NSManagedObjectContext Class Reference

Data Management

2010-04-21

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

Simultaneously published in the United States and Canada.

Contents

NSManagedObjectContext Class Reference

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

Overview

An instance of `NSManagedObjectContext` represents a single "object space" or scratch pad in an application. Its primary responsibility is to manage a collection of managed objects. These objects form a group of related model objects that represent an internally consistent view of one or more persistent stores. A single managed object instance exists in one and only one context, but multiple copies of an object can exist in different contexts. Thus object uniquing is scoped to a particular context.

Life-cycle Management

The context is a powerful object with a central role in the life-cycle of managed objects, with responsibilities from life-cycle management (including faulting) to validation, inverse relationship handling, and undo/redo. Through a context you can retrieve or "fetch" objects from a persistent store, make changes to those objects, and then either discard the changes or—again through the context—commit them back to the persistent store. The context is responsible for watching for changes in its objects and maintains an undo manager so you can have finer-grained control over undo and redo. You can insert new objects and delete ones you have fetched, and commit these modifications to the persistent store.

Persistent Store Coordinator

A context always has a "parent" persistent store coordinator which provides the model and dispatches requests to the various persistent stores containing the data. Without a coordinator, a context is not fully functional. The context's coordinator provides the managed object model and handles persistency. All objects fetched from an external store are registered in a context together with a global identifier (an instance of `NSManagedObjectID`) that's used to uniquely identify each object to the external store.

Subclassing Notes

You are strongly discouraged from subclassing `NSManagedObjectContext`. The change tracking and undo management mechanisms are highly optimized and hence intricate and delicate. Interposing your own additional logic that might impact `processPendingChanges` can have unforeseen consequences. In situations such as store migration, Core Data will create instances of `NSManagedObjectContext` for its own use. Under these circumstances, you cannot rely on any features of your custom subclass. Any `NSManagedObject` subclass must always be fully compatible with `NSManagedObjectContext` (as opposed to any subclass of `NSManagedObjectContext`).

Tasks

Registering and Fetching Objects

- `executeFetchRequest:error:` (page 12)
  Returns an array of objects that meet the criteria specified by a given fetch request.
- `countForFetchRequest:error:` (page 10)
  Returns the number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:`.
- `objectRegisteredForID:` (page 16)
  Returns the object for a specified ID, if the object is registered with the receiver.
- `objectWithID:` (page 17)
  Returns the object for a specified ID.
- `existingObjectWithID:error:` (page 13)
  Returns the object for the specified ID.
- `registeredObjects` (page 20)
  Returns the set of objects registered with the receiver.

Managed Object Management

- `insertObject:` (page 14)
  Registers an object to be inserted in the receiver's persistent store the next time changes are saved.
- `deleteObject:` (page 11)
  Specifies an object that should be removed from its persistent store when changes are committed.

- `assignObject:toPersistentStore:` (page 9)
  Specifies the store in which a newly-inserted object will be saved.
- `obtainPermanentIDsForObjects:error:` (page 17)
  Converts to permanent IDs the object IDs of the objects in a given array.
- `detectConflictsForObject:` (page 11)
  Marks an object for conflict detection.
- `refreshObject:mergeChanges:` (page 19)
  Updates the persistent properties of a managed object to use the latest values from the persistent store.
- `processPendingChanges` (page 18)
  Forces the receiver to process changes to the object graph.
- `insertedObjects` (page 14)
  Returns the set of objects that have been inserted into the receiver but not yet saved in a persistent store.
- `updatedObjects` (page 27)
  Returns the set of objects registered with the receiver that have uncommitted changes.
- `deletedObjects` (page 10)
  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 15)
  Merges the changes specified in a given notification.

Undo Management

- `undoManager` (page 27)
  Returns the undo manager of the receiver.
- `setUndoManager:` (page 25)
  Sets the undo manager of the receiver.
- `undo` (page 26)
  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 19)
  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 21)
  Returns the receiver to its base state.
- `rollback` (page 21)
  Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values.
- `save:` (page 22)
  Attempts to commit unsaved changes to registered objects to their persistent store.

- `hasChanges` (page 13)
  Returns a Boolean value that indicates whether the receiver has uncommitted changes.

Locking

- `lock` (page 15)
  Attempts to acquire a lock on the receiver.
- `unlock` (page 27)
  Relinquishes a previously acquired lock.
- `tryLock` (page 26)
  Attempts to acquire a lock.

Delete Propagation

- `propagatesDeletesAtEndOfEvent` (page 19)
  Returns a Boolean that indicates whether the receiver propagates deletes at the end of the event in which a change was made.
- `setPropagatesDeletesAtEndOfEvent:` (page 23)
  Sets whether the context propagates deletes to related objects at the end of the event.

Retaining Registered Objects

- `retainsRegisteredObjects` (page 21)
  Returns a Boolean that indicates whether the receiver sends a `retain` message to objects upon registration.
- `setRetainsRegisteredObjects:` (page 24)
  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 18)
  Returns the persistent store coordinator of the receiver.
- `setPersistentStoreCoordinator:` (page 23)
  Sets the persistent store coordinator of the receiver.

Managing the Staleness Interval

- `stalenessInterval` (page 25)
  Returns the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

- `setStalenessInterval:` (page 24)
  Sets the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

Managing the Merge Policy

- `mergePolicy` (page 16)
  Returns the merge policy of the receiver.
- `setMergePolicy:` (page 22)
  Sets the merge policy of the receiver.

Instance Methods

assignObject:toPersistentStore:

Specifies the store in which a newly-inserted object will be saved.

```
- (void)assignObject:(id)object toPersistentStore:(NSPersistentStore *)store
```

Parameters

object
  A managed object.

store
  A persistent store.

Discussion
You can obtain a store from the persistent store coordinator, using for example `persistentStoreForURL:`.

Special Considerations

It is only necessary to use this method if the receiver's persistent store coordinator manages multiple writable stores that have *object*'s entity in their configuration. Maintaining configurations in the managed object model can eliminate the need for invoking this method directly in many situations. If the receiver's persistent store coordinator manages only a single writable store, or if only one store has *object*'s entity in its model, *object* will automatically be assigned to that store.

Availability
Available in iOS 3.0 and later.

See Also
- `insertObject:` (page 14)
- `persistentStoreCoordinator` (page 18)

Declared In
`NSManagedObjectContext.h`

countForFetchRequest:error:

Returns the number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:`.

```
- (NSUInteger)countForFetchRequest:(NSFetchRequest *)request error:(NSError **)error
```

Parameters

request

A fetch request that specifies the search criteria for the fetch.

error

If there is a problem executing the fetch, upon return contains an instance of `NSError` that describes the problem.

Return Value

The number of objects a given fetch request would have returned if it had been passed to `executeFetchRequest:error:` (page 12).

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

deletedObjects

Returns the set of objects that will be removed from their persistent store during the next save operation.

```
- (NSSet *)deletedObjects
```

Return Value

The set of objects that will be removed from their persistent store during the next save operation.

Discussion

The returned set does not necessarily include all the objects that have been deleted (using `deleteObject:` (page 11))—if an object has been inserted and deleted without an intervening save operation, it is not included in the set.

A managed object context does not post key-value observing notifications when the return value of `deletedObjects` changes. A context does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 31) notification when a change is made, and a `NSManagedObjectContextWillSaveNotification` (page 31) notification and a `NSManagedObjectContextDidSaveNotification` (page 31) notification before and after changes are committed respectively (although again the set of deleted objects given for a `NSManagedObjectContextDidSaveNotification` (page 31) does not include objects that were inserted and deleted without an intervening save operation—that is, they had never been saved to a persistent store).

Availability

Available in iOS 3.0 and later.

See Also

- `deleteObject:` (page 11)
- `insertedObjects` (page 14)
- `registeredObjects` (page 20)

- updatedObjects (page 27)
- isDeleted (NSManagedObject)

Declared In
NSManagedObjectContext.h

deleteObject:

Specifies an object that should be removed from its persistent store when changes are committed.

```
- (void)deleteObject:(NSManagedObject *)object
```

Parameters

object
A managed object.

Discussion
When changes are committed, *object* will be removed from the uniquing tables. If *object* has not yet been saved to a persistent store, it is simply removed from the receiver.

Availability
Available in iOS 3.0 and later.

See Also
- deletedObjects (page 10)
- isDeleted (NSManagedObject)

Declared In
NSManagedObjectContext.h

detectConflictsForObject:

Marks an object for conflict detection.

```
- (void)detectConflictsForObject:(NSManagedObject *)object
```

Parameters

object
A managed object.

Discussion
If on the next invocation of save: (page 22) *object* has been modified in its persistent store, the save fails. This allows optimistic locking for unchanged objects. Conflict detection is always performed on changed or deleted objects.

Availability
Available in iOS 3.0 and later.

Declared In
NSManagedObjectContext.h

executeFetchRequest:error:

Returns an array of objects that meet the criteria specified by a given fetch request.

```
- (NSArray *)executeFetchRequest:(NSFetchRequest *)request error:(NSError **)error
```

Parameters

request

A fetch request that specifies the search criteria for the fetch.

error

If there is a problem executing the fetch, upon return contains an instance of `NSError` that describes the problem.

Return Value

An array of objects that meet the criteria specified by *request* fetched from the receiver and from the persistent stores associated with the receiver's persistent store coordinator. If an error occurs, returns `nil`. If no objects match the criteria specified by *request*, returns an empty array.

Discussion

Returned objects are registered with the receiver.

The following points are important to consider:

- If the fetch request has no predicate, then all instances of the specified entity are retrieved, modulo other criteria below.
- An object that meets the criteria specified by *request* (it is an instance of the entity specified by the request, and it matches the request's predicate if there is one) and that has been inserted into a context but which is not yet saved to a persistent store, is retrieved if the fetch request is executed on that context.
- If an object in a context has been modified, a predicate is evaluated against its modified state, not against the current state in the persistent store. Therefore, if an object in a context has been modified such that it meets the fetch request's criteria, the request retrieves it even if changes have not been saved to the store and the values in the store are such that it does not meet the criteria. Conversely, if an object in a context has been modified such that it does not match the fetch request, the fetch request will not retrieve it even if the version in the store does match.
- If an object has been deleted from the context, the fetch request does not retrieve it even if that deletion has not been saved to a store.

Objects that have been realized (populated, faults fired, "read from", and so on) as well as pending updated, inserted, or deleted, are never changed by a fetch operation without developer intervention. If you fetch some objects, work with them, and then execute a new fetch that includes a superset of those objects, you do not get new instances or update data for the existing objects—you get the existing objects with their current in-memory state.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

existingObjectWithID:error:

Returns the object for the specified ID.

```
- (NSManagedObject *)existingObjectWithID:(NSManagedObjectID *)objectIDerror:(NSError
    **)error
```

Parameters

objectID

The object ID for the requested object.

error

If there is a problem in retrieving the object specified by *objectID*, upon return contains an error that describes the problem.

Return Value

The object specified by *objectID*. If the object cannot be fetched, or does not exist, or cannot be faulted, it returns `nil`.

Discussion

If there is a managed object with the given ID already registered in the context, that object is returned directly; otherwise the corresponding object is faulted into the context.

This method might perform I/O if the data is uncached.

Unlike `objectWithID:` (page 17), this method never returns a fault.

Availability

Available in iOS 3.0 and later.

See Also

- `objectWithID:` (page 17)
- `objectRegisteredForID:` (page 16)

Declared In

`NSManagedObjectContext.h`

hasChanges

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

```
- (BOOL)hasChanges
```

Return Value

`YES` if the receiver has uncommitted changes, otherwise `NO`.

Discussion

On Mac OS X v10.6 and later, this property is key-value observing compliant (see *Key-Value Observing Programming Guide*).

Prior to Mac OS X v10.6, this property is not key-value observing compliant—for example, if you are using Cocoa bindings, you cannot bind to the `hasChanges` property of a managed object context.

Special Considerations

If you are observing this property using key-value observing (KVO) you should not touch the context or its objects within your implementation of `observeValueForKeyPath:ofObject:change:context:` for this notification. (This is because of the intricacy of the locations of the KVO notifications—for example, the context may be in the middle of an undo operation, or repairing a merge conflict.) If you need to send messages to the context of change any of its managed objects as a result of a change to the value of `hasChanges`, you must do so after the call stack unwinds (typically using `performSelector:withObject:afterDelay:` or a similar method).

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

insertedObjects

Returns the set of objects that have been inserted into the receiver but not yet saved in a persistent store.

```
- (NSSet *)insertedObjects
```

Return Value
The set of objects that have been inserted into the receiver but not yet saved in a persistent store.

Discussion
A managed object context does not post key-value observing notifications when the return value of `insertedObjects` changes—it does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 31) notification when a change is made, and a `NSManagedObjectContextWillSaveNotification` (page 31) and a `NSManagedObjectContextDidSaveNotification` (page 31) notification before and after changes are committed respectively.

Availability
Available in iOS 3.0 and later.

See Also
- `deletedObjects` (page 10)
- `insertObject:` (page 14)
- `registeredObjects` (page 20)
- `updatedObjects` (page 27)

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

Availability

Available in iOS 3.0 and later.

See Also

- `insertedObjects` (page 14)

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 27).

Sending this message to a managed object context helps the framework to understand the scope of a transaction in a multi-threaded environment. It is preferable to use the `NSManagedObjectContext`'s implementation of `NSLocking` instead using of a separate mutex object.

If you lock (or successfully `tryLock`) a managed object context, the thread in which the lock call is made must have a retain until it invokes unlock. If you do not properly retain a context in a multi-threaded environment, this will result in deadlock.

Availability

Available in iOS 3.0 and later.

See Also

- `tryLock` (page 26)
- `unlock` (page 27)

Declared In

`NSManagedObjectContext.h`

mergeChangesFromContextDidSaveNotification:

Merges the changes specified in a given notification.

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

Parameters

notification

An instance of an `NSManagedObjectContextWillSaveNotification` (page 31) 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 11): on those which have been deleted.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

mergePolicy

Returns the merge policy of the receiver.

```
- (id)mergePolicy
```

Return Value

The receiver's merge policy.

Discussion

The default is `NSErrorMergePolicy`.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObjectContext.h`

objectRegisteredForID:

Returns the object for a specified ID, if the object is registered with the receiver.

```
- (NSManagedObject *)objectRegisteredForID:(NSManagedObjectID *)objectID
```

Parameters

objectID

An object ID.

Return Value

The object for the specified ID if it is registered with the receiver, otherwise `nil`.

Availability

Available in iOS 3.0 and later.

See Also

- `objectWithID:` (page 17)
- `existingObjectWithID:error:` (page 13)

Declared In
NSManagedObjectContext.h

objectWithID:

Returns the object for a specified ID.

```
- (NSManagedObject *)objectWithID:(NSManagedObjectID *)objectID
```

Parameters

objectID
An object ID.

Return Value
The object for the specified ID.

Discussion
If the object is not registered in the context, it may be fetched or returned as a fault. This method always returns an object. The data in the persistent store represented by *objectID* is assumed to exist—if it does not, the returned object throws an exception when you access any property (that is, when the fault is fired). The benefit of this behavior is that it allows you to create and use faults, then create the underlying rows later or in a separate context.

Availability
Available in iOS 3.0 and later.

See Also
- objectRegisteredForID: (page 16)
- existingObjectWithID:error: (page 13)
- managedObjectIDForURIRepresentation:
- URIRepresentation

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` until it is saved. Any object that already has a permanent ID is ignored.

Any object not already assigned to a store is assigned based on the same rules Core Data uses for assignment during a save operation (first writable store supporting the entity, and appropriate for the instance and its related items).

Special Considerations

This method results in a transaction with the underlying store which changes the file's modification date.

This results an additional consideration if you invoke this method on the managed object context associated with an instance of `NSPersistentDocument`. Instances of `NSDocument` need to know that they are in sync with the underlying content. To avoid problems, after invoking this method you must therefore update the document's modification date (using `setFileModificationDate:`).

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

persistentStoreCoordinator

Returns the persistent store coordinator of the receiver.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
```

Return Value
The persistent store coordinator of the receiver.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

processPendingChanges

Forces the receiver to process changes to the object graph.

```
- (void)processPendingChanges
```

Discussion
This method causes changes to registered managed objects to be recorded with the undo manager.

In AppKit-based applications, this method is invoked automatically at least once during the event loop (at the end of the loop)—it may be called more often than that if the framework needs to coalesce your changes before doing something else. You can also invoke it manually to coalesce any pending unprocessed changes.

Availability
Available in iOS 3.0 and later.

See Also
- `redo` (page 19)
- `undo` (page 26)
- `undoManager` (page ?)

Declared In
`NSManagedObjectContext.h`

propagatesDeletesAtEndOfEvent

Returns a Boolean that indicates whether the receiver propagates deletes at the end of the event in which a change was made.

```
- (BOOL)propagatesDeletesAtEndOfEvent
```

Return Value
`YES` if the receiver propagates deletes at the end of the event in which a change was made, `NO` if it propagates deletes only immediately before saving changes.

Availability
Available in iOS 3.0 and later.

See Also
- `setPropagatesDeletesAtEndOfEvent:` (page 23)

Declared In
`NSManagedObjectContext.h`

redo

Sends an redo message to the receiver's undo manager, asking it to reverse the latest undo operation applied to objects in the object graph.

```
- (void)redo
```

Availability
Available in iOS 3.0 and later.

See Also
- `undo` (page 26)
- `processPendingChanges` (page 18)

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 25)) has not been exceeded, any available cached data is reused instead of executing a new fetch. If *flag* is YES, this method does not affect any transient properties; if *flag* is NO, transient properties are released.

You typically use this method to ensure data freshness if more than one managed object context may use the same persistent store simultaneously, in particular if you get an optimistic locking failure when attempting to save.

It is important to note that turning *object* into a fault (*flag* is NO) also causes related managed objects (that is, those to which *object* has a reference) to be released, so you can also use this method to trim a portion of your object graph you want to constrain memory usage.

Availability

Available in iOS 3.0 and later.

See Also

- `detectConflictsForObject:` (page 11)
- `reset` (page 21)
- `setStalenessInterval:` (page 24)

Declared In

`NSManagedObjectContext.h`

registeredObjects

Returns the set of objects registered with the receiver.

```
- (NSSet *)registeredObjects
```

Return Value

The set of objects registered with the receiver.

Discussion

A managed object context does not post key-value observing notifications when the return value of `registeredObjects` changes.

Availability

Available in iOS 3.0 and later.

See Also
- `deletedObjects` (page 10)
- `insertedObjects` (page 14)
- `updatedObjects` (page 27)

Declared In
`NSManagedObjectContext.h`

reset

Returns the receiver to its base state.

```
- (void)reset
```

Discussion
All the receiver's managed objects are “forgotten.” If you use this method, you should ensure that you also discard references to any managed objects fetched using the receiver, since they will be invalid afterwards.

Availability
Available in iOS 3.0 and later.

See Also
- `rollback` (page 21)
- `setStalenessInterval:` (page 24)
- `undo` (page 26)

Declared In
`NSManagedObjectContext.h`

retainsRegisteredObjects

Returns a Boolean that indicates whether the receiver sends a `retain` message to objects upon registration.

```
- (BOOL)retainsRegisteredObjects
```

Return Value
`YES` if the receiver sends a `retain` message to objects upon registration, otherwise `NO`.

Availability
Available in iOS 3.0 and later.

See Also
- `setRetainsRegisteredObjects:` (page 24)

Declared In
`NSManagedObjectContext.h`

rollback

Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values.

```
- (void)rollback
```

Discussion
This method does not refetch data from the persistent store or stores.

Availability
Available in iOS 3.0 and later.

See Also
- `reset` (page 21)
- `setStalenessInterval:` (page 24)
- `undo` (page 26)
- `processPendingChanges` (page 18)

Declared In
`NSManagedObjectContext.h`

save:

Attempts to commit unsaved changes to registered objects to their persistent store.

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

Parameters
error
A pointer to an `NSError` object. You do not need to create an `NSError` object. The save operation aborts after the first failure if you pass `NULL`.

Return Value
`YES` if the save succeeds, otherwise `NO`.

Discussion
If there were multiple errors (for example several edited objects had validation failures) the description of `NSError` returned indicates that there were multiple errors, and its userInfo dictionary contains the key `NSDetailedErrors`. The value associated with the `NSDetailedErrors` key is an array that contains the individual `NSError` objects.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

setMergePolicy:

Sets the merge policy of the receiver.

```
- (void)setMergePolicy:(id)mergePolicy
```

Parameters
mergePolicy
The merge policy of the receiver. For possible values, see "Merge Policies" (page 29).

Availability
Available in iOS 3.0 and later.

Declared In
NSManagedObjectContext.h

setPersistentStoreCoordinator:

Sets the persistent store coordinator of the receiver.

```
- (void)setPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
```

Parameters

coordinator

The persistent store coordinator of the receiver.

Discussion
The coordinator provides the managed object model and handles persistency. Note that multiple contexts can share a coordinator.

This method raises an exception if *coordinator* is nil. If you want to “disconnect" a context from its persistent store coordinator, you should simply release all references to the context and allow it to be deallocated normally.

Availability
Available in iOS 3.0 and later.

Declared In
NSManagedObjectContext.h

setPropagatesDeletesAtEndOfEvent:

Sets whether the context propagates deletes to related objects at the end of the event.

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

Parameters

Flag

A Boolean value that indicates whether the context propagates deletes to related objects at the end of the event (YES) or not (NO).

Discussion
The default is YES. If the value is NO, then deletes are propagated during a save operation.

Availability
Available in iOS 3.0 and later.

See Also
- propagatesDeletesAtEndOfEvent (page 19)

Declared In
NSManagedObjectContext.h

setRetainsRegisteredObjects:

Sets whether or not the receiver retains all registered objects, or only objects necessary for a pending save (those that are inserted, updated, deleted, or locked).

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

Parameters

flag

A Boolean value.

If *flag* is `NO`, then registered objects are retained only when they are inserted, updated, deleted, or locked.

If *flag* is `YES`, then all registered objects are retained.

Discussion
The default is `NO`.

Availability
Available in iOS 3.0 and later.

See Also
`- retainsRegisteredObjects` (page 21)

Declared In
`NSManagedObjectContext.h`

setStalenessInterval:

Sets the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

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

Parameters

expiration

The maximum length of time that may have elapsed since the store previously fetched data before *fulfilling a fault* issues a new fetch rather than using the previously-fetched data.

A negative value represents an infinite value; `0.0` represents "no staleness acceptable".

Discussion
The staleness interval controls whether *fulfilling a fault* uses data previously fetched by the application, or issues a new fetch (see also `refreshObject:mergeChanges:` (page 19)). The staleness interval does *not* affect objects currently in use (that is, it is *not* used to automatically update property values from a persistent store after a certain period of time).

The expiration value is applied on a per object basis. It is the relative time until cached data (snapshots) should be considered stale. For example, a value of 300.0 informs the context to utilize cached information for no more than 5 minutes after an object was originally fetched.

Note that the staleness interval is a hint and may not be supported by all persistent store types. It is not used by XML and binary stores, since these stores maintain all current values in memory.

Availability
Available in iOS 3.0 and later.

See Also
- `reset` (page 21)
- `rollback` (page 21)
- `stalenessInterval` (page 25)
- `undo` (page 26)
- `refreshObject:mergeChanges:` (page 19)

Declared In
`NSManagedObjectContext.h`

setUndoManager:

Sets the undo manager of the receiver.

```
- (void)setUndoManager:(NSUndoManager *)undoManager
```

Parameters

undoManager
The undo manager of the receiver.

Discussion
You can set the undo manager to `nil` to disable undo support. This provides a performance benefit if you do not want to support undo for a particular context, for example in a large import process—see *Core Data Programming Guide*.

If a context does not have an undo manager, you can enable undo support by setting one. You may also replace a context's undo manager if you want to integrate the context's undo operations with another undo manager in your application.

Important: On Mac OS X, a context provides an undo manager by default; on iOS, the undo manager is `nil` by default.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

stalenessInterval

Returns the maximum length of time that may have elapsed since the store previously fetched data before fulfilling a fault issues a new fetch rather than using the previously-fetched data.

```
- (NSTimeInterval)stalenessInterval
```

Return Value
The maximum length of time that may have elapsed since the store previously fetched data before *fulfilling a fault* issues a new fetch rather than using the previously-fetched data.

Discussion
The default is infinite staleness, represented by an interval of `-1` (although any negative value represents an infinite value); `0.0` represents "no staleness acceptable".

For a full discussion, see `setStalenessInterval:` (page 24).

Availability
Available in iOS 3.0 and later.

See Also
- `setStalenessInterval:` (page 24)

Declared In
`NSManagedObjectContext.h`

tryLock

Attempts to acquire a lock.

```
- (BOOL)tryLock
```

Return Value
`YES` if a lock was acquired, `NO` otherwise.

Discussion
This method returns immediately after the attempt to acquire a lock.

Availability
Available in iOS 3.0 and later.

See Also
- `lock` (page 15)
- `unlock` (page 27)

Declared In
`NSManagedObjectContext.h`

undo

Sends an undo message to the receiver's undo manager, asking it to reverse the latest uncommitted changes applied to objects in the object graph.

```
- (void)undo
```

Availability
Available in iOS 3.0 and later.

See Also
- `reset` (page 21)
- `rollback` (page 21)
- `undoManager` (page ?)
- `processPendingChanges` (page 18)

Declared In
NSManagedObjectContext.h

undoManager

Returns the undo manager of the receiver.

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

Return Value
The undo manager of the receiver.

Discussion
For a discussion, see `setUndoManager:` (page ?).

Important: On Mac OS X, a context provides an undo manager by default; on iOS, the undo manager is `nil` by default.

Availability
Available in iOS 3.0 and later.

Declared In
NSManagedObjectContext.h

unlock

Relinquishes a previously acquired lock.

```
- (void)unlock
```

Availability
Available in iOS 3.0 and later.

See Also
- `lock` (page 15)
- `tryLock` (page 26)

Declared In
NSManagedObjectContext.h

updatedObjects

Returns the set of objects registered with the receiver that have uncommitted changes.

```
- (NSSet *)updatedObjects
```

Return Value
The set of objects registered with the receiver that have uncommitted changes.

Discussion
A managed object context does not post key-value observing notifications when the return value of `updatedObjects` changes. A context does, however, post a `NSManagedObjectContextObjectsDidChangeNotification` (page 31) notification when a change is made, and a `NSManagedObjectContextWillSaveNotification` (page 31) notification and a `NSManagedObjectContextDidSaveNotification` (page 31) notification before and after changes are committed respectively.

Availability
Available in iOS 3.0 and later.

See Also
- `deletedObjects` (page 10)
- `insertedObjects` (page 14)
- `registeredObjects` (page 20)

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 aNSManagedObjectContextObjectsDidChangeNotification (page 31) notification.

```
NSString * const NSInsertedObjectsKey;
NSString * const NSUpdatedObjectsKey;
NSString * const NSDeletedObjectsKey;
NSString * const NSRefreshedObjectsKey;
NSString * const NSInvalidatedObjectsKey;
NSString * const NSInvalidatedAllObjectsKey;
```

Constants

`NSInsertedObjectsKey`
Key for the set of objects that were inserted into the context.
Available in iOS 3.0 and later.
Declared in `NSManagedObjectContext.h`.

`NSUpdatedObjectsKey`
Key for the set of objects that were updated.
Available in iOS 3.0 and later.
Declared in `NSManagedObjectContext.h`.

`NSDeletedObjectsKey`
Key for the set of objects that were marked for deletion during the previous event.
Available in iOS 3.0 and later.
Declared in `NSManagedObjectContext.h`.

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

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

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

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

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

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

Declared In
`NSManagedObjectContext.h`

Merge Policies

Merge policy constants define the way conflicts are handled during a save operation.

```
id NSErrorMergePolicy;
id NSMergeByPropertyStoreTrumpMergePolicy;
id NSMergeByPropertyObjectTrumpMergePolicy;
id NSOverwriteMergePolicy;
id NSRollbackMergePolicy;
```

Constants
`NSErrorMergePolicy`
This policy causes a save to fail if there are any merge conflicts.

In the case of failure, the save method returns with an error with a userInfo dictionary that contains the key `@"conflictList"`; the corresponding value is an array of conflict records.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSMergeByPropertyStoreTrumpMergePolicy`
This policy merges conflicts between the persistent store's version of the object and the current in-memory version, giving priority to external changes.

The merge occurs by individual property. For properties that have been changed in both the external source and in memory, the external changes trump the in-memory ones.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSMergeByPropertyObjectTrumpMergePolicy`
This policy merges conflicts between the persistent store's version of the object and the current in-memory version, giving priority to in-memory changes.

The merge occurs by individual property. For properties that have been changed in both the external source and in memory, the in-memory changes trump the external ones.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSOverwriteMergePolicy`

This policy overwrites state in the persistent store for the changed objects in conflict.

Changed objects' current state is forced upon the persistent store.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

`NSRollbackMergePolicy`

This policy discards in-memory state changes for objects in conflict.

The persistent store's version of the objects' state is used.

Available in iOS 3.0 and later.

Declared in `NSManagedObjectContext.h`.

Discussion

The default policy is the `NSErrorMergePolicy`. It is the only policy that requires action to correct any conflicts; the other policies make a save go through silently by making changes following their rules.

Declared In

`NSManagedObjectContext.h`

The following constants, defined in `CoreDataErrors.h`, relate to errors returned following validation failures or problems encountered during a save operation.

`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 18), after the changes have been processed, but before it is safe to call save: (page 22) again (if you try, you will generate an infinite loop).

The notification object is the managed object context. The *userInfo* dictionary contains the following keys: `NSInsertedObjectsKey`, `NSUpdatedObjectsKey`, and `NSDeletedObjectsKey`.

Note that this notification is posted only when managed objects are *changed*; it is not posted when managed objects are added to a context as the result of a fetch.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

NSManagedObjectContextDidSaveNotification

Posted whenever a managed object context completes a save operation.

The notification object is the managed object context. The *userInfo* dictionary contains the following keys: `NSInsertedObjectsKey`, `NSUpdatedObjectsKey`, and `NSDeletedObjectsKey`.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

NSManagedObjectContextWillSaveNotification

Posted whenever a managed object context is about to perform a save operation.

The notification object is the managed object context. There is no *userInfo* dictionary.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObjectContext.h`

Document Revision History

This table describes the changes to *NSManagedObjectContext Class Reference*.

Date	Notes
2010-04-21	Corrected description of NSManagedObjectContextWillSaveNotification (there is no user info dictionary).
2009-10-06	Corrected descriptions of save notifications.
2009-08-12	Corrected definition of stalenessInterval.
2009-05-04	Enhanced discussion of undo manager.
2009-02-25	Updated for iOS 3.0.
2007-07-19	Updated for Mac OS X v10.5.
2007-03-06	Clarified description of NSManagedObjectContextObjectsDidChangeNotification.
2007-01-08	Noted that hasChanges is not KVO-compliant, and enhanced discussion of setPersistentStoreCoordinator:.
2006-11-07	Clarified effect of refreshObject:mergeChanges:.
2006-05-23	First publication of this content as a separate document.

Document Revision History