The EOControl Framework

Framework: System/Library/Frameworks/EOInterface.framework

Header File Directories: System/Library/Frameworks/EOInterface.framework/Headers

Introduction

The EOInterface framework defines one of the layers of the Enterprise Objects Framework architecture—the control layer. It provides an infrastructure for enterprise objects that is independent of your application's user interface and its storage mechanism. The control layer dynamically manages the interaction between enterprise objects, the access layer, and the interface layer by:

- Tracking changes to enteprise objects
- Prompting the user interface to change when object values change
- Prompting the database to change when changes to objects are committed
- Managing undo in the object graph
- Managing uniquing (the mechanism by which Enterprise Objects Framework uniquely identifies enterprise objects and maintains their mapping to stored data in the database)

The control layer's major areas of responsibility and the key classes involved are described in the following table:

Responsibility	Classes
Tracking Enterprise Objects Changes	EOObserverCenter EODelayedObserverQueue EODelayedObserver EOObserverProxy EOObserving (protocol)
Object Storage Abstraction	EOObjectStore EOCooperatingObjectStore EOObjectStoreCoordinator EOGlobalID EOKeyGlobalID EOTemporaryGlobalID

Responsibility	Classes
Query specification	EOFetchSpecification EOQualifier EOSortOrdering
Interaction with enterprise objects	EOClassDescription (validation) NSObjectAdditions (basic enterprise object behavior)
Simple source of objects (for display groups)	EODataSource, EODetailDataSource

The following sections describe each responsibility in greater detail.

Tracking Enterprise Objects Changes

EOControl provides four classes and a protocol that form an efficient, specialized mechanism for tracking changes to enterprise objects and for managing the notification of those changes to interested observers. EOObserverCenter is the central manager of change notification. It records observers and the objects they observe, and it distributes notifications when the observable objects change. Observers implement the EOObserving protocol, which defines one method, **objectWillChange:**. Observable objects (generally enterprise objects) invoke their **willChange** method before altering their state, which causes all observers to receive an **objectWillChange:** message.

The other three classes add to the basic observation mechanism. EODelayedObserverQueue alters the basic, synchronous change notification mechanism by offering different priority levels, which allows observers to specify the order in which they're notified of changes. EODelayedObserver is an abstract superclass for objects that observe other objects (such as the EOInterface layer's EOAssociation classes). Finally, EOObserverProxy is a subclass of EODelayedObserver that forwards change messages to a target object, allowing objects that don't inherit from EODelayedObserver to take advantage of this mechanism.

The major observer in Enterprise Objects Framework is EOEditingContext, which implements its **objectWillChange:** method to record a snapshot for the object about to change, register undo operations in an NSUndoManager, and record the changes needed to update objects in its EOObjectStore. Because some of these actions—such as examining the object's new state—can only be performed after the object has changed, an EOEditingContext sets up a delayed message to itself, which it gets at the end of the run loop. Observers that only need to examine an object after it has changed can use the delayed observer mechanism, described in the EODelayedObserver and EODelayedObserverQueue class specifications.

Object Storage Abstraction

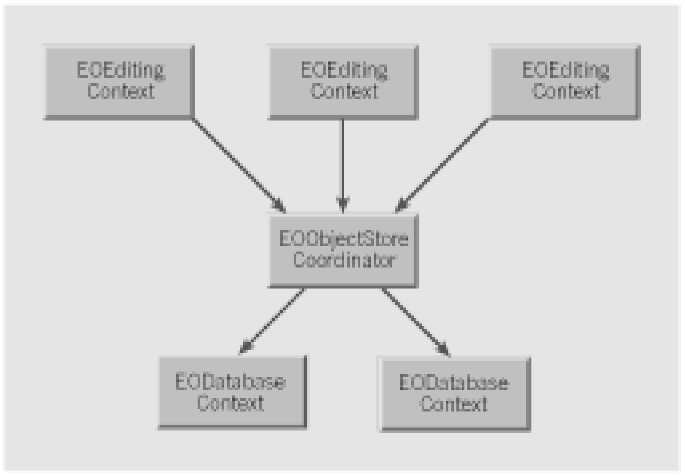
The control layer provides an infrastructure that's independent of your application's storage mechanism (typically a database) by defining an API for an "intelligent" repository of objects, whether it's based on external data or whether it manages objects entirely in memory. EOObjectStore is an abstract class that defines that basic API, setting up the framework for constructing and registering enterprise objects, servicing object faults, and committing changes made in an EOEditingContext. Subclasses of EOObjectStore implement the API in terms of their specific storage mechanism.

Subclasses of EOObjectStore

EOEditingContext is the principal subclass of EOObjectStore and is used for managing objects in memory. For stores based on external data, there are several subclasses. EOCooperatingObjectStore defines stores that work together to manage data from several distinct sources (such as different databases). The access layer's EODatabaseContext is actually a subclass of this class. A group of cooperating stores is managed by another subclass of EOObjectStore, EOObjectStoreCoordinator. If you're defining a subclass of EOObjectStore, it's probably one based on an external data repository, and it should therefore inherit from EOCooperatingObjectStore so as to work well with an EOObjectStoreCoordinator—though this isn't required.

EODatabaseContext provides objects from relational databases and is therefore provided by Enterprise Objects Framework's access layer. It is the class that defines the interaction between the control and access layers. Database contexts and other object stores based on external data are often shared by several editing contexts to conserve database connections.

Object store subclasses cooperate with one another as illustrated in the following:



Registering Enterprise Objects

An object store identifies its objects in two ways:

- By reference for identification within a specific editing context
- By global ID for universal identification of the same record among multiple stores.

A global ID is defined by three classes: EOGlobalID, EOKeyGlobalID, and EOTemporaryGlobalID. EOGlobalID is an abstract class that forms the basis for uniquing in Enterprise Objects Framework.

EOKeyGlobalID is a concrete subclass of EOGlobalID whose instances represent persistent IDs based on the access layer's EOModel information: an entity and the primary key values for the object being identified. An EOTemporaryGlobalID object is used to identify a newly created enterprise object before it's saved to an external store. For more information, see the EOGlobalID class specification.

Servicing Faults

For external repositories, an object store might delay fetching an object's data, instead creating an EOFault as a placeholder. When a fault is accessed (sent a message), it triggers its object store to fetch its data and transform it into an instance of the appropriate object class. This preserves both the object's **id** and its EOGlobalID, while saving the cost of fetching data that might not be used. Faults are typically created for the destinations of relationships for objects that are explicitly fetched. See the EOFault and EOFaultHandler class specifications for more information.

EOArrayDataSource

Inherits From:	EODataSource : NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	EOControl/EOArrayDataSource.h

Class Description

EOArrayDataSource is a concrete subclass of EODataSource that can be used to provide enterprise objects to a display group (EODisplayGroup from EOInterface or WODisplayGroup from WebObjects) without having to fetch them from the database. In an EOArrayDataSource, objects are maintained in an in-memory NSArray.

EOArrayDataSource can fetch, insert, and delete objects—operations it performs directly with its array. It can also provide a detail data source.

Adopted Protocols

NSCoding	
	encodeWithCoder:
	initWithCoder:

Instance Methods

initWithClassDescription:editingContext:

– **initWithClassDescription:**(EOClassDescription *)*classDescription*
  editingContext:(EOEditingContext *)*editingContext*

The designated initializer of the EOArrayDataSource class, this method initializes a newly allocated EOArrayDataSource object with *classDescription* and *editingContext*, both of which it retains. *classDescription* contains information about the objects provided by the EOArrayDataSource and *editingContext* is the EOArrayDataSource's EOEditingContext. Either argument may be nil. Returns **self**.

setArray:

– (void)**setArray:**(NSArray *)*array*

Sets the receiver’s array of objects to *array.*

EOAndQualifier

Inherits From:	EOQualifier : NSObject
Conforms To:	EOQualifierEvaluation EOQualifierSQLGeneration
Declared In:	EOControl/EOQualifier.h

Class Description

EOAndQualifier is a subclass of EOQualifier that contains multiple qualifiers. EOAndQualifier adopts the EOQualifierEvaluation protocol, which defines the method **evaluateWithObject:** for in-memory evaluation. When an EOAndQualifier object receives an **evaluateWithObject:** message, it evaluates each of its qualifiers until one of them returns NO. If one of its qualifiers returns NO, the EOAndQualifier object returns NO immediately. If all of its qualifiers return YES, the EOAndQualifier object returnsYES.

Adopted Protocols

EOQualifierEvaluation	
	– evaluateWithObject:
EOQualifierSQLGeneration	
	– sqlStringForSQLExpression:
	– schemaBasedQualifierWithRootEntity:

Instance Methods

evaluateWithObject:

@protocol EOQualifierEvaluation
– (BOOL)**evaluateWithObject:**(id)*anObject*

Returns YES if *anObject* satisfies the qualifier, NO otherwise. When an EOAndQualifier object receives an **evaluateWithObject:** message, it evaluates each of its qualifiers until one of them returns NO. If any of its qualifiers returns NO, the EOAndQualifier object returns NO immediately. If all of its qualifiers return YES, the object returns YES. This method can raise one of several possible exceptions if an error occurs. If your application allows users to construct arbitrary qualifiers (such as through a user interface), you may want to write code to catch any exceptions and properly respond to errors (for example, by displaying a panel saying that the user typed a poorly formed qualifier).

initWithQualifierArray:

– **initWithQualifierArray:**(NSArray *)*qualifiers*

Initializes the receiver with the qualifiers *qualifiers* and returns **self**. This method is the designated initializer for EOAndQualifier.

initWithQualifiers:

– **initWithQualifiers:**(EOQualifier *)*qualifiers, ...*

Initializes the receiver with the **nil**-terminated list of qualifiers *qualifiers*. Works by invoking **initWithQualifierArray:**. For example, the following code excerpt constructs two qualifiers, **qual1** and **qual2**. It then uses these qualifiers to initialize an EOAndQualifier, **andQual**. **andQual** is then used to filter an in-memory array.

```
NSArray *guests;    /* Assume this exists. */
EOQualifier *qual1, *qual2, *andQual;

qual1 = [EOQualifier qualifierWithQualifierFormat:@"lastName = 'Nunez'"];
qual2 = [EOQualifier qualifierWithQualifierFormat:@"firstName = 'Maria'"];
andQual = [[EOAndQualifier alloc] initWithQualifiers:qual1, qual2, nil];
return [guests filteredArrayUsingQualifier:andQual];
```

qualifiers

– (NSArray *)**qualifiers**

Returns the receiver’s qualifiers.

EOClassDescription

Inherits From:	NSObject
Declared In:	EOControl/EOClassDescription.h

Class Description

The EOClassDescription class provides a mechanism for extending classes by giving them access to metadata not available in the run-time system. This is achieved as follows:

- EOClassDescription provides a bridge between enterprise objects and the metadata contained in an external source of information, such as an EOModel (EOAccess). It defines a standard API for accessing the information in an external source. It also manages the registration of EOClassDescription objects in your application.
- The EOEnterpriseObject informal protocol declares several EOClassDescription-related methods that define basic enterprise objects behavior, such as undo and validation. The Enterprise Objects Framework extends NSObject by providing implementations of these methods. An enterprise object class can either accept the default implementations or it can provide its own implementation by overriding. This is discussed in more detail in the section “Using EOClassDescription.”

Enterprise Objects Framework implements a default subclass of EOClassDescription in EOAccess, EOEntityClassDescription. EOEntityClassDescription extends the behavior of enterprise objects by deriving information about them (such as NULL constraints and referential integrity rules) from an associated EOModel.

For more information on using EOClassDescription, see the sections

- How Does It Work?
- Using EOClassDescription
- EOEntityClassDescription
- The EOClassDescription’s Delegate

Method Types

Managing EOClassDescriptions	
	+ invalidateClassDescriptionCache
	+ registerClassDescription:forClass:

Getting EOClassDescriptions	
	+ classDescriptionForClass:
	+ classDescriptionForEntityName:
Creating new object instances	
	– createInstanceWithEditingContext:globalID:zone:
Propagating delete	
	– propagateDeleteForObject:editingContext:
Returning information from the EOClassDescription	
	– entityName
	– attributeKeys
	– classDescriptionForDestinationKey:
	– toManyRelationshipKeys
	– toOneRelationshipKeys
	– inverseForRelationshipKey:
	– ownsDestinationObjectsForRelationshipKey:
	– deleteRuleForRelationshipKey:
Performing validation	
	– validateObjectForDelete:
	– validateObjectForSave:
	– validateValue:forKey:
Providing default characteristics for key display	
	– defaultFormatterForKey:
	– defaultFormatterForKeyPath:
	– displayNameForKey:
Handling newly inserted and newly fetched objects	
	– awakeObject:fromFetchInEditingContext:
	– awakeObject:fromInsertionInEditingContext:
Setting the delegate	
	+ classDelegate
	+ setClassDelegate:
Getting an object’s description	
	– userPresentableDescriptionForObject:

Class Methods

classDelegate

+ (id)**classDelegate**

Returns the delegate for the EOClassDescription class (as opposed to EOClassDescription instances).

See also: + **setClassDelegate:**

classDescriptionForClass:

+ (EOClassDescription *)**classDescriptionForClass:**(Class)*aClass*

Invoked by the default implementations of the EOEnterpriseObject informal protocol method **classDescription** to return the EOClassDescription for *aClass*. It's generally not safe to use this method directly—for example, individual EOGenericRecord instances can have different class descriptions. If a class description for *aClass* isn't found, this method posts an EOClassDescriptionNeededForClassNotification on behalf of the receiver's class, allowing an observer to register a an EOClassDescription.

classDescriptionForEntityName:

+ (EOClassDescription *)**classDescriptionForEntityName:**(NSString *)*entityName*

Returns the EOClassDescription registered under *entityName*.

invalidateClassDescriptionCache

+ (void)**invalidateClassDescriptionCache**

Flushes the EOClassDescription cache. Because the EOModel objects in an application supply and register EOClassDescriptions on demand, the cache continues to be repopulated as needed after you invalidate it. (The EOModel class is defined in EOAccess.)

You'd use this method when a provider of EOClassDescriptions (such as an EOModel) has newly become available, or is about to go away. However, you should rarely need to directly invoke this method unless you're using an external source of information other than an EOModel.

registerClassDescription:forClass:

+ (void)**registerClassDescription:**(EOClassDescription *)*description* **forClass:**(Class)*class*

Registers an EOClassDescription object for *class* in the EOClassDescription cache. You should rarely need to directly invoke this method unless you're using an external source of information other than an EOModel (EOAccess).

setClassDelegate:

+ (void)**setClassDelegate:**(id)*delegate*

Sets the delegate for the EOClassDescription class (as opposed to EOClassDescription instances) to *delegate*, without retaining it. For more information on the class delegate, see the EOClassDescriptionClassDelegate informal protocol specification.

See also: + **classDelegate**

Instance Methods

attributeKeys

– (NSArray *)**attributeKeys**

Overridden by subclasses to return an array of attribute keys (NSStrings) for objects described by the receiver. "Attributes" contain immutable data (such as NSNumbers and NSStrings), as opposed to "relationships" that are references to other enterprise objects. For example, a class description that describes Movie objects could return the attribute keys "title," "dateReleased," and "rating."

EOClassDescription's implementation of this method simply returns .

See also: – **entityName**, – **toOneRelationshipKeys,** – **toManyRelationshipKeys**

awakeObject:fromFetchInEditingContext:

– (void)**awakeObject:**(id)*object*
    fromFetchInEditingContext:(EOEditingContext *)*anEditingContext*

Overridden by subclasses to perform standard post-fetch initialization for *object* in *anEditingContext.* EOClassDescription's implementation of this method does nothing.

awakeObject:fromInsertionInEditingContext:

– (void)**awakeObject:**(id)*object*
    fromInsertionInEditingContext:(EOEditingContext *)*anEditingContext*

Assigns empty arrays to to-many relationship properties of newly inserted enterprise objects. Can be overridden by subclasses to propagate inserts for the newly inserted *object* in *anEditingContext*. More specifically, if *object* has a relationship (or relationships) that propagates the object's primary key and if no object yet exists at the destination of that relationship, subclasses should create the new object at the destination of the relationship. Use this method to put default values in your enterprise object.

classDescriptionForDestinationKey:

– (EOClassDescription *)**classDescriptionForDestinationKey:**(NSString *)*detailKey*

Overridden by subclasses to return the class description for objects at the destination of the to-one relationship identified by *detailKey*. For example, the statement:

```
[movie classDescriptionForDestinationKey:@"studio"]
```

might return the class description for the Studio class. EOClassDescription's implementation of this method returns **nil**.

createInstanceWithEditingContext:globalID:zone:

– (id)**createInstanceWithEditingContext:**(EOEditingContext *)*anEditingContext*
    globalID:(EOGlobalID *)*globalID*
    zone:(NSZone *)*zone*

Overridden by subclasses to create an object of the appropriate class in *anEditingContext* with *globalID* and in *zone*. In typical usage, all three of the method's arguments are **nil**. If the object responds to **initWithEditingContext:classDescription:globalID** subclasses should invoke that method, otherwise they should invoke **init**. Implementations of this method should return an autoreleased object. Enterprise Objects Framework uses this method to create new instances of objects when fetching existing enterprise objects or inserting new ones in an interface layer EODisplayGroup. EOClassDescription's implementation of this method returns **nil**.

defaultFormatterForKey:

– (NSFormatter *)**defaultFormatterForKey:**(NSString *)*key*

Returns the default NSFormatter to use when parsing values for assignment to *key*. EOClassDescription's implementation returns **nil**. The access layer's EOEntityClassDescription's implementation returns an NSFormatter based on the Objective-C valueClass specified for *key* in the associated model file. Code that creates a user interface, like a wizard, can use this method to assign formatters to user interface elements.

defaultFormatterForKeyPath:

– (NSFormatter *)**defaultFormatterForKeyPath:**(NSString *)*keyPath*

Similar to **defaultFormatterForKey:**, except this method traverses *keyPath* and returns the formatter for the key at the end of the path (using **defaultFormatterForKey:**).

deleteRuleForRelationshipKey:

– (EODeleteRule)**deleteRuleForRelationshipKey:**(NSString *)*relationshipKey*

Overridden by subclasses to return a delete rule indicating how to treat the destination of the given relationship when the receiving object is deleted. The delete rule is one of:

Constant	Description
EODeleteRuleNullify	When the source object is deleted, any references a destination object has to the source are removed or “nullified.” For example, suppose a department has a to-many relationship to multiple employees. When the department is deleted, any back references an employee has to the department are set to nil.
EODeleteRuleCascade	When the source object (department) is deleted, any destination objects (employees) are also deleted.
EODeleteRuleDeny	If the source object (department) has any destination objects (employees), a delete operation is refused.
EODeleteRuleNoAction	When the source object is deleted, its relationship is ignored and no action is taken to propagate the deletion to destination objects. This rule is useful for tuning performance. To perform a deletion, Enterprise Objects Framework fires all the faults of the deleted object and then fires any to-many faults that point back to the deleted object. For example, suppose you have a simple application based on the sample Movies database. Deleting a Movie object has the effect of firing a to-one fault for the Movie’s **studio** relationship, and then firing the to-many **movies** fault for that studio. In this scenario, it would make sense to set the delete rule EODeleteRuleNoAction for Movie’s **studio** relationship. However, you should use this delete rule with great caution since it can result in dangling references in your object graph.

EOClassDescription’s implementation of this method returns the delete rule EODeleteRuleNullify. In the common case, the delete rule for an enterprise object is defined in its EOModel. (The EOModel class is defined in EOAccess.)

See also: – **propagateDeleteWithEditingContext:** (EOEnterpriseObject)

displayNameForKey:

– (NSString *)**displayNameForKey:**(NSString *)*key*

Returns the default string to use in the user interface when displaying *key*. By convention, lowercase words are capitalized (for example, “revenue” becomes “Revenue”), and spaces are inserted into words with mixed case (for example, “firstName” becomes “First Name”). This method is useful if you’re creating a user interface from only a class description, such as with a wizard or a Direct To Web application.

entityName

– (NSString *)**entityName**

Overridden by subclasses to return a unique type name for objects of this class. For example, the access layer’s EOEntityClassDescription returns its EOEntity’s name. EOClassDescription’s implementation of this method returns **nil**.

See also: – **attributeKeys**, – **toOneRelationshipKeys,** – **toManyRelationshipKeys**

inverseForRelationshipKey:

– (NSString *)**inverseForRelationshipKey:**(NSString *)*relationshipKey*

Overridden by subclasses to return the name of the relationship pointing back at the receiver from the destination of the relationship specified by *relationshipKey*. For example, suppose an Employee object has a relationship called **department** to a Department object, and Department has a relationship called **employees** back to Employee. The statement:

```
[employee inverseForRelationshipKey:@"department"]
```

returns the string “employees”.

EOClassDescription’s implementation of this method returns **nil**.

ownsDestinationObjectsForRelationshipKey:

– (BOOL)**ownsDestinationObjectsForRelationshipKey:**(NSString *)*relationshipKey*

Overridden by subclasses to return YES or NO to indicate whether the objects at the destination of the relationship specified by *relationshipKey* should be deleted if they are removed from the relationship (and not transferred to the corresponding relationship of another object). For example, an Invoice object owns its line items. If a LineItem object is removed from an Invoice it should be deleted since it can’t exist outside of an Invoice. EOClassDescription’s implementation of this method returns NO. In the common case, this behavior for an enterprise object is defined in its EOModel. (The EOModel class is defined in EOAccess.)

propagateDeleteForObject:editingContext:

– (void)**propagateDeleteForObject:**(id)*object*
    editingContext:(EOEditingContext *)*anEditingContext*

Propagates a delete operation for *object* in *anEditingContext*, according to the delete rules specified in the EOModel. This method is invoked whenever a delete operation needs to be propagated, as indicated by the delete rule specified for the corresponding EOEntity's relationship key. (The EOModel and EOEntity classes are defined in EOAccess.) For more discussion of delete rules, see the EOEnterpriseObject informal protocol specification.

See also: – **deleteRuleForRelationshipKey:**

toManyRelationshipKeys

– (NSArray *)**toManyRelationshipKeys**

Overridden by subclasses to return the keys for the to-many relationship properties of the receiver. To-many relationship properties contain arrays of enterprise objects. EOClassDescription's implementation of this method returns **nil**.

See also: – **entityName**, – **toOneRelationshipKeys,** – **attributeKeys**

toOneRelationshipKeys

– (NSArray *)**toOneRelationshipKeys**

Overridden by subclasses to return the keys for the to-one relationship properties of the receiver. To-one relationship properties are other enterprise objects. EOClassDescription's implementation of this method returns **nil**.

See also: – **entityName**, – **toManyRelationshipKeys,** – **attributeKeys**

userPresentableDescriptionForObject:

– (NSString *)**userPresentableDescriptionForObject:**(id)*anObject*

Returns a short (no longer than 60 characters) description of *anObject* based on its data. This method enumerates *anObject*'s attributeKeys and returns each attribute's value, separated by commas and with the default formatter applied for numbers and dates.

validateObjectForDelete:

– (NSException *)**validateObjectForDelete:**(id)*object*

Overridden by subclasses to determine whether it's permissible to delete *object*. Subclasses should return **nil** if the delete operation should proceed, or return an exception containing a user-presentable (localized) error message if not. EOClassDescription's implementation of this method returns **nil**.

validateObjectForSave:

– (NSException *)**validateObjectForSave:**(id)*object*

Overridden by subclasses to determine whether the values being saved for *object* are acceptable. Subclasses should return **nil** if the values are acceptable and the save operation should proceed, or return an exception containing a user-presentable (localized) error message if not. EOClassDescription's implementation of this method returns **nil**.

validateValue:forKey:

– (NSException *)**validateValue:**(id *)*valueP* **forKey:**(NSString *)*key*

Overridden by subclasses to validate the value pointed to by *valueP*. Subclasses should return **nil** if the value is acceptable, or return an exception containing a user-presentable (localized) error message if not. Implementations can replace *valueP* with a converted value (for example, an EOAttribute might convert an NSString to an NSNumber). EOClassDescription's implementation of this method returns **nil**.

Notifications

The following notifications are declared by EOClassDescription and posted by enterprise objects in your application.

EOClassDescriptionNeededForClassNotification

One of the EOClassDescription-related methods that Enterprise Objects Framework adds to NSObject to extend the behavior of enterprise objects is classDescription. The first time an enterprise object receives a classDescription message (for example, when changes to the object are being saved to the database), it posts EOClassDescriptionNeededForClassNotification to notify observers that a class description is needed. The observer then locates the appropriate class description and registers it in the application. By default, EOModel objects are registered as observers for this notification and register EOClassDescriptions on demand.

Notification Object	Enterprise object class

userInfo Dictionary	None

EOClassDescriptionNeededForEntityNameNotification

When **classDescriptionForEntityName:** is invoked for a previously unregistered entity name, this notification is broadcast with the requested entity name as the object of the notification. By default, EOModel objects are registered as observers for this notification and register EOClassDescriptions on demand.

Notification Object	Entity name (NSString)
userInfo Dictionary	None

EOClassDescription

How Does It Work?

As noted above, Enterprise Objects Framework implements a default subclass of EOClassDescription in EOAccess, EOEntityClassDescription. In the typical scenario in which an enterprise object has a corresponding model file, a particular operation (such as validating a value) results in the broadcast of an EOClassDescriptionNeeded... notification (an EOClassDescriptionNeededForClassNotification or an EOClassDescriptionNeededForEntityNameNotification). When an EOModel object receives such a notification, it registers the metadata (class description) for the EOEntity on which the enterprise object is based. (EOModel and EOEntity are defined in EOAccess.)

An enterprise object takes advantage of the metadata registered for it by using the EOClassDescription-related methods defined in the EOEnterpriseObject informal protocol (and implemented in a category of NSObject). Primary among these methods is **classDescription**, which returns the class description associated with the enterprise object. Through this class description the enterprise object has access to all of the information relating to its entity in a model file.

In addition to methods that return information based on an enterprise object's class description, the EOClassDescription-related methods the EnterpriseObject informal protocol defines include methods that are automatically invoked when a particular operation occurs. These include validation methods and methods that are invoked whenever an enterprise object is inserted or fetched.

All of this comes together in your running application. When a user tries to perform a particular operation on an enterprise object (such as attempting to delete it), the EOEditingContext sends these validation messages to your enterprise object, which in turn (by default) forwards them to its EOClassDescription. Based on the result, the operation is either accepted or refused. For example, referential integrity constraints in your model might state that you can't delete a department object that has employees. If a user attempts to delete a department that has employees, an exception is returned and the deletion is refused.

Using EOClassDescription

For the most part, you don't need to programmatically interact with EOClassDescription. It extends the behavior of your enterprise objects transparently. However, there are two cases in which you do need to programmatically interact with it:

- When you override EOClassDescription-related EOEnterpriseObject methods in an enterprise object class. These methods are used to perform validation and to intervene when enterprise objects based on EOModels are created and fetched. (The EOModel class is defined in EOAccess.) For objects that don't have EOModels, you can override a different set of EOEnterpriseObject methods; this is described in more detail in the section "Working with Objects That Don't Have EOModels."
- When you create a subclass of EOClassDescription

Overriding Methods in an Enterprise Object

As described above, EOEnterpriseObject defines several EOClassDescription-related methods. It’s common for enterprise object classes to override the following methods to either perform validation, to assign default values (**awakeFromInsertionInEditingContext:**), or to provide additional initialization to newly fetched objects (**awakeFromFetchInEditingContext:**):

- validateForSave
- validateForDelete
- validateForInsert
- validateForUpdate
- awakeFromInsertionInEditingContext:
- awakeFromFetchInEditingContext:
- userPresentableDescriptionForObject:

For example, an enterprise object class can implement a **validateForSave** method that checks the values of **salary** and **jobLevel** properties before allowing the values to be saved to the database:

```
- (NSException *)validateForSave
{
    if (salary > 1500 && jobLevel < 2)
        return [NSException validationExceptionWithFormat:
            @"The salary is too high for that position!"];
    // pass the check on to the EOClassDescription
    return [super validateForSave];
}
```

For more discussion of this subject, see the chapter “Designing Enterprise Objects” in the *Enterprise Objects Framework Developer’s Guide*, and the EOEnterpriseObject informal protocol specification.

Working with Objects That Don’t Have EOModels

Although an EOModel is the most common source of an EOClassDescription for a class, it isn’t the only one. Objects that don’t have an EOModel can implement EOClassDescription methods directly as instance methods, and the rest of the Framework will treat them just as it does enterprise objects that have this information provided by an external EOModel.

There are a few reasons you might want to do this. First of all, if your object implements the methods **entityName**, **attributeKeys**, **toOneRelationshipKeys**, and **toManyRelationshipKeys**, EOEditingContexts can snapshot the object and thereby provide undo for it.

For example, the following code excerpt shows an implementation of **attributeKeys** for a Circle class:

```
- (NSArray *)attributeKeys {
    static NSArray *array = nil;
    if (!array)
        array = [[NSArray alloc] initWithObjects:@"radius", @"x",
          @"y", @"color", nil];
    return array;
}
```

Secondly, you might want to implement EOClassDescription’s validation or referential integrity methods to add these features to your classes.

Implementing EOClassDescription methods on a per-class basis in this way is a good alternative to creating a subclass of EOClassDescription.

Creating a Subclass of EOClassDescription

You create a subclass of EOClassDescription when you want to use an external source of information other than an EOModel to extend your objects. Another possible scenario is if you’ve added information to an EOModel (such as in its user dictionary) and you want that information to become part of your class description—in that case, you’d probably want to create a subclass of the access layer’s EOEntityClassDescription.

When you create a subclass of EOClassDescription, you only need to implement the methods that have significance for your subclass.

If you’re using an external source of information other than an EOModel, you need to decide when to register class descriptions, which you do by invoking the method **registerClassDescription:forClass:**. You can either register class descriptions in response to a EOClassDescriptionNeeded... notification (an EOClassDescriptionNeededForClassNotification or an EOClassDescriptionNeededForEntityNameNotification), or you can register class descriptions at the time you initialize your application (in other words, you can register all potential class descriptions ahead of time). The default implementation in Enterprise Objects Framework is based on responding to the EOClassDescriptionNeeded... notifications. When an EOModel receives one of these notifications, it supplies a class description for the specified class or entity name by invoking **registerClassDescription: forClass:**

EOEntityClassDescription

There are only three methods in EOClassDescription that have meaningful implementations (that is, that don’t either return **nil** or simply return without doing anything): **invalidateClassDescriptionCache**, **registerClassDescription:forClass:**, and **propagateDeleteForObject:editingContext:**. The default behavior of the rest of the methods in Enterprise Objects Framework comes from the implementation in the access layer’s EOClassDescription subclass EOEntityClassDescription. For more information, see the EOEntityClassDescription class specification.

The EOClassDescription’s Delegate

You can assign a delegate to the EOClassDescription class. EOClassDescription sends the message **shouldPropagateDeleteForObject:inEditingContext:forRelationshipKey:** to its delegate when delete propagation is about to take place for a particular object. The delegate can either allow or deny the operation for a specified relationship key. For more information, see the method description for **shouldPropagateDeleteForObject:inEditingContext:forRelationshipKey:**.

EOCooperatingObjectStore

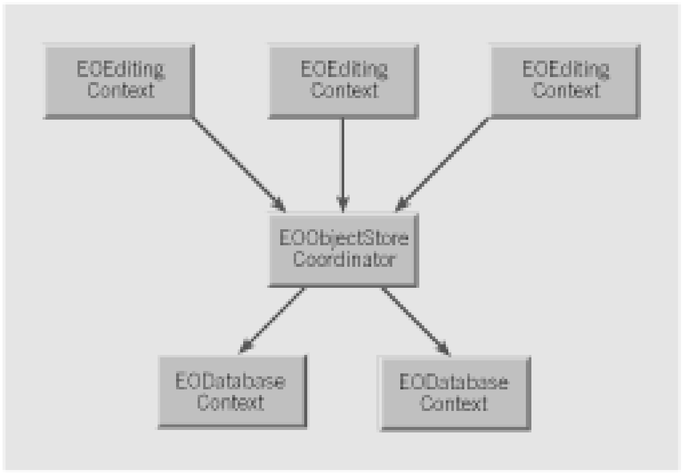
Inherits From:	EOObjectStore : NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOObjectStoreCoordinator.h

Class Description

EOCooperatingObjectStore is a part of the control layer's object storage abstraction. It is an abstract class that defines the basic API for object stores that work together to manage data from several distinct data repositories. For more general information on the object storage abstraction, see "Object Storage Abstraction" in the introduction to the EOControl Framework.

The interaction between EOCooperatingObjectStores is managed by another class, EOObjectStoreCoordinator. The EOObjectStoreCoordinator communicates changes to its EOCooperatingObjectStores by passing them an EOEditingContext. Each cooperating store examines the modified objects in the editing context and determines if it's responsible for handling the changes. When a cooperating store has changes that need to be handled by another store, it communicates the changes to the other store back through the coordinator.

For relational databases, Enterprise Objects Framework provides a concrete subclass of EOCooperatingObjectStore, EODatabaseContext (EOAccess). A database context represents a single connection to a database server, fetching and saving objects on behalf of one or more editing contexts. However, a database context and an editing context don't interact with each other directly—a coordinator acts as a mediator between them.



Method Types

Committing or discarding changes

– commitChanges
– performChanges
– rollbackChanges
– prepareForSaveWithCoordinator:editingContext:
– recordChangesInEditingContext
– recordUpdateForObject:changes:

Returning information about objects

– valuesForKeys:object:

Determining if the EOCooperatingObjectStore is responsible for an operation

– ownsObject:
– ownsGlobalID:
– handlesFetchSpecification:

Instance Methods

commitChanges

– (void)**commitChanges**

Overridden by subclasses to commit the transaction. Raises an exception if an error occurs; the error message indicates the nature of the problem.

See also: – **performChanges**, – **commitChanges**, – **saveChangesInEditingContext:** (EOObjectStoreCoordinator)

handlesFetchSpecification:

– (BOOL)**handlesFetchSpecification:**(EOFetchSpecification *)*fetchSpecification*

Overridden by subclasses to return YES if the receiver is responsible for fetching the objects described by *fetchSpecification*. For example, EODatabaseContext (EOAccess) determines whether it's responsible based on *fetchSpecification*'s entity name.

See also: – **ownsGlobalID:**, – **ownsObject:**

ownsGlobalID:

– (BOOL)**ownsGlobalID:**(EOGlobalID *)*globalID*

Overridden by subclasses to return YES if the receiver is responsible for fetching and saving the object identified by *globalID*. For example, EODatabaseContext (EOAccess) determines whether it's responsible based on the entity associated with *globalID*.

See also: – **handlesFetchSpecification:**, – **ownsObject:**

ownsObject:

– (BOOL)**ownsObject:**(id)*object*

Overridden by subclasses to return YES if the receiver is responsible for fetching and saving *object*. For example, EODatabaseContext (EOAccess) determines whether it's responsible based on the entity associated with *object*.

See also: – **ownsGlobalID:**, – **handlesFetchSpecification:**

performChanges

– (void)**performChanges**

Overridden by subclasses to transmit changes to the receiver's underlying database. Raises an exception if an error occurs; the error message indicates the nature of the problem.

See also: – **commitChanges**, – **rollbackChanges**, – **saveChangesInEditingContext:** (EOObjectStoreCoordinator)

prepareForSaveWithCoordinator:editingContext:

– (void)**prepareForSaveWithCoordinator:**(EOObjectStoreCoordinator *)*coordinator* **editingContext:**(EOEditingContext *)*anEditingContext*

Overridden by subclasses to notify the receiver that a multi-store save operation overseen by *coordinator* is beginning for *anEditingContext*. For example, the receiver might prepare primary keys for newly inserted objects so that they can be handed out to other EOCooperatingObjectStores upon request. The receiver should be prepared to receive the messages **recordChangesInEditingContext** and **recordUpdateForObject:changes:**.

After performing these methods, the receiver should be prepared to receive the possible messages **performChanges** and then **commitChanges** or **rollbackChanges**.

recordChangesInEditingContext

– (void)**recordChangesInEditingContext**

Overridden by subclasses to instruct the receiver to examine the changed objects in the receiver's EOEditingContext, record any operations that need to be performed, and notify the receiver's EOObjectStoreCoordinator of any changes that need to be forwarded to other EOCooperatingObjectStores.

See also: – **prepareForSaveWithCoordinator:editingContext:**, – **recordUpdateForObject:changes:**

recordUpdateForObject:changes:

– (void)**recordUpdateForObject:**(id)*object* **changes:**(NSDictionary *)*changes*

Overridden by subclasses to communicate from one EOCooperatingObjectStore to another (through the EOObjectStoreCoordinator) that *changes* need to be made to an *object*. For example, an insert of an object in a relationship property might require changing a foreign key property in an object owned by another EOCooperatingObjectStore. This method is primarily used to manipulate relationships.

See also: – **prepareForSaveWithCoordinator:editingContext:**, – **recordChangesInEditingContext**

rollbackChanges

– (void)**rollbackChanges**

Overridden by subclasses to roll back changes to the underlying database. Raises one of several possible exceptions if an error occurs; the error message should indicate the nature of the problem.

See also: – **commitChanges**, – **performChanges**, – **saveChangesInEditingContext:** (EOObjectStoreCoordinator)

valuesForKeys:object:

– (NSDictionary *)**valuesForKeys:**(NSArray *)*keys*
object:(id)*object*

Overridden by subclasses to return values (as identified by *keys*) held by the receiver that augment properties in *object*. For instance, an EODatabaseContext (EOAccess) stores foreign keys for the objects it owns (and primary keys for new objects). These foreign and primary keys may well not be defined as properties of the object. Other database contexts can find out these keys by sending the database context that owns the object a **valuesForKeys:object:** message. Note that you use this for properties that are *not* stored in the object, so using key-value coding directly on the object won't always work.

EODataSource

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EODataSource.h

Class Description

EODataSource is an abstract class that defines a basic API for providing enterprise objects. It exists primarily as a simple means for a display group (EODisplayGroup from EOInterface or WODisplayGroup from WebObjects) or other higher-level class to access a store of objects. EODataSource defines functional implementations of very few methods; concrete subclasses, such as EODatabaseDataSource (defined in EOAccess) and EODetailDataSource, define working data sources by implementing the others. EODatabaseDataSource, for example, provides objects fetched through an EOEditingContext, while EODetailDataSource provides objects from a relationship property of a master object. For information on creating your own EODataSource subclass, see the section "Creating a Subclass."

An EODataSource provides its objects with its **fetchObjects** method. **insertObject:** and **deleteObject:** add and remove individual objects, and **createObject** instantiates a new object. Other methods provide information about the objects, as described below.

Method Types

Accessing the objects	
	– fetchObjects
Inserting and deleting objects	
	– createObject
	– insertObject:
	– deleteObject:
Creating detail data sources	
	– dataSourceQualifiedByKey:
	– qualifyWithRelationshipKey:ofObject:
Accessing the editing context	
	– editingContext
Accessing the class description	
	– classDescriptionForObjects

Instance Methods

classDescriptionForObjects

– (EOClassDescription *)**classDescriptionForObjects**

Implemented by subclasses to return an EOClassDescription that provides information about the objects provided by the receiver. EODataSource's implementation returns **nil**.

createObject

– (id)**createObject**

Creates a new object, inserts it in the receiver's collection of objects if appropriate, and returns the object. Returns **nil** if the receiver can't create the object or can't insert it. You should invoke **insertObject:** after this method to actually add the new object to the receiver.

As a convenience, EODataSource's implementation sends the receiver's EOClassDescription a **createInstanceWithEditingContext:globalID:zone:** message to create the object. If this succeeds and the receiver has an EOEditingContext, it sends the EOEditingContext an **insertObject:** message to register the new object with the EOEditingContext (note that this does *not* insert the object into the EODataSource). Subclasses that don't use EOClassDescriptions or EOEditingContexts should override this method *without* invoking **super**'s implementation.

See also: – **classDescriptionForObjects**, – **editingContext**

dataSourceQualifiedByKey:

– (EODataSource *)**dataSourceQualifiedByKey:**(NSString *)*relationshipKey*

Implemented by subclasses to return a detail EODataSource that provides the destination objects of the relationship named by *relationshipKey*. The detail EODataSource can be qualified using **qualifyWithRelationshipKey:ofObject:** to set a specific master object (or to change the relationship key). EODataSource's implementation merely raises an NSInvalidArgumentException; subclasses shouldn't invoke **super**'s implementation.

deleteObject:

– (void)**deleteObject:**(id)*anObject*

Implemented by subclasses to delete *anObject*. EODataSource's implementation merely raises an NSInvalidArgumentException; subclasses shouldn't invoke **super**'s implementation.

editingContext

– (EOEditingContext *)**editingContext**

Implemented by subclasses to return the receiver’s EOEditingContext. EODataSource’s implementation returns **nil**.

fetchObjects

– (NSArray *)**fetchObjects**

Implemented by subclasses to fetch and return the objects provided by the receiver. EODataSource’s implementation returns **nil**.

insertObject:

– (void)**insertObject:**(id)*object*

Implemented by subclasses to insert *object*. EODataSource’s implementation merely raises an NSInvalidArgumentException; subclasses shouldn’t invoke **super**’s implementation.

qualifyWithRelationshipKey:ofObject:

– (void)**qualifyWithRelationshipKey:**(NSString *)*key*
    ofObject:(id)*sourceObject*

Implemented by subclasses to qualify the receiver, a detail EODataSource, to display destination objects for the relationship named *key* belonging to *sourceObject*. *key* should be the same as the key specified in the **dataSourceQualifiedByKey:** message that created the receiver. If *sourceObject* is **nil**, the receiver qualifies itself to provide no objects. EODataSource’s implementation merely raises an NSInvalidArgumentException; subclasses shouldn’t invoke **super**’s implementation.

EODataSource

Creating a Subclass

The job of an EODataSource is to provide objects that share a set of properties so that they can be managed uniformly by its client, such as an EODisplayGroup (defined in EOInterface) or a WODisplayGroup (defined in WebObjects). Typically, these objects are all of the same class or share a superclass that defines the common properties managed by the client. All that's needed, however, is that every object have the properties expected by the client. For example, if an EODataSource provides Member and Guest objects, they can be implemented as subclasses of a more general Customer class, or they can be independent classes defining the same properties (**lastName**, **firstName**, and **address**, for example). You typically specify the kind of objects an EODataSource provides when you initialize it. Subclasses usually define a special **init...** method whose arguments describe the objects. EODatabaseDataSource, for example, defines **initWithEditingContext:entityName:**, which uses an EOEntity to describe the set of objects. Another subclass might use an EOClassDescription, a class or superclass for the objects, or even a collection of existing instances.

A subclass can provide two other pieces of information about its objects, using methods declared by EODataSource. First, if your subclass keeps its objects in an EOEditingContext, it should override the **editingContext** method to return that EOEditingContext. It doesn't have to use an EOEditingContext, though, in which case it can just use the default implementation of **editingContext**, which returns **nil**. Keep in mind, however, the amount of work EOEditingContexts do for you, especially when you use EODisplayGroups. For example, EODisplayGroups depend on change notifications from EOEditingContexts to update changes in the objects displayed. If your subclass or its clients depend on change notification, you should use an EOEditingContext for object storage and change notification. If you don't use one, you'll have to implement that functionality yourself. For more information, see these class specifications:

- EOObjectStore
- EOEditingContext
- EODisplayGroup (EOInterface)
- EODelayedObserverQueue
- EODelayedObserver

The other piece of information—also optional—is an EOClassDescription for the objects. Interface Builder uses an EOClassDescription to get the keys it displays in its Connections Inspector, and EODataSource uses it by default when creating new objects. Your subclass should override **classDescriptionForObjects** to return the class description if it uses one and if it's providing objects of a single superclass. Your subclass can either record an EOClassDescription itself, or get it from some other object, such as an EOEntity or from the objects it provides (through the EOEnterpriseObject method **classDescription**, which is implemented in a category of NSObject and also by and EOGenericRecord). If your EODataSource subclass doesn't use an EOClassDescription at all it, can use the default implementation of **classDescriptionForObjects**, which returns **nil**.

Manipulating Objects

A concrete subclass of EODataSource must at least provide objects by implementing **fetchObjects**. If it supports insertion of new objects, it should implement **insertObject:**, and if it supports deletion it should also implement **deleteObject:**. An EODataSource that implements its own store must define these methods from scratch. An EODataSource that uses another object as a store can forward these messages to that store. For example, an EODatabaseDataSource turns these three requests into **objectsWithFetchSpecification:**, **insertObject:**, and **deleteObject:** messages to its EOEditingContext.

Implementing Master-Detail Data Sources

An EODataSource subclass can also implement a pair of methods that allow it to be used in master-detail configurations. The first method, **dataSourceQualifiedByKey:**, should create and return a new data source, set up to provide objects of the destination class for a relationship in a master-detail setup. In a master-detail setup, changes to the detail apply to the objects in the master; for example, adding an object to the detail also adds it to the relationship of the master object. The standard EODetailDataSource class works well for this purpose, so you can simply implement **dataSourceQualifiedByKey:** to create and return one of these. Once you have a detail EODataSource, you can set the master object by sending the detail a **qualifyWithRelationshipKey:ofObject:** message. The detail then uses the master object in evaluating the relationship and applies inserts and deletes to that master object.

Another kind of paired EODataSource setup, called master-peer, is exemplified by the EODatabaseDataSource class. In a master-peer setup, the two EODataSources are independent, so that changes to one don’t affect the other. Inserting into the “peer,” for example, does not update the relationship property of the master object. See that class description for more information.

EODelayedObserver

Inherits From:	NSObject
Conforms To:	EODelayedObserving NSObject (NSObject)
Declared In:	EOControl/EOObserver.h

Class Description

The EODelayedObserver class is a part of EOControl's change tracking mechanism. It is an abstract superclass that defines the basic functionality for coalescing change notifications for multiple objects and postponing notification according to a prioritized queue. For an overview of the general change tracking mechanism, see "Tracking Enterprise Objects Changes" in the introduction to the EOControl Framework.

EODelayedObserver is primarily used to implement the interface layer's associations and wouldn't ordinarily be used outside the scope of a Java Client or Yellow Box application (not in a command line tool or WebObjects application, for example). See the EODelayedObserverQueue class specification for general information.

You would never create an instance of EODelayedObserver. Instead, you use subclasses—typically EOAssociations (EOInterface). For information on creating your own EODelayedObserver subclass, see "Creating a Subclass of EODelayedObserver."

Constants

The following integer constants are defined to represent the priority of a notification in the queue:

EOObserverPriorityImmediate	EOObserverPriorityFourth
EOObserverPriorityFirst	EOObserverPriorityFifth
EOObserverPrioritySecond	EOObserverPrioritySixth
EOObserverPriorityThird	EOObserverPriorityLater

Adopted Protocols

EOObserving	
	– objectWillChange:

Method Types

Change notification	
	– subjectChanged
Canceling change notification	
	– discardPendingNotification
Getting the queue and priority	
	– observerQueue
	– priority

Instance Methods

discardPendingNotification

– (void)**discardPendingNotification**

Sends a **dequeueObserver:** message to the receiver's EODelayedObserverQueue to clear it from receiving a change notification. A subclass of EODelayedObserver should invoke this method in its implementation of **dealloc**.

See also: **observerQueue**

objectWillChange:

@protocol EOObserving
– (void)**objectWillChange:**(id)*anObject*

Implemented by EODelayedObserver to enqueue the receiver on its EODelayedObserverQueue. Subclasses shouldn't need to override this method; if they do, they must be sure to invoke **super**'s implementation.

See also: **observerQueue**, – **enqueueObserver:** (EODelayedObserverQueue), **objectWillChange:** (EOObserving)

observerQueue

– (EODelayedObserverQueue *)**observerQueue**

Overridden by subclasses to return the receiver’s designated EODelayedObserverQueue. EODelayedObserver’s implementation returns the default EODelayedObserverQueue.

See also: **defaultObserverQueue** (EODelayedObserverQueue)

priority

– (EOObserverPriority)**priority**

Overridden by subclasses to return the receiver’s change notification priority, one of:

- EOObserverPriorityImmediate
- EOObserverPriorityFirst
- EOObserverPrioritySecond
- EOObserverPriorityThird
- EOObserverPriorityFourth
- EOObserverPriorityFifth
- EOObserverPrioritySixth
- EOObserverPriorityLater

EODelayedObserver’s implementation returns EOObserverPriorityThird. See the EODelayedObserverQueue class specification for more information on priorities.

subjectChanged

– (void)**subjectChanged**

Implemented by subclasses to examine the receiver’s observed objects and take whatever action is necessary. EODelayedObserver’s implementation does nothing.

EODelayedObserver

Creating a Subclass of EODelayedObserver

EODelayedObserver implements the basic **objectWillChange:** method to simply enqueue the receiver on an EODelayedObserverQueue. Regardless of how many of these messages the receiver gets during the run loop, it receives a single **subjectChanged** message from the queue—at the end of the run loop. In this method the delayed observer can check for changes and take whatever action is necessary. Subclasses should record objects they're interested in, perhaps in an **init** method, and examine them in **subjectChanged**. An EOAssociation.(EOInterface) for example, examines each of the EODisplayGroups (EOInterface) it's bound to in order to find out what has changed. Another kind of subclass might record each changed object for later examination by overriding **objectWillChange:**, but it must be sure to invoke **super**'s implementation when doing so.

The rest of EODelayedObserver's methods have meaningful, if static, default implementations. EODelayedObserverQueue sends change notifications according to the priority of each enqueued observer. EODelayedObserver's implementation of the **priority** method returns EOObserverPriorityThird. Your subclass can override it to return a higher or lower priority, or to have a settable priority. The other method a subclass might override is **observerQueue**, which returns a default EODelayedObserverQueue normally shared by all EODelayedObservers. Because sharing a single queue keeps all EODelayedObserver's synchronized according to their priority, you should rarely override this method, doing so only if your subclass is involved in a completely independent system.

A final method, **discardPendingNotification**, need never be overridden by subclasses, but must be invoked from their implementation of **dealloc**. This prevents observers from being sent change notifications after they've been deallocated.

EODelayedObserverQueue

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOObserver.h

Class Description

The EODelayedObserverQueue class is a part of EOControl's change tracking mechanism. An EODelayedObserverQueue collects change notifications for observers of multiple objects and notifies them of the changes *en masse* during the application's run loop, according to their individual priorities. For an overview of the general change tracking mechanism, see "Tracking Enterprise Objects Changes" in the introduction to the EOControl Framework.

EODelayedObserverQueue's style of notification is particularly useful for coalescing and prioritizing multiple changes; the interface layer's EOAssociation classes use it extensively to update Java Client and Yellow Box user interfaces, for example. Instead of being told that an object will change, an EODelayedObserver is told that it did change, with a **subjectChanged** message, as described in the EODelayedObserver class specification. Delayed observation is thus not useful for comparing old and new states, but only for examining the new state. Delayed observation also isn't ordinarily used outside the scope of a Java Client or Yellow Box application (in a command line tool or WebObjects application, for example).

The motivation for a delayed change notification mechanism arises mainly from issues in observing multiple objects. Any single change to an observed object typically requires the observer to update some state or perform an action. When many such objects change, it makes no sense to recalculate the new state and perform the action for each object. EODelayedObserverQueue allows these changes to be collected into a single notification. It further orders change notifications according to priorities, allowing observers to be updated in sequence according to dependencies among them. For example, an EOMasterDetailAssociation (EOInterface), which must update its detail EODisplayGroup (EOInterface) according to the selection in the master *before* any redisplay occurs, has an earlier priority than the default for EOAssociations. This prevents regular EOAssociations from redisplaying old values and then displaying the new values after the EOMasterDetailAssociation updates.

For more information on using EODelayedObserverQueues, see the sections

- Enqueuing a Delayed Observer
- Change Notification
- Observer Proxies

Method Types

Creating instances	
	– init
Getting the default queue	
	+ defaultObserverQueue
Enqueuing and dequeuing observers	
	– enqueueObserver:
	– dequeueObserver:
Sending change notifications	
	– notifyObserversUpToPriority:
Configuring notification behavior	
	– runLoopModes
	– setRunLoopModes:

Class Methods

defaultObserverQueue

+ (EODelayedObserverQueue *)**defaultObserverQueue**

Returns the EODelayedObserverQueue that EODelayedObservers use by default.

Instance Methods

dequeueObserver:

– (void)**dequeueObserver:**(EODelayedObserver *)*anObserver*

Removes *anObserver* from the receiver.

See also: – **enqueueObserver:**

enqueueObserver:

– (void)**enqueueObserver:**(EODelayedObserver *)*anObserver*

Records *anObserver* to be sent **subjectChanged** messages. If *anObserver*'s priority is EOObserverPriorityImmediate, it's immediately sent the message and not enqueued. Otherwise *anObserver* is sent the message the next time **notifyObserversUpToPriority:** is invoked with a priority later than or equal to *anObserver*'s. Does nothing if *anObserver* is already recorded.

The first time this method is invoked during the run loop with an observer whose priority isn't EOObserverPriorityImmediate, it registers the receiver to be sent a **notifyObserversUpToPriority:** message at the end of the run loop, using EOFlushDelayedObserversRunLoopOrdering and the receiver's run loop modes. This causes enqueued observers up to a priority of EOObserverPrioritySixth to be notified automatically during each pass of the run loop.

This method does *not* retain *anObserver*. When *anObserver* is deallocated, it should invoke **dequeueObserver:** to remove itself from the queue.

See also: – **dequeueObserver:**, – **priority** (EODelayedObserver), – **discardPendingNotification** (EODelayedObserver), – **runLoopModes**, – **performSelector: target:argument:order:modes:** (NSRunLoop class of the Foundation Kit)

init

– (id)**init**

Initializes a newly allocated EODelayedObserverQueue with NSDefaultRunLoopMode as its only run loop mode. This is the designated initializer for the EODelayedObserverQueue class. Returns **self**.

notifyObserversUpToPriority:

– (void) **notifyObserversUpToPriority:**(EOObserverPriority)*priority*

Sends **subjectChanged** messages to all of the receiver's enqueued observers whose priority is *priority* or earlier. This method cycles through the receiver's enqueued observers in priority order, sending each a **subjectChanged** message and then returning to the very beginning of the queue, in case another observer with an earlier priority was enqueued as a result of the message.

EODelayedObserverQueue invokes this method automatically as needed during the run loop, with a *priority* of EOObserverPrioritySixth.

See also: – **enqueueObserver:**, – **priority** (EODelayedObserver)

runLoopModes

– (NSArray *)**runLoopModes**

Returns the receiver's run loop modes.

setRunLoopModes:

– (void)**setRunLoopModes:**(NSArray *)*modes*

Sets the receiver’s run loop modes to *modes*, an array of NSString objects representing run loop modes. For more information see the Foundation class NSRunLoop.

EODelayedObserverQueue

Enqueuing a Delayed Observer

The **enqueueObserver:** method records an EODelayedObserver for later change notification. However, enqueuing is usually performed automatically by an EODelayedObserver in its **objectWillChange:** method. Hence, it's typically enough that an object being observed invoke **willChange** as needed. For example, in Java Client and Yellow Box applications, an EODisplayGroup (EOInterface) does this (among many other things) on receiving an EOObjectsChangedInEditingContextNotification from its EOEditingContext.

Although you can create individual EODelayedObserverQueues using **alloc** and **init**, you typically use the single instance provided by the class method **defaultObserverQueue**. Using separate queues bypasses the prioritization mechanism, which may cause problems between the objects using the separate queues. If you do use separate queues, your EODelayedObserver subclasses should record a designated EODelayedObserverQueue that they always use, and implement **observerQueue** to return that object.

If you need to remove an enqueued observer, you can do so using the **dequeueObserver:** method. EODelayedObserver also defines the **discardPendingNotification** method, which removes the receiver from its designated queue. This is useful in an object's implementation of **dealloc**, for example, to prevent a change notification being sent to it.

Change Notification

The actual process of change notification is initiated by the **enqueueObserver:** messages that line observers up to receive notifications. Regardless of how many times **enqueueObserver:** is invoked for a particular observer, that observer is only put in the queue once. The first observer enqueued during the run loop also triggers the EODelayedObserverQueue to set up a delayed invocation of **notifyObserversUpToPriority:**, which causes it to receive that message at the end of the run loop. EODelayedObserver sets up this delayed invocation in NSDefaultRunLoopMode, but you can change the mode or add additional modes in which delayed invocation occurs using **setRunLoopModes:**.

notifyObserversUpToPriority: cycles through the queue of EODelayedObservers in priority order, from EOObserverPriorityFirst to the priority given, sending each observer a **subjectChanged** message. Each time, it returns to the earliest priority (rather than continuing through the queue) in case the message resulted in another EODelayedObserver with a earlier priority being enqueued. This guarantees an optimal delivery of change notifications.

Observer Proxies

It may not always be possible for a custom observer class to inherit from EODelayedObserver. To aid such objects in participating in delayed change notifications, the Framework defines a subclass of EODelayedObserver, EOObserverProxy, which implements its **subjectChanged** method to invoke an action method of your custom object. You create an EOObserverProxy,using the **initWithTarget:action:**

priority: method, which records the “real” observer, the action method to invoke, and the priority at which the EOObserverProxy should be enqueued. Then, instead of registering the custom object as an observer of objects, you register the proxy (using EOObserverCenter’s **addObserver:forObject:)**. When the proxy receives an **objectWillChange:** message, it enqueues itself for delayed change notification, receives the **subjectChanged** message from the EODelayedObserverQueue, and then sends the action message to the “real” observer.

EODetailDataSource

Inherits From:	EODataSource : NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EODetailDataSource.h

Class Description

EODetailDataSource defines a data source for use in master-detail configurations, where operations in the detail data source are applied directly to properties of a master object. EODetailDataSource implements the standard **fetchObjects**, **insertObject:**, and **deleteObject:** methods to operate on a relationship property of its master object, so it works for any concrete subclass of EODataSource, including another EODetailDataSource (for a chain of three master and detail data sources).

To set up an EODetailDataSource programmatically, you typically create it by sending a **dataSourceQualifiedByKey:** message to the master data source, then establish the master object with a **qualifyWithRelationshipKey:ofObject:** message. The latter method records the name of a relationship for a particular object to resolve in **fetchObjects** and to modify in **insertObject:**, and **deleteObject:**. These three methods then manipulate the relationship property of the master object to perform the operations requested. See the individual method descriptions for more information.

Method Types

Creating instances	
	– initWithMasterClassDescription:detailKey:
	– initWithMasterDataSource:detailKey:
Qualifying instances	
	– qualifyWithRelationshipKey:ofObject:
Examining instances	
	– masterDataSource
	– detailKey
	– masterObject
Accessing the master class description	
	– masterClassDescription
	– setMasterClassDescription:

Accessing the objects	
	– fetchObjects
Inserting and deleting objects	
	– insertObject:
	– deleteObject:
Accessing the master editing context	
	– editingContext

Instance Methods

deleteObject:

– (void)**deleteObject:**(id)*anObject*

Sends a **removeObject:fromPropertyWithKey:** message (defined in the EORelationshipManipulation informal protocol) to the master object with *anObject* and the receiver's detail key as the arguments. Raises an NSInternalInconsistencyException if there's no master object or no detail key set.

detailKey

– (NSString *)detailKey

Returns the name of the relationship for which the receiver provides objects, as provided to **initWithMasterDataSource:detailKey:** or as set in **qualifyWithRelationshipKey:ofObject:**. If none has been set yet, returns **nil**.

editingContext

– (EOEditingContext *)**editingContext**

Returns the EOEditingContext of the master object, or **nil** if there isn't one.

fetchObjects

– (NSArray *)**fetchObjects**

Sends **valueForKey:** (defined in the EOKeyValueCoding informal protocol) to the master object with the receiver's detail key as the argument, constructs an array for the returned object or objects, and returns it. Returns an empty array if there's no master object, or returns an array containing the master object itself if no detail key is set.

initWithMasterClassDescription:detailKey:

– **initWithMasterClassDescription:**(EOClassDescription *)*masterClassDescription*
detailKey:(NSString *)*relationshipKey*

Initializes a newly allocated EODetailDataSource to provide objects based on a relationship of objects in the master object associated with *masterClassDescription*. Invokes **qualifyWithRelationshipKey: ofObject:** with *relationshipKey* specified as the relationship key and **nil** specified as the object. The receiver initially has no master object selected; to select one, use **qualifyWithRelationshipKey:ofObject:**. This is the designated initializer for the EODetailDataSource class. Returns **self**.

See also: – **masterClassDescription**, – **detailKey**

initWithMasterDataSource:detailKey:

– (id)**initWithMasterDataSource:**(EODataSource *)*masterDataSource*
detailKey:(NSString *)*relationshipKey*

Initializes a newly allocated EODetailDataSource to provide objects based on a relationship of objects in *masterDataSource* named by *relationshipKey*. Invokes **initWithMasterClassDescription:detailKey:** with **nil** specified for the class description and *relationshipKey* specified as the detail key. The receiver initially has no master object selected; to select one, use **qualifyWithRelationshipKey:ofObject:**. Returns **self**.

See also: – **masterDataSource**, – **detailKey**

insertObject:

– (void)**insertObject:**(id)*anObject*

Sends an **addObject:toBothSidesOfRelationshipWithKey:** message (defined in the EORelationshipManipulation informal protocol) to the master object with *anObject* and the receiver's detail key as the arguments. Raises an NSInternalInconsistencyException if there's no master object or no detail key set.

masterClassDescription

– (EOClassDescription *)**masterClassDescription**

Returns the EOClassDescription of the receiver's master object.

See also: – **setMasterClassDescription:**, – initWithMasterClassDescription:detailKey:

masterDataSource

– (EODataSource *)**masterDataSource**

Returns the receiver’s master data source.

See also: – **detailKey**, – initWithMasterDataSource:detailKey:

masterObject

– (id)**masterObject**

Returns the object in the master data source for which the receiver provides objects. You can change this with a **qualifyWithRelationshipKey:ofObject:** message.

See also: – **detailKey**

qualifyWithRelationshipKey:ofObject:

– (void)**qualifyWithRelationshipKey:**(NSString *)*relationshipKey*
ofObject:(id)*masterObject*

Configures the receiver to provide objects based on the relationship of *masterObject* named by *relationshipKey. relationshipKey* can be different from the one used with **initWithMasterDataSource: detailKey:**, which changes the relationship the receiver operates on. If *masterObject* is **nil**, this method causes the receiver to return an empty array when sent a **fetchObjects** message.

See also: – **detailKey**

setMasterClassDescription:

– (void)**setMasterClassDescription:**(EOClassDescription *)*classDescription*

Assigns *classDescription* as the EOClassDescription for the receiver’s master object.

See also: – **masterClassDescription**

EOEditingContext

Inherits From:	EOObjectStore : NSObject
Conforms To:	EOObserving NSLocking
Declared In:	EOControl/EOEditingContext.h

Class at a Glance

Purpose

An EOEditingContext object manages a graph of enterprise objects in an application; this object graph represents an internally consistent view of one or more external stores (most often a database).

Principal Attributes

- The set of enterprise objects managed by the EOEditingContext
- The EOEditingContext's parent EOObjectStore
- The set of EOEditor objects messaged by the EOEditingContext
- The EOEditingContext's EOMessageHandler

Creation

– initWithParentObjectStore:	Designated initializer.

Commonly Used Methods

– objectsWithFetchSpecification:	Fetches objects from an external store.
– insertObject:	Registers a new object to be inserted into the parent EOObjectStore when changes are saved.
– deleteObject:	Registers that an object should be removed from the parent EOObjectStore when changes are saved.
– lockObject:	Attempts to lock an object in the external store.
– hasChanges	Returns YES if any of the receiver has any pending changes to the parent EOObjectStore.
– saveChanges	Commits changes made in the receiver to the parent EOObjectStore.
– revert	Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their original values.
– objectForGlobalID:	Given a globalID, returns its associated object.
– globalIDForObject:	Given an object, returns its globalID.
– setDelegate:	Sets the receiver's delegate.
– parentObjectStore	Returns the receiver's parent EOObjectStore.
– rootObjectStore	Returns the receiver's root EOObjectStore.

Class Description

An EOEditingContext object represents a single “object space” or document in an application. Its primary responsibility is managing a graph of enterprise objects. This *object graph* is a group of related business objects that represent an internally consistent view of one or more external stores (usually a database).

All objects fetched from an external store are registered in an editing context along with a global identifier (EOGlobalID) that’s used to uniquely identify each object to the external store. The editing context is responsible for watching for changes in its objects (using the EOObserving protocol) and recording snapshots for object-based undo. A single enterprise object instance exists in one and only one editing context, but multiple copies of an object can exist in different editing contexts. Thus object uniquing is scoped to a particular editing context.

For more information on EOEditingContext, see the sections:

- Other Classes that Participate in Object Graph Management
- Programmatically Creating an EOEditingContext
- Using EOEditingContexts in Different Configurations
- Fetching Objects
- Managing Changes in Your Application
- Methods for Managing the Object Graph
- General Guidelines for Managing the Object Graph
- Using EOEditingContext to Archive Custom Objects in Web Objects Framework

Constants

The following string constants name notifications EOEditingContext posts:

- EOEditingContextDidSaveChangesNotification
- EOObjectsChangedInEditingContextNotification

See the Notifications section for more information on the notifications.

The following string constants are the keys to the EOObjectsChangedInEditingContextNotification’s user info dictionary:

- updated
- deleted
- inserted
- invalidated

EditingContextFlushChangesRunLoopOrdering, is an integer that defines the order in which the editing context performs end of event processing in **processRecentChanges**. Messages with lower order numbers are processed before messages with higher order numbers. In an application built with the Application Kit, the constant order value schedules the editing context to perform its processing before the undo stack group is closed or window display is updated.

Adopted Protocols

EOObserving	
	– objectWillChange:
NSLocking	
	– lock
	– unlock

Method Types

Initializing an EOEditingContext	
	– initWithParentObjectStore:
Controlling EOEditingContext’s memory management strategy	
Fetching objects	
	– objectsWithFetchSpecification:
Committing or discarding changes	
	– saveChanges
	– saveChanges:
	– tryToSaveChanges
	– refaultObjects
	– refault:
	– refetch:
	– revert
	– revert:
	– invalidateAllObjects
Registering changes	
	– deleteObject:
	– insertObject:
	– insertObject:withGlobalID:
	– objectWillChange:
	– processRecentChanges
Checking changes	
	– deletedObjects
	– insertedObjects
	– updatedObjects
	– hasChanges

Object registration and snapshotting	
	– forgetObject:
	– recordObject:globalID:
	– committedSnapshotForObject:
	– currentEventSnapshotForObject:
	– objectForGlobalID:
	– globalIDForObject:
	– registeredObjects
Locking objects	
	– lockObject:
	– lockObjectWithGlobalID:editingContext:
	– isObjectLockedWithGlobalID:editingContext:
	– setLocksObjectsBeforeFirstModification:
	– locksObjectsBeforeFirstModification
Undoing operations	
	– redo:
	– undo:
	– setUndoManager:
	– undoManager
Deletion and Validation Behavior	
	– setPropagatesDeletesAtEndOfEvent:
	– propagatesDeletesAtEndOfEvent
	– setStopsValidationAfterFirstError:
	– stopsValidationAfterFirstError
Returning related object stores	
	– parentObjectStore
	– rootObjectStore
Managing editors	
	– editors
	– addEditor:
	– removeEditor:
Setting the delegate	
	– setDelegate:
	– delegate
Setting the message handler	
	– setMessageHandler:
	– messageHandler
Invalidating objects	
	– setInvalidatesObjectsWhenFreed:
	– invalidatesObjectsWhenFreed

Locking	
	– lock
– unlockWorking with raw rows	
	– faultForRawRow:entityNamed:
Unarchiving from nib	
	+ defaultParentObjectStore
	+ setDefaultParentObjectStore:
	+ setSubstitutionEditingContext:
	+ substitutionEditingContext
Nested EOEditingContext support	
	– objectsWithFetchSpecification:editingContext:
	– objectsForSourceGlobalID:relationshipName:editingContext:
	– arrayFaultWithSourceGlobalID:relationshipName:editingContext:
	– faultForGlobalID:editingContext:
	– saveChangesInEditingContext:
	– refaultObject:withGlobalID:editingContext:
	– invalidateObjectsWithGlobalIDs:
	– initializeObject:withGlobalID:editingContext:
Archiving and unarchiving objects	
	+ encodeObject:withCoder:
	+ initObject:withCoder:
	+ setUsesContextRelativeEncoding:
	+ usesContextRelativeEncoding

Class Methods

defaultParentObjectStore

+ (EOObjectStore *)**defaultParentObjectStore**

Returns the EOObjectStore that is the default parent object store for new editing contexts. Normally this is the EOObjectStoreCoordinator returned from the EOObjectStoreCoordinator class method **defaultCoordinator**.

See also: + **setDefaultParentObjectStore:**

encodeObject:withCoder:

+ (void)**encodeObject:**(id)*object*
  withCoder:(NSCoder *)*encoder*

Invoked by an enterprise object *object* to ask the EOEditingContext to encode *object* using *encoder*. For more discussion of this subject, see “Using EOEditingContext to Archive Custom Objects in Web Objects Framework” in the class description.

See also: + **initObject:withCoder:**, + **setUsesContextRelativeEncoding:**, + **usesContextRelativeEncoding**

initObject:withCoder:

+ (id)**initObject:**(id)*object*
  withCoder:(NSCoder *)*decoder*

Invoked by an enterprise object *object* to ask the EOEditingContext to initialize *object* from data in *decoder*. For more discussion of this subject, see “Using EOEditingContext to Archive Custom Objects in Web Objects Framework” in the class description.

See also: + **encodeObject:withCoder:**, + **setUsesContextRelativeEncoding:**, + **usesContextRelativeEncoding**

setDefaultParentObjectStore:

+ (void)**setDefaultParentObjectStore:**(EOObjectStore *)*store*

Sets the default parent EOObjectStore to *store*. You use this method before loading a nib file to change the default parent EOObjectStores of the EOEditingContexts in the nib file. The object you supply for *store* can be a different EOObjectStoreCoordinator or another EOEditingContext (if you’re using a nested EOEditingContext). After loading a nib with an EOEditingContext substituted as the default parent EOObjectStore, you should restore the default behavior by setting the default parent EOObjectStore to **nil**. For example:

```
[EOEditingContext setDefaultParentObjectStore:editingContext];
nibLoaded = [NSBundle loadNibNamed:@"thirdNib" owner:self];
[EOEditingContext setDefaultObjectStore:nil]; // Restore default
```

A default parent object store is global until it is changed again. For more discussion of this topic, see the chapter “Application Configurations” in the *Enterprise Objects Framework Developer’s Guide*.

See also: + **defaultParentObjectStore**

setSubstitutionEditingContext:

+ (void)**setSubstitutionEditingContext:**(EOEditingContext *)*anEditingContext*

Assigns *anEditingContext* as the EOEditingContext to substitute for the one specified in a nib file you're about to load. Using this method causes all of the connections in your nib file to be redirected to *anEditingContext*. This can be useful when you want an interface loaded from a second nib file to use an existing EOEditingContext. After loading a nib with a substitution EOEditingContext, you should restore the default behavior by setting the substitution EOEditingContext to **nil**. For example:

```
[EOEditingContext setSubstitutionEditingContext:editingContext];
nibLoaded = [NSBundle loadNibNamed:@"thirdNib" owner:self];
[EOEditingContext setSubstitutionEditingContext:nil]; // Restore default
```

A substitution editing context is global until it is changed again. For more discussion of this topic, see the chapter "Application Configurations" in the *Enterprise Objects Framework Developer's Guide*.

See also: + **substitutionEditingContext**

setUsesContextRelativeEncoding:

+ (void)**setUsesContextRelativeEncoding:**(BOOL)*flag*

Sets according to *flag* whether **encodeObject:withCoder:** uses context-relative encoding. For more discussion of this subject, see "Using EOEditingContext to Archive Custom Objects in Web Objects Framework" in the class description.

See also: + **usesContextRelativeEncoding**, + **encodeObject:withCoder:**,

substitutionEditingContext

+ (EOEditingContext *)**substitutionEditingContext**

Returns the substitution EOEditingContext if one has been specified. Otherwise returns **nil**.

See also: + **setSubstitutionEditingContext:**

usesContextRelativeEncoding

+ (BOOL)**usesContextRelativeEncoding**

Returns YES to indicate that **encodeObject:withCoder:** uses context relative encoding, NO otherwise. For more discussion of this subject, see "Using EOEditingContext to Archive Custom Objects in Web Objects Framework" in the class description.

See also: + **setUsesContextRelativeEncoding:**

Instance Methods

addEditor:

– (void)**addEditor:**(id)*editor*

Adds *editor* to the receiver's set of EOEditors. For more explanation, see the method description for **editors** and the EOEditors informal protocol specification.

See also: – **removeEditor:**

arrayFaultWithSourceGlobalID:relationshipName:editingContext:

– (NSArray *)**arrayFaultWithSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*
editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore. If the objects associated with the EOGlobalID *globalID* are already registered in the receiver, returns those objects. Otherwise, propagates the message down the object store hierarchy, through the parent object store, ultimately to the associated EODatabaseContext. The EODatabaseContext creates and returns a to-many fault.

When a parent EOEditingContext receives this on behalf of a child EOEditingContext and the EOGlobalID *globalID* identifies a newly inserted object in the parent, the parent returns a copy of its object's relationship array with the member objects translated into objects in the child EOEditingContext.

For more information on faults, see the EOObjectStore, EODatabaseContext (EOAccess), EOFault, and EOFaultHandler class specifications.

See also: – **faultForGlobalID:editingContext:**

committedSnapshotForObject:

– (NSDictionary *)**committedSnapshotForObject:**(id)*object*

Returns a dictionary containing a snapshot of *object* that reflects its committed values (that is, its values as they were last committed to the database). In other words, this snapshot represents the state of the object before any modifications were made to it. The snapshot is updated to the newest object state after a save.

See also: – **currentEventSnapshotForObject:**

currentEventSnapshotForObject:

– (NSDictionary *)**currentEventSnapshotForObject:**(id)*object*

Returns a dictionary containing a snapshot of *object* that reflects its state as it was at the beginning of the current event loop. After the end of the current event—upon invocation of **processRecentChanges**—this snapshot is updated to hold the modified state of the object.

See also: – **committedSnapshotForObject:**, – **processRecentChanges**

delegate

– (id)**delegate**

Returns the receiver’s delegate.

See also: – **setDelegate:**

deleteObject:

– (void)**deleteObject:**(id)*object*

Specifies that *object* should be removed from the receiver’s parent EOObjectStore when changes are committed. At that time, the object will be removed from the uniquing tables.

See also: – **deletedObjects**

deletedObjects

– (NSArray *)**deletedObjects**

Returns the objects that have been deleted from the receiver’s object graph.

See also: **– updatedObjects**, **– insertedObjects**

editors

– (NSArray *)**editors**

Returns the receiver’s editors. Editors are special-purpose delegate objects that may contain uncommitted changes that need to be validated and applied to enterprise objects before the EOEditingContext saves changes. For example, EODisplayGroups (EOInterface) register themselves as editors with the EOEditingContext of their data sources so that they can save any changes in the key text field. For more information, see the EOEditors informal protocol specification and the EODisplayGroup class specification.

See also: **– addEditor:**, **– removeEditor:**

faultForGlobalID:editingContext:

– (id)**faultForGlobalID:**(EOGlobalID *)*globalID*
  editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore. If the object associated with the EOGlobalID *globalID* is already registered in the receiver, this method returns that object. Otherwise, the method propagates the message down the object store hierarchy, through the parent object store, ultimately to the associated EODatabaseContext. The EODatabaseContext creates and returns a to-one fault.

For example, suppose you want the department object whose **deptID** has a particular value. The most efficient way to get it is to look it up by its globalID using **faultForGlobalID:editingContext:**:

```
EOEntity *entity = [[[editingContext rootObjectStore] modelGroup] entityNamed:
entityName];
EOGlobalID *gid = [entity globalIDForRow:[NSDictionary
    dictionaryWithObjectsAndKeys:deptIdentifier, @"deptID", nil]];
return [editingContext faultForGlobalID:gid editingContext:editingContext];
```

If the department object is already registered in the EOEditingContext, this code returns the object (without going to the database). If not, a fault for this object is created, and the object is fetched only when you trigger the fault.

In a nested editing context configuration, when a parent EOEditingContext is sent **faultForGlobalID: editingContext:** on behalf of a child EOEditingContext and *globalID* identifies a newly inserted object in the parent, the parent registers a copy of the object in the child.

For more discussion of this method, see the section “Working with Objects Across Multiple EOEditingContexts” in the class description. For more information on faults, see the EOObjectStore, EODatabaseContext (EOAccess), EOFault, and EOFaultHandler class specifications.

See also: – **arrayFaultWithSourceGlobalID:relationshipName:editingContext:**

faultForRawRow:entityNamed:

– (id)**faultForRawRow:**(id)*row*
  entityNamed:(NSString *)*entityName*

Returns a fault for the raw row *row* by invoking **faultForRawRow:entityNamed:editingContext:** with **self** as the editing context.

forgetObject:

– (void)**forgetObject:**(id)*object*

Removes *object* from the uniquing tables and causes the receiver to remove itself as the object's observer. This method is invoked whenever an object being observed by an EOEditingContext is deallocated. Note that this method does *not* have the effect of releasing and freeing the object. You should never invoke this method directly. The correct way to remove an object from its editing context is to remove every reference to the object by refaulting any object that references it (using **refaultObjects** or **invalidateAllObjects**). Also note that this method does *not* have the effect of deleting an object—to delete an object you should either use the **deleteObject:** method or remove the object from an owning relationship.

globalIDForObject:

– (EOGlobalID *)**globalIDForObject:***object*

Returns the EOGlobalID for *object*. All objects fetched from an external store are registered in an EOEditingContext along with a global identifier (EOGlobalID) that's used to uniquely identify each object to the external store. If *object* hasn't been registered in the EOEditingContext (that is, if no match is found), this method returns **nil**. Objects are registered in an EOEditingContext using the **insertObject:** method, or, when fetching, with **recordObject:globalID:**.

See also: – **objectForGlobalID:**

hasChanges

– (BOOL)**hasChanges**

Returns YES if any of the objects in the receiver's object graph have been modified—that is, if any objects have been inserted, deleted, or updated.

initWithParentObjectStore:

– **initWithParentObjectStore:**(EOObjectStore *)*anObjectStore*

Initializes the receiver with *anObjectStore* as its parent EOObjectStore. Returns **self**. This method is the designated initializer for EOEditingContext. For more discussion of parent EOObjectStores, see “Other Classes that Participate in Object Graph Management” in the class description.

initializeObject:withGlobalID:editingContext:

– (void)**initializeObject:**(id)*object*
    withGlobalID:(EOGlobalID *)*globalID*
    editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to build the properties for the *object* identified by *globalID*. When a parent EOEditingContext receives this on behalf of a child EOEditingContext (as represented by *anEditingContext*), and the *globalID* identifies an object instantiated in the parent, the parent returns properties extracted from its object and translated into the child's context. This ensures that a nested context "inherits" modified values from its parent EOEditingContext. If the receiver doesn't have *object*, the request is forwarded the receiver's parent EOObjectStore.

insertedObjects

– (NSArray *)**insertedObjects**

Returns the objects that have been inserted into the receiver's object graph.

See also: **– deletedObjects**, **– updatedObjects**

insertObject:

– (void)**insertObject:**(id)*object*

Registers (by invoking **insertObject:withGlobalID:**) *object* to be inserted in the receiver's parent EOObjectStore the next time changes are saved. In the meantime, *object* is registered in the receiver with a temporary globalID.

See also: – **insertedObjects**, – **deletedObjects**, – **insertObject:withGlobalID:**

insertObject:withGlobalID:

– (void)**insertObject:***object* **withGlobalID:**(EOGlobalID *)*globalID*

Registers a new *object* identified by *globalID* that should be inserted in the parent EOObjectStore when changes are saved. Works by invoking **recordObject:globalID:**, unless the receiver already contains the object. Sends *object* the message **awakeFromInsertionInEditingContext:**. *globalID* must respond YES to **isTemporary**. When the external store commits *object*, it re-records it with the appropriate permanent globalID.

It is an error to insert an object that's already registered in an editing context unless you are effectively undeleting the object by reinserting it.

See also: – **insertObject:**

invalidateAllObjects

– (void)**invalidateAllObjects**

Overrides the implementation inherited from EOObjectStore to discard the values of objects cached in memory and refault them, which causes them to be refetched from the external store the next time they're accessed. This method sends the message **invalidateObjectsWithGlobalIDs:** to the parent object store with the globalIDs of all of the objects cached in the receiver. When an EOEditingContext receives this message, it propagates the message down the object store hierarchy. EODatabaseContexts discard their snapshots for invalidated objects and broadcast an EOObjectsChangedInStoreNotification. (EODatabaseContext is defined in EOAccess.)

The final effect of this method is to refault all objects currently in memory. This refaulting in turn releases all objects not retained by your application or by an EODisplayGroup. The next time you access one of these objects, it's refetched from the database.

To flush the entire application's cache of all values fetched from an external store, use a statement such as the following:

```
[[editingContext rootObjectStore] invalidateAllObjects];
```

If you just want to discard uncommitted changes but you don't want to sacrifice the values cached in memory, use the EOEditingContext **revert** method, which reverses all changes and clears the undo stack. For more discussion of this topic, see the section "Methods for Managing the Object Graph" in the class description.

See also: – **refetch:**, – **invalidateObjectsWithGlobalIDs:**

invalidateObjectsWithGlobalIDs:

– (void)**invalidateObjectsWithGlobalIDs:**(NSArray *)*globalIDs*

Overrides the implementation inherited from EOObjectStore to signal to the parent object store that the cached values for the objects identified by *globalID*s should no longer be considered valid and that they should be refaulted. Invokes **processRecentChanges** before refaulting the objects. This message is propagated to any underlying object store, resulting in a refetch the next time the objects are accessed. Any related (child or peer) object stores are notified that the objects are no longer valid. All uncommitted changed to the objects are lost. For more discussion of this topic, see the section "Methods for Managing the Object Graph" in the class description.

See also: – **invalidateAllObjects**

invalidatesObjectsWhenFreed

– (BOOL)**invalidatesObjectsWhenFreed**

Returns YES to indicate that the receiver clears and “booby-traps” all of the objects registered with it when the receiver is deallocated, NO otherwise. The default is YES. In this method, “invalidate” has a different meaning than it does in the other **invalidate...** methods. For more discussion of this topic, see the method description for **setInvalidatesObjectsWhenFreed:**.

isObjectLockedWithGlobalID:editingContext:

– (BOOL)**isObjectLockedWithGlobalID:**(EOGlobalID *)*globalID*
editingContext:(EOEditingContext *)*anEditingContext*

Returns YES if the object identified by *globalID* in *anEditingContext* is locked, NO otherwise. This method works by forwarding the message **isObjectLockedWithGlobalID:editingContext:** to its parent object store.

See also: – **lockObject:**, – **lockObjectWithGlobalID:editingContext:**, – **locksObjectsBeforeFirstModification**

lock

Locks access to the receiver to prevent other threads from accessing it. You should lock an editing context when you are accessing or modifying objects managed by the editing context. The thread-saftey provided by Enterprise Objects Framework allows one thread to be active in each EOEditingContext and one thread to be active in each EODatabaseContext (EOAccess). In other words, multiple threads can access and modify objects concurrently in different editing contexts, but only one thread can access the database at a time (to save, fetch, or fault).

Warning: This method creates an NSAutoreleasePool that is released when **unlock** is called. Consequently, objects that have been autoreleased within the scope of a **lock**/**unlock** pair may not be valid after the **unlock**.

See also: – **unlock**

lockObject:

– (void)**lockObject:**(id)*anObject*

Attempts to lock *anObject* in the external store. This method works by invoking **lockObjectWithGlobalID: editingContext:**. Raises an NSInvalidArgumentException if it can’t find the globalID for *anObject* to pass to **lockObjectWithGlobalID:editingContext:**.

See also: – **isObjectLockedWithGlobalID:editingContext:**, – **locksObjectsBeforeFirstModification**

lockObjectWithGlobalID:editingContext:

– (void)**lockObjectWithGlobalID:**(EOGlobalID *)*globalID*
  editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to attempt to lock the object identified by *globalID* in *anEditingContext* in the external store. Raises an NSInternalInconsistencyException if unable to obtain the lock. This method works by forwarding the message **lockObjectWithGlobalID: editingContext:** to its parent object store.

See also: – **lockObject:**, – **isObjectLockedWithGlobalID:editingContext:**, – **locksObjectsBeforeFirstModification**

locksObjectsBeforeFirstModification

– (BOOL)**locksObjectsBeforeFirstModification**

Returns YES if the receiver locks *object* in the external store (with **lockObject:**) the first time *object* is modified.

See also: – **setLocksObjectsBeforeFirstModification:**, – **isObjectLockedWithGlobalID: editingContext:**, – **lockObject:**, – **lockObjectWithGlobalID:editingContext:**

messageHandler

– (id)**messageHandler**

Returns the EOEditingContext's message handler. A message handler is a special-purpose delegate responsible for presenting errors to the user. Typically, an EODisplayGroup (EOInterface) registers itself as the message handler for its EOEditingContext. For more information, see the EOMessageHandlers informal protocol specification.

See also: – **setMessageHandler:**

objectForGlobalID:

– (id)**objectForGlobalID:**(EOGlobalID *)*globalID*

Returns the object identified by *globalID*, or **nil** if no object has been registered in the EOEditingContext with *globalID*.

See also: – **globalIDForObject:**

objectsForSourceGlobalID:relationshipName:editingContext:

– (NSArray *)**objectsForSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*
editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to service a to-many fault for a relationship named *name*. When a parent EOEditingContext receives a **objectsForSourceGlobalID: relationshipName:editingContext:** message on behalf of a child editing context and *globalID* matches an object instantiated in the parent, the parent returns a copy of its relationship array and translates its objects into the child editing context. This ensures that a child editing context "inherits" modified values from its parent. If the receiving editing context does not have the specified object or if the parent's relationship property is still a fault, the request is fowarded to its parent object store.

objectsWithFetchSpecification:

– (NSArray *)**objectsWithFetchSpecification:**(EOFetchSpecification *)*fetchSpecification*

Invokes **objectsWithFetchSpecification:editingContext:** with **self** as the EOEditingContext and returns the result.

objectsWithFetchSpecification:editingContext:

– (NSArray *)**objectsWithFetchSpecification:**(EOFetchSpecification *)*fetchSpecification*
editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to fetch objects from an external store according to the criteria specified by *fetchSpecification* and return them in an array. If one of these objects is already present in memory, this method doesn't overwrite its values with the new values from the database. This method raises an exception if an error occurs; the error message indicates the nature of the problem.

When an EOEditingContext receives this message, it forwards the message to its root object store. Typically the root object store is an EOObjectStoreCoordinator with underlying EODatabaseContexts. In this case, the object store coordinator forwards the request to the appropriate database context based on the entity name in *fetchSpecification*. The database context then obtains an EODatabaseChannel and performs the fetch, registering all fetched objects in *anEditingContext*. (EODatabaseContext and EODatabaseChannel are defined in EOAccess.)

objectWillChange:

– (void)**objectWillChange:**(id)*object*

This method is automatically invoked when any of the objects registered in the receiver invokes its **willChange** method. This method is EOEditingContext’s implementation of the EOObserving protocol.

parentObjectStore

– (EOObjectStore *)**parentObjectStore**

Returns the EOObjectStore from which the receiver fetches and to which it saves objects.

processRecentChanges

– (void)**processRecentChanges**

Forces the receiver to process pending insertions, deletions, and updates. Normally, when objects are changed, the processing of the changes is deferred until the end of the current event. At that point, an EOEditingContext moves objects to the inserted, updated, and deleted lists, delete propagation is performed, undos are registered, and EOObjectsChangedInStoreNotification and EOObjectsChangedInEditingContextNotification are posted (In a Yellow Box application, this usually causes the user interface to update). You can use this method to explicitly force changes to be processed. An EOEditingContext automatically invokes this method on itself before performing certain operations such as **saveChanges**.

propagatesDeletesAtEndOfEvent

– (BOOL)**propagatesDeletesAtEndOfEvent**

Returns YES if the receiver propagates deletes at the end of the event in which a change was made, NO if it propagates deletes only right before saving changes. The default is YES.

See also: – **setPropagatesDeletesAtEndOfEvent:**

recordObject:globalID:

– (void)**recordObject:**(id)*object*
    globalID:(EOGlobalID *)*globalID*

Makes the receiver aware of an object identified by *globalID* existing in its parent object store. EOObjectStores (such as the access layer’s EODatabaseContext) usually invoke this method for each object fetched. When it receives this message, the receiver enters the object in its uniquing table and registers itself as an observer of the object.

redo:

– (void)**redo:**(id)*sender*

This method forwards a **redo** message to the receiver’s NSUndoManager, asking it to reverse the latest undo operation applied to objects in the object graph.

See also: – **undo:**

refault:

– (void)**refault:**(id)*sender*

This action method simply invokes **refaultObjects**.

refaultObjects

– (void)**refaultObjects**

Refaults all objects cached in the receiver that haven’t been inserted, deleted, or updated. Invokes **processRecentChanges**, then invokes **refaultObject:withGlobalID:editingContext:** for all objects that haven’t been inserted, deleted, or updated. For more discussion of this topic, see the section “Methods for Managing the Object Graph” in the class description.

refaultObject:withGlobalID:editingContext:

– (void)**refaultObject:**(id)*anObject*
withGlobalID:(EOGlobalID *)*globalID*
editingContext:(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to refault the enterprise object *object* identified by *globalID* in *anEditingContext*. This method should be used with caution since refaulting an object does not remove the object snapshot from the undo stack. Objects that have been newly inserted or deleted should not be refaulted.

The main purpose of this method is to break retain cycles between enterprise objects. For example, suppose you have an Employee object that has a to-one relationship to its Department, and the Department object in turn has an array of Employee objects. You can use this method to break the retain cycle. Note that retain cycles are automatically broken if you release the EOEditingContext. For more discussion of this topic, see the section “Methods for Managing the Object Graph” in the class description.

See also: – **invalidateObjectsWithGlobalIDs:**

refetch:

– (void)**refetch:**(id)*sender*

This action method simply invokes the **invalidateAllObjects** method.

registeredObjects

– (NSArray *)**registeredObjects**

Returns the enterprise objects managed by the receiver.

removeEditor:

– (void)**removeEditor:**(id)*editor*

Unregisters *editor* from the receiver. For more discussion of EOEditors, see the **editors** method description and the EOEditors informal protocol specification.

See also: – **addEditor:**

revert

– (void)**revert**

Removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values. Does not refetch from the database. Note that **revert** doesn't automatically cause higher level display groups (WebObject's WODisplayGroups or the interface layer's EODisplayGroups) to refetch. Display groups that allow insertion and deletion of objects need to be explicitly synchronized whenever this method is invoked on their EOEditingContext.

See also: – **invalidateAllObjects**

revert:

– (void)**revert:**(id)*sender*

This action method simply invokes **revert**.

rootObjectStore

– (EOObjectStore *)**rootObjectStore**

Returns the EOObjectStore at the base of the object store hierarchy (usually an EOObjectStoreCoordinator).

saveChanges

– (void)**saveChanges**

Commits changes made in the receiver to its parent EOObjectStore by sending it the message **saveChangesInEditingContext:**. If the parent is an EOObjectStoreCoordinator, it guides its EOCooperatingObjectStores, typically EODatabaseContexts, through a multi-pass save operation (see the EOObjectStoreCoordinator class specification for more information). If a database error occurs, an exception is raised; the error message indicates the nature of the problem.

saveChanges:

– (void)**saveChanges**:(id)*sender*

This action method invokes **saveChanges**, handling an exception by passing it to the message handler. For example, if a validation error occurs, the message handler (usually an EODisplayGroup) presents an alert panel with the text of the validation exception.

See also: – **editingContext:presentErrorMessage:**(EOMessageHandlers), – **editingContext:shouldPresentException:** (EOEditingContext Delegate)

saveChangesInEditingContext:

– (void)**saveChangesInEditingContext:**(EOEditingContext *)*anEditingContext*

Overrides the implementation inherited from EOObjectStore to tell the receiver's EOObjectStore to accept changes from a child EOEditingContext. This method shouldn't be invoked directly. It's invoked by a nested EOEditingContext when it's committing changes to a parent EOEditingContext. The receiving parent EOEditingContext incorporates all changes from the nested EOEditingContext into its own copies of the objects, but it doesn't immediately save those changes to the database. If the parent itself is later sent **saveChanges**, it propagates any changes received from the child along with any other changes to its parent EOObjectStore. Raises an exception if an error occurs; the error message indicates the nature of the problem.

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Set the receiver's delegate to be *anObject*, without retaining it.

See also: – **delegate**

setInvalidatesObjectsWhenFreed:

– (void)**setInvalidatesObjectsWhenFreed:**(BOOL)*flag*

Sets according to *flag* whether the receiver clears and "booby-traps" all of the objects registered with it when the receiver is deallocated. If an editing context invalidates objects when it's deallocated, it sends a **clearProperties** message to all of its objects, thereby breaking any retain cycles between objects that would prevent them from being deallocated. This method leaves the objects in a state in which sending them any message other than **dealloc** or **release** raises an exception.

The default is YES, and as a general rule, this setting must be YES for enterprise objects with cyclic references to be freed when their EOEditingContext is freed.

Note that the word "invalidate" in this method name has a different meaning than it does in the other **invalidate...** methods, which discard object values and refault them.

See also: – **invalidatesObjectsWhenFreed**

setLocksObjectsBeforeFirstModification:

– (void)**setLocksObjectsBeforeFirstModification:**(BOOL)*flag*

Sets according to *flag* whether the receiver locks *object* in the external store (with **lockObject:**) the first time *object* is modified. The default is NO. If *flag* is YES, an exception will be thrown raised if a lock can't be obtained when *object* invokes **willChange**. There are two reasons a lock might fail: because the row is already locked in the server, or because your snapshot is out of date. If your snapshot is out of date, you can explicitly refetch the object using an EOFetchSpecification with **setRefreshesRefetchedObjects:** set to YES. To handle the exception, you can implement the EODatabaseContext delegate method **databaseContextShouldRaiseExceptionForLockFailure:**.

You should avoid using this method or pessimistic locking in an interactive end-user application. For example, a user might make a change in a text field and neglect to save it, thereby leaving the data locked in the server indefinitely. Consider using optimistic locking or application level explicit check-in/check-out instead.

See also: – **locksObjectsBeforeFirstModification**

setMessageHandler:

– (void)**setMessageHandler:**(id)*handler*

Set the receiver’s message handler to be *handler.*

See also: – **messageHandler**

setPropagatesDeletesAtEndOfEvent:

– (void)**setPropagatesDeletesAtEndOfEvent:**(BOOL)*flag*

Sets according to *flag* whether the receiver propagates deletes at the end of the event in which a change was made, or only just before saving changes.

If *flag* is YES, deleting an enterprise object triggers delete propagation at the end of the event in which the deletion occurred (this is the default behavior). If *flag* is NO, delete propagation isn’t performed until **saveChanges** is invoked.

You can delete enterprise objects explicitly by using the **deleteObject:** method or implicitly by removing the enterprise object from an owning relationship. Delete propagation uses the delete rules in the EOClassDescription to determine whether objects related to the deleted object should also be deleted (for more information, see the EOClassDescription class specification and the EOEnterpriseObject informal protocol specification). If delete propagation fails (that is, if an enterprise object refuses to be deleted—possibly due to a deny rule), all changes made during the event are rolled back.

See also: – **propagatesDeletesAtEndOfEvent**

setStopsValidationAfterFirstError:

– (void)**setStopsValidationAfterFirstError:**(BOOL)*flag*

Sets according to *flag* whether the receiver stops validating after the first error is encountered, or continues for all objects (validation typically occurs during a save operation). The default is YES. Setting it to NO is useful if the delegate implements **editingContext:shouldPresentException:** to handle the presentation of aggregate exceptions.

See also: – **stopsValidationAfterFirstError**

setUndoManager:

– (void)**setUndoManager:**(NSUndoManager *)*undoManager*

Sets the receiver's NSUndoManager to *undoManager*. You might invoke this method with **nil** if your application doesn't need undo and you want to avoid the overhead of an undo stack. For more information on editing context's undo support, see the section "Undo and Redo."

See also: – **undoManager**

stopsValidationAfterFirstError

– (BOOL)**stopsValidationAfterFirstError**

Returns YES to indicate that the receiver should stop validating after it encounters the first error, or NO to indicate that it should continue for all objects.

See also: – **setStopsValidationAfterFirstError:**

tryToSaveChanges

– (NSException *)**tryToSaveChanges**

Invokes the **saveChanges** method, and catches and returns any exceptions that are raised.

undo:

– (void)**undo:**(id)*sender*

This action method forwards an **undo** message to the receiver's NSUndoManager, asking it to reverse the latest uncommitted changes applied to objects in the object graph. For more information on editing context's undo support, see the section "Undo and Redo."

See also: **redo:**

undoManager

– (NSUndoManager *)**undoManager**

Returns the receiver's NSUndoManager.

See also: – **setUndoManager:**

unlock

– (void)**unlock**

Unlocks access to the receiver so that other threads may access it.

Warning: This method creates an NSAutoreleasePool that is released when **unlock** is called. Consequently, objects that have been autoreleased within the scope of a **lock**/**unlock** pair may not be valid after the **unlock**.

See also: – **lock**

updatedObjects

– (NSArray *)**updatedObjects**

Returns the objects in the receiver's object graph that have been updated.

See also: **– deletedObjects**, **– insertedObjects**

Notifications

The following notifications are declared (except where otherwise noted) and posted by EOEditingContext.

EOEditingContextDidSaveChangesNotification

This notification is broadcast after changes are saved to the EOEditingContext's parent EOObjectStore. The notification contains:

Notification Object The EOEditingContext

userInfo Dictionary

Key	Value
updated	An NSArray containing the changed objects
deleted	An NSArray containing the deleted objects
inserted	An NSArray containing the inserted objects

EOInvalidatedAllObjectsInStoreNotification

This notification is defined by EOObjectStore. When posted by an EOEditingContext, it's the result of the editing context invalidating all its objects. When an EOEditingContext receives an EOInvalidatedAllObjectsInStoreNotification from its parent EOObjectStore, it clears its lists of inserted, updated, and deleted objects, and resets its undo stack. The notification contains:

Notification Object	The EOEditingContext
userInfo Dictionary	None.

An interface layer EODisplayGroup (not a WebObjects WODisplayGroup) listens for this notification to refetch its contents. See the EOObjectStore class specification for more information on this notification.

EOObjectsChangedInStoreNotification

This notification is defined by EOObjectStore. When posted by an EOEditingContext, it's the result of the editing context processing **objectWillChange:** observer notifications in **processRecentChanges**, which is usually as the end of the event in which the changes occurred. See the EOObjectStore class specification for more information on EOObjectsChangedInStoreNotification.

This notification contains:

Notification Object	The EOEditingContext
userInfo Dictionary	
Key	**Value**
updated	An NSArray of EOGlobalIDs for objects whose properties have changed. A receiving EOEditingContext typically responds by refaulting the objects.
inserted	An NSArray of EOGlobalIDs for objects that have been inserted into the EOObjectStore.
deleted	An NSArray of EOGlobalIDs for objects that have been deleted from the EOObjectStore.
invalidated	An NSArray of EOGlobalIDs for objects that have been turned into faults. Invalidated objects are those for which the cached view should no longer be trusted. Invalidated objects should be refaulted so that they are refetched when they're next examined.

EOObjectsChangedInEditingContextNotification

This notification is broadcast whenever changes are made in an EOEditingContext. It's similar to EOObjectsChangedInStoreNotification, except that it contains objects rather than globalIDs. The notification contains:

Notification Object	The EOEditingContext
userInfo Dictionary	
Key	**Value**
updated	An NSArray containing the changed objects
deleted	An NSArray containing the deleted objects
inserted	An NSArray containing the inserted objects
invalidated	An NSArray containing invalidated objects.

Interface layer EODisplayGroups (not WebObjects WODisplayGroups) listen for this notification to redisplay their contents.

EOEditingContext

Other Classes that Participate in Object Graph Management

EOEditingContexts work in conjunction with instances of other classes to manage the object graph. Two other classes that play a significant role in object graph management are NSUndoManager and EOObserverCenter. NSUndoManager objects provide a general-purpose undo stack. As a client of NSUndoManager, EOEditingContext registers undo events for all changes made the enterprise objects that it watches.

EOObserverCenter provides a notification mechanism for an observing object to find out when another object is about to change its state. “Observable” objects (typically all enterprise objects) are responsible for invoking their **willChange** method prior to altering their state (in a “set” method, for instance). Objects (such as instances of EOEditingContext) can add themselves as observers to the objects they care about in the EOObserverCenter. They then receive a notification (as an **objectWillChange:** message) whenever an observed object invokes **willChange**.

The **objectWillChange:** method is defined in the EOObserving protocol. EOEditingContext implements the EOObserving interface. For more information about the object change notification mechanism, see the EOObserving protocol specification.

Programmatically Creating an EOEditingContext

Typically, an EOEditingContext is created automatically for your application as a by product of some other operation. For example, the following operations result in the creation of network of objects that include an EOEditingContext:

- Running the EOF Wizard in Project Builder to create an OpenStep application with a graphical user interface
- Dragging an entity from EOModeler into a nib file in Interface Builder
- Accessing the default editing context of a WebObjects WOSession in a WebObjects application

Under certain circumstances, however, you may need to create an EOEditingContext programmatically—for example, if you’re writing an application that doesn’t include a graphical interface. To create an EOEditingContext, do this:

```
EOEditingContext *editingContext = [[EOEditingContext alloc] init];
```

This creates an editing context that’s connected to the default EOObjectStoreCoordinator. You can change this default setting by initializing an EOEditingContext with a particular parent EOObjectStore. This is useful if you want your EOEditingContext to use a different EOObjectStoreCoordinator than the default, or if your EOEditingContext is nested. For example, the following code excerpt initializes **childEditingContext** with a parent object store **parentEditingContext**:

```
EOEditingContext *parentEditingContext;     // Assume this exists.
EOEditingContext *childEditingContext = [[EOEditingContext alloc]
    initWithParentObjectStore:parentEditingContext];
```

For more discussion of working programmatically with EOEditingContexts, see the chapter “Application Configurations” in the *Enterprise Objects Framework Developer’s Guide*.

Accessing An Editing Context’s Adaptor Level Objects

You can use an EOEditingContext with any EOObjectStore. However, in a typical configuration, you use an EOEditingContext with the objects in the access layer. To access an EOEditingContext’s adaptor level objects, you get the editing context’s EOObjectStoreCoordinator from the editing context, you get an EODatabaseContext (EOAccess) from the object store coordinator, and you get the adaptor level objects from there. The following code demonstrates the process.

```
EOEditingContext *editingContext;    // Assume this exists.
NSString *myEntityName;              // Assume this exists.
EOFetchSpecification *fspec;
EOObjectStoreCoordinator *rootStore;
EODatabaseContext *dbContext;
EOAdaptor *adaptor;
EOAdaptorContext *adContext;

fspec = [EOFetchSpecification fetchSpecificationWithEntityName:myEntityName
    qualifier:nil
    sortOrderings:nil];

rootStore = (EOCooperatingObjectStore *)[editingContext rootObjectStore];

dbContext = [rootStore objectStoreForFetchSpecification:fspec];

adaptor = [[dbContext database] adaptor];
adContext = [dbContext adaptorContext];
```

This example first creates a fetch specification, providing just the entity name as an argument. Of course, you can use a fetch specification that has non-**nil** values for all of its arguments, but only the entity name is used by the EOObjectStore **objectStoreForFetchSpecification:** method. Next, the example gets the editing context’s EOObjectStoreCoordinator using the EOEditingContext method **rootObjectStore**. **rootObjectStore** returns an EOObjectStore and not an EOObjectStoreCoordinator, because it’s possible to substitute a custom object store in place of an object store coordinator. Similarly, the EOObjectStoreCoordinator method **objectStoreForFetchSpecification:** returns an EOCooperatingObjectStore instead of an access layer EODatabaseContext because it’s possible to substitute a custom cooperating object store in place of a database context. If your code performs any such substitutions, you should alter the above code example to match your custom object store’s API. See the class specifications for EOObjectStore, EOObjectStoreCoordinator, and EOCooperatingObjectStore for more information.

An EOEditingContext's EOObjectStoreCoordinator can have more than one set of database and adaptor level objects. Consequently, to get a database context from the object store coordinator, you have to provide information that the coordinator can use to choose the correct database context. The code example above provides an EOFetchSpecification using the method **objectStoreForFetchSpecification:**, but you could specify different criteria by using one of the following EOObjectStoreCoordinator methods instead:

Method	Description
cooperatingObjectStores	Returns an array of the EOObjectStoreCoordinator's cooperating object stores.
objectStoreForGlobalID:	Returns the cooperating object store for the enterprise object identified by the provided EOGlobalID.
objectStoreForObject:	Returns the cooperating object store for the provided enterprise object.

After you have the EODatabaseContext, you can get the corresponding EOAdaptor and EOAdaptorContext as shown above. (EODatabaseContext, EOAdaptor, and EOAdaptorContext are all defined in EOAccess.)

Using EOEditingContexts in Different Configurations

The fundamental relationship an EOEditingContext has is to its parent EOObjectStore, which creates the object graph the EOEditingContext monitors. EOObjectStore is an abstract class that defines a source and sink of objects for an EOEditingContext. The EOObjectStore is responsible for constructing and registering objects, servicing object faults, and committing changes made in an EOEditingContext.

You can augment the basic configuration of an EOEditingContext and its parent EOObjectStore in several different ways. For example, multiple EOEditingContexts can share the same EOObjectStore, one EOEditingContext can act as an EOObjectStore for another, and so on. The most commonly used scenarios are described in the following sections.

Peer EOEditingContexts

One or more "peer" EOEditingContexts can share a single EOObjectStore (Figure 1). Each EOEditingContext has its own object graph—so, for example, a given Employee row in a database can have separate object instances in each EOEditingContext. Changes to an object in one EOEditingContext don't affect the corresponding object in another EOEditingContext until all changes are successfully committed to the shared object store. At that time the objects in all EOEditingContexts are synchronized with the committed changes. This arrangement is useful when an application allows the user to edit multiple independent "documents."

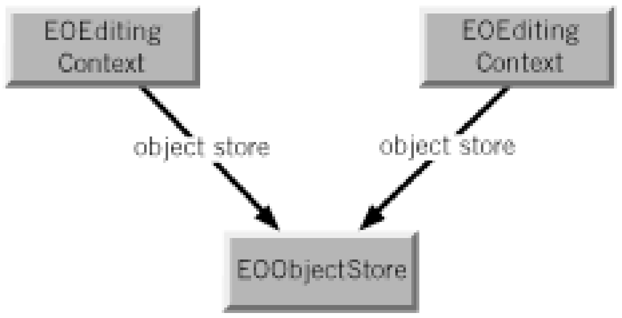


Figure 1 Peer EOEditingContexts

Nested EOEditingContexts

EOEditingContext is a subclass of EOObjectStore, which gives its instances the ability to act as EOObjectStores for other EOEditingContexts. In other words, EOEditingContexts can be nested (Figure 2), thereby allowing a user to make edits to an object graph in one EOEditingContext and then discard or commit those changes to another object graph (which, in turn, may commit them to an external store). This is useful in a “drill down” style of user interface where changes in a nested dialog can be okayed (committed) or canceled (rolled back) to the previous panel.

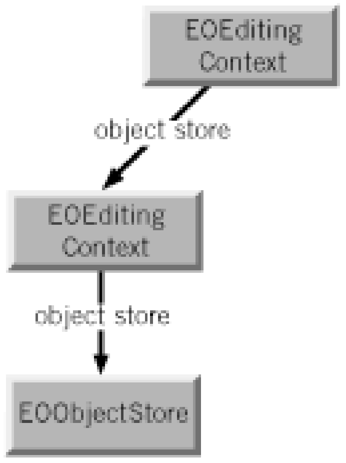


Figure 2 Nested EOEditingContexts

When an object is fetched into a nested EOEditingContext, it incorporates any uncommitted changes that were made to it in its parent EOEditingContext. For example, suppose that in one panel you have a list of employees that allows you to edit salaries, and that the panel includes a button to display a nested panel where you can edit detail information. If you edit the salary in the parent panel, you see the modified salary in the nested panel, not the old (committed) salary from the database. Thus, conceptually, nested EOEditingContexts fetch through their parents.

EOEditingContext overrides several of EOObjectStore's methods:

- – arrayFaultWithSourceGlobalID:relationshipName:editingContext:
- – faultForGlobalID:editingContext:
- – invalidateAllObjects
- – invalidateObjectsWithGlobalIDs:
- – objectsForSourceGlobalID:relationshipName:editingContext:
- – objectsWithFetchSpecification:editingContext:
- – refaultObject:withGlobalID:editingContext:
- – saveChangesInEditingContext:

These methods are generally used when an EOEditingContext acts as an EOObjectStore for another EOEditingContext. For more information, see the individual method descriptions. For information on setting up this configuration for interfaces loaded from nib files, see the method description for **setDefaultParentObjectStore:**.

For a description of how to implement nested EOEditingContexts, see the chapter "Application Configurations" in the *Enterprise Objects Framework Developer's Guide*.

Getting Data from Multiple Sources

An EOEditingContext's object graph can contain objects from more than one external store (Figure 3). In this scenario, the object store is an EOObjectStoreCoordinator, which provides the abstraction of a single object store by redirecting operations to one or more EOCooperatingObjectStores.

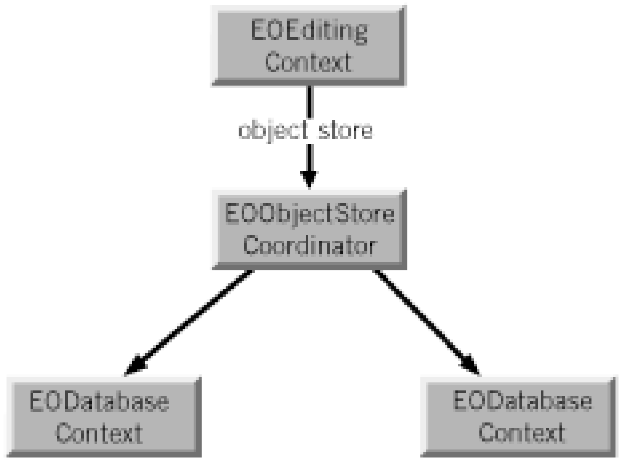


Figure 3 An EOEditingContext Containing Objects from Multiple Sources

In writing an application, it's likely that you'll use combinations of the different scenarios described in the above sections.

Fetching Objects

The most common way to explicitly fetch objects from an external store in an Enterprise Objects Framework application is to use EOEditingContext's **objectsWithFetchSpecification:** method. This method takes a fetch specification and returns an array of objects. A fetch specification includes the name of the entity for which you want to fetch objects, the qualifier (query) you want to use in the fetch, and the sort order in which you want the objects returned (if any). For example, the following code excerpt uses **objectsWithFetchSpecification:** to fetch all video store members who have Visa credit cards:

```
EOFetchSpecification *fetchSpec;
NSArray *results;
fetchSpec = [EOFetchSpecification
    fetchSpecificationWithEntityName:@"Member"
    qualifier:[EOQualifier qualifierWithQualifierFormat:
    @"cardType = 'Visa' "]
    sortOrderings:nil];
results = [editingContext objectsWithFetchSpecification:fetchSpec];
```

Note that objects are allocated in the same zone as the EOEditingContext into which they're fetched.

Managing Changes in Your Application

EOEditingContext provides several methods for managing the changes made to objects in your application. You can use these methods to get information about objects that have changed, to selectively undo and redo changes, and to discard all changes made to objects before these changes are committed to the database. These methods are described in the following sections.

Getting Information About Changed Objects

An EOEditingContext maintains information about three different kinds of changes to objects in its object graph: insertions, deletions, and updates. After these changes have been made and before they're committed to the database, you can find out which objects have changes in each of these categories by using the **insertedObjects**, **deletedObjects**, and **updatedObjects** methods. Each method returns an array containing the objects that have been inserted, deleted, and updated, respectively. The **hasChanges** method returns YES or NO to indicate whether any of the objects in the object graph have been inserted, deleted, or updated.

Undo and Redo

EOEditingContext includes the **undo:**, **redo:**, and **revert:** methods for managing changes to objects in the object graph. **undo:** asks the EOEditingContext's NSUndoManager to reverse the latest changes to objects in the object graph. **redo:** asks the NSUndoManager to reverse the latest undo operation. **revert:** clears the undo stack, discards all insertions and deletions, and restores updated objects to their last committed (saved) values.

EOEditingContext's undo support is arbitrarily deep; you can undo an object repeatedly until you restore it to the state it was in when it was first created or fetched into its editing context. Even after saving, you can undo a change. To support this feature, the NSUndoManager can keep a lot of data in memory.

For example, whenever an object is removed from a relationship, the corresponding editing context creates a snapshot of the modified, source object. The snapshot, which retains the removed object, is retained by the editing context and by the undo manager. The editing context releases the snapshot when the change is saved, but the undo manager doesn't. It continues holding the snapshot, so it can undo the deletion if requested.

If the typical usage patterns for your application generate a lot of change processing, you might want to limit the undo feature to keep its memory usage in check. For example, you could clear an undo manager whenever its editing context saves. To do so, simply send the undo manager a **removeAllActions** message (or a **removeAllActionsWithTarget:** message with the editing context as the argument). If your application doesn't need undo at all, you can avoid any undo overhead by setting the editing context's undo manager to **nil** with **setUndoManager:**.

Saving Changes

The **saveChanges** method commits changes made to objects in the object graph to an external store. When you save changes, EOEditingContext's lists of inserted, updated, and deleted objects are flushed.

Upon a successful save operation, the EOEditingContext's parent EOObjectStore broadcasts an EOObjectsChangedInStoreNotification. Peers of the saved EOEditingContext receive this notification and respond by synchronizing their objects with the committed versions. See also

Methods for Managing the Object Graph

EOEditingContext provides methods for managing the enterprise objects in the context's object graph. This section describes these methods, as well as other techniques you can use to manage the object graph.

At different points in your application, you might want to do the following:

- Break retain cycles between enterprise objects
- Discard changes that have been made to enterprise objects
- Make sure that when you refetch objects from the database, any changed database values are used instead of the original values
- Discard the view of objects cached in memory
- Work with objects across multiple editing contexts

These scenarios are discussed in the following sections.

Breaking Retain Cycles

You use the EOEditingContext methods **refaultObjects** and **refaultObject:withGlobalID: editingContext:** to break retain cycles between your enterprise objects. For example, suppose you have an Employee object that has a to-one relationship to its Department, and the Department object in turn has an array of Employee objects. This circular reference constitutes a retain cycle, which you can break using the **refault...** methods.

Note: Retain cycles are automatically broken if you release the EOEditingContext.

You should use the **refault...** methods with caution, since refaulting an object doesn't remove the object snapshot from the undo stack. Objects that have been newly inserted or deleted should not be refaulted. In general, it's safer to use **refaultObjects** than it is to use **refaultObject:withGlobalID:editingContext:** since **refaultObjects** only refaults objects that haven't been inserted, deleted or updated. The method **refaultObject:withGlobalID:editingContext:** doesn't make this distinction, so you should only use it when you're sure you know what you're doing.

If you want to reset your EOEditingContext and free all of its objects, do the following:

```
EOEditingContext *editingContext;     // Assume this exists.
[editingContext revert];               // Discard uncommitted changes
[editingContext refaultObjects];
```

Note that you must release any other retains on the enterprise objects in the EOEditingContext for them to actually be freed. For example, to clear a display group that references a list of enterprise objects, you'd do something like the following:

```
[displayGroup setObjectArray:nil];
```

Releasing the EODisplayGroup (and any user interface objects that refer to it) also has the effect of releasing the object array.

Using the **invalidate...** methods (described below) also has the effect of breaking retain cycles, but these methods have a more far-reaching effect. It's not recommended that you use them simply to break retain cycles.

Discarding Changes to Enterprise Objects

EOEditingContext provides different techniques for discarding changes to enterprise objects. These techniques are as follows:

- Perform a simple **undo:**, which reverses the latest uncommitted changes applied to objects in the object graph.
- Invoke the EOEditingContext method **revert**, which removes everything from the undo stack, discards all insertions and deletions, and restores updated objects to their last committed values. If you just want to discard uncommitted changes but you don't want to sacrifice the original values from the database cached in memory, use the **revert** method.

A different approach is to use the **invalidate...** methods, described in ““Discarding the View of Objects Cached in Memory”.”

Refreshing Objects

One characteristic of an object graph is that it represents an internally consistent view of your application’s data. By default, when you refetch data, Enterprise Objects Framework maintains the integrity of your object graph by not overwriting your object values with database values that have been changed by someone else. But what if you want your application to see those changes? You can accomplish this by using the EOFetchSpecification method **setRefreshesRefetchedObjects:**. Invoking **setRefreshesRefetchedObjects:** with the argument YES causes existing objects to be overwritten with fetched values that have been changed. Alternatively, you can use the EODatabaseContext (EOAccess) delegate method **databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID:channel:**.

Normally, when you set an EOFetchSpecification to refresh using **setRefreshesRefetchedObjects:**, it only refreshes the objects you’re fetching. For example, if you refetch employees, you don’t also refetch the employees’ departments. However, if you use the EOPrefetchingRelationshipHintKey with an EOFetchSpecification in the EODatabaseContext method **objectsWithFetchSpecification: editingContext:**, the refetch is propagated for all of the fetched objects’ relationships that are specified for the hint. For more discussion of this topic, see the EODatabaseContext class specification.

Refreshing refetched objects only affects the objects you specify. If you want to refetch your entire object graph, you can use the EOEditingContext **invalidate...** methods, described below.

Discarding the View of Objects Cached in Memory

As described in the section “Discarding Changes to Enterprise Objects,” you can use **undo:** or **revert** to selectively discard the changes you’ve made to enterprise objects. Using these methods preserves the original cache of values fetched from the database. But what if you want to flush your in-memory object view all together—in the most likely scenario, to see changes someone else has made to the database? You can invalidate your enterprise objects using the **invalidateAllObjects** method or the **invalidateObjectsWithGlobalIDs:** method. (You can also use the action method **refetch:**, which simply invokes **invalidateAllObjects**). Unlike fetching with the EOFetchSpecification method **setRefreshesRefetchedObjects:** set to YES (described above), the **invalidate...** methods result in the refetch of your entire object graph.

The effect of the **invalidateAllObjects** method depends on how you use it. For example, if you send **invalidateAllObjects** to an EOEditingContext, it sends **invalidateObjectsWithGlobalIDs:** to its parent object store with all the globalIDs for the objects registered in it. If the EOEditingContext is nested, its parent object store is another EOEditingContext; otherwise its parent object store is typically an EOObjectStoreCoordinator. Regardless, the message is propagated down the object store hierarchy. Once it reaches the EOObjectStoreCoordinator, it’s propagated to the EODatabaseContext(s). The EODatabaseContext discards the row snapshots for these globalIDs and sends an EOObjectsChangedInStoreNotification, thereby refaulting all the enterprise objects in the object graph.

This refaulting in turn releases all objects not retained by your application or by an EODisplayGroup. The next time you access one of these objects, it's refetched from the database.

Sending **invalidateAllObjects** to an EOEditingContext affects not only that context's objects, but objects with the same globalIDs in other EOEditingContexts. For example, suppose *editingContext1* has *objectA* and *objectB*, and *editingContext2* has *objectA*, *objectB*, and *objectC*. When you send **invalidateAllObjects** to *editingContext1*, *objectA* and *objectB* are refaulted in both *editingContext1* and *editingContext2*. However, *objectC* in *editingContext2* is left intact since *editingContext1* doesn't have an *objectC*.

If you send **invalidateAllObjects** directly to the EOObjectStoreCoordinator, it sends **invalidateAllObjects** to all of its EODatabaseContexts, which then discard all of the snapshots in your application and refault every single enterprise object in all of your EOEditingContexts.

The **invalidate...** methods are the only way to get rid of a database lock without saving your changes.

Working with Objects Across Multiple EOEditingContexts

Any time your application is using more than one EOEditingContext (as described in the section ""Using EOEditingContexts in Different Configurations""), it's likely that one editing context will need to access objects in another.

On the face of it, it may seem like the most reasonable solution would be for the first editing context to just get a pointer to the desired object in the second editing context and modify the object directly. But this would violate the cardinal rule of keeping each editing context's object graph internally consistent. Instead of modifying the second editing context's object, the first editing context needs to get its own copy of the object. It can then modify its copy without affecting the original. When it saves changes, they're propagated to the original object, down the object store hierarchy. The method that you use to give one editing context its own copy of an object that's in another editing context is **faultForGlobalID:editingContext:**.

For example, suppose you have a nested editing context configuration in which a user interface displays a list of objects—this maps to the parent editing context. From the list, the user can select an object to inspect and modify in a "detail view"—this maps to the child editing context. To give the child its own copy of the object to modify in the detail view, you would do something like the following:

```
EOEditingContext *childEC, *parentEC;      // Assume these exist.
id newObject = [childEC faultForGlobalID:[parentEC globalIDForObject:origObject]
    editingContext:childEC];
```

where **origObject** is the object the user selected for inspection from the list.

The child can make changes to **newObject** without affecting **origObject** in the parent. Then when the child saves changes, **origObject** is updated accordingly.

Updates from the Parent EOObjectStore

When changes are successfully saved in an EOObjectStore, it broadcasts an EOObjectsChangedInStoreNotification. An EOEditingContext receiving this notification will synchronize its objects with the committed values by refaulting objects needing updates so the new values will be retrieved from the EOObjectStore the next time they are needed. However, locally uncommitted changes to objects in the EOEditingContext are by default reapplied to the objects, in effect preserving the uncommitted changes in the object graph. After the update, the uncommitted changes remain uncommitted, but the committed snapshots have been updated to reflect the values in the EOObjectStore.

You can control this process by implementing two delegate methods. Before any updates have occurred, the delegate method **editingContext:shouldMergeChangesForObject:** will be invoked for each of the objects that has both uncommitted changes and an update in the EOObjectStore. If the delegate returns YES, the uncommitted chnages will be merged with the update (the default behavior). If it returns NO, then the object will be invalidated (and refaulted) without preserving ay uncommitted changes. As a side effect, the delgate might cache information about the object (globalID, snapshot, etc.) so that a specialized merging behavior could be implemented. At this point, the delegate should not make changes to the object becuse it is about to be invalidated. However, the delegate method **editingContextDidMergeChanges:** is invoked after all of the updates for the EOObjectsChangedInStoreNotification have been completed, including the merginf of all uncommitted changes. By default, it does nothing, but this delegate method might perform the customized merging behavior based on whatever information was cached in **editingContext: shouldMergeChangesForObject:** for each of the objects that needed an update. See the informal protocol EOValueMerging for the descriptions of the methods **changesFromSnapshot:** and **reapplyChangesFromDictionary:**, which might be useful for implementing custom merging behaviors.

General Guidelines for Managing the Object Graph

When you fetch objects into your application, you create a graph of objects instantiated from database data. From that point on, your focus should be on working with the object graph—not on interacting with your database. This distinction is an important key to working with Enterprise Objects Framework.

You don’t have to worry about the database...

One of the primary benefits of Enterprise Objects Framework is that it insulates you from having to worry about database details. Once you’ve defined the mapping between your database and your enterprise objects in a model file, you don’t need to think about issues such as foreign key propagation, how object deletions are handled, how operations in the object graph are reflected in your database tables, and so on.

This can be illustrated by considering the common scenario in which one object has a relationship to another. For example, suppose an Employee has a relationship to a Department. In the object graph, this relationship is simply expressed as an Employee object having a pointer to its Department object. The Department object might in turn have a pointer to an array of Employee objects. When you manipulate relationships in the object graph (for example, by moving an Employee to a different Department), Enterprise Objects Framework changes the appropriate relationship pointers. For example, moving an Employee to a different Department changes the Employee’s department pointer and adds the Employee to the new Department’s employee array. When you save your changes to the database, Enterprise Objects Framework knows how to translate these object graph manipulations into database operations.

...but you do have to worry about the object graph

As described above, you generally don’t need to concern yourself with how changes to the object graph are saved to the database. However, you do need to concern yourself with managing the object graph itself. Since the object graph is intended to represent an internally consistent view of your application’s data, one of your primary considerations should be maintaining its consistency. For example, suppose you have a relationship from Employee to Project, and from Employee to Manager. When you create a new Employee object, you must make sure that it has relationships to the appropriate Projects and to a Manager.

Just as you need to maintain the internal consistency of an EOEditingContext’s object graph, you should never directly modify the objects in one EOEditingContext from another EOEditingContext. If you do so, you risk creating major synchronization problems in your application. If you need to access the objects in one EOEditingContext from another, use the method **faultForGlobalID:editingContext:**, as described in “Working with Objects Across Multiple EOEditingContexts.” This gives the receiving EOEditingContext its own copy of the object, which it can modify without affecting the original. Then when it saves its changes, the original is updated accordingly.

One of the implications of needing to maintain the consistency of your object graph is that you should never copy an enterprise object (though you can snapshot its properties), since this would be in conflict with uniquing. Uniquing dictates that an EOEditingContext can have one and only one copy of a particular object. For more discussion of uniquing, see the chapter “Behind the Scenes” in the *Enterprise Objects Framework Developer’s Guide*. Similarly, your enterprise objects shouldn’t override the **isEqual:** method. Enterprise Objects Framework relies on the default NSObject implementation which checks instance (pointer) equality rather than value equality.

Using EOEditingContext to Archive Custom Objects in Web Objects Framework

In WebObjects, applications that use the Enterprise Objects Framework must enlist the help of the EOEditingContext to archive enterprise objects. The primary reason is so that the EOEditingContext can keep track, from one transaction to the next, of the objects it manages. But using an EOEditingContext for archiving also benefits your application in these other ways:

- During archiving, an EOEditingContext stores only as much information about its enterprise objects as is needed to reconstitute the object graph at a later time. For example, unmodified objects are stored as simple references (by globalID) that will allow the EOEditingContext to recreate the object from the database. Thus, your application can store state very efficiently by letting an EOEditingContext archive your enterprise objects.
- During unarchiving, an EOEditingContext can recreate individual objects in the graph only as they are needed by the application. This approach can significantly improve application performance.

An enterprise object (like any other object that uses the OpenStep archiving scheme) makes itself available for archiving by declaring that it conforms to the NSCoding protocol, by implementing the protocol’s two methods, **encodeWithCoder:** and **initWithCoder:**. It implements these methods like this:

```
- (void)encodeWithCoder:(NSCoder *)aCoder {
    [EOEditingContext encodeObject:self withCoder:aCoder];
}

- (id)initWithCoder:(NSCoder *)aDecoder {
    return [EditingContext initObject:self withCoder:aDecoder];
}
```

The enterprise object simply passes on responsibility for archiving and unarchiving itself to the EOEditingContext class, by invoking the **encodeObject:withCoder:** and **initObject:withCoder:** class methods. The EOEditingContext takes care of the rest. For more discussion of **encodeWithCoder:** and **initWithCoder:**, see the NSCoding protocol specification in the *Foundation Framework Reference*.

EOEditingContext includes two additional methods that affect the archiving and unarchiving of objects: **setUsesContextRelativeEncoding:** and **usesContextRelativeEncoding**. When you use context relative encoding, it means that enterprise objects that archive themselves using the EOEditingContext **encodeObject:withCoder:** method archive their current state (that is, all of their class properties) only if they (the objects) are marked as inserted or updated in the EOEditingContext. Otherwise, they archive just their globalID's since their state matches what's stored in the database and can be retrieved from there. If **usesContextRelativeEncoding** returns NO, it means the current state will always be archived, even if the enterprise object is unmodified. The default is NO for OpenStep applications, and YES for WebObjects applications.

EOFault

Inherits From:	none *(EOFault is a root class)*
Declared In:	EOControl/EOFault.h

Class Description

EOFault and EOFaultHandler form a general mechanism for substituting placeholder objects that convert themselves into regular objects. An EOFault is most commonly used by the Access Layer to represent an object not yet fetched from the database, but that must nonetheless exist as an instance in the application—typically because it's the destination of a relationship. EOFault is a completely general class; there's no need to create subclasses to customize fault handling. Instead, you create subclasses of EOFaultHandler to accommodate different means of converting faults into regular objects.

The faulting mechanism provides for continuity of an object's **id** even when that object's state isn't yet available. An EOFault simply holds the place for an ultimate "real" object, handling all methods that it can without causing the state to be loaded. When an EOFault receives a message that it can't handle, it calls upon its EOFaultHandler to *fire* it, converting it into a "real" object. This often involves accessing the external, persistent state of the object.

Creating an EOFault

Rather than allocating and initializing an EOFault, you turn an existing object into one using EOFault's **makeObjectIntoFault:withHandler:** class method. When you do so, you must provide an EOFaultHandler that will later help the fault to fire. **makeObjectIntoFault:withHandler:** preserves the **id** of the original object, overlaying its **isa** pointer with that of the EOFault class and slipping the EOFaultHandler among its instance variables. Once this is done, the original object is an EOFault that will fire when accessed.

The EOFaultHandler should be considered completely private property of the EOFault once you've created it. You should neither retain the EOFaultHandler or send it any other messages, instead dealing exclusively with the newly created EOFault or the EOFault class itself.

EOFault Behavior

EOFault implements many basic object methods in a manner that doesn't cause the receiver to fire. The following methods all behave as though normal for the original object:

– retain	– isMemberOfClass:

– release	– conformsToProtocol:
– autorelease	– isProxy
– retainCount	– methodSignatureForSelector:
– class	– respondsToSelector:
– superclass	– zone
– isKindOfClass:	– doesNotRecognizeSelector:

doesNotRecognizeSelector: is a special case here, in that it's only invoked if the selector in question isn't found for the original class. Normally, methods not implemented by EOFault, but implemented by the original class, cause the receiver to fire as described below.

These methods don't cause the receiver to fire, but also don't hide the presence of the EOFault class:

– description	– descriptionWithLocale:
– descriptionWithIndent:	– descriptionWithLocale:indent:
– eoDescription	– eoShallowDescription

The following common methods, along with any others not explicitly mentioned in this section, do cause the receiving EOFault to fire.

- – dealloc
- – self
- – forwardInvocation:

When an EOFault receives one of these messages, it fires in one of a few different ways. **dealloc** invokes the – clearFault: class method to revert the receiver back to its original state, then reinvokes **dealloc** to clean up instance variables and deallocate the object. The other methods all send a special message, **completeInitializationOfObject:**, to the EOFaultHandler to transform the EOFault into a regular object, possibly different from its original state. In addition, **forwardInvocation:** sends a **shouldPerformInvocation:** to the EOFaultHandler first, which allows it to perform the method itself without causing the EOFault to be transformed. If the EOFaultHandler returns YES, though, the EOFault then sends it a **completeInitializationOfObject:** message.

Examining an EOFault

Three additional EOFault methods allow you to explicitly check whether an object is an EOFault without causing it to fire, and to get its original class and EOFaultHandler if it is an EOFault. These methods are:

- + isFault:
- + targetClassForFault:
- + handlerForFault:

You can use these methods to base some decisions on whether an object is an EOFault, though you should rarely need to do so.

Method Types

Creating and examining faults

+ makeObjectIntoFault:withHandler:
+ isFault:
+ clearFault:
+ handlerForFault:
+ targetClassForFault:
+ respondsToSelector:

Checking class informatio

– class
– isKindOfClass:
– isMemberOfClass:
– respondsToSelector:
– conformsToProtocol:
– methodSignatureForSelector:

Run-time support

– forwardInvocation:
– doesNotRecognizeSelector:

Getting a fault’s description

– description
– descriptionWithIndent:
– descriptionWithLocale:
– descriptionWithLocale:indent:
– eoDescription
– eoShallowDescription

Reference-counting

– retain
– release
– retainCount
– autorelease
– dealloc

Miscellaneous object methods

- – self
- – isProxy
- – superclass
- – zone

Class Methods

clearFault:

+ (void)**clearFault:**(id)*aFault*

Restores *aFault* to its status prior to the **makeObjectIntoFault:withHandler:** message that created it. Raises an NSInvalidArgumentException if *aFault* isn't an EOFault.

You rarely use this method. Faults typically fire automatically when accessed, using EOFaultHandler's **completeInitializationOfObject:** method. See the EOFaultHandler class specification for more information.

handlerForFault:

+ (EOFaultHandler *)**handlerForFault:**(id)*aFault*

Returns the EOFaultHandler that will help *aFault* to fire. Returns **nil** if *aFault* isn't an EOFault.

isFault:

+ (BOOL)**isFault:**(id)*anObject*

Returns YES if *anObject* is an EOFault, NO otherwise.

makeObjectIntoFault:withHandler:

+ (void)**makeObjectIntoFault:**(id)*anObject* **withHandler:**(EOFaultHandler *)*aFaultHandler*

Converts *anObject* into an EOFault, assigning *aFaultHandler* as the object that stores its original state and later converts the EOFault back into a normal object (typically by fetching data from an external repository). The new EOFault becomes the owner of *aFaultHandler*; you shouldn't assign it to another object.

respondsToSelector:

+ (BOOL)**respondsToSelector:**(SEL)*aSelector*

Returns YES if the receiving class responds to *aSelector*, NO otherwise.

targetClassForFault:

+ (Class)**targetClassForFault:**(id)*aFault*

Returns the original class of the object that was turned into *aFault*, or **nil** if *aFault* isn't an EOFault. When the EOFault fires, it's guaranteed to be an instance of this class or possibly of a subclass. To get the actual class, you must send a **class** message to the EOFault, which may fire to determine its actual class membership.

Instance Methods

autorelease

– (id)**autorelease**

Performs as NSObject's **autorelease** method.

class

– (Class)**class**

Returns the class of the object that the receiving EOFault will become. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **classForFault:** (EOFaultHandler), + **targetClassForFault:**

conformsToProtocol:

– (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol*

Returns YES if the object that the receiving EOFault will become conforms to *aProtocol*, NO if it doesn't. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **conformsToProtocol:forFault:** (EOFaultHandler)

dealloc

– (void)**dealloc**

Invokes the **clearFault:** class method to revert the receiving EOFault to its original class membership and state, then reinvokes **dealloc**.

description

– (NSString *)**description**

Sends **descriptionForObject:** to the receiver's EOFaultHandler and returns the result.

descriptionWithIndent:

– (NSString *)**descriptionWithIndent:**(unsigned int)*indentLevel*

Invokes **description** and returns the result.

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Invokes **description** and returns the result.

descriptionWithLocale:indent:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale* **indent:**(unsigned int)*indentLevel*

Invokes **description** and returns the result.

doesNotRecognizeSelector:

– (void)**doesNotRecognizeSelector:**(SEL)*aSelector*

Raises an NSInvalidArgumentException.

eoDescription

– (NSString *)**eoDescription**

Invokes **description** and returns the result.

See also: – **eoDescription** (NSObject Additions)

eoShallowDescription

– (NSString *)**eoShallowDescription**

Invokes **description** and returns the result.

See also: – **eoShallowDescription** (NSObject Additions)

forwardInvocation:

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*

Causes the receiving EOFault to fire, if allowed by its EOFaultHandler, and forward *anInvocation* to its new incarnation. Sends a **shouldPerformInvocation:** to the receiver's EOFaultHandler first, giving it a chance to bypass the conversion. If the EOFaultHandler returns NO, returns immediately. If it returns YES, sends a **completeInitializationOfObject:** message to the EOFaultHandler with **self** as the argument. Once the receiver has fired it invokes *anInvocation*.

isKindOfClass:

– (BOOL)**isKindOfClass:**(Class)*aClass*

Returns YES if *aClass* is the class, or a superclass, of the object that the receiving EOFault will become, NO otherwise. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **isMemberOfClass:**, – **isKindOfClass:forFault:** (EOFaultHandler)

isMemberOfClass:

– (BOOL)**isMemberOfClass:**(Class)*aClass*

Returns YES if *aClass* is the class of the object that the receiving EOFault will become, NO otherwise. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **isKindOfClass:**, – **isMemberOfClass:forFault:** (EOFaultHandler)

isProxy

– (BOOL)**isProxy**

Returns NO.

methodSignatureForSelector:

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector*

Returns a method signature for *aSelector* for the object that the receiving EOFault will become, or **nil** if one can't be found. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **methodSignatureForSelector:** (EOFaultHandler)

release

– (void)**release**

Performs as NSObject's **release** method.

respondsToSelector:

– (BOOL)**respondsToSelector:**(SEL)*aSelector*

Returns YES if the object that the receiving EOFault will become responds to *aSelector*, NO otherwise. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **respondsToSelector:forFault:** (EOFaultHandler)

retain

– (id)**retain**

Performs as NSObject's **retain** method.

retainCount

– (unsigned int)**retainCount**

Performs as NSObject's **retainCount** method.

self

– (id)**self**

Fires the receiver and returns **self**. This is the recommended way to simply fire an EOFault.

superclass

– (Class)**superclass**

Returns the superclass of the object that the receiving EOFault will become. This may cause the EOFault to fire in order to determine its actual class membership.

See also: – **classForFault:** (EOFaultHandler)

zone

– (NSZone *)**zone**

Performs as NSObject's **zone** method.

EOFaultHandler

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOFault.h

Class Description

EOFaultHandler is an abstract class that defines the mechanisms that create EOFaults (or faults) and help them to fire. *Faults* are used as placeholders for an enterprise object's relationship destinations. For example, suppose an Employee object has a **department** relationship to the employee's department. When an employee is fetched, faults are created for its relationship destinations. In the case of the **department** relationship, an empty Department object is created. The Department object's data isn't fetched until the Department is accessed, at which time the fault is said to *fire*.

Subclasses of EOFaultHandler perform the specific steps necessary to get data for the fault and fire it. The Access Layer, for example, uses private subclasses to fetch data using an EODatabaseContext (defined in EOAccess). Most of EOFaultHandler's methods are properly defined; you need only override **completeInitializationOfObject:** to provide appropriate behavior. In Yellow Box applications, you can optionally implement **faultWillFire:** to prepare for conversion, and **shouldPerformInvocation:** to intercept particular messages sent to the fault without causing it to fire.

In a Yellow Box application you create an EOFaultHandler using the standard **alloc** and **init** methods, possibly using a more specific **init** method with your subclass. To create a fault you invoke EOFault's **makeObjectIntoFault:withHandler:** class method with the object to turn into a fault and the EOFaultHandler. An EOFaultHandler belongs exclusively to a single fault, and shouldn't be shared or used by any other object.

Firing a Fault

When a fault receives a message that requires it to fire, it sends a **completeInitializationOfObject:** method to its EOFaultHandler. This method is responsible for invoking EOFault's **clearFault:**class method to revert the fault to its original state, and then do whatever is necessary to complete initialization of the object. Doing so typically involves fetching data from an external repository and passing it to the object.

As a trivial example, consider a subclass called FileFaultHandler, that simply stores a filename whose contents it reads from disk. Its initialization and **completeInitializationOfObject:** methods might look like these:

```
- (id)initWithFile:(NSString *)path
{
    self = [super init];
    filename = [path copy];
    return self;
}

- (void)completeInitializationOfObject:(id)anObject
{
    NSString *fileContents;

    [self retain];      // retain self so we won’t get released by clearing the
                        // fault. Otherwise, accessing “filename” will cause a
crash.

    [EOFault clearFault:anObject];

    fileContents = [NSString stringWithContentsOfFile:filename];
    [anObject takeValue:fileContents forKey:@"fileContents"];
    [self release];
    return;
}
```

initWithFile: just stores the path of the file to read in the instance variable **filename**. **completeInitializationOfObject:** invokes EOFault’s **clearFault:** method, which reverts the fault into its original state (and also releases the fault handler, so references to **self** after this are illegal). It then gets the contents of the file it was created with and passes them to the reverted object. Note that this implementation doesn’t assume the class of the cleared EOFault, instead using the generic **takeValue:forKey:** method to assign the file contents to it.

Method Types

Setting the target class and extra data	
	– setTargetClass:extraData:
	– targetClass
	– extraData
Reference counting	
	– incrementExtraRefCount
	– decrementExtraRefCountIsZero
	– extraRefCount
Getting the original class	
	– classForFault:

Firing a fault	
	– completeInitializationOfObject:
	– faultWillFire:
	– shouldPerformInvocation:
Getting a description	
	– descriptionForObject:
Checking class information	
	– isKindOfClass:forFault:
	– isMemberOfClass:forFault:
	– conformsToProtocol:forFault:
	– methodSignatureForSelector:forFault:
	– respondsToSelector:forFault:

Instance Methods

classForFault:

– (Class)**classForFault:**(id)*fault*

Returns the target class of the receiver's EOFault, which must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **targetClass**

completeInitializationOfObject:

– (void)**completeInitializationOfObject:**(id)*aFault*

Implemented by subclasses to revert *aFault* to its original state and complete its initialization in whatever means is appropriate to the subclass. For example, the Access layer subclasses of EOFaultHandler fetch data from the database and pass it to the object. This method is invoked automatically by an EOFaultwhen it's sent a message it can't handle without fetching its data. EOFaultHandler's implementation merely throws an exception.

conformsToProtocol:forFault:

– (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol* **forFault:**(id)*aFault*

Returns YES if the target class of the receiver's EOFault conforms to *aProtocol*. This EOFault must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **completeInitializationOfObject:**

decrementExtraRefCountIsZero

– (BOOL)**decrementExtraRefCountIsZero**

Decrements the reference count for the receiver's EOFault. An object's reference count is the number of objects that are accessing it. Newly created objects have a reference count of one. If another object is referencing an object, the object is said to have an *extra reference count*.

If, after decrementing the reference count, the fault's new reference count is zero, this method returns YES, If the reference count has not become zero, this method returns NO. Objects that have a zero reference count are released at the end of the current event loop.

This method is used by EOFaultHandler's internal reference counting mechanism—it functions as the Foundation function **NSDecrementExtraRefCountWasZero()** for the receiver's EOFault.

descriptionForObject:

– (NSString *)**descriptionForObject:**(id)*aFault*

Returns a string naming the original class of the receiver's EOFault and giving *aFault*'s **id**, and also noting that it's a fault; for example: "<Employee (Fault 0x3a07)>". (The fault must be passed as *aFault* because EOFaultHandlers don't store back pointers to their faults.)

extraData

– (void *)**extraData**

Returns the bytes replaced by the receiver's **id** in the original object's state, as a pointer to **void**. When the receiver's EOFault is reverted to its original state, both its **isa** pointer and this data are replaced.

extraRefCount

– (unsigned int)**extraRefCount**

Returnsthe receiver’s current reference count. This method is used by EOFaultHandler’s internal reference counting mechanism and functions as the Foundation function **NSExtraRefCount()** for the receiver’s EOFault.

faultWillFire:

– (void)**faultWillFire:**(id)*aFault*

Informs the receiver that *aFault* is about to be reverted to its original state. EOFaultHandler’s implementation does nothing. This method is invoked by EOFault’s **clearFault:** method.

incrementExtraRefCount

– (void)**incrementExtraRefCount**

Increments the reference count for the receiver’s EOFault. An object’s reference count is the number of objects that are accessing it. Newly created objects have a reference count of one. If another object is referencing an object, the object is said to have an *extra reference count*.

This method is used by EOFaultHandler’s internal reference counting mechanism and functions as the Foundation function **NSIncrementExtraRefCount()** for the receiver’s EOFault.

See also: – **extraRefCount**

isKindOfClass:forFault:

– (BOOL)**isKindOfClass:**(Class)*aClass* **forFault:**(id)*aFault*

Returns YES if the target class of the receiver’s EOFault is *aClass* or a subclass of *aClass*. The fault must be passed in as *aFault* in case the receiver needs to fire it (EOFaultHandlers don’t store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **completeInitializationOfObject:**

isMemberOfClass:forFault:

– (BOOL)**isMemberOfClass:**(Class)*aClass* **forFault:**(id)*aFault*

Returns YES if the target class of the receiver’s EOFault is *aClass*. This fault must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don’t store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **completeInitializationOfObject:**

methodSignatureForSelector:forFault:

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector* **forFault:**(id)*aFault*

Returns the NSMethodSignature for *aSelector* in the target class of the receiver's EOFault, which must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **completeInitializationOfObject:**

respondsToSelector:forFault:

– (BOOL)**respondsToSelector:**(SEL)*aSelector* **forFault:**(id)*aFault*

Returns YES if the target class of the receiver's EOFault responds to *aSelector*. This fault must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't store back pointers to their faults). For example, to support entity inheritance, the Access layer fires faults for entities with subentities to confirm their precise class membership.

See also: – **completeInitializationOfObject:**

setTargetClass:extraData:

– (void)**setTargetClass:**(Class)*targetClass* **extraData:**(void *)*extraData*

Stores *targetClass* and *extraData* as state of the original object overwritten when an EOFault is created by EOFault's **makeObjectIntoFault:withHandler:** <<should be XRef>> method, which replaces *targetClass* with the EOFault class, and *extraData* with the EOFaultHandler's **id**.

shouldPerformInvocation:

– (BOOL)**shouldPerformInvocation:**(NSInvocation *)*anInvocation*

Overridden by subclasses to circumvent reversion of an EOFault to its original state. Returns YES if the EOFault should revert and perform *anInvocation*, NO if it shouldn't. If this method returns NO, the receiver should set *anInvocation*'s return value appropriately. EOFaultHandler's implementation returns YES.

See also: – **setReturnValue:** (NSInvocation class of the Foundation Framework)

targetClass

– (Class)**targetClass**

Returns the target class of the receiver’s EOFault . The EOFault may, however, be converted to a member of this class or of a subclass of this class. For example, to support entity inheritance, the Access layer fires EOFaults for entities with subentities into the appropriate class on fetching their data.

EOFetchSpecification

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	EOControl/EOFetchSpecification.h

Class Description

An EOFetchSpecification collects the criteria needed to select and order a group of records or enterprise objects, whether from an external repository such as a relational database or an internal store such as an EOEditingContext. An EOFetchSpecification contains these elements:

- The name of an entity for which to fetch records or objects. This is the only mandatory element.
- An EOQualifier, indicating which properties to select by and how to do so.
- An array of EOSortOrderings, which indicate how the selected records or objects should be ordered when fetched.
- An indicator of whether to produce distinct results or not. Normally if a record or object is selected several times, such as when forming a join, it appears several times in the fetched results. An EOFetchSpecification that makes distinct selections causes duplicates to be filtered out, so each record or object selected appears exactly once in the result set.
- An indicator of whether to fetch deeply or not. This is used with inheritance hierarchies when fetching for an entity with sub-entities. A deep fetch produces all instances of the entity and its sub-entities, while a shallow fetch produces instances only of the entity in the fetch specification.
- A fetch limit indicating how many objects to fetch before giving the user or program an opportunity to intervene.
- A listing of relationships for which the destination of the relationship should be prefetched along with the entity being fetched. Proper use of this feature allows for substantially increased performance in some cases.
- A dictionary of hints, which an EODatabaseContext or other object can use to optimize or alter the results of the fetch.

EOFetchSpecifications are most often used with the method **objectsWithFetchSpecification: editingContext:**, defined by EOObjectStore, EOEditingContext, and EODatabaseContext, as well as

objectsWithFetchSpecification:, defined by EOEditingContext alone. EOAdaptorChannel and EODatabaseChannel also define methods that use EOFetchSpecifications.

Adopted Protocols

NSCoding	
	– encodeWithCoder:
	– initWithCoder:
NSCopying	
	– copyWithZone:

Method Types

Creating instances	
	+ fetchSpecificationWithEntityName:qualifier:sortOrderings:
	– fetchSpecificationWithQualifierBindings:
	– init
	– initWithEntityName:qualifier:sortOrderings:usesDistinct: isDeep:hints:
Setting the qualifier	
	– setQualifier:
	– qualifier
Sorting	
	– setSortOrderings:
	– sortOrderings:
Removing duplicates	
	– setUsesDistinct:
	– usesDistinct:
Fetching objects in an inheritance hierarchy	
	– setIsDeep:
	– isDeep
	– setEntityName:
	– entityName

Controlling fetching behavior	
	– setFetchLimit:
	– fetchLimit
	– setFetchesRawRows:
	– fetchesRawRows
	– setPrefetchingRelationshipKeyPaths:
	– prefetchingRelationshipKeyPaths
	– setPromptsAfterFetchLimit:
	– promptsAfterFetchLimit
	– setRawRowKeyPaths:
	– rawRowKeyPaths
	– setRequiresAllQualifierBindingVariables:
	– requiresAllQualifierBindingVariables
	– setHints:
	– hints
Locking objects	
	– setLocksObjects:
	– locksObjects
Refreshing refetched objects	
	– setRefreshesRefetchedObjects:
	– refreshesRefetchedObjects

Class Methods

fetchSpecificationWithEntityName:qualifier:sortOrderings:

+ (EOFetchSpecification *)**fetchSpecificationWithEntityName:**(NSString *)*entityName*
qualifier:(EOQualifier *)*qualifier*
sortOrderings:(NSArray *)*sortOrderings*

Returns an EOFetchSpecification for *entityName*, using *qualifier* to select and *sortOrderings* to sort the results. The EOFetchSpecification created with this method is deep, doesn't perform distinct selection, and has no hints.

See also: – **initWithEntityName:qualifier:sortOrderings:usesDistinct:isDeep:hints:**

Instance Methods

entityName

– (NSString *)**entityName**

Returns the name of the entity to be fetched.

See also: – **isDeep**, – **setEntityName:**

fetchLimit

– (unsigned)**fetchLimit**

Returns the fetch limit value which indicates the maximum number of objects to fetch. Depending on the value of promptsAfterFetchLimit, the EODatabaseContext will either stop fetching objects when this limit is reached or it will ask the editing context's message handler to prompt the user as to whether or not it should continue fetching. Use 0 (zero) to indicate no fetch limit. The default is 0.

See also: – **setFetchLimit:**

fetchesRawRows

– (BOOL)**fetchesRawRows**

Returns YES if **rawRowKeyPaths** returns non-nil.

See also: – **rawRowKeyPaths**, – **setFetchesRawRows:**

fetchSpecificationWithQualifierBindings:

– (EOFetchSpecification *)**fetchSpecificationWithQualifierBindings:**(NSDictionary *)*bindings*

Applies bindings from *bindings* to its qualifier if there is one, and returns a new fetch specification that can be used in a fetch. The default behavior is to prune any nodes for which there are no bindings. Invoke **setRequiresAllQualifierBindingVariables:** with an argument of YES to force an exception to be raised if a binding is missing during variable substitution.

See also: – **setRequiresAllQualifierBindingVariables:**

hints

– (NSDictionary *)**hints**

Returns the receiver’s hints, which other objects can use to alter or optimize fetch operations.

See also: – **setHints:**

init

– (id)**init**

Initializes a new EOFetchSpecification with no state, except that it fetches deeply and doesn’t use distinct. Use the **set...** methods to add other parts of the specification. This is the designated initializer for the EOFetchSpecification class. Returns **self**.

See also: – **initWithEntityName:qualifier:sortOrderings:usesDistinct:isDeep:hints:**

initWithEntityName:qualifier:sortOrderings:usesDistinct:isDeep:hints:

– (id)**initWithEntityName:**(NSString *)*entityName*
qualifier:(EOQualifier *)*qualifier*
sortOrderings:(NSArray *)*sortOrderings*
usesDistinct:(BOOL)*distinctFlag*
isDeep:(BOOL)*deepFlag*
hints:(NSDictionary *)*hints*

Initializes a new EOFetchSpecification with the given arguments. Returns **self**.

See also: + **fetchSpecificationWithEntityName:qualifier:sortOrderings:**

isDeep

– (BOOL)**isDeep**

Returns YES if a fetch should include sub-entities of the receiver’s entity, NO if it shouldn’t. EOFetchSpecifications are deep by default.

For example, if you have a Person entity with two sub-entities, Employee and Customer, fetching Persons deeply also fetches all Employees and Customers matching the qualifier. Fetching Persons shallowly fetches only Persons matching the qualifier.

See also: – **setIsDeep:**

locksObjects

– (BOOL)**locksObjects**

Returns YES if a fetch should result in the selected objects being locked in the data repository, NO if it shouldn’t. The default is NO.

See also: – **setLocksObjects:**

prefetchingRelationshipKeyPaths

– (NSArray *)**prefetchingRelationshipKeyPaths**

Returns an array of relationship key paths that should be prefetched along with the main fetch. For example, if fetching from the Movie entity, you might specify paths of the form (@"directors", @"roles.talent", @"plotSummary").

See also: – **setPrefetchingRelationshipKeyPaths:**

promptsAfterFetchLimit

– (BOOL)**promptsAfterFetchLimit**

Returns whether to prompt user after the fetch limit has been reached. Default is NO.

See also: – **setPromptsAfterFetchLimit:**

qualifier

– (EOQualifier *)**qualifier**

Returns the EOQualifier that indicates which records or objects the receiver is to fetch.

See also: – **setQualifier:**

rawRowKeyPaths

– (NSArray *)**rawRowKeyPaths**

Returns an array of attribute key paths that should be fetched as raw data and returned as an array of dictionaries (instead of the normal result of full objects). The raw fetch can increase speed, but forgoes most of the benefits of full Enterprise Objects. The default value is nil, indicating that full objects will be returned from the fetch. An empty array may be used to indicate that the fetch should query the entity named by the fetch specification using the method **attributesToFetch**. As long as the primary key attributes are included in the raw attributes, the raw row may be used to generate a fault for the corresponding object using EOEditingContext's **faultForRawRow:entityNamed:** method.

See also: – **fetchesRawRows**, – **setFetchesRawRows:**, – **setRawRowKeyPaths:**

refreshesRefetchedObjects

– (BOOL)**refreshesRefetchedObjects**

Returns YES if existing objects are overwritten with fetched values when they've been updated or changed. Returns NO if existing objects aren't touched when their data is refetched (the fetched data is simply discarded). The default is NO. Note that this setting does not affect relationships

See also: – **setRefreshesRefetchedObjects:**

requiresAllQualifierBindingVariables

– (BOOL)**requiresAllQualifierBindingVariables**

Returns YES to indicate that a missing binding will cause an exception to be raised during variable substitution. The default value is NO, which says to prune any nodes for which there are no bindings.

See also: – **setRequiresAllQualifierBindingVariables:**

setEntityName:

– (void)**setEntityName:**(NSString *)*entityName*

Sets the name of the root entity to be fetched to *entityName*.

See also: – **isDeep**, – **entityName**

setFetchesRawRows:

– (void)**setFetchesRawRows:**(BOOL)*fetchRawRows*

Sets the behavior for fetching raw rows. If set to YES, the behavior is the same as if **setRawRowKeyPaths:** were called with an empty array. If set to NO, the behavior is as if **setRawRowKeyPaths:** were called with a nil argument.

See also: – **fetchesRawRows**, – **setRawRowKeyPaths:**, – **rawRowKeyPaths**

setFetchLimit:

– (void)**setFetchLimit:**(unsigned)*fetchLimit*

Sets the fetch limit value which indicates the maximum number of objects to fetch. Depending on the value of promptsAfterFetchLimit, the EODatabaseContext will either stop fetching objects when this limit is reached or it will ask the editing context's message handler to prompt the user as to whether or not it should continue fetching. Use 0 (zero) to indicate no fetch limit. The default is 0.

See also: – **fetchLimit**

setHints:

– (void)**setHints:**(NSDictionary *)*hints*

Sets the receiver's hints to *hints*. Any object that uses an EOFetchSpecification can define its own hints that it uses to alter or optimize fetch operations. For example, EODatabaseContext uses a hint identified by the key EOCustomQueryExpressionHintKey. EODatabaseContext is the only class in Enterprise Objects Framework that defines fetch specification hints. For information about EODatabaseContext's hints, see the EODatabaseContext class specification.

See also: – **hints**

setIsDeep:

– (void)**setIsDeep:**(BOOL)*flag*

Controls whether a fetch should include sub-entities of the receiver's entity. If *flag* is YES, sub-entities are also fetched; if *flag* is NO, they aren't. EOFetchSpecifications are deep by default.

For example, if you have a Person entity /class /table with two sub-entities and subclasses, Employee and Customer, fetching Persons deeply also fetches all Employees and Customers matching the qualifier, while fetching Persons shallowly fetches only Persons matching the qualifier.

See also: – **isDeep**

setLocksObjects:

– (void)**setLocksObjects:**(BOOL)*flag*

Controls whether a fetch should result in the selected objects being locked in the data repository. If *flag* is YES it should, if NO it shouldn't. The default is NO.

See also: – **locksObjects**

setPrefetchingRelationshipKeyPaths:

– (void)**setPrefetchingRelationshipKeyPaths:**(NSArray *)*prefetchingRelationshipKeyPaths*

Sets an array of relationship key paths that should be prefetched along with the main fetch. For example, if fetching from the Movie entity, you might specify paths of the form (@"directors", @"roles.talent", @"plotSummary").

See also: – **prefetchingRelationshipKeyPaths**

setPromptsAfterFetchLimit:

– (void)**setPromptsAfterFetchLimit:**(BOOL)*promptsAfterFetchLimit*

Sets whether to prompt user after the fetch limit has been reached. Default is NO.

See also: – **promptsAfterFetchLimit**

setQualifier:

– (void)**setQualifier:**(EOQualifier *)*qualifier*

Sets the receiver’s qualifier to *qualifier.*

See also: – **qualifier**

setRawRowKeyPaths:

– (void)setRawRowKeyPaths:(NSArray *)*rawRowKeyPaths*

Sets an array of attribute key paths that should be fetched as raw data and returned as an array of dictionaries (instead of the normal result of full objects). The raw fetch can increase speed, but forgoes most of the benefits of full Enterprise Objects. The default value is nil, indicating that full objects will be returned from the fetch. An empty array may be used to indicate that the fetch should query the entity named by the fetch specification using the method **attributesToFetch**. As long as the primary key attributes are included in the raw attributes, the raw row may be used to generate a fault for the corresponding object using EOEditingContext’s **faultForRawRow:entityNamed:** method.

See also: – **fetchesRawRows**, – **rawRowKeyPaths**, – **setFetchesRawRows:**

setRefreshesRefetchedObjects:

– (void)**setRefreshesRefetchedObjects:**(BOOL)*flag*

Controls whether existing objects are overwritten with fetched values when they have been updated or changed. If *flag* is YES, they are; if *flag* is NO, they aren't (the fetched data is simply discarded). The default is NO.

For example, suppose that you fetch an employee object and then refetch it, without changing the employee between fetches. In this case, you want to refresh the employee when you refetch it, because another application might have updated the object since your first fetch. To keep your employee in sync with the employee data in the external repository, you'd need to replace the employee's outdated values with the new ones. On the other hand, if you were to fetch the employee, change it, and then refetch it, you would not want to refresh the employee. If you to refreshed it—whether or not another application had changed the employee—you would lose the changes that you had made to the object.

You can get finer-grain control on an EODatabaseContext's refreshing behavior than you can with an EOFetchSpecification by using the delegate method **databaseContext:shouldUpdateCurrentSnapshot: newSnapshot:globalID:databaseChannel:**. For more information see the EODatabaseContext class specification.

See also: – **refreshesRefetchedObjects**

setRequiresAllQualifierBindingVariables:

– (void)**setRequiresAllQualifierBindingVariables:**(BOOL)*allVariablesRequired*

Sets the behavior when a missing binding is encountered during variable substitution. If *allVariablesRequired* is YES, then a missing binding will cause an exception to be raised during variable substitution. The default value is NO, which says to prune any nodes for which there are no bindings.

See also: – **fetchSpecificationWithQualifierBindings:**, – **requiresAllQualifierBindingVariables**

setSortOrderings:

– (void)**setSortOrderings:**(NSArray *)*sortOrderings*

Sets the receiver's array of EOSortOrderings to *sortOrderings*. When a fetch is performed with the receiver, the results are sorted by applying each EOSortOrdering in the array.

See also: – **sortedArrayUsingKeyOrderArray:** (NSArray Additions), – **sortOrderings:**

setUsesDistinct:

– (void)**setUsesDistinct:**(BOOL)*flag*

Controls whether duplicate objects or records are removed after fetching. If *flag* is YES they're removed; if *flag* is NO they aren't. EOFetchSpecifications by default don't use distinct.

See also: – **usesDistinct:**

sortOrderings:

– (NSArray *) **sortOrderings**

Returns the receiver's array of EOSortOrderings. When a fetch is performed with the receiver, the results are sorted by applying each EOSortOrdering in the array.

See also: – **sortedArrayUsingKeyOrderArray:** (NSArray Additions), – **setSortOrderings:**

usesDistinct:

– (BOOL)**usesDistinct**

Returns YES if duplicate objects or records are removed after fetching, NO if they aren't. EOFetchSpecifications by default don't use distinct.

See also: – **setUsesDistinct:**

EOGenericRecord

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOGenericRecord.h

Class Description

EOGenericRecord is a generic enterprise object class that can be used in place of custom classes when you don't need custom behavior. It implements the EOEnterpriseObject interface to provide the basic enterprise object behavior. An EOGenericRecord object has an EOClassDescription that provides metadata about the generic record, including the name of the entity that the generic record represents and the names of the record's attributes and relationships. A generic record stores its properties in a dictionary using its attribute and relationship names as keys.

In the typical case of applications that access a relational database, the access layer's modeling objects are an important part of how generic records map to database rows: If an EOModel doesn't have a custom enterprise object class defined for a particular entity, an EODatabaseChannel using that model creates EOGenericRecords when fetching objects for that entity from the database server. During this process, the EODatabaseChannel also sets each generic record's class description to an EOEntityClassDescription, providing the link to the record's associated modeling objects. (EOModel, EODatabaseChannel, and EOEntityClassDescription are defined in EOAccess.)

Creating an Instance of EOGenericRecord

The best way to create an instance of EOGenericRecord is using the EOClassDescription method **createInstanceWithEditingContext:globalID:zone:** as follows:

```
id newEO;
NSString *entityName;             // Assume this exists.

newEO = [[EOClassDescription classDescriptionForEntityName:entityName]
    createInstanceWithEditingContext:nil
    globalID:nil
     zone:nil];
```

createInstanceWithEditingContext:globalID:zone: is preferable to EOGenericRecord's **init...** method because the same code works if you later use a custom enterprise object class instead of EOGenericRecord. You can get an EOClassDescription for an entity name as shown above. Alternatively, you can get an EOClassDescription for a destination key of an existing enterprise object as follows:

```
id newEO;
id existingEO;              // Assume this exists.
NSString *relationshipName; // Assume this exists.
EOClassDescription *description = [existingEO classDescription];

newEO = [[description classDescriptionForDestinationKey:relationshipName]
    createInstanceWithEditingContext:editingContext
    lobalID:nil
    zone:nil];
```

The technique in this example is useful for inserting a new destination object into an existing enterprise object—for creating a new Movie object to add to a Studio's array of Movies, for example.

Instance Methods

initWithEditingContext:classDescription:globalID:

– (id)**initWithEditingContext:**(EOEditingContext *)*anEditingContext*
    classDescription:(EOClassDescription *)*aClassDescription*
    globalID:(EOGlobalID *)*globalID*

The designated initializer, this method initializes a newly allocated EOGenericRecord to get its metadata from *aClassDescription*. You should pass **nil** for *anEditingContext* and *globalID*, because the arguments are optional: EOGenericRecord's implementation does nothing with them. Raises an NSInternalInconsistencyException if *aClassDescription* is **nil**. Returns **self**.

You shouldn't use this method to create new EOGenericRecords. Rather, use EOClassDescription's **createInstanceWithEditingContext:globalID:zone:** method. See the class description for more information.

storedValueForKey:

– (id)**storedValueForKey:**(NSString *)*key*

Overrides the default implementation to simply invoke **valueForKey:**.

See also: **storedValueForKey:** (EOKeyValueCoding)

takeStoredValue:forKey:

– (void)**takeStoredValue:**(id)*value*
  forKey:(NSString *)*key*

Overrides the default implementation to simply invoke **takeValue:forKey:**.

See also: **takeStoredValue:forKey:** (EOKeyValueCoding)

takeValue:forKey:

– (void)**takeValue:**(id)*value*
  forKey:(NSString *)*key*

Invokes the receiver's **willChange** method, and sets the value for the property identified by *key* to *value*. If *value* is **nil**, this method removes the receiver's dictionary entry for *key*. (EOGenericRecord overrides the default implementation.) If *key* is not one of the receiver's attribute or relationship names, EOGenericRecord's implementation does not invoke **handleTakeValue:forUnboundKey:**. Instead, EOGenericRecord's implementation does nothing.

valueForKey:

– (id)**valueForKey:**(NSString *)*key*

Returns the value for the property identified by *key*. (EOGenericRecord overrides the default implementation.) If *key* is not one of the receiver's attribute or relationship names, EOGenericRecord's implementation does not invoke **handleQueryWithUnboundKey:**. Instead, EOGenericRecord's implementation simply returns **nil**.

EOGlobalID

Inherits From:	NSObject
Conforms To:	NSCopying NSObject (NSObject)
Declared In:	EOControl/EOGlobalID.h

Class Description

An EOGlobalID is a compact, universal identifier for a persistent object, forming the basis for uniquing in Enterprise Objects Framework. An EOGlobalID uniquely identifies the same object or record both between EOEditingContexts in a single application and in multiple applications (as in distributed systems). EOGlobalID is an abstract class, declaring only the methods needed for identification. A concrete subclass must define appropriate storage for identifying values (such as primary keys), as well as an initialization or creation method to build IDs. See the EOKeyGlobalID class specification for an example of a concrete ID class.

Temporary Identifiers

EOEditingContexts and other object stores support the insertion of new objects without established IDs, creating temporary IDs that get replaced with permanent ones as soon as the new objects are saved to their persistent stores. The temporary IDs are instances of the EOTemporaryGlobalID class.

When an EOObjectStore saves these newly inserted objects, it must replace the temporary IDs with persistent ones. When it does this, it must post an EOGlobalIDChangedNotification announcing the change so that observers can update their accounts of which objects are identified by which global IDs. The notification's **userInfo** dictionary contains a mapping from the temporary IDs (the keys) to their permanent replacements (the values).

Adopted Protocols

NSCopying	– copyWithZone:

Instance Methods

isTemporary

– (BOOL)**isTemporary**

Returns NO. See the class description for more information.

Notifications

EOGlobalIDChangedNotification

Posted whenever EOTemporaryGlobalIDs are replaced by permanent EOGlobalIDs. The notification contains:

Notification Object	**nil**
Userinfo	A mapping from the temporary IDs (keys) to permanent IDs (values)

EOKeyComparisonQualifier

Inherits From:	EOQualifier : NSObject
Conforms To:	EOQualifierEvaluation EOQualifierSQLGeneration
Declared In:	EOControl/EOQualifier.h

Class Description

EOKeyComparisonQualifier is a subclass of EOQualifier that compares a named property of an object with a named value of another object. For example, to return all of the employees whose salaries are greater than those of their managers, you might use an expression such as "salary > manager.salary", where "salary" is the *left key* and "manager.salary" is the *right key*. The "left key" is the property of the first object that's being compared to a property in a second object; the property in the second object is the "right key." Both the left key and the right key might be key paths. You can use EOKeyComparisonQualifier to compare properties of two different objects or to compare two properties of the same object.

EOKeyComparisonQualifier adopts the EOQualifierEvaluation protocol, which defines the method **evaluateWithObject:** for in-memory evaluation. When an EOKeyComparisonQualifier object receives an **evaluateWithObject:** message, it evaluates the given object to determine if it satisfies the qualifier criteria.

In addition to performing in-memory filtering, EOKeyComparisonQualifier can be used to generate SQL. When it's used for this purpose, the key should be a valid property name of the root entity for the qualifier (or a valid key path).

Adopted Protocols

EOQualifierEvaluation	
	– evaluateWithObject:
EOQualifierSQLGeneration	
	– sqlStringForSQLExpression: – schemaBasedQualifierWithRootEntity:

Instance Methods

evaluateWithObject:

@protocol EOQualifierEvaluation
– (BOOL)**evaluateWithObject:***object*

Returns YES if the object *object* satisfies the qualifier, NO otherwise. When an EOKeyComparisonQualifier object receives an **evaluateWithObject:** message, it evaluates *object* to determine if it meets the qualifier criteria. This method can raise one of several possible exceptions if an error occurs. If your application allows users to construct arbitrary qualifiers (such as through a user interface), you may want to write code to catch any exceptions and properly respond to errors (for example, by displaying a panel saying that the user typed a poorly formed qualifier).

initWithLeftKey:operatorSelector:rightKey:

– **initWithLeftKey:**(NSString *)*leftKey* **operatorSelector:**(SEL)*selector* **rightKey:**
(NSString *)*rightKey*

Initializes the receiver to compare the properties named by *leftKey* and *rightKey*, using the operator selector *selector.*

- EOQualifierOperatorEqual
- EOQualifierOperatorNotEqual
- EOQualifierOperatorLessThan
- EOQualifierOperatorGreaterThan
- EOQualifierOperatorLessThanOrEqualTo
- EOQualifierOperatorGreaterThanOrEqualTo
- EOQualifierOperatorContains
- EOQualifierOperatorLike
- EOQualifierOperatorCaseInsensitiveLike

Enterprise Objects Framework supports SQL generation for these selectors only.

For example, the following excerpt creates an EOKeyComparisonQualifier **qual** that has the left key “lastName”, the operator selector EOQualifierOperatorEqual, and the right key “member.lastName”. Once constructed, the qualifier **qual** is used to filter an in-memory array. The code excerpt returns an array of Guest objects whose **lastName** properties have the same value as the **lastName** property of the guest’s sponsoring member (this example is based on the Rentals sample database).

```
NSArray *guests;    /* Assume this exists. */
EOQualifier *qual = [[EOKeyComparisonQualifier alloc]
    initWithLeftKey:@"lastName"
    operatorSelector:EOQualifierOperatorEqual
    rightKey:@"member.lastName"];

return [guests filteredArrayUsingQualifier:qual];
```

leftKey

– (NSString *)**leftKey**

Returns the receiver's left key.

rightKey

– (NSString *)**rightKey**

Returns the receiver's right key.

selector

– (SEL)**selector**

Returns the receiver's selector.

EOKeyGlobalID

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying (EOGlobalID) NSObject (NSObject)
Declared In:	EOAccess/EOKeyGlobalID.h

Class Description

EOKeyGlobalID is a concrete subclass of EOGlobalID whose instances represent persistent IDs based on EOModel information: an entity and the primary key values for the object being identified. When creating an EOKeyGlobalID, the key values must be supplied following alphabetical order for their attribute names. EOKeyGlobalID defines the **globalIDWithEntityName:keys:keyCount:zone:** for creating instances, but it's much more convenient to create instances from fetched rows using EOEntity's **globalIDForRow:** method. (EOEntity and EOModel are defined in EOAccess.)

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Creating instances

+ globalIDWithEntityName:keys:keyCount:zone:

Getting the entity name

– entityName

Getting the key values

– keyValues
– keyCount
– keyValuesArray

Comparison

– isEqual:

Class Methods

globalIDWithEntityName:keys:keyCount:zone:

+ (id)**globalIDWithEntityName:**(NSString *)*entityName*
 keys:(id *)*keyValues*
 keyCount:(unsigned int)*count*
 zone:(NSZone *)*zone*

Returns an EOKeyGlobalID based on *entityName* and *keyValues*. For performance reasons, the key values are given as a C array of **id**; *count* indicates how many key values there are. The object returned is allocated from *zone*.

EOKeyGlobalIDs are more conveniently created using EOEntity's **globalIDForRow:** method (EOAccess).

Instance Methods

entityName

– (NSString *)**entityName**

Returns the name of the entity governing the object identified by the receiver. This is used by EODatabaseContexts (EOAccess) to identify an EOEntity (EOAccess) in methods such as **faultForGlobalID:editingContext:**.

hash– (unsigned int)**hash**

Returns an integer that can be used as a table address in a hash table structure. If two objects are equal (as determined by **isEqual:**), they must have the same hash value. For more information, see the descriptions of this method in the NSObject class and protocol specifications of the Foundation Framework.

isEqual:

@protocol NSObject
– (BOOL)**isEqual:**(id)*anObject*

Returns YES if the receiver and *anObject* share the same entity name and key values, NO if they don't. For more information, see the descriptions of this method in the NSObject class and protocol specifications of the Foundation Framework.

See also: – **entityName**, – **keyValues**

keyCount

– (unsigned int)**keyCount**

Returns the number of key values in the receiver.

keyValues

– (id *)**keyValues**

Returns the receiver’s key values as a C array of **id** (for performance reasons).

keyValuesArray

– (NSArray *)**keyValuesArray**

Returns the receiver’s key values as an NSArray.

EOKeyValueQualifier

Inherits From:	EOQualifier : NSObject
Conforms To:	EOQualifierEvaluation EOQualifierSQLGeneration
Declared In:	EOControl/EOQualifier.h

Class Description

EOKeyValueQualifier is a subclass of EOQualifier that compares a named property of an object with a supplied value, for example, “salary > 1500”. EOKeyValueQualifier adopts the EOQualifierEvaluation protocol, which defines the method **evaluateWithObject:** for in-memory evaluation. When an EOKeyValueQualifier object receives an **evaluateWithObject:** message, it evaluates the given object to determine if it satisfies the qualifier criteria.

In addition to performing in-memory filtering, EOKeyValueQualifier can be used to generate SQL. When it’s used for this purpose, the key should be a valid property name of the root entity for the qualifier (or a valid key path).

Adopted Protocols

EOQualifierEvaluation	
	– evaluateWithObject:
EOQualifierSQLGeneration	
	– sqlStringForSQLExpression: – schemaBasedQualifierWithRootEntity:

Instance Methods

evaluateWithObject

@protocol EOQualifierEvaluation
– (BOOL)**evaluateWithObject:***anObject*

Returns YES if the object *anObject* satisfies the qualifier, NO otherwise. When an EOKeyValueQualifier object receives the **evaluateWithObject:** message, it evaluates *anObject* to determine if it meets the qualifier criteria. This method can raise one of several possible exceptions if an error occurs. If your application allows users to construct arbitrary qualifiers (such as through a user interface), you may want to write code to catch any exceptions and properly respond to errors (for example, by displaying a panel saying that the user typed a poorly formed qualifier).

initWithKey:operatorSelector:value:

– **initWithKey:**(NSString *)*key* **operatorSelector:**(SEL)*selector* **value:**(id)*value*

Initializes the receiver to compare values for *key* to *value* using the operator selector *selector*. The possible values for *selector* are as follows:

- EOQualifierOperatorEqual
- EOQualifierOperatorNotEqual
- EOQualifierOperatorLessThan
- EOQualifierOperatorGreaterThan
- EOQualifierOperatorLessThanOrEqualTo
- EOQualifierOperatorGreaterThanOrEqualTo
- EOQualifierOperatorContains
- EOQualifierOperatorLike
- EOQualifierOperatorCaseInsensitiveLike

Enterprise Objects Framework supports SQL generation for these selectors only.

For example, the following excerpt creates an EOKeyValueQualifier **qual** that has the key “name”, the operator selector EOQualifierOperatorEqual, and the value “Smith”. Once constructed, the qualifier **qual** is used to filter an in-memory array.

```
NSArray *employees;    /* Assume this exists. */
EOQualifier *qual = [[EOKeyValueQualifier alloc] initWithKey:@"name"
        operatorSelector:EOQualifierOperatorEqual
        value:@"Smith"];
return [employees filteredArrayUsingQualifier:qual];
```

key

– (NSString *)**key**

Returns the receiver’s key.

selector

– (SEL)**selector**

Returns the receiver’s selector.

value

– (id)**value**

Returns the receiver’s value.

EONotQualifier

Inherits From:	EOQualifier : NSObject
Conforms To:	EOQualifierEvaluation EOQualifierSQLGeneration
Declared In:	EOControl/EOQualifier.h

Class Description

EONotQualifier is a subclass of EOQualifier that contains a single qualifier. When an EONotQualifier object is evaluated, it returns the inverse of the result obtained by evaluating the qualifier it contains.

EONotQualifier adopts the EOQualifierEvaluation protocol, which defines the method **evaluateWithObject:** for in-memory evaluation. When an EONotQualifier object receives an **evaluateWithObject:** message, it evaluates the given object to determine if it satisfies the qualifier criteria.

Adopted Protocols

EOQualifierEvaluation	
	– evaluateWithObject:
EOQualifierSQLGeneration	
	– sqlStringForSQLExpression: – schemaBasedQualifierWithRootEntity:

Instance Methods

evaluateWithObject:

@protocol EOQualifierEvaluation
– (BOOL)**evaluateWithObject:***anObject*

Returns YES if the object *anObject* satisfies the EONotQualifier, NO otherwise. This method can raise one of several possible exceptions if an error occurs. If your application allows users to construct arbitrary qualifiers (such as through a user interface), you may want to put exception handlers around this method to properly respond to errors (for example, by displaying a panel saying that the user typed a poorly formed qualifier).

initWithQualifier:

– **initWithQualifier:**(EOQualifier *)*aQualifier*

Initializes the receiver with the EOQualifier *aQualifier*. For example, the following code excerpt constructs a qualifier, **baseQual**, and uses it to initialize an EONotQualifier, **negQual**. The EONotQualifier **negQual** is then used to filter an in-memory array. The code excerpt returns an array of Guest objects whose **lastName** properties do *not* have the same value as the **lastName** property of the guest's sponsoring member (this example is based on the Rentals sample database). In other words, the EONotQualifier **negQual** inverts the effects of **baseQual**.

```
NSArray *guests;     /* Assume this exists. */
EOQualifier *baseQual, *negQual;

baseQual = [EOQualifier qualifierWithQualifierFormat:@"lastName =
   member.lastName"];
negQual = [[EONotQualifier alloc] initWithQualifier:baseQual];
return [guests filteredArrayUsingQualifier:negQual];
```

qualifier

– (EOQualifier *)**qualifier**

Returns the receiver's qualifier.

EONull

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	EOControl/EONull.h

Class Description

The EONull class defines a unique object used to represent null values in collection objects (which don't allow **nil** values). For example, NSDictionaries fetched by an EOAdaptorChannel contain an EONull instance for such values. EONull is automatically translated to **nil** in enterprise objects, however, so most applications should rarely need to account for this class. See the NSObject Additions class specification for details on where this translation is performed.

EONull has exactly one instance, returned by the **null** class method. This object isn't reference-counted, can't be copied (**copyWithZone:** returns **self**), and is never deallocated. You can thus safely cache this instance and use pointer comparison to test for the presence of a null value:

```
static id NULL_VALUE;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    /* ... */
    NULL_VALUE = [EONull null];
    return;
}

if (value == NULL_VALUE) {
    /* ... */
}
```

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

EOSortOrderingComparison	
	– compareAscending:
	– compareCaseInsensitiveAscending:
	– compareCaseInsensitiveDescending:
	– compareDescending:
NSCopying	
	– copyWithZone:

Class Methods

null

+ (EONull *)**null**

Returns the unique instance of EONull.

EOObjectStore

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOObjectStore.h

Class Description

EOObjectStore is the abstract class that defines the API for an “intelligent” repository of objects, the control layer’s object storage abstraction. An object store is responsible for constructing and registering objects, servicing object faults, and saving changes made to objects. For more information on the object storage abstraction, see “Object Storage Abstraction” in the introduction to the EOControl Framework.

EOEditingContext is the principal EOObjectStore subclass and is used for managing objects in memory—in fact, the primary purpose of the EOObjectStore class is to define an API for servicing editing contexts, not to define a completely general API. Other subclasses of EOObjectStore are:

- EOCooperatingObjectStore
- EOObjectStoreCoordinator
- EODatabaseContext (EOAccess)

A subclass of EOObjectStore must implement all of its methods. The default implementations simply raise exceptions.

Method Types

Initializing objects	
	– initializeObject:withGlobalID:editingContext:
Getting objects	
	– objectsWithFetchSpecification:editingContext:
	– objectsForSourceGlobalID:relationshipName:editingContext:
Getting faults	
	– faultForGlobalID:editingContext:
	– arrayFaultWithSourceGlobalID:relationshipName:editingContext:
	– refaultObject:withGlobalID:editingContext:
	– faultForRawRow:entityNamed:editingContext:

Locking objects	
	– lockObjectWithGlobalID:editingContext:
	– isObjectLockedWithGlobalID:editingContext:
Saving changes to objects	
	– saveChangesInEditingContext:
Invalidating objects	
	– invalidateAllObjects
	– invalidateObjectsWithGlobalIDs:

Instance Methods

arrayFaultWithSourceGlobalID:relationshipName:editingContext:

– (NSArray *)**arrayFaultWithSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*relationshipName*
editingContext:(EOEditingContext *)*anEditingContext*

Implemented by subclasses to return the destination objects for a to-many relationship, whether as real instances or as faults (EOFault objects). *globalID* identifies the source object for the relationship (which doesn't necessarily exist in memory yet), and *relationshipName* is the name of the relationship. The object identified by *globalID* and the destination objects for the relationship all belong to *anEditingContext*.

If you implement this method to return a fault, you must define an EOFaultHandler subclass that stores *globalID* and *relationshipName*, using them to fetch the objects in a later **objectsForSourceGlobalID: relationshipName:editingContext:** message and that turns the fault into an array containing those objects. See the EOFaultHandler and EOFault class specifications for more information on faults.

See the EOEditingContext and EODatabaseContext (EOAccess) class specifications for more information on how this method works in concrete subclasses.

See also: – **faultForGlobalID:editingContext:**

faultForGlobalID:editingContext:

– (id)**faultForGlobalID:**(EOGlobalID *)*globalID*
editingContext:(EOEditingContext *)*anEditingContext*

If the receiver is *anEditingContext* and the object associated with *globalID* is already registered in *anEditingContext*, this method returns that object. Otherwise it creates a to-one fault, registers it in *anEditingContext*, and returns the fault. This method is always directed first at *anEditingContext*, which forwards the message to its parent object store if needed to create a fault.

If you implement this method to return a fault (an EOFault object), you must define an EOFaultHandler subclass that stores *globalID*, uses it to fetch the object and turn the EOFault into that object, and initializes the object with EOObjectStore's **initializeObject:withGlobalID:editingContext:**. See the EOFaultHandler and EOFault class specifications for more information on faults.

See the EOEditingContext and EODatabaseContext (EOAccess) class specifications for more information on how this method works in concrete subclasses.

See also: – **arrayFaultWithSourceGlobalID:relationshipName:editingContext:**, – **recordObject: globalID:** (EOEditingContext)

faultForRawRow:entityNamed:editingContext:

– (id)**faultForRawRow:**(id)*row*
entityNamed:(NSString *)*entityName*
editingContext:(EOEditingContext *)*anEditingContext*

Returns a fault for the enterprise object corresponding to *row*, which is a dictionary of values containing at least the primary key of the corresponding enterprise object. This is especially useful if you have fetched raw rows and now want a unique enterprise object.

initializeObject:withGlobalID:editingContext:

– (void)**initializeObject:**(id)*anObject*
withGlobalID:(EOGlobalID *)*globalID*
editingContext:(EOEditingContext *)*anEditingContext*

Implemented by subclasses to set *anObject*'s properties, as obtained for *globalID*. This method is typically invoked after *anObject* has been created using EOClassDescription's **createInstanceWithEditingContext:globalID:zone:** or using NSObject's **initWithEditingContext: classDescription:globalID:**. This method is also invoked after a fault has been fired.

See also: – **initWithEditingContext:classDescription:globalID:** (NSObject Additions),
– **awakeFromInsertionInEditingContext:** (NSObject Additions),
– **awakeFromFetchInEditingContext:** (NSObject Additions)

invalidateAllObjects

– (void)**invalidateAllObjects**

Discards the values of all objects held by the receiver and turns them into faults (EOFault objects). This causes all locks to be dropped and any transaction to be rolled back. The next time any object is accessed, its data is fetched anew. Any child object stores are also notified that the objects are no longer valid. See the EOEditingContext class specification for more information on how this method works in concrete subclasses.

This method should also post an EOInvalidatedAllObjectsInStoreNotification.

See also: – **invalidateObjectsWithGlobalIDs:**, – **refaultObject:withGlobalID:editingContext:**

invalidateObjectsWithGlobalIDs:

– (void)**invalidateObjectsWithGlobalIDs:**(NSArray *)*globalIDs*

Signals that the objects identified by the EOGlobalIDs in *globalIDs* should no longer be considered valid and that they should be turned into faults (EOFault objects). This causes data for each object to be refetched the next time it's accessed. Any child object stores are also notified that the objects are no longer valid.

See also: – **invalidateAllObjects**, – **refaultObject:withGlobalID:editingContext:**

isObjectLockedWithGlobalID:editingContext:

– (BOOL)**isObjectLockedWithGlobalID:**(EOGlobalID *)*globalID*
    editingContext:(EOEditingContext *)*anEditingContext*

Returns YES if the object identified by *globalID* is locked, NO if it isn't. See the EODatabaseContext (EOAccess) class specification for more information on how this method works in concrete subclasses.

lockObjectWithGlobalID:editingContext:

– (void)**lockObjectWithGlobalID:**(EOGlobalID *)*globalID*
    editingContext:(EOEditingContext *)*anEditingContext*

Locks the object identified by *globalID*. See the EODatabaseContext (EOAccess) class specification for more information on how this method works in concrete subclasses.

objectsForSourceGlobalID:relationshipName:editingContext:

– (NSArray *)**objectsForSourceGlobalID:**(EOGlobalID *)*globalID*
    relationshipName:(NSString *)*relationshipName*
    editingContext:(EOEditingContext *)*anEditingContext*

Returns the destination objects for a to-many relationship. This method is used by an array fault previously constructed using **arrayFaultWithSourceGlobalID:relationshipName:editingContext:**. *globalID* identifies the source object for the relationship (which doesn't necessarily exist in memory yet), and *relationshipName* is the name of the relationship. The object identified by *globalID* and the destination objects for the relationship all belong to *anEditingContext*.

See the EOEditingContext and EODatabaseContext (EOAccess) class specifications for more information on how this method works in concrete subclasses.

objectsWithFetchSpecification:editingContext:

– (NSArray *)**objectsWithFetchSpecification:**(EOFetchSpecification *)*aFetchSpecification*
    editingContext:(EOEditingContext *)*anEditingContext*

Fetches objects from an external store according to the criteria specified by *fetchSpecification* and returns them in an array for inclusion in *anEditingContext*. If one of these objects is already present in memory, this method doesn't overwrite its values with the new values from the database. Raises an exception if an error occurs.

See the EOEditingContext and EODatabaseContext (EOAccess) class specifications for more information on how this method works in concrete subclasses.

refaultObject:withGlobalID:editingContext:

– (void)**refaultObject:**(id)*anObject*
    withGlobalID:(EOGlobalID *)*globalID*
    editingContext:(EOEditingContext *)*anEditingContext*

Turns *anObject* into a fault (an EOFault), identified by *globalID* in *anEditingContext*. Objects that have been inserted but not saved, or that have been deleted, shouldn't be refaulted. When using the Yellow Box, use this method with caution since refaulting an object doesn't remove the object snapshot from the undo stack.

saveChangesInEditingContext:

– (void)**saveChangesInEditingContext:**(EOEditingContext *)*anEditingContext*

Saves any changes in *anEditingContext* to the receiver's repository. Sends **insertedObjects**, **deletedObjects**, and **updatedObjects** messages to *anEditingContext* and applies the changes to the receiver's data repository as appropriate. For example, EODatabaseContext (EOAccess) implements this method to send operations to an EOAdaptor (EOAccess) for making the changes in a database.

Notifications

EOInvalidatedAllObjectsInStoreNotification

Posted whenever an EOObjectStore receives an **invalidateAllObjects** message. The notification contains:

Notification Object	The EOObjectStore that received the **invalidateAllObjects** message.
Userinfo	None

EOObjectsChangedInStoreNotification

Posted whenever an EOObjectStore observes changes to its objects. The notification contains:

Notification Object The EOObjectStore that observed the change.

Userinfo

Key	Value
updated	An NSArray of EOGlobalIDs for objects whose properties have changed. A receiving EOEditingContext typically responds by refaulting its corresponding objects.
inserted	An NSArray of EOGlobalIDs for objects that have been inserted into the EOObjectStore.
deleted	An NSArray of EOGlobalIDs for objects that have been deleted from the EOObjectStore.
invalidated	An NSArray of EOGlobalIDs for objects that have been turned into faults.

EOObjectStoreCoordinator

Inherits From:	EOObjectStore : NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOObjectStoreCoordinator.h

Class Description

EOObjectStoreCoordinator is a part of the control layer's object storage abstraction. An EOObjectStoreCoordinator object acts as a single object store by directing one or more EOCooperatingObjectStores in managing objects from distinct data repositories. For more general information on the object storage abstraction, see "Object Storage Abstraction" in the introduction to the EOControl Framework.

EOObjectStore Methods

EOObjectStoreCoordinator overrides the following EOObjectStore methods:

- – objectsWithFetchSpecification:editingContext:
- – objectsForSourceGlobalID:relationshipName:editingContext:
- – faultForGlobalID:editingContext:
- – arrayFaultWithSourceGlobalID:relationshipName:editingContext:
- – refaultObject:withGlobalID:editingContext:
- – saveChangesInEditingContext:
- – invalidateAllObjects
- – invalidateObjectsWithGlobalIDs:

With the exception of **saveChangesInEditingContext:**, EOObjectStoreCoordinator's implementation of these methods simply forwards the message to an EOCooperatingObjectStore or stores. The message **invalidateAllObjects** is forwarded to all of a coordinator's cooperating stores. The rest of the messages are forwarded to the appropriate store based on which store responds YES to the messages **ownsGlobalID:**, **ownsObject:**, and **handlesFetchSpecification:** (which message is used depends on the context). The EOObjectStore methods listed above aren't documented in this class specification (except for **saveChangesInEditingContext:**)—for descriptions of them, see the EOObjectStore and EODatabaseContext (EOAccess) class specifications

For the method **saveChangesInEditingContext:**, the coordinator guides its cooperating stores through a multi-pass save protocol in which each cooperating store saves its own changes and forwards remaining changes to the other of the coordinator's stores. For example, if in its **recordChangesInEditingContext** method one cooperating store notices the removal of an object from an "owning" relationship but that object belongs to another cooperating store, it informs the other store by sending the coordinator a **forwardUpdateForObject:changes:** message. For a more details, see the method description for **saveChangesInEditingContext:**.

Although it manages objects from multiple repositories, EOObjectStoreCoordinator doesn't absolutely guarantee consistent updates when saving changes across object stores. If your application requires guaranteed distributed transactions, you can either provide your own solution by creating a subclass of EOObjectStoreCoordinator that integrates with a TP monitor, use a database server with built-in distributed transaction support, or design your application to write to only one object store per save operation (though it may read from multiple object stores). For more discussion of this subject, see the method description for **saveChangesInEditingContext:**.

Method Types

Initializing instances	
	– init
Setting the default coordinator	
	+ setDefaultCoordinator:
	+ defaultCoordinator
Managing EOCooperatingObjectStores	
	– addCooperatingObjectStore:
	– removeCooperatingObjectStore:
	– cooperatingObjectStores
Saving changes	
	– saveChangesInEditingContext:
Communication between EOCooperatingObjectStores	
	– forwardUpdateForObject:changes:
	– valuesForKeys:object:
Returning EOCooperatingObjectStores	
	– objectStoreForGlobalID:
	– objectStoreForFetchSpecification:
	– objectStoreForObject:
Getting the userInfo dictionary	
	– userInfo
	– setUserInfo:

Class Methods

defaultCoordinator

+ (id)**defaultCoordinator**

Returns a shared instance of EOObjectStoreCoordinator.

setDefaultCoordinator:

+ (void)**setDefaultCoordinator:**(EOObjectStoreCoordinator *)*coordinator*

Sets a shared instance EOObjectStoreCoordinator.

Instance Methods

addCooperatingObjectStore:

– (void)**addCooperatingObjectStore:**(EOCooperatingObjectStore *)*store*

Adds *store* to the list of EOCooperatingObjectStores that need to be queried and notified about changes to enterprise objects. Posts the notification EOCooperatingObjectStoreWasAdded.

See also: – **removeCooperatingObjectStore:**, – **cooperatingObjectStores**

cooperatingObjectStores

– (NSArray *)**cooperatingObjectStores**

Returns the receiver's EOCooperatingObjectStores.

See also: – **addCooperatingObjectStore:**, – **removeCooperatingObjectStore:**

forwardUpdateForObject:changes:

– (void)**forwardUpdateForObject:**(id)*object*
  changes:(NSDictionary *)*changes*

Tells the receiver to forward a message from an EOCooperatingObjectStore to another store, informing it that *changes* need to be made to *object*. For example, inserting an object in a relationship property of one EOCooperatingObjectStore might require changing a foreign key property in an object owned by another EOCooperatingObjectStore.

This method first locates the EOCooperatingObjectStore that's responsible for applying *changes*, and then it sends the store the message **recordUpdateForObject:changes:**.

init

– **init**

Initializes a newly allocated EOObjectStoreCoordinator and returns **self**. This is the designated initializer for the EOObjectStoreCoordinator class.

objectStoreForFetchSpecification:

– (EOCooperatingObjectStore *)**objectStoreForFetchSpecification:**
(EOFetchSpecification *)*fetchSpecification*

Returns the EOCooperatingObjectStore responsible for fetching objects with *fetchSpecification*. Returns **nil** if no EOCooperatingObjectStore can be found that responds YES to **handlesFetchSpecification:**.

See also: – **objectStoreForGlobalID:**, – **objectStoreForObject:**

objectStoreForGlobalID:

– (EOCooperatingObjectStore *)**objectStoreForGlobalID:**(EOGlobalID *)*globalID*

Returns the EOCooperatingObjectStore for the object identified by *globalID*. Returns **nil** if no EOCooperatingObjectStore can be found that responds YES to **ownsGlobalID:**.

See also: – **objectStoreForFetchSpecification:**, – **objectStoreForObject:**

objectStoreForObject:

– (EOCooperatingObjectStore *)**objectStoreForObject:**(id)*object*

Returns the EOCooperatingObjectStore that owns *object*. Returns **nil** if no EOCooperatingObjectStore can be found that responds YES to **ownsObject:**.

See also: – **objectStoreForFetchSpecification:**, – **objectStoreForGlobalID:**

removeCooperatingObjectStore:

– (void)**removeCooperatingObjectStore:**(EOCooperatingObjectStore *)*store*

Removes *store* from the list of EOCooperatingObjectStores that need to be queried and notified about changes to enterprise objects. Posts the notification EOCooperatingObjectStoreWasRemoved.

See also: – **addCooperatingObjectStore:**, – **cooperatingObjectStores**

saveChangesInEditingContext:

– (void)**saveChangesInEditingContext:**(EOEditingContext *)*anEditingContext*

Overrides the EOObjectStore implementation to save the changes made in *anEditingContext*. This message is sent by an EOEditingContext to an EOObjectStoreCoordinator to commit changes. When an EOObjectStoreCoordinator receives this message, it guides its EOCooperatingObjectStores through a multi-pass save protocol in which each EOCooperatingObjectStore saves its own changes and forwards remaining changes to other EOCooperatingObjectStores. When this method is invoked, the following sequence of events occurs:

1. The receiver sends each of its EOCooperatingObjectStores the message **prepareForSaveWithCoordinator: editingContext:**, which informs them that a multi-pass save operation is beginning. When the EOCooperatingObjectStore is an EODatabaseContext (EOAccess), it takes this opportunity to generate primary keys for any new objects in the EOEditingContext.
2. The receiver sends each of its EOCooperatingObjectStores the message **recordChangesInEditingContext**, which prompts them to examine the changed objects in the editing context, record operations that need to be performed, and notify the receiver of any changes that need to be forwarded to other stores. For example, if in its **recordChangesInEditingContext** method one EOCooperatingObjectStore notices the removal of an object from an "owning" relationship but that object belongs to another EOCooperatingObjectStore, it informs the other store by sending the coordinator a **forwardUpdateForObject:changes:** message.
3. The receiver sends each of its EOCooperatingObjectStores the message **performChanges**. This tells the stores to transmit their changes to their underlying databases. When the EOCooperatingObjectStore is an EODatabaseContext, it responds to this message by taking the EODatabaseOperations (EOAccess) that were constructed in the previous step, constructing EOAdaptorOperations (EOAccess) from them, and giving the EOAdaptorOperations to an available EOAdaptorChannel(EOAccess) for execution.
4. If **performChanges** fails for any of the EOCooperatingObjectStores, all stores are sent the message **rollbackChanges**.
5. If **performChanges** succeeds for all EOCooperatingObjectStores, the receiver sends them the message **commitChanges**, which has the effect of telling the adaptor to commit the changes.
6. If **commitChanges** fails for a particular EOCooperatingObjectStore, that store and all subsequent ones are sent the message **rollbackChanges**. However, the stores that have already committed their changes do not roll back. In other words, the coordinator doesn't perform the two-phase commit protocol necessary to guarantee consistent distributed update.

This method raises an exception if an error occurs.

setUserInfo:

– (void)**setUserInfo**:(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs.

See also: – **userInfo**

userInfo

– (NSDictionary *)**userInfo**

Returns a dictionary of user data. Your application can use this to store any auxiliary information it needs.

See also: – **setUserInfo:**

valuesForKeys:object:

– (NSDictionary *)**valuesForKeys:**(NSArray *)*keys*
object:(id)*object*

Communicates with the appropriate EOCooperatingObjectStore to get the values identified by *keys* for *object*, so that it can then forward them on to another EOCooperatingObjectStore. EOCooperatingObjectStores can hold values for an object that augment the properties in the object. For instance, an EODatabaseContext (EOAccess) stores foreign key information for the objects it owns. These foreign keys may well not be defined as properties of the object. Other EODatabaseContexts can find out the object's foreign keys by sending the EODatabaseContext that owns the object a **valuesForKeys:object:** message (through the coordinator).

Notifications

The following notifications are declared and posted by EOObjectStoreCoordinator.

EOCooperatingObjectStoreWasAdded

When an EOObjectStoreCoordinator receives an **addCooperatingObjectStore:** message and adds an EOCooperatingObjectStore to its list, it posts EOCooperatingObjectStoreWasAdded to notify observers.

Notification Object	The EOObjectStoreCoordinator
userInfo Dictionary	None

EOCooperatingObjectStoreWasRemoved

When an EOObjectStoreCoordinator receives a **removeCooperatingObjectStore:** message and removes an EOCooperatingObjectStore from its list, it posts EOCooperatingObjectStoreWasRemoved to notify observers.

Notification Object	The EOObjectStoreCoordinator

userInfo Dictionary	None

EOCooperatingObjectStoreNeeded

Posted when an EOObjectStoreCoordinator receives a request that it can't service with any of its currently registered EOCooperatingObjectStores. The observer can call back to the coordinator to register an appropriate EOCooperatingObjectStore based on the information in the userInfo dictionary.

Notification Object	The EOObjectStoreCoordinator
userInfo Dictionary	One of the following key-value pairs
Key	**Value**
globalID	globalID for the operation
fetchSpecification	fetch specification for the operation
object	object for the operation

EOObserverCenter

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOObserver.h

Class Description

EOObserverCenter is the central player in EOControl's change tracking mechanism. EOObserverCenter records observers and the objects they observe, and it distributes notifications when the observable objects change. For an overview of the change tracking mechanism, see "Tracking Enterprise Objects Changes" in the introduction to the EOControl Framework.

You don't ever create instances of EOObserverCenter. Instead, the class itself acts as the central manager of change notification, registering observers and notifying them of changes. The EOObserverCenter API is provided entirely in class methods.

Registering an Observer

Objects that directly observe others must adopt the EOObserving protocol, which consists of the single method **objectWillChange:**. To register an object as an observer, invoke EOObserverCenter's **addObserver:forObject:** with the observer and the object to be observed. Once this is done, any time the observed object invokes its **willChange** method, the observer is sent an **objectWillChange:** message informing it of the pending change. You can also register an observer to be notified when any object changes using **addOmniscientObserver:**. This can be useful in certain situations, but as it's very costly to deal out frequent change notifications, you should use omniscient observers sparingly. To unregister either kind of observer, simply use the corresponding **remove...** method.

Change Notification

Objects that are about to change invoke **willChange**, a method that the Framework adds to NSObject. The implementations of this method invoke EOObserverCenter's **notifyObserversObjectWillChange:**, which sends an **objectWillChange:** message to all observers registered for the object that's changing, as well as to any omniscient observers. **notifyObserversObjectWillChange:** optimizes the process by suppressing redundant **objectWillChange:** messages when the same object invokes **willChange** several times in a row (as often happens when multiple properties are changed). Change notification is immediate, and takes place *before* the object's state changes. If you need to compare the object's state before and after the change, you must arrange to examine the new state at the end of the run loop.

You can suppress change notification when necessary, using the **suppressObserverNotification** and **enableObserverNotification** methods. While notification is suppressed, neither regular nor omniscient observers are informed of changes. These methods nest, so you can invoke **suppressObserverNotification** multiple times, and notification isn't re-enabled until a matching number of **enableObserverNotification** message have been sent.

Method Types

Registering and unregistering observers
- + addObserver:forObject:
- + removeObserver:forObject:
- + addOmniscientObserver:
- + removeOmniscientObserver:

Notifying observers of change
- + notifyObserversObjectWillChange:

Getting observers
- + observersForObject:
- + observerForObject:ofClass:

Suppressing change notification
- + suppressObserverNotification
- + enableObserverNotification
- + observerNotificationSuppressCount

Class Methods

addObserver:forObject:

+ (void)**addObserver:**(id <EOObserving>)*anObserver*
    forObject:(id)*anObject*

Records *anObserver* to be notified with an **objectWillChange:** message when *anObject* changes.

See also: + **removeObserver:forObject:**

addOmniscientObserver:

+ (void)**addOmniscientObserver:**(id <EOObserving>)*anObserver*

Records *anObserver* to be notified with an **objectWillChange:** message when any object changes. This can cause significant performance degradation, and so should be used with care. The ominiscient observer must be prepared to receive the **objectWillChange:** message with a **nil** argument.

See also: + **addObserver:forObject:**, + **removeOmniscientObserver:**

enableObserverNotification

+ (void)**enableObserverNotification**

Counters a prior **suppressObserverNotification** message. When no such messages remain in effect, the **notifyObserversObjectWillChange:** method is re-enabled. Raises an NSInternalInconsistencyException if not paired with a prior **suppressObserverNotification** message.

notifyObserversObjectWillChange:

+ (void)**notifyObserversObjectWillChange:**(id)*anObject*

Unless change notification is suppressed, sends an **objectWillChange:** to all observers registered for *anObject* with that object as the argument, and sends that message to all omniscient observers as well. If invoked several times in a row with the same object, only the first invocation has any effect, since subsequent change notifications are redundant.

If an observer wants to ensure that it receives notification the next time the last object to change changes again, it should use the statement:

```
[EOObserverCenter notifyObserversObjectWillChange:nil];
```

An observable object (typically an enterprise object) invokes this method from its **willChange** implementation, so you should never have to invoke this method directly.

See also: + **suppressObserverNotification**, + **addObserver:forObject:**, + **addOmniscientObserver:**

observerForObject:ofClass:

+ (id)**observerForObject:**(id)*anObject*
    ofClass:(Class)*aClass*

Returns an observer for *anObject* that's a kind of *aClass*. If more than one observer of *anObject* is a kind of *aClass*, the specific observer returned is undetermined. You can use **observersForObject:** instead to get all observers and examine their class membership.

observerNotificationSuppressCount

+ (unsigned int)**observerNotificationSuppressCount**

Returns the number of **suppressObserverNotification** messages in effect.

See also: + **enableObserverNotification**

observersForObject:

+ (NSArray *)**observersForObject:**(id)*anObject*

Returns all observers of *anObject*.

removeObserver:forObject:

+ (void)**removeObserver:**(id <EOObserving>)*anObserver* **forObject:**(id)*anObject*

Removes *anObserver* as an observer of *anObject*.

See also: – **addObserver:forObject:**

removeOmniscientObserver:

+ (void)**removeOmniscientObserver:**(id <EOObserving>)*anObserver*

Unregisters *anObserver* as an observer of all objects.

See also: + **removeObserver:forObject:**, + **addOmniscientObserver:**

suppressObserverNotification

+ (void)**suppressObserverNotification**

Disables the **notifyObserversObjectWillChange:** method, so that no change notifications are sent. This method can be invoked multiple times; **enableObserverNotification** must then be invoked an equal number of times to re-enable change notification.

EOObserverProxy

Inherits From:	EODelayedObserver : NSObject
Conforms To:	EOObserving (EODelayedObserver) NSObject (NSObject)
Declared In:	EOControl/EOObserver.h

Class Description

The EOObserverProxy class is a part of EOControl's change tracking mechanism. It provides a means for objects that can't inherit from EODelayedObserver to handle **subjectChanged** messages. For an overview of the general change tracking mechanism, see "Tracking Enterprise Objects Changes" in the introduction to the EOControl Framework.

An EOObserverProxy has a target object on whose behalf it observes objects. EOObserverProxy overrides **subjectChanged** to send an action message to its target object, allowing the target to act as though it had received **subjectChanged** directly from an EODelayedObserverQueue. See the EOObserverCenter and EODelayedObserverQueue class specifications for more information.

Instance Methods

initWithTarget:action:priority:

– (id)**initWithTarget:**(id)*anObject*
  action:(SEL)*anAction*
  priority:(EOObserverPriority)*priority*

Initializes a new EOObserverProxy to send *anAction* to *anObject* upon receiving a **subjectChanged** message. *anAction* should be a selector for a typical action method, taking one **id** argument and returning **void**. *priority* indicates when the receiver is sent this message from EODelayedObserverQueue's **notifyObserversUpToPriority:** method. This is the designated initializer for the EOObserverProxy class. Returns **self**.

EOOrQualifier

Inherits From:	EOQualifier : NSObject
Conforms To:	EOQualifierEvaluation EOQualifierSQLGeneration
Declared In:	EOControl/EOQualifier.h

Class Description

EOOrQualifier is a subclass of EOQualifier that contains multiple qualifiers. EOOrQualifier adopts the EOQualifierEvaluation protocol, which defines the method **evaluateWithObject:** for in-memory evaluation. When an EOOrQualifier object receives an **evaluateWithObject:** message, it evaluates each of its qualifiers until one of them returns YES. If one of its qualifiers returns YES, the EOOrQualifier object returns YES immediately. If all of its qualifiers return NO, the EOOrQualifier object returns NO.

Adopted Protocols

EOQualifierEvaluation	
	– evaluateWithObject:
EOQualifierSQLGeneration	
	– sqlStringForSQLExpression:
	– schemaBasedQualifierWithRootEntity:

Instance Methods

evaluateWithObject:

@protocol EOQualifierEvaluation
– (BOOL)**evaluateWithObject:**(id)*anObject*

Returns YES if *anObject* satisfies the qualifier, NO otherwise. When an EOOrQualifier object receives an **evaluateWithObject:** message, it evaluates each of its qualifiers until one of them returns YES. If any of its qualifiers returns YES, the EOOrQualifier object returns YES immediately. If all of its qualifiers return NO, the EOOrQualifier object returns NO. This method can raise one of several possible exceptions if an error occurs. If your application allows users to construct arbitrary qualifiers (such as through a user interface), you may want to write code to catch any exceptions and respond to errors (for example, by displaying a panel saying that the user typed a poorly formed qualifier).

initWithQualifierArray:

– **initWithQualifierArray:**(NSArray *)*qualifiers*

Initializes the receiver with the qualifiers *qualifiers* and returns **self**. This method is the designated initializer for EOOrQualifier.

initWithQualifiers:

– **initWithQualifiers:**(EOQualifier *)*qualifiers*,...

Initializes the receiver with the **nil**-terminated list of qualifiers *qualifiers*. Works by invoking **initWithQualifierArray:**. For example, the following code excerpt constructs three qualifiers, **qual1**, **qual2**, and **qual3**. It then uses these qualifiers to initialize an EOOrQualifier, **orQual**. **orQual** is then used to filter an in-memory array.

```
NSArray *guests;    /* Assume this exists. */
EOQualifier *qual1, *qual2, *qual3, *orQual;

qual1 = [EOQualifier qualifierWithQualifierFormat:@"lastName = 'Nunez'"];
qual2 = [EOQualifier qualifierWithQualifierFormat:@"lastName = 'Wren'"];
qual3 = [EOQualifier qualifierWithQualifierFormat:@"lastName = 'Wilson'"];

/* Initialize the EOOrQualifier orQual using a nil-terminated list of
 * qualifiers.
 */
orQual = [[EOOrQualifier alloc] initWithQualifiers:qual1, qual2, qual3, nil];
/* Use orQual to filter the array guests. */
return [guests filteredArrayUsingQualifier:orQual];
```

qualifiers

– (NSArray *)**qualifiers**

Returns the receiver’s qualifiers.

EOQualifier

Inherits From:	NSObject
Conforms To:	NSCopying
Declared In:	EOControl/EOQualifier.h

Class Description

EOQualifier is an abstract class for objects that hold information used to restrict selections on objects or database rows according to specified criteria. With the exception of EOSQLQualifier (EOAccess), qualifiers aren't based on SQL and they don't rely upon an EOModel (EOAccess). Thus, the same qualifier can be used both to perform in-memory searches and to fetch from the database.

You never instantiate an instance of EOQualifier. Rather, you use one of its subclasses—one of the following or your own custom EOQualifier subclass:

Subclass	Purpose
EOKeyValueQualifier	Compares the named property of an object to a supplied value, for example, "weight > 150".
EOKeyComparisonQualifier	Compares the named property of one object with the named property of another, for example "name = wife.name".
EOAndQualifier	Contains multiple qualifiers, which it conjoins. For example, "name = 'Fred' AND age < 20".
EOOrQualifier	Contains multiple qualifiers, which it disjoins. For example, "name = 'Fred' OR name = 'Ethel'".
EONotQualifier	Contains a single qualifier, which it negates. For example, "NOT (name = 'Fred')".
EOSQLQualifier	Contains unstructured text that can be transformed into a SQL expression. EOSQLQualifier provides a way to create SQL expressions with any arbitrary SQL. Because EOSQLQualifiers can't be evaluated against objects in memory and because they contain database and SQL-specific content, you should use EOQualifier wherever possible. EOSQLQualifier is also provided for backward compatibility with pre-2.0 Enterprise Objects Framework releases, which didn't offer a SQL-independent qualifier.

The protocol EOQualifierEvaluation defines how qualifiers are evaluated in memory. To evaluate qualifiers in a database, methods in EOSQLExpression (EOAccess) and EOEntity (EOAccess) are used to generate SQL for qualifiers. Note that all of the SQL generation functionality is contained in the access layer.

For more information on using EOQualifiers, see the sections

- Creating a Qualifier
- Constructing Format Strings
- Checking for NULL Values
- Using Wildcards and the like Operator
- Using Selectors in Qualifier Expressions
- Using Different Data Types in Format Strings
- Using EOQualifier’s Subclasses
- Creating Subclasses

Constants

The following selector constants are defined to represent the different qualifier operators:

EOQualifierOperatorEqual	EOQualifierOperatorLessThanOrEqualTo
EOQualifierOperatorNotEqual	EOQualifierOperatorGreaterThanOrEqualTo
EOQualifierOperatorLessThan	EOQualifierOperatorContains
EOQualifierOperatorGreaterThan	EOQualifierOperatorLike
	EOQualifierOperatorCaseInsensitiveLike

Adopted Protocols

NSCopying

Method Types

Creating a qualifier	
	+ qualifierWithQualifierFormat:
	+ qualifierWithQualifierFormat:arguments:
	+ qualifierToMatchAllValues:
	+ qualifierToMatchAnyValue:
	– qualifierWithBindings:requiresAllVariables:
Converting strings and operators	
	+ operatorSelectorForString:
	+ stringForOperatorSelector:
Get EOQualifier operators	
	+ allQualifierOperators
	+ relationalQualifierOperators
Accessing a qualifier's keys	
	– bindingKeys
	– keyPathForBindingKey:
Validating a qualifier's keys	
	– validateKeysWithRootClassDescription:

Class Methods

allQualifierOperators

+ (NSArray *)**allQualifierOperators**

Returns an NSArray containing all of the operators supported by EOQualifier: =, !=, <, <=, >, >=, "like", and "caseInsensitiveLike".

See also: + **relationalQualifierOperators**

operatorSelectorForString:

+ (SEL)**operatorSelectorForString:**(NSString *)*aString*

Returns an operator selector based on the string *aString*. This method is used in parsing a qualifier. For example, the following statement returns the selector **isNotEqualTo:**.

```
selector = [EOQualifier operatorSelectorForString:@"!="];
```

The possible values of *aString* are =, ==, !=, <, >, <=, >=, "like", and "caseInsensitiveLike".

You'd probably only use this method if you were writing your own qualifier parser.

See also: + **stringForOperatorSelector:**

qualifierToMatchAllValues:

+ (EOQualifier *)qualifierToMatchAllValues:(NSDictionary *)*values*;

Takes a dictionary of search criteria, from which the method creates EOKeyValueQualifiers (one for each dictionary entry). The method ANDs these qualifiers together, and returns the resulting EOAndQualifier.

See also:

qualifierToMatchAnyValue:

+ (EOQualifier *)qualifierToMatchAnyValue:(NSDictionary *)*values*;

Takes a dictionary of search criteria, from which the method creates EOKeyValueQualifiers (one for each dictionary entry). The method ORs these qualifiers together, and returns the resulting EOOrQualifier.

See also:

qualifierWithQualifierFormat:

+ (EOQualifier *)**qualifierWithQualifierFormat:**(NSString *)*qualifierFormat*, ...

Parses the format string *qualifierFormat*, usesit to create an EOQualifier, and returns the EOQualifier. Based on the content of *qualifierFormat*, this method generates a tree of the basic qualifier types. For example, the format string "firstName = 'Joe' AND department = 'Facilities'" generates an EOAndQualifier that contains two "sub" EOKeyValueQualifiers. The following code excerpt shows a typical way to use the **qualifierWithQualifierFormat:** method. The excerpt constructs an EOFetchSpecification, which includes an entity name and a qualifier. It then applies the EOFetchSpecification to the EODisplayGroup's data source and tells the EODisplayGroup to fetch.

```
EODisplayGroup *displayGroup;     /* Assume this exists.*/
EOFetchSpecification *fetchSpec;
EODatabaseDataSource *dataSource;

dataSource = [displayGroup dataSource];
fetchSpec = [EOFetchSpecification
    fetchSpecificationWithEntityName:@"Member"
    qualifier:[EOQualifier qualifierWithQualifierFormat:
    @"cardType = 'Visa' "]
    sortOrderings:nil];
[dataSource setFetchSpecification:fetchSpec];
[displayGroup fetch];
```

qualifierWithQualifierFormat performs no verification to ensure that keys referred to by the format string *qualifierFormat* exist. It raises an NSInvalidArgumentException if *qualifierFormat* contains any syntax errors.

qualifierWithQualifierFormat:arguments:

+ (EOQualifier *)**qualifierWithQualifierFormat:**(NSString *)*qualifierFormat*
arguments:(NSArray *)*arguments*

Parses the format string *qualifierFormat* and the specified *arguments*, uses them to create an EOQualifier, and returns the EOQualifier. This method is equivalent to **qualifierWithQualifierFormat:** except that format characters (for example, %@, %d, %f) in *qualifierFormat* cause the method to search in the arguments array for values rather than in a variable argument list. Note that although %d and %f can be used when constructing qualifiers, they don’t work with most other string formatting methods such as NSString’s stringWithFormat:.

relationalQualifierOperators

+ (NSArray *)**relationalQualifierOperators**

Returns an NSArray containing all of the relational operators supported by EOQualifier: =, !=, <, <=, >, and >=. In other words, returns all of the EOQualifier operators except for the ones that work exclusively on strings: “like” and “caseInsensitiveLike”.

See also: + **allQualifierOperators**

stringForOperatorSelector:

+ (NSString *)**stringForOperatorSelector:**(SEL)*aSelector*

Returns an NSString representation of the selector *aSelector.* For example, the following statement returns the string “!=”:

```
operator = [EOQualifier stringForOperatorSelector:EOQualifierOperatorNotEqual];
```

The possible values for *selector* are as follows:

- EOQualifierOperatorEqual
- EOQualifierOperatorNotEqual
- EOQualifierOperatorLessThan
- EOQualifierOperatorGreaterThan
- EOQualifierOperatorLessThanOrEqualTo
- EOQualifierOperatorGreaterThanOrEqualTo
- EOQualifierOperatorContains
- EOQualifierOperatorLike

- EOQualifierOperatorCaseInsensitiveLike

You'd probably only use this method if you were writing your own parser.

See also: + **operatorSelectorForString:**

Instance Methods

bindingKeys

– (NSArray *)**bindingKeys**

Returns an array of strings which are the names of the known variables. Multiple occurrences of the same variable will only appear once in this list.

keyPathForBindingKey:

– (NSString *)**keyPathForBindingKey:**(NSString *)*key*

Returns a string which is the "left-hand-side" of the variable in the qualifier. e.g. If you have a qualifier "salary > $amount and manager.lastName = $manager", then calling bindingKeys would return the array ("amount", "manager"). Calling **keyPathForBindingKey** would return salary for amount, and manager.lastname for manager.

qualifierWithBindings:requiresAllVariables:

– (EOQualifier *)qualifierWithBindings:(NSDictionary *)*bindings* requiresAllVariables:
(BOOL)*requiresAll*;

Returns a new qualifier substituting all variables with values found in *bindings*. If *requiresAll* is YES, any variable not found in *bindings* will cause an EOQualifierVariableSubstitutionException to be raised. If *requiresAll* is NO, missing variable values will cause the qualifier node to be pruned from the tree.

validateKeysWithRootClassDescription:

– (NSException *)**validateKeysWithRootClassDescription:**(EOClassDescription *)*classDesc*

Validates that the receiver contains keys and key paths that belong to or originate from *classDesc*. This method returns an NSInternalInconsistencyException if an unknown key is found, otherwise it returns nil to indicate that the keys contained by the qualifier are valid.

EOQualifier

Creating a Qualifier

As described above, there are several EOQualifier subclasses, each of which represents a different semantic. However, in most cases you simply create a qualifier using the EOQualifier class method **qualifierWithQualifierFormat:**, as follows:

```
EOQualifier *qual = [EOQualifier qualifierWithQualifierFormat:@"lastName =
'Smith'"];
```

The qualifier or group of qualifiers that result from such a statement is based on the contents of the format string you provide. For example, giving the format string “lastName = ’Smith’" as an argument to **qualifierWithQualifierFormat:** returns an EOKeyValueQualifier object. But you don’t normally need to be concerned with this level of detail.

The format strings you use to create a qualifier can be compound logical expressions, such as “firstName = ’Fred’ AND age < 20”. When you create a qualifier, compound logical expressions are translated into a tree of EOQualifier nodes. Logical operators such as AND and OR become EOAndQualifiers and EOOrQualifiers, respectively. These qualifiers conjoin (AND) or disjoin (OR) a group of sub-qualifiers. This is illustrated in Figure 4, in which the format string “salary > 300 AND firstName = ’Angela’ AND manager.name = ’Fred’" has been translated into a tree of qualifiers.

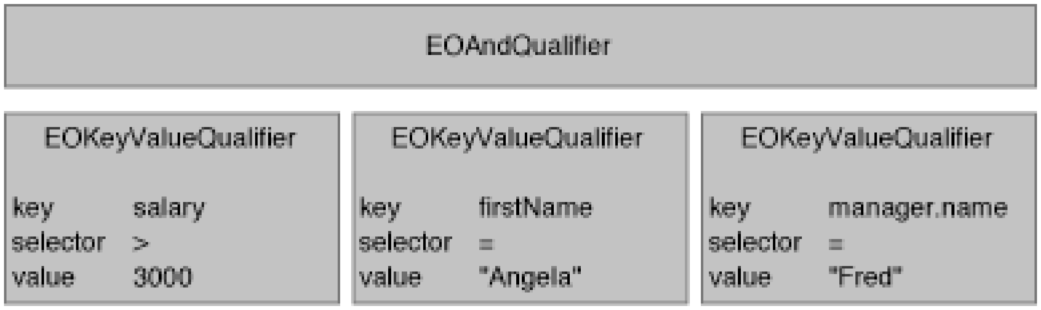


Figure 4 EOQualifier Tree for **salary > 300 AND firstName = “Angela” AND manager.name = “Fred”**

Note: The **qualifierWithQualifierFormat:** method can’t be used to create an instance of EOSQLQualifier. This is because EOSQLQualifier uses a non-structured syntax to provide backward compatibility with pre-2.0 Enterprise Objects Framework releases. It also requires an entity. To create an instance of EOSQLQualifier, you’d use a statement such as the following:

```
myQual = [[EOSQLQualifier alloc] initWithEntity:myEntity format:myFormatString];
```

Constructing Format Strings

As described above, you typically create a qualifier from a format string by using **qualifierWithQualifierFormat:**. This method takes as an argument a format string somewhat like that used with the standard C **printf()** function. The format string can embed strings, numbers, and objects using the conversion specifications listed below. This allows qualifiers to be built dynamically. The following table lists the conversion specifications you can use in a format string and their corresponding data types.

Conversion Specification	Expected Value or Result
%s	A constant C string (**const char ***).
%d	An **int**.
%f	A **float** or **double**.
%@	An **id** argument. The behavior of this conversion specification depends on its position. It can either be an object whose description method returns a key (in other words, an NSString), or a value such as an NSString, NSNumber, NSCalendarDate, and so on.
%%	Results in a literal **%** character.

Note: If you use an unrecognized character in a conversion specification (for example, %x), an NSInvalidArgumentException is raised.

For example, suppose you have an Employee entity with the properties **empID**, **firstName**, **lastName**, **salary**, and **department** (representing a to-one relationship to the employee's department), and a Department entity with properties deptID, and name. You could construct simple qualifier strings like the following:

lastName = 'Smith'
salary > 2500
department.name = 'Personnel'

The following examples build qualifiers similar to the qualifier strings described above, but take the specific values from already-fetched enterprise objects:

```
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ = %@",
    @"lastName", [anEmployee lastName]];
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ > %f",
    @"salary", [anEmployee salary]];
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ = %@",
    @"department.name", [aDept name]];
```

The enterprise objects here implement methods for directly accessing the given attributes: **lastName** and **salary** for Employee objects, and **name** for Department objects.

Note: Unlike a string literal, the %@ conversion specification is never surrounded by single quotes:

```
// For a literal string value such as Smith, you use single quotes.
[EOQualifier qualifierWithQualifierFormat:@"lastName = 'Smith'", null)];

// For the conversion specification %@, you don’t use quotes
[EOQualifier qualifierWithQualifierFormat:@"lastName = %@", @"Jones"];
```

Typically format strings include only two data types: strings and numbers. Single-quoted or double-quoted strings are NSStrings, non-quoted numbers are NSNumbers, and non-quoted strings are keys. You can get around this limitation by performing explicit casting, as described in the section “Using Different Data Types in Format Strings”.

The operators you can use in constructing qualifiers are =, ==, !=, <, >, <=, >=, “like”, and “caseInsensitiveLike”. The **like** and **caseInsensitiveLike** operators can be used with wildcards to perform pattern matching, as described in “Using Wildcards and the like Operator,” below.

Checking for NULL Values

To construct a qualifier that fetches rows matching null values, use either of the approaches shown in the following example:

```
[EOQualifier qualifierWithQualifierFormat:@"bonus = nil"];
[EOQualifier qualifierWithQualifierFormat:@"bonus = %@", [EONull null]];
[EOQualifier qualifierWithQualifierFormat:@"bonus = %@", nil];
```

Using Wildcards and the like Operator

When you use the **like** or **caseInsensitiveLike** operator in a qualifier expression, you can use the wildcard characters * and ? to perform pattern matching, for example:

```
@"lastName like ’Jo*’"
```

matches Jones, Johnson, Jolsen, Josephs, and so on.

The ? character just matches a single character, for example:

```
@"lastName like ’Jone?’"
```

matches Jones.

The asterisk character (*) is only interpreted as a wildcard in expressions that use the **like** or **caseInsensitiveLike** operator. For example, in the following statement, the character * is treated as a literal value, not as a wildcard:

```
@"lastName = ’Jo*’"//  The * character doesn’t act as a wildcard in this statement.
```

Using Selectors in Qualifier Expressions

The format strings you use to initialize a qualifier can include selectors. The parser recognizes a selector as an unquoted string followed by a colon, such as **myMethod:**. For example:

```
point1 isInside: area
firstName isAnagramOfString: "Computer"
```

Selectors in a qualifier are parsed and applied only in memory; that is, they can’t be used in SQL generation.

Using Different Data Types in Format Strings

As stated in the section “Constructing Format Strings”, format strings normally include only two data types: strings and numbers. To get around this limitation, you can perform explicit casting.

For example, NSCalendarDate and NSDecimalNumber are two classes that are likely to be used in qualifiers. You can construct format strings for objects of these classes as follows:

```
hireDate = (NSCalendarDate)’1990-03-16 00:00:00 +0000’
salary = (NSDecimalNumber)’15000.02’
```

When you use this approach, qualifiers are constructed by looking up the class and invoking `[[class alloc] initWithString:stringValue]`. Therefore, this technique only works for classes that implement **initWithString:**.

Note that to construct a date qualifier using a format string, you must use the default CalendarDate format, which is %Y-%m-%d %H:%M:%S %z—for example:

```
EOQualifier *qual = [EOQualifier qualifierWithQualifierFormat:
    @"dateReleased < (NSCalendarDate)'1990-01-26 00:00:00 +0000'"];
```

This limitation doesn’t apply when you’re working with NSCalendarDate objects—you can just construct a qualifier in the usual way:

```
 EOQualifier *qual = [EOQualifier qualifierWithQualifierFormat:
    @"dateReleased > %@", [NSCalendarDate calendarDate]];
```

Using EOQualifier’s Subclasses

You rarely need to explicitly create an instance of EOAndQualifier, EOOrQualifier, or EONotQualifier. However, you may want to create instances of EOKeyValueQualifier and EOKeyComparisionQualifier. The primary advantage of this is that it lets you exercise more control over how the qualifier is constructed, which is desirable in some cases.

If you want to explicitly create a qualifier subclass, you can do it using code such as the following excerpt, which uses EOKeyValueQualifier to select all objects whose “isOut” key is equal to YES. In the excerpt, the qualifier is used to filter an in-memory array.

```
// Create the qualifier
EOQualifier *qual = [[EOKeyValueQualifier alloc] initWithKey:@"isOut"
        operatorSelector:EOQualifierOperatorEqual
        value:[NSNumber numberWithBool:YES]];

// Filter an array and return it
return [[self allRentals] filteredArrayUsingQualifier:qual];
```

filteredArrayUsingQualifier: is a method that Enterprise Objects Framework adds to NSArray. It's used for filtering in-memory arrays.

Creating Subclasses

EOQualifier offers extensibility across two dimensions: new classes can be added to extend qualifier semantics, and categories can be added to extend functionality (for example, to provide in-memory evaluation).

Subclasses used to evaluate objects in memory must implement the EOQualifierEvaluation protocol. Subclasses used to generate SQL queries must conform to the EOQualifierSQLGeneration protocol.

EOSortOrdering

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	EOControl/EOSortOrdering.h

Class Description

An EOSortOrdering object specifies the way that a group of objects should be sorted, using a property key and a method selector for comparing values of that property. EOSortOrderings are used both to generate SQL when fetching rows from a database server, and to sort objects in memory. Both the EOFetchSpecification class and the added NSArray sorting methods accept an array of EOSortOrderings, which are applied in series to perform sorts by more than one property.

Sorting with SQL

When an EOSortOrdering is used to fetch data from a relational database, it's rendered into an ORDER BY clause for a SQL SELECT statement according to the concrete adaptor you're using. For more information, see the class description for EOSQLExpression. The Framework predefines symbols for four comparison selectors, listed in the table below. The table also shows an example of how the comparison selectors can be mapped to SQL.

Defined Name	SQL Expression
EOCompareAscending	(*key*) asc
EOCompareDescending	(*key*) desc
EOCompareCaseInsensitiveAscending	upper(*key*) asc
EOCompareCaseInsensitiveDescending	upper(*key*) desc

Using the mapping in the table above, the array of EOSortOrderings (**nameOrdering**) created in the following code example:

```
NSArray *nameOrdering = [NSArray arrayWithObjects:
    [EOSortOrdering sortOrderingWithKey:@"lastName" selector:EOCompareAscending],
    [EOSortOrdering sortOrderingWithKey:@"firstName" selector:EOCompareAscending],
    nil];
```

results in this ORDER BY clause:

```
order by (lastName) asc, (firstName) asc
```

In-Memory Sorting

Enterprise Objects Framework adds a method each to NSArray and NSMutableArray for sorting objects in memory. NSArray’s **sortedArrayUsingKeyOrderArray:** returns a new array of objects sorted according to the specified EOSortOrderings. Similarly, NSMutableArray’s **sortUsingKeyOrderArray:** sorts the array of objects. This code fragment, for example, sorts an array of Employee objects by last name, then first name using the array of EOSortOrderings created above:

```
NSArray *sortedEmployees = [employees sortedArrayUsingKeyOrderArray:nameOrdering];
```

Comparison Methods

The predefined comparison selectors are:

Defined Name	
EOCompareAscending	– compareAscending:
EOCompareDescending	– compareDescending:
EOCompareCaseInsensitiveAscending	– compareCaseInsensitiveAscending:
EOCompareCaseInsensitiveDescending	– compareCaseInsensitiveDescending:

The first two can be used with any value class; the second two with NSString objects only. The sorting methods extract property values using key-value coding and apply the selectors to the values. If you use custom value classes, you should be sure to implement the appropriate comparison methods to avoid exceptions when sorting objects.

Adopted Protocols

NSCoding	
	– encodeWithCoder:
	– initWithCoder:

Method Types

Creating instances

+ sortOrderingWithKey:selector:
– initWithKey:selector:

Examining a sort ordering

– key
– selector

Class Methods

sortOrderingWithKey:selector:

+ (EOSortOrdering *)**sortOrderingWithKey:**(NSString *)*key* **selector:**(SEL)*selector*

Creates and returns an EOSortOrdering based on *key* and *selector*.

See also: – **initWithKey:selector:**

Instance Methods

initWithKey:selector:

– (id)**initWithKey:**(NSString *)*key* **selector:**(SEL)*aSelector*

Initializes a newly allocated EOSortOrdering based on *key* and *selector* and returns **self**. This is the designated initializer for the EOSortOrdering class.

See also: + **sortOrderingWithKey:selector:**

key

– (NSString *)**key**

Returns the key by which the receiver orders items.

See also: – **selector**

selector

– (SEL)**selector**

Returns the method selector used to compare values when sorting.

See also: – **key**

EOTemporaryGlobalID

Inherits From:	EOGlobalID : NSObject
Conforms To:	NSCoding NSCopying (EOGlobalID) NSObject (NSObject)
Declared In:	EOControl/EOGlobalID.h

Class Description

An EOTemporaryGlobalID object identifies a newly created enterprise object before it's saved to an external store. When the object is saved, the temporary ID is converted to a permanent one, as described in the EOGlobalID class specification.

Adopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:

Class Methods

assignGloballyUniqueBytes:

+ (void)**assignGloballyUniqueBytes:**(unsigned char *)*buffer*

Assigns a network-wide unique ID of the format:

```
< Sequence [2], ProcessID [2] , Time [4], IP Addr [4] >
```

buffer should have space for EOUniqueBinaryKeyLength (12) bytes.

Instance Methods

init

– (id)**init**

Initializes a newly allocated EOTemporaryGlobalID as a unique instance. The new temporary global ID contains a byte string obtained from **assignGloballyUniqueBytes:** that's guaranteed to be unique network-wide. As a result, EOTemporaryGlobalIDs can be safely passed between processes and machines while still preserving global uniqueness.

buffer should have space for EOUniqueBinaryKeyLength (12) bytes.

isTemporary

– (BOOL)**isTemporary**

Returns YES.

EOUndoManager

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOUndoManager.h

Class Description

EOUndoManager is a general purpose recorder of operations for undo and redo. You register an undo operation by specifying the object that's changing (or the owner of that object), along with a method to invoke to revert its state, and the arguments for that method. EOUndoManager automatically groups all operations within a single cycle of the run loop, so that performing an undo reverts all changes that occurred during the loop. Also, when performing undo an EOUndoManager saves the operations reverted so that you can redo the undos.

Operations and Groups

An *undo operation* is a method for reverting a change to an object, along with the arguments needed to revert the change (for example, its state before the change). Undo operations are typically collected into *undo groups*, which represent whole undoable actions. *Redo operations* and groups are simply undo operations stored on a separate stack (described below). When an EOUndoManager performs undo or redo, it's actually undoing or redoing an entire group of operations. To undo a single operation, it must be packaged in a group.

EOUndoManager normally creates undo groups automatically during the run loop. The first time it's asked to record an undo operation in the run loop, it creates a new group. Then, at the end of the loop, it closes the group. You can create additional, nested undo groups within these default groups using the **beginUndoGrouping** and **endUndoGrouping** methods. You can also turn off the default grouping behavior using **setGroupsByEvent:**.

The Undo and Redo Stacks

Undo groups are stored on a stack, with the oldest groups at the bottom and the newest at the top. The undo stack is unlimited by default, but you can restrict it to a maximum number of groups using the **setLevelsOfUndo:** method. When the stack exceeds the maximum, the oldest undo groups are dropped from the bottom.

Initially, both stacks are empty. Recording undo operations adds to the undo stack, but the redo stack remains empty until undo is performed. Performing undo causes the undo operations in the latest group to be applied to their objects. Since these operations cause changes to the objects' states, the objects presumably register new operations with the EOUndoManager, this time in the reverse direction from the original operations. Since the EOUndoManager is in the process of performing undo, it records these operations as redo operations on the redo stack. Consecutive undos add to the redo stack. Subsequent redo operations pull the operations off the redo stack and apply them to the objects.

The redo stack's contents last as long as undo and redo is performed. However, because applying a new change to objects invalidates the previous changes, as soon as a new undo operation is registered, the redo stack is cleared. This prevents redo from returning objects to an inappropriate prior state. You can check for the ability to undo and redo with the **canUndo** and **canRedo** methods.

Registering Undo Operations

EOUndoManager supports two types of undo operations: one based on a simple selector with a single object argument, and one based on a general NSInvocation (which allows any number and type of arguments). The first method is commonly used by EOEditingContext for changes to enterprise objects. When an object changes, the EOEditingContext records a simple undo operation with an NSDictionary containing the old property values of the object. Performing undo then applies this object snapshot via the key-value coding protocol's **takeValues:forKeys:** method. Invocation-based undo is useful for undoing specific state-changing methods, such as a document object's **setFont:color:**. This more general undo operation is useful for already-defined methods, especially when their arguments aren't objects.

Regardless of the type of operations recorded, a single instance of EOUndoManager typically belongs to a single document or container of objects, called the EOUndoManager's *client*. Each EOEditingContext in an application, for example, has its own private EOUndoManager. This keeps each pair of undo and redo stacks separate so that when an undo is performed, it applies to the focal document in the application (typically the one displayed in the key window). It also relieves the individual objects from having to know the identity of their EOUndoManager.

In order to use undo effectively, either the client must claim exclusive right to alter its undoable objects—in order to record undo operations for all changes—or the objects themselves must participate in recording their changes. The first case is exemplified by a text document that holds a private NSTextView, handling all text operations by registering undo operations and forwarding the change to the NSTextView. For the second case, the **willChange** method defined by Enterprise Objects Framework allows any object to notify observers that it's about to change. EOEditingContexts, being containers for enterprise objects, receive these change notifications and record undo operations (among many other things). Even in this case, interaction with the EOUndoManager is handled exclusively by the container object.

Simple Undo

To record a simple undo operation, you need only invoke **registerUndoWithTarget:selector:arg:**, giving the object to be sent the undo operation selector, the selector to invoke, and an argument to pass with that message. The target object is rarely the actual object whose state is changing; instead, it's the client object, a document or container that holds many undoable objects. An object like EOEditingContext, for example, can record an undo operation for **insertObject:** by registering a **deleteObject:** message with the object inserted (**undoManager** is an instance variable):

```
[undoManager registerUndoWithTarget:self selector:@selector(deleteObject:)
    arg:anObject];
```

An update might be recorded for undo like so:

```
NSDictionary *updateDict = [NSDictionary dictionaryWithObjectsAndKeys:anObject,
    @"object", [anObject snapshot], @"snapshot"];

[undoManager registerUndoWithTarget:self
    selector:@selector(revertUpdate:)
    arg:updateDict];
```

This fragment is likely to be executed as a result of **anObject** invoking the standard **willChange** method, which announces that the object's state is going to change. Since it hasn't changed yet, the state can be recorded for later undo. This fragment, then, registers the client (**self**) to be sent a **revertUpdate:** message with the object and its old state when undo is performed. The old values are retrieved with a **snapshot** message. **revertUpdate:** can be implemented to pass the old state back to the object:

```
- (void)revertUpdate:(NSDictionary *)updateDict
{
    [[updateDict objectForKey:@"object"]
        udpateFromSnapshot:[updateDict objectForKey:@"snapshot"]];
    return;
}
```

Both **snapshot** and **updateFromSnapshot:** are methods added to NSObject by the Framework. See the NSObject Additions specification for more information.

Invocation-Based Undo

For other changes involving specific methods or arguments that aren't objects, you can use invocation-based undo, which records an actual message to revert the target object's state. As with simple undo, you record a message that reverts the object to its state before the change. However, in this case you do so by sending the message directly to the EOUndoManager, after preparing it with a special message to note the target:

```
[[myUndoManager prepareWithInvocationTarget:textObject]
   setFont:[textObject font] color:[textObject textColor]];
[textObject setFont:newFont color:newColor];
```

prepareWithInvocationTarget: records the argument as the target of the undo operation about to be established. Following this, you send the message that will revert the target's state—in this case, **setFont:color:**. Because EOUndoManager doesn't respond to this method, **forwardInvocation:** is invoked, which EOUndoManager implements to record the NSInvocation containing the target, selector, and all arguments. Performing undo later results in **textObject** being sent a **setFont:color:** message with the old values.

Performing Undo and Redo

Performing undo and redo is usually as simple as sending **undo** and **redo** messages to the EOUndoManager. **undo** closes the last open undo group and then applies all of the undo operations in that group (recording any undo operations as redo operations instead). **redo** likewise applies all of the redo operations on the top redo group.

undo is intended for undoing top-level groups, and shouldn't be used for nested undo groups. If any unclosed, nested undo groups are on the stack when **undo** is invoked, it raises an exception. To undo nested groups, you must use explicitly close the group with an **endUndoGrouping** message, then use **undoNestedGroup** to undo it. Note also that if you turn off automatic grouping by event with **setGroupsByEvent:**, you must explicitly close the current undo group with **endUndoGrouping** before invoking either undo method.

Cleaning the Undo Stack

EOUndoManager doesn't retain the targets of undo operations, for several reasons. Foremost is that the client—the object registering operations—typically owns the EOUndoManager, so retaining it would create cycles. The EOUndoManager does contain references to the targets of undo operations, however, which it uses to send undo messages when undo is performed. If a target object has been deallocated, this will cause errors.

To remedy this, the client must take care to clear undo operations for targets that are being deallocated. This typically occurs in one of three ways:

- The client is the exclusive owner of the EOUndoManager and the target of all undo operations. In this case the client can simply release the EOUndoManager in its **dealloc** method.
- The client shares the EOUndoManager with other clients. To handle this the client should send **forgetAllWithTarget:** to the EOUndoManager before releasing it in its **dealloc** method.
- The client registers objects other than itself for undo operations. Here either the client must watch for the other objects being deallocated in order to send **forgetAllWithTarget:**, or the other objects must do so themselves when deallocated (which requires that they have a reference to the EOUndoManager). This is likely to be needed with invocation-based undo.

In a more general sense, it sometimes makes sense to clear all undo and redo operations. Some applications might want to do this when saving a document, for example. To this end, EOUndoManager defines the **forgetAll** method, which clears both stacks.

Undo Checkpoint Notifications

Objects sometimes delay performing changes, for various reasons. This means they may also delay registering undo operations for those changes. Because EOUndoManager collects individual operations into groups, it must be sure to synchronize its client with the creation of these groups so that operations are entered into the proper undo groups. To this end, whenever an EOUndoManager opens or closes a new undo group (except when it opens a top-level group), it posts an “UndoManagerCheckpointNotification” so that observers can apply their pending undo operations to the group in effect. The EOUndoManager’s client should register itself as an observer for this notification and record undo operations for all pending changes upon receiving it.

Method Types

Registering undo operations	
	– registerUndoWithTarget:selector:arg:
	– prepareWithInvocationTarget:
	– forwardInvocation:
Checking undo ability	
	– canUndo
	– canRedo
Performing undo and redo	
	– undo
	– undoNestedGroup
	– redo
Limiting the undo stack	
	– setLevelsOfUndo:
	– levelsOfUndo
Creating undo groups	
	– beginUndoGrouping
	– endUndoGrouping
	– setGroupsByEvent:
	– groupsByEvent
Disabling undo	
	– disableUndoRegistration
	– reenableUndoRegistration
Checking whether undo or redo is being performed	
	– isUndoing
	– isRedoing
Clearing undo operations	

Instance Methods

beginUndoGrouping

– (void)**beginUndoGrouping**

– forgetAll
– forgetAllWithTarget:

Marks the beginning of an undo group. All individual undo operations before a subsequent **endUndoGrouping** message are grouped together and reversed by a later **undo** message. Undo groups can be nested, thus providing functionality similar to nested transactions.

This method posts an "UndoManagerCheckpointNotification".

canRedo

– (BOOL)**canRedo**

Returns YES if the receiver has any actions to redo, NO if it doesn't.

Because any undo operation registered clears the redo stack, this method posts an "UndoManagerCheckpointNotification" to allow clients to apply their pending operations before testing the redo stack.

See also: – **canUndo**, – **redo**

canUndo

– (BOOL)**canUndo**

Returns YES if the receiver has any actions to undo, NO if it doesn't. This does *not* mean that you can safely invoke **undo** or **undoNestedGroup**; you may have to close open undo groups first.

See also: – **canRedo**, – **endUndoGrouping**, – **registerUndoWithTarget:selector:arg:**

disableUndoRegistration

– (void)**disableUndoRegistration**

Disables the recording of undo operations, whether by **registerUndoWithTarget:selector:arg:** or by invocation-based undo. This method can be invoked multiple times; **reenableUndoRegistration** must be invoked an equal number of times to actually re-enable undo registration.

endUndoGrouping

– (void)**endUndoGrouping**

Marks the end of an undo group. All individual undo operations back to the matching **beginUndoGrouping** message are grouped together and reversed by a later **undo** or **undoNestedGroup** message. Undo groups can be nested, thus providing functionality similar to nested transactions. Raises an NSInternalInconsistencyException if there's no **beginUndoGrouping** message in effect.

This method posts an "UndoManagerCheckpointNotification".

forgetAll

– (void)**forgetAll**

Clears the undo and redo stacks and reenables the receiver.

See also: – **levelsOfUndo**,– **reenableUndoRegistration**, – **forgetAllWithTarget:**

forgetAllWithTarget:

– (void)**forgetAllWithTarget:**(id)*target*

Clears the undo and redo stacks of all operations involving *target* as the recipient of the undo message. Doesn't re-enable the receiver if it's disabled. An object that shares an EOUndoManager with other clients should invoke this message in its implementation of **dealloc**.

See also: – **reenableUndoRegistration**, – **forgetAll**

forwardInvocation:

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*

Overrides NSObject's implementation to record *anInvocation* as an undo operation. Also clears the redo stack. Raises an NSInternalInconsistencyException if **prepareWithInvocationTarget:** wasn't invoked before this method; this method then clears the prepared invocation target. See ""Invocation-Based Undo"" in the class description for more information.

Raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **registerUndoWithTarget:selector:arg:**, – **groupsByEvent**

groupsByEvent

– (BOOL)**groupsByEvent**

Returns YES if the receiver automatically creates undo groups around each pass of the run loop, NO if it doesn't. The default is YES.

See also: – **beginUndoGrouping**, – **setGroupsByEvent:**

isRedoing

– (BOOL)**isRedoing**

Returns YES if the receiver is in the process of performing its **redo** method, NO otherwise.

See also: – **isRedoing**

isUndoing

– (BOOL)**isUndoing**

Returns YES if the receiver is in the process of performing its **undo** or **undoNestedGroup** method, NO otherwise.

levelsOfUndo

– (unsigned int)**levelsOfUndo**

Returns the maximum number of top-level undo groups the receiver will hold. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

See also: – **endUndoGrouping**, – **isUndoing**, – **setLevelsOfUndo:**

prepareWithInvocationTarget:

– (id)**prepareWithInvocationTarget:**(id)*target*

Prepares the receiver for invocation-based undo with *target* as the subject of the next undo operation and returns **self**. See ""Invocation-Based Undo"" in the class description for more information.

See also: – **forwardInvocation:**

redo

– (void)**redo**

Performs the operations in the last group on the redo stack, if there are any, recording them on the undo stack as a single group.

This method posts an “UndoManagerCheckpointNotification”.

See also: – **redo**, – **registerUndoWithTarget:selector:arg:**

reenableUndoRegistration

– (void)**reenableUndoRegistration**

Balances a prior **disableUndoRegistration** message. Undo registration isn’t actually re-enabled until a re-enable message balances the last disable message in effect. Raises an NSInternalInconsistencyException if invoked while no **disableUndoRegistration** message is in effect.

registerUndoWithTarget:selector:arg:

– (void)**registerUndoWithTarget:**(id)*target* **selector:**(SEL)*aSelector* **arg:**(id)*anObject*

Records a single undo operation for *target*, so that when undo is performed it’s sent *aSelector* with *anObject* as the sole argument. Also clears the redo stack. Doesn’t retain *target*. See ““Simple Undo”” in the class description for more information.

Raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **forwardInvocation:**, – **groupsByEvent**

setGroupsByEvent:

– (void)**setGroupsByEvent:**(BOOL)*flag*

Sets whether the receiver automatically groups undo operations during the run loop. If *flag* is YES, the receiver creates undo groups around each pass through the run loop; if *flag* is NO it doesn’t. The default is YES.

If you turn automatic grouping off, you must close groups explicitly before invoking either **undo** or **undoNestedGroup**.

See also: – **groupsByEvent**

setLevelsOfUndo:

– (void)**setLevelsOfUndo:**(unsigned int)*anInt*

Sets the maximum number of top-level undo groups the receiver will hold to *anInt*. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

If invoked with a limit below the prior limit, old undo groups are immediately dropped.

See also: – **endUndoGrouping**, – **levelsOfUndo**

undo

– (void)**undo**

Closes the top-level undo group if necessary and invokes **undoNestedGroup**. Raises an NSInternalInconsistencyException if more than one undo group is open (that is, if the last group isn’t at the top level).

This method posts an “UndoManagerCheckpointNotification”.

See also: – **endUndoGrouping**, – **groupsByEvent**

undoNestedGroup

– (void)**undoNestedGroup**

Performs the undo operations in the last undo group (whether top-level or nested), recording the operations on the redo stack as a single group. Raises an NSInternalInconsistencyException if any undo operations have been registered since the last **endUndoGrouping** message.

This method posts an “UndoManagerCheckpointNotification”.

Notifications

UndoManagerCheckpointNotification

Posted whenever an EOUndoManager opens or closes an undo group (except when it opens a top-level group), and when checking the redo stack in **canRedo**. The notification contains:

Notification Object	The EOUndoManager
Userinfo	**nil**

See also: – **undo**

NSArray Additions

Class Cluster Description

Enterprise Objects Framework adds some methods to the Foundation Framework's NSArray class cluster, for filtering objects according to an EOQualifier and sorting them according to a series of EOSortOrderings. It also adds methods for key-value coding, with special support for aggregates, and a convenience method for filtering an array with a specified qualifier.

NSArray

Inherits From: NSObject

Declared In: EOControl/EOQualifier.h
EOControl/EOSortOrdering.h
EOControl/EOClassDescription.h
EOControl/EOKeyValueCoding.h

Class Description

Enterprise Objects Framework adds two methods to the Foundation Framework's NSArray class, for filtering objects according to an EOQualifier and sorting them according to a series of EOSortOrderings.

Instance Methods

computeAvgForKey:

– (id)**computeAvgForKey:**(NSString *)*key*

Returns as an NSDecimalNumber the average of the values the receiver's objects have for *key*. If the array is empty, returns **nil**.

See also: – **valueForKey:**, – **computeCountForKey:**, – **computeMaxForKey:**, – **computeMinForKey:**, – **computeSumForKey:**

computeCountForKey:

– (id)**computeCountForKey:**(NSString *)*key*

Returns the number of elements in the receiver as an NSNumber; the argument *key* is ignored.

See also: – **valueForKey:**, – **computeAvgForKey:**, – **computeMaxForKey:**, – **computeMinForKey:**, – **computeSumForKey:**

computeMaxForKey:

– (id)**computeMaxForKey:**(NSString *)*key*

Returns the value for *key* that is the highest for all of the objects in the receiver. If the array is empty, returns **nil**.

See also: – **valueForKey:**, – **computeAvgForKey:**, – **computeCountForKey:**, – **computeMinForKey:**, – **computeSumForKey:**

computeMinForKey:

– (id)**computeMinForKey:**(NSString *)*key*

Returns the object in the receiver that has the lowest value for *key*. If the array is empty, returns **nil**.

See also: – **valueForKey:**, – **computeAvgForKey:**, – **computeCountForKey:**, – **computeMaxForKey:**, – **computeSumForKey:**

computeSumForKey:

– (id)**computeSumForKey:**(NSString *)*key*

Returns as an NSDecimalNumber the sum of the values the receiver's objects have for *key*.

See also: – **valueForKey:**,– **computeAvgForKey:**, – **computeCountForKey:**, – **computeMaxForKey:**, – **computeMinForKey:**, – **computeSumForKey:**

filteredArrayUsingQualifier:

– (NSArray *)**filteredArrayUsingQualifier:**(EOQualifier *)*aQualifier*

Returns a new NSArray that contains only the objects from the receiver matching *aQualifier*.

shallowCopy

– (NSArray *)**shallowCopy**

Returns an NSArray that represents a shallow copy of the receiver. Used by Enterprise Objects Framework to snapshot to-many relationship properties.

sortedArrayUsingKeyOrderArray:

– (NSArray *)**sortedArrayUsingKeyOrderArray:**(NSArray *)*orderings*

Creates and returns a new NSArray by sorting the objects of the receiver according to the EOSortOrderings in *orderings*. The objects are compared by extracting the sort properties using the added NSObject method **valueForKey:** and sending them **compare:** messages.

See also: – **sortUsingKeyOrderArray:** (NSMutableArray)

valueForKey:

– (id)**valueForKey:**(NSString *)*key*

When passed a "normal" key, returns an array composed of the results of sending **valueForKey:** to all elements of the array. When passed a key prefixed with "@", returns a single value that is the result of invoking an aggregate function on the values of the array.

For instance, if this method were passed the key `@sum.budget`, it would invoke `computeSumForKey:@"budget"` on the array, which would add the values for the budget keys for all of the objects in the array. The returned value would be the sum of all of the objects' budgets. The following aggregates are defined: @sum, @count, @avg, @min, @max. You can extend this set by adding methods to NSArray of the form **compute*Name*ForKey:**.

See also: – **computeAvgForKey:**, – **computeCountForKey:**, – **computeMaxForKey:**, – **computeMinForKey:**, – **computeSumForKey:**

NSMutableArray

Inherits From: NSObject

Declared In: EOControl/EOSortOrdering.h

NSMutableArray has one added method for sorting its elements according to a series of EOSortOrderings.

Instance Methods

sortUsingKeyOrderArray:

– (void)**sortUsingKeyOrderArray:**(NSArray *)*orderings*

Sorts the objects of the receiver according to the EOSortOrderings in *orderings*. The objects are compared by extracting the sort properties using the added NSObject method **valueForKey:** and sending them **compare:** messages.

See also: – **sortedArrayUsingKeyOrderArray:** (NSArray)

NSException Additions

Inherits From:	NSObject
Declared In:	EOControl/EOClassDescription.h

Class Description

Enterprise Objects Framework adds methods to the Foundation Framework's NSException class for handling validating errors and augmenting an exception's **userInfo** dictionary. The methods used for validation errors are **validationExceptionWithFormat:** and **aggregateExceptionWithExceptions:**. You use **validationExceptionWithFormat:** in an enterprise object's **validateFor...** or **validate***Property***:** method, as described in the NSObject Additions class specification. The other method used for validation errors, **aggregateExceptionWithExceptions:**, is used internally by the Framework to group multiple validation exceptions together.

The method **exceptionAddingEntriesToUserInfo:** is used to augment an exception's **userInfo** dictionary.

Method Types

Creating a validation exception
+ validationExceptionWithFormat:

Collecting exceptions
+ aggregateExceptionWithExceptions:

Returning an exception with an augmented userInfo dictionary
– exceptionAddingEntriesToUserInfo:

Class Methods

aggregateExceptionWithExceptions:

+ (NSException *)**aggregateExceptionWithExceptions:**(NSArray *)*subexceptions*

Returns an NSException with the same name, reason, and **userInfo** dictionary of the first exception in the *subexceptions* array, but with the **userInfo** dictionary augmented with the list of subexceptions under the key EOAdditionalExceptionsKey.

See also: – **exceptionAddingEntriesToUserInfo:**

validationExceptionWithFormat:

+ (NSException *)**validationExceptionWithFormat:**(NSString *)*format*, ...

Returns an NSException whose name is EOValidationException and whose reason is an NSString created from *format* and subsequent arguments. For example:

```
[NSException validationExceptionWithFormat:@"invalid name \"%@\": entity names
cannot be nil or empty", name];
```

Instance Methods

exceptionAddingEntriesToUserInfo:

– (NSException *)**exceptionAddingEntriesToUserInfo:**(NSDictionary *)*additions*

Returns an NSException whose **userInfo** dictionary has been augmented with the object and property information contained in *additions*. When exceptions are raised by certain validation methods such as **validateValue:forKey:**, this message is sent to the exception to create a duplicate exception with object and property information added to the new exception's **userInfo** dictionary. This information is stored in the **userInfo** dictionary under the keys EOValidatedObjectUserInfoKey and EOValidatedPropertyUserInfoKey, respectively. The exception that's returned by this method has the same class with the same name and reason as the original exception; the only difference is the augmented **userInfo** dictionary.

NSObject Additions

Inherits From:	none *(NSObject is a root class)*
Declared In:	EOControl/EOClassDescription.h EOControl/EOEditingContext.h EOControl/EOKeyValueCoding.h EOControl/EOObserver.h

Class at a Glance

Purpose

Defines basic functionality for all enterprise objects. Create a subclass when you need a custom enterprise object class to perform business logic; otherwise use EOGenericRecords.

Principal Attributes

- EOClassDescription
- EOEditingContext

Creation

– init	Designated initializer.
– initWithEditingContext:classDescription:globalID:	Optional initializer.
– awakeFromFetchInEditingContext:	Performs additional initialization after the object is fetched.
– awakeFromInsertionInEditingContext:	Performs additional initialization after the object is created in memory.

Commonly Used Methods

– willChange	Notifies observers of a change in state.
– editingContext	Returns the receiver's EOEditingContext.
– addObject:toBothSidesOfRelationshipWithKey:	Adds an object to a relationship property and the receiver to the reciprocal relationship.
– removeObject:fromBothSidesOfRelationshipWithKey:	Removes an object from a relationship property and the receiver from the reciprocal relationship.

Methods to Implement or Override

The following methods are invoked by the Framework.

– set*Key*:	Sets the value for the property named *key.*
– *key*	Retrieves the value for the property named *key.*

– addTo*Key*:	Adds an object to a relationship property named *key*.
– removeFrom*Key*:	Removes an object from the property named *key*.
– handleTakeValue:forUnboundKey:	Handles a failure of **takeValue:forKey:** to find a property.
– handleQueryWithUnboundKey:	Handles a failure of **valueForKey:** to find a property.
– unableToSetNilForKey:	Handles an attempt to set a non-object property's value to **nil**.
– validate*Key*:	Validates a value for the property named *key*.
– validateForDelete	Validates all properties before deleting the receiver.
– validateForInsert	Validates all properties before inserting the receiver.
– validateForSave	Validates all properties before saving the receiver.
– validateForUpdate	Validates all properties before updating the receiver.

Class at a Glance

Class Description

Enterprise Objects Framework adds a number of methods to NSObject for supporting operations common to all enterprise objects. Among these are methods for initializing instances, announcing changes, setting and retrieving property values, and performing validation of state. Some of these methods are for enterprise objects to implement or override, and some are meant to be used as defined by the Framework. Many methods are used internally by the Framework and rarely invoked by application code. The implementation or use of each method is highlighted in the following sections, which describe the major functional groups.

Initialization Methods

Enterprise objects are initialized using the **initWithEditingContext:classDescription:globalID:**, which by default simply invokes **init**. You can place your custom initialization code in **init**, or you can override **initWithEditingContext:classDescription:globalID:** to take advantage of the extra information available with this method.

After initialization, an enterprise object receives an **awake...** message. The particular message depends on whether the object has been fetched from a database or created anew in the application. In the former case, it receives an **awakeFromFetchInEditingContext:** message. In the latter, it receives an **awakeFromInsertionInEditingContext:** message. The receiver can override either method to perform extra initialization—such as setting default values—based on how it was created.

Announcing Changes

For the Framework to keep all areas of an application synchronized, enterprise objects must notify their observers when their state changes. Objects do this by simply invoking **willChange** before altering any instance variable or other kind of state. This method informs all observers that the invoker is about to change. See the EOObserverCenter class specification for more information on change notification.

The primary observer of changes in an object is its EOEditingContext. EOEditingContext is a subclass of EOObjectStore that manages collections of objects in memory, tracking inserts, deletes, and updates, and propagating changes to the persistent store as needed. You can get the EOEditingContext that contains an object by sending the object an **editingContext** message.

Getting Object and Class Metadata

One of the larger groups of methods added to NSObject provides information about an object's properties. Most of these methods consult an EOClassDescription to provide their answers. The **classDescription** method return an object's EOClassDescription. See that class specification for the methods it implements. Methods you can send directly to any object include **entityName**, which provides the name of the entity mapped to the receiver's class; **allPropertyKeys**, which returns the names of all the receiver's class properties, attributes and relationships alike; and **attributeKeys**, which returns just the names of the attributes.

Some methods return information about relationships. **toOneRelationshipKeys** and **toManyRelationshipKeys** return the names of the receiver's relationships, while **isToManyKey:** tells which kind a particular relationship is. **deleteRuleForRelationshipKey:** indicates what should happen to the receiver's relationships when it's deleted. Similarly, **ownsDestinationObjectsForRelationshipKey:** indicates what should happen when another object is added to or removed from the receiver's relationship. Another method, **classDescriptionForDestinationKey:**, returns the EOClassDescription for the objects at the destination of a relationship.

These methods are all properly implemented in terms of the receiver's EOClassDescription, so unless your object class doesn't have an EOClassDescription, there's little need to override them. One method you might override in your enterprise object class, however, is **inverseForRelationshipKey:**. Given the name of one of the receiver's relationships, this method finds the destination object's class data and determines the name of the relationship that points back at the receiver. The default implementation of this method looks for a relationship predicated on the same attributes in both the source and destination, which works correctly in most cases. If, however, you define a reciprocal pair of relationships on different attributes, you should override this method to take that into account. See the method description for an example.

Key-Value Coding Methods

A special set of methods form the Framework's main data transport mechanism, in which the properties of an enterprise object are accessed indirectly by name (or key), rather than directly through invocation of an accessor method or as instance variables. Thus, any object's state can be accessed in a consistent manner.

The basic methods for accessing an enterprise object's values are **takeValue:forKey:** and **valueForKey:**. These two methods are defined by NSObject to use the accessor methods normally implemented by objects (or to access instance variables directly if need be), so that you don't have to write special code simply to integrate your enterprise objects into the Framework. Another pair of methods, **takeValuesFromDictionary:** and **valuesForKeys:**, gives access to groups of properties. Lastly, **valueForKeyPath:** and **valueForKeyPath:** give access to properties across key paths of the form *relationship.property*; for example, "department.name".

All of the **takeValue...** methods have corresponding **takeStoredValue...** methods for setting object values from object store values: **takeStoredValue:forKey:**, **takeStoredValue:forKeyPath:**, and **takeStoredValuesFromDictionary:**. To enable them, override **useStoredAccessor** to return YES. This is discussed in more detail below.

Default Implementations; Handling Access Errors

The Framework provides default implementations of **takeValue:forKey:** and **valueForKey:** that work for all objects. The other four access methods are implemented in terms of these two. These implementations are general enough that your enterprise object classes should rarely need to override either key-value coding method. In accessing an object's property, the default NSObject implementations of the key-value coding methods use the class definition as follows:

1. The key-value coding method looks for an accessor method based on the key. For example, with a key of "lastName", **takeValue:forKey:** looks for a method named **setLastName:** (note that the first letter of the property name is made uppercase), and **valueForKey:** looks for a method of the form **lastName**.
2. If the key-value coding method doesn't find an accessor method, and the class responds YES to an **accessInstanceVariablesDirectly** message (which it does by default), it looks for an instance variable whose name is the same as the key and sets or retrieves its value directly. In setting an instance variable that's an object, **takeValue:forKey:** retains the new value and autoreleases the old one.
3. If neither an accessor method nor an instance variable can be found, the default implementations invoke methods that your custom objects can override to handle failures. **handleTakeValue:forUnboundKey:** is invoked from **takeValue:forKey:**, and **handleQueryWithUnboundKey:** is invoked from **valueForKey:**. Normally these methods raise an exception, but you can implement them to set or get a value in whatever way is needed.

The Framework also provides methods for setting object values from object store values: **takeStoredValue:forKey:**, **takeStoredValue:forKeyPath:**, and **takeStoredValuesFromDictionary:**. You cause these methods to be used instead of their **takeValue...** counterparts by overriding **useStoredAccessor** to return YES. When you override **useStoredAccessor** to return YES, the **takeStoredValue...** methods are used whenever an object moves from one object store to another—for example, when you instantiate objects from database data, or when you transfer objects between EOEditingContexts. In all other cases the regular **takeValue...** methods are used, such as when a user modifies an object by providing a new value for it in a user interface. To think of it another way, the **takeStoredValue...** methods let you bypass the logic in your **set...** methods, whereas the **takeValue...** methods execute that logic.

The **takeStoredValue...** methods are especially useful in cases where an object has instance variables whose values are interdependent. For example, suppose you have a Product object with **status** and **dateOfSale** attributes. When the Product's **status** changes to "sold," you'd also want to set its **dateOfSale** value—in all likelihood, by invoking **setDateOfSale:** from the object's **setStatus:** method. If you were using **takeValue:forKey:**, initializing a Product object from database data would have the effect of invoking the object's **setStatus:** method, which in turn would attempt to change the object's **dateOfSale** date. You can prevent this from happening from using the **takeStoredValue...** methods.

When you override **useStoredAccessor** to return YES and you're changing the value of an object's property, the default NSObject implementations of the key-value coding methods use the class definition as follows:

1. The key-value coding method looks for an instance variable whose name is the same as the key, but preceded by an underbar. It then sets the instance variable's value directly. For example, with a key of "lastName", **takeStoredValue:forKey:** looks for an instance variable called **_lastName**.
2. If the key-value coding method doesn't find an instance variable, it looks for an accessor method based on key, preceded by an underbar. For example, with a key of "lastName", **takeStoredValue:forKey:** looks for a method called **_setLastName.**
3. If the key-value coding method doesn't find an underbar-preceded instance variable or accessor method, it looks for an instance variable whose name is the same as the key (**lastName**) and sets its value directly.
4. Finally, the key-value coding method looks for an accessor method based on the key. For the key "lastName", this would be **setLastName:**.
5. If none of the above instance variables or accessor methods can be found, the default implementations invoke methods that your custom objects can override to handle failures. **handleTakeValue:forUnboundKey:** is invoked from **takeValue:forKey:**, and **handleQueryWithUnboundKey:** is invoked from **valueForKey:**. Normally these methods raise an exception, but you can implement them to set or get a value in whatever way is needed.

The key-value coding methods cache attribute bindings for both accessor methods and instance variables, making lookups efficient. If you need to clear these bindings—as when you add or remove a class from the run-time system—you can invoke **flushAllKeyBindings** to do so.

Some subclasses of NSObject override the default implementations. EOGenericRecord's implementations, for example, simply store and retrieve the properties in an NSDictionary object held by the EOGenericRecord. NSDictionary and NSMutableDictionary, though not suitable for use as enterprise objects, meaningfully implement these methods by directly accessing their key-value pairs.

Type Checking and Type Conversion

The default implementations of the key-value coding methods accept any object as a value, and do no type checking or type conversion among object classes. It's possible, for example, to pass an NSString to **takeValue:forKey:** as the value for a property the receiver expects to be an NSDate. The sender of a key-value coding message is thus responsible for ensuring that a values is of the proper class, typically by using the **validateValue:forKey:** method to coerce it to the proper type. The interface layer's

EODisplayGroup uses this on all values received from interface user objects, for example, as well as relying on number and date formatters to interpret string values typed by the user. For more information on the **validateValue:forKey:** method, see the EOClassDescription and EOEntityClassDescription class specifications.

The key-value coding methods handle one special case with regard to value types. For enterprise objects that access numeric values as C scalar types, these methods automatically convert between the scalar types and NSNumber objects. For example, suppose your enterprise object defines these accessor methods:

– (void)**setSalary:**(unsigned int)*salary*
– (unsigned int)**salary**

For the **setSalary:** method, **takeValue:forKey:** converts the object value for the “salary” key in the dictionary to an **unsigned int** and passes it as *salary*. Similarly, **valueForKey:** converts the return value of the **salary** method to an NSNumber and returns that.

The default implementations support the following scalar types:

char	unsigned char
short	unsigned short
int	unsigned int
long	unsigned long
float	double

Object values are converted to these types with the standard messages **charValue**, **intValue**, **floatValue**, and so on. Note that the key-value coding methods don’t check that an object value actually responds to these messages; this can result in a run-time error if the object doesn’t respond to the appropriate message.

One type of conversion these methods can’t perform is that from a **nil** object value to a scalar value. C scalar values define no equivalent of a database system’s NULL value, so these must be handled by the object itself. Upon encountering an **nil** while setting a scalar value, the **takeValue:forKey:** invokes **unableToSetNilForKey:**, which by default simply raises an exception. Enterprise object classes that use scalar values which may be NULL in the database should override this method to substitute the appropriate scalar value for **nil**, reinvoking **takeValue:forKey:** to set the substitute value. This method works in general to handle setting scalar properties to **nil**.

EONull in Collections

Because collection objects such as NSArray and NSDictionary can’t contain **nil** as a value, it must represented by a special object, EONull. EONull provides a single instance that represents the NULL value for object attributes. The default implementations of **takeValuesFromDictionary:** and **valuesForKeys:**

translate EONull and **nil** between NSDictionaries and enterprise objects, removing the need for your objects to explicitly test for EONull values.

Relationship Accessor Methods

Building on the key-value coding methods, another group of methods allows you to modify relationship properties by adding and removing single objects, rather than replacing the entire content of the relationship, and to modify relationships so that reciprocal relationships are automatically adjusted. **addObject:toPropertyWithKey:** and **removeObject:fromPropertyWithKey:** handle the first situation, doing all the work of altering arrays for to-many relationships. They both check first for a method you might implement, **addTo***Key***:** or **removeFrom***Key***:**, invoking that method if it's implemented, otherwise using the basic key-value coding methods to do the work.

Reciprocal relationships are handled by **addObject:toBothSidesOfRelationshipWithKey:** and **removeObject:fromBothSidesOfRelationshipWithKey:**. For example, when you add an Employee to a Department's **employees** relationship, or remove it, you also want the Employee's **department** relationship altered to suit. These two methods take care of tracing the inverse relationship and use **addObject: toPropertyWithKey:** and **removeObject:fromPropertyWithKey:** to alter both relationships, whether they're to-one or to-many. Unless you have specific reasons to do otherwise, you should always use the methods that handle reciprocal relationships so that back pointers are properly updated.

Two other methods that affect relationships are typically used internally by the Framework. You should rarely have a need either to invoke or override them. **propagateDeleteWithEditingContext:** applies an object's delete rule to the destinations of its relationships. The delete rule specifies whether a destination object should be ignored, also deleted, or whether the deletion should be disallowed if a destination exists. **clearProperties** simply sets all of the receiver's relationship properties to **nil**. An EOEditingContext uses this method to break circular references between its objects when the context is deallocated.

Snapshots

The key-value coding methods define a general mechanism for accessing an object's properties, but you first have to know what those properties are. Sometimes, however, you just want to preserve an object's entire state for later use, whether to undo changes to the object, compare the values that have changed, or just keep a record of the changes. The snapshotting methods provide this service, extracting or setting all properties at once and performing the necessary conversions for proper behavior. **snapshot** returns an NSDictionary containing all the receiver's properties, with EONull substituted for **nil** and arrays reproduced as shallow, immutable copies. **updateFromSnapshot:** sets properties of the receiver to the values in a snapshot, converting EONull to **nil**, and making shallow, mutable copies of any array values (allowing the object to add to and remove from the array).

Validation

Validating new values is a vital part of business logic. Several methods added to NSObject support validation at different stages of an object's life. Validation methods check for illegal value types, values outside of established limits, illegal relationships, and so on. All validation methods return **nil** if the values under consideration are valid, or an NSException indicating how the values aren't valid.

There are two kinds of validation methods that you can override. The first covers individual properties, when it's important to validate a value before it changes. These methods are invoked automatically by the Framework when it changes a property value, such as when the user makes an edit in the user interface. These methods are dynamically invoked based on the property name. The second kind covers operations to the external store—inserting, updating, and deleting. These methods are invoked when the associated operation is performed. You can override these methods in your custom enterprise object classes to perform delayed validation of properties, to compare multiple properties against one another, and to allow or refuse the operation based on property values. For example, a Fee object might refuse to be deleted if it hasn't been paid yet.

Immediate Validation of Individual Properties

The most general method, **validateValue:forKey:**, is used by the Framework when an EODisplayGroup passes an updated value to the object and when the object is saved. This method does two things: coerce the value into an appropriate type for the object, and validate it according to the object's rules. Coercion is performed automatically for you, so all you need handle is validation itself.

The default implementation of **validateValue:forKey:** consults the object's EOClassDescription for basic errors, such as a **nil** value when that isn't allowed. If no basic errors exist, this method then examines the object's class itself for a method of the form **validate***Key***:** and invokes that. These are the methods that your custom classes can implement to validate individual properties, such as **validateAge:** to check that the value the user entered is within acceptable limits.

For example, suppose that Member objects have an **age** attribute stored as an integer. This attribute has an lower limit of 16, defined by the Member class. Now, suppose the user types "12" into a text field for the age. Before the EODisplayGroup updates the selected object, it sends the object a **validateValue:forKey:** message. The object uses its EOEntityClassDescription to convert the string "12" into an NSNumber, then invokes **validateAge:** with that NSNumber. Member's implementation of this method compares the age to its limit of 16 and returns an EOValidationException:

```
- (NSException *)validateAge:(NSNumber *)age
{
    if ([age intValue] < 16) {
        return [NSException
            validationExceptionWithFormat:@"Age of %@ is below minimum.", age];
    }
    return nil;
}
```

The Framework adds the **validationExceptionWithFormat:** method to NSException for convenient creation of validation exceptions. The userInfo dictionaries in the exceptions raised by these methods contain the enterprise object being validated and the key (where applicable).

Validation for Specific Operations

The other validation methods are invoked at specific times—such as before the object is written to or deleted from the external store—and are particularly useful when properties must be compared or when expensive calculation is necessary. The methods are **validateForInsert**, **validateForUpdate**, **validateForSave**, and **validateForDelete**, and they're invoked automatically for the operations indicated by the method name. You can override these methods to check values individually or in groups; for example, you might verify that a pair of dates is in the proper temporal order:

```
- (NSException *)validateForSave
{
    if ([startDate compare:endDate] == NSOrderedDescending) {
        return [NSException
            validationExceptionWithFormat:@"Start date must not follow end date."];
    }
    return [super validateForSave];
}
```

Note that this method also invokes **super**'s implementation. This is important, as the default implementations of the **validateFor...** pass the check on to the object's EOClassDescription, which performs basic checking among properties. The access layer's EOEntityClassDescription class verifies constraints based on an EOModel, such as delete rules. For example, the delete rule for a Department object might state that it can't be deleted if it still contains Employee objects.

validateForSave is the generic validation method for when an object is written to the external store. The default implementations of **validateForInsert** and **validateForUpdate** both invoke this method. If an object performs validation that isn't specific to insertion or to updating, it should go in **validateForSave**.

Method Types

Initializing enterprise objects	
	– initWithEditingContext:classDescription:globalID:
	– awakeFromFetchInEditingContext:
	– awakeFromInsertionInEditingContext:
Announcing changes	
	– willChange
Getting an object's EOEditingContext	
	– editingContext

Getting class description information

– allPropertyKeys
– attributeKeys
– classDescription
– classDescriptionForDestinationKey:
– deleteRuleForRelationshipKey:
– entityName
– inverseForRelationshipKey:
– isToManyKey:
– ownsDestinationObjectsForRelationshipKey:
– toManyRelationshipKeys
– toOneRelationshipKeys

Key-value coding

– takeValue:forKey:
– valueForKey:
– takeValuesFromDictionary:
– valuesForKeys:
– takeValue:forKeyPath:
– valueForKeyPath:
– takeStoredValue:forKey:
– takeStoredValue:forKeyPath:
– takeStoredValuesFromDictionary:
– handleQueryWithUnboundKey:
– handleTakeValue:forUnboundKey:
– unableToSetNilForKey:
+ accessInstanceVariablesDirectly
+ flushClassKeyBindings
+ flushAllKeyBindings
+ useStoredAccessor

Modifying relationships

– addObject:toPropertyWithKey:
– removeObject:fromPropertyWithKey:
– addObject:toBothSidesOfRelationshipWithKey:
– removeObject:fromBothSidesOfRelationshipWithKey:
– propagateDeleteWithEditingContext:
– clearProperties

Working with snapshots

– snapshot
– updateFromSnapshot:

Validating values	
	– validateForDelete
	– validateForInsert
	– validateForSave
	– validateForUpdate
	– validateValue:forKey:
Getting descriptions	
	– eoDescription
	– eoShallowDescription
	– userPresentableDescription

Class Methods

accessInstanceVariablesDirectly

+ (BOOL)**accessInstanceVariablesDirectly**

Returns YES if the default implementations of the key-value coding methods, on finding no accessor method for a property, should access the corresponding instance variable directly. Returns NO if they shouldn't. NSObject's implementation of this method returns YES. Subclasses can override it to return NO, in which case the other methods won't access instance variables.

flushAllKeyBindings

+ (void)**flushAllKeyBindings**

Invalidates the cached key binding information for all classes (caches are kept of key-to-method or instance variable bindings in order to make key-value coding efficient).

See also: + **flushClassKeyBindings**

flushClassKeyBindings

+ (void)**flushClassKeyBindings**

Invalidates the cached key binding information for the receiving class. This method should be invoked whenever a class is modified or removed from the run-time system.

See also: + **flushAllKeyBindings**

useStoredAccessor

+ (BOOL)**useStoredAccessor**

Returns YES if the default implementations of the key-value coding methods should be accessed using the **takeStoredValue...** methods, NO otherwise. NSObject’s implementation of this method returns NO. Subclasses can override it to return YES. For more discussion of this topic, see the section ““Key-Value Coding Methods”” in the class description.

See also: – **takeStoredValue:forKey:**, – **takeStoredValue:forKeyPath:**, – **takeStoredValuesFromDictionary:**

Instance Methods

addObject:toBothSidesOfRelationshipWithKey:

– (void)**addObject:**(id)*anObject* **toBothSidesOfRelationshipWithKey:**(NSString *)*key*

Sets or adds *anObject* as the destination for the receiver’s relationship identified by *key*, and also sets or adds the receiver for *anObject*’s reciprocal relationship if there is one. For a to-one relationship, *anObject* is set using **takeValue:forKey:**. For a to-many relationship, *anObject* is added using **addObject: toPropertyWithKey:**.

This method also properly handles removing **self** and *anObject* from their previous relationship as needed. For example, if an Employee object belongs to the Research department, invoking this method with the Maintenance department removes the Employee from the Research department as well as setting the Employee’s department to Maintenance.

See also: – **removeObject:fromBothSidesOfRelationshipWithKey:**

addObject:toPropertyWithKey:

– (void)**addObject:**(id)*anObject* **toPropertyWithKey:**(NSString *)*key*

Adds *anObject* to the receiver’s to-many relationship identified by *key*, without setting a reciprocal relationship. Similar to the implementation of **takeValue:forKey:**, NSObject’s implementation of this method first attempts to invoke a method of the form **addTo***Key***:**. If the receiver doesn’t have such a method, this method gets the property array using **valueForKey:** and operates directly on that. If the array is mutable, this method simply adds *anObject*. Otherwise it constructs a new array containing any existing objects and *anObject*, then sets it using **takeValue:forKey:**.

See also: – **removeObject:fromPropertyWithKey:**, – **addObject: toBothSidesOfRelationshipWithKey:**

allPropertyKeys

– (NSArray *)**allPropertyKeys**

Returns all of the receiver’s property keys, as returned by **attributeKeys**, **toOneRelationshipKeys**, and **toManyRelationshipKeys**.

attributeKeys

– (NSArray *)**attributeKeys**

Returns the names of the receiver’s attributes, as determined from the EOClassDescription. You might wish to override this method to add keys for attributes not defined by the EOClassDescription. The access layer’s subclass of EOClassDescription, EOEntityClassDescription, returns the names of attributes designated as class properties.

See also: – **toOneRelationshipKeys**, – **toManyRelationshipKeys**.,
– **attributeKeys** (EOClassDescription)

awakeFromFetchInEditingContext:

– (void)**awakeFromFetchInEditingContext:**(EOEditingContext *)*anEditingContext*

Overridden by subclasses to perform additional initialization on the receiver upon its being fetched from the external repository into *anEditingContext*. NSObject’s implementation merely sends an **awakeObject: fromFetchInEditingContext:** to the receiver’s EOClassDescription. Subclasses should invoke **super**’s implementation before performing their own initialization.

See also: – **awakeFromInsertionInEditingContext:**

awakeFromInsertionInEditingContext:

– (void)**awakeFromInsertionInEditingContext:**(EOEditingContext *)*anEditingContext*

Overridden by subclasses to perform additional initialization on the receiver upon its being inserted into *anEditingContext*. This is commonly used to assign default values or record the time of insertion. NSObject’s implementation merely sends an **awakeObject:fromInsertionInEditingContext:** to the receiver’s EOClassDescription. Subclasses should invoke **super**’s implementation before performing their own initialization.

See also: – **awakeFromFetchInEditingContext:**

classDescription

– (EOClassDescription *)**classDescription**

Returns the EOClassDescription registered for the receiver's class. If none is found, posts an EOClassDescriptionNeededForClassNotification on behalf of the receiver's class, allowing an observer to register a an EOClassDescription. See the EOClassDescription class specification for more information.

See also: + **registerClassDescription:forClass:** (EOClassDescription)

classDescriptionForDestinationKey:

– (EOClassDescription *)**classDescriptionForDestinationKey:**(NSString *)*key*

Returns the EOClassDescription for the destination objects of the relationship identified by *key*. If none is found, posts an EOClassDescriptionNeededForClassNotification on behalf of the destination objects' class, allowing an observer to register a an EOClassDescription. See the EOClassDescription class specification for more information.

See also: + **registerClassDescription:forClass:** (EOClassDescription),
– classDescriptionForDestinationKey: (EOClassDescription)

clearProperties

– (void)**clearProperties**

Sets all of the receiver's to-one and to-many relationships to **nil**. EOEditingContexts use this method to break circular references among objects when they're deallocated. You should never need to invoke this method or override it.

See also: – **toOneRelationshipKeys**, – **toManyRelationshipKeys**, – **takeValue:forKey:**

deleteRuleForRelationshipKey:

– (EODeleteRule)**deleteRuleForRelationshipKey:**(NSString *)*relationshipKey*

Returns a rule indicating how to handle the destination of the receiver's relationship named by *relationshipKey* when the receiver is deleted. The delete rule is one of:

- EODeleteRuleNullify
- EODeleteRuleCascade
- EODeleteRuleDeny
- EODeleteRuleNoAction

For example, an Invoice object might return EODeleteRuleCascade for the relationship named “lineItems”, since when an invoice is deleted, its line items should be deleted as well.

See also: – **propagateDeleteWithEditingContext:**, – **validateForDelete**,
– **deleteRuleForRelationshipKey:** (EOClassDescription)

editingContext

– (EOEditingContext *)**editingContext**

Returns the EOEditingContext that holds the receiver.

entityName

– (NSString *)**entityName**

Returns the name of the receiver’s entity, or **nil** if it doesn’t have one.

See also: – **entityName** (EOClassDescription)

eoDescription

– (NSString *)**eoDescription**

Returns a full description of the receiver’s property values by extracting them using the key-value coding methods. An object referenced through relationships is listed with the results of an **eoShallowDescription** message (to avoid infinite recursion through cyclical relationships).

This method is useful for debugging. You can implement a **description** method that invokes this one, and the debugger’s print-object command (**po** on the command line) automatically displays this description. You can also invoke this method directly on the command line of the debugger.

See also: – **userPresentableDescription**

eoShallowDescription

– (NSString *)**eoShallowDescription**

Returns a string containing the receiver’s class and entity names, along with the memory address of its **id**.

See also: – **userPresentableDescription**

handleQueryWithUnboundKey:

– (id)**handleQueryWithUnboundKey:**(NSString *)*key*

Invoked from **valueForKey:** when it finds no property binding for *key*. NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the query in some other way.

handleTakeValue:forUnboundKey:

– (void)**handleTakeValue:**(id)*value* **forUnboundKey:**(NSString *)*key*

Invoked from **takeValue:forKey:** when it finds no property binding for *key*. NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the request in some other way.

initWithEditingContext:classDescription:globalID:

– **initWithEditingContext:**(EOEditingContext *)*anEditingContext*
    classDescription:(EOClassDescription *)*aClassDescription*
    globalID:(EOGlobalID *)*globalID*

Overridden by subclasses to perform initialization with the extra arguments provided. NSObject's implementation simply invokes **init**.

See also: – **createInstanceWithEditingContext:globalID:zone:** (EOClassDescription)

inverseForRelationshipKey:

– (NSString *)**inverseForRelationshipKey:**(NSString *)*relationshipKey*

Returns the name of the relationship pointing back to the receiver's class or entity from that named by *relationshipKey*, or **nil** if there isn't one. With the access layer's EOEntity and EORelationship, for example, reciprocality is determined by the join attributes of the two EORelationships.

You might override this method for reciprocal relationships that aren't defined using the same join attributes. For example, if a Member object has a relationship to CreditCard based on the card number, but a CreditCard has a relationship to Member based on the Member's primary key, both classes need to override this method. This is how Member might implement it:

```
- (NSString *)inverseForRelationshipKey:(NSString *)relationshipKey
{
    if ([relationshipKey isEqual:@"creditCard"]) return @"member";
        return [super inverseForRelationshipKey:relationshipKey];
}
```

See also: – **inverseForRelationshipKey:** (EOClassDescription)

isToManyKey:

– (BOOL)**isToManyKey:**(NSString *)*key*

Returns YES if the receiver has a to-many relationship identified by *key*, NO otherwise.

See also: – **toManyRelationshipKeys**, – **toOneRelationshipKeys**

ownsDestinationObjectsForRelationshipKey:

– (BOOL)**ownsDestinationObjectsForRelationshipKey:**(NSString *)*key*

Returns YES if the receiver has a relationship identified by *key* that owns its destination, NO otherwise. If an object owns the destination for a relationship, then when that destination object is removed from the relationship, it's automatically deleted. Ownership of a relationship thus contrasts with a delete rule, in that the first applies when the destination is removed and the second applies when the source is deleted.

See also: – **deleteRuleForRelationshipKey:**, – **ownsDestinationObjectsForRelationshipKey:** (EOClassDescription), – **ownsDestination** (the access layer's EORelationship)

propagateDeleteWithEditingContext:

– (void)**propagateDeleteWithEditingContext:**(EOEditingContext *)*anEditingContext*

Sends a **propagateDeleteForObject:editingContext:** message to the receiver's EOClassDescription. This causes the destination objects of the receiver's relationships to be deleted according to the delete rule for each relationship:

Delete Rule	Action
EODeleteRuleNullify	The destination object is simply removed from the relationship, and the receiver is likewise removed from the destination's reciprocal l relationship if there is one.
EODeleteRuleCascade	As above, but the destination object is also deleted and sent a **propagateDeleteWithEditingContext:** message.
EODeleteRuleDeny	Applied in **validateForDelete**, not in this method.
EODeleteRuleNoAction	The relationship is ignored when the receiver is deleted. The EODeleteRuleNoAction option is useful for tuning performance. In order to perform a deletion, Enterprise Objects Framework fires all the faults of the deleted object and then fires any to-many faults that point back to the deleted object. For example, suppose you have a simple application based on the sample Movies database. Deleting a Movie object has the effect of firing a to-one fault for the Movie's **studio** relationship, and then firing the to-many **movies** fault for that studio. In this scenario, it would make sense to set the delete rule EODeleteRuleNoAction for Movie's **studio** relationship. However, you should use this delete rule with great caution since it can result in dangling references in your object graph.

See also: – **deleteRuleForRelationshipKey:**

removeObject:fromBothSidesOfRelationshipWithKey:

– (void)**removeObject:**(id)*anObject* **fromBothSidesOfRelationshipWithKey:**(NSString *)*key*

Removes *anObject* from the receiver's relationship identified by *key*, and also removes the receiver from *anObject*'s reciprocal relationship if there is one. For a to-one relationship, *anObject* is removed using **takeValue:forKey:** with **nil** as the value. For a to-many relationship, *anObject* is removed using **removeObject:fromPropertyWithKey:**.

See also: – **addObject:toBothSidesOfRelationshipWithKey:**

removeObject:fromPropertyWithKey:

– (void)**removeObject:**(id)*anObject* **fromPropertyWithKey:**(NSString *)*key*

Removes *anObject* from the receiver's to-many relationship identified by *key*, without modifying a reciprocal relationship. Similar to the implementation of **takeValue:forKey:**, NSObject's implementation of this method first attempts to invoke a method of the form **removeFrom***Key***:**. If the receiver doesn't have such a method, this method gets the property array using **valueForKey:** and operates directly on that. If the array is mutable, this method simply locates *anObject* and removes it. Otherwise it constructs a new array containing any existing objects minus *anObject*, then sets it using **takeValue:forKey:**.

See also: – **addObject:toPropertyWithKey:**, – **removeObject: fromBothSidesOfRelationshipWithKey:**

snapshot

– (NSDictionary *)**snapshot**

Returns a dictionary whose keys are those of the receiver's attributes, to-one relationships, and to-many relationships, and whose values are the values of those properties, with EONull substituted for **nil**. For to-many relationships, the dictionary contains shallow copies of the arrays to preserve the **id**s of the contents.

See also: – **updateFromSnapshot:**, – **allPropertyKeys**, – **valueForKey:**

takeStoredValue:forKey:

– (void)**takeStoredValue:**(id)*value* **forKey:**(NSString *)*key*

Sets the property identified by *key* to *value*. If you haven't overridden **useStoredAccessor** to return YES, this method simply invokes **takeValue:forKey:**. If you have overridden **useStoredAccessor** to return YES, the default implementation does the following:

1. Searches for an instance variable whose name is the same as the key, but preceded by an underbar. Sets its value directly. For example, with a key of "lastName", **takeStoredValue:forKey:** looks for an instance variable called **_lastName**.
2. If the instance variable isn't found, searches for an accessor method based on the key, but preceded by an underbar. For example, with a key of "lastName", **takeStoredValue:forKey:** looks for a method called **_setLastName.**
3. If neither an underbar-preceded instance variable or accessor method is found, searches for an instance variable whose name is the same as the key and sets its value directly.
4. Finally, searches for an accessor method based on the key. For the key "lastName", this would be **setLastName:**.

Classes can override this method to add custom behavior. The default implementation raises an exception if an unknown key is passed in. For more discussion of key-value coding, see the section ““Key-Value Coding Methods”” in the class description.

See also: + **useStoredAccessor**, – **takeStoredValue:forKeyPath:**, – **takeStoredValuesFromDictionary:**

takeStoredValue:forKeyPath:

– (void)**takeStoredValue:**(id)*value* **forKeyPath:**(NSString *)*keyPath*

Sets the value for the derived property identified by *keyPath* to *value*. For example, suppose you have the following code:

```
[myEmployee takeStoredValue:aStreet forKeyPath:@"address.street"];
```

This code would first get the address object by invoking `[myEmployee valueForKey: @"address"]`, and then it would set the value using `[address setStoredValue:aStreet forKey:@"street"]`.

See also: + **useStoredAccessor**, – **takeValue:forKey:**, – **takeStoredValuesFromDictionary:**

takeStoredValuesFromDictionary:

– (void)**takeStoredValuesFromDictionary:**(NSDictionary *)*aDictionary*

Sets properties of the receiver with values from *aDictionary*, using their keys to identify the properties. NSObject’s implementation invokes **takeStoredValue:forKey:** for each key-value pair, substituting **nil** for EONull values in *aDictionary.*

See also: + **useStoredAccessor**, – **takeValue:forKey:**, – **takeValue:forKeyPath:**

takeValue:forKey:

– (void)**takeValue:**(id)*value* **forKey:**(NSString *)*key*

Sets the value for the property identified by *key* to *value*. NSObject’s implementation does so by first checking the receiver for a selector of the form **set***Key***:**, invoking it if there is one. If there’s no such method, and **accessInstanceVariablesDirectly** returns YES, NSObject’s implementation checks for an instance variable named *key* and sets the value directly, autoreleasing the old value and retaining the new one.

If there’s neither an accessor method nor an instance variable matching *key*, NSObject’s implementation invokes **handleTakeValue:forUnboundKey:** as a fallback mechanism. Subclasses can override **handleTakeValue:forUnboundKey:** to handle the request in some other way. For more discussion of key-value coding, see the section “Key-Value Coding Methods” in the class description.

See also: – **takeValue:forKeyPath:**, – **takeValuesFromDictionary:**, – **valueForKey:**

takeValue:forKeyPath:

– (void)**takeValue:**(id)*value* **forKeyPath:**(NSString *)*keyPath*

Sets the value for the derived property identified by *keyPath* to *value*. A key path has the form *relationship.property* (with one or more relationships); for example "department.name". NSObject's implementation of this method gets the destination object for each relationship using **valueForKey:**, and sends the final object a **takeValue:forKey:** message with *value* and *property*.

See also: – **takeValuesFromDictionary:**, – **valueForKeyPath:**

takeValuesFromDictionary:

– (void)**takeValuesFromDictionary:**(NSDictionary *)*aDictionary*

Sets properties of the receiver with values from *aDictionary*, using the keys to identify the properties. NSObject's implementation invokes **takeValue:forKey:** for each key-value pair, substituting **nil** for EONull values in *aDictionary*.

See also: – **updateFromSnapshot:**, – **takeValue:forKeyPath:**, – **valuesForKeys:**

toManyRelationshipKeys

– (NSArray *)**toManyRelationshipKeys**

Returns the names of the receiver's to-many relationships, as determined from the EOClassDescription. You might wish to override this method to add keys for relationships not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-many relationships designated as class properties.

See also: – **toOneRelationshipKeys**., – **attributeKeys**,
– **toManyRelationshipKeys** (EOClassDescription)

toOneRelationshipKeys

– (NSArray *)**toOneRelationshipKeys**

Returns the names of the receiver's to-one relationships, as determined from the EOClassDescription. You might wish to override this method to add keys for relationships not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-one relationships designated as class properties.

See also: – **attributeKeys**, – **toManyRelationshipKeys**.,
– **toOneRelationshipKeys** (EOClassDescription)

unableToSetNilForKey:

– (void)**unableToSetNilForKey:**(NSString *)*key*

Invoked from **takeValue:forKey:** when it's given a **nil** value for a scalar property (such as an **int** or a **float**). NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the request in some other way, such as by substituting zero or a sentinel value and invoking **takeValue: forKey:** again.

updateFromSnapshot:

– (void)**updateFromSnapshot:**(NSDictionary *)*aSnapshot*

Takes the values from *aSnapshot*, setting each one according to its key using **takeValue:forKey:**. In the process, EONull values are converted to **nil**, and array values are set as shallow mutable copies to preserve the **id**s of the contents.

See also: – **takeValuesFromDictionary:**, – **snapshot**

userPresentableDescription

– (NSString *)**userPresentableDescription**

Returns a short (no longer than 60 characters) description of an enterprise object based on its data. NSObject's implementation first checks to see if the enterprise object has an attribute called "name" and if so, it returns its value. Otherwise, checks for an attribute called "title". If neither of those attributes exists, this method enumerates the object's **attributeKeys** and returns the values of all of its properties, separated by commas (applying the default formatter for numbers and dates).

See also: – **eoDescription**, – **eoShallowDescription**

validateForDelete

– (NSException *)**validateForDelete**

Confirms that the receiver can be deleted in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. For example, an object can't be deleted if it has a relationship with a delete rule of EODeleteRuleDeny and that relationship has a destination object.

NSObject's implementation sends the receiver's EOClassDescription a **validateObjectForDelete:** message (which performs basic checking based on the presence or absence of values). Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForDelete
{
    NSException *exception = [super validateForDelete];

    if (/* some other violation */ ) {
        NSException *newException = /* the extra exception */;
        exception = [NSException aggregateExceptionWithExceptions:[NSArray
             arrayWithObjects:exception, newException, nil]];
    }

    return exception;
}
```

See also: – **validateForInsert**, – **validateForSave**, – **validateForUpdate**, – **validateValue:forKey:**,
+ **validationExceptionWithFormat:** (NSException Additions)

validateForInsert

– (NSException *)**validateForInsert**

Confirms that the receiver can be inserted in its current state, returning **nil** if it can or an NSException that can be raised if it can't. NSObject's implementation simply invokes **validateForSave**.

See also: – **validateForDelete**, – **validateForUpdate**, – **validateValue:forKey:**,
+ **validationExceptionWithFormat:** (NSException Additions)

validateForSave

– (NSException *)**validateForSave**

Confirms that the receiver can be saved in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation sends the receiver's EOClassDescription a **validateObjectForSave:** message, then iterates through all of the receiver's properties, invoking **validateValue:forKey:** for each one. If this results in more than one exception, the exception returned contains the additional ones in its **userInfo** dictionary under the EOAdditionalExceptions key . Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForSave
{
    NSException *exception = [super validateForSave];

    if (/* some other violation */ ) {
        NSException *newException = /* the extra exception */;
        exception = [NSException aggregateExceptionWithExceptions:[NSArray
             arrayWithObjects:exception, newException, nil]];
    }

    return exception;
}
```

Enterprise objects can implement this method to check that certain relations between properties hold; for example, that the end date of a vacation period follows the begin date. To validate an individual property, you can simply implement a method for it as described under **validateValue:forKey:**.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForUpdate**,
+ **validationExceptionWithFormat:** (NSException Additions),
+ **aggregateExceptionWithExceptions:** (NSException Additions)

validateForUpdate

– (NSException *)**validateForUpdate**

Confirms that the receiver can be updated in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation simply invokes **validateForSave**.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForSave**, – **validateValue:forKey:**,
+ **validationExceptionWithFormat:** (NSException Additions)

validateValue:forKey:

– (NSException *)**validateValue:**(id *)*valuePointer* **forKey:**(NSString *)*key*

Confirms that the value referenced by *valuePointer* is legal for the receiver's property named by *key*. Returns **nil** if it can confirm that the value is legal or an EOValidationException that the sender may raise if it can't. NSObject's implementation sends a **validateValue:forKey:** message to the receiver's EOClassDescription. If that message doesn't return an exception, it checks for a method of the form **validate***Key***:** (for example, **validateBudget:** for a *key* of "budget") and invokes it, returning the result.

Enterprise objects can implement individual **validate***Key***:** methods to check limits, test for nonsense values, and otherwise confirm individual properties. To validate multiple properties based on relations among them, override the appropriate **validateFor...** method.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForSave**, – **validateForUpdate**, + **validationExceptionWithFormat:** (NSException Additions)

valueForKey:

– (id)**valueForKey:**(NSString *)*key*

Returns the value for the property identified by *key*. NSObject's implementation does so by first checking the receiver for a method named *key*, invoking it if there is one. If there's no such method, and **accessInstanceVariablesDirectly** returns YES, NSObject's implementation checks for an instance variable named *key* and returns the instance variable. If there's neither an accessor method nor an instance variable matching *key*, NSObject's implementation invokes **handleQueryWithUnboundKey:** as a fallback mechanism. Subclasses can override **handleQueryWithUnboundKey:** to handle the request in some other way.

See also: – **valueForKeyPath:**, – **valuesForKeys:**, – **takeValue:forKey:**

valueForKeyPath:

– (id)**valueForKeyPath:**(NSString *)*keyPath*

Returns the value for the derived property identified by *keyPath*. A key path has the form *relationship.property* (with one or more relationships); for example "movieRole.roleName" or "movieRole.Talent.lastName". NSObject's implementation of this method gets the destination object for each relationship using **valueForKey:**, and returns the result of a **valueForKey:** message to the final object.

See also: – **valuesForKeys:**, – **takeValue:forKeyPath:**

valuesForKeys:

– (NSDictionary *)**valuesForKeys:**(NSArray *)*keys*

Returns a dictionary containing the property values identified by each of *keys*. NSObject's implementation invokes **valueForKey:** for each key in *keys*, substituting EONull in the dictionary for returned **nil** values.

See also: – **valueForKeyPath:**, – **takeValuesFromDictionary:**

willChange

– (void)**willChange**

Notifies any observers that the receiver’s state is about to change, by sending each an **objectWillChange:** message (see the EOObserverCenter class specification for more information). A subclass should not override this method, but should invoke it prior to altering their state, most typically in “set” methods such as the following:

```
- (void)setRoleName:(NSString *)value {
    [self willChange];
    [roleName autorelease];
    roleName = [value retain];
}
```

EOClassDescriptionClassDelegate
(informal protocol)

Category Of: NSObject

Declared In: EOControl/EOClassDescription.h

Category Description

The EOClassDescriptionClassDelegate informal protocol defines a method that the EOClassDescription class can invoke in its delegate. Delegates are not required to provide an implementation for the method. Instead, declare and implement the method if you need it, and use the EOClassDescription method **setClassDelegate:** method to assign your object as the class delegate. The EOClassDescription class can determine if the delegate doesn't implement the delegate method and only attempts to invoke it if it's actually implemented.

Instance Methods

shouldPropagateDeleteForObject:inEditingContext:forRelationshipKey:

– (BOOL)**shouldPropagateDeleteForObject:**(id)*anObject*
  inEditingContext:(EOEditingContext *)*anEditingContext*
  forRelationshipKey:(NSString *)*key*

Invoked from **propagateDeleteForObject:editingContext:**. If the class delegate returns NO, it prevents *anObject* in *anEditingContext* from propagating deletion to the objects at the destination of *key*. This can be useful if you have a large model and a small application that only deals with a subset of the model's entities. In such a case you might want to disable delete propagation to entities that will never be accessed. You should use this method with caution, however—returning NO and not propagating deletion can lead to dangling references in your object graph.

EOEditingContextDelegate

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOEditingContext.h

Category Description

The EOEditingContextDelegate informal protocol defines methods that an EOEditingContext can invoke in its delegate. Delegates are not required to provide implementations for all of the methods in the informal protocol. Instead, declare and implement any subset of the methods declared in the informal protocol that you need, and use the EOEditingContext method **setDelegate:** method to assign your object as the delegate. An editing context can determine if the delegate doesn't implement a delegate method and only attempts to invoke the methods the delegate actually implements.

Method Types

Fetching objects	
	– editingContext: shouldFetchObjectsDescribedByFetchSpecification:
Invalidating objects	
	– editingContext:shouldInvalidateObject:globalID:
Saving changes	
	– editingContextWillSaveChanges:
Handling failures	
	– editingContextShouldValidateChanges:
	– editingContext:shouldPresentException:
	– editingContextShouldUndoUserActionsAfterFailure:
Merging changes	
	– editingContext:shouldMergeChangesForObject:
	– editingContextDidMergeChanges:

Instance Methods

editingContextDidMergeChanges:

– (void)**editingContextDidMergeChanges:**(EOEditingContext *)*anEditingContext*

Invoked once after a batch of objects has been updated in *anEditingContext*'s parent object store (in response to a EOObjectsChangedInStoreNotification). A delegate might implement this method to define custom merging behavior, most likely in conjunction with **editingContext: shouldMergeChangesForObject:**. It is safe for this method to make changes to the objects in the editing context.

editingContext:shouldFetchObjectsDescribedByFetchSpecification:

– (NSArray *)**editingContext:**(EOEditingContext *)*editingContext*
shouldFetchObjectsDescribedByFetchSpecification:(EOFetchSpecification *)*fetchSpecification*

Invoked from **objectsWithFetchSpecification:editingContext:**. If the delegate has appropriate results cached it can return them and the fetch will be bypassed. Returning **nil** causes the fetch to be propagated to the parent object store.

editingContext:shouldInvalidateObject:globalID:

– (BOOL)**editingContext:**(EOEditingContext *)*anEditingContext*
shouldInvalidateObject:(id)*object*
globalID:(EOGlobalID *)*globalID*

Sent when an *object* identified by *globalID* has been explicitly invalidated. If the delegate returns NO, the invalidation is refused. This allows the delegate to selectively override object invalidations.

See also: **– invalidateAllObjects**, **– revert**

editingContext:shouldMergeChangesForObject:

– (BOOL)**editingContext:**(EOEditingContext *)*anEditingContext*
shouldMergeChangesForObject:(id)*object*

When an EOObjectsChangedInStoreNotification is received, *anEditingContext* invokes this method in its delegate once for each of the objects that has both uncommitted changes and an update from the EOObjectStore. This method is invoked before any updates actually occur.

If this method returns YES, all of the uncommitted changes should be merged into the object after the update is applied, in effect preserving the uncommitted changes (the default behavior). The delegate method **editingContext:shouldInvalidateObject:globalID:** will not be sent for the object in question.

If this method returns NO, no uncommitted changes are applied. Thus, the object is updated to reflect the values from the database exactly. This method should not make any changes to the object since it is about to be invalidated.

If you want to provide custom merging behavior, you need to implement both this method and **editingContextDidMergeChanges:**. You use **editingContext:shouldMergeChangesForObject:** to save information about each changed object and return YES to allow merging to continue. After the default merging behavior occurs, **editingContextDidMergeChanges:** is invoked, at which point you implement your custom behavior.

editingContext:shouldPresentException:

– (BOOL)**editingContext:**(EOEditingContext *)*anEditingContext*
  shouldPresentException:(NSException *)*exception*

Sent whenever an exception is caught by an EOEditingContext. If the delegate returns NO, *exception* is ignored. Otherwise (if the delegate returns YES, if the editing context doesn't have a delegate, or if the delegate doesn't implement this method) *exception* is passed to the message handler for further processing,

See also: – **messageHandler**

editingContextShouldUndoUserActionsAfterFailure:

– (BOOL)**editingContextShouldUndoUserActionsAfterFailure:**(EOEditingContext *)*anEditingContext*

Sent when a validation error occurs while processing a **processRecentChanges** message. If the delegate returns NO, it disables the automatic undoing of user actions after validation has resulted in an error.

By default, if a user attempts to perform an action that results in a validation failure (such as deleting a department object that has a delete rule stating that the department can't be deleted if it contains employees), the user's action is immediately rolled back. However, if this delegate method returns NO, the user action is allowed to stand (though attempting to save the changes to the database without solving the validation error will still result in a failure). Returning NO gives the user an opportunity to correct the validation problem so that the operation can proceed (for example, the user might delete all of the department's employees so that the department itself can be deleted).

editingContextShouldValidateChanges:

– (BOOL)**editingContextShouldValidateChanges:**(EOEditingContext *)*anEditingContext*

Sent when an EOEditingContext receives a **saveChanges** message. If the delegate returns NO, changes are saved without first performing validation. This method can be useful if the delegate wants to provide its own validation mechanism.

editingContextWillSaveChanges:

– (void)**editingContextWillSaveChanges:**(EOEditingContext *)*editingContext*

Sent when an EOEditingContext receives a **saveChanges** message. The delegate can raise an exception to abort the save operation.

EOEditors

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOEditingContext.h

Category Description

The EOEditors informal protocol defines methods for objects that act as higher-level editors of the objects an EOEditingContext contains. An editing context sends messages to its editors to determine whether they have any changes that need to be saved, and to allow them to flush pending changes before a save (possibly raising an exception to abort the save). See the EOEditingContext and EODisplayGroup (EOInterface) class specifications for more information.

Instance Methods

editingContextWillSaveChanges:

– (void)**editingContextWillSaveChanges:**(EOEditingContext *)*anEditingContext*

Invoked by *anEditingContext* in its **saveChanges** method, this method allows the receiver to flush any pending edits and, if necessary, prohibit a save operation. The receiver should validate and flush any unprocessed edits it has, raising an exception if it can't do so to prevent *anEditingContext* from saving.

editorHasChangesForEditingContext:

– (BOOL)**editorHasChangesForEditingContext:**(EOEditingContext *)*anEditingContext*

Invoked by *anEditingContext*, this method should return YES if the receiver has any unapplied edits that need to be saved, NO if it doesn't.

EOEnterpriseObject

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOClassDescription.h EOControl/EOEditingContext.h EOControl/EOKeyValueCoding.h EOControl/EOObserver.h

Protocol Description

The EOEnterpriseObject informal protocol identifies basic enterprise object behavior, defining methods for supporting operations common to all enterprise objects. Among these are methods for initializing instances, announcing changes, setting and retrieving property values, and performing validation of state. Some of these methods are for enterprise objects to implement or override, and some are meant to be used as defined by the Framework. Many methods are used internally by the Framework and rarely invoked by application code.

Many of the functional areas are defined in smaller, more specialized informal protocols and incorporated in the overarching EOEnterpriseObject informal protocol:

- EOKeyValueCoding defines Enterprise Objects Framework's main data transport mechanism, in which the properties of an object are accessed indirectly by name (or *key*), rather than directly through invocation of an accessor method or as instance variables.
- EOKeyValueCodingAdditions defines extensions to the basic EOKeyValueCoding informal protocol, giving access to groups of properties and to properties across relationships.
- EORelationshipManipulation builds on the basic EOKeyValueCoding informal protocol to allow you to modify to-many relationship properties.
- EOValidation defines the way that enterprise objects validate their values.

The remaining methods are introduced in the EOEnterpriseObject informal protocol itself and can be broken down into three functional groups discussed in the following sections:

- Initialization
- Change Notification
- Object and Class Metadata Access
- Snapshots

You rarely need to implement the EOEnterpriseObject informal protocol from scratch. The Framework provides default implementations of the methods in categories on NSObject. Use EOGenericRecords to represent enterprise objects that don't require custom behavior, and create subclasses of NSObject to represent enterprise objects that do. The section "Writing an Enterprise Object Class" highlights the methods that you typically provide or override in a custom enterprise object class.

Informal Protocols Incorporated

EOKeyValueCoding	
	+ accessInstanceVariablesDirectly
	+ flushAllKeyBindings
	+ useStoredAccessor
	– handleQueryWithUnboundKey:
	– handleTakeValue:forUnboundKey:
	– storedValueForKey:
	– takeStoredValue:forKey:
	– takeValue:forKey:
	– unableToSetNullForKey:
	– valueForKey:
EOKeyValueCodingAdditions	
	– takeValue:forKeyPath:
	– takeValuesFromDictionary:
	– valueForKeyPath:
	– valuesForKeys:
EORelationshipManipulation	
	– addObject:toBothSidesOfRelationshipWithKey:
	– addObject:toPropertyWithKey:
	– removeObject:fromBothSidesOfRelationshipWithKey:
	– removeObject:fromPropertyWithKey:
EOValidation	
	– validateForDelete
	– validateForInsert
	– validateForSave
	– validateForUpdate
	– validateValue:forKey:

Method Types

Initializing enterprise objects	
	– initWithEditingContext:classDescription:globalID:
	– awakeFromFetchInEditingContext:
	– awakeFromInsertionInEditingContext:
Announcing changes	
	– willChange
Getting an object’s EOEditingContext	
	– editingContext

Getting class description information	
	– allPropertyKeys
	– attributeKeys
	– classDescription
	– classDescriptionForDestinationKey:
	– deleteRuleForRelationshipKey:
	– entityName
	– inverseForRelationshipKey:
	– isToManyKey:
	– ownsDestinationObjectsForRelationshipKey:
	– toManyRelationshipKeys
	– toOneRelationshipKeys
Modifying relationships	
	– propagateDeleteWithEditingContext:
	– clearProperties
Working with snapshots	
	– snapshot
	– updateFromSnapshot:
Merging values	
	– changesFromSnapshot
	– reapplyChangesFromDictionary:
Getting descriptions	
	– eoDescription
	– eoShallowDescription
	– userPresentableDescription

Instance Methods

allPropertyKeys

– (NSArray *)**allPropertyKeys**

Returns all of the receiver’s property keys. NSObject’s implementation returns the union of the keys returned by **attributeKeys**, **toOneRelationshipKeys**, and **toManyRelationshipKeys**.

attributeKeys

– (NSArray *)**attributeKeys**

Returns the names of the receiver's attributes (not relationship properties). NSObject's implementation simply invokes **attributeKeys** in the object's EOClassDescription and returns the results. You might wish to override this method to add keys for attributes not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of attributes designated as class properties.

See also: – **toOneRelationshipKeys**, – **toManyRelationshipKeys**

awakeFromFetchInEditingContext:

– (void)**awakeFromFetchInEditingContext:**(EOEditingContext *)*anEditingContext*

Overridden by subclasses to perform additional initialization on the receiver upon its being fetched from the external repository into *anEditingContext*. NSObject's implementation merely sends an **awakeObject:fromFetchInEditingContext:** to the receiver's EOClassDescription. Subclasses should invoke **super**'s implementation before performing their own initialization.

awakeFromInsertionInEditingContext:

– (void)**awakeFromInsertionInEditingContext:**(EOEditingContext *)*anEditingContext*

Overridden by subclasses to perform additional initialization on the receiver upon its being inserted into *anEditingContext*. This is commonly used to assign default values or record the time of insertion. NSObject's implementation merely sends an **awakeObject:fromInsertionInEditingContext:** to the receiver's EOClassDescription. Subclasses should invoke **super**'s implementation before performing their own initialization.

changesFromSnapshot

– (NSDictionary *)**changesFromSnapshot:**(NSDictionary *)*snapshot*

Returns a dictionary whose keys correspond to the receiver's properties with uncommitted changes relative to *snapshot*, and whose values are the uncommitted values. In both *snapshot* and the returned dictionary, where a key represents a to-many relationship, the corresponding value is an NSArray containing two other NSArrays: the first is an array of objects to be added to the relationship property, and the second is an array of objects to be removed.

See also: – **reapplyChangesFromDictionary:**

classDescription

– (EOClassDescription *)**classDescription**

Returns the EOClassDescription registered for the receiver’s class.NSObject’s implementation invokes the EOClassDescription class method a **classDescriptionForClass:**.

classDescriptionForDestinationKey:

– (EOClassDescription *)**classDescriptionForDestinationKey:**(NSString *)*key*

Returns the EOClassDescription for the destination objects of the relationship identified by *key*. NSObject’s implementation sends a **classDescriptionForDestinationKey:** message to the receiver’s EOClassDescription.

clearProperties

– (void)**clearProperties**

Sets all of the receiver’s to-one and to-many relationships to **nil**. EOEditingContexts use this method to break cyclic references among objects when they’re deallocated. NSObject’s implementation should be sufficient for all purposes. If your enterprise object maintains references to other objects and these references are not to-one or to-many keys, then you should probably subclass this method ensure unused objects can be deallocated.

deleteRuleForRelationshipKey:

– (EODeleteRule)**deleteRuleForRelationshipKey:**(NSString *)*relationshipKey*

Returns a rule indicating how to handle the destination of the receiver’s relationship named by *relationshipKey* when the receiver is deleted. The delete rule is one of:

- EODeleteRuleNullify
- EODeleteRuleNullify
- EODeleteRuleNullify
- EODeleteRuleNullify

For example, an Invoice object might return EODeleteRuleNullify for the relationship named “lineItems”, since when an invoice is deleted, its line items should be deleted as well. For more information on the delete rules, see the method description for EOClassDescription’s **deleteRuleForRelationshipKey:** in the class specification for EOClassDescription.

NSObject’s implementation of this method simply sends a **deleteRuleForRelationshipKey:** message to the receiver’s EOClassDescription.

See also: – **propagateDeleteWithEditingContext:**, – **validateForDelete** (EOValidation)

editingContext

– (EOEditingContext *)**editingContext**

Returns the EOEditingContext that holds the receiver.

entityName

– (NSString *)**entityName**

Returns the name of the receiver's entity, or **nil** if it doesn't have one. NSObject's implementation simply sends an **entityName** message to the receiver's EOClassDescription.

eoDescription

– (NSString *)**eoDescription**

Returns a string that describes the receiver. NSObject's implementation returns a full description of the receiver's property values by extracting them using the key-value coding methods. An object referenced through relationships is listed with the results of an **eoShallowDescription** message (to avoid infinite recursion through cyclical relationships).

This method is useful for debugging. You can implement a **description** method that invokes this one, and the debugger's print-object command (**po** on the command line) automatically displays this description. You can also invoke this method directly on the command line of the debugger.

See also: – **userPresentableDescription**

eoShallowDescription

– (NSString *)**eoShallowDescription**

Similar to **eoDescription**, but doesn't descend into relationships. **eoDescription** invokes this method for relationship destinations to avoid infinite recursion through cyclical relationships. NSObject's implementation simply returns a string containing the receiver's class and entity names, along with the memory address of its **id**.

See also: – **userPresentableDescription**

initWithEditingContext:classDescription:globalID:

– **initWithEditingContext:**(EOEditingContext *)*anEditingContext*
  classDescription:(EOClassDescription *)*aClassDescription*
  globalID:(EOGlobalID *)*globalID*

Initializes the receiver with the arguments provided. NSObject's implementation simply invokes **init**, and ingores *anEditingContext*.

See also: – **createInstanceWithEditingContext:globalID:zone:** (EOClassDescription)

inverseForRelationshipKey:

– (NSString *)**inverseForRelationshipKey:**(NSString *)*relationshipKey*

Returns the name of the relationship pointing back to the receiver's class or entity from that named by *relationshipKey*, or **nil** if there isn't one. With the access layer's EOEntity and EORelationship, for example, reciprocality is determined by the join attributes of the two EORelationships. NSObject's implementation simply sends an **inverseForRelationshipKey:** message to the receiver's EOClassDescription.

You might override this method for reciprocal relationships that aren't defined using the same join attributes. For example, if a Member object has a relationship to CreditCard based on the card number, but a CreditCard has a relationship to Member based on the Member's primary key, both classes need to override this method. This is how Member might implement it:

```
public String inverseForRelationshipKey(java.lang.String relationshipKey) {
    if (relationshipKey.equals("creditCard"))
- (NSString *)inverseForRelationshipKey:(NSString *)relationshipKey
{
    if ([relationshipKey isEqual:@"creditCard"]) return @"member";
    return [super inverseForRelationshipKey:relationshipKey];
}
```

isToManyKey:

– (BOOL)**isToManyKey:**(NSString *)*key*

Returns YES if the receiver has a to-many relationship identified by *key*, NO otherwise. NSObject's implementation of this method simply checks its **toManyRelationshipKeys** array for *key*.

ownsDestinationObjectsForRelationshipKey:

– (BOOL)**ownsDestinationObjectsForRelationshipKey:**(NSString *)*key*

Returns YES if the receiver has a relationship identified by *key* that owns its destination, NO otherwise. If an object owns the destination for a relationship, then when that destination object is removed from the relationship, it’s automatically deleted. Ownership of a relationship thus contrasts with a delete rule, in that the first applies when the destination is removed and the second applies when the source is deleted. NSObject’s implementation of this method simply sends an **ownsDestinationObjectsForRelationshipKey:** message to the receiver’s EOClassDescription.

See also: – **deleteRuleForRelationshipKey:**, – **ownsDestination** (the access layer’s EORelationship)

propagateDeleteWithEditingContext:

– (void)**propagateDeleteWithEditingContext:**(EOEditingContext *)*anEditingContext*

Deletes the destination objects of the receiver’s relationships according to the delete rule for each relationship. NSObject’s implementation simply sends a **propagateDeleteForObject:editingContext:** message to the receiver’s EOClassDescription. For more information on delete rules, see the method description for **deleteRuleForRelationshipKey:** in the EOClassDescription class specification.

See also: – **deleteRuleForRelationshipKey:**

reapplyChangesFromDictionary:

– (void)**reapplyChangesFromDictionary:**(NSDictionary *)*changes*

Similar to **takeValuesFromDictionary:**, but the *changes* dictionary can contain arrays for to-many relationships. Where a key represents a to-many relationship, the dictionary’s value is an NSArray containing two other NSArrays: the first is an array of objects to be added to the relationship property, and the second is an array of objects to be removed. NSObject’s implementation should be sufficient for all purposes; you shouldn’t have to override this method.

See also: – **changesFromSnapshot**

snapshot

– (NSDictionary *)**snapshot**

Returns a dictionary whose keys are those of the receiver’s attributes, to-one relationships, and to-many relationships, and whose values are the values of those properties, with EONull substituted for **nil**. For to-many relationships, the dictionary contains shallow copies of the arrays to preserve the **id**s of the contents. NSObject’s implementation should be sufficient for all purposes; you shouldn’t have to override this method.

See also: – **updateFromSnapshot:**

toManyRelationshipKeys

– (NSArray *)**toManyRelationshipKeys**

Returns the names of the receiver's to-many relationships. NSObject's implementation simply invokes **toManyRelationshipKeys** in the object's EOClassDescription and returns the results. You might wish to override this method to add keys for relationships not defined by the EOClassDescription, but it's rarely necessary: The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-many relationships designated as class properties.

See also: – **attributeKeys**, – **toOneRelationshipKeys**

toOneRelationshipKeys

– (NSArray *)**toOneRelationshipKeys**

Returns the names of the receiver's to-one relationships. NSObject's implementation simply invokes **toOneRelationshipKeys** in the object's EOClassDescription and returns the results. You might wish to override this method to add keys for relationships not defined by the EOClassDescription, but it's rarely necessary: The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-one relationships designated as class properties.

See also: – **attributeKeys**, – **toManyRelationshipKeys**

updateFromSnapshot:

– (void)**updateFromSnapshot:**(NSDictionary *)*aSnapshot*

Takes the values from *aSnapshot*, and sets the receiver's properties to them. NSObject's implementation sets each one using **takeStoredValue:forKey:**. In the process, EONull values are converted to **nil**, and array values are set as shallow mutable copies to preserve the **id**s of the contents.

See also: – **snapshot**

userPresentableDescription

– (NSString *)**userPresentableDescription**

Returns a short (no longer than 60 characters) description of an enterprise object based on its data. NSObject's implementation enumerates the object's **attributeKeys** and returns the values of all of its properties, separated by commas (applying the default formatter for numbers and dates).

See also: – **eoDescription**, – **eoShallowDescription**

willChange

– (void)**willChange**

Notifies any observers that the receiver's state is about to change, by sending each an **objectWillChange:** message (see the EOObserverCenter class specification for more information). A subclass should not override this method, but should invoke it prior to altering the subclass's state, most typically in "set" methods such as the following:

```
- (void)setRoleName:(NSString *)value {
    [self willChange];
    [roleName autorelease];
    roleName = [value retain];
}
```

EOEnterpriseObject

Initialization

Enterprise objects are initialized using **initWithEditingContext:classDescription:globalID:**, which by default simply invokes **init**. You can place your custom initialization code in **init**, or you can override **initWithEditingContext:classDescription:globalID:** to take advantage of the extra information available with this method.

After an enterprise object is created, it receives an **awake...** message. The particular message depends on whether the object has been fetched from a database or created anew in the application. In the former case, it receives an **awakeFromFetchInEditingContext:** message. In the latter, it receives an **awakeFromInsertionInEditingContext:** message. The receiver can override either method to perform extra initialization—such as setting default values—based on how it was created.

Change Notification

For the Framework to keep all areas of an application synchronized, enterprise objects must notify their observers when their state changes. Objects do this by invoking **willChange** before altering any instance variable or other kind of state. This method informs all observers that the invoker is about to change. See the EOObserverCenter class specification for more information on change notification.

The primary observer of changes in an object is the object's EOEditingContext. EOEditingContext is a subclass of EOObjectStore that manages collections of objects in memory, tracking inserts, deletes, and updates, and propagating changes to the persistent store as needed. You can get the EOEditingContext that contains an object by sending the object an **editingContext** message.

Object and Class Metadata Access

One of the larger groups of methods in the EOEnterpriseObject interface provides information about an object's properties. Most of these methods consult an EOClassDescription to provide their answers. An object's **classDescription** method returns it's class description. See the EOClassDescription class specification for the methods it implements. Methods you can send directly to an enterprise object include **entityName**, which provides the name of the entity mapped to the receiver's class; **allPropertyKeys**, which returns the names of all the receiver's class properties, attributes and relationships alike; and **attributeKeys**, which returns just the names of the attributes.

Some methods return information about relationships. **toOneRelationshipKeys** and **toManyRelationshipKeys** return the names of the receiver's relationships, while **isToManyKey:** tells which kind a particular relationship is. **deleteRuleForRelationshipKey:** indicates what should happen to the receiver's relationships when it's deleted. Similarly, **ownsDestinationObjectsForRelationshipKey:** indicates what should happen when another object is added to or removed from the receiver's relationship. Another method, **classDescriptionForDestinationKey:**, returns the EOClassDescription for the objects at the destination of a relationship.

Snapshots

The key-value coding methods define a general mechanism for accessing an object's properties, but you first have to know what those properties are. Sometimes, however, the Framework needs to preserve an object's entire state for later use, whether to undo changes to the object, compare the values that have changed, or just keep a record of the changes. The snapshotting methods provide this service, extracting or setting all properties at once and performing the necessary conversions for proper behavior. **snapshot** returns a dictionary containing all the receiver's properties, and **updateFromSnapshot:** sets properties of the receiver to the values in a snapshot.

A special kind of snapshot is also used to merge an object's uncommitted changes with changes that have been committed to the external store since the object was fetched. These methods are **changesFromSnapshot** and **reapplyChangesFromDictionary:**.

Writing an Enterprise Object Class

Some of the EOEnterpriseObject methods are for enterprise objects to implement or override, and some are meant to be used as defined by the Framework. Many methods are used internally by the Framework and rarely invoked by application code. The tables in this section highlight the methods that you typically override or implement in a custom enterprise object.

Creation

– init	Designated initializer.
– initWithEditingContext:classDescription:globalID:	Optional initializer.
– awakeFromFetchInEditingContext:	Performs additional initialization after the object is fetched.
– awakeFromInsertionInEditingContext:	Performs additional initialization after the object is created in memory.

Key-Value Coding: Accessing Properties and Relationships

– set*Key*:	Sets the value for the property named *key*.
– *key*	Retrieves the value for the property named *key*.
– addTo*Key*:	Adds an object to a relationship property named *key*.
– removeFrom*Key*:	Removes an object from the property named *key*.
– handleTakeValue:forUnboundKey:	Handles a failure of **takeValue:forKey:** to find a property.

Key-Value Coding: Accessing Properties and Relationships

– handleQueryWithUnboundKey:	Handles a failure of **valueForKey:** to find a property.
– unableToSetNullForKey:	Handles an attempt to set a non-object property's value to **nil**.

Validation

– validate*Key*:	Validates a value for the property named *key*.
– validateForDelete	Validates all properties before deleting the receiver.
– validateForInsert	Validates all properties before inserting the receiver.
– validateForSave	Validates all properties before saving the receiver.
– validateForUpdate	Validates all properties before updating the receiver.

EOKeyValueCoding

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOKeyValueCoding.h

Protocol Description

The EOKeyValueCoding informal protocol defines Enterprise Objects Framework's main data transport mechanism, in which the properties of an object are accessed indirectly by name (or *key*), rather than directly through invocation of an accessor method or as instance variables. Thus, all of an object's properties can be accessed in a consistent manner. the Framework additions to NSObject provide default implementations of EOKeyValueCoding, which are sufficient for most purposes.

The basic methods for accessing an object's values are **takeValue:forKey:**, which sets the value for the property identified by the specified key, and **valueForKey:**, which returns the value for the property identified by the specified key. The default implementations provided by NSObject use the accessor methods normally implemented by objects (or to access instance variables directly if need be), so that you don't have to write special code simply to integrate your objects into the Enterprise Objects Framework.

The corresponding methods **takeStoredValue:forKey:** and **storedValueForKey:** are similar, but they're considered to be a private API, for use by the Framework for transporting data to and from *trusted* sources. For example, **takeStoredValue:forKey:** is used to initialize an object's properties with values fetched from the database, whereas **takeValue:forKey:** is used to modify an object's properties to values provided by a user or other business logic. How these methods work and how they're used by the framework is discussed in more detail in the section "Stored Value Methods."

Both the basic and stored value key-value coding methods cache attribute bindings for both accessor methods and instance variables, making lookups efficient. The method **flushAllKeyBindings** is provided to clear these bindings—as you should when you add or modify a class in the run-time system.

The the methods **accessInstanceVariablesDirectly** and **useStoredAccessor** are used by enterprise object classes to modify the behavior of the default implementations of key-value coding methods. The remaining methods, **handleQueryWithUnboundKey:**, **handleTakeValue:forUnboundKey:**, and **unableToSetNullForKey:**, are provided to handle error conditions. The default versions of **handleQueryWithUnboundKey:** and **handleTakeValue:forUnboundKey:** raise EOUnknownKeyException, with the target object (*EOTargetObjectUserInfoKey*) and key (*EOUnknownUserInfoKey*) in the user info.

For more information on EOKeyValueCoding, see the sections:

- Stored Value Methods
- Type Checking and Type Conversion

Method Types

Accessing values	– storedValueForKey: – takeStoredValue:forKey: – takeValue:forKey: – valueForKey:
Changing default behavior	+ accessInstanceVariablesDirectly + useStoredAccessor
Flushing key bindings	+ flushAllKeyBindings
Handling error conditions	– handleQueryWithUnboundKey: – handleTakeValue:forUnboundKey: – unableToSetNullForKey:

Class Methods

accessInstanceVariablesDirectly

+ (BOOL)**accessInstanceVariablesDirectly**

Returns YES if the key-value coding methods should access the corresponding instance variable directly on finding no accessor method for a property. Returns NO if they shouldn't. NSObject's implementation of this method returns YES. Subclasses can override it to return NO, in which case the key-value coding methods won't access instance variables.

flushAllKeyBindings

+ (void)**flushAllKeyBindings**

Invalidates the cached key binding information for all classes (caches are kept of key-to-method or instance variable bindings in order to make key-value coding efficient). This method should be invoked whenever a class is modified in or removed from the run-time system.

useStoredAccessor

+ (BOOL)**useStoredAccessor**

Returns YES if the stored value methods (**storedValueForKey:** and **takeStoredValue:forKey:**) should use private accessor methods in preference to public accessors. Returning NO causes the stored value methods to use the same accessor method-instance variable search order as the corresponding basic key-value coding methods (**valueForKey:** and **takeValue:forKey:**). NSObject's implementation of this method returns YES.

Instance Methods

handleQueryWithUnboundKey:

– (id)**handleQueryWithUnboundKey:**(NSString *)*key*

Invoked from **valueForKey:** when it finds no property binding for *key*. NSObject's implementation raises an EOUnknownKeyException, with the target object (*EOTargetObjectUserInfoKey*) and key (*EOUnknownUserInfoKey*) in the user info. Subclasses can override this method to handle the query in some other way.

handleTakeValue:forUnboundKey:

– (void)**handleTakeValue:**(id)*value*
forUnboundKey:(NSString *)*key*

Invoked from **takeValue:forKey:** when it finds no property binding for *key*. NSObject's implementation raises an EOUnknownKeyException, with the target object (*EOTargetObjectUserInfoKey*) and key (*EOUnknownUserInfoKey*) in the user info. Subclasses can override it to handle the request in some other way.

storedValueForKey:

– (id)**storedValueForKey:**(NSString *)*key*

Returns the property identified by *key*. This method is used when the value is retrieved for storage in an object store (generally, this is ultimately in a database) or for inclusion in a snapshot. The default implementation provided by the Framework additions to NSObject is similar to the implementation of **valueForKey:**, but it resolves *key* with a different method-instance variable search order:

1. Searches for a private accessor method based on *key* (a method preceded by an underbar). For example, with a key of "lastName", **storedValueForKey:** looks for a method named **_getLastName** or **_lastName.**
2. If a private accessor isn't found, searches for an instance variable based on *key* and returns its value directly. For example, with a key of "lastName", **storedValueForKey:** looks for an instance variable named **_lastName** or **lastName**.
3. If neither a private accessor or an instance variable is found, **storedValueForKey:** searches for a public accessor method based on *key*. For the key "lastName", this would be **getLastName** or **lastName**.
4. If *key* is unknown, **storedValueForKey:** calls **handleTakeValue:forUnboundKey:**.

This different search order allows an object to bypass processing that is performed before returning a value through public API. However, if you always want to use the search order in **valueForKey:**, you can implement the class method **useStoredAccessor** to return NO. And as with **valueForKey:**, you can prevent direct access of an instance variable with the method the class method **accessInstanceVariablesDirectly**.

takeStoredValue:forKey:

– (void)**takeStoredValue:**(id)*value*
  forKey:(NSString *)*key*

Sets the property identified by *key* to *value*. This method is used to initialize the receiver with values from an object store (generally, this is ultimately from a database) or to restore a value from a snapshot. The default implementation provided by the Framework additions to NSObject is similar to the implementation of **takeValue:forKey:**, but it resolves *key* with a different method-instance variable search order:

1. Searches for a private accessor method based on *key* (a method preceded by an underbar). For example, with a key of "lastName", **takeStoredValue:forKey:** looks for a method named **_setLastName**:**.**
2. If a private accessor isn't found, searches for an instance variable based on *key* and and sets its value directly. For example, with a key of "lastName", **takeStoredValue:forKey:** looks for an instance variable named **_lastName** or **lastName**.
3. If neither a private accessor or an instance variable is found, **takeStoredValue:forKey:** searches for a public accessor method based on *key*. For the key "lastName", this would be **setLastName**:.
4. If *key* is unknown, **storedValueForKey:** calls **handleTakeValue:forUnboundKey:**.

This different search order allows an object to bypass processing that is performed before setting a value through public API. However, if you always want to use the search order in **takeValue:forKey:**, you can implement the class method **useStoredAccessor** to return NO. And as with **valueForKey:**, you can prevent direct access of an instance variable with the method the class method **accessInstanceVariablesDirectly**.

takeValue:forKey:

– (void)**takeValue:**(id)*value*
  forKey:(NSString *)*key*

Sets the value for the property identified by *key* to *value*, invoking **handleTakeValue:forUnboundKey:** if the receiver doesn't recognize *key* and **unableToSetNullForKey:** if *value* is **nil** and *key* identifies a scalar property.

The default implementation provided by the Framework additions to NSObject works as follows:

1. Searches for a public accessor method of the form **set***Key***:**, invoking it if there is one.
2. If a public accessor method isn't found, searches for a private accessor method of the form **_set***Key***:**, invoking it if there is one.

3. If an accesor method isn't found and the class method **accessInstanceVariablesDirectly** returns YES, **takeValue:forKey:** searches for an instance variable based on *key* and sets the value directly, autoreleasing the old value and retaining the new one. For the key "lastName", this would be **_lastName** or **lastName**.
4. If neither an accessor method nor an instance variable is found, the default implementation invokes **handleTakeValue:forUnboundKey:**.

unableToSetNullForKey:

– (void)**unableToSetNilForKey:**(NSString *)*key*

Invoked from **takeValue:forKey:** (and **takeStoredValue:forKey:**) when it's given a **nil** value for a scalar property (such as an **int** or a **float**). NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the request in some other way, such as by substituting zero or a sentinel value and invoking **takeValue:forKey:** again.

valueForKey:

– (id)**valueForKey:**(NSString *)*key*

Returns the value for the property identified by *key*, invoking **handleQueryWithUnboundKey:** if the receiver doesn't recognize *key.*

The default implementation provided by the Framework additions to NSObject works as follows:

1. Searches for a public accessor method based on *key*. For example, with a key of "lastName", **valueForKey:** looks for a method named **getLastName** or **lastName**.
2. If a public accessor method isn't found, searches for a private accessor method based on *key* (a method preceded by an underbar). For example, with a key of "lastName", **valueForKey:** looks for a method named **_getLastName** or **_lastName.**
3. If an accesor method isn't found and the class method **accessInstanceVariablesDirectly** returns YES, **valueForKey:** searches for an instance variable based on *key* and returns its value directly. For the key "lastName", this would be **_lastName** or **lastName**.
4. If neither an accessor method nor an instance variable is found, the default implementation invokes **handleQueryWithUnboundKey:**.

EOKeyValueCoding

Stored Value Methods

The stored value methods, **storedValueForKey:** and **takeStoredValue:forKey:**, are used by the framework to store and restore an enterprise object's properties, either from the database or from an in-memory snapshot. This access is considered private to the enterprise object and is invoked by the framework to effect persistence on the object's behalf.

On the other hand, the basic key-value coding methods, **valueForKey:** and **takeValue:forKey:**, are the public API to an enterprise object. They are invoked by clients external to the object, such as for interactions with the user interface or with other enterprise objects.

All of the key-value coding methods access an object's properties by invoking property-specific accessor methods or by directly accessing instance variables. The basic methods resolve the specified property key as follows:

1. Search for a public accessor method based on the specified key, invoking it if there is one. For example, with a key of "lastName", **takeValue:forKey:** looks for a method named **set***Key***:**, and **valueForKey:** looks for a method named **getLastName** or **lastName**.
2. If a public accessor method isn't found and **useStoredAccessor** returns YES, the basic methods search for a private accessor method based on the key. For example, with a key of "lastName", **takeValue:forKey:** looks for a method named **_set***Key***:**, and **valueForKey:** looks for a method named **_getLastName** or **_lastName**.
3. If an accesor method isn't found, the basic methods search for an instance variable based on the key and set the value directly. For the key "lastName", this would be **_lastName** or **lastName**. Note that **_lastName** is used only if **useStoredAccessor** returns YES.

The stored value methods use a different search order for resolving the property key: they search for a private accessor first, then for an instance variable, and finally for a public accessor. Enterprise object classes can take advantage of this distinction to simply set or get values when properties are accessed through the private API (on behalf of a trusted source) and to perform additional processing when properties are accessed through the public API. Put another way, the stored value methods allow you bypass the logic in your public accessor methods, whereas the basic key-value coding methods execute that logic.

The stored value methods are especially useful in cases where property values are interdependent. For example, suppose you need to update a total whenever an object's **bonus** property is set:

```
- (void)setBonus:(double)newBonus {
    [self willChange];
    _total += (newBonus - _bonus);
    _bonus = newBonus;
}
```

This total-updating code should be activated when the object is updated with values provided by a user (through the user interface), but not when the **bonus** property is restored from the database. Since the

Framework restores the property using **takeStoredValue:forKey:** and since this method accesses the **_bonus** instance variable in preference to calling the public accessor, the unnecessary (and possibly harmful) recomputation of **_total** is avoided. If the object actually wants to intervene when a property is set from the database, it has two options:

- Implement **_setBonus:.**
- Replace the Framework's default stored value search order with the same search order used by the basic methods by overriding the class method **useStoredAccessor** to return NO.

Type Checking and Type Conversion

The default implementations of the key-value coding methods accept any object as a value, and do no type checking or type conversion among object classes. It's possible, for example, to pass an NSString to **takeValue:forKey:** as the value for a property the receiver expects to be an NSDate. The sender of a key-value coding message is thus responsible for ensuring that a value is of the proper class, typically by using the **validateValue:forKey:** method to coerce it to the proper type. The interface layer's EODisplayGroup uses this on all values received from interface user objects, for example, as well as relying on number and date formatters to interpret string values typed by the user. For more information on the **validateValue:forKey:** method, see the EOValidation informal protocol specification.

The key-value coding methods handle one special case with regard to value types. For enterprise objects that access numeric values as C scalar types, these methods automatically convert between the scalar types and NSNumber objects. For example, suppose your enterprise object defines these accessor methods:

– (void)**setSalary:**(unsigned int)*salary*
– (unsigned int)**salary**

For the **setSalary:** method, **takeValue:forKey:** converts the object value it receives as the argument for the "salary" key to an **unsigned int** and passes it as *salary* to **setSalary:**. Similarly, **valueForKey:** converts the return value of the **salary** method to an NSNumber and returns that.

The default implementations of the key-value coding methods support the following scalar types:

char	unsigned char
short	unsigned short
int	unsigned int
long	unsigned long
float	double

Object values are converted to these types with the standard messages **charValue**, **intValue**, **floatValue**, and so on. Note that the key-value coding methods don't check that an object value actually responds to these messages; this can result in a run-time error if the object doesn't respond to the appropriate message.

One type of conversion these methods can't perform is that from **nil** to a scalar value. C scalar values define no equivalent of a database system's NULL value, so these must be handled by the object itself. Upon encountering **nil** while setting a scalar value, **takeValue:forKey:** invokes **unableToSetNullForKey:**, which by default simply raises an exception. Enterprise object classes that use scalar values which may be NULL in the database should override this method to substitute the appropriate scalar value for **nil**, reinvoking **takeValue:forKey:** to set the substitute value.

EOKeyValueCodingAdditions

(informal protocol)

Category Of: NSObject

Declared In: EOControl/EOKeyValueCoding.h

Protocol Description

The EOKeyValueCodingAdditions informal protocol defines extensions to the basic EOKeyValueCoding informal protocol. One pair of methods, **takeValuesFromDictionary:** and **valuesForKeys:**, gives access to groups of properties. Another pair of methods, **takeValue:forKeyPath:** and **valueForKeyPath:** give access to properties across relationships with key paths of the form *relationship.property*; for example, “department.name”. the Framework additions to NSObject provide default implementations of EOKeyValueCodingAdditions, which you rarely (if ever) need to override.

EONull in Collections

Because collection objects such as NSArray and NSDictionary can’t contain **nil** as a value, **nil** must be represented by a special object, EONull. EONull provides a single instance that represents the NULL value for object attributes. The default implementations of **takeValuesFromDictionary:** and **valuesForKeys:** translate EONull and **nil** between NSDictionaries and enterprise objects so your objects don’t have to explicitly test for EONull values.

Instance Methods

takeValue:forKeyPath:

– (void)**takeValue:**(id)*value*
  forKeyPath:(NSString *)*keyPath*

Sets the value for the property identified by *keyPath* to *value*. A key path has the form *relationship.property* (with one or more relationships); for example “movieRole.roleName” or “movieRole.Talent.lastName”. NSObject’s implementation of this method gets the destination object for each relationship using **valueForKey:**, and sends the final object a **takeValue:forKey:**message with *value* and *property.*

takeValuesFromDictionary:

– (void)**takeValuesFromDictionary:**(NSDictionary *)*aDictionary*

Sets properties of the receiver with values from *aDictionary*, using its keys to identify the properties. NSObject's implementation invokes **takeValue:forKey:** for each key-value pair, substituting **nil** for EONull values in *aDictionary.*

valueForKeyPath:

– (id)**valueForKeyPath:**(NSString *)*keyPath*

Returns the value for the derived property identified by *keyPath*. A key path has the form *relationship.property* (with one or more relationships); for example "movieRole.roleName" or "movieRole.Talent.lastName". NSObject's implementation of this method gets the destination object for each relationship using **valueForKey:**, and returns the result of a **valueForKey:** message to the final object.

valuesForKeys:

– (NSDictionary *)**valuesForKeys:**(NSArray *)*keys*

Returns a dictionary containing the property values identified by each of *keys*. NSObject's implementation invokes **valueForKey:** for each key in *keys*, substituting EONull values in the dictionary for returned **nil** values.

EOMessageHandlers

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOEditingContext.h

Category Description

The EOMessageHandlers informal protocol declares methods used for error reporting and determining fetch limits. See the EOEditingContext, EODatabaseContext (EOAccess), and EODisplayGroup (EOInterface) class specifications for more information.

Message handlers are primarily used to implement exception handling in the interface layer's EODisplayGroup, and wouldn't ordinarily be used in a command line tool or WebObjects application.

Instance Methods

editingContext:presentErrorMessage:

– (void)**editingContext:**(EOEditingContext *)*anEditingContext*
    presentErrorMessage:(NSString *)*message*

Invoked by *anEditingContext*, this method should present *message* to the user in whatever way is appropriate (whether by opening an attention panel or printing the message in a terminal window, for example). This message is sent only if the method is implemented.

editingContext: shouldContinueFetchingWithCurrentObjectCount:originalLimit: objectStore:

– (BOOL)**editingContext:**(EOEditingContext *)*anEditingContext*
    shouldContinueFetchingWithCurrentObjectCount:(unsigned)*count*
    originalLimit:(unsigned)*limit*
    objectStore:(EOObjectStore *)*objectStore*

Invoked by an *objectStore* (such as an access layer EODatabaseContext) to allow the message handler for *anEditingContext* (often an interface layer EODisplayGroup) to prompt the user about whether or not to continue fetching the current result set. The *count* argument is the number of objects fetched so far. *limit* is the original limit specified an EOFetchSpecification. This message is sent only if the method is implemented.

EOObserving

Adopted By:	EODelayedObserver EOEditingContext
Declared In:	EOControl/EOObserver.h

Protocol Description

The EOObserving protocol,a part of EOControl's change tracking mechanism, declares the **objectWillChange:** method, used by observers to receive notifications that an object has changed. This message is sent by EOObserverCenter to all observers registered using its **addObserver:forObject:** method. For an overview of the general change tracking mechanism, see "Tracking Enterprise Objects Changes" in the introduction to the EOControl Framework. The EOObserving protocol

Instance Methods

objectWillChange:

– (void)**objectWillChange:**(id)*anObject*

Informs the receiver that *anObject*'s state is about to change. The receiver can record *anObject*'s state, mark or record it as changed, and examine it later (such as at the end of the run loop) to see how it's changed.

EOQualifierEvaluation

Adopted By:

EOKeyValueQualifier
EOKeyComparisonQualifier
EOAndQualifier
EOOrQualifier
EONotQualifier

Protocol Description

The EOQualifierEvaluation protocol defines a method, **evaluateWithObject:**, that performs in-memory evaluation of qualifiers. All qualifier classes whose objects can be evaluated in memory must implement this protocol.

Instance Methods

evaluateWithObject:

– (BOOL)**evaluateWithObject:***object*

Returns YES if the argument *object* satisfies the qualifier, NO otherwise. This method can raise one of several possible exceptions if an error occurs, depending on the implementation.

EORelationshipManipulation

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOClassDescription.h

Protocol Description

The EORelationshipManipulation informal protocol builds on the basic EOKeyValueCoding informal protocol to allow you to modify to-many relationship properties. the Framework additions to NSObject provide default implementations of EORelationshipManipulation, which you rarely (if ever) need to override.

The primitive methods **addObject:toPropertyWithKey:** and **removeObject:fromPropertyWithKey:** add and remove single objects from to-many relationship arrays. The two other methods in the informal protocol, **addObject:toBothSidesOfRelationshipWithKey:** and **removeObject: fromBothSidesOfRelationshipWithKey:**, are implemented in terms of the two primitives to handle reciprocal relationships. These methods find the inverse relationship to the one identified by the specified key (if there is such an inverse relationship) and use **addObject:toPropertyWithKey:** and **removeObject: fromPropertyWithKey:** to alter both relationships, whether they're to-one or to-many.

The primitive methods check first for a method you might implement, **addTo***Key* or **removeFrom***Key*, invoking that method if it's implemented, otherwise using the basic key-value coding methods to do the work. Consequently, you rarely need to provide your own implementations of EORelationshipManipulation. Rather, you can provide relationship accessors (**addTo***Key* or **removeFrom***Key*) whenever you need to implement custom business logic.

Instance Methods

addObject:toBothSidesOfRelationshipWithKey:

– (void)**addObject:**(id)*anObject*
    toBothSidesOfRelationshipWithKey:(NSString *)*key*

Sets or adds *anObject* as the destination for the receiver's relationship identified by *key*, and also sets or adds the receiver for *anObject*'s reciprocal relationship if there is one. For a to-one relationship, *anObject* is set using **takeValue:forKey:**. For a to-many relationship, *anObject* is added using **addObject: toBothSidesOfRelationshipWithKey:**.

This method also properly handles removing **self** and *anObject* from their previous relationship as needed. For example, if an Employee object belongs to the Research department, invoking this method with the Maintenance department removes the Employee from the Research department as well as setting the Employee's department to Maintenance.

addObject:toPropertyWithKey:

– (void)**addObject:**(id)*anObject*
  toPropertyWithKey:(NSString *)*key*

Adds *anObject* to the receiver's to-many relationship identified by *key*, without setting a reciprocal relationship. Similar to the implementation of **takeValue:forKey:**, NSObject's implementation of this method first attempts to invoke a method of the form **addTo***Key***:**. If the receiver doesn't have such a method, this method gets the property array using **valueForKey:** and operates directly on that. For a to-many relationship, this method adds *anObject* to the array if it is not already in the array. For a to-one relationship, this method replaces the previous value with *anObject* .

removeObject:fromBothSidesOfRelationshipWithKey:

– (void)**removeObject:**(id)*anObject*
  fromBothSidesOfRelationshipWithKey:(NSString *)*key*

Removes *anObject* from the receiver's relationship identified by *key*, and also removes the receiver from *anObject*'s reciprocal relationship if there is one. For a to-one relationship, *anObject* is removed using **takeValue:forKey:** with **nil** as the value. For a to-many relationship, *anObject* is removed using **removeObject:fromPropertyWithKey:**.

removeObject:fromPropertyWithKey:

– (void)**removeObject:**(id)*anObject*
  fromPropertyWithKey:(NSString *)*key*

Removes *anObject* from the receiver's to-many relationship identified by *key*, without modifying a reciprocal relationship. Similar to the implementation of **takeValue:forKey:**, NSObject's implementation of this method first attempts to invoke a method of the form **removeFrom***Key***:**. If the receiver doesn't have such a method, this method gets the property array using **valueForKey:** and operates directly on that. For a to-many relationship, this method removes *anObject* from the array. For a to-one relationship, this method replaces *anObject* with nil.

EOQualifierComparison

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOQualifier.h

Protocol Description

The EOQualifierComparison informal protocol defines methods for comparing values. These methods are used for evaluating qualifiers in memory. Though declared for NSObject, most of these methods work properly only with value classes: NSString, NSDate, NSNumber, NSDecimalNumber, and EONull

Method Types

Testing value objects

– doesContain:
– isEqualTo:
– isGreaterThan:
– isGreaterThanOrEqualTo:
– isLessThan:
– isLessThanOrEqualTo:
– isLike:
– isCaseInsensitiveLike:
– isNotEqualTo:

Instance Methods

doesContain:

– (BOOL)**doesContain:**(id)*anObject*

Returns YES if the receiver contains *anObject*, NO if it doesn’t. NSObject’s implementation of this method returns YES only if the receiver is a kind of NSArray and contains *anObject*. In all other cases it returns NO.

isCaseInsensitiveLike:

– (BOOL)**isCaseInsensitiveLike:**(NSString *)*anObject*

Returns YES if the receiver is a case-insensitive match for *aString*, NO if it isn't. See "Using Wildcards" in the EOQualifier class specification for the wildcard characters allowed. NSObject's implementation returns NO; NSString's performs a proper case-insensitive comparison.

See also: – **isLike:**, – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLessThanOrEqualTo:**,– **isNotEqualTo:**

isEqualTo:

– (BOOL)**isEqualTo:**(id)*anObject*

Returns YES if the receiver is equal to *anObject*, NO if it isn't. NSObject's implementation invokes **isEqual:** and returns the result.

See also: – **doesContain:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLessThanOrEqualTo:**, – **isLike:**, – **isCaseInsensitiveLike:**, – **isNotEqualTo:**

isGreaterThan:

– (BOOL)**isGreaterThan:**(id)*anObject*

Returns YES if the receiver is greater than *anObject*, NO if it isn't. NSObject's implementation invokes **compare:** and returns YES if the result is NSOrderedDescending.

See also: – **doesContain:**, – **isEqualTo:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLessThanOrEqualTo:**, – **isLike:**, – **isCaseInsensitiveLike:**, – **isNotEqualTo:**

isGreaterThanOrEqualTo:

– (BOOL)**isGreaterThanOrEqualTo:**(id)*anObject*

Returns YES if the receiver is greater than or equal to *anObject*, NO if it isn't. NSObject's implementation invokes **compare:** and returns YES if the result is NSOrderedAscending.

See also: – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isLessThan:**, – **isLessThanOrEqualTo:**, – **isLike:**, – **isCaseInsensitiveLike:**, – **isNotEqualTo:**

isLessThan:

– (BOOL)**isLessThan:**(id)*anObject*

Returns YES if the receiver is less than *anObject*, NO if it isn't. NSObject's implementation invokes **compare:** and returns YES if the result is NSOrderedAscending.

See also: – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThanOrEqualTo:**, – **isLike:**, – **isCaseInsensitiveLike:**, – **isNotEqualTo:**

isLessThanOrEqualTo:

– (BOOL)**isLessThanOrEqualTo:**(id)*anObject*

Returns YES if the receiver is less than or equal to *anObject*, NO if it isn't. NSObject's implementation invokes **compare:** and returns YES if the result is NSOrderedAscending or NSOrderedSame.

See also: – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLike:**, – **isCaseInsensitiveLike:**, – **isNotEqualTo:**

isLike:

– (BOOL)**isLike:**(NSString *)*aString*

Returns YES if the receiver matches *aString* according to the semantics of the SQL **like** comparison operator, NO if it doesn't. See "Using Wildcards" in the EOQualifier class specification for the wildcard characters allowed. NSObject's implementation returns NO; NSString's performs a proper comparison.

See also: – **isCaseInsensitiveLike:**, – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLessThanOrEqualTo:**, – **isNotEqualTo:**

isNotEqualTo:

– (BOOL)**isNotEqualTo:**(id)*anObject*

Returns YES if the receiver is not equal to *anObject*, NO if it is. NSObject's implementation invokes **isEqual:**, inverts the result, and returns it.

See also: – **doesContain:**, – **isEqualTo:**, – **isGreaterThan:**, – **isGreaterThanOrEqualTo:**, – **isLessThan:**, – **isLessThanOrEqualTo:**, – **isLike:**, – **isCaseInsensitiveLike:**

EOSortOrderingComparison

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOSortOrdering.h

Protocol Description

The EOSortOrderingComparison informal protocol defines methods for comparing values. These methods are used for sorting value objects. Though declared for NSObject, most of these methods work properly only with value classes: NSString, NSDate, NSNumber, NSDecimalNumber, and EONull

Sorting value objects	
	– compareAscending:
	– compareCaseInsensitiveAscending:
	– compareCaseInsensitiveDescending:
	– compareDescending:

Instance Methods

compareAscending:

– (NSComparisonResult)**compareAscending:**(id)*anObject*

Returns NSOrderedAscending if *anObject* is naturally ordered after the receiver, NSOrderedDescending if it's naturally ordered before the receiver, and NSOrderedSame if they're equivalent for ordering purposes. NSObject's implementation of this method simply invokes **compare:**.

See also: – **compareDescending:**, – **compareCaseInsensitiveAscending:**, – **compareCaseInsensitiveDescending:**

compareCaseInsensitiveAscending:

– (NSComparisonResult)**compareCaseInsensitiveAscending:**(id)*anObject*

Returns NSOrderedAscending if *anObject* is naturally ordered—ignoring case—after the receiver, NSOrderedDescending if it's naturally ordered before the receiver, and NSOrderedSame if they're equivalent for ordering purposes. NSObject's implementation of this method invokes **compare:**, while NSString's invokes **caseInsensitiveCompare:**.

See also: – **compareCaseInsensitiveDescending:**, – **compareAscending:**, – **compareDescending:**

compareCaseInsensitiveDescending:

– (NSComparisonResult)**compareCaseInsensitiveDescending:**(id)*anObject*

Returns NSOrderedAscending if *anObject* is naturally ordered—ignoring case—*before* the receiver, NSOrderedDescending if it's naturally ordered *after* the receiver, and NSOrderedSame if they're equivalent for ordering purposes. NSObject's implementation of this method invokes **compare:** and inverts the result, while NSString's invokes **caseInsensitiveCompare:** and inverts the result.

See also: – **compareCaseInsensitiveAscending:**, – **compareDescending:**, – **compareAscending:**

compareDescending:

– (NSComparisonResult)**compareDescending:**(id)*anObject*

Returns NSOrderedAscending if *anObject* is naturally ordered *before* the receiver, NSOrderedDescending if it's naturally ordered *after* the receiver, and NSOrderedSame if they're equivalent for ordering purposes. NSObject's implementation of this method simply invokes **compare:** and inverts the result.

See also: – **compareAscending:**, – **compareCaseInsensitiveDescending:**, – **compareCaseInsensitiveAscending:**

EOValidation
(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOClassDescription.h

Inherits From:	java.lang.Object
Package:	com.apple.client.eocontrol

Protocol Description

The EOValidation informal protocol defines the way that enterprise objects validate their values. The validation methods check for illegal value types, values outside of established limits, illegal relationships, and so on. the Framework additions to NSObject provide default implementations of EOValidation, which are described in detail in this specification.

There are two kinds of validation methods. The first validates individual properties, and the second validates an entire object to see if it's ready for a specific operation (inserting, updating, and deleting). The two different types are discussed in more detail in the sections "Validating Individual Properties" and "Validating Before an Operation."

Instance Methods

validateForDelete

– (NSException *)**validateForDelete**

Confirms that the receiver can be deleted in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. For example, an object can't be deleted if it has a relationship with a delete rule of EODeleteRuleDeny and that relationship has a destination object.

NSObject's implementation sends the receiver's EOClassDescription a message (which performs basic checking based on the presence or absence of values). Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForDelete
{
    NSException *exception = [super validateForDelete];

    if ([balance intValue] != 0) {
        NSException *validationExample = [NSException
            validationExceptionWithFormat:@"The balance must be zero."];
      if (!exception)
         exception = validationException;
      else
         exception = [NSException aggregateExceptionWithExceptions:
            [NSArray arrayWithObjects:exception, validationException, nil]];
   }
   return exception;
}
```

See also: – **propagateDeleteWithEditingContext:** (EOEnterpriseObject), + **validationExceptionWithFormat:** (NSException Additions)

validateForInsert

– (NSException *)**validateForInsert**

Confirms that the receiver can be inserted in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation simply invokes **validateForSave**.

The method **validateForSave** is the generic validation method for when an object is written to the external store. If an object performs validation that isn't specific to insertion, it should go in **validateForSave**.

validateForSave

– (NSException *)**validateForSave**

Confirms that the receiver can be saved in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation sends the receiver's EOClassDescription a **validateObjectForSave:** message, then iterates through all of the receiver's properties, invoking **validateValue:forKey:** for each one. If this results in more than one exception, the exception returned contains the additional ones in its **userInfo** dictionary under the EOAdditionalExceptions key. Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForSave
{
    NSException *exception = [super validateForDelete];

    if ([balance intValue] != 0) {
        NSException *validationExample = [NSException
            validationExceptionWithFormat:@"The balance must be zero."];
      if (!exception)
         exception = validationException;
      else
         exception = [NSException aggregateExceptionWithExceptions:
            [NSArray arrayWithObjects:exception, validationException, nil]];
   }
   return exception;
}
```

Enterprise objects can implement this method to check that certain relations between properties hold; for example, that the end date of a vacation period follows the begin date. To validate an individual property, you can simply implement a method for it as described under **validateValue:forKey:**.

See also: + **validationExceptionWithFormat:** (NSException Additions),
+ **aggregateExceptionWithExceptions:** (NSException Additions)

validateForUpdate

– (NSException *)**validateForUpdate**

Confirms that the receiver can be inserted in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation simply invokes **validateForSave**.

The method **validateForSave** is the generic validation method for when an object is written to the external store. If an object performs validation that isn't specific to updating, it should go in **validateForSave**.

validateValue:forKey:

– (NSException *)**validateValue:**(id *)*valuePointer* **forKey:**(NSString *)*key*

Confirms that the value referenced by *valuePointer* is legal for the receiver's property named by *key*. Returns **nil** if it can confirm that the value is legal or an NSException that the sender may raise if it can't. The implementation can provide a coerced value by putting the new value into ***valuePointer**. This lets you convert strings to dates or numbers or maybe convert strings to an enumerated type value. NSObject's implementation sends a **validateValue:forKey:** message to the receiver's EOClassDescription. If that message doesn't return an exception, it checks for a method of the form **validate***Key***:** (for example, **validateBudget:** for a *key* of "budget") and invokes it, returning the result.

Enterprise objects can implement individual **validate***Key***:** methods to check limits, test for nonsense values, and otherwise confirm individual properties. To validate multiple properties based on relations among them, override the appropriate **validateFor...** method.

See also: + **validationExceptionWithFormat:** (NSException Additions)

EOValidation

Validating Individual Properties

The most general method for validating individual properties, **validateValue:forKey:**, validates a property indirectly by name (or key). 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 NSObject consults the object's EOClassDescription (using the EOEnterpriseObject informal protocol method **classDescription**) to coerce the value and to check for basic errors, such as a **null** value when that isn't allowed. If no basic errors exist, this default implementation then validates the value according to the object itself. It searches for a method of the form **validate***Key*: and invokes it if it exists. These are the methods that your custom classes can implement to validate individual properties, such as **validateAge**: to check that the value the user entered is within acceptable limits. The **validateAge** method sholuld return nil, indicating the value is acceptable, or an NSException created by calling the NSException method **validationException:withFormat:**.

Coercion is performed automatically for you (by the EOClassDescription), so all you need handle is validation itself. Since you can implement custom validation logic in the **validate***Key*: methods, you rarely need to override the EOValidation method **validateValue:forKey:**. Rather, the default implementation provided by NSObject is generally sufficient.

As an example of how validating a single property works, suppose that Member objects have an **age** attribute stored as an integer. This attribute has a lower limit of 16, defined by the Member class. Now, suppose a user types "12" into a text field for the age of a member. The value comes into the Framework as a string. When **validateValue:forKey:** is invoked to validate the new value, the method uses its EOClassDescription to convert the string "12" into an NSNumber, then invokes **validateAge:** with that NSNumber. The **validateAge:** method compares the age to its limit of 16 and returns an exception to indicate that the new value is not acceptable:

```
public void validateAge(java.lang.Object age) throws EOvalidation.Exception {
    if ((((Number)age).intValue) < 16)
        throw new EOValidation.Exception("Age of " + age + " is below minimum.");
}
- (NSException *)validateAge:(NSNumber *)age
{
    if ([age intValue] < 16) {
        return [NSException
            validationExceptionWithFormat:@"Age of %@ is below minimum.", age];
    }
    return nil;
}
```

The method **validationExceptionWithFormat:** used in the above example is a method that the Framework adds to NSException for convenient creation of validation exceptions.

When Properties are Validated

The Framework validates all of an object’s properties before the object is saved to an external source—either inserted or updated. Additionally, you can design your application so that changes to a property’s value are validated immediately, as soon as a user attempts to leave an editable field in the user interface (in Java Client and Application Kit applications only). Whenever an EODisplayGroup sets a value in an object, it sends the object a **validateValue:forKey:** message, allowing the object to coerce the value’s type, perform any additional validation, and return an exception if the value isn't valid. By default, the display group leaves validation errors to be handled when the object is saved, using **validateValue:forKey:** only for type coersion. However, you can use the EODisplayGroup method **setValidatesChangesImmediately:** with an argument of YES to tell the display group to immediately present an attention panel whenever a validation error is encountered.

Validating Before an Operation

The remaining EOValidation methods—**validateForInsert**, **validateForUpdate**, **validateForSave**, and **validateForDelete**—validate an entire object to see if it’s valid for a particular operation. These methods are invoked automatically by the Framework when the associated operation is initiated. NSObject provides default implementations, so you only have to implement them yourself when special validation logic is required. For example, you can override these methods in your custom enterprise object classes to allow or refuse the operation based on property values. For example, a Fee object might refuse to be deleted if it hasn’t been paid yet. Or you can override these methods to perform delayed validation of properties or to compare multiple properties against one another; for example, you might verify that a pair of dates is in the proper temporal order.

```
- (NSException *)validateForSave
{
    NSException *exception = [super validateForSave];
    NSException *myException = nil;

    if ([startDate compare:endDate] == NSOrderedDescending) {
        myException = [NSException
            validationExceptionWithFormat:@"Start date must precede end date."];
    }
    if (exception && myException) {
        exception = [NSException aggregateExceptionWithExceptions:
            [NSArray arrayWithObjects:exception, myException, nil]];
    } else if (myException) {
        exception = myException;
    }
    return exception;
}
```

Note that this method also invokes **super**’s implementation. This is important, as the default implementations of the **validateFor...** methods pass the check on to the object’s EOClassDescription, which performs basic checking among properties, including invoking **validateValue:forKey:** for each property.

The access layer’s EOEntityClassDescription class verifies constraints based on an EOModel, such as delete rules. For example, the delete rule for a Department object might state that it can’t be deleted if it still contains Employee objects.

The method **validateForSave** is the generic validation method for when an object is written to the external store. If an object performs validation that isn’t specific to insertion or to updating, it should go in **validateForSave**.

EOValueMerging

(informal protocol)

Category Of:	NSObject
Declared In:	EOControl/EOClassDescription.h

Protocol Description

Description forthcoming.

Method Types

Merging values	– changesFromSnapshot:
	– reapplyChangesFromDictionary:

Instance Methods

changesFromSnapshot:

– (NSDictionary *)**changesFromSnapshot:**(NSDictionary *)*snapshot*

The result is like a snapshot except that it contains only those keys that refer to uncommitted changes in the object relative to the given snapshot. For to-many keys, the uncommitted value is an array of two arrays: uncommitted additions and uncommitted deletions. The return value is autoreleased.

reapplyChangesFromDictionary:

– (void)**reapplyChangesFromDictionary:**(NSDictionary *)*changes*

Similar to **takeValuesFromDictionary:** but the *changes* dictionary is not quite the same as a snapshot. For to-many relationship keys, the value is an array with exactly two arrays in it: the first is an array of objects to be added to the relation, and the second is an array of objects to be removed from the relation. Attribute and to-one relationship keys refer to values that should replace the current value. An instance of EONull is used in the *changes* dictionary as a placeholder for nil.