NSManagedObject Class Reference

Data Management

2010-05-25

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

Simultaneously published in the United States and Canada.

Contents

NSManagedObject Class Reference

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

Overview

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

A managed object is associated with an entity description (an instance of `NSEntityDescription`) that provides metadata about the object (including the name of the entity that the object represents and the names of its attributes and relationships) and with a managed object context that tracks changes to the object graph. It is important that a managed object is properly configured for use with Core Data. If you instantiate a managed object directly, you must call the designated initializer (`initWithEntity:insertIntoManagedObjectContext:` (page 19)).

Data Storage

In some respects, an `NSManagedObject` acts like a dictionary—it is a generic container object that efficiently provides storage for the properties defined by its associated `NSEntityDescription` object. `NSManagedObject` provides support for a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). There is therefore commonly no need to define instance variables in subclasses. Sometimes, however, you want to use types that are not supported directly, such as colors and C structures. For example, in a graphics application you might want to define a Rectangle entity that has attributes color and bounds that are an instance of `NSColor` and an `NSRect` struct respectively. For some types you can use a transformable attribute, for others this may require you to create a subclass of `NSManagedObject`—see “Non-Standard Persistent Attributes”.

Faulting

Managed objects typically represent data held in a persistent store. In some situations a managed object may be a "fault"—an object whose property values have not yet been loaded from the external data store—see "Faulting and Uniquing" for more details. When you access persistent property values, the fault "fires" and the data is retrieved from the store automatically. This can be a comparatively expensive process (potentially requiring a round trip to the persistent store), and you may wish to avoid unnecessarily firing a fault.

You can safely invoke the following methods on a fault without causing it to fire: `isEqual:`, `hash`, `superclass`, `class`, `self`, `zone`, `isProxy`, `isKindOfClass:`, `isMemberOfClass:`, `conformsToProtocol:`, `respondsToSelector:`, `retain`, `release`, `autorelease`, `retainCount`, `description`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, `faultingState`, and `isFault`. Since `isEqual` and `hash` do not cause a fault to fire, managed objects can typically be placed in collections without firing a fault. Note, however, that invoking key-value coding methods on the collection object might in turn result in an invocation of `valueForKey:` on a managed object, which would fire the fault.

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

Subclassing Notes

In combination with the entity description in the managed object model, `NSManagedObject` provides a rich set of default behaviors including support for arbitrary properties and value validation. There are, however, many reasons why you might wish to subclass `NSManagedObject` to implement custom features. It is important, though, not to disrupt Core Data's behavior.

Methods you Must Not Override

`NSManagedObject` itself customizes many features of `NSObject` so that managed objects can be properly integrated into the Core Data infrastructure. Core Data relies on `NSManagedObject`'s implementation of the following methods, which you therefore absolutely must not override: `primitiveValueForKey:`, `setPrimitiveValue:forKey:`, `isEqual:`, `hash`, `superclass`, `class`, `self`, `zone`, `isProxy`, `isKindOfClass:`, `isMemberOfClass:`, `conformsToProtocol:`, `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 19)).

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 12)—see the method description for details.

Custom Accessor Methods

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

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

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

Custom Instance Variables

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

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

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

If you define custom instance variables, for example, to store derived attributes or other transient properties, you should clean up these variables in `didTurnIntoFault` (page 18) 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 30), 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 29) 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 29)), you should call the superclass's implementation first. This ensures that individual property validation methods are also invoked. If there are multiple validation failures in one operation, you should collect them in an array and add the array—using the key `NSDetailedErrorsKey`—to the userInfo dictionary in the `NSError` object you return. For an example, see "Model Object Validation".

Tasks

Initializing a Managed Object

`- initWithEntity:insertIntoManagedObjectContext:` (page 19)
Initializes the receiver and inserts it into the specified managed object context.

Getting a Managed Object's Identity

- `entity` (page 18)
  Returns the entity description of the receiver.
- `objectID` (page 24)
  Returns the object ID of the receiver.
- `self` (page 26)
  Returns the receiver.

Getting State Information

- `managedObjectContext` (page 23)
  Returns the managed object context with which the receiver is registered.
- `isInserted` (page 22)
  Returns a Boolean value that indicates whether the receiver has been inserted in a managed object context.
- `isUpdated` (page 22)
  Returns a Boolean value that indicates whether the receiver has unsaved changes.
- `isDeleted` (page 20)
  Returns a Boolean value that indicates whether the receiver will be deleted during the next save.
- `isFault` (page 21)
  Returns a Boolean value that indicates whether the receiver is a fault.
- `faultingState` (page 18)
  Returns a value that indicates the faulting state of the receiver.
- `hasFaultForRelationshipNamed:` (page 19)
  Returns a Boolean value that indicates whether the relationship for a given key is a fault.

Managing Life Cycle and Change Events

+ `contextShouldIgnoreUnmodeledPropertyChanges` (page 12)
  Returns a Boolean value that indicates whether instances of the class should be marked as having changes if an unmodeled property is changed.
- `awakeFromFetch` (page 12)
  Invoked automatically by the Core Data framework after the receiver has been fetched.
- `awakeFromInsert` (page 13)
  Invoked automatically by the Core Data framework when the receiver is first inserted into a managed object context.
- `awakeFromSnapshotEvents:` (page 13)
  Invoked automatically by the Core Data framework when the receiver is reset due to an undo, redo, or other multi-property state change.
- `changedValues` (page 14)
  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 15)
  Returns a dictionary of the last fetched or saved values of the receiver for the properties specified by the given keys.
- `prepareForDeletion` (page 24)
  Invoked automatically by the Core Data framework when the receiver is about to be deleted.
- `dealloc` (page 15)
  Deallocates the memory occupied by the receiver.
- `willSave` (page 33)
  Invoked automatically by the Core Data framework when the receiver's managed object context is saved.
- `didSave` (page 17)
  Invoked automatically by the Core Data framework after the receiver's managed object context completes a save operation.
- `willTurnIntoFault` (page 34)
  Invoked automatically by the Core Data framework before receiver is converted to a fault.
- `didTurnIntoFault` (page 18)
  Invoked automatically by the Core Data framework when the receiver is turned into a fault.

Supporting Key-Value Coding

- `valueForKey:` (page 31)
  Returns the value for the property specified by *key*.
- `setValue:forKey:` (page 28)
  Sets the specified property of the receiver to the specified value.
- `mutableSetValueForKey:` (page 23)
  Returns a mutable set that provides read-write access to the unordered to-many relationship specified by a given key.
- `primitiveValueForKey:` (page 25)
  Returns from the receiver's private internal storage the value for the specified property.
- `setPrimitiveValue:forKey:` (page 26)
  Sets in the receiver's private internal storage the value of a given property.

Validation

- `validateValue:forKey:error:` (page 30)
  Validates a property value for a given key.
- `validateForDelete:` (page 28)
  Determines whether the receiver can be deleted in its current state.
- `validateForInsert:` (page 29)
  Determines whether the receiver can be inserted in its current state.
- `validateForUpdate:` (page 29)
  Determines whether the receiver's current state is valid.

Supporting Key-Value Observing

`+ automaticallyNotifiesObserversForKey:` (page 11)
Returns a Boolean value that indicates whether the receiver provides automatic support for key-value observing change notifications for the given key.

`- didAccessValueForKey:` (page 15)
Provides support for key-value observing access notification.

`- observationInfo` (page 24)
Returns the observation info of the receiver.

`- setObservationInfo:` (page 26)
Sets the observation info of the receiver.

`- willAccessValueForKey:` (page 32)
Provides support for key-value observing access notification.

`- didChangeValueForKey:` (page 16)
Invoked to inform the receiver that the value of a given property has changed.

`- didChangeValueForKey:withSetMutation:usingObjects:` (page 17)
Invoked to inform the receiver that the specified change was made to a specified to-many relationship.

`- willChangeValueForKey:` (page 32)
Invoked to inform the receiver that the value of a given property is about to change.

`- willChangeValueForKey:withSetMutation:usingObjects:` (page 32)
Invoked to inform the receiver that the specified change is about to be made to a specified to-many relationship.

Class Methods

automaticallyNotifiesObserversForKey:

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

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

Parameters

key
The name of one of the receiver's properties.

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

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

Special Considerations

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

contextShouldIgnoreUnmodeledPropertyChanges

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

```
+ (BOOL)contextShouldIgnoreUnmodeledPropertyChanges
```

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

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

Availability
Available in iOS 3.0 and later.

See Also
`- changedValues` (page 14)
`hasChanges` (`NSManagedObjectContext`)

Declared In
`NSManagedObject.h`

Instance Methods

awakeFromFetch

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

```
- (void)awakeFromFetch
```

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

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

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

Availability
Available in iOS 3.0 and later.

See Also
`- awakeFromInsert` (page 13)
`- awakeFromSnapshotEvents:` (page 13)
`- primitiveValueForKey:` (page 25)
`- setPrimitiveValue:forKey:` (page 26)

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 26) or—better—the appropriate custom primitive accessors). This ensures that the new values are treated as baseline values rather than being recorded as undoable changes for the properties in question.

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

Special Considerations

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `awakeFromFetch` (page 12)
- `awakeFromSnapshotEvents:` (page 13)

Declared In
`NSManagedObject.h`

awakeFromSnapshotEvents:

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

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

Parameters

flags

A bitmask of `didChangeValueForKey:` (page 16) constants to denote the event or events that led to the method being invoked.

For possible values, see "NSSnapshotEventType" (page 34).

Discussion

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

If you want to set attribute values and need to avoid emitting key-value observation change notifications, you should use primitive accessor methods (either `setPrimitiveValue:forKey:` (page 26) or—better—the appropriate custom primitive accessors). This ensures that the new values are treated as baseline values rather than being recorded as undoable changes for the properties in question.

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

Availability

Available in iOS 3.0 and later.

See Also

- `awakeFromFetch` (page 12)
- `awakeFromInsert` (page 13)

Declared In

`NSManagedObject.h`

changedValues

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

```
- (NSDictionary *)changedValues
```

Return Value

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

Discussion

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

Availability

Available in iOS 3.0 and later.

See Also

- `committedValuesForKeys:` (page 15)

Declared In

`NSManagedObject.h`

committedValuesForKeys:

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

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

Parameters

keys

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

Return Value

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

Discussion

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

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

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

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

Availability

Available in iOS 3.0 and later.

See Also

- `changedValues` (page 14)

Declared In

`NSManagedObject.h`

dealloc

Deallocates the memory occupied by the receiver.

```
- (void)dealloc
```

Discussion

This method first invokes `didTurnIntoFault` (page 18).

You should typically not override this method—instead you should put "clean-up" code in `prepareForDeletion` (page 24) or `didTurnIntoFault` (page 18).

See Also

- `prepareForDeletion` (page 24)
- `didTurnIntoFault` (page 18)

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

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

Availability

Available in iOS 3.0 and later.

See Also

`- willAccessValueForKey:` (page 32)

Declared In

`NSManagedObject.h`

didChangeValueForKey:

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

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

Parameters

key

The name of the property that changed.

Discussion

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

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObject.h`

didChangeValueForKey:withSetMutation:usingObjects:

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

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

Parameters

inKey

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

inMutationKind

The type of change that was made.

inObjects

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

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

You must not override this method.

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObject.h`

didSave

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

```
- (void)didSave
```

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

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

Note that the sense of "save" in the method name is that of a database commit statement and so applies to deletions as well as to updates to objects. For subclasses, this method is therefore an appropriate locus for code to be executed when an object deleted as well as "saved to disk." You can find out if an object is marked for deletion with `isDeleted` (page 20).

Special Considerations

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

Availability
Available in iOS 3.0 and later.

See Also
- `willSave` (page 33)

Declared In
`NSManagedObject.h`

didTurnIntoFault

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

```
- (void)didTurnIntoFault
```

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

Availability
Available in iOS 3.0 and later.

See Also
- `willTurnIntoFault` (page 34)

Declared In
`NSManagedObject.h`

entity

Returns the entity description of the receiver.

```
- (NSEntityDescription *)entity
```

Return Value
The entity description of the receiver.

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

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObject.h`

faultingState

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

```
- (NSUInteger)faultingState
```

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

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

Availability
Available in iOS 3.0 and later.

See Also
`- isFault` (page 21)

Declared In
`NSManagedObject.h`

hasFaultForRelationshipNamed:

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

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

Parameters
key
The name of one of the receiver's relationships.

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

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

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObject.h`

initWithEntity:insertIntoManagedObjectContext:

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

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

Parameters
entity
The entity of which to create an instance.

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

context
The context into which the new instance is inserted.

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

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

If *context* is not `nil`, this method invokes `[context insertObject:self]` (which causes awakeFromInsert (page 13) to be invoked).

You are discouraged from overriding this method—you should instead override awakeFromInsert (page 13) and/or awakeFromFetch (page 12) (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:`. If your application has multiple stores and you do need to assign an object to a specific store, you should not do so in a managed object's initializer method. Such an assignment is controller- not model-level logic.

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

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

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

Availability
Available in iOS 3.0 and later.

See Also
`insertNewObjectForEntityForName:inManagedObjectContext:`

Declared In
`NSManagedObject.h`

isDeleted

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

```
- (BOOL)isDeleted
```

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

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `- isFault` (page 21)
- `- isInserted` (page 22)
- `- isUpdated` (page 22)

`deletedObjects` (`NSManagedObjectContext`)

`NSManagedObjectContextObjectsDidChangeNotification` (`NSManagedObjectContext`)

Declared In
`NSManagedObject.h`

isFault

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

```
- (BOOL)isFault
```

Return Value
`YES` if the receiver is a fault, otherwise `NO`.

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

If this method returns `NO`, then the receiver's data must be in memory. However, if this method returns `YES`, it does *not* imply that the data is not in memory. The data may be in memory, or it may not, depending on many factors influencing caching

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

Availability
Available in iOS 3.0 and later.

See Also
- `- faultingState` (page 18)
- `- isDeleted` (page 20)
- `- isInserted` (page 22)
- `- isUpdated` (page 22)

Declared In
`NSManagedObject.h`

isInserted

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

```
- (BOOL)isInserted
```

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `isDeleted` (page 20)
- `isFault` (page 21)
- `isUpdated` (page 22)

Declared In
`NSManagedObject.h`

isUpdated

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

```
- (BOOL)isUpdated
```

Return Value
`YES` if the receiver has unsaved changes, otherwise `NO`.

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `isDeleted` (page 20)
- `isFault` (page 21)
- `isInserted` (page 22)

Declared In
`NSManagedObject.h`

managedObjectContext

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

```
- (NSManagedObjectContext *)managedObjectContext
```

Return Value
The managed object context with which the receiver is registered.

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

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

Availability
Available in iOS 3.0 and later.

Declared In
`NSManagedObject.h`

mutableSetValueForKey:

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

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

Parameters

key

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

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

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

Important: You must not override this method.

Special Considerations

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

See Also
- `valueForKey:` (page 31)
- `primitiveValueForKey:` (page 25)
- `setObservationInfo:` (page 26)

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 iOS 3.0 and later.

See Also
`URIRepresentation` (`NSManagedObjectID`)

Declared In
`NSManagedObject.h`

observationInfo

Returns the observation info of the receiver.

```
- (id)observationInfo
```

Return Value
The observation info of the receiver.

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

Important: You must not override this method.

Availability
Available in iOS 3.0 and later.

See Also
`- setObservationInfo:` (page 26)

Declared In
`NSManagedObject.h`

prepareForDeletion

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

```
- (void)prepareForDeletion
```

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

Availability
Available in iOS 3.0 and later.

See Also
- `willTurnIntoFault` (page 34)
- `didTurnIntoFault` (page 18)

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 32) and `didAccessValueForKey:` (page 15)). This method is used primarily by subclasses that implement custom accessor methods that need direct access to the receiver's private storage.

Special Considerations

Subclasses should not override this method.

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `setObservationInfo:` (page 26)

- valueForKey: (page 31)
- mutableSetValueForKey: (page 23)

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 32).

setObservationInfo:

Sets the observation info of the receiver.

```
- (void)setObservationInfo:(id)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 iOS 3.0 and later.

See Also
- observationInfo (page 24)

Declared In
NSManagedObject.h

setPrimitiveValue:forKey:

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

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

Parameters

value

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

key

The name of one of the receiver's properties.

Discussion

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

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

Special Considerations

You must not override this method.

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

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

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

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

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

Availability

Available in iOS 3.0 and later.

See Also

- `primitiveValueForKey:` (page 25)
- `valueForKey:` (page 31)
- `mutableSetValueForKey:` (page 23)
- `awakeFromFetch` (page 12)

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: You must not override this method.

Availability

Available in iOS 3.0 and later.

See Also

- `valueForKey:` (page 31)
- `primitiveValueForKey:` (page 25)
- `setObservationInfo:` (page 26)

Declared In

`NSManagedObject.h`

validateForDelete:

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

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

Parameters

error

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

Return Value

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

Discussion

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

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

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

Availability
Available in iOS 3.0 and later.

See Also
- `validateForInsert:` (page 29)
- `validateForUpdate:` (page 29)
- `validateValue:forKey:error:` (page 30)

Declared In
`NSManagedObject.h`

validateForInsert:

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

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

Parameters

error

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

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

Special Considerations

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

Availability
Available in iOS 3.0 and later.

See Also
- `validateForDelete:` (page 28)
- `validateForUpdate:` (page 29)
- `validateValue:forKey:error:` (page 30)

Declared In
`NSManagedObject.h`

validateForUpdate:

Determines whether the receiver's current state is valid.

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

Parameters

error

If the receiver's current state is invalid, upon return contains an instance of `NSError` that describes the problem.

Return Value

`YES` if the receiver's current state is valid, otherwise `NO`.

Discussion

`NSManagedObject`'s implementation iterates through all of the receiver's properties validating each in turn. If this results in more than one error, the *`userInfo`* dictionary in the `NSError` returned in *`error`* contains a key `NSDetailedErrorsKey`; the corresponding value is an array containing the individual validation errors. If you pass `NULL` as the error, validation will abort after the first failure.

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

Availability

Available in iOS 3.0 and later.

See Also

- `validateForDelete:` (page 28)
- `validateForInsert:` (page 29)
- `validateValue:forKey:error:` (page 30)

Declared In

`NSManagedObject.h`

validateValue:forKey:error:

Validates a property value for a given key.

```
- (BOOL)validateValue:(id *)value forKey:(NSString *)key error:(NSError **)error
```

Parameters

value

A pointer to an object.

key

The name of one of the receiver's properties.

error

If *`value`* is not a valid value for *`key`* (and cannot be coerced), upon return contains an instance of `NSError` that describes the problem.

Return Value

`YES` if *`value`* is a valid value for *`key`* (or if *`value`* can be coerced into a valid value for *`key`*), otherwise `NO`.

Discussion

This method is responsible for two things: coercing the value into an appropriate type for the object, and validating it according to the object's rules.

The default implementation provided by `NSManagedObject` consults the object's entity description to coerce the value and to check for basic errors, such as a null value when that isn't allowed and the length of strings when a field width is specified for the attribute. It then searches for a method of the form `validate<Key>:error:` and invokes it if it exists.

You can implement methods of the form `validate<Key>:error:` to perform validation that is not possible using the constraints available in the property description. If it finds an unacceptable value, your validation method should return `NO` and in *error* an `NSError` object that describes the problem. For more details, see "Model Object Validation". For inter-property validation (to check for combinations of values that are invalid), see `validateForUpdate:` (page 29) and related methods.

Availability
Available in iOS 3.0 and later.

See Also
- `validateForDelete:` (page 28)
- `validateForInsert:` (page 29)
- `validateForUpdate:` (page 29)

Declared In
`NSManagedObject.h`

valueForKey:

Returns the value for the property specified by *key*.

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

Parameters

key

The name of one of the receiver's properties.

Return Value
The value of the property specified by *key*.

Discussion
If *key* is not a property defined by the model, the method raises an exception. This method is overridden by `NSManagedObject` to access the managed object's generic dictionary storage unless the receiver's class explicitly provides key-value coding compliant accessor methods for *key*.

Important: You must not override this method.

Availability
Available in iOS 3.0 and later.

See Also
- `primitiveValueForKey:` (page 25)
- `setValue:forKey:` (page 28)
- `setObservationInfo:` (page 26)

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 15) for more details. You can invoke this method with the *key* value of `nil` to ensure that a fault has been fired, as illustrated by the following example.

```
[aManagedObject willAccessValueForKey:nil];
```

Availability

Available in iOS 3.0 and later.

See Also

`- didAccessValueForKey:` (page 15)

Declared In

`NSManagedObject.h`

willChangeValueForKey:

Invoked to inform the receiver that the value of a given property is about to change.

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

Parameters

key

The name of the property that will change.

Discussion

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

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObject.h`

willChangeValueForKey:withSetMutation:usingObjects:

Invoked to inform the receiver that the specified change is about to be made to a specified to-many relationship.

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

Parameters

inKey

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

inMutationKind

The type of change that will be made.

inObjects

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

Discussion

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

You must not override this method.

Availability

Available in iOS 3.0 and later.

Declared In

`NSManagedObject.h`

willSave

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

```
- (void)willSave
```

Discussion

This method can have "side effects" on persistent values. You can use it to, for example, compute persistent values from other transient or scratchpad values.

If you want to update a persistent property value, you should typically test for equality of any new value with the existing value before making a change. If you change property values using standard accessor methods, Core Data will observe the resultant change notification and so invoke `willSave` again before saving the object's managed object context. If you continue to modify a value in `willSave`, `willSave` will continue to be called until your program crashes.

For example, if you set a last-modified timestamp, you should check whether either you previously set it in the same save operation, or that the existing timestamp is not less than a small delta from the current time. Typically it's better to calculate the timestamp once for all the objects being saved (for example, in response to an `NSManagedObjectContextWillSaveNotification`).

If you change property values using primitive accessors, you avoid the possibility of infinite recursion, but Core Data will not notice the change you make.

Note that the sense of "save" in the method name is that of a database commit statement and so applies to deletions as well as to updates to objects. For subclasses, this method is therefore an appropriate locus for code to be executed when an object deleted as well as "saved to disk." You can find out if an object is marked for deletion with `isDeleted` (page 20).

Availability

Available in iOS 3.0 and later.

See Also

`- didSave` (page 17)

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 18) method. You can use it to (re)set state which requires access to property values (for example, observers across keypaths). The default implementation does nothing.

Availability
Available in iOS 3.0 and later.

See Also
- `didTurnIntoFault` (page 18)

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.

NSSnapshotEventType

Constants returned from `awakeFromSnapshotEvents:` (page 13) to denote the reason why a managed object may need to reinitialize values.

```
enum {
        NSSnapshotEventUndoInsertion = 1 << 1,
        NSSnapshotEventUndoDeletion = 1 << 2,
        NSSnapshotEventUndoUpdate = 1 << 3,
        NSSnapshotEventRollback = 1 << 4,
        NSSnapshotEventRefresh = 1 << 5,
        NSSnapshotEventMergePolicy = 1 << 6
};
typedef NSUInteger NSSnapshotEventType;
```

Constants

`NSSnapshotEventUndoInsertion`

Specifies a change due to undo from insertion.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventUndoDeletion`

Specifies a change due to undo from deletion.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventUndoUpdate`

Specifies a change due to a property-level undo.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventRollback`

Specifies a change due to the managed object context being rolled back.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventRefresh`

Specifies a change due to the managed object being refreshed.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

`NSSnapshotEventMergePolicy`

Specifies a change due to conflict resolution during a save operation.

Available in iOS 3.0 and later.

Declared in `NSManagedObject.h`.

Document Revision History

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

Date	Notes
2010-05-25	Edited overview.
2009-05-05	Corrected typographical errors.
2009-02-20	Updated for iOS 3.0.
2008-10-15	Clarified descriptions of primitiveValueForKey: and setPrimitiveValue:forKey:.
2007-10-31	Enhanced the discussion of working with Objective-C 2.0 features.
2007-08-23	Corrected description of isDeleted method.
2007-07-19	Enhanced subclassing notes for Mac OS X v10.5.
2007-03-06	Augmented discussion of init method.
2007-02-08	Corrected the description of the awakeFromInsert method.
2006-09-05	Enhanced subclassing notes.
Leopard WWDC	Updated for Mac OS X v10.5.
2006-05-23	First publication of this content as a separate document.

Document Revision History