# The EOControl Framework

**Framework:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

**Header File Directories:** System/Developer/Java/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 ChangesEOControl provides four classes and an interface 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 interface, 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. | EOObserverCenter<br>EODelayedObserverQueue<br>EODelayedObserver<br>EOObserverProxy<br>EOObserving (interface) |

| Responsibility | Classes |
|---|---|
| Object Storage Abstraction | EOObjectStore<br>EOCooperatingObjectStore (Yellow Box only)<br>EOObjectStoreCoordinator (Yellow Box only)<br>EOGlobalID<br>EOKeyGlobalID<br>EOTemporaryGlobalID |
| Query specification | EOFetchSpecification<br>EOQualifier<br>EOSortOrdering |
| Interaction with enterprise objects | EOClassDescription (validation)<br>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 an interface 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 interface, 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

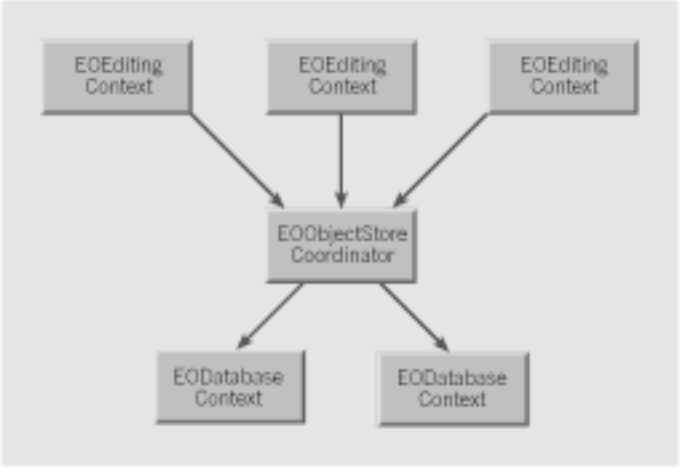
The control layer provides an infrastructure that's independent of your application's storage mechanism (typically a database) by defining an API for an "intelligent" repository of objects, whether it's based on external data or whether it manages objects entirely in memory. EOObjectStore is an abstract 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:




Note that EOCooperatingObjectStore, EOObjectStoreCoordinator, and EODatabaseContext are not provided by Java Client.

### 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 empty enterprise object (called a *fault*). When a fault is accessed (sent a message), it triggers its object store to fetch its data and fill the fault with its data. This preserves both the object's reference 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 EOFaultHandler class specification for more information.

# EOArrayDataSource

**Inherits From:** EODataSource : Object (Java Client)
EODataSource : NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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.

## Constructors

### EOArrayDataSource

public **EOArrayDataSource**(
EOClassDescription *classDescription*,
EOEditingContext *editingContext*)

Creates and returns an EOArrayDataSource object where *classDescription* contains information about the objects provided by the EOArrayDataSource and *editingContext* is the EOArrayDataSource's editing context. Either argument may be null

## Instance Methods

### setArray

public void **setArray**(foundation.NSArray *array*)

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

# EOAndQualifier

| | |
|---|---|
| **Inherits From:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | EOQualifierEvaluation<br>NSCoding (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## Class Description

EOAndQualifier is a subclass of EOQualifier that contains multiple qualifiers. EOAndQualifier implements the EOQualifierEvaluation interface, 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 **false**. If one of its qualifiers returns **false**, the EOAndQualifier object returns **false** immediately. If all of its qualifiers return **true**, the EOAndQualifier object returns **true**.

## Interfaces Implemented

| | |
|---|---|
| EOQualifierEvaluation | |
| | evaluateWithObject |
| NSCoding (Java Client only) | |
| | classForCoder |
| | encodeWithCoder |

## Constructors

### EOAndQualifier

public **EOAndQualifier**(NSArray *qualifiers*)

Creates a new EOAndQualifier. If *qualifiers* is provided, the new EOAndQualifier is initialized with the EOQualifier objects in *qualifiers*.

## Instance Methods

### evaluateWithObject

EOQualifierEvaluation Interface
Java Client:
public boolean **evaluateWithObject**(EOKeyValueCodingAdditions *anObject*)
Yellow Box:
public boolean **evaluateWithObject**(java.lang.Object *anObject*)

Returns **true** if *anObject* satisfies the qualifier, **false** otherwise. When an EOAndQualifier object receives an **evaluateWithObject** message, it evaluates each of its qualifiers until one of them returns **false**. If any of its qualifiers returns **false**, the EOAndQualifier object returns **false** immediately. If all of its qualifiers return **true**, the object returns **true**. This method can throw 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).

### qualifiers

public NSArray **qualifiers**()

Returns the receiver's qualifiers.

# EOClassDescription

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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 interface declares several EOClassDescription-related methods that define basic enterprise objects behavior, such as undo and validation. The EOCustomObject and EOGenericRecord classes implement the EOEnterpriseObject inteface. An enterprise object class can either accept the default implementations by subclassing from EOCustomObject 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

## Constants

EOClassDescription defines the following string constants for the names of the notifications it posts:
- EOClassDescriptionNeededForClassNotification
- EOClassDescriptionNeededForEntityNameNotification

See the Notifications section for more information on the notifications.

Additionally, EOClassDescription defines an integer constant for each delete rule:

- DeleteRuleCascade
- DeleteRuleDeny
- DeleteRuleNullify
- DeleteRuleNoAction

For more information on the delete rules, see the method description for **deleteRuleForRelationshipKey**.

## Method Types

Managing EOClassDescriptions

invalidateClassDescriptionCache (Yellow Box only)
registerClassDescription

Getting EOClassDescriptions

classDescriptionForClass
classDescriptionForEntityName

Creating new object instances

createInstanceWithEditingContext

Propagating delete

propagateDeleteForObject

Returning information from the EOClassDescription

entityName
attributeKeys
classDescriptionForDestinationKey
toManyRelationshipKeys
toOneRelationshipKeys
inverseForRelationshipKey
ownsDestinationObjectsForRelationshipKey
deleteRuleForRelationshipKey

Performing validation

validateObjectForDelete
validateObjectForSave
validateValueForKey

Providing default characteristics for key display (Yellow Box only)

defaultFormatterForKey (Yellow Box only)
defaultFormatterForKeyPath (Yellow Box only)
displayNameForKey (Yellow Box only)

Handling newly inserted and newly fetched objects
awakeObjectFromFetch
awakeObjectFromInsertion

Setting the delegate
classDelegate
setClassDelegate

Getting an object's description (Yellow Box only)
userPresentableDescriptionForObject (Yellow Box only)

## Static Methods

### classDelegate

public static java.lang.Object **classDelegate**()

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

**See also: setClassDelegate**

### classDescriptionForClass

public static EOClassDescription **classDescriptionForClass**(java.lang.Class *aClass*)

Invoked by the default implementations of the EOEnterpriseObject interface 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

public static EOClassDescription
**classDescriptionForEntityName**(java.lang.String *entityName*)

Returns the EOClassDescription registered under *entityName*.

### invalidateClassDescriptionCache

public static void **invalidateEOClassDescriptionCache**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public static void **registerClassDescription**(
com.apple.client.eocontrol.EOClassDescription *description*,
java.lang.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

public static void **setClassDelegate**(java.lang.Object *delegate*)

Sets the delegate for the EOClassDescription class (as opposed to EOClassDescription instances) to *delegate*. For more information on the class delegate, see the EOClassDescription.ClassDelegate interface specification.

**See also:** **classDelegate**

## Instance Methods

### attributeKeys

public NSArray **attributeKeys**()

Overridden by subclasses to return an array of attribute keys (Strings) for objects described by the receiver. "Attributes" contain immutable data (such as Numbers and Strings), 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**

## awakeObjectFromFetch

public void **awakeObjectFromFetch**(
EOEnterpriseObject *object*,
EOEditingContext *anEditingContext*)

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

## awakeObjectFromInsertion

public void **awakeObjectFromInsertion**(
EOEnterpriseObject *object*,
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

public EOClassDescription **classDescriptionForDestinationKey**(java.lang.String *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 **null**.

## createInstanceWithEditingContext

public EOEnterpriseObject **createInstanceWithEditingContext**(
EOEditingContext *anEditingContext*,
EOGlobalID *globalID*)

Overridden by subclasses to create an object of the appropriate class in *anEditingContext* with *globalID*. In typical usage, both of the method's arguments are **null**. To create the object, the subclass should pass *anEditingContext*, itself, and *aGlobalID* to the appropriate constructor. 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 **null**.

## defaultFormatterForKey

public NSFormatter **defaultFormatterForKey**(java.lang.String *key*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns the default NSFormatter to use when parsing values for assignment to *key*. EOClassDescription's implementation returns **null**. The access layer's EOEntityClassDescription's implementation returns an NSFormatter based on the Java 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

public NSFormatter **defaultFormatterForKey**(java.lang.String *key*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

## deleteRuleForRelationshipKey

public int **deleteRuleForRelationshipKey**(java.lang.String *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 |
|---|---|
| DeleteRuleNullify | 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 null. |
| DeleteRuleCascade | When the source object (department) is deleted, any destination objects (employees) are also deleted. |
| DeleteRuleDeny | If the source object (department) has any destination objects (employees), a delete operation is refused. |

| Constant | Description |
|---|---|
| DeleteRuleNoAction | When the source object is deleted, its relationship is ignored and no action is taken to propagate the deletion to destination objects.<br><br>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

public java.lang.String **displayNameForKey**(java.lang.String *key*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public java.lang.String **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 **null**.

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

## inverseForRelationshipKey

public java.lang.String **inverseForRelationshipKey**(java.lang.String *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 **null**.

## ownsDestinationObjectsForRelationshipKey

public boolean **ownsDestinationObjectsForRelationshipKey**(java.lang.String *relationshipKey*)

Overridden by subclasses to return **true** or **false** 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 **false**. In the common case, this behavior for an enterprise object is defined in its EOModel. (The EOModel class is defined in EOAccess.)

## propagateDeleteForObject

public void **propagateDeleteForObject**(
EOEnterpriseObject *object*,
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 interfacespecification.

**See also: deleteRuleForRelationshipKey**

## toManyRelationshipKeys

public 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 **null**.

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

## toOneRelationshipKeys

public 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 **null**.

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

## userPresentableDescriptionForObject

public java.lang.String **userPresentableDescriptionForObject**(java.lang.Object *anObject*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public void **validateObjectForDelete**(EOEnterpriseObject *object*)
    throws EOValidation.Exception

Overridden by subclasses to determine whether it's permissible to delete *object*. Subclasses should complete normally if the delete operation should proceed, or raise an exception containing a user-presentable (localized) error message. EOClassDescription's implementation of this method completes normally.

### validateObjectForSave

public void **validateObjectForSave**(EOEnterpriseObject *object*)
throws EOValidation.Exception

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

### validateValueForKey

public java.lang.Object **validateValueForKey**(
java.lang.Object *value*,
java.lang.String *key*)
throws EOValidation.Exception

Overridden by subclasses to validate *value*. Subclasses should complete normally if the value is acceptable, or raise an exception containing a user-presentable (localized) error message if not. Implementations can replace *value* by returning a new value.

## Notifications

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

### EOClassDescriptionNeededForClassNotification

One of the EOClassDescription-related methods in the EOEnterpriseObject interface 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 (String) |
| **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 interface (and implemented in EOCustomObject and EOGenericRecord). 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 interface 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 (**awakeFromInsertion**), or to provide additional initialization to newly fetched objects (**awakeFromFetch**):

- 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:

```
public void validateForSave() throw EOValidation.Exception {
    if (salary > 1500 && jobLevel < 2) {
        throw new EOValidation.Exception(
            "The salary is too high for that position!");
        }
    // pass the check on to the EOClassDescription
    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 interface 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.

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**. 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**

## EOEntityClassDescription

There are only three methods in EOClassDescription that have meaningful implementations (that is, that don't either return **null** or simply return without doing anything): **invalidateClassDescriptionCache**, **registerClassDescription**, and **propagateDeleteForObject**. 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** 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**.

# EOCooperatingObjectStore

**Inherits From:** EOObjectStore : NSObject

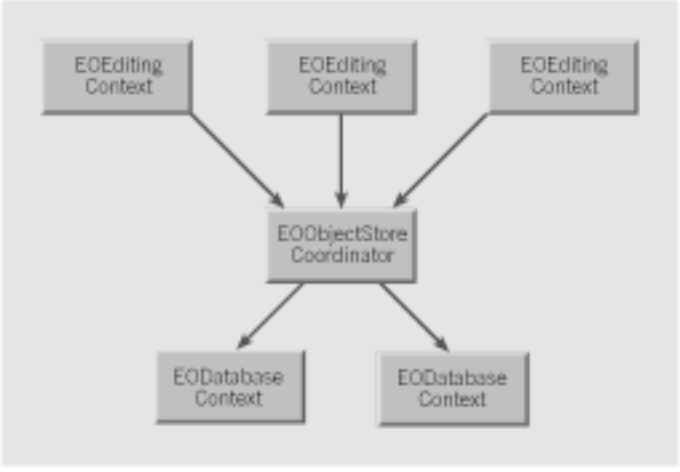
**Package:** com.apple.yellow.eocontrol (Yellow Box)

## 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. It is for use in WebObjects and Yellow Box applications only; there is no equivalent class for Java Client applications. 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
recordChangesInEditingContext
recordUpdateForObject

Returning information about objects

valuesForKeys

Determining if the EOCooperatingObjectStore is responsible for an operation

ownsObject
ownsGlobalID
handlesFetchSpecification

## Instance Methods

### commitChanges

public abstract void **commitChanges**()

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

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

### handlesFetchSpecification

public abstract boolean **handlesFetchSpecification**(EOFetchSpecification *fetchSpecification*)

Overridden by subclasses to return **true** 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

public abstract boolean **ownsGlobalID**(EOGlobalID *globalID*)

Overridden by subclasses to return **true** 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

public abstract boolean **ownsObject**(java.lang.Object *object*)

Overridden by subclasses to return **true** 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

public abstract 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

public abstract void **prepareForSaveWithCoordinator**(
EOObjectStoreCoordinator *coordinator*,
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**.

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

## recordChangesInEditingContext

public abstract 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**, **recordUpdateForObject**

## recordUpdateForObject

public abstract void **recordUpdateForObject**(
java.lang.Object *object*,
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**, **recordChangesInEditingContext**

## rollbackChanges

public abstract 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

public abstract NSDictionary **valuesForKeys**(
NSArray *keys*,
java.lang.Object *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** 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.

# EOCustomObject

| | |
|---|---|
| **Inherits From:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | EOEnterpriseObject<br>EOKeyValueCoding (EOKeyValueCodingAdditions)<br>EOKeyValueCodingAdditions (EOEnterpriseObject)<br>EORelationshipManipulation (EOEnterpriseObject)<br>EOValidation (EOEnterpriseObject)<br>EOFaulting (EOEnterpriseObject) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (\Yellow Box) |

## Class Description

The EOCustomObject class provides a default implementation of the EOEnterpriseObject interface. If you need to create a custom enterprise object class, you can subclass EOCustomObject and inherit the Framework's default implementations. Some of the methods are for subclasses to implement or override, but most are meant to be used as defined by EOCustomObject. For information on which methods you should implement in your subclass, see the EOEnterpriseObject interface specification.

EOCustomObject's method implementations are described in the specification for the interface that declares them. For example, you can find a description of how EOCustomObject implements **valueForKey** (introduced in the EOKeyValueCoding interface) in the specification for EOKeyValueCoding, and you can find a description of how EOCustomObject implements **classDescription** (introduced in the EOEnterpriseObject interface) in the specification for EOEnterpriseObject.

The only methods provided in EOCustomObject that aren't defined in the EOEnterpriseObject interface are the following three static methods:

- accessInstanceVariablesDirectly
- flushAllKeyBindings
- useStoredAccessor

You would never invoke these methods, rather, they are provided in EOCustomObject to demonstrate the additional API your custom enterprise objects can implement. Similarly, EOCustomObject's constructors are not meant to be invoked; you would never create an instance of EOCustomObject. Rather, EOCustomObject provides the constructors to demonstrate the constructors your custom enterprise objects should implement.

## Interfaces Implemented

EOKeyValueCoding

- handleQueryWithUnboundKey
- handleTakeValueForUnboundKey
- storedValueForKey
- takeStoredValueForKey
- takeValueForKey
- unableToSetNullForKey
- valueForKey

EOKeyValueCodingAdditions

- takeValueForKeyPath
- takeValuesFromDictionary
- valueForKeyPath
- valuesForKeys

EORelationshipManipulation

- addObjectToBothSidesOfRelationshipWithKey
- addObjectToPropertyWithKey
- removeObjectFromBothSidesOfRelationshipWithKey
- removeObjectFromPropertyWithKey

EOValidation

- validateForDelete
- validateForInsert
- validateForSave
- validateForUpdate
- validateValueForKey

EOEnterpriseObject

allPropertyKeys
attributeKeys
awakeFromFetch
awakeFromInsertion
changesFromSnapshot (Yellow Box only)
classDescription
classDescriptionForDestinationKey
clearProperties
deleteRuleForRelationshipKey
editingContext
entityName
eoDescription
eoShallowDescription
inverseForRelationshipKey
invokeRemoteMethod (Java Client only)
isToManyKey
ownsDestinationObjectsForRelationshipKey
propagateDeleteWithEditingContext
reapplyChangesFromDictionary (Yellow Box only)
snapshot
toManyRelationshipKeys
toOneRelationshipKeys
updateFromSnapshot
userPresentableDescription (Yellow Box only)
willChange

EOFaulting

clearFault
isFault
turnIntoFault
willRead

## Constructors

public **EOCustomObject**(EOEditingContext *anEOEditingContext*,
EOClassDescription *anEOClassDescription*, EOGlobalID *anEOGlobalID*)

You would never create an instance of EOCustomObject; rather, your subclasses can create constructors of this same form. A subclass's constructors should create a new object and initialize it with the arguments provided.

**See also:** **createInstanceWithEditingContext** (EOClassDescription)

## Static Methods

### accessInstanceVariablesDirectly

public static boolean **accessInstanceVariablesDirectly**()

Subclasses implement this method to return **false** if the key-value coding methods should never access the corresponding instance variable directly on finding no accessor method for a property. You don't have to implement this method if the default behavior of accessing instance variables directly is correct for your objects.

**See also:** **valueForKey**, **takeValueForKey**

### flushAllKeyBindings

public static 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.

**See also:**

### useStoredAccessor

public static boolean **useStoredAccessor**()

Subclasses implement this method to return **false** if the stored value methods (**storedValueForKey** and **takeStoredValueForKey**) should not use private accessor methods in preference to public accessors. Returning **false** 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 **takeValueForKey**). You don't have to implement this method if the default default stored value search order is correct for your objects.

# EODataSource

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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
- qualifyWithRelationshipKeyAndObject

Accessing the editing context
- editingContext

Accessing the class description
- classDescriptionForObjects

## Instance Methods

### classDescriptionForObjects

public EOClassDescription **classDescriptionForObjects**()

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

### createObject

public java.lang.Object **createObject**()

Creates a new object, inserts it in the receiver's collection of objects if appropriate, and returns the object. Returns **null** 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** 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

public abstract EODataSource **dataSourceQualifiedByKey**(java.lang.String *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 **qualifyWithRelationshipKeyAndObject** to set a specific master object (or to change the relationship key). EODataSource's implementation merely throws an exception; subclasses shouldn't invoke **super**'s implementation.

### deleteObject

public abstract void **deleteObject**(java.lang.Object *anObject*)

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

## editingContext

public EOEditingContext **editingContext**()

Implemented by subclasses to return the receiver's EOEditingContext. EODataSource's implementation returns **null**.

## fetchObjects

public NSArray **fetchObjects**()

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

## insertObject

public abstract void **insertObject**(java.lang.Object *object*)

Implemented by subclasses to insert *object*. EODataSource's implementation merely throws an exception; subclasses shouldn't invoke **super**'s implementation.

## qualifyWithRelationshipKeyAndObject

public abstract void **qualifyWithRelationshipKey**(
    java.lang.String *key*,
    java.lang.Object *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 message that created the receiver. If *sourceObject* is **null**, the receiver qualifies itself to provide no objects. EODataSource's implementation merely throws an exception; 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 constructor whose arguments describe the objects. The EODatabaseDataSource constructor, for exampleuses 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 **null**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. EODataSource uses an EOClassDescription 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 by EOCustomObject and EOGenericRecord). If your EODataSource subclass doesn't use an EOClassDescription at all it, can use the default implementation of **classDescriptionForObjects**, which returns **null**.

### 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 **qualifyWithRelationshipKeyAndObject** 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:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | EOObserving |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## 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 ChangesEOControl provides four classes and an interface 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 interface, 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." 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:

| | |
|---|---|
| ObserverPriorityImmediate | ObserverPriorityFourth |
| ObserverPriorityFirst | ObserverPriorityFifth |
| ObserverPrioritySecond | ObserverPrioritySixth |
| ObserverPriorityThird | ObserverPriorityLater |

## Interfaces Implemented

EOObserving

objectWillChange

## Method Types

Change notification

subjectChanged

Canceling change notification

discardPendingNotification

Getting the queue and priority

observerQueue
priority

## Instance Methods

### discardPendingNotification

public 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 when its done observing changes.

**See also:** **observerQueue**

## objectWillChange

interface EOObserving
public void **objectWillChange**(java.lang.Object *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

public EODelayedObserverQueue **observerQueue**()

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

**See also:** **defaultObserverQueue** (EODelayedObserverQueue)

## priority

public int **priority**()
()

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

- ObserverPriorityImmediate
- ObserverPriorityFirst
- ObserverPrioritySecond
- ObserverPriorityThird
- ObserverPriorityFourth
- ObserverPriorityFifth
- ObserverPrioritySixth
- ObserverPriorityLater

EODelayedObserver's implementation returns ObserverPriorityThird. See the EODelayedObserverQueue class specification for more information on priorities.

## subjectChanged

public abstract 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 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 ObserverPriorityThird. 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 when a delayed observer is done observing changes. This prevents observers from being sent change notifications after they've been finalized.

# EODelayedObserverQueue

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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 ChangesEOControl provides four classes and an interface 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 interface, 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." 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

## Constants

EODelayedObserverQueue defines the following constant:

| Constant | Type | Description |
|---|---|---|
| FlushDelayedObserversRunLoop | int | Determines when to notify delayed observers are notified, during end of event processing. |

## Method Types

Constructors

EODelayedObserverQueue

Getting the default queue

defaultObserverQueue

Enqueuing and dequeuing observers

enqueueObserver
dequeueObserver

Sending change notifications

notifyObserversUpToPriority

Configuring notification behavior

runLoopModes (Yellow Box only)
setRunLoopModes (Yellow Box only)

## Constructors

### EODelayedObserverQueue

public **EODelayedObserverQueue**()

Creates and returns a new EODelayedObserverQueue with NSRunLoop.DefaultRunLoopMode as its only run loop mode.

**See also:** **runLoopModes** (Yellow Box only)

## Static Methods

### defaultObserverQueue

public static EODelayedObserverQueue **defaultObserverQueue**()

Returns the EODelayedObserverQueue that EODelayedObservers use by default.

## Instance Methods

### dequeueObserver

public void **dequeueObserver**(EODelayedObserver *anObserver*)

Removes *anObserver* from the receiver.

**See also:** **enqueueObserver**

### enqueueObserver

public void **enqueueObserver**(EODelayedObserver *anObserver*)

Records *anObserver* to be sent **subjectChanged** messages. If *anObserver*'s priority is ObserverPriorityImmediate, 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 ObserverPriorityImmediate, it registers the receiver to be sent a **notifyObserversUpToPriority** message at the end of the run loop, using FlushDelayedObserversRunLoopOrdering and the receiver's run loop modes. This causes enqueued observers up to a priority of ObserverPrioritySixth to be notified automatically during each pass of the run loop.

When *anObserver* is done observing changes, it should invoke **discardPendingNotification** to remove itself from the queue.

**See also:** **dequeueObserver**, **priority** (EODelayedObserver), **discardPendingNotification** (EODelayedObserver), **runLoopModes** (Yellow Box only)

## notifyObserversUpToPriority

public void **notifyObserversUpToPriority**(int *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 ObserverPrioritySixth.

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

## runLoopModes

public NSArray **runLoopModes**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns the receiver's run loop modes.

## setRunLoopModes

public void **setRunLoopModes**(NSArray *modes*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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 Siva and Yellow Box applications, an EODisplayGroup (EOInterface) does this (among many other things) on receiving an ObjectsChangedInEditingContextNotification from its EOEditingContext.

Although you can create individual EODelayedObserverQueues, you typically use the single instance provided by the static 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.

## 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 sets up the EODelayedObserverQueue to receive a message at the end of the run loop. EODelayedObserver sets up this delayed invocation in NSRunLoop.DefaultRunLoopMode, but you can change the mode or add additional modes in which delayed invocation occurs using **setRunLoopModes** (Yellow Box only).

**notifyObserversUpToPriority** cycles through the queue of EODelayedObservers in priority order, from ObserverPriorityFirst 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, providing 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)**. 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 : Object (Java Client)
EODataSource : NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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** 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

Constructors
- EODetailDataSource

Qualifying instances
- qualifyWithRelationshipKey

Examining instances
- masterDataSource
- detailKey
- masterObject

Accessing the master class description
- masterClassDescription
- setMasterClassDescription (Yellow Box only)

Accessing the objects
- fetchObjects

Inserting and deleting objects

insertObject
deleteObject

Accessing the master editing context

editingContext

## Constructors

### EODetailDataSource

public **EODetailDataSource**(
EOClassDescription *masterClassDescription*,
java.lang.String *relationshipKey*)

Creates and returns a new EODetailDataSource object. The new data source's **masterObject** is associated with *masterClassDescription*, and *relationshipKey* is assigned to the new data source's **detailKey**. The constructor invokes **qualifyWithRelationshipKey** specifying *relationshipKey* as the relationship key and **null** as the object.

public **EODetailDataSource**(
EODataSource *masterDataSource*,
java.lang.String *relationshipKey*)

Creates and returns a new EODetailDataSource object. The new data source provides destination objects for the relationship named by *relationshipKey* from a **masterObject** in *masterDataSource*.

**See also:** **masterClassDescription**, **masterDataSource**

## Instance Methods

### deleteObject

public void **deleteObject**(java.lang.Object *anObject*)

Sends a **removeObjectFromPropertyWithKey** message (defined in the EORelationshipManipulation interface) to the master object with *anObject* and the receiver's detail key as the arguments. Throws an exception if there's no master object or no detail key set.

## detailKey

public java.lang.String **detailKey**()

Returns the name of the relationship for which the receiver provides objects, as provided to the constructor when the receiver was created or as set in **qualifyWithRelationshipKey**. If none has been set yet, returns **null**.

**See also:** "Constructors"

## editingContext

public EOEditingContext **editingContext**()

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

## fetchObjects

public NSArray **fetchObjects**()

Sends **valueForKey** (defined in the EOKeyValueCoding interface) 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.

## insertObject

public void **insertObject**(java.lang.Object *anObject*)

Sends an **addObjectToBothSidesOfRelationshipWithKey** message (defined in the EORelationshipManipulation interface) to the master object with *anObject* and the receiver's detail key as the arguments. Throws an exception if there's no master object or no detail key set.

## masterClassDescription

public EOClassDescription **masterClassDescription**()

Returns the EOClassDescription of the receiver's master object.

**See also:** **setMasterClassDescription**, "Constructors"

## masterDataSource

public EODataSource **masterDataSource**()

Returns the receiver's master data source.

**See also:** **detailKey**, "Constructors"

## masterObject

public java.lang.Object **masterObject**()

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

**See also:** **detailKey**

## qualifyWithRelationshipKey

public void **qualifyWithRelationshipKey**(
    java.lang.String *relationshipKey*,
    java.lang.Object *masterObject*)

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

**See also:** **detailKey**

## setMasterClassDescription

public void **setMasterClassDescription**(EOClassDescription *anEOClassDescription*)

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

**See also:** **masterClassDescription**

# EOEditingContext

| | |
|---|---|
| **Inherits From:** | EOObjectStore : Object (Java Client)<br>EOObjectStore : NSObject (Yellow Box) |
| **Implements:** | EOObserving<br>NSLocking (Yellow Box only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## 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 |

**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 true 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 at a Glance

## 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 interface) 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
- For more discussion of working programmatically with EOEditingContexts, see the chapter “Application Configurations” in the Enterprise Objects Framework Developer’s Guide.
- Fetching Objects
- Managing Changes in Your Application
- Managing Changes in Your Application
- General Guidelines for Managing the Object Graph
- Methods for Managing the Object Graph

## Constants

The following string constants name notifications EOEditingContext posts:

- EditingContextDidSaveChangesNotification
- ObjectsChangedInEditingContextNotification

See the Notifications section for more information on the notifications.

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

- UpdatedKey
- DeletedKey
- InsertedKey
- InvalidatedKey

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.

## Interfaces Implemented

EOObserving

objectWillChange

## Method Types

Constructors

EOEditingContext

Controlling EOEditingContext’s memory management strategy

Fetching objects

objectsWithFetchSpecification

Committing or discarding changes

saveChanges
refaultObjects
refetch
revert (Yellow Box only)
invalidateAllObjects

Registering changes

deleteObject
insertObject
insertObjectWithGlobalID
objectWillChange
processRecentChanges

Checking changes

deletedObjects
insertedObjects
updatedObjects
hasChanges

Object registration and snapshotting

forgetObject
recordObject
committedSnapshotForObject
currentEventSnapshotForObject
objectForGlobalID
globalIDForObject
registeredObjects

Locking objects

lockObject
lockObjectWithGlobalID
isObjectLockedWithGlobalID
setLocksObjectsBeforeFirstModification
locksObjectsBeforeFirstModification

Undoing operations (Yellow Box only)
- redo (Yellow Box only)
- undo (Yellow Box only)
- setUndoManager (Yellow Box only)
- undoManager (Yellow Box only)

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 (Yellow Box only)
- setInvalidatesObjectsWhenFreed (Yellow Box only)
- invalidatesObjectsWhenFreed (Yellow Box only)

Interacting with the server (Java Client only)
- invokeRemoteMethod (Java Client only)

Locking (Yellow Box only)
- lock (Yellow Box only)
- unlock (Yellow Box only)

Working with raw rows (Yellow Box only)
- faultForRawRow (Yellow Box only)

Unarchiving from nib
- defaultParentObjectStore
- setDefaultParentObjectStore
- setSubstitutionEditingContext
- substitutionEditingContext

Nested EOEditingContext support

objectsWithFetchSpecification
objectsForSourceGlobalID
arrayFaultWithSourceGlobalID
faultForGlobalID
saveChangesInEditingContext
refaultObject
invalidateObjectsWithGlobalIDs
initializeObject

Archiving and unarchiving objects (Yellow Box only)

encodeObjectWithCoder (Yellow Box only)
initObjectWithCoder (Yellow Box only)
setUsesContextRelativeEncoding (Yellow Box only)
usesContextRelativeEncoding (Yellow Box only)

## Constructors

### EOEditingContext

public **EOEditingContext**()

Creates a new EOEditingContext object with the default parent object store as its parent object store.

public **EOEditingContext**(EOObjectStore *anObjectStore*)

Creates a new EOEditingContext object with *anObjectStore* as its parent object store. For more discussion of parent object stores, see "Other Classes that Participate in Object Graph Management" in the class description.

**See also:** **parentObjectStore**, **defaultParentObjectStore**

## Static Methods

### defaultParentObjectStore

public static 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 static method **defaultCoordinator**.

**See also:** **setDefaultParentObjectStore**

## encodeObjectWithCoder

public static void **encodeObjectWithCoder**(
java.lang.Object *object*,
NSCoder *encoder*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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:** **initObjectWithCoder**, **setUsesContextRelativeEncoding**, **usesContextRelativeEncoding**

## initObjectWithCoder

public static java.lang.Object **initObjectWithCoder**(
java.lang.Object *object*,
NSCoder *decoder*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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:** **encodeObjectWithCoder**, **setUsesContextRelativeEncoding**, **usesContextRelativeEncoding**

## setDefaultParentObjectStore

public static 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 **null**.

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

public static 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 **null**.

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

public static void **setUsesContextRelativeEncoding**(boolean *flag*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Sets according to *flag* whether **encodeObjectWithCoder** 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**, **encodeObjectWithCoder**,

## substitutionEditingContext

public static EOEditingContext **substitutionEditingContext**()

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

**See also:** **setSubstitutionEditingContext**

## usesContextRelativeEncoding

public static boolean **usesContextRelativeEncoding**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns **true** to indicate that **encodeObjectWithCoder** uses context relative encoding, **false** 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

public void **addEditor**(java.lang.Object *editor*)

Adds *editor* to the receiver's set of EOEditingContext.Editor. For more explanation, see the method description for **editors** and the EOEditingContext.Editor interface specification.

**See also:** **removeEditor**

### arrayFaultWithSourceGlobalID

public NSArray **arrayFaultWithSourceGlobalID**(
EOGlobalID *globalID*,
java.lang.String *name*,
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), and EOFaultHandler class specifications.

**See also:** **faultForGlobalID**

### committedSnapshotForObject

public NSDictionary **committedSnapshotForObject**(EOEnterpriseObject *object*)

This method is only available in Yellow Box; there is no Java Client equivalent.

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

public NSDictionary **currentEventSnapshotForObject**(EOEnterpriseObject *object*)

This method is only available in Yellow Box; there is no Java Client equivalent.

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

public java.lang.Object **delegate**()

Returns the receiver's delegate.

**See also:** **setDelegate**

## deleteObject

public void **deleteObject**(EOEnterpriseObject *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

public NSArray **deletedObjects**()

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

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

## editors

public 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 EOEditingContext.Editor interface specification and the EODisplayGroup class specification.

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

## faultForGlobalID

public EOEnterpriseObject **faultForGlobalID**(
EOGlobalID *globalID*,
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**.

If the department object is already registered in the EOEditingContext, **faultForGlobalID** 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** 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), and EOFaultHandler class specifications.

**See also:** **arrayFaultWithSourceGlobalID**

## faultForRawRow

public EOEnterpriseObject **faultForRawRow**(
java.lang.Object *row*,
java.lang.String *entityName*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns a fault for the raw row *row* by invoking **faultForRawRow** with **this** as the editing context.

## forgetObject

public void **forgetObject**(EOEnterpriseObject *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 finalized. 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

public EOGlobalID **globalIDForObject**(EOEnterpriseObject *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 **null**. Objects are registered in an EOEditingContext using the **insertObject** method, or, when fetching, with **recordObject**.

**See also: objectForGlobalID**

## hasChanges

public boolean **hasChanges**()
()

Returns **true** 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.

## initializeObject

public void **initializeObject**(
EOEnterpriseObject *object*,
EOGlobalID *globalID*,
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

public NSArray **insertedObjects**()

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

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

## insertObject

public void **insertObject**(EOEnterpriseObject *object*)

Registers (by invoking **insertObjectWithGlobalID**) *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**, **insertObjectWithGlobalID**

## insertObjectWithGlobalID

public void **insertObjectWithGlobalID**(EOEnterpriseObject *anEOEnterpriseObject*,
EOGlobalID *anEOGlobalID*)

Registers a new *object* identified by *globalID* that should be inserted in the parent EOObjectStore when changes are saved. Works by invoking **recordObject**, unless the receiver already contains the object. Sends *object* the message **awakeFromInsertion**. *globalID* must respond **true** 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

public 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 ObjectsChangedInStoreNotification. (EODatabaseContext is defined in EOAccess.)

The final effect of this method is to refault all objects currently in memory. 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:

```
EOEditingContext.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 (Yellow Box only), 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

public 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

public boolean **invalidatesObjectsWhenFreed**()
()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns **true** to indicate that the receiver clears and "booby-traps" all of the objects registered with it when the receiver is finalized, **false** otherwise. The default is **true**. 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**.

## invokeRemoteMethod

public java.lang.Object **invokeRemoteMethod**(
EOEditingContext *anEditingContext*,
EOGlobalID *globalID*,
java.lang.String *methodName*,
java.lang.Object[] *objects*)

This method is available for Java Client applications only; there is no Yellow Box equivalent.

**See also:**

## isObjectLockedWithGlobalID

public boolean **isObjectLockedWithGlobalID**(
EOGlobalID *globalID*,
EOEditingContext *anEditingContext*)

Returns **true** if the object identified by *globalID* in *anEditingContext* is locked, **false** otherwise. This method works by forwarding the message **isObjectLockedWithGlobalID** to its parent object store.

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

## lock

public void **lock**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public void **lockObject**(EOEnterpriseObject *anObject*)

Attempts to lock *anObject* in the external store. This method works by invoking **lockObjectWithGlobalID**. Throws an exception if it can't find the globalID for *anObject* to pass to **lockObjectWithGlobalID**.

**See also:** **isObjectLockedWithGlobalID**, **locksObjectsBeforeFirstModification**

## lockObjectWithGlobalID

public void **lockObjectWithGlobalID**(
EOGlobalID *globalID*,
EOEditingContext *anEditingContext*)

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

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

## locksObjectsBeforeFirstModification

public boolean **locksObjectsBeforeFirstModification**()
()

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

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

## messageHandler

public java.lang.Object **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 EOEditingContext.MessageHandler interface specification.

**See also:** **setMessageHandler**

## objectForGlobalID

public EOEnterpriseObject **objectForGlobalID**(EOGlobalID *globalID*)

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

**See also:** **globalIDForObject**

## objectsForSourceGlobalID

public NSArray **objectsForSourceGlobalID**(
    EOGlobalID *globalID*,
    java.lang.String *name*,
    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** 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

public NSArray **objectsWithFetchSpecification**(EOFetchSpecification *fetchSpecification*)
public NSArray **objectsWithFetchSpecification**(
    EOFetchSpecification *fetchSpecification*,
    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 throws 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

public void **objectWillChange**(java.lang.Object *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

public EOObjectStore **parentObjectStore**()

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

## processRecentChanges

public 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 ObjectsChangedInStoreNotification and ObjectsChangedInEditingContextNotification 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**. This method does nothing on Java Client.

## propagatesDeletesAtEndOfEvent

public boolean **propagatesDeletesAtEndOfEvent**()
()

Returns **true** if the receiver propagates deletes at the end of the event in which a change was made, **false** if it propagates deletes only right before saving changes. The default is **true**.

**See also: setPropagatesDeletesAtEndOfEvent**

## recordObject

public void **recordObject**(
EOEnterpriseObject *object*,
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

public void **redo**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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:

public void **refault**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

This method simply invokes **refaultObjects**.

## refaultObject

public void **refaultObject**(
EOEnterpriseObject *anObject*,
EOGlobalID *globalID*,
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 reference cycles between enterprise objects. When you are using Java APIs to access Objective-C Enterprise Objects Framework classes, you have to take into consideration the way objects are deallocated on the Objective-C side of the Java Bridge. This means that you might still need to break reference cycles to help keep your application's memory in check. 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 reference cycle. Note that reference cycles are automatically broken if the EOEditingContext is finalized. For more discussion of this topic, see the section “Methods for Managing the Object Graph” in the class description.

**See also:** **invalidateObjectsWithGlobalIDs**

## refaultObjects

public void **refaultObjects**()
()

Refaults all objects cached in the receiver that haven’t been inserted, deleted, or updated. Invokes **processRecentChanges**, then invokes **refaultObject** 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.

## refetch

public void **refetch**()

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

## registeredObjects

public NSArray **registeredObjects**()

Returns the enterprise objects managed by the receiver.

## removeEditor

public void **removeEditor**(java.lang.Object *anObject*)

Unregisters *editor* from the receiver. For more discussion of EOEditors, see the **editors** method description and the EOEditingContext.Editor interface specification.

**See also:** **addEditor**

## revert

public void **revert**()
()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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**

## rootObjectStore

public EOObjectStore **rootObjectStore**()

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

## saveChanges

public void **saveChanges**()
public void **saveChanges**(java.lang.Object *anObject*) (Siva only)
()

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 thrown; the error message indicates the nature of the problem.

## saveChangesInEditingContext

public 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. Throws an exception if an error occurs; the error message indicates the nature of the problem.

## setDelegate

public void **setDelegate**(java.lang.Object *anObject*)

Set the receiver's delegate to be *anObject*.

**See also:** **delegate**

## setInvalidatesObjectsWhenFreed

public void **setInvalidatesObjectsWhenFreed**(boolean *flag*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

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

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.

When you are using Java APIs to access Objective-C Enterprise Objects Framework classes, you have to take into consideration the way objects are deallocated on the Objective-C side of the Java Bridge. This means that you might still need to break reference cycles to help keep your application' the objects usage in check.

**See also:** **invalidatesObjectsWhenFreed**

## setLocksObjectsBeforeFirstModification

public void **setLocksObjectsBeforeFirstModification**(boolean *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 **false**. If *flag* is **true**, 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

**true**. 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

public void **setMessageHandler**(java.lang.Object *handler*)

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

**See also:** **messageHandler**

## setPropagatesDeletesAtEndOfEvent

public void **setPropagatesDeletesAtEndOfEvent**(boolean *flag*)

This method is only available on Yellow Box; it has no effect in Java Client.

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 **true**, 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 **false**, 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 interface 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

public void **setStopsValidationAfterFirstError**(boolean *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 **true**. Setting it to **false**

is useful if the delegate implements **editingContextShouldPresentException** to handle the presentation of aggregate exceptions.

**See also:** **stopsValidationAfterFirstError**

## setUndoManager

public void **setUndoManager**(NSUndoManager *undoManager*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Sets the receiver's NSUndoManager to *undoManager*. You might invoke this method with **null** 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

public boolean **stopsValidationAfterFirstError**()
()

Returns **true** to indicate that the receiver should stop validating after it encounters the first error, or **false** to indicate that it should continue for all objects.

**See also:** **setStopsValidationAfterFirstError**

## undo

public void **undo**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

This 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

public NSUndoManager **undoManager**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns the receiver's NSUndoManager.

**See also:** **setUndoManager**

### unlock

public void **unlock**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public 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.

### EditingContextDidSaveChangesNotification

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 |

## InvalidatedAllObjectsInStoreNotification

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 InvalidatedAllObjectsInStoreNotification 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.

## ObjectsChangedInStoreNotification

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 ObjectsChangedInStoreNotification.

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. |

## ObjectsChangedInEditingContextNotification

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

**Notification Object** The EOEditingContext

userInfo Dictionary

| Key | Value |
|---|---|
| UpdatedKey | An NSArray containing the changed objects |
| DeletedKey | An NSArray containing the deleted objects |
| InsertedKey | An NSArray containing the inserted objects |
| InvalidatedKey | 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 interface. EOEditingContext implements the EOObserving interface. For more information about the object change notification mechanism, see the EOObserving interface 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 EOEditingContext = new EOEditingContext();
```

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 = new EOEditingContext(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.
String entityName;                // Assume this exists.
EOFetchSpecification fspec;
EOObjectStoreCoordinator rootStore;
com.apple.yellow.eoaccess.EODatabaseContext dbContext;
com.apple.yellow.eoaccess.EOAdaptor adaptor;
com.apple.yellow.eoaccess.EOAdaptorContext adContext;

fspec = new EOFetchSpecification(entityName, null, null);
rootStore = (EOObjectStoreCoordinator)editingContext.rootObjectStore();
dbContext = (EODatabaseContext)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-**null** 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."

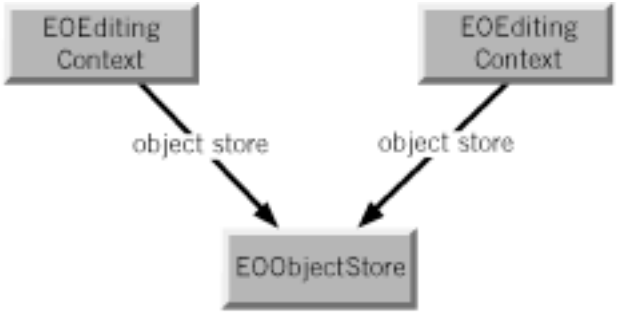



**Figure 1** Peer EOEditingContexts

#### 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.

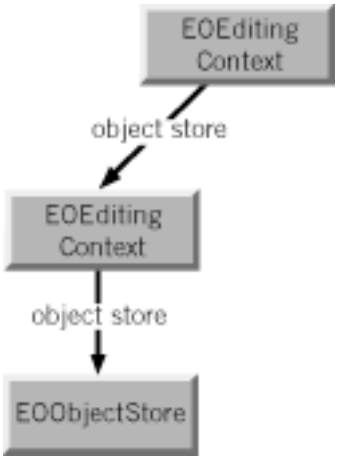



**Figure 2** Nested EOEditingContexts

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
- faultForGlobalID
- invalidateAllObjects
- invalidateObjectsWithGlobalIDs
- objectsForSourceGlobalID
- objectsWithFetchSpecification
- – 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).

## 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 arraycontaining the objects that have been inserted, deleted, and updated, respectively. The **hasChanges** method returns **true** or **false** 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 has a reference to the removed object, is referenced by the editing context and by the undo manager. The editing context releases the reference to

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 **null** 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 ObjectsChangedInStoreNotification. 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 reference 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 Reference Cycles

When you are using Java APIs to access Objective-C Enterprise Objects Framework classes, you have to take into consideration the way objects are deallocated on the Objective-C side of the Java Bridge. This means that you might still need to break reference cycles to help keep your application's memory usage in check.

You use the EOEditingContext methods **refaultObjects** and **refaultObject:withGlobalID: editingContext:** to break reference 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 reference cycle, which you can break using the **refault...** methods.

**Note:** Reference cycles are automatically broken if the EOEditingContext is finalized.

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 EOEditingContext;     // Assume this exists.
EOEditingContext.revert();           // Discard uncommitted changes.
EOEditingContext.refaultObjects();
```

Note that you must remove any other references to 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(null);
```

Using the **invalidate...** methods (described below) also has the effect of breaking reference cycles, but these methods have a more far-reaching effect. It's not recommended that you use them simply to break reference 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 **true** causes existing objects to be overwritten with fetched values that have been changed. Alternatively, you can use the EODatabaseContext (EOAccess) delegate method **databaseContextShouldUpdateCurrentSnapshot**.

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**, 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 method **refetch**, which simply invokes **invalidateAllObjects**). Unlike fetching with the EOFetchSpecification method **setRefreshesRefetchedObjects** set to **true** (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 ObjectsChangedInStoreNotification, thereby refaulting all the enterprise objects in the object graph. 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 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**.

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.
Object origObject;               // Assume this exists.
Object newObject;

newObject = childEC.faultForGlobalID(parentEC.globalIDForObject(origObject,
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 **editingContextShouldMergeChangesForObject** will be invoked for each of the objects that has both uncommitted changes and an update in the EOObjectStore. If the delegate returns **true**, the uncommitted chnages will be merged with the update (the default behavior). If it returns **false**, 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 **editingContextShouldMergeChangesForObject** 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 an instance variable for its Department object. The Department object might in turn have an instance variable an ’s 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 references. For example, moving an Employee to a different Department changes the Employee’s department instance variable 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**, 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 **equals** method. Enterprise Objects Framework relies on this method checking implementation which checks instance 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 implements the NSCoding interface, by implementing the interface's method **encodeWithCoder** and by providing a constructor that takes an NSCoder object.The enterprise object simply passes on responsibility for archiving and unarchiving itself to the EOEditingContext class, by invoking the **encodeObjectWithCoder** and **initObjectWithCoder** static methods. The

EOEditingContext takes care of the rest. For more discussion of **encodeWithCoder** and the corresponding constructor, see the NSCoding interface 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 **encodeObjectWithCoder** 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 **false**, it means the current state will always be archived, even if the enterprise object is unmodified. The default is **false** for OpenStep applications, and **true** for WebObjects applications.

# EOFaultHandler

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## Class Description

EOFaultHandler is an abstract class that defines the mechanisms that create 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.

In a Yellow Box application you create an EOFaultHandler using the standard constructor. To create a fault in a Yellow Box application, you invoke the static method **makeObjectIntoFault** 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.

In a Java Client application you also create an EOFaultHandler using the standard constructor. To create a fault in a Java Client application, though, you send a newly-created object a **turnIntoFault** message and provide an EOFaultHandler that will help the fault to fire. In order for that newly-created object to be able to respond to **turnIntoFault**, the object must conform to the EOFaulting interface. An EOFaultHandler belongs exclusively to a single fault, and shouldn't be shared or used by any other object. In Java Client applications, the fault handler is the private property of the fault; you shouldn't send any messages to the fault hander, instead dealing exclusively with the fault.

### 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 the **clearFault** 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.

## Method Types

| | |
|---|---|
| Constructors | EOFaultHandler |
| Creating and examining faults | clearFault<br>isFault<br>makeObjectIntoFault<br>handlerForFault |
| Reference counting | incrementExtraRefCount<br>decrementExtraRefCountIsZero<br>extraRefCount |
| Getting the original class | classForFault |
| Firing a fault | completeInitializationOfObject<br>faultWillFire |
| Getting a description | descriptionForObject |
| Checking class information | respondsToSelectorForFault |

## Constructors

### EOFaultHandler

public **EOFaultHandler**()

Creates and returns an EOFaultHandler object.

## Static Methods

### clearFault

public static void **clearFault**(java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Restores *aFault* to its status prior to the **makeObjectIntoFault** message that created it. Throws an exception if *aFault* isn't a fault.

You rarely use this method. Faults typically fire automatically when accessed, using the **completeInitializationOfObject** method.

### handlerForFault

public static EOFaultHandler **handlerForFault**(java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns the EOFaultHandler that will help *aFault* to fire. Returns **null** if *aFault* isn't a fault.

### isFault

public static boolean **isFault**(java.lang.Object *anObject*)

Returns **true** if *anObject* is a fault, **false** otherwise.

### makeObjectIntoFault

public static void **makeObjectIntoFault**(java.lang.Object *anObject*, EOFaultHandler *aFaultHandler*)

This method is only available in Yellow Box; there is no Java Client equivalent. Converts *anObject* into a fault, assigning *aFaultHandler* as the object that stores its original state and later converts the fault back into a normal object (typically by fetching data from an external repository). The new fault becomes the owner of *aFaultHandler*; you shouldn't assign it to another object.

### targetClassForFault

public static java.lang.Class **targetClassForFault**(java.lang.Object *anObject*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns the class that will be instantiated when the fault fires. The returned class could be a superclass of the actual class instantiated.

## Instance Methods

### classForFault

public java.lang.Class **classForFault**(java.lang.Object *fault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns the target class of the receiver's fault object, which must be passed as *aFault* in case the receiver needs to fire it

(EOFaultHandlers don’t keep references 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:** **targetClassForFault**

## completeInitializationOfObject

public void **completeInitializationOfObject**(java.lang.Object *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 a fault when it’s sent a message it can’t handle without fetching its data. EOFaultHandler’s implementation merely throws an exception.

## decrementExtraRefCountIsZero

public boolean **decrementExtraRefCountIsZero**()

This method is only available in Yellow Box; there is no Java Client equivalent. Decrements the reference count for the receiver’s fault. 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 **true**, If the reference count has not become zero, this method returns **false**. Objects that have a zero reference count are marked for garbage collection.

This method is used by EOFaultHandler’s internal reference counting mechanism.

## descriptionForObject

public java.lang.String **descriptionForObject**(java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns a string naming the original class of the receiver’s fault and giving *aFault*’s address, and also noting that it’s a fault. (The fault must be passed as *aFault* because EOFaultHandlers don’t keep references to their faults.)

## extraRefCount

public int **extraRefCount**

This method is only available in Yellow Box; there is no Java Client equivalent. Returnsthe receiver’s current reference count. This method is used by EOFaultHandler’s internal reference counting mechanism.

## faultWillFire

public void **faultWillFire**(java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Informs the receiver that *aFault* is about to be reverted to its original state. EOFaultHandler's implementation does nothing.

## incrementExtraRefCount

public void **incrementExtraRefCount**()

This method is only available in Yellow Box; there is no Java Client equivalent. Increments the reference count for the receiver's fault. 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.

**See also:** **extraRefCount**

## isKindOfClass

public boolean **isKindOfClass**(java.lang.Class *aClass*, java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns **true** if the target class of the receiver's fault 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 keep references 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

public boolean **isMemberOfClass**(java.lang.Class *aClass*, java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns **true** if the target class of the receiver's fault is *aClass*. This fault must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't keep references 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**

## respondsToSelectorForFault

public boolean **respondsToSelectorForFault**(NSSelector *aSelector*, java.lang.Object *aFault*)

This method is only available in Yellow Box; there is no Java Client equivalent. Returns **true** if the target class of the receiver's fault responds to *aSelector*. This fault must be passed as *aFault* in case the receiver needs to fire it (EOFaultHandlers don't store references 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**

## targetClass

public java.lang.Class **targetClass**()

This method is only available in Yellow Box; there is no Java Client equivalent. Returns the target class of the receiver's fault . The fault 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 faults for entities with subentities into the appropriate class on fetching their data.

# EOFetchSpecification

| | |
|---|---|
| **Inherits From:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | NSCoding (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## 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. EOAdaptorChannel and EODatabaseChannel also define methods that use EOFetchSpecifications.

## Adopted Protocols

NSCoding (Java Client only)

classForCoder
encodeWithCoder
initWithCoder

## Method Types

Constructors

EOFetchSpecification

Creating instances

fetchSpecificationWithQualifierBindings (Yellow Box only)

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 |

## Constructors

### EOFetchSpecification

public **EOFetchSpecification**()
public **EOFetchSpecification**(java.lang.String *entityName*, EOQualifier *qualifier*, NSArray *sortOrderings*)
public **EOFetchSpecification**(java.lang.String *entityName*, EOQualifier *qualifier*, NSArray *sortOrderings*, boolean *distinctFlag*, boolean *deepFlag*, NSDictionary *hints*)

Creates a new EOFetchSpecification with the arguments specified. If no arguments are provided, the new EOFetchSpecification has no state, except that it fetches deeply and doesn't use distinct. Use the **set...** methods to add other parts of the specification. Minimally, you must set the entity name.

If only *entityName*, *qualifier*, and *sortOrderings* are provided, the new EOFetchSpecification is deep, doesn't perform distinct selection, and has no hints.

## Instance Methods

### entityName

public java.lang.String **entityName**()

Returns the name of the entity to be fetched.

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

### fetchLimit

public int **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

public boolean **fetchesRawRows**()

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

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

### fetchSpecificationWithQualifierBindings

public EOFetchSpecification **fetchSpecificationWithQualifierBindings**(NSDictionary *bindings*)

This method is only available in Yellow Box; there is no equivalent in Java Client. 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 **true** to force an exception to be raised if a binding is missing during variable substitution.

**See also:** **setRequiresAllQualifierBindingVariables**

## hints

public NSDictionary **hints**()

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

**See also:** **setHints**

## isDeep

public boolean **isDeep**

Returns **true** if a fetch should include sub-entities of the receiver's entity, **false** 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

public boolean **locksObjects**

()Returns **true** if a fetch should result in the selected objects being locked in the data repository, **false** if it shouldn't. The default is **false**.

**See also:** **setLocksObjects**

## prefetchingRelationshipKeyPaths

public 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

public boolean **promptsAfterFetchLimit**()

Returns whether to prompt user after the fetch limit has been reached. Default is **false**.

**See also: setPromptsAfterFetchLimit**

## qualifier

EOQualifier **qualifier**()

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

**See also: setQualifier**

## rawRowKeyPaths

public 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** method. (Note that this faulting behavior does not occur in Java Client.)

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

## refreshesRefetchedObjects

public boolean **refreshesRefetchedObjects**()

()Returns **true** if existing objects are overwritten with fetched values when they've been updated or changed. Returns **false** if existing objects aren't touched when their data is refetched (the fetched data is simply discarded). The default is **false**. Note that this setting does not affect relationships

**See also: setRefreshesRefetchedObjects**

## requiresAllQualifierBindingVariables

public boolean **requiresAllQualifierBindingVariables**()

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

**See also:** **setRequiresAllQualifierBindingVariables**

## setEntityName

public void **setEntityName**(java.lang.String *entityName*)

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

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

## setFetchesRawRows

public void **setFetchesRawRows**(boolean *fetchRawRows*)

Sets the behavior for fetching raw rows. If set to **true**, the behavior is the same as if **setRawRowKeyPaths** were called with an empty array. If set to **false**, the behavior is as if **setRawRowKeyPaths** were called with a nil argument.

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

## setFetchLimit

public void **setFetchLimit**(int *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

public 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 CustomQueryExpressionHintKey. 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

public void **setIsDeep**(boolean *flag*)Controls whether a fetch should include sub-entities of the receiver's entity. If *flag* is **true**, sub-entities are also fetched; if *flag* is **false**, 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

public void **setLocksObjects**(boolean *flag*)

Controls whether a fetch should result in the selected objects being locked in the data repository. If *flag* is **true** it should, if **false** it shouldn't. The default is **false**.

**See also:** **locksObjects**

## setPrefetchingRelationshipKeyPaths

public 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

public void **setPromptsAfterFetchLimit**(boolean *promptsAfterFetchLimit*)

Sets whether to prompt user after the fetch limit has been reached. Default is **false**.

**See also:** **promptsAfterFetchLimit**

## setQualifier

public void **setQualifier**(EOQualifier *qualifier*)

Sets the receiver's qualifier to *qualifier*.

**See also:** **qualifier**

## setRawRowKeyPaths

public 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** method. (Note that this faulting behavior does not occur in Java Client.)

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

## setRefreshesRefetchedObjects

public void **setRefreshesRefetchedObjects**(boolean *flag*)

Controls whether existing objects are overwritten with fetched values when they have been updated or changed. If *flag* is **true**, they are; if *flag* is **false**, they aren't (the fetched data is simply discarded). The default is **false**.

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 in Yellow Box than you can with an EOFetchSpecification by using the delegate method **databaseContextShouldUpdateCurrentSnapshot**. For more information see the EODatabaseContext class specification and EODatabaseContext.Delegate interface specification.

**See also:** **refreshesRefetchedObjects**

## setRequiresAllQualifierBindingVariables

public void **setRequiresAllQualifierBindingVariables**(boolean *allVariablesRequired*)

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

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

## setSortOrderings

public 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** (EOSortOrdering), **sortArrayUsingKeyOrderArray** (EOSortOrdering), **sortOrderings**

## setUsesDistinct

public void **setUsesDistinct** (boolean *flag)*

Controls whether duplicate objects or records are removed after fetching. If *flag* is **true** they're removed; if *flag* is **false** they aren't. EOFetchSpecifications by default don't use distinct.

**See also:** **usesDistinct**

## sortOrderings

public NSArray **sortOrderings**()

**See also:** 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.**sortedArrayUsingKeyOrderArray** (EOSortOrdering), **sortArrayUsingKeyOrderArray** (EOSortOrdering), **setSortOrderings**

## usesDistinct

public boolean **usesDistinct**

(Returns **true** if duplicate objects or records are removed after fetching, **false** if they aren't. EOFetchSpecifications by default don't use distinct.

**See also:** **setUsesDistinct**

# EOGenericRecord

| | |
|---|---|
| **Inherits From:** | EOCustomObject : Object(Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | EOEnterpriseObject<br>EOKeyValueCoding (EOEnterpriseObject)<br>EOKeyValueCodingAdditions (EOEnterpriseObject)<br>EORelationshipManipulation (EOEnterpriseObject)<br>EOValidation (EOEnterpriseObject)<br>EOFaulting (EOEnterpriseObject) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## 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** as follows:

```
EOEnterpriseObject newEO;
String entityName;       // Assume this exists.

ClassDescription description =
    ClassDescription.classDescriptionForEntityName(entityName);
newEO = description.createInstanceWithEditingContext(null, null);
```

**createInstanceWithEditingContext** is preferable to using the constructor 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:

```
EOEnterpriseObject newEO;
EOEnterpriseObject existingEO;   // Assume this exists.
String relationshipName;       // Assume this exists.
ClassDescription sourceDesc = existingEO.classDescription();
ClassDescription desc =
sourceDesc.classDescriptionForDestinationKey(relationshipName);

newEO = desc.createInstanceWithEditingContext(null, null);
```

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.

## Constructors

### EOGenericRecord

public **EOGenericRecord**(EOEditingContext *anEditingContext*,
EOClassDescription *aClassDescription*, EOGlobalID *globalID*)

Creates a new EOGenericRecord. The new EOGenericRecord gets its metadata from *aClassDescription*. You should pass **null** for *anEditingContext* and *globalID*, because the arguments are optional: EOGenericRecord's implementation does nothing with them. Throws an exception if *aClassDescription* is **null**.

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

## Instance Methods

### storedValueForKey

public abstract java.lang.Object **storedValueForKey**(java.lang.String *key*)

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

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

### takeStoredValueForKey

public abstract void **takeStoredValueForKey**(
java.lang.Object *value*,
java.lang.String *key*)

Overrides the default implementation to simply invoke **takeValueForKey**.

**See also:** **takeStoredValueForKey** (EOKeyValueCoding)

### takeValueForKey

public void **takeValueForKey**(
java.lang.Object *value*,
java.lang.String *key*)

Invokes the receiver's **willChange** method, and sets the value for the property identified by *key* to *value*. If *value* is **null**, 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 **handleTakeValueForUnboundKey**. Instead, EOGenericRecord's implementation does nothing.

### valueForKey

public java.lang.Object **valueForKey**(java.lang.String *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 **null**. This method calls **willRead**.

# EOGlobalID

| | |
|---|---|
| **Inherits From:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | java.lang.Cloneable (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (WebObjects and Yellow Box) |

## 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 GlobalIDChangedNotification 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).

## Constants

The string constant, GlobalIDChangedNotification, defines the name of EOGlobalID's single notification. For more information, see the section "Notifications" below.

## Interfaces Implemented

java.lang.Cloneable (Java Client only)

## Instance Methods

### isTemporary

public boolean **isTemporary**()

Returns **false**. See the class description for more information.

## Notifications

### GlobalIDChangedNotification

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

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

# EOKeyComparisonQualifier

| | |
|---|---|
| **Inherits From:** | EOQualifier |
| **Implements:** | EOQualifierEvaluation<br>NSCoding (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## 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 implements the EOQualifierEvaluation interface, 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).

## Interfaces Implemented

| | |
|---|---|
| EOQualifierEvaluation | evaluateWithObject |
| NSCoding (Java Client only) | classForCoder<br>encodeWithCoder |

## Constructors

### EOKeyComparisonQualifier

public **EOKeyComparisonQualifier**(java.lang.String *leftKey*, NSSelector *selector*,
java.lang.String *rightKey*)

Creates and returns a new EOKeyComparisonQualifier object that compares the properties named by *leftKey* and *rightKey*, using the operator method *selector*.

- QualifierOperatorEqual
- QualifierOperatorNotEqual
- QualifierOperatorLessThan
- QualifierOperatorGreaterThan
- QualifierOperatorLessThanOrEqualTo
- QualifierOperatorGreaterThanOrEqualTo
- QualifierOperatorContains
- QualifierOperatorLike
- QualifierOperatorCaseInsensitiveLike

Enterprise Objects Framework supports SQL generation for these methods only. You can generate SQL using the SQLExpression static method **sqlStringForKeyComparisonQualifier**.

For example, the following excerpt creates an EOKeyComparisonQualifier **qual** that has the left key "lastName", the operator method 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 */
EOKeyComparisonQualifier qual = new EOKeyComparisonQualifier("lastName",
    EOQualifier.QualifierOperatorEqual,
    "member.lastName");

return EOQualifier.filteredArrayWithQualifier(guests, qual);
```

## Instance Methods

### evaluateWithObject

EOQualifierEvaluation interface
public boolean **evaluateWithObject**(java.lang.Object *object*)

Returns **true** if the object *object* satisfies the qualifier, **false** otherwise. When an EOKeyComparisonQualifier object receives an **evaluateWithObject** message, it evaluates *object* to determine if it meets the qualifier criteria. This method can throw 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).

### leftKey

public java.lang.String **leftKey**()

Returns the receiver's left key.

### rightKey

public java.lang.String **rightKey**()

Returns the receiver's right key.

### selector

public NSSelector **selector**()

Returns the receiver's selector.

# EOKeyGlobalID

**Inherits From:** EOGlobalID : Object (Java Client)
EOGlobalID : NSObject (Yellow Box)

**Implements:** com.apple.client.foundation.NSCoding (Java Client only)
java.lang.Cloneable (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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** 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.) Note that you don't use a constructor to create EOKeyGlobalIDs.

## Interfaces Implemented

NSCoding

classForCoder (Java Client only)
encodeWithCoder (Java Client only)

## Method Types

Creating instances

globalIDWithEntityName

Getting the entity name

entityName

Getting the key values

keyValues
keyCount
– keyValuesArray

Comparison

equals

## Static Methods

### globalIDWithEntityName

public static EOKeyGlobalID **globalIDWithEntityName**(
java.lang.String *entityName*,
NSArray *keyValues*)

Returns an EOKeyGlobalID based on *entityName* and *keyValues*.

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

## Instance Methods

### entityName

public java.lang.String **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**.

### equals

public boolean **equals**(java.lang.Object *anObject*)

Returns **true** if the receiver and *anObject* share the same entity name and key values, **false** if they don't.

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

### hashcode

public int **hashCode**()

Returns an integer that can be used as a table address in a hash table structure. If two objects are equal (as determined by **equals**), they must have the same hash value.

### keyCount

public int **keyCount**()

Returns the number of key values in the receiver.

### keyValues

public java.lang.Object[] **keyValues**()

Returns the receiver’s key values.

### keyValuesArray

public NSArray **keyValuesArray**()

Returns the receiver’s key values as an NSArray.

# EOKeyValueQualifier

**Inherits From:** EOQualifier

**Implements:** EOQualifierEvaluation
NSCoding (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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 implements the EOQualifierEvaluation interface, 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).

## Interfaces Implemented

EOQualifierEvaluation
evaluateWithObject

NSCoding (Java Client only)
classForCoder
encodeWithCoder

## Constructors

### EOKeyValueQualifier

public **EOKeyValueQualifier**(java.lang.String *key*, NSSelector *selector*, java.lang.Object *value*)

Creates and returns a new EOKeyValueQualifier.

If *key*, *selector*, and *value* are provided, the EOKeyValueQualifier compares values for *key* to *value* using the operator method *selector*. The possible values for *selector* are as follows:

- QualifierOperatorEqual
- QualifierOperatorNotEqual
- QualifierOperatorLessThan
- QualifierOperatorGreaterThan
- QualifierOperatorLessThanOrEqualTo
- QualifierOperatorGreaterThanOrEqualTo
- QualifierOperatorContains
- QualifierOperatorLike
- QualifierOperatorCaseInsensitiveLike

Enterprise Objects Framework supports SQL generation for these methods only. You can generate SQL using the SQLExpression static method **sqlStringForKeyValueQualifier**.

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

```
NSArray employees /* Assume this exists */
EOKeyValueQualifier qual = new EOKeyValueQualifier("name",
    EOQualifier.QualifierOperatorEqual, "Smith");
return EOQualifier.filteredArrayWithQualifier(employees, qual);
```

## Instance Methods

### evaluateWithObject

EOQualifierEvaluation interface
public boolean **evaluateWithObject**(java.lang.Object *anObject*)

Returns **true** if the object *anObject* satisfies the qualifier, **false** otherwise. When an EOKeyValueQualifier object receives the **evaluateWithObject** message, it evaluates *anObject* to determine if it meets the qualifier criteria. This method can throw 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).

### key

public java.lang.String **key**()

Returns the receiver’s key.

### selector

public NSSelector **selector**()

Returns the receiver’s selector.

### value

public java.lang.Object **value**()

Returns the receiver’s value.

# EONotQualifier

**Inherits From:** EOQualifier

**Implements:** EOQualifierEvaluation
NSCoding (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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 implements the EOQualifierEvaluation interface, 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.

## Interfaces Implemented

EOQualifierEvaluation
evaluateWithObject

NSCoding (Java Client only)
classForCoder
encodeWithCoder

## Constructors

### EONotQualifier

public com.apple.yellow.eocontrol.**EONotQualifier**(EOQualifier *aQualifier*)

Creates and returns a new EONotQualifier

If *aQualifier* is specified, it is used as the qualifier. 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;
EONotQualifier negQual;

baseQual = EOQualifier.qualifierWithQualifierFormat("lastName = member.lastName",
null);
negQual = new EONotQualifier(baseQual);
return EOQualifier.filteredArrayWithQualifier(guests, negQual);
```

## Instance Methods

### evaluateWithObject

EOQualifierEvaluation interface
public boolean **evaluateWithObject**(java.lang.Object *anObject*)

Returns **true** if the object *anObject* satisfies the EONotQualifier, **false** otherwise. This method can throw 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).

### qualifier

EOQualifier **qualifier**()

Returns the receiver's qualifier.

# EONullValue

| | |
|---|---|
| **Inherits From:** | Object (Java Client)<br>NSObject (Yellow Box) |
| **Implements:** | NSCoding (Java Client only)<br>EOSortOrderingComparison (Java Client only)<br>java.lang.Cloneable (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## Class Description

The EONullValue class defines a unique object used to represent null values in collection objects (which don't allow **null** values). For example, NSDictionaries fetched by an EOAdaptorChannel contain an EONullValue instance for such values. EONullValue is automatically translated to **null** in enterprise objects, however, so most applications should rarely need to account for this class.

EONullValue has exactly one instance, returned by the **nullValue** class method. t You can safely cache this instance and use the == operator to test for the presence of a null value:

```
EONullValue myNull = EONullValue.nullValue();
/* ... */
if (value == myNull) {
    /* ... */
}
```

## Interfaces Implemented

NSCoding (Java Client only)

– classForCoder
– encodeWithCoder

EOSortOrderingComparison

compareAscending
compareCaseInsensitiveAscending
compareCaseInsensitiveDescending
compareDescending

java.lang.Cloneable (Java Client only)

## Constructors

### EONullValue

public **EONullValue**()

Returns the unique instance of EONullValue.

## Static Methods

### nullValue

public static EONullValue **nullValue**()

Returns the unique instance of EONullValue.

# EOObjectStore

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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 throw exceptions.

## Constants

EOObjectStore defines the following string constants for the names of the notifications it posts:
- InvalidatedAllObjectsInStoreNotification
- ObjectsChangedInStoreNotification

See the Notifications section for more information on the notifications.

## Method Types

Initializing objects

initializeObject

| | |
|---|---|
| Getting objects | |
| | objectsWithFetchSpecification |
| | objectsForSourceGlobalID |
| Getting faults | |
| | faultForGlobalID |
| | arrayFaultWithSourceGlobalID |
| | refaultObject |
| | faultForRawRow (Yellow Box only) |
| Locking objects | |
| | lockObjectWithGlobalID |
| | isObjectLockedWithGlobalID |
| Saving changes to objects | |
| | saveChangesInEditingContext |
| Invalidating objects | |
| | invalidateAllObjects |
| | invalidateObjectsWithGlobalIDs |
| Interacting with the server (Java Client only) | |
| | invokeRemoteMethod (Java Client only) |

## Instance Methods

### arrayFaultWithSourceGlobalID

public abstract NSArray **arrayFaultWithSourceGlobalID**(
EOGlobalID *globalID*,
java.lang.String *relationshipName*,
EOEditingContext *anEditingContext*)

Implemented by subclasses to return the destination objects for a to-many relationship, whether as real instances or as faults (empty enterprise 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** message and that turns the fault into an array containing those objects. See the EOFaultHandler class specification 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**

## faultForGlobalID

public abstract EOEnterpriseObject **faultForGlobalID**(
EOGlobalID *globalID*,
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 empty enterprise object), you must define an EOFaultHandler subclass that stores *globalID*, uses it to fetch the object's data, and initializes the object with EOObjectStore's **initializeObject**. See the EOFaultHandler class specification 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**, **recordObject** (EOEditingContext)

## faultForRawRow

public abstract EOEnterpriseObject **faultForRawRow**(
java.lang.Object *row*,
java.lang.String *entityName*,
EOEditingContext *anEOEditingContext*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public abstract void **initializeObject**(
EOEnterpriseObject *anObject*,
EOGlobalID *globalID*,
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**

or using EOGenericRecord's or EOCustomObject's constructors. This method is also invoked after a fault has been fired.

**See also:** **awakeFromInsertionInEditingContext** (EnterpriseObject), **awakeFromFetchInEditingContext** (EnterpriseObject)

## invalidateAllObjects

public abstract void **invalidateAllObjects**()

Discards the values of all objects held by the receiver and turns them into faults (empty enterprise 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 InvalidatedAllObjectsInStoreNotification.

**See also:** **invalidateObjectsWithGlobalIDs**, **refaultObject**

## invalidateObjectsWithGlobalIDs

public abstract 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 (empty enterprise 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**

## invokeRemoteMethod

public abstract void **invokeRemoteMethod**(
EOEditingContext *anEditingContext*,
EOGlobalID *receiverGID*,
java.lang.String *methodName*,
java.lang.Object[] *arguments*)

This method is available for Java Client applications only; there is no Yellow Box equivalent.

Invokes *methodName* on the enterprise object identified by *receiverGID* in *anEditingContext*, using *arguments*. To pass an enterprise object as an argument, use its global ID. This method has the side effect of saving all the changes from the editing context all the way down to the editing context in the server session.

## isObjectLockedWithGlobalID

public abstract boolean **isObjectLockedWithGlobalID**(
EOGlobalID *globalID*,
EOEditingContext *anEditingContext*)

Returns **true** if the object identified by *globalID* is locked, **false** if it isn't. See the EODatabaseContext (EOAccess) class specification for more information on how this method works in concrete subclasses.

## lockObjectWithGlobalID

public abstract void **lockObjectWithGlobalID**(
EOGlobalID *globalID*,
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

public abstract NSArray **objectsForSourceGlobalID**(
EOGlobalID *globalID*,
java.lang.String *relationshipName*,
EOEditingContext *anEditingContext*)

Returns the destination objects for a to-many relationship. This method is used by an array fault previously constructed using **arrayFaultWithSourceGlobalID**. *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

public abstract NSArray **objectsWithFetchSpecification**(
EOFetchSpecification *aFetchSpecification*,
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. Throws 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

public abstract void **refaultObject**(
EOEnterpriseObject *anObject*,
EOGlobalID *globalID*,
EOEditingContext *anEditingContext*)

Turns *anObject* into a fault (an empty enterprise object), 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

public abstract 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

### InvalidatedAllObjectsInStoreNotification

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

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

### ObjectsChangedInStoreNotification

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

**Package:** com.apple.yellow.eocontrol (Yellow Box)

## 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. It is for use in WebObjects and Yellow Box applications only; there is no equivalent class for Java Client applications. 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
- objectsForSourceGlobalID
- faultForGlobalID
- arrayFaultWithSourceGlobalID
- refaultObject
- 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 **true** 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** 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**.

## Constants

The following string constants define the names of EOObjectStoreCoordinator's notifications:

- CooperatingObjectStoreWasAdded
- CooperatingObjectStoreWasRemoved
- CooperatingObjectStoreNeeded

For more information, see the section "Notifications" below.

## Method Types

Constructors
EOObjectStoreCoordinator

Setting the default coordinator
setDefaultCoordinator
defaultCoordinator

Managing EOCooperatingObjectStores
addCooperatingObjectStore
removeCooperatingObjectStore
cooperatingObjectStores

Saving changes
saveChangesInEditingContext

Communication between EOCooperatingObjectStores
forwardUpdateForObject
valuesForKeys

Returning EOCooperatingObjectStores
objectStoreForGlobalID
objectStoreForFetchSpecification
objectStoreForObject

Getting the userInfo dictionary
userInfo
setUserInfo

## Constructors

### EOObjectStoreCoordinator

public **EOObjectStoreCoordinator**()

Creates and returns an EOObjectStoreCoordinator.

## Static Methods

### defaultCoordinator

public static java.lang.Object **defaultCoordinator**()

Returns a shared instance of EOObjectStoreCoordinator.

### setDefaultCoordinator

public static void **setDefaultCoordinator**(EOObjectStoreCoordinator *coordinator*)

Sets a shared instance EOObjectStoreCoordinator.

## Instance Methods

### addCooperatingObjectStore

public 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 CooperatingObjectStoreWasAdded.

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

## cooperatingObjectStores

public NSArray **cooperatingObjectStores**()

Returns the receiver's EOCooperatingObjectStores.

**See also:** **addCooperatingObjectStore**, **removeCooperatingObjectStore**

## forwardUpdateForObject

public void **forwardUpdateForObject**(
java.lang.Object *object*,
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**.

## objectStoreForFetchSpecification

public EOCooperatingObjectStore **objectStoreForFetchSpecification**(
EOFetchSpecification *fetchSpecification*)

Returns the EOCooperatingObjectStore responsible for fetching objects with *fetchSpecification*. Returns **null** if no EOCooperatingObjectStore can be found that responds **true** to **handlesFetchSpecification**.

**See also:** **objectStoreForGlobalID**, **objectStoreForObject**

## objectStoreForGlobalID

public EOCooperatingObjectStore **objectStoreForGlobalID**(EOGlobalID *globalID*)

Returns the EOCooperatingObjectStore for the object identified by *globalID*. Returns **null** if no EOCooperatingObjectStore can be found that responds **true** to **ownsGlobalID**.

**See also:** **objectStoreForFetchSpecification**, **objectStoreForObject**

## objectStoreForObject

public EOCooperatingObjectStore **objectStoreForObject**(java.lang.Object *object*)

Returns the EOCooperatingObjectStore that owns *object*. Returns **null** if no EOCooperatingObjectStore can be found that responds **true** to **ownsObject**.

**See also:** **objectStoreForFetchSpecification**, **objectStoreForGlobalID**

## removeCooperatingObjectStore

public 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 CooperatingObjectStoreWasRemoved.

**See also:** **addCooperatingObjectStore**, **cooperatingObjectStores**

## saveChangesInEditingContext

public 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**, 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** 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

public void **setUserInfo**(NSDictionary *dictionary*)

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs.

**See also: userInfo**

## userInfo

public NSDictionary **userInfo**()

Returns a dictionary of user data. Your application can use this to store any auxiliary information it needs.

**See also: setUserInfo**

## valuesForKeys

public NSDictionary **valuesForKeys**(
NSArray *keys*,
java.lang.Object *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** message (through the coordinator).

## Notifications

The following notifications are declared and posted by EOObjectStoreCoordinator.

### CooperatingObjectStoreWasAdded

When an EOObjectStoreCoordinator receives an **addCooperatingObjectStore** message and adds an EOCooperatingObjectStore to its list, it posts CooperatingObjectStoreWasAdded to notify observers.

| | |
|---|---|
| **Notification Object** | The EOObjectStoreCoordinator |
| **userInfo Dictionary** | None |

### CooperatingObjectStoreWasRemoved

When an EOObjectStoreCoordinator receives a **removeCooperatingObjectStore** message and removes an EOCooperatingObjectStore from its list, it posts CooperatingObjectStoreWasRemoved to notify observers.

| | |
|---|---|
| **Notification Object** | The EOObjectStoreCoordinator |
| **userInfo Dictionary** | None |

### CooperatingObjectStoreNeeded

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:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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 ChangesEOControl provides four classes and an interface 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 interface, 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." 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 static methods.

### Registering an Observer

Objects that directly observe others must implement the EOObserving interface, which consists of the single method **objectWillChange**. To register an object as an observer, invoke EOObserverCenter's **addObserver** 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 defined by the EOEnterpriseObject interface. 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
- removeObserver
- addOmniscientObserver
- removeOmniscientObserver

Notifying observers of change
- notifyObserversObjectWillChange

Getting observers
- observersForObject
- observerForObject

Suppressing change notification
- suppressObserverNotification
- enableObserverNotification
- observerNotificationSuppressCount

## Static Methods

### addObserver

public static void **addObserver**(
EOObserving *anObserver*,
java.lang.Object *anObject*)

Records *anObserver* to be notified with an **objectWillChange** message when *anObject* changes.

**See also:** **removeObserver**

## addOmniscientObserver

public static void **addOmniscientObserver**(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 **null** argument.

**See also:** **addObserver**, **removeOmniscientObserver**

## enableObserverNotification

public static void **enableObserverNotification**()

Counters a prior **suppressObserverNotification** message. When no such messages remain in effect, the **notifyObserversObjectWillChange** method is re-enabled. Throws an exception if not paired with a prior **suppressObserverNotification** message.

## notifyObserversObjectWillChange

public static void **notifyObserversObjectWillChange**(java.lang.Object *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(null);
```

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**, **addOmniscientObserver**

## observerForObject

public static EOObserving **observerForObject**(
    java.lang.Object *anObject*,
    java.lang.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

public static int **observerNotificationSuppressCount**()

Returns the number of **suppressObserverNotification** messages in effect.

**See also:** **enableObserverNotification**

## observersForObject

public static NSArray **observersForObject**(java.lang.Object *anObject*)

Returns all observers of *anObject*.

## removeObserver

public static void **removeObserver**(
EOObserving *anObserver*,
java.lang.Object *anObject*)

Removes *anObserver* as an observer of *anObject*.

**See also:** **addObserver**

## removeOmniscientObserver

public static void **removeOmniscientObserver**(EOObserving *anObserver*)

Unregisters *anObserver* as an observer of all objects.

**See also:** **removeObserver**, **addOmniscientObserver**

## suppressObserverNotification

public static 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 : Object (Java Client)
EODelayedObserver : NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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 ChangesEOControl provides four classes and an interface 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 interface, 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." 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.

## Constructors

### EOObserverProxy

public **EOObserverProxy**(
java.lang.Object *anObject*,
NSSelector *anAction*,
int *priority*)

Creates 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 java.util.Object argument and returning **void**. *priority* indicates when the receiver is sent this message from EODelayedObserverQueue's **notifyObserversUpToPriority** method.

# EOOrQualifier

| | |
|---|---|
| **Inherits From:** | EOQualifier |
| **Implements:** | EOQualifierEvaluation<br>NSCoding (Java Client only) |
| **Package:** | com.apple.client.eocontrol (Java Client)<br>com.apple.yellow.eocontrol (Yellow Box) |

## Class Description

EOOrQualifier is a subclass of EOQualifier that contains multiple qualifiers. EOOrQualifier implements the EOQualifierEvaluation interface, 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 true. If one of its qualifiers returns true, the EOOrQualifier object returns true immediately. If all of its qualifiers return false, the EOOrQualifier object returns false.

## Interfaces Implemented

| | |
|---|---|
| EOQualifierEvaluation | evaluateWithObject |
| NSCoding (Java Client only) | classForCoder<br>encodeWithCoder |

## Constructors

### EOOrQualifier

public **EOOrQualifier**(NSArray *qualifiers*)

Creates and returns a new EOOrQualifier.

If *qualifiers* is specified, the EOOrQualifier is initialized with the qualifiers in *qualifiers*.

## Instance Methods

### evaluateWithObject:

EOQualifierEvaluation interface
public boolean **evaluateWithObject**(java.lang.Object *anObject*)

Returns **true** if *anObject* satisfies the qualifier, **false** otherwise. When an EOOrQualifier object receives an **evaluateWithObject** message, it evaluates each of its qualifiers until one of them returns **true**. If any of its qualifiers returns **true**, the EOOrQualifier object returns **true** immediately. If all of its qualifiers return **false**, the EOOrQualifier object returns **false**. This method can throw 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).

### qualifiers

NSArray **qualifiers**()

Returns the receiver's qualifiers.

# EOQualifier

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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. |

The interface 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 NSSelector constants are defined to represent the different qualifier operators:

| | |
|---|---|
| QualifierOperatorEqual | QualifierOperatorLessThanOrEqualTo |
| QualifierOperatorNotEqual | QualifierOperatorGreaterThanOrEqualTo |
| QualifierOperatorLessThan | QualifierOperatorContains |
| QualifierOperatorGreaterThan | QualifierOperatorLike |
| | QualifierOperatorCaseInsensitiveLike |

## Method Types

Constructors

EOQualifier

Creating a qualifier

qualifierWithQualifierFormat (Yellow Box only)
qualifierToMatchAllValues
qualifierToMatchAnyValue
qualifierWithBindings

| | |
|---|---|
| In-memory filtering | |
| | filteredArrayWithQualifier |
| | filterArrayWithQualifier |
| Converting strings and operators | |
| | operatorSelectorForString |
| | stringForOperatorSelector |
| Get EOQualifier operators | |
| | allQualifierOperators |
| | relationalQualifierOperators |
| Accessing a qualifier's keys | |
| | bindingKeys |
| | keyPathForBindingKey |
| Validating a qualifier's keys | |
| | validateKeysWithRootClassDescription |

## Constructors

### EOQualifier

public **com.apple.yellow.eocontrol.EOQualifier**(java.lang.String *qualifierFormat*, next.util.ImmutableVector *arguments*)

Creates and returns a new EOQualifier object. Parses the format string *qualifierFormat* and the specified *arguments*, initializes the new EOQualifier with them, and returns that EOQualifier. Conversion specifications (occurrences of %@) in *qualifierFormat* are replaced using the value objects in *arguments*. For more information on how *qualifierFormat* and *arguments* are used, see the method description for the static method **qualifierWithQualifierFormat**.

You would never use this constructor to create an EOQualifier. Instead, you'd use the static method **qualifierWithQualifierFormat** to create an instance of one of the qualifier subclasses.

A subclass of EOQualifier should write a constructor with the same *formatString* and *arguments* arguments that invokes the EOQualifier implementation.

## Static Methods

### allQualifierOperators

public static NSArray **allQualifierOperators**()

Returns an NSArray containing all of the operators supported by EOQualifier: =, !=, <, <=, >, >=, “like”, and “caseInsensitiveLike”.

**See also:** **relationalQualifierOperators**

### filterArrayWithQualifier

public static void **filterArrayWithQualifier**(NSMutableArray *objects*, EOQualifier *aQualifier*)

Filters *objects* in place so that it contains only objects matching *aQualifier.*

**See also:** **filteredArrayWithQualifier**

### filteredArrayWithQualifier

public static NSArray **filteredArrayWithQualifier**(NSArray *objects*, EOQualifier *aQualifier*)

Returns a new array that contains only the objects from *objects* matching *aQualifier.*

**See also:** **filterArrayWithQualifier**

### operatorSelectorForString

public static NSSelector **operatorSelectorForString**(java.lang.String *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 QualifierOperatorNotEqual.

```
Selector selector = Qualifier.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

public static EOQualifier **qualifierToMatchAllValues**(NSDictionary *aNSDictionary*)

This method is only available in Yellow Box; there is no equivalent in Java Client. 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

public static EOQualifier **qualifierToMatchAnyValue**(NSDictionary *aNSDictionary*)

This method is only available in Yellow Box; there is no equivalent in Java Client. 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

public static EOQualifier **qualifierWithQualifierFormat**(java.lang.String *qualifierFormat*,
NSArray *arguments*)

This method is only available in Yellow Box; there is no equivalent in Java Client. Parses the format string *qualifierFormat* and the specified *arguments*, uses them to create an EOQualifier, and returns the EOQualifier. Conversion specifications (occurrences of %@) in *qualifierFormat* are replaced using the value objects in *arguments*.

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.*/
EOQualifier qualifier;
EOFetchSpecification fetchSpec;
EODatabaseDataSource dataSource;

dataSource = (EODatabaseDataSource)displayGroup.dataSource();
qualifier = EOQualifier.qualifierWithQualifierFormat("cardType = 'Visa'");
fetchSpec = new EOFetchSpecification("Member", qualifier, null), null);

dataSource.setFetchSpecification(fetchSpec);
displayGroup.fetch();
```

**qualifierWithQualifierFormat** performs no verification to ensure that keys referred to by the format string *qualifierFormat* exist. It throws an exception if *qualifierFormat* contains any syntax errors.

## relationalQualifierOperators

public static 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

public static java.lang.String **stringForOperatorSelector**(NSSelector *aSelector*)

Returns a string representation of the selector *aSelector*. For example, the following statement returns the string “!=”:

```
java.lang.String operator =
    EOQualifier.stringForOperatorSelector(EOQualifier.QualifierOperatorNotEqual);
```

The possible values for *selector* are as follows:

- QualifierOperatorEqual
- QualifierOperatorNotEqual
- QualifierOperatorLessThan
- QualifierOperatorGreaterThan
- QualifierOperatorLessThanOrEqualTo
- QualifierOperatorGreaterThanOrEqualTo
- QualifierOperatorContains
- QualifierOperatorLike
- QualifierOperatorCaseInsensitiveLike

You'd probably only use this method if you were writing your own parser.

**See also:** **operatorSelectorForString**

## Instance Methods

### bindingKeys

NSArray **bindingKeys**()

This method is only available in Yellow Box; there is no equivalent in Java Client. 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

public java.lang.String **keyPathForBindingKey**(java.lang.String *key*)

This method is only available in Yellow Box; there is no equivalent in Java Client. 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

public abstract EOQualifier **qualifierWithBindings**(NSDictionary *bindings*, boolean *requiresAll*)

This method is only available in Yellow Box; there is no equivalent in Java Client. Returns a new qualifier substituting all variables with values found in *bindings*. If *requiresAll* is YES, any variable not found in *bindings* will throw a QualifierVariableSubstitutionException. If *requiresAll* is NO, missing variable values will cause the qualifier node to be pruned from the tree.

### validateKeysWithRootClassDescription

public abstract java.lang.Throwable
**validateKeysWithRootClassDescription**(EOClassDescription *classDesc*)

This method is only available in Yellow Box; there is no equivalent in Java Client. Validates that the receiver contains keys and key paths that belong to or originate from *classDesc*. This method returns an exception if an unknown key is found, otherwise it returns **null** 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 simplycreate a qualifier using the EOQualifier static method **qualifierWithQualifierFormat:**, as follows:

```
EOQualifier qual = Qualifier.qualifierWithQualifierFormat("lastName = 'Smith'",
null);
```

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.

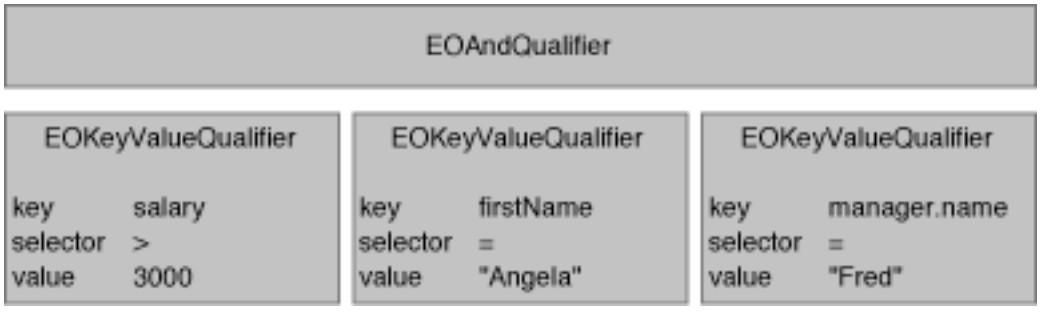



**Figure 4** EOQualifier Tree for **salary > 300 AND firstName = “Angela” AND manager.name = “Fred”**

**Note:** The **qualifierWithQualifierFormat** method can’t be used to create an instance of EOSQLQualifier. This is because EOSQLQualifier uses a non-structured syntax. It also requires an entity. To create an instance of EOSQLQualifier, you’d use a statement such as the following:

```
EOQualifier myQual = new EOSQLQualifier(myEntity, 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 other objects using the conversion specification %@. The second argument to **qualifierWithQualifierFormat** is an array that contains the value or result to substitute for any %@ conversion specifications. 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 |
|---|---|
| %@ | It can either be an object whose **toString** (or **description**) method returns a key (in other words, a java.lang.String), or a value object such as an java.lang.String, java.lang.Number, java.util.CalendarDate, and so on. |
| %% | Results in a literal **%** character. |

**Note:** If you use an unrecognized character in a conversion specification (for example, %x), an exception is thrown.

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:

```
Employee anEmployee;    // Assume this exists.
Department aDept;       // Assume this exists.
EOQualifier myQualifier;
MutableVector args = new MutableVector();

args.addElement("lastName");
args.addElement(anEmployee.lastName());
myQualifier = EOQualifier.qualifierWithQualifierFormat("%@ = %@", args);

args.removeAllElements();
args.addElement("salary");
args.addElement(anEmployee.salary());
myQualifier = EOQualifier.qualifierWithQualifierFormat("%@ > %f", args);

args.removeAllElements();
args.addElement("department.name");
args.addElement(aDept.name());
myQualifier = EOQualifier.qualifierWithQualifierFormat("%@ = %@", args);
```

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
args.removeAllElements();
args.addElement("Jones");
EOQualifier.qualifierWithQualifierFormat("lastName = %@", args);
```

Typically format strings include only two data types: strings and numbers. Single-quoted or double-quoted strings correspond to java.lang.String objects in the argument array, non-quoted numbers correspond to java.lang.Numbers, 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 NULLvalues, use either of the approaches shown in the following example:

```
NSMutableArrayargs = new NSMutableArray();

// Approach 1
EOQualifier.qualifierWithQualifierFormat("bonus = nil", null);

// Approach 2
args.addElement(NullValue.nullValue());
EOQualifier.qualifierWithQualifierFormat("bonus = %@", args);
```

## 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*'"
```

## Using Selectors in Qualifier Expressions

The format strings you use to initialize a qualifier can include methods. The parser recognizes an unquoted string followed by a colon (such as **myMethod:**) as a method. For example:

```
point1 isInside: area
firstName isAnagramOfString: "Computer"
```

Methods specified in a qualifier are parsed and applied only in memory; that is, they can't be used in to qualify fetches in a database.

## 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:

```
"dateReleased < (NSCalendarDate)'1990-01-26 00:00:00 +0000'"
"salary = (NSDecimalNumber)'15000.02'"
```

When you use this approach, qualifiers are constructed by looking up the class and invoking a constructor that takes a java.lang.String argument. Therefore, this technique only works for classes that have such a constructor.

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. This limitation doesn't apply when you're working with NSCalendarDate objects—you can just construct a qualifier in the usual way:

```
NSMutableArray args = new NSMutableArray();
args.addElement(new CalendarDate());
qual = EOQualifier.qualifierWithQualifierFormat("dateReleased > %@", args);
```

## 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 1 (meaning true). In the excerpt, the qualifier is used to filter an in-memory array.

```
// Create the qualifier
EOQualifier qual = new KeyValueQualifier("isOut", Qualifier.QualifierOperatorEqual,
    new Integer(1));

// Filter an array and return it
return Qualifier.filteredVectorWithQualifier(allRentals(), qual);
```

**filteredArrayWithQualifier** is a method that returns an array containing objects from the provided array that match the provided qualifier.

## Creating Subclasses

A custom subclass of EOQualifier must implement the EOQualifierEvaluation interface if they are to be evaluated in memory. <<Would they have to do something special to be translatable to SQL?>>

# EOSortOrdering

**Inherits From:** Object (Java Client)
NSObject (Yellow Box)

**Implements:** NSCoding (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## 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. EOFetchSpecification objects use 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 |
|---|---|
| CompareAscending | (*key*) asc |
| CompareDescending | (*key*) desc |
| CompareCaseInsensitiveAscending | upper(*key*) asc |
| CompareCaseInsensitiveDescending | upper(*key*) desc |

Using the mapping in the table above, the array of EOSortOrderings (**nameOrdering**) created in the following code example:

```
EOSortOrdering lastNameOrdering =
    EOSortOrdering.sortOrderingWithKey("lastName", EOSortOrdering.CompareAscending);
EOSortOrdering firstNameOrdering =
    (EOSortOrdering.sortOrderingWithKey("firstName",
EOSortOrdering.CompareAscending);
NSMutableArray nameOrdering = new NSMutableArray();
nameOrdering.addObject(lastNameOrdering);
nameOrdering.addObject(firstNameOrdering);
```

results in this ORDER BY clause:

```
order by (lastName) asc, (firstName) asc
```

## In-Memory Sorting

The methods **sortedArrayUsingKeyOrderArray** and **sortArrayUsingKeyOrderArray** are used to sort objects in memory. Given an array of objects and an array of EOSortOrderings, **sortedArrayUsingKeyOrderArray** returns a new array of objects sorted according to the specified EOSortOrderings. Similarly, **sortArrayUsingKeyOrderArray** sorts the provided array of objects in place. This code fragment, for example, sorts an array of Employee objects in place, by last name, then first name using the array of EOSortOrderings created above:

```
SortOrdering.sortVectorUsingKeyOrderVector(employees, nameOrdering);
```

## Comparison Methods

The predefined comparison methods are:

| Defined Name | Method |
|---|---|
| CompareAscending | compareAscending |
| CompareDescending | compareDescending |
| CompareCaseInsensitiveAscending | compareCaseInsensitiveAscending |
| CompareCaseInsensitiveDescending | compareCaseInsensitiveDescending |

The first two can be used with any value class; the second two with java.lang.String 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.

## Interfaces Implemented

NSCoding (Java Client only)

encodeWithCoder

## Method Types

Constructors

EOSortOrdering

Creating instances

sortOrderingWithKey

Examining a sort ordering

key
selector

In-memory sorting

sortedArrayUsingKeyOrderArray
sortArrayUsingKeyOrderArray

## Constructors

### EOSortOrdering

public **EOSortOrdering**(java.lang.String *key*, NSSelector *selector*)

Creates and returns a new EOSortOrdering object. If *key* and *selector* are provided, the new EOSortOrdering is initialized with them.

**See also:** **sortOrderingWithKey**

## Static Methods

### sortArrayUsingKeyOrderArray

public static void **sortArrayUsingKeyOrderArray**(NSMutableArray *objects*, NSArray *sortOrderings*)

Sorts *objects* in place according to the EOSortOrderings in *sortOrderings*. The objects are compared by extracting the sort properties using the EnterpriseObject method **valueForKey** and sending them **compare...** messages. See the table in “Sorting with SQL” for a list of the compare methods.

**See also:** **sortedArrayUsingKeyOrderArray**

### sortOrderingWithKey

public static EOSortOrdering **sortOrderingWithKey**(java.lang.String *key*, NSSelector *selector*)

Creates and returns an EOSortOrdering based on *key* and *selector.*

**See also:** "Constructors"

### sortedArrayUsingKeyOrderArray

public static NSArray **sortedArrayUsingKeyOrderArray**(NSArray *objects*, NSArray *sortOrderings*)

Creates and returns a new array by sorting *objects* according to the SortOrderings in *sortOrderings*. The objects are compared by extracting the sort properties using the added NSObject method **valueForKey** and sending them **compare...** messages. See the table in "Sorting with SQL" for a list of the compare methods.

## Instance Methods

### key

public java.lang.String **key**()

Returns the key by which the receiver orders items.

**See also:** **selector**

### selector

public NSSelector **selector**()

Returns the method selector used to compare values when sorting.

**See also:** **key**

# EOTemporaryGlobalID

**Inherits From:** EOGlobalID : NSObject

**Implements:** java.lang.Cloneable (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## 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.

## Constructors

### EOTemporaryGlobalID

public **EOTemporaryGlobalID**()

Creates and returns an EOTemporaryGlobalID as a unique instance. The returned object contains a byte string 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. The returned byte string has the format:

```
< Sequence [2], ProcessID [2] , Time [4], IP Addr [4] >
```

## Instance Methods

### isTemporary

public boolean **isTemporary**()

Returns **true**.

# EOClassDescription.ClassDelegate

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## Interface Description

The EOClassDescription.ClassDelegate interface defines a method that the EOClassDescription class can invoke in its delegate. Delegates are not required to provide an implementation for the method, and you don't have to use the **implements** keyword to specify that the object implements the ClassDelegate interface. 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

public abstract boolean **shouldPropagateDeleteForObject**(
EOEnterpriseObject *anObject*,
EOEditingContext *anEditingContext*,
java.lang.String *key*);

Invoked from **propagateDeleteForObject**. If the class delegate returns **false**, 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 **false** and not propagating deletion can lead to dangling references in your object graph.

# EOEditingContext.Delegate

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Interface Description

The EOEditingContext.Delegate interface defines methods that an EOEditingContext can invoke in its delegate. Delegates are not required to provide implementations for all of the methods in the interface, and you don't have to use the **implements** keyword to specify that the object implements the Delegate interface. Instead, declare and implement any subset of the methods declared in the interface 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
- editingContextShouldFetchObjects

Invalidating objects
- editingContextShouldInvalidateObject (Yellow Box only)

Saving changes
- editingContextWillSaveChanges

Handling failures
- editingContextShouldValidateChanges
- editingContextShouldPresentException
- editingContextShouldUndoUserActionsAfterFailure (Yellow Box only)

Merging changes (Yellow Box only)
- editingContextShouldMergeChangesForObject (Yellow Box only)
- editingContextDidMergeChanges (Yellow Box only)

## Instance Methods

### editingContextDidMergeChanges

public abstract void **editingContextDidMergeChanges**(EOEditingContext *anEditingContext*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Invoked once after a batch of objects has been updated in *anEditingContext*'s parent object store (in response to a ObjectsChangedInStoreNotification). A delegate might implement this method to define custom merging behavior, most likely in conjunction with **editingContextShouldMergeChangesForObject**. It is safe for this method to make changes to the objects in the editing context.

## editingContextShouldFetchObjects

public abstract NSArray **editingContextShouldFetchObjects**(
EOEditingContext *editingContext*,
EOFetchSpecification *fetchSpecification*)

Invoked from **objectsWithFetchSpecification**. If the delegate has appropriate results cached it can return them and the fetch will be bypassed. Returning **null** causes the fetch to be propagated to the parent object store.

## editingContextShouldInvalidateObject

public abstract boolean **editingContextShouldInvalidateObject**(
EOEditingContext *anEOEditingContext*,
EOEnterpriseObject *anObject*,
EOGlobalID *anEOGlobalID*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Sent when an *object* identified by *globalID* has been explicitly invalidated. If the delegate returns **false**, the invalidation is refused. This allows the delegate to selectively override object invalidations.

**See also:** **invalidateAllObjects**, **revert**

## editingContextShouldMergeChangesForObject

public abstract boolean **editingContextShouldMergeChangesForObject**(
EOEditingContext *anEditingContext*,
EOEnterpriseObject *object*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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 **true**, 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 **editingContextShouldInvalidateObject** will not be sent for the object in question.

If this method returns **false**, 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 **editingContextShouldMergeChangesForObject** to save information about each changed object and return **true** to allow merging to continue. After the default merging behavior occurs, **editingContextDidMergeChanges** is invoked, at which point you implement your custom behavior.

## editingContextShouldPresentException

Java Client:
public abstract boolean **editingContextShouldPresentException**(
    EOEditingContext *anEditingContext*,
    java.lang.Exception *exception*)
Yellow Box:
public abstract boolean **editingContextShouldPresentException**(
    EOEditingContext *anEditingContext*,
    java.lang.Throwable *exception*)

Sent whenever an exception is caught by an EOEditingContext. If the delegate returns **false**, *exception* is ignored. Otherwise (if the delegate returns **true**, 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

public abstract boolean **editingContextShouldUndoUserActionsAfterFailure**(
    EOEditingContext *anEditingContext*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Sent when a validation error occurs while processing a **processRecentChanges** message. If the delegate returns **false**, 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 **false**, 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 **false** 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

public abstract boolean **editingContextShouldValidateChanges**(
EOEditingContext *anEditingContext*)

Sent when an EOEditingContext receives a **saveChanges** message. If the delegate returns **false**, changes are saved without first performing validation. This method can be useful if the delegate wants to provide its own validation mechanism.

## editingContextWillSaveChanges

public abstract void **editingContextWillSaveChanges**(EOEditingContext *editingContext*)

Sent when an EOEditingContext receives a **saveChanges** message. The delegate can throw an exception to abort the save operation.

# EOEditingContext.Editor

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## Interface Description

The EOEditingContext.Editor interface 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 throwing an exception to abort the save). See the EOEditingContext and EODisplayGroup (EOInterface) class specifications for more information.

Editors are not required to provide implementations for all of the methods in the interface. When you write an editor, you don't have to use the **implements** keyword to specify that the object implements the Editors interface. Instead, simply use the EOEditingContext method **addEditor** method to assign your object as one of the EOEditingContext's editors and then declare and implement any subset of the methods declared in the Editors interface. An EOEditingContext can determine if the editor doesn't implement a method and only attempts to invoke the methods the editor actually implements.

## Instance Methods

### editingContextWillSaveChanges

public abstract 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, throwing an exception if it can't do so to prevent *anEditingContext* from saving.

### editorHasChangesForEditingContext

public abstract boolean **editorHasChangesForEditingContext**(EOEditingContext *anEditingContext*)

Invoked by *anEditingContext*, this method should return **true** if the receiver has any unapplied edits that need to be saved, **false** if it doesn't.

# EOEnterpriseObject

**Implemented By:** EOCustomObject
EOGenericRecord

**Implements:** EOFaulting
EOKeyValueCoding (EOKeyValueCodingAdditions)
EOKeyValueCodingAdditions
EORelationshipManipulation
EOValidation

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Interface Description

The EOEnterpriseObject interface 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 interfaces and incorporated in the overarching EOEnterpriseObject interface:

- 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 interface, giving access to groups of properties and to properties across relationships.
- EORelationshipManipulation builds on the basic EOKeyValueCoding interface to allow you to modify to-many relationship properties.
- EOValidation defines the way that enterprise objects validate their values.
- EOFaulting forms a general mechanism for postponing an object's initialization until its actually needed.

The remaining methods are introduced in the EOEnterpriseObject interface 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 interface from scratch. The Framework provides default implementations of the methods in EOCustomObject and EOGenericRecord. Use EOGenericRecords to represent enterprise objects that don't require custom behavior, and create subclasses of EOCustomObject 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.

## Interfaces Implemented

| | |
|---|---|
| EOKeyValueCoding | |
| | handleQueryWithUnboundKey<br>handleTakeValueForUnboundKey<br>storedValueForKey<br>takeStoredValueForKey<br>takeValueForKey<br>unableToSetNullForKey<br>valueForKey |
| EOKeyValueCodingAdditions | |
| | takeValueForKeyPath<br>takeValuesFromDictionary<br>valueForKeyPath<br>valuesForKeys |
| EORelationshipManipulation | |
| | addObjectToBothSidesOfRelationshipWithKey<br>addObjectToPropertyWithKey<br>removeObjectFromBothSidesOfRelationshipWithKey<br>removeObjectFromPropertyWithKey |
| EOValidation | |
| | validateForDelete<br>validateForInsert<br>validateForSave<br>validateForUpdate<br>validateValueForKey |
| EOFaulting | |
| | clearFault<br>isFault<br>turnIntoFault<br>willRead |

## Method Types

Initializing enterprise objects

awakeFromFetch
awakeFromInsertion

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 (Yellow Box only)

changesFromSnapshot (Yellow Box only)
reapplyChangesFromDictionary (Yellow Box only)

Invoking behavior on the server (Java Client only)

invokeRemoteMethod (Java Client only)

Getting descriptions

eoDescription
eoShallowDescription
userPresentableDescription (Yellow Box only)

## Instance Methods

### allPropertyKeys

public abstract NSArray **allPropertyKeys**()

Returns all of the receiver's property keys. EOCustomObject's implementation returns the union of the keys returned by **attributeKeys**, **toOneRelationshipKeys**, and **toManyRelationshipKeys**.

### attributeKeys

public abstract NSArray **attributeKeys**()

Returns the names of the receiver's attributes (not relationship properties). EOCustomObject'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**

### awakeFromFetch

public abstract void **awakeFromFetch**(EOEditingContext *anEditingContext*)

Overridden by subclasses to perform additional initialization on the receiver upon its being fetched from the external repository into *anEditingContext*. EOCustomObject's implementation merely sends an **awakeObjectFromFetch** to the receiver's EOClassDescription. Subclasses should invoke **super**'s implementation before performing their own initialization.

### awakeFromInsertion

public abstract void **awakeFromInsertion**(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. EOCustomObject's implementation merely sends an **awakeObjectFromInsertion** to the receiver's EOClassDescription. Subclasses should invoke **super**'s implementation before performing their own initialization.

## changesFromSnapshot

public abstract NSDictionary **changesFromSnapshot**(NSDictionary *snapshot*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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

public abstract EOClassDescription **classDescription**()

Returns the EOClassDescription registered for the receiver's class.EOCustomObject's implementation invokes the EOClassDescription static method a **classDescriptionForClass**.

## classDescriptionForDestinationKey

public abstract EOClassDescription **classDescriptionForDestinationKey**(java.lang.String *key*)

Returns the EOClassDescription for the destination objects of the relationship identified by *key.* EOCustomObject's implementation sends a **classDescriptionForDestinationKey** message to the receiver's EOClassDescription.

## clearProperties

public abstract void **clearProperties**()

Sets all of the receiver's to-one and to-many relationships to **null**. EOEditingContexts use this method to break cyclic references among objects when they're finalized. EOCustomObject'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 finalized.

## deleteRuleForRelationshipKey

public abstract int **deleteRuleForRelationshipKey**(java.lang.String *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:

- DeleteRuleNullify
- DeleteRuleNullify
- DeleteRuleNullify
- DeleteRuleNullify

For example, an Invoice object might return DeleteRuleNullify 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, the class in which they're defined.

EOCustomObject's implementation of this method simply sends a **deleteRuleForRelationshipKey** message to the receiver's EOClassDescription.

**See also:** **propagateDeleteWithEditingContext**, **validateForDelete** (EOValidation)

## editingContext

public abstract EOEditingContext **editingContext**()

Returns the EOEditingContext that holds the receiver.

## entityName

public abstract java.lang.String **entityName**()

Returns the name of the receiver's entity, or **null** if it doesn't have one. EOCustomObject's implementation simply sends an **entityName** message to the receiver's EOClassDescription.

## eoDescription

public abstract java.lang.String **eoDescription**()

Returns a string that describes the receiver. EOCustomObject'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 **toString** 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

public abstract java.lang.String **eoShallowDescription**()

Similar to **eoDescription**, but doesn't descend into relationships. **eoDescription** invokes this method for relationship destinations to avoid infinite recursion through cyclical relationships. EOCustomObject's implementation simply returns a string containing the receiver's class and entity names.

**See also:** **userPresentableDescription**

## inverseForRelationshipKey

public abstract java.lang.String **inverseForRelationshipKey**(java.lang.String *relationshipKey*)

Returns the name of the relationship pointing back to the receiver's class or entity from that named by *relationshipKey*, or **null** 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. EOCustomObject'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"))
       return "member";
    else
       return super.inverseForRelationshipKey(relationshipKey);
}
```

## invokeRemoteMethod

public abstract java.lang.Object **invokeRemoteMethod**(
    java.lang.String *methodName*,
    java.lang.Object[] *arguments*)

This method is available for Java Client applications only; there is no Yellow Box equivalent.

Invokes *methodName* using *arguments*. To pass an enterprise object as an argument, use its global ID. This method has the side effect of saving all the changes from the receiver's editing context all the way down to the editing context in the server session.

### isToManyKey

public abstract boolean **isToManyKey**(java.lang.String *key*)

Returns **true** if the receiver has a to-many relationship identified by *key*, **false** otherwise. EOCustomObject's implementation of this method simply checks its **toManyRelationshipKeys** array for *key*.

### ownsDestinationObjectsForRelationshipKey

public abstract boolean **ownsDestinationObjectsForRelationshipKey**(java.lang.String *key*)

Returns **true** if the receiver has a relationship identified by *key* that owns its destination, **false** 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. EOCustomObject'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

public abstract void **propagateDeleteWithEditingContext**(EOEditingContext *anEditingContext*)

Deletes the destination objects of the receiver's relationships according to the delete rule for each relationship. EOCustomObject's implementation simply sends a **propagateDeleteForObject** 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

public abstract void **reapplyChangesFromDictionary**(NSDictionary *changes*)

This method is available for Yellow Box applications only; there is no Java Client equivalent.

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. EOCustomObject's implementation should be sufficient for all purposes; you shouldn't have to override this method.

**See also:** **changesFromSnapshot**

## snapshot

public abstract 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 EONullValue substituted for **null**. For to-many relationships, the dictionary contains shallow copies of the arrays. EOCustomObject's implementation should be sufficient for all purposes; you shouldn't have to override this method.

**See also:** **updateFromSnapshot**

## toManyRelationshipKeys

public abstract NSArray **toManyRelationshipKeys**()

Returns the names of the receiver's to-many relationships. EOCustomObject'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

public abstract NSArray **toOneRelationshipKeys**()

Returns the names of the receiver's to-one relationships. EOCustomObject'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

public abstract void **updateFromSnapshot**(NSDictionary *aSnapshot*)

Takes the values from *aSnapshot*, and sets the receiver's properties to them. EOCustomObject's implementation sets each one using **takeStoredValueForKey**. In the process, EONullValues are converted to **null**, and array values are set as shallow mutable copies.

**See also: snapshot**

## userPresentableDescription

public abstract java.lang.String **userPresentableDescription**()

This method is available for Yellow Box applications only; there is no Java Client equivalent.

Returns a short (no longer than 60 characters) description of an enterprise object based on its data. EOCustomObject'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

public abstract 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:

```
public void setRoleName(String value) {
    willChange();
    roleName = value;
}
```

In Java Client, this method invokes **willRead:**.

# EOEnterpriseObject

## Initialization

The Framework creates enterprise objects with a constructor of the following form:

public **MyClass**(
EOEditingContext *anEditingContext*,
EOClassDescription *classDescription*,
EOGlobalID *globalID*)

This constructor should create a new instance of your enterprise object class with the provided arguments and it can perform any custom initialization that you require. Enterprise objects created in a Java client (with Java Client) and enterprise objects created on the server (with Yellow Box) require this constructor.

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 **awakeFromFetch** message. In the latter, it receives an **awakeFromInsertion** 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 (in Yellow Box only). 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**

| | |
|---|---|
| *MyClass*(EOEditingContext, EOClassDescription, EOGlobalID) | The framework creates enterprise objects with this method if it exists. Yellow Box resorts to the empty constructor if this constructor doesn't exist, but Java Client requires this constructor. |
| awakeFromFetch | Performs additional initialization after the object is fetched. |
| awakeFromInsertion | 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-Value Coding: Accessing Properties and Relationships

| | |
|---|---|
| *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*. |
| handleTakeValueForUnboundKey | Handles a failure of **takeValueForKey** to find a property. |
| handleQueryWithUnboundKey | Handles a failure of **valueForKey** to find a property. |
| unableToSetNullForKey | Handles an attempt to set a non-object property's value to **null**. |

### 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. |

# EOFaulting

**Implemented By:** EOEnterpriseObject
EOCustomObject
EOGenericRecord

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## Interface Description

The EOFaulting interface together with the EOFaultHandler class forms a general mechanism for postponing an object's initialization until its actually needed. In it's pre-initialization state, an EOFaulting object is known as a *fault*. When the object is sent a message to which it can't respond without initializing, it uses a fault handler to *fire*, or to finish initializing. Faults are 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. Consequently, a fault typically fires when an attempt is made to access any of its data. In this case, firing a fault involves fetching the object's data.

The default implementations of EOFaulting in EOCustomObject and EOGenericRecord are sufficient for most purposes. If you need custom faulting behavior, you typically create a subclass of EOFaultHandler to accommodate different means of converting faults into regular objects; there's rarely a need to override the default implementations of EOFaulting.

### Creating a Fault

In Yellow Box, you create a fault with the EOFaultHandler method **makeObjectIntoFault**. In Java Client, you create a fault by sending an newly created object a **turnIntoFault** message, providing an EOFaultHandler that will later help the fault to fire. This fault handler should be considered completely the private property of the fault. You shouldn't send it any messages, instead dealing exclusively with the fault.

### Firing a Fault

A fault is fired when it can't respond to a message without completing its initialization. Any of the object's methods that requires initialization trigger the firing, This is generally accomplished by invoking the **willRead** method. For example, in the typical case of an object that needs to fetch it's data from a database upon firing, **willRead** is invoked from the object's "get" methods, such as the following:

```
public String roleName() {
    willRead();
    return roleName;
}
```

The default implementations of **willRead** provided by EOCustomObject and EOGenericRecord take care of using the object's fault handler to finish initialization. For more information on a fault handler's role, see the EOFaultHandler class specification.

## Instance Methods

### clearFault

public abstract void **clearFault**()

This method is available for Java Client applications only; there is no Yellow Box equivalent.

Restores the receiver to its status prior to the **turnIntoFault** message that turned the object into a fault. Throws an exception if the receiver isn't a fault.

You rarely use this method. Rather, it's invoked by an EOFaultHandler during the process of firing the fault. For more information, see the EOFaultHandler class specification.

### isFault

public abstract boolean **isFault**()

This method is available for Java Client applications only; there is no Yellow Box equivalent.

Returns **true** if *anObject* is an EOFault, **false** otherwise.

### turnIntoFault

public abstract void **turnIntoFault**(EOFaultHandler *aFaultHandler*)

This method is available for Java Client applications only; there is no Yellow Box equivalent.

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

## willRead

public abstract void **willRead**()

Fills the receiver with values fetched from the database. Before your application attempts to message an object, you must ensure that it has been filled with its data. To do this, enterprise objects invoke the method **willRead** prior to any attempt to access the object's state, most typically in "get" methods such as the following:

```
public String roleName() {
    willRead();
    return roleName;
}
```

# EOKeyValueCoding

**Implemented By:** EOEnterpriseObject
EOCustomObject
EOGenericRecord

**Implements:** com.apple.client.foundation.NSKeyValueCoding (Java Client only)

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Interface Description

The EOKeyValueCoding interface 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. EOCustomObject and EOGenericRecord provide default implementations of EOKeyValueCoding, which are sufficient for most purposes.

The basic methods for accessing an object's values are **takeValueForKey**, 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 EOCustomObject 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 **takeStoredValueForKey** 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, **takeStoredValueForKey** is used to initialize an object's properties with values fetched from the database, whereas **takeValueForKey** 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."

The remaining methods, **handleQueryWithUnboundKey**, **handleTakeValueForUnboundKey**, and **unableToSetNullForKey**, are provided to handle error conditions. The default versions of **handleQueryWithUnboundKey** and **handleTakeValueForUnboundKey** throw an exception.

For more information on EOKeyValueCoding, see the sections:

- Stored Value Methods
- Type Checking and Type Conversion

## Interfaces Implemented

NSKeyValueCoding (Java Client only)

takeValueForKey
valueForKey

## Method Types

Accessing values

storedValueForKey
takeStoredValueForKey
takeValueForKey
valueForKey

Handling error conditions

handleQueryWithUnboundKey
handleTakeValueForUnboundKey
unableToSetNullForKey

## Instance Methods

### handleQueryWithUnboundKey

public abstract java.lang.Object **handleQueryWithUnboundKey**(java.lang.String *key*)

Invoked from **valueForKey** when it finds no property binding for *key*. EOCustomObject's implementation throws an exception. Subclasses can override this method to handle the query in some other way.

### handleTakeValueForUnboundKey

public abstract void **handleTakeValueForUnboundKey**(
    java.lang.Object *value*,
    java.lang.String *key*)

Invoked from **takeValueForKey** when it finds no property binding for *key*. EOCustomObject's implementation throws an exception. Subclasses can override it to handle the request in some other way.

## storedValueForKey

public abstract java.lang.Object **storedValueForKey**(java.lang.String *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 EOCustomObject 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 **handleTakeValueForUnboundKey**.

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 static method **useStoredAccessor** to return **false**. And as with **valueForKey**, you can prevent direct access of an instance variable with the method the static method **accessInstanceVariablesDirectly**.

## takeStoredValueForKey

public abstract void **takeStoredValueForKey**(
java.lang.Object *value*,
java.lang.String *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 EOCustomObject is similar to the implementation of **takeValueForKey**, 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”, **takeStoredValueForKey** 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”, **takeStoredValueForKey** looks for an instance variable named **_lastName** or **lastName**.
3. If neither a private accessor or an instance variable is found, **takeStoredValueForKey** searches for a public accessor method based on *key*. For the key “lastName”, this would be **setLastName**.
4. If *key* is unknown, **storedValueForKey** calls **handleTakeValueForUnboundKey**.

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 **takeValueForKey**, you can implement the static method **useStoredAccessor** to return **false**. And as with **valueForKey**, you can prevent direct access of an instance variable with the method the static method **accessInstanceVariablesDirectly**.

## takeValueForKey

public abstract void **takeValueForKey**(
java.lang.Object *value*,
java.lang.String *key*)

Sets the value for the property identified by *key* to *value*, invoking **handleTakeValueForUnboundKey** if the receiver doesn't recognize *key* and **unableToSetNullForKey** if *value* is **null** and *key* identifies a scalar property.

The default implementation provided by EOCustomObject 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 static method **accessInstanceVariablesDirectly** returns **true**, **takeValueForKey** searches for an instance variable based on *key* and sets the 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 **handleTakeValueForUnboundKey**.

## unableToSetNullForKey

public abstract void **unableToSetNullForKey**(java.lang.String *key*)

Invoked from **takeValueForKey** (and **takeStoredValueForKey**) when it's given a **null** value for a scalar property (such as an **int** or a **float**). EOCustomObject's implementation throws an exception. Subclasses can override it to handle the request in some other way, such as by substituting zero or a sentinel value and invoking **takeValueForKey** again.

## valueForKey

public abstract java.lang.Object **valueForKey**(java.lang.String *key*)

Returns the value for the property identified by *key*, invoking **handleQueryWithUnboundKey** if the receiver doesn't recognize *key*.

The default implementation provided by EOCustomObject 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 static method **accessInstanceVariablesDirectly** returns **true**, **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 **takeStoredValueForKey**, 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 **takeValueForKey**, 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", **takeValueForKey** 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 the basic methods search for a private accessor method based on the key. For example, with a key of "lastName", **takeValueForKey** 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**.

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) {
    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 **takeStoredValueForKey** 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 static method **useStoredAccessor** to return **false**.

## 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 a String to **takeValueForKey** 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 **validateValueForKey** 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 **validateValueForKey** method, see the EOValidation interface specification.

The key-value coding methods handle one special case with regard to value types. For enterprise objects that access numeric values as scalar types, these methods automatically convert between the scalar types and java.lang.Number objects. For example, suppose your enterprise object defines these accessor methods:

```
public void setSalary(int salary)
public int salary()
```

For the **setSalary** method, **takeValueForKey** converts the object value it receives as the argument for the "salary" key to an **int** and passes it as *salary* to **setSalary**. Similarly, **valueForKey** converts the return value of the **salary** method to a java.lang.Number and returns that.

The default implementations of the key-value coding methods support the scalar types **int**, **float**, and **double**. Object values are converted to these types with the standard messages **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 **null** to a scalar value. Scalar values define no equivalent of a database system's NULL value, so these must be handled by the object itself. Upon encountering **null** while setting a scalar value, **takeValueForKey** invokes **unableToSetNullForKey**, which by default simply throws 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 **null**, reinvoking **takeValueForKey** to set the substitute value.

# EOKeyValueCodingAdditions

**Implemented By:** EOEnterpriseObject
EOCustomObject
EOGenericRecord

**Implements:** com.apple.client.foundation.NSKeyValueCoding (Java Client only)
EOKeyValueCoding

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Interface Description

The EOKeyValueCodingAdditions interface defines extensions to the basic EOKeyValueCoding interface. One pair of methods, **takeValuesFromDictionary** and **valuesForKeys**, gives access to groups of properties. Another pair of methods, **takeValueForKeyPath** and **valueForKeyPath** give access to properties across relationships with key paths of the form *relationship.property*; for example, "department.name". EOCustomObject and EOGenericRecord provide default implementations of EOKeyValueCodingAdditions, which you rarely (if ever) need to override.

### EONullValue in Collections

Because collection objects such as NSArray and NSDictionary can't contain **null** as a value, **null** must be represented by a special object, EONullValue. EONullValue provides a single instance that represents the NULL value for object attributes. The default implementations of **takeValuesFromDictionary** and **valuesForKeys** translate EONullValue and **null** between  between NSDictionaries and enterprise objects so your objects don't have to explicitly test for EONullValues.

## Instance Methods

### takeValueForKeyPath

public abstract void **takeValueForKeyPath**(
java.lang.Object *value*,
java.lang.String *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".

EOCustomObject's implementation of this method gets the destination object for each relationship using **valueForKey**, and sends the final object a **takeValueForKey**message with *value* and *property.*

### takeValuesFromDictionary

public abstract void **takeValuesFromDictionary**(NSDictionary *aDictionary*)

Sets properties of the receiver with values from *aDictionary*, using its keys to identify the properties. EOCustomObject's implementation invokes **takeValueForKey** for each key-value pair, substituting **null** for EONullValues in *aDictionary.*

### valueForKeyPath

public abstract java.lang.Object **valueForKeyPath**(java.lang.String *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". EOCustomObject'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

public abstract NSDictionary **valuesForKeys**(NSArray *keys*)

Returns a dictionary containing the property values identified by each of *keys*. EOCustomObject's implementation invokes **valueForKey** for each key in *keys*, substituting EONullValues in the dictionary for returned **null** values.

# EOEditingContext.MessageHandler

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (Yellow Box)

## Interface Description

The EOEditingContext.MessageHandler interface 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.

Message handlers are not required to provide implementations for all of the methods in the interface. When you write a handler, you don't have to use the **implements** keyword to specify that the object implements the MessageHandler interface. Instead, simply use the EOEditingContext method **setMessageHandler** method to assign your object as the EOEditingContext's handler and then declare and implement any subset of the methods declared in the MessageHandler interface. An EOEditingContext can determine if the handler doesn't implement a method and only attempts to invoke the methods the handler actually implements

## Instance Methods

### editingContextPresentException

public abstract void **editingContextPresentException**(
EOEditingContext *anEditingContext*,
java.lang.Exception *anException*)

This method is available for Java Client applications only; the Yellow Box equivalent is **editingContextPresentErrorMessage**.

Invoked by *anEditingContext*, this method should present an error 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). The error message can be derived from *anException*, an exception that was thrown as the result of some error.

## editingContextPresentErrorMessage

public abstract void **editingContextPresentErrorMessage**(
EOEditingContext *anEditingContext*,
java.lang.String *message*)

This method is available for Yellow Box applications only; the Java Client equivalent is **editingContextPresentException**.

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.

## editingContextShouldContinueFetching

public abstract boolean **editingContextShouldContinueFetching**(
EOEditingContext *anEditingContext*,
int *count*,
int *limit*,
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

**Implemented By:** EODelayedObserver
EOEditingContext

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Protocol Description

The EOObserving interface,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** method. For an overview of the general change tracking mechanism, see "Tracking Enterprise Objects ChangesEOControl provides four classes and an interface 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 interface, 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." in the introduction to the EOControl Framework. The EOObserving interface

## Instance Methods

### objectWillChange

public abstract void **objectWillChange**(java.lang.Object *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.

# EOQualifier.Comparison

**Implemented By:** NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)

## Interface Description

The EOQualifierComparison interface defines methods for comparing values. These methods are used for evaluating qualifiers in memory. Though declared for NSObject in Yellow Box, most of these methods work properly only with value classes: NSString, NSDate, NSNumber, NSDecimalNumber, and EONullValue. Yellow Box implements these methods as part of NSObject—there is no separate interface. In Java Client, support for these methods is provided for java.lang.String, java.lang.Number, and java.lang.Date using EOQualifier.ComparisonSupport. You should implement this interface for any value classes you write that you want to be evaluated in memory by EOQualifier instances.

## Method Types

Testing value objects

doesContain
isEqualTo
isGreaterThan
isGreaterThanOrEqualTo
isLessThan
isLessThanOrEqualTo
isLike
isCaseInsensitiveLike
isNotEqualTo

## Instance Methods

### doesContain

public abstract boolean **doesContain**(java.lang.Object *anObject*)

Returns **true** if the receiver contains *anObject*, **false** if it doesn't. NSObject's implementation of this method returns **true** only if the receiver is a kind of NSArray and contains *anObject*. In all other cases it returns **false**.

## isCaseInsensitiveLike

public abstract boolean **isCaseInsensitiveLike**(java.lang.Object *anObject*)

Returns **true** if the receiver is a case-insensitive match for *aString*, **false** if it isn't. See "Using Wildcards" in the EOQualifier class specification for the wildcard characters allowed. NSObject's implementation returns **false**; NSString's performs a proper case-insensitive comparison.

**See also:** **isLike**, **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**,**isNotEqualTo**

## isEqualTo

public abstract boolean **isEqualTo**(java.lang.Object *anObject*)

Returns **true** if the receiver is equal to *anObject*, **false** if it isn't. NSObject's implementation invokes **isEqual** and returns the result.

**See also:** **doesContain**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isGreaterThan

public abstract boolean **isGreaterThan**(java.lang.Object *anObject*)

Returns **true** if the receiver is greater than *anObject*, **false** if it isn't. NSObject's implementation invokes **compare:** and returns **true** if the result is NSOrderedDescending.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isGreaterThanOrEqualTo

public abstract boolean **isGreaterThanOrEqualTo**(java.lang.Object *anObject*)

Returns **true** if the receiver is greater than or equal to *anObject*, **false** if it isn't. NSObject's implementation invokes **compare:** and returns **true** if the result is NSOrderedAscending.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

### isLessThan

public abstract boolean **isLessThan**(java.lang.Object *anObject*)

Returns **true** if the receiver is less than *anObject*, **false** if it isn't. NSObject's implementation invokes **compare:** and returns **true** if the result is NSOrderedAscending.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

### isLessThanOrEqualTo

public abstract boolean **isLessThanOrEqualTo**(java.lang.Object *anObject*)

Returns **true** if the receiver is less than or equal to *anObject*, **false** if it isn't. NSObject's implementation invokes **compare:** and returns **true** if the result is NSOrderedAscending or NSOrderedSame.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

### isLike

public abstract boolean **isLike**(java.lang.Object *anObject*)

Returns **true** if the receiver matches *aString* according to the semantics of the SQL **like** comparison operator, **false** if it doesn't. See "Using Wildcards" in the EOQualifier class specification for the wildcard characters allowed. NSObject's implementation returns **false**; NSString's performs a proper comparison.

**See also:** **isCaseInsensitiveLike**, **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isNotEqualTo**

### isNotEqualTo

public abstract boolean **isNotEqualTo**(java.lang.Object *anObject*)

Returns **true** if the receiver is not equal to *anObject*, **false** if it is. NSObject's implementation invokes **isEqual**, inverts the result, and returns it.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**

# EOQualifierEvaluation

**Implemented By:**

EOKeyValueQualifier
EOKeyComparisonQualifier
EOAndQualifier
EOOrQualifier
EONotQualifier

## Protocol Description

The EOQualifierEvaluation interface defines a method, **evaluateWithObject**, that performs in-memory evaluation of qualifiers. All qualifier classes whose objects can be evaluated in memory must implement this interface.

## Instance Methods

### evaluateWithObject

public boolean **evaluateWithObject**(java.lang.Object *object*)

Returns **true** if the argument *object* satisfies the qualifier, **false** otherwise. This method can throw one of several possible exceptions if an error occurs, depending on the implementation.

# EORelationshipManipulation

**Implemented By:** EOEnterpriseObject
EOCustomObject
EOGenericRecord

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

## Interface Description

The EORelationshipManipulation interface builds on the basic EOKeyValueCoding interface to allow you to modify to-many relationship properties. EOCustomObject and EOGenericRecord provide default implementations of EORelationshipManipulation, which you rarely (if ever) need to override.

The primitive methods **addObjectToPropertyWithKey** and **removeObjectFromPropertyWithKey** add and remove single objects from to-many relationship arrays. The two other methods in the interface, **addObjectToBothSidesOfRelationshipWithKey** and **removeObjectFromBothSidesOfRelationshipWithKey**, 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 **addObjectToPropertyWithKey** and **removeObjectFromPropertyWithKey** 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

### addObjectToBothSidesOfRelationshipWithKey

public abstract void **addObjectToBothSidesOfRelationshipWithKey**(
EORelationshipManipulation *anObject*,
java.lang.String *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 **takeValueForKey**. For a to-many relationship, *anObject* is added using **addObjectToBothSidesOfRelationshipWithKey**.

This method also properly handles removing **this** 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.

### addObjectToPropertyWithKey

public abstract void **addObjectToPropertyWithKey**(
java.lang.Object *anObject*,
java.lang.String *key*)

Adds *anObject* to the receiver's to-many relationship identified by *key*, without setting a reciprocal relationship. Similar to the implementation of **takeValueForKey**, EOCustomObject'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* .

### removeObjectFromBothSidesOfRelationshipWithKey

public abstract void **removeObjectFromBothSidesOfRelationshipWithKey**(
EORelationshipManipulation *anObject*,
java.lang.String *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 **takeValueForKey** with **null** as the value. For a to-many relationship, *anObject* is removed using **removeObjectFromPropertyWithKey**.

### removeObjectFromPropertyWithKey

public abstract void **removeObjectFromPropertyWithKey**(
java.lang.Object *anObject*,
java.lang.String *key*)

Removes *anObject* from the receiver's to-many relationship identified by *key*, without modifying a reciprocal relationship. Similar to the implementation of **takeValueForKey**, EOCustomObject'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 null.

# EOSortOrdering.Comparison

**Implemented By:** EONullValue (Java Client)
NSObject (Yellow Box)

**Package:** com.apple.client.eocontrol (Java Client)

## Interface Description

The EOSortOrdering.Comparison interface defines methods for comparing values. These methods are used for sorting value objects. Though declared for NSObject in Yellow Box, most of these methods work properly only with value classes: NSString, NSDate, NSNumber, NSDecimalNumber, and EONullValue. Yellow Box implements these methods as part of NSObject—there is no separate interface. In Java Client, support for these methods is provided for java.lang.String, java.lang.Number, and java.lang.Date using EOSortOrdering.ComparisonSupport. EONullValue implements the interface directly. You should implement this interface for any value classes you write that you want to be properly sorted by EOSortOrdering instances.

Sorting value objects

compareAscending
compareCaseInsensitiveAscending
compareCaseInsensitiveDescending
compareDescending

## Instance Methods

### compareAscending

public abstract int **compareAscending**(java.lang.Object *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

public abstract int **compareCaseInsensitiveAscending**(java.lang.Object *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

public abstract int **compareCaseInsensitiveDescending**(java.lang.Object *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

public abstract int **compareDescending**(java.lang.Object *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

**Implemented By:** EOEnterpriseObject
EOCustomObject
EOGenericRecord

**Package:** com.apple.client.eocontrol (Java Client)
com.apple.yellow.eocontrol (WebObjects and Yellow Box)

**Inherits From:** java.lang.Object

**Package:** com.apple.client.eocontrol

## Interface Description

The EOValidation interface 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. EOCustomObject and EOGenericRecord 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

public abstract void **validateForDelete**() throws EOValidation.Exception

Confirms that the receiver can be deleted in its current state, throwing an EOValidation.Exception if it can't. For example, an object can't be deleted if it has a relationship with a delete rule of EOClassDescription.DeleteRuleDeny and that relationship has a destination object.

EOCustomObject'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 thrown by **super**'s implementation with their own.

**See also:** **propagateDeleteWithEditingContext** (EOEnterpriseObject), "Constructors" (EOValidationException)

## validateForInsert

public abstract void **validateForInsert**() throws EOValidation.Exception

Confirms that the receiver can be inserted in its current state, throwing an EOValidation.Exception if it can't. EOCustomObject'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

public abstract void **validateForSave**()

Confirms that the receiver can be saved in its current state, throwing an EOValidation.Exception if it can't. EOCustomObject's implementation sends the receiver's EOClassDescription a **validateObjectForSave** message, then iterates through all of the receiver's properties, invoking **validateValueForKey** for each one. If this results in more than one exception, the exception returned contains the additional ones in its **userInfo** dictionary under the EOValidation.Exception.AdditionalExceptions key. Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception thrown by **super**'s implementation with their own.

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 **validateValueForKey**.

**See also:** "Constructors" (EOValidationException)

## validateForUpdate

public abstract void **validateForUpdate**() throws EOValidation.Exception

Confirms that the receiver can be inserted in its current state, throwing an EOValidation.Exception if it can't. EOCustomObject'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**.

## validateValueForKey

public abstract java.lang.Object **validateValueForKey**(
java.lang.Object *value*,
java.lang.String *key*) throws EOValidation.Exception

Confirms that *value* is legal for the receiver's property named by *key*. Throws an EOValidation.Exception if it can't confirm that the value is legal. The implementation can provide a coerced value by returning the value. This lets you convert strings to dates or numbers or maybe convert strings to an enumerated type value. EOCustomObject's implementation sends a **validateValueForKey** message to the receiver's EOClassDescription.

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.

"Constructors" (EOValidationException)

# EOValidation

## Validating Individual Properties

The most general method for validating individual properties, **validateValueForKey**, 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 EOCustomObject consults the object's EOClassDescription (using the EOEnterpriseObject interface 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.

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 **validateValueForKey**. Rather, the default implementation provided by EOCustomObject 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 **validateValueForKey** 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 throws 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.");
}
```

### 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 **validateValueForKey** message, allowing the object to coerce the value's type, perform any additional validation, and throw 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 **validateValueForKey** only for type coersion. However, you can use the EODisplayGroup method **setValidatesChangesImmediately:** with an argument of **true** 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. EOCustomObject 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.

If you override any of these operation-specific validation methods, be sure to invoke **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 **validateValueForKey** 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**.

# EOQualifier.ComparisonSupport

**Inherits From:** java.lang.Object

**Package:** com.apple.client.eocontrol (Siva)

## Class Description

Siva's class EOQualifier.ComparisonSupport provides default implementations of the EOQualifierComparison interface. It is for use in client-side enterprise object classes only; there is no equivalent Yellow Box class for server-side enterprise objects.

Siva's EOCustomObject uses EOQualifier.ComparisonSupport's default implementations. Typically your custom enterprise object classes inherit from EOCustomObject and inherit the default implementations. If your custom enterprise object class doesn't inherit from EOCustomObject, you should implement the EOQualifierComparison interface directly.

## Method Types

Setting up automatic support

setSupportForClass
supportForClass

Comparing two objects

compareValues

EOQualifierComparison methods

compareValues
setSupportForClass
supportForClass
EOQualifier.ComparisonSupport
doesContain
isCaseInsensitiveLike
isEqualTo
isGreaterThan
isGreaterThanOrEqualTo
isLessThan
isLessThanOrEqualTo
isLike
isNotEqualTo

## Static Methods

### compareValues

public static int **compareValues**(java.lang.Object *anObject*, java.lang.Object *anotherObject*, com.apple.client.foundation.NSSelector *selector*)

Compares the two objects using *selector*. You should use this method to compare value objects instead of calling *selector* directly. This method is the entry point for the comparison support, and calls methods in support classes if appropriate.

**See also:** **setSupportForClass**, **supportForClass**

### setSupportForClass

public static void **setSupportForClass**(EOSortOrdering. ComparisonSupport *supportClass*, java.lang.Class *aClass*)

Sets *supportClass* as the support class to be used for comparing instances of *aClass*. When **compareValues** is called, the methods in *supportClass* will be used to do the comparison for instances of *aClass*.

**See also:** **compareValues**

### supportForClass

public static EOSortOrdering. ComparisonSupport **supportForClass**(java.lang.Class *aClass*)

Returns the support class used for doing sort ordering comparisons for instances of *aClass*.

**See also:** **compareValues**, **setSupportForClass**

## Instance Methods

### doesContain

public boolean **doesContain**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns YES if *receiver* contains *anObject*, NO if it doesn't. NSObject's implementation of this method returns YES only if *receiver* is a kind of NSArray and contains *anObject*. In all other cases it returns NO. This method is used in the Framework only by EOQualifier for in-memory evaluation.

## isCaseInsensitiveLike

public boolean **isCaseInsensitiveLike**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns YES if *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. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **isLike**, **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**,**isNotEqualTo**

## isEqualTo

public boolean **isEqualTo**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **isEqual:** and returns the result. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isGreaterThan

public boolean **isGreaterThan**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **compare:** and returns YES if the result is NSOrderedDescending. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isGreaterThanOrEqualTo

public boolean **isGreaterThanOrEqualTo**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **compare:** and returns YES if the result is NSOrderedDescending or NSOrderedSame. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isLessThan

public boolean **isLessThan**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **compare:** and returns YES if the result is NSOrderedAscending. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isLessThanOrEqualTo

public boolean **isLessThanOrEqualTo**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **compare:** and returns YES if the result is NSOrderedAscending or NSOrderedSame. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLike**, **isCaseInsensitiveLike**, **isNotEqualTo**

## isLike

public boolean **isLike**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns YES if *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. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **isCaseInsensitiveLike**, **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isNotEqualTo**

## isNotEqualTo

public boolean **isNotEqualTo**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Invokes **isEqual:**, inverts the result, and returns it. This method is used in the Framework only by EOQualifier for in-memory evaluation.

**See also:** **doesContain**, **isEqualTo**, **isGreaterThan**, **isGreaterThanOrEqualTo**, **isLessThan**, **isLessThanOrEqualTo**, **isLike**, **isCaseInsensitiveLike**

# EOSortOrdering.ComparisonSupport

**Inherits From:** java.lang.Object

**Package:** com.apple.client.eocontrol (Siva)

## Class Description

Siva's class EOSortOrdering.ComparisonSupport provides default implementations of the EOSortOrderingComparison interface. It is for use in client-side enterprise object classes only; there is no equivalent Yellow Box class for server-side enterprise objects.

Siva's EOCustomObject uses EOSortOrdering.ComparisonSupport's default implementations. Typically your custom enterprise object classes inherit from EOCustomObject and inherit the default implementations. If your custom enterprise object class doesn't inherit from EOCustomObject, you should implement the EOSortOrderingComparison interface directly.

## Method Types

Setting up automatic support
- setSupportForClass
- supportForClass

Comparing two objects
- compareValues

EOSortOrderingComparison methods
- compareAscending
- compareCaseInsensitiveAscending
- compareCaseInsensitiveDescending
- compareDescending

## Static Methods

### compareValues

public static int **compareValues**(java.lang.Object *anObject*, java.lang.Object *anotherObject*, com.apple.client.foundation.NSSelector *selector*)

Compares the two objects using *selector*. You should use this method to compare value objects instead of calling *selector* directly. This method is the entry point for the comparison support, and calls methods in support classes if appropriate.

**See also:** **setSupportForClass**, **supportForClass**

### setSupportForClass

public static void **setSupportForClass**(EOSortOrdering. ComparisonSupport *supportClass*, java.lang.Class *aClass*)

Sets *supportClass* as the support class to be used for comparing instances of *aClass*. When **compareValues** is called, the methods in *supportClass* will be used to do the comparison for instances of *aClass*.

**See also:** **compareValues**

### supportForClass

public static EOSortOrdering. ComparisonSupport **supportForClass**(java.lang.Class *aClass*)

Returns the support class used for doing sort ordering comparisons for instances of *aClass*.

**See also:** **compareValues**, **setSupportForClass**

## Instance Methods

### compareAscending

public int **compareAscending**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns NSOrderedAscending if *anObject* is naturally ordered after *receiver*, NSOrderedDescending if it's naturally ordered before *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

public int **compareCaseInsensitiveAscending**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns NSOrderedAscending if *anObject* is naturally ordered—ignoring case—after *receiver*, NSOrderedDescending if it's naturally ordered before *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

public int **compareCaseInsensitiveDescending**(java.lang.Object *receiver*, java.lang.Object *anObject*)

Returns NSOrderedAscending if *anObject* is naturally ordered—ignoring case—before *receiver*, NSOrderedDescending if it's naturally ordered after *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

public int **compareDescending**(java.lang.Object *anObject*, java.lang.Object *anObject*)

Returns NSOrderedAscending if *anObject* is naturally ordered before *receiver*, NSOrderedDescending if it's naturally ordered after *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**