Core Data Framework Reference

Cocoa > Objective-C Language

2007-07-24

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

Simultaneously published in the United States and Canada.

Contents

Introduction **Introduction 9**

Part I **Classes 11**

Chapter 1 **NSAtomicStore Class Reference 13**

Overview 13
Tasks 14
Instance Methods 15

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 35

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 NSFetchedPropertyDescription Class Reference 77

Overview 77
Tasks 78
Instance Methods 78

Chapter 8 NSFetchRequest Class Reference 81

Overview 81
Tasks 82
Instance Methods 83
Constants 93

Chapter 9 NSFetchRequestExpression Class Reference 95

Overview 95
Tasks 95
Class Methods 96
Instance Methods 96
Constants 97

Chapter 10 NSManagedObject Class Reference 99

Overview 99
Tasks 103
Class Methods 105
Instance Methods 105
Constants 124

Chapter 11 NSManagedObjectContext Class Reference 125

Overview 125
Tasks 126
Instance Methods 129
Constants 152
Notifications 155

Chapter 12 NSManagedObjectID Class Reference 157

Overview 157
Tasks 157
Instance Methods 158

Chapter 13 NSManagedObjectModel Class Reference 161

Overview 161
Tasks 163
Class Methods 164
Instance Methods 167

Chapter 14 NSMappingModel Class Reference 177

Overview 177
Tasks 177
Class Methods 178
Instance Methods 178

Chapter 15 NSMigrationManager Class Reference 181

Overview 181
Tasks 181
Instance Methods 182

Chapter 16 NSPersistentStore Class Reference 193

Overview 193
Tasks 194
Class Methods 195
Instance Methods 196

Chapter 17 NSPersistentStoreCoordinator Class Reference 203

Overview 203
Tasks 204
Class Methods 205
Instance Methods 209
Constants 217
Notifications 221

Chapter 18 NSPropertyDescription Class Reference 223

Overview 223
Tasks 224
Instance Methods 225

Chapter 19 NSPropertyMapping Class Reference 233

Overview 233
Tasks 233

Instance Methods 234

Chapter 20 NSRelationshipDescription Class Reference 237

Overview 237
Tasks 238
Instance Methods 239
Constants 244

Part II Constants 245

Chapter 21 Core Data Constants Reference 247

Overview 247
Constants 247

Document Revision History 257

Index 259

Tables

Chapter 13 **NSManagedObjectModel Class Reference 161**

Table 13-1 Key and value pattern for the localization dictionary. 174

Introduction

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 NSFetchRequest.h NSFetchRequestExpression.h NSFetchedPropertyDescription.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

Classes

PART I

Classes

NSAtomicStore Class Reference

Inherits from	NSPersistentStore : NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Declared in	NSAtomicStore.h
Availability	Available in Mac OS X v10.5 and later.
Companion guides	Atomic Store Programming Topics Core Data Programming Guide
Related sample code	Core Data HTML Store CustomAtomicStoreSubclass

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 17)	Loads the cache nodes for the receiver.
`newReferenceObjectForManagedObject:` (page 19)	Returns a new reference object for a given managed object.
`save:` (page 21)	Saves the cache nodes.

`updateCacheNode:fromManagedObject:` (page 22)	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 201)	Returns the type string of the receiver.
`identifier` (page 197)	Returns the unique identifier for the receiver.
`setIdentifier:` (page 199)	Sets the unique identifier for the receiver.
`metadata` (page 198)	Returns the metadata for the receiver.
`metadataForPersistentStoreWithURL:error:` (page 195)	Returns the metadata from the persistent store at the given URL.
`setMetadata:forPersistentStoreWithURL:error:` (page 195)	Sets the metadata for the store at a given URL.

Tasks

Initializing a Store

- `initWithPersistentStoreCoordinator:configurationName:URL:options:` (page 16)
  Returns an atomic store, initialized with the given arguments.

Loading a Store

- `load:` (page 17)
  Loads the cache nodes for the receiver.
- `objectIDForEntity:referenceObject:` (page 20)
  Returns a managed object ID from the reference data for a specified entity.
- `addCacheNodes:` (page 15)
  Registers a set of cache nodes with the receiver.

Updating Cache Nodes

- `newCacheNodeForManagedObject:` (page 19)
  Returns a new cache node for a given managed object.
- `newReferenceObjectForManagedObject:` (page 19)
  Returns a new reference object for a given managed object.
- `updateCacheNode:fromManagedObject:` (page 22)
  Updates the given cache node using the values in a given managed object.

- `willRemoveCacheNodes:` (page 22)
  Method invoked before the store removes the given collection of cache nodes.

Saving a Store

- `save:` (page 21)
  Saves the cache nodes.

Utility Methods

- `cacheNodes` (page 16)
  Returns the set of cache nodes registered with the receiver.
- `cacheNodeForObjectID:` (page 16)
  Returns the cache node for a given managed object ID.
- `referenceObjectForObjectID:` (page 20)
  Returns the reference object for a given managed object ID.

Managing Metadata

- `metadata` (page 18)
  Returns the metadata for the receiver.
- `setMetadata:` (page 21)
  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 17) to register the loaded information with the store.

Availability
Available in Mac OS X v10.5 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 Mac OS X v10.5 and later.

Related Sample Code

CustomAtomicStoreSubclass

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 15): and `willRemoveCacheNodes:` (page 22).

Availability

Available in Mac OS X v10.5 and later.

Related Sample Code

CustomAtomicStoreSubclass

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 209), 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 17) 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 17). 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 21).

Special Considerations

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

Availability

Available in Mac OS X v10.5 and later.

See Also

- `load:` (page 17)
- `setMetadata:` (page 21)

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 16) 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 16) 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 Mac OS X v10.5 and later.

See Also

`- addCacheNodes:` (page 15)

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 Value

A 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 Mac OS X v10.5 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 Value

A 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
`- addCacheNodes:` (page 15)

Related Sample Code
CustomAtomicStoreSubclass

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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also

- `newReferenceObjectForManagedObject:` (page 19)
- `updateCacheNode:fromManagedObject:` (page 22)
- `willRemoveCacheNodes:` (page 22)

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 18)

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 Mac OS X v10.5 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 21) 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 21) method.

You should not invoke this method directly in a subclass.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- save:` (page 21)

Declared In
`NSAtomicStore.h`

NSAtomicStoreCacheNode Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Declared in	NSAtomicStoreCacheNode.h
Availability	Available in Mac OS X v10.5 and later.
Companion guide	Core Data Programming Guide
Related sample code	Core Data HTML Store CustomAtomicStoreSubclass

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 27)
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 Mac OS X v10.5 and later.

Related Sample Code
Core Data HTML Store
CustomAtomicStoreSubclass

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 Mac OS X v10.5 and later.

Related Sample Code
CustomAtomicStoreSubclass

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: and -setValue:forKey: 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Related Sample Code
CustomAtomicStoreSubclass

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 27) 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 Mac OS X v10.5 and later.

Related Sample Code
Core Data HTML Store

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 27) 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 Mac OS X v10.5 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 Mac OS X v10.4 and later.
Declared in	NSAttributeDescription.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass

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 Mac OS X v10.4 and later.

See Also
- `attributeValueClassName` (page 31)
- `setAttributeType:` (page 32)

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.5 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 Mac OS X v10.4 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
`- setValueTransformerName:` (page 33)

Related Sample Code
CustomAtomicStoreSubclass

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 231) information from `NSPropertyDescription` and the attribute type.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- versionHash` (page 231) (`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
} 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 Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSInteger16AttributeType`

Specifies a 16-bit signed integer attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSInteger32AttributeType`

Specifies a 32-bit signed integer attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSInteger64AttributeType`

Specifies a 64-bit signed integer attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSDecimalAttributeType`

Specifies an `NSDecimalNumber` attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSDoubleAttributeType`
Specifies a double attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSFloatAttributeType`
Specifies a float attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSStringAttributeType`
Specifies an `NSString` attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSBooleanAttributeType`
Specifies a Boolean attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSDateAttributeType`
Specifies an `NSDate` attribute.

Times are specified in GMT.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSBinaryDataAttributeType`
Specifies an `NSData` attribute.

Available in Mac OS X v10.4 and later.

Declared in `NSAttributeDescription.h`.

`NSTransformableAttributeType`
Specifies an attribute that uses a value transformer.

Available in Mac OS X v10.5 and later.

Declared in `NSAttributeDescription.h`.

Availability
Available in Mac OS X v10.4 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 Mac OS X v10.4 and later.
Declared in	CoreData/NSEntityDescription.h
Companion guides	Core Data Programming Guide Core Data Utility Tutorial
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass Departments and Employees QTMetadataEditor

Overview

Instances of `NSEntityDescription` are used to describe entities in terms of their name, their properties—attributes and relationships as expressed by `NSAttributeDescription` and `NSRelationshipDescription`—and the class by which they are represented. Entities are to managed objects what `Class` is to `id`, or—to use a database analogy—what tables are to rows.

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 `NSManagedObjectModel` 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 (`NSMapTable`) with retain callbacks.

Fast Enumeration

In Mac OS X v10.5 and later, `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 45)
  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.
- `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 51)
  Returns the user info dictionary of the receiver.
- `setUserInfo:` (page 49)
  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 44)
  Returns a Boolean value that indicates whether the receiver is a subentity of another given entity.

Working with Properties

- `propertiesByName` (page 46)
  Returns a dictionary containing the properties of the receiver.
- `properties` (page 45)
  Returns an array containing the properties of the receiver.
- `setProperties:` (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 47)
  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 a new autoreleased 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 the *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 Mac OS X v10.4 and later.

See Also
- entitiesByName (page 168)

Related Sample Code
Core Data HTML Store
CoreRecipes
Departments and Employees
QTMetadataEditor

Declared In
NSEntityDescription.h

insertNewObjectForEntityForName:inManagedObjectContext:

Creates, configures, and returns a new autoreleased 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
Note that despite the word "new" in the method name, the object returned is autoreleased ("new" is not the first word in the method name—see Memory Management Rules)

This method makes it easier for you to create instances of a given entity without having to know the class used to represent the entity, which may be particularly useful early in the development life-cycle when classes and class names are volatile. It also takes care of the details of managed object creation.

This method makes it easier for you to create instances of a given entity without worrying about the details of managed object creation when there is no need to explicitly assign a new managed object to a specific persistent store.

This is particularly useful on Mac OS X v10.4 as you can use this method to create a new managed object without having to know the class used to represent the entity, especially 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, `initWithEntity:insertIntoManagedObjectContext:` (page 111) returns an instance of the appropriate class for the entity. The equivalent code for Mac OS X v10.5 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];
```

Availability
Available in Mac OS X v10.4 and later.

See Also
`- initWithEntity:insertIntoManagedObjectContext:` (page 111)

Related Sample Code
CoreRecipes
Departments and Employees
QTMetadataEditor

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 Mac OS X v10.4 and later.

See Also
- `propertiesByName` (page 46)
- `relationshipsByName` (page 46)
- `relationshipsWithDestinationEntity:` (page 47)

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- setAbstract: (page 47)

Declared In
NSEntityDescription.h

isKindOfEntity:

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

```
- (BOOL)isKindOfEntity:(NSEntityDescription *)entity
```

Parameters

entity
An entity.

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

Availability
Available in Mac OS X v10.5 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

See Also
`setEntities:` (page 172) (`NSManagedObjectModel`)
`setEntities:forConfiguration:` (page 173): (`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 Mac OS X v10.4 and later.

See Also
`- setName:` (page 48)

Related Sample Code
Core Data HTML Store
CoreRecipes
ManagedObjectDataFormatter

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 Mac OS X v10.4 and later.

See Also
`- propertiesByName` (page 46)
`- setProperties:` (page 48)
`- attributesByName` (page 42)
`- relationshipsByName` (page 46)

Related Sample Code
ManagedObjectDataFormatter

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 Mac OS X v10.4 and later.

See Also
- `attributesByName` (page 42)
- `relationshipsByName` (page 46)
- `relationshipsWithDestinationEntity:` (page 47)

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 Mac OS X v10.4 and later.

See Also
- `attributesByName` (page 42)
- `propertiesByName` (page 46)
- `relationshipsWithDestinationEntity:` (page 47)

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 and later.

See Also

- `attributesByName` (page 42)
- `propertiesByName` (page 46)
- `relationshipsByName` (page 46)

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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

See Also

- `name` (page 45)

Declared In

`NSEntityDescription.h`

setProperties:

Sets the properties array of the receiver.

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

Parameters

properties

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

Special Considerations

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

Availability
Available in Mac OS X v10.4 and later.

See Also
- `properties` (page 45)
- `propertiesByName` (page 46)
- `attributesByName` (page 42)
- `relationshipsByName` (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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

See Also
- `userInfo` (page 51)

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 Mac OS X v10.5 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

See Also
- `setUserInfo:` (page 49)

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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 184) method to retrieve the entity description for this entity name.

Availability
Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
- sourceEntityVersionHash (page 64)
- setDestinationEntityVersionHash: (page 60)

Declared In
NSEntityMapping.h

setSourceExpression:

Sets the source expression for the receiver.

- (void)setSourceExpression:(NSExpression *)*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 Mac OS X v10.5 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 Mac OS X v10.5 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 189) method on the migration manager to retrieve the entity description for this entity name.

Availability

Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
- `setUserInfo:` (page 63)

Declared In
`NSEntityMapping.h`

Constants

Entity Mapping Types

These constants specify the types of entity mapping.

```
enum {
    NSUndefinedEntityMappingType   = 0x00,
    NSCustomEntityMappingType      = 0x01,
    NSAddEntityMappingType         = 0x02,
    NSRemoveEntityMappingType      = 0x03,
    NSCopyEntityMappingType        = 0x04,
    NSTransformEntityMappingType   = 0x05
};
```

Constants

`NSUndefinedEntityMappingType`
Specifies that the developer handles destination instance creation.
Available in Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

`NSCustomEntityMappingType`
Specifies a custom mapping.
Available in Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

`NSAddEntityMappingType`
Specifies that this is a new entity in the destination model.
Instances of the entity only exist in the destination.
Available in Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

`NSRemoveEntityMappingType`
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 Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

`NSCopyEntityMappingType`
Specifies that source instances are migrated as-is.
Available in Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

`NSTransformEntityMappingType`
Specifies that entity exists in source and destination and is mapped.
Available in Mac OS X v10.5 and later.
Declared in `NSEntityMapping.h`.

Declared In
NSEntityMapping.h

NSEntityMappingType

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

```
typedef NSUInteger NSEntityMappingType;
```

Discussion
For possible values, see "Entity Mapping Types" (page 66).

Availability
Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 182) method.

Special Considerations

If you override this method and do not invoke `super`, you must invoke `NSMigrationManager`'s `associateSourceInstance:withDestinationInstance:forEntityMapping:` (page 182) to associate the source and destination instances as required. .

Availability

Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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;
```

Constants

`NSMigrationManagerKey`

Key for the migration manager.

Available in Mac OS X v10.5 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationSourceObjectKey`

Key for the source object.

Available in Mac OS X v10.5 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationDestinationObjectKey`

Key for the destination object.

Available in Mac OS X v10.5 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationEntityMappingKey`

Key for the entity mapping object.

Available in Mac OS X v10.5 and later.

Declared in `NSEntityMigrationPolicy.h`.

`NSMigrationPropertyMappingKey`

Key for the property mapping object.

Available in Mac OS X v10.5 and later.

Declared in `NSEntityMigrationPolicy.h`.

Discussion

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

Declared In

`NSEntityMigrationPolicy.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 Mac OS X v10.4 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 143) (`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 78)
  Returns the fetch request of the receiver.
- `setFetchRequest:` (page 79)
  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
Available in Mac OS X v10.4 and later.

See Also
- `setFetchRequest:` (page 79)

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

Available in Mac OS X v10.4 and later.

See Also

- `fetchRequest` (page 78)

Declared In

`NSFetchedPropertyDescription.h`

NSFetchRequest Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSCopying NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.4 and later.
Declared in	CoreData/NSFetchRequest.h
Companion guides	Core Data Programming Guide Predicate Programming Guide
Related sample code	Core Data HTML Store CoreRecipes Departments and Employees QTMetadataEditor

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 an 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 '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 134) 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 87), `includesSubentities` (page 85), `includesPropertyValues` (page 84), and `returnsObjectsAsFaults` (page 87)).

You use `NSFetchRequest` objects with the method `executeFetchRequest:error:` (page 134), defined by `NSManagedObjectContext`.

You often predefine fetch requests in a managed object model—`NSManagedObjectModel` 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 84)
  Returns the entity specified for the receiver.
- `setEntity:` (page 88)
  Sets the entity of the receiver.
- `includesSubentities` (page 85)
  Returns a Boolean value that indicates whether the receiver includes subentities in the results.
- `setIncludesSubentities:` (page 89)
  Sets whether the receiver includes subentities.

Fetch Constraints

- `predicate` (page 85)
  Returns the predicate of the receiver.
- `setPredicate:` (page 90)
  Sets the predicate of the receiver.
- `fetchLimit` (page 84)
  Returns the fetch limit of the receiver.
- `setFetchLimit:` (page 89)
  Sets the fetch limit of the receiver.
- `affectedStores` (page 83)
  Returns an array containing the persistent stores specified for the receiver.
- `setAffectedStores:` (page 88)
  Sets the array of persistent stores that will be searched by the receiver.

Sorting

- `sortDescriptors` (page 92)
  Returns the sort descriptors of the receiver.
- `setSortDescriptors:` (page 92)
  Sets the array of sort descriptors of the receiver.

Prefetching

- `relationshipKeyPathsForPrefetching` (page 86)
  Returns the array of relationship keypaths to prefetch along with the entity for the request.
- `setRelationshipKeyPathsForPrefetching:` (page 90)
  Sets an array of relationship keypaths to prefetch along with the entity for the request.

Managing How Results Are Returned

- `resultType` (page 87)
  Returns the result type of the receiver.
- `setResultType:` (page 91)
  Sets the result type of the receiver.
- `includesPropertyValues` (page 84)
  Returns a Boolean value that indicates whether, when the fetch is executed, property data is obtained from the persistent store.
- `setIncludesPropertyValues:` (page 89)
  Sets if, when the fetch is executed, property data is obtained from the persistent store.
- `returnsObjectsAsFaults` (page 87)
  Returns a Boolean value that indicates whether the objects resulting from a fetch using the receiver are faults.
- `setReturnsObjectsAsFaults:` (page 91)
  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 Mac OS X v10.4 and later.

See Also
- setAffectedStores: (page 88)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- setEntity: (page 88)

Declared In
NSFetchRequest.h

fetchLimit

Returns the fetch limit of the receiver.

```
- (NSUInteger)fetchLimit
```

Return Value
The fetch limit of the receiver.

Availability
Available in Mac OS X v10.4 and later.

See Also
- setFetchLimit: (page 89)

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 87)). 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 Mac OS X v10.5 and later.

See Also
`- setIncludesPropertyValues:` (page 89)

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`.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- setIncludesSubentities:` (page 89)

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 Predicates Programming Guide.

Availability
Available in Mac OS X v10.4 and later.

See Also
`- setPredicate:` (page 90)

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 Mac OS X v10.5 and later.

See Also
`- setRelationshipKeyPathsForPrefetching:` (page 90)

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 91) to set the instance type of objects returned from executing the request—for possible values, see "Fetch request result types" (page 93). If you set the value to `NSManagedObjectIDResultType`, this will demote any sort orderings to "best effort" hints if you do not include the property values in the request.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- setResultType:` (page 91)

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 87)) 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 Mac OS X v10.5 and later.

See Also
`- setReturnsObjectsAsFaults:` (page 91)

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 Mac OS X v10.4 and later.

See Also
- affectedStores (page 83)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- entity (page 84)

Related Sample Code
Core Data HTML Store
CoreRecipes
Departments and Employees
QTMetadataEditor

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

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 Mac OS X v10.4 and later.

See Also

- `fetchLimit` (page 84)

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 84).

Availability

Available in Mac OS X v10.5 and later.

See Also

- `includesPropertyValues` (page 84)

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).

Discussion
The default is YES.

Availability
Available in Mac OS X v10.5 and later.

See Also
- includesSubentities (page 85)

Declared In
NSFetchRequest.h

setPredicate:

Sets the predicate of the receiver.

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

Parameters

predicate

The predicate of the receiver.

Discussion
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 Predicates Programming Guide.

Availability
Available in Mac OS X v10.4 and later.

See Also
- predicate (page 85)

Related Sample Code
CoreRecipes
QTMetadataEditor

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 86).

Availability

Available in Mac OS X v10.5 and later.

See Also

- `relationshipKeyPathsForPrefetching` (page 86)

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 87).

Availability

Available in Mac OS X v10.5 and later.

See Also

- `resultType` (page 87)

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 113) 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 87).

Availability
Available in Mac OS X v10.5 and later.

See Also
- `returnsObjectsAsFaults` (page 87)

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.

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 Mac OS X v10.4 and later.

See Also
- `sortDescriptors` (page 92)

Related Sample Code
CoreRecipes

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 are used to order the array of objects returned when the fetch is executed.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `setSortDescriptors:` (page 92)

Declared In
NSFetchRequest.h

Constants

Fetch request result types

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

```
enum {
    NSManagedObjectResultType        = 0x00,
    NSManagedObjectIDResultType      = 0x01
};
```

Constants

`NSManagedObjectResultType`
Specifies that the request returns managed objects.
Available in Mac OS X v10.5 and later.
Declared in `NSFetchRequest.h`.

`NSManagedObjectIDResultType`
Specifies that the request returns managed object IDs.
Available in Mac OS X v10.5 and later.
Declared in `NSFetchRequest.h`.

Discussion
These constants are used by `resultType` (page 87).

NSFetchRequestResultType

Defines the type for the fetch request result type.

```
typedef NSUInteger NSFetchRequestResultType;
```

Availability
Available in Mac OS X v10.5 and later.

Declared In
NSFetchRequest.h

NSFetchRequestExpression Class Reference

Inherits from	NSExpression : NSObject
Conforms to	NSCoding (NSExpression) NSCopying (NSExpression) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.5 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 96)
Returns an expression which will evaluate to the result of executing a fetch request on a context.

Examining a Fetch Request Expression

`- requestExpression` (page 97)
Returns the expression for the receiver's fetch request.

- `contextExpression` (page 96)
  Returns the expression for the receiver's managed object context.
- `isCountOnlyRequest` (page 97)
  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.

```
+ (NSExpression *)expressionForFetch:(NSExpression *)fetch
    context:(NSExpression *)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 `countForFetchRequest:error:` (page 132) with the fetch request (from *fetch*). If `NO`, when the new expression is evaluated the managed object context will perform `executeFetchRequest:error:` (page 134) 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 Mac OS X v10.5 and later.

Declared In
`NSFetchRequestExpression.h`

Instance Methods

contextExpression

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

```
- (NSExpression *)contextExpression
```

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

Availability
Available in Mac OS X v10.5 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 96)) will perform `executeFetchRequest:error:` (page 134): with the `requestExpression` (page 97); if this method returns `YES`, the managed object context will perform `countForFetchRequest:error:` (page 132) with the `requestExpression` (page 97).

Availability
Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.4 and later.
Declared in	CoreData/NSManagedObject.h
Companion guides	Core Data Programming Guide Model Object Implementation Guide Core Data Utility Tutorial
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass Departments and Employees QTMetadataEditor

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 111)).

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 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:`, `respondsToSelector:`, `retain`, `release`, `autorelease`, `retainCount`, `description`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, and `isFault`. Since `isEqual` and `hash` do not cause a fault to fire, managed objects can typically be placed in collections without firing a fault. Note, however, that invoking key-value coding methods on the collection object might in turn result in an invocation of `valueForKey:` on a managed object, which would fire 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:`, `respondsToSelector:`, `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 111)).

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 105)—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 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 109) 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 122), 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 121) 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 121)), 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 Validation.

Tasks

Initializing a Managed Object

- `initWithEntity:insertIntoManagedObjectContext:` (page 111)
  Initializes the receiver and inserts it into the specified managed object context.

Getting a Managed Object's Identity

- `entity` (page 110)
  Returns the entity description of the receiver.
- `objectID` (page 115)
  Returns the object ID of the receiver.
- `self` (page 117)
  Returns the receiver.

Getting State Information

- `managedObjectContext` (page 114)
  Returns the managed object context with which the receiver is registered.
- `isInserted` (page 113)
  Returns a Boolean value that indicates whether the receiver has been inserted in a managed object context.
- `isUpdated` (page 114)
  Returns a Boolean value that indicates whether the receiver has unsaved changes.
- `isDeleted` (page 112)
  Returns a Boolean value that indicates whether the receiver will be deleted during the next save.
- `isFault` (page 113)
  Returns a Boolean value that indicates whether the receiver is a fault.
- `hasFaultForRelationshipNamed:` (page 110)
  Returns a Boolean value that indicates whether the relationship for a given key is a fault.

Managing Life Cycle and Change Events

- `awakeFromFetch` (page 105)
  Invoked automatically by the Core Data framework after the receiver has been fetched.
- `awakeFromInsert` (page 106)
  Invoked automatically by the Core Data framework when the receiver is first inserted into a managed object context.
- `changedValues` (page 107)
  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 107)
  Returns a dictionary of the last fetched or saved values of the receiver for the properties specified by the given keys.
- `dealloc` (page 108)
  Deallocates the memory occupied by the receiver.
- `didSave` (page 109)
  Invoked automatically by the Core Data framework after the receiver's managed object context completes a save operation.
- `willTurnIntoFault` (page 124)
  Invoked automatically by the Core Data framework before receiver is converted to a fault.
- `didTurnIntoFault` (page 109)
  Invoked automatically by the Core Data framework when the receiver is turned into a fault.
- `willSave` (page 124)
  Invoked automatically by the Core Data framework when the receiver's managed object context is saved.

Supporting Key-Value Coding

- `valueForKey:` (page 122)
  Returns the value for the property specified by *key*.
- `setValue:forKey:` (page 119)
  Sets the specified property of the receiver to the specified value.
- `mutableSetValueForKey:` (page 114)
  Returns a mutable set that provides read-write access to the unordered to-many relationship specified by a given key.
- `primitiveValueForKey:` (page 116)
  Returns from the receiver's private internal storage the value for the specified property.
- `setPrimitiveValue:forKey:` (page 118)
  Sets in the receiver's private internal storage the value of a given property.

Validation

- `validateValue:forKey:error:` (page 122)
  Validates a property value for a given key.
- `validateForDelete:` (page 120)
  Determines whether the receiver can be deleted in its current state.
- `validateForInsert:` (page 120)
  Determines whether the receiver can be inserted in its current state.
- `validateForUpdate:` (page 121)
  Determines whether the receiver's current state is valid.

Supporting Key-Value Observing

`+ automaticallyNotifiesObserversForKey:` (page 105)
Returns a Boolean value that indicates whether the receiver provides automatic support for key-value observing change notifications for the given key.

`- didAccessValueForKey:` (page 108)
Provides support for key-value observing access notification.

`- observationInfo` (page 116)
Returns the observation info of the receiver.

`- setObservationInfo:` (page 117)
Sets the observation info of the receiver.

`- willAccessValueForKey:` (page 123)
Provides support for key-value observing access notification.

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 more about key-value observation, see *Key-Value Observing Programming Guide*.

You should only override this to return `YES` for properties that are not defined for the corresponding entity in the managed object model—see Subclassing Notes.

Instance Methods

awakeFromFetch

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

```
- (void)awakeFromFetch
```

Discussion
This method is commonly used to compute derived values or to recreate transient relationships from the receiver's persistent properties.

The change processing is explicitly disabled around this method so that you can conveniently 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.

If you want to set attribute values in an implementation of this method, you should typically use primitive accessor methods (either `setPrimitiveValue:forKey:` (page 118) 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 Mac OS X v10.4 and later.

See Also
- `awakeFromInsert` (page 106)
- `primitiveValueForKey:` (page 116)
- `setPrimitiveValue:forKey:` (page 118)

Related Sample Code
CoreRecipes

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 118) 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.

Availability
Available in Mac OS X v10.4 and later.

See Also
- awakeFromFetch (page 105)

Related Sample Code
CoreRecipes
Departments and Employees
QTMetadataEditor

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 Mac OS X v10.4 and later.

See Also
- committedValuesForKeys: (page 107)

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 Mac OS X v10.4 and later.

See Also
- `changedValues` (page 107)

Declared In
`NSManagedObject.h`

dealloc

Deallocates the memory occupied by the receiver.

```
- (void)dealloc
```

Discussion
This method first invokes `didTurnIntoFault` (page 109).

You should typically not override this method—instead you should put "clean-up" code in `didTurnIntoFault` (page 109).

See Also
- `didTurnIntoFault` (page 109)

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 123), 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 Mac OS X v10.4 and later.

See Also
`- willAccessValueForKey:` (page 123)

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
This method is commonly used to notify other objects after a save, and to compute persisted values from other transient values, to set time-stamps, and so on. This method can have "side effects" on the persistent values. Note however that `setValue:forKey:` should not be used within the implementation of this method (because it generates additional change notifications)—you should use `setPrimitiveValue:forKey:` instead.

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 112).

Availability
Available in Mac OS X v10.4 and later.

See Also
`- willSave` (page 124)

Declared In
`NSManagedObject.h`

didTurnIntoFault

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

```
- (void)didTurnIntoFault
```

Discussion
This method may be used 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 143)).

Availability
Available in Mac OS X v10.4 and later.

See Also
`- willTurnIntoFault` (page 124)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

Related Sample Code
Core Data HTML Store
CoreRecipes
ManagedObjectDataFormatter

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 Mac OS X v10.5 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 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 106) to be invoked).

You are discouraged from overriding this method—you should instead override awakeFromInsert (page 106) and/or awakeFromFetch (page 105) (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 129). 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 should 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
{
    if (self = [super initWithEntity:entity
insertIntoManagedObjectContext:context])
    {
        // perform additional initialization
    }
    return self;
}
```

Availability
Available in Mac OS X v10.4 and later.

See Also
`+ insertNewObjectForEntityForName:inManagedObjectContext:` (page 41)

Related Sample Code
Core Data HTML Store

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 Mac OS X v10.4 and later.

See Also
`- isFault` (page 113)
`- isInserted` (page 113)
`- isUpdated` (page 114)
`- deletedObjects` (page 132) (`NSManagedObjectContext`)
`NSManagedObjectContextObjectsDidChangeNotification` (page 155) (`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 Mac OS X v10.4 and later.

See Also
- `isDeleted` (page 112)
- `isInserted` (page 113)
- `isUpdated` (page 114)

Related Sample Code
ManagedObjectDataFormatter

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 Mac OS X v10.4 and later.

See Also
- `isDeleted` (page 112)
- `isFault` (page 113)
- `isUpdated` (page 114)

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 Mac OS X v10.4 and later.

See Also
- isDeleted (page 112)
- isFault (page 113)
- isInserted (page 113)

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

Availability
Available in Mac OS X v10.4 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: Subclasses should 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 122)
- `primitiveValueForKey:` (page 116)
- `setObservationInfo:` (page 117)

objectID

Returns the object ID of the receiver.

```
- (NSManagedObjectID *)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 Mac OS X v10.4 and later.

See Also

`URIRepresentation` (page 159) (`NSManagedObjectID`)

Related Sample Code

Core Data HTML Store

CoreRecipes

CustomAtomicStoreSubclass

Declared In
NSManagedObject.h

observationInfo

Returns the observation info of the receiver.

```
- (void *)observationInfo
```

Return Value
The observation info of the receiver.

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

Important: Subclasses should not override this method.

Availability
Available in Mac OS X v10.4 and later.

See Also
- setObservationInfo: (page 117)

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 123) and didAccessValueForKey: (page 108)). 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 from ADC 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 Mac OS X v10.4 and later.

See Also
- `- setObservationInfo:` (page 117)
- `- valueForKey:` (page 122)
- `- mutableSetValueForKey:` (page 114)

Related Sample Code
CoreRecipes

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 123).

setObservationInfo:

Sets the observation info of the receiver.

```
- (void)setObservationInfo:(void *)value
```

Parameters
value
The new observation info for the receiver.

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

Availability
Available in Mac OS X v10.4 and later.

See Also
- `- observationInfo` (page 116)

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

Subclasses should 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 ADC 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 Mac OS X v10.4 and later.

See Also
- `primitiveValueForKey:` (page 116)
- `valueForKey:` (page 122)
- `mutableSetValueForKey:` (page 114)
- `awakeFromFetch` (page 105)

Related Sample Code
CoreRecipes

Declared In
`NSManagedObject.h`

setValue:forKey:

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: Subclasses should not override this method.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `valueForKey:` (page 122)
- `primitiveValueForKey:` (page 116)
- `setObservationInfo:` (page 117)

Related Sample Code
CoreRecipes
QTMetadataEditor

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 Validation).

Availability
Available in Mac OS X v10.4 and later.

See Also
- validateForInsert: (page 120)
- validateForUpdate: (page 121)
- validateValue:forKey:error: (page 122)

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 Validation).

Availability

Available in Mac OS X v10.4 and later.

See Also

- `validateForDelete:` (page 120)
- `validateForUpdate:` (page 121)
- `validateValue:forKey:error:` (page 122)

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 Validation).

Availability

Available in Mac OS X v10.4 and later.

See Also

- `validateForDelete:` (page 120)
- `validateForInsert:` (page 120)
- `validateValue:forKey:error:` (page 122)

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. If no basic errors exist, this default implementation then validates the value according to the object itself—it searches for a method of the form `validate<Key>:error:` and invokes it if it exists. These are the methods that your custom classes can implement to validate individual properties, such as `validateAge:error:` to check that the value the user entered is within acceptable limits. If it finds an unacceptable value, the `validateAge:error:` method should return `NO` and in *error* an `NSError` object that describes the problem.

Availability

Available in Mac OS X v10.4 and later.

See Also

- `validateForDelete:` (page 120)
- `validateForInsert:` (page 120)
- `validateForUpdate:` (page 121)

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: Subclasses should not override this method.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `primitiveValueForKey:` (page 116)
- `setObservationInfo:` (page 117)
- `setObservationInfo:` (page 117)

Related Sample Code
Core Data HTML Store
CoreRecipes
QTMetadataEditor

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 108) 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 Mac OS X v10.4 and later.

See Also
- `didAccessValueForKey:` (page 108)

Declared In
`NSManagedObject.h`

willSave

Invoked automatically by the Core Data framework when the receiver's managed object context is saved.

```
- (void)willSave
```

Availability
Available in Mac OS X v10.4 and later.

See Also
- `didSave` (page 109)

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 109) 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 Mac OS X v10.5 and later.

See Also
- `didTurnIntoFault` (page 109)

Declared In
`NSManagedObject.h`

Constants

The following constants relate to errors returned following validation failures.

`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.
`NSValidationKeyErrorKey`	Key for the key that failed to validate for a validation error.
`NSValidationPredicateErrorKey`	For predicate-based validation, key for the predicate for the condition that failed to validate.
`NSValidationValueErrorKey`	If non-nil, the key for the value for the key that failed to validate for a validation error.

NSManagedObjectContext Class Reference

Inherits from	NSObject
Conforms to	NSCoding NSLocking NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.4 and later.
Declared in	NSManagedObjectContext.h
Companion guides	Core Data Programming Guide NSPersistentDocument Core Data Tutorial Core Data Utility Tutorial Predicate Programming Guide
Related sample code	Core Data HTML Store CoreRecipes Departments and Employees QTMetadataEditor

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.

If you are using Cocoa Bindings, the context can also serve as a controller. It implements the `NSEditor` and `NSEditorRegistration` informal protocols, although there should typically be little reason for you to directly invoke any method other than `commitEditing` (page 130) or `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131), and then only rarely.

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

- `objectRegisteredForID:` (page 139)
  Returns the object for a specified ID, if the object is registered with the receiver.
- `objectWithID:` (page 140)
  Returns the object for a specified ID.
- `executeFetchRequest:error:` (page 134)
  Returns an array of objects that meet the criteria specified by a given fetch request.
- `countForFetchRequest:error:` (page 132)
  Returns the number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:`.
- `registeredObjects` (page 144)
  Returns the set of objects registered with the receiver.

Managed Object Management

- `insertObject:` (page 136)
  Registers an object to be inserted in the receiver's persistent store the next time changes are saved.

- `deleteObject:` (page 133)
  Specifies an object that should be removed from its persistent store when changes are committed.
- `assignObject:toPersistentStore:` (page 129)
  Specifies the store in which a newly-inserted object will be saved.
- `obtainPermanentIDsForObjects:error:` (page 141)
  Converts to permanent IDs the object IDs of the objects in a given array.
- `detectConflictsForObject:` (page 134)
  Marks an object for conflict detection.
- `refreshObject:mergeChanges:` (page 143)
  Updates the persistent properties of a managed object to use the latest values from the persistent store.
- `processPendingChanges` (page 142)
  Forces the receiver to process changes to the object graph.
- `insertedObjects` (page 136)
  Returns the set of objects that have been inserted into the receiver but not yet saved in a persistent store.
- `updatedObjects` (page 151)
  Returns the set of objects registered with the receiver that have uncommitted changes.
- `deletedObjects` (page 132)
  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 137)
  Merges the changes specified in a given notification.

Undo Management

- `undoManager` (page 150)
  Returns the undo manager of the receiver.
- `setUndoManager:` (page 149)
  Sets the undo manager of the receiver.
- `undo` (page 150)
  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 143)
  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 145)
  Returns the receiver to its base state.
- `rollback` (page 145)
  Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values.

- `save:` (page 146)
  Attempts to commit unsaved changes to registered objects to their persistent store.
- `hasChanges` (page 135)
  Returns a Boolean value that indicates whether the receiver has uncommitted changes.

Locking

- `lock` (page 137)
  Attempts to acquire a lock on the receiver.
- `unlock` (page 151)
  Relinquishes a previously acquired lock.
- `tryLock` (page 150)
  Attempts to acquire a lock.

Delete Propagation

- `propagatesDeletesAtEndOfEvent` (page 142)
  Returns a Boolean that indicates whether the receiver propagates deletes at the end of the event in which a change was made.
- `setPropagatesDeletesAtEndOfEvent:` (page 147)
  Sets whether the context propagates deletes to related objects at the end of the event.

Retaining Registered Objects

- `retainsRegisteredObjects` (page 145)
  Returns a Boolean that indicates whether the receiver sends a `retain` message to objects upon registration.
- `setRetainsRegisteredObjects:` (page 148)
  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 142)
  Returns the persistent store coordinator of the receiver.
- `setPersistentStoreCoordinator:` (page 147)
  Sets the persistent store coordinator of the receiver.

Managing the Staleness Interval

- `stalenessInterval` (page 149)
  Returns the staleness interval of the receiver.

- `setStalenessInterval:` (page 148)
  Sets the staleness interval of the receiver.

Managing the Merge Policy

- `mergePolicy` (page 138)
  Returns the merge policy of the receiver.
- `setMergePolicy:` (page 146)
  Sets the merge policy of the receiver.

Supporting NSKeyValueObserving Protocol

- `observeValueForKeyPath:ofObject:change:context:` (page 140)
  This message is sent to the receiver when the value at the specified key path relative to the given object has changed.

Supporting NSEditor and NSEditorRegistration Protocols

- `commitEditing` (page 130)
  Returns a Boolean that indicates whether the receiver was able to commit any pending edits in known editors.
- `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131)
  Attempts to commit any pending changes in known editors of the receiver.
- `discardEditing` (page 134)
  Causes the receiver to discard any changes in known editors, restoring the previous values
- `objectDidBeginEditing:` (page 138)
  Provides support for the `NSEditorRegistration` informal protocol.
- `objectDidEndEditing:` (page 139)
  Provides support for the`NSEditorRegistration` informal protocol.

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 213).

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 Mac OS X v10.4 and later.

See Also
- `insertObject:` (page 136)
- `persistentStoreCoordinator` (page 142)

Related Sample Code
CoreRecipes

Declared In
`NSManagedObjectContext.h`

commitEditing

Returns a Boolean that indicates whether the receiver was able to commit any pending edits in known editors.

```
- (BOOL)commitEditing
```

Return Value
`YES` if the changes were successfully applied, otherwise `NO`.

Discussion
This method attempts to commit pending changes in known *external editors*—it does *not* commit unsaved changes to registered objects to their persistent store (contrast `save:` (page 146)). Known editors are either instances of a subclass of NSController or (more rarely) user interface controls that may contain pending edits—such as text fields—that registered with the context using `objectDidBeginEditing:` and have not yet unregistered using a subsequent invocation of `objectDidEndEditing:`. Note that controllers only register with the context as being an editor if their `content` binding is *not* bound—if they have content of any kind, then they do not register.

The receiver iterates through the array of its known editors and invokes `commitEditing` on each until either it reaches the end of the array or an editor returns `NO`. If an editor returns `NO`, then the context also returns `NO`; otherwise the context returns `YES`.

You may find this method useful in some situations (typically if you are using Cocoa Bindings) when you want to ensure that pending changes are applied before a change in user interface state. For example, you may need to ensure that changes pending in a text field are applied before a window is closed. See also `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131) which performs a similar function but which allows you to specify a delegate that will handle any errors—the delegate is informed which editor failed to commit, which may be useful if you want to display an alert panel on the editor's window.

See Also
- `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131)
- `discardEditing` (page 134)
- `objectDidBeginEditing:` (page 138)
- `objectDidEndEditing:` (page 139)

commitEditingWithDelegate:didCommitSelector:contextInfo:

Attempts to commit any pending changes in known editors of the receiver.

```
-(void)commitEditingWithDelegate:(id)delegate
    didCommitSelector:(SEL)didCommitSelector   contextInfo:(void *)contextInfo
```

Parameters

delegate

An object that can serve as the receiver's delegate. It should implement the method specified by *didCommitSelector*.

didCommitSelector

A selector that is invoked on delegate. The method specified by the selector must have the same signature as the following method:

```
- (void)editor:(id)editor didCommit:(BOOL)didCommit contextInfo:(void
*)contextInfo
```

contextInfo

Contextual information that is sent as the `contextInfo` argument to delegate when *didCommitSelector* is invoked.

Discussion

Provides support for the `NSEditor` informal protocol. This method attempts to commit pending changes in known *external editors*—it does *not* commit unsaved changes to registered objects to their persistent store (contrast `save:` (page 146)). Known editors are either instances of a subclass of `NSController` or (more rarely) user interface controls that may contain pending edits—such as text fields—that registered with the context using `objectDidBeginEditing:` and have not yet unregistered using a subsequent invocation of `objectDidEndEditing:`. Note that controllers only register with the context as being an editor if their `content` binding is *not* bound—if they have content of any kind, then they do not register.

The receiver iterates through the array of its known editors and invokes `commitEditing` on each. The receiver then sends the message specified by the *didCommitSelector* selector to the specified delegate.

The `didCommit` argument is the value returned by the editor specified by `editor` from the `commitEditing` message. The `contextInfo` argument is the same value specified as the *contextInfo* parameter—you may use this value however you wish.

If an error occurs while attempting to commit, for example if key-value coding validation fails, your implementation of this method should typically send the view in which editing is being performed a `presentError:modalForWindow:delegate:didRecoverSelector:contextInfo:` message, specifying the view's containing window.

You may find this method useful in some situations (typically if you are using Cocoa Bindings) when you want to ensure that pending changes are applied before a change in user interface state. For example, you may need to ensure that changes pending in a text field are applied before a window is closed. See also `commitEditing` (page 130) which performs a similar function but which allows you to handle any errors directly, although it provides no information beyond simple success/failure.

See Also

- `commitEditing` (page 130)
- `discardEditing` (page 134)
- `objectDidBeginEditing:` (page 138)
- `objectDidEndEditing:` (page 139)

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 134). If an error occurrs during the processing of the request, returns `NSNotFound`.

Availability

Available in Mac OS X v10.5 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

Note that the returned set does not necessarily include all the objects that have been deleted (using `deleteObject:` (page 133))—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—it does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 155) notification when a change is made, and a `NSManagedObjectContextDidSaveNotification` (page 155) notification when changes are committed (although again note that the set of deleted objects given for a `NSManagedObjectContextDidSaveNotification` (page 155) does not include objects that were inserted and deleted without an intervening save operation—that is, they had not been saved to a persistent store).

Availability
Available in Mac OS X v10.4 and later.

See Also
- `deleteObject:` (page 133)
- `insertedObjects` (page 136)
- `registeredObjects` (page 144)
- `updatedObjects` (page 151)
- `isDeleted` (page 112) (`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 Mac OS X v10.4 and later.

See Also
- `deletedObjects` (page 132)
- `isDeleted` (page 112) (`NSManagedObject`)

Related Sample Code
Core Data HTML Store
CoreRecipes
Departments and Employees
QTMetadataEditor

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 146) *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 Mac OS X v10.4 and later.

Declared In

`NSManagedObjectContext.h`

discardEditing

Causes the receiver to discard any changes in known editors, restoring the previous values

```
- (void)discardEditing
```

Discussion

Provides support for the `NSEditor` informal protocol. Causes the receiver to discard any changes in known editors, restoring the previous values. This method only applies to known editors (see `commitEditing` (page 130)). To discard general edits, use `rollback` (page 145) or `reset` (page 145).

See Also

- `commitEditing` (page 130)
- `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131)
- `objectDidBeginEditing:` (page 138)
- `objectDidEndEditing:` (page 139)
- `reset` (page 145)
- `rollback` (page 145)

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 Mac OS X v10.4 and later.

Related Sample Code
Core Data HTML Store
CoreRecipes
Departments and Employees
QTMetadataEditor

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
This property is not key-value observing compliant (see *Key-Value Observing Programming Guide*)—if you are using Cocoa bindings, you cannot bind to the `hasChanges` property of a managed object context.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `save:` (page 146)
- `reset` (page 145)
- `rollback` (page 145)

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 155) notification when a change is made, and a `NSManagedObjectContextDidSaveNotification` (page 155) notification when changes are committed.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `deletedObjects` (page 132)
- `insertObject:` (page 136)
- `registeredObjects` (page 144)
- `updatedObjects` (page 151)

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 145) message) before a save operation, the object is unregistered from the receiver.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `insertedObjects` (page 136)

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 151).

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 Mac OS X v10.4 and later.

See Also
- `tryLock` (page 150)
- `unlock` (page 151)

Declared In
`NSManagedObjectContext.h`

mergeChangesFromContextDidSaveNotification:

Merges the changes specified in a given notification.

```
- (void)mergeChangesFromContextDidSaveNotification:(NSNotification *)notification
```

Parameters

notification

An instance of an `NSManagedObjectContextDidSaveNotification` (page 155) 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 133): on those which have been deleted.

You can use this method to, for example, update a managed object context on the main thread with work completed in another context in another thread. You must, though, `lock` (page 137) the receiver or otherwise ensure thread safety (that is, the notification contents are handled safely by Core Data, but the receiver's usage is still expected to conform to the standard Core Data threading policies). For example, you might implement a method to handle a notification that a worker thread had finished saving as follows:

```
- (void)workerThreadObjectContextDidSave:(NSNotification*)saveNotification {
    NSManagedObjectContext *appMOC = [[NSApp delegate] managedObjectContext];
    [appMOC
performSelectorOnMainThread:@selector(mergeChangesFromContextDidSaveNotification:)
                           withObject:saveNotification
                        waitUntilDone:NO];
}
```

In this case, serialization is enforced by the main thread's run loop.

Availability

Available in Mac OS X v10.5 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 Mac OS X v10.4 and later.

Declared In

`NSManagedObjectContext.h`

objectDidBeginEditing:

Provides support for the `NSEditorRegistration` informal protocol.

```
- (void)objectDidBeginEditing:(id)editor
```

Parameters

editor

An external editor that has changes that may affect the receiver.

Discussion

This message should be sent to the receiver when *editor* has uncommitted changes that can affect the receiver. There should typically be no reason for you to invoke this method directly.

See Also

- `commitEditing` (page 130)
- `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131)
- `discardEditing` (page 134)
- `objectDidEndEditing:` (page 139)

objectDidEndEditing:

Provides support for the`NSEditorRegistration` informal protocol.

```
- (void)objectDidEndEditing:(id)editor
```

Parameters

editor

An external editor that has made changes that affect the receiver.

Discussion

This message should be sent to the receiver when *editor* has finished editing a property belonging to the receiver. There should typically be no reason for you to invoke this method directly.

See Also

- `commitEditing` (page 130)
- `commitEditingWithDelegate:didCommitSelector:contextInfo:` (page 131)
- `discardEditing` (page 134)
- `objectDidBeginEditing:` (page 138)

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 Mac OS X v10.4 and later.

See Also

- `objectWithID:` (page 140)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- objectRegisteredForID: (page 139)
- managedObjectIDForURIRepresentation: (page 210)
- URIRepresentation (page 159)

Related Sample Code
CoreRecipes

Declared In
NSManagedObjectContext.h

observeValueForKeyPath:ofObject:change:context:

This message is sent to the receiver when the value at the specified key path relative to the given object has changed.

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context
```

Parameters

keyPath
The key path, relative to *object*, to the value that has changed.

object
The source object of the key path *keyPath*.

change
A dictionary that describes the changes that have been made to the value of the property at the key path *keyPath* relative to *object*. For possible values, see `NSKeyValueObserving`.

context
The value that was provided when the receiver was registered to receive key-value observation notifications.

Availability
Available in Mac OS X v10.4 and later.

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 113) 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 Mac OS X v10.5 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 Mac OS X v10.4 and later.

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `redo` (page 143)
- `undo` (page 150)
- `undoManager` (page 150)

Related Sample Code
CoreRecipes
Departments and Employees

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 Mac OS X v10.4 and later.

See Also
- setPropagatesDeletesAtEndOfEvent: (page 147)

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 Mac OS X v10.4 and later.

See Also
- undo (page 150)
- processPendingChanges (page 142)

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 149)) 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 Mac OS X v10.4 and later.

See Also
- `detectConflictsForObject:` (page 134)
- `reset` (page 145)
- `setStalenessInterval:` (page 148)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `deletedObjects` (page 132)
- `insertedObjects` (page 136)
- `updatedObjects` (page 151)

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 Mac OS X v10.4 and later.

See Also
- `rollback` (page 145)
- `setStalenessInterval:` (page 148)
- `undo` (page 150)

Related Sample Code
QTMetadataEditor

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 Mac OS X v10.4 and later.

See Also
- `setRetainsRegisteredObjects:` (page 148)

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 Mac OS X v10.4 and later.

See Also
- `reset` (page 145)
- `setStalenessInterval:` (page 148)
- `undo` (page 150)
- `processPendingChanges` (page 142)

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 Mac OS X v10.4 and later.

See Also
- `reset` (page 145)
- `rollback` (page 145)
- `hasChanges` (page 135)

Related Sample Code
CoreRecipes

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 153).

Availability
Available in Mac OS X v10.4 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 Mac OS X v10.4 and later.

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `propagatesDeletesAtEndOfEvent` (page 142)

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 Mac OS X v10.4 and later.

See Also
- `retainsRegisteredObjects` (page 145)

Declared In
`NSManagedObjectContext.h`

setStalenessInterval:

Sets the staleness interval of the receiver.

```
- (void)setStalenessInterval:(NSTimeInterval)expiration
```

Parameters

expiration

The staleness interval of the receiver.

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 143)). 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.

The default is infinite staleness (represented by an interval of `0`).

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 Mac OS X v10.4 and later.

See Also
- `reset` (page 145)
- `rollback` (page 145)
- `stalenessInterval` (page 149)
- `undo` (page 150)
- `refreshObject:mergeChanges:` (page 143)

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
By default, a context provides its own undo manager. You can set the undo manager to `nil` to disable undo support, for example in a large import process. For more details, see *Core Data Programming Guide*.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `undoManager` (page 150)

Declared In
`NSManagedObjectContext.h`

stalenessInterval

Returns the staleness interval of the receiver.

```
- (NSTimeInterval)stalenessInterval
```

Return Value
The staleness interval of the receiver.

Discussion
For more details, see `setStalenessInterval:` (page 148).

Availability
Available in Mac OS X v10.4 and later.

See Also
- `setStalenessInterval:` (page 148)

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 Mac OS X v10.4 and later.

See Also
- `lock` (page 137)
- `unlock` (page 151)

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 Mac OS X v10.4 and later.

See Also
- `reset` (page 145)
- `rollback` (page 145)
- `undoManager` (page 150)
- `processPendingChanges` (page 142)

Declared In
`NSManagedObjectContext.h`

undoManager

Returns the undo manager of the receiver.

```
- (NSUndoManager *)undoManager
```

Return Value
The undo manager of the receiver.

Discussion
By default, a context provides its own undo manager. For more details, see `setUndoManager:` (page 149).

Availability
Available in Mac OS X v10.4 and later.

See Also
- `setUndoManager:` (page 149)

Related Sample Code
Departments and Employees

Declared In
`NSManagedObjectContext.h`

unlock

Relinquishes a previously acquired lock.

```
- (void)unlock
```

Availability
Available in Mac OS X v10.4 and later.

See Also
- `lock` (page 137)
- `tryLock` (page 150)

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—it does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 155) notification when a change is made, and a `NSManagedObjectContextDidSaveNotification` (page 155) notification when changes are committed.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `deletedObjects` (page 132)
- `insertedObjects` (page 136)
- `registeredObjects` (page 144)

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 managed object context notifications (NSManagedObjectContextObjectsDidChangeNotification (page 155) and NSManagedObjectContextDidSaveNotification (page 155)).

```
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 Mac OS X v10.4 and later.
Declared in `NSManagedObjectContext.h`.

`NSUpdatedObjectsKey`
Key for the set of objects that were updated.
Available in Mac OS X v10.4 and later.
Declared in `NSManagedObjectContext.h`.

`NSDeletedObjectsKey`
Key for the set of objects that were marked for deletion during the previous event.
Note that the set of deleted objects given for a `NSManagedObjectContextDidSaveNotification` (page 155) does not include objects that were inserted and deleted without an intervening save operation—that is, they had not been saved to a persistent store. See also `deletedObjects` (page 132) (`NSManagedObjectContext`) and `isDeleted` (page 112) (`NSManagedObject`).
Available in Mac OS X v10.4 and later.
Declared in `NSManagedObjectContext.h`.

`NSRefreshedObjectsKey`
Key for the set of objects that were refreshed.

Available in Mac OS X v10.5 and later.

Declared in `NSManagedObjectContext.h`.

`NSInvalidatedObjectsKey`
Key for the set of objects that were invalidated.

Available in Mac OS X v10.5 and later.

Declared in `NSManagedObjectContext.h`.

`NSInvalidatedAllObjectsKey`
Key that specifies that all objects in the context have been invalidated.

Available in Mac OS X v10.5 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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.

`NSValidationObjectErrorKey`	Key for the object that failed to validate for a validation error.
`NSAffectedStoresErrorKey`	The key for stores prompting an error.
`NSAffectedObjectsErrorKey`	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
`@"object"`	The managed object that could not be saved.
`@"snapshot"`	A dictionary of key-value pairs for the properties that represents the managed object context's last saved state for this managed object.
`@"cachedRow"`	A dictionary of key-value pairs for the properties that represents the persistent store's last saved state for this managed object.
`@"databaseRow"`	A dictionary of key-value pairs for the properties that represents the database's current state for this managed object.
`@"newVersion"`	An `NSNumber` object whose value is latest version number of this managed object.
`@"oldVersion"`	As `NSNumber` 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 142), after the changes have been processed, but before it is safe to call save: (page 146) 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.
Declared in	CoreData/NSManagedObjectID.h
Companion guide	Core Data Programming Guide
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass

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 210) (`NSPersistentStoreCoordinator`) and `objectWithID:` (page 140) (`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 158)
Returns the entity description associated with the receiver.

- `isTemporaryID` (page 158)
  Returns a Boolean value that indicates whether the receiver is temporary.
- `persistentStore` (page 159)
  Returns the persistent store that contains the object whose ID is the receiver.
- `URIRepresentation` (page 159)
  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 Mac OS X v10.4 and later.

See Also
`entity` (page 110) (`NSManagedObject`)

Related Sample Code
Core Data HTML Store

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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

See Also
`managedObjectIDForURIRepresentation:` (page 210) (`NSPersistentStoreCoordinator`)
`objectWithID:` (page 140): (`NSManagedObjectContext`)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 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
Related sample code	Core Data HTML Store CoreRecipes

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 an 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 `fetchRequestTemplateForName:` (page 170) and `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169). You can create fetch request templates programmatically, and associate them with a model using `setFetchRequestTemplate:forName:` (page 173); 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 173) or using the Xcode design tool, and retrieve the entities for a given configuration name using `entitiesForConfiguration:` (page 168).

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 Versioning in *Core Data 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, `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 171)
  Initializes the receiver using the model file at the specified URL.
+ `mergedModelFromBundles:` (page 164)
  Returns a model created by merging all the models found in given bundles.
+ `mergedModelFromBundles:forStoreMetadata:` (page 165)
  Returns a merged model from a specified array for the version information in provided metadata.
+ `modelByMergingModels:` (page 166)
  Creates a single model from an array of existing models.
+ `modelByMergingModels:forStoreMetadata:` (page 166)
  Returns, for the version information in given metadata, a model merged from a given array of models.

Entities and Configurations

- `entities` (page 167)
  Returns the entities in the receiver.
- `entitiesByName` (page 168)
  Returns the entities of the receiver in a dictionary.
- `setEntities:` (page 172)
  Sets the entities array of the receiver.
- `configurations` (page 167)
  Returns all the available configuration names of the receiver.
- `entitiesForConfiguration:` (page 168)
  Returns the entities of the receiver for a specified configuration.
- `setEntities:forConfiguration:` (page 173)
  Associates the specified entities with the receiver using the given configuration name.

Getting Fetch Request Templates

- `fetchRequestTemplatesByName` (page 170)
  Returns a dictionary of the receiver's fetch request templates.
- `fetchRequestTemplateForName:` (page 170)
  Returns the fetch request with a specified name.
- `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169)
  Returns a copy of the fetch request template with the variables substituted by values from the substitutions dictionary.
- `setFetchRequestTemplate:forName:` (page 173)
  Associates the specified fetch request with the receiver using the given name.

Localization

- `localizationDictionary` (page 172)
  Returns the localization dictionary of the receiver.
- `setLocalizationDictionary:` (page 174)
  Sets the localization dictionary of the receiver.

Versioning and Migration

- `isConfiguration:compatibleWithStoreMetadata:` (page 171)
  Returns a Boolean value that indicates whether a given configuration in the receiver is compatible with given metadata from a persistent store.
- `entityVersionHashesByName` (page 169)
  Returns a dictionary of the version hashes for the entities in the receiver.
- `versionIdentifiers` (page 175)
  Returns the collection of developer-defined version identifiers for the receiver.
- `setVersionIdentifiers:` (page 175)
  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 Mac OS X v10.4 and later.

See Also
+ mergedModelFromBundles:forStoreMetadata: (page 165)
+ modelByMergingModels: (page 166)
+ modelByMergingModels:forStoreMetadata: (page 166)
- initWithContentsOfURL: (page 171)

Related Sample Code
Core Data HTML Store
CoreRecipes

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 164).

Availability
Available in Mac OS X v10.5 and later.

See Also
+ mergedModelFromBundles: (page 164)
+ modelByMergingModels: (page 166)
+ modelByMergingModels:forStoreMetadata: (page 166)
- initWithContentsOfURL: (page 171)

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 Mac OS X v10.4 and later.

See Also

`+ mergedModelFromBundles:` (page 164)
`+ mergedModelFromBundles:forStoreMetadata:` (page 165)
`+ modelByMergingModels:forStoreMetadata:` (page 166)
`- initWithContentsOfURL:` (page 171)

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 Value

A 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 165).

Availability

Available in Mac OS X v10.5 and later.

See Also

`+ mergedModelFromBundles:` (page 164)
`+ mergedModelFromBundles:forStoreMetadata:` (page 165)

+ modelByMergingModels: (page 166)
- initWithContentsOfURL: (page 171)

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 Mac OS X v10.4 and later.

See Also
- entitiesForConfiguration: (page 168)
- setEntities:forConfiguration: (page 173)

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 Mac OS X v10.4 and later.

See Also
- entitiesByName (page 168)
- entitiesForConfiguration: (page 168)
- setEntities: (page 172)
- setEntities:forConfiguration: (page 173)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- entities (page 167)
- entitiesForConfiguration: (page 168)
- setEntities: (page 172)
- setEntities:forConfiguration: (page 173)
+ entityForName:inManagedObjectContext: (page 40) (NSEntityDescription)

Related Sample Code
Core Data HTML Store
CoreRecipes
CustomAtomicStoreSubclass

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 Mac OS X v10.4 and later.

See Also
- entities (page 167)
- entitiesByName (page 168)
- setEntities: (page 172)
- setEntities:forConfiguration: (page 173)

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 Mac OS X v10.5 and later.

See Also
- isConfiguration:compatibleWithStoreMetadata: (page 171)

Declared In
NSManagedObjectModel.h

fetchRequestFromTemplateWithName:substitutionVariables:

Returns a copy of the fetch request template with the variables substituted by values from the substitutions dictionary.

```
- (NSFetchRequest *)fetchRequestFromTemplateWithName:(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 Mac OS X v10.4 and later.

See Also
- `- fetchRequestTemplatesByName` (page 170)
- `- fetchRequestTemplateForName:` (page 170)
- `- setFetchRequestTemplate:forName:` (page 173)

Declared In
`NSManagedObjectModel.h`

fetchRequestTemplateForName:

Returns the fetch request with a specified name.

```
- (NSFetchRequest *)fetchRequestTemplateForName:(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 `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169) to create a new fetch request.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `- fetchRequestTemplatesByName` (page 170)
- `- fetchRequestFromTemplateWithName:substitutionVariables:` (page 169)
- `- setFetchRequestTemplate:forName:` (page 173)

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 `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169) to create a new fetch request.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `fetchRequestTemplateForName:` (page 170)
- `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169)

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 Mac OS X v10.4 and later.

See Also
- `+ mergedModelFromBundles:` (page 164)
- `+ mergedModelFromBundles:forStoreMetadata:` (page 165)
- `+ modelByMergingModels:` (page 166)
- `+ modelByMergingModels:forStoreMetadata:` (page 166)

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 169) and perform an entity-by-entity comparison.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- entityVersionHashesByName` (page 169)

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 174).

Note that in the implementation in 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 Mac OS X v10.4 and later.

See Also
`- setLocalizationDictionary:` (page 174)

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 Mac OS X v10.4 and later.

See Also
- `entities` (page 167)
- `entitiesByName` (page 168)
- `entitiesForConfiguration:` (page 168)
- `setEntities:forConfiguration:` (page 173)

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 Mac OS X v10.4 and later.

See Also
- `entities` (page 167)
- `entitiesByName` (page 168)
- `entitiesForConfiguration:` (page 168)
- `setEntities:` (page 172)

Declared In
`NSManagedObjectModel.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 Mac OS X v10.4 and later.

See Also

- `fetchRequestTemplatesByName` (page 170)
- `fetchRequestTemplateForName:` (page 170)
- `fetchRequestFromTemplateWithName:substitutionVariables:` (page 169)

Declared In

`NSManagedObjectModel.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 13-1 (page 174).

Discussion

Table 13-1 (page 174) describes the key and value pattern for the localization dictionary.

Table 13-1 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 Mac OS X v10.4 and later.

See Also

- `localizationDictionary` (page 172)

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 Mac OS X v10.5 and later.

See Also
- versionIdentifiers (page 175)

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 Mac OS X v10.5 and later.

See Also
- setVersionIdentifiers: (page 175)

Declared In
NSManagedObjectModel.h

NSMappingModel Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.5 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 178)
Returns the mapping model to translate data from the source to the destination model.

`- initWithContentsOfURL:` (page 179)
Returns a mapping model initialized from a given URL.

Managing Entity Mappings

`- entityMappings` (page 178)
Returns the collection of entity mappings for the receiver.

`- setEntityMappings:` (page 180)
Sets the collection of entity mappings for the receiver

`- entityMappingsByName` (page 179)
Returns a dictionary of the entity mappings for the receiver.

Class Methods

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 164) and `mergedModelFromBundles:forStoreMetadata:` (page 165) 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 Mac OS X v10.5 and later.

See Also

`- initWithContentsOfURL:` (page 179)

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 Mac OS X v10.5 and later.

See Also
- `setEntityMappings:` (page 180)
- `entityMappingsByName` (page 179)

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 178).

Availability
Available in Mac OS X v10.5 and later.

See Also
- `entityMappings` (page 178)

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 Mac OS X v10.5 and later.

See Also
+ `mappingModelFromBundles:forSourceModel:destinationModel:` (page 178)

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 Mac OS X v10.5 and later.

See Also

- `entityMappings` (page 178)

Declared In

`NSMappingModel.h`

NSMigrationManager Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.5 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 186)
Initializes a migration manager instance with given source and destination models.

`- setUserInfo:` (page 189)
Sets the user info for the receiver.

Performing Migration Operations

`- migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` (page 187)
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.

`- reset` (page 188)
Resets the association tables for the migration.

`- cancelMigrationWithError:` (page 183)
Cancels the migration with a given error.

Monitoring Migration Progress

- `migrationProgress` (page 188)
  Returns a number from `0` to `1` that indicates the proportion of completeness of the migration.
- `currentEntityMapping` (page 184)
  Returns the entity mapping currently being processed.

Working with Source and Destination Instances

- `associateSourceInstance:withDestinationInstance:forEntityMapping:` (page 182)
  Associates a given source instance with an array of destination instances for a given property mapping.
- `destinationInstancesForEntityMappingNamed:sourceInstances:` (page 185)
  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 190)
  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 187)
  Returns the mapping model for the receiver.
- `sourceModel` (page 191)
  Returns the source model for the receiver.
- `destinationModel` (page 185)
  Returns the destination model for the receiver.
- `sourceEntityForEntityMapping:` (page 189)
  Returns the entity description for the source entity of a given entity mapping.
- `destinationEntityForEntityMapping:` (page 184)
  Returns the entity description for the destination entity of a given entity mapping.
- `sourceContext` (page 189)
  Returns the managed object context the receiver uses for reading the source persistent store.
- `destinationContext` (page 184)
  Returns the managed object context the receiver uses for writing the destination persistent store.
- `userInfo` (page 191)
  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 Mac OS X v10.5 and later.

See Also

- `sourceInstancesForEntityMappingNamed:destinationInstances:` (page 190)
- `destinationInstancesForEntityMappingNamed:sourceInstances:` (page 185)

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 187) to abort the migration and return *error*—you should provide an appropriate error to indicate the reason for the cancellation.

Availability

Available in Mac OS X v10.5 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 Mac OS X v10.5 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 lazily as part of the initialization of the Core Data stacks used for migration.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- sourceContext` (page 189)

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 Mac OS X v10.5 and later.

See Also
- `sourceEntityForEntityMapping:` (page 189)

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 Mac OS X v10.5 and later.

See Also
- `sourceInstancesForEntityMappingNamed:destinationInstances:` (page 190)

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 Mac OS X v10.5 and later.

See Also
- `mappingModel` (page 187)
- `sourceModel` (page 191)
- `initWithSourceModel:destinationModel:` (page 186)

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 187).

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 187), as with `NSPersistentStoreCoordinator` once models are added to the migration manager they are immutable and cannot be altered.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` (page 187)
- `mappingModel` (page 187)
- `sourceModel` (page 191)
- `destinationModel` (page 185)

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 Mac OS X v10.5 and later.

See Also
- `sourceModel` (page 191)
- `destinationModel` (page 185)
- `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` (page 187)

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 Mac OS X v10.5 and later.

See Also
`- cancelMigrationWithError:` (page 183)

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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
- userInfo (page 191)

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 lazily as part of the initialization of the Core Data stacks used for migration.

Availability
Available in Mac OS X v10.5 and later.

See Also
- destinationContext (page 184)

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 Value

The 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 Mac OS X v10.5 and later.

See Also

- destinationEntityForEntityMapping: (page 184)

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 Value

An 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 Mac OS X v10.5 and later.

See Also

- destinationInstancesForEntityMappingNamed:sourceInstances: (page 185)

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 Mac OS X v10.5 and later.

See Also
- `mappingModel` (page 187)
- `destinationModel` (page 185)
- `initWithSourceModel:destinationModel:` (page 186)

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 Mac OS X v10.5 and later.

See Also
- `setUserInfo:` (page 189)

Declared In
`NSMigrationManager.h`

NSPersistentStore Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Declared in	NSPersistentStore.h
Availability	Available in Mac OS X v10.5 and later.
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; 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 197). When you implement the initializer, you must ensure you load metadata during initialization and set it using `setMetadata:` (page 200).

You must override these methods:

- `type` (page 201)
- `metadata` (page 198)
- `metadataForPersistentStoreWithURL:error:` (page 195)
- `setMetadata:forPersistentStoreWithURL:error:` (page 195)

Tasks

Initializing a Persistent Store

- `initWithPersistentStoreCoordinator:configurationName:URL:options:` (page 197)
  Returns a store initialized with the given arguments.

Working with State Information

- `type` (page 201)
  Returns the type string of the receiver.
- `persistentStoreCoordinator` (page 199)
  Returns the persistent store coordinator which loaded the receiver.
- `configurationName` (page 196)
  Returns the name of the managed object model configuration used to create the receiver.
- `options` (page 198)
  Returns the options with which the receiver was created.
- `URL` (page 201)
  Returns the URL for the receiver.
- `setURL:` (page 200)
  Sets the URL for the receiver.
- `identifier` (page 197)
  Returns the unique identifier for the receiver.
- `setIdentifier:` (page 199)
  Sets the unique identifier for the receiver.
- `isReadOnly` (page 198)
  Returns a Boolean value that indicates whether the receiver is read-only.
- `setReadOnly:` (page 200)
  Sets whether the receiver is read-only.

Managing Metadata

+ `metadataForPersistentStoreWithURL:error:` (page 195)
  Returns the metadata from the persistent store at the given URL.
+ `setMetadata:forPersistentStoreWithURL:error:` (page 195)
  Sets the metadata for the store at a given URL.
- `metadata` (page 198)
  Returns the metadata for the receiver.
- `setMetadata:` (page 200)
  Sets the metadata for the receiver.

Setup and Teardown

`- didAddToPersistentStoreCoordinator:` (page 196)
Invoked after the receiver has been added to the persistent store coordinator.

`- willRemoveFromPersistentStoreCoordinator:` (page 201)
Invoked before the receiver is removed from the persistent store coordinator.

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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

See Also
- `setIdentifier:` (page 199)
- `setMetadata:` (page 200)

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.

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 the managed object model configuration *configurationName*.

Discussion
You must ensure that you load metadata during initialization and set it using `setMetadata:` (page 200).

Special Considerations

This is the designated initializer for persistent stores.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- setMetadata:` (page 200)

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 Mac OS X v10.5 and later.

Related Sample Code
CustomAtomicStoreSubclass

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 Mac OS X v10.5 and later.

Related Sample Code
Core Data HTML Store

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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Related Sample Code
CustomAtomicStoreSubclass

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 Mac OS X v10.5 and later.

See Also
- `identifier` (page 197)
- `metadata` (page 198)

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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared In
NSPersistentStore.h

setURL:

Sets the URL for the receiver.

```
- (void)setURL:(NSURL *)url
```

Parameters

url

The URL for the receiver.

Discussion
To alter the location of a store, send the persistent store coordinator a `setURL:forPersistentStore:` (page 215) message.

Availability
Available in Mac OS X v10.5 and later.

See Also
– URL (page 201)

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 Mac OS X v10.5 and later.

Related Sample Code
Core Data HTML Store

Declared In
NSPersistentStore.h

URL

Returns the URL for the receiver.

```
- (NSURL *)URL
```

Return Value
The URL for the receiver.

Availability
Available in Mac OS X v10.5 and later.

See Also
- setURL: (page 200)

Related Sample Code
CustomAtomicStoreSubclass

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 Mac OS X v10.5 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 Mac OS X v10.4 and later.
Declared in	CoreData/NSPersistentStoreCoordinator.h
Companion guides	Core Data Programming Guide Atomic Store Programming Topics Core Data Utility Tutorial
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass Departments and Employees File Wrappers with Core Data Documents

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 209)), or by using `persistentStoreForURL:` (page 213) or `persistentStores` (page 213). This allows you to 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 212).
- You can set metadata for a given store using the persistent store coordinator (`setMetadata:forPersistentStore:` (page 214)).

For more details about these tasks, see Persistent Stores.

Tasks

Registered Store Types

`+ registeredStoreTypes` (page 207)
Returns a dictionary of the registered store types.

`+ registerStoreClass:forStoreType:` (page 207)
Registers a given `NSPersistentStore` subclass for a given store type string.

Initializing

`- initWithManagedObjectModel:` (page 209)
Initializes the receiver with a managed object model.

`- managedObjectModel` (page 211)
Returns the receiver's managed object model.

Configuring Persistent Stores

`- addPersistentStoreWithType:configuration:URL:options:error:` (page 209)
Adds a new persistent store of a specified type at a given location, and returns the new store.

`- setURL:forPersistentStore:` (page 215)
Sets the URL for a given persistent store.

`- removePersistentStore:error:` (page 214)
Removes a given persistent store.

`- migratePersistentStore:toURL:options:withType:error:` (page 212)
Moves a persistent store to a new location, changing the storage type if necessary.

`- persistentStores` (page 213)
Returns an array of persistent stores associated with the receiver.

`- persistentStoreForURL:` (page 213)
Returns the persistent store for the specified URL.

- `URLForPersistentStore:` (page 216)
  Returns the URL for a given persistent store.

Locking

- `lock` (page 210)
  Attempts to acquire a lock.
- `tryLock` (page 216)
  Attempts to acquire a lock.
- `unlock` (page 216)
  Relinquishes a previously acquired lock.

Working with Metadata

- `metadataForPersistentStore:` (page 211)
  Returns a dictionary that contains the metadata currently stored or to-be-stored in a given persistent store.
- `setMetadata:forPersistentStore:` (page 214)
  Sets the metadata stored in the persistent store during the next save operation executed on it to *`metadata`*.
+ `setMetadata:forPersistentStoreOfType:URL:error:` (page 208)
  Sets the metadata for a given store.
+ `metadataForPersistentStoreOfType:URL:error:` (page 205)
  Returns a dictionary containing the metadata stored in the persistent store at a given URL.
+ `metadataForPersistentStoreWithURL:error:` (page 206) Deprecated in Mac OS X v10.5
  Returns a dictionary that contains the metadata stored in the persistent store at the specified location. (Deprecated. Use `metadataForPersistentStoreOfType:URL:error:` (page 205) instead.)

Discovering Object IDs

- `managedObjectIDForURIRepresentation:` (page 210)
  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 `NSStoreTypeKey` and `NSStoreUUIDKey`.

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 Mac OS X v10.5 and later.

See Also

`+ setMetadata:forPersistentStoreOfType:URL:error:` (page 208)

`- metadataForPersistentStore:` (page 211)

`- setMetadata:forPersistentStore:` (page 214)

Declared In

`NSPersistentStoreCoordinator.h`

metadataForPersistentStoreWithURL:error:

Returns a dictionary that contains the metadata stored in the persistent store at the specified location. (Deprecated in Mac OS X v10.5. Use `metadataForPersistentStoreOfType:URL:error:` (page 205) instead.)

```
+ (NSDictionary *)metadataForPersistentStoreWithURL:(NSURL *)url
    error:(NSError **)error
```

Parameters

url

An URL object that specifies 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 instance of `NSError` that describes the problem.

Return Value

A dictionary containing the metadata for the persistent store at *url*. If no store is found, or there is a problem accessing its contents, returns `nil`.

The keys guaranteed to be in this dictionary are `NSStoreTypeKey` and `NSStoreUUIDKey`.

Discussion
This method allows you to access the metadata in a persistent store without initializing a Core Data stack.

Availability
Available in Mac OS X v10.4 and later.
Deprecated in Mac OS X v10.5.

See Also
- `- metadataForPersistentStore:` (page 211)
- `- setMetadata:forPersistentStore:` (page 214)
- `+ metadataForPersistentStoreOfType:URL:error:` (page 205)
- `+ setMetadata:forPersistentStoreOfType:URL:error:` (page 208)

Related Sample Code
CoreRecipes

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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Related Sample Code
Core Data HTML Store

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 Mac OS X v10.5 and later.

See Also
`+ metadataForPersistentStoreOfType:URL:error:` (page 205)
`- metadataForPersistentStore:` (page 211)
`- setMetadata:forPersistentStore:` (page 214)

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 217) 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 `"Migration Options"` (page 219). 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 Mac OS X v10.4 and later.

See Also

`- migratePersistentStore:toURL:options:withType:error:` (page 212)

`- removePersistentStore:error:` (page 214)

Related Sample Code

Core Data HTML Store

CoreRecipes

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 Mac OS X v10.4 and later.

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- tryLock (page 216)
- unlock (page 216)

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 Mac OS X v10.4 and later.

See Also
`URIRepresentation` (page 159) (`NSManagedObjectID`)
`objectWithID:` (page 140) (`NSManagedObjectContext`)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

Related Sample Code
Core Data HTML Store
CoreRecipes
CustomAtomicStoreSubclass

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 Mac OS X v10.4 and later.

See Also
- `- setMetadata:forPersistentStore:` (page 214)
- `+ metadataForPersistentStoreOfType:URL:error:` (page 205)
- `+ setMetadata:forPersistentStoreOfType:URL:error:` (page 208)

Related Sample Code
CoreRecipes
Departments and Employees

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 219).

storeType
A string constant (such as `NSSQLiteStoreType`) that specifies the type of the new store—see "Store Types" (page 217).

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 Stores.

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 Mac OS X v10.4 and later.

See Also
- `addPersistentStoreWithType:configuration:URL:options:error:` (page 209)
- `removePersistentStore:error:` (page 214)

Related Sample Code
File Wrappers with Core Data Documents

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 Mac OS X v10.4 and later.

See Also
- `persistentStores` (page 213)
- `URLForPersistentStore:` (page 216)

Related Sample Code
CoreRecipes
Departments and Employees
File Wrappers with Core Data Documents

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 Mac OS X v10.4 and later.

See Also
- `persistentStoreForURL:` (page 213)

- URLForPersistentStore: (page 216)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- addPersistentStoreWithType:configuration:URL:options:error: (page 209)
- migratePersistentStore:toURL:options:withType:error: (page 212)

Related Sample Code
CoreRecipes
File Wrappers with Core Data Documents

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 Mac OS X v10.4 and later.

See Also
- `- metadataForPersistentStore:` (page 211)
- `+ setMetadata:forPersistentStoreOfType:URL:error:` (page 208)
- `+ metadataForPersistentStoreOfType:URL:error:` (page 205)

Related Sample Code
CoreRecipes
Departments and Employees

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 Mac OS X v10.5 and later.

Related Sample Code
File Wrappers with Core Data Documents

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 210) which blocks.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `lock` (page 210)
- `unlock` (page 216)

Declared In
`NSPersistentStoreCoordinator.h`

unlock

Relinquishes a previously acquired lock.

```
- (void)unlock
```

Availability
Available in Mac OS X v10.4 and later.

See Also
- `lock` (page 210)
- `tryLock` (page 216)

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 Mac OS X v10.4 and later.

See Also
- `persistentStoreForURL:` (page 213)
- `persistentStores` (page 213)

Related Sample Code
CoreRecipes

Declared In
`NSPersistentStoreCoordinator.h`

Constants

Store Types

Types of persistent store.

```
NSString * const NSSQLiteStoreType;
NSString * const NSXMLStoreType;
NSString * const NSBinaryStoreType;
NSString * const NSInMemoryStoreType;
```

Constants

`NSSQLiteStoreType`
The SQLite database store type.
Available in Mac OS X v10.4 and later.
Declared in `NSPersistentStoreCoordinator.h`.

`NSXMLStoreType`
The XML store type.
Available in Mac OS X v10.4 and later.
Declared in `NSPersistentStoreCoordinator.h`.

`NSBinaryStoreType`
The binary store type.
Available in Mac OS X v10.4 and later.
Declared in `NSPersistentStoreCoordinator.h`.

`NSInMemoryStoreType`
The in-memory store type.
Available in Mac OS X v10.4 and later.
Declared in `NSPersistentStoreCoordinator.h`.

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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Declared In

`NSPersistentStoreCoordinator.h`

Stores Change Notification User Info Keys

An `NSPersistentStoreCoordinatorStoresDidChangeNotification` 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 Mac OS X v10.4 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSRemovedPersistentStoresKey`

Key for the array of stores that were removed.

Available in Mac OS X v10.4 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSUUIDChangedPersistentStoresKey`

Key for the array of stores whose UUIDs changed.

Available in Mac OS X v10.4 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 209) and `migratePersistentStore:toURL:options:withType:error:` (page 212).

```
NSString * const NSReadOnlyPersistentStoreOption;
NSString * const NSValidateXMLStoreOption;
NSString * const NSPersistentStoreTimeoutOption;
NSString * const NSSQLitePragmasOption;
```

Constants

`NSReadOnlyPersistentStoreOption`

A flag that indicates whether a store is treated as read-only or not.

The default value is `NO`.

Available in Mac OS X v10.4 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSValidateXMLStoreOption`

A flag that indicates whether an XML file should be validated with the DTD while opening.

The default value is `NO`.

Available in Mac OS X v10.4 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `NSPersistentStoreCoordinator.h`.

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 209).

```
NSString * const NSIgnorePersistentStoreVersioningOption;
NSString * const NSMigratePersistentStoresAutomaticallyOption;
```

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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `NSPersistentStoreCoordinator.h`.

Declared In

`NSPersistentStoreCoordinator.h`

Versioning Support

Keys in store metadata to support versioning.

```
NSString * const NSStoreModelVersionHashesKey;
NSString * const NSStoreModelVersionIdentifiersKey;
NSString * const NSPersistentStoreOSCompatibility;
```

Constants

`NSStoreModelVersionHashesKey`

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 Mac OS X v10.5 and later.

Declared in `NSPersistentStoreCoordinator.h`.

`NSStoreModelVersionIdentifiersKey`

Key to represent the version identifier for the model used to create the store.

This key is used in the metadata for a persistent store.

Available in Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `NSPersistentStoreCoordinator.h`.

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`	An array of stores that were added.
`NSRemovedPersistentStoresKey`	An array of stores that were removed.
`NSUUIDChangedPersistentStoresKey`	An array of stores whose UUIDs changed.

Availability
Available in Mac OS X v10.4 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 Mac OS X v10.4 and later.
Declared in	CoreData/NSPropertyDescription.h
Companion guide	Core Data Programming Guide
Related sample code	Core Data HTML Store

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 225)
  Returns the entity description of the receiver.
- `isIndexed` (page 225)
  Returns a Boolean value that indicates whether the receiver is important for searching.
- `isOptional` (page 226)
  Returns a Boolean value that indicates whether the receiver is optional.
- `isTransient` (page 226)
  Returns a Boolean value that indicates whether the receiver is transient.
- `name` (page 226)
  Returns the name of the receiver.
- `userInfo` (page 230)
  Returns the user info dictionary of the receiver.

Setting Features of a Property

- `setIndexed:` (page 227)
  Sets the optionality flag of the receiver.
- `setName:` (page 227)
  Sets the name of the receiver.
- `setOptional:` (page 228)
  Sets the optionality flag of the receiver.
- `setTransient:` (page 228)
  Sets the transient flag of the receiver.
- `setUserInfo:` (page 229)
  Sets the user info dictionary of the receiver.

Validation

- `validationPredicates` (page 231)
  Returns the validation predicates of the receiver.
- `validationWarnings` (page 231)
  Returns the error strings associated with the receiver's validation predicates.

- `setValidationPredicates:withValidationWarnings:` (page 229)
  Sets the validation predicates and warnings of the receiver.

Versioning Support

- `versionHash` (page 231)
  Returns the version hash for the receiver.
- `versionHashModifier` (page 232)
  Returns the version hash modifier for the receiver.
- `setVersionHashModifier:` (page 230)
  Sets the version hash modifier 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 Mac OS X v10.4 and later.

See Also
`setProperties:` (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 Mac OS X v10.5 and later.

See Also
- setIndexed: (page 227)

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 Mac OS X v10.4 and later.

See Also
- setOptional: (page 228)

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 Mac OS X v10.4 and later.

See Also
- setTransient: (page 228)

Declared In
NSPropertyDescription.h

name

Returns the name of the receiver.

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

Return Value
The name of the receiver.

Availability
Available in Mac OS X v10.4 and later.

See Also
- `setName:` (page 227)

Related Sample Code
CoreRecipes

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 Mac OS X v10.5 and later.

See Also
- `isIndexed` (page 225)

Declared In
`NSPropertyDescription.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 `NSManagedObject`. 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 Mac OS X v10.4 and later.

See Also
- `name` (page 226)

Declared In
`NSPropertyDescription.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 Mac OS X v10.4 and later.

See Also
- `isOptional` (page 226)

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 Mac OS X v10.4 and later.

See Also
- `isTransient` (page 226)

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 Mac OS X v10.4 and later.

See Also
- `userInfo` (page 230)

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 Mac OS X v10.4 and later.

See Also
- `validationPredicates` (page 231)
- `validationWarnings` (page 231)

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
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.)

Availability
Available in Mac OS X v10.5 and later.

See Also
- `versionHash` (page 231)
- `versionHashModifier` (page 232)

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 Mac OS X v10.4 and later.

See Also
- `setUserInfo:` (page 229)

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 Mac OS X v10.4 and later.

See Also
- `validationWarnings` (page 231)
- `setValidationPredicates:withValidationWarnings:` (page 229)

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 Mac OS X v10.4 and later.

See Also
- `validationPredicates` (page 231)
- `setValidationPredicates:withValidationWarnings:` (page 229)

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 232) 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 Mac OS X v10.5 and later.

See Also
- `versionHashModifier` (page 232)
- `setVersionHashModifier:` (page 230)

Declared In
`NSPropertyDescription.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. See `setVersionHashModifier:` (page 230) for a full discussion.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `versionHash` (page 231)
- `setVersionHashModifier:` (page 230)

Declared In
`NSPropertyDescription.h`

NSPropertyMapping Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.5 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 234)
  Returns the name of the property in the destination entity for the receiver.
- `setName:` (page 234)
  Sets the name of the property in the destination entity for the receiver.
- `valueExpression` (page 235)
  Returns the value expression for the receiver.
- `setValueExpression:` (page 235)
  Sets the value expression for the receiver.
- `userInfo` (page 235)
  Returns the user info for the receiver.
- `setUserInfo:` (page 234)
  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 Mac OS X v10.5 and later.

See Also
- setName: (page 234)

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 Mac OS X v10.5 and later.

See Also
- name (page 234)

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 Mac OS X v10.5 and later.

See Also
- `userInfo` (page 235)

Declared In
`NSPropertyMapping.h`

setValueExpression:

Sets the value expression for the receiver.

```
- (void)setValueExpression:(NSExpression *)expression
```

Parameters

expression
The the value expression for the receiver.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `setValueExpression:` (page 235)

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 Mac OS X v10.5 and later.

See Also
- `setUserInfo:` (page 234)

Declared In
`NSPropertyMapping.h`

valueExpression

Returns the value expression for the receiver.

```
- (NSExpression *)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 Mac OS X v10.5 and later.

See Also
`- setValueExpression:` (page 235)

Declared In
`NSPropertyMapping.h`

NSRelationshipDescription Class Reference

Inherits from	NSPropertyDescription : NSObject
Conforms to	NSCoding (NSPropertyDescription) NSCopying (NSPropertyDescription) NSObject (NSObject)
Framework	/System/Library/Frameworks/CoreData.framework
Availability	Available in Mac OS X v10.4 and later.
Declared in	CoreData/NSRelationshipDescription.h
Companion guide	Core Data Programming Guide
Related sample code	Core Data HTML Store CoreRecipes CustomAtomicStoreSubclass

Overview

The `NSRelationshipDescription` class is used to describe relationships of an entity in an `NSEntityDescription` object.

`NSRelationshipDescription` extends `NSPropertyDescription` 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). Note that 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

Getting and Setting Type Information

- `destinationEntity` (page 239)
  Returns the entity description of the receiver's destination.
- `setDestinationEntity:` (page 242)
  Sets the entity description for the receiver's destination.
- `inverseRelationship` (page 239)
  Returns the relationship that represents the inverse of the receiver.
- `setInverseRelationship:` (page 242)
  Sets the inverse relationship of the receiver.

Getting and Setting Delete Rules

- `deleteRule` (page 239)
  Returns the delete rule of the receiver.
- `setDeleteRule:` (page 241)
  Sets the delete rule of the receiver.

Cardinality

- `maxCount` (page 240)
  Returns the maximum count of the receiver.
- `setMaxCount:` (page 242)
  Sets the maximum count of the receiver.
- `minCount` (page 241)
  Returns the minimum count of the receiver.
- `setMinCount:` (page 243)
  Sets the minimum count of the receiver.
- `isToMany` (page 240)
  Returns a Boolean value that indicates whether the receiver represents a to-many relationship.

Versioning Support

- `versionHash` (page 243)
  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 Mac OS X v10.4 and later.

See Also
- setDeleteRule: (page 241)

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 Mac OS X v10.4 and later.

See Also
- setDestinationEntity: (page 242)

Related Sample Code
Core Data HTML Store
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `setInverseRelationship:` (page 242)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `maxCount` (page 240)
- `setMaxCount:` (page 242)

Related Sample Code
CoreRecipes

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 Mac OS X v10.4 and later.

See Also
- `isToMany` (page 240)
- `minCount` (page 241)
- `setMaxCount:` (page 242)
- `setMinCount:` (page 243)

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 Mac OS X v10.4 and later.

See Also
- `maxCount` (page 240)
- `setMaxCount:` (page 242)
- `setMinCount:` (page 243)

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 Mac OS X v10.4 and later.

See Also
- `deleteRule` (page 239)

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 Mac OS X v10.4 and later.

See Also
- destinationEntity (page 239)

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 Mac OS X v10.4 and later.

See Also
- inverseRelationship (page 239)

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 Mac OS X v10.4 and later.

See Also
- `isToMany` (page 240)
- `maxCount` (page 240)
- `minCount` (page 241)
- `setMinCount:` (page 243)

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 Mac OS X v10.4 and later.

See Also
- `maxCount` (page 240)
- `minCount` (page 241)
- `setMaxCount:` (page 242)

Declared In
`NSRelationshipDescription.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 231) information from `NSPropertyDescription`, the name of the destination entity and the inverse relationship, and the min and max count.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `versionHash` (page 231) (`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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

Declared in `NSRelationshipDescription.h`.

`NSDenyDeleteRule`
If the destination of this relationship is not `nil`, the delete creates a validation error.

Available in Mac OS X v10.4 and later.

Declared in `NSRelationshipDescription.h`.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`NSRelationshipDescription.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 Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationObjectErrorKey`

Key for the object that failed to validate for a validation error.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationKeyErrorKey`

Key for the key that failed to validate for a validation error.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationPredicateErrorKey`
: For predicate-based validation, key for the predicate for the condition that failed to validate.

  Available in Mac OS X v10.4 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 Mac OS X v10.4 and later.

  Declared in `CoreDataErrors.h`.

`NSAffectedStoresErrorKey`
: The key for stores prompting an error.

  Available in Mac OS X v10.4 and later.

  Declared in `CoreDataErrors.h`.

`NSAffectedObjectsErrorKey`
: The key for objects prompting an error.

  Available in Mac OS X v10.4 and later.

  Declared in `CoreDataErrors.h`.

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 Mac OS X v10.5 and later.

  Declared in `CoreDataErrors.h`.

Declared In
`CoreDataErrors.h`

Validation Error Codes

Error codes related to validation.

```
NSManagedObjectValidationError                  = 1550,
NSValidationMultipleErrorsError                 = 1560,
NSValidationMissingMandatoryPropertyError       = 1570,
NSValidationRelationshipLacksMinimumCountError  = 1580,
NSValidationRelationshipExceedsMaximumCountError = 1590,
NSValidationRelationshipDeniedDeleteError       = 1600,
NSValidationNumberTooLargeError                 = 1610,
NSValidationNumberTooSmallError                 = 1620,
NSValidationDateTooLateError                    = 1630,
NSValidationDateTooSoonError                    = 1640,
NSValidationInvalidDateError                    = 1650,
NSValidationStringTooLongError                  = 1660,
NSValidationStringTooShortError                 = 1670,
NSValidationStringPatternMatchingError          = 1680,
```

Constants

`NSManagedObjectValidationError`

Error code to denote a generic validation error.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationMultipleErrorsError`

Error code to denote an error containing multiple validation errors.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationMissingMandatoryPropertyError`

Error code for a non-optional property with a nil value.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipLacksMinimumCountError`

Error code to denote a to-many relationship with too few destination objects.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipExceedsMaximumCountError`

Error code to denote a bounded to-many relationship with too many destination objects.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationRelationshipDeniedDeleteError`

Error code to denote some relationship with delete rule `NSDeleteRuleDeny` is non-empty.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationNumberTooLargeError`

Error code to denote some numerical value is too large.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationNumberTooSmallError`
Error code to denote some numerical value is too small.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationDateTooLateError`
Error code to denote some date value is too late.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationDateTooSoonError`
Error code to denote some date value is too soon.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationInvalidDateError`
Error code to denote some date value fails to match date pattern.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringTooLongError`
Error code to denote some string value is too long.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringTooShortError`
Error code to denote some string value is too short.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSValidationStringPatternMatchingError`
Error code to denote some string value fails to match some pattern.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

Discussion
For additional error codes, including `NSValidationErrorMinimum` and `NSValidationErrorMaximum`, see `NSError`.

Declared In
`CoreDataErrors.h`

Object Graph Management Error Codes

These error codes specify Core Data errors related to object graph management.

```
NSManagedObjectContextLockingError              = 132000,
NSPersistentStoreCoordinatorLockingError        = 132010,
NSManagedObjectReferentialIntegrityError        = 133000,
NSManagedObjectExternalRelationshipError        = 133010,
NSManagedObjectMergeError                       = 133020,
```

Constants

`NSManagedObjectContextLockingError`

Error code to denote an inability to acquire a lock in a managed object context.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreCoordinatorLockingError`

Error code to denote an inability to acquire a lock in a persistent store.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectReferentialIntegrityError`

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 Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectExternalRelationshipError`

Error code to denote that an object being saved has a relationship containing an object from another store.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSManagedObjectMergeError`

Error code to denote that a merge policy failed—Core Data is unable to complete merging.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

Declared In

`CoreDataErrors.h`

Persistent Store Error Codes

Error codes related to persistent stores.

```
NSPersistentStoreInvalidTypeError               = 134000,
NSPersistentStoreTypeMismatchError              = 134010,
NSPersistentStoreIncompatibleSchemaError        = 134020,
NSPersistentStoreSaveError                      = 134030,
NSPersistentStoreIncompleteSaveError            = 134040,
NSPersistentStoreOperationError                 = 134070,
NSPersistentStoreOpenError                      = 134080,
NSPersistentStoreTimeoutError                   = 134090,
NSPersistentStoreIncompatibleVersionHashError   = 134100,
```

Constants

`NSPersistentStoreInvalidTypeError`

Error code to denote an unknown persistent store type/format/version.

Available in Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 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 Mac OS X v10.4 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreOperationError`

Error code to denote that a persistent store operation failed.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSPersistentStoreOpenError`

Error code to denote an error occurred while attempting to open a persistent store.

Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

Declared In
`CoreDataErrors.h`

Migration Error Codes

Error codes related to store migration.

```
NSMigrationError                                = 134110,
NSMigrationCancelledError                       = 134120,
NSMigrationMissingSourceModelError              = 134130,
NSMigrationMissingMappingModelError             = 134140,
NSMigrationManagerSourceStoreError              = 134150,
NSMigrationManagerDestinationStoreError         = 134160,
NSEntityMigrationPolicyError                    = 134170,
```

Constants
`NSMigrationError`
Error code to denote a general migration error.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationCancelledError`
Error code to denote that migration failed due to manual cancellation.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationMissingSourceModelError`
Error code to denote that migration failed due to a missing source data model.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSMigrationMissingMappingModelError`
Error code to denote that migration failed due to a missing mapping model.

Available in Mac OS X v10.5 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 Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSEntityMigrationPolicyError`
Error code to denote that migration failed during processing of an entity migration policy.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

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 Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

`NSSQLiteError`
Error code to denote a general SQLite error.

Available in Mac OS X v10.5 and later.

Declared in `CoreDataErrors.h`.

Declared In
`CoreDataErrors.h`

Core Data Version Number

Specifies the current Core Data version number.

```
COREDATA_EXTERN double NSCoreDataVersionNumber;
```

Constants
`NSCoreDataVersionNumber`
Specifies the version of Core Data available in the current process.

Available in Mac OS X v10.4 and later.

Declared in `CoreDataDefines.h`.

Discussion
See "Core Data Version Numbers" (page 255) for defined versions.

Declared In
`CoreDataDefines.h`

Core Data Version Numbers

Specify Core Data version numbers.

```
#define NSCoreDataVersionNumber10_4       46.0
#define NSCoreDataVersionNumber10_4_3     77.0
```

Constants

`NSCoreDataVersionNumber10_4`

Specifies the Core Data version number released with Mac OS X v10.4.0.

Available in Mac OS X v10.5 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 Mac OS X v10.5 and later.

Declared in `CoreDataDefines.h`.

Discussion

See "Core Data Version Number" (page 254) for the current version.

Declared In

`CoreDataDefines.h`

Document Revision History

This table describes the changes to *Core Data Framework Reference*.

Date	Notes
2007-07-24	Updated for Mac OS X v10.5.
2006-05-23	First publication of this content as a collection of separate documents.

Document Revision History

Index

Symbols

@"cachedRow" constant 154
@"databaseRow" constant 154
@"newVersion" constant 154
@"object" constant 154
@"oldVersion" constant 154
@"snapshot" constant 154

A

addCacheNodes: instance method 15
addPersistentStoreWithType:configuration:URL: options:error: instance method 209
affectedStores instance method 83
assignObject:toPersistentStore: instance method 129
associateSourceInstance:withDestinationInstance: forEntityMapping: instance method 182
attributeMappings instance method 57
attributesByName instance method 42
attributeType instance method 31
attributeValueClassName instance method 31
automaticallyNotifiesObserversForKey: class method 105
awakeFromFetch instance method 105
awakeFromInsert instance method 106

B

beginEntityMapping:manager:error: instance method 70

C

cacheNodeForObjectID: instance method 16
cacheNodes instance method 16
cancelMigrationWithError: instance method 183
changedValues instance method 107
commitEditing instance method 130
commitEditingWithDelegate:didCommitSelector: contextInfo: instance method 131
committedValuesForKeys: instance method 107
configurationName instance method 196
configurations instance method 167
contextExpression instance method 96
copy instance method 43
Core Data Version Number 254
Core Data Version Numbers 255
countForFetchRequest:error: instance method 132
createDestinationInstancesForSourceInstance: entityMapping:manager:error: instance method 70
createRelationshipsForDestinationInstance: entityMapping:manager:error: instance method 71
currentEntityMapping instance method 184

D

dealloc instance method 108
defaultValue instance method 31
deletedObjects instance method 132
deleteObject: instance method 133
deleteRule instance method 239
destinationContext instance method 184
destinationEntity instance method 239
destinationEntityForEntityMapping: instance method 184
destinationEntityName instance method 57
destinationEntityVersionHash instance method 58
destinationInstancesForEntityMappingNamed: sourceInstances: instance method 185
destinationModel instance method 185
detectConflictsForObject: instance method 134
didAccessValueForKey: instance method 108

didAddToPersistentStoreCoordinator: instance method 196
didSave instance method 109
didTurnIntoFault instance method 109
discardEditing instance method 134

E

endEntityMapping:manager:error: instance method 72
endInstanceCreationForEntityMapping:manager:error: instance method 73
endRelationshipCreationForEntityMapping:manager:error: instance method 73
entities instance method 167
entitiesByName instance method 168
entitiesForConfiguration: instance method 168
entity instance method 84, 110, 158, 225
Entity Mapping Types 66
entityForName:inManagedObjectContext: class method 40
entityMappings instance method 178
entityMappingsByName instance method 179
entityMigrationPolicyClassName instance method 58
entityVersionHashesByName instance method 169
Error Domain 248
Error User Info Keys 247
executeFetchRequest:error: instance method 134
expressionForFetch:context:countOnly: class method 96

F

Fetch request expression type 97
Fetch request result types 93
fetchLimit instance method 84
fetchRequest instance method 78
fetchRequestFromTemplateWithName:substitutionVariables: instance method 169
fetchRequestTemplateForName: instance method 170
fetchRequestTemplatesByName instance method 170

G

General Error Codes 254

H

hasChanges instance method 135
hasFaultForRelationshipNamed: instance method 110

I

identifier instance method 197
includesPropertyValues instance method 84
includesSubentities instance method 85
initWithContentsOfURL: instance method 171, 179
initWithEntity:insertIntoManagedObjectContext: instance method 111
initWithManagedObjectModel: instance method 209
initWithObjectID: instance method 26
initWithPersistentStoreCoordinator:configurationName:URL:options: instance method 16, 197
initWithSourceModel:destinationModel: instance method 186
insertedObjects instance method 136
insertNewObjectForEntityForName:inManagedObjectContext: class method 41
insertObject: instance method 136
inverseRelationship instance method 239
isAbstract instance method 43
isConfiguration:compatibleWithStoreMetadata: instance method 171
isCountOnlyRequest instance method 97
isDeleted instance method 112
isFault instance method 113
isIndexed instance method 225
isInserted instance method 113
isKindOfEntity: instance method 44
isOptional instance method 226
isReadOnly instance method 198
isTemporaryID instance method 158
isToMany instance method 240
isTransient instance method 226
isUpdated instance method 114

L

load: instance method 17
localizationDictionary instance method 172
lock instance method 137, 210

M

managedObjectClassName instance method 44
managedObjectContext instance method 114
managedObjectIDForURIRepresentation: instance method 210
managedObjectModel instance method 44, 211
mappingModel instance method 187
mappingModelFromBundles:forSourceModel: destinationModel: class method 178
mappingType instance method 58
maxCount instance method 240
Merge Policies 153
mergeChangesFromContextDidSaveNotification: instance method 137
mergedModelFromBundles: class method 164
mergedModelFromBundles:forStoreMetadata: class method 165
mergePolicy instance method 138
metadata instance method 18, 198
metadataForPersistentStore: instance method 211
metadataForPersistentStoreOfType:URL:error: class method 205
metadataForPersistentStoreWithURL:error: class method 195, 206
migratePersistentStore:toURL:options:withType:error: instance method 212
migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error: instance method 187
Migration Error Codes 253
Migration Options 219
migrationProgress instance method 188
minCount instance method 241
modelByMergingModels: class method 166
modelByMergingModels:forStoreMetadata: class method 166
mutableSetValueForKey: instance method 114

N

name instance method 45, 59, 226, 234
newCacheNodeForManagedObject: instance method 19
newReferenceObjectForManagedObject: instance method 19
NSAddedPersistentStoresKey constant 218
NSAddEntityMappingType constant 66
NSAffectedObjectsErrorKey constant 248
NSAffectedStoresErrorKey constant 248
NSAttributeType data type 35
NSBinaryDataAttributeType constant 36
NSBinaryStoreType constant 217
NSBooleanAttributeType constant 36
NSCascadeDeleteRule constant 244
NSCopyEntityMappingType constant 66
NSCoreDataError constant 254
NSCoreDataVersionNumber constant 254
NSCoreDataVersionNumber10_4 constant 255
NSCoreDataVersionNumber10_4_3 constant 255
NSCustomEntityMappingType constant 66
NSDateAttributeType constant 36
NSDecimalAttributeType constant 35
NSDeletedObjectsKey constant 152
NSDeleteRule data type 244
NSDenyDeleteRule constant 244
NSDetailedErrorsKey constant 247
NSDoubleAttributeType constant 36
NSEntityMappingType data type 67
NSEntityMigrationPolicyError constant 254
NSErrorMergePolicy constant 153
NSFetchRequestExpressionType constant 98
NSFetchRequestResultType data type 93
NSFloatAttributeType constant 36
NSIgnorePersistentStoreVersioningOption constant 220
NSInMemoryStoreType constant 217
NSInsertedObjectsKey constant 152
NSInteger16AttributeType constant 35
NSInteger32AttributeType constant 35
NSInteger64AttributeType constant 35
NSInvalidatedAllObjectsKey constant 153
NSInvalidatedObjectsKey constant 153
NSManagedObjectContext Change Notification User Info Keys 152
NSManagedObjectContextDidSaveNotification notification 155
NSManagedObjectContextLockingError constant 251
NSManagedObjectContextObjectsDidChangeNotification notification 155
NSManagedObjectExternalRelationshipError constant 251
NSManagedObjectIDResultType constant 93
NSManagedObjectMergeError constant 251
NSManagedObjectReferentialIntegrityError constant 251
NSManagedObjectResultType constant 93
NSManagedObjectValidationError constant 249
NSMergeByPropertyObjectTrumpMergePolicy constant 153
NSMergeByPropertyStoreTrumpMergePolicy constant 153

NSMigratePersistentStoresAutomaticallyOption constant 220
NSMigrationCancelledError constant 253
NSMigrationDestinationObjectKey constant 75
NSMigrationEntityMappingKey constant 75
NSMigrationError constant 253
NSMigrationManagerDestinationStoreError constant 254
NSMigrationManagerKey constant 75
NSMigrationManagerSourceStoreError constant 253
NSMigrationMissingMappingModelError constant 253
NSMigrationMissingSourceModelError constant 253
NSMigrationPropertyMappingKey constant 75
NSMigrationSourceObjectKey constant 75
NSNoActionDeleteRule constant 244
NSNullifyDeleteRule constant 244
NSOverwriteMergePolicy constant 154
NSPersistentStoreCoordinatorLockingError constant 251
NSPersistentStoreCoordinatorStoresDidChange-Notification notification 221
NSPersistentStoreIncompatibleSchemaError constant 252
NSPersistentStoreIncompatibleVersionHashError constant 253
NSPersistentStoreIncompleteSaveError constant 252
NSPersistentStoreInvalidTypeError constant 252
NSPersistentStoreOpenError constant 252
NSPersistentStoreOperationError constant 252
NSPersistentStoreOSCompatibility constant 221
NSPersistentStoreSaveError constant 252
NSPersistentStoreTimeoutError constant 253
NSPersistentStoreTimeoutOption constant 219
NSPersistentStoreTypeMismatchError constant 252
NSReadOnlyPersistentStoreOption constant 219
NSRefreshedObjectsKey constant 153
NSRemovedPersistentStoresKey constant 218
NSRemoveEntityMappingType constant 66
NSRollbackMergePolicy constant 154
NSSQLiteError constant 254
NSSQLiteErrorDomain constant 248
NSSQLitePragmasOption constant 219
NSSQLiteStoreType constant 217
NSStoreModelVersionHashesKey constant 220
NSStoreModelVersionIdentifiersKey constant 220
NSStoreTypeKey constant 218
NSStoreUUIDKey constant 218
NSStringAttributeType constant 36
NSTransformableAttributeType constant 36
NSTransformEntityMappingType constant 66
NSUndefinedAttributeType constant 35
NSUndefinedEntityMappingType constant 66
NSUpdatedObjectsKey constant 152
NSUUIDChangedPersistentStoresKey constant 218
NSValidateXMLStoreOption constant 219
NSValidationDateTooLateError constant 250
NSValidationDateTooSoonError constant 250
NSValidationInvalidDateError constant 250
NSValidationKeyErrorKey constant 247
NSValidationMissingMandatoryPropertyError constant 249
NSValidationMultipleErrorsError constant 249
NSValidationNumberTooLargeError constant 249
NSValidationNumberTooSmallError constant 250
NSValidationObjectErrorKey constant 247
NSValidationPredicateErrorKey constant 248
NSValidationRelationshipDeniedDeleteError constant 249
NSValidationRelationshipExceedsMaximumCountError constant 249
NSValidationRelationshipLacksMinimumCountError constant 249
NSValidationStringPatternMatchingError constant 250
NSValidationStringTooLongError constant 250
NSValidationStringTooShortError constant 250
NSValidationValueErrorKey constant 248
NSXMLStoreType constant 217

O

Object Graph Management Error Codes 250
objectDidBeginEditing: instance method 138
objectDidEndEditing: instance method 139
objectID instance method 26, 115
objectIDForEntity:referenceObject: instance method 20
objectRegisteredForID: instance method 139
objectWithID: instance method 140
observationInfo instance method 116
observeValueForKeyPath:ofObject:change:context: instance method 140
obtainPermanentIDsForObjects:error: instance method 141
options instance method 198

P

performCustomValidationForEntityMapping:manager:error: instance method 74
Persistent Store Error Codes 251
persistentStore instance method 159
persistentStoreCoordinator instance method 142, 199
persistentStoreForURL: instance method 213
persistentStores instance method 213
predicate instance method 85
primitiveValueForKey: instance method 116
processPendingChanges instance method 142
propagatesDeletesAtEndOfEvent instance method 142
properties instance method 45
propertiesByName instance method 46
propertyCache instance method 27

R

redo instance method 143
referenceObjectForObjectID: instance method 20
refreshObject:mergeChanges: instance method 143
registeredObjects instance method 144
registeredStoreTypes class method 207
registerStoreClass:forStoreType: class method 207
relationshipKeyPathsForPrefetching instance method 86
relationshipMappings instance method 59
relationshipsByName instance method 46
relationshipsWithDestinationEntity: instance method 47
removePersistentStore:error: instance method 214
requestExpression instance method 97
reset instance method 145, 188
resultType instance method 87
retainsRegisteredObjects instance method 145
returnsObjectsAsFaults instance method 87
rollback instance method 145

S

save: instance method 21, 146
self instance method 117
setAbstract: instance method 47
setAffectedStores: instance method 88
setAttributeMappings: instance method 60
setAttributeType: instance method 32
setAttributeValueClassName: instance method 32
setDefaultValue: instance method 33
setDeleteRule: instance method 241
setDestinationEntity: instance method 242
setDestinationEntityName: instance method 60
setDestinationEntityVersionHash: instance method 60
setEntities: instance method 172
setEntities:forConfiguration: instance method 173
setEntity: instance method 88
setEntityMappings: instance method 180
setEntityMigrationPolicyClassName: instance method 61
setFetchLimit: instance method 89
setFetchRequest: instance method 79
setFetchRequestTemplate:forName: instance method 173
setIdentifier: instance method 199
setIncludesPropertyValues: instance method 89
setIncludesSubentities: instance method 89
setIndexed: instance method 227
setInverseRelationship: instance method 242
setLocalizationDictionary: instance method 174
setManagedObjectClassName: instance method 47
setMappingType: instance method 61
setMaxCount: instance method 242
setMergePolicy: instance method 146
setMetadata: instance method 21, 200
setMetadata:forPersistentStore: instance method 214
setMetadata:forPersistentStoreOfType:URL:error: class method 208
setMetadata:forPersistentStoreWithURL:error: class method 195
setMinCount: instance method 243
setName: instance method 48, 62, 227, 234
setObservationInfo: instance method 117
setOptional: instance method 228
setPersistentStoreCoordinator: instance method 147
setPredicate: instance method 90
setPrimitiveValue:forKey: instance method 118
setPropagatesDeletesAtEndOfEvent: instance method 147
setProperties: instance method 48
setPropertyCache: instance method 27
setReadOnly: instance method 200
setRelationshipKeyPathsForPrefetching: instance method 90
setRelationshipMappings: instance method 62
setResultType: instance method 91

setRetainsRegisteredObjects: instance method 148
setReturnsObjectsAsFaults: instance method 91
setSortDescriptors: instance method 92
setSourceEntityName: instance method 62
setSourceEntityVersionHash: instance method 63
setSourceExpression: instance method 63
setStalenessInterval: instance method 148
setSubentities: instance method 49
setTransient: instance method 228
setUndoManager: instance method 149
setURL: instance method 200
setURL:forPersistentStore: instance method 215
setUserInfo: instance method 49, 63, 189, 229, 234
setValidationPredicates:withValidationWarnings: instance method 229
setValue:forKey: instance method 27, 119
setValueExpression: instance method 235
setValueTransformerName: instance method 33
setVersionHashModifier: instance method 50, 230
setVersionIdentifiers: instance method 175
sortDescriptors instance method 92
sourceContext instance method 189
sourceEntityForEntityMapping: instance method 189
sourceEntityName instance method 64
sourceEntityVersionHash instance method 64
sourceExpression instance method 65
sourceInstancesForEntityMappingNamed: destinationInstances: instance method 190
sourceModel instance method 191
stalenessInterval instance method 149
Store Metadata 217
Store Options 219
Store Types 217
Stores Change Notification User Info Keys 218
subentities instance method 50
subentitiesByName instance method 51
superentity instance method 51

T

tryLock instance method 150, 216
type instance method 201

U

undo instance method 150
undoManager instance method 150
unlock instance method 151, 216
updateCacheNode:fromManagedObject: instance method 22
updatedObjects instance method 151
URIRepresentation instance method 159
URL instance method 201
URLForPersistentStore: instance method 216
userInfo instance method 51, 65, 191, 230, 235

V

validateForDelete: instance method 120
validateForInsert: instance method 120
validateForUpdate: instance method 121
validateValue:forKey:error: instance method 122
Validation Error Codes 248
validationPredicates instance method 231
validationWarnings instance method 231
Value Expression Keys 75
valueExpression instance method 235
valueForKey: instance method 28, 122
valueTransformerName instance method 34
versionHash instance method 34, 52, 231, 243
versionHashModifier instance method 52, 232
versionIdentifiers instance method 175
Versioning Support 220

W

willAccessValueForKey: instance method 123
willRemoveCacheNodes: instance method 22
willRemoveFromPersistentStoreCoordinator: instance method 201
willSave instance method 124
willTurnIntoFault instance method 124