changes managedObject's position in frc's
    fetchedObjects array#>;
NSIndexPath *afterIndexPath = [frc indexPathForObject:managedObject];
```

```
if ([beforeIndexPath compare:afterIndexPath] == NSOrderedSame) {
    NSLog(@"same");
}
```

Handling Object Invalidation

When a managed object context notifies the fetched results controller that *individual* objects are invalidated, the controller treats these as deleted objects and sends the proper delegate calls.

It's possible for *all* the objects in a managed object context to be invalidated simultaneously. (For example, as a result of calling [reset](#) (page 169), or if a store is removed from the the persistent store coordinator.) When this happens, `NSFetchedResultsController` does not invalidate all objects, nor does it send individual notifications for object deletions. Instead, you must call [performFetch:](#) (page 96) to reset the state of the controller then reload the data in the table view (`reloadData`).

Subclassing Notes

You create a subclass of this class if you want to customize the creation of sections and index titles. You override [sectionIndexTitleForSectionName:](#) (page 97) if you want the section index title to be something other than the capitalized first letter of the section name. You override [sectionIndexTitles](#) (page 93) if you want the index titles to be something other than the array created by calling [sectionIndexTitleForSectionName:](#) (page 97) on all the known sections.

Tasks

Initialization

- [initWithFetchRequest:managedObjectContext:sectionNameKeyPath:cacheName:](#) (page 95)
Returns a fetch request controller initialized using the given arguments.
- [performFetch:](#) (page 96)
Executes the receiver's fetch request.

Configuration Information

- [fetchRequest](#) (page 92) *property*
The fetch request used to do the fetching.
- [managedObjectContext](#) (page 93) *property*
The managed object context used to fetch objects.
- [sectionNameKeyPath](#) (page 93) *property*
The key path on the fetched objects used to determine the section they belong to.
- [cacheName](#) (page 91) *property*
The name of the file used to cache section information.

- `delegate` (page 92) *property*
The object that is notified when the fetched results changed.
- + `deleteCacheWithName:` (page 94)
Deletes the cached section information with the given name.

Accessing Results

- `fetchedObjects` (page 92) *property*
The results of the fetch.
- `objectAtIndexPath:` (page 96)
Returns the object at the given index path in the fetch results.
- `indexPathForObject:` (page 94)
Returns the index path of a given object.

Querying Section Information

- `sections` (page 94) *property*
The sections for the receiver's fetch results.
- `sectionForSectionIndexTitle:atIndex:` (page 97)
Returns the section number for a given section title and index in the section index.

Configuring Section Information

- `sectionIndexTitleForSectionName:` (page 97)
Returns the corresponding section index entry for a given section name.
- `sectionIndexTitles` (page 93) *property*
The array of section index titles.

Properties

For more about Objective-C properties, see “Properties” in *The Objective-C Programming Language*.

cacheName

The name of the file used to cache section information.

```
@property (nonatomic, readonly) NSString *cacheName
```

Discussion

The file itself is stored in a private directory; you can only access it by name using `deleteCacheWithName:` (page 94)

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

delegate

The object that is notified when the fetched results changed.

```
@property(nonatomic, assign) id <NSFetchedResultsControllerDelegate> delegate
```

Special Considerations

If you do not specify a delegate, the controller does not track changes to managed objects associated with its managed object context.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

fetchedObjects

The results of the fetch.

```
@property (nonatomic, readonly) NSArray *fetchedObjects
```

Discussion

Returns `nil` if `performFetch:` (page 96) hasn't been called.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

fetchRequest

The fetch request used to do the fetching.

```
@property (nonatomic, readonly) NSFetchRequest *fetchRequest
```

Discussion

Typically the sort descriptor used in the request groups objects into sections.

Important: You must not modify the fetch request. For example, you must not change its predicate or the sort orderings.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

managedObjectContext

The managed object context used to fetch objects.

```
@property (nonatomic, readonly) NSManagedObjectContext *managedObjectContext
```

Discussion

The controller registers to listen to change notifications on this context and properly update its result set and section information.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

sectionIndexTitles

The array of section index titles.

```
@property (nonatomic, readonly) NSArray *sectionIndexTitles
```

Discussion

The default implementation returns the array created by calling [sectionIndexTitleForSectionName:](#) (page 97) on all the known sections. You should override this method if you want to return a different array for the section index.

Special Considerations

You only need this method if you use a section index.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

sectionNameKeyPath

The key path on the fetched objects used to determine the section they belong to.

```
@property (nonatomic, readonly) NSString *sectionNameKeyPath
```

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

sections

The sections for the receiver's fetch results.

```
@property (nonatomic, readonly) NSArray *sections
```

Discussion

The objects in the sections array implement the `NSFetchedResultsControllerSectionInfo` protocol.

You typically use the sections array when implementing `UITableViewDataSource` methods, such as `numberOfSectionsInTableView:` and `tableView:titleForHeaderInSection:`.

Availability

Available in iOS 3.0 and later.

Declared In

`NSFetchedResultsController.h`

Class Methods

deleteCacheWithName:

Deletes the cached section information with the given name.

```
+ (void)deleteCacheWithName:(NSString *)name
```

Parameters

name

The name of the cache file to delete.

If *name* is `nil`, deletes all cache files.

Availability

Available in iOS 3.0 and later.

Declared In

`NSFetchedResultsController.h`

Instance Methods

indexPathForObject:

Returns the index path of a given object.

```
- (NSIndexPath *)indexPathForObject:(id)object
```

Parameters

object

An object in the receiver's fetch results.

Return Value

The index path of *object* in the receiver's fetch results, or *nil* if *object* could not be found.

Special Considerations

In versions of iOS before 3.2, this method raises an exception if *object* is not contained in the receiver's fetch results.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

initWithFetchRequest:managedObjectContext:sectionNameKeyPath:cacheName:

Returns a fetch request controller initialized using the given arguments.

```
- (id)initWithFetchRequest:(NSFetchRequest *)fetchRequest
    managedObjectContext:(NSManagedObjectContext *)context
    sectionNameKeyPath:(NSString *)sectionNameKeyPath
    cacheName:(NSString *)name
```

Parameters

fetchRequest

The fetch request used to get the objects.

The fetch request must have at least one sort descriptor. If the controller generates sections, the first sort descriptor in the array is used to group the objects into sections; its key must either be the same as *sectionNameKeyPath* or the relative ordering using its key must match that using *sectionNameKeyPath*.

Important: You must not modify *fetchRequest* after invoking this method. For example, you must not change its predicate or the sort orderings.

context

The managed object against which *fetchRequest* is executed.

sectionNameKeyPath

A key path on result objects that returns the section name. Pass *nil* to indicate that the controller should generate a single section.

The section name is used to pre-compute the section information.

If this key path is not the same as that specified by the first sort descriptor in *fetchRequest*, they must generate the same relative orderings. For example, the first sort descriptor in *fetchRequest* might specify the key for a persistent property; *sectionNameKeyPath* might specify a key for a transient property derived from the persistent property.

name

The name of the cache file the receiver should use. Pass `nil` to prevent caching.

Pre-computed section info is cached to a private directory under this name. If Core Data finds a cache stored with this name, it is checked to see if it matches the *fetchRequest*. If it does, the cache is loaded directly—this avoids the overhead of computing the section and index information. If the cached information doesn't match the request, the cache is deleted and recomputed when the fetch happens.

Return Value

The receiver initialized with the specified fetch request, context, key path, and cache name.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

objectAtIndexPath:

Returns the object at the given index path in the fetch results.

```
- (id)objectAtIndexPath:(NSIndexPath *)indexPath
```

Parameters

indexPath

An index path in the fetch results.

If *indexPath* does not describe a valid index path in the fetch results, an exception is raised.

Return Value

The object at a given index path in the fetch results.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

performFetch:

Executes the receiver's fetch request.

```
- (BOOL)performFetch:(NSError **)error
```

Parameters

error

If the fetch is not successful, upon return contains an error object that describes the problem.

Return Value

YES if the fetch executed successfully, otherwise NO.

Discussion

After executing this method, you can access the receiver's the fetched objects with the property [fetchedObjects](#) (page 92).

Special Considerations

This method returns `NO` (and a suitable error in *error*) if the fetch request doesn't include a sort descriptor that uses the section name key path specified in

`initWithFetchRequest:managedObjectContext:sectionNameKeyPath:cacheName:` (page 95).

Availability

Available in iOS 3.0 and later.

Declared In

`NSFetchedResultsController.h`

sectionForSectionIndexTitle:atIndex:

Returns the section number for a given section title and index in the section index.

```
- (NSInteger)sectionForSectionIndexTitle:(NSString *)title
    atIndex:(NSInteger)sectionIndex
```

Parameters

title

The title of a section

sectionIndex

The index of a section.

Return Value

The section number for the given section title and index in the section index

Discussion

You would typically call this method when executing `UITableViewDataSource`'s `tableView:sectionForSectionIndexTitle:atIndex:` method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSFetchedResultsController.h`

sectionIndexTitleForSectionName:

Returns the corresponding section index entry for a given section name.

```
- (NSString *)sectionIndexTitleForSectionName:(NSString *)sectionName
```

Parameters

sectionName

The name of a section.

Return Value

The section index entry corresponding to the section with name *sectionName*.

Discussion

The default implementation returns the capitalized first letter of the section name.

You should override this method if you need a different way to convert from a section name to its name in the section index.

Special Considerations

You only need this method if you use a section index.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

NSFetchRequest Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSCopying NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSFetchRequest.h
Companion guides	Core Data Programming Guide Predicate Programming Guide

Overview

The `NSFetchRequest` class is used to describe search criteria used to retrieve data from a persistent store.

An instance collects the criteria needed to select and—optionally—order a group of persistent objects, whether from a repository such as a file or an in-memory store such as a managed object context. A fetch request contains the following elements:

- An entity description (an instance of `NSEntityDescription`) that specifies which entity to search, and hence what type of object (if any) will be returned. This is the only mandatory element.
- A predicate (an instance of `NSPredicate`) that specifies which properties to select by and the constraints on selection, for example “last name begins with a ‘J’”. If you don’t specify a predicate, then all instances of the specified entity are selected (subject to other constraints, see [executeFetchRequest:error:](#) (page 160) for full details).
- An array of sort descriptors (instances of `NSSortDescriptor`) that specify how the returned objects should be ordered, for example by last name then by first name.

You can also specify other aspects of a fetch request—the maximum number of objects that a request should return, and which data stores the request should access. With Mac OS X v10.5 and later you can also specify, for example, whether the fetch returns managed objects or just object IDs, and whether objects are fully populated with their properties (see [resultType](#) (page 107), [includesSubentities](#) (page 105), [includesPropertyValues](#) (page 105), and [returnsObjectsAsFaults](#) (page 108)). With Mac OS X v10.6 and later and on iOS, you can further specify, for example, what properties to fetch, the fetch offset, and whether, when the fetch is executed it matches against currently unsaved changes in the managed object context (see [resultType](#) (page 107), [propertiesToFetch](#) (page 106), [fetchOffset](#) (page 104), and [includesPendingChanges](#) (page 104)).

You use `NSFetchRequest` objects with the method `executeFetchRequest:error:` (page 160), defined by `NSManagedObjectContext`.

You often predefine fetch requests in a managed object model—`NSManagedObjectContext` provides API to retrieve a stored fetch request by name. Stored fetch requests can include placeholders for variable substitution, and so serve as templates for later completion. Fetch request templates therefore allow you to pre-define queries with variables that are substituted at runtime.

Tasks

Entity

- `entity` (page 102)
Returns the entity specified for the receiver.
- `setEntity:` (page 109)
Sets the entity of the receiver.
- `includesSubentities` (page 105)
Returns a Boolean value that indicates whether the receiver includes subentities in the results.
- `setIncludesSubentities:` (page 112)
Sets whether the receiver includes subentities.

Fetch Constraints

- `predicate` (page 106)
Returns the predicate of the receiver.
- `setPredicate:` (page 112)
Sets the predicate of the receiver.
- `fetchLimit` (page 103)
Returns the fetch limit of the receiver.
- `setFetchLimit:` (page 110)
Sets the fetch limit of the receiver.
- `fetchOffset` (page 104)
Returns the fetch offset of the receiver.
- `setFetchOffset:` (page 110)
Sets the fetch offset of the receiver.
- `fetchBatchSize` (page 102)
Returns the batch size of the receiver.
- `setFetchBatchSize:` (page 110)
Sets the fetch offset of the receiver.
- `affectedStores` (page 102)
Returns an array containing the persistent stores specified for the receiver.
- `setAffectedStores:` (page 109)
Sets the array of persistent stores that will be searched by the receiver.

Sorting

- [sortDescriptors](#) (page 115)
Returns the sort descriptors of the receiver.
- [setSortDescriptors:](#) (page 115)
Sets the array of sort descriptors of the receiver.

Prefetching

- [relationshipKeyPathsForPrefetching](#) (page 107)
Returns the array of relationship keypaths to prefetch along with the entity for the request.
- [setRelationshipKeyPathsForPrefetching:](#) (page 113)
Sets an array of relationship keypaths to prefetch along with the entity for the request.

Managing How Results Are Returned

- [resultType](#) (page 107)
Returns the result type of the receiver.
- [setResultType:](#) (page 113)
Sets the result type of the receiver.
- [includesPendingChanges](#) (page 104)
Returns a Boolean value that indicates whether, when the fetch is executed it matches against currently unsaved changes in the managed object context.
- [setIncludesPendingChanges:](#) (page 111)
Sets if, when the fetch is executed, it matches against currently unsaved changes in the managed object context.
- [propertiesToFetch](#) (page 106)
Returns an array of `NSPropertyDescription` objects that specify which properties should be returned by the fetch.
- [setPropertyToFetch:](#) (page 112)
Specifies which properties should be returned by the fetch.
- [returnsDistinctResults](#) (page 108)
Returns a Boolean value that indicates whether the fetch request returns only distinct values for the fields specified by `propertiesToFetch`.
- [setReturnsDistinctResults:](#) (page 114)
Sets whether the request should return only distinct values for the fields specified by `propertiesToFetch`.
- [includesPropertyValues](#) (page 105)
Returns a Boolean value that indicates whether, when the fetch is executed, property data is obtained from the persistent store.
- [setIncludesPropertyValues:](#) (page 111)
Sets if, when the fetch is executed, property data is obtained from the persistent store.
- [returnsObjectsAsFaults](#) (page 108)
Returns a Boolean value that indicates whether the objects resulting from a fetch using the receiver are faults.

- [setReturnsObjectsAsFaults:](#) (page 114)
Sets whether the objects resulting from a fetch request are faults.

Instance Methods

affectedStores

Returns an array containing the persistent stores specified for the receiver.

- (NSArray *)affectedStores

Return Value

An array containing the persistent stores specified for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setAffectedStores:](#) (page 109)

Declared In

NSFetchRequest.h

entity

Returns the entity specified for the receiver.

- (NSEntityDescription *)entity

Return Value

The entity specified for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setEntity:](#) (page 109)

Declared In

NSFetchRequest.h

fetchBatchSize

Returns the batch size of the receiver.

- (NSUInteger)fetchBatchSize

Return Value

The batch size of the receiver.

Discussion

The default value is 0. A batch size of 0 is treated as infinite, which disables the batch faulting behavior.

If you set a non-zero batch size, the collection of objects returned when the fetch is executed is broken into batches. When the fetch is executed, the entire request is evaluated and the identities of all matching objects recorded, but no more than *batchSize* objects' data will be fetched from the persistent store at a time. The array returned from executing the request will be a proxy object that transparently faults batches on demand. (In database terms, this is an in-memory cursor.)

You can use this feature to restrict the working set of data in your application. In combination with [fetchLimit](#) (page 103), you can create a subrange of an arbitrary result set.

Special Considerations

For purposes of thread safety, you should consider the array proxy returned when the fetch is executed as being owned by the managed object context the request is executed against, and treat it as if it were a managed object registered with that context.

Availability

Available in iOS 3.0 and later.

See Also

- [setFetchBatchSize](#): (page 110)
- [fetchLimit](#) (page 103)

Declared In

NSFetchRequest.h

fetchLimit

Returns the fetch limit of the receiver.

- (NSUInteger)fetchLimit

Return Value

The fetch limit of the receiver.

Discussion

The fetch limit specifies the maximum number of objects that a request should return when executed.

Special Considerations

If you set a fetch limit, the framework makes a best effort, but does not guarantee, to improve efficiency. For every object store except the SQL store, a fetch request executed with a fetch limit in effect simply performs an unlimited fetch and throws away the unasked for rows.

Availability

Available in iOS 3.0 and later.

See Also

- [setFetchLimit](#): (page 110)
- [fetchOffset](#) (page 104)

Declared In

NSFetchRequest.h

fetchOffset

Returns the fetch offset of the receiver.

- (NSUInteger)fetchOffset

Return Value

The fetch offset of the receiver.

Discussion

The default value is 0.

This setting allows you to specify an offset at which rows will begin being returned. Effectively, the request will skip over the specified number of matching entries. For example, given a fetch which would normally return a, b, c, d, specifying an offset of 1 will return b, c, d, and an offset of 4 will return an empty array. Offsets are ignored in nested requests such as subqueries.

This can be used to restrict the working set of data. In combination with `-fetchLimit`, you can create a subrange of an arbitrary result set.

Availability

Available in iOS 3.0 and later.

See Also

- [setFetchOffset:](#) (page 110)
- [fetchLimit](#) (page 103)

Declared In

NSFetchRequest.h

includesPendingChanges

Returns a Boolean value that indicates whether, when the fetch is executed it matches against currently unsaved changes in the managed object context.

- (BOOL)includesPendingChanges

Return Value

YES if, when the fetch is executed it will match against currently unsaved changes in the managed object context, otherwise NO.

Discussion

The default value is YES.

If the value is NO, the fetch request skips checking unsaved changes and only returns objects that matched the predicate in the persistent store.

Availability

Available in iOS 3.0 and later.

See Also

- [setIncludesPendingChanges:](#) (page 111)

Declared In

NSFetchRequest.h

includesPropertyValues

Returns a Boolean value that indicates whether, when the fetch is executed, property data is obtained from the persistent store.

- (BOOL)includesPropertyValues

Return Value

YES if, when the fetch is executed, property data is obtained from the persistent store, otherwise NO.

Discussion

The default value is YES.

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 database.

During a normal fetch (`includesPropertyValues` is YES), Core Data fetches the object ID *and* property data for the matching records, fills the row cache with the information, and returns managed object as faults (see [returnsObjectsAsFaults](#) (page 108)). These faults are managed objects, but all of their property data still resides in the row cache until the fault is fired. When the fault is fired, Core Data retrieves the data from the row cache—there is no need to go back to the database.

If `includesPropertyValues` is NO, then Core Data fetches *only* the object ID information for the matching records—it does not populate the row cache. Core Data still returns managed objects since it only needs managed object IDs to create faults. However, if you subsequently fire the fault, Core Data looks in the (empty) row cache, doesn't find any data, and then goes back to the store a second time for the data.

Availability

Available in iOS 3.0 and later.

See Also

- [setIncludesPropertyValues:](#) (page 111)

Declared In

NSFetchRequest.h

includesSubentities

Returns a Boolean value that indicates whether the receiver includes subentities in the results.

- (BOOL)includesSubentities

Return Value

YES if the request will include all subentities of the entity for the request, otherwise NO.

Discussion

The default is YES.

Availability

Available in iOS 3.0 and later.

See Also

- [setIncludesSubentities:](#) (page 112)

Declared In

NSFetchRequest.h

predicate

Returns the predicate of the receiver.

- (NSPredicate *)predicate

Return Value

The predicate of the receiver.

Discussion

The predicate is used to constrain the selection of objects the receiver is to fetch. For more about predicates, see *Predicate Programming Guide*.

If the predicate is empty—for example, if it is an AND predicate whose array of elements contains no predicates—the receiver has its predicate set to `nil`. For more about predicates, see *Predicate Programming Guide*.

Availability

Available in iOS 3.0 and later.

See Also

- [setPredicate:](#) (page 112)

Declared In

NSFetchRequest.h

propertiesToFetch

Returns an array of `NSPropertyDescription` objects that specify which properties should be returned by the fetch.

- (NSArray *)propertiesToFetch

Return Value

An array of `NSPropertyDescription` objects that specify which properties should be returned by the fetch.

Discussion

For a full discussion, see [setPropertyToFetch:](#) (page 112).

Availability

Available in iOS 3.0 and later.

See Also

- [setPropertyToFetch:](#) (page 112)

- [resultType](#) (page 107)

- [returnsDistinctResults](#) (page 108)

Declared In

NSFetchRequest.h

relationshipKeyPathsForPrefetching

Returns the array of relationship keypaths to prefetch along with the entity for the request.

- (NSArray *)relationshipKeyPathsForPrefetching

Return Value

The array of relationship keypaths to prefetch along with the entity for the request.

Discussion

The default value is an empty array (no prefetching).

Prefetching allows Core Data to obtain related objects in a single fetch (per entity), rather than incurring subsequent access to the store for each individual record as their faults are tripped. 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, it may be that a fault has to be fired for each individual Department object (for more details, see “Core Data Performance” in *Core Data Programming Guide*). This can represent a significant overhead. You could avoid this by prefetching the department relationship in the Employee fetch, as illustrated in the following example:

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

Availability

Available in iOS 3.0 and later.

See Also

- [setRelationshipKeyPathsForPrefetching:](#) (page 113)

Declared In

NSFetchRequest.h

resultType

Returns the result type of the receiver.

- (NSFetchRequestResultType)resultType

Return Value

The result type of the receiver.

Discussion

The default value is `NSManagedObjectResultType`.

You use [setResultType:](#) (page 113) to set the instance type of objects returned from executing the request—for possible values, see “Fetch request result types” (page 116). If you set the value to `NSManagedObjectIDResultType`, this will demote any sort orderings to “best efforts” hints if you do not include the property values in the request.

Availability

Available in iOS 3.0 and later.

See Also

- [setResultType:](#) (page 113)

Declared In

NSFetchRequest.h

returnsDistinctResults

Returns a Boolean value that indicates whether the fetch request returns only distinct values for the fields specified by `propertiesToFetch`.

- (BOOL)returnsDistinctResults

Return Value

YES if, when the fetch is executed, it returns only distinct values for the fields specified by `propertiesToFetch`, otherwise NO.

Discussion

The default value is NO.

Special Considerations

This value is only used if a value has been set for [propertiesToFetch](#) (page 106).

Availability

Available in iOS 3.0 and later.

See Also

- [setReturnsDistinctResults:](#) (page 114)
- [propertiesToFetch](#) (page 106)

Declared In

NSFetchRequest.h

returnsObjectsAsFaults

Returns a Boolean value that indicates whether the objects resulting from a fetch using the receiver are faults.

- (BOOL)returnsObjectsAsFaults

Return Value

YES if the objects resulting from a fetch using the receiver are faults, otherwise NO.

Discussion

The default value is YES. This setting is not used if the result type (see [resultType](#) (page 107)) is `NSManagedObjectIDResultType`, as object IDs do not have property values. You can set `returnsObjectsAsFaults` to NO to gain a performance benefit if you know you will need to access the property values from the returned objects.

By default, when you execute a fetch `returnsObjectsAsFaults` is YES; Core Data fetches the object data for the matching records, fills the row cache with the information, and returns managed object as faults. These faults are managed objects, but all of their property data resides in the row cache until the fault is fired. When the fault is fired, Core Data retrieves the data from the row cache. Although the overhead for this operation is small for large datasets it may become non-trivial. If you *need* to access the property values

from the returned objects (for example, if you iterate over all the objects to calculate the average value of a particular attribute), then it is more efficient to set `returnsObjectsAsFaults` to `NO` to avoid the additional overhead.

Availability

Available in iOS 3.0 and later.

See Also

- [setReturnsObjectsAsFaults:](#) (page 114)

Declared In

NSFetchRequest.h

setAffectedStores:

Sets the array of persistent stores that will be searched by the receiver.

```
- (void)setAffectedStores:(NSArray *)stores
```

Parameters

stores

An array containing identifiers for the stores to be searched when the receiver is executed.

Availability

Available in iOS 3.0 and later.

See Also

- [affectedStores](#) (page 102)

Declared In

NSFetchRequest.h

setEntity:

Sets the entity of the receiver.

```
- (void)setEntity:(NSEntityDescription *)entity
```

Parameters

entity

The entity of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [entity](#) (page 102)

Declared In

NSFetchRequest.h

setFetchBatchSize:

Sets the fetch offset of the receiver.

```
- (void)setFetchBatchSize:(NSUInteger)batchSize
```

Parameters

batchSize

The batch size of the receiver.

A batch size of 0 is treated as infinite, which disables the batch faulting behavior.

Discussion

For a full discussion, see [fetchBatchSize](#) (page 102).

Availability

Available in iOS 3.0 and later.

See Also

- [fetchBatchSize](#) (page 102)

- [fetchLimit](#) (page 103)

Declared In

NSFetchRequest.h

setFetchLimit:

Sets the fetch limit of the receiver.

```
- (void)setFetchLimit:(NSUInteger)limit
```

Parameters

limit

The fetch limit of the receiver. 0 specifies no fetch limit.

Discussion

For a full discussion, see [fetchLimit](#) (page 103).

Availability

Available in iOS 3.0 and later.

See Also

- [fetchLimit](#) (page 103)

- [fetchOffset](#) (page 104)

Declared In

NSFetchRequest.h

setFetchOffset:

Sets the fetch offset of the receiver.

```
- (void)setFetchOffset:(NSUInteger)limit
```

Parameters*limit*

The fetch offset of the receiver.

Discussion

For a full discussion, see [fetchOffset](#) (page 104).

Availability

Available in iOS 3.0 and later.

See Also

- [fetchOffset](#) (page 104)
- [fetchLimit](#) (page 103)

Declared In

NSFetchRequest.h

setIncludesPendingChanges:

Sets if, when the fetch is executed, it matches against currently unsaved changes in the managed object context.

```
- (void)setIncludesPendingChanges:(BOOL)yesNo
```

Parameters*yesNo*

If YES, when the fetch is executed it will match against currently unsaved changes in the managed object context.

Discussion

For a full discussion, see [includesPendingChanges](#) (page 104).

Availability

Available in iOS 3.0 and later.

See Also

- [includesPendingChanges](#) (page 104)

Declared In

NSFetchRequest.h

setIncludesPropertyValues:

Sets if, when the fetch is executed, property data is obtained from the persistent store.

```
- (void)setIncludesPropertyValues:(BOOL)yesNo
```

Parameters*yesNo*

If NO, the request will not obtain property information, but only information to identify each object (used to create managed object IDs).

Discussion

For a full discussion, see [includesPropertyValues](#) (page 105).

Availability

Available in iOS 3.0 and later.

See Also

- [includesPropertyValues](#) (page 105)

Declared In

NSFetchRequest.h

setIncludesSubentities:

Sets whether the receiver includes subentities.

- (void)setIncludesSubentities:(BOOL)*yesNo*

Parameters

yesNo

If NO, the receiver will fetch objects of exactly the entity type of the request; if YES, the receiver will include all subentities of the entity for the request (if any).

Availability

Available in iOS 3.0 and later.

See Also

- [includesSubentities](#) (page 105)

Declared In

NSFetchRequest.h

setPredicate:

Sets the predicate of the receiver.

- (void)setPredicate:(NSPredicate *)*predicate*

Parameters

predicate

The predicate for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [predicate](#) (page 106)

Declared In

NSFetchRequest.h

setPropertiesToFetch:

Specifies which properties should be returned by the fetch.

- (void)setPropertiesToFetch:(NSArray *)*values*

Parameters

values

An array of `NSPropertyDescription` objects that specify which properties should be returned by the fetch.

Discussion

The property descriptions may represent attributes, to-one relationships, or expressions. The name of an attribute or relationship description must match the name of a description on the fetch request's entity.

Availability

Available in iOS 3.0 and later.

See Also

- [propertiesToFetch](#) (page 106)
- [resultType](#) (page 107)
- [returnsDistinctResults](#) (page 108)

Declared In

`NSFetchRequest.h`

setRelationshipKeyPathsForPrefetching:

Sets an array of relationship keypaths to prefetch along with the entity for the request.

- (void)setRelationshipKeyPathsForPrefetching:(NSArray *)*keys*

Parameters

keys

An array of relationship key-path strings in `NSKeyValueCoding` notation (as you would normally use with `valueForKeyPath:`).

Discussion

For a full discussion, see [relationshipKeyPathsForPrefetching](#) (page 107).

Availability

Available in iOS 3.0 and later.

See Also

- [relationshipKeyPathsForPrefetching](#) (page 107)

Declared In

`NSFetchRequest.h`

setResultType:

Sets the result type of the receiver.

- (void)setResultType:(NSFetchRequestResultType)*type*

Parameters*type*

The result type of the receiver.

Discussion

For further discussion, see [resultType](#) (page 107).

Availability

Available in iOS 3.0 and later.

See Also

- [resultType](#) (page 107)

Declared In

NSFetchRequest.h

setReturnsDistinctResults:

Sets whether the request should return only distinct values for the fields specified by `propertiesToFetch`.

```
- (void)setReturnsDistinctResults:(BOOL)values
```

Parameters*values*

If YES, the request returns only distinct values for the fields specified by `propertiesToFetch`.

Discussion

For a full discussion, see [returnsDistinctResults](#) (page 108).

Special Considerations

This value is only used if a value has been set for `propertiesToFetch`.

Availability

Available in iOS 3.0 and later.

See Also

- [returnsDistinctResults](#) (page 108)

- [propertiesToFetch](#) (page 106)

Declared In

NSFetchRequest.h

setReturnsObjectsAsFaults:

Sets whether the objects resulting from a fetch request are faults.

```
- (void)setReturnsObjectsAsFaults:(BOOL)yesNo
```

Parameters*yesNo*

If `NO`, the objects returned from the fetch are pre-populated with their property values (making them fully-faulted objects, which will immediately return `NO` if sent the `isFault` (page 137) message). If `YES`, the objects returned from the fetch are not pre-populated (and will receive a `didFireFault` message when the properties are accessed the first time).

Discussion

For a full discussion, see [returnsObjectsAsFaults](#) (page 108).

Availability

Available in iOS 3.0 and later.

See Also

- [returnsObjectsAsFaults](#) (page 108)

Declared In

`NSFetchRequest.h`

setSortDescriptors:

Sets the array of sort descriptors of the receiver.

```
- (void)setSortDescriptors:(NSArray *)sortDescriptors
```

Parameters*sortDescriptors*

The array of sort descriptors of the receiver. `nil` specifies no sort descriptors.

Availability

Available in iOS 3.0 and later.

See Also

- [sortDescriptors](#) (page 115)

Declared In

`NSFetchRequest.h`

sortDescriptors

Returns the sort descriptors of the receiver.

```
- (NSArray *)sortDescriptors
```

Return Value

The sort descriptors of the receiver.

Discussion

The sort descriptors specify how the objects returned when the fetch request is issued should be ordered—for example by last name then by first name. The sort descriptors are applied in the order in which they appear in the *sortDescriptors* array (serially in lowest-array-index-first order).

Availability

Available in iOS 3.0 and later.

See Also

- [setSortDescriptors:](#) (page 115)

Declared In

NSFetchRequest.h

Constants

NSFetchRequestResultType

These constants specify the possible result types a fetch request can return.

```
enum {
    NSManagedObjectResultType      = 0x00,
    NSManagedObjectIDResultType    = 0x01,
    NSDictionaryResultType          = 0x02
};
typedef NSUInteger NSFetchRequestResultType;
```

Constants

NSManagedObjectResultType

Specifies that the request returns managed objects.

Available in iOS 3.0 and later.

Declared in `NSFetchRequest.h`.

NSManagedObjectIDResultType

Specifies that the request returns managed object IDs.

Available in iOS 3.0 and later.

Declared in `NSFetchRequest.h`.

NSDictionaryResultType

Specifies that the request returns dictionaries.

Available in iOS 3.0 and later.

Declared in `NSFetchRequest.h`.

Discussion

These constants are used by [resultType](#) (page 107).

NSFetchRequestExpression Class Reference

Inherits from	NSExpression : NSObject
Conforms to	NSCoding (NSExpression) NSCopying (NSExpression) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSFetchRequestExpression.h
Companion guides	Core Data Programming Guide Predicate Programming Guide

Overview

Instances of `NSFetchRequestExpression` represent expressions which evaluate to the result of executing a fetch request on a managed object context.

`NSFetchRequestExpression` inherits from `NSExpression`, which provides most of the basic behavior. The first argument must be an expression which evaluates to an `NSFetchRequest` object, and the second must be an expression which evaluates to an `NSManagedObjectContext` object. If you simply want the count for the request, the `countOnly` argument should be YES.

Tasks

Creating a Fetch Request Expression

+ [expressionForFetch:context:countOnly:](#) (page 118)

Returns an expression which will evaluate to the result of executing a fetch request on a context.

Examining a Fetch Request Expression

- [requestExpression](#) (page 119)

Returns the expression for the receiver's fetch request.

- [contextExpression](#) (page 118)
Returns the expression for the receiver's managed object context.
- [isCountOnlyRequest](#) (page 119)
Returns a Boolean value that indicates whether the receiver represents a count-only fetch request.

Class Methods

expressionForFetch:context:countOnly:

Returns an expression which will evaluate to the result of executing a fetch request on a context.

```
+ (NSEExpression *)expressionForFetch:(NSEExpression *)fetch context:(NSEExpression *)context countOnly:(BOOL)countFlag
```

Parameters

fetch

An expression that evaluates to an instance of `NSFetchRequest`.

context

An expression that evaluates to an instance of `NSManagedObjectContext`.

countFlag

If YES, when the new expression is evaluated the managed object context (from *context*) will perform `countFetchRequest:error:` (page 158) with the fetch request (from *fetch*). If NO, when the new expression is evaluated the managed object context will perform `executeFetchRequest:error:` (page 160) with the fetch request.

Return Value

An expression which will evaluate to the result of executing a fetch request (from *fetch*) on a managed object context (from *context*).

Availability

Available in iOS 3.0 and later.

Declared In

`NSFetchRequestExpression.h`

Instance Methods

contextExpression

Returns the expression for the receiver's managed object context.

```
- (NSEExpression *)contextExpression
```

Return Value

The expression for the receiver's managed object context. Evaluating the expression must return an `NSManagedObjectContext` object.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchRequestExpression.h

isCountOnlyRequest

Returns a Boolean value that indicates whether the receiver represents a count-only fetch request.

- (BOOL)isCountOnlyRequest

Return Value

YES if the receiver represents a count-only fetch request, otherwise NO.

Discussion

If this method returns NO, the managed object context (from the [contextExpression](#) (page 118)) will perform [executeFetchRequest:error:](#) (page 160): with the [requestExpression](#) (page 119); if this method returns YES, the managed object context will perform [countForFetchRequest:error:](#) (page 158) with the [requestExpression](#) (page 119).

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchRequestExpression.h

requestExpression

Returns the expression for the receiver's fetch request.

- (NSExpression *)requestExpression

Return Value

The expression for the receiver's fetch request. Evaluating the expression must return an `NSFetchRequest` object.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchRequestExpression.h

Constants

Fetch request expression type

This constant specifies the fetch request expression type.

```
enum {  
    NSFetchRequestExpressionType = 50  
};
```

Constants

NSFetchRequestExpressionType

Specifies the fetch request expression type.

Available in iOS 3.0 and later.

Declared in `NSFetchRequestExpression.h`.

Declared In

`NSFetchRequestExpression.h`

NSManagedObject Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSManagedObject.h
Companion guides	Core Data Programming Guide Model Object Implementation Guide Core Data Utility Tutorial

Overview

`NSManagedObject` is a generic class that implements all the basic behavior required of a Core Data model object. It is not possible to use instances of direct subclasses of `NSObject` (or any other class not inheriting from `NSManagedObject`) with a managed object context. You may create custom subclasses of `NSManagedObject`, although this is not always required. If no custom logic is needed, a complete object graph can be formed with `NSManagedObject` instances.

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. It is important that a managed object is properly configured for use with Core Data. If you instantiate a managed object directly, you must call the designated initializer (`initWithEntity:insertIntoManagedObjectContext:` (page 135)).

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. Sometimes, however, 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. For some types you can use a transformable attribute, for others this may require you to create a subclass of `NSManagedObject`—see “Non-Standard Persistent Attributes”.

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” 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.

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:`, `respondToSelector:`, `retain`, `release`, `autorelease`, `retainCount`, `description`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, `faultingState`, 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 the fault.

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.

Subclassing Notes

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, however, many reasons why you might wish to subclass `NSManagedObject` to implement custom features. It is important, though, not to disrupt Core Data’s behavior.

Methods you Must Not Override

`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 therefore absolutely must not override: `primitiveValueForKey:`, `setPrimitiveValue:forKey:`, `isEqual:`, `hash`, `superclass`, `class`, `self`, `zone`, `isProxy`, `isKindOfClass:`, `isMemberOfClass:`, `conformsToProtocol:`, `respondToSelector:`, `retain`, `release`, `autorelease`, `retainCount`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, and `isFault`.

In addition to the methods listed above, on Mac OS X v10.5, you must not override: `alloc`, `allocWithZone:`, `new`, `instancesRespondToSelector:`, `instanceMethodForSelector:`, `methodForSelector:`, `methodSignatureForSelector:`, `instanceMethodSignatureForSelector:`, or `isSubclassOfClass:`.

Methods you Are Discouraged From Overriding

As with any class, you are strongly discouraged from overriding the key-value observing methods such as `willChangeValueForKey:` and `didChangeValueForKey:withSetMutation:usingObjects:`. You are discouraged from overriding `description`—if this method fires a fault during a debugging operation, the results may be unpredictable. You are also discouraged from overriding `initWithEntity:insertIntoManagedObjectContext:`, `dealloc`, or `finalize`. Changing values in the `initWithEntity:insertIntoManagedObjectContext:` method will not be noticed by the context

and if you are not careful, those changes may not be saved. Most initialization customization should be performed in one of the `awake...` methods. If you do override

`initWithEntity:insertIntoManagedObjectContext:`, you must make sure you adhere to the requirements set out in the method description (see [initWithEntity:insertIntoManagedObjectContext:](#) (page 135)).

You are discouraged from overriding `dealloc` or `finalize` because `didTurnIntoFault` is usually a better time to clear values—a managed object may not be reclaimed for some time after it has been turned into a fault. Core Data does not guarantee that either `dealloc` or `finalize` will be called in all scenarios (such as when the application quits); you should therefore not in these methods include required side effects (like saving or changes to the file system, user preferences, and so on).

In summary, for `initWithEntity:insertIntoManagedObjectContext:`, `dealloc`, and `finalize` it is important to remember that Core Data reserves exclusive control over the life cycle of the managed object (that is, raw memory management). This is so that the framework is able to provide features such as uniquing and by consequence relationship maintenance as well as much better performance than would be otherwise possible.

Methods to Override Considerations

The following methods are intended to be fine grained and not perform large scale operations. You must not fetch or save in these methods. In particular, they should not have side effects on the managed object context:

- `initWithEntity:insertIntoManagedObjectContext:`
- `didTurnIntoFault`
- `willTurnIntoFault`
- `dealloc`
- `finalize`

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. Note that you should not modify relationships in [awakeFromFetch](#) (page 128)—see the method description for details.

Custom Accessor Methods

Typically, there is no need to write custom accessor methods for properties that are defined in the entity of a managed object's corresponding managed object model. Should you wish or need to do so, though, there are several implementation patterns you must follow. These are described in "Managed Object Accessor Methods" in *Core Data Programming Guide*.

On Mac OS X v10.5, Core Data automatically generates accessor methods (and primitive accessor methods) for you. For attributes and to-one relationships, Core Data generates the standard get and set accessor methods; for to-many relationships, Core Data generates the indexed accessor methods as described in "Key-Value Coding Accessor Methods" in *Key-Value Coding Programming Guide*. You do however need to declare the accessor methods or use Objective-C properties to suppress compiler warnings. For a full discussion, see "Managed Object Accessor Methods" in *Core Data Programming Guide*.

On Mac OS X v10.4, you can access properties using standard key-value coding methods such as `valueForKey:`. It may, however, be convenient to implement custom accessors to benefit from compile-time type checking and to avoid errors with misspelled key names.

Custom Instance Variables

By default, `NSManagedObject` stores its properties in an internal structure as objects, and in general Core Data is more efficient working with storage under its own control rather using custom instance variables.

`NSManagedObject` provides support for a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). If you want to use types that are not supported directly, such as colors and C structures, you can either use transformable attributes or create a subclass of `NSManagedObject`, as described in “Non-Standard Persistent Attributes”.

Sometimes it may be convenient to represent variables as scalars—in a drawing applications, for example, where variables represent dimensions and x and y coordinates and are frequently used in calculations. To represent attributes as scalars, you declare instance variables as you would in any other class. You also need to implement suitable accessor methods as described in “Managed Object Accessor Methods”.

If you define custom instance variables, for example, to store derived attributes or other transient properties, you should clean up these variables in `didTurnIntoFault` (page 134) rather than `dealloc`.

Validation Methods

`NSManagedObject` provides consistent hooks for validating property and inter-property values. You typically should not override `validateValue:forKey:error:` (page 146), 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:` (page 145) 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:` (page 145)), 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. For an example, see “Model Object Validation”.

Tasks

Initializing a Managed Object

- `initWithEntity:insertIntoManagedObjectContext:` (page 135)
Initializes the receiver and inserts it into the specified managed object context.

Getting a Managed Object's Identity

- [entity](#) (page 134)
Returns the entity description of the receiver.
- [objectID](#) (page 140)
Returns the object ID of the receiver.
- [self](#) (page 142)
Returns the receiver.

Getting State Information

- [managedObjectContext](#) (page 139)
Returns the managed object context with which the receiver is registered.
- [isInserted](#) (page 138)
Returns a Boolean value that indicates whether the receiver has been inserted in a managed object context.
- [isUpdated](#) (page 138)
Returns a Boolean value that indicates whether the receiver has unsaved changes.
- [isDeleted](#) (page 136)
Returns a Boolean value that indicates whether the receiver will be deleted during the next save.
- [isFault](#) (page 137)
Returns a Boolean value that indicates whether the receiver is a fault.
- [faultingState](#) (page 134)
Returns a value that indicates the faulting state of the receiver.
- [hasFaultForRelationshipNamed:](#) (page 135)
Returns a Boolean value that indicates whether the relationship for a given key is a fault.

Managing Life Cycle and Change Events

- + [contextShouldIgnoreUnmodeledPropertyChanges](#) (page 128)
Returns a Boolean value that indicates whether instances of the class should be marked as having changes if an unmodeled property is changed.
- [awakeFromFetch](#) (page 128)
Invoked automatically by the Core Data framework after the receiver has been fetched.
- [awakeFromInsert](#) (page 129)
Invoked automatically by the Core Data framework when the receiver is first inserted into a managed object context.
- [awakeFromSnapshotEvents:](#) (page 129)
Invoked automatically by the Core Data framework when the receiver is reset due to an undo, redo, or other multi-property state change.
- [changedValues](#) (page 130)
Returns a dictionary containing the keys and (new) values of persistent properties that have been changed since last fetching or saving the receiver.

- [committedValuesForKeys:](#) (page 131)
Returns a dictionary of the last fetched or saved values of the receiver for the properties specified by the given keys.
- [prepareForDeletion](#) (page 140)
Invoked automatically by the Core Data framework when the receiver is about to be deleted.
- [dealloc](#) (page 131)
Deallocates the memory occupied by the receiver.
- [willSave](#) (page 149)
Invoked automatically by the Core Data framework when the receiver's managed object context is saved.
- [didSave](#) (page 133)
Invoked automatically by the Core Data framework after the receiver's managed object context completes a save operation.
- [willTurnIntoFault](#) (page 150)
Invoked automatically by the Core Data framework before receiver is converted to a fault.
- [didTurnIntoFault](#) (page 134)
Invoked automatically by the Core Data framework when the receiver is turned into a fault.

Supporting Key-Value Coding

- [valueForKey:](#) (page 147)
Returns the value for the property specified by *key*.
- [setValue:forKey:](#) (page 144)
Sets the specified property of the receiver to the specified value.
- [mutableSetValueForKey:](#) (page 139)
Returns a mutable set that provides read-write access to the unordered to-many relationship specified by a given key.
- [primitiveValueForKey:](#) (page 141)
Returns from the receiver's private internal storage the value for the specified property.
- [setPrimitiveValue:forKey:](#) (page 142)
Sets in the receiver's private internal storage the value of a given property.

Validation

- [validateValue:forKey:error:](#) (page 146)
Validates a property value for a given key.
- [validateForDelete:](#) (page 144)
Determines whether the receiver can be deleted in its current state.
- [validateForInsert:](#) (page 145)
Determines whether the receiver can be inserted in its current state.
- [validateForUpdate:](#) (page 145)
Determines whether the receiver's current state is valid.

Supporting Key-Value Observing

- + [automaticallyNotifiesObserversForKey:](#) (page 127)
Returns a Boolean value that indicates whether the receiver provides automatic support for key-value observing change notifications for the given key.
- [didAccessValueForKey:](#) (page 131)
Provides support for key-value observing access notification.
- [observationInfo](#) (page 140)
Returns the observation info of the receiver.
- [setObservationInfo:](#) (page 142)
Sets the observation info of the receiver.
- [willAccessValueForKey:](#) (page 148)
Provides support for key-value observing access notification.
- [didChangeValueForKey:](#) (page 132)
Invoked to inform the receiver that the value of a given property has changed.
- [didChangeValueForKey:withSetMutation:usingObjects:](#) (page 133)
Invoked to inform the receiver that the specified change was made to a specified to-many relationship.
- [willChangeValueForKey:](#) (page 148)
Invoked to inform the receiver that the value of a given property is about to change.
- [willChangeValueForKey:withSetMutation:usingObjects:](#) (page 148)
Invoked to inform the receiver that the specified change is about to be made to a specified to-many relationship.

Class Methods

automaticallyNotifiesObserversForKey:

Returns a Boolean value that indicates whether the receiver provides automatic support for key-value observing change notifications for the given key.

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString *)key
```

Parameters

key

The name of one of the receiver's properties.

Return Value

YES if the receiver provides automatic support for key-value observing change notifications for *key*, otherwise NO.

Discussion

The default implementation for `NSManagedObject` returns NO for modeled properties, and YES for unmodeled properties. For more about key-value observation, see *Key-Value Observing Programming Guide*.

Special Considerations

On Mac OS X v10.4, this method returns NO for all properties.

contextShouldIgnoreUnmodeledPropertyChanges

Returns a Boolean value that indicates whether instances of the class should be marked as having changes if an unmodeled property is changed.

+ (BOOL)contextShouldIgnoreUnmodeledPropertyChanges

Return Value

YES if instances of the class should be marked as having changes if an unmodeled property is changed, otherwise NO.

Discussion

For programs targeted at Mac OS X v10.5 and earlier, the default value is NO. For programs targeted at Mac OS X v10.6 and later, the default value is YES.

Availability

Available in iOS 3.0 and later.

See Also

- [changedValues](#) (page 130)
[hasChanges](#) (page 161) (NSManagedObjectContext)

Declared In

NSManagedObject.h

Instance Methods

awakeFromFetch

Invoked automatically by the Core Data framework after the receiver has been fetched.

- (void)awakeFromFetch

Discussion

You typically use this method to compute derived values or to recreate transient relationships from the receiver's persistent properties.

The managed object context's change processing is explicitly disabled around this method so that you can use public setters to establish transient values and other caches without dirtying the object or its context. Because of this, however, you should not modify relationships in this method as the inverse will not be set.

Important: Subclasses must invoke super's implementation before performing their own initialization.

Availability

Available in iOS 3.0 and later.

See Also

- [awakeFromInsert](#) (page 129)
- [awakeFromSnapshotEvents:](#) (page 129)
- [primitiveValueForKey:](#) (page 141)
- [setPrimitiveValue:forKey:](#) (page 142)

Declared In

NSManagedObject.h

awakeFromInsert

Invoked automatically by the Core Data framework when the receiver is first inserted into a managed object context.

```
- (void)awakeFromInsert
```

Discussion

You typically use this method to initialize special default property values. This method is invoked only once in the object's lifetime.

If you want to set attribute values in an implementation of this method, you should typically use primitive accessor methods (either `setPrimitiveValue:forKey:` (page 142) or—better—the appropriate custom primitive accessors). This ensures that the new values are treated as baseline values rather than being recorded as undoable changes for the properties in question.

Important: Subclasses must invoke super's implementation before performing their own initialization.

Special Considerations

If you create a managed object then perform undo operations to bring the managed object context to a state prior to the object's creation, then perform redo operations to bring the managed object context back to a state after the object's creation, `awakeFromInsert` is *not* invoked a second time.

You are typically discouraged from performing fetches within an implementation of `awakeFromInsert`. Although it is allowed, execution of the fetch request can trigger the sending of internal Core Data notifications which may have unwanted side-effects. For example, on Mac OS X, an instance of `NSArrayController` may end up inserting a new object into its content array twice.

Availability

Available in iOS 3.0 and later.

See Also

- [awakeFromFetch](#) (page 128)
- [awakeFromSnapshotEvents:](#) (page 129)

Declared In

NSManagedObject.h

awakeFromSnapshotEvents:

Invoked automatically by the Core Data framework when the receiver is reset due to an undo, redo, or other multi-property state change.

```
- (void)awakeFromSnapshotEvents:(NSSnapshotEventType)flags
```

Parameters*flags*

A bitmask of `didChangeValueForKey:` (page 132) constants to denote the event or events that led to the method being invoked.

For possible values, see “`NSSnapshotEventType`” (page 150).

Discussion

You typically use this method to compute derived values or to recreate transient relationships from the receiver’s persistent properties.

If you want to set attribute values and need to avoid emitting key-value observation change notifications, you should use primitive accessor methods (either `setPrimitiveValue:forKey:` (page 142) or—better—the appropriate custom primitive accessors). This ensures that the new values are treated as baseline values rather than being recorded as undoable changes for the properties in question.

Important: Subclasses must invoke super’s implementation before performing their own initialization.

Availability

Available in iOS 3.0 and later.

See Also

- [awakeFromFetch](#) (page 128)
- [awakeFromInsert](#) (page 129)

Declared In

`NSManagedObject.h`

changedValues

Returns a dictionary containing the keys and (new) values of persistent properties that have been changed since last fetching or saving the receiver.

- (`NSDictionary *`)`changedValues`

Return Value

A dictionary containing as keys the names of persistent properties that have changed since the receiver was last fetched or saved, and as values the new values of the properties.

Discussion

Note that this method only reports changes to properties that are defined as persistent properties of the receiver, not changes to transient properties or custom instance variables. This method does not unnecessarily fire relationship faults.

Availability

Available in iOS 3.0 and later.

See Also

- [committedValuesForKeys:](#) (page 131)

Declared In

`NSManagedObject.h`

committedValuesForKeys:

Returns a dictionary of the last fetched or saved values of the receiver for the properties specified by the given keys.

```
- (NSDictionary *)committedValuesForKeys:(NSArray *)keys
```

Parameters

keys

An array containing names of properties of the receiver, or `nil`.

Return Value

A dictionary containing the last fetched or saved values of the receiver for the properties specified by *keys*.

Discussion

This method only reports values of properties that are defined as persistent properties of the receiver, not values of transient properties or of custom instance variables.

You can invoke this method with the *keys* value of `nil` to retrieve committed values for all the receiver's properties, as illustrated by the following example.

```
NSDictionary *allCommittedValues =  
    [aManagedObject committedValuesForKeys:nil];
```

It is more efficient to use `nil` than to pass an array of all the property keys.

Availability

Available in iOS 3.0 and later.

See Also

- [changedValues](#) (page 130)

Declared In

`NSManagedObject.h`

dealloc

Deallocates the memory occupied by the receiver.

```
- (void)dealloc
```

Discussion

This method first invokes [didTurnIntoFault](#) (page 134).

You should typically not override this method—instead you should put “clean-up” code in [prepareForDeletion](#) (page 140) or [didTurnIntoFault](#) (page 134).

See Also

- [prepareForDeletion](#) (page 140)

- [didTurnIntoFault](#) (page 134)

didAccessValueForKey:

Provides support for key-value observing access notification.

```
- (void)didAccessValueForKey:(NSString *)key
```

Parameters

key

The name of one of the receiver's properties.

Discussion

Together with [willAccessValueForKey:](#) (page 148), this method is used to fire faults, to maintain inverse relationships, and so on. Each read access must be wrapped in this method pair (in the same way that each write access must be wrapped in the `willChangeValueForKey:/didChangeValueForKey:` method pair). In the default implementation of `NSManagedObject` these methods are invoked for you automatically. If, say, you create a custom subclass that uses explicit instance variables, you must invoke them yourself, as in the following example.

```
- (NSString *)firstName
{
    [self willAccessValueForKey:@"firstName"];
    NSString *rtn = firstName;
    [self didAccessValueForKey:@"firstName"];
    return rtn;
}
```

Availability

Available in iOS 3.0 and later.

See Also

- [willAccessValueForKey:](#) (page 148)

Declared In

`NSManagedObject.h`

didChangeValueForKey:

Invoked to inform the receiver that the value of a given property has changed.

```
- (void)didChangeValueForKey:(NSString *)key
```

Parameters

key

The name of the property that changed.

Discussion

For more details, see *Key-Value Observing Programming Guide*.

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObject.h`

didChangeValueForKey:withSetMutation:usingObjects:

Invoked to inform the receiver that the specified change was made to a specified to-many relationship.

```
- (void)didChangeValueForKey:(NSString *)inKey
    withSetMutation:(NSKeyValueSetMutationKind)inMutationKind usingObjects:(NSSet *)inObjects
```

Parameters

inKey

The name of a property that is a to-many relationship.

inMutationKind

The type of change that was made.

inObjects

The objects that were involved in the change (see `NSKeyValueSetMutationKind`).

Discussion

For more details, see *Key-Value Observing Programming Guide*.

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObject.h`

didSave

Invoked automatically by the Core Data framework after the receiver's managed object context completes a save operation.

```
- (void)didSave
```

Discussion

You can use this method to notify other objects after a save, and to compute transient values from persistent values.

This method can have “side effects” on the persistent values, however note that any changes you make using standard accessor methods will by default dirty the managed object context and leave your context with unsaved changes. Moreover, if the object's context has an undo manager, such changes will add an undo operation. For document-based applications, changes made in `didSave` will therefore come into the next undo grouping, which can lead to “empty” undo operations from the user's perspective. You may want to disable undo registration to avoid this issue.

Note that the sense of “save” in the method name is that of a database commit statement and so applies to deletions as well as to updates to objects. For subclasses, this method is therefore an appropriate locus for code to be executed when an object deleted as well as “saved to disk.” You can find out if an object is marked for deletion with `isDeleted` (page 136).

Special Considerations

You cannot attempt to resurrect a deleted object in `didSave`.

Availability

Available in iOS 3.0 and later.

See Also

- [willSave](#) (page 149)

Declared In

NSManagedObject.h

didTurnIntoFault

Invoked automatically by the Core Data framework when the receiver is turned into a fault.

- (void)didTurnIntoFault

Discussion

You use this method to clear out custom data caches—transient values declared as entity properties are typically already cleared out by the time this method is invoked (see, for example, [refreshObject:mergeChanges:](#) (page 167)).

Availability

Available in iOS 3.0 and later.

See Also

- [willTurnIntoFault](#) (page 150)

Declared In

NSManagedObject.h

entity

Returns the entity description of the receiver.

- (NSEntityDescription *)entity

Return Value

The entity description of the receiver.

Discussion

If the receiver is a fault, calling this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObject.h

faultingState

Returns a value that indicates the faulting state of the receiver.

- (NSInteger)faultingState

Return Value

0 if the object is fully initialized as a managed object and not transitioning to or from another state, otherwise some other value.

Discussion

The method allow you to determine if an object is in a transitional phase when receiving a key-value observing change notification.

Availability

Available in iOS 3.0 and later.

See Also

- [isFault](#) (page 137)

Declared In

NSManagedObject.h

hasFaultForRelationshipNamed:

Returns a Boolean value that indicates whether the relationship for a given key is a fault.

```
- (BOOL)hasFaultForRelationshipNamed:(NSString *)key
```

Parameters

key

The name of one of the receiver's relationships.

Return Value

YES if the relationship for for the key *key* is a fault, otherwise NO.

Discussion

If the specified relationship is a fault, calling this method does not result in the fault firing.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObject.h

initWithEntity:insertIntoManagedObjectContext:

Initializes the receiver and inserts it into the specified managed object context.

```
- (id)initWithEntity:(NSEntityDescription *)entity
    insertIntoManagedObjectContext:(NSManagedObjectContext *)context
```

Parameters

entity

The entity of which to create an instance.

The model associated with *context*'s persistent store coordinator must contain *entity*.

context

The context into which the new instance is inserted.

Return Value

An initialized instance of the appropriate class for *entity*.

Discussion

`NSManagedObject` uses dynamic class generation to support the Objective-C 2 properties feature (see “Declared Properties”) by automatically creating a subclass of the class appropriate for *entity*. `initWithEntity:insertIntoManagedObjectContext:` therefore returns an instance of the appropriate class for *entity*. The dynamically-generated subclass will be based on the class specified by the entity, so specifying a custom class in your model will supersede the class passed to `alloc`.

If *context* is not `nil`, this method invokes `[context insertObject:self]` (which causes [awakeFromInsert](#) (page 129) to be invoked).

You are discouraged from overriding this method—you should instead override [awakeFromInsert](#) (page 129) and/or [awakeFromFetch](#) (page 128) (if there is logic common to these methods, it should be factored into a third method which is invoked from both). If you do perform custom initialization in this method, you may cause problems with undo and redo operations.

In many applications, there is no need to subsequently assign a newly-created managed object to a particular store—see [assignObject:toPersistentStore:](#) (page 157). If your application has multiple stores and you do need to assign an object to a specific store, you should not do so in a managed object's initializer method. Such an assignment is controller- not model-level logic.

Important: This method is the designated initializer for `NSManagedObject`. You must not initialize a managed object simply by sending it `init`.

Special Considerations

If you override `initWithEntity:insertIntoManagedObjectContext:`, you *must* ensure that you set `self` to the return value from invocation of super's implementation, as shown in the following example:

```
- (id)initWithEntity:(NSEntityDescription*)entity
insertIntoManagedObjectContext:(NSManagedObjectContext*)context
{
    self = [super initWithEntity:entity insertIntoManagedObjectContext:context];
    if (self != nil) {
        // Perform additional initialization.
    }
    return self;
}
```

Availability

Available in iOS 3.0 and later.

See Also

[insertNewObjectForEntityForName:inManagedObjectContext:](#) (page 41)

Declared In

`NSManagedObject.h`

isDeleted

Returns a Boolean value that indicates whether the receiver will be deleted during the next save.

- (BOOL)isDeleted

Return Value

YES if the receiver will be deleted during the next save, otherwise NO.

Discussion

The method returns YES if Core Data will ask the persistent store to delete the object during the next save operation. It may return NO at other times, particularly after the object has been deleted. The immediacy with which it will stop returning YES depends on where the object is in the process of being deleted.

If the receiver is a fault, invoking this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

See Also

- [isFault](#) (page 137)

- [isInserted](#) (page 138)

- [isUpdated](#) (page 138)

[deletedObjects](#) (page 158) (NSManagedObjectContext)

[NSManagedObjectContextObjectsDidChangeNotification](#) (page 179) (NSManagedObjectContext)

Declared In

NSManagedObject.h

isFault

Returns a Boolean value that indicates whether the receiver is a fault.

- (BOOL)isFault

Return Value

YES if the receiver is a fault, otherwise NO.

Discussion

Knowing whether an object is a fault is useful in many situations when computations are optional. It can also be used to avoid growing the object graph unnecessarily (which may improve performance as it can avoid time-consuming fetches from data stores).

If this method returns NO, then the receiver's data must be in memory. However, if this method 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.

If the receiver is a fault, calling this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

See Also

- [faultingState](#) (page 134)

- [isDeleted](#) (page 136)

- [isInserted](#) (page 138)

- [isUpdated](#) (page 138)

Declared In

NSManagedObject.h

isInserted

Returns a Boolean value that indicates whether the receiver has been inserted in a managed object context.

- (BOOL)isInserted

Return Value

YES if the receiver has been inserted in a managed object context, otherwise NO.

Discussion

If the receiver is a fault, calling this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

See Also

- [isDeleted](#) (page 136)
- [isFault](#) (page 137)
- [isUpdated](#) (page 138)

Declared In

NSManagedObject.h

isUpdated

Returns a Boolean value that indicates whether the receiver has unsaved changes.

- (BOOL)isUpdated

Return Value

YES if the receiver has unsaved changes, otherwise NO.

Discussion

The receiver has unsaved changes if it has been updated since its managed object context was last saved.

If the receiver is a fault, calling this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

See Also

- [isDeleted](#) (page 136)
- [isFault](#) (page 137)
- [isInserted](#) (page 138)

Declared In

NSManagedObject.h

managedObjectContext

Returns the managed object context with which the receiver is registered.

- (NSManagedObjectContext *)managedObjectContext

Return Value

The managed object context with which the receiver is registered.

Discussion

This method may return `nil` if the receiver has been deleted from its context.

If the receiver is a fault, calling this method does not cause it to fire.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObject.h

mutableSetValueForKey:

Returns a mutable set that provides read-write access to the unordered to-many relationship specified by a given key.

- (NSMutableSet *)mutableSetValueForKey:(NSString *)key

Parameters

key

The name of one of the receiver's to-many relationships.

Discussion

If *key* is not a property defined by the model, the method raises an exception.

This method is overridden by `NSManagedObject` to access the managed object's generic dictionary storage unless the receiver's class explicitly provides key-value coding compliant accessor methods for *key*.

Important: You must not override this method.

Special Considerations

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 (see "Managed Object Accessor Methods" in *Core Data Programming Guide*).

See Also

- [valueForKey:](#) (page 147)
- [primitiveValueForKey:](#) (page 141)
- [setObservationInfo:](#) (page 142)

objectID

Returns the object ID of the receiver.

- (NSNumber *)objectID

Return Value

The object ID of the receiver.

Discussion

If the receiver is a fault, calling this method does not cause it to fire.

Important: If the receiver has not yet been saved, the object ID is a temporary value that will change when the object is saved.

Availability

Available in iOS 3.0 and later.

See Also

[URIRepresentation](#) (page 183) (NSManagedObjectID)

Declared In

NSManagedObject.h

observationInfo

Returns the observation info of the receiver.

- (id)observationInfo

Return Value

The observation info of the receiver.

Discussion

For more about observation information, see *Key-Value Observing Programming Guide*.

Important: You must not override this method.

Availability

Available in iOS 3.0 and later.

See Also

- [setObservationInfo:](#) (page 142)

Declared In

NSManagedObject.h

prepareForDeletion

Invoked automatically by the Core Data framework when the receiver is about to be deleted.

- (void)prepareForDeletion

Discussion

You can implement this method to perform any operations required before the object is deleted, such as custom propagation before relationships are torn down, or reconfiguration of objects using key-value observing.

Availability

Available in iOS 3.0 and later.

See Also

- [willTurnIntoFault](#) (page 150)
- [didTurnIntoFault](#) (page 134)

Declared In

NSManagedObject.h

primitiveValueForKey:

Returns from the receiver's private internal storage the value for the specified property.

- (id)primitiveValueForKey:(NSString *)key

Parameters

key

The name of one of the receiver's properties.

Return Value

The value of the property specified by *key*. Returns *nil* if no value has been set.

Discussion

This method does not invoke the access notification methods ([willAccessValueForKey:](#) (page 148) and [didAccessValueForKey:](#) (page 131)). This method is used primarily by subclasses that implement custom accessor methods that need direct access to the receiver's private storage.

Special Considerations

Subclasses should not override this method.

On Mac OS X v10.5 and later, the following points also apply:

- Primitive accessor methods are only supported on *modeled* properties. If you invoke a primitive accessor on an unmodeled property, it will instead operate upon a random modeled property. (The debug libraries and frameworks (available from [Apple Developer website](#)) have assertions to test for passing unmodeled keys to these methods.)
- You are strongly encouraged to use the dynamically-generated accessors rather than using this method directly (for example, `primitiveName:` instead of `primitiveValueForKey:@"name"`). The dynamic accessors are much more efficient, and allow for compile-time checking.

Availability

Available in iOS 3.0 and later.

See Also

- [setObservationInfo:](#) (page 142)

- [valueForKey:](#) (page 147)
- [mutableSetValueForKey:](#) (page 139)

Declared In

NSManagedObject.h

self

Returns the receiver.

- (id)self

Discussion

Subclasses must not override this method.

Note for EOF developers: Core Data does not rely on this method for faulting—see instead [willAccessValueForKey:](#) (page 148).

setObservationInfo:

Sets the observation info of the receiver.

- (void)setObservationInfo:(id)value

Parameters*value*

The new observation info for the receiver.

DiscussionFor more about observation information, see *Key-Value Observing Programming Guide*.**Availability**

Available in iOS 3.0 and later.

See Also

- [observationInfo](#) (page 140)

Declared In

NSManagedObject.h

setPrimitiveValue:forKey:

Sets in the receiver's private internal storage the value of a given property.

- (void)setPrimitiveValue:(id)value forKey:(NSString *)key

Parameters*value*The new value for the property specified by *key*.*key*

The name of one of the receiver's properties.

Discussion

Sets in the receiver's private internal storage the value of the property specified by *key* to *value*. If *key* identifies a to-one relationship, relates the object specified by *value* to the receiver, unrelating the previously related object if there was one. Given a collection object and a key that identifies a to-many relationship, relates the objects contained in the collection to the receiver, unrelating previously related objects if there were any.

This method does not invoke the change notification methods (`willChangeValueForKey:` and `didChangeValueForKey:`). It is typically used by subclasses that implement custom accessor methods that need direct access to the receiver's private internal storage. It is also used by the Core Data framework to initialize the receiver with values from a persistent store or to restore a value from a snapshot.

Special Considerations

You must not override this method.

You should typically use this method only to modify attributes (usually transient), not relationships. If you try to set a to-many relationship to a new `NSMutableSet` object, it will (eventually) fail. In the unusual event that you need to modify a relationship using this method, you first get the existing set using `primitiveValueForKey:` (ensure the method does not return `nil`), create a mutable copy, and then modify the copy—as illustrated in the following example:

```
NSMutableSet *recentHires = [[dept primitiveValueForKey:@"recentHires"]
mutableCopy];
if (recentHires != nil) {
    [recentHires removeAllObjects];
    [dept setPrimitiveValue:recentHires forKey:@"recentHires"];
}
```

Note that if the relationship is bi-directional (that is, if an inverse relationship is specified) then you are also responsible for maintaining the inverse relationship (regardless of cardinality)—in contrast with Core Data's normal behavior described in "Using Managed Objects".

On Mac OS X v10.5 and later, the following points also apply:

- Primitive accessor methods are only supported on *modeled* properties. If you invoke a primitive accessor on an unmodeled property, it will instead operate upon a random modeled property. (The debug libraries and frameworks from (available from the [Apple Developer Website](#)) have assertions to test for passing unmodeled keys to these methods.)
- You are strongly encouraged to use the dynamically-generated accessors rather than using this method directly (for example, `setPrimitiveName:` instead of `setPrimitiveValue:newName forKey:@"name"`). The dynamic accessors are much more efficient, and allow for compile-time checking.

Availability

Available in iOS 3.0 and later.

See Also

- [primitiveValueForKey:](#) (page 141)
- [valueForKey:](#) (page 147)
- [mutableSetValueForKey:](#) (page 139)
- [awakeFromFetch](#) (page 128)

Declared In

`NSManagedObject.h`

setValueForKey:

Sets the specified property of the receiver to the specified value.

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

Parameters

value

The new value for the property specified by *key*.

key

The name of one of the receiver's properties.

Discussion

If *key* is not a property defined by the model, the method raises an exception. If *key* identifies a to-one relationship, relates the object specified by *value* to the receiver, unrelating the previously related object if there was one. Given a collection object and a key that identifies a to-many relationship, relates the objects contained in the collection to the receiver, unrelating previously related objects if there were any.

This method is overridden by `NSManagedObject` to access the managed object's generic dictionary storage unless the receiver's class explicitly provides key-value coding compliant accessor methods for *key*.

Important: You must not override this method.

Availability

Available in iOS 3.0 and later.

See Also

- [valueForKey:](#) (page 147)
- [primitiveValueForKey:](#) (page 141)
- [setObservationInfo:](#) (page 142)

Declared In

NSManagedObject.h

validateForDelete:

Determines whether the receiver can be deleted in its current state.

```
- (BOOL)validateForDelete:(NSError **)error
```

Parameters

error

If the receiver cannot be deleted in its current state, upon return contains an instance of `NSError` that describes the problem.

Return Value

YES if the receiver can be deleted in its current state, otherwise NO.

Discussion

An object cannot be deleted if it has a relationship has a “deny” delete rule and that relationship has a destination object.

NSManagedObject’s implementation sends the receiver’s entity description a message which performs basic checking based on the presence or absence of values.

Important: Subclasses should invoke super’s implementation before performing their own validation, and should combine any error returned by super’s implementation with their own (see “Model Object Validation”).

Availability

Available in iOS 3.0 and later.

See Also

- [validateForInsert:](#) (page 145)
- [validateForUpdate:](#) (page 145)
- [validateValue:forKey:error:](#) (page 146)

Declared In

NSManagedObject.h

validateForInsert:

Determines whether the receiver can be inserted in its current state.

```
- (BOOL)validateForInsert:(NSError **)error
```

Parameters

error

If the receiver cannot be inserted in its current state, upon return contains an instance of `NSError` that describes the problem.

Return Value

YES if the receiver can be inserted in its current state, otherwise NO.

Special Considerations

Subclasses should invoke super’s implementation before performing their own validation, and should combine any error returned by super’s implementation with their own (see “Model Object Validation”).

Availability

Available in iOS 3.0 and later.

See Also

- [validateForDelete:](#) (page 144)
- [validateForUpdate:](#) (page 145)
- [validateValue:forKey:error:](#) (page 146)

Declared In

NSManagedObject.h

validateForUpdate:

Determines whether the receiver’s current state is valid.

```
- (BOOL)validateForUpdate:(NSError **)error
```

Parameters*error*

If the receiver's current state is invalid, upon return contains an instance of `NSError` that describes the problem.

Return Value

YES if the receiver's current state is valid, otherwise NO.

Discussion

`NSManagedObject`'s implementation iterates through all of the receiver's properties validating each in turn. If this results in more than one error, the `userInfo` dictionary in the `NSError` returned in *error* contains a key `NSDetailedErrorsKey`; the corresponding value is an array containing the individual validation errors. If you pass NULL as the error, validation will abort after the first failure.

Important: Subclasses should invoke super's implementation before performing their own validation, and should combine any error returned by super's implementation with their own (see "Model Object Validation").

Availability

Available in iOS 3.0 and later.

See Also

- [validateForDelete:](#) (page 144)
- [validateForInsert:](#) (page 145)
- [validateValue:forKey:error:](#) (page 146)

Declared In

`NSManagedObject.h`

validateValue:forKey:error:

Validates a property value for a given key.

```
- (BOOL)validateValue:(id *)value forKey:(NSString *)key error:(NSError **)error
```

Parameters*value*

A pointer to an object.

key

The name of one of the receiver's properties.

error

If *value* is not a valid value for *key* (and cannot be coerced), upon return contains an instance of `NSError` that describes the problem.

Return Value

YES if *value* is a valid value for *key* (or if *value* can be coerced into a valid value for *key*), otherwise NO.

Discussion

This method is responsible for two things: coercing the value into an appropriate type for the object, and validating it according to the object's rules.

The default implementation provided by `NSManagedObject` consults the object's entity description to coerce the value and to check for basic errors, such as a null value when that isn't allowed and the length of strings when a field width is specified for the attribute. It then searches for a method of the form `validate<Key>:error:` and invokes it if it exists.

You can implement methods of the form `validate<Key>:error:` to perform validation that is not possible using the constraints available in the property description. If it finds an unacceptable value, your validation method should return `NO` and in `error` an `NSError` object that describes the problem. For more details, see "Model Object Validation". For inter-property validation (to check for combinations of values that are invalid), see `validateForUpdate:` (page 145) and related methods.

Availability

Available in iOS 3.0 and later.

See Also

- `validateForDelete:` (page 144)
- `validateForInsert:` (page 145)
- `validateForUpdate:` (page 145)

Declared In

`NSManagedObject.h`

valueForKey:

Returns the value for the property specified by *key*.

```
- (id)valueForKey:(NSString *)key
```

Parameters

key

The name of one of the receiver's properties.

Return Value

The value of the property specified by *key*.

Discussion

If *key* is not a property defined by the model, the method raises an exception. This method is overridden by `NSManagedObject` to access the managed object's generic dictionary storage unless the receiver's class explicitly provides key-value coding compliant accessor methods for *key*.

Important: You must not override this method.

Availability

Available in iOS 3.0 and later.

See Also

- `primitiveValueForKey:` (page 141)
- `setValue:forKey:` (page 144)
- `setObservationInfo:` (page 142)

Declared In

`NSManagedObject.h`

willAccessValueForKey:

Provides support for key-value observing access notification.

```
- (void)willAccessValueForKey:(NSString *)key
```

Parameters

key

The name of one of the receiver's properties.

Discussion

See [didAccessValueForKey:](#) (page 131) for more details. You can invoke this method with the *key* value of *nil* to ensure that a fault has been fired, as illustrated by the following example.

```
[aManagedObject willAccessValueForKey:nil];
```

Availability

Available in iOS 3.0 and later.

See Also

- [didAccessValueForKey:](#) (page 131)

Declared In

NSManagedObject.h

willChangeValueForKey:

Invoked to inform the receiver that the value of a given property is about to change.

```
- (void)willChangeValueForKey:(NSString *)key
```

Parameters

key

The name of the property that will change.

Discussion

For more details, see *Key-Value Observing Programming Guide*.

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObject.h

willChangeValueForKey:withSetMutation:usingObjects:

Invoked to inform the receiver that the specified change is about to be made to a specified to-many relationship.

```
- (void)willChangeValueForKey:(NSString *)inKey  
  withSetMutation:(NSKeyValueSetMutationKind)inMutationKind usingObjects:(NSSet  
  *)inObjects
```

Parameters*inKey*

The name of a property that is a to-many relationship

inMutationKind

The type of change that will be made.

*inObjects*The objects that were involved in the change (see `NSKeyValueSetMutationKind`).**Discussion**For more details, see *Key-Value Observing Programming Guide*.

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In`NSManagedObject.h`**willSave**

Invoked automatically by the Core Data framework when the receiver's managed object context is saved.

- (void)willSave

Discussion

This method can have “side effects” on persistent values. You can use it to, for example, compute persistent values from other transient or scratchpad values.

If you want to update a persistent property value, you should typically test for equality of any new value with the existing value before making a change. If you change property values using standard accessor methods, Core Data will observe the resultant change notification and so invoke `willSave` again before saving the object's managed object context. If you continue to modify a value in `willSave`, `willSave` will continue to be called until your program crashes.

For example, if you set a last-modified timestamp, you should check whether either you previously set it in the same save operation, or that the existing timestamp is not less than a small delta from the current time. Typically it's better to calculate the timestamp once for all the objects being saved (for example, in response to an `NSManagedObjectContextWillSaveNotification`).

If you change property values using primitive accessors, you avoid the possibility of infinite recursion, but Core Data will not notice the change you make.

Note that the sense of “save” in the method name is that of a database commit statement and so applies to deletions as well as to updates to objects. For subclasses, this method is therefore an appropriate locus for code to be executed when an object deleted as well as “saved to disk.” You can find out if an object is marked for deletion with `isDeleted` (page 136).

Availability

Available in iOS 3.0 and later.

See Also- [didSave](#) (page 133)

Declared In

NSManagedObject.h

willTurnIntoFault

Invoked automatically by the Core Data framework before receiver is converted to a fault.

- (void)willTurnIntoFault

Discussion

This method is the companion of the [didTurnIntoFault](#) (page 134) method. You can use it to (re)set state which requires access to property values (for example, observers across keypaths). The default implementation does nothing.

Availability

Available in iOS 3.0 and later.

See Also

- [didTurnIntoFault](#) (page 134)

Declared In

NSManagedObject.h

Constants

The following constants relate to errors returned following validation failures.

NSDetailedErrorsKey (page 285)	If multiple validation errors occur in one operation, they are collected in an array and added with this key to the “top-level error” of the operation.
NSValidationKeyErrorKey (page 285)	Key for the key that failed to validate for a validation error.
NSValidationPredicateErrorKey (page 286)	For predicate-based validation, key for the predicate for the condition that failed to validate.
NSValidationValueErrorKey (page 286)	If non-nil, the key for the value for the key that failed to validate for a validation error.

NSSnapshotEventType

Constants returned from [awakeFromSnapshotEvents:](#) (page 129) to denote the reason why a managed object may need to reinitialize values.

```
enum {
    NSSnapshotEventUndoInsertion = 1 << 1,
    NSSnapshotEventUndoDeletion = 1 << 2,
    NSSnapshotEventUndoUpdate = 1 << 3,
    NSSnapshotEventRollback = 1 << 4,
    NSSnapshotEventRefresh = 1 << 5,
    NSSnapshotEventMergePolicy = 1 << 6
};
typedef NSUInteger NSSnapshotEventType;
```

Constants

`NSSnapshotEventUndoInsertion`

Specifies a change due to undo from insertion.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventUndoDeletion`

Specifies a change due to undo from deletion.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventUndoUpdate`

Specifies a change due to a property-level undo.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventRollback`

Specifies a change due to the managed object context being rolled back.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventRefresh`

Specifies a change due to the managed object being refreshed.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventMergePolicy`

Specifies a change due to conflict resolution during a save operation.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

NSManagedObjectContext Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSLocking NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSManagedObjectContext.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial Core Data Snippets Predicate Programming Guide

Overview

An instance of `NSManagedObjectContext` 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. A single managed object instance exists in one and only one context, but multiple copies of an object can exist in different contexts. Thus object uniquing is scoped to a particular context.

Life-cycle Management

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. Through a context you can retrieve or “fetch” objects from a persistent store, make changes to those objects, and then either discard the changes or—again through the context—commit them back to the persistent store. The context is responsible for watching for changes in its objects and maintains an undo manager so you can have finer-grained control over undo and redo. You can insert new objects and delete ones you have fetched, and commit these modifications to the persistent store.

Persistent Store Coordinator

A context always has a “parent” persistent store coordinator which provides the model and dispatches requests to the various persistent stores containing the data. Without a coordinator, a context is not fully functional. The context’s coordinator provides the managed object model and handles persistency. All objects fetched from an external store are registered in a context together with a global identifier (an instance of `NSManagedObjectID`) that’s used to uniquely identify each object to the external store.

Subclassing Notes

You are strongly discouraged from subclassing `NSManagedObjectContext`. The change tracking and undo management mechanisms are highly optimized and hence intricate and delicate. Interposing your own additional logic that might impact `processPendingChanges` can have unforeseen consequences. In situations such as store migration, Core Data will create instances of `NSManagedObjectContext` for its own use. Under these circumstances, you cannot rely on any features of your custom subclass. Any `NSManagedObject` subclass must always be fully compatible with `NSManagedObjectContext` (as opposed to any subclass of `NSManagedObjectContext`).

Tasks

Registering and Fetching Objects

- [executeFetchRequest:error:](#) (page 160)
Returns an array of objects that meet the criteria specified by a given fetch request.
- [countForFetchRequest:error:](#) (page 158)
Returns the number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:`.
- [objectRegisteredForID:](#) (page 164)
Returns the object for a specified ID, if the object is registered with the receiver.
- [objectWithID:](#) (page 165)
Returns the object for a specified ID.
- [existingObjectWithID:error:](#) (page 161)
Returns the object for the specified ID.
- [registeredObjects](#) (page 168)
Returns the set of objects registered with the receiver.

Managed Object Management

- [insertObject:](#) (page 162)
Registers an object to be inserted in the receiver’s persistent store the next time changes are saved.
- [deleteObject:](#) (page 159)
Specifies an object that should be removed from its persistent store when changes are committed.

- [assignObject:toPersistentStore:](#) (page 157)
Specifies the store in which a newly-inserted object will be saved.
- [obtainPermanentIDsForObjects:error:](#) (page 165)
Converts to permanent IDs the object IDs of the objects in a given array.
- [detectConflictsForObject:](#) (page 159)
Marks an object for conflict detection.
- [refreshObject:mergeChanges:](#) (page 167)
Updates the persistent properties of a managed object to use the latest values from the persistent store.
- [processPendingChanges](#) (page 166)
Forces the receiver to process changes to the object graph.
- [insertedObjects](#) (page 162)
Returns the set of objects that have been inserted into the receiver but not yet saved in a persistent store.
- [updatedObjects](#) (page 175)
Returns the set of objects registered with the receiver that have uncommitted changes.
- [deletedObjects](#) (page 158)
Returns the set of objects that will be removed from their persistent store during the next save operation.

Merging Changes from Another Context

- [mergeChangesFromContextDidSaveNotification:](#) (page 163)
Merges the changes specified in a given notification.

Undo Management

- [undoManager](#) (page 175)
Returns the undo manager of the receiver.
- [setUndoManager:](#) (page 173)
Sets the undo manager of the receiver.
- [undo](#) (page 174)
Sends an undo message to the receiver's undo manager, asking it to reverse the latest uncommitted changes applied to objects in the object graph.
- [redo](#) (page 167)
Sends an redo message to the receiver's undo manager, asking it to reverse the latest undo operation applied to objects in the object graph.
- [reset](#) (page 169)
Returns the receiver to its base state.
- [rollback](#) (page 169)
Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values.
- [save:](#) (page 170)
Attempts to commit unsaved changes to registered objects to their persistent store.

- [hasChanges](#) (page 161)
Returns a Boolean value that indicates whether the receiver has uncommitted changes.

Locking

- [lock](#) (page 163)
Attempts to acquire a lock on the receiver.
- [unlock](#) (page 175)
Relinquishes a previously acquired lock.
- [tryLock](#) (page 174)
Attempts to acquire a lock.

Delete Propagation

- [propagatesDeletesAtEndOfEvent](#) (page 167)
Returns a Boolean that indicates whether the receiver propagates deletes at the end of the event in which a change was made.
- [setPropagatesDeletesAtEndOfEvent:](#) (page 171)
Sets whether the context propagates deletes to related objects at the end of the event.

Retaining Registered Objects

- [retainsRegisteredObjects](#) (page 169)
Returns a Boolean that indicates whether the receiver sends a `retain` message to objects upon registration.
- [setRetainsRegisteredObjects:](#) (page 172)
Sets whether or not the receiver retains all registered objects, or only objects necessary for a pending save (those that are inserted, updated, deleted, or locked).

Managing the Persistent Store Coordinator

- [persistentStoreCoordinator](#) (page 166)
Returns the persistent store coordinator of the receiver.
- [setPersistentStoreCoordinator:](#) (page 171)
Sets the persistent store coordinator of the receiver.

Managing the Staleness Interval

- [stalenessInterval](#) (page 173)
Returns the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

- [setStalenessInterval:](#) (page 172)
Sets the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

Managing the Merge Policy

- [mergePolicy](#) (page 164)
Returns the merge policy of the receiver.
- [setMergePolicy:](#) (page 170)
Sets the merge policy of the receiver.

Instance Methods

assignObject:toPersistentStore:

Specifies the store in which a newly-inserted object will be saved.

```
- (void)assignObject:(id)object toPersistentStore:(NSPersistentStore *)store
```

Parameters

object

A managed object.

store

A persistent store.

Discussion

You can obtain a store from the persistent store coordinator, using for example [persistentStoreForURL:](#) (page 237).

Special Considerations

It is only necessary to use this method if the receiver's persistent store coordinator manages multiple writable stores that have *object's* entity in their configuration. Maintaining configurations in the managed object model can eliminate the need for invoking this method directly in many situations. If the receiver's persistent store coordinator manages only a single writable store, or if only one store has *object's* entity in its model, *object* will automatically be assigned to that store.

Availability

Available in iOS 3.0 and later.

See Also

- [insertObject:](#) (page 162)
- [persistentStoreCoordinator](#) (page 166)

Declared In

NSManagedObjectContext.h

countForFetchRequest:error:

Returns the number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:`.

```
- (NSUInteger)countForFetchRequest:(NSFetchRequest *)request error:(NSError **)error
```

Parameters

request

A fetch request that specifies the search criteria for the fetch.

error

If there is a problem executing the fetch, upon return contains an instance of `NSError` that describes the problem.

Return Value

The number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:` (page 160).

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

deletedObjects

Returns the set of objects that will be removed from their persistent store during the next save operation.

```
- (NSSet *)deletedObjects
```

Return Value

The set of objects that will be removed from their persistent store during the next save operation.

Discussion

The returned set does not necessarily include all the objects that have been deleted (using `deleteObject:` (page 159))—if an object has been inserted and deleted without an intervening save operation, it is not included in the set.

A managed object context does not post key-value observing notifications when the return value of `deletedObjects` changes. A context does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 179) notification when a change is made, and a `NSManagedObjectContextWillSaveNotification` (page 179) notification and a `NSManagedObjectContextDidSaveNotification` (page 179) notification before and after changes are committed respectively (although again the set of deleted objects given for a `NSManagedObjectContextDidSaveNotification` (page 179) does not include objects that were inserted and deleted without an intervening save operation—that is, they had never been saved to a persistent store).

Availability

Available in iOS 3.0 and later.

See Also

- `deleteObject:` (page 159)
- `insertedObjects` (page 162)
- `registeredObjects` (page 168)

- [updatedObjects](#) (page 175)
- [isDeleted](#) (page 136) (NSManagedObject)

Declared In

NSManagedObjectContext.h

deleteObject:

Specifies an object that should be removed from its persistent store when changes are committed.

```
- (void)deleteObject:(NSManagedObject *)object
```

Parameters

object

A managed object.

Discussion

When changes are committed, *object* will be removed from the uniquing tables. If *object* has not yet been saved to a persistent store, it is simply removed from the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [deletedObjects](#) (page 158)
- [isDeleted](#) (page 136) (NSManagedObject)

Declared In

NSManagedObjectContext.h

detectConflictsForObject:

Marks an object for conflict detection.

```
- (void)detectConflictsForObject:(NSManagedObject *)object
```

Parameters

object

A managed object.

Discussion

If on the next invocation of [save:](#) (page 170) *object* has been modified in its persistent store, the save fails. This allows optimistic locking for unchanged objects. Conflict detection is always performed on changed or deleted objects.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

executeFetchRequest:error:

Returns an array of objects that meet the criteria specified by a given fetch request.

```
- (NSArray *)executeFetchRequest:(NSFetchRequest *)request error:(NSError **)error
```

Parameters

request

A fetch request that specifies the search criteria for the fetch.

error

If there is a problem executing the fetch, upon return contains an instance of `NSError` that describes the problem.

Return Value

An array of objects that meet the criteria specified by *request* fetched from the receiver and from the persistent stores associated with the receiver's persistent store coordinator. If an error occurs, returns `nil`. If no objects match the criteria specified by *request*, returns an empty array.

Discussion

Returned objects are registered with the receiver.

The following points are important to consider:

- If the fetch request has no predicate, then all instances of the specified entity are retrieved, modulo other criteria below.
- An object that meets the criteria specified by *request* (it is an instance of the entity specified by the request, and it matches the request's predicate if there is one) and that has been inserted into a context but which is not yet saved to a persistent store, is retrieved if the fetch request is executed on that context.
- If an object in a context has been modified, a predicate is evaluated against its modified state, not against the current state in the persistent store. Therefore, if an object in a context has been modified such that it meets the fetch request's criteria, the request retrieves it even if changes have not been saved to the store and the values in the store are such that it does not meet the criteria. Conversely, if an object in a context has been modified such that it does not match the fetch request, the fetch request will not retrieve it even if the version in the store does match.
- If an object has been deleted from the context, the fetch request does not retrieve it even if that deletion has not been saved to a store.

Objects that have been realized (populated, faults fired, "read from", and so on) as well as pending updated, inserted, or deleted, are never changed by a fetch operation without developer intervention. If you fetch some objects, work with them, and then execute a new fetch that includes a superset of those objects, you do not get new instances or update data for the existing objects—you get the existing objects with their current in-memory state.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

existingObjectWithID:error:

Returns the object for the specified ID.

```
- (NSManagedObject *)existingObjectWithID:(NSManagedObjectID *)objectIDerror:(NSError **)error
```

Parameters

objectID

The object ID for the requested object.

error

If there is a problem in retrieving the object specified by *objectID*, upon return contains an error that describes the problem.

Return Value

The object specified by *objectID*. If the object cannot be fetched, or does not exist, or cannot be faulted, it returns `nil`.

Discussion

If there is a managed object with the given ID already registered in the context, that object is returned directly; otherwise the corresponding object is faulted into the context.

This method might perform I/O if the data is uncached.

Unlike [objectWithID:](#) (page 165), this method never returns a fault.

Availability

Available in iOS 3.0 and later.

See Also

- [objectWithID:](#) (page 165)
- [objectRegisteredForID:](#) (page 164)

Declared In

NSManagedObjectContext.h

hasChanges

Returns a Boolean value that indicates whether the receiver has uncommitted changes.

```
- (BOOL)hasChanges
```

Return Value

YES if the receiver has uncommitted changes, otherwise NO.

Discussion

On Mac OS X v10.6 and later, this property is key-value observing compliant (see *Key-Value Observing Programming Guide*).

Prior to Mac OS X v10.6, this property is not key-value observing compliant—for example, if you are using Cocoa bindings, you cannot bind to the `hasChanges` property of a managed object context.

Special Considerations

If you are observing this property using key-value observing (KVO) you should not touch the context or its objects within your implementation of `observeValueForKeyPath:ofObject:change:context:` for this notification. (This is because of the intricacy of the locations of the KVO notifications—for example, the context may be in the middle of an undo operation, or repairing a merge conflict.) If you need to send messages to the context of change any of its managed objects as a result of a change to the value of `hasChanges`, you must do so after the call stack unwinds (typically using `performSelector:withObject:afterDelay:` or a similar method).

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

insertedObjects

Returns the set of objects that have been inserted into the receiver but not yet saved in a persistent store.

- (NSSet *)insertedObjects

Return Value

The set of objects that have been inserted into the receiver but not yet saved in a persistent store.

Discussion

A managed object context does not post key-value observing notifications when the return value of `insertedObjects` changes—it does, however, post a [NSManagedObjectContextObjectsDidChangeNotification](#) (page 179) notification when a change is made, and a [NSManagedObjectContextWillSaveNotification](#) (page 179) and a [NSManagedObjectContextDidSaveNotification](#) (page 179) notification before and after changes are committed respectively.

Availability

Available in iOS 3.0 and later.

See Also

- [deletedObjects](#) (page 158)
- [insertObject:](#) (page 162)
- [registeredObjects](#) (page 168)
- [updatedObjects](#) (page 175)

Declared In

`NSManagedObjectContext.h`

insertObject:

Registers an object to be inserted in the receiver's persistent store the next time changes are saved.

- (void)insertObject:(NSManagedObject *)object

Parameters*object*

A managed object.

Discussion

The managed object (*object*) is registered in the receiver with a temporary global ID. It is assigned a permanent global ID when changes are committed. If the current transaction is rolled back (for example, if the receiver is sent a [rollback](#) (page 169) message) before a save operation, the object is unregistered from the receiver.

Availability

Available in iOS 3.0 and later.

See Also- [insertedObjects](#) (page 162)**Declared In**

NSManagedObjectContext.h

lock

Attempts to acquire a lock on the receiver.

- (void)lock

Discussion

This method blocks a thread's execution until the lock can be acquired. An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking [unlock](#) (page 175).

Sending this message to a managed object context helps the framework to understand the scope of a transaction in a multi-threaded environment. It is preferable to use the `NSManagedObjectContext`'s implementation of `NSLocking` instead using of a separate mutex object.

If you `lock` (or successfully `tryLock`) a managed object context, the thread in which the lock call is made must have a retain until it invokes `unlock`. If you do not properly retain a context in a multi-threaded environment, this will result in deadlock.

Availability

Available in iOS 3.0 and later.

See Also- [tryLock](#) (page 174)- [unlock](#) (page 175)**Declared In**

NSManagedObjectContext.h

mergeChangesFromContextDidSaveNotification:

Merges the changes specified in a given notification.

- (void)mergeChangesFromContextDidSaveNotification:(NSNotification *)*notification*

Parameters*notification*

An instance of an [NSManagedObjectContextWillSaveNotification](#) (page 179) notification posted by another context.

Discussion

This method refreshes any objects which have been updated in the other context, faults in any newly-inserted objects, and invokes [deleteObject:](#) (page 159): on those which have been deleted.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

mergePolicy

Returns the merge policy of the receiver.

- (id)mergePolicy

Return Value

The receiver's merge policy.

Discussion

The default is `NSErrorMergePolicy`.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

objectRegisteredForID:

Returns the object for a specified ID, if the object is registered with the receiver.

- (NSManagedObject *)objectRegisteredForID:(NSManagedObjectID *)*objectID*

Parameters*objectID*

An object ID.

Return Value

The object for the specified ID if it is registered with the receiver, otherwise `nil`.

Availability

Available in iOS 3.0 and later.

See Also

- [objectWithID:](#) (page 165)
- [existingObjectWithID:error:](#) (page 161)

Declared In

NSManagedObjectContext.h

objectWithID:

Returns the object for a specified ID.

- (NSManagedObject *)objectWithID:(NSManagedObjectID *)*objectID***Parameters***objectID*

An object ID.

Return Value

The object for the specified ID.

Discussion

If the object is not registered in the context, it may be fetched or returned as a fault. This method always returns an object. The data in the persistent store represented by *objectID* is assumed to exist—if it does not, the returned object throws an exception when you access any property (that is, when the fault is fired). The benefit of this behavior is that it allows you to create and use faults, then create the underlying rows later or in a separate context.

Availability

Available in iOS 3.0 and later.

See Also

- [objectRegisteredForID:](#) (page 164)
- [existingObjectWithID:error:](#) (page 161)
- [managedObjectIDForURIRepresentation:](#) (page 235)
- [URIRepresentation](#) (page 183)

Declared In

NSManagedObjectContext.h

obtainPermanentIDsForObjects:error:

Converts to permanent IDs the object IDs of the objects in a given array.

- (BOOL)obtainPermanentIDsForObjects:(NSArray *)*objects* error:(NSError **)*error***Parameters***objects*

An array of managed objects.

*error*If an error occurs, upon return contains an `NSError` object that describes the problem.**Return Value**YES if permanent IDs are obtained for all the objects in *objects*, otherwise NO.

Discussion

This method converts the object ID of each managed object in *objects* to a permanent ID. Although the object will have a permanent ID, it will still respond positively to `isInserted` (page 138) until it is saved. Any object that already has a permanent ID is ignored.

Any object not already assigned to a store is assigned based on the same rules Core Data uses for assignment during a save operation (first writable store supporting the entity, and appropriate for the instance and its related items).

Special Considerations

This method results in a transaction with the underlying store which changes the file's modification date.

This results an additional consideration if you invoke this method on the managed object context associated with an instance of `NSPersistentDocument`. Instances of `NSDocument` need to know that they are in sync with the underlying content. To avoid problems, after invoking this method you must therefore update the document's modification date (using `setFileModificationDate:`).

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

persistentStoreCoordinator

Returns the persistent store coordinator of the receiver.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
```

Return Value

The persistent store coordinator of the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

processPendingChanges

Forces the receiver to process changes to the object graph.

```
- (void)processPendingChanges
```

Discussion

This method causes changes to registered managed objects to be recorded with the undo manager.

In AppKit-based applications, this method is invoked automatically at least once during the event loop (at the end of the loop)—it may be called more often than that if the framework needs to coalesce your changes before doing something else. You can also invoke it manually to coalesce any pending unprocessed changes.

Availability

Available in iOS 3.0 and later.

See Also

- [redo](#) (page 167)
- [undo](#) (page 174)
- [undoManager](#) (page ?)

Declared In

NSManagedObjectContext.h

propagatesDeletesAtEndOfEvent

Returns a Boolean that indicates whether the receiver propagates deletes at the end of the event in which a change was made.

- (BOOL)propagatesDeletesAtEndOfEvent

Return Value

YES if the receiver propagates deletes at the end of the event in which a change was made, NO if it propagates deletes only immediately before saving changes.

Availability

Available in iOS 3.0 and later.

See Also

- [setPropagatesDeletesAtEndOfEvent:](#) (page 171)

Declared In

NSManagedObjectContext.h

redo

Sends an redo message to the receiver's undo manager, asking it to reverse the latest undo operation applied to objects in the object graph.

- (void)redo

Availability

Available in iOS 3.0 and later.

See Also

- [undo](#) (page 174)
- [processPendingChanges](#) (page 166)

Declared In

NSManagedObjectContext.h

refreshObject:mergeChanges:

Updates the persistent properties of a managed object to use the latest values from the persistent store.

- (void)refreshObject:(NSManagedObject *)*object* mergeChanges:(BOOL)*flag*

Parameters*object*

A managed object.

flag

A Boolean value.

If *flag* is `NO`, then *object* is turned into a fault and any pending changes are lost. The object remains a fault until it is accessed again, at which time its property values will be reloaded from the store or last cached state.

If *flag* is `YES`, then *object*'s property values are reloaded from the values from the store or the last cached state then any changes that were made (in the local context) are re-applied over those (now newly updated) values. (If *flag* is `YES` the merge of the values into *object* will always succeed—in this case there is therefore no such thing as a “merge conflict” or a merge that is not possible.)

Discussion

If the staleness interval (see [stalenessInterval](#) (page 173)) has not been exceeded, any available cached data is reused instead of executing a new fetch. If *flag* is `YES`, this method does not affect any transient properties; if *flag* is `NO`, transient properties are released.

You typically use this method to ensure data freshness if more than one managed object context may use the same persistent store simultaneously, in particular if you get an optimistic locking failure when attempting to save.

It is important to note that turning *object* into a fault (*flag* is `NO`) also causes related managed objects (that is, those to which *object* has a reference) to be released, so you can also use this method to trim a portion of your object graph you want to constrain memory usage.

Availability

Available in iOS 3.0 and later.

See Also

- [detectConflictsForObject:](#) (page 159)
- [reset](#) (page 169)
- [setStalenessInterval:](#) (page 172)

Declared In

NSManagedObjectContext.h

registeredObjects

Returns the set of objects registered with the receiver.

- (NSSet *)registeredObjects

Return Value

The set of objects registered with the receiver.

Discussion

A managed object context does not post key-value observing notifications when the return value of `registeredObjects` changes.

Availability

Available in iOS 3.0 and later.

See Also

- [deletedObjects](#) (page 158)
- [insertedObjects](#) (page 162)
- [updatedObjects](#) (page 175)

Declared In

NSManagedObjectContext.h

reset

Returns the receiver to its base state.

```
- (void)reset
```

Discussion

All the receiver's managed objects are “forgotten.” If you use this method, you should ensure that you also discard references to any managed objects fetched using the receiver, since they will be invalid afterwards.

Availability

Available in iOS 3.0 and later.

See Also

- [rollback](#) (page 169)
- [setStalenessInterval:](#) (page 172)
- [undo](#) (page 174)

Declared In

NSManagedObjectContext.h

retainsRegisteredObjects

Returns a Boolean that indicates whether the receiver sends a `retain` message to objects upon registration.

```
- (BOOL)retainsRegisteredObjects
```

Return Value

YES if the receiver sends a `retain` message to objects upon registration, otherwise NO.

Availability

Available in iOS 3.0 and later.

See Also

- [setRetainsRegisteredObjects:](#) (page 172)

Declared In

NSManagedObjectContext.h

rollback

Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values.

- (void)rollback

Discussion

This method does not refetch data from the persistent store or stores.

Availability

Available in iOS 3.0 and later.

See Also

- [reset](#) (page 169)
- [setStalenessInterval:](#) (page 172)
- [undo](#) (page 174)
- [processPendingChanges](#) (page 166)

Declared In

NSManagedObjectContext.h

save:

Attempts to commit unsaved changes to registered objects to their persistent store.

- (BOOL)save:(NSError **)error

Parameters

error

A pointer to an `NSError` object. You do not need to create an `NSError` object. The save operation aborts after the first failure if you pass `NULL`.

Return Value

YES if the save succeeds, otherwise NO.

Discussion

If there were multiple errors (for example several edited objects had validation failures) the description of `NSError` returned indicates that there were multiple errors, and its `userInfo` dictionary contains the key `NSDetailedErrors`. The value associated with the `NSDetailedErrors` key is an array that contains the individual `NSError` objects.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

setMergePolicy:

Sets the merge policy of the receiver.

- (void)setMergePolicy:(id)mergePolicy

Parameters

mergePolicy

The merge policy of the receiver. For possible values, see “[Merge Policies](#)” (page 177).

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

setPersistentStoreCoordinator:

Sets the persistent store coordinator of the receiver.

```
- (void)setPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
```

Parameters

coordinator

The persistent store coordinator of the receiver.

Discussion

The coordinator provides the managed object model and handles persistency. Note that multiple contexts can share a coordinator.

This method raises an exception if *coordinator* is *nil*. If you want to “disconnect” a context from its persistent store coordinator, you should simply release all references to the context and allow it to be deallocated normally.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

setPropagatesDeletesAtEndOfEvent:

Sets whether the context propagates deletes to related objects at the end of the event.

```
- (void)setPropagatesDeletesAtEndOfEvent:(BOOL) flag
```

Parameters

Flag

A Boolean value that indicates whether the context propagates deletes to related objects at the end of the event (YES) or not (NO).

Discussion

The default is YES. If the value is NO, then deletes are propagated during a save operation.

Availability

Available in iOS 3.0 and later.

See Also

– [propagatesDeletesAtEndOfEvent](#) (page 167)

Declared In

NSManagedObjectContext.h

setRetainsRegisteredObjects:

Sets whether or not the receiver retains all registered objects, or only objects necessary for a pending save (those that are inserted, updated, deleted, or locked).

```
- (void)setRetainsRegisteredObjects:(BOOL)flag
```

Parameters

flag

A Boolean value.

If *flag* is NO, then registered objects are retained only when they are inserted, updated, deleted, or locked.

If *flag* is YES, then all registered objects are retained.

Discussion

The default is NO.

Availability

Available in iOS 3.0 and later.

See Also

- [retainsRegisteredObjects](#) (page 169)

Declared In

NSManagedObjectContext.h

setStalenessInterval:

Sets the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

```
- (void)setStalenessInterval:(NSTimeInterval)expiration
```

Parameters

expiration

The maximum length of time that may have elapsed since the store previously fetched data before *fulfilling a fault* issues a new fetch rather than using the previously-fetched data.

A negative value represents an infinite value; 0.0 represents “no staleness acceptable”.

Discussion

The staleness interval controls whether *fulfilling a fault* uses data previously fetched by the application, or issues a new fetch (see also [refreshObject:mergeChanges:](#) (page 167)). The staleness interval does *not* affect objects currently in use (that is, it is *not* used to automatically update property values from a persistent store after a certain period of time).

The expiration value is applied on a per object basis. It is the relative time until cached data (snapshots) should be considered stale. For example, a value of 300.0 informs the context to utilize cached information for no more than 5 minutes after an object was originally fetched.

Note that the staleness interval is a hint and may not be supported by all persistent store types. It is not used by XML and binary stores, since these stores maintain all current values in memory.

Availability

Available in iOS 3.0 and later.

See Also

- [reset](#) (page 169)
- [rollback](#) (page 169)
- [stalenessInterval](#) (page 173)
- [undo](#) (page 174)
- [refreshObject:mergeChanges:](#) (page 167)

Declared In

NSManagedObjectContext.h

setUndoManager:

Sets the undo manager of the receiver.

```
- (void)setUndoManager:(NSUndoManager *)undoManager
```

Parameters*undoManager*

The undo manager of the receiver.

Discussion

You can set the undo manager to `nil` to disable undo support. This provides a performance benefit if you do not want to support undo for a particular context, for example in a large import process—see *Core Data Programming Guide*.

If a context does not have an undo manager, you can enable undo support by setting one. You may also replace a context's undo manager if you want to integrate the context's undo operations with another undo manager in your application.

Important: On Mac OS X, a context provides an undo manager by default; on iOS, the undo manager is `nil` by default.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

stalenessInterval

Returns the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

```
- (NSTimeInterval)stalenessInterval
```

Return Value

The maximum length of time that may have elapsed since the store previously fetched data before *fulfilling a fault* issues a new fetch rather than using the previously-fetched data.

Discussion

The default is infinite staleness, represented by an interval of -1 (although any negative value represents an infinite value); 0.0 represents “no staleness acceptable”.

For a full discussion, see [setStalenessInterval:](#) (page 172).

Availability

Available in iOS 3.0 and later.

See Also

- [setStalenessInterval:](#) (page 172)

Declared In

NSManagedObjectContext.h

tryLock

Attempts to acquire a lock.

- (BOOL)tryLock

Return Value

YES if a lock was acquired, NO otherwise.

Discussion

This method returns immediately after the attempt to acquire a lock.

Availability

Available in iOS 3.0 and later.

See Also

- [lock](#) (page 163)
- [unlock](#) (page 175)

Declared In

NSManagedObjectContext.h

undo

Sends an undo message to the receiver’s undo manager, asking it to reverse the latest uncommitted changes applied to objects in the object graph.

- (void)undo

Availability

Available in iOS 3.0 and later.

See Also

- [reset](#) (page 169)
- [rollback](#) (page 169)
- [undoManager](#) (page ?)
- [processPendingChanges](#) (page 166)

Declared In

NSManagedObjectContext.h

undoManager

Returns the undo manager of the receiver.

- (NSUndoManager *)undoManager

Return Value

The undo manager of the receiver.

Discussion

For a discussion, see [setUndoManager:](#) (page ?).

Important: On Mac OS X, a context provides an undo manager by default; on iOS, the undo manager is `nil` by default.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectContext.h

unlock

Relinquishes a previously acquired lock.

- (void)unlock

Availability

Available in iOS 3.0 and later.

See Also

- [lock](#) (page 163)
- [tryLock](#) (page 174)

Declared In

NSManagedObjectContext.h

updatedObjects

Returns the set of objects registered with the receiver that have uncommitted changes.

- (NSSet *)updatedObjects

Return Value

The set of objects registered with the receiver that have uncommitted changes.

Discussion

A managed object context does not post key-value observing notifications when the return value of `updatedObjects` changes. A context does, however, post a [NSManagedObjectContextObjectsDidChangeNotification](#) (page 179) notification when a change is made, and a [NSManagedObjectContextWillSaveNotification](#) (page 179) notification and a [NSManagedObjectContextDidSaveNotification](#) (page 179) notification before and after changes are committed respectively.

Availability

Available in iOS 3.0 and later.

See Also

- [deletedObjects](#) (page 158)
- [insertedObjects](#) (page 162)
- [registeredObjects](#) (page 168)

Declared In

NSManagedObjectContext.h

Constants

NSManagedObjectContext Change Notification User Info Keys

Core Data uses these string constants as keys in the user info dictionary in a [NSManagedObjectContextObjectsDidChangeNotification](#) (page 179) notification.

```
NSString * const NSInsertedObjectsKey;
NSString * const NSUpdatedObjectsKey;
NSString * const NSDeletedObjectsKey;
NSString * const NSRefreshedObjectsKey;
NSString * const NSInvalidatedObjectsKey;
NSString * const NSInvalidatedAllObjectsKey;
```

Constants

NSInsertedObjectsKey

Key for the set of objects that were inserted into the context.

Available in iOS 3.0 and later.

Declared in NSManagedObjectContext.h.

NSUpdatedObjectsKey

Key for the set of objects that were updated.

Available in iOS 3.0 and later.

Declared in NSManagedObjectContext.h.

NSDeletedObjectsKey

Key for the set of objects that were marked for deletion during the previous event.

Available in iOS 3.0 and later.

Declared in NSManagedObjectContext.h.

`NSRefreshedObjectsKey`

Key for the set of objects that were refreshed.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSInvalidatedObjectsKey`

Key for the set of objects that were invalidated.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSInvalidatedAllObjectsKey`

Key that specifies that all objects in the context have been invalidated.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

Declared In

`NSManagedObjectContext.h`

Merge Policies

Merge policy constants define the way conflicts are handled during a save operation.

```
id NSErrorMergePolicy;
id NSMergeByPropertyStoreTrumpMergePolicy;
id NSMergeByPropertyObjectTrumpMergePolicy;
id NSOverwriteMergePolicy;
id NSRollbackMergePolicy;
```

Constants

`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.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSMergeByPropertyStoreTrumpMergePolicy`

This policy merges conflicts between the persistent store’s version of the object and the current in-memory version, giving priority to external changes.

The merge occurs by individual property. For properties that have been changed in both the external source and in memory, the external changes trump the in-memory ones.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSMergeByPropertyObjectTrumpMergePolicy`

This policy merges conflicts between the persistent store’s version of the object and the current in-memory version, giving priority to in-memory changes.

The merge occurs by individual property. For properties that have been changed in both the external source and in memory, the in-memory changes trump the external ones.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSOverwriteMergePolicy`

This policy overwrites state in the persistent store for the changed objects in conflict.

Changed objects' current state is forced upon the persistent store.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSRollbackMergePolicy`

This policy discards in-memory state changes for objects in conflict.

The persistent store's version of the objects' state is used.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

Discussion

The default policy is the `NSErrorMergePolicy`. It is the only policy that requires action to correct any conflicts; the other policies make a save go through silently by making changes following their rules.

Declared In

`NSManagedObjectContext.h`

The following constants, defined in `CoreDataErrors.h`, relate to errors returned following validation failures or problems encountered during a save operation.

<code>NSValidationObjectErrorKey</code> (page 285)	Key for the object that failed to validate for a validation error.
<code>NSAffectedStoresErrorKey</code> (page 286)	The key for stores prompting an error.
<code>NSAffectedObjectsErrorKey</code> (page 286)	The key for objects prompting an error.

Each conflict record in the `@"conflictList"` array in the `userInfo` dictionary for an error from the `NSErrorMergePolicy` is a dictionary containing some of the keys described in the following table. Of the `cachedRow`, `databaseRow`, and `snapshot` keys, only two will be present depending on whether the conflict is between the managed object context and the persistent store coordinator (`snapshot` and `cachedRow`) or between the persistent store coordinator and the persistent store (`cachedRow` and `databaseRow`).

Constant	Description
<code>@"object"</code>	The managed object that could not be saved.
<code>@"snapshot"</code>	A dictionary of key-value pairs for the properties that represents the managed object context's last saved state for this managed object.
<code>@"cachedRow"</code>	A dictionary of key-value pairs for the properties that represents the persistent store's last saved state for this managed object.
<code>@"databaseRow"</code>	A dictionary of key-value pairs for the properties that represents the database's current state for this managed object.
<code>@"newVersion"</code>	An <code>NSNumber</code> object whose value is latest version number of this managed object.
<code>@"oldVersion"</code>	As <code>NSNumber</code> object whose value is the version number that this managed object context last saved for this managed object.

Notifications

NSManagedObjectContextObjectsDidChangeNotification

Posted when values of properties of objects contained in a managed object context are changed.

The notification is posted during [processPendingChanges](#) (page 166), after the changes have been processed, but before it is safe to call [save](#): (page 170) again (if you try, you will generate an infinite loop).

The notification object is the managed object context. The *userInfo* dictionary contains the following keys: [NSInsertedObjectsKey](#), [NSUpdatedObjectsKey](#), and [NSDeletedObjectsKey](#).

Note that this notification is posted only when managed objects are *changed*; it is not posted when managed objects are added to a context as the result of a fetch.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

NSManagedObjectContextDidSaveNotification

Posted whenever a managed object context completes a save operation.

The notification object is the managed object context. The *userInfo* dictionary contains the following keys: [NSInsertedObjectsKey](#), [NSUpdatedObjectsKey](#), and [NSDeletedObjectsKey](#).

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

NSManagedObjectContextWillSaveNotification

Posted whenever a managed object context is about to perform a save operation.

The notification object is the managed object context. There is no *userInfo* dictionary.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

NSManagedObjectID Class Reference

Inherits from	NSObject
Conforms to	NSCopying NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSManagedObjectID.h
Companion guide	Core Data Programming Guide

Overview

An `NSManagedObjectID` object is a compact, universal, identifier for a managed object. This forms the 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). Identifiers contain the information needed to exactly describe an object in a persistent store (like the primary key in the database), although the detailed information is not exposed. The framework completely encapsulates the “external” information and presents a clean object oriented interface.

Object IDs can be transformed into a URI representation which can be archived and recreated later to refer back to a given object (using [managedObjectIDForURIRepresentation:](#) (page 235) (`NSPersistentStoreCoordinator`) and [objectWithID:](#) (page 165) (`NSManagedObjectContext`)). For example, the last selected group in an application could be stored in the user defaults through the group object’s ID. You can also use object ID URI representations to store “weak” relationships across persistent stores (where no hard join is possible).

Tasks

Information About a Managed Object ID

- [entity](#) (page 182)
Returns the entity description associated with the receiver.
- [isTemporaryID](#) (page 182)
Returns a Boolean value that indicates whether the receiver is temporary.

- [persistentStore](#) (page 182)
Returns the persistent store that contains the object whose ID is the receiver.
- [URIRepresentation](#) (page 183)
Returns a URI that provides an archiveable reference to the object which the receiver represents.

Instance Methods

entity

Returns the entity description associated with the receiver.

- (NSEntityDescription *)entity

Return Value

The entity description object associated with the receiver

Availability

Available in iOS 3.0 and later.

See Also

[entity](#) (page 134) (NSManagedObject)

Declared In

NSManagedObjectID.h

isTemporaryID

Returns a Boolean value that indicates whether the receiver is temporary.

- (BOOL)isTemporaryID

Return Value

YES if the receiver is temporary, otherwise NO.

Discussion

Most object IDs return NO. New objects inserted into a managed object context are assigned a temporary ID which is replaced with a permanent one once the object gets saved to a persistent store.

Availability

Available in iOS 3.0 and later.

Declared In

NSManagedObjectID.h

persistentStore

Returns the persistent store that contains the object whose ID is the receiver.

- (NSPersistentStore *)persistentStore

Return Value

The persistent store that contains the object whose ID is the receiver, or `nil` if the ID is for a newly-inserted object that has not yet been saved to a persistent store.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectID.h`

URIRepresentation

Returns a URI that provides an archiveable reference to the object which the receiver represents.

- (NSURL *)URIRepresentation

Return Value

An `NSURL` object containing a URI that provides an archiveable reference to the object which the receiver represents.

Discussion

If the corresponding managed object has not yet been saved, the object ID (and hence URI) is a temporary value that will change when the corresponding managed object is saved.

Availability

Available in iOS 3.0 and later.

See Also

[managedObjectIDForURIRepresentation:](#) (page 235) (`NSPersistentStoreCoordinator`)

[objectWithID:](#) (page 165): (`NSManagedObjectContext`)

Declared In

`NSManagedObjectID.h`

NSManagedObjectModel Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSCopying NSFastEnumeration NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSManagedObjectModel.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial Core Data Model Versioning and Data Migration Programming Guide

Overview

An `NSManagedObjectModel` object describes a schema—a collection of entities (data models) that you use in your application.

The model contains one or more `NSEntityDescription` objects representing the entities in the schema. Each `NSEntityDescription` object has property description objects (instances of subclasses of `NSPropertyDescription`) that represent the properties (or fields) of the entity in the schema. The Core Data framework uses this description in several ways:

- Constraining UI creation in Interface Builder
- Validating attribute and relationship values at runtime
- Mapping between your managed objects and a database or file-based schema for object persistence.

A managed object model maintains a mapping between each of its entity objects and a corresponding managed object class for use with the persistent storage mechanisms in the Core Data Framework. You can determine the entity for a particular managed object with the `entity` method.

You typically create managed object models using the data modeling tool in Xcode, but it is possible to build a model programmatically if needed.

Loading a Model File

Managed object model files are typically stored in a project or a framework. To load a model, you provide an URL to the constructor. Note that loading a model doesn't have the effect of loading all of its entities.

Stored Fetch Requests

It is often the case that in your application you want to get hold of a collection of objects that share features in common. Sometimes you can define those features (property values) in advance; sometimes you need to be able to supply values at runtime. For example, you might want to be able to retrieve all movies owned by Pixar; alternatively you might want to be able to retrieve all movies that earned more than an amount specified by the user at runtime.

Fetch requests are often predefined in a managed object model as templates. They allow you to pre-define named queries and their parameters in the model. Typically they contain variables that need to be substituted at runtime. `NSManagedObjectModel` provides API to retrieve a stored fetch request by name, and to perform variable substitution—see [fetchRequestTemplateName:](#) (page 193) and [fetchRequestFromTemplateWithName:substitutionVariables:](#) (page 193). You can create fetch request templates programmatically, and associate them with a model using [setFetchRequestTemplate:forName:](#) (page 197); typically, however, you define them using the Xcode design tool.

Configurations

Sometimes a model—particularly one in a framework—may be used in different situations, and you may want to specify different sets of entities to be used in different situations. There might, for example, be certain entities that should only be available if a user has administrative privileges. To support this requirement, a model may have more than one configuration. Each configuration is named, and has an associated set of entities. The sets may overlap. You establish configurations programmatically using [setEntities:forConfiguration:](#) (page 196) or using the Xcode design tool, and retrieve the entities for a given configuration name using [entitiesForConfiguration:](#) (page 192).

Changing Models

Since 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.

Editing Models Programmatically

Managed object models are editable until they are used by an object graph manager (a managed object context or a persistent store coordinator). This allows you to create or modify them dynamically. However, once a model is being used, it *must not* be changed. This is enforced at runtime—when the object manager

first fetches data using a model, the whole of that model becomes uneditable. Any attempt to mutate a model or any of its sub-objects after that point causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

Fast Enumeration

In Mac OS X v10.5 and later and on iOS, `NSManagedObjectModel` supports the `NSFastEnumeration` protocol. You can use this to enumerate over a model's entities, as illustrated in the following example:

```
NSManagedObjectModel *aModel = ...;
for (NSEntityDescription *entity in aModel) {
    // entity is each instance of NSEntityDescription in aModel in turn
}
```

Tasks

Initializing a Model

- [initWithContentsOfURL:](#) (page 194)
Initializes the receiver using the model file at the specified URL.
- + [mergedModelFromBundles:](#) (page 188)
Returns a model created by merging all the models found in given bundles.
- + [mergedModelFromBundles:forStoreMetadata:](#) (page 189)
Returns a merged model from a specified array for the version information in provided metadata.
- + [modelByMergingModels:](#) (page 189)
Creates a single model from an array of existing models.
- + [modelByMergingModels:forStoreMetadata:](#) (page 190)
Returns, for the version information in given metadata, a model merged from a given array of models.

Entities and Configurations

- [entities](#) (page 191)
Returns the entities in the receiver.
- [entitiesByName](#) (page 191)
Returns the entities of the receiver in a dictionary.
- [setEntities:](#) (page 196)
Sets the entities array of the receiver.
- [configurations](#) (page 191)
Returns all the available configuration names of the receiver.
- [entitiesForConfiguration:](#) (page 192)
Returns the entities of the receiver for a specified configuration.
- [setEntities:forConfiguration:](#) (page 196)
Associates the specified entities with the receiver using the given configuration name.

Getting Fetch Request Templates

- [fetchRequestTemplatesByName](#) (page 194)
Returns a dictionary of the receiver's fetch request templates.
- [fetchRequestTemplateForName:](#) (page 193)
Returns the fetch request with a specified name.
- [fetchRequestFromTemplateWithName:substitutionVariables:](#) (page 193)
Returns a copy of the fetch request template with the variables substituted by values from the substitutions dictionary.
- [setFetchRequestTemplate:forName:](#) (page 197)
Associates the specified fetch request with the receiver using the given name.

Localization

- [localizationDictionary](#) (page 196)
Returns the localization dictionary of the receiver.
- [setLocalizationDictionary:](#) (page 198)
Sets the localization dictionary of the receiver.

Versioning and Migration

- [isConfiguration:compatibleWithStoreMetadata:](#) (page 195)
Returns a Boolean value that indicates whether a given configuration in the receiver is compatible with given metadata from a persistent store.
- [entityVersionHashesByName](#) (page 192)
Returns a dictionary of the version hashes for the entities in the receiver.
- [versionIdentifiers](#) (page 199)
Returns the collection of developer-defined version identifiers for the receiver.
- [setVersionIdentifiers:](#) (page 198)
Sets the identifiers for the receiver.

Class Methods

mergedModelFromBundles:

Returns a model created by merging all the models found in given bundles.

```
+ (NSManagedObjectModel *)mergedModelFromBundles:(NSArray *)bundles
```

Parameters

bundles

An array of instances of `NSBundle` to search. If you specify `nil`, then the main bundle is searched.

Return Value

A model created by merging all the models found in *bundles*.

Availability

Available in iOS 3.0 and later.

See Also

- + [mergedModelFromBundles:forStoreMetadata:](#) (page 189)
- + [modelByMergingModels:](#) (page 189)
- + [modelByMergingModels:forStoreMetadata:](#) (page 190)
- [initWithContentsOfURL:](#) (page 194)

Declared In

NSManagedObjectModel.h

mergedModelFromBundles:forStoreMetadata:

Returns a merged model from a specified array for the version information in provided metadata.

```
+ (NSManagedObjectModel *)mergedModelFromBundles:(NSArray *)bundles
  forStoreMetadata:(NSDictionary *)metadata
```

Parameters

bundles

An array of bundles.

metadata

A dictionary containing version information from the metadata for a persistent store.

Return Value

The managed object model used to create the store for the metadata. If a model cannot be created to match the version information specified by *metadata*, returns *nil*.

Discussion

This method is a companion to [mergedModelFromBundles:](#) (page 188).

Availability

Available in iOS 3.0 and later.

See Also

- + [mergedModelFromBundles:](#) (page 188)
- + [modelByMergingModels:](#) (page 189)
- + [modelByMergingModels:forStoreMetadata:](#) (page 190)
- [initWithContentsOfURL:](#) (page 194)

Declared In

NSManagedObjectModel.h

modelByMergingModels:

Creates a single model from an array of existing models.

```
+ (NSManagedObjectModel *)modelByMergingModels:(NSArray *)models
```

Parameters*models*An array of instances of `NSManagedObjectModel`.**Return Value**A single model made by combining the models in *models*.**Discussion**

You use this method to combine multiple models (typically from different frameworks) into one.

Availability

Available in iOS 3.0 and later.

See Also

- + [mergedModelFromBundles:](#) (page 188)
- + [mergedModelFromBundles:forStoreMetadata:](#) (page 189)
- + [modelByMergingModels:forStoreMetadata:](#) (page 190)
- [initWithContentsOfURL:](#) (page 194)

Declared In`NSManagedObjectModel.h`**modelByMergingModels:forStoreMetadata:**

Returns, for the version information in given metadata, a model merged from a given array of models.

```
+ (NSManagedObjectModel *)modelByMergingModels:(NSArray *)models
  forStoreMetadata:(NSDictionary *)metadata
```

Parameters*models*An array of instances of `NSManagedObjectModel`.*metadata*

A dictionary containing version information from the metadata for a persistent store.

Return ValueA merged model from *models* for the version information in *metadata*. If a model cannot be created to match the version information in *metadata*, returns `nil`.**Discussion**This is the companion method to [mergedModelFromBundles:forStoreMetadata:](#) (page 189).**Availability**

Available in iOS 3.0 and later.

See Also

- + [mergedModelFromBundles:](#) (page 188)
- + [mergedModelFromBundles:forStoreMetadata:](#) (page 189)
- + [modelByMergingModels:](#) (page 189)
- [initWithContentsOfURL:](#) (page 194)

Declared In`NSManagedObjectModel.h`

Instance Methods

configurations

Returns all the available configuration names of the receiver.

- (NSArray *)configurations

Return Value

An array containing the available configuration names of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [entitiesForConfiguration:](#) (page 192)
- [setEntities:forConfiguration:](#) (page 196)

Declared In

NSManagedObjectModel.h

entities

Returns the entities in the receiver.

- (NSArray *)entities

Return Value

An array containing the entities in the receiver.

Discussion

Entities are instances of `NSEntityDescription`.

Availability

Available in iOS 3.0 and later.

See Also

- [entitiesByName](#) (page 191)
- [entitiesForConfiguration:](#) (page 192)
- [setEntities:](#) (page 196)
- [setEntities:forConfiguration:](#) (page 196)

Declared In

NSManagedObjectModel.h

entitiesByName

Returns the entities of the receiver in a dictionary.

- (NSDictionary *)entitiesByName

Return Value

The entities of the receiver in a dictionary, where the keys in the dictionary are the names of the corresponding entities.

Availability

Available in iOS 3.0 and later.

See Also

- [entities](#) (page 191)
- [entitiesForConfiguration:](#) (page 192)
- [setEntities:](#) (page 196)
- [setEntities:forConfiguration:](#) (page 196)
- + [entityForName:inManagedObjectContext:](#) (page 40) (NSEntityDescription)

Declared In

NSManagedObjectModel.h

entitiesForConfiguration:

Returns the entities of the receiver for a specified configuration.

```
- (NSArray *)entitiesForConfiguration:(NSString *)configuration
```

Parameters

configuration

The name of a configuration in the receiver.

Return Value

An array containing the entities of the receiver for the configuration specified by *configuration*.

Availability

Available in iOS 3.0 and later.

See Also

- [entities](#) (page 191)
- [entitiesByName](#) (page 191)
- [setEntities:](#) (page 196)
- [setEntities:forConfiguration:](#) (page 196)

Declared In

NSManagedObjectModel.h

entityVersionHashesByName

Returns a dictionary of the version hashes for the entities in the receiver.

```
- (NSDictionary *)entityVersionHashesByName
```

Return Value

A dictionary of the version hashes for the entities in the receiver, keyed by entity name.

Discussion

The dictionary of version hash information is used by Core Data to determine schema compatibility.

Availability

Available in iOS 3.0 and later.

See Also

- [isConfiguration:compatibleWithStoreMetadata:](#) (page 195)

Declared In

NSManagedObjectModel.h

fetchRequestFromTemplateName:substitutionVariables:

Returns a copy of the fetch request template with the variables substituted by values from the substitutions dictionary.

```
- (NSFetchRequest *)fetchRequestFromTemplateName:(NSString *)name  
    substitutionVariables:(NSDictionary *)variables
```

Parameters

name

A string containing the name of a fetch request template.

variables

A dictionary containing key-value pairs where the keys are the names of variables specified in the template; the corresponding values are substituted before the fetch request is returned. The dictionary must provide values for all the variables in the template.

Return Value

A copy of the fetch request template with the variables substituted by values from *variables*.

Discussion

The *variables* dictionary must provide values for all the variables. If you want to test for a nil value, use [NSNull null].

This method provides the usual way to bind an “abstractly” defined fetch request template to a concrete fetch. For more details on using this method, see [Creating Predicates](#).

Availability

Available in iOS 3.0 and later.

See Also

- [fetchRequestTemplatesByName](#) (page 194)
- [fetchRequestTemplateForName:](#) (page 193)
- [setFetchRequestTemplate:forName:](#) (page 197)

Declared In

NSManagedObjectModel.h

fetchRequestTemplateForName:

Returns the fetch request with a specified name.

- (NSFetchRequest *)fetchRequestTemplateName:(NSString *)name

Parameters

name

A string containing the name of a fetch request template.

Return Value

The fetch request named *name*.

Discussion

If the template contains substitution variables, you should instead use

[fetchRequestFromTemplateName:substitutionVariables:](#) (page 193) to create a new fetch request.

Availability

Available in iOS 3.0 and later.

See Also

- [fetchRequestTemplatesByName](#) (page 194)
- [fetchRequestFromTemplateName:substitutionVariables:](#) (page 193)
- [setFetchRequestTemplate:forName:](#) (page 197)

Declared In

NSManagedObjectModel.h

fetchRequestTemplatesByName

Returns a dictionary of the receiver's fetch request templates.

- (NSDictionary *)fetchRequestTemplatesByName

Return Value

A dictionary of the receiver's fetch request templates, keyed by name.

Discussion

If the template contains a predicate with substitution variables, you should instead use

[fetchRequestFromTemplateName:substitutionVariables:](#) (page 193) to create a new fetch request.

Availability

Available in iOS 3.0 and later.

See Also

- [fetchRequestTemplateName:](#) (page 193)
- [fetchRequestFromTemplateName:substitutionVariables:](#) (page 193)

Declared In

NSManagedObjectModel.h

initWithContentsOfURL:

Initializes the receiver using the model file at the specified URL.

```
- (id)initWithContentsOfURL:(NSURL *)url
```

Parameters

url

An URL object specifying the location of a model file.

Return Value

A managed object model initialized using the file at *url*.

Availability

Available in iOS 3.0 and later.

See Also

+ [mergedModelFromBundles:](#) (page 188)

+ [mergedModelFromBundles:forStoreMetadata:](#) (page 189)

+ [modelByMergingModels:](#) (page 189)

+ [modelByMergingModels:forStoreMetadata:](#) (page 190)

Declared In

NSManagedObjectModel.h

isConfiguration:compatibleWithStoreMetadata:

Returns a Boolean value that indicates whether a given configuration in the receiver is compatible with given metadata from a persistent store.

```
- (BOOL)isConfiguration:(NSString *)configuration
    compatibleWithStoreMetadata:(NSDictionary *)metadata
```

Parameters

configuration

The name of a configuration in the receiver. Pass *nil* to specify no configuration.

metadata

Metadata for a persistent store.

Return Value

YES if the configuration in the receiver specified by *configuration* is compatible with the store metadata given by *metadata*, otherwise NO.

Discussion

This method compares the version information in the store metadata with the entity versions of a given configuration. For information on specific differences, use [entityVersionHashesByName](#) (page 192) and perform an entity-by-entity comparison.

Availability

Available in iOS 3.0 and later.

See Also

- [entityVersionHashesByName](#) (page 192)

Declared In

NSManagedObjectModel.h

localizationDictionary

Returns the localization dictionary of the receiver.

- (NSDictionary *)localizationDictionary

Return Value

The localization dictionary of the receiver.

Discussion

The key-value pattern is described in [setLocalizationDictionary:](#) (page 198).

Special Considerations

On Mac OS X v10.4, `localizationDictionary` may return `nil` until Core Data lazily loads the dictionary for its own purposes (for example, reporting a localized error).

Availability

Available in iOS 3.0 and later.

See Also

- [setLocalizationDictionary:](#) (page 198)

Declared In

NSManagedObjectModel.h

setEntities:

Sets the entities array of the receiver.

- (void)setEntities:(NSArray *)*entities*

Parameters

entities

An array of instances of `NSEntityDescription`.

Special Considerations

This method raises an exception if the receiver has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [entities](#) (page 191)
- [entitiesByName](#) (page 191)
- [entitiesForConfiguration:](#) (page 192)
- [setEntities:forConfiguration:](#) (page 196)

Declared In

NSManagedObjectModel.h

setEntities:forConfiguration:

Associates the specified entities with the receiver using the given configuration name.

- (void)setEntities:(NSArray *)*entities* forConfiguration:(NSString *)*configuration*

Parameters

entities

An array of instances of `NSEntityDescription`.

configuration

A name for the configuration.

Special Considerations

This method raises an exception if the receiver has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [entities](#) (page 191)
- [entitiesByName](#) (page 191)
- [entitiesForConfiguration:](#) (page 192)
- [setEntities:](#) (page 196)

Declared In

`NSManagedObjectContext.h`

setFetchRequestTemplate:forName:

Associates the specified fetch request with the receiver using the given name.

- (void)setFetchRequestTemplate:(NSFetchRequest *)*fetchRequest* forName:(NSString *)*name*

Parameters

fetchRequest

A fetch request, typically containing predicates with variables for substitution.

name

A string that specifies the name of the fetch request template.

Discussion

For more details on using this method, see [Creating Predicates](#).

Special Considerations

This method raises an exception if the receiver has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [fetchRequestTemplatesByName](#) (page 194)
- [fetchRequestTemplateForName:](#) (page 193)
- [fetchRequestFromTemplateWithName:substitutionVariables:](#) (page 193)

Declared In

`NSManagedObjectContext.h`

setLocalizationDictionary:

Sets the localization dictionary of the receiver.

```
- (void)setLocalizationDictionary:(NSDictionary *)localizationDictionary
```

Parameters

localizationDictionary

A dictionary containing localized string values for entities, properties, and error strings related to the model. The key and value pattern is described in [Table 15-1](#) (page 198).

Discussion

[Table 15-1](#) (page 198) describes the key and value pattern for the localization dictionary.

Table 15-1 Key and value pattern for the localization dictionary.

Key	Value	Note
"Entity/NonLocalizedEntityName"	"LocalizedEntityName"	
"Property/NonLocalizedPropertyName/Entity/EntityName"	"LocalizedPropertyName"	(1)
"Property/NonLocalizedPropertyName"	"LocalizedPropertyName"	
"ErrorString/NonLocalizedErrorString"	"LocalizedErrorString"	

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

Availability

Available in iOS 3.0 and later.

See Also

- [LocalizationDictionary](#) (page 196)

Declared In

NSManagedObjectModel.h

setVersionIdentifiers:

Sets the identifiers for the receiver.

```
- (void)setVersionIdentifiers:(NSSet *)identifiers
```

Parameters

identifiers

An array of identifiers for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [versionIdentifiers](#) (page 199)

Declared In

NSManagedObjectModel.h

versionIdentifiers

Returns the collection of developer-defined version identifiers for the receiver.

```
- (NSSet *)versionIdentifiers
```

Return Value

The collection of developer-defined version identifiers for the receiver. Merged models return the combined collection of identifiers.

Discussion

The Core Data framework does not give models a default identifier, nor does it depend this value at runtime. For models created in Xcode, you set this value in the model inspector.

This value is meant to be used as a debugging hint to help you determine the models that were combined to create a merged model.

Availability

Available in iOS 3.0 and later.

See Also

- [setVersionIdentifiers:](#) (page 198)

Declared In

NSManagedObjectModel.h

NSMappingModel Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSMappingModel.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSMappingModel` specify how to map from a source to a destination managed object model.

Tasks

Creating a Mapping

- + [mappingModelFromBundles:forSourceModel:destinationModel:](#) (page 202)
Returns the mapping model to translate data from the source to the destination model.
- + [inferredMappingModelForSourceModel:destinationModel:error:](#) (page 202)
Returns a newly-created mapping model to migrate data from the source to the destination model.
- [initWithContentsOfURL:](#) (page 204)
Returns a mapping model initialized from a given URL.

Managing Entity Mappings

- [entityMappings](#) (page 203)
Returns the collection of entity mappings for the receiver.
- [setEntityMappings:](#) (page 204)
Sets the collection of entity mappings for the receiver
- [entityMappingsByName](#) (page 203)
Returns a dictionary of the entity mappings for the receiver.

Class Methods

inferredMappingModelForSourceModel:destinationModel:error:

Returns a newly-created mapping model to migrate data from the source to the destination model.

```
+ (NSMappingModel *)inferredMappingModelForSourceModel:(NSManagedObjectModel *)source
  destinationModel:(NSManagedObjectModel *)destination error:(NSError **)error
```

Parameters

source

The source managed object model.

destination

The destination managed object model.

error

If a problem occurs, on return contains an `NSInferredMappingModelError` error that describes the problem.

The error's `userInfo` will contain additional details about why inferring the mapping model failed (check for the following keys: `reason`, `entity`, `property`).

Return Value

A newly-created mapping model to migrate data from the source to the destination model. If the mapping model can not be created, returns `nil`.

Discussion

A model will be created only if all changes are simple enough to be able to reasonably infer a mapping (for example, removing or renaming an attribute, adding an optional attribute or relationship, or adding renaming or deleting an entity). Element IDs are used to track renamed properties and entities.

Availability

Available in iOS 3.0 and later.

Declared In

`NSMappingModel.h`

mappingModelFromBundles:forSourceModel:destinationModel:

Returns the mapping model to translate data from the source to the destination model.

```
+ (NSMappingModel *)mappingModelFromBundles:(NSArray *)bundles
  forSourceModel:(NSManagedObjectModel *)sourceModel
  destinationModel:(NSManagedObjectModel *)destinationModel
```

Parameters

bundles

An array of bundles in which to search for mapping models.

sourceModel

The managed object model for the source store.

destinationModel

The managed object model for the destination store.

Return Value

Returns the mapping model to translate data from *sourceModel* to *destinationModel*. If a suitable mapping model cannot be found, returns `nil`.

Discussion

This method is a companion to the [mergedModelFromBundles:](#) (page 188) and [mergedModelFromBundles:forStoreMetadata:](#) (page 189) methods. In this case, the framework uses the version information from the models to locate the appropriate mapping model in the available bundles.

Availability

Available in iOS 3.0 and later.

See Also

- [initWithContentsOfURL:](#) (page 204)

Declared In

NSMappingModel.h

Instance Methods

entityMappings

Returns the collection of entity mappings for the receiver.

- (NSArray *)entityMappings

Return Value

The collection of entity mappings for the receiver.

Special Considerations

The order of the mappings in the array specifies the order in which they will be processed during migration.

Availability

Available in iOS 3.0 and later.

See Also

- [setEntityMappings:](#) (page 204)

- [entityMappingsByName](#) (page 203)

Declared In

NSMappingModel.h

entityMappingsByName

Returns a dictionary of the entity mappings for the receiver.

- (NSDictionary *)entityMappingsByName

Return Value

A dictionary of the entity mappings for the receiver, keyed by their respective name.

Discussion

You can use this method to quickly access to mapping by name, rather than iterating the ordered array returned by [entityMappings](#) (page 203).

Availability

Available in iOS 3.0 and later.

See Also

- [entityMappings](#) (page 203)

Declared In

NSMappingModel.h

initWithContentsOfURL:

Returns a mapping model initialized from a given URL.

```
- (id)initWithContentsOfURL:(NSURL *)url
```

Parameters

url

The location of an archived mapping model.

Return Value

A mapping model initialized from *url*.

Availability

Available in iOS 3.0 and later.

See Also

+ [mappingModelFromBundles:forSourceModel:destinationModel:](#) (page 202)

Declared In

NSMappingModel.h

setEntityMappings:

Sets the collection of entity mappings for the receiver

```
- (void)setEntityMappings:(NSArray *)mappings
```

Parameters

mappings

The collection of entity mappings for the receiver.

Special Considerations

The order of the mappings specifies the order in which they will be processed during migration.

Availability

Available in iOS 3.0 and later.

See Also

- [entityMappings](#) (page 203)

Declared In

NSMappingModel.h

NSMigrationManager Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSMigrationManager.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSMigrationManager` perform a migration of data from one persistent store to another using a given mapping model.

Tasks

Initializing a Manager

- [initWithSourceModel:destinationModel:](#) (page 212)
Initializes a migration manager instance with given source and destination models.
- [setUserInfo:](#) (page 215)
Sets the user info for the receiver.

Performing Migration Operations

- [migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:](#) (page 213)
Migrates the store at a given source URL to the store at a given destination URL, performing all of the mappings specified in a given mapping model.
- [reset](#) (page 214)
Resets the association tables for the migration.
- [cancelMigrationWithError:](#) (page 209)
Cancels the migration with a given error.

Monitoring Migration Progress

- `migrationProgress` (page 214)
Returns a number from 0 to 1 that indicates the proportion of completeness of the migration.
- `currentEntityMapping` (page 210)
Returns the entity mapping currently being processed.

Working with Source and Destination Instances

- `associateSourceInstance:withDestinationInstance:forEntityMapping:` (page 208)
Associates a given source instance with an array of destination instances for a given property mapping.
- `destinationInstancesForEntityMappingNamed:sourceInstances:` (page 211)
Returns the managed object instances created in the destination store for a named entity mapping for a given array of source instances.
- `sourceInstancesForEntityMappingNamed:destinationInstances:` (page 216)
Returns the managed object instances in the source store used to create a given destination instance for a given property mapping.

Getting Information About a Migration Manager

- `mappingModel` (page 213)
Returns the mapping model for the receiver.
- `sourceModel` (page 217)
Returns the source model for the receiver.
- `destinationModel` (page 211)
Returns the destination model for the receiver.
- `sourceEntityForEntityMapping:` (page 215)
Returns the entity description for the source entity of a given entity mapping.
- `destinationEntityForEntityMapping:` (page 210)
Returns the entity description for the destination entity of a given entity mapping.
- `sourceContext` (page 215)
Returns the managed object context the receiver uses for reading the source persistent store.
- `destinationContext` (page 210)
Returns the managed object context the receiver uses for writing the destination persistent store.
- `userInfo` (page 217)
Returns the user info for the receiver.

Instance Methods

`associateSourceInstance:withDestinationInstance:forEntityMapping:`

Associates a given source instance with an array of destination instances for a given property mapping.


```
- (void)associateSourceInstance:(NSManagedObject *)sourceInstance
    withDestinationInstance:(NSManagedObject *)destinationInstance
    forEntityMapping:(NSEntityMapping *)entityMapping
```

Parameters

sourceInstance

A source managed object.

destinationInstance

The destination manage object for *sourceInstance*.

entityMapping

The entity mapping to use to associate *sourceInstance* with the object in *destinationInstances*.

Discussion

Data migration is performed as a three-stage process (first create the data, then relate the data, then validate the data). You use this method to associate data between the source and destination stores, in order to allow for relationship creation or fix-up after the creation stage.

This method is called in the default implementation of `NSEntityMigrationPolicy`'s `createDestinationInstancesForSourceInstance:entityMapping:manager:error:` (page 70) method.

Availability

Available in iOS 3.0 and later.

See Also

- `sourceInstancesForEntityMappingNamed:destinationInstances:` (page 216)
- `destinationInstancesForEntityMappingNamed:sourceInstances:` (page 211)

Declared In

`NSMigrationManager.h`

cancelMigrationWithError:

Cancels the migration with a given error.

```
- (void)cancelMigrationWithError:(NSError *)error
```

Parameters

error

An error object that describes the reason why the migration is canceled.

Discussion

You can invoke this method from anywhere in the migration process to abort the migration. Calling this method causes `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` (page 213) to abort the migration and return *error*—you should provide an appropriate error to indicate the reason for the cancellation.

Availability

Available in iOS 3.0 and later.

Declared In

`NSMigrationManager.h`

currentEntityMapping

Returns the entity mapping currently being processed.

- (NSEntityMapping *)currentEntityMapping

Return Value

The entity mapping currently being processed.

Discussion

Each entity is processed a total of three times (instance creation, relationship creation, validation).

Special Considerations

You can observe this value using key-value observing.

Availability

Available in iOS 3.0 and later.

Declared In

NSMigrationManager.h

destinationContext

Returns the managed object context the receiver uses for writing the destination persistent store.

- (NSManagedObjectContext *)destinationContext

Return Value

The managed object context the receiver uses for writing the destination persistent store.

Discussion

This context is created on demand as part of the initialization of the Core Data stacks used for migration.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceContext](#) (page 215)

Declared In

NSMigrationManager.h

destinationEntityForEntityMapping:

Returns the entity description for the destination entity of a given entity mapping.

- (NSEntityDescription *)destinationEntityForEntityMapping:(NSEntityMapping *)mEntity

Parameters

mEntity

An entity mapping.

Return Value

The entity description for the destination entity of *mEntity*.

Discussion

Entity mappings do not store the actual description objects, but rather the name and version information of the entity.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceEntityForEntityMapping:](#) (page 215)

Declared In

NSMigrationManager.h

destinationInstancesForEntityMappingNamed:sourceInstances:

Returns the managed object instances created in the destination store for a named entity mapping for a given array of source instances.

```
- (NSArray *)destinationInstancesForEntityMappingNamed:(NSString *)mappingName
    sourceInstances:(NSArray *)sourceInstances
```

Parameters

mappingName

The name of an entity mapping in use.

sourceInstances

A array of managed objects in the source store.

Return Value

An array containing the managed object instances created in the destination store for the entity mapping named *mappingName* for *sourceInstances*. If *sourceInstances* is *nil*, all of the destination instances created by the specified property mapping are returned.

Special Considerations

This method throws an `NSInvalidArgumentException` exception if *mappingName* is not a valid mapping name.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceInstancesForEntityMappingNamed:destinationInstances:](#) (page 216)

Declared In

NSMigrationManager.h

destinationModel

Returns the destination model for the receiver.

```
- (NSManagedObjectModel *)destinationModel
```

Return Value

The destination model for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [mappingModel](#) (page 213)
- [sourceModel](#) (page 217)
- [initWithSourceModel:destinationModel:](#) (page 212)

Declared In

NSMigrationManager.h

initWithSourceModel:destinationModel:

Initializes a migration manager instance with given source and destination models.

```
- (id)initWithSourceModel:(NSManagedObjectModel *)sourceModel
    destinationModel:(NSManagedObjectModel *)destinationModel
```

Parameters

sourceModel

The source managed object model for the migration manager.

destinationModel

The destination managed object model for the migration manager.

Return Value

A migration manager instance initialized to migrate data in a store that uses *sourceModel* to a store that uses *destinationModel*.

Discussion

You specify the mapping model in the migration method, [migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:](#) (page 213).

Special Considerations

This is the designated initializer for NSMigrationManager.

Although validation of the models is performed during [migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:](#) (page 213), as with NSPersistentStoreCoordinator once models are added to the migration manager they are immutable and cannot be altered.

Availability

Available in iOS 3.0 and later.

See Also

- [migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:](#) (page 213)
- [mappingModel](#) (page 213)
- [sourceModel](#) (page 217)
- [destinationModel](#) (page 211)

Declared In

NSMigrationManager.h

mappingModel

Returns the mapping model for the receiver.

- (NSMappingModel *)mappingModel

Return Value

The mapping model for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [sourceModel](#) (page 217)

- [destinationModel](#) (page 211)

- [migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:](#) (page 213)

Declared In

NSMigrationManager.h

migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:

Migrates of the store at a given source URL to the store at a given destination URL, performing all of the mappings specified in a given mapping model.

```
- (BOOL)migrateStoreFromURL:(NSURL *)sourceURL type:(NSString *)sStoreType
options:(NSDictionary *)sOptions withMappingModel:(NSMappingModel *)mappings
toDestinationURL:(NSURL *)dURL destinationType:(NSString *)dStoreType
destinationOptions:(NSDictionary *)dOptions error:(NSError **)error
```

Parameters

sourceURL

The location of an existing persistent store. A store must exist at this URL.

sStoreType

The type of store at *sourceURL* (see [NSPersistentStoreCoordinator](#) for possible values).

sOptions

A dictionary of options for the source (see [NSPersistentStoreCoordinator](#) for possible values).

mappings

The mapping model to use to effect the migration.

dURL

The location of the destination store.

dStoreType

The type of store at *dURL* (see [NSPersistentStoreCoordinator](#) for possible values).

dOptions

A dictionary of options for the destination (see [NSPersistentStoreCoordinator](#) for possible values).

error

If an error occurs during the validation or migration, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the migration proceeds without errors during the compatibility checks or migration, otherwise NO.

Discussion

This method performs compatibility checks on the source and destination models and the mapping model.

Special Considerations

If a store does not exist at the destination URL (*dURL*), one is created; otherwise, the migration appends to the existing store.

Availability

Available in iOS 3.0 and later.

See Also

- [cancelMigrationWithError:](#) (page 209)

Declared In

`NSMigrationManager.h`

migrationProgress

Returns a number from 0 to 1 that indicates the proportion of completeness of the migration.

```
- (float)migrationProgress
```

Return Value

A number from 0 to 1 that indicates the proportion of completeness of the migration. If a migration is not taking place, returns 1.

Special Considerations

You can observe this value using key-value observing.

Availability

Available in iOS 3.0 and later.

Declared In

`NSMigrationManager.h`

reset

Resets the association tables for the migration.

```
- (void)reset
```

Special Considerations

This method does not reset the source or destination contexts.

Availability

Available in iOS 3.0 and later.

Declared In

NSMigrationManager.h

setUserInfo:

Sets the user info for the receiver.

```
- (void)setUserInfo:(NSDictionary *)dict
```

Parameters*dict*

The user info for the receiver.

Discussion

You can use the user info dictionary to aid the customization of your migration process.

Availability

Available in iOS 3.0 and later.

See Also

- [userInfo](#) (page 217)

Declared In

NSMigrationManager.h

sourceContext

Returns the managed object context the receiver uses for reading the source persistent store.

```
- (NSManagedObjectContext *)sourceContext
```

Return Value

The managed object context the receiver uses for reading the source persistent store.

Discussion

This context is created on demand as part of the initialization of the Core Data stacks used for migration.

Availability

Available in iOS 3.0 and later.

See Also

- [destinationContext](#) (page 210)

Declared In

NSMigrationManager.h

sourceEntityForEntityMapping:

Returns the entity description for the source entity of a given entity mapping.

```
- (NSEntityDescription *)sourceEntityForEntityMapping:(NSEntityMapping *)mEntity
```

Parameters*mEntity*

An entity mapping.

Return ValueThe entity description for the source entity of *mEntity*.**Discussion**

Entity mappings do not store the actual description objects, but rather the name and version information of the entity.

Availability

Available in iOS 3.0 and later.

See Also- [destinationEntityForEntityMapping:](#) (page 210)**Declared In**

NSMigrationManager.h

sourceInstancesForEntityMappingNamed:destinationInstances:

Returns the managed object instances in the source store used to create a given destination instance for a given property mapping.

```
- (NSArray *)sourceInstancesForEntityMappingNamed:(NSString *)mappingName
    destinationInstances:(NSArray *)destinationInstances
```

Parameters*mappingName*

The name of an entity mapping in use.

destinationInstances

A array of managed objects in the destination store.

Return ValueAn array containing the managed object instances in the source store used to create *destinationInstances* using the entity mapping named *mappingName*. If *destinationInstances* is nil, all of the source instances used to create the destination instance for this property mapping are returned.**Special Considerations**This method throws an `NSInvalidArgumentException` exception if *mappingName* is not a valid mapping name.**Availability**

Available in iOS 3.0 and later.

See Also- [destinationInstancesForEntityMappingNamed:sourceInstances:](#) (page 211)**Declared In**

NSMigrationManager.h

sourceModel

Returns the source model for the receiver.

- (NSManagedObjectModel *)sourceModel

Return Value

The source model for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [mappingModel](#) (page 213)
- [destinationModel](#) (page 211)
- [initWithSourceModel:destinationModel:](#) (page 212)

Declared In

NSMigrationManager.h

userInfo

Returns the user info for the receiver.

- (NSDictionary *)userInfo

Return Value

The user info for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setUserInfo:](#) (page 215)

Declared In

NSMigrationManager.h

NSPersistentStore Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSPersistentStore.h
Companion guides	Core Data Programming Guide Atomic Store Programming Topics

Overview

This class is the abstract base class for all Core Data persistent stores.

Core Data provides four store types—SQLite, Binary, XML, and In-Memory (the XML store is not available on iOS); these are described in [Persistent Stores](#). Core Data also provides a subclass of `NSPersistentStore`, `NSAtomicStore`. The Binary and XML stores are examples of atomic stores that inherit functionality from `NSAtomicStore`.

Subclassing Notes

You should not subclass `NSPersistentStore` directly. Core Data currently only supports subclassing of `NSAtomicStore`.

The designated initializer is

[initWithPersistentStoreCoordinator:configurationName:URL:options:](#) (page 224). When you implement the initializer, you must ensure you load metadata during initialization and set it using [setMetadata:](#) (page 226).

You must override these methods:

- [type](#) (page 227)
- [metadata](#) (page 225)
- [metadataForPersistentStoreWithURL:error:](#) (page 221)
- [setMetadata:forPersistentStoreWithURL:error:](#) (page 222)

Tasks

Initializing a Persistent Store

- [initWithPersistentStoreCoordinator:configurationName:URL:options:](#) (page 224)
Returns a store initialized with the given arguments.

Working with State Information

- [type](#) (page 227)
Returns the type string of the receiver.
- [persistentStoreCoordinator](#) (page 226)
Returns the persistent store coordinator which loaded the receiver.
- [configurationName](#) (page 222)
Returns the name of the managed object model configuration used to create the receiver.
- [options](#) (page 225)
Returns the options with which the receiver was created.
- [URL](#) (page 228)
Returns the URL for the receiver.
- [setURL:](#) (page 227)
Sets the URL for the receiver.
- [identifier](#) (page 223)
Returns the unique identifier for the receiver.
- [setIdentifier:](#) (page 226)
Sets the unique identifier for the receiver.
- [isReadOnly](#) (page 224)
Returns a Boolean value that indicates whether the receiver is read-only.
- [setReadOnly:](#) (page 227)
Sets whether the receiver is read-only.

Managing Metadata

- + [metadataForPersistentStoreWithURL:error:](#) (page 221)
Returns the metadata from the persistent store at the given URL.
- + [setMetadata:forPersistentStoreWithURL:error:](#) (page 222)
Sets the metadata for the store at a given URL.
- [metadata](#) (page 225)
Returns the metadata for the receiver.
- [loadMetadata:](#) (page 225)
Instructs the receiver to load its metadata.
- [setMetadata:](#) (page 226)
Sets the metadata for the receiver.

Setup and Teardown

- `didAddToPersistentStoreCoordinator:` (page 223)
Invoked after the receiver has been added to the persistent store coordinator.
- `willRemoveFromPersistentStoreCoordinator:` (page 228)
Invoked before the receiver is removed from the persistent store coordinator.

Supporting Migration

- + `migrationManagerClass` (page 221)
Returns the `NSMigrationManager` class for this store class.

Class Methods

metadataForPersistentStoreWithURL:error:

Returns the metadata from the persistent store at the given URL.

```
+ (NSDictionary *)metadataForPersistentStoreWithURL:(NSURL *)url error:(NSError **)error
```

Parameters

url

The location of the store.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

The metadata from the persistent store at *url*. Returns `nil` if there is an error.

Special Considerations

Subclasses must override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStore.h`

migrationManagerClass

Returns the `NSMigrationManager` class for this store class.

```
+ (Class)migrationManagerClass
```

Return Value

The `NSMigrationManager` class for this store class

Discussion

In a subclass of `NSPersistentStore`, you can override this to provide a custom migration manager subclass (for example, to take advantage of store-specific functionality to improve migration performance).

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStore.h`

setMetadata:forPersistentStoreWithURL:error:

Sets the metadata for the store at a given URL.

```
+ (BOOL)setMetadata:(NSDictionary *)metadata forPersistentStoreWithURL:(NSURL *)url
      error:(NSError **)error
```

Parameters

metadata

The metadata for the store at *url*.

url

The location of the store.

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the metadata was written correctly, otherwise NO.

Special Considerations

Subclasses must override this method to set metadata appropriately.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStore.h`

Instance Methods

configurationName

Returns the name of the managed object model configuration used to create the receiver.

```
- (NSString *)configurationName
```

Return Value

The name of the managed object model configuration used to create the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

didAddToPersistentStoreCoordinator:

Invoked after the receiver has been added to the persistent store coordinator.

```
- (void)didAddToPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
```

Parameters*coordinator*

The persistent store coordinator to which the receiver was added.

Discussion

The default implementation does nothing. You can override this method in a subclass in order to perform any kind of setup necessary before the load method is invoked.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

identifier

Returns the unique identifier for the receiver.

```
- (NSString *)identifier
```

Return Value

The unique identifier for the receiver.

Discussion

The identifier is used as part of the managed object IDs for each object in the store.

Special Considerations

`NSPersistentStore` provides a default implementation to provide a globally unique identifier for the store instance.

Availability

Available in iOS 3.0 and later.

See Also

- [setIdentifier:](#) (page 226)

- [setMetadata:](#) (page 226)

Declared In

NSPersistentStore.h

initWithPersistentStoreCoordinator:configurationName:URL:options:

Returns a store initialized with the given arguments.

```
- (id)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)root
    configurationName:(NSString *)name URL:(NSURL *)url options:(NSDictionary
    *)options
```

Parameters

coordinator

A persistent store coordinator.

configurationName

The name of the managed object model configuration to use. Pass `nil` if you do not want to specify a configuration.

url

The URL of the store to load.

options

A dictionary containing configuration options.

Return Value

A new store object, associated with *coordinator*, that represents a persistent store at *url* using the options in *options* and—if it is not `nil`—the managed object model configuration *configurationName*.

Discussion

You must ensure that you load metadata during initialization and set it using [setMetadata:](#) (page 226).

Special Considerations

This is the designated initializer for persistent stores.

Availability

Available in iOS 3.0 and later.

See Also

- [setMetadata:](#) (page 226)

Declared In

NSPersistentStore.h

isReadOnly

Returns a Boolean value that indicates whether the receiver is read-only.

```
- (BOOL)isReadOnly
```

Return Value

YES if the receiver is read-only, otherwise NO.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

loadMetadata:

Instructs the receiver to load its metadata.

```
- (BOOL)loadMetadata:(NSError **)error
```

Parameters

error

If an error occurs, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the metadata was loaded correctly, otherwise NO.

Special Considerations

There is no way to return an error if the store is invalid.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStore.h`

metadata

Returns the metadata for the receiver.

```
- (NSDictionary *)metadata
```

Return Value

The metadata for the receiver. The dictionary must include the store type (`NSStoreTypeKey`) and UUID (`NSStoreUUIDKey`).

Special Considerations

Subclasses must override this method to provide storage and persistence for the store metadata.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStore.h`

options

Returns the options with which the receiver was created.

```
- (NSDictionary *)options
```

Return Value

The options with which the receiver was created.

Discussion

See `NSPersistentStoreCoordinator` for a list of key names for options in this dictionary.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

persistentStoreCoordinator

Returns the persistent store coordinator which loaded the receiver.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
```

Return Value

The persistent store coordinator which loaded the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

setIdentifier:

Sets the unique identifier for the receiver.

```
- (void)setIdentifier:(NSString *)identifier
```

Parameters

identifier

The unique identifier for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [identifier](#) (page 223)

- [metadata](#) (page 225)

Declared In

NSPersistentStore.h

setMetadata:

Sets the metadata for the receiver.

```
- (void)setMetadata:(NSDictionary *)storeMetadata
```

Parameters

storeMetadata

The metadata for the receiver.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

setReadOnly:

Sets whether the receiver is read-only.

```
- (void)setReadOnly:(BOOL)flag
```

Parameters*flag*

YES if the receiver is read-only, otherwise NO.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

setURL:

Sets the URL for the receiver.

```
- (void)setURL:(NSURL *)url
```

Parameters*url*

The URL for the receiver.

DiscussionTo alter the location of a store, send the persistent store coordinator a [setURL:forPersistentStore:](#) (page 239) message.**Availability**

Available in iOS 3.0 and later.

See Also

- [URL](#) (page 228)

Declared In

NSPersistentStore.h

type

Returns the type string of the receiver.

```
- (NSString *)type
```

Return Value

The type string of the receiver.

Discussion

This string is used when specifying the type of store to add to a persistent store coordinator.

Special Considerations

Subclasses must override this method to provide a unique type.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

URL

Returns the URL for the receiver.

- (NSURL *)URL

Return Value

The URL for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setURL:](#) (page 227)

Declared In

NSPersistentStore.h

willRemoveFromPersistentStoreCoordinator:

Invoked before the receiver is removed from the persistent store coordinator.

```
- (void)willRemoveFromPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
```

Parameters

coordinator

The persistent store coordinator from which the receiver was removed.

Discussion

The default implementation does nothing. You can override this method in a subclass in order to perform any clean-up before the store is removed from the coordinator (and deallocated).

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStore.h

NSPersistentStoreCoordinator Class Reference

Inherits from	NSObject
Conforms to	NSLocking NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSPersistentStoreCoordinator.h
Companion guides	Core Data Programming Guide Atomic Store Programming Topics Core Data Spotlight Integration Programming Guide

Overview

Instances of `NSPersistentStoreCoordinator` associate persistent stores (by type) with a model (or more accurately, a configuration of a model) and serve to mediate between the persistent store or stores and the managed object context or contexts. Instances of `NSManagedObjectContext` use a coordinator to save object graphs to persistent storage and to retrieve model information. A context without a coordinator is not fully functional as it cannot access a model except through a coordinator. The coordinator is designed to present a façade to the managed object contexts such that a group of persistent stores appears as an aggregate store. A managed object context can then create an object graph based on the union of all the data stores the coordinator covers.

Coordinators do the opposite of providing for concurrency—they **serialize** operations. If you want to use multiple threads for different write operations you use multiple coordinators. Note that if multiple threads work directly with a coordinator, they need to lock and unlock it explicitly.

Each coordinator (and thus container) may use different copies, and hence different versions, of a managed object model. This allows you to cleanly deal with file versioning.

The coordinator gives access to its underlying object stores. You can retrieve an object store when you first add one (using `addPersistentStoreWithType:configuration:URL:options:error:` (page 233)), or by using `persistentStoreForURL:` (page 237) or `persistentStores` (page 238). This allows you to determine, for example, whether a store has already been added, or whether two objects come from the same store.

- You move a store from one location to another, or change the type of a store, using `migratePersistentStore:toURL:options:withType:error:` (page 236).
- You can set metadata for a given store using the persistent store coordinator (`setMetadata:forPersistentStore:` (page 238)).

For more details about these tasks, see Persistent Store Features in *Core Data Programming Guide*.

Tasks

Registered Store Types

- + [registeredStoreTypes](#) (page 232)
Returns a dictionary of the registered store types.
- + [registerStoreClass:forStoreType:](#) (page 232)
Registers a given `NSPersistentStore` subclass for a given store type string.

Initializing a Coordinator

- [initWithManagedObjectModel:](#) (page 234)
Initializes the receiver with a managed object model.
- [managedObjectModel](#) (page 235)
Returns the receiver's managed object model.

Configuring Persistent Stores

- [addPersistentStoreWithType:configuration:URL:options:error:](#) (page 233)
Adds a new persistent store of a specified type at a given location, and returns the new store.
- [setURL:forPersistentStore:](#) (page 239)
Sets the URL for a given persistent store.
- [removePersistentStore:error:](#) (page 238)
Removes a given persistent store.
- [migratePersistentStore:toURL:options:withType:error:](#) (page 236)
Moves a persistent store to a new location, changing the storage type if necessary.
- [persistentStores](#) (page 238)
Returns an array of persistent stores associated with the receiver.
- [persistentStoreForURL:](#) (page 237)
Returns the persistent store for the specified URL.
- [URLForPersistentStore:](#) (page 240)
Returns the URL for a given persistent store.

Locking

- [lock](#) (page 235)
Attempts to acquire a lock.
- [tryLock](#) (page 240)
Attempts to acquire a lock.

- `unlock` (page 240)
Relinquishes a previously acquired lock.

Working with Metadata

- `metadataForPersistentStore:` (page 236)
Returns a dictionary that contains the metadata currently stored or to-be-stored in a given persistent store.
- `setMetadata:forPersistentStore:` (page 238)
Sets the metadata stored in the persistent store during the next save operation executed on it to *metadata*.
- + `setMetadata:forPersistentStoreOfType:URL:error:` (page 233)
Sets the metadata for a given store.
- + `metadataForPersistentStoreOfType:URL:error:` (page 231)
Returns a dictionary containing the metadata stored in the persistent store at a given URL.

Discovering Object IDs

- `managedObjectIDForURIRepresentation:` (page 235)
Returns an object ID for the specified URI representation of an object ID if a matching store is available, or `nil` if a matching store cannot be found.

Class Methods

`metadataForPersistentStoreOfType:URL:error:`

Returns a dictionary containing the metadata stored in the persistent store at a given URL.

```
+ (NSDictionary *)metadataForPersistentStoreOfType:(NSString *)storeType URL:(NSURL *)url error:(NSError **)error
```

Parameters

storeType

The type of the store at *url*. If this value is `nil`, Core Data determines which store class should be used to get or set the store file's metadata by inspecting the file contents.

url

The location of a persistent store.

error

If no store is found at *url* or if there is a problem accessing its contents, upon return contains an `NSError` object that describes the problem.

Return Value

A dictionary containing the metadata stored in the persistent store at *url*, or `nil` if the store cannot be opened or if there is a problem accessing its contents.

The keys guaranteed to be in this dictionary are [NSToreTypeKey](#) (page 241) and [NSToreUUIDKey](#) (page 242).

Discussion

You can use this method to retrieve the metadata from a store without the overhead of creating a Core Data stack.

Availability

Available in iOS 3.0 and later.

See Also

- + [setMetadata:forPersistentStoreOfType:URL:error:](#) (page 233)
- [metadataForPersistentStore:](#) (page 236)
- [setMetadata:forPersistentStore:](#) (page 238)

Declared In

NSPersistentStoreCoordinator.h

registeredStoreTypes

Returns a dictionary of the registered store types.

```
+ (NSDictionary *)registeredStoreTypes
```

Return Value

A dictionary of the registered store types—the keys are the store type strings, and the values are the `NSPersistentStore` subclasses.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStoreCoordinator.h

registerStoreClass:forStoreType:

Registers a given `NSPersistentStore` subclass for a given store type string.

```
+ (void)registerStoreClass:(Class)storeClass forStoreType:(NSString *)storeType
```

Parameters

storeClass

The `NSPersistentStore` subclass to use for the store of type *storeType*.

storeType

A unique string that identifies a store type.

Discussion

You must invoke this method before a custom subclass of `NSPersistentStore` can be loaded into a persistent store coordinator.

You can pass `nil` for *storeClass* to unregister the store type.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStoreCoordinator.h

setMetadata:forPersistentStoreOfType:URL:error:

Sets the metadata for a given store.

```
+ (BOOL)setMetadata:(NSDictionary *)metadata forPersistentStoreOfType:(NSString *)storeType URL:(NSURL *)url error:(NSError **)error
```

Parameters

metadata

A dictionary containing metadata for the store.

storeType

The type of the store at *url*. If this value is *nil*, Core Data will determine which store class should be used to get or set the store file's metadata by inspecting the file contents.

url

The location of a persistent store.

error

If no store is found at *url* or if there is a problem setting its metadata, upon return contains an `NSError` object that describes the problem.

Return Value

YES if the metadata was set correctly, otherwise NO.

Discussion

You can use this method to set the metadata for a store without the overhead of creating a Core Data stack.

Availability

Available in iOS 3.0 and later.

See Also

- + [metadataForPersistentStoreOfType:URL:error:](#) (page 231)
- [metadataForPersistentStore:](#) (page 236)
- [setMetadata:forPersistentStore:](#) (page 238)

Declared In

NSPersistentStoreCoordinator.h

Instance Methods

addPersistentStoreWithType:configuration:URL:options:error:

Adds a new persistent store of a specified type at a given location, and returns the new store.

```
- (NSPersistentStore *)addPersistentStoreWithType:(NSString *)storeType
    configuration:(NSString *)configuration URL:(NSURL *)storeURL
    options:(NSDictionary *)options error:(NSError **)error
```

Parameters*storeType*

A string constant (such as `NSSQLiteStoreType`) that specifies the store type—see “Store Types” (page 241) for possible values.

configuration

The name of a configuration in the receiver's managed object model that will be used by the new store. The configuration can be `nil`, in which case no other configurations are allowed.

storeURL

The file location of the persistent store.

options

A dictionary containing key-value pairs that specify whether the store should be read-only, and whether (for an XML store) the XML file should be validated against the DTD before it is read. For key definitions, see “Store Options” (page 242) and “Migration Options” (page 243). This value may be `nil`.

error

If a new store cannot be created, upon return contains an instance of `NSError` that describes the problem

Return Value

The newly-created store or, if an error occurs, `nil`.

Availability

Available in iOS 3.0 and later.

See Also

- [migratePersistentStore:toURL:options:withType:error:](#) (page 236)
- [removePersistentStore:error:](#) (page 238)

Declared In

`NSPersistentStoreCoordinator.h`

initWithManagedObjectModel:

Initializes the receiver with a managed object model.

```
- (id)initWithManagedObjectModel:(NSManagedObjectModel *)model
```

Parameters*model*

A managed object model.

Return Value

The receiver, initialized with *model*.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStoreCoordinator.h`

lock

Attempts to acquire a lock.

- (void)lock

Discussion

This method blocks a thread's execution until the lock can be acquired. An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking `unlock`.

Availability

Available in iOS 3.0 and later.

See Also

- [tryLock](#) (page 240)
- [unlock](#) (page 240)

Declared In

NSPersistentStoreCoordinator.h

managedObjectIDForURIRepresentation:

Returns an object ID for the specified URI representation of an object ID if a matching store is available, or `nil` if a matching store cannot be found.

- (NSManagedObjectID *)managedObjectIDForURIRepresentation:(NSURL *)URL

Parameters

URL

An URL object containing a URI that specify a managed object.

Return Value

An object ID for the object specified by *URL*.

Discussion

The URI representation contains a UUID of the store the ID is coming from, and the coordinator can match it against the stores added to it.

Availability

Available in iOS 3.0 and later.

See Also

- [URIRepresentation](#) (page 183) (NSManagedObjectID)
- [objectWithID:](#) (page 165) (NSManagedObjectContext)

Declared In

NSPersistentStoreCoordinator.h

managedObjectModel

Returns the receiver's managed object model.

- (NSManagedObjectModel *)managedObjectModel

Return Value

The receiver's managed object model.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStoreCoordinator.h

metadataForPersistentStore:

Returns a dictionary that contains the metadata currently stored or to-be-stored in a given persistent store.

```
- (NSDictionary *)metadataForPersistentStore:(NSPersistentStore *)store
```

Parameters

store

A persistent store.

Return Value

A dictionary that contains the metadata currently stored or to-be-stored in *store*.

Availability

Available in iOS 3.0 and later.

See Also

- [setMetadata:forPersistentStore:](#) (page 238)
- + [metadataForPersistentStoreOfType:URL:error:](#) (page 231)
- + [setMetadata:forPersistentStoreOfType:URL:error:](#) (page 233)

Declared In

NSPersistentStoreCoordinator.h

migratePersistentStore:toURL:options:withType:error:

Moves a persistent store to a new location, changing the storage type if necessary.

```
- (NSPersistentStore *)migratePersistentStore:(NSPersistentStore *)store toURL:(NSURL *)URL options:(NSDictionary *)options withType:(NSString *)storeType error:(NSError **)error
```

Parameters

store

A persistent store.

URL

An URL object that specifies the location for the new store.

options

A dictionary containing key-value pairs that specify whether the store should be read-only, and whether (for an XML store) the XML file should be validated against the DTD before it is read. For key definitions, see [“Store Options”](#) (page 242).

storeType

A string constant (such as `NSSQLiteStoreType`) that specifies the type of the new store—see “Store Types” (page 241).

error

If an error occurs, upon return contains an instance of `NSError` that describes the problem.

Return Value

If the migration is successful, the new store, otherwise `nil`.

Discussion

This method is typically used for “Save As” operations. Performance may vary depending on the type of old and new store. For more details of the action of this method, see Persistent Store Features in *Core Data Programming Guide*.

Important: After invocation of this method, the specified store is removed from the coordinator thus `store` is no longer a useful reference.

Availability

Available in iOS 3.0 and later.

See Also

- [addPersistentStoreWithType:configuration:URL:options:error:](#) (page 233)
- [removePersistentStore:error:](#) (page 238)

Declared In

`NSPersistentStoreCoordinator.h`

persistentStoreForURL:

Returns the persistent store for the specified URL.

```
- (NSPersistentStore *)persistentStoreForURL:(NSURL *)URL
```

Parameters

URL

An URL object that specifies the location of a persistent store.

Return Value

The persistent store at the location specified by *URL*.

Availability

Available in iOS 3.0 and later.

See Also

- [persistentStores](#) (page 238)
- [URLForPersistentStore:](#) (page 240)

Declared In

`NSPersistentStoreCoordinator.h`

persistentStores

Returns an array of persistent stores associated with the receiver.

- (NSArray *)persistentStores

Return Value

An array persistent stores associated with the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [persistentStoreForURL:](#) (page 237)
- [URLForPersistentStore:](#) (page 240)

Declared In

NSPersistentStoreCoordinator.h

removePersistentStore:error:

Removes a given persistent store.

- (BOOL)removePersistentStore:(NSPersistentStore *)store error:(NSError **)error

Parameters

store

A persistent store.

error

If an error occurs, upon return contains an instance of NSError that describes the problem.

Return Value

YES if the store is removed, otherwise NO.

Availability

Available in iOS 3.0 and later.

See Also

- [addPersistentStoreWithType:configuration:URL:options:error:](#) (page 233)
- [migratePersistentStore:toURL:options:withType:error:](#) (page 236)

Declared In

NSPersistentStoreCoordinator.h

setMetadata:forPersistentStore:

Sets the metadata stored in the persistent store during the next save operation executed on it to *metadata*.

- (void)setMetadata:(NSDictionary *)metadata forPersistentStore:(NSPersistentStore *)store

Parameters*metadata*

A dictionary containing metadata for the store.

store

A persistent store.

Discussion

The store type and UUID (`NSStoreTypeKey` and `NSStoreUUIDKey`) are always added automatically, however `NSStoreUUIDKey` is only added if it is not set manually as part of the dictionary argument.

Important: Setting the metadata for a store does not change the information on disk until the store is actually saved.

Availability

Available in iOS 3.0 and later.

See Also

- [metadataForPersistentStore:](#) (page 236)
- + [setMetadata:forPersistentStoreOfType:URL:error:](#) (page 233)
- + [metadataForPersistentStoreOfType:URL:error:](#) (page 231)

Declared In

NSPersistentStoreCoordinator.h

setURL:forPersistentStore:

Sets the URL for a given persistent store.

```
-(BOOL)setURL:(NSURL *)url forPersistentStore:(NSPersistentStore *)store
```

Parameters*url*The new location for *store*.*store*

A persistent store associated with the receiver.

Return Value

YES if the store was relocated, otherwise NO.

Discussion

For atomic stores, this method alters the location to which the next save operation will write the file; for non-atomic stores, invoking this method will release the existing connection and create a new one at the specified URL. (For non-atomic stores, a store must already exist at the destination URL; a new store will not be created.)

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStoreCoordinator.h

tryLock

Attempts to acquire a lock.

- (BOOL)tryLock

Return Value

YES if successful, otherwise NO.

Discussion

Returns immediately—contrast [lock](#) (page 235) which blocks.

Availability

Available in iOS 3.0 and later.

See Also

- [lock](#) (page 235)
- [unlock](#) (page 240)

Declared In

NSPersistentStoreCoordinator.h

unlock

Relinquishes a previously acquired lock.

- (void)unlock

Availability

Available in iOS 3.0 and later.

See Also

- [lock](#) (page 235)
- [tryLock](#) (page 240)

Declared In

NSPersistentStoreCoordinator.h

URLForPersistentStore:

Returns the URL for a given persistent store.

- (NSURL *)URLForPersistentStore:(NSPersistentStore *)store

Parameters

store

A persistent store.

Return Value

The URL for *store*.

Availability

Available in iOS 3.0 and later.

See Also

- [persistentStoreForURL:](#) (page 237)
- [persistentStores](#) (page 238)

Declared In

NSPersistentStoreCoordinator.h

Constants

Store Types

Types of persistent store.

```
NSString * const NSSQLiteStoreType;
NSString * const NSBinaryStoreType;
NSString * const NSInMemoryStoreType;
```

Constants

NSSQLiteStoreType

The SQLite database store type.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

NSBinaryStoreType

The binary store type.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

NSInMemoryStoreType

The in-memory store type.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Store Metadata

Keys used in a store's metadata dictionary.

```
NSString * const NSStoreTypeKey;
NSString * const NSStoreUUIDKey;
```

Constants

NSStoreTypeKey

The key in the metadata dictionary to identify the store type.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

NSStoreUUIDKey

The key in the metadata dictionary to identify the store UUID.

The store UUID is useful to identify stores through URI representations, but it is *not* guaranteed to be unique. The UUID generated for new stores is unique—users can freely copy files and thus the UUID stored inside—so if you track or reference stores explicitly you need to be aware of duplicate UUIDs and potentially override the UUID when a new store is added to the list of known stores in your application.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Declared In

`NSPersistentStoreCoordinator.h`

Stores Change Notification User Info Keys

An `NSPersistentStoreCoordinatorStoresDidChangeNotification` (page 245) notification is posted whenever persistent stores are added to or removed from a persistent store coordinator, or when store UUIDs change. The `userInfo` dictionary contains information about the stores that were added or removed using these keys.

```
NSString * const NSAddedPersistentStoresKey;
NSString * const NSRemovedPersistentStoresKey;
NSString * const NSUUIDChangedPersistentStoresKey;
```

Constants

`NSAddedPersistentStoresKey`

Key for the array of stores that were added.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSRemovedPersistentStoresKey`

Key for the array of stores that were removed.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSUUIDChangedPersistentStoresKey`

Key for the array of stores whose UUIDs changed.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Declared In

`NSPersistentStoreCoordinator.h`

Store Options

Keys for the options dictionary used in

`addPersistentStoreWithType:configuration:URL:options:error:` (page 233),

`migratePersistentStore:toURL:options:withType:error:` (page 236), and

`importStoreWithIdentifier:fromExternalRecordsDirectory:toURL:options:withType:error:`.

```

NSString * const NSReadOnlyPersistentStoreOption;
NSString * const NSPersistentStoreTimeoutOption;
NSString * const NSSQLitePragmasOption;
NSString * const NSSQLiteAnalyzeOption;
NSString * const NSSQLiteManualVacuumOption;

```

Constants

`NSReadOnlyPersistentStoreOption`

A flag that indicates whether a store is treated as read-only or not.

The default value is `NO`.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSPersistentStoreTimeoutOption`

Options key that specifies the connection timeout for Core Data stores.

The corresponding value is an `NSNumber` object that represents the duration in seconds that Core Data will wait while attempting to create a connection to a persistent store. If a connection is cannot be made within that timeframe, the operation is aborted and an error is returned.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSSQLitePragmasOption`

Options key for a dictionary of SQLite pragma settings with pragma values indexed by pragma names as keys.

All pragma values must be specified as `NSString` objects. The `fullfsync` and `synchronous` pragmas control the tradeoff between write performance (write to disk speed & cache utilization) and durability (data loss/corruption sensitivity to power interruption). For more information on pragma settings, see <http://sqlite.org/pragma.html>.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSSQLiteAnalyzeOption`

Option key to run an analysis of the store data to optimize indices based on statistical information when the store is added to the coordinator.

This invokes SQLite's `ANALYZE` command. It is ignored by stores other than the SQLite store.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSSQLiteManualVacuumOption`

Option key to rebuild the store file, forcing a database wide defragmentation when the store is added to the coordinator.

This invokes SQLite's `VACUUM` command. It is ignored by stores other than the SQLite store.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Migration Options

Migration options, specified in the dictionary of options when adding a persistent store using `addPersistentStoreWithType:configuration:URL:options:error:` (page 233).

```
NSString * const NSIgnorePersistentStoreVersioningOption;
NSString * const NSMigratePersistentStoresAutomaticallyOption;
NSString * const NSInferMappingModelAutomaticallyOption;
```

Constants

`NSIgnorePersistentStoreVersioningOption`

Key to ignore the built-in versioning provided by Core Data.

The corresponding value is an `NSNumber` object. If the `boolValue` of the number is `YES`, Core Data will not compare the version hashes between the managed object model in the coordinator and the metadata for the loaded store. (It will, however, continue to update the version hash information in the metadata.) This key and corresponding value of `YES` is specified by default for all applications linked on or before Mac OS X 10.4.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSMigratePersistentStoresAutomaticallyOption`

Key to automatically attempt to migrate versioned stores.

The corresponding value is an `NSNumber` object. If the `boolValue` of the number is `YES` and if the version hash information for the added store is determined to be incompatible with the model for the coordinator, Core Data will attempt to locate the source and mapping models in the application bundles, and perform a migration.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSInferMappingModelAutomaticallyOption`

Key to attempt to create the mapping model automatically.

The corresponding value is an `NSNumber` object. If the `boolValue` of the number is `YES` and the value of the `NSMigratePersistentStoresAutomaticallyOption` is `YES`, the coordinator will attempt to infer a mapping model if none can be found.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Versioning Support

Keys in store metadata to support versioning.

```
NSString * const NSSStoreModelVersionHashesKey;
NSString * const NSSStoreModelVersionIdentifiersKey;
NSString * const NSPersistentStoreOSCompatibility;
```

Constants

`NSSStoreModelVersionHashesKey`

Key to represent the version hash information for the model used to create the store.

This key is used in the metadata for a persistent store.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

NSStoreModelVersionIdentifiersKey

Key to represent the version identifiers for the model used to create the store.

If you add your own annotations to a model's version identifier (see [versionIdentifiers](#) (page 199)), they are stored in the persistent store's metadata. You can use this key to retrieve the identifiers from the metadata dictionaries available from `NSPersistentStore` ([metadata](#) (page 225)) and `NSPersistentStoreCoordinator` ([metadataForPersistentStore:](#) (page 236) and related methods). The corresponding value is a Foundation collection (an `NSArray` or `NSSet` object).

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

NSPersistentStoreOSCompatibility

Key to represent the earliest version of Mac OS X the persistent store supports.

The corresponding value is an `NSNumber` object that takes the form of the constants defined by the Mac OS X availability macros (defined in `/usr/include/AvailabilityMacros.h`), for example 1040 represents Mac OS X version 10.4.0.

Backward compatibility may preclude some features.

Available in iOS 3.0 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Notifications

NSPersistentStoreCoordinatorStoresDidChangeNotification

Posted whenever persistent stores are added to or removed from a persistent store coordinator, or when store UUIDs change.

The notification's object is the persistent store coordinator that was affected. The notification's *userInfo* dictionary contains information about the stores that were added or removed, specified using the following keys:

NSAddedPersistentStoresKey (page 242)	Key for the array of stores that were added.
NSRemovedPersistentStoresKey (page 242)	Key for the array of stores that were removed.
NSUUIDChangedPersistentStoresKey (page 242)	Key for the array of stores whose UUIDs changed.

Availability

Available in iOS 3.0 and later.

Declared In

`NSPersistentStoreCoordinator.h`

NSPersistentStoreCoordinatorWillRemoveStoreNotification

Posted whenever a persistent store is removed from a persistent store coordinator.

The notification is sent during the invocation of `NSPersistentStore`'s `willRemoveFromPersistentStoreCoordinator` method during store deallocation or removal. The notification's object is the persistent store coordinator will be removed.

Availability

Available in iOS 3.0 and later.

Declared In

NSPersistentStoreCoordinator.h

NSPropertyDescription Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSCopying NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSPropertyDescription.h
Companion guide	Core Data Programming Guide

Overview

The `NSPropertyDescription` class is used to define properties of an entity in a Core Data managed object model. Properties are to entities what instance variables are to classes.

A property describes a single value within an object managed by the Core Data Framework. There are different types of property, each represented by a subclass which encapsulates the specific property behavior—see `NSAttributeDescription`, `NSRelationshipDescription`, and `NSFetchedPropertyDescription`.

Note that 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". There are hundreds of methods on `NSObject` which may conflict with property names—and this list can grow without warning from frameworks or other libraries. You should avoid very general words (like "font," and "color") and words or phrases which overlap with Cocoa paradigms (such as "isEditing" and "objectSpecifier").

Properties—relationships as well as attributes—may be transient. A managed object context knows about transient properties and tracks changes made to them. Transient properties are ignored by the persistent store, and not just during saves: you cannot fetch using a predicate based on transients (although you can use transient properties to filter in memory yourself).

Editing Property Descriptions

Property descriptions are editable until they are used by an object graph manager (such as a persistent store coordinator). This allows you to create or modify them dynamically. However, once a description is used (when the managed object model to which it belongs is associated with a persistent store coordinator), it *must not* (indeed cannot) be changed. This is enforced at runtime: any attempt to mutate a model or any of

its sub-objects after the model is associated with a persistent store coordinator causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

Tasks

Getting Features of a Property

- [entity](#) (page 249)
Returns the entity description of the receiver.
- [isIndexed](#) (page 249)
Returns a Boolean value that indicates whether the receiver is important for searching.
- [isOptional](#) (page 250)
Returns a Boolean value that indicates whether the receiver is optional.
- [isTransient](#) (page 250)
Returns a Boolean value that indicates whether the receiver is transient.
- [name](#) (page 250)
Returns the name of the receiver.
- [userInfo](#) (page 255)
Returns the user info dictionary of the receiver.

Setting Features of a Property

- [setIndexed:](#) (page 251)
Sets the optionality flag of the receiver.
- [setName:](#) (page 252)
Sets the name of the receiver.
- [setOptional:](#) (page 252)
Sets the optionality flag of the receiver.
- [setTransient:](#) (page 253)
Sets the transient flag of the receiver.
- [setUserInfo:](#) (page 254)
Sets the user info dictionary of the receiver.

Validation

- [validationPredicates](#) (page 255)
Returns the validation predicates of the receiver.
- [validationWarnings](#) (page 256)
Returns the error strings associated with the receiver's validation predicates.
- [setValidationPredicates:withValidationWarnings:](#) (page 254)
Sets the validation predicates and warnings of the receiver.

Versioning Support

- [versionHash](#) (page 256)
Returns the version hash for the receiver.
- [versionHashModifier](#) (page 257)
Returns the version hash modifier for the receiver.
- [setVersionHashModifier:](#) (page 255)
Sets the version hash modifier for the receiver.
- [renamingIdentifier](#) (page 251)
Returns the renaming identifier for the receiver.
- [setRenamingIdentifier:](#) (page 253)
Sets the renaming identifier for the receiver.

Instance Methods

entity

Returns the entity description of the receiver.

- (NSEntityDescription *)entity

Return Value

The entity description of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

[setProperty:](#) (page 48) (NSEntityDescription)

Declared In

NSPropertyDescription.h

isIndexed

Returns a Boolean value that indicates whether the receiver is important for searching.

- (BOOL)isIndexed

Return Value

YES if the receiver is important for searching, otherwise NO.

Discussion

Object stores can optionally use this information upon store creation for operations such as defining indexes.

Availability

Available in iOS 3.0 and later.

See Also

- [setIndexed:](#) (page 251)

Declared In

NSPropertyDescription.h

isOptional

Returns a Boolean value that indicates whether the receiver is optional.

- (BOOL)isOptional

Return Value

YES if the receiver is optional, otherwise NO.

Availability

Available in iOS 3.0 and later.

See Also

- [setOptional:](#) (page 252)

Declared In

NSPropertyDescription.h

isTransient

Returns a Boolean value that indicates whether the receiver is transient.

- (BOOL)isTransient

Return Value

YES if the receiver is transient, otherwise NO.

Availability

Available in iOS 3.0 and later.

See Also

- [setTransient:](#) (page 253)

Declared In

NSPropertyDescription.h

name

Returns the name of the receiver.

- (NSString *)name

Return Value

The name of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setName:](#) (page 252)

Declared In

NSPropertyDescription.h

renamingIdentifier

Returns the renaming identifier for the receiver.

```
- (NSString *)renamingIdentifier
```

Return Value

The renaming identifier for the receiver.

Discussion

This is used to resolve naming conflicts between models. When creating an entity mapping between entities in two managed object models, a source entity property and a destination entity property that share the same identifier indicate that a property mapping should be configured to migrate from the source to the destination. If unset, the identifier will return the property's name.

Availability

Available in iOS 3.0 and later.

See Also

- [setRenamingIdentifier:](#) (page 253)

Declared In

NSPropertyDescription.h

setIndexed:

Sets the optionality flag of the receiver.

```
- (void)setIndexed:(BOOL)flag
```

Parameters

flag

A Boolean value that indicates whether whether the receiver is important for searching (YES) or not (NO).

Discussion

Object stores can optionally use this information upon store creation for operations such as defining indexes.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [isIndexed](#) (page 249)

Declared In

NSObjectPropertyDescription.h

setName:

Sets the name of the receiver.

```
- (void)setName:(NSString *)name
```

Parameters

name

The name of the receiver.

Special Considerations

A property name cannot be the same as any no-parameter method name of NSObject or NSObjectManagedObject. Since there are hundreds of methods on NSObject which may conflict with property names, you should avoid very general words (like “font,” and “color”) and words or phrases which overlap with Cocoa paradigms (such as “isEditing” and “objectSpecifier”).

This method raises an exception if the receiver’s model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [name](#) (page 250)

Declared In

NSObjectPropertyDescription.h

setOptional:

Sets the optionality flag of the receiver.

```
- (void)setOptional:(BOOL)flag
```

Parameters

flag

A Boolean value that indicates whether whether the receiver is optional (YES) or not (NO).

Discussion

The optionality flag specifies whether a property’s value can be nil before an object can be saved to a persistent store.

Special Considerations

This method raises an exception if the receiver’s model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [isOptional](#) (page 250)

Declared In

NSPropertyDescription.h

setRenamingIdentifier:

Sets the renaming identifier for the receiver.

```
- (void)setRenamingIdentifier:(NSString *)value
```

Parameters

value

The renaming identifier for the receiver.

Discussion

See [renamingIdentifier](#) (page 251) for a full discussion.

Availability

Available in iOS 3.0 and later.

See Also

- [renamingIdentifier](#) (page 251)

Declared In

NSPropertyDescription.h

setTransient:

Sets the transient flag of the receiver.

```
- (void)setTransient:(BOOL)flag
```

Parameters

flag

A Boolean value that indicates whether whether the receiver is transient (YES) or not (NO).

Discussion

The transient flag specifies whether or not a property's value is ignored when an object is saved to a persistent store. Transient properties are not saved to the persistent store, but are still managed for undo, redo, validation, and so on.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [isTransient](#) (page 250)

Declared In

NSPropertyDescription.h

setUserInfo:

Sets the user info dictionary of the receiver.

```
- (void)setUserInfo:(NSDictionary *)dictionary
```

Parameters

dictionary

The user info dictionary of the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [userInfo](#) (page 255)

Declared In

NSPropertyDescription.h

setValidationPredicates:withValidationWarnings:

Sets the validation predicates and warnings of the receiver.

```
- (void)setValidationPredicates:(NSArray *)validationPredicates  
withValidationWarnings:(NSArray *)validationWarnings
```

Parameters

validationPredicates

An array containing the validation predicates for the receiver.

validationWarnings

An array containing the validation warnings for the receiver.

Discussion

The *validationPredicates* and *validationWarnings* arrays should contain the same number of elements, and corresponding elements should appear at the same index in each array.

Instead of implementing individual validation methods, you can use this method to provide a list of predicates that are evaluated against the managed objects and a list of corresponding error messages (which can be localized).

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [validationPredicates](#) (page 255)

- [validationWarnings](#) (page 256)

Declared In

NSPropertyDescription.h

setVersionHashModifier:

Sets the version hash modifier for the receiver.

```
- (void)setVersionHashModifier:(NSString *)modifierString
```

Parameters

modifierString

The version hash modifier for the receiver.

Discussion

See [versionHashModifier](#) (page 257) for a full discussion.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 256)
- [versionHashModifier](#) (page 257)

Declared In

NSPropertyDescription.h

userInfo

Returns the user info dictionary of the receiver.

```
- (NSDictionary *)userInfo
```

Return Value

The user info dictionary of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setUserInfo:](#) (page 254)

Declared In

NSPropertyDescription.h

validationPredicates

Returns the validation predicates of the receiver.

```
- (NSArray *)validationPredicates
```

Return Value

An array containing the receiver's validation predicates.

Availability

Available in iOS 3.0 and later.

See Also

- [validationWarnings](#) (page 256)
- [setValidationPredicates:withValidationWarnings:](#) (page 254)

Declared In

NSPropertyDescription.h

validationWarnings

Returns the error strings associated with the receiver's validation predicates.

- (NSArray *)validationWarnings

Return Value

An array containing the error strings associated with the receiver's validation predicates.

Availability

Available in iOS 3.0 and later.

See Also

- [validationPredicates](#) (page 255)
- [setValidationPredicates:withValidationWarnings:](#) (page 254)

Declared In

NSPropertyDescription.h

versionHash

Returns the version hash for the receiver.

- (NSData *)versionHash

Return Value

The version hash for the receiver.

Discussion

The version hash is used to uniquely identify a property based on its configuration. The version hash uses only values which affect the persistence of data and the user-defined [versionHashModifier](#) (page 257) value. (The values which affect persistence are the name of the property, and the flags for `isOptional`, `isTransient`, and `isReadOnly`.) This value is stored as part of the version information in the metadata for stores, as well as a definition of a property involved in an `NSPropertyMapping` object.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHashModifier](#) (page 257)
- [setVersionHashModifier:](#) (page 255)

Declared In

NSObjectPropertyDescription.h

versionHashModifier

Returns the version hash modifier for the receiver.

- (NSString *)versionHashModifier

Return Value

The version hash modifier for the receiver.

Discussion

This value is included in the version hash for the property. You use it to mark or denote a property as being a different “version” than another even if all of the values which affect persistence are equal. (Such a difference is important in cases where the attributes of a property are unchanged but the format or content of its data are changed.)

This value is included in the version hash for the property.

Availability

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 256)
- [setVersionHashModifier:](#) (page 255)

Declared In

NSObjectPropertyDescription.h

NSPropertyMapping Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSPropertyMapping.h
Companion guide	Core Data Model Versioning and Data Migration Programming Guide

Overview

Instances of `NSPropertyMapping` specify in a mapping model how to map from a property in a source entity to a property in a destination entity.

Tasks

Managing Mapping Attributes

- `name` (page 260)
Returns the name of the property in the destination entity for the receiver.
- `setName:` (page 260)
Sets the name of the property in the destination entity for the receiver.
- `valueExpression` (page 261)
Returns the value expression for the receiver.
- `setValueExpression:` (page 261)
Sets the value expression for the receiver.
- `userInfo` (page 261)
Returns the user info for the receiver.
- `setUserInfo:` (page 260)
Sets the user info for the receiver.

Instance Methods

name

Returns the name of the property in the destination entity for the receiver.

- (NSString *)name

Return Value

The name of the property in the destination entity for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setName:](#) (page 260)

Declared In

NSPropertyMapping.h

setName:

Sets the name of the property in the destination entity for the receiver.

- (void)setName:(NSString *)name

Parameters

name

The name of the property in the destination entity for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [name](#) (page 260)

Declared In

NSPropertyMapping.h

setUserInfo:

Sets the user info for the receiver.

- (void)setUserInfo:(NSDictionary *)userInfo

Parameters

userInfo

The user info for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [userInfo](#) (page 261)

Declared In

NSPropertyMapping.h

setValueExpression:

Sets the value expression for the receiver.

```
- (void)setValueExpression:(NSEExpression *)expression
```

Parameters

expression

The the value expression for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setValueExpression:](#) (page 261)

Declared In

NSPropertyMapping.h

userInfo

Returns the user info for the receiver.

```
- (NSDictionary *)userInfo
```

Return Value

The user info for the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [setUserInfo:](#) (page 260)

Declared In

NSPropertyMapping.h

valueExpression

Returns the value expression for the receiver.

```
- (NSEExpression *)valueExpression
```

Return Value

The value expression for the receiver.

Discussion

The expression is used to create the value for the destination property.

Availability

Available in iOS 3.0 and later.

See Also

- [setValueExpression:](#) (page 261)

Declared In

NSPropertyMapping.h

NSRelationshipDescription Class Reference

Inherits from	NSObject (NSObject)
Conforms to	NSCoding (NSObject) NSCopying (NSObject) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	NSRelationshipDescription.h
Companion guide	Core Data Programming Guide

Overview

The `NSRelationshipDescription` class is used to describe relationships of an entity in an `NSEntityDescription` object.

`NSRelationshipDescription` extends `NSObject` to describe features appropriate to relationships, including cardinality (the number of objects allowed in the relationship), the destination entity, and delete rules.

Cardinality

The maximum and minimum counts for a relationship indicate the number of objects referenced (1 for a to-one relationship, -1 means undefined). The counts are only enforced if the relationship value in the containing object is not `nil`. That is, provided that the relationship value is optional, there may be zero objects in the relationship, which might be less than the minimum count.

Editing Relationship Descriptions

Relationship descriptions are editable until they are used by an object graph manager. This allows you to create or modify them dynamically. However, once a description is used (when the managed object model to which it belongs is associated with a persistent store coordinator), it *must not* (indeed cannot) be changed. This is enforced at runtime: any attempt to mutate a model or any of its sub-objects after the model is associated with a persistent store coordinator causes an exception to be thrown. If you need to modify a model that is in use, create a copy, modify the copy, and then discard the objects with the old model.

Tasks

Managing Type Information

- [destinationEntity](#) (page 265)
Returns the entity description of the receiver's destination.
- [setDestinationEntity:](#) (page 267)
Sets the entity description for the receiver's destination.
- [inverseRelationship](#) (page 265)
Returns the relationship that represents the inverse of the receiver.
- [setInverseRelationship:](#) (page 268)
Sets the inverse relationship of the receiver.

Getting and Setting Delete Rules

- [deleteRule](#) (page 265)
Returns the delete rule of the receiver.
- [setDeleteRule:](#) (page 267)
Sets the delete rule of the receiver.

Cardinality

- [maxCount](#) (page 266)
Returns the maximum count of the receiver.
- [setMaxCount:](#) (page 268)
Sets the maximum count of the receiver.
- [minCount](#) (page 267)
Returns the minimum count of the receiver.
- [setMinCount:](#) (page 269)
Sets the minimum count of the receiver.
- [isToMany](#) (page 266)
Returns a Boolean value that indicates whether the receiver represents a to-many relationship.

Versioning Support

- [versionHash](#) (page 269)
Returns the version hash for the receiver.

Instance Methods

deleteRule

Returns the delete rule of the receiver.

- (NSDeleteRule)deleteRule

Return Value

The receiver's delete rule.

Availability

Available in iOS 3.0 and later.

See Also

- [setDeleteRule:](#) (page 267)

Declared In

NSRelationshipDescription.h

destinationEntity

Returns the entity description of the receiver's destination.

- (NSEntityDescription *)destinationEntity

Return Value

The entity description for the receiver's destination.

Availability

Available in iOS 3.0 and later.

See Also

- [setDestinationEntity:](#) (page 267)

Declared In

NSRelationshipDescription.h

inverseRelationship

Returns the relationship that represents the inverse of the receiver.

- (NSRelationshipDescription *)inverseRelationship

Return Value

The relationship that represents the inverse of the receiver.

Discussion

Given a to-many relationship “employees” between a Department entity and an Employee entity (a department may have many employees), and a to-one relationship “department” between an Employee entity and a Department entity (an employee may belong to only one department), the inverse of the “department” relationship is the “employees” relationship.

Availability

Available in iOS 3.0 and later.

See Also

- [setInverseRelationship:](#) (page 268)

Declared In

NSRelationshipDescription.h

isToMany

Returns a Boolean value that indicates whether the receiver represents a to-many relationship.

- (BOOL)isToMany

Return Value

YES if the receiver represents a to-many relationship (its `maxCount` is greater than 1) otherwise NO.

Availability

Available in iOS 3.0 and later.

See Also

- [maxCount](#) (page 266)

- [setMaxCount:](#) (page 268)

Declared In

NSRelationshipDescription.h

maxCount

Returns the maximum count of the receiver.

- (NSUInteger)maxCount

Return Value

The maximum count of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [isToMany](#) (page 266)

- [minCount](#) (page 267)

- [setMaxCount:](#) (page 268)

- [setMinCount:](#) (page 269)

Declared In

NSRelationshipDescription.h

minCount

Returns the minimum count of the receiver.

```
- (NSUInteger)minCount
```

Return Value

The minimum count of the receiver.

Availability

Available in iOS 3.0 and later.

See Also

- [maxCount](#) (page 266)
- [setMaxCount:](#) (page 268)
- [setMinCount:](#) (page 269)

Declared In

NSRelationshipDescription.h

setDeleteRule:

Sets the delete rule of the receiver.

```
- (void)setDeleteRule:(NSDeleteRule)rule
```

Parameters*rule*

The delete rule for the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [deleteRule](#) (page 265)

Declared In

NSRelationshipDescription.h

setDestinationEntity:

Sets the entity description for the receiver's destination.

```
- (void)setDestinationEntity:(NSEntityDescription *)entity
```

Parameters*entity*

The destination entity for the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [destinationEntity](#) (page 265)

Declared In

NSRelationshipDescription.h

setInverseRelationship:

Sets the inverse relationship of the receiver.

```
- (void)setInverseRelationship:(NSRelationshipDescription *)relationship
```

Parameters*relationship*

The inverse relationship for the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [inverseRelationship](#) (page 265)

Declared In

NSRelationshipDescription.h

setMaxCount:

Sets the maximum count of the receiver.

```
- (void)setMaxCount:(NSUInteger)maxCount
```

Parameters*maxCount*

The maximum count of the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [isToMany](#) (page 266)
- [maxCount](#) (page 266)
- [minCount](#) (page 267)
- [setMinCount:](#) (page 269)

Declared In

NSRelationshipDescription.h

setMinCount:

Sets the minimum count of the receiver.

```
- (void)setMinCount:(NSUInteger)minCount
```

Parameters*minCount*

The minimum count of the receiver.

Special Considerations

This method raises an exception if the receiver's model has been used by an object graph manager.

Availability

Available in iOS 3.0 and later.

See Also

- [maxCount](#) (page 266)
- [minCount](#) (page 267)
- [setMaxCount:](#) (page 268)

Declared In

NSRelationshipDescription.h

versionHash

Returns the version hash for the receiver.

```
- (NSData *)versionHash
```

Return Value

The version hash for the receiver.

DiscussionThe version hash is used to uniquely identify an attribute based on its configuration. This value includes the [versionHash](#) (page 256) information from [NSPropertyDescription](#), the name of the destination entity and the inverse relationship, and the min and max count.**Availability**

Available in iOS 3.0 and later.

See Also

- [versionHash](#) (page 256) ([NSPropertyDescription](#))

Declared In

NSRelationshipDescription.h

Constants

NSDeleteRule

These constants define what happens to relationships when an object is deleted.

```
typedef enum {
    NSNoActionDeleteRule,
    NSNullifyDeleteRule,
    NSCascadeDeleteRule,
    NSDenyDeleteRule
} NSDeleteRule;
```

Constants

NSNoActionDeleteRule

If the object is deleted, no modifications are made to objects at the destination of the relationship.

If you use this rule, you are responsible for maintaining the integrity of the object graph. This rule is strongly discouraged for all but advanced users. You should normally use `NSNullifyDeleteRule` instead.

Available in iOS 3.0 and later.

Declared in `NSRelationshipDescription.h`.

NSNullifyDeleteRule

If the object is deleted, back pointers from the objects to which it is related are nullified.

Available in iOS 3.0 and later.

Declared in `NSRelationshipDescription.h`.

NSCascadeDeleteRule

If the object is deleted, the destination object or objects of this relationship are also deleted.

Available in iOS 3.0 and later.

Declared in `NSRelationshipDescription.h`.

NSDenyDeleteRule

If the destination of this relationship is not `nil`, the delete creates a validation error.

Available in iOS 3.0 and later.

Declared in `NSRelationshipDescription.h`.

Availability

Available in iOS 3.0 and later.

Declared In

NSRelationshipDescription.h

Protocols

NSFetchedResultsControllerDelegate Protocol Reference

Framework	/System/Library/Frameworks/CoreData.framework
Declared in	NSFetchedResultsController.h
Companion guide	Core Data Programming Guide

Overview

An instance of `NSFetchedResultsController` uses methods in this protocol to notify its delegate that the controller's fetch results have been changed due to an add, remove, move, or update operations.

You should consider carefully whether you want to update the table view as each change is made. If a large number of modifications are made simultaneously—for example, if you are reading data from a background thread—it may be computationally expensive to animate all the changes. Rather than responding to changes individually (as illustrated in “Typical Use”), you could just implement `controllerDidChangeContent:` (page 278) (which is sent to the delegate when all pending changes have been processed) to reload the table view.

The fetched results controller reports changes to its section before changes to the fetched objects themselves.

Typical Use

You can use `controllerWillChangeContent:` (page 278) and `controllerDidChangeContent:` (page 278) to bracket updates to a table view whose content is provided by the fetched results controller as illustrated in the following example:

```

/*
 Assume self has a property 'tableView' -- as is the case for an instance of a
 UITableViewController
 subclass -- and a method configureCell:atIndexPath: which updates the contents
 of a given cell
 with information from a managed object at the given index path in the fetched
 results controller.
 */

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

- (void)controller:(NSFetchedResultsController *)controller didChangeSection:(id
<NSFetchedResultsSectionInfo>)sectionInfo
    atIndex:(NSUInteger)sectionIndex
    forChangeType:(NSFetchedResultsChangeType)type {

```

```

    switch(type) {
        case NSFetchedResultsControllerChangeInsert:
            [self.tableView insertSections:[NSIndexSet
indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsControllerChangeDelete:
            [self.tableView deleteSections:[NSIndexSet
indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

- (void)controller:(NSFetchedResultsController *)controller
didChangeObject:(id)anObject
    atIndexPath:(NSIndexPath *)indexPath
forChangeType:(NSFetchedResultsControllerChangeType)type
    newIndexPath:(NSIndexPath *)newIndexPath {

    UITableView *tableView = self.tableView;

    switch(type) {

        case NSFetchedResultsControllerChangeInsert:
            [tableView insertRowsAtIndexPaths:[NSArray
arrayWithObject:newIndexPath]
                withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsControllerChangeDelete:
            [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsControllerChangeUpdate:
            [self configureCell:[tableView cell]ForRowAtIndexPath:indexPath]
                atIndexPath:indexPath];
            break;

        case NSFetchedResultsControllerChangeMove:
            [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                withRowAnimation:UITableViewRowAnimationFade];
            [tableView insertRowsAtIndexPaths:[NSArray
arrayWithObject:newIndexPath]
                withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

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

```

User-Driven Updates

In general, `NSFetchedResultsController` is designed to respond to changes at the model layer. If you allow a user to reorder table rows, then your implementation of the delegate methods must take this into account.

Typically, if you allow the user to reorder table rows, your model object has an attribute that specifies its index. When the user moves a row, you update this attribute accordingly. This, however, has the side effect of causing the controller to notice the change, and so inform its delegate of the update (using `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`). If you simply use the implementation of this method shown in “Typical Use,” then the delegate attempts to update the table view. The table view, however, is already in the appropriate state because of the user’s action.

In general, therefore, if you support user-driven updates, you should set a flag if a move is initiated by the user. In the implementation of your delegate methods, if the flag is set, you bypass main method implementations; for example:

```
- (void)controller:(NSFetchedResultsController *)controller
didChangeObject:(id)anObject
  atIndexPath:(NSIndexPath *)indexPath
forChangeType:(NSFetchedResultsControllerChangeType)type
  newIndexPath:(NSIndexPath *)newIndexPath {

    if (!changeIsUserDriven) {
        UITableView *tableView = self.tableView;
        // Implementation continues...
```

Note: Prior to iOS 4.0, `NSFetchedResultsController` did not support sections being deleted as a result of a UI-driven change.

Tasks

Responding to Changes

- [controllerWillChangeContent:](#) (page 278)
Notifies the receiver that the fetched results controller is about to start processing of one or more changes due to an add, remove, move, or update.
- [controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:](#) (page 276)
Notifies the receiver that a fetched object has been changed due to an add, remove, move, or update.
- [controller:didChangeSection:atIndex:forChangeType:](#) (page 277)
Notifies the receiver of the addition or removal of a section.
- [controllerDidChangeContent:](#) (page 278)
Notifies the receiver that the fetched results controller has completed processing of one or more changes due to an add, remove, move, or update.

Customizing Section Names

- [controller:sectionIndexTitleForSectionName:](#) (page 277)

Returns the name for a given section.

Instance Methods

controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:

Notifies the receiver that a fetched object has been changed due to an add, remove, move, or update.

```
- (void)controller:(NSFetchedResultsController *)controller
  didChangeObject:(id)anObject
  atIndexPath:(NSIndexPath *)indexPath
  forChangeType:(NSFetchedResultsControllerChangeType)type
  newIndexPath:(NSIndexPath *)newIndexPath
```

Parameters

controller

The fetched results controller that sent the message.

anObject

The object in controller’s fetched results that changed.

indexPath

The index path of the changed object (this value is `nil` for insertions).

type

The type of change. For valid values see “[NSFetchedResultsControllerChangeType](#)” (page 278).

newIndexPath

The destination path for the object for insertions or moves (this value is `nil` for a deletion).

Discussion

The fetched results controller reports changes to its section before changes to the fetch result objects.

Changes are reported with the following heuristics:

- On add and remove operations, only the added/removed object is reported.

It’s assumed that all objects that come after the affected object are also moved, but these moves are not reported.
- A move is reported when the changed attribute on the object is one of the sort descriptors used in the fetch request.

An update of the object is assumed in this case, but no separate update message is sent to the delegate.
- An update is reported when an object’s state changes, but the changed attributes aren’t part of the sort keys.

Special Considerations

This method may be invoked many times during an update event (for example, if you are importing data on a background thread and adding them to the context in a batch). You should consider carefully whether you want to update the table view on receipt of each message.

controller:didChangeSection:atIndex:forChangeType:

Notifies the receiver of the addition or removal of a section.

```
- (void)controller:(NSFetchedResultsController *)controller
  didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
    atIndex:(NSUInteger)sectionIndex
    forChangeType:(NSFetchedResultsChangeType)type
```

Parameters

controller

The fetched results controller that sent the message.

sectionInfo

The section that changed.

sectionIndex

The index of the changed section.

type

The type of change (insert or delete). Valid values are [NSFetchedResultsChangeInsert](#) (page 279) and [NSFetchedResultsChangeDelete](#) (page 279).

Discussion

The fetched results controller reports changes to its section before changes to the fetched result objects.

Special Considerations

This method may be invoked many times during an update event (for example, if you are importing data on a background thread and adding them to the context in a batch). You should consider carefully whether you want to update the table view on receipt of each message.

controller:sectionIndexTitleForSectionName:

Returns the name for a given section.

```
- (NSString *)controller:(NSFetchedResultsController *)controller
  sectionIndexTitleForSectionName:(NSString *)sectionName
```

Parameters

controller

The fetched results controller that sent the message.

sectionName

The default name of the section.

Return Value

The string to use as the name for the specified section.

Discussion

This method does not enable change tracking. It is only needed if a section index is used.

If the delegate doesn't implement this method, the default implementation returns the capitalized first letter of the section name (see [sectionIndexTitleForSectionName:](#) (page 97) in [NSFetchedResultsController](#)).

controllerDidChangeContent:

Notifies the receiver that the fetched results controller has completed processing of one or more changes due to an add, remove, move, or update.

```
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
```

Parameters

controller

The fetched results controller that sent the message.

Discussion

This method is invoked after all invocations of

[controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:](#) (page 276) and [controller:didChangeSection:atIndex:forChangeType:](#) (page 277) have been sent for a given change event (such as the controller receiving a [NSManagedObjectContextDidSaveNotification](#) (page 179) notification).

controllerWillChangeContent:

Notifies the receiver that the fetched results controller is about to start processing of one or more changes due to an add, remove, move, or update.

```
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller
```

Parameters

controller

The fetched results controller that sent the message.

Discussion

This method is invoked before all invocations of

[controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:](#) (page 276) and [controller:didChangeSection:atIndex:forChangeType:](#) (page 277) have been sent for a given change event (such as the controller receiving a [NSManagedObjectContextDidSaveNotification](#) (page 179) notification).

Constants

NSFetchedResultsControllerChangeType

Specify types of change.

```
enum {
    NSFetchedResultsControllerChangeInsert = 1,
    NSFetchedResultsControllerChangeDelete = 2,
    NSFetchedResultsControllerChangeMove = 3,
    NSFetchedResultsControllerChangeUpdate = 4
};
typedef NSUInteger NSFetchedResultsControllerChangeType;
```

Constants

`NSFetchedResultsControllerChangeInsert`
Specifies that an object was inserted.

`NSFetchedResultsControllerChangeDelete`
Specifies that an object was deleted.

`NSFetchedResultsControllerChangeMove`
Specifies that an object was moved.

`NSFetchedResultsControllerChangeUpdate`
Specifies that an object was changed.

NSFetchedResultsController Protocol Reference

Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in iOS 3.0 and later.
Declared in	CoreData/NSFetchedResultsController.h
Companion guide	Core Data Programming Guide

Overview

This protocol defines the interface for section objects vended by an instance of `NSFetchedResultsController`.

Tasks

Accessing Objects

- `numberOfObjects` (page 282) *required property*
The number of objects (rows) in the section. (required)
- `objects` (page 282) *required property*
The array of objects in the section. (required)

Accessing the Name and Title

- `name` (page 282) *required property*
The name of the section. (required)
- `indexTitle` (page 282) *required property*
The index title of the section. (required)

Properties

For more about Objective-C properties, see “Properties” in *The Objective-C Programming Language*.

indexTitle

The index title of the section. (required)

```
@property (nonatomic, readonly) NSString *indexTitle
```

Discussion

This is used when displaying the index.

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

name

The name of the section. (required)

```
@property (nonatomic, readonly) NSString *name
```

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

numberOfObjects

The number of objects (rows) in the section. (required)

```
@property (nonatomic, readonly) NSUInteger numberOfObjects
```

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

objects

The array of objects in the section. (required)

```
@property (nonatomic, readonly) NSArray *objects
```

Availability

Available in iOS 3.0 and later.

Declared In

NSFetchedResultsController.h

Constants

Core Data Constants Reference

Framework: CoreData/CoreData.h

Overview

This document describes the constants defined in the Core Data framework and not described in a document for an individual class.

Constants

Error User Info Keys

Keys in the user info dictionary in errors Core Data creates.

```
const NSString *NSDetailedErrorsKey;
const NSString *NSValidationObjectErrorKey;
const NSString *NSValidationKeyErrorKey;
const NSString *NSValidationPredicateErrorKey;
const NSString *NSValidationValueErrorKey;
const NSString *NSAffectedStoresErrorKey;
const NSString *NSAffectedObjectsErrorKey;
```

Constants

`NSDetailedErrorsKey`

If multiple validation errors occur in one operation, they are collected in an array and added with this key to the “top-level error” of the operation.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationObjectErrorKey`

Key for the object that failed to validate for a validation error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationKeyErrorKey`

Key for the key that failed to validate for a validation error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationPredicateErrorKey`

For predicate-based validation, key for the predicate for the condition that failed to validate.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationValueErrorKey`

If non-nil, the key for the value for the key that failed to validate for a validation error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSAffectedStoresErrorKey`

The key for stores prompting an error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSAffectedObjectsErrorKey`

The key for objects prompting an error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Error Domain

Constant to identify the SQLite error domain.

```
const NSString *NSSQLiteErrorDomain;
```

Constants

`NSSQLiteErrorDomain`

Domain for SQLite errors.

The value of "code" corresponds to preexisting values in SQLite.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Validation Error Codes

Error codes related to validation.

<code>NSManagedObjectValidationError</code>	= 1550,
<code>NSValidationMultipleErrorsError</code>	= 1560,
<code>NSValidationMissingMandatoryPropertyError</code>	= 1570,
<code>NSValidationRelationshipLacksMinimumCountError</code>	= 1580,
<code>NSValidationRelationshipExceedsMaximumCountError</code>	= 1590,
<code>NSValidationRelationshipDeniedDeleteError</code>	= 1600,
<code>NSValidationNumberTooLargeError</code>	= 1610,
<code>NSValidationNumberTooSmallError</code>	= 1620,
<code>NSValidationDateTooLateError</code>	= 1630,
<code>NSValidationDateTooSoonError</code>	= 1640,
<code>NSValidationInvalidDateError</code>	= 1650,
<code>NSValidationStringTooLongError</code>	= 1660,
<code>NSValidationStringTooShortError</code>	= 1670,
<code>NSValidationStringPatternMatchingError</code>	= 1680,

Constants

`NSManagedObjectValidationError`

Error code to denote a generic validation error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationMultipleErrorsError`

Error code to denote an error containing multiple validation errors.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationMissingMandatoryPropertyError`

Error code for a non-optional property with a nil value.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipLacksMinimumCountError`

Error code to denote a to-many relationship with too few destination objects.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipExceedsMaximumCountError`

Error code to denote a bounded to-many relationship with too many destination objects.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipDeniedDeleteError`

Error code to denote some relationship with delete rule `NSDeleteRuleDeny` is non-empty.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationNumberTooLargeError`

Error code to denote some numerical value is too large.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationNumberTooSmallError`

Error code to denote some numerical value is too small.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationDateTooLateError`

Error code to denote some date value is too late.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationDateTooSoonError`

Error code to denote some date value is too soon.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationInvalidDateError`

Error code to denote some date value fails to match date pattern.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringTooLongError`

Error code to denote some string value is too long.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringTooShortError`

Error code to denote some string value is too short.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringPatternMatchingError`

Error code to denote some string value fails to match some pattern.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Discussion

For additional error codes, including `NSValidationErrorMinimum` and `NSValidationErrorMaximum`, see `NSError`.

Object Graph Management Error Codes

These error codes specify Core Data errors related to object graph management.

<code>NSManagedObjectContextLockingError</code>	= 132000,
<code>NSPersistentStoreCoordinatorLockingError</code>	= 132010,
<code>NSManagedObjectContextReferentialIntegrityError</code>	= 133000,
<code>NSManagedObjectContextExternalRelationshipError</code>	= 133010,
<code>NSManagedObjectContextMergeError</code>	= 133020,

Constants

`NSManagedObjectContextLockingError`

Error code to denote an inability to acquire a lock in a managed object context.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreCoordinatorLockingError`

Error code to denote an inability to acquire a lock in a persistent store.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectContextReferentialIntegrityError`

Error code to denote an attempt to fire a fault pointing to an object that does not exist.

The store is accessible, but the object corresponding to the fault cannot be found.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectContextExternalRelationshipError`

Error code to denote that an object being saved has a relationship containing an object from another store.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectContextMergeError`

Error code to denote that a merge policy failed—Core Data is unable to complete merging.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Persistent Store Error Codes

Error codes related to persistent stores.

<code>NSPersistentStoreInvalidTypeError</code>	= 134000,
<code>NSPersistentStoreTypeMismatchError</code>	= 134010,
<code>NSPersistentStoreIncompatibleSchemaError</code>	= 134020,
<code>NSPersistentStoreSaveError</code>	= 134030,
<code>NSPersistentStoreIncompleteSaveError</code>	= 134040,
<code>NSPersistentStoreOperationError</code>	= 134070,
<code>NSPersistentStoreOpenError</code>	= 134080,
<code>NSPersistentStoreTimeoutError</code>	= 134090,
<code>NSPersistentStoreIncompatibleVersionHashError</code>	= 134100,

Constants

`NSPersistentStoreInvalidTypeError`

Error code to denote an unknown persistent store type/format/version.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreTypeMismatchError`

Error code returned by a persistent store coordinator if a store is accessed that does not match the specified type.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreIncompatibleSchemaError`

Error code to denote that a persistent store returned an error for a save operation.

This code pertains to database level errors such as a missing table.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreSaveError`

Error code to denote that a persistent store returned an error for a save operation.

This code pertains to errors such as permissions problems.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreIncompleteSaveError`

Error code to denote that one or more of the stores returned an error during a save operations.

The stores or objects that failed are in the corresponding user info dictionary of the `NSError` object.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreOperationError`

Error code to denote that a persistent store operation failed.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreOpenError`

Error code to denote an error occurred while attempting to open a persistent store.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreTimeoutError`

Error code to denote that Core Data failed to connect to a persistent store within the time specified by `NSPersistentStoreTimeoutOption`.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreIncompatibleVersionHashError`

Error code to denote that entity version hashes in the store are incompatible with the current managed object model.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Migration Error Codes

Error codes related to store migration.

<code>NSMigrationError</code>	= 134110,
<code>NSMigrationCancelledError</code>	= 134120,
<code>NSMigrationMissingSourceModelError</code>	= 134130,
<code>NSMigrationMissingMappingModelError</code>	= 134140,
<code>NSMigrationManagerSourceStoreError</code>	= 134150,
<code>NSMigrationManagerDestinationStoreError</code>	= 134160,
<code>NSEntityMigrationPolicyError</code>	= 134170,
<code>NSInferredMappingModelError</code>	= 134190,
<code>NSExternalRecordImportError</code>	= 134200,

Constants

`NSMigrationError`

Error code to denote a general migration error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationCancelledError`

Error code to denote that migration failed due to manual cancellation.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationMissingSourceModelError`

Error code to denote that migration failed due to a missing source data model.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationMissingMappingModelError`

Error code to denote that migration failed due to a missing mapping model.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationManagerSourceStoreError`

Error code to denote that migration failed due to a problem with the source data store.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationManagerDestinationStoreError`

Error code to denote that migration failed due to a problem with the destination data store.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSEntityMigrationPolicyError`

Error code to denote that migration failed during processing of an entity migration policy.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSInferredMappingModelError`

Error code to denote a problem with the creation of an inferred mapping model.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSExternalRecordImportError`

Error code to denote a general error encountered while importing external records.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

General Error Codes

Error codes that denote a general error.

`NSCoreDataError`

= 134060,

`NSSQLiteError`

= 134180,

Constants

`NSCoreDataError`

Error code to denote a general Core Data error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

`NSSQLiteError`

Error code to denote a general SQLite error.

Available in iOS 3.0 and later.

Declared in `CoreDataErrors.h`.

Core Data Version Number

Specifies the current Core Data version number.

```
double NSCoreDataVersionNumber;
```

Constants

`NSCoreDataVersionNumber`

Specifies the version of Core Data available in the current process.

Available in iOS 3.0 and later.

Declared in `CoreDataDefines.h`.

Discussion

See “[Core Data Version Numbers](#)” (page 293) for defined versions.

Core Data Version Numbers

Specify Core Data version numbers.

```
#define NSCoreDataVersionNumber10_4          46.0
#define NSCoreDataVersionNumber10_4_3       77.0
#define NSCoreDataVersionNumber10_5         185.0
#define NSCoreDataVersionNumber10_5_3      186.0
#define NSCoreDataVersionNumber10_6        246.0
#define NSCoreDataVersionNumber10_6_2      250.0
#define NSCoreDataVersionNumber10_6_3      251.0
#define NSCoreDataVersionNumber_iOS_3_0     241.0
#define NSCoreDataVersionNumber_iOS_3_1     248.0
#define NSCoreDataVersionNumber_iOS_3_2     310.2
```

Constants

`NSCoreDataVersionNumber10_4`

Specifies the Core Data version number released with Mac OS X v10.4.0.

Available in iOS 3.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_4_3`

Specifies the Core Data version number released with Mac OS X v10.4.3.

Available in iOS 3.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_5`

Specifies the Core Data version number released with Mac OS X v10.5.0.

Available in iOS 3.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_5_3`

Specifies the Core Data version number released with Mac OS X v10.5.3.

Available in iOS 3.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_6`

Specifies the Core Data version number released with Mac OS X v10.6.0.

Available in iOS 4.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_6_2`

Specifies the Core Data version number released with Mac OS X v10.6.2.

Available in iOS 4.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber10_6_3`

Specifies the Core Data version number released with Mac OS X v10.6.3.

Available in iOS 4.0 and later.

Declared in `CoreDataDefines.h`.

`NSCoreDataVersionNumber_iOS_3_0`

Specifies the Core Data version number released with iOS v3.0.

`NSCoreDataVersionNumber_iOS_3_1`

Specifies the Core Data version number released with iOS v3.1.

`NSCoreDataVersionNumber_iOS_3_2`

Specifies the Core Data version number released with iOS v3.2.

Discussion

See [“Core Data Version Number”](#) (page 292) for the current version.

Document Revision History

This table describes the changes to *Core Data Framework Reference*.

Date	Notes
2009-03-10	First release for iOS.
2009-02-07	Updated for Mac OS X v10.6.
2007-07-24	Updated for Mac OS X v10.5.
2006-05-23	First publication of this content as a collection of separate documents.

REVISION HISTORY

Document Revision History