NSObject Additions

Inherits From: none *(NSObject is a root class)*

Declared In: EOControl/EOClassDescription.h
EOControl/EOEditingContext.h
EOControl/EOKeyValueCoding.h
EOControl/EOObserver.h

Class at a Glance

Purpose

Defines basic functionality for all enterprise objects.

Principal Attributes

- EOClassDescription

Creation

– init	Designated initializer.
– initWithEditingContext:classDescription:globalID:	Optional initializer.
– awakeFromFetchInEditingContext:	Additional initialization after being fetched.
– awakeFromInsertionInEditingContext:	Additional initialization after being created in memory.

Commonly Used Methods

– willChange	Notifies observers of a change in state.
– editingContext	Returns the receiver's EOEditingContext.
– addObject:toPropertyWithKey:	Adds an object to a relationship property.
– removeObject:fromPropertyWithKey:	Removes an object from a relationship property.
– addObject:toBothSidesOfRelationshipWithKey:	Adds an object to a relationship property and the receiver to the reciprocal relationship.
– removeObject:fromBothSidesOfRelationshipWithKey:	Removes an object from a relationship property and the receiver from the reciprocal relationship.

Methods to Implement or Override

These methods are invoked by the Framework.

– set*Key*:	Sets the value for the property named *key*.
– *key*	Retrieves the value for the property named *key*.
– addTo*Key*:	Adds an object to a relationship property named *key*.
– removeFrom*Key*:	Removes an object from the property named *key*.
– handleTakeValue:forUnboundKey:	Handles a failure of **takeValue:forKey:** to find a property.
– handleQueryWithUnboundKey:	Handles a failure of **valueForKey:** to find a property.
– unableToSetNilForKey:	Handles an attempt to set a non-object property's value to **nil**.
– validate*Key*:	Validates a value for the property named *key*.
– validateForDelete	Validates all properties before deleting the receiver.
– validateForInsert	Validates all properties before inserting the receiver.
– validateForSave	Validates all properties before saving the receiver.
– validateForUpdate	Validates all properties before updating the receiver.

Class Description

Enterprise Objects Framework adds a number of methods to NSObject for supporting operations common to all enterprise objects. Among these are methods for initializing instances, announcing changes, setting and retrieving property values, and performing validation of state. Some of these methods are for enterprise objects to implement or override, and some are meant to be used as defined by the Framework. Many methods are used internally by the Framework and rarely invoked by application code. The implementation or use of each method is highlighted in the following sections, which describe the major functional groups.

Initialization Methods

Enterprise objects are initialized using the **initWithEditingContext:classDescription:globalID:**, which by default simply invokes **init**. You can place your custom initialization code in **init**, or you can override **initWithEditingContext:classDescription:globalID:** to take advantage of the extra information available with this method.

After initialization, an enterprise object receives an **awake...** message. The particular message depends on whether the object has been fetched from a database or created anew in the application. In the former case, it receives an **awakeFromFetchInEditingContext:** message. In the latter, it receives an **awakeFromInsertionInEditingContext:** message. The receiver can override either method to perform extra initialization—such as setting default values—based on how it was created.

Announcing Changes

For the Framework to keep all areas of an application synchronized, enterprise objects must notify their observers when their state changes. Objects do this by simply invoking **willChange** before altering any instance variable or other kind of state. This method informs all observers that the invoker is about to change. See the EOObserverCenter class specification for more information on change notification.

The primary observer of changes in an object is its EOEditingContext. EOEditingContext is a subclass of EOObjectStore that manages collections of objects in memory, tracking inserts, deletes, and updates, and propagating changes to the persistent store as needed. You can get the EOEditingContext that contains an object by sending the object an **editingContext** message.

Getting Object and Class Metadata

One of the larger groups of methods added to NSObject provides information about an object's properties. Most of these methods consult an EOClassDescription to provide their answers. The **classDescription** method return an object's EOClassDescription. See that class specification for the methods it implements. Methods you can send directly to any object include **entityName**, which provides the name of the entity mapped to the receiver's class; **allPropertyKeys**, which returns the names of all the receiver's class properties, attributes and relationships alike; and **attributeKeys**, which returns just the names of the attributes.

Some methods return information about relationships. **toOneRelationshipKeys** and **toManyRelationshipKeys** return the names of the receiver's relationships, while **isToManyKey:** tells which kind a particular relationship is. **deleteRuleForRelationshipKey:** indicates what should happen to the receiver's relationships when it's deleted. Similarly, **ownsDestinationObjectsForRelationshipKey:** indicates what should happen when another object is added to or removed from the receiver's relationship. Another method, **classDescriptionForDestinationKey:**, returns the EOClassDescription for the objects at the destination of a relationship.

These methods are all properly implemented in terms of the receiver's EOClassDescription, so unless your object class doesn't have an EOClassDescription, there's little need to override them. One method you might override in your enterprise object class, however, is **inverseForRelationshipKey:**. Given the name of one of the receiver's relationships, this method finds the destination object's class data and determines the name of the relationship that points back at the receiver. The default implementation of this method looks for a relationship predicated on the same attributes in both the source and destination, which works correctly in most cases. If, however, you define a reciprocal pair of relationships on different attributes, you should override this method to take that into account. See the method description for an example.

Key-Value Coding Methods

A special set of methods form the Framework's main data transport mechanism, in which the properties of an enterprise object are accessed indirectly by name (or key), rather than directly through invocation of an accessor method or as instance variables. Thus, any object's state can be accessed in a consistent manner.

The basic methods for accessing an enterprise object's values are **takeValue:forKey:** and **valueForKey:**. These two methods are defined by NSObject to use the accessor methods normally implemented by objects (or to access instance variables directly if need be), so that you don't have to write special code simply to integrate your enterprise objects into the Framework. Another pair of methods, **takeValuesFromDictionary:** and **valuesForKeys:**, gives access to groups of properties. Lastly, **valueForKeyPath:** and **valueForKeyPath:** give access to flattened properties, of the form *relationship.property*; for example, "department.name".

Default Implementations; Handling Access Errors

The Framework provides default implementations of **takeValue:forKey:** and **valueForKey:** that work for all objects. The other four access methods are implemented in terms of these two. These implementations are general enough that your enterprise object classes should rarely need to override either key-value coding method. In accessing an object's property, the default NSObject implementations of the key-value coding methods use the class definition as follows:

1. The key-value coding method looks for an accessor method based on the key. For example, with a key of "lastName", **takeValue:forKey:** looks for a method named **setLastName:** (note that the first letter of the property name is made uppercase), and **valueForKey:** looks for a method of the form **lastName**.

2. If the key-value coding method doesn't find an accessor method, and the class responds YES to an **accessInstanceVariablesDirectly** message (which it does by default), it looks for an instance variable whose name is the same as the key and sets or retrieves its value directly. In setting an instance variable that's an object, **takeValue:forKey:** retains the new value and autoreleases the old one.

3. If neither an accessor method nor an instance variable can be found, the default implementations invoke methods that your custom objects can override to handle failures. **handleTakeValue:forUnboundKey:** is invoked from **takeValue:forKey:**, and **handleQueryWithUnboundKey:** is invoked from **valueForKey:**. Normally these methods raise an exception, but you can implement them to set or get a value in whatever way is needed.

The key-value coding methods cache attribute bindings for both accessor methods and instance variables, making lookups efficient. If you need to clear these bindings—as when you add or remove a class from the run-time system—you can invoke **flushAllKeyBindings** to do so.

Some subclasses of NSObject override the default implementations. EOGenericRecord's implementations, for example, simply store and retrieve the properties in an NSDictionary object held by the EOGenericRecord. NSDictionary and NSMutableDictionary, though not suitable for use as enterprise objects, meaningfully implement these methods by directly accessing their key-value pairs.

Type Checking and Type Conversion

The default implementations of the key-value coding methods accept any object as a value, and do no type checking or type conversion among object classes. It's possible, for example, to pass an NSString to **takeValue:forKey:** as the value for a property the receiver expects to be an NSDate. The sender of a key-value coding message is thus responsible for ensuring that a values is of the proper class, typically by

using the **validateValue:forKey:** method to coerce it to the proper type. The interface layer's EODisplayGroup uses this on all values received from interface user objects, for example, as well as relying on number and date formatters to interpret string values typed by the user. For more information on the **validateValue:forKey:** method, see the EOClassDescription and EOEntityClassDescription class specifications.

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

– (void)**setSalary:**(unsigned int)*salary*
– (unsigned int)**salary**

For the **setSalary:** method, **takeValue:forKey:** converts the object value for the "salary" key in the dictionary to an **unsigned int** and passes it as *salary.* Similarly, **valueForKey:** converts the return value of the **salary** method to an NSNumber and returns that.

The default implementations support the following scalar types:

char	unsigned char
short	unsigned short
int	unsigned int
long	unsigned long
float	double

Object values are converted to these types with the standard messages **charValue**, **intValue**, **floatValue**, and so on. Note that the key-value coding methods don't check that an object value actually responds to these messages; this can result in a run-time error if the object doesn't respond to the appropriate message.

One type of conversion these methods can't perform is that from a **nil** object value to a scalar value. C scalar values define no equivalent of a database system's NULL value, so these must be handled by the object itself. Upon encountering an **nil** while setting a scalar value, the **takeValue:forKey:** invokes **unableToSetNilForKey:**, which by default simply raises an exception. Enterprise object classes that use scalar values which may be NULL in the database should override this method to substitute the appropriate scalar value for **nil**, reinvoking **takeValue:forKey:** to set the substitute value. This method works in general to handle setting scalar properties to **nil**.

EONull in Collections

Because collection objects such as NSArray and NSDictionary can't contain **nil** as a value, it must represented by a special object, EONull. EONull provides a single instance that represents the NULL value for object attributes. The default implementations of **takeValuesFromDictionary:** and **valuesForKeys:** translate EONull and **nil** between NSDictionaries and enterprise objects, removing the need for your objects to explicitly test for EONull values.

Relationship Accessor Methods

Building on the key-value coding methods, another group of methods allows you to modify relationship properties by adding and removing single objects, rather than replacing the entire content of the relationship, and to modify relationships so that reciprocal relationships are automatically adjusted. **addObject:toPropertyWithKey:** and **removeObject:fromPropertyWithKey:** handle the first situation, doing all the work of altering arrays for to-many relationships. They both check first for a method you might implement, **addTo***Key***:** or **removeFrom***Key***:**, invoking that method if it's implemented, otherwise using the basic key-value coding methods to do the work.

Reciprocal relationships are handled by **addObject:toBothSidesOfRelationshipWithKey:** and **removeObject:fromBothSidesOfRelationshipWithKey:**. For example, when you add an Employee to a Department's **employees** relationship, or remove it, you also want the Employee's **department** relationship altered to suit. These two methods take care of tracing the inverse relationship and use **addObject:toPropertyWithKey:** and **removeObject:fromPropertyWithKey:** to alter both relationships, whether they're to-one or to-many. Unless you have specific reasons to do otherwise, you should always use the methods that handle reciprocal relationships so that back pointers are properly updated.

Two other methods that affect relationships are typically used internally by the Framework. You should rarely have a need either to invoke or override them. **propagateDeleteWithEditingContext:** applies an object's delete rule to the destinations of its relationships. The delete rule specifies whether a destination object should be ignored, also deleted, or whether the deletion should be disallowed if a destination exists. **clearProperties** simply sets all of the receiver's relationship properties to **nil**. An EOEditingContext uses this method to break circular references between its objects when the context is deallocated.

Snapshots

The key-value coding methods define a general mechanism for accessing an object's properties, but you first have to know what those properties are. Sometimes, however, you just want to preserve an object's entire state for later use, whether to undo changes to the object, compare the values that have changed, or just keep a record of the changes. The snapshotting methods provide this service, extracting or setting all properties at once and performing the necessary conversions for proper behavior. **snapshot** returns an NSDictionary containing all the receiver's properties, with EONull substituted for **nil** and arrays reproduced as shallow, immutable copies. **updateFromSnapshot:** sets properties of the receiver to the values in a snapshot, converting EONull to **nil**, and making shallow, mutable copies of any array values (allowing the object to add to and remove from the array).

Validation

Validating new values is a vital part of business logic. Several methods added to NSObject support validation at different stages of an object's life. Validation methods check for illegal value types, values outside of established limits, illegal relationships, and so on. All validation methods return **nil** if the values under consideration are valid, or an NSException indicating how the values aren't valid.

There are two kinds of validation methods that you can override. The first covers individual properties, when it's important to validate a value before it changes. These methods are invoked automatically by the Framework when it changes a property value, such as when the user makes an edit in the user interface. These methods are dynamically invoked based on the property name. The second kind covers operations to the external store—inserting, updating, and deleting. These methods are invoked when the associated operation is performed. You can override these methods in your custom enterprise object classes to perform delayed validation of properties, to compare multiple properties against one another, and to allow or refuse the operation based on property values. For example, a Fee object might refuse to be deleted if it hasn't been paid yet.

Immediate Validation of Individual Properties

The most general method, **validateValue:forKey:**, is used by the Framework when an EODisplayGroup passes an updated value to the object and when the object is saved. This method does two things: coerce the value into an appropriate type for the object, and validate it according to the object's rules. Coercion is performed automatically for you, so all you need handle is validation itself.

The default implementation of **validateValue:forKey:** consults the object's EOClassDescription for basic errors, such as a **nil** value when that isn't allowed. If no basic errors exist, this method then examines the object's class itself for a method of the form **validate***Key***:** and invokes that. These are the methods that your custom classes can implement to validate individual properties, such as **validateAge:** to check that the value the user entered is within acceptable limits.

For example, suppose that Member objects have an **age** attribute stored as an integer. This attribute has an lower limit of 16, defined by the Member class. Now, suppose the user types "12" into a text field for the age. Before the EODisplayGroup updates the selected object, it sends the object a **validateValue:forKey:** message. The object uses its EOEntityClassDescription to convert the string "12" into an NSNumber, then invokes **validateAge:** with that NSNumber. Member's implementation of this method compares the age to its limit of 16 and returns an EOValidationException:

```
- (NSException *)validateAge:(NSNumber *)age
{
    if ([age intValue] < 16) {
        return [NSException
            validationExceptionWithFormat:@"Age of %@ is below minimum.", age];
    }
    return nil;
}
```

The Framework adds the **validationExceptionWithFormat:** method to NSException for convenient creation of validation exceptions. The userInfo dictionaries in the exceptions raised by these methods contain the enterprise object being validated and the key (where applicable).

Validation for Specific Operations

The other validation methods are invoked at specific times—such as before the object is written to or deleted from the external store—and are particularly useful when properties must be compared or when expensive calculation is necessary. The methods are **validateForInsert**, **validateForUpdate**, **validateForSave**, and **validateForDelete**, and they're invoked automatically for the operations indicated by the method name. You can override these methods to check values individually or in groups; for example, you might verify that a pair of dates is in the proper temporal order:

```
- (NSException *)validateForSave
{
    if ([startDate compare:endDate] == NSOrderedDescending) {
        return [NSException
            validationExceptionWithFormat:@"Start date must not follow end date."];
    }
    return [super validateForSave];
}
```

Note that this method also invokes **super**'s implementation. This is important, as the default implementations of the **validateFor...** pass the check on to the object's EOClassDescription, which performs basic checking among properties. The access layer's EOEntityClassDescription class verifies constraints based on an EOModel, such as delete rules. For example, the delete rule for a Department object might state that it can't be deleted if it still contains Employee objects.

validateForSave is the generic validation method for when an object is written to the external store. The default implementations of **validateForInsert** and **validateForUpdate** both invoke this method. If an object performs validation that isn't specific to insertion or to updating, it should go in **validateForSave**.

Method Types

Initializing enterprise objects	– initWithEditingContext:classDescription:globalID: – awakeFromFetchInEditingContext: – awakeFromInsertionInEditingContext:
Announcing changes	– willChange
Getting an object's EOEditingContext	– editingContext

Getting class description information	– allPropertyKeys – attributeKeys – classDescription – classDescriptionForDestinationKey: – deleteRuleForRelationshipKey: – entityName – inverseForRelationshipKey: – isToManyKey: – ownsDestinationObjectsForRelationshipKey: – toManyRelationshipKeys – toOneRelationshipKeys
Key-value coding	– takeValue:forKey: – valueForKey: – takeValuesFromDictionary: – valuesForKeys: – takeValue:forKeyPath: – valueForKeyPath: – handleQueryWithUnboundKey: – handleTakeValue:forUnboundKey: – unableToSetNilForKey: + accessInstanceVariablesDirectly + flushClassKeyBindings + flushAllKeyBindings
Modifying relationships	– addObject:toPropertyWithKey: – removeObject:fromPropertyWithKey: – addObject:toBothSidesOfRelationshipWithKey: – removeObject:fromBothSidesOfRelationshipWithKey: – propagateDeleteWithEditingContext: – clearProperties
Working with snapshots	– snapshot – updateFromSnapshot:
Validating values	– validateForDelete – validateForInsert – validateForSave – validateForUpdate – validateValue:forKey:
Getting descriptions	– eoDescription – eoShallowDescription

Class Methods

accessInstanceVariablesDirectly

+ (BOOL)**accessInstanceVariablesDirectly**

Returns YES if the default implementations of the key-value coding methods, on finding no accessor method for a property, should access the corresponding instance variable directly. Returns NO if they shouldn't. NSObject's implementation of this method returns YES. Subclasses can override it to return NO, in which case the other methods won't access instance variables.

flushAllKeyBindings

+ (void)**flushAllKeyBindings**

Invalidates the cached key binding information for all classes (caches are kept of key-to-method or instance variable bindings in order to make key-value coding efficient).

See also: + **flushClassKeyBindings**

flushClassKeyBindings

+ (void)**flushClassKeyBindings**

Invalidates the cached key binding information for the receiving class. This method should be invoked whenever a class is modified or removed from the run-time system.

See also: + **flushAllKeyBindings**

Instance Methods

addObject:toBothSidesOfRelationshipWithKey:

– (void)**addObject:**(id)*anObject* **toBothSidesOfRelationshipWithKey:**(NSString *)*key*

Sets or adds *anObject* as the destination for the receiver's relationship identified by *key*, and also sets or adds the receiver for *anObject*'s reciprocal relationship if there is one. For a to-one relationship, *anObject* is set using **takeValue:forKey:**. For a to-many relationship, *anObject* is added using **addObject:toPropertyWithKey:**.

This method also properly handles removing **self** and *anObject* from their previous relationship as needed. For example, if an Employee object belongs to the Research department, invoking this method with the Maintenance department removes the Employee from the Research department as well as setting the Employee's department to Maintenance.

See also: – **removeObject:fromBothSidesOfRelationshipWithKey:**

addObject:toPropertyWithKey:

– (void)**addObject:**(id)*anObject* **toPropertyWithKey:**(NSString *)*key*

Adds *anObject* to the receiver's to-many relationship identified by *key*, without setting any reciprocal relationship. Similar to **takeValue:forKey:**, this method first attempts to invoke a method of the form **addTo***Key***:**. If the receiver doesn't have such a method, this method gets the property array using **valueForKey:** and operates directly on that.

If the array is mutable, this method simply adds *anObject* using **addObject:**. Otherwise it constructs a new array containing any existing objects and *anObject*, then sets it using **takeValue:forKey:**.

See also: – **removeObject:fromPropertyWithKey:**, – **addObject:toBothSidesOfRelationshipWithKey:**

allPropertyKeys

– (NSArray *)**allPropertyKeys**

Returns all of the receiver's property keys, as returned by **attributeKeys**, **toOneRelationshipKeys**, and **toManyRelationshipKeys**.

attributeKeys

– (NSArray *)**attributeKeys**

Returns the names of the receiver's attributes, as determined from the EOClassDescription. You might wish to override this method to add keys for attributes not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of attributes designated as class properties.

See also: – **toOneRelationshipKeys**, – **toManyRelationshipKeys**.,
– **attributeKeys** (EOClassDescription)

awakeFromFetchInEditingContext:

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

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

See also: – **awakeFromInsertionInEditingContext:**

awakeFromInsertionInEditingContext:

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

Overridden by subclasses to perform additional initialization upon being inserted into *anEditingContext*. This is commonly used to assign default values or record the time of insertion. NSObject's implementation merely sends an **awakeObject:fromInsertionInEditingContext:** to the receiver's EOClassDescription. Subclasses should invoke **super**'s implementation before performing their own initialization.

See also: – **awakeFromFetchInEditingContext:**

classDescription

– (EOClassDescription *)**classDescription**

Returns the EOClassDescription registered for the receiver's class. If none is found, posts an EOClassDescriptionNeededNotification on behalf of the receiver's class, allowing an observer to register a an EOClassDescription. See the EOClassDescription class specification for more information.

See also: + **registerClassDescription:forClass:** (EOClassDescription)

classDescriptionForDestinationKey:

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

Returns the EOClassDescription for the destination objects of the relationship identified by *key*. If none is found, posts an EOClassDescriptionNeededNotification on behalf of the destination objects' class, allowing an observer to register a an EOClassDescription. See the EOClassDescription class specification for more information.

See also: + **registerClassDescription:forClass:** (EOClassDescription),
– **classDescriptionForDestinationKey:**

clearProperties

– (void)**clearProperties**

Sets all of the receiver's to-one and to-many relationships to **nil**. EOEditingContexts use this method to break circular references among objects when they're deallocated. You should never need to invoke this method or override it.

See also: – **toOneRelationshipKeys**, – **toManyRelationshipKeys**, – **takeValue:forKey:**

deleteRuleForRelationshipKey:

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

Returns a rule indicating how to handle the destination of the receiver's relationship named by *relationshipKey* when the receiver is deleted. The delete rule is one of:

EODeleteRuleNullify
EODeleteRuleCascade
EODeleteRuleDeny

For example, an Invoice object might return EODeleteRuleCascade for the relationship named "lineItems", since when an invoice is deleted, its line items should be as well.

See also: – **propagateDeleteWithEditingContext:**, – **validateForDelete**, – **deleteRuleForRelationshipKey:** (EOClassDescription)

editingContext

– (EOEditingContext *)**editingContext**

Returns the EOEditingContext that holds the receiver.

entityName

– (NSString *)**entityName**

Returns the name of the receiver's entity, or **nil** if it doesn't have one.

See also: – **entityName** (EOClassDescription)

eoDescription

– (NSString *)**eoDescription**

Returns a full description of the receiver's property values by extracting them via 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 define your enterprise object's **description** method to invoke this one, so that 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.

See also: – **eoShallowDescription**

eoShallowDescription

– (NSString *)**eoShallowDescription**

Returns a string containing the receiver's class and entity names, along with the memory address of its **id**.

See also: – **eoDescription**

handleQueryWithUnboundKey:

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

Invoked from **valueForKey:** when it finds no property binding for *key*. NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the query in some other way.

See also: – **handleTakeValue:forUnboundKey:**

handleTakeValue:forUnboundKey:

– (void)**handleTakeValue:**(id)*value* **forUnboundKey:**(NSString *)*key*

Invoked from **takeValue:forKey:** when it finds no property binding for *key*. NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the request in some other way.

See also: – **handleQueryWithUnboundKey:**

initWithEditingContext:classDescription:globalID:

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

Overridden by subclasses to perform initialization with the extra arguments provided. NSObject's implementation simply invokes **init**.

See also: – **createInstanceWithEditingContext:classDescription:globalID:** (EOClassDescription),

inverseForRelationshipKey:

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

Returns the name of the relationship pointing back to the receiver's class or entity from that named by *relationshipKey*, or **nil** if there isn't one. With EOEntity and EORelationship, for example, reciprocality is determined by the join attributes of the two EORelationships.

You might override this method for reciprocal relationships that aren't defined using the same join attributes. For example, if a Member object has a relationship to CreditCard based on the card number, but a CreditCard has a relationship to Member based on the Member's primary key, both classes need to override this method. This is how Member might implement it:

```
- (NSString *)inverseForRelationshipKey:(NSString *)relationshipKey
{
    if ([relationshipKey isEqual:@"creditCard"]) return @"member";
    return [super inverseForRelationshipKey:relationshipKey];
}
```

See also: – **inverseForRelationshipKey:** (EOClassDescription)

isToManyKey:

– (BOOL)**isToManyKey:**(NSString *)*key*

Returns YES if the receiver has a to-many relationship identified by *key,* NO otherwise.

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

ownsDestinationObjectsForRelationshipKey:

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

Returns YES if the receiver has a relationship identified by *key* that owns its destination, NO otherwise. If an object owns the destination for a relationship, then when that destination object is removed from the relationship, it's automatically deleted. Ownership of a relationship thus contrasts with a delete rule, in that the first applies when the destination is removed and the second applies when the source is deleted.

See also: – **deleteRuleForRelationshipKey:**,
– **ownsDestinationObjectsForRelationshipKey:** (EOClassDescription),
– **ownsDestination** (EORelationship)

propagateDeleteWithEditingContext:

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

Sends a **propagateDeleteForObject:editingContext:** message to the receiver's EOClassDescription. This causes the destination objects of the receiver's relationships to be deleted according to the delete rule for each relationship:

Delete Rule	Action
EODeleteRuleNullify	The destination object is simply removed from the relationship, and the receiver is likewise removed from the destination's reciprocal relationship if there is one.
EODeleteRuleCascade	As above, but the destination object is also deleted and sent a **propagateDeleteWithEditingContext:** message.
EODeleteRuleDeny	Applied in **validateForDelete**, not in this method.

See also: – **deleteRuleForRelationshipKey:**

removeObject:fromBothSidesOfRelationshipWithKey:

– (void)**removeObject:**(id)*anObject* **fromBothSidesOfRelationshipWithKey:**(NSString *)*key*

Removes *anObject* from the receiver's relationship identified by *key*, and also removes the receiver from *anObject*'s reciprocal relationship if there is one. For a to-one relationship, *anObject* is removed using **takeValue:forKey:** with **nil** as the value. For a to-many relationship, *anObject* is removed using **removeObject:fromPropertyWithKey:**.

See also: – **addObject:toBothSidesOfRelationshipWithKey:**

removeObject:fromPropertyWithKey:

– (void)**removeObject:**(id)*anObject* **fromPropertyWithKey:**(NSString *)*key*

Removes *anObject* from the receiver's to-many relationship identified by *key*, without modifying any reciprocal relationship. Similar to **takeValue:forKey:**, this method first attempts to invoke a method of the form **removeFrom***Key***:**. If the receiver doesn't have such a method, this method gets the property array using **valueForKey:** and operates directly on that.

If the array is mutable, this method simply locates *anObject* and removes it using **removeObjectAtIndex:**. Otherwise it constructs a new array containing any existing objects minus *anObject*, then sets it using **takeValue:forKey:**.

See also: – **addObject:toPropertyWithKey:**, – **removeObject:fromBothSidesOfRelationshipWithKey:**

snapshot

– (NSDictionary *)**snapshot**

Returns an NSDictionary whose keys are those of the receiver's attributes, to-one relationships, and to-many relationships, and whose values are the values of those properties, with EONull substituted for **nil**.

For to-many relationships, the dictionary contains shallow copies of the arrays to preserve the **id**s of the contents.

See also: – **updateFromSnapshot:**, – **allPropertyKeys**, – **valueForKey:**

takeValue:forKey:

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

Sets the value for the property identified by *key* to *value*. NSObject's implementation does so by first checking the receiver for a selector of the form **set***Key***:**, invoking it if there is one. If there's no such method, and **accessInstanceVariablesDirectly** returns YES, NSObject's implementation checks for an instance variable named *key* and sets the value directly, autoreleasing the old value and retaining the new one.

If there's neither an accessor method nor an instance variable matching *key*, NSObject's implementation invokes **handleTakeValue:forUnboundKey:** as a fallback mechanism. Subclasses can override **handleTakeValue:forUnboundKey:** to handle the request in some other way.

See also: – **takeValue:forKeyPath:**, – **takeValuesFromDictionary:**, – **valueForKey:**

takeValue:forKeyPath:

– (void)**takeValue:**(id)*value* **forKeyPath:**(NSString *)*keyPath*

Sets the value for the derived property identified by *keyPath* to *value*. A key path has the form *relationship.property* (with one or more relationships); for example "department.name". NSObject's implementation of this method gets the destination object for each relationship using **valueForKey:**, and sends the final object a **takeValue:forKey:** message with *value* and *property*.

See also: – **takeValuesFromDictionary:**, – **valueForKeyPath:**

takeValuesFromDictionary:

– (void)**takeValuesFromDictionary:**(NSDictionary *)*aDictionary*

Sets properties of the receiver with values from *aDictionary*, using their keys to identify the properties. NSObject's implementation invokes **takeValue:forKey:** for each key-value pair, substituting **nil** for EONull values in *aDictionary*.

See also: – **updateFromSnapshot:**, – **takeValue:forKeyPath:**, – **valuesForKeys:**

toManyRelationshipKeys

– (NSArray *)**toManyRelationshipKeys**

Returns the names of the receiver's to-many relationships, as determined from the EOClassDescription. You might wish to override this method to add keys for relationships not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-many relationships designated as class properties.

See also: – **toOneRelationshipKeys**., – **attributeKeys**, – **attributeKeys**,
– **toManyRelationshipKeys** (EOClassDescription)

toOneRelationshipKeys

– (NSArray *)**toOneRelationshipKeys**

Returns the names of the receiver's to-one relationships, as determined from the EOClassDescription. You might wish to override this method to add keys for relationships not defined by the EOClassDescription. The access layer's subclass of EOClassDescription, EOEntityClassDescription, returns the names of to-one relationships designated as class properties.

See also: – **toManyRelationshipKeys**., – **attributeKeys**, – **attributeKeys**,
– **toOneRelationshipKeys** (EOClassDescription)

unableToSetNilForKey:

– (void)**unableToSetNilForKey:**(NSString *)*key*

Invoked from **takeValue:forKey:** when it's given a **nil** value for a scalar property (such as an **int** or a **float**). NSObject's implementation raises an NSInvalidArgumentException. Subclasses can override it to handle the request in some other way, such as by substituting zero or a sentinel value and invoking **takeValue:forKey:** again.

updateFromSnapshot:

– (void)**updateFromSnapshot:**(NSDictionary *)*aSnapshot*

Takes the values from *aSnapshot*, setting each one according to its key using **takeValue:forKey:**. In the process, EONull values are converted to **nil**, and array values are set as shallow mutable copies to preserve the **id**s of the contents.

See also: – **takeValuesFromDictionary:**, – **snapshot**

validateForDelete

– (NSException *)**validateForDelete**

Confirms that the receiver can be deleted in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. For example, an object can't be deleted if it has a relationship with a delete rule of EODeleteRuleDeny and that relationship has a destination object.

NSObject's implementation sends the receiver's EOClassDescription a **validateObjectForDelete:** message (which performs basic checking based on the presence or absence of values). Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForDelete
{
    NSException *exception = [super validateForDelete];

    if (/* some other violation */ ) {
        NSException *newException = /* the extra exception */;
        exception = [NSException aggregateExceptionWithExceptions:[NSArray
             arrayWithObjects:exception, newException, nil]];
    }

    return exception;
}
```

See also: – **validateForInsert**, – **validateForSave**, – **validateForUpdate**, – **validateValue:forKey:**, + **validationExceptionWithFormat:** (NSException Additions)

validateForInsert

– (NSException *)**validateForInsert**

Confirms that the receiver can be inserted in its current state, returning **nil** if it can or an NSException that can be raised if it can't. NSObject's implementation simply invokes **validateForSave**.

See also: – **validateForDelete**, – **validateForSave**, – **validateForUpdate**, – **validateValue:forKey:**, + **validationExceptionWithFormat:** (NSException Additions)

validateForSave

– (NSException *)**validateForSave**

Confirms that the receiver can be saved in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation sends the receiver's EOClassDescription a **validateObjectForSave:** message, then iterates through all of the receiver's properties, invoking **validateValue:forKey:** for each one. If this results in more than one exception, the exception returned

contains the additional ones in its **userInfo** dictionary under the EOAdditionalExceptions key. Subclasses should invoke **super**'s implementation before performing their own validation, and should combine any exception returned by **super**'s implementation with their own:

```
- (NSException *)validateForSave
{
    NSException *exception = [super validateForSave];

    if (/* some other violation */ ) {
        NSException *newException = /* the extra exception */;
        exception = [NSException aggregateExceptionWithExceptions:[NSArray
             arrayWithObjects:exception, newException, nil]];
    }

    return exception;
}
```

Enterprise objects can implement this method to check that certain relations between properties hold; for example, that the end date of a vacation period follows the begin date. To validate an individual property, you can simply implement a method for it as described under **validateValue:forKey:**.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForUpdate**,
+ **validationExceptionWithFormat:** (NSException Additions),
+ **aggregateExceptionWithExceptions:** (NSException Additions)

validateForUpdate

– (NSException *)**validateForUpdate**

Confirms that the receiver can be updated in its current state, returning **nil** if it can or an NSException that the sender may raise if it can't. NSObject's implementation simply invokes **validateForSave**.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForSave**, – **validateValue:forKey:**,
+ **validationExceptionWithFormat:** (NSException Additions)

validateValue:forKey:

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

Confirms that the value referenced by *valuePointer* is legal for the receiver, returning **nil** if it can or an EOValidationException that the sender may raise if it can't. NSObject's implementation sends a **validateValue:forKey:** message to the receiver's EOClassDescription. If that message doesn't return an exception, it checks for a method of the form **validate***Key***:** (for example, **validateBudget:** for a *key* of "budget") and invokes it, returning the result.

Enterprise objects can implement individual **validate***Key***:** methods to check limits, test for nonsense values, and otherwise confirm individual properties. To validate multiple properties based on relations among them, override the appropriate **validateFor...** method.

See also: – **validateForDelete**, – **validateForInsert**, – **validateForSave**, – **validateForUpdate**, + **validationExceptionWithFormat:** (NSException Additions)

valueForKey:

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

Returns the value for the property identified by *key.* NSObject's implementation does so by first checking the receiver for a method named *key*, invoking it if there is one. If there's no such method, and **accessInstanceVariablesDirectly** returns YES, NSObject's implementation checks for an instance variable named *key* and returns the instance variable.

If there's neither an accessor method nor an instance variable matching *key*, NSObject's implementation invokes **handleQueryWithUnboundKey:** as a fallback mechanism. Subclasses can override **handleQueryWithUnboundKey:** to handle the request in some other way.

See also: – **valueForKeyPath:**, – **valuesForKeys:**, – **takeValue:forKey:**

valueForKeyPath:

– (id)**valueForKeyPath:**(NSString *)*keyPath*

Returns the value for the derived property identified by *keyPath*. A key path has the form *relationship.property* (with one or more relationships); for example "department.name". NSObject's implementation of this method gets the destination object for each relationship using **valueForKey:**, and returns the result of a **valueForKey:** message to the final object.

See also: – **valuesForKeys:**, – **takeValue:forKeyPath:**

valuesForKeys:

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

Returns an NSDictionary containing the property values identified by each of *keys*. NSObject's implementation invokes **valueForKey:** for each key in *keys*, substituting EONull in the NSDictionary for returned **nil** values.

See also: – **valueForKeyPath:**, – **takeValuesFromDictionary:**

willChange

– (void)**willChange**

Notifies any observers that the receiver's state is about to change, by sending each an **objectWillChange:** message. See the EOObserverCenter class specification for more information.