The EOAccess Framework

Framework: /System/Library/Frameworks/EOAccess.framework

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

Introduction

The EOAccess framework is one of a group of frameworks known collectively as the Enterprise Objects Framework. The classes and protocols that make up the EOAccess framework allow your applications to interact with database servers at a high level of abstraction. These classes make up what is known as the *access layer*. The access layer is divided into two main parts:

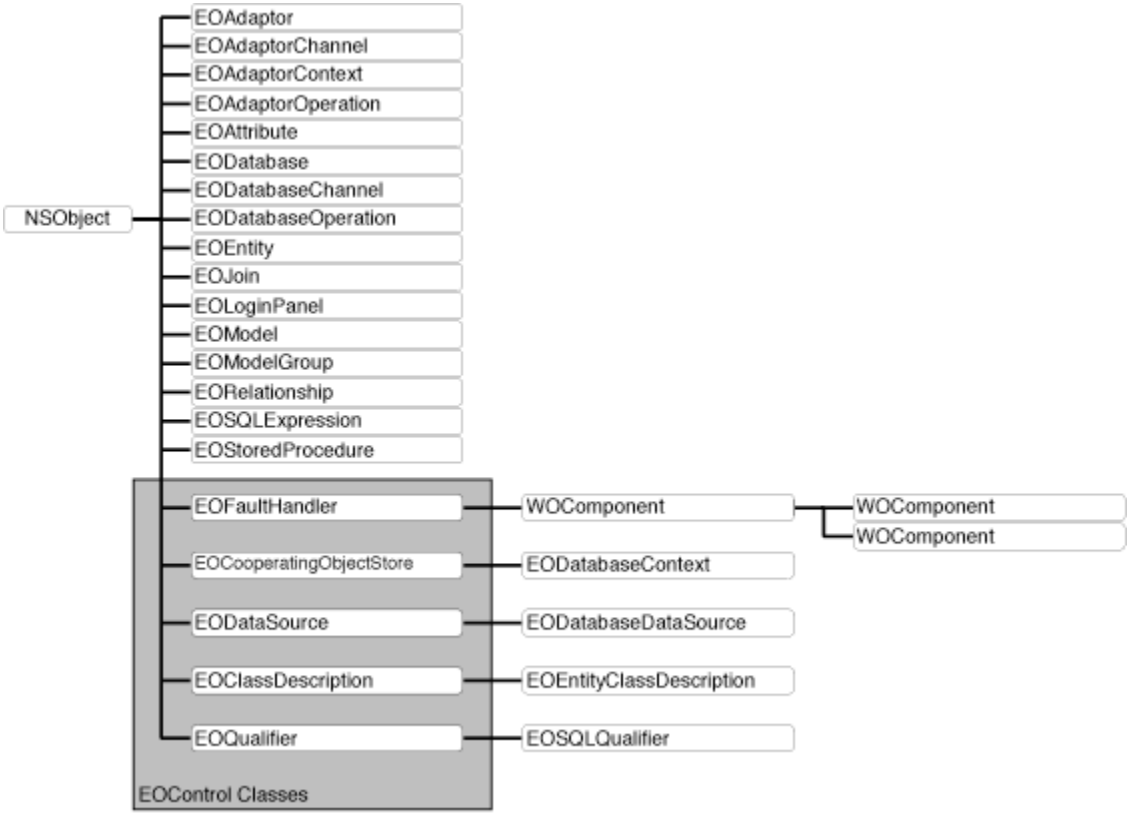
- The *database level*, which allows applications to treat records as full-fledged enterprise objects.
- The *adaptor level*, which provides server-independent database access.

Working with the access layer allows you to have a finer level of control over database operations.

EOAccess Framework Class Hierarchy

The EOAccess class hierarchy is rooted in the Foundation Framework's NSObject class (see Figure 1). The remainder of the EOAccess Framework consists of several related groups of classes, a few miscellaneous classes, and a number of protocols.

Figure 1 The EOAccess Framework class hierarchy



The Database Level

The database level is where enterprise objects are created from the dictionaries retreived by the adaptor level. It's also where snapshotting is performed. The database level is primarily made up of the following classes:

- **EODatabase** is a class that represents a single database server.
- **EODatabaseChannel** is a class that represents an independent communication channel to the database server.

- **EODatabaseContext** is subclass of EOObjectStore for accessing relational databases, creating and saving objects based on EOEntity definitions in an EOModel.
- **EODatabaseOperation** is a class that represents an operation—insert, update, or delete—to perform on an enterprise object and all the necessary information required to perform the operation.

The Adaptor Level

The adaptor level deals with database rows packaged as dictionaries. The adaptor level is primarily made up of the following classes:

- **EOAdaptor** is an abstract class that provides concrete subclasses with a structure for connecting to a database.
- **EOAdaptorChannel** is an abstract class that provides its concrete subclasses with a structure for performing database operations.
- **EOAdaptorContext** is an abstract class that defines transaction handling in Enterprise Objects Framework applications.
- **EOAdaptorOperation** is a class that represents a primitive operation in a database server and all the necessary information required by the operation.

The Modeling Classes

A model defines, in entity-relationship terms, the mapping between enterprise object classes and a database. The following are the principal modeling classes in the EOAccess framework:

- **EOAttribute** is a class that represents a column, field or property in a database, and associates an internal name with an external name or expression by which the property is known to the database.
- **EOEntity** is a class that describes a table in a database and associates a name internal to the Framework with an external name by which the table is known to the database.
- **EOJoin** is a class that describes one source-destination attribute pair for an EORelationship.
- **EOModel** is a class that represents a mapping between a database schema and a set of classes based on the entity-relationship model.
- **EOModelGroup** is a class that represents an aggregation of related models.
- **EORelationship** is a class that describes an association between two entities, based on attributes of those two entities.

Faulting

These classes implement or are used to implement object faulting:

EOAccessArrayFaultHandler is a subclass of EOAccessGenericFaultHandler that implements a fault for an array of enterprise objects.

EOAccessFaultHandler is a subclass of EOAccessGenericFaultHandler that implements an object fault for enterprise objects.

EOAccessGenericFaultHandler is an abstract class that helps an EOAccessFault to fire by fetching data using an EODatabaseContext.

Additions to Other Frameworks

The EOAccess framework adds methods to a number of classes in different frameworks:

EOGenericRecord Additions adds one method to the control layer's class, for returning a generic record's associated EOEntity.

EOObjectStoreCoordinator Additions adds two methods to the EOControl class for accessing the coordinator's EOModelGroup.

EOQualifier Additions adds one method to the class, for "rerooting" a qualifier to another entity.

NSString Additions adds two methods to the class, to convert modeling object names to database schema names, and database schema names to modeling object names

Miscellaneous Classes

The EOAccess framework also has a number of other useful classes, including:

- **EODatabaseDataSource** is a concrete subclass of EODataSource that fetches objects based on an EOModel, using an EODatabaseContext that services the data source's EOEditingContext.
- **EOEntityClassDescription** is a subclass of the control layer's EOClassDescription and extends the behavior of enterprise objects by deriving information about them from an associated EOModel.
- **EOLoginPanel** is an abstract class that defines how users provide database login information.
- **EOSQLExpression** is an abstract superclass that defines how to build SQL statements for adaptor channels.
- **EOSQLQualifier** is a subclass of EOQualifier that contains unstructured text that can be transformed into an SQL expression.
- **EOStoredProcedure** is a class that represents a stored procedure defined in a database, and associates a name internal to EOF with an external name known to the database.

Delegates

A number of EOAccess classes delegate behavior. The delegate methods are defined in these Objective-C protocols:

- An **EOAdaptorChannel** delegate receives messages for nearly every operation that would affect data in the database server, and it can preempt, modify, or track these operations.
- A **EOAdaptorContext** delegate receives messages for any transaction begin, commit, or rollback, and it can preempt, modify, or track these operations.
- An **EOAdaptor** delegate implements a method that can perform a database-specific transformations on a value.
- An **EODatabaseContext** delegate can intervene when objects are created and when they're fetched from the database.
- An **EOModelGroupClass** delegate implements a method that returns the default model group.
- An **EOModelGroup** delegate influences how the model group finds and loads models.

Miscellaneous Protocols

- **EOCustomClassArchiving** is an informal protocol that defines methods that can write any object that conforms to NSCoding to the database as binary data, as generated by NSArchiver.
- **EOEditingContext Additions** is a collection of convenience methods intended to make common operations with EOF easier.
- **EOPropertyListEncoding** declares methods that read and write objects to property lists.
- **EOQualifierSQLGeneration** declares two methods that are adopted by qualifier classes to qualify fetches from a database.

EOAccessArrayFaultHandler

Inherits From: EOAccessGenericFaultHandler :EOFaultHandler (EOControl) : NSObject

Declared In: EOAccess/EOAccessFault.h

Class Description

EOAccessArrayFaultHandler is a subclass of EOAccessGenericFaultHandler that implements a fault for an array of enterprise objects.

Instance Methods

completeInitializationOfObject

– (void)completeInitializationOfObject:(id)object;

Asks the receiver's database context to fetch the object if it is not already in memory. This method is called when the fault is fired and uses the EOObjectStore protocol to get the information from the reciever's editing context

databaseContext

– (EODatabaseContext *)**databaseContext**

Returns the receiver's database context.

editingContext

– (EOEditingContext *)**editingContext**

Returns the receiver's editing context.

initWithSourceGlobalID:relationshipName:databaseContext:editingContext:

– **initWithSourceGlobalID:**(EOKeyGlobalID *)*sourceGID*
relationshipName:(NSString *)*relationshipName*
databaseContext:(EODatabaseContext *)*databaseContext*
editingContext:(EOEditingContext *)*editingContext*

Initializes the handler with all of the information necessary to fetch the appropriate objects when the fault is fired. When the fault is fired, the database context asks the editing context for the required objects using the EOObjectStore protocol.

relationshipName

– (NSString *)**relationshipName**

Returns the receiver’s relationship name.

sourceGlobalID

– (EOKeyGlobalID *)**sourceGlobalID**

Returns the receiver’s source global ID.

EOAccessFaultHandler

Inherits From: EOAccessGenericFaultHandler :EOFaultHandler (EOControl) : NSObject

Declared In: EOAccess/EOAccessFault.h

Class Description

EOAccessFaultHandler is a subclass of EOAccessGenericFaultHandler that implements an object fault for enterprise objects.

Instance Methods

completeInitializationOfObject

– (void)**completeInitializationOfObject:**(id)*anObject*;

Asks the receiver's database context to fetch *anObject* if it is not already in memory. This method is called called when the fault is fired and uses the EOObjectStore protocol to get the information from the receiver's editing context.

databaseContext

– (EODatabaseContext *)**databaseContext**

Returns the receiver's database context.

editingContext

– (EOEditingContext *)**editingContext**

Returns the receiver's editing context.

globalID

– (EOKeyGlobalID *)**globalID**

Returns the receiver's global ID.

initWithglobalID:relationshipName:databaseContext:editingContext:

– **initWithGlobalID:**(EOKeyGlobalID *)*globalID*
databaseContext:(EODatabaseContext *)*databaseContext*
editingContext:(EOEditingContext *)*editingContext*

Initializes the handler with all of the information necessary to fetch the object when the fault is fired. When the fault is fired, this object calls **completeInitializationOfObject** on the object.

EOAccessGenericFaultHandler

Inherits From: EOFaultHandler (EOControl) : NSObject

Declared In: EOAccess/EOAccessFault.h

Class Description

EOAccessGenericFaultHandler is an abstract class that helps an EOAccessFault to fire by fetching data using an EODatabaseContext. Don't use EOAcceessGenericFaultHandler directly; instead, use its subclasses EOAccessFaultHandler and EOAccessArrayFaultHandler.

EOAccessGenericFaultHandler lets you chain together all the fault handlers in the access layer, so the batch faulting mechanism can find other faults related to the one that triggered the batch. Use **linkAfter: usingGeneration:** to link one fault after another. Use **next** and **previous** to traverse the chain.

Instance Methods

generation

- (unsigned int)**generation**

Returns the the receiver's generation, a number that represents when the fault handler was built.

linkAfter:usingGeneration:

- (void)**linkAfter:**(EOAccessGenericFaultHandler *)*faultHandler*
  usingGeneration:(unsigned int)*generation*

Adds the receiver to a chain of fault handlers, after *faultHandler. generation* is a number that represents when the handler was built. All faults in an access layer can be chained together, so the batch faulting mechanism can find other faults related to the one that triggered the batch.

See also: – **next**, – **previous**

next

- (EOAccessGenericFaultHandler *)**next**

Returns the next fault in the chain.

previous

public EOAccessGenericFaultHandler **previous**()
- (EOAccessGenericFaultHandler *)**previous**

Returns the previous fault in the chain.

EOAdaptor

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOAdaptor.h

Class Description

EOAdaptor is an abstract class that provides concrete subclasses with a structure for connecting to a database. A concrete subclass of EOAdaptor provides database-specific method implementations and represents a single database server. You never interact with instances of the EOAdaptor class, but you use its class methods, **adaptorWithName:** and **adaptorWithModel:**, to create instances of a concrete subclass. The EOAdaptor class defines the methods that find and load the concrete adaptors from bundles. However, you rarely interact with a concrete adaptor either. Generally, adaptors are automatically created and used by other classes in the Enterprise Objects Framework.

The EOAdaptor class has the following principal attributes:

- Dictionary of connection information
- Login panel
- Array of adaptor contexts
- Expression class

Other framework classes create EOAdaptor objects. **adaptorWithModel:** creates a new adaptor with the adaptor name in the specified model. **adaptorWithName:** creates a new adaptor with the specified name.

The following table lists the most commonly-used methods in the EOAdaptor class:

assertConnectionDictionaryIsValid	Verifies that the adaptor can connect with its connection information.
setConnectionDictionary:	Sets the connection dictionary.

– assertConnectionDictionaryIsValid	Verifies that the adaptor can connect with its connection information.
– runLoginPanel	Runs the login panel without affecting the connection dictionary.
– runLoginPanelAndValidateConnectionDictionary	Runs the login panel until the user enters valid connection information or cancels the panel.
– setConnectionDictionary:	Sets the connection dictionary.

For information on subclassing an EOAdaptor, see “Creating an EOAdaptor Subclass”.

Method Types

Creating an EOAdaptor

+ adaptorWithName:
+ adaptorWithModel:
– initWithName:

Accessing an adaptor’s name

– name

Accessing the names of all available adaptors

+ availableAdaptorNames

Accessing connection information

– assertConnectionDictionaryIsValid
– connectionDictionary
– setConnectionDictionary:
– runLoginPanelAndValidateConnectionDictionary
– runLoginPanel
– databaseEncoding

Performing database-specific transformations on values

– fetchedValueForValue:attribute:
– fetchedValueForDataValue:attribute:
– fetchedValueForDateValue:attribute:
– fetchedValueForNumberValue:attribute:
– fetchedValueForStringValue:attribute:

Servicing models

– canServiceModel:
+ internalTypeForExternalType:model:
+ externalTypesWithModel:
+ assignExternalInfoForEntireModel:
+ assignExternalInfoForEntity:
+ assignExternalInfoForAttribute:
– isValidQualifierTypeIn:model:

Creating adaptor contexts

– createAdaptorContext
– contexts

Checking connection status

– hasOpenChannels

Accessing a default expression class

+ setExpressionClassName:adaptorClassName:
– expressionClass
– defaultExpressionClass

Accessing an adaptor's login panel

+ sharedLoginPanelInstance
– runLoginPanelAndValidateConnectionDictionary
– runLoginPanel

Accessing the delegate

– delegate
– setDelegate:

Other

– createDatabaseWithAdministrativeConnectionDictionary:
– dropDatabaseWithAdministrativeConnectionDictionary:
– prototypeAttributes

Class Methods

adaptorWithModel:

+(id)**adaptorWithModel:**(EOModel *)*model*

Creates and returns a new adaptor by extracting the adaptor name from *model*, invoking **adaptorWithName:**, and assigning *model*'s connection dictionary to the new adaptor. Raises an NSInvalidArgumentException if *model* is **nil,** if *model*'s adaptor name is **nil**, or if the adaptor named in *model* can't be loaded.

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: **– adaptorName** (EOModel), – **setConnectionDictionary:**

adaptorWithName:

+ (id)**adaptorWithName:**(NSString *)*name*

Creates and returns a new adaptor, loading it from the framework named *name* if necessary and sending it an **initWithName:** message. For example, this code excerpt creates an adaptor from a framework named **AcmeEOAdaptor.framework**:

```
EOAdaptor *myAdaptor = [EOAdaptor adaptorWithName:@"Acme"];
```

This method searches the application's main bundle, **~/Library/Frameworks**, **Network/Library/Frameworks**, and **System/Library/Frameworks** for the first framework whose base filename (that is, the filename without the ".framework" extension) corresponds to *name*. However, note that dynamic loading isn't available on PDO platforms. Consequently, you must statically link your adaptor into applications for PDO: In this case, **adaptorWithName:** simply looks in the runtime for an adaptor class corresponding with the specified name. Raises an NSInvalidArgumentException if *name* is **nil** or if an adaptor class corresponding with *name* can't be found.

Usually you'd use **adaptorWithModel:** to create a new adaptor, but you can use this method when you don't have a model. In fact, this method is typically used when you're creating an adaptor for the purpose of creating a model from an existing database.

assignExternalInfoForAttribute:

+ (void)**assignExternalInfoForAttribute:**(EOAttribute *)*attribute*

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

Overridden by adaptor subclasses to assign database-specific characteristics to *attribute*. EOAdaptor's implementation invokes **assignExternalTypeForAttribute:** to assign an external type and then assigns a column name based on the attribute name. For example, **assignExternalInfoForAttribute:** assigns the column name "FIRST_NAME" to an attribute named "firstName". The method makes no changes to *attribute*'s column name if *attribute* is derived.

assignExternalInfoForEntireModel:

+ (void)**assignExternalInfoForEntireModel:**(EOModel *)*model*

Assigns database-specific characteristics to *model*. Used in EOModeler to switch a model's adaptor. This method examines each entity in *model*. If an entity's external name is not set and all of the entity's attribute's external names are not set, then this method uses **assignExternalInfoForEntity:** and **assignExternalInfoForAttribute:** to assign external names. If the entity's external name is set or if any of the entity's attributes' external names are set, then the method doesn't assign external names to the entity or any of its attributes. Regardless, this method invokes **assignExternalTypeForAttribute:** for each attribute in the model to assign external types.

A subclass of EOAdaptor doesn't need to override this method.

assignExternalInfoForEntity:

+ (void)**assignExternalInfoForEntity:**(EOEntity *)*entity*

Overridden by adaptor subclasses to assign database-specific characteristics to *entity*. EOAdaptor's implementation assigns an external name to *entity* based on *entity*'s name. For example, **assignExternalInfoForEntity:** assigns the external name "MOVIE" to an entity named "Movie". An adaptor subclass should override this method to assign additional database-specific characteristics, if any.

See also: + **assignExternalInfoForEntireModel:**

assignExternalTypeForAttribute:

+ (void)**assignExternalTypeForAttribute:**(EOAttribute *)*attribute*

Overridden by adaptor subclasses to assign the external type to *attribute*. EOAdaptor's implementation does nothing. A subclass of EOAdaptor should override this method to assign an external type using *attribute*'s internal type, precision, and length information.

See also: + **assignExternalInfoForEntireModel:**

availableAdaptorNames

+ (NSArray *)**availableAdaptorNames**

Returns an array containing the names of all available adaptors, as found by searching the paths returned by **NSStandardLibraryPaths()**. If no adaptors are found, this method returns an empty array.

See also: + **assignExternalInfoForEntireModel:**

externalTypesWithModel:

+ (NSArray *)**externalTypesWithModel:**(EOModel *)*model*

Implemented by subclasses to return the names of the database types (such as Sybase "varchar" or Oracle "NUMBER") for use with the adaptor. *model* is an optional argument that can be used to supplement the adaptor's set of database types with additional, user-defined database types. See your adaptor's documentation for information on if and how it uses *model*.

internalTypeForExternalType:model:

+ (NSString *)**internalTypeForExternalType:**(NSString *)*extType* **model:**(EOModel *)*model*

Implemented by subclasses to return the name of the Objective-C class used to represent values stored in the database as *extType*. *model* is an optional argument that can be used to supplement the adaptor's set of

type mappings with additional mappings for user-defined database types. See your adaptor's documentation for information on if and how it uses *model*. Returns **nil** if no mapping for *extType* is found.

An adaptor subclass should override this method without invoking EOAdaptor's implementation.

setExpressionClassName:adaptorClassName:

+ (void)**setExpressionClassName:**(NSString *)*sqlExpressionClassName* **adaptorClassName:** (NSString *)*adaptorClassName*

Sets the expression class for instances of the class named *adaptorClassName* to *sqlExpressionClassName*. If *sqlExpressionClassName* is **nil**, restores the expression class to the default. Raises an NSInvalidArgumentException if *adaptorClassName* is **nil** or the empty string.

Use this method to substitute a subclass of EOSQLExpression for the expression class provided by the adaptor. For example, the default expression class for the Oracle adaptor is OracleSQLExpression. The following statement substitutes the class named MySQLExpression:

```
[EOAdaptor setExpressionClassName:@"MySQLExpression" adaptorClassName:
@"OracleAdaptor"];
```

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: – **defaultExpressionClass**

Instance Methods

assertConnectionDictionaryIsValid

– (void)**assertConnectionDictionaryIsValid**

Implemented by subclasses to verify that the adaptor can connect to the database server with its connection dictionary. Briefly forms a connection to the server to validate the connection dictionary and then closes the connection. Raises an EOGeneralAdaptorException if the connection dictionary contains invalid information.

An adaptor subclass must override this method without invoking EOAdaptor's implementation.

See also: – **setConnectionDictionary:**, – **runLoginPanel**, – **runLoginPanelAndValidateConnectionDictionary**

canServiceModel:

– (BOOL)**canServiceModel:**(EOModel *)*model*

Returns YES if the receiver can service *model*, NO otherwise. EOAdaptor's implementation returns YES if the receiver's connection dictionary is equal to *model*'s connection dictionary as determined by NSDictionary's **isEqual:** method.

A subclass of EOAdaptor doesn't need to override this method.

connectionDictionary

– (NSDictionary *)**connectionDictionary**

Returns the receiver's connection dictionary, or **nil** if the adaptor doesn't have one. The connection dictionary contains the values, such as user name and password, needed to connect to the database server. The dictionary's keys identify the information the server expects, and its values are the values that the adaptor will try when connecting. Each adaptor uses different keys; see your adaptor's documentation for keys it uses.

A subclass of EOAdaptor doesn't need to override this method.

See also: **setConnectionDictionary:**

contexts

– (NSArray *)**contexts**

Returns the adaptor contexts created by the receiver, or **nil** if no adaptor contexts have been created. A subclass of EOAdaptor doesn't need to override this method.

See also: – **createAdaptorContext**

createAdaptorContext

– (EOAdaptorContext *)**createAdaptorContext**

Implemented by subclasses to create and return a new EOAdaptorContext, or **nil** if a new context can't be created.The new context retains the receiver. A newly created EOAdaptor has no contexts.

An adaptor subclass must override this method without invoking EOAdaptor's implementation.

See also: – **contexts**, – **initWithAdaptor:** (EOAdaptorContext)

createDatabaseWithAdministrativeConnectionDictionary:

– (void)**createDatabaseWithAdministrativeConnectionDictionary:**
(NSDictionary *)*connectionDictionary*

Uses the administrative login information to create the database (or user for Oracle) defined by the *connectionDictionary.*

See also: – **dropDatabaseWithAdministrativeConnectionDictionary:**, EOLoginPanel class

databaseEncoding

– (NSStringEncoding)**databaseEncoding**

Returns the string encoding used to encode and decode database strings. An adaptor's database encoding is stored in the connection dictionary with the key "databaseEncoding". If the connection dictionary doesn't have an entry for the database encoding, the default C string encoding is used. This method raises an NSInvalidArgumentException if the receiver's database encoding isn't valid.

A database system stores strings in a particular character set. The Framework needs to know what character set the database system uses so it can encode and decode strings coming from and going to the database server. The string encoding returned from this method specifies the character set the Framework uses.

A subclass of EOAdaptor doesn't need to override this method.

See also: – **availableStringEncodings** (NSString), – **defaultCStringEncoding** (NSString)

defaultExpressionClass

– (Class)**defaultExpressionClass**

Implemented by subclasses to return the subclass of EOSQLExpression used as the default expression class for the adaptor. You wouldn't ordinarily invoke this method directly. It's invoked automatically to determine which class should be used to represent query language expressions.

An adaptor subclass must override this method without invoking EOAdaptor's implementation.

See also: + **setExpressionClassName:adaptorClassName:**

delegate

– (id)**delegate**

Returns the receiver's delegate or **nil** if a delegate has not been assigned. A subclass of EOAdaptor doesn't need to override this method.

See also: – **setDelegate:**

dropDatabaseWithAdministrativeConnectionDictionary:

– (void)**dropDatabaseWithAdministrativeConnectionDictionary:**
(NSDictionary *)*connectionDictionary*

Uses the administrative login information to drop the database (or user for Oracle) defined by the *connectionDictionary.*

See also: – **createDatabaseWithAdministrativeConnectionDictionary:**, EOLoginPanel class

expressionClass

– (Class)**expressionClass**

Returns the subclass of EOSQLExpression used by the receiver for query language expressions. Returns the expression class assigned using the class method + **setExpressionClassName:adaptorClassName:**. If no class has been set for the receiver's class, this method determines the expression class by sending **defaultExpressionClass** to **self**.

You wouldn't ordinarily invoke this method directly. It's invoked automatically to determine which class should be used to represent query language expressions.

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

fetchedValueForDataValue:attribute:

– (NSData *)**fetchedValueForDataValue:**(NSData *)*value* **attribute:**(EOAttribute *)*attribute*

Overridden by subclasses to return the value that the receiver's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute*. This method is invoked from **fetchedValueForValue:attribute:** when the value argument is an NSData.

EOAdaptor's implementation returns *value* unchanged. An adaptor subclass should override this method if the adaptor's database performs transformations on binary types, such as BLOBs.

fetchedValueForDateValue:attribute:

– (NSCalendarDate *)**fetchedValueForDateValue:**(NSCalendarDate *)*value*
attribute:(EOAttribute *)*attribute*

Overridden by subclasses to return the value that the receiver's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute*. This method is invoked from **fetchedValueForValue:attribute:** when the value argument is a date.

EOAdaptor's implementation returns *value* unchanged. An adaptor subclass should override this method to convert or format date values. For example, a concrete adaptor subclass could set *value*'s millisecond value to 0.

fetchedValueForNumberValue:attribute:

– (NSNumber *)**fetchedValueForNumberValue:**(NSNumber *)*value*
attribute:(EOAttribute *)*attribute*

Overridden by subclasses to return the value that the receiver's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute*. This method is invoked from **fetchedValueForValue:attribute:** when the value argument is a number.

EOAdaptor's implementation returns *value* unchanged. An adaptor subclass should override this method to convert or format numeric values. For example, a concrete adaptor subclass should probably round *value* according to the precision and scale *attribute*.

fetchedValueForStringValue:attribute:

– (NSString*)**fetchedValueForStringValue:**(NSString *)*value* **attribute:**(EOAttribute *)*attribute*

Overridden by subclasses to return the value that the receiver's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute*. This method is invoked from **fetchedValueForValue:attribute:** when the value argument is a string.

EOAdaptor's implementation trims trailing spaces and returns **nullnil** for zero-length strings. An adaptor subclass should override this method to perform any additional conversion or formatting on string values. For example, a concrete adaptor subclass could trim trailing spaces.

fetchedValueForValue:attribute:

– (id)**fetchedValueForValue:**(id)*value* **attribute:**(EOAttribute *)*attribute*

Returns the value that the receiver's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute*. The Framework uses this method to keep enterprise object snapshots in sync with database values. For example, assume that a product's price is marked down 15%. If the product's original price is 5.25, the sale price is 5.25*.85, or 4.4625. When the Framework updates the product's price, the database server truncates the price to 4.46 (assuming the scale of the database's price column is 2). Before performing the update, the Framework sends the adaptor a **fetchedValueForValue: attribute:** message with the value 4.4625. The adaptor performs the database-specific transformation and returns 4.46. The Framework assigns the truncated value to the product object and to the product object's snapshot and then proceeds with the update.

An adaptor subclass can override this method or one of the data type-specific **fetchedValue...** methods. EOAdaptor's implementation of **fetchedValueForValue:attribute:** invokes one of the data type-specific methods depending on *value*'s class. If *value* is not a string, number, date, or data object (that is, an instance of NSString, NSNumber, NSDate, NSData, or any of their subclasses), **fetchedValueForValue:attribute:** returns *value* unchanged.

This method invokes the delegate method **adaptor:fetchedValueForValue:attribute:** which can override the adaptor's default behavior.

See also: – **fetchedValueForDataValue:attribute:**, – **fetchedValueForDateValue:attribute:**, – **fetchedValueForNumberValue:attribute:**, – **fetchedValueForStringValue:attribute:**, **– valueFactoryMethod** (EOAttribute)

hasOpenChannels

– (BOOL)**hasOpenChannels**

Returns YES if any of the receiver's contexts have open channels, NO otherwise. A subclass of EOAdaptor doesn't need to override this method.

See also: **– hasOpenChannels** (EOAdaptorContext)

initWithName:

– (id)**initWithName:**(NSString *)*name*

The designated initializer for the EOAdaptor class, this method is overridden by adaptor subclasses to initialize a newly allocated EOAdaptor subclass with *name*. *name* is usually derived from the base filename (that is, the filename without the ".framework" extension) of the framework from which the adaptor is loaded. For example, an adaptor named "Acme" is loaded from the framework **AcmeEOAdaptor.framework**. Returns **self**.

Never invoke this method directly. It is invoked automatically from **adaptorWithName:** and **adaptorWithModel:**—EOAdaptor class methods you use to create a new adaptor.

A subclass of EOAdaptor doesn't need to override this method, but may override it to perform additional initialization. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

isValidQualifierTypeIn:model:

– (BOOL)**isValidQualifierType:**(NSString *)*typeName* **model:**(EOModel *)*model*

Implemented by subclasses to return YES if an attribute of type *typeName* can be used in a qualifier (a SQL WHERE clause) sent to the database server, or NO otherwise. *typeName* is the name of a type as required

by the database server, such as Sybase “varchar” or Oracle “NUMBER”. *model* is an optional argument that can be used to supplement the adaptor’s set of type mappings with additional mappings for user-defined database types. See your adaptor’s documentation for information on if and how it uses *model*.

An adaptor subclass must override this method without invoking EOAdaptor’s implementation.

name

– (NSString *)**name**

Returns the adaptor’s name; this is usually the base filename of the framework from which the adaptor was loaded. For example, if an adaptor was loaded from a framework named **AcmeEOAdaptor.framework**, this method returns “Acme”.

A subclass of EOAdaptor doesn’t need to override this method.

See also: + **adaptorWithName:**, – **initWithName:**

prototypeAttributes

– (NSArray *)**prototypeAttributes**

Returns an array of prototype attributes specific to the adaptor class. Adaptor implementers should note that this method looks for an EOModel named EO*adaptorName*Prototypes in the resources directory of the adaptor.

runLoginPanel

– (NSDictionary *)**runLoginPanel**

Runs the adaptor’s login panel by sending a **runPanelForAdaptor:validate:** message to the adaptor’s login panel object with the validate flag NO. Returns connection information entered in the panel without affecting the adaptor’s connection dictionary. The connection dictionary returned isn’t validated by this method.

A subclass of EOAdaptor doesn’t need to override this method. A subclass that does override this method must incorporate the superclass’s version through a message to **super**.

See also: – **runLoginPanelAndValidateConnectionDictionary**, – **setConnectionDictionary:**, – **assertConnectionDictionaryIsValid**, + **sharedLoginPanelInstance**

runLoginPanelAndValidateConnectionDictionary

– (BOOL)**runLoginPanelAndValidateConnectionDictionary**

Runs the adaptor's login panel by sending a **runPanelForAdaptor:validate:** message to the adaptor's login panel object with the validate flag YES. Returns YES if the user enters valid connection information, or NO if the user cancels the panel.

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: – **runLoginPanel**, – **setConnectionDictionary:**, – **assertConnectionDictionaryIsValid**, + **sharedLoginPanelInstance**

setConnectionDictionary:

– (void)**setConnectionDictionary:**(NSDictionary *)*dictionary*

Sets the adaptor's connection dictionary to *dictionary*, which must only contain NSString, NSData,NSDictionary, and NSArray objects. Raises an NSInvalidArgumentException if there are any open channels—you can't change connection information while the adaptor is connected.

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: – **connectionDictionary**, – **hasOpenChannels**, – **assertConnectionDictionaryIsValid**, – **runLoginPanelAndValidateConnectionDictionary**, **– runPanelForAdaptor:validate:** (EOLoginPanel)

setDelegate:

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

Sets the receiver's delegate to *delegate*, or removes its delegate if *delegate* is **nil**. The receiver does not retain *delegate*. A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: – **delegate**

sharedLoginPanelInstance

+ (EOLoginPanel *)**sharedLoginPanelInstance**

Returns the receiver's login panel in applications that have a graphical user interface. Returns **nil** if the application doesn't have an NSApplication object. Otherwise, looks for the bundle named "LoginPanel" in the resources for the adaptor framework, loads the bundle, and returns an instance of the bundle's principal

class (see the NSBundle class specification for information on loading bundles). The returned object is used to implement **runLoginPanelAndValidateConnectionDictionary** and **runLoginPanel**.

A subclass of EOAdaptor doesn't need to override this method. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

Creating an EOAdaptor Subclass

Enterprise Objects Framework provides concrete adaptors for three standard relational database management systems—Informix, Oracle, and Sybase—as well as a concrete adaptor for ODBC-compliant databases. You may want to create a subclass of one of these adaptors to extend its behavior, or you may want to create a concrete EOAdaptor subclass for a different database or persistent storage system. EOAdaptor provides many default method implementations that are sufficient for concrete subclasses:

- + assignExternalInfoForEntireModel:
- – connectionDictionary
- – contexts
- – databaseEncoding
- – delegate
- – hasOpenChannels
- – name

The following methods establish structure and conventions that other Enterprise Objects Framework classes depend on and should be overridden with caution:

- + adaptorWithModel:
- + adaptorWithName:
- + setExpressionClassName:adaptorClassName:
- + sharedLoginPanelInstance
- – initWithName:
- – expressionClass
- – runLoginPanel
- – runLoginPanelAndValidateConnectionDictionary
- – setConnectionDictionary:
- – setDelegate:

If you override any of the above methods, your implementations should incorporate the superclass's implementation through a message to **super**.

The remaining EOAdaptor methods must be overridden by concrete adaptor subclasses in terms of the persistent storage system with which it interacts:

- + assignExternalInfoForAttribute:
- + assignExternalInfoForEntity:
- + externalTypesWithModel:
- + internalTypeForExternalType:model:
- – assertConnectionDictionaryIsValid
- – createAdaptorContext
- – defaultExpressionClass
- – fetchedValueForDataValue:attribute:
- – fetchedValueForDateValue:attribute:
- – fetchedValueForNumberValue:attribute:

- – fetchedValueForStringValue:attribute:
- – fetchedValueForValue:attribute:
- – isValidQualifierTypeIn:model:

EOAdaptorChannel

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOAdaptorChannel.h

Class Description

EOAdaptorChannel is an abstract class that provides its concrete subclasses with a structure for performing database operations. It's associated with EOAdaptor and EOAdaptorContext, which, together with EOAdaptorChannel, form the *adaptor level* of Enterprise Objects Framework's access layer. See the EOAdaptor class specification for more information about accessing, creating, and using adaptor level objects.

A concrete subclass of EOAdaptorChannel provides database-specific method implementations and represents an independent communication channel to the database server to which its EOAdaptor object is connected. You never interact with instances of the EOAdaptorChannel class, rather your Enterprise Objects Framework applications use instances of concrete subclasses that are written to interact with a specific database or other persistent storage system. To create an instance of a concrete EOAdaptorChannel subclass, you send a **createAdaptorChannel** message to an instance of the corresponding EOAdaptorContext subclass. You rarely create adaptor channels yourself. They are generally created automatically by other framework objects.

You use an adaptor channel to manipulate rows (records) by selecting, fetching, inserting, deleting, and updating them. An adaptor channel also gives you access to some of the metadata on the server, such as what stored procedures exist, what tables exist, and what their basic attributes and relationships are.

All of an adaptor channel's operations take place within the context of transactions controlled or tracked by its EOAdaptorContext. An adaptor context may manage several channels (though not all can), but a channel is associated with only one context.

Notifying the Adaptor Channel's Delegate

You can assign a delegate to an adaptor channel. The EOAdaptorChannel sends certain messages directly to the delegate, and the delegate responds to these messages on the channel's behalf. Many of the adaptor channel methods notify the channel's delegate before and after an operation is performed. Some delegate methods, such as **adaptorChannel:shouldEvaluateExpression:**, let the delegate determine whether the channel should perform an operation. Others, such as **adaptorChannel:didEvaluateExpression:**, are simply notifications that an operation has occurred. The delegate has an opportunity to respond by implementing the delegate methods. If the delegate wants to intervene, it implements **adaptorChannel:**

shouldEvaluateExpression:. If it simply wants notification when a transaction has begun, it implements **adaptorChannel:didEvaluateExpression:**.

The principal attributes of the EOAdaptorChannel class are:

- Adaptor context
- Delegate

Other framework classes create EOAdaptorChannel objects, using EOAdaptorContext's **createAdaptorChannel** method, which both creates an adaptor channel and assigns its context.

The following table lists EOAdaptorChannel's more commonly-used methods:

openChannel	Opens the channel so it can perform database operations.
closeChannel	Close the channel.
selectAttributes:fetchSpecification:lock:entity:	Selects rows matching the specified qualifier.
fetchRowWithZone:	Fetches a row resulting from the last **selectAttributes: fetchSpecification:lock:entity:**, **executeStoredProcedure:withValues:**, or **evaluateExpression:**.
insertRow:forEntity:	Inserts the specified row.
updateValues:inRowsDescribedByQualifier:entity:	Updates the row described by the specified qualifier.
deleteRowDescribedByQualifier:entity:	Deletes the row described by the specified qualifier.
executeStoredProcedure:withValues:	Performs the specified stored procedure.
evaluateExpression:	Sends the specified expression to the database.

– openChannel	Opens the channel so it can perform database operations.
– closeChannel	Close the channel.
– selectAttributes:fetchSpecification:lock:entity:	Selects rows matching the specified qualifier.
– fetchRowWithZone:	Fetches a row resulting from the last **select...**, **executeStoredProcedure...**, or **evaluateExpression:**.
– insertRow:forEntity:	Inserts the specified row.
– updateValues:inRowDescribedByQualifier:entity:	Updates the row described by the specified qualifier.

– deleteRowDescribedByQualifier:entity:	Deletes the row described by the specified qualifier.
– executeStoredProcedure:withValues:	Performs the specified stored procedure.
– evaluateExpression:	Sends the specified expression to the database.
– performAdaptorOperation:	Performs an adaptor operation by invoking the EOAdaptorChannel method appropriate for performing the specified operation.

For more information on subclassing EOAdaptorChannel, see “Creating an EOAdaptorChannel Subclass”.

Method Types

Accessing the adaptor context

– adaptorContext

Opening and closing a channel

– openChannel
– closeChannel
– isOpen

Creating an EOAdaptorChannel

– initWithAdaptorContext:

Modifying rows

– insertRow:forEntity:
– updateValues:inRowDescribedByQualifier:entity:
– updateValues:inRowsDescribedByQualifier:entity:
– deleteRowDescribedByQualifier:entity:
– deleteRowsDescribedByQualifier:entity:

Fetching rows

– selectAttributes:fetchSpecification:lock:entity:
– describeResults
– setAttributesToFetch:
– attributesToFetch
– fetchRowWithZone:
– dictionaryWithObjects:forAttributes:zone:
– cancelFetch
– isFetchInProgress

Invoking stored procedures

– executeStoredProcedure:withValues:
– returnValuesForLastStoredProcedureInvocation

Assigning primary keys
– primaryKeyForNewRowWithEntity:

Sending SQL to the server
– evaluateExpression:

Batch processing operations
– performAdaptorOperation:
– performAdaptorOperations:

Accessing schema information
– describeTableNames
– describeStoredProcedureNames
– addStoredProceduresNamed:toModel:
– describeModelWithTableNames:

Debugging
– setDebugEnabled:
– isDebugEnabled

Accessing the delegate
– delegate
– setDelegate:

Instance Methods

adaptorContext

– (EOAdaptorContext *)**adaptorContext**

Returns the receiver's EOAdaptorContext. A subclass of EOAdaptorChannel doesn't need to override this method.

See also: – **initWithAdaptorContext:**

addStoredProceduresNamed:toModel:

– (void)**addStoredProceduresNamed:**(NSArray *)*storedProcedureNames*
toModel:(EOModel *)*model*

Overridden by subclasses to create EOStoredProcedure objects for the stored procedures named in *storedProcedureNames* and then to add them to *model*. This method is used in conjunction with **describeStoredProcedureNames** to build a default model in EOModeler. Raises an exception if an error occurs.

attributesToFetch

– (NSArray *)**attributesToFetch**

Implemented by subclasses to return the set of attributes to retrieve when **fetchRowWithZone:** is next invoked. An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: **setAttributesToFetch:**

cancelFetch

– (void)**cancelFetch**

Implemented by subclasses to clear all result sets established by the last **selectAttributes: fetchSpecification:lock:entity:**, **executeStoredProcedure:withValues:**, or **evaluateExpression:** message and terminate the current fetch, so that **isFetchInProgress** returns NO.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

closeChannel

– (void)**closeChannel**

Implemented by subclasses to close the EOAdaptorChannel so that it can't perform operations with the server. Any fetch in progress is canceled. If the receiver is the last open channel in an adaptor context and if the channel's adaptor context has outstanding transactions, closing the channel has server-dependent results: some database servers roll back all outstanding transactions but others do nothing. Regardless of whether outstanding transactions are rolled back, this method has the side effect of closing the receiver's adaptor context's connection with the database if the receiver is its adaptor context's last open channel.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **cancelFetch**, – **transactionNestingLevel** (EOAdaptorContext)

delegate

– (id)**delegate**

Returns the receiver's delegate, or **nil** if the receiver doesn't have a delegate. A subclass of EOAdaptorChannel doesn't need to override this method.

See also: **setDelegate:**

deleteRowDescribedByQualifier:entity:

– (void)**deleteRowDescribedByQualifier:**(EOQualifier *)*qualifier* **entity:**(EOEntity *)*entity*

Deletes the row described by *qualifier* from the database table corresponding to *entity*. Invokes **deleteRowsDescribedByQualifier:entity:** and raises an exception unless exactly one row is deleted. A subclass of EOAdaptorChannel doesn't need to override this method.

deleteRowsDescribedByQualifier:entity:

– (unsigned int)**deleteRowsDescribedByQualifier:**(EOQualifier *)*qualifier* **entity:**(EOEntity *)*entity*

Implemented by subclasses to delete the rows described by *qualifier* from the database table corresponding to *entity*. Returns the number of rows deleted. Raises an exception on failure. Some possible reasons for failure are:

- The adaptor channel isn't open
- The adaptor channel is in an invalid state (for example, it's fetching).
- An error occurs in the database server

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **deleteRowDescribedByQualifier:entity:**, – **isOpen**, – **isFetchInProgress**, – **transactionNestingLevel** (EOAdaptorContext)

describeModelWithTableNames:

– (EOModel *)**describeModelWithTableNames:**(NSArray *)*tableNames*

Overridden by subclasses to create and return a default model containing entities for the tables specified in *tableNames*. Assigns the adaptor name and connection dictionary to the new model. This method is typically used in conjunction with **describeTableNames** and **describeStoredProcedureNames**.

EOAdaptorChannel's implementation does nothing. An adaptor channel subclass should override this method to create a default model from the database's metadata.

describeResults

– (NSArray *)**describeResults**

Implemented by subclasses to return an array of EOAttributes describing the properties available in the current result set, as determined by **selectAttributes:describedByQualifier:fetchOrder:lock:**, **executeStoredProcedure:withValues:**, or a statement evaluated by **evaluateExpression:**. Only invoke this method if a fetch is in progress as determined by **isFetchInProgress**.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

describeStoredProcedureNames

– (NSArray *)**describeStoredProcedureNames**

Overridden by subclasses to read and return an array of stored procedure names from the database. This method is used in conjunction with **addStoredProceduresNamed:toModel:** to build a default model in EOModeler. Raises an exception if an error occurs.

describeTableNames

– (NSArray *)**describeTableNames**

Overridden by subclasses to read and return an array of table names from the database. This method in conjunction with **describeModelWithTableNames:** is used to build a default model.

EOAdaptorChannel's implementation simply returns **nil**. An adaptor channel subclass should override this method to construct an array of table names from database metadata.

dictionaryWithObjects:forAttributes:zone:

– (NSMutableDictionary *)**dictionaryWithObjects:**(id *)*objects*
forAttributes:(NSArray *)*attributes*
zone:(NSZone *)*zone*

Used by EOAdaptorChannel subclasses to create dictionaries that can be returned from **fetchRowWithZone:**. You don't ordinarily invoke this method unless you are writing your own concrete adaptor. If you are writing a concrete adaptor, use of this method is optional but strongly recommended because it enables performance optimizations. The objects in *objects* are the values for the row that correspond to the EOAttribute objects in *attributes*. The dictionary representation of the row is created from *zone*.

A subclass of EOAdaptorChannel shouldn't override this method.

evaluateExpression:

– (void)**evaluateExpression:**(EOSQLExpression *)*expression*

Implemented by subclasses to send *expression* to the database server for evaluation, beginning a transaction first and committing it after evaluation if a transaction isn't already in progress. Raises an exception if an error occurs. An EOAdaptorChannel uses this method to send SQL expressions to the database.

If *expression* results in a select operation being performed, you can fetch the results as you would if you had sent a **selectAttributes:fetchSpecification:lock:entity:**. You must use the method **setAttributesToFetch:** before you begin fetching. Also, if *expression* evaluates to multiple result sets, you must invoke **setAttributesToFetch:** before you begin fetching each subsequent set.

evaluateExpression: invokes the delegate methods **adaptorChannel:shouldEvaluateExpression:** and **adaptorChannel:didEvaluateExpression:**.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation. Note, however, that the upper layers of the Framework never invoke **evaluateExpression:** directly. Thus, adaptors for data stores that don't naturally support an expression language (for example, flat file adaptors) don't need to implement this method to work with the Framework.

See also: – **fetchRowWithZone:**

executeStoredProcedure:withValues:

– (void)**executeStoredProcedure:**(EOStoredProcedure *)*storedProcedure*
withValues:(NSDictionary *)*values*

Implemented by subclasses to execute *storedProcedure*. Any arguments to the stored procedure are in *values*, a dictionary whose keys are the argument names. Use **fetchRowWithZone:** to get result rows and **returnValuesForLastStoredProcedureInvocation** to get return arguments and result status, if any. Raises an exception if an error occurs.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation. Note, however, that the upper layers of the Framework never invoke **executeStoredProcedure:withValues:** directly. Thus, adaptors for data stores that don't support stored procedures (for example, flat file adaptors) don't need to implement this method to work with the Framework

fetchRowWithZone:

– (NSMutableDictionary *)**fetchRowWithZone:**(NSZone *)*zone*

Implemented by subclasses to fetch the next row from the result set of the last **selectAttributes:fetchSpecification:lock:entity:**, **executeStoredProcedure:withValues:**, or **evaluateExpression:** message sent to the receiver. Returns values for the receiver's **attributesToFetch** in a dictionary whose keys are the attribute names. When there are no more rows in the current result set, this method returns **nil**, and invokes the delegate method **adaptorChannelDidChangeResultSet:** if there are more results sets. When there are no more rows or result sets, this method returns **nil**, ends the fetch, and invokes **adaptorChannelDidFinishFetching:**. **isFetchInProgress** returns YES until the fetch is canceled or until this method exhausts all result sets and returns **nil**. This method also invoke the delegate methods **adaptorChannelWillFetchRow:** and **adaptorChannel:didFetchRow:**. Raises an exception if an error occurs.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **setAttributesToFetch:**

initWithAdaptorContext:

– **initWithAdaptorContext:**(EOAdaptorContext *)*adaptorContext*

The designated initializer for the EOAdaptorChannel class, this method is overridden by subclasses to initialize a newly allocated EOAdaptorChannel subclass and retain *adaptorContext*. Returns **self**.

You never invoke this method directly unless you are implementing a concrete adaptor context. It is invoked automatically from **createAdaptorChannel—**the EOAdaptorContext method you use to create a new adaptor channel.

A subclass of EOAdaptorChannel doesn't need to override this method, but may override it to perform additional initialization. A subclass that does override this method must incorporate the superclass's version through a message to **super**.

See also: – **adaptorContext**

insertRow:forEntity:

– (void)**insertRow:**(NSDictionary *)*row* **forEntity:**(EOEntity *)*entity*

Implemented by subclasses to insert the values of *row* into the table in the database that corresponds to *entity. row* is a dictionary whose keys are attribute names and whose values are the values to insert. Raises an exception on failure. Some possible reasons for failure are:

- The user logged in to the database doesn't have permission to insert a new row.
- The adaptor channel is in an invalid state (for example, fetching).
- The row fails to satisfy a constraint defined in the database server.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

isDebugEnabled

– (BOOL)**isDebugEnabled**

Returns YES if the adaptor channel logs evaluated SQL and other useful information to the console (or to the standard error stream), NO if not. A subclass of EOAdaptorChannel doesn't need to override this method.

See also: – **setDebugEnabled:**, – **setDebugEnabled:** (EOAdaptorContext)

isFetchInProgress

– (BOOL)**isFetchInProgress**

Implemented by subclasses to return YES if the receiver is fetching, NO otherwise. An adaptor channel is fetching if:

- It's been sent a successful **selectAttributes:describedByQualifier:fetchOrder:lock:** message.
- A stored procedure that returns rows has been successfully executed using **executeStoredProcedure: withValues:**.
- An expression sent through **evaluateExpression:** resulted in a select operation being performed.

An adaptor channel stops fetching when there are no more records to fetch or when it's sent a **cancelFetch** message.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **fetchRowWithZone:**

isOpen

– (BOOL)**isOpen**

Implemented by subclasses to return YES if the channel has been opened with **openChannel**, NO if not. An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **closeChannel**

lockRowComparingAttributes:entity:qualifier:snapshot:

– (void)**lockRowComparingAttributes:**(NSArray *)*attributes*
entity:(EOEntity *)*entity*
qualifier:(EOQualifier *)*qualifier*
snapshot:(NSDictionary *)*snapshot*

Attempts to lock a row in the database by selecting it with locking on. The lock operation succeeds if a select statement generated with *qualifier* retrieves exactly one row and the values in the row match the values in *snapshot*, a dictionary whose keys are attribute names and whose values are the values that were last fetched from the database.

lockRowComparingAttributes:entity:qualifier:snapshot: invokes **selectAttributes:fetchSpecification: lock:entity:** with *attributes* as the attributes to select, a fetch specification built from *qualifier*, locking on, and *entity* as the entity. If the select returns no rows or more than one row, the method raises an EOGeneralAdaptorException. It also raises an EOGeneralAdaptorException if the values in the returned row don't match the corresponding values in *snapshot*.

The Framework uses this method whenever it needs to lock a row. When the Framework invokes it, *qualifier* specifies the primary key of the row to be locked and attributes used for locking to be compared in the database server. If any of the values specified in *qualifier* are different from the values in the database row, the select operation will not retrieve or lock the row. When this happens, the row to be locked has been updated in the database since it was last retrieved, and it isn't safe to update it.

Some attributes (such as BLOB types) can't be compared in the database. *attributes* should specify any such attributes. (If the row doesn't contain any such attributes, *attributes* can be **nil**.) If *qualifier* generates a select statement that returns and locks a single row, this method performs an in-memory comparison between the value in the retrieved row and the value in *snapshot* for each attribute in *attributes*. Therefore, *snapshot* must contain an entry for each attribute in *attributes*. In addition, it must contain an entry for the row's primary key.

A subclass of EOAdaptorChannel doesnt need to override this method.

openChannel

– (void)**openChannel**

Implemented by subclasses to put the channel and both its context and adaptor into a state where they are ready to perform database operations. Raises an exception if an error occurs. An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **isOpen**, – **closeChannel**

performAdaptorOperation:

– (void)**performAdaptorOperation:**(EOAdaptorOperation *)*adaptorOperation*

Performs *adaptorOperation* by invoking the adaptor channel method appropriate for performing the specified operation. For example, if the adaptor operator for *adaptorOperation* is EOAdaptorInsertOperator, this method invokes **insertRow:forEntity:** using information in *adaptorOperation* to supply the arguments. Raises an exception if an error occurs.

A subclass of EOAdaptorChannel doesn't need to override this method.

See also: – **performAdaptorOperations:**

performAdaptorOperations:

– (void)**performAdaptorOperations:**(NSArray *)*adaptorOperations*

Performs adaptor operations by invoking **performAdaptorOperation:** with each EOAdaptorOperation object in the array *adaptorOperations*. An adaptor channel subclass may be able to override this method to

take advantage of database-specific batch processing capabilities. Invokes the delegate methods **adaptorChannel:willPerformOperations:** and **adaptorChannel:didPerformOperations:exception:**.

This method raises an exception if an error occurs. The exception's userInfo dictionary contains these keys:

- EOAdaptorOperationsKey

  Corresponds to the array of adaptor operations that's being executed.

- EOFailedAdaptorOperationKey

  Corresponds to the particular adaptor operation that failed.

- EOAdaptorFailureKey

  If present, offers additional information on the type of error that occurred. Currently, the only possible value for this key is EOAdaptorOptimisticLockingFailure, which indicates that an update or lock operation failed because the row found in the database did not match the snapshot taken when the row was last fetched into the application.

A subclass of EOAdaptorChannel doesn't need to override the **performAdaptorOperations:** method.

primaryKeyForNewRowWithEntity:

– (NSDictionary *)**primaryKeyForNewRowWithEntity:**(EOEntity *)*entity*

Overridden by subclasses to return a primary key for a new row in the database table that corresponds to *entity*. The primary key returned from this method is a dictionary whose keys are the primary key attribute names. For example, suppose you've got a table MOVIE with primary key MOVIE_ID, and the corresponding Movie Entity's primary key attribute is **movieID**. In this scenario, the dictionary returned from **primaryKeyForNewRowWithEntity:** has one entry whose key is **movieID** and whose value is the unique value to assign. If the primary key is compound (made up of more than one attribute), the dictionary should contain an entry for each primary key attribute. Note, however, that the Enterprise Objects Frameworks adaptors don't handle compound primary keys; they return **nil** from **primaryKeyForNewRowWithEntity:** if the primary key is compound.

If information in *entity* specifies an adaptor-specific means to assign a new primary key (for example, a sequence name or stored procedure), then this method returns a new primary key. Otherwise, if the key is a simple integer, the method tries to fetch a new primary key from the database using an adaptor-specific scheme. Otherwise, the method returns **nil**.

EOAdaptorChannel's implementation simply returns **nil**. See your adaptor channel's documentation for information on how it generates primary keys.

A subclass of EOAdaptorChannel must override this method. For example, to return a value generated by a sequence, you'd create the proper SQL statement (using EOSQLExpression's **expressionForString:** method) and evaluate it (using the **evaluateExpression:** method).

returnValuesForLastStoredProcedureInvocation

– (NSDictionary *)**returnValuesForLastStoredProcedureInvocation**

Implemented by subclasses to return stored procedure parameter and return values. Used in conjunction with **executeStoredProcedure:withValues:**. The dictionary returned by this method has entries whose keys are stored procedure parameter names and whose values are the parameter values. The dictionary also contains a special entry for the stored procedures return value with the key “returnValue”. Returns an empty dictionary for stored procedures that have void return types. Returns **nil** if the stored procedure has results to fetch. In this case, you must use **fetchRowWithZone:** until there are no more results to fetch before the return value will be available.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel’s implementation.

selectAttributes:fetchSpecification:lock:entity:

– (void)**selectAttributes:**(NSArray *)*attributes*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification*
lock:(BOOL)*flag*
entity:(EOEntity *)*entity*

Implemented by subclasses to select *attributes* in rows matching the qualifier in *fetchSpecification* and set the receiver’s attributes to fetch. The selected rows compose one or more result sets, each row of which will be returned by subsequent **fetchRowWithZone:** messages according to *fetchSpecification*’s sort orderings. If *flag* is YES, the rows are locked if possible so that no other user can modify them (the lock specification in *fetchSpecification* is ignored). Raises an exception if an error occurs. Some possible reasons for failure are:

- The adaptor channel is in an invalid state (for example, fetching).
- The database failed to lock the specified rows.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel’s implementation.

See also: – **setAttributesToFetch:**

setAttributesToFetch:

– (void)**setAttributesToFetch:**(NSArray *)*attributes*

Implemented by subclasses to specify the set of attributes used to describe fetch data from a corresponding select. *attributes* is an array of the attributes to fetch. This method is invoked after **evaluateExpression:** but before the first call to **fetchRowWithZone:**. For more information on using this method, see “Sending SQL Statements Directly to the Server” in the “WebObjects Programming Topics.” Is that a good cross-reference? This method raises if invoked when there is no fetch in progress.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **attributesToFetch**, – **selectAttributes:fetchSpecification:lock:entity:**

setDebugEnabled:

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

Enables debugging in the receiver and all its channels. If *flag* is YES, enables debugging; otherwise, disables debugging. When debugging is enabled, the adaptor channel logs evaluated SQL and other useful debugging information to the console (or to the standard error stream). The information provided may vary from adaptor to adaptor and may change from release to release.

A subclass of EOAdaptorChannel doesn't need to override this method. A subclass that does override it must incorporate the superclass's version through a message to **super**.

See also: – **isDebugEnabled**, – **setDebugEnabled:** (EOAdaptorContext)

setDelegate:

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

Sets the receiver's delegate to *delegate*, or removes its delegate if *delegate* is **nil**. The receiver does not retain its delegate. A subclass of EOAdaptorChannel doesn't need to override this method. A subclass that does override it must incorporate the superclass's version through a message to **super**.

See also: – **delegate**

updateValues:inRowDescribedByQualifier:entity:

– (void)**updateValues:**(NSDictionary *)*values*
inRowDescribedByQualifier:(EOQualifier *)*qualifier*
entity:(EOEntity *)*entity*

Updates the row described by *qualifier*. Invokes **updateValues:inRowsDescribedByQualifier:entity:** and raises an exception unless exactly one row is updated.

A subclass of EOAdaptorChannel doesn't need to override this method.

updateValues:inRowsDescribedByQualifier:entity:

– (unsigned int)**updateValues:**(NSDictionary *)*values*
inRowsDescribedByQualifier:(EOQualifier *)*qualifier*
entity:(EOEntity *)*entity*

Implemented by subclasses to update the rows described by *qualifier* with the values in *values*. *values* is a dictionary whose keys are attribute names and whose values are the new values for those attributes (the dictionary need only contain entries for the attributes being changed). Returns the number of updated rows. Raises an exception if an error occurs. Some possible reasons for failure are:

- The user logged in to the database doesn't have permission to update.
- The adaptor channel is in an invalid state (for example, fetching).
- The new values fail to satisfy a constraint defined in the database server.

An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See also: – **updateValues:inRowDescribedByQualifier:entity:**

Creating an EOAdaptorChannel Subclass

EOAdaptorChannel provides many default method implementations that are sufficient for concrete subclasses:

- – adaptorContext
- – delegate
- – deleteRowDescribedByQualifier:entity:
- – isDebugEnabled
- – lockRowComparingAttributes:entity:qualifier:snapshot:
- – performAdaptorOperation:
- – performAdaptorOperations:
- – updateValues:inRowDescribedByQualifier:entity:

The following methods establish structure and conventions that other Enterprise Objects Framework classes depend on and should be overridden with caution:

- – dictionaryWithObjects:forAttributes:zone:
- – initWithAdaptorContext:
- – setDebugEnabled:
- – setDelegate:

If you override any of the above methods, your implementations should incorporate the superclass's implementation through a message to **super**.

The remaining EOAdaptorChannel methods must be overridden by concrete subclasses in terms of the persistent storage system with which it interacts:

- – attributesToFetch
- – cancelFetch
- – closeChannel
- – deleteRowsDescribedByQualifier:entity:
- – describeModelWithTableNames:
- – describeResults
- – describeStoredProcedureNames
- – describeTableNames
- – evaluateExpression:
- – executeStoredProcedure:withValues:
- – fetchRowWithZone:
- – insertRow:forEntity:
- – isFetchInProgress
- – isOpen
- – openChannel
- – primaryKeyForNewRowWithEntity:
- – returnValuesForLastStoredProcedureInvocation
- – selectAttributes:fetchSpecification:lock:entity:

- – setAttributesToFetch:
- – updateValues:inRowsDescribedByQualifier:entity:

EOAdaptorContext

Inherits From: NSObject

Declared In: EOAccess/EOAdaptorContext.h

Class Description

EOAdaptorContext is an abstract class that defines transaction handling in Enterprise Objects Framework applications. You typically don't interact with EOAdaptorContext API directly; rather, a concrete adaptor context subclass inherits from EOAdaptorContext and overrides many of its methods, which are invoked automatically by the Enterprise Objects Framework. If you're not creating a concrete adaptor context subclass, there aren't very many methods you need to use, and you'll rarely invoke them directly.

The EOAdaptorContext class has the following principal attributes:

- Array of adaptor channels
- Delegate
- Adaptor

Other framework classes create EOAdaptorContext objects automatically. This is typically done with EOAdaptor's – createAdaptorContext method, which creates an adaptor context and assigns its adaptor.

The following table lists the most commonly-used EOAdaptorContext methods:

– beginTransaction	Begins a transaction in the database server.
– commitTransaction	Commits the last transaction begun.
– rollbackTransaction	Rolls back the last transaction begun.
– setDebugEnabled:	Enables debugging in all the adaptor context's channels.

For more information, see "EOAdaptorContext".

Method Types

Creating an EOAdaptorContext

– initWithAdaptor:

Accessing the adaptor

– adaptor

Creating adaptor channels

– createAdaptorChannel
– channels

Checking connection status

– hasOpenChannels
– hasBusyChannels

Controlling transactions

– beginTransaction
– commitTransaction
– rollbackTransaction
– transactionDidBegin
– transactionDidCommit
– transactionDidRollback
– canNestTransactions
– transactionNestingLevel

Debugging

+ setDebugEnabledDefault:
+ debugEnabledDefault
– setDebugEnabled:
– isDebugEnabled

Accessing the delegate

– delegate
– setDelegate:

Class Methods

debugEnabledDefault

+ (BOOL)**debugEnabledDefault**

Returns YES if new adaptor context instances have debugging enabled by default, NO otherwise. By default, adaptor contexts have debugging enabled if the user default EOAdaptorDebugEnabled is YES. (For more information on user defaults, see the NSUserDefaults class specification in the *Foundation Framework Reference*.) You can override the user default using the class method **setDebugEnabledDefault:** , or you can set debugging behavior for a specific instance with the instance method **setDebugEnabled:**.

setDebugEnabledDefault:

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

Sets default debugging behavior for new instances of EOAdaptorContext. If *flag* is YES, debugging is enabled for new instances. If *flag* is NO, debugging is disabled. Use the instance method **setDebugEnabled:** to enable debugging for a specific adaptor context.

See also: + **debugEnabledDefault**, – **isDebugEnabled**

Instance Methods

adaptor

– (EOAdaptor *)**adaptor**

Returns the receiver's EOAdaptor.

See also: – **initWithAdaptor:**

beginTransaction

– (void)**beginTransaction**

Implemented by subclasses to attempt to begin a new transaction, nested within the current one if nested transactions are supported. Each successful invocation of **beginTransaction** must be paired with an invocation of either **commitTransaction** or **rollbackTransaction** to end the transaction.

The Enterprise Objects Framework automatically wraps database operations in transactions, so you don't have to begin and end transactions explicitly. In fact, letting the framework manage transactions is sometimes more efficient. You typically use **beginTransaction** only to execute more than one database operation in the same transaction scope.

This method invokes the delegate method **adaptorContextShouldBegin:** before beginning the transaction. If the transaction is begun successfully, sends **self** a **transactionDidBegin** message and invokes the delegate method **adaptorContextDidBegin:**. Raises if the attempt is unsuccessful. Some possible reasons for failure are:

- A connection to the database hasn't been established.
- Nested transactions aren't supported, and a transaction is already in progress.
- A fetch is in progress.
- The delegate refuses**.**
- The database server fails to begin a transaction.

An adaptor context subclass should override this method without invoking EOAdaptorContext's implementation.

See also: – **commitTransaction**, – **rollbackTransaction**, – **canNestTransactions**, – **transactionNestingLevel**

canNestTransactions

– (BOOL)**canNestTransactions**

Implemented by subclasses to return YES if the database server and the adaptor context can nest transactions, NO otherwise. An adaptor context subclass should override this method without invoking EOAdaptorContext's implementation.

See also: – **transactionNestingLevel**

channels

– (NSArray *)**channels**

Returns an array of channels created by this context.

See also: **createAdaptorChannel**

commitTransaction

– (void)**commitTransaction**

Implemented by subclasses to attempt to commit the last transaction begun. Invokes the delegate method **adaptorContextShouldCommit:** before committing the transaction. If the transaction is committed successfully, sends **self** a **transactionDidCommit** message and invokes the delegate method **adaptorContextDidCommit:**. Raises if the attempt is unsuccessful. Some possible reasons for failure are:

- A transaction is not in progress.
- Fetches are in progress.
- The delegate refuses.
- The database server fails to commit.

An adaptor context subclass should override this method without invoking EOAdaptorContext's implementation.

See also: – **beginTransaction**, – **createAdaptorChannel**, – **transactionDidCommit**, – **hasBusyChannels**

createAdaptorChannel

– (EOAdaptorChannel *)**createAdaptorChannel**

Implemented by subclasses to create and return a new AdaptorChannel, or **nil** if a new channel cannot be created. Initializes the new channel by sending it **initWithAdaptorContext:self**. The newly created channel retains its context. A newly created adaptor context has no channels. Specific adaptors have different limits on the maximum number of channels a context can have, and **createAdaptorChannel** fails if a newly created channel would exceed the limits.

See also: – **channels**

delegate

– **delegate**

Returns the receiver's delegate, or **nil** if the receiver doesn't have a delegate.

See also: – **setDelegate:**

hasBusyChannels

– (BOOL)**hasBusyChannels**

Returns YES if any of the receiver's channels have outstanding operations (that is, have a fetch in progress), NO otherwise.

See also: – **isFetchInProgress** (EOAdaptorChannel)

hasOpenChannels

– (BOOL)**hasOpenChannels**

Returns YES if any of the receiver's channels are open, NO otherwise.

See also: – **openChannel** (EOAdaptorChannel), – **isOpen** (EOAdaptorChannel)

initWithAdaptor:

– **initWithAdaptor:**(EOAdaptor *)*adaptor*

The designated initializer for the EOAdaptorContext class, this method is overridden by subclasses to initialize a newly allocated EOAdaptorContext subclass and retain *adaptor*. Returns **self**.

You never invoke this method directly. You must use the EOAdaptor method **createAdaptorContext** to create a new adaptor context.

See also: – **adaptor**

isDebugEnabled

– (BOOL)**isDebugEnabled**

Returns YES if debugging is enabled in the receiver, NO otherwise.

See also: – **setDebugEnabled:**, + **debugEnabledDefault**, + **setDebugEnabledDefault:**

rollbackTransaction

– (void)**rollbackTransaction**

Implemented by subclasses to attempt to roll back the last transaction begun. Invokes the delegate method **adaptorContextShouldRollback:** before rolling back the transaction. If the transaction is begun successfully, sends **self** a **transactionDidRollback** message and invokes the delegate method **adaptorContextDidRollback:**. Raises if the attempt is unsuccessful. Some possible reasons for failure are:

- A transaction is not in progress.
- Fetches are in progress.
- The delegate refuses.
- The database server fails to rollback.

An adaptor context subclass should override this method without invoking EOAdaptorContext's implementation.

See also: – **beginTransaction**, – **commitTransaction**

setDebugEnabled:

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

Enables debugging in the receiver and all its channels. If *flag* is YES, enables debugging; otherwise, disables debugging.

See also: – **setDebugEnabled:** (EOAdaptorChannel), – **isDebugEnabled**, + **setDebugEnabledDefault:**, – **channels**

setDelegate:

– (void)**setDelegate:***delegate*

Sets the receiver's delegate and the delegate of all the receiver's channels to *delegate*, or removes their delegates if *delegate* is **nil**. The receiver does not retain *delegate*.

See also: – **delegate**, – **channels**

transactionDidBegin

– (void)**transactionDidBegin**

Informs the adaptor context that a transaction has begun in the database server, so the receiver can update its state to reflect this fact and send an EOAdaptorContextBeginTransactionNotification. This method is invoked from **beginTransaction** after a transaction has successfully been started. It is also invoked when the Enterprise Objects Framework implicitly begins a transaction.

You don't need to invoke this method unless you are implementing a concrete adaptor. Your concrete adaptor should invoke this method from within your adaptor context's implementation of **beginTransaction** method and anywhere else it begins a transaction—either implicitly or explicitly. For example, an adaptor channel's implementation of **evaluateExpression:** should check to see if a transaction is in progress. If no transaction is in progress, it can start one explicitly by invoking **beginTransaction**. Alternatively, it can start an implicit transaction by invoking **transactionDidBegin**.

A subclass of EOAdaptorContext doesn't need to override this method. A subclass that does override it must incorporate the superclass's version through a message to **super**.

See also: – **transactionDidCommit**, – **transactionDidRollback**

transactionDidCommit

– (void)**transactionDidCommit**

Informs the adaptor context that a transaction has committed in the database server, so the receiver can update its state to reflect this fact and send an EOAdaptorContextCommitTransactionNotification. This method is invoked from **commitTransaction** after a transaction has successfully committed.

You don't need to invoke this method unless you are implementing a concrete adaptor. Your concrete adaptor should invoke this method from within your adaptor context's implementation of **commitTransaction** method and anywhere else it commits a transaction—either implicitly or explicitly.

A subclass of EOAdaptorContext doesn't need to override this method. A subclass that does override it must incorporate the superclass's version through a message to **super**.

See also: – **transactionDidBegin**, – **transactionDidRollback**

transactionDidRollback

– (void)**transactionDidRollback**

Informs the receiver that a transaction has rolled back in the database server, so the adaptor context can update its state to reflect this fact and send an EOAdaptorContextRollbackTransactionNotification. This method is invoked from **rollbackTransaction** after a transaction has successfully been rolled back.

You don't need to invoke this method unless you are implementing a concrete adaptor. Your concrete adaptor should invoke this method from within your adaptor context's implementation of **rollbackTransaction** method and anywhere else it rolls back a transaction—either implicitly or explicitly.

A subclass of EOAdaptorContext doesn't need to override this method. A subclass that does override it must incorporate the superclass's version through a message to **super**.

See also: – **transactionDidBegin**, – **transactionDidCommit**

transactionNestingLevel

– (unsigned)**transactionNestingLevel**

Returns the number of transactions in progress. If the database server and the adaptor support nested transactions, this number may be greater than 1.

See also: – **canNestTransactions**

Notifications

EOAdaptorContextBeginTransactionNotification

Sent from **transactionDidBegin** to tell observers that a transaction has begun. The notification contains:

Notification Object	The notifying EOAdaptorContext object
Userinfo	None

EOAdaptorContextCommitTransactionNotification

Sent from **transactionDidCommit** to tell observers that a transaction has been committed. The notification contains:

Notification Object	The notifying EOAdaptorContext object

Userinfo	None

EOAdaptorContextRollbackTransactionNotification

Sent from **transactionDidRollback** to tell observers that a transaction has been rolled back. The notification contains:

Notification Object	The notifying EOAdaptorContext object
Userinfo	None

EOAdaptorContext

EOAdaptorContext is an abstract class that provides its concrete subclasses with a structure for handling database transactions. It's associated with EOAdaptor and EOAdaptorChannel, which, together with EOAdaptorContext, form the *adaptor level* of Enterprise Objects Framework's access layer. See the EOAdaptor class specification for more information about accessing, creating, and using adaptor level objects.

A concrete subclass of EOAdaptorContext provides database-specific method implementations and represents a single transaction scope (logical user) on the database server to which its EOAdaptor object is connected. You never interact with instances of the EOAdaptorContext class, rather your Enterprise Objects Framework applications use instances of concrete subclasses that are written to work with a specific database or other persistent storage system. To create an instance of a concrete EOAdaptorContext subclass, you send a **createAdaptorContext** message to an instance of the corresponding EOAdaptor subclass. You rarely create adaptor contexts yourself. They are generally created automatically by other framework objects.

If a database server supports multiple concurrent transaction sessions, an adaptor context's EOAdaptor can have several contexts. When you use multiple EOAdaptorContexts for a single EOAdaptor, you can have several database server transactions in progress simultaneously. You should be aware of the issues involved in concurrent access if you do this.

An EOAdaptorContext has an EOAdaptorChannel, which handles actual access to the data on the server. If the database server supports it, a context can have multiple channels. See your adaptor context's documentation to find out if your adaptor supports multiple channels. An EOAdaptorContext by default has no EOAdaptorChannels; to create a new channel send your EOAdaptorContext a **createAdaptorChannel** message.

Controlling Transactions

EOAdaptorContext defines a simple set of methods for explicitly controlling transactions: **beginTransaction**, **commitTransaction**, and **rollbackTransaction**. Each of these messages confirms the requested action with the adaptor context's delegate, then performs the action if possible.

There's also a set of methods for notifying an adaptor context that a transaction has been started, committed, or rolled back without using the **beginTransaction**, **commitTransaction**, or **rollbackTransaction** methods. For example, if you invoke a stored procedure in the server that begins a transaction, you need to notify the adaptor context that a transaction has been started. Use the following methods to keep an adaptor context synchronized with the state of the database server: **transactionDidBegin**, **transactionDidCommit**, and **transactionDidRollback**. These methods post notifications.

The Adaptor Context's Delegate and Notifications

You can assign a delegate to an adaptor context. The delegate responds to certain messages on behalf of the context. An EOAdaptorContext sends these messages directly to its delegate. The transaction-controlling

methods—**beginTransaction**, **commitTransaction**, and **rollbackTransaction**—notify the adaptor context's delegate before and after a transaction operation is performed. Some delegate methods, such as **adaptorContextShouldBegin:**, let the delegate determine whether the context should perform an operation. Others, such as **adaptorContextDidBegin:**, are simply notifications that an operation has occurred. The delegate has an opportunity to respond by implementing the delegate methods. If the delegate wants to intervene, it implements **adaptorContextShouldBegin:**. If it simply wants notification when a transaction has begun, it implements **adaptorContextDidBegin:**.

EOAdaptorContext also posts notifications to the application's default notification center. Any object may register to receive one or more of the notifications posted by an adaptor context by sending the message **addObserver:selector:name:object:** to the default notification center (an instance of the NSNotificationCenter class). For more information on notifications, see the NSNotificationCenter class specification in the *Foundation Framework Reference*.

Creating an EOAdaptorContext Subclass

EOAdaptorContext provides many default method implementations that are sufficient for concrete subclasses. The following methods establish structure and conventions that other Enterprise Objects Framework classes depend on and should never be overridden:

- + setDebugEnabledDefault:
- – transactionDidBegin
- – transactionDidCommit
- – transactionDidRollback
- – transactionNestingLevel

Other methods require database-specific implementations that can be provided only by a concrete adaptor context subclass. A subclass must override the following methods in terms of the persistent storage system to which it interacts:

- – beginTransaction
- – canNestTransactions
- – commitTransaction
- – createAdaptorChannel
- – rollbackTransaction

EOAdaptorOperation

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EODatabaseOperation.h

Class Description

An EOAdaptorOperation object represents a primitive operation in a database server—lock, insert, update, or delete a row; or execute a stored procedure—and all the necessary information required by the operation. An EOAdaptorOperation is processed by an EOAdaptorChannel object in the method **performAdaptorOperation:**. You don't ordinarily create instances of EOAdaptorOperation; rather, the Framework automatically creates an EOAdaptorOperation object and sends it to an adaptor channel when your application needs the database server to perform an operation. You generally interact with EOAdaptorOperation objects only if you need to specify the order in which a set of operations are carried out (see the description for the EODatabaseContext delegate method **databaseContext: willOrderAdaptorOperationsFromDatabaseOperations:**).

An EOAdaptorOperation has an entity and an operator (the type of operation the object represents). An adaptor operation's operator (EOAdaptorLockOperator, EOAdaptorInsertOperator, EOAdaptorUpdateOperator, EOAdaptorDeleteOperator, or EOAdaptorStoredProcedureOperator) determines additional, operator-dependent information used by the EOAdaptorOperation object. For example, only a stored procedure operation has an EOStoredProcedure object. The operator-dependent information is accessible using the methods described below.

Method Types

Creating a new EOAdaptorOperation
- initWithEntity:

Accessing the entity
- entity

Accessing the operator
- setAdaptorOperator:
- adaptorOperator

Accessing the qualifier
- setStoredProcedure:
- qualifier

Accessing locking attributes
– setAttributes:
– attributes

Accessing operation values
– setChangedValues:
– changedValues

Accessing a stored procedure
– setStoredProcedure:
– storedProcedure

Handling errors during the operation
– setException:
– exception

Comparing operations
– compareAdaptorOperation:

Instance Methods

adaptorOperator

– (EOAdaptorOperator)**adaptorOperator**

Returns the receiver's adaptor operator. The operator indicates which of the other adaptor operation attributes are valid. For example, an adaptor operation whose operator is EOAdaptorInsertOperator uses **changedValues**, but not **attributes**, **qualifier**, or **storedProcedure**.

See also: **setAdaptorOperator:**

attributes

– (NSArray *)**attributes**

Returns the array of attributes to select when locking the row. If attributes have not been assigned to the receiver, the primary key attributes are selected. Only valid for adaptor operations with the EOAdaptorLockOperator.

See also: – **setAttributes:**

changedValues

– (NSDictionary *)**changedValues**

Returns the dictionary of values that need to be updated, inserted, or compared for locking purposes.

See also: – **setChangedValues:**

compareAdaptorOperation:

– (NSComparisonResult)**compareAdaptorOperation:**(EOAdaptorOperation *)*operation*

Orders adaptor operations alphabetically by entity name and by adaptor operator within the same entity. The adaptor operators are ordered as follows:

- EOAdaptorLockOperator
- EOAdaptorInsertOperator
- EOAdaptorUpdateOperator
- EOAdaptorDeleteOperator
- EOAdaptorStoredProcedureOperator

EOAdaptorLockOperator precedes EOAdaptorInsertOperator, EOAdaptorInsertOperator precedes EOAdaptorUpdateOperator, and so on.

An EODatabaseContext uses **compareAdaptorOperation:** to order adaptor operations before invoking EOAdaptorChannel's **performAdaptorOperations:** method.

entity

– (EOEntity *)**entity**

Returns the entity to which the operation will be applied.

See also: – **initWithEntity:**

exception

– (NSException *)**exception**

Returns the exception that was raised when an adaptor channel attempted to process the receiver. Returns **nil** if no exception was raised or if the receiver hasn't been processed yet.

See also: – **setException:**

qualifier

– (EOQualifier *)**qualifier**

Returns the qualifier that identifies the specific row to which the operation applies. Not valid with adaptor operations with the operators EOAdaptorInsertOperator and EOAdaptorStoredProcedureOperator.

initWithEntity:

– **initWithEntity:**(EOEntity *)*entity*

The designated initializer, initializes a new EOAdaptorOperation instance, and sets the entity to which the operation will be applied. Returns **self**.

See also: – **entity**

setAdaptorOperator:

– (void)**setAdaptorOperator:**(EOAdaptorOperator)*adaptorOperator*

Sets the receiver's operator to *adaptorOperator,* which is one of the following:

- EOAdaptorLockOperator
- EOAdaptorInsertOperator
- EOAdaptorUpdateOperator
- EOAdaptorDeleteOperator
- EOAdaptorStoredProcedureOperator

For more information, see the discussion on adaptor operators in the class description above.

See also: – **adaptorOperator**

setAttributes:

– (void)**setAttributes:**(NSArray *)*attributes*

Sets the array of attributes to select when locking the row. The selected values are compared in memory to the corresponding snapshot values to determine if a row has changed since the application last fetched it. *attributes* is an array of EOAttribute objects that can't be compared in a qualifier (generally BLOB types); it should not be **nil** or empty. Generally, an adaptor operation's qualifier contains all the comparisons needed to verify that a row hasn't changed since the application last fetched, inserted, or updated it. In this case (if there aren't any attributes that can't be compared in a qualifier), *attributes* should contain primary key attributes. This method is only valid for adaptor operations with the EOAdaptorLockOperator.

See also: – **attributes**, – **entity**

setChangedValues:

– (void)**setChangedValues:**(NSDictionary *)*changedValues*

Sets the dictionary of values that need to be updated, inserted, or compared for locking purposes. *changedValues* is a dictionary object whose keys are attribute names and whose values are the values for those attributes. As summarized in the following table, the contents of *changedValues* depends on the receiver's operator:

Operator	Contents of changedValues Dictionary
EOAdaptorLockOperator	snapshot values used to verify that the database row hasn't changed since this application last fetched it
EOAdaptorInsertOperator	the values to insert
EOAdaptorUpdateOperator	the new values for the columns to update
EOAdaptorDeleteOperator	snapshot values (*changedValues* is only valid for AdaptorDeleteOperation if the receiver's entity uses a stored procedure to perform delete operations.)
EOAdaptorStoredProcedureOperator	snapshot values

See also: – **changedValues**

setException:

– (void)**setException:**(NSException *)*exception*

Sets the receiver's exception to *exception*. This method is typically invoked from EOAdaptorChannel's **performAdaptorOperations:** method. If a database error occurs while processing an adaptor operation, the adaptor channel creates an exception and assigns it to the adaptor operation.

See also: – **exception**

setQualifier:

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

Sets the qualifier that identifies the row to which the adaptor operation is to be applied to *qualifier.*

See also: – **qualifier**

setStoredProcedure:

– (void)**setStoredProcedure:**(EOStoredProcedure *)*storedProcedure*

Sets the receiver's stored procedure to *storedProcedure*.

See also: – **storedProcedure**

storedProcedure

– (EOStoredProcedure *)**storedProcedure**

Returns the receiver's stored procedure. Only valid with adaptor operations with the EOAdaptorStoredProcedureOperation.

See also: – **setStoredProcedure:**

EOAttribute

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOAttribute.h

Class Description

An EOAttribute represents a column, field or property in a database, and associates an internal name with an external name or expression by which the property is known to the database. The property an EOAttribute represents may be a meaningful value, such as a salary or a name, or it may be an arbitrary value used for identification but with no real-world applicability (ID numbers and foreign keys for relationships fall into this category). An EOAttribute also maintains type information for binding values to the instance variables of objects.

EOAttributes are also used to represent arguments for EOStoredProcedures.

You usually define attributes in your EOModel with the EOModeler application, which is documented in *WebObjects Tools and Techniques*. Your code probably won't need to programmatically interact with EOAttribute unless you're working at the adaptor level. See "Creating Attributes" for information on creating your own attribute objects.

Fore detailed discussion of using attribute objects to map database data types to JavaObjective-C objects, see "Mapping Attributes." EOAttributes can also alter the way values are selected, inserted, and updated in the database by defining special format strings; see "SQL Statement Formats" for more information.

Adopted Protocols

EOPropertyListEncoding

– awakeWithPropertyList
– encodeIntoPropertyList:
– initWithPropertyList:owner:

Method Types

Accessing the entity

– entity
– parent

Accessing the name

– setName:
– name
– validateName:
– beautifyName

Accessing date information

– serverTimeZone
– setServerTimeZone:

Accessing external definitions

– setColumnName:
– columnName
– setDefinition:
– definition
– setExternalType:
– externalType

Accessing value type information

– setValueClassName:
– valueClassName
– setValueType:
– valueType
– setAllowsNull:
– allowsNull
– setPrecision:
– precision
– setScale:
– scale
– setWidth:
– width
– validateValue:

Converting to adaptor value types

– adaptorValueByConvertingAttributeValue:
– adaptorValueType

Working with custom value types

- setValueFactoryMethodName:
- valueFactoryMethod
- valueFactoryMethodName
- setFactoryMethodArgumentType:
- factoryMethodArgumentType
- setAdaptorValueConversionMethodName:
- adaptorValueConversionMethod
- adaptorValueConversionMethodName

Accessing attribute characteristics

- setReadOnly:
- isReadOnly
- isDerived
- isFlattened

Accessing SQL statement formats

- setReadFormat:
- readFormat
- setWriteFormat:
- writeFormat

Accessing the user dictionary

- setUserInfo:
- userInfo

Methods used by the adaptor

- newDateForYear:month:day:hour:minute:second:millisecond: timezone:zone:
- newValueForBytes:length:
- newValueForBytes:length:encoding:

Working with stored procedures

- setParameterDirection:
- parameterDirection
- storedProcedure

Working with prototypes

- overridesPrototypeDefinitionForKey:
- prototype
- prototypeName
- setPrototype:

<<Need to add more info to this on the implications to custom value classes.>>

Instance Methods

adaptorValueByConvertingAttributeValue:

– (id)**adaptorValueByConvertingAttributeValue:**(id)*value*

Ensures that *value* is eitheran NSString, NSNumber, NSData, or NSDate, converting it if necessary. If *value* needs to be converted, **adaptorValueByConvertingAttributeValue:** uses the adaptor conversion method to convert *value* to one of these four primitive types. If the attribute hasn't a specific adaptor conversion method, and the type to be fetched from the database is EOAdaptorBytesType, "archiveData" will be invoked to convert the attribute value.

See also: – **adaptorValueConversionMethod**, – **adaptorValueType**

adaptorValueConversionMethod

– (SEL)**adaptorValueConversionMethod**

Returns the method used to convert a custom class into one of the primitive types that the adaptor knows how to manipulate: NSString, NSNumber, NSData, or NSDate. The return value of this method is derived from the attribute's adaptor value conversion method name. If that name doesn't map to a valid selector in the Objective-C run-time, **nil** is returned.

See also: **– adaptorValueByConvertingAttributeValue:**, **– adaptorValueConversionMethodName**

adaptorValueConversionMethodName

– (NSString *)**adaptorValueConversionMethodName**

Returns the name of the method used to convert a custom class into one of the primitive types that the adaptor knows how to manipulate: NSString, NSNumber, NSData, or NSDate.

See also: **– adaptorValueByConvertingAttributeValue:**

adaptorValueType

– (EOAdaptorValueType)**adaptorValueType**

Returns an EOAdaptorValueType that indicates the data type that will be fetched from the database. Currently, this method returns one of the following values:

Constant	Description
EOAdaptorNumberType	A number value
EOAdaptorCharactersType	A string of characters
EOAdaptorBytesType	Raw bytes
EOAdaptorDateType	A date

See also: – **factoryMethodArgumentType**

allowsNull

– (BOOL)**allowsNull**

Returns YES to indicate that the attribute can have a **nil** value, NO otherwise. If the attribute maps directly to a column in the database, it also is used to determine whether the database column can have a NULL value.

See also: – **setAllowsNull:**

beautifyName

– (void)**beautifyName**

Makes the attribute name conform to a standard convention. Names that conform to this style are all lower-case except for the initial letter of each embedded word other than the first, which is upper case. Thus, “NAME” becomes “name”, and “FIRST_NAME” becomes “firstName”. This method is used in reverse-engineering an EOModel.

See also: – **validateName:**, – **beautifyNames** (EOModel)

columnName

– (NSString *)**columnName**

Returns the name of the column in the database that corresponds to this attribute, or **nil** if the attribute isn't simple (that is, if it's derived or flattened). An adaptor uses this name to identify the column corresponding to the attribute. Your application should never need to use this name. Note that **columnName** and **definition** are mutually exclusive; if one returns a value, the other returns **nil**.

See also: , **– externalType**

definition

– (NSString *)**definition**

Returns a derived or flattened attribute's definition, or **nil** if the attribute is simple. An attribute's definition is either a value expression defining a derived attribute, such as "salary * 12", or a data path for a flattened attribute, such as "toAuthor.name". Note that **columnName** and **definition** are mutually exclusive; if one returns a value, the other returns **nil**.

See also: **– externalType, – setDefinition:**

entity

– (EOEntity *)**entity**

Returns the entity that owns the attribute, or **nil** if this attribute is acting as an argument for a stored procedure.

See also: – **storedProcedure**

externalType

– (NSString *)**externalType**

Returns the attribute's type as understood by the database; for example, a Sybase "varchar" or an Oracle "NUMBER".

See also: – **columnName, – setExternalType:**

factoryMethodArgumentType

– (EOFactoryMethodArgumentType)**factoryMethodArgumentType**

Returns the type of argument that should be passed to the "factory method"—which is invoked by the attribute to create an attribute value for a custom class. This method returns one of the following values:

Constant	Argument Type
EOFactoryMethodArgumentIsNSData	NSData
EOFactoryMethodArgumentIsNSString	NSString
EOFactoryMethodArgumentIsBytes	raw bytes

See also: – **valueFactoryMethod**, – **setFactoryMethodArgumentType:**

isDerived

– (BOOL)**isDerived**

Returns NO if the attribute corresponds exactly to one column in the table associated with its entity, and YES if it doesn't. For example, an attribute with a definition of "otherAttributeName + 1" is derived.

Note that flattened attributes are also considered as derived attributes.

See also: – **isFlattened**, – **definition**

isFlattened

– (BOOL)**isFlattened**

Returns YES if the attribute is flattened, NO otherwise. A flattened attribute is one that's accessed through an entity's relationships but belongs to another entity.

Note that flattened attributes are also considered to be derived attributes.

See also: – **isDerived**, – **definition**

isReadOnly

– (BOOL)**isReadOnly**

Returns YES if the value of the attribute can't be modified, NO if it can.

See also: – **setReadOnly:**

name

– (NSString *)**name**

Returns the attribute's name.

See also: – **columnName**, – **definition, – setName:**

newDateForYear:month:day:hour:minute:second:millisecond:timezone:zone:

– (NSCalendarDate *)**newDateForYear:**(int)*year* **month:**(unsigned)*month* **day:**(unsigned)*day* **hour:** (unsigned)*hour* **minute:**(unsigned)*minute* **second:**(unsigned)*second* **millisecond:** (unsigned)*millisecond* **timezone:**(NSTimeZone *)*timezone* **zone:**(NSZone *)*zone*

Returns an NSCalendarDate given discrete values for year, month, day, and so on. This method is used by EOAdaptorChannel subclasses to create a calendar date object to return in an adaptor row. For efficiency reasons, the caller is responsible for releasing the return value.

newValueForBytes:length:

– (id)**newValueForBytes:**(const void *)*bytes* **length:**(int)*length*

Generates an NSString or custom class value object from a supplied set of bytes. This method is called by the adaptor during value creation while fetching from the database. For efficiency reasons, the caller is responsible for releasing the return value.

newValueForBytes:length:encoding:

– (id)**newValueForBytes:**(const void *)*bytes* **length:**(int)*length* **encoding:** (NSStringEncoding)*encoding*

Generates an NSData or custom class value object from a supplied set of bytes with a given NSStringEncoding. This method is called by the adaptor during value creation while fetching from the database. For efficiency reasons, the caller is responsible for releasing the return value.

overridesPrototypeDefinitionForKey:

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

Returns NO if the requested key gets its value from the prototype attribute. If the attribute has an override, then this method returns YES. Valid values for key include @"columnName," @"valueClass," and so on.

See also: – **prototype**

parameterDirection

– (EOParameterDirection)**parameterDirection**

Returns the parameter direction for attributes that are arguments to a stored procedure. This method returns one of the following values:

Constant	Description
EOVoid	No parameters
EOInParameter	Input only parameters
EOOutParameter	Output only parameters
EOInOutParameter	Bidirectional parameters (input and output)

See also: – **storedProcedure**, – **storedProcedureForOperation:** (EOEntity), – **setParameterDirection:**

parent

– (id)**parent**

Returns the attribute's parent, which is either an EOEntity or an EOStoredProcedure. Use this method when you need to find the model for an attribute:

```
EOModel *myModel = [[anAttribute parent] model];
```

precision

– (unsigned)**precision**

Returns the precision of the database representation for attributes with a value class of NSNumber or NSDecimalNumber.

See also: – **scale**

prototype

– (EOAttribute *)**prototype**

Returns the prototype attribute that is used to define default settings for the receiver.

See also: – **overridesPrototypeDefinitionForKey:**

prototypeName

– (NSString *)**prototypeName**

Returns the name of the prototype attribute of the receiver.

See also: – **prototype**

readFormat

– (NSString *)**readFormat**

Returns a format string of the appropriate type that can be used when building an expression that contains the value of the attribute.

See also: – **setReadFormat:**, **– writeFormat**

scale

– (int)**scale**

Returns the scale of the database representation for attributes with a value class of NSNumber or NSDecimalNumber. The returned value can be negative.

See also: – **precision, – setScale:**

serverTimeZone

– (NSTimeZone *)**serverTimeZone**

Returns the time zone assumed for NSDates in the database server, or the local time zone if one hasn't been set. An EOAdaptorChannel automatically converts dates between the time zones used by the server and the client when fetching and saving values. Applies only to attributes that represent dates.

See also: + **localTimeZone** (NSTimeZone), – **setServerTimeZone:**

setAdaptorValueConversionMethodName:

– (void)**setAdaptorValueConversionMethodName:**(NSString *)*conversionMethodName*

Sets to *conversionMethodName* the name of the method used to convert a custom class into one of the primitive types that the adaptor knows how to manipulate: NSString, NSNumber, NSData, or NSDate. Note that your adaptor value conversion method should return an autoreleased object.

See also: **– adaptorValueConversionMethodName**

setAllowsNull:

– (void)**setAllowsNull:**(BOOL)*allowsNull*

Sets according to *allowsNull* whether or not the attribute can have a **nil** value. If the attribute maps directly to a column in the database, it also controls whether the database column can have a NULL value.

See also: – **allowsNull**

setColumnName:

– (void)**setColumnName:**(NSString *)*columnName*

Sets to *columnName* the name of the attribute used in communication with the database server. An adaptor uses this name to identify the column corresponding to the attribute; this name must match the name of a column in the database table corresponding to the attribute's entity.

This method makes a derived or flattened attribute simple; the **definition** is released and the column name takes its place for use with the server.

Note: **setColumnName:** and **setDefinition:** are closely related. Only one can be set at any given time. Invoking either of these methods causes the other value to be set to **nil**.

See also: **– columnName**

setDefinition:

– (void)**setDefinition:**(NSString *)*definition*

Sets to *definition* the attribute's definition as recognized by the database server. *definition* should be either a value expression defining a derived attribute, such as "salary * 12", or a data path for a flattened attribute, such as "toAuthor.name".

Prior to invoking this method, the attribute's entity must have been set by adding the attribute to an entity. This method will not function correctly if the attribute's entity has not been set.

This method converts a simple attribute into a derived or flattened attribute; the **columnName** is released and the definition takes its place for use with the server.

Note: **setColumnName:** and **setDefinition:** are closely related. Only one can be set at any given time. Invoking either of these methods causes the other value to be set to **nil**.

See also: **– definition**

setExternalType:

– (void)**setExternalType:**(NSString *)*typeName*

Sets to *typeName* the type used for the attribute in the database adaptor; for example, a Sybase “varchar” or an Oracle7 “NUMBER”. Each adaptor defines the set of types that can be supplied to **setExternalType:**. The external type you specify for a given attribute must correspond to the type used in the database server.

See also: – **setDefinition:, – externalType**

setFactoryMethodArgumentType:

– (void)**setFactoryMethodArgumentType:**(EOFactoryMethodArgumentType)*argumentType*

Sets the type of argument that should be passed to the “factory method”—which is invoked by the receiver to create a value for a custom class. Factory methods can accept NSStrings, NSDatas, or raw bytes; specify an *argumentType* as EOFactoryMethodArgumentIsNSString, EOFactoryMethodArgumentIsNSData, or EOFactoryMethodArgumentIsBytes as appropriate.

See also: – **setValueFactoryMethodName:, – factoryMethodArgumentType**

setName:

– (void)**setName:**(NSString *)*name*

Sets the attribute’s name to *name*. Raises an NSInvalidArgumentException if *name* is already in use by another attribute or relationship of the same entity, or if *name* is not a valid attribute name.

See also: – **validateName:, – name**

setParameterDirection:

– (void)**setParameterDirection:**(EOParameterDirection)*parameterDirection*

Sets the parameter direction for attributes that are arguments to a stored procedure. *parameterDirection* should be one of the following values:

- EOVoid
- EOInParameter
- EOOutParameter
- EOInOutParameter

See also: **– setStoredProcedure:forOperation:** (EOEntity), – **parameterDirection**

setPrecision:

– (void)**setPrecision:**(unsigned)*precision*

Sets to *precision* the precision of the database representation for attributes with a value class of NSNumber or NSDecimalNumber.

See also: – **setScale:, – precision**

setPrototype:

– (void)**setPrototype:**(EOAttribute *)*prototype*

Sets the prototype attribute. This overrides any existing settings in the attribute.

See also: – **prototype**

setReadFormat:

– (void)**setReadFormat:**(NSString *)*aString*

Sets the format string that's used to format the attribute's value for SELECT statements. In *aString*, %P is replaced by the attribute's external name. For example:

```
[myAttribute setReadFormat:@"TO_UPPER(%P)"];
```

The read format string is used whenever the attribute is referenced in a select list or qualifier.

See also: **– setWriteFormat:, – readFormat**

setReadOnly:

– (void)**setReadOnly:**(BOOL)*flag*

Sets whether the value of the attribute can be modified according to *flag*. Raises an NSInvalidArgumentException if *flag* is NO and the argument is derived but not flattened.

See also: – **isDerived**, – **isFlattened, – isReadOnly**

setScale:

– (void)**setScale:**(int)*scale*

Sets to *scale* the scale of the database representation for attributes with a value class of NSNumber or NSDecimalNumber. *scale* can be negative.

See also: – **setPrecision:, – scale**

setServerTimeZone:

– (void)**setServerTimeZone:**(NSTimeZone *)*aTimeZone*

Sets to *aTimeZone* the time zone used for NSDates in the database server. If *aTimeZone* is **nil** then the local time zone is used. An EOAdaptorChannel automatically converts dates between the time zones used by the server and the client when fetching and saving values. Applies only to attributes that represent dates.

See also: – **serverTimeZone**

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets to *dictionary* the dictionary of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types (that is, NSDictionary, NSArray, NSData, and NSString).

See also: – **userInfo**

setValueClassName:

– (void)**setValueClassName:**(NSString *)*name*

Sets the class name for values of this attribute to *name*. When an EOAdaptorChannel fetches data for the attribute, it's presented to the application as an instance of this class.

The class need not exist in the run-time system when this message is sent, but it must exist when an adaptor channel performs a fetch; if the class isn't present the result depends on the adaptor. See your adaptor's documentation for information on how absent value classes are handled.

As an example, if your attribute's values are instances of NSImage, send the following:

```
[myAttribute setValueClassName:@"NSImage"];
```

See also: – **setValueType:, – valueClassName**

setValueFactoryMethodName:

– (void)**setValueFactoryMethodName:**(NSString *)*factoryMethodName*

Sets the "factory method"—which is invoked by the attribute to create an attribute value for a custom class—to *factoryMethodName*. The factory method should be a class method returning an autoreleased object of your custom value class. Use **setFactoryMethodArgumentType:**to specify the type of argument that is to be passed to your factory method.

See also: – **valueFactoryMethodName**

setValueType:

– (void)**setValueType:**(NSString *)*typeName*

Sets to *typeName* the conversion character (such as “i” or “d”) for the data type an NSNumber attribute is converted to and from in your application. Value types are scalars such as **int**, **float**, and **double**. Each adaptor supports a different set of conversion characters for numeric types. However, in most (if not all) cases it’s safe to supply a value of “i” (int) or “d” (double).

See also: – **setValueClassName:, – valueType**

setWidth:

– (void)**setWidth:**(unsigned)*length*

Sets to *length* the maximum amount of bytes the attribute’s value may contain. Adaptors may use this information to allocate space for fetch buffers.

See also: **– width**

setWriteFormat:

– (void)**setWriteFormat:**(NSString *)*string*

Sets the format string that’s used to format the attribute’s value for INSERT or UPDATE expressions. In *string*, %P is replaced by the attribute’s value. For example:

```
[myAttribute setWriteFormat:@"TO_LOWER(%P)"];
```

See also: **– setReadFormat:, – writeFormat**

storedProcedure

– (EOStoredProcedure *)**storedProcedure**

Returns the stored procedure for which this attribute is an argument. If this attribute isn’t an argument to a stored procedure but instead is owned by an entity, this method returns **nil**.

See also: – **entity**

userInfo

– (NSDictionary *)**userInfo**

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

See also: – **setUserInfo:**

validateName:

– (NSException *)**validateName:**(NSString *)*name*

Validates *name* and returns **nil** if it is a valid name, or an exception if it isn't. A name is invalid if it has zero length; starts with a character other than a letter, a number, or "@", "#", or "_"; or contains a character other than a letter, a number, "@", "#", "_", or "$". A name is also invalid if the receiver's EOEntity already has an EOAttribute with the same name, or if the model has a stored procedure that has an argument with the same name.

setName: uses this method to validate its argument.

validateValue:

– (NSException *)**validateValue:**(id *)*valueP*

Validates the argument by converting it to the attribute's value type and by testing other attribute validation constraints (such as **allowsNull**, **width**, and so on). Returns **nil** if **valueP* is deemed to be a legal value for this attribute. Returns a validation exception otherwise. If, during the validation process, any coercion was performed, the converted value is assigned to **valueP*.

See also: **– adaptorValueByConvertingAttributeValue:**, – **allowsNull**, – **valueType**, – **valueClassName**, – **width**

valueClassName

– (NSString *)**valueClassName**

Returns the name of the class for custom value types. When data is fetched for the attribute, it's presented to the application as an instance of this class. For example, if a column from the database is represented by instances of NSImage, this method returns "NSImage".

This class must be present in the run-time system when an EOAdaptorChannel fetches data for the attribute; if the class isn't present the result depends on the adaptor. See your adaptor's documentation for information on how absent value classes are handled.

See also: **– valueType, – setValueClassName:**

valueFactoryMethod

– (SEL)**valueFactoryMethod**

Returns the factory method that's invoked by the attribute when creating an attribute value that's of a custom class. The value returned from this method is derived from the attribute's **valueFactoryMethodName**. If that name doesn't map to a valid selector in the Objective-C run-time, this method returns **nil**.

valueFactoryMethodName

– (NSString *)**valueFactoryMethodName**

Returns the name of the factory method that's used for creating a custom class value.

See also: – **valueFactoryMethod, – setValueFactoryMethodName:**

valueType

– (NSString *)**valueType**

Returns the conversion character (such as "i" or "d") for the data type an NSNumber attribute is converted to and from in your application. Value types are scalars such as **int**, **float**, and **double**.

See also: – **valueClassName, – setValueType:**

width

– (unsigned)**width**

Returns the maximum length (in bytes) for values that are mapped to this attribute. Returns zero for numeric and date types.

See also: – **setWidth:**

writeFormat

– (NSString *)**writeFormat**

Returns the format string that's used to format the attribute's value for INSERT or UPDATE expressions. In the returned string, %P is replaced by the attribute's value.

See also: **– readFormat, – setWriteFormat:**

Creating Attributes

An attribute may be simple, derived, or flattened. A simple attribute typically corresponds to a single column in the database, and may be read or updated directly from or to the database. A simple EOAttribute may also be set as read-only with its **setReadOnly:** method. Read-only attributes of enterprise objects are never updated.

A derived attribute doesn't necessarily correspond to a single database column in its entity's database table, and is usually based on some other attribute, which is modified in some way. For example, if an Employee entity has a simple monthly salary attribute, you can define a derived **annualSalary** attribute as "salary * 12". Derived attributes, since they don't correspond to actual values in the database, are effectively read-only; it makes no sense to write a derived value.

A flattened attribute of an entity is actually an attribute of some other entity that's fetched through a relationship with a database join. A flattened attribute's external definition is a data path ending in an attribute name. For example, if the Employee entity has the relationship **toAddress** and the Address entity has the attribute **street**, you can define **streetName** as an attribute of your Employee EOEntity by creating an EOAttribute for it with a definition of "toAddress.street".

Creating a Simple Attribute

A simple attribute needs at least the following characteristics:

- A name unique within its EOEntity
- The name of a column in the database table for its entity (the EOAttribute's external name)
- A declaration of the type of that column as defined by the database and adaptor (the EOAttribute's external type)
- A declaration of the Objective-C class used to represent values outside the context of an enterprise object
- For Objective-C value classes that require it, a subtype for such distinctions as between numeric types

You also have to set whether the attribute is part of its entity's primary key, is a class property, or is used for locking. See the EOEntity class description for more information on these three groups of attributes. This code excerpt gives an example of creating a simple EOAttribute and adding it to an EOEntity:

```
EOEntity *employeeEntity;    /* Assume this exists. */
EOAttribute *salaryAttribute;
NSArray *empClassProps;
NSArray *empLockAttributes;
BOOL result;

salaryAttribute = [[EOAttribute alloc] init];
[salaryAttribute setName:@"salary"];
[salaryAttribute setColumnName:@"SALARY"];
[salaryAttribute setExternalType:@"money"];
[salaryAttribute setValueClassName:"NSDecimalNumber"];
```

```
[employeeEntity addAttribute:salaryAttribute];
[salaryAttribute release];

empClassProps = [[employeeEntity classProperties] mutableCopy];
[empClassProps addObject:salaryAttribute];
[employeeEntity setClassProperties:empClassProps];
[empClassProps release];

empLockAttributes = [[employeeEntity attributesUsedForLocking]
    mutableCopy];
[empLockAttributes addObject:salaryAttribute];
result = [employeeEntity setAttributesUsedForLocking:empLockAttributes];
[empLockAttributes release];
```

Creating a Derived Attribute

A derived attribute depends on another attribute, so you base it on a definition including that attribute's name rather than on an external name. Because a derived attribute isn't mapped directly to anything in the database, you shouldn't include it in the entity's set of primary key attributes or attributes used for locking:

```
EOEntity *employeeEntity;    /* Assume this exists. */
EOAttribute *bonusAttribute;
NSArray *empClassProps;
BOOL result;

bonusAttribute = [[EOAttribute alloc] init];
[bonusAttribute setName:@"bonus"];
[bonusAttribute setDefinition:@"salary * 0.5"];
[bonusAttribute setValueClassName:@"NSDecimalNumber"];
[employeeEntity addAttribute:bonusAttribute];
[bonusAttribute release];

empClassProps = [[employeeEntity classProperties] mutableCopy];
[empClassProps addObject:bonusAttribute];
result = [employeeEntity setClassProperties:empClassProps];
[empClassProps release];
```

Creating a Flattened Attribute

A flattened attribute depends on a relationship, so you base it on a definition including that relationship's name rather than on an external name. Because a flattened attribute doesn't correspond directly to anything in its entity's table, you don't have to specify an external name, and shouldn't include it in the entity's set of primary key attributes or attributes used for locking:

```
EOEntity *employeeEntity;    /* Assume this exists. */
EOAttribute *deptNameAttribute;
```

```
NSArray *empClassProps;
BOOL result;

deptNameAttribute = [[EOAttribute alloc] init];
[deptNameAttribute setName:@"departmentName"];
[deptNameAttribute setValueClassName:"NSString"];
[deptNameAttribute setExternalType:@"varchar"];
[employeeEntity addAttribute:deptNameAttribute];
[deptNameAttribute setDefinition:@"toDepartment.name"];
[deptNameAttribute release];

empClassProps = [[employeeEntity classProperties] mutableCopy];
[empClassProps addObject:deptNameAttribute];
result = [employeeEntity setClassProperties:empClassProps];
[empClassProps release];
```

Instead of flattening attributes in your model, a better approach is often to directly traverse the object graph through relationships. See the chapter “Using EOModeler” in the *Enterprise Objects Framework Developer’s Guide* for a discussion on when to use flattened attributes.

Mapping Attributes

Mapping from Database to Objects

Every EOAttribute has an external type, which is the type used by the database to store its associated data, and an Objective-C class used as the type for that data in the client application. The type used by the database is accessed with the **setExternalType:** and **externalType** methods. The class type used by the application is accessed with the **valueClassName** method. You can map database types to a set of standard value classes, which includes:

- NSString
- NSNumber
- NSDecimalNumber
- NSData
- NSDate

Database-specific adaptors automatically handle value conversions for these classes. You can also create your own custom value class, so long as you define a format that it uses to interpret data. Your value class must also implement the EOCustomClassArchiving protocol to work as a customvalue; see that protocol specification for more information. For more information on using EOAttribute methods to work with custom data types, see the next section, “Working with Custom Data Types.”

The handling of dates assumes by default that both the database server and the client application are running in the same, local, time zone. You can alter the server time zone with the **setServerTimeZone:** method. If you alter the server time zone, the adaptor automatically converts dates as they pass into and out of the server.

Working with Custom Data Types

When you create a new model, EOModeler maps each attribute in your model to one of the primitive data types the adaptor knows how to manipulate: NSString, NSNumber, NSDecimalNumber, NSData, and NSDate. For example, suppose you have a **photo** attribute that’s stored in the database as a LONG RAW. When you create a new model, this attribute is mapped to NSData. However, NSData is just an object wrapper for binary data—for instance, it doesn’t have any methods for operating on images, which would limit what you’d be able to do with the image in your application. This is a case in which you’d probably choose to use a custom data type, such as NSImage.

For a custom data type to be usable in Enterprise Objects Framework, it must supply methods for importing and exporting itself as one of the primitive types so that it can be read from and written to the database. Specifically, to use a custom data type you need to do the following:

- Set the attribute’s value class using the method **setValueClassName:**.
- Set the factory method that will be used to create instances of your class from raw data using the method **setValueFactoryMethodName:**.

- Set the type of the argument that should be passed to the factory method using the method **setFactoryMethodArgumentType:**.
- Set the conversion method that is used to convert your data back into one of the primitive data types the adaptor can work with using the method **setAdaptorValueConversionMethodName:**; this enables the data to be stored in the database.

If an EOAttribute represents a binary column in the database, the factory method argument type can be either EOFactoryMethodArgumentIsNSData or EOFactoryMethodArgumentIsBytes, indicating that the method takes an NSData object or raw bytes as an argument. If the EOAttribute represents a string or character column, the factory method argument type can be either EOFactoryMethodArgumentIsNSString or EOFactoryMethodArgumentIsBytes, indicating that the method takes an NSString object or raw bytes as an argument. These types apply when fetching custom values, as described below.

The following code excerpt demonstrates how these methods work together. The example shows two custom data types: an image that's initialized with an NSData, and a custom zip code that's initialized with a string.

```
[imageAttribute setValueClassName:@"NSImage"];
[imageAttribute setFactoryMethodArgumentType:EOFactoryMethodArgumentIsNSData];
[imageAttribute setValueFactoryMethodName:@"imageWithData:"];
[imageAttribute setAdaptorValueConversionMethodName:@"TIFFRepresentation"];

[zipCodeAttribute setValueClassName:@"MyZipCodeClass"];
[zipCodeAttribute setFactoryMethodArgumentType:EOFactoryMethodArgumentIsBytes];
[zipCodeAttribute setValueFactoryMethodName:@"zipCodeWithBytes:length:"];
[zipCodeAttribute setAdaptorValueConversionMethodName:@"zipCodeString"];
```

Instead of setting the class information programmatically, you can use the Attributes Inspector in EOModeler, which is more common. For more information, see the chapter "Advanced Modeling Techniques" in the *Enterprise Objects Framework Developer's Guide*.

Fetching Custom Values

Custom values are created during fetching in EOAdaptorChannel's **fetchRowWithZone:** method. This method fetches data in the external (server) type and converts it to a value object. For scalar database types such as numbers and dates, the EOAdaptorChannel converts the value itself. For binary and string database types, it calls upon the EOAttribute being fetched to perform the conversion, into either a standard or custom value class. EOAttribute's methods for performing this conversion are **newValueForBytes:length:** for binary data and **newValueForBytes:length:encoding:** for strings. These methods either convert the raw data directly into an NSData or NSString, or apply the custom value factory method to convert it into the custom class. Once the value is converted, the EOAdaptorChannel puts it into the dictionary for the row being fetched.

newValueForBytes:length: can handle NSData and raw bytes (**void ***). It converts the raw bytes into an NSData if the custom value argument type is EOFactoryMethodArgumentIsNSData, then invokes the

custom value factory method with the NSData or bytes. If the EOAttribute has no custom value factory method, this method simply returns an NSData object containing the bytes.

newValueForBytes:length:encoding: can handle NSString and raw bytes. It converts the raw bytes into an NSString if the custom value argument type is EOFactoryMethodArgumentIsNSString, then it invokes the custom value factory method with the string or bytes. If the EOAttribute has no custom value factory method, this method simply returns an NSString object created from the bytes.

Converting Custom Values

Custom values are converted back to binary or character data in EOAdaptorChannel's **evaluateExpression:** method. For each value in the EOSQLExpression to be evaluated, the EOAdaptorChannel sends the appropriate EOAttribute an **adaptorValueByConvertingAttributeValue:** message to convert it. If the value is any of the standard value classes, it's returned unchanged. If the value is of a custom class, though, it's converted by applying the conversion method (**adaptorValueConversionMethod**) specified in the EOAttribute.

SQL Statement Formats

In addition to mapping database values to object values, an EOAttribute can alter the way values are selected, inserted, and updated in the database by defining special format strings. These format strings allow a client application to extend its reach right down to the server for certain operations. For example, you might want to view an employee's salary on a yearly basis, without defining a derived attribute as in a previous example. In this case, you could set the salary attribute's SELECT statement format to "salary * 12" (with **setReadFormat:**) and the INSERT and UPDATE statement formats to "salary / 12" (**setWriteFormat:**). Thus, whenever your application retrieves values for the salary attribute they're multiplied by 12, and when it writes values back to the database they're divided by 12.

Your application can use any legal SQL value expression in a format string, and can even access server-specific features such as functions and stored procedures (see EOEntity's **setStoredProcedure: forOperation:** method description for more information). Accessing server-specific features can offer your application great flexibility in dealing with its server, but does limit its portability. You're responsible for ensuring that your SQL is well-formed and will be understood by the database server.

Format strings for the **setReadFormat:** and **setWriteFormat:** methods should use "%P" as the substitution character for the value that is being formatted. "%@" will not work. For example:

```
[myAttribute setReadFormat:@"TO_UPPER(%P)"];
[myAttribute setWriteFormat:@"TO_LOWER(%P)"];
```

Instead of setting the read and write formats programmatically, you can set them in EOModeler, which is more common. For more information, see the chapter "Using EOModeler" in *WebObjects Tools and Techniques*.

EODatabase

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EODatabase.h

Class Description

An EODatabase object represents a single database server. It contains an EOAdaptor which is capable of communicating with the server, a list of EOModels that describe the server's schema, a list of EODatabaseContexts that are connected to the server, and a set of snapshots representing the state of all objects stored in the server.

For more information, see "EODatabase".

Method Types

Creating instances	– initWithModel: – initWithAdaptor:
Adding and removing models	– addModel: – addModelIfCompatible: – removeModel: – models
Accessing entities	– entityForObject: – entityNamed:

Recording snapshots

– recordSnapshot:forGlobalID:
– forgetSnapshotForGlobalID:
– forgetSnapshotsForGlobalIDs:
– recordSnapshots:
– forgetAllSnapshots
– snapshotForGlobalID:
– snapshots
– recordSnapshot:forSourceGlobalID:relationshipName:
– recordToManySnapshots:
– snapshotForSourceGlobalID:relationshipName:

Registering database contexts

– registerContext:
– unregisterContext:
– registeredContexts

Accessing the adaptor

– adaptor

Managing the result cache

– invalidateResultCache
– invalidateResultCacheForEntityNamed:
– resultCacheForEntityNamed:
– setResultCache:forEntityNamed:

Instance Methods

adaptor

– (EOAdaptor *)**adaptor**

Returns the EOAdaptor used by the receiver for communication with the database server. Your application can interact directly with the EOAdaptor, but should avoid altering its state (for example, by starting a transaction with one of its adaptor contexts).

addModel:

– (void)**addModel:**(EOModel *)*aModel*

Adds *aModel* to the receiver's list of EOModels. This allows EODatabases to load entities and their properties only as they're needed, by dividing them among separate EOModels. *aModel* must use the same EOAdaptor as the receiver and use the same connection dictionary as the receiver's other EOModels.

See also: – **addModelIfCompatible:**, – **models**, – **removeModel:**

addModelIfCompatible:

– (BOOL)**addModelIfCompatible:**(EOModel *)*aModel*

Adds *aModel* to the receiver's list of EOModels, checking first to see whether it's compatible with those other EOModels. Returns YES if *aModel* is already in the list or if it's successfully added. Returns NO if *aModel*'s adaptor name differs from that of the receivers or if the receiver's **adaptor** returns NO to a **canServiceModel:** message.

See also: – **addModel:**, – **models**, – **removeModel:**

entityForObject:

– (EOEntity *)**entityForObject:**(id)*anObject*

Returns the EOEntity from one of the receiver's Models that's mapped to *anObject*, or **nil** if there is no such EOEntity. This method works by sending **entityForObject:** messages to each of the receiver's EOModels and returning the first one found.

See also: – **entityNamed:**

entityNamed:

– (EOEntity *)**entityNamed:**(NSString *)*entityName*

Returns the EOEntity from one of the receiver's Models that's named *entityName*, or **nil** if there is no such EOEntity. This method works by sending **entityNamed:** messages to each of the receiver's EOModels and returning the first one found.

See also: – **entityForObject:**

forgetAllSnapshots

– (void)**forgetAllSnapshots**

Clears all of the receiver's snapshots and posts an EOObjectsChangedInStoreNotification (defined in the EOControl framework's EOObjectStore class) describing the invalidated object. For a description of snapshots and their role in an application, see the class description.

See also: – **forgetSnapshotForGlobalID:**, – **forgetSnapshotsForGlobalIDs:**, – **recordSnapshot:forGlobalID:**, – **recordSnapshots:**, – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **recordToManySnapshots:**

forgetSnapshotForGlobalID:

– (void)**forgetSnapshotForGlobalID:**(EOGlobalID *)*globalID*

Clears the snapshot made for the enterprise object identified by *globalID* and posts an EOObjectsChangedInStoreNotification (defined in the EOControl framework's EOObjectStore class) describing the invalidated object. For a description of snapshots and their role in an application, see the class description.

See also: – **forgetSnapshotsForGlobalIDs:**, – **forgetAllSnapshots**, – **recordSnapshot:forGlobalID:**

forgetSnapshotsForGlobalIDs:

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

Clears the snapshots made for the enterprise objects identified by each of the EOGlobalIDs in *globalIDs* and posts an EOObjectsChangedInStoreNotification (defined in the EOControl framework's EOObjectStore class) describing the invalidated objects. For a description of snapshots and their role in an application, see the class description.

See also: – **forgetSnapshotForGlobalID:**, – **forgetAllSnapshots**, – **recordSnapshots:**

initWithAdaptor:

– **initWithAdaptor:**(EOAdaptor *)*anAdaptor*

The designated initializer, this method initializes a newly allocated EODatabase with *anAdaptor* as its adaptor and returns **self**.

Typically, you don't need to programmatically create EODatabase objects. Rather, they are created automatically by the control layer. See the class description for more information. If you do need to create an EODatabase programmatically, you should never associate more than one EODatabase with a given EOAdaptor. In general, use **initWithModel:**, which automatically selects the adaptor.

initWithModel:

– **initWithModel:**(EOModel *)*aModel*

Initializes a newly allocated EODatabase by creating an instance of EOAdaptor named in *aModel* and invoking **initWithAdaptor:**. Returns **self**. Typically, you don't need to programmatically create EODatabase objects. Rather, they are created automatically by the control layer. See the class description for more information.

See also: + **adaptorWithModel:** (EOAdaptor), – **adaptorName** (EOModel)

invalidateResultCache

– (void)**invalidateResultCache**

Invalidates the receiver's result cache. See the class description for more discussion of this topic.

See also: – **invalidateResultCacheForEntityNamed:**, – **resultCacheForEntityNamed:**

invalidateResultCacheForEntityNamed:

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

Invalidates the result cache containing an array of globalIDs for the objects associated with the entity *entityName*. See the class description for more discussion of this topic.

See also: – **invalidateResultCache**, – **resultCacheForEntityNamed:**

models

– (NSArray *)**models**

Returns the receiver's EOModels.

See also: – **initWithModel:**, – **addModel:**, – **addModelIfCompatible:**, – **removeModel:**

recordSnapshot:forGlobalID:

– (void)**recordSnapshot:**(NSDictionary *)*aSnapshot* **forGlobalID:**(EOGlobalID *)*globalID*

Records *aSnapshot* under *globalID*. For a description of snapshots and their role in an application, see the class description.

See also: – **globalIDForRow:** (EOEntity), – **recordSnapshots:**, – **forgetSnapshotForGlobalID:**

recordSnapshot:forSourceGlobalID:relationshipName:

– (void)**recordSnapshot:**(NSArray *)*globalIDs*
forSourceGlobalID:(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*

For the object identified by *globalID*, records an NSArray of *globalIDs* for the to-many relationship named *name*. These *globalIDs* identify the objects at the destination of the relationship. For a description of snapshots and their role in an application, see the class description.

See also: – **recordSnapshot:forGlobalID:**, – **recordSnapshots:**, – **recordSnapshot:forGlobalID:**, – **snapshotForSourceGlobalID:relationshipName:**

recordSnapshots:

– (void)**recordSnapshots:**(NSDictionary *)*snapshots*

Records the snapshots in *snapshots*. *snapshots* is a dictionary whose keys are EOGlobalIDs and whose values are the snapshots for those global IDs. For a description of snapshots and their role in an application, see the class description.

See also: – **recordSnapshot:forGlobalID:**, – **forgetSnapshotsForGlobalIDs:**

recordToManySnapshots:

– (void)**recordToManySnapshots:**(NSDictionary *)*snapshots*

Records the objects in *snapshots*. *snapshots* should be an NSDictionary of NSDictionaries, in which the top-level dictionary has as its key the globaID of the enterprise object for which to-many relationships are being recorded. The key's value is a dictionary whose keys are the names of the enterprise object's to-many relationships. Each of these keys in turn has as its value an array of globalIDs that identify the objects at the destination of the relationship. For a description of snapshots and their role in an application, see the class description.

See also: – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **recordSnapshot:forGlobalID:**, – **snapshotForSourceGlobalID:relationshipName:**

registerContext:

– (void)**registerContext:**(EODatabaseContext *)*aContext*

Records *aContext* as one of the receiver's EODatabaseContexts, without retaining it. . *aContext* must have been created with the receiver using EODatabaseContext's **initWithDatabase:** method, which invokes this method automatically. You should never need to invoke this method directly.

See also: – **unregisterContext:**, – **registeredContexts**

registeredContexts

– (NSArray *)**registeredContexts**

Returns all the EODatabaseContexts that have been registered with the receiver, generally all the database contexts that were created with the receiver as their EODatabase object.

See also: – **registerContext:**, – **unregisterContext:**

removeModel:

– (void)**removeModel:**(EOModel *)*aModel*

Removes *aModel* from the receiver's list of EOModels. Raises an exception if *aModel* isn't one of the receiver's models.

See also: – **addModel:**, – **addModelIfCompatible:**, – **models**

resultCacheForEntityNamed:

– (NSArray *)**resultCacheForEntityNamed:**(NSString *)*entityName*

Returns an array containing the globalIDs of the objects associated with *entityName*. See the class description for more discussion of this topic.

See also: – **invalidateResultCache**, – **invalidateResultCacheForEntityNamed:**

setResultCache:forEntityNamed:

– (void)**setResultCache:**(NSArray *)*cache* **forEntityNamed:**(NSString *)*entityName*

Updates the receiver's cache for *entityName* with *cache*, an array of EOGlobalID objects, for all the enterprise objects associated with the EOEntity named *entityName*. This method is invoked automatically, and you should never need to invoke it directly. For more information on this topic, see the class description.

See also: – **invalidateResultCache**, – **invalidateResultCacheForEntityNamed:**,
– **resultCacheForEntityNamed:**

snapshotForGlobalID:

– (NSDictionary *)**snapshotForGlobalID:**(EOGlobalID *)*globalID*

Returns the snapshot associated with *globalID* if there is one; otherwise returns **nil**. For a description of snapshots and their role in an application, see the class description.

See also: – **recordSnapshot:forGlobalID:**, – **forgetSnapshotForGlobalID:**

snapshotForSourceGlobalID:relationshipName:

– (NSArray *)**snapshotForSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*

Returns a snapshot that consists of an array of globalIDs. These globalIDs identify the objects at the destination of the to-many relationship named *name*, which is a property of the object identified by

globalID. If there is no snapshot, returns **nil**. For a description of snapshots and their role in an application, see the class description.

snapshots

– (NSDictionary *)**snapshots**

Returns all of the receiver's snapshots, stored in a dictionary under their EOGlobalIDs.

See also: – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **recordToManySnapshots:**

unregisterContext:

– (void)**unregisterContext:**(EODatabaseContext *)*aContext*

Removes *aContext* as one of the receiver's EODatabaseContexts, without releasing it. An EODatabaseContext automatically invokes this method when deallocated; you should never need to invoke it directly.

See also: – **registerContext:**, – **registeredContexts**

EODatabase

Each of an EODatabase's EODatabaseContexts forms a separate transaction scope, and is in effect a separate logical user to the server. An EODatabaseContext uses one or more pairs of EODatabaseChannel and EOAdaptorChannel objects to manage data operations (insert, update, delete, and fetch). Adaptors may support a limited number of contexts per database or channels per context, but an application is guaranteed at least one of each.

The EODatabase, EODatabaseContext, and EODatabaseChannel classes form the *database level* of the Enterprise Objects Framework. The database level is a client of the *adaptor level*, which is defined by the adaptor classes: EOAdaptor, EOAdaptorContext, and EOAdaptorChannel. Together, the database and adaptor levels make up the *access layer* of the Enterprise Objects Framework.

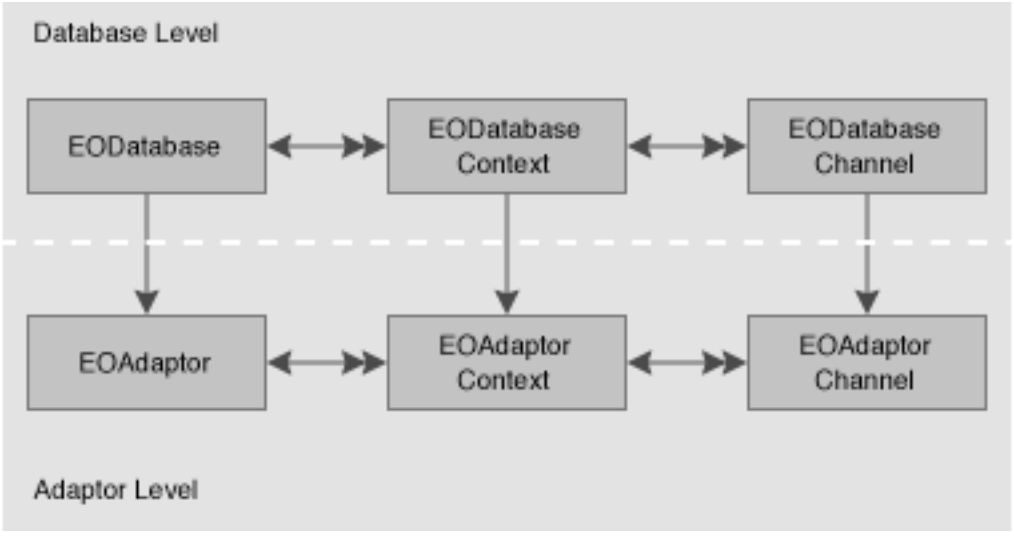


Figure 2 The Access Layer

The database level acts as an intermediary between the adaptor level and the *control layer*, which includes an EOObjectStoreCoordinator and an EOEditingContext (Figure 3). The control layer operates in terms of enterprise objects, while the adaptor level operates in terms of database rows packaged as NSDictionaries. It's the job of the database level to perform the necessary object-to-relational translation between the two.

There's little need for your code to interact directly with an EODatabase object. An EOEditingContext creates its own database level objects, which create their own corresponding adaptor level objects. Once the network of objects is in place, your code might interact with an EODatabase to access its corresponding EOAdaptor object, but additional programmatic interaction is usually unnecessary.

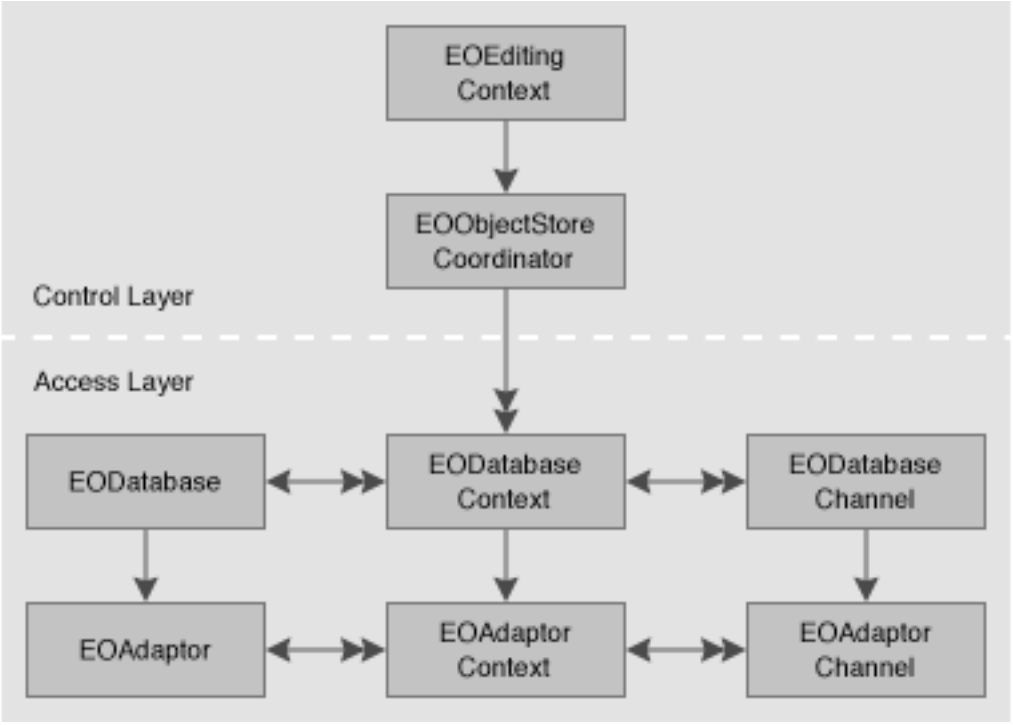


Figure 3 The EODatabase Level as an Intermediary Between the Adaptor Level and the Control Layer

Snapshots

EODatabase's most significant responsibility is to record *snapshots* for its EODatabaseContexts. A snapshot is a dictionary whose keys are attribute names and whose values are the corresponding, last-known database values. Enterprise Objects Framework records snapshots as it successfully fetches, inserts and updates enterprise objects. Snapshot information is used when changes to enterprise objects are saved back out to the database to ensure that row data has not been changed by someone else since it was last recorded by the application.

A snapshot contains entries for a row's primary key, class properties, foreign keys for class property relationships, and attributes used for locking. They are recorded under the globalIDs of their enterprise objects. (EOGlobalIDs are based on an object's primary key and its associated entity; see the class specification for EOGlobalID in the EOControl framework for more information.)

EODatabase also records snapshots for to-many relationships. These snapshots consist of an NSDictionary of NSDictionaries, in which the top-level dictionary has as its key the globaID of the enterprise object for which to-many relationships are being recorded. The key's value is a dictionary whose keys are the names of the enterprise object's to-many relationships. Each of these keys in turn has as its value an array of globalIDs that identify the objects at the destination of the relationship.

The snapshots made by an EODatabase form the global view of data for nearly every other part of the application, representing the current view of data in the server as far as the application is concerned (though other applications may have made changes). This global view is temporarily overridden locally by EODatabaseContexts, which form their own snapshots as they make changes during a transaction. When an EODatabaseContext commits its top-level transaction, it reconciles all changed snapshots with the global view of the database object, so that other database contexts (except those with open transactions) immediately use the new snapshots as well. EODatabaseContexts automatically use their EODatabase to record snapshots, so there's no need for your application to intervene in an EODatabase's snapshotting mechanism.

For more information on snapshots and how they relate to an application's update strategy, see the EODatabaseContext class specification.

Result Cache

An EODatabase object also performs the function of caching enterprise objects for entities that cache their objects (see the EOEntity class specification). An EODatabase's result cache stores the globalIDs of enterprise objects for entities that cache their objects. The first time you perform a fetch against such an entity, all of its objects are fetched, regardless of the fetch specification used. The globalIDs of the resulting objects are stored in the EODatabase's result cache by entity name. Whenever possible, subsequent fetches are performed against the cache (in memory) rather than against the database. With a globalID, Enterprise Objects Framework can look up the values for the corresponding object in its snapshot using EODatabase's or EODatabaseContext's **snapshotForGlobalID:** method.

As an example, suppose that you have an entity named Rating that contains all the valid ratings for Movies (G, PG, R, and so on). Rather than store a Movie's rating directly in the Movie as an attribute, Movie maintains a relationship to a Rating. To specify a rating for a movie, users select the rating from a pop-up list of the possible values. This Rating entity should cache its objects. The values that populate the rating pop-up list are only fetched once, and there's no need to fetch them again; the relationships between Movies and Ratings can be maintained without subsequent fetches.

The result cache is managed automatically; you shouldn't have to manipulate it explicitly. However, if you need to access or alter the cache, EODatabase provides several methods for interacting with it.

EODatabaseChannel

Inherits From: NSObject

Declared In: EOAccess/EODatabaseChannel.h

Class Description

An EODatabaseChannel represents an independent communication channel to the database server. It's associated with an EODatabaseContext and an EODatabase, which, together with the EODatabaseChannel, form the *database level* of Enterprise Objects Framework's access layer. See the EODatabase class specification for more information.

An EODatabaseChannel has an EOAdaptorChannel that it uses to connect to the database server its EODatabase object represents. An EODatabaseChannel fetches database records as instances of enterprise object classes that are specified in its EODatabase's EOModel objects. An EODatabaseChannel also has an EODatabaseContext, which uses the channel to perform fetches and to lock rows in the database. All of the database level objects are used automatically by EOEditingContexts and other components of Enterprise Objects Framework. You rarely need to interact with them directly. In particular, you wouldn't ordinarily use an EODatabaseChannel to fetch objects. Rather, you'd use an EOEditingContext.

Method Types

Creating instances	– initWithDatabaseContext:
Accessing cooperating objects	– adaptorChannel – databaseContext
Fetching objects	– selectObjectsWithFetchSpecification:editingContext: – isFetchInProgress – fetchObject – cancelFetch
Accessing internal fetch state	– setCurrentEntity: – setCurrentEditingContext: – setIsLocking: – isLocking – setIsRefreshingObjects: – isRefreshingObjects
Accessing the delegate	– setDelegate: – delegate

Instance Methods

adaptorChannel

– (EOAdaptorChannel *)**adaptorChannel**

Returns the EOAdaptorChannel used by the receiver for communication with the database server.

cancelFetch

– (void)**cancelFetch**

Cancels any fetch in progress.

See also: – **isFetchInProgress**, – **selectObjectsWithFetchSpecification:editingContext:**, – **fetchObject**

databaseContext

– (EODatabaseContext *)**databaseContext**

Returns the EODatabaseContext that controls transactions for the receiver.

delegate

– (id)**delegate**

Returns the receiver's delegate. An EODatabaseChannel shares the delegate of its EODatabaseContext. See the EODatabaseContext class specification for the delegate methods you can implement.

See also: – **setDelegate:**

fetchObject

– (id)**fetchObject**

Fetches and returns the next object in the result set produced by a **selectObjectsWithFetchSpecification: editingContext:** message; returns **nil** if there are no more objects in the current result set or if an error occurs. This method uses the receiver's EOAdaptorChannel to fetch a row, records a snapshot with the EODatabaseContext if necessary, and creates an enterprise object from the row if a corresponding object doesn't already exist. The new object is sent an **awakeFromFetchInEditingContext:** message to allow it to finish setting up its state.

If no snapshot exists for the fetched object, the receiver sends its EODatabase a **recordSnapshot: forGlobalID:** message to record one. If a snapshot already exists (because the object was previously fetched), the receiver checks whether it should overwrite the old snapshot with the new one. It does so by

asking the delegate with a **databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID: databaseChannel:** method. If the delegate doesn't respond to this method, the EODatabaseChannel overwrites the snapshot if it's locking or refreshing fetched objects. Further, if the EODatabaseChannel is refreshing fetched objects, it posts an EOObjectsChangedInStoreNotification on behalf of its EODatabaseContext (which causes any EOEditingContext using that EODatabaseContext to update its enterprise object with the values recorded in the new snapshot).

For information on locking and update strategies, see the EODatabaseContext class specification. For information on refreshing fetched objects, see the EOFetchSpecification class specification.

Ordinarily, you don't directly use an EODatabaseChannel to fetch objects. Rather, you use an EOEditingContext, which uses an underlying EODatabaseChannel to do its work.

See also: – **cancelFetch**, – **isFetchInProgress**, – **isLocking**, – **isRefreshingObjects**

initWithDatabaseContext:

– **initWithDatabaseContext:**(EODatabaseContext *)*aDatabaseContext*

The designated initializer, this method initializes a newly allocated EODatabaseChannel with *aDatabaseContext* as the EODatabaseContext in which it works. The new EODatabaseChannel retains *aDatabaseContext*, and creates an EOAdaptorChannel to communicate with the database server. Returns **self**. Raises if the underlying adaptor context can't create a corresponding adaptor channel.

Typically, you don't need to programmatically create EODatabaseChannel objects. Rather, they are created automatically by the control layer. See the EODatabase class description for more information.

isFetchInProgress

– (BOOL)**isFetchInProgress**

Returns YES if the receiver is fetching, NO otherwise. An EODatabaseChannel is fetching if it's been sent a successful **selectObjectsWithFetchSpecification:editingContext:** message. An EODatabaseChannel stops fetching when there are no more objects to fetch or when it is sent a **cancelFetch** message.

isLocking

– (BOOL)**isLocking**

Returns YES if the receiver is locking the objects selected, as determined by its EODatabaseContext's update strategy or the EOFetchSpecification used to perform the select. Returns NO otherwise. This method always returns NO when no fetch is in progress.

See also: – **locksObjects** (EOFetchSpecification), – **setIsLocking:**

isRefreshingObjects

– (BOOL)**isRefreshingObjects**

Returns YES if the receiver overwrites existing snapshots with fetched values and causes the current EOEditingContext to overwrite existing enterprise objects with those values as well. Returns NO otherwise. This behavior is controlled by the EOFetchSpecification used in a **selectObjectsWithFetchSpecification: editingContext:** message.

See also: – **refreshesRefetchedObjects** (EOFetchSpecification), – **fetchObject**, – **setIsRefreshingObjects:**

selectObjectsWithFetchSpecification:editingContext:

– (void)**selectObjectsWithFetchSpecification:**(EOFetchSpecification *)*fetchSpecification*
editingContext:(EOEditingContext *)*anEditingContext*

Selects objects described by *fetchSpecification* so that they'll be fetched into *anEditingContext*. The selected objects compose one or more result sets, each object of which will be returned by subsequent **fetchObject** messages in the order prescribed by *fetchSpecification*'s EOSortOrderings.

Raises an exception if an error occurs; the particular exception depends on the specific error, and is indicated in the exception's description. Some possible reasons for failure are:

- *fetchSpecification* is invalid.
- The receiver's EODatabaseContext has no transaction in progress.
- The delegate disallows the select operation.
- The receiver's EOAdaptorChannel fails to perform the select operation.

This method invokes the delegate methods **databaseContext: shouldSelectObjectsWithFetchSpecification:databaseChannel:**, **databaseContext: shouldUsePessimisticLockWithFetchSpecification: databaseChannel:**, and **databaseContext: didSelectObjectsWithFetchSpecification:databaseChannel:**. See their descriptions in the EODatabaseContext class specification for more information.

You wouldn't ordinarily invoke this method directly; rather, you'd use an EOEditingContext to select and fetch enterprise objects.

See also: – **fetchObject**

setCurrentEditingContext:

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

Sets the EOEditingContext that's made the owner of fetched objects to *anEditingContext*. This method is automatically invoked by **selectObjectsWithFetchSpecification:editingContext:**. You should never invoke it directly.

See also: – **setCurrentEntity:**

setCurrentEntity:

– (void)**setCurrentEntity:**(EOEntity *)*anEntity*

Sets the EOEntity used when fetching enterprise objects to *anEntity*. Subsequent **fetchObject** messages during a fetch operation create an object of the class associated with *anEntity*. This method is invoked automatically by **selectObjectsWithFetchSpecification:editingContext:**. You should never need to invoke it directly.

See also: – **setCurrentEditingContext:**

setDelegate:

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

Sets the receiver's delegate to *anObject*. An EODatabaseChannel shares the delegate of its EODatabaseContext; you should never invoke this method directly. See the EODatabaseContext class specification for the delegate methods you can implement.

See also: **delegate**

setIsLocking:

– (void)**setIsLocking:**(BOOL)*flag*

Records whether the receiver locks the records it selects. A EODatabaseChannel modifies its interaction with the database server and its snapshotting behavior based on this setting. If *flag* is YES the EODatabaseChannel modifies its fetching behavior to lock objects; if *flag* is NO it simply fetches them.

An EODatabaseChannel automatically sets this flag according to the fetch specification used in a **selectObjectsWithFetchSpecification:editingContext:** message. You might invoke this method directly if evaluating SQL directly with EOAdaptorChannel's method.

See also: – **locksObjects** (EOFetchSpecification), – **setIsLocking:**

setIsRefreshingObjects:

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

Records whether the receiver causes existing snapshots and enterprise objects to be overwritten with fetched values. If *flag* is YES the receiver overwrites existing snapshots with fetched values and posts an EOObjectsChangedInStoreNotification on behalf of its EODatabaseContext (which typically causes the an existing object's EOEditingContext to replace its values with the new ones). If *flag* is NO, the receiver relies on the delegate to determine whether snapshots should be overwritten, and doesn't cause enterprise objects to be overwritten.

An EODatabaseChannel automatically sets this flag according to the fetch specification used in a **selectObjectsWithFetchSpecification:editingContext:** message. You might invoke this method directly if evaluating SQL directly with EOAdaptorChannel's **evaluateExpression:** method.

See also: – **refreshesRefetchedObjects** (EOFetchSpecification)

EODatabaseContext

Inherits From: EOCooperatingObjectStore : EOObjectStore : NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EODatabaseContext.h

Class Description

An EODatabaseContext object is an EOObjectStore for accessing relational databases, creating and saving objects based on EOEntity definitions in an EOModel.

An EODatabaseContext represents a single connection to a database server, and it determines the updating and locking strategy used by its EODatabaseChannel objects. An EODatabaseContext has a corresponding EODatabase object. If the server supports multiple concurrent transactions, the EODatabase object may have several database contexts. If the server and adaptor support it, a database context may in turn have several database channels, which handle access to the data on the server.

For a more information, see “EODatabaseContext”.

Method Types

Initializing instances
- initWithDatabase:

Fetching objects
- objectsWithFetchSpecification:editingContext:
- objectsForSourceGlobalID:relationshipName:editingContext:
- arrayFaultWithSourceGlobalID:relationshipName:editingContext:
- batchFetchRelationship:forSourceObjects:editingContext:

Accessing the adaptor context
- adaptorContext

Accessing the database object
- database

Accessing the coordinator
- coordinator

Managing channels

– availableChannel
– registerChannel:
– registeredChannels
– unregisterChannel:

Accessing the delegate

– setDelegate:
– delegate

Committing or discarding changes

– invalidateAllObjects
– invalidateObjectsWithGlobalIDs:
– rollbackChanges
– saveChangesInEditingContext:
– commitChanges
– performChanges
– prepareForSaveWithCoordinator:editingContext:
– recordUpdateForObject:changes:
– recordChangesInEditingContext
– refaultObject:withGlobalID:editingContext:

Determining if the EODatabaseContext is responsible for a particular operation

– ownsObject:
– ownsGlobalID:
– handlesFetchSpecification:

Managing Snapshots

– forgetSnapshotForGlobalID:
– forgetSnapshotsForGlobalIDs:
– localSnapshotForGlobalID:
– recordSnapshot:forGlobalID:
– recordSnapshots:
– snapshotForGlobalID:
– recordSnapshot:forSourceGlobalID:relationshipName:
– snapshotForSourceGlobalID:relationshipName:
– localSnapshotForSourceGlobalID:relationshipName:
– recordToManySnapshots:

Initializing objects

– initializeObject:withGlobalID:editingContext:

Obtaining an EODatabaseContext

+ registeredDatabaseContextForModel:editingContext:

Locking objects
– setUpdateStrategy:
– updateStrategy
– registerLockedObjectWithGlobalID:
– isObjectLockedWithGlobalID:
– isObjectLockedWithGlobalID:editingContext:
– forgetAllLocks
– forgetLocksForObjectsWithGlobalIDs:
– lockObjectWithGlobalID:editingContext:

Returning information about objects
– valuesForKeys:object:

Setting the context class
+ contextClassToRegister
+ setContextClassToRegister:

Checking connection status
– hasBusyChannels

Other
+ forceConnectionWithModel:connectionDictionaryOverrides:
editingContext:
– lock
– unlock

Class Methods

contextClassToRegister

+ (Class)**contextClassToRegister**

Returns the class that is registered with an EOObjectStoreCoordinator when the coordinator broadcasts an EOCooperatingObjectStoreNeeded notification. By default this is EODatabaseContext, but you can use **setContextClassToRegister:** to specify your own subclass of EODatabaseContext.

When an EOObjectStoreCoordinator sends an EOCooperatingObjectStoreNeeded notification for an EOEntity in the default model group, if **contextClassToRegister** is non-**nil** (and it should be—it makes no sense to set **contextClassToRegister** to **nil**), an instance of the that class is created, the EOModel for the EOEntity is registered, and the context class is registered with the requesting EOObjectStoreCoordinator.

forceConnectionWithModel:connectionDictionaryOverrides:editingContext:

+ (EODatabaseContext *)**forceConnectionWithModel:**(EOModel *)*amodel*
  connectionDictionaryOverrides:(NSDictionary *)*overrides*
  editingContext:(EOEditingContext *)*anEditingContext*

Forces the stack of objects in the EOAccess layer to be instantiated, if necessary, and then makes a connection to the database. If there is an existing connection for *amodel*, it is first closed and then reconnected. The new connection dictionary is effectively made up of the model's connection dictionary, overlaid with *overrides*. All compatible models in the model's group also are associated with the new connection (so they share the same adaptor). Returns the EODatabaseContext associated with the model for *anEditingContext*.

registeredDatabaseContextForModel:editingContext:

+ (EODatabaseContext *)**registeredDatabaseContextForModel:**(EOModel *)*aModel*
  editingContext:(EOEditingContext *)*anEditingContext*

Finds the EOObjectStoreCoordinator for *anEditingContext* and checks to see if it already contains an EODatabaseContext cooperating store for *aModel*. If it does, it returns that EODatabaseContext. Otherwise it instantiates a new EODatabaseContext, adds it to the EOObjectStoreCoordinator, and returns the EODatabaseContext.

setContextClassToRegister:

+ (void)**setContextClassToRegister:**(Class)*contextClass*

Sets to *contextClass* the "contextClassToRegister." For more discussion of this topic, see the method description for **contextClassToRegister**.

Instance Methods

adaptorContext

– (EOAdaptorContext *)**adaptorContext**

Returns the EOAdaptorContext used by the EODatabaseContext for communication with the database server.

arrayFaultWithSourceGlobalID:relationshipName:editingContext:

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

Overrides the inherited implementation to create a to-many fault for *anEditingContext*. *name* must correspond to an EORelationship in the EOEntity for the specified *globalID*.

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

availableChannel

– (EODatabaseChannel *)**availableChannel**

Returns an EODatabaseChannel that's registered with the receiver and that isn't busy. If the method can't find a channel that meets these criteria, it posts an EODatabaseChannelNeededNotification in the hopes that someone will provide a new channel. After posting the notification, the receiver checks its list of channels again. If there are still no available channels, the receiver creates an EODatabaseChannel itself. However, if the list is not empty and there are no available channels, the method returns **nil**.

See also: – **registerChannel:**, – **registeredChannels**, – **unregisterChannel:**

batchFetchRelationship:forSourceObjects:editingContext:

– (void)**batchFetchRelationship:**(EORelationship *)*relationship*
forSourceObjects:(NSArray *)*objects*
editingContext:(EOEditingContext *)*anEditingContext*

Clear all the faults for the *relationship* of *anEditingContext*'s *objects* and performs a single, efficient, fetch (at most two fetches, if the relationship is many-to-many). This method provides a way to fetch the same relationship for multiple objects. For example, given an array of Employee objects, this method can fetch all of their departments with one round trip to the server, rather than asking the server for each of the employee's departments individually.

commitChanges

– (void)**commitChanges**

Overrides the inherited implementation to instruct the adaptor to commit the transaction. If the commit is successful, any primary and foreign key changes are written back to the saved objects, database locks are released, and an EOObjectsChangedInStoreNotification (defined in EOObjectStore) is posted describing

the committed changes. Raises an exception if the adaptor is unable to commit the transaction; the error message indicates the nature of the problem. You should never need to invoke this method directly.

See also: – **performChanges**, – **rollbackChanges**

coordinator

– (EOObjectStoreCoordinator *)**coordinator**

Returns the receiver's EOObjectStoreCoordinator or **nil** if there is none. This method is only valid during a save operation.

database

– (EODatabase *)**database**

Returns the receiver's EODatabase.

See also: – **initWithDatabase:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: **– setDelegate:**

faultForGlobalID:editingContext:

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

Overrides the inherited implementation to create a to-one fault for the object identified by *globalID* and register it in *anEditingContext.*

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

faultForRawRow:entityNamed:editingContext:

– (id <EOEnterpriseObject>)**faultForRawRow:**(id)*row* **entityNamed:**(NSString *)*entityName* **editingContext:**(EOEditingContext *)*context*

Returns a fault for a raw row. *row* is the raw data, typically in the form of an NSDictionary. *entityName* is the name of the appropriate entity for the EO you want to create (as a fault). *editingContext* is the EOEditingContext in which to create the fault

forgetAllLocks

– (void)**forgetAllLocks**

Clears all of the receiver's locks. Doesn't cause the locks to be forgotten in the server, only in the receiver. This method is useful when something has happened to cause the server to forget the locks and the receiver needs to be synced up. This method is invoked whenever a transaction is committed or rolled back.

See also: – **registerLockedObjectWithGlobalID:**, – **isObjectLockedWithGlobalID:**, – **isObjectLockedWithGlobalID:editingContext:**, – **forgetLocksForObjectsWithGlobalIDs:**, – **lockObjectWithGlobalID:editingContext:**, – **lockObject:** (EOEditingContext)

forgetLocksForObjectsWithGlobalIDs:

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

Clears the locks made for the enterprise objects identified by each of the EOGlobalIDs in *globalIDs*. Doesn't cause the locks to be forgotten in the server, only in the receiver.

See also: – **registerLockedObjectWithGlobalID:**, – **isObjectLockedWithGlobalID:**, – **isObjectLockedWithGlobalID:editingContext:**, – **forgetAllLocks**, – **lockObjectWithGlobalID:editingContext:**, – **lockObject:** (EOEditingContext)

forgetSnapshotForGlobalID:

– (void)**forgetSnapshotForGlobalID:**(EOGlobalID *)*globalID*

Deletes the snapshot made for the enterprise object identified by *globalID*.

See also: – **recordSnapshot:forGlobalID:**, – **localSnapshotForGlobalID:**, – **recordSnapshots:**, – **snapshotForGlobalID:**, – **forgetSnapshotsForGlobalIDs:**

forgetSnapshotsForGlobalIDs:

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

Deletes the snapshots made for the enterprise objects identified by *globalIDs*, an array of EOGlobalID objects.

See also: – **recordSnapshot:forGlobalID:**, – **localSnapshotForGlobalID:**, – **recordSnapshots:**, – **snapshotForGlobalID:**

handlesFetchSpecification:

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

Overrides the inherited implementation to return YES if the receiver is responsible for fetching the objects described by the entity name in *fetchSpecification*.

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

hasBusyChannels

– (BOOL)**hasBusyChannels**

Returns YES if the receiver's EOAdaptorContext has channels that have outstanding operations (that is, have a fetch in progress), NO otherwise.

initializeObject:withGlobalID:editingContext:

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

Overrides the inherited implementation initialize *object* for *anEditingContext* by filling it with properties based on row data fetched from the adaptor. The snapshot for *globalID* is looked up and those attributes in the snapshot that are marked as class properties in the EOEntity are assigned to *object*. For relationship class properties, faults are constructed and assigned to the object.

initWithDatabase:

– **initWithDatabase:**(EODatabase *)*aDatabase*

Initializes a newly allocated EODatabaseContext with *aDatabase* as the EODatabase object it works with. The new EODatabaseContext retains *aDatabase*. Returns **self**, or **nil** if unable to create another EOAdaptorContext for the EOAdaptor of *aDatabase*. This is the designated initializer for the EODatabaseContext class.

invalidateAllObjects

– (void)**invalidateAllObjects**

Overrides the inherited implementation to discard all snapshots in the receiver's EODatabase, forget all locks, and post an EOInvalidatedAllObjectsInStoreNotification, as well as an EOObjectsChangedInStoreNotification with the invalidated global IDs in the **userInfo** dictionary. Both of these notifications are defined in EOObjectStore. This method works by invoking **– invalidateObjectsWithGlobalIDs:** for all of the snapshots in the receiver's EODatabase.

invalidateObjectsWithGlobalIDs:

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

Overrides the inherited implementation to discard the snapshots for the objects identified by the EOGlobalIDs in *globalIDs* and broadcasts an EOObjectsChangedInStoreNotification (defined in EOObjectStore), which causes the EOEditingContext containing objects fetched from the receiver to refault those objects. The result is that these objects will be refetched from the database the next time they're accessed.

isObjectLockedWithGlobalID:

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

Returns YES if the enterprise object identified by *globalID* is locked, NO otherwise.

See also: – **registerLockedObjectWithGlobalID:**, – **forgetAllLocks**, **– isObjectLockedWithGlobalID: editingContext:**, – **forgetLocksForObjectsWithGlobalIDs:**, – **lockObjectWithGlobalID: editingContext:**, – **lockObject:** (EOEditingContext)

isObjectLockedWithGlobalID:editingContext:

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

Overrides the EOObjectStore method **isObjectLockedWithGlobalID:editingContext:** to return YES if the database row corresponding to *globalID* has been locked in an open transaction held by the receiver.

See also: – **registerLockedObjectWithGlobalID:**, – **isObjectLockedWithGlobalID:**, – **forgetAllLocks**, – **forgetLocksForObjectsWithGlobalIDs:**, – **lockObjectWithGlobalID:editingContext:**, – **lockObject:** (EOEditingContext)

localSnapshotForGlobalID:

– (NSDictionary *)**localSnapshotForGlobalID:**(EOGlobalID *)*globalID*

Returns the snapshot for the object identified by *globalID*, if there is one; else returns **nil**. Only searches locally (in the transaction scope), not in the EODatabase.

See also: – **recordSnapshot:forGlobalID:**, – **forgetSnapshotForGlobalID:**, – **recordSnapshots:**, – **snapshotForGlobalID:**

localSnapshotForSourceGlobalID:relationshipName:

– (NSArray *)**localSnapshotForSourceGlobalID:**(EOGlobalID *)*globalID* **relationshipName:** (NSString *)*name*

Returns an array that is the snapshot for the objects at the destination of the to-many relationship named *name*, which is a property of the object identified by *globalID*. The returned array contains the globalIDs of the destination objects. If there is no snapshot, returns **nil**. Only searches locally (in the transaction scope), not in the EODatabase.

See also: – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **snapshotForSourceGlobalID: relationshipName:**

lock

– (void)**lock**

Used internally to protect access to the receiver in a multi-threaded environment. Do not confuse this with any methods which work with the database locking mechanism.

See also: – **unlock**

lockObjectWithGlobalID:editingContext:

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

Overrides the inherited implementation to attempt to lock the database row corresponding to *globalID* in the underlying database server, on behalf of *anEditingContext*. If a transaction is not already open at the time of the lock request, the transaction is begun and is held open until either **commitChanges** or **invalidateAllObjects** is invoked. At that point all locks are released. Raises an NSInternalInconsistencyException if unable to obtain the lock.

See also: – **registerLockedObjectWithGlobalID:**, – **isObjectLockedWithGlobalID:**, – **forgetAllLocks**, – **forgetLocksForObjectsWithGlobalIDs:**, – **lockObject:** (EOEditingContext)

objectsForSourceGlobalID:relationshipName:editingContext:

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

Overrides the inherited implementation to service a to-many fault. The snapshot for the source object identified by *globalID* is located and the EORelationship named *name* is used to construct a qualifier from that snapshot. This qualifier is then used to fetch the requested objects into *anEditingContext* using the method **objectsWithFetchSpecification:editingContext:**.

objectsWithFetchSpecification:editingContext:

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

Overrides the inherited implementation to fetch objects from an external store into *anEditingContext*. The receiver obtains an available EODatabaseChannel and issues a fetch with *fetchSpecification*. If one of these objects is already present in memory, by default this method doesn't overwrite its values with the new values from the database (you can change this behavior; see the **setRefreshesRefetchedObjects:** method in the EOFetchSpecification class specification).

You can fine-tune the fetching behavior by adding hints to *fetchSpecification*'s **hints** dictionary. For this purpose, Enterprise Objects Framework defines the following keys (NSStrings):

Constant	Corresponding value in the hints dictionary
EOCustomQueryExpressionHintKey	An NSString specifying raw SQL with which to perform the fetch. There is no way to pass down parameters with this hint.
EOStoredProcedureNameHintKey	An NSString specifying a name for a stored procedure in the model that should be used rather than building the SQL statement. The stored procedure must query the the exact same attributes in the same order as EOF would query if generating the SELECT expression dynamically. If this key is supplied, other aspects of the EOFetchSpecification such as **isDeep**, **qualifier**, and **sortOrderings** are ignored (in that sense, this key is more of a directive than a hint). There is no way to pass down parameters with this hint.

The class description contains additional information on using these hints. See "Using a Custom Query."

You can also use this method to implement "on-demand" locking by using a *fetchSpecification* that includes locking. For more discussion of this subject, see "Updating And Locking Strategies" in the class description.

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

See also: – **objectsWithFetchSpecification:** (EOEditingContext)

ownsGlobalID:

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

Overrides the inherited implementation to return YES if the receiver is responsible for fetching and saving the object identified by *globalID*, NO otherwise. The receiver is determined to be responsible if *globalID* is a subclass of EOKeyGlobalID and *globalID* has an entity from one of the receiver's EODatabase's EOModels.

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

ownsObject:

– (BOOL)**ownsObject:**(id)*object*

Overrides the inherited implementation to return YES if the receiver is responsible for fetching and saving *object*, NO otherwise. The receiver is determined to be responsible if the entity corresponding to *object* is in one of the receiver's EODatabase's EOModels.

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

performChanges

– (void)**performChanges**

Overrides the inherited implementation to construct EOAdaptorOperations from the EODatabaseOperations produced during **recordChangesInEditingContext** and **recordUpdateForObject:changes:**. Invokes the delegate method **databaseContext: willOrderAdaptorOperationsFromDatabaseOperations:** to give the delegate an opportunity to construct alternative EOAdaptorOperations from the EODatabaseOperations. Then invokes the delegate method **databaseContext:willPerformAdaptorOperations:adaptorChannel:** to let the delegate substitute its own array of EOAdaptorOperations. Gives the EOAdaptorOperations to an available EOAdaptorChannel for execution. If the save succeeds, updates the snapshots in the receiver to reflect the new state of the server. You should never need to invoke this method directly.

This method raises an exception if the adaptor is unable to perform the operations. The exception's userInfo dictionary contains these keys:

- EODatabaseContextKey

  The EODatabaseContext object that was trying to save to its underlying repository when the exception was raised.

- EODatabaseOperationsKey

  The list of database operations the EODatabaseContext was trying to perform when the failure occurred.

- EOFailedDatabaseOperationKey

  The database operation the EODatabaseContext failed to perform.

The userInfo dictionary may also contain some of the keys listed in the method description for the EOAdaptorChannel method **performAdaptorOperation:**. For more information, see the EOAdaptorChannel class specification.

See also: – **commitChanges**, – **rollbackChanges**

prepareForSaveWithCoordinator:editingContext:

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

Overrides the inherited implementation to do whatever is necessary to prepare to save changes. If needed, generates primary keys for any new objects in *anEditingContext* that are owned by the receiver. This method is invoked before the object graph is analyzed and foreign key assignments are performed. You should never need to invoke this method directly.

recordChangesInEditingContext

– (void)**recordChangesInEditingContext**

Overrides the inherited implementation to construct a list of EODatabaseOperations for all changes to objects in the EOEditingContext that are owned by the receiver. Forwards any relationship changes discovered but not owned by the receiver to the EOObjectStoreCoordinator. This method is typically invoked in the course of an EOObjectStoreCoordinator saving changes through its **saveChangesInEditingContext:** method. It's invoked after **prepareForSaveWithCoordinator: editingContext:** and before **performChanges**. You should never need to invoke this method directly.

recordSnapshot:forGlobalID:

– (void)**recordSnapshot:**(NSDictionary *)*snapshot* **forGlobalID:**(EOGlobalID *)*globalID*

Records *aSnapshot* under *globalID*. This method only records snapshots locally (in the transaction scope). If you want to record snapshots globally, use the corresponding EODatabase method.

See also: – **forgetSnapshotForGlobalID:**, – **localSnapshotForGlobalID:**, – **recordSnapshots:**, – **snapshotForGlobalID:**

recordSnapshot:forSourceGlobalID:relationshipName:

– (void)**recordSnapshot:**(NSArray *)*globalIDs* **forSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*

For the object identified by *globalID*, records an NSArray of *globalIDs* for the to-many relationship named *name*. These *globalIDs* identify the objects at the destination of the relationship. This method only records snapshots locally (in the transaction scope). If you want to record snapshots globally, use the corresponding EODatabase method.

See also: – **snapshotForSourceGlobalID:relationshipName:**, – **localSnapshotForSourceGlobalID:relationshipName:**, – **recordToManySnapshots:**

recordSnapshots:

– (void)**recordSnapshots:**(NSDictionary *)*snapshots*

Records the objects in *snapshots*, a dictionary of snapshots. The *snapshots; keys* are GlobalIDs and its values are the corresponding snapshots represented as NSDicationaries. This method only records snapshots locally (in the transaction scope). If you want to record snapshots globally, use the corresponding EODatabase method.

See also: – **recordSnapshot:forGlobalID:**, – **localSnapshotForGlobalID:**, – **forgetSnapshotForGlobalID:**, – **snapshotForGlobalID:**

recordToManySnapshots:

– (void)**recordToManySnapshots:**(NSDictionary *)*snapshots*

Records the objects in *snapshots*. *snapshots* should be an NSDictionary of NSDictionaries, in which the top-level dictionary has as its key the globaID of the enterprise object for which to-many relationships are being recorded. The key's value is a dictionary whose keys are the names of the Enterprise Object's to-many relationships. Each of these keys in turn has as its value an array of globalIDs that identify the objects at the destination of the relationship.

This method only records snapshots locally (in the transaction scope). If you want to record snapshots globally, use the corresponding EODatabase method.

See also: – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **snapshotForSourceGlobalID:relationshipName:**, – **localSnapshotForSourceGlobalID:relationshipName:**

recordUpdateForObject:changes:

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

Overrides the inherited implementation to communicate to the receiver that *changes* from another EOCooperatingObjectStore (through the EOObjectStoreCoordinator) need to be made to an *object* in the receiver. For example, an insert of an object in a relationship property might require changing a foreign key property in an object owned by another cooperating store. This method can be invoked any time after **prepareForSaveWithCoordinator:editingContext:** and before **performChanges**.

refaultObject:withGlobalID:editingContext:

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

Overrides the inherited implementation to refault the enterprise object object identified by *globalID* in *anEditingContext*. Newly-inserted objects should not be refaulted, since they can't be refetched from the external store. If you attempt to do this, an exception will be raised. Don't refault to-many relationship arrays, just recreate them.

This method should be used with caution since refaulting an object doesn't remove the object snapshot from the undo stack, after which the object snapshot may not refer to the proper object..

registerChannel:

– (void)**registerChannel:**(EODatabaseChannel *)*channel*

Registers *channel*, which means that it adds it to the pool of available channels used to service fetch and fault requests. Registered channels are retained by the receiver. You use this method if you need to perform more than one fetch simultaneously.

See also: – **availableChannel**, – **registeredChannels**, – **unregisterChannel:**

registeredChannels

– (NSArray *)**registeredChannels**

Returns all of the EODatabaseChannels that have been registered for use with the receiver.

See also: – **registerChannel:**, – **availableChannel**, – **unregisterChannel:**

registerLockedObjectWithGlobalID:

– (void)**registerLockedObjectWithGlobalID:**(EOGlobalID *)*globalID*

Registers as a locked object the enterprise object identified by *globalID*. This method is used internally to keep track of objects corresponding to rows that are locked in the database.

See also: – **forgetAllLocks**, – **isObjectLockedWithGlobalID:**,
– **forgetLocksForObjectsWithGlobalIDs:**, – **lockObjectWithGlobalID:editingContext:**,
– **lockObject:** (EOEditingContext)

rollbackChanges

– (void)**rollbackChanges**

Overrides the inherited implementation to instruct the adaptor to roll back the transaction. Rolls back any changed snapshots, and releases all locks.

See also: – **performChanges**, – **commitChanges**

saveChangesInEditingContext:

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

Overrides the inherited implementation to save the changes made in *anEditingContext*. This message is sent by an EOEditingContext to its EOObjectStore to commit changes. Normally an editing context doesn't send this message to an EODatabaseContext, but to an EOObjectStoreCoordinator. Raises an exception if an error occurs; the error message indicates the nature of the problem.

setDelegate:

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

Sets the receiver's delegate to *delegate*, and propagates the delegate to all of the receiver's EODatabaseChannels. EODatabaseChannels share the delegate of their EODatabaseContext.

See also: – **delegate**

setUpdateStrategy:

– (void)**setUpdateStrategy:**(EOUpdateStrategy)*strategy*

Sets the update strategy used by the EODatabaseContext to *strategy.* See "Updating And Locking Strategies" in the class description for information on the update strategies:

- EOUpdateWithOptimisticLocking

- EOUpdateWithPessimisticLocking

Raises an NSInvalidArgumentException if the receiver has any transactions in progress or if you try to set *strategy* to EOUpdateWithPessimisticLocking and the receiver's EODatabase already has snapshots.

See also: – **updateStrategy**

snapshotForGlobalID:

– (NSDictionary *)**snapshotForGlobalID:**(EOGlobalID *)*globalID*

Returns the snapshot for the object identified by *globalID*, if there is one; else returns **nil**. Searches first locally (in the transaction scope) and then in the EODatabase.

See also: – **recordSnapshot:forGlobalID:**, – **localSnapshotForGlobalID:**, – **forgetSnapshotForGlobalID:**, – **recordSnapshots:**

snapshotForSourceGlobalID:relationshipName:

– (NSArray *)**snapshotForSourceGlobalID:**(EOGlobalID *)*globalID*
relationshipName:(NSString *)*name*

Returns a snapshot that consists of an array of global IDs. These global IDs identify the objects at the destination of the to-many relationship named *name*, which is a property of the object identified by *globalID*. If there is no snapshot, returns **nil**.

See also: – **recordSnapshot:forSourceGlobalID:relationshipName:**, – **localSnapshotForSourceGlobalID:relationshipName:**, – **recordToManySnapshots:**

unlock

– (void)**unlock**

Used internally to release the lock that protects access to the receiver in a multi-threaded environment.

See also: – **lock**

unregisterChannel:

– (void)**unregisterChannel:**(EODatabaseChannel *)*channel*

Unregisters the EODatabaseChannel *channel*, which means that it removes it from the pool of available channels used for database communication (for example, to service fetch and fault requests).

See also: – **registerChannel:**, – **registeredChannels**, – **availableChannel**

updateStrategy

– (EOUpdateStrategy)**updateStrategy**

Returns the update strategy used by the receiver, one of:

- EOUpdateWithOptimisticLocking
- EOUpdateWithPessimisticLocking

The default strategy is EOUpdateWithOptimisticLocking. See the class description for information on update strategies.

See also: – **setUpdateStrategy:**

valuesForKeys:object:

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

Overrides the inherited implementation to return values for the specified *keys* from the snapshot of *object*. The returned values are used primarily by another EODatabaseContext to extract foreign key properties for objects owned by the receiver.

Notifications

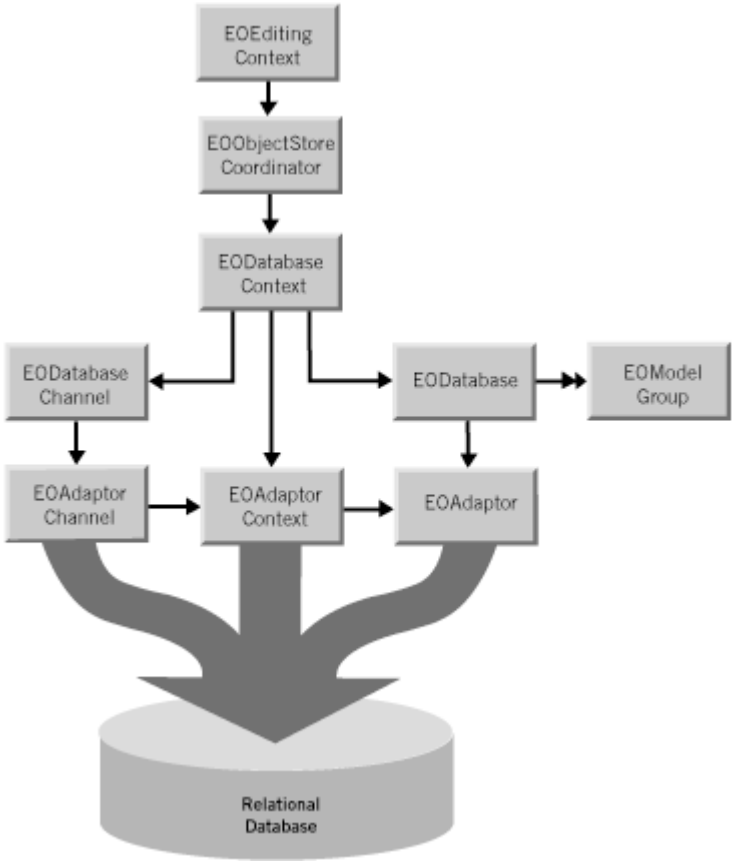
EODatabaseChannelNeededNotification

This notification is broadcast whenever an EODatabaseContext is asked to perform an object store operation and it doesn't have an available EODatabaseChannel. Subscribers can create a new channel and add it to the EODatabaseContext at this time.

Notification Object	The EODatabaseContext.
userInfo Dictionary	None.

EODatabaseContext

The relationship between EODatabaseContext and other classes in the control and access layers is illustrated in the following diagram.



As a subclass of EOCooperatingObjectStore, EODatabaseContext acts as one of possibly several EOCooperatingObjectStores for an EOObjectStoreCoordinator, which mediates between EOEditingContexts and EOCooperatingObjectStores.

An EODatabaseContext creates an EOAdaptorContext when initialized, and uses this object to communicate with the database server.

Creating and Using an EODatabaseContext

Though you can create an EODatabaseContext explicitly by using the class method **registeredDatabaseContextForModel:editingContext:**, you should rarely need to do so. If you're using the "higher-level" objects EOEditingContexts and EODatabaseDataSources, the database contexts those

objects need are created automatically, on demand. When you create database data source (typically for use with a display group—one ofEODisplayGroup, EODisplayGroup, or WODisplayGroup), it registers a database context that's capable of fetching objects for the data source's entities. If objects fetched into an editing context (described more in the following section) have references to objects from EOModels that are based on another database, an EODatabaseContext is creates and registered for each of the additional databases.

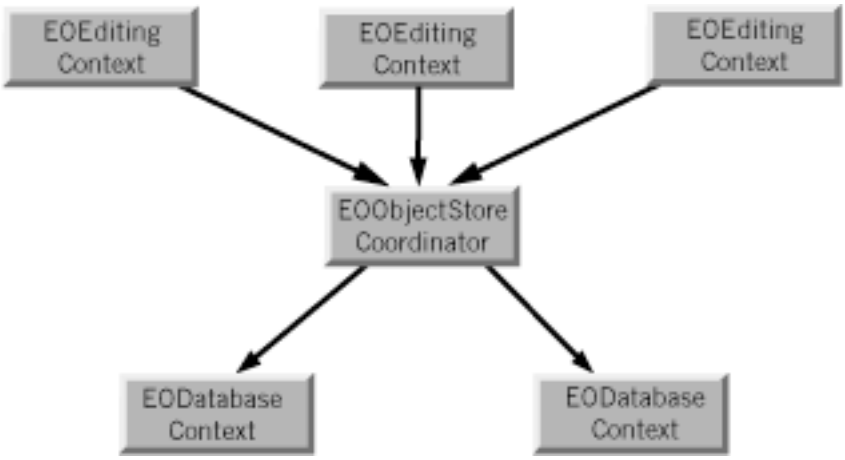
EODatabaseContexts are created on demand when an EOObjectStoreCoordinator posts an EOCooperatingObjectStoreNeeded notification. The EODatabaseContext class registers for the notification, and it provides the coordinator with a new EODatabaseContext instance that can handle the request. For more discussion of this topic, see the chapter "Application Configurations" in the *Enterprise Objects Framework Developer's Guide*.

For the most part, you don't need to programmatically interact with an EODatabaseContext. However, some of the reasons you might want to are as follows:

- To implement your own locking strategy, either application-wide, or on a per-fetch basis. This is described in the section "Updating And Locking Strategies."
- To do performance tuning, which is described in the section "Faulting."
- To intervene when objects are created and fetched to provide custom behavior. This is described in the section "Delegate Methods," and in the individual delegate method descriptions in the section "Instance Methods."

Fetching and Saving Objects

Conceptually, an EODatabaseContext fetches and saves objects on behalf of an EOEditingContext. However, the two objects don't interact with each other directly—an EOObjectStoreCoordinator acts as a mediator between them. The relationship between EOEditingContext, EOObjectStoreCoordinator, and EODatabaseContext is illustrated in the following figure. This configuration includes one EOObjectStoreCoordinator, and can include one or more EOEditingContexts, and one or more EODatabaseContexts.



When an editing context fetches objects, the request is passed through the coordinator, which forwards it to the appropriate database context based on the fetch specification or global ID. When the database context receives a request to fetch or write information to the database, it tries to use one of its EODatabaseChannels. If all of its channels are busy, it broadcasts an EODatabaseChannelNeededNotification in the hopes that an observer can provide a new channel or that an existing channel can be freed up. This observer could be a manager that decides how many database cursors can be opened by a particular client.

EODatabaseContext knows how to interact with other EOCooperatingObjectStores to save changes made to an object graph in more than one database server. For a more detailed discussion of this subject, see the class specifications for EOObjectStoreCoordinator and EOCooperatingObjectStore.

Setting a Fetch Limit

EODatabaseContext defines a hint for use with an EOFetchSpecification in the **objectsWithFetchSpecification:editingContext:** method. Named by the key EOFetchLimitHintKey, the hint's value is an NSNumber containing an unsigned integer value indicating the maximum number of objects to fetch. Depending on the value of the EOPromptAfterFetchLimitHintKey (NO or YES), the EODatabaseContext will either stop fetching objects when this limit is reached or it will ask the EOEditingContext's message handler to ask the user whether it should continue fetching. For more information on hint keys, see the method description for **objectsWithFetchSpecification:editingContext:**.

Using a Custom Query

EODatabaseContext defines a hint for use with an EOFetchSpecification in the **objectsWithFetchSpecification:editingContext:** method. Named by the key EOCustomQueryExpressionHintKey, the hint's value is a SQL string for performing the fetch. The expression must query the same attributes in the same order that Enterprise Objects Framework would if it were generating the SELECT expression dynamically. If this key is supplied, other characteristics of the

EOFetchSpecification such as **isDeep**, **qualifier**, and **sortOrderings** are ignored—in that sense this key is more of a directive than a hint. For more information on hint keys, see the method description for **objectsWithFetchSpecification:editingContext:**.

Faulting

When an EODatabaseContext fetches an object, it examines the relationships defined in the model and creates objects representing the destinations of the fetched object's relationships. For example, if you fetch an employee object, you can ask for its manager and immediately receive an object; you don't have to get the manager's employee ID from the object you just fetched and fetch the manager yourself.

However, EODatabaseContext doesn't immediately fetch data for the destination objects of relationships since fetching is fairly expensive. To avoid this waste of time and resources, the destination objects are created as EOFault objects which act as placeholders. EOFaults (or faults) come in two varieties: single object faults for to-one relationships, and array faults for to-many relationships.

When an EOFault is accessed (sent a message), it triggers its EODatabaseContext to fetch its data and transform it into an instance of the appropriate object class. This preserves both the object's **id** and its EOGlobalID.

You can fine-tune faulting behavior for additional performance gains by using two different mechanisms: batch faulting, and prefetching relationships.

Batch Faulting

When you access a fault, its data is fetched from the database. However, triggering one fault has no effect on other faults—it just fetches the object or array of objects for the one fault. You can take advantage of this expensive round trip to the database server by batching faults together. EODatabaseContext provides the **batchFetchRelationship:forSourceObjects:editingContext:** method for doing this. For example, given an array of Employee objects, this method can fetch all of their departments with one round trip to the server, rather than asking the server for each of the employee's departments individually. You can use the delegate methods **databaseContext:shouldFetchArrayFault:** and **databaseContext: shouldFetchObjectFault:** to fine-tune batch faulting behavior.

You can also set batch faulting in an EOModel. In that approach, you specify the *number* of faults that should be triggered along with the first fault; you don't actually control which faults are triggered the way you do with **batchFetchRelationship:forSourceObjects:editingContext:**. For more information on setting batch faulting in an EOModel, see the chapter "Using EOModeler" in the *Enterprise Objects Framework Developer's Guide*.

Prefetching Relationships

EODatabaseContext defines a hint for use with an EOFetchSpecification in the **objectsWithFetchSpecification:editingContext:** method. Named by the key EOPrefetchingRelationshipHintKey, the hint's value specifies relationships whose destinations should be

fetched along with the objects matching the fetch specification. Although prefetching increases the initial fetch cost, it can improve overall performance by reducing the number of round trips made to the database server. For more information on this and other hint keys, see the method description for **objectsWithFetchSpecification:editingContext:**.

Using this key also has an effect on how an EOFetchSpecification refreshes. “Refreshing” refers to existing objects being overwritten with fetched values—this allows your application to see changes to the database that have been made by someone else. Normally, when you set an EOFetchSpecification to refresh using **setRefreshesRefetchedObjects:**, it only refreshes the objects you’re fetching. For example, if you fetch employees, you don’t also fetch the employees’ departments. However, if you have the EOPrefetchingRelationshipHintKey set, the refetch is propagated for all of the relationships specified for the hint.

Delegate Methods

An EODatabaseContext shares its delegate with its EODatabaseChannels. These delegate methods are actually sent from EODatabaseChannel, but they’re defined in EODatabaseContext for ease of access:

- databaseContext:didSelectObjectsWithFetchSpecification:databaseChannel:
- databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:
- databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID:databaseChannel:
- databaseContext:shouldUsePessimisticLockWithFetchSpecification: databaseChannel:

– databaseContext:didSelectObjectsWithFetchSpecification:databaseChannel:

– databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:

– databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID:databaseChannel:

– databaseContext:shouldUsePessimisticLockWithFetchSpecification: databaseChannel:

You can use the EODatabaseContext delegate methods to intervene when objects are created and when they’re fetched from the database. This gives you more fine-grained control over such issues as how an object’s primary key is generated (**databaseContextNewPrimaryKeyForObjectdatabaseContext: newPrimaryKeyForObject:entity:**), how and if objects are locked (**databaseContextShouldLockObjectWithGlobalIDdatabaseContext: shouldLockObjectWithGlobalID:snapshot:**), what fetch specification is used to fetch objects (**databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:**), how batch faulting is performed (**databaseContext:shouldFetchArrayFault:** and **databaseContext: shouldFetchObjectFault:**), and so on. For more information, see the individual delegate method descriptions in the section “Instance Methods.”

Snapshots

An EODatabase records snapshots for its EODatabaseContexts. These snapshots form the application’s view of the current state of the database server. This global view is overridden locally by database contexts,

which form their own snapshots as they make changes during a transaction. When a database context commits its top-level transaction, it reconciles all changed snapshots with the global view of the database object, so that other database contexts (except those with open transactions) immediately use the new snapshots as well.

Updating And Locking Strategies

EODatabaseContext supports two updating strategies defined by the **EOUpdateStrategy** type as integer values:

Type	Description
EOUpdateWithOptimisticLocking	The default update strategy. Under optimistic locking, objects aren't locked immediately on being fetched from the server. Instead, whenever you attempt to save updates to an object in the database, the object's snapshot is used to ensure that the values in the corresponding database row haven't changed since the object was fetched. As long as the snapshot matches the values in the database, the update is allowed to proceed.
EOUpdateWithPessimisticLocking	Causes objects to be locked in the database when they're selected. This ensures that no one else can modify the objects until the transaction ends. However, this doesn't necessarily mean that either the select or the update operation will succeed.

EODatabaseContext also supports "on-demand" locking, in which specific optimistic locks can be promoted to database locks during the course of program execution. You can either use **lockObjectWithGlobalID:editingContext:** to lock a database row for a particular object, or **objectsWithFetchSpecification:editingContext:** to fetch objects with a fetch specification that includes locking.

For more discussion of locking strategies, see the chapter "Behind the Scenes" in the *Enterprise Objects Framework Developer's Guide*.

EODatabaseDataSource

Inherits From: EODataSource : NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EODatabaseDataSource.h

Class Description

EODatabaseDataSource is a concrete subclass of EODataSource (defined in EOControl) that fetches objects based on an EOModel, using an EODatabaseContext that services the data source's EOEditingContext (defined in EOControl). An EODatabaseDataSource can be set up to fetch all objects for its root entity, to fetch objects matching a particular EOFetchSpecification, and to further filter its fetching with an auxiliary qualifier.

EODatabaseDataSource implements all the functionality defined by EODataSource: In addition to fetching objects, it can insert and delete them (provided the entity isn't read-only). See the EODataSource class specification for more information on these topics.

As with other data sources, EODatabaseDataSource can also provide a detail data source. The most significant consequence of using an master-detail configuration is that the detail operates directly on the master's object graph. The EODetailDataSource has a *master object* and a *detail key* through which the detail data source accesses the its objects. The master object is simply the object that's selected in the master display group, and the detail key is the name of a relationship property in the master object. When the detail display group asks its data source to fetch, the EODetailDataSource simply gets the value for the relationship property named *detail key* from its master object and returns it. When you add and remove objects from the detail, you're directly modifying the master's relationship array. In fact, you can think of EODetailDataSource as an interface to its master object's relationship property.

Method Types

Creating instances

– initWithEditingContext:entityName:fetchSpecificationName:
– initWithEditingContext:entityName:fetchSpecificationName:

Accessing selection criteria

– auxiliaryQualifier
– fetchSpecification
– fetchSpecificationForFetch
– fetchSpecificationName
– setAuxiliaryQualifier:
– setFetchSpecification:
– setFetchSpecificationByName:

Accessing objects used for fetching

– entity
– databaseContext

Enabling fetching

– setFetchEnabled:
– isFetchEnabled

Accessing qualifier bindings

– qualifierBindingKeys
– qualifierBindings
– setQualifierBindings:

Instance Methods

auxiliaryQualifier

– (EOQualifier *)**auxiliaryQualifier**

Returns the EOQualifier used to further filter the objects fetched by the receiver's EOFetchSpecification (in EOControl).

See also: – **setAuxiliaryQualifier:**, – **fetchSpecificationForFetch**, – **fetchSpecification**

databaseContext

– (EODatabaseContext *)**databaseContext**

Returns the EODatabaseContext that the receiver uses to access the external database. This is either the root EOObjectStore for the receiver's EOEditingContext, or if the root is an EOCooperatingObjectStore, it's the EODatabaseContext under that EOCooperatingObjectStore that services the EOModel containing the EOEntity for the receiver. (EOObjectStore, EOEditingContext, and EOCooperatingObjectStore are all defined in EOControl.)

Deletes *anObject* from the data source. This method raises an exception on failure. If the receiver registers undos for the deletion, the receiver may receive a possibly redundant **insertObject** call.

entity

– (EOEntity *)**entity**

Returns the EOEntity from which the receiver fetches objects.

fetchSpecification

– (EOFetchSpecification *)**fetchSpecification**

Returns the receiver's basic EOFetchSpecification. Its EOQualifier is conjoined with the receiver's auxiliary EOQualifier when the receiver fetches objects. The sender of this message can alter the EOFetchSpecification directly, or replace it using **setFetchSpecification:**.

See also: **fetchSpecificationForFetch**, **auxiliaryQualifier**

fetchSpecificationForFetch

– (EOFetchSpecification *)**fetchSpecificationForFetch**

Returns a copy of the EOFetchSpecification that the receiver uses to fetch. This is constructed by conjoining the EOQualifier of the receiver's EOFetchSpecification with its auxiliary EOQualifier. Modifying the returned EOFetchSpecification doesn't affect the receiver's fetching behavior; use **setFetchSpecification:** and **setAuxiliaryQualifier:** for that purpose.

See also: – **fetchSpecification**, – **auxiliaryQualifier**

fetchSpecificationName

– (NSString *)**fetchSpecificationName**

Returns the name of the fetch specification (or **nil** if there is no name).

See also: – **setFetchSpecificationByName:**

initWithEditingContext:entityName:

– (id)**initWithEditingContext:**(EOEditingContext *)*anEditingContext* **entityName:**
(NSString *)*anEntityName*

Initializes a newly allocated EODatabaseDataSource to fetch objects into *anEditingContext* for the EOEntity named by *anEntityName*. This method checks *anEditingContext*'s EOObjectStoreCoordinator for an EODatabaseChannel that services the EOModel containing the named EOEntity. If none exists, this method creates one. This method works by calling **initWithEditingContext:entityName: fetchSpecificationName:** and specifying **nil** for the fetchSpecificationName.

initWithEditingContext:entityName:fetchSpecificationName:

– (id)**initWithEditingContext:**(EOEditingContext *)*anEditingContext*
entityName:(NSString *)*anEntityName*
fetchSpecificationName:(NSString *)*fetchSpecificationName*

Initializes a newly allocated EODatabaseDataSource to fetch objects into *anEditingContext* for the EOEntity named by *anEntityName*. This method checks *anEditingContext*'s EOObjectStoreCoordinator for an EODatabaseChannel that services the EOModel containing the named EOEntity. If none exists, this method creates one. The *fetchSpecificationName* argument is used to find the named fetch specification in the entity. If the *fetchSpecificationName* is **nil**, a new fetch specification will be instantiated that will fetch all objects of the entity. This is the primitive initializer. Returns **self**.

Inserts *object* into the data source.

isFetchEnabled

– (BOOL)**isFetchEnabled**

Returns YES if the receiver's **fetchObjects** method actually fetches objects, NO if it returns an empty array without fetching. Fetching is typically disabled in a master-peer configuration when no object is selected in the master.

See also: – **setFetchEnabled:**

qualifierBindingKeys

– (NSArray *)**qualifierBindingKeys**

Returns an array of strings which is a union of the binding keys from the fetch specification's qualifier and the data source's auxiliary qualifier.

See also: – **setQualifierBindings:**

qualifierBindings

– (NSDictionary *)**qualifierBindings**

Returns a set of bindings that will be used for variable replacement on the fetch specification's qualifier and the auxiliary qualifier before the fetch is executed.

See also: – **setQualifierBindings:**

setAuxiliaryQualifier:

– (void)**setAuxiliaryQualifier:**(EOQualifier *)*aQualifier*

Sets the receiver's auxiliary qualifier to *aQualifier*. The auxiliary qualifier usually adds conditions to the primary qualifier and is useful for narrowing the scope of a data source without altering its primary qualifier. This is especially useful for setting a qualifier on a qualified peer data source, since a peer's primary qualifiers specifies the matching criteria for the relationship it fetches for. For more information on auxiliary qualifiers, see "Creating a Master-Peer Configuration" in the "WebObjects Programming Topics."

See also: – **fetchSpecificationForFetch**, – **fetchSpecification**, – **auxiliaryQualifier**

setFetchEnabled:

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

Controls whether the receiver can fetch. If *flag* is YES the receiver's **fetchObjects** method actually fetches objects, if NO it returns an empty array without fetching. Fetching is typically disabled in a master-peer configuration when no object is selected in the master. For example, EODatabaseDataSource's implementation of **qualifyWithRelationshipKey:ofObject:** invokes this method to enable or disable fetching based on whether a master object is provided.

See also: – **isFetchEnabled**

setFetchSpecification:

– (void)**setFetchSpecification:**(EOFetchSpecification *)*aFetchSpecification*

Sets the receiver's basic EOFetchSpecification to *aFetchSpecification*. Its EOQualifier is conjoined with the receiver's auxiliary EOQualifier when the receiver fetches objects. This method also sets the name of the fetch specification to nil.

See also: – **setAuxiliaryQualifier:**, – **fetchSpecificationForFetch**, – **fetchSpecification**, – **setFetchSpecificationByName:**

setFetchSpecificationByName:

– (void)**setFetchSpecificationByName:**(NSString *)*fetchSpecificationName*

Sets the *fetchSpecificationName* as given, and sets the fetch specification (used when supplying objects) to the named fetch specification of the entity that was used to initialize the data source. This method is an alternative to **setFetchSpecification:**.

See also: – **fetchSpecificationName**

setQualifierBindings:

– (NSDictionary *)**setQualifierBindings:**(NSDictionary *)*bindings*

Sets a set of bindings that will be used for variable replacement on the fetch specification's qualifier and the auxiliary qualifier before the fetch is executed.

See also: – **qualifierBindingKeys**, – **qualifierBindings**

EODatabaseOperation

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EODatabaseOperation.h

Class Description

An EODatabaseOperation object represents an operation—insert, update, or delete—to perform on an enterprise object and all the necessary information required to perform the operation. You don't ordinarily create instances of EODatabaseOperation; rather, the Framework automatically creates an EODatabaseOperation object for each new, updated, or deleted object in an EOEditingContext. An EODatabaseContext object analyzes a set of database operations and maps each operation to one or more adaptor operations. The adaptor operations are then performed by an EOAdaptorChannel object. You generally interact with EODatabaseOperation objects only if you need to specify the order in which a set of operations are carried out (see the description for the EODatabaseContext delegate method **databaseContextWillOrderAdaptorOperationsFromDatabaseOperationsdatabaseContext: willOrderAdaptorOperationsFromDatabaseOperations:**).

An EODatabaseOperation specifies an enterprise object (called "object") on which the operation is performed, the EOGlobalID for the object, and the object's entity. In addition, the database operation has a snapshot containing the last known database values for the object and a **newRow** dictionary of new or updated values to save in the database. Finally, a database operation specifies one of the following operators (the type of operation represented by the database operation).

- EODatabaseNothingOperator
- EODatabaseInsertOperator
- EODatabaseUpdateOperator
- EODatabaseDeleteOperator

Method Types

Creating a new EODatabaseOperation

– initWithGlobalID:object:entity:

Accessing the global ID object

– globalID

Accessing the object

– object

Accessing the entity
– entity

Accessing the operator
– setDatabaseOperator:
– databaseOperator

Accessing the database snapshot
– setDBSnapshot:
– dbSnapshot

Accessing the row
– setNewRow:
– newRow

Accessing the adaptor operations
– addAdaptorOperation:
– removeAdaptorOperation:
– adaptorOperations

Comparing new row and snapshot values
– rowDiffs
– rowDiffsForAttributes:

Working with to-many snapshots
– recordToManySnapshot:relationshipName:
– toManySnapshots

Instance Methods

adaptorOperations

– (NSArray *)**adaptorOperations**

Returns the EOAdaptorOperation objects that need to be performed to carry out the operation represented by the receiver.

See also: – **addAdaptorOperation:**, – **removeAdaptorOperation:**

addAdaptorOperation:

– (void)**addAdaptorOperation:**(EOAdaptorOperation *)*adaptorOperation*

Adds *adaptorOperation* to the receiver's list of adaptor operations. Raises an exception if *adaptorOperation* is **nil**.

See also: – **adaptorOperations**, – **removeAdaptorOperation:**

databaseOperator

– (EODatabaseOperator)**databaseOperator**

Returns the receiver's database operator.

See also: **setDatabaseOperator:**

dbSnapshot

– (NSDictionary *)**dbSnapshot**

Returns the database snapshot for the receiver's enterprise object. The snapshot contains the last known database values for the enterprise object. The dictionary returned from this method will be empty if the receiver's object has just been inserted into an EOEditingContext and has not yet been saved in persistent storage. For more information on EOEditingContexts, see the EOEditingContext class specification in the EOControl framework.

See also: – **setDBSnapshot:**, – **setDatabaseOperator:**

entity

– (EOEntity *)**entity**

Returns the entity that corresponds to the receiver's enterprise object.

See also: – **initWithGlobalID:object:entity:**

globalID

– (EOGlobalID *)**globalID**

Returns the EOGlobalID object that corresponds to the receiver's enterprise object.

initWithGlobalID:object:entity:

– **initWithGlobalID:**(EOGlobalID *)*globalID* **object:**(id)*object* **entity:**(EOEntity *)*entity*

The designated initializer, this method initializes a new EODatabaseOperation instance. Sets the enterprise object to which the operation will be applied, the object's global ID, and the object's entity. Returns **self**.

See also: – **object**, – **entity**

newRow

– (NSMutableDictionary *)**newRow**

Returns a dictionary representation of the receiver's enterprise object. In addition to all the properties of the enterprise object that are stored in the database, the dictionary contains values for the non-derived attribute's of the enterprise object's entity that aren't visible in the enterprise object. For example, primary and foreign keys aren't ordinarily properties of an enterprise object but are attributes of the object's entity.

The **newRow** dictionary is initialized with the values in the receiver's snapshot. New or updated values are added to the **newRow** dictionary (replacing out-of-date values) as the Framework maps changes in the object to an operation.

See also: – **setNewRow:**

object

– (id)**object**

Returns the receiver's enterprise object.

primaryKeyDiffs

– (NSDictionary *)**primaryKeyDiffs**

See also: Returns a dictionary that contains any primary key values in **newRow** that are different from those in the **dbSnapshot**. Returns **nil** if the receiver doesn't have EODatabaseUpdateOperator set as its database operator.– **setDatabaseOperator:**, – **newRow**

recordToManySnapshot:relationshipName:

– (void)**recordToManySnapshot:**(NSArray *)*globalIDs* **relationshipName:**(NSString *)*name*

Records the objects in *globalIDs*. *globalIDs* is an array of the globalIDs that identify the objects at the destination of the to-many relationship named *name*; *name* is a property of the receiver's enterprise object.

See also: – **toManySnapshots**

removeAdaptorOperation:

– (void)**removeAdaptorOperation:**(EOAdaptorOperation *)*adaptorOperation*

Removes *adaptorOperation* from the receiver's list of adaptor operations.

See also: – **adaptorOperations**, – **addAdaptorOperation:**

rowDiffs

– (NSDictionary *)**rowDiffs**

Returns values in the receiver's **newRow** dictionary that are different than the corresponding values in its **dbSnapshot**. The dictionary returned from this method contains the new values from the enterprise object.

See also: – **primaryKeyDiffs**

rowDiffsForAttributes:

– (NSDictionary *)**rowDiffsForAttributes:**(NSArray *)*attributes*

For the EOAttribute objects in *attributes*, this method returns values in the receiver's **newRow** dictionary that are different than the corresponding values in its **dbSnapshot**. The dictionary returned contains the new values from the enterprise object.

setDatabaseOperator:

– (void)**setDatabaseOperator:**(EODatabaseOperator)*databaseOperator*

Sets the receiver's database operator. *databaseOperator* can be one of the following:

- EODatabaseNothingOperator
- EODatabaseInsertOperator
- EODatabaseUpdateOperator
- EODatabaseDeleteOperator

See also: – **databaseOperator**

setDBSnapshot:

– (void)**setDBSnapshot:**(NSDictionary *)*dbSnapshot*

Sets the snapshot for the receiver's enterprise object. If the object has just been inserted into an an EOEditingContext, it won't have a snapshot. In this case, *dbSnapshot* should be an empty dictionary.

See also: – **dbSnapshot**

setNewRow:

– (void)**setNewRow:**(NSMutableDictionary *)*newRow*

Sets the dictionary representation of the receiver's enterprise object. *newRow* should contain values for all the properties of the enterprise object that are stored in the database and for the non-derived attribute's of the enterprise object's entity that aren't visible in the enterprise object.

See also: – **newRow**, – **databaseOperator**

toManySnapshots

– (NSDictionary *)**toManySnapshots**

Returns the NSDictionary containing the snapshots for the to-many relationships of the receiver's enterprise object.

See also: – **recordToManySnapshot:relationshipName:**

EOEntity

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOAccess/EOEntity.h

Class Description

An EOEntity describes a table, file, or collection in a database and associates a name internal to the Framework with an external name by which the table is known to the database. An EOEntity maintains a group of attributes and relationships, which are collectively called properties. These are represented by the EOAttribute and EORelationship classes, respectively; see their specifications for more information.

You usually define entities in a model with the EOModeler application, which is documented in *WebObjects Tools and Techniques*. EOEntity objects are primarily used by the Enterprise Objects Framework for mapping tables in the database to enterprise objects; your code will probably make limited use of them unless you're specifically working with models.

An EOEntity is associated with a specific class whose instances are used to represent records (rows) from the database in applications using layers at or above the database layer of the Enterprise Objects Framework. If an EOEntity doesn't have a specific class associated with it, instances of EOGenericRecord (defined in EOControl) are created.

An EOEntity may be marked as read-only, in which case any changes to rows or objects for that entity made by the database level objects are denied.

You can define an external query for an EOEntity to be used when a selection is attempted with an unrestricted qualifier (one that would select all rows in the entity's table). An external query is sent unaltered to the database server and so can use database-specific features such as stored procedures; external queries are thus useful for hiding records or invoking database-specific features. You can also assign stored procedures to be invoked upon particular database operations through the use of EOEntity's **setStoredProcedure:forOperation:** method.

Like the other major modeling classes, EOEntity provides a user dictionary for your application to store any application-specific information related to the entity.

For more information on programmatically creating EOEntity objects, see "Creating an Entity."

Adopted Protocols

EOPropertyListEncoding

– awakeWithPropertyList
– encodeIntoPropertyList:
– initWithPropertyList:owner:

Method Types

Accessing the name

– setName:
– name
– validateName:
– beautifyName

Accessing the model

– model

Specifying fetching behavior for the entity

– setExternalQuery:
– externalQuery
– setRestrictingQualifier:
– restrictingQualifier

Accessing primary key qualifiers

– qualifierForPrimaryKey:
– isQualifierForPrimaryKey:

Accessing attributes

– addAttribute:
– anyAttributeNamed:
– attributeNamed:
– attributes
– removeAttribute:
– attributesToFetch

Accessing relationships

– addRelationship:
– anyRelationshipNamed:
– relationships
– relationshipNamed:
– removeRelationship:

Checking referential integrity

– externalModelsReferenced
– referencesProperty:

Accessing primary keys
- globalIDForRow:
- isPrimaryKeyValidInObject:
- primaryKeyForGlobalID:
- primaryKeyForRow:

Accessing primary key attributes
- setPrimaryKeyAttributes:
- primaryKeyAttributes
- primaryKeyAttributeNames
- primaryKeyRootName:
- isValidPrimaryKeyAttribute:

Accessing class properties
- setClassProperties:
- classProperties
- classPropertyNames
- isValidClassProperty:

Accessing the enterprise object class
- classDescriptionForInstances
- setClassName:
- className

Accessing locking attributes
- setAttributesUsedForLocking:
- attributesUsedForLocking
- isValidAttributeUsedForLocking:

Accessing external name
- setExternalName:
- externalName

Accessing whether an entity is read only
- setReadOnly:
- isReadOnly

Accessing the user dictionary
- setUserInfo:
- userInfo

Working with stored procedures
- setStoredProcedure:forOperation:
- storedProcedureForOperation:

Working with fetch specifications

– addFetchSpecification:withName:
– fetchSpecificationNamed:
– fetchSpecificationNames
– removeFetchSpecificationNamed:

Working with entity inheritance hierarchies

– parentEntity
– subEntities
– addSubEntity:
– removeSubEntity:
– setIsAbstractEntity:
– isAbstractEntity

Specifying fault behavior

– setMaxNumberOfInstancesToBatchFetch:
– maxNumberOfInstancesToBatchFetch

Caching objects

– setCachesObjects:
– cachesObjects

Instance Methods

addAttribute:

– (void)**addAttribute:**(EOAttribute *)*anAttribute*

Adds *anAttribute* to the receiver. Raises an NSInvalidArgumentException if *anAttribute*'s name is already in use by another attribute or relationship. Sets *anAttribute*'s entity to **self**.

See also: – **removeAttribute:**, – **attributes**, – **attributeNamed:**

addFetchSpecification:withName:

– (void)**addFetchSpecification:**(EOFetchSpecification *)*fetchSpec*
withName:(NSString *)*fetchSpecName*

Adds the fetch specification and associates *fetchSpecName* with it.

See also: – **fetchSpecificationNamed:**, – **fetchSpecificationNames**, – **removeFetchSpecificationNamed:**

addRelationship:

– (void)**addRelationship:**(EORelationship *)*aRelationship*

Adds *aRelationship* to the receiver. Raises an NSInvalidArgumentException if *aRelationship*'s name is already in use by another attribute or relationship. Sets *aRelationship*'s entity to **self**.

See also: – **removeRelationship:**, – **relationships**, – **relationshipNamed:**

addSubEntity:

– (void)**addSubEntity:**(EOEntity *)*child*

Causes the child entity *child* to "inherit" from the receiver. This is the first step in setting up an inheritance hierarchy between entities.

See also: – **subEntities** , – **removeSubEntity:**

anyAttributeNamed:

– (EOAttribute *)**anyAttributeNamed:**(NSString *)*attributeName*

Returns the user-created attribute identified by *attributeName*. If no such attribute exists, this method looks through the "hidden" attributes created by the Enterprise Objects Framework for one with the given name. Hidden attributes are used for such things as primary keys on target entities of flattened attributes. If none is found, **nil** is returned.

See also: – **attributeNamed:**, – **attributes**

anyRelationshipNamed:

– (EORelationship *)**anyRelationshipNamed:**(NSString *)*relationshipName*

Returns the user-created relationship identified by *relationshipName*. If none exists, this method looks through the "hidden" relationships created by the Enterprise Objects Framework for one with the given name. If none is found, **nil** is returned.

See also: – **relationshipNamed:**, – **relationships**

attributeNamed:

– (EOAttribute *)**attributeNamed:**(NSString *)*attributeName*

Returns the attribute named *attributeName*, or **nil** if no such attribute exists.

See also: – **anyAttributeNamed:**, – **attributes**, – **relationshipNamed:**

attributes

– (NSArray *)**attributes**

Returns all of the receiver's attributes, or **nil** if the receiver has none.

See also: – **anyAttributeNamed:**, – **attributeNamed:**

attributesToFetch

– (NSArray *)**attributesToFetch**

Returns an array of the EOAttributes that need to be fetched so that they can be included in the row snapshot. The set of attributes includes:

1. Attributes that are class properties, "used for locking," or primary keys.
2. Source attributes of any to-many relationship (flattened or non-flattened) that is a class property.
3. Source attributes of any non-flattened, to-one relationship that is a class property or that is used by a flattened attribute that is a class property.
4. The foreign key attributes of any flattened, to-one relationship that is a class property or that is used by a class property.

attributesUsedForLocking

– (NSArray *)**attributesUsedForLocking**

Returns an array containing those properties whose values must match a snapshot any time a row is updated.

Attributes used for locking are those whose values are compared when a database-level object performs an update. When the database-level classes fetch an enterprise object, they cache these attributes' values in a snapshot. Later, when the enterprise object is updated, the values of these attributes in the object are checked with those in the snapshot—if they differ, the update fails. See the EODatabaseContext class specification for more information.

beautifyName

– (void)**beautifyName**

Makes the receiver's name conform to a standard convention. EOEntity names that conform to this style are all lower-case except for the initial letter of each word, which is upper case. Thus, "MOVIE" becomes "Movie", and "MOVIE_ROLE" becomes "MovieRole".

See also: – **setName:**, – **validateName:**, – **beautifyNames** (EOModel)

cachesObjects

– (BOOL)**cachesObjects**

Returns YES if all of the objects from the receiver are to be cached in memory and queries are to be evaluated in-memory using this cache rather than in the database. This method should only be used for fairly small tables of read-only objects, since the first access to the receiver will trigger fetching the entire table. You should generally restrict this method to read-only entities to avoid cached data getting out of sync with database data. Also, you shouldn't use this method if your application will be making queries against the entity that can't be evaluated in memory.

See also: – **setCachesObjects:**

classDescriptionForInstances

– (EOClassDescription *)**classDescriptionForInstances**

Returns the EOClassDescription associated with the receiver. The EOClassDescription class provides a mechanism for extending classes by giving them access to the metadata contained in an EOModel (or another external source of information). In an application, EOClassDescriptions are registered on demand for the EOEntity on which an enterprise object is based. For more information, see the class specifications for EOClassDescription (in EOControl) and EOEntityClassDescription.

className

– (NSString *)**className**

Returns the name of the enterprise object class associated with the receiver. When a row is fetched for the receiver by a database-level object, it's returned as an instance of this class. This class might not be present in the run-time system, and in fact your application may have to load it on demand. If your application doesn't load a class, EOGenericRecord is used.

An enterprise object class other than EOGenericRecord can be mapped to only one entity.

classProperties

– (NSArray *)**classProperties**

Returns an array containing the properties that are bound to the receiver's class (so that instances of the class will be passed values corresponding to those properties). This is a subset of the receiver's attributes and relationships.

See also: – **classPropertyNames**

classPropertyNames

– (NSArray *)**classPropertyNames**

Returns an array containing the names of those properties that are bound to the receiver's class (so that instances of the class will be passed values corresponding to those properties). This is a subset of the receiver's attributes and relationships.

See also: – **classProperties**

externalModelsReferenced

– (NSArray *)**externalModelsReferenced**

Examines each of the receiver's relationships and returns a list of all external models referenced by the receiver.

See also: – **referencesProperty:**

externalName

– (NSString *)**externalName**

Returns the name of the receiver as understood by the database server.

externalQuery

– (NSString *)**externalQuery**

Returns a query statement that's used by an EOAdaptorChannel to select rows for the receiver when a qualifier is empty, or **nil** if the receiver has no external query. An empty qualifier is one that specifies only the entity, and would thus fetch all enterprise objects for that entity.

External queries are useful for hiding records or invoking database-specific features such as stored procedures when an application attempts to select all records for an entity. You can also use the EOStoredProcedure class to work with stored procedures; for more information see the EOStoredProcedure class specification.

See also: – **setExternalQuery:**

fetchSpecificationNamed:

– (EOFetchSpecification *)**fetchSpecificationNamed:**(NSString *)*fetchSpecName*

Returns the fetch specification associated with *fetchSpecName*.

See also: – **addFetchSpecification:withName:**, – **fetchSpecificationNames**, – **removeFetchSpecificationNamed:**

fetchSpecificationNames

– (NSArray *)**fetchSpecificationNames**

Returns an alphabetically sorted array of names of the entity's fetch specifications.

See also: – **addFetchSpecification:withName:**, – **fetchSpecificationNamed:**, – **removeFetchSpecificationNamed:**

globalIDForRow:

– (EOGlobalID *)**globalIDForRow:**(NSDictionary *)*aRow*

Constructs a global identifier from the specified row for the receiver.

See also: – **primaryKeyForGlobalID:**

isAbstractEntity

– (BOOL)**isAbstractEntity**

Returns YES to indicate that the receiver is abstract, NO otherwise. An abstract entity is one that has no corresponding enterprise objects in your application. Abstract entities are used to model inheritance relationships. For example, you might have a Person abstract entity that acts as the parent of Customer and Employee entities. Customer and Employee would inherit certain characteristics from Person (such as name and address attributes). However, though your application might have Customer and Employee objects, it would never have a Person object.

See also: – **setIsAbstractEntity:**

isPrimaryKeyValidInObject:

– (BOOL)**isPrimaryKeyValidInObject:**(id)*anObject*

Returns YES if every key attribute is present in *anObject* and has a value that is not **nil**. Returns NO otherwise. This method uses the key-value coding protocol so a dictionary may be provided instead of an enterprise object.

See also: – **primaryKeyForRow:**

isQualifierForPrimaryKey:

– (BOOL)**isQualifierForPrimaryKey:**(EOQualifier *)*aQualifier*

Returns YES if *aQualifier* describes the primary key and nothing but the primary key, NO otherwise.

isReadOnly

– (BOOL)**isReadOnly**

Returns YES if the receiver can't be modified, NO if it can. If an entity can't be modified, then enterprise objects fetched for that entity also can't be modified (that is, inserted, deleted, or updated).

isValidAttributeUsedForLocking:

– (BOOL)**isValidAttributeUsedForLocking:**(EOAttribute *)*anAttribute*

Returns NO if *anAttribute* isn't an EOAttribute, if the EOAttribute doesn't belong to the receiver, or if *anAttribute* is derived. Otherwise returns YES. An attribute that isn't valid for locking will cause **setAttributesUsedForLocking:** to fail.

See also: – **attributesUsedForLocking**

isValidClassProperty:

– (BOOL)**isValidClassProperty:**(id)*aProperty*

Returns NO if either *aProperty* isn't an EOAttribute or EORelationship, or if *aProperty* doesn't belong to the receiver. Otherwise returns YES. Note that this method doesn't tell you whether *aProperty* is a member of the array returned by **classProperties**. In other words, unlike **classProperties**, **classPropertyNames**, and **setClassProperties:**, this method doesn't interact with the properties bound to the entity's enterprise object class.

isValidPrimaryKeyAttribute:

– (BOOL)**isValidPrimaryKeyAttribute:**(EOAttribute *)*anAttribute*

Returns NO if *anAttribute* isn't an EOAttribute, doesn't belong to the receiver, or is derived. Otherwise returns YES.

See also: – **setPrimaryKeyAttributes:**

maxNumberOfInstancesToBatchFetch

– (unsigned int)**maxNumberOfInstancesToBatchFetch**

Returns the maximum number of to-one EOFaults from the receiver to fire at one time. See the method description for **setMaxNumberOfInstancesToBatchFetch:** for more explanation of what this means.

model

– (EOModel *)**model**

Returns the model that contains the receiver.

See also: – **addEntity:** (EOModel)

name

– (NSString *)**name**

Returns the receiver's name.

parentEntity

– (EOEntity *)**parentEntity**

Returns the entity from which the receiver inherits.

See also: – **subEntities**

primaryKeyAttributeNames

– (NSArray *)**primaryKeyAttributeNames**

Returns an array containing the names of the attributes that make up the receiver's primary key.

See also: – **primaryKeyAttributes**

primaryKeyAttributes

– (NSArray *)**primaryKeyAttributes**

Returns an array of those attributes that make up the receiver's primary key.

See also: – **primaryKeyAttributeNames**

primaryKeyForGlobalID:

– (NSDictionary *)**primaryKeyForGlobalID:**(EOKeyGlobalID *)*globalID*

Returns the primary key for the object identified by *globalID*.

See also: – **globalIDForRow:**

primaryKeyForRow:

– (NSDictionary *)**primaryKeyForRow:**(NSDictionary *)*aRow*

Returns the primary key for *aRow*, or **nil** if the primary key can't be computed. The primary key is aDictionary whose keys are attribute names and whose values are values for those attributes.

See also: – **primaryKeyForGlobalID:**

primaryKeyRootName:

– (NSString *)**primaryKeyRootName**

Returns the external name (that is, the name as it's understood by the database) of the receiver's root entity. If the receiver has no parent entity, returns the receiver's external name.

See also: – **externalName**, – **name**, – **parentEntity**

qualifierForPrimaryKey:

– (EOQualifier *)**qualifierForPrimaryKey:**(NSDictionary *)*aRow*

Returns a qualifier for the receiver that can be used to fetch an instance of the receiver with the primary key extracted from *aRow*.

See also: – **isQualifierForPrimaryKey:**, – **restrictingQualifier**

referencesProperty:

– (BOOL)**referencesProperty:**(id)*aProperty*

Returns YES if any of the receiver's attributes or relationships reference *aProperty*, NO otherwise. A property can be referenced by a flattened attribute or by a relationship. For example, suppose a model has an Employee entity with a **toDepartment** relationship. If you flatten the department's name attribute into the Employee entity, creating a **departmentName** attribute, that flattened attribute references the **toDepartment** relationship.

If an entity has any outstanding references to a property, you shouldn't remove the property.

See also: – **removeAttribute:**, – **removeRelationship:**

relationshipNamed:

– (EORelationship *)**relationshipNamed:**(NSString *)*name*

Returns the relationship named *name*, or **nil** if the receiver has no such relationship.

See also: – **anyRelationshipNamed:**, – **attributeNamed:**, – **relationships**

relationships

– (NSArray *)**relationships**

Returns all of the receiver's relationships, or **nil** if the receiver has none.

See also: – **attributes**

removeAttribute:

– (void)**removeAttribute:**(EOAttribute *)*name*

Removes the attribute named *name* if it exists. You should always use **referencesProperty:** to check that an attribute isn't referenced by another property before removing it.

See also: – **addAttribute:**, – **attributes**

removeFetchSpecificationNamed:

– (void)**removeFetchSpecificationNamed:**(NSString *)*fetchSpecName*

Removes the fetch specification referred to by *fetchSpecName*.

See also: – **addFetchSpecification:withName:**, – **fetchSpecificationNamed:**, – **fetchSpecificationNames**

removeRelationship:

– (void)**removeRelationship:**(EORelationship *)*name*

Removes the relationship named *name* if it exists. You should always use **referencesProperty:** to check that a relationship isn't referenced by another property before removing it.

See also: – **addRelationship:**, – **relationships**

removeSubEntity:

– (void)**removeSubEntity:**(EOEntity *)*child*

Removes *child* from the receiver's list of sub-entities.

See also: – **addSubEntity:**, – **subEntities**

restrictingQualifier

– (EOQualifier *)**restrictingQualifier**

Returns the qualifier used to restrict all queries made against the receiver. Restricting qualifiers are useful when there is not a one-to-one mapping between an entity and a particular database table, or when you always want to filter the data that's returned for a particular entity.

For example, if you're using the "one table" inheritance model in which parent and child data is contained in the same table, you'd use a restricting qualifier to fetch objects of the appropriate type. To give a non-inheritance example, for an Employees table you might create a "Sales" entity that has a restricting qualifier that only fetches employees who are in the Sales department.

See also: – **setRestrictingQualifier:**

setAttributesUsedForLocking:

– (BOOL)**setAttributesUsedForLocking:**(NSArray *)*attributes*

Sets *attributes* as the attributes used when an EODatabaseChannel locks enterprise objects for updates. Returns NO and doesn't set the attributes used for locking if any of the attributes in *attributes* responds NO to **isValidAttributeUsedForLocking:**; returns YES otherwise. See the EODatabase, EODatabaseContext, and EODatabaseChannel class specifications for information on locking.

setCachesObjects:

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

Sets according to *flag* whether all of the receiver's objects are cached the first time the associated table is queried.

See also: – **cachesObjects**

setClassName:

– (void)**setClassName:**(NSString *)*name*

Assigns *name* as the name of the class associated with the receiver. This class need not be present in the run-time system when this message is sent. When an EODatabaseChannel fetches objects for the receiver, they're created as instances of this class. Your application may have to load the class on demand if it isn't present in the run-time system; if it doesn't load the class, EOGenericRecord will be used.

Note: If you set the class name to **nil**, the **className** method returns "EOGenericRecord".

An enterprise object class other than EOGenericRecord can be mapped to only one entity.

See also: – **className**

setClassProperties:

– (BOOL)**setClassProperties:**(NSArray *)*properties*

Sets the receiver's class properties to the EOAttributes and EORelationships in *properties* and returns YES, unless the receiver responds NO to **isValidClassProperty:** for any of the objects in the array. In this event, the receiver's class properties aren't changed and NO is returned.

setExternalName:

– (void)**setExternalName:**(NSString *)*name*

Sets the name of the receiver as understood by the database server to *name*. For example, though your application may know the entity as "JobTitle" the database may require a form such as "JOB_TTL". An adaptor uses the external name to communicate with the database; your application should never need to use the external name.

setExternalQuery:

– (void)**setExternalQuery:**(NSString *)*aQuery*

Assigns *aQuery* as the query statement used for selecting rows from the receiver when there is no qualifier.

External queries are useful for hiding records or invoking database-specific features such as stored procedures when an application attempts to select all records for an entity. You can also use the EOStoredProcedure class to work with stored procedures; for more information see the EOStoredProcedure class specification.

An external query is sent unaltered to the database server, and so must contain the external (column) names instead of the names of EOAttributes. However, to work properly with the adaptor the external query must use the columns in alphabetical order by their corresponding EOAttributes' names.

See also: – **columnName** (EOAttribute), – **externalQuery**

setIsAbstractEntity:

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

Sets according to *flag* whether the receiver is an abstract entity. For more discussion of abstract entities, see the method description for **isAbstractEntity**.

setMaxNumberOfInstancesToBatchFetch:

– (void)**setMaxNumberOfInstancesToBatchFetch:**(unsigned int)*size*

Sets the maximum number of EOFaults from the receiver to trigger at one time. By default, only one object is fetched from the database when you trigger an EOFault. You can optionally use this method to set to size the number of EOFaults of the same entity should be fetched from the database along with the first one. Using this technique helps to optimize performance by taking advantage of round trips to the database.

See also: – **maxNumberOfInstancesToBatchFetch**

setName:

– (void)**setName:**(NSString *)*name*

Sets the receiver's name to *name*. Raises an NSInvalidArgumentException if *name* is already in use by another entity in the same EOModel or if *name* is not a valid entity name.

See also: – **beautifyName**, – **validateName:**

setPrimaryKeyAttributes:

– (BOOL)**setPrimaryKeyAttributes:**(NSArray *)*keys*

If the receiver responds NO to **isValidPrimaryKeyAttribute:** for any of the objects in *keys*, this method returns NO. Otherwise, this method sets the primary key attributes to the attributes in *keys* and returns YES.

You should exercise care in choosing primary key attributes. Floating-point numbers, for example, can't be reliably compared for equality, and are thus unsuitable for use in primary keys. Integer and string types are the safest choice for primary keys. NSDecimalNumbers will work, but they'll entail more overhead than integers.

See also: – **isValidPrimaryKeyAttribute:**

setReadOnly:

– (void)**setReadOnly:**(BOOL)*flag*

Sets according to *flag* whether the database rows for the receiver can be modified by the database level objects.

See also: – **isReadOnly**

setRestrictingQualifier:

– (void)**setRestrictingQualifier:**(EOQualifier *)*aQualifier*

Assigns *aQualifier* as the qualifier used to restrict all queries made against the receiver. The restricting qualifier can be used to map an entity to a subset of the rows in a table. For more discussion of this subject, see the description for **restrictingQualifier**.

setStoredProcedure:forOperation:

– (void)**setStoredProcedure:**(EOStoredProcedure *)*storedProcedure*
forOperation:(NSString *)*operation*

Sets *storedProcedure* for *operation*. *operation* can be one of the following:

Constant	Description
EOFetchAllProcedureOperation	Procedure that fetches all records from the database.
EOFetchWithPrimaryKeyProcedureOperation	Procedure that performs a fetch with primary key.
EOInsertProcedureOperation	Procedure that performs an insert.

Constant	Description
EODeleteProcedureOperation	Procedure that performs a delete.
EONextPrimaryKeyProcedureOperation	Procedure that performs generates a new primary key.

This information is used when changes from the object graph have been transformed into EODatabaseOperations that are being used to construct EOAdaptorOperations. At this point, Enterprise Objects Framework checks the entities associated with the changed objects to see if the entities have any stored procedures defined for the operation being performed.

See also: – **storedProcedureForOperation:**

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types—that is, NSString, NSDictionary, NSArray, and NSData.

storedProcedureForOperation:

– (EOStoredProcedure *)**storedProcedureForOperation:**(NSString *)*operation*

Returns the stored procedure for the specified *operation*, if one has been set. Otherwise, returns **nil**. *operation* can be one of the following:

- EOFetchAllProcedureOperation
- EOFetchWithPrimaryKeyProcedureOperation
- EOInsertProcedureOperation
- EODeleteProcedureOperation
- EONextPrimaryKeyProcedureOperation

See also: **– setStoredProcedure:forOperation:**, – **parameterDirection** (EOAttribute), – **storedProcedure** (EOAttribute)

subEntities

– (NSArray *)**subEntities**

Returns a list of those entities which inherit from the receiver.

See also: – **addSubEntity:**, – **parentEntity**, – **removeSubEntity:**

userInfo

– (NSDictionary *)**userInfo**

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

See also: – **setUserInfo:**

validateName:

– (NSException *)**validateName:**(NSString *)*name*

Validates *name* and returns **nil** if it is a valid name, or an exception if it isn’t. A name is invalid if it has zero length; starts with a character other than a letter, a number, or “@”, “#”, or “_”; or contains a character other than a letter, a number, “@”, “#”, “_”, or “$”. A name is also invalid if the receiver's model already has an EOEntity that has the same name or a stored procedure with an argument that has the same name.

setName: uses this method to validate its argument.

Creating an Entity

An EOEntity requires at least the following to be usable:

- A name
- The name of a table in the database (the external name)
- The name of an enterprise object class
- A set of attributes to be used as the primary key

Note that if an entity has no enterprise object class name, the database-level objects use EOGenericRecord. This code excerpt gives an example of creating an EOEntity and adding it to an EOModel:

```
EOModel *myModel;          /* Assume this exists. */
NSArray *keyAttributes;    /* Assume this exists. */
EOEntity *employeeEntity;
BOOL result;

employeeEntity = [[[EOEntity alloc] init] autorelease];
[employeeEntity setName:@"employee"];
[employeeEntity setExternalName:@"EMPLOYEE"];
[employeeEntity setClassName:@"Employee"];

/* Create at least the primary key attributes. */
result = [employeeEntity setPrimaryKeyAttributes:keyAttributes];

/* Add the entity to the model. */
[myModel addEntity:employeeEntity];
```

EOEntityClassDescription

Inherits From: EOClassDescription : NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOEntity.h

Class Description

EOEntityClassDescription is the subclass of the control layer's EOClassDescription. The EOClassDescription class provides a mechanism for extending classes by giving them access to metadata not available in the run-time system. EOEntityClassDescription extends the behavior of enterprise objects by deriving information about them (such as NULL constraints and referential integrity rules) from an associated EOModel.

In the typical scenario in which an enterprise object has a corresponding model file, the first time a particular operation is performed on a class (such as validating a value), an EOClassDescriptionNeeded... notification (either an EOClassDescriptionNeededForClassNotification or an EOClassDescriptionNeededForEntityNameNotification) is broadcast. When an EOModel object receives this notification it registers the metadata (class description) for the EOEntity on which the enterprise object is based. This class description is used from that point on.

For a more detailed discussion of this subject, see the EOClassDescription class specification.

Instance Methods

entity

– (EOEntity *)**entity**

Returns the entity associated with the receiver.

See also: – **initWithEntity:**

initWithEntity:

– **initWithEntity:**(EOEntity *)*anEntity*

Initializes a newly allocated EOEntityClassDescription with *anEntity.* Returns **self**.

EOGenericRecord Additions

Inherits From: NSObject

Declared In: EOAccess/EOGenericRecord.h

Class Description

The access layer adds one method to the control layer's EOGenericRecord class, for returning a generic record's associated EOEntity. Strictly speaking, EOGenericRecord doesn't rely on the access layer. However, in 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, an EODatabaseChannel also sets each generic record's **classDescription** to an EOEntityClassDescription, providing the link to the record's associated modeling objects.

Instance Methods

entity

– (EOEntity *)**entity**

Returns the receiver's EOEntity.

EOJoin

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOJoin.h

Class Description

An EOJoin describes one source-destination attribute pair for an EORelationship. See the EORelationship class specification for more information and for examples.

Method Types

Initializing new instances

– initWithSourceAttribute:destinationAttribute:

Querying the join

– destinationAttribute
– isReciprocalToJoin:
– sourceAttribute

Instance Methods

destinationAttribute

– (EOAttribute *)**destinationAttribute**

Returns the destination (“right”) attribute used by the join.

See also: – **destinationAttributes** (EORelationship)

initWithSourceAttribute:destinationAttribute:

– **initWithSourceAttribute:**(EOAttribute *)*source* **destinationAttribute:**(EOAttribute *)*destination*

Initializes a newly allocated EOJoin with the given source and destination attributes. This is the designated initializer for the EOJoin class. Returns **self**.

See the EORelationship class specification for an example of creating a relationship using EOJoins.

See also: – **addJoin:** (EORelationship)

isReciprocalToJoin:

– (BOOL)**isReciprocalToJoin:**(EOJoin *)*otherJoin*

Returns YES if this join's source attribute is equal to *otherJoin*'s destination attribute and *otherJoin*'s source attribute is equal to this join's destination attribute. This is known as a back-referencing join.

See also: – **inverseRelationship** (EORelationship)

sourceAttribute

– (EOAttribute *)**sourceAttribute**

Returns the source ("left") attribute used by the join.

See also: – **sourceAttributes** (EORelationship)

EOLoginPanel

Inherits From: NSObject

Declared In: EOAccess/EOAdaptor.h

Class Description

EOLoginPanel is an abstract class that defines how users of an Enterprise Objects Framework application provide database login information. Concrete subclasses of EOLoginPanel override its one method to run a modal login panel. Unless you are writing a concrete adaptor subclass, you shouldn't need to interact with this class. Generally, the Framework automatically creates and runs an instance of a concrete login panel object when your application needs connection information for the user. If you want to control when or how the login panel is run, use the EOAdaptor methods **runLoginPanelAndValidateConnectionDictionary** and **runLoginPanel**. When invoked, these methods create a concrete EOLoginPanel and interact with it for you.

If you are writing a concrete adaptor, you must provide a concrete subclass of EOLoginPanel and a graphical user interface (usually a **.nib** file). Enterprise Objects Framework expects these resources to be provided in a bundle named "LoginPanel" in the adaptor's framework. See the class specification for EOAdaptor for more information.

Instance Methods

administrativeConnectionDictionaryForAdaptor:

– (NSDictionary *)**administrativeConnectionDictionaryForAdaptor:**(EOAdaptor *)*adaptor*

Adaptor subclass should implement a subclass that implements this. Returns **nil** if the user cancels the panel.

runPanelForAdaptor:validate:allowsCreation:

– (NSDictionary *)**runPanelForAdaptor:**(EOAdaptor *)*adaptor*
validate:(BOOL)*flag*
allowsCreation:(BOOL)*allowsCreation*

Implemented by subclasses to run the login panel, allowing a user to enter new connection information. Returns the new connection information or **nil** if the user cancels the panel. If *flag* is YES, this method runs the login panel until the user enters valid connection information or cancels the panel. If *allowsCreation* is

YES, the panel will have an additional button that allows the user to creat a new database, and will prompt them for any necessary administrative information. When valid login information is entered in the panel, it is stored in *adaptor*'s connection dictionary and returned. Login information is validated by sending *adaptor* an **assertConnectionDictionaryIsValid** message.

If *flag* is NO, login information entered in the panel isn't validated and is returned without affecting the adaptor's connection dictionary.

A subclass must override this method without invoking EOAdaptor's implementation.

See also: – **setConnectionDictionary:** (EOAdaptor), – **assertConnectionDictionaryIsValid** (EOAdaptor), – **runLoginPanelAndValidateConnectionDictionary** (EOAdaptor), – **runLoginPanel** (EOAdaptor)

EOModel

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOModel.h

Class Description

An EOModel represents a mapping between a database schema and a set of classes based on the entity-relationship model. The model contains a number of EOEntity objects representing the entities (tables) of the database schema. Each EOEntity object has a number of EOAttribute and EORelationship objects representing the properties (columns or fields) of the entity in the database schema. For more information on attributes and relationships, see their respective class specifications.

An EOModel maintains a mapping between each of its EOEntity objects and a corresponding enterprise object class for use with the database level of the Enterprise Objects Framework. You can determine the EOEntity for a particular enterprise object with the **entityForObject:** method.

An EOModel is specific to a particular database server, and stores information needed to connect to that server. This includes the name of an adaptor framework to load so that the Enterprise Objects Framework can communicate with the database. Models are stored in the file system in a manner similar to adaptor framework. EOModel objects are usually loaded from model files built with the EOModeler application rather than built programmatically. If you need to programmatically load a model file, see the discussion in “Loading a Model File.”

Models can have relationships that reference other models in the same model group. The other models may map to different databases and types of servers.

Models are organized into model groups; see the EOModelGroup class specification for more information.

Creating an EOModel Programmatically

The EOAdaptorChannel class declares methods for reading basic schema information from a relational database. You can use this information to build up an EOModel programmatically, and then enhance that model by defining extra relationships, flattening attributes, and so on. See the class description in the EOAdaptorChannel class specification for information on reading basic schema information, and see the other modeling classes’ specifications for information on creating additional attributes and relationships.

Method Types

Initializing an EOModel instance
- initWithContentsOfFile:
- initWithTableOfContentsPropertyList:path:

Saving a model
- encodeTableOfContentsIntoPropertyList:
- writeToFile:

Loading a model's objects
- loadAllModelObjects

Working with entities
- addEntity:
- removeEntity:
- removeEntityAndReferences:
- entityNames
- entityNamed:
- entities

Naming a model's components
- beautifyNames

Accessing the model's name
- setName:
- name
- path

Checking references
- referencesToProperty:
- externalModelsReferenced

Getting an object's entity
- entityForObject:

Accessing the adaptor bundle
- adaptorName
- setAdaptorName:

Accessing the connection dictionary
- setConnectionDictionary:
- connectionDictionary

Accessing the user dictionary
- setUserInfo:
- userInfo

Working with stored procedures
– addStoredProcedure:
– removeStoredProcedure:
– storedProcedureNames
– storedProcedureNamed:
– storedProcedures

Accessing the model's group
– setModelGroup:
– modelGroup

Instance Methods

adaptorName

– (NSString *)**adaptorName**

Returns the name of the adaptor for the receiver. This name can be used with EOAdaptor's **adaptorWithName:** class method to create an adaptor.

addEntity:

– (void)**addEntity:**(EOEntity *)*anEntity*

Adds *anEntity* to the receiver. Raises an NSInvalidArgumentException if an error occurs (for example, if *anEntity* doesn't exist, if the entity belongs to another model, or if an entity of the same name is already in the receiver).

See also: – **entities**, – **removeEntity:**, **– removeEntityAndReferences:**

addStoredProcedure:

– (void)**addStoredProcedure:**(EOStoredProcedure *)*storedProcedure*

Adds *storedProcedure* to the receiver. Raises an NSInvalidArgumentException if an error occurs (for example, if a stored procedure of the same name is already in the receiver).

See also: – **removeStoredProcedure:**, – **storedProcedures**, – **storedProcedureNamed:**, – **storedProcedureNames**

availablePrototypeAttributeNames

– (NSArray *)**availablePrototypeAttributeNames**

Returns a list of available prototype names.

See also: – **prototypeAttributeNamed:**

beautifyNames

– (void)**beautifyNames**

Makes all of the receiver's named components conform to a standard convention. Names that conform to this style are all lower-case except for the initial letter of each embedded word other than the first, which is upper case. Thus, "NAME" becomes "name", and "FIRST_NAME" becomes "firstName".

See also: , – **name**

connectionDictionary

– (NSDictionary *)**connectionDictionary**

Returns a dictionary containing information used to connect to the database server. The connection dictionary is the place to specify default login information for applications using the model. See the EOAdaptor class specification for more information.

encodeTableOfContentsIntoPropertyList:

– (void)**encodeTableOfContentsIntoPropertyList:**(NSMutableDictionary *)*propertyList*

Encodes the receiver into *propertyList*. This method is used to get an ASCII representation of an EOModel in property list format.

See also: – **initWithTableOfContentsPropertyList:path:**

entities

– (NSArray *)**entities**

Returns an array containing the receiver's entities. Note that this method loads every entity, and thus defeats the benefits of incremental model loading.

See also: – **entityNames**

entityForObject:

– (EOEntity *)**entityForObject:**(id)*anEO*

Returns the entity associated with *anEO*, whether *anEO* is an instance of an enterprise object class, an instance of EOGenericRecord, or a fault object (see the EOFault class specification for information on faults). Returns **nil** if *anEO* has no associated entity.

entityNamed:

– (EOEntity *)**entityNamed:**(NSString *)*name*

Returns the entity named *name*, or **nil** if no such entity exists. Posts an EOEntityLoadedNotification when the entity is loaded.

See also: – **entityNames**, – **entities**

entityNames

– (NSArray *)**entityNames**

Returns an array containing the names of the EOModel's entities.

See also: – **entities**, – **entityNamed:**

externalModelsReferenced

– (NSArray *)**externalModelsReferenced**

Returns an array containing those models that are referenced by this model.

See also: – **referencesToProperty:**

initWithContentsOfFile:

– **initWithContentsOfFile:**(NSString *)*path*

Initializes a newly-allocated EOModel by reading the contents of the file named *path* as a model archive. The file specified by path can either be an old-style (**.eomodel**) or new-style (**.eomodeld**) model file. Sets the EOModel's name and path. **initWithContentsOfFile:** raises an NSInvalidArgumentException if for any reason it cannot initialize the model from the file specified by *path*.

See also: – **name**, – **path**

initWithTableOfContentsPropertyList:path:

– **initWithTableOfContentsPropertyList:**(NSDictionary *)*tableOfContents* **path:**(NSString *)*path*

Uses *tableOfContents* (which is the property list representation of an EOModel) with the file name *path* to initialize the receiver.

See also: – **encodeTableOfContentsIntoPropertyList:**

loadAllModelObjects

– (void)**loadAllModelObjects**

Loads any of the receiver's entities, stored procedures, attributes, and relationships that have not yet been loaded.

See also: – **attributes** (EOEntity), – **entities**, – **relationships** (EOEntity), – **storedProcedures**

modelGroup

– (EOModelGroup *)**modelGroup**

Returns the model group of which the receiver is a part.

See also: – **setModelGroup:**

name

– (NSString *)**name**

Returns the receiver's name.

See also: – **path**

path

– (NSString *)**path**

Returns the name of the EOModel file used to create the receiver, or **nil** if the model wasn't initialized from a file.

See also: – **name**

prototypeAttributeNamed:

– (EOAttribute *)**prototypeAttributeNamed:**(NSString *)*attributeName*

Returns the prototype attribute for the given *attributeName*. **prototypeAttributeNamed:** first looks for the prototype in EO*adaptorName*Prototypes. If the prototype isn't found there, it then looks in EOPrototypes. If the search is still unsuccessful, this method finally looks for the prototype in the list of prototypes provided by the adaptor itself.

See also: – **availablePrototypeAttributeNames**

referencesToProperty:

– (NSArray *)**referencesToProperty:**(id)*aProperty*

Returns an array of all properties in the receiver that reference *aProperty*, whether derived attributes, relationships that reference *aProperty*, and so on. Returns **nil** if *aProperty* isn't referenced by any of the properties in the model.

See also: – **externalModelsReferenced**

removeEntity:

– (void)**removeEntity:**(EOEntity *)*name*

Removes the entity with the given *name* without performing any referential integrity checking.

See also: – **addEntity:, – removeEntityAndReferences:**

removeEntityAndReferences:

– (void)**removeEntityAndReferences:**(EOEntity *)*entity*

Removes *entity* and any attributes or relationships in other entities that reference entity.

See also: – **removeEntity:**, – **addEntity:**

removeStoredProcedure:

– (void)**removeStoredProcedure:**(EOStoredProcedure *)*storedProcedure*

Removes *aStoredProcedure* without checking to see if an entity uses it.

See also: – **addStoredProcedure:**, **– storedProcedures**

setAdaptorName:

– (void)**setAdaptorName:**(NSString *)*adaptorName*

Sets the name of the receiver's adaptor to *adaptorName*.

See also: **availableAdaptorNames** (EOAdaptor)

setConnectionDictionary:

– (void)**setConnectionDictionary:**(NSDictionary *)*connectionDictionary*

Sets the dictionary containing information used to connect to the database to *connectionDictionary*. See the EOAdaptor class specification for more information on working with connection dictionaries.

See also: **adaptorWithModel:** (EOAdaptor)

setModelGroup:

– (void)**setModelGroup:**(EOModelGroup *)*group*

Sets the model group of which the receiver should be a part.

Note: You shouldn't change an EOModel's model group after it has been bound to other models in its group.

See also: – **modelGroup**

setName:

– (void)**setName:**(NSString *)*name*

Sets the name of the receiver to *name*.

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types—that is, NSString, NSDictionary, NSArray, and NSData.

storedProcedureNamed:

– (EOStoredProcedure *)**storedProcedureNamed:**(NSString *)*name*

Returns the stored procedure named *name*, or **nil** if the model doesn't contain a stored procedure with the given name.

See also: – **storedProcedureNames**, – **storedProcedures**

storedProcedureNames

– (NSArray *)**storedProcedureNames**

Returns an array containing the names of all of the model's stored procedures.

See also: – **storedProcedureNamed:**, – **storedProcedures**

storedProcedures

– (NSArray *)**storedProcedures**

Returns an array containing all of the model's stored procedures. Note that this method loads each of the model's stored procedures, thus defeating the benefits of incremental model loading.

See also: – **storedProcedureNames**, – **storedProcedureNamed:**

userInfo

– (NSDictionary *)**userInfo**

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

See also: – **setUserInfo:**

writeToFile:

– (void)**writeToFile:**(NSString *)*path*

Saves the receiver in the directory specified by *path*. If the file specified by path already exists, a backup copy is first created (using path with a "~" character appended). As a side-effect, this method resets the current path.

writeToFile: raises an NSInvalidArgumentException on any error which prevents the file from being written.

See also: – **path**

Notifications

EOModel declares and posts the following notification.

EOEntityLoadedNotification

Posted after an EOEntity is loaded into memory. The notification contains:

Notification Object	The entity that was loaded.
Userinfo	None

Loading a Model File

EOModels are usually loaded from model files built with the EOModeler application rather than built programmatically. EOModel files are typically stored in a project or a framework.

You use **initWithContentsOfFile:** to load an EOModel. Note that loading an EOModel doesn't have the effect of loading all of its entities. EOModel files can be quite large, so to reduce start-up time, entity definitions are only loaded as needed. This incremental model loading is possible because an EOModel actually consists of one index file and two files for each entity. Models have an **.eomodeld** file wrapper (which is actually a directory), and the individual entity files within the model are in ASCII format. The index file has the name **index.eomodeld**, and it contains the connection dictionary, the adaptor name, and a list of all of the entities in the model. It is this file that gets loaded when you use **initWithContentsOfFile:** . Thereafter, when an entity is loaded, EOModel posts an EOEntityLoadedNotification. The entity files are a **.plist** file that describes the entity and a .**fspec** file that describes any named fetch specifications for that entity.

Some of the EOModel methods contain the string "TableOfContents". An EOModel's "table of contents" corresponds to its **index.eomodeld** file, which is used to access the model's entities. **index.eomodeld** is just the ASCII representation of a model's table of contents.

EOModelGroup

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOModelGroup.h

Class Description

An EOModelGroup represents an aggregation of related models (see the EOModel class specification for more information on models). When a model in the group needs to resolve a relationship to an entity in another model, it looks for that model in its group. Model groups allow applications to load entities and their properties only as they're needed, by distributing them among separate EOModels.

The *default model group* contains all models for an application, as well as any frameworks the application references. It is automatically created on demand. The entity name space among all of these models is global; consequently, the same entity name shouldn't appear in any two of the models. All cross-model information is represented in the models by entity name only. Binding the entity name to an actual entity is done at run-time within the EOModelGroup.

In the majority of applications, the automatic creation of the default model group is sufficient. However, your code can override this automatic creation; see "Setting Up A Model Group Programmatically."

EOModelGroup Delegates

Your EOModelGroup object should have a delegate which can influence how it finds and loads models. In addition to the delegates you assign to EOModelGroup instances, the EOModelGroup class itself can have a delegate. The class delegate implements a single method—**defaultModelGroup**—while the instance delegate can implement the methods defined in the EOModelGroupDelegation protocol. For more information on EOModelGroup class delegate and instance delegate methods, see the EOModelGroupClassDelegation and EOModelGroupDelegation protocol specifications, respectively.Note that the following delegate methods are set on EOModelGroup, rather than EOEntity, to provide a single point in the code where you can alter the database-to-objects mapping:

- **entity:classForObjectWithGlobalID:**
- **entity:failedToLookupClassNamed:**
- **entity:relationshipForRow:relationship:**
- **subEntityForEntity:primaryKey:isFinal:**

Method Types

Accessing the group

– addModel:
– addModelWithFile:
– modelNamed:
– modelNames
– models
– modelWithPath:
– removeModel:

Accessing model groups

+ defaultGroup
+ setDefaultGroup:
+ globalModelGroup

Searching a group

– entityNamed:
– entityForObject:
– fetchSpecificationNamed:entityNamed:
– storedProcedureNamed:

Loading all of a group's objects

– loadAllModelObjects

Assigning a delegate

+ classDelegate
– delegate
+ setClassDelegate:
– setDelegate:

Class Methods

classDelegate

+ (id)**classDelegate**

Returns the EOModelGroup's class delegate. This delegate optionally implements the **defaultModelGroup** method (see the EOModelGroupClassDelegation protocol specification for more information).

See also: + **setClassDelegate:**

defaultGroup

+ (EOModelGroup *)**defaultGroup**

Returns the default EOModelGroup. Unless you've either specified a default model group with **setDefaultGroup:** or implemented the **defaultModelGroup** class delegate method to return a non-**nil** value, this method is equivalent to **globalModelGroup**.

See also: + **classDelegate**

globalModelGroup

+ (EOModelGroup *)**globalModelGroup**

Returns an EOModelGroup composed of all models in the resource directory of the main bundle, as well as those in all the bundles and frameworks loaded into the application.

See also: + **defaultGroup**

setClassDelegate:

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

Assigns *anObject* as the EOModelGroup's class delegate. The class delegate is optional; it allows you to determine the default model group (see the EOModelGroupClassDelegation protocol specification for more information).

See also: + **classDelegate**, – **defaultModelGroup**

setDefaultGroup:

+ (void)**setDefaultGroup:**(EOModelGroup *)*group*

Sets the default model group to *group*. If you've implemented the **defaultModelGroup** class delegate method to return a non-**nil** value, the delegate's return value overrides *group* as the default model group.

See also: + **defaultGroup**,+ **setClassDelegate:**

Instance Methods

addModel:

– (void)**addModel:**(EOModel *)*model*

Adds a *model* to the receiver, sets the *model*'s model group to the receiver, posts EOModelAddedNotification, then returns the newly-created EOModel. Raises if the receiver already contains an EOModel with the same name as the specified *model*.

See also: – **models**, – **removeModel:**

addModelWithFile:

– (EOModel *)**addModelWithFile:**(NSString *)*path*

Creates an EOModel object with the contents of the file identified by *path*, adds the newly-created model to the receiver, and returns it. Uses the EOModel method – **initWithContentsOfFile:** to initialize the new model, and adds it to the receiver with **addModel:**.

delegate

– (id)**delegate**

Returns the receiver's delegate, which is different from the EOModelGroup's class delegate. Each EOModelGroup object can have it's own delegate in addition to the delegate that's assigned to the EOModelGroup class. See the EOModelGroupDelegation protocol specification for more information.

See also: – **setDelegate:**, + **classDelegate**

entityForObject:

– (EOEntity *)**entityForObject:**(id)*object*

Returns the EOEntity associated with *object* from any of the models in the receiver that handle *object*, or **nil** if none of the entities in the receiver handles *object*.

See also: – **entityForObject:** (EOModel)

entityNamed:

– (EOEntity *)**entityNamed:**(NSString *)*entityName*

Searches each of the EOModels in the receiver for the entity specified by *entityName*, and returns the entity if found. Returns **nil** if it is unable to find the specified entity.

See also: – **entityNamed:** (EOModel)

fetchSpecificationNamed:entityNamed:

– (EOFetchSpecification *)**fetchSpecificationNamed:**(NSString *)*fetchSpecName*
entityNamed:(NSString *)*entityName*

Returns the named fetch specification from the entity specified by *entityName* in the receiving model group.

loadAllModelObjects

– (void)**loadAllModelObjects**

Sends **loadAllModelObjects** to each of the receiver's EOModels, thereby loading any EOEntities, EOAttributes, EORelationships, and EOStoredProcedures that haven't yet been loaded from each of the EOModels in the receiver.

See also: – **loadAllModelObjects** (EOModel)

modelNamed:

– (EOModel *)**modelNamed:**(NSString *)*modelName*

Returns the EOModel named *modelName* if it's part of the receiver, or **nil** if the receiver doesn't contain an EOModel with the specified name.

See also: – **modelNames**, – **models**

modelNames

– (NSArray *)**modelNames**

Returns an array containing the names of all of the EOModels in the receiver, or an empty array if the receiver contains no EOModels. The order of the model names in the array isn't defined.

See also: – **modelNamed:**, – **models**

models

– (NSArray *)**models**

Returns an array containing the receiver's EOModels, or an empty array if the receiver contains no EOModels. The order of the models in the array isn't defined.

See also: – **modelNamed:**, – **modelNames**, – **models**

modelWithPath:

– (EOModel *)**modelWithPath:**(NSString *)*path*

If the receiver contains an EOModel whose path (as determined by sending **path** to the EOModel object) is equal to *path*, that EOModel is returned. Otherwise, returns **nil**. NSString's **isEqual:** method is used to compare the paths, and each path is standardized (with **stringByStandardizingPath**) before comparison.

See also: – **modelNamed:**:, – **path** (EOModel)

removeModel:

– (void)**removeModel:**(EOModel *)*aModel*

Removes *aModel* from the receiver, and unbinds any connections to *aModel* from other EOModels in the receiver. Posts EOModelInvalidatedNotification to the default notification center after removing *aModel* from the receiver.

setDelegate:

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

Sets the receiver's delegate to *anObject*. See the EOModelGroupDelegation protocol specification for more information.

See also: **– delegate**

storedProcedureNamed:

– (EOStoredProcedure *)**storedProcedureNamed:**(NSString *)*aName*

Returns the stored procedure in the receiving model group having the given name.

Notifications

EOModelGroup declares and posts the following notifications.

EOModelAddedNotification

Posted by an EOModelGroup when an EOModel is added to the group. This notification is sent, for instance, inside Interface Builder when the user has saved changes to a model in EOModeler and the objects in Interface Builder must be brought back in sync. The old model is flushed and receivers of the notification (like data sources) can invoke **modelNamed:** to re-fetch their models.

Notification Object	The newly added model.
Userinfo	None

EOModelInvalidatedNotification

Posted by an EOModelGroup when an EOModel is removed from the group. This notification is sent, for instance, inside Interface Builder when the user has saved changes to a model in EOModeler and the objects in Interface Builder must be brought back in sync. The old model is flushed and receivers of the notification (like data sources) can invoke **modelNamed:** to re-fetch their models.

Notification Object	The invalidated model.
Userinfo	None

Setting Up A Model Group Programmatically

In the majority of applications, the automatic creation of the default model group is sufficient. However, if your particular application requires different model grouping semantics, you can create your own EOModelGroup instance, add the appropriate models, and then use that instance to replace the default EOModelGroup. The following code demonstrates the process:

```
NSString *modelPath;                        // Assume this exists
EOModelGroup *group = [EOModelGroup new];

[group addModelWithFile:modelPath];

[EOModelGroup setDefaultGroup:group];
[group release];
```

EOObjectStoreCoordinator Additions

Inherits From: EOObjectStoreCoordinator : NSObject

Declared In: EOAccess/EOModelGroup.h

Class Description

The EOAccess framework adds two methods to EOControl's EOObjectStoreCoordinator class for accessing the coordinator's EOModelGroup. An application can have multiple EOObjectStoreCoordinators, and each coordinator can have a different EOModelGroup. (For more discussion of this subject, see the chapter "Application Configurations" in the *Enterprise Objects Framework Developer's Guide*.) Application and framework code needing access to the EOModelGroup for a given EOEditingContext can get that information by asking the EOEditingContext's EOObjectStoreCoordinator for its EOModelGroup.

The methods are defined in a category of EOObjectStoreCoordinator in EOAccess (instead of in EOControl's EOObjectStoreCoordinator interface) to preserve the EOControl framework's independence of the EOAccess framework.

Instance Methods

modelGroup

– (EOModelGroup *)**modelGroup**

Returns the receiver's EOModelGroup. By default, this method returns the results of the statement `[EOModelGroup defaultGroup]`. If your application is using more than one EOObjectStoreCoordinator, each coordinator can have its own EOModelGroup.

setModelGroup:

– (void)**setModelGroup:**(EOModelGroup *)*group*

Sets to *group* the EOModelGroup used by the receiver. By default, an EOObjectStore's EOModelGroup is the model group returned from the statement `[EOModelGroup defaultGroup]`. However, you can override this by using **setModelGroup:** to explicitly set a different EOModelGroup for the receiver. Other parts of Enterprise Objects Framework (such as EODatabaseContext) use the EOModelGroup bound to their EOObjectStoreCoordinator.

EOQualifier Additions

Inherits From: NSObject

Declared In: EOAccess/EOSQLQualifier.h

Class Description

The access layer adds one method to the EOQualifier class, for "rerooting" a qualifier to another entity. EOQualifiers (except EOSQLQualifier) aren't based on SQL and they don't rely upon an EOModel. Because this method reroots a qualifier in terms of model objects, it is only useful to the classes in the access layer. It is not used in in-memory searches.

Instance Methods

qualifierMigratedFromEntity:relationshipPath:

– (EOQualifier *)**qualifierMigratedFromEntity:**(EOEntity *)*entity*
relationshipPath:(NSString *)*relationshipPath*

Creates a copy of the receiver, translates all the copy's keys to work with the entity specified in *relationshipPath*, and returns the copy. The receiver's keys are all specified in terms of *entity*. For example, assume that an Employee entity has a relationship to a Department entity named "department". You could migrate a qualifier described in terms of the Employee entity (department.name = 'Finance', for example) to a qualifier described in terms of the Department entity (name = 'Finance'). To do so, you send a **qualifierMigratedFromEntity:relationshipPath:** message with the Employee entity as the entity and "department" as the relationship path.

EORelationship

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EORelationship.h

Class Description

An EORelationship describes an association between two entities, based on attributes of those two entities. By defining EORelationships in your application's EOModel, you can cause the relationships defined in the database to be automatically resolved as enterprise objects are fetched. For example, a Movie entity may contain its **studioId** as an attribute, but without an EORelationship **studioId** will only appear in a movie enterprise object as a number. With an EORelationship explicitly connecting the Movie entity to a Studio entity, a movie enterprise object will automatically be given its studio enterprise object when an EODatabaseChannel fetches it from the database. The two entities that make up a relationship can be in the same model or two different models, as long as they are in the same model group.

You usually define relationships in your EOModel with the EOModeler application, which is documented in *WebObjects Tools and Techniques*. EORelationships are primarily for use by the Enterprise Objects Framework; unless you have special needs you shouldn't need to access them in your application's code. If you have such a need, you can create your own EORelationship objects as outlined in "Creating Relationships."

A relationship is directional: One entity is considered the source, and the other is considered the destination. The relationship belongs to the source entity, and may only be traversed from source to destination. To simulate a two-way relationship you have to create an EORelationship for each direction. Although the relationship is directional, no inverse is implied (although an inverse relationship may exist).

A relationship maintains an array of joins identifying attributes from the related entities (see the EOJoin class specification for more information). Most relationships simply relate the objects of one entity to those of another by comparing attribute values between them. Such a relationship must be defined as to-one or to-many based on how many objects of the destination match each object of the source. This is called the *cardinality* of the relationship. In a to-one relationship, there must be exactly one destination object for each source object; in a to-many relationship there can be any number of destination objects for each source object. See "Creating a Simple Relationship" for more information.

A chain of relationships across several entities can be flattened, creating a single relationship that spans them all. For example, suppose you have a relationship between movies and directors, and a relationship between directors and talent. You can traverse these relationships to create a flattened relationship going directly from movies to talent. A flattened relationship is determined to be to-many or to-one based on the

relationships it spans; if all are to-one, then the flattened relationship is to-one, but if any of them is to-many the flattened relationship is to-many. See “Creating a Flattened Relationship” for more information.

Like the other major modeling classes, EORelationship provides a user dictionary that the application can use to store application-specific information related to the relationship.

Specifying the Join Semantic

The relationship holds the join semantic; you specify this semantic with **setJoinSemantic:**. There are four types of join semantic, as specified by the EOJoinSemantic type: EOInnerJoin, EOFullOuterJoin, EOLeftOuterJoin, and EORightOuterJoin. An inner join produces results only for destinations of the join relationship that have non-NULL values. A full outer join produces results for all source records, regardless of the values of the relationships. A left outer join preserves rows in the left (source) table, keeping them even if there’s no corresponding row in the right table, while a right outer join preserves rows in the right (destination) table.

Note: Not all join semantics are supported by all database servers.

Adopted Protocols

EOPropertyListEncoding

– awakeWithPropertyList
– encodeIntoPropertyList:
– initWithPropertyList:owner:

Method Types

Accessing the relationship name

– beautifyName
– name
– setName:
– validateName:

Using joins

– addJoin:
– joins
– joinSemantic
– removeJoin:
– setJoinSemantic:

Accessing attributes joined on

– destinationAttributes
– sourceAttributes

Accessing the definition

– componentRelationships
– definition
– setDefinition:

Accessing the entities joined

– anyInverseRelationship
– destinationEntity
– entity
– inverseRelationship
– setEntity:

Checking the relationship type

– isCompound
– isFlattened
– isMandatory
– setIsMandatory:
– validateValue:

Accessing whether the relationship is to-many

– isToMany
– setToMany:

Relationship qualifiers

– qualifierWithSourceRow:

Checking references

– referencesProperty:

Controlling batch fetches

– numberOfToManyFaultsToBatchFetch
– setNumberOfToManyFaultsToBatchFetch:

Taking action upon a change

– deleteRule
– propagatesPrimaryKey
– setDeleteRule:
– setPropagatesPrimaryKey:
– ownsDestination
– setOwnsDestination:

Accessing the user dictionary

– setUserInfo:
– userInfo

Instance Methods

addJoin:

– (void)**addJoin:**(EOJoin *)*aJoin*

Adds a source-destination attribute pair to the relationship. Raises an NSInvalidArgumentException if the relationship is flattened, if either the source or destination attributes are flattened, or if either of *aJoin*'s attributes already belongs to another join of the relationship.

See also: – **joins**, – **isFlattened**, – **setDefinition:**

anyInverseRelationship

– (EORelationship *)**anyInverseRelationship**

Searches the relationship's destination entity for a user-created, back-pointing relationship joining on the same keys. If none is found, it looks for a "hidden" inverse relationship that was manufactured by the Framework. If none is found, the Enterprise Objects Framework creates a "hidden" inverse relationship and returns that. Hidden relationships are used internally by the Framework.

See also: – **inverseRelationship**

beautifyName

– (void)**beautifyName**

Makes the relationship's name conform to a standard convention. Names that conform to this style are all lower-case except for the initial letter of each embedded word other than the first, which is upper case. Thus, "NAME" becomes "name", and "FIRST_NAME" becomes "firstName". This method is used in reverse-engineering a model.

See also: – **setName:**, – **validateName:**, – **beautifyNames** (EOModel)

componentRelationships

– (NSArray *)**componentRelationships**

Returns an array of base relationships making up a flattened relationship, or **nil** if the relationship isn't flattened.

See also: – **definition**

definition

– (NSString *)**definition**

Returns the data path of a flattened relationship; for example “department.facility”. If the relationship isn’t flattened, **definition** returns **nil**.

See also: – **componentRelationships**

deleteRule

– (EODeleteRule)**deleteRule**

Returns a rule that describes the action to take when an object is being deleted. The returned rule is one of the following:

Value	Type	Description
EODeleteRuleNullify	int	Delete the department and remove any back pointer the employee has to the department.
EODeleteRuleCascade	int	Delete the department and all of the employees it contains.
EODeleteRuleDeny	int	Refuse the deletion if the department contains employees.
EODeleteRuleNoAction	int	Delete the department, but ignore the department’s **employees** relationship. You should use this delete rule with caution since it can leave dangling references in your object graph.

destinationAttributes

– (NSArray *)**destinationAttributes**

Returns the destination attributes of the relationship. These correspond one-to-one with the attributes returned by **sourceAttributes**. Returns **nil** if the relationship is flattened.

See also: – **joins**, – **destinationAttribute** (EOJoin)

destinationEntity

– (EOEntity *)**destinationEntity**

Returns the relationship's destination entity, which is determined by the destination entity of its joins for a simple relationship, and by whatever ends the data path for a flattened relationship. For example, if a flattened relationship's definition is "department.facility", the destination entity is the Facility entity.

See also: – **entity**

entity

– (EOEntity *)**entity**

Returns the relationship's source entity.

See also: – **destinationEntity**, – **addRelationship:** (EOEntity)

inverseRelationship

– (EORelationship *)**inverseRelationship**

Searches the relationship's destination entity for a user-created, back-pointing relationship joining on the same keys. Returns the inverse relationship if one is found, **nil** otherwise.

See also: – **anyInverseRelationship**

isCompound

– (BOOL)**isCompound**

Returns YES if the relationship contains more than one join (that is, if it joins more than one pair of attributes), NO if it has only one join. See "Creating a Simple Relationship" for information on compound relationships.

See also: – **joins**, – **joinSemantic**

isFlattened

– (BOOL)**isFlattened**

Returns YES if the relationship traverses more than two entities, NO otherwise. See "Creating a Flattened Relationship" for an example of a flattened relationship.

isMandatory

– (BOOL)**isMandatory**

Returns YES if the target of the relationship is required, NO if it can be **nil**.

See also: – **setIsMandatory:**

isToMany

– (BOOL)**isToMany**

Returns YES if the relationship is to-many, NO if it's to-one.

See also: – **setToMany:**

joinSemantic

– (EOJoinSemantic)**joinSemantic**

Returns the semantic used to create SQL expressions for this relationship. The returned join semantic is one of the following:

Constant	Description
EOInnerJoin	Produces results only for destinations of the join relationship that have non-NULL values.
EOFullOuterJoin	Produces results for all source records, regardless of the values of the relationships.
EOLeftOuterJoin	Preserves rows in the left (source) table, keeping them even if there's no corresponding row in the right table.
EORightOuterJoin	Preserves rows in the right (destination) table, keeping them even if there's no corresponding row in the left table.

See also: – **joins**

joins

– (NSArray *)**joins**

Returns all joins used by relationship.

See also: – **destinationAttributes,** – **joinSemantic,** – **sourceAttributes**

name

– (NSString *)**name**

Returns the relationship's name.

numberOfToManyFaultsToBatchFetch

– (unsigned int)**numberOfToManyFaultsToBatchFetch**

Returns the number of to-many faults that are triggered at one time.

ownsDestination

– (BOOL)**ownsDestination**

Returns YES if the receiver's source object owns its destination objects, NO otherwise. See the method description for **setOwnsDestination:** for more discussion of this topic.

See also: – **destinationAttributes**

propagatesPrimaryKey

– (BOOL)**propagatesPrimaryKey**

Returns YES if objects should propagate their primary key to related objects through this relationship. Objects only propagate their primary key values if the corresponding values in the destination object aren't already set.

qualifierWithSourceRow:

– (EOQualifier *)**qualifierWithSourceRow:**(NSDictionary *)*sourceRow*

Returns a qualifier that can be used to fetch the destination of the receiving relationship, given *sourceRow.*

referencesProperty:

– (BOOL)**referencesProperty:**(id)*aProperty*

Returns YES if *aProperty* is in the relationship's data path or is an attribute belonging to one of the relationship's joins; otherwise, it returns NO. See the class description for information on how relationships reference properties.

See also: **– referencesProperty:** (EOEntity)

removeJoin:

– (void)**removeJoin:**(EOJoin *)*aJoin*

Deletes *aJoin* from the relationship. Does nothing if the relationship is flattened.

See also: – **addJoin:**

setDefinition:

– (void)**setDefinition:**(NSString *)*definition*

Changes the relationship to a flattened relationship by releasing any joins and attributes (both source and destination) associated with the relationship and setting *definition* as its data path. “department.facility” is an example of a definition that could be supplied to this method.

If the relationship’s entity hasn’t been set, this method won’t work correctly. See “Creating a Flattened Relationship” for more information on flattened relationships.

See also: – **addJoin:**, – **setEntity:**

setDeleteRule:

– (void)**setDeleteRule:**(EODeleteRule)*deleteRule*

Set a rule describing the action to take when object is being deleted. *deleteRule* can be one of the following:

- EODeleteRuleNullify
- EODeleteRuleCascade
- EODeleteRuleDeny
- EODeleteRuleNoAction

For more discussion of what these rules mean, see the method description for **deleteRule**.

setEntity:

– (void)**setEntity:**(EOEntity *)*anEntity*

Sets the entity of the relationship to *anEntity*. If the relationship is currently owned by a different entity, this method will remove the relationship from that entity. This method doesn’t add the relationship to the new entity. EOEntity’s **addRelationship:** method invokes this method.

You only need to use this method when creating a flattened relationship; use EOEntity’s **addRelationship:** to associate an existing relationship with an entity.

See also: – **setDefinition:**

setIsMandatory:

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

Specifies according to *flag* whether the target of the relationship must be supplied or can be **nil**.

setJoinSemantic:

– (void)**setJoinSemantic:**(EOJoinSemantic)*joinSemantic*

Sets the semantic used to create SQL expressions for this relationship. *joinSemantic* should be one of the following:

- EOInnerJoin
- EOFullOuterJoin
- EOLeftOuterJoin
- EORightOuterJoin

See also: – **addJoin:**, – **joinSemantic**

setName:

– (void)**setName:**(NSString *)*name*

Sets the relationship's name to *name*. Raises a verification exception if *name* is not a valid relationship name, and NSInvalidArgumentException if *name* is already in use by an attribute or another relationship in the same entity.

This method forces all objects in the model to be loaded into memory.

See also: – **beautifyName**, – **validateName:**

setNumberOfToManyFaultsToBatchFetch:

– (void)**setNumberOfToManyFaultsToBatchFetch:**(unsigned int)*size*

Sets the number of "toMany" faults that are fired at one time to *size*.

See also: – **isToMany**, – **numberOfToManyFaultsToBatchFetch**

setOwnsDestination:

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

Sets according to *flag* whether a receiver's source object owns its destination objects. The default is NO. When a source object owns its destination objects, it means that the destination objects can't exist

independently. For example, in a personnel database, dependents can't exist without having an associated employee. Removing a dependent from an employee's **dependents** array would have the effect of also deleting the dependent from the database, unless you transferred the dependent to a different employee.

See also: – **deleteRule**, – **setDeleteRule:**, – **ownsDestination**

setPropagatesPrimaryKey:

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

Specifies according to *flag* whether objects should propagate their primary key to related objects through this relationship. For example, an Employee object might propagate its primary key to an EmployeePhoto object. Objects only propagate their primary key values if the corresponding values in the destination object aren't already set.

setToMany:

– (void)**setToMany:**(BOOL)*flag*

Sets a simple relationship as to-many according to *flag*. Raises an NSInvalidArgumentException if the receiver is flattened. See the class description for considerations in setting this flag.

See also: – **isFlattened**

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types (that is, NSDictionary, NSString, NSArray, and NSData).

sourceAttributes

– (NSArray *)**sourceAttributes**

Returns the source attributes of a simple (non-flattened) relationship. These correspond one-to-one with the attributes returned by **destinationAttributes**. Returns **nil** if the relationship is flattened.

See also: – **joins**, – **sourceAttribute** (EOJoin)

userInfo

– (NSDictionary *)**userInfo**

Returns a dictionary of user data. Your application can use this data for whatever it needs.

validateName:

– (NSException *)**validateName:**(NSString *)*name*

Validates *name* and returns **nil** if its a valid name, or an exception if it isn't. A name is invalid if it has zero length; starts with a character other than a letter, a number, or "@", "#", or "_"; or contains a character other than a letter, a number, "@", "#", "_", or "$". A name is also invalid if the receiver's EOEntity already has an EORelationship with the same name, or if the model has a stored procedure that has an argument with the same name.

setName: uses this method to validate its argument.

validateValue:

– (NSException *)**validateValue:**(id *)*valueP*

For relationships marked as mandatory, returns a validation exception if the receiver is to-one and *valueP* is **nil**, or if the receiver is to-many an *valueP* has a count of 0. A mandatory relationship is one in which the target of the relationship is required. Returns **nil** to indicate success.

See also: – **isMandatory**, – **setIsMandatory:**

Creating Relationships

Creating a Simple Relationship

A simple relationship is defined by the attributes it compares in connecting its source and destination entities. Each source-destination pair of attributes is encapsulated in an EOJoin object. For example, to create a relationship from the Movie entity to the Studio entity, a join has to be created from the **studioId** attribute of the Movie entity to the same attribute of the Studio entity. The values of these two attributes must be equal for a match to result. Note that **studioId** is the primary key attribute for the Studio entity, so there can only be one studio per movie; this relationship is therefore to-one.

This code excerpt creates an EORelationship for the relationship described above and adds it to the EOEntity for the Movie entity:

```
EOEntity *movieEntity;       // Assume this exists.
EOEntity *studioEntity;      // Assume this exists.
EOAttribute *studioIDAttribute;
EOAttribute *movieStudioIDAttribute;
EOJoin *toStudioJoin;
EORelationship *toStudioRelationship;

studioIDAttribute = [studioEntity attributeNamed:@"studioId"];
movieStudioIDAttribute = [movieEntity attributeNamed:@"studioId"];

toStudioJoin = [[[EOJoin alloc]
    initWithSourceAttribute:movieStudioIDAttribute
    destinationAttribute:studioIDAttribute] autorelease];

toStudioRelationship = [[[EORelationship alloc] init] autorelease];
[toStudioRelationship setName:@"studio"];
[movieEntity addRelationship:toStudioRelationship];
[toStudioRelationship addJoin:toStudioJoin];
[toStudioRelationship setToMany:NO];
[toStudioRelationship setJoinSemantic:EOInnerJoin];
```

This code first gets the attributes from the source and destination entities, and then creates an EOJoin with them. Next, a new EORelationship is created, its name is set, and it's added to **movieEntity**. The EOJoin is added to the relationship and the relationship is set to be to-one. Finally, in the **setJoinSemantic:** line, EOInnerJoin indicates that only objects that actually have a matching destination object will be included in the result when the relationship is traversed.

Creating a to-many relationship in the opposite direction merely swaps the source and destination attributes, and assigns the relationship to the EOEntity for the Studio entity:

```
EOJoin *toMoviesJoin;
EORelationship *toMoviesRelationship;

toMoviesJoin = [[[EOJoin alloc]
    initWithSourceAttribute:studioIDAttribute
    destinationAttribute:movieStudioIDAttribute] autorelease];

toMoviesRelationship = [[[EORelationship alloc] init] autorelease];
[toMoviesRelationship setName:@"movies"];
[studioEntity addRelationship:toMoviesRelationship];
[toMoviesRelationship addJoin:toMoviesJoin];
[toMoviesRelationship setToMany:YES];
[toMoviesRelationship setJoinSemantic:EOInnerJoin];
```

Note that this relationship is to-many precisely because the destination attribute isn't the primary key for its entity (Movie), and therefore isn't unique with regard to that entity.

A relationship isn't restricted to only one EOJoin. It's entirely possible for a relationship to be defined based on two or more attributes in the source and destination entities. For example, consider an employees database that contains a picture of each employee identified by first and last name. You'd define the relationship by joining each of the first and last names in the Employee entity to the same attribute in the **EmpPhoto** attribute.

A simple relationship is considered to reference all of the attributes in its joins. You can use the **referencesProperty:** method to find out if an EORelationship references a particular attribute.

Creating a Flattened Relationship

A flattened relationship depends on several simple relationships already existing. Assuming that several do exist, creating a flattened relationship is straightforward. For example, suppose that the Movie entity has a to-many relationship to the Director entity, called **toDirectors**. The Director entity in turn has a relationship to the Talent entity called **toTalent**. In the Movies database, the Director table acts as an intermediate table between Movie and Talent. In this situation, it make sense to flatten the relationship Movies has to Director (**toDirectors**) to give Movie access to the Talent table through Director's **toTalent** relationship. For more discussion of when to use flattened relationships, see the chapters "Designing Enterprise Objects" and "Advanced Enterprise Object Modeling" in the *Enterprise Objects Framework Developer's Guide*.

This code excerpt creates a flattened relationship from Movie to Talent:

```
EOEntity *movieEntity;   // Assume this exists.
EORelationship *toDirectorsRelationship;

toDirectorsRelationship = [[[EORelationship alloc] init] autorelease];
[toDirectorsRelationship setName:@"directors"];
[toDirectorsRelationship setEntity:movieEntity];
[movieEntity addRelationship:toDirectorsRelationship];
[toDirectorsRelationship setDefinition:@"toDirector.toTalent"];
```

All that's needed to establish the relationship is a data path (also called the definition) naming each component relationship connected, with the names separated by periods. Note that because the cardinality of a flattened relationship is determinable from its components, no **setToMany:** message is required here.

A simple relationship is considered to reference all of the relationships in its definition, plus every attribute referenced by the component relationships. You can use the **referencesProperty:** method to find out if an EORelationship references another relationship or attribute.

EOSQLExpression

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOSQLExpression.h
EOAccess/EOSchemaGeneration.h

Class Description

EOSQLExpression is an abstract superclass that defines how to build SQL statements for adaptor channels. You don't typically use instances of EOSQLExpression; rather, you use EOSQLExpression subclasses written to work with a particular RDBMS and corresponding adaptor. A concrete subclass of EOSQLExpression overrides many of its methods in terms of the query language syntax for its specific RDBMS. EOSQLExpression objects are used internally by the Framework, and unless you're creating a concrete adaptor, you won't ordinarily need to interact with EOSQLExpression objects yourself. You most commonly create and use an EOSQLExpression object when you want to send an SQL statement directly to the server. In this case, you simply create an expression with the EOSQLExpression class method **expressionForString:**, and send the expression object to an adaptor channel using EOAdaptorChannel's **evaluateExpression:** method.

For more information, see "EOSQLExpression".

Method Types

Creating an EOSQLExpression object

+ selectStatementForAttributes:lock:fetchSpecification:entity:
+ insertStatementForRow:entity:
+ updateStatementForRow:qualifier:entity:
+ deleteStatementWithQualifier:entity:
+ expressionForString:
– initWithEntity:

Building SQL Expressions

– prepareSelectExpressionWithAttributes:lock:fetchSpecification:
– prepareInsertExpressionWithRow:
– prepareUpdateExpressionWithRow:qualifier:
– prepareDeleteExpressionForQualifier:
– setStatement:
– statement

Generating SQL for attributes and values

+ formatSQLString:format:
+ formatValue:forAttribute:
+ formatStringValue:
– sqlStringForValue:attributeNamed:
– sqlStringForAttributeNamed:
– sqlStringForAttribute:
– sqlStringForAttributePath:

Generating SQL for names of database objects

– sqlStringForSchemaObjectName:
+ setUseQuotedExternalNames:
+ useQuotedExternalNames
– externalNameQuoteCharacter

Generating an attribute list

– addSelectListAttribute:
– addInsertListAttribute:value:
– addUpdateListAttribute:value:
– appendItem:toListString:
– listString

Generating a value list

– addInsertListAttribute:value:
– addUpdateListAttribute:value:
– valueList

Generating a table list

– tableListWithRootEntity:
– aliasesByRelationshipPath

Generating the join clause

– joinExpression
– addJoinClause
– assembleJoinClauseWithLeftName:rightName:joinSemantic:
– joinClauseString

Generating a search pattern

+ sqlPatternFromShellPattern:
+ sqlPatternFromShellPattern:withEscapeCharacter:

Generating a relational operator

– sqlStringForSelector:value:

Accessing the where clause

– whereClauseString

Generating an order by clause

– addOrderByAttributeOrdering:
– orderByString

Accessing the lock clause

– lockClause

Assembling a statement

– assembleSelectStatementWithAttributes:lock:qualifier:fetchOrder:
selectString:columnList:tableList:whereClause:joinClause:
orderByClause:lockClause:
– assembleInsertStatementWithRow:tableList:columnList:valueList:
– assembleUpdateStatementWithRow:qualifier:tableList:updateList:
whereClause:
– assembleDeleteStatementWithQualifier:tableList:whereClause:

Generating SQL for qualifiers

– sqlStringForConjoinedQualifiers:
– sqlStringForDisjoinedQualifiers:
– sqlStringForKeyComparisonQualifier:
– sqlStringForKeyValueQualifier:
– sqlStringForNegatedQualifier:

Managing bind variables

+ setUseBindVariables:
+ useBindVariables
– addBindVariableDictionary:
– bindVariableDictionaries
– bindVariableDictionaryForAttribute:value:
– mustUseBindVariableForAttribute:
– shouldUseBindVariableForAttribute:

Using table aliases

– setUseAliases:
– useAliases

Accessing the entity

entity

Creating a schema generation script

+ schemaCreationStatementsForEntities:options:
– createDatabaseStatementsForConnectionDictionary:
administrativeConnectionDictionary:
– dropDatabaseStatementsForConnectionDictionary:
administrativeConnectionDictionary:

createDatabaseStatementsForConnectionDictionary: administrativeConnectionDictionary:

+ (NSArray *)**createDatabaseStatementsForConnectionDictionary:**
(NSDictionary *)*connectionDictionary*
administrativeConnectionDictionary:(NSDictionary *)*adminDictionary*

Generates the SQL statements that will create a database (or user, for Oracle) that can be accessed by the provided connection dictionary and administrative connection dictionary.

See also: + **dropDatabaseStatementsForConnectionDictionary:administrativeConnectionDictionary:**

deleteStatementWithQualifier:entity:

+ (EOSQLExpression *)**deleteStatementWithQualifier:**(EOQualifier *)*qualifier* **entity:**(id)*entity*

Creates and returns an SQL DELETE expression to delete the rows described by *qualifier*. Creates an instance of EOSQLExpression, initializes it with *entity* (an EOEntity object), and sends it a **prepareDeleteExpressionForQualifier:** message. Raises an NSInvalidArgumentException if *qualifier* is **nil**.

The expression created with this method does not use table aliases because Enterprise Objects Framework assumes that all INSERT, UPDATE, and DELETE statements are single-table operations. As a result, all keys in *qualifier* should be simple key names; no key paths are allowed. To generate DELETE statements that do use table aliases, you must override **prepareDeleteExpressionForQualifier:** to send a **setUseAliases:**YES message prior to invoking **super**'s version.

dropDatabaseStatementsForConnectionDictionary: administrativeConnectionDictionary:

+ (NSArray *)**dropDatabaseStatementsForConnectionDictionary:**
(NSDictionary *)*connectionDictionary*
administrativeConnectionDictionary:(NSDictionary *)*adminDictionary*

Generates the SQL statements to drop the database (or user, for Oracle).

See also: + **createDatabaseStatementsForConnectionDictionary:**
administrativeConnectionDictionary:

expressionForString:

+ (EOSQLExpression *)**expressionForString:**(NSString *)*string*

Creates and returns an SQL expression for *string*. *string* should be a valid expression in the target query language. This method does not perform substitutions or formatting of any kind.

See also: – **setStatement:**

formatSQLString:format:

+ (NSString *)**formatSQLString:**(NSString *)*sqlString* **format:**(NSString *)*format*

Applies *format* (an EOAttribute object's "read" or "write" format) to *sqlString* (a value for the attribute). If *format* is **nil**, this method returns *sqlString* unchanged.

See also: **– readFormat** (EOAttribute), **– writeFormat** (EOAttribute)

formatStringValue:

+ (NSString *)**formatStringValue:**(NSString *)*string*

Formats *string* for use as a string constant in a SQL statement. EOSQLExpression's implementation encloses the string in single quotes, escaping any single quotes already present in *string*. Raises an NSInternalInconsistencyException if *string* is **nil**.

formatValue:forAttribute:

+ (NSString *)**formatValue:**(id)*value* **forAttribute:**(EOAttribute *)*attribute*

Overridden by subclasses to return a string representation of *value* suitable for use in an SQL statement. EOSQLExpression's implementation returns *value* unchanged. A subclass should override this method to format *value* depending on *attribute*'s **externalType**. For example, a subclass might format a date using a special database-specific syntax or standard form or truncate numbers to *attribute*'s precision and scale.

insertStatementForRow:entity:

+ (EOSQLExpression *)**insertStatementForRow:**(NSDictionary *)*row* **entity:**(EOEntity *)*entity*

Creates and returns an SQL INSERT expression to insert *row*. Creates an instance of EOSQLExpression, initializes it with *entity*, and sends it **prepareInsertExpressionWithRow:**. Raises an NSInvalidArgumentException if *entity* is **nil**.

The expression created with this method does not use table aliases because Enterprise Objects Framework assumes that all INSERT, UPDATE, and DELETE statements are single-table operations. To generate

INSERT statements that do use table aliases, you must override **prepareInsertExpressionWithRow:** to send a **setUseAliases:**YES message prior to invoking **super**'s version.

schemaCreationStatementsForEntities:options:

+ (NSArray *)**schemaCreationStatementsForEntities:**(NSArray *)*entities*
options:(NSDictionary *)*options*

Returns an array of SQLExpressions suitable to create the schema for the Entity objects in *entities*. The *options* dictionary specifies the aspects of the schema for which to create SQLExpressions:

Dictionary Key	Acceptable Values (java.util.Strings)	Default
createTables	"YES" or "NO"	YES
dropTables	"YES" or "NO"	YES
createPrimaryKeySupport	"YES" or "NO"	YES
dropPrimaryKeySupport	"YES" or "NO"	YES
primaryKeyConstraints	"YES" or "NO"	YES
foreignKeyConstraints	"YES" or "NO"	NO
createDatabase	"YES" or "NO"	NO
dropDatabase	"YES" or "NO"	NO

If you specify "createDatabase" or "dropDatabase," the SQL for those statements must be executed by an administrative user.

EOSQLExpression's implementation uses the following methods:

- createTableStatementsForEntityGroups
- dropTableStatementsForEntityGroups
- primaryKeySupportStatementsForEntityGroups
- dropPrimaryKeySupportStatementsForEntityGroups
- primaryKeyConstraintStatementsForEntityGroups
- foreignKeyConstraintStatementsForRelationship

to generate SQLExpressions for the support identified in *options*.

selectStatementForAttributes:lock:fetchSpecification:entity:

+ (EOSQLExpression *)**selectStatementForAttributes:**(NSArray *)*attributes*
lock:(BOOL)*flag*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification*
entity:(EOEntity *)*entity*

Creates and returns an SQL SELECT expression. Creates an instance of EOSQLExpression, initializes it with *entity*, and sends it **prepareSelectExpressionWithAttributes:lock:fetchSpecification:**. The expression created with this method uses table aliases. Raises an NSInvalidArgumentException if *attributes* is **nil** or empty, *fetchSpecification* is **nil**, or *entity* is **nil**.

The expression created with this method uses table aliases. To generate SELECT statements that don't use them, you must override **prepareSelectExpressionWithAttributes:lock:fetchSpecification:** to send a **setUseAliases:**NO message prior to invoking **super**'s version.

setUseBindVariables:

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

Sets according to *flag* whether all instances of EOSQLExpression subclasses use bind variables. By default, instances don't use bind variables; if the value for the global user default named EOAdaptorUseBindVariables is YES, though, instances do use them. For more information on bind variables, see the discussion in the class description.

See also: + **useBindVariables**

setUseQuotedExternalNames:

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

Sets whether all instances of EOSQLExpression subclasses quote external names when they are referenced in SQL statements. By setting *flag* to YES, you can access database tables with names such as "%return", "1st year", and "TABLE" that you couldn't otherwise access. By default, instances don't quote external names; if the value for the global user default named EOAdaptorQuotesExternalNames is YES, though, instances do use quotes.

See also: + **useQuotedExternalNames**, – **sqlStringForSchemaObjectName:**,
– **externalNameQuoteCharacter**

sqlPatternFromShellPattern:

+ (NSString *)**sqlPatternFromShellPattern:**(NSString *)*pattern*

Translates a “like” qualifier to an SQL “like” expression. Invoked from **sqlStringForKeyValueQualifier:** when the qualifier argument is an EOKeyValueQualifier object whose selector is **isLike:**. EOSQLExpression’s implementation performs the following substitutions

Character in pattern	Substitution string
*	%
?	_
%	[%] *(unless the percent character appears in square brackets)*
_	[_] *(unless the underscore character appears in square brackets)*

See also: + **sqlPatternFromShellPattern:withEscapeCharacter:**

sqlPatternFromShellPattern:withEscapeCharacter:

+ (NSString *)**sqlPatternFromShellPattern:**(NSString *)*pattern*
withEscapeCharacter:(unichar)*escapeCharacter*

Like **sqlPatternFromShellPattern:** except the argument *escapeCharacter* allows you to specify a character for escaping the wild card characters “%” and “_”.

updateStatementForRow:qualifier:entity:

+ (EOSQLExpression *)**updateStatementForRow:**(NSDictionary *)*row*
qualifier:(EOQualifier *)*qualifier*
entity:(EOEntity *)*entity*

Creates and returns an SQL UPDATE expression to update the row identified by *qualifier* with the values in *row*. *row* should only contain entries for values that have actually changed. Creates an instance of EOSQLExpression, initializes it with *entity*, and sends it **prepareUpdateExpressionWithRow:qualifier:** . Raises an NSInvalidArgumentException if *row* is **nil** or empty, *qualifier* is **nil**, or *entity* is **nil**.

The expression created with this method does not use table aliases because Enterprise Objects Framework assumes that all INSERT, UPDATE, and DELETE statements are single-table operations. As a result, all keys in *qualifier* should be simple key names; no key paths are allowed. To generate UPDATE statements

that do use table aliases, you must override **prepareUpdateExpressionWithRow:qualifier:** to send a **setUseAliases:**YES message prior to invoking **super**'s version.

See also: – **setUseAliases:**

useBindVariables

+ (BOOL)**useBindVariables**

Returns YES if instances use bind variables, NO otherwise. For more information on bind variables, see the discussion in the class description.

See also: + **setUseBindVariables:**

useQuotedExternalNames

+ (BOOL)**useQuotedExternalNames**

Returns YES if instances use quoted external names, NO otherwise.

See also: + **setUseQuotedExternalNames:**, – **sqlStringForSchemaObjectName:**, – **externalNameQuoteCharacter**

Instance Methods

addBindVariableDictionary:

– (void)**addBindVariableDictionary:**(NSMutableDictionary *)*binding*

Adds *binding* to the receiver's array of bind variable dictionaries. *binding* is generally created using the method **bindVariableDictionaryForAttribute:value:** and is added to the receiver's bind variable dictionaries in **sqlStringForValue:attributeNamed:** when the receiver uses a bind variable for the specified attribute. See the method description for **bindVariableDictionaryForAttribute:value:** for a description of the contents of a bind variable dictionary, and for more information on bind variables, see the discussion in the class description.

See also: – **bindVariableDictionaries**

addInsertListAttribute:value:

– (void)**addInsertListAttribute:**(EOAttribute *)*attribute* **value:**(NSString *)*value*

Adds the SQL string for *attribute* to a comma-separated list of attributes and *value* to a comma-separated list of values. Both lists are constructed for use in an INSERT statement. Use the methods **listString** and **valueList** to access the attributes and value lists.

Invokes **appendItem:toListString:** to add an SQL string for *attribute* to the receiver's **listString**, and again to add a formatted SQL string for *value* to the receiver's **valueList**.

See also: – **sqlStringForAttribute:**, – **sqlStringForValue:attributeNamed:**, + **formatValue: forAttribute:**

addJoinClause

– (void)**addJoinClauseWithLeftName:**(NSString *)*leftName* **rightName:**(NSString *)*rightName* **joinSemantic:**(EOJoinSemantic)*semantic*

Creates a new join clause by invoking **assembleJoinClauseWithLeftName:rightName:joinSemantic:** and adds it to the receiver's join clause string. Separates join conditions already in the join clause string with the word "and". Invoked from **joinExpression**.

See also: **joinClauseString**

addOrderByAttributeOrdering:

– (void)**addOrderByAttributeOrdering:**(EOSortOrdering *)*sortOrdering*

Adds an attribute-direction pair ("LAST_NAME asc", for example) to the receiver's ORDER BY string. If *sortOrdering*'s selector is **c**ompareCaseInsensitiveAscending**:** or **c**ompareCaseInsensitiveDescending**:**, the string generated has the format "upper(attribute) direction". Use the method **orderByString** to access the ORDER BY string. **addOrderByAttributeOrdering:** invokes **appendItem:toListString:** to add the attribute-direction pair.

See also: **sqlStringForAttributeNamed:**

addSelectListAttribute:

– (void)**addSelectListAttribute:**(EOAttribute *)*attribute*

Adds an SQL string for *attribute* to a comma-separated list of attribute names for use in a SELECT statement. The SQL string for *attribute* is formatted with *attribute*'s "read" format. Use **listString** to access the list. **addSelectListAttribute:** invokes **appendItem:toListString:** to add the attribute name.

See also: – **sqlStringForAttribute:**, + **formatSQLString:format:**, **– readFormat** (EOAttribute)

addUpdateListAttribute:value:

– (void)**addUpdateListAttribute:**(EOAttribute *)*attribute* **value:**(NSString *)*value*

Adds a attribute-value assignment ("LAST_NAME = 'Thomas'", for example) to a comma-separated list for use in an UPDATE statement. Formats *value* with *attribute*'s "write" format. Use **listString** to access the list. **addUpdateListAttribute:value:** invokes **appendItem:toListString:** to add the attribute-value assignment.

See also: + **formatSQLString:format:**

aliasesByRelationshipPath

– (NSMutableDictionary *)**aliasesByRelationshipPath**

Returns a dictionary of table aliases. The keys of the dictionary are relationship paths—"department" and "department.location", for example. The values are the table aliases for the corresponding table—"t1" and "t2", for example. The **aliasesByRelationshipPath** dictionary always has at least one entry: an entry for the EOSQLExpression's entity. The key of this entry is the empty string (@"") and the value is "t0". The dictionary returned from this method is built up over time with successive calls to **sqlStringForAttributePath:**.

See also: – **tableListWithRootEntity:**

appendItem:toListString:

– (void)**appendItem:**(NSString *)*itemString* **toListString:**(NSMutableString *)*listString*

Adds *itemString* to a comma-separated list. If *listString* already has entries, this method appends a comma followed by *itemString*. Invoked from **addSelectListAttribute:**, **addInsertListAttribute:value:**, **addUpdateListAttribute:value:**, and **addOrderByAttributeOrdering:**

assembleDeleteStatementWithQualifier:tableList:whereClause:

– (NSString *)**assembleDeleteStatementWithQualifier:**(EOQualifier *)*qualifier*
tableList:(NSString *)*tableList*
whereClause:(NSString *)*whereClause*

Invoked from **prepareDeleteExpressionForQualifier:** to return an SQL DELETE statement of the form:

```
DELETE FROM tableList
SQL_WHERE whereClause
```

qualifier is the argument to **prepareDeleteExpressionForQualifier:** from which *whereClause* was derived. It is provided for subclasses that need to generate the WHERE clause in a particular way.

assembleInsertStatementWithRow:tableList:columnList:valueList:

– (NSString *)**assembleInsertStatementWithRow:**(NSDictionary *)*row*
tableList:(NSString *)*tableList*
columnList:(NSString *)*columnList*
valueList:(NSString *)*valueList*

Invoked from **prepareInsertExpressionWithRow:** to return an SQL INSERT statement of the form:

```
INSERT INTO tableList (columnList)
VALUES valueList
```

or, if *columnList* is **nil**:

```
INSERT INTO tableList
VALUES valueList
```

row is the argument to **prepareInsertExpressionWithRow:** from which *columnList* and *valueList* were derived. It is provided for subclasses that need to generate the list of columns and values in a particular way.

assembleJoinClauseWithLeftName:rightName:joinSemantic:

– (NSString *)**assembleJoinClauseWithLeftName:**(NSString *)*leftName*
rightName:(NSString *)*rightName*
joinSemantic:(EOJoinSemantic)*semantic*

Returns a join clause of the form:

```
leftName operator rightName
```

Where operator is “=” for an inner join, “*=” for a left-outer join, and “=*” for a right-outer join. Invoked from **addJoinClause**.

assembleSelectStatementWithAttributes:lock:qualifier:fetchOrder: selectString:columnList:tableList:whereClause:joinClause: orderByClause:lockClause:

– (NSString *)**assembleSelectStatementWithAttributes:**(NSArray *)*attributes*
lock:(BOOL)*lock*
qualifier:(EOQualifier *)*qualifier*
fetchOrder:(NSArray *)*fetchOrder*
selectString:(NSString *)*selectString*
columnList:(NSString *)*columnList*
tableList:(NSString *)*tableList*
whereClause:(NSString *)*whereClause*
joinClause:(NSString *)*joinClause*
orderByClause:(NSString *)*orderByClause*
lockClause:(NSString *)*lockClause*

Invoked from **prepareSelectExpressionWithAttributes:lock:fetchSpecification:** to return an SQL SELECT statement of the form:

```
SELECT columnList
FROM tableList lockClause
WHERE whereClause AND joinClause
ORDER BY orderByClause
```

If *lockClause* is **nil**, it is omitted from the statement. Similarly, if *orderByClause* is **nil**, the "ORDER BY *orderByClause*" is omitted. If either *whereClause* or *joinClause* is **nil**, the "AND" and **nil**-valued argument are omitted. If both are **nil**, the entire WHERE clause is omitted.

attributes, *lock*, *qualifier*, and *fetchOrder* are the arguments to **prepareSelectExpressionWithAttributes: lock:fetchSpecification:** from which the other **assembleSelect...** arguments were derived. They are provided for subclasses that need to generate the clauses of the SELECT statement in a particular way.

assembleUpdateStatementWithRow:qualifier:tableList:updateList:whereClause:

– (NSString *)**assembleUpdateStatementWithRow:**(NSDictionary *)*row*
qualifier:(EOQualifier *)*qualifier*
tableList:(NSString *)*tableList*
updateList:(NSString *)*updateList*
whereClause:(NSString *)*whereClause*

Invoked from **prepareUpdateExpressionWithRow:qualifier:** to return an SQL UPDATE statement of the form:

```
UPDATE tableList
SET updateList
WHERE whereClause
```

row and *qualifier* are the arguments to **prepareUpdateExpressionWithRow:qualifier:** from which *updateList* and *whereClause* were derived. They are provided for subclasses that need to generate the clauses of the UPDATE statement in a particular way.

bindVariableDictionaries

– (NSArray *)**bindVariableDictionaries**

Returns the receiver's bind variable dictionaries. For more information on bind variables, see the discussion in the class description.

See also: – **addBindVariableDictionary:**

bindVariableDictionaryForAttribute:value:

– (NSMutableDictionary *)**bindVariableDictionaryForAttribute:**(EOAttribute *)*attribute* **value:** (id)*value*

Implemented by subclasses to create and return the bind variable dictionary for *attribute* and *value*. The dictionary returned from this method must contain at least the following key-value pairs:

Key	Value
EOBindVariableNameKey	the name of the bind variable for *attribute*
EOBindVariablePlaceHolderKey	the placeholder string used in the SQL statement
EOBindVariableAttributeKey	*attribute*
EOBindVariableValueKey	*value*

An adaptor subclass may define additional entries as required by its RDBMS.

Invoked from **sqlStringForValue:attributeNamed:** when the message **mustUseBindVariableForAttribute:***attribute* returns YES or when the receiver's class uses bind variables and the message **shouldUseBindVariableForAttribute:***attribute* returns YES. For more information on bind variables, see the discussion in the class description.

A subclass that uses bind variables should implement this method without invoking EOSQLExpression's implementation. The subclass implementation must return a dictionary with entries for the keys listed above and may add additional keys.

See also: – **bindVariableDictionaryForAttribute:value:**, + **useBindVariables**

entity

– (EOEntity *)**entity**

Returns the receiver's entity.

See also: – **initWithEntity:**

externalNameQuoteCharacter

– (NSString *)**externalNameQuoteCharacter**

Returns the string '\"' (an escaped quote character) if the receiver uses quoted external names, or the empty string ("") otherwise.

See also: + **useQuotedExternalNames**, – **sqlStringForSchemaObjectName:**

initWithEntity:

– **initWithEntity:**(EOEntity *)*entity*

Initializes a new instance of EOSQLExpression with *entity.*

See also: – **entity**

joinClauseString

– (NSMutableString *)**joinClauseString**

Returns the part of the receiver's WHERE clause that specifies join conditions. Together, the **joinClauseString** and the **whereClauseString** make up a statement's WHERE clause. If the receiver's statement doesn't contain join conditions, this method returns an empty string.

An EOSQLExpression's **joinClauseString** is generally set by invoking **joinExpression**.

See also: – **addJoinClause**

joinExpression

– (void)**joinExpression**

Builds up the **joinClauseString** for use in a SELECT statement. For each relationship path in the **aliasesByRelationshipPath** dictionary, this method invokes **addJoinClause** for each of the relationship's EOJoin objects.

If the **aliasesByRelationshipPath** dictionary only has one entry (the entry for the EOSQLExpression's entity), the **joinClauseString** is empty.

You must invoke this method *after* invoking **addSelectListAttribute:** for each attribute to be selected and after sending **sqlStringForSQLExpression:self** to the qualifier for the SELECT statement. (These methods build up the **aliasesByRelationshipPath** dictionary by invoking **sqlStringForAttributePath:**.)

See also: – **whereClauseString**, **– sqlStringForSQLExpression:** (EOQualifierSQLGeneration protocol)

listString

– (NSMutableString *)**listString**

Returns a comma-separated list of attributes or "attribute = value" assignments. **listString** is built up with successive invocations of **addInsertListAttribute:value:**, **addSelectListAttribute:**, or **addUpdateListAttribute:value:** for INSERT statements, SELECT statements, and UPDATE statements, respectively. The contents of **listString** vary according to the type of statement the receiver is building:

Type of Statement	Sample listString Contents
INSERT	FIRST_NAME, LAST_NAME, EMPLOYEE_ID
UPDATE	FIRST_NAME = "Timothy", LAST_NAME = "Richardson"
SELECT	t0.FIRST_NAME, t0.LAST_NAME, t1.DEPARTMENT_NAME

lockClause

– (NSString *)**lockClause**

Overridden by subclasses to return the SQL string used in a SELECT statement to lock selected rows. A concrete subclass of EOSQLExpression must override this method to return the string used by its adaptor's RDBMS.

mustUseBindVariableForAttribute:

– (BOOL)**mustUseBindVariableForAttribute:**(EOAttribute *)*attribute*

Returns YES if the receiver must use bind variables for *attribute*, NO otherwise. EOSQLExpression's implementation returns NO. An SQL expression subclass that uses bind variables should override this method to return YES if the underlying RDBMS requires that bind variables be used for attributes with *attribute*'s external type.

See also: – **shouldUseBindVariableForAttribute:**, – **bindVariableDictionaryForAttribute:value:**

orderByString

– (NSMutableString *)**orderByString**

Returns the comma-separated list of "attribute direction" pairs ("LAST_NAME asc, FIRST_NAME asc", for example) for use in a SELECT statement.

See also: – **addOrderByAttributeOrdering:**

prepareDeleteExpressionForQualifier:

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

Generates a DELETE statement by performing the following steps:

1. Sends an **sqlStringForSQLExpression:self** message to *qualifier* to generate the receiver's **whereClauseString**.
2. Invokes **tableListWithRootEntity:** to get the table name for the FROM clause.
3. Invokes **assembleDeleteStatementWithQualifier:tableList:whereClause:**.

See also: + **deleteStatementWithQualifier:entity:**

prepareInsertExpressionWithRow:

– (void)**prepareInsertExpressionWithRow:**(NSDictionary *)*row*

Generates an INSERT statement by performing the following steps:

1. Invokes **addInsertListAttribute:value:** for each entry in *row* to prepare the comma-separated list of attributes and the corresponding list of values.
2. Invokes **tableListWithRootEntity:** to get the table name.
3. Invokes **assembleInsertStatementWithRow:tableList:columnList:valueList:**.

See also: + **insertStatementForRow:entity:**

prepareSelectExpressionWithAttributes:lock:fetchSpecification:

– (void)**prepareSelectExpressionWithAttributes:**(NSArray *)*attributes*
lock:(BOOL)*flag*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification*

Generates a SELECT statement by performing the following steps:

1. Invokes **addSelectListAttribute:** for each entry in *attributes* to prepare the comma-separated list of attributes.

2. Sends an **sqlStringForSQLExpression:self** message to *fetchSpecification*'s qualifier to generate the receiver's **whereClauseString**.
3. Invokes **addOrderByAttributeOrdering:** for each EOAttributeOrdering object in *fetchSpecification*. First conjoins the qualifier in *fetchSpecification* with the restricting qualifier, if any, of the receiver's entity.
4. Invokes **joinExpression** to generate the receiver's **joinClauseString**.
5. Invokes **tableListWithRootEntity:** to get the comma-separated list of tables for the FROM clause.
6. If *flag* is YES, invokes **lockClause** to get the SQL string to lock selected rows.
7. Invokes **assembleSelectStatementWithAttributes:lock:qualifier:fetchOrder: selectString:columnList: tableList:whereClause:joinClause: orderByClause:lockClause:**.

See also: + **selectStatementForAttributes:lock:fetchSpecification:entity:**

prepareUpdateExpressionWithRow:qualifier:

– (void)**prepareUpdateExpressionWithRow:**(NSDictionary *)*row* **qualifier:**(EOQualifier *)*qualifier*

Generates an UPDATE statement by performing the following steps:

1. Invokes **addUpdateListAttribute:value:** for each entry in *row* to prepare the comma-separated list of "attribute = value" assignments.
2. Sends an **sqlStringForSQLExpression:self** message to *qualifier* to generate the receiver's **whereClauseString**.
3. Invokes **tableListWithRootEntity:** to get the table name for the FROM clause.
4. Invokes **assembleUpdateStatementWithRow:qualifier:tableList:updateList:whereClause:**.

See also: + **updateStatementForRow:qualifier:entity:**

setStatement:

– (void)**setStatement:**(NSString *)*string*

Sets the receiver's SQL statement to *string*, which should be a valid expression in the target query language. Use this method—instead of a **prepare...** method—to directly assign an SQL string to an EOSQLExpression object. This method does not perform substitutions or formatting of any kind.

See also: + **expressionForString:**, – **statement**

setUseAliases:

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

Tells the receiver whether or not to use table aliases.

See also: – **useAliases**

shouldUseBindVariableForAttribute:

– (BOOL)**shouldUseBindVariableForAttribute:**(EOAttribute *)*attribute*

Returns YES if the receiver can provide a bind variable dictionary for *attribute*, NO otherwise. Bind variables aren't used for values associated with this attribute when the class method **useBindVariables** returns NO. EOSQLExpression's implementation returns NO. An SQL expression subclass should override this method to return YES if the receiver should use bind variables for attributes with *attribute*'s external type. It should also return YES for any attribute for which the receiver must use bind variables.

See also: – **mustUseBindVariableForAttribute:**

sqlStringForAttribute:

– (NSString *)**sqlStringForAttribute:**(EOAttribute *)*attribute*

Returns the SQL string for *attribute*, complete with a table alias if the receiver uses table aliases. Invoked from **sqlStringForAttributeNamed:** when the attribute name is not a path.

See also: – **sqlStringForAttributePath:**

sqlStringForAttributeNamed:

– (NSString *)**sqlStringForAttributeNamed:**(NSString *)*name*

Returns the SQL string for the attribute named *name*, complete with a table alias if the receiver uses table aliases. Generates the return value using **sqlStringForAttributePath:** if *name* is an attribute path ("department.name", for example); otherwise, uses **sqlStringForAttribute:**.

sqlStringForAttributePath:

– (NSString *)**sqlStringForAttributePath:**(NSArray *)*path*

Returns the SQL string for *path*, complete with a table alias if the receiver uses table aliases. Invoked from **sqlStringForAttributeNamed:** when the specified attribute name is a path ("department.location.officeNumber", for example). *path* is an array of any number of EORelationship objects followed by an EOAttribute object. The EORelationship and EOAttribute objects each correspond

to a component in path. For example, if the attribute name argument to **sqlStringForAttributeNamed:** is "department.location.officeNumber", *path* is an array containing the following objects in the order listed:

- The EORelationship object in the receiver's entity named "department". (Assume the relationship's destination entity is named "Department".)
- The EORelationship object in the Department entity named "location". (Assume the relationship's destination entity is named "Location".)
- The EOAttribute object in the Location entity named "officeNumber".

Assuming that the receiver uses aliases and the alias for the Location table is t2, the SQL string for this sample attribute path is "t2.officeNumber".

If the receiver uses table aliases, this method has the side effect of adding a "relationship path"-"alias name" entry to the **aliasesByRelationship** dictionary.

See also: – **sqlStringForAttribute:**, – **aliasesByRelationshipPath**

sqlStringForConjoinedQualifiers:

– (NSString *)**sqlStringForConjoinedQualifiers:**(NSArray *)*qualifiers*

Creates and returns an SQL string that is the result of interposing the word "AND" between the SQL strings for the qualifiers in *qualifiers*. Generates an SQL string for each qualifier by sending **sqlStringForSQLExpression:** messages to the qualifiers with **self** as the argument. If the SQL string for a qualifier contains only white space, it isn't included in the return value. The return value is enclosed in parentheses if the SQL strings for two or more qualifiers were ANDed together.

sqlStringForDisjoinedQualifiers:

– (NSString *)**sqlStringForDisjoinedQualifiers:**(NSArray *)*qualifiers*

Creates and returns an SQL string that is the result of interposing the word "OR" between the SQL strings for the qualifiers in *qualifiers*. Generates an SQL string for each qualifier by sending **sqlStringForSQLExpression:** messages to the qualifiers with **self** as the argument. If the SQL string for a qualifier contains only white space, it isn't included in the return value. The return value is enclosed in parentheses if the SQL strings for two or more qualifiers were ORed together.

sqlStringForKeyComparisonQualifier:

– (NSString *)**sqlStringForKeyComparisonQualifier:**(EOKeyComparisonQualifier *)*qualifier*

Creates and returns an SQL string that is the result of interposing an operator between the SQL strings for the right and left keys in *qualifier*. Determines the SQL operator by invoking **sqlStringForSelector:value:** with *qualifier*'s selector and **nil** for the value. Generates SQL strings for *qualifier*'s keys by invoking

sqlStringForAttributeNamed: to get SQL strings. This method also formats the strings for the right and left keys using **formatSQLString:format:** with the corresponding attributes' "read" formats.

sqlStringForKeyValueQualifier:

– (NSString *)**sqlStringForKeyValueQualifier:**(EOKeyValueQualifier *)*qualifier*

Creates and returns an SQL string that is the result of interposing an operator between the SQL strings for *qualifier*'s key and value. Determines the SQL operator by invoking **sqlStringForSelector:value:** with *qualifier*'s selector and value. Generates an SQL string for *qualifier*'s key by invoking **sqlStringForAttributeNamed:** to get an SQL string and **formatSQLString:format:** with the corresponding attribute's "read" format. Similarly, generates an SQL string for qualifier's value by invoking **sqlStringForValue:attributeNamed:** to get an SQL string and **formatValue:forAttribute:** to format it. (First invokes **sqlPatternFromShellPattern:** for the value if *qualifier*'s selector is **isLike:**.)

sqlStringForNegatedQualifier:

– (NSString *)**sqlStringForNegatedQualifier:**(EOQualifier *)*qualifier*

Creates and returns an SQL string that is the result of surrounding the SQL string for *qualifier* in parentheses and appending it to the word "not". For example, if the string for *qualifier* is "FIRST_NAME = 'John'", **sqlStringForNegatedQualifier:** returns the string "not (FIRST_NAME = 'John')".

Generates an SQL string for *qualifier* by sending an **sqlStringForSQLExpression::** message to *qualifier* with **self** as the argument. If the SQL string for *qualifier* contains only white space, this method returns **nil**.

sqlStringForSchemaObjectName:

– (NSString *)**sqlStringForSchemaObjectName:**(NSString *)*name*

Returns *name* enclosed in the external name quote character if the receiver uses quoted external names, otherwise simply returns *name* unaltered.

See also: + **useQuotedExternalNames**, – **externalNameQuoteCharacter**

sqlStringForSelector:value:

– (NSString *)**sqlStringForSelector:**(SEL)*selector* **value:**(id)*value*

Returns an SQL operator for *selector* and *value*. The following table summarizes EOSQLExpression's default mapping:

Selector	SQL Operator
isEqualTo:	"is" if value is an EONull, "=" otherwise
isNotEqualTo:	"is not" if *value* is an EONull, "<> otherwise
isLessThan:	"<"
isGreaterThan:	">"
isLessThanOrEqualTo:	"<="
isGreaterThanOrEqualTo:	">="
isLike:	"like"

Raises an NSInternalInconsistencyException if selector is an unknown operator.

See also: – **sqlStringForKeyComparisonQualifier:**, – **sqlStringForKeyValueQualifier:**

sqlStringForValue:attributeNamed:

– (NSString *)**sqlStringForValue:**(id)*value* **attributeNamed:**(NSString *)*name*

Returns a string for *value* appropriate for use in an SQL statement. If the receiver uses a bind variable for the attribute named *name*, then **sqlStringForValue:attributeNamed:** gets the bind variable dictionary for the attribute, adds it to the receiver's array of bind variables dictionaries, and returns the value for the binding's EOBindVariablePlaceHolderKey. Otherwise, this method invokes **formatValue:forAttribute:** and returns the formatted string for *value*.

See also: – **mustUseBindVariableForAttribute:**, – **shouldUseBindVariableForAttribute:**, + **useBindVariables**, – **bindVariableDictionaries**, – **addBindVariableDictionary:**

statement

– (NSString *)**statement**

Returns the complete SQL statement for the receiver. An SQL statement can be assigned to an EOSQLExpression object directly using the class method **expressionForString:** or using the instance method **setStatement:**. Generally, however, an EOSQLExpression's statement is built up using one of the following methods:

- – prepareSelectExpressionWithAttributes:lock:fetchSpecification:
- – prepareInsertExpressionWithRow:
- – prepareUpdateExpressionWithRow:qualifier:
- – prepareDeleteExpressionForQualifier:

tableListWithRootEntity:

– (NSString *)**tableListWithRootEntity:**(EOEntity *)*entity*

Returns the comma-separated list of tables for use in a SELECT, UPDATE, or DELETE statement's FROM clause. If the receiver doesn't use table aliases, the table list consists only of the table name for *entity*—"EMPLOYEE", for example. If the receiver does use table aliases (only in SELECT statements by default), the table list is a comma separated list of table names and their aliases, for example:

```
EMPLOYEE t0, DEPARTMENT t1
```

tableListWithRootEntity: creates a string containing the table name for *entity* and a corresponding table alias ("EMPLOYEE t0", for example). For each entry in **aliasesByRelationshipPath**, this method appends a new table name and table alias.

See also: – **useAliases**, – **aliasesByRelationshipPath**

useAliases

– (BOOL)**useAliases**

Returns YES if the receiver generates statements with table aliases, NO otherwise. For example, the following SELECT statement uses table aliases:

```
SELECT t0.FIRST_NAME, t0.LAST_NAME, t1.NAME
FROM EMPLOYEE t0, DEPARTMENT t1
WHERE t0.DEPARTMENT_ID = t1.DEPARTMENT_ID
```

The EMPLOYEE table has the alias t0, and the DEPARTMENT table has the alias t1.

By default, EOSQLExpression uses table aliases only in SELECT statements. Enterprise Objects Framework assumes that INSERT, UPDATE, and DELETE statements are single-table operations. For more information, see the discussion in the class description.

See also: – **setUseAliases:**, – **aliasesByRelationshipPath**

valueList

– (NSMutableString *)**valueList**

Returns the comma-separated list of values used in an INSERT statement. For example, the value list for the following INSERT statement:

```
INSERT EMPLOYEE (FIRST_NAME, LAST_NAME, EMPLOYEE_ID, DEPARTMENT_ID, SALARY)
VALUES ('Shaun', 'Hayes', 1319, 23, 4600)
```

is "'Shaun', 'Hayes', 1319, 23, 4600". An EOSQLExpression's **valueList** is generated a value at a time with **addInsertListAttribute:value:** messages.

whereClauseString

– (NSString *)**whereClauseString**

Returns the part of the receiver's WHERE clause that qualifies rows. The whereClauseString does not specify join conditions; the **joinClauseString** does that. Together, the **whereClauseString** and the **joinClauseString** make up a statement's where clause. For example, a qualifier for an Employee entity specifies that a statement only affects employees who belong to the Finance department and whose monthly salary is greater than $4500. Assume the corresponding where clause looks like this:

```
WHERE EMPLOYEE.SALARY > 4500 AND DEPARTMENT.NAME = 'Finance'
    AND EMPLOYEE.DEPARTMENT_ID = DEPARTMENT.DEPARTMENT_ID
```

EOSQLExpression generates both a **whereClauseString** and a **joinClauseString** for this qualifier. The **whereClauseString** qualifies the rows and looks like this:

```
EMPLOYEE.SALARY > 4500 AND DEPARTMENT.NAME = 'Finance'
```

The **joinClauseString** specifies the join conditions between the EMPLOYEE table and the DEPARTMENT table and looks like this:

```
EMPLOYEE.DEPARTMENT_ID = DEPARTMENT.DEPARTMENT_ID
```

An EOSQLExpression's **whereClauseString** is generally set by sending a **sqlStringForSQLExpression:** message to an EOQualifier object.

See also: – **sqlStringForSQLExpression:** (EOQualifierSQLGeneration protocol)

EOSQLExpression

Building Expressions

The following four methods create EOSQLExpression objects for the four basic database operations—select, insert, update, and delete:

- + selectStatementForAttributes:lock:fetchSpecification:entity:
- + insertStatementForRow:entity:
- + updateStatementForRow:qualifier:entity:
- + deleteStatementWithQualifier:entity:

Unless you're implementing an EOSQLExpression subclass, these and the class method **expressionForString:** are the only EOSQLExpression methods you should ever need. If, on the other hand, you are creating a subclass, you need to understand the mechanics of how EOSQLExpression builds SQL statements. Each of the creation methods above creates an EOSQLExpression, initializes the expression with a specified entity, and sends the new expression object one of the following **prepare...** methods:

- – prepareSelectExpressionWithAttributes:lock:fetchSpecification:
- – prepareInsertExpressionWithRow:
- – prepareUpdateExpressionWithRow:qualifier:
- – prepareDeleteExpressionForQualifier:

The **prepare...** methods, in turn, invoke a corresponding **assemble...** method, first generating values for the **assemble...** method's arguments. The **assemble...** methods:

- – assembleSelectStatementWithAttributes:lock:qualifier:fetchOrder: selectString:columnList:tableList: whereClause:joinClause: orderByClause:lockClause:
- – assembleInsertStatementWithRow:tableList:columnList:valueList:
- – assembleUpdateStatementWithRow:qualifier:tableList:updateList:whereClause:
- – assembleDeleteStatementWithQualifier:tableList:whereClause:

combine their arguments into SQL statements that the database server can understand.

These three sets of methods establish a framework in which SQL statements are generated. The bulk of the remaining methods generate pieces of an SQL statement.

An individual SQL statement is constructed by combining the SQL strings for any model or value objects specified in the "build" method in the appropriate form. An SQL string for a modeling or value object is a string representation of the object that the database understands; for example, the SQL string for an EOEntity is ultimately its table name. An EOSQLExpression gets the SQL strings for attributes and values with the methods **sqlStringForAttributeNamed:** and **sqlStringForValue:attributeNamed:**. If necessary, it also formats the SQL strings according to an EOAttribute's "read" or "write" format with the class method **formatSQLString:format:**.

Each of the "build" methods above invokes a number of instance methods. These methods are documented individually below.

Using Table Aliases

By default, EOSQLExpression uses table aliases in SELECT statements. For example, the following SELECT statement uses table aliases:

```
SELECT t0.FIRST_NAME, t0.LAST_NAME, t1.NAME
FROM EMPLOYEE t0, DEPARTMENT t1
WHERE t0.DEPARTMENT_ID = t1.DEPARTMENT_ID
```

The EMPLOYEE table is aliased t0, and the DEPARTMENT table is aliased t1. Table aliases are necessary in some SELECT statements—when a table contains a self-referential relationship, for example. Assume the EMPLOYEE table contains a manager column. Managers are also employees, so to retrieve all the employees whose manager is Bob Smith, the SELECT statement looks like this:

```
SELECT t0.FIRST_NAME, t0.LAST_NAME
FROM EMPLOYEE t0, EMPLOYEE t1
WHERE t1.FIRST_NAME = "BOB" AND t1.LAST_NAME = "SMITH" AND
    t0.MANAGER_ID = t1.EMPLOYEE_ID
```

When the Framework maps operations on enterprise objects to operations on database rows, it reduces insert, update, and delete operations to one or more single-table operations. As a result, EOSQLExpression assumes that INSERT, UPDATE, and DELETE statements are always single-table operations, and does not use table aliases in the statements of these types.

In addition, if EOSQLExpression detects that all the attributes in a SELECT statement's attribute list are flattened attributes and they're all flattened from the same table, the expression doesn't use table aliases. For example, suppose that an EOSQLExpression object is created to select a customer's credit card. In the application, a customer object has a credit card object as one of its properties, and all operations on credit cards are described in terms of a customer. As a result, the expression object is initialized with the entity for the Customer object. Rather than create a statement like the following:

```
SELECT t1.TYPE, t1.NUMBER, t1.EXPIRATION, t1.CREDIT_LIMIT, t1.CUSTOMER_ID
FROM CUSTOMER t0, CREDIT_CARD t1
WHERE t1.CUSTOMER_ID = t0.CUSTOMER_ID AND t1.CUSTOMER_ID = 459
```

EOSQLExpression detects that all the attributes correspond to columns in the CREDIT_CARD table and creates the following statement:

```
SELECT TYPE, NUMBER, EXPIRATION, CREDIT_LIMIT, CUSTOMER_ID
FROM CREDIT_CARD
WHERE CUSTOMER_ID = 459
```

Bind Variables

Some RDBMS client libraries use bind variables. A bind variable is a placeholder used in an SQL statement that is replaced with an actual value after the database server determines an execution plan. If you are writing an adaptor for a database server that uses bind variables, you must override the following EOSQLExpression variables:

- – bindVariableDictionaryForAttribute:value:
- – mustUseBindVariableForAttribute:
- – shouldUseBindVariableForAttribute:

If your adaptor doesn’t need to use bind variables, the default implementations of the bind variable methods are sufficient.

EOSQLQualifier

Inherits From: EOQualifier : NSObject

Conforms To: EOQualifierSQLGeneration, NSObject (NSObject)

Declared In: EOAccess/EOSQLQualifier.h

Class Description

EOSQLQualifier is a subclass of EOQualifier that contains unstructured text that can be transformed into an SQL expression. EOSQLQualifier is provided for backwards compatibility with pre-2.0 Enterprise Objects Framework releases and to provide a way to create SQL expressions with any arbitrary SQL. EOSQLQualifier formats are not parsed, they simply perform substitution for keys and format characters. The qualifying information is expressed in the database server's query language (nearly always SQL), and you're responsible for ensuring that the query language statement is valid for your database server. EOSQLQualifiers can't be evaluated against objects in memory. As a result, you should use EOQualifier whenever possible and only use EOSQLQualifier in cases that absolutely require it.

You create an SQL qualifier using **alloc...** and **initWithEntity:qualifierFormat:**. This method takes as arguments the root entity for the qualifier and a format string like that used with the standard creation method **qualifierWithQualifierFormat:**.

Note: Because an SQL qualifier must be rooted to an entity, you can't use **qualifierWithQualifierFormat:** to create EOSQLQualifier objects.

Adopted Protocols

EOQualifierSQLGeneration

– schemaBasedQualifierWithRootEntity:
– sqlStringForSQLExpression:

Class Methods

qualifierWithQualifierFormat:

+ (EOQualifier *)**qualifierWithQualifierFormat:**(NSString *)*format*, ...

Raises an exception. An EOSQLQualifier must be created with an entity, and this method does not provide one. Use **alloc...** and **initWithEntity:qualifierFormat:** to create an EOSQLQualifier.

Instance Methods

initWithEntity:qualifierFormat:

– **initWithEntity:**(EOEntity *)*entity* **qualifierFormat:**(NSString *)*qualifierFormat*, ...

Initializes a newly allocated EOSQLQualifier rooted in *entity* and built from a format string. *qualifierFormat* is a **printf()**-style format string like that used with EOQualifier's **qualifierWithQualifierFormat:** method. This is the designated initializer for the EOSQLQualifier class. Returns **self** if *qualifierFormat* is successfully parsed, **nil** otherwise.

EOStoredProcedure

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: EOAccess/EOStoredProcedure.h

Class Description

An EOStoredProcedure represents a stored procedure defined in a database, and associates a name internal to the Framework with an external name by which the stored procedure is known to the database. If a stored procedure has arguments, its EOStoredProcedure object also maintains a group of EOAttributes which represent the stored procedure's arguments. See the EOAttribute class specification for more information

You usually define stored procedures in your EOModel with the EOModeler application, which is documented in the *Enterprise Objects Framework Developer's Guide*. EOStoredProcedures are primarily used by the Enterprise Objects Framework to map operations for an EOEntity to stored procedures (see the description for EOEntity's **setStoredProcedure:forOperation:** method). You can assign stored procedures to an entity for any of the following scenarios:

- Fetching all the objects for the entity
- Fetching a single object by its primary key
- Inserting a new object
- Deleting an object
- Generating a new primary key

Your code probably won't use EOStoredProcedures unless you're working at the adaptor level.

Like the other major modeling classes, EOStoredProcedure provides a user dictionary for your application to store any application-specific information related to the stored procedure.

Method Types

Creating a new EOStoredProcedure

– initWithName:

Accessing the model

– model

Accessing the name

– setName:
– beautifyName
– name

Accessing the external name

– setExternalName:
– externalName

Accessing the arguments

– setArguments:
– arguments

Accessing the user dictionary

– setUserInfo:
– userInfo

Instance Methods

arguments

– (NSArray *)**arguments**

Returns the EOAttribute objects that describe the stored procedure's arguments or **nil** if the stored procedure has no arguments.

beautifyName

– (void)**beautifyName**

Renames the receiver's name and its arguments to conform to the Framework's naming conventions. For example, "NAME" is renamed "name" and "FIRST_NAME" is renamed "firstName".

See also: – **setArguments:**, – **beautifyNames** (EOModel)

externalName

– (NSString *)**externalName**

Returns the name of the stored procedure as it is defined in the database, or **nil** if the receiver doesn't have an external name.

See also: – **setExternalName:**

initWithName:

– (EOStoredProcedure *)**initWithName:**(NSString *)*name*

The designated initializer for EOStoredProcedure, this method initializes a new EOStoredProcedure object and sets its name to *name*. Returns **self**.

See also: – **setName:**, – **name**

model

– (EOModel *)**model**

Returns the model to which the receiver belongs.

See also: – **addStoredProcedure:** (EOModel)

name

– (NSString *)**name**

Returns the name of the receiver.

See also: – **setName:**, – **initWithName:**

setArguments:

– (void)**setArguments:**(NSArray *)*arguments*

Sets *arguments* as the array of EOAttributes that describe the receiver's arguments. The EOAttribute objects in *arguments* must be ordered to match the database stored procedure definition.

See also: – **arguments**

setExternalName:

– (void)**setExternalName:**(NSString *)*name*

Sets the external name of the stored procedure to *name*. *name* should be the name of the stored procedure as it is defined in the database.

See also: – **externalName**

setName:

– (void)**setName:**(NSString *)*name*

Sets the name of the receiver.

See also: – **name**, – **initWithName:**

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types (that is, NSString, NSDictionary, NSArray, and NSData).

See also: – **userInfo**

userInfo

– (NSDictionary *)**userInfo**

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

See also: – **setUserInfo:**

NSString Additions

Inherits From: NSObject

Declared In: EOAccess/EOEntity.h

Class Description

The access layer adds two methods to the NSString class, to enable the conversion of modeling object names to database schema names, and database schema names to modeling object names.

Class Methods

externalNameForInternalName:separatorString:useAllCaps:

+ (NSString *)**externalNameForInternalName:**(NSString *)*name*,
separatorString:(NSString *)*separatorString*,
useAllCaps:(BOOL)*useAllCaps*)

Used by the Framework to convert modeling object names to database schema names that conform to a standard convention. A conforming database schema name is upper-case and uses "_" to separate words. Consequently "name" becomes "NAME" and "firstName" becomes "FIRST_NAME".

separatorString is a character that is used to separate words. The Framework uses "_" by default as in the examples above. *useAllCaps* indicates whether to capitalize the name. For example, providing NO converts "firstName" to "first_name".

nameForExternalName:separatorString:initialCaps:

+ (NSString *)**nameForExternalName:**(NSString *)*name*,
separatorString:(NSString *)*separatorString*,
initialCaps:(BOOL)*initialCaps*)

Used by name beautification to convert database schema names to modeling object names that conform to a standard convention. A conforming attribute, relationship, or stored procedure name is lower-case except for the initial letter of each embedded word other than the first. Consequently "NAME" becomes "name" and "FIRST_NAME" becomes "firstName". A conforming entity is all lower-case except for the initial letter of each word. Consequently "CUSTOMER_ACCOUNT" becomes "CustomerAccount".

separatorString is a character that is used to separate words. The Framework uses "_" by default as in the examples above. *initialCaps* indicates whether to capitalize the first letter of the first word. By default, the Framework uses YES for entities and NO for everything else.

See also: – **beautifyNames** (EOModel), – **beautifyName** (EOAttribute, EOEntity, EORelationship, EOStoredProcedure)

EOAdaptorChannelDelegate

Adopted By: EOAdaptorChannel delegate objects

Declared In: EOAccess/EOAdaptorChannel.h

Protocol Description

EOAdaptorChannel sends messages to its delegate for nearly every operation that would affect data in the database server. The delegate can use these methods to preempt these operations, modify their results, or simply track activity.

Instance Methods

adaptorChannelDidChangeResultSet:

– (void)**adaptorChannelDidChangeResultSet:**(id)*channel*

Invoked from **fetchRowWithZone:** when a select operation resulted in multiple result sets. This method tells the delegate that the next invocation of **fetchRowWithZone:** will fetch from the next result set. This method is invoked when **fetchRowWithZone:** returns **nil** and there are still result sets left to fetch. The delegate can invoke **setAttributesToFetch:** to prepare for fetching the new rows.

adaptorChannel:didEvaluateExpression:

– (void)**adaptorChannel:**(id)*channel*
didEvaluateExpression:(EOSQLExpression *)*expression*

Invoked from **evaluateExpression:** to tell the delegate that a query language expression has been evaluated by the database server.

adaptorChannel:didExecuteStoredProcedure:withValues:

– (void)**adaptorChannel:**(id)*channel*
didExecuteStoredProcedure:(EOStoredProcedure *)*procedure*
withValues:(NSDictionary *)*values*

Invoked from **executeStoredProcedure:withValues:** after *procedure* is executed successfully.

adaptorChannel:didFetchRow:

– (void)**adaptorChannel:**(id)*channel* **didFetchRow:**(NSMutableDictionary *)*row*

Invoked from **fetchRowWithZone:** after a row is fetched successfully. This method is not invoked if an exception occurs during the fetch or if the same returns **nil** because there are no more rows in the current result set. The delegate may modify *row*, which will be returned from **fetchRowWithZone:**.

adaptorChannelDidFinishFetching:

– (void)**adaptorChannelDidFinishFetching:**(id)*channel*

Invoked from **fetchRowWithZone:** to tell the delegate that fetching is finished for the current select operation. This method is invoked when a fetch ends in **fetchRowWithZone:** because there are no more result sets.

adaptorChannel:didPerformOperations:exception:

– (NSException *)**adaptorChannel:**(id)*channel*
didPerformOperations:(NSArray *)*operations*
exception:(NSException *)*exception*

Invoked from **performAdaptorOperations:**. *exception* is **nil** if no exception was raised while *operations* were performed. Otherwise, *exception* is the raised exception. The delegate can return the same or a different exception, which is re-raised by **performAdaptorOperations:**, or it can return **nil** to prevent the adaptor channel from raising an exception.

adaptorChannel:didSelectAttributes:fetchSpecification:lock:entity:

– (void)**adaptorChannel:**(id)*channel*
didSelectAttributes:(NSArray *)*attributes*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification*
lock:(BOOL)*flag*
entity:(EOEntity *)*entity*

Invoked from **selectAttributes:fetchSpecification:lock:entity:** to tell the delegate that rows have been selected in the database server.

adaptorChannelShouldConstructStoredProcedureReturnValues:

– (NSDictionary *)**adaptorChannelShouldConstructStoredProcedureReturnValues:**(id)*channel*

Invoked from **returnValuesForLastStoredProcedureInvocation** to tell the delegate that *channel* is constructing return values for the last stored procedure evaluated. If the delegate returns a value other than **nil**, that value will be returned immediately from **returnValuesForLastStoredProcedureInvocation**.

adaptorChannel:shouldEvaluateExpression:

– (BOOL)**adaptorChannel:**(id)*channel*
shouldEvaluateExpression:(EOSQLExpression *)*expression*

Invoked from **evaluateExpression:** to tell the delegate that *channel* is sending an expression to the database server. The delegate returns YES to permit the adaptor channel to send *expression* to the server. If the delegate returns NO, the adaptor channel does not send the expression and returns immediately. When the delegate returns NO, the adaptor channel expects that the implementor of the delegate has done the work that **evaluateExpression:** would have done. The delegate can create a new EOSQLExpression and send the expression itself before returning NO.

adaptorChannel:shouldExecuteStoredProcedure:withValues:

– (NSDictionary *)**adaptorChannel:**(id)*channel*
shouldExecuteStoredProcedure:(EOStoredProcedure *)*procedure*
withValues:(NSDictionary *)*values*

Invoked from **executeStoredProcedure:withValues:** to tell the delegate that *channel* is executing a stored procedure. If the delegate returns a value other than **nil**, that value is used as the arguments to the stored procedure instead of *values*.

adaptorChannel:shouldReturnValuesForStoredProcedure:

– (NSDictionary *)**adaptorChannel:**(id)*channel*
shouldReturnValuesForStoredProcedure:(NSDictionary *)*returnValues*

Invoked from **returnValuesForLastStoredProcedureInvocation** to tell the delegate that *channel* is returning values for a stored procedure. If the delegate returns a value other than **nil**, that value is returned from **returnValuesForLastStoredProcedureInvocation** instead of *returnValues*.

adaptorChannel:shouldSelectAttributes:fetchSpecification:lock:entity:

– (BOOL)**adaptorChannel:**(id)*channel*
shouldSelectAttributes:(NSArray *)*attributes*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification*
lock:(BOOL)*flag*
entity:(EOEntity *)*entity*

Invoked from **selectAttributes:fetchSpecification:lock:entity:** to ask the delegate whether a select operation should be performed. The delegate should not modify *fetchSpecification*. Instead, if the delegate wants to perform a different select it should invoke **selectAttributes:fetchSpecification:lock:entity:** itself with a new fetch specification, and return NO (indicating that the adaptor channel should not perform the select itself).

adaptorChannelWillFetchRow:

– (void)**adaptorChannelWillFetchRow:**(id)*channel*

Invoked from **fetchRowWithZone:** to tell the delegate that a single row will be fetched. The delegate can determine the attributes used by the fetch by sending **attributesToFetch** to *channel*, and can change the set of attributes to fetch by sending **setAttributesToFetch:** to *channel*. The adaptor channel performs the actual fetch.

adaptorChannel:willPerformOperations:

– (NSArray *)**adaptorChannel:**(id)*channel* **willPerformOperations:**(NSArray *)*operations*

Invoked from **performAdaptorOperations:** to tell the delegate that *channel* is performing the EOAdaptorOperations in *operations*. The delegate may return *operations* or a different NSArray for the adaptor channel to perform. If the delegate returns **nil**, the adaptor channel does not perform the operations and returns from the method immediately.

EOAdaptorContextDelegate

Adopted By: EOAdaptorContext delegate objects

Declared In: EOAccess/EOAdaptorContext.h

Protocol Description

EOAdaptorContext sends messages to its delegate for any transaction begin, commit, or rollback. The delegate can use these methods to preempt these operations, modify their results, or simply track activity.

Instance Methods

adaptorContextDidBegin:

– (void)**adaptorContextDidBegin:***context*

Invoked from **beginTransaction** to tell the delegate that a transaction has begun.

adaptorContextDidCommit:

– (void)**adaptorContextDidCommit:***context*

Invoked from **commitTransaction** to tell the delegate that a transaction has been committed.

adaptorContextDidRollback:

– (void)**adaptorContextDidRollback:***context*

Invoked from **rollbackTransaction** to tell the delegate that a transaction has been rolled back.

adaptorContextShouldBegin:

– (BOOL)**adaptorContextShouldBegin:***context*

Invoked from **beginTransaction** to tell the delegate that *context* is beginning a transaction. If this method returns NO, the adaptor context does not begin a transaction. Return YES to allow the adaptor context to begin a transaction.

adaptorContextShouldCommit:

– (BOOL)**adaptorContextShouldCommit:***context*

Invoked from **commitTransaction** to tell the delegate that *context* is committing a transaction. If this method returns NO, the adaptor context does not commit the transaction. Return YES to allow the adaptor context to commit.

Note that if you implement this delegate method to return NO, your delegate must perform the database COMMIT itself; the rest of the Enterprise Objects Framework assumes that the commit has taken place. **adaptorContextShouldCommit:** doesn't specify whether or not the commit should take place; it only specifies whether or not the adaptor context should do it for you.

adaptorContextShouldConnect:

– (BOOL)**adaptorContextShouldConnect:***context*

Invoked before the adaptor attempts to connect. The delegate can return NO if it wants to override the connect, YES if it wants the adaptor to attempt to connect in the usual way. The delegate should raise an exception if it fails to connect.

adaptorContextShouldRollback:

– (BOOL)**adaptorContextShouldRollback:***context*

Invoked from **rollbackTransaction** to tell the delegate that *context* is rolling back a transaction. If this method returns NO, the adaptor context does not roll back the transaction. Return YES to allow the adaptor context to roll back.

EOAdaptorDelegate

Adopted By: EOAdaptor delegate objects

Declared In: EOAccess/EOAdaptor.h

Protocol Description

The delegate for EOAdaptor can implement the method **adaptor:fetchedValueForValue:attribute:** to perform a database-specific transformations on a value.

Instance Methods

adaptor:fetchedValueForValue:attribute:

– (id)**adaptor:**(EOAdaptor *)*adaptor*
fetchedValueForValue:(id)*value*
attribute:(EOAttribute *)*attribute*

Invoked from **fetchedValueForValue:attribute:** to allow the delegate to perform a database-specific transformation on *value*. The delegate should return the value that the adaptor's database server would ultimately store for *value* if it was inserted or updated in the column described by *attribute.*

Ordinarily, **fetchedValueForValue:attribute:** invokes one of the type-specific **fetchedValue...** methods depending on the type of *value*. If you implement this delegate method, **fetchedValueForValue:attribute:** does not invoke the other **fetchedValue...** methods. It simply invokes your delegate method and returns the value returned from it. Therefore, an implementation of **adaptor:fetchedValueForValue:attribute:** must handle values of all types.

EOCustomClassArchiving

(informal protocol)

Category Of: NSObject

Declared In: EOAccess/EOAttribute.h

Category Description

EOCustomClassArchiving defines methods that can be used to write any object that conforms to NSCoding to the database as binary data, as generated by NSArchiver. Since data in this fomat is neither human-readable nor readable by non-OpenStep applications, it's usually preferable to supply other custom archiving methods for your custom value classes. For a comprehensive discussion of working with custom data types, see the EOAttribute class specification and the chapter "Advanced Enterprise Object Modeling" in the *Enterprise Objects Framework Developer's Guide*.

For more information, see "Archiving Attributes with Custom Types" in "WebObjects Programming Topics." Is this cross-reference correct?

Class Methods

objectWithArchiveData:

+ (id)**objectWithArchiveData:**(NSData *)*data*

Returns an object created from *data*. NSObject's implementation of this method invokes NSUnarchiver's **unarchiveObjectWithData:** method and returns the result. Your custom value class can therefore take advantage of this method merely by implementing the NSCoding protocol method **initWithCoder:**.

See also: – **archiveData**

Instance Methods

archiveData

– (NSData *)**archiveData**

Return the receiver's value as an NSData object whose bytes can be stored in an external repository. NSObject's implementation of this method invokes NSArchiver's **archivedDataWithRootObject:** method and returns the result. Your custom value class can therefore take advantage of this method merely by implementing the NSCoding protocol method **encodeWithCoder:**.

See also: **+ objectWithArchiveData:**

EODatabaseContextDelegation

Declared In: EOAccess/EODatabaseContext.h

Protocol Description

An EODatabaseContext shares its delegate with its EODatabaseChannels. These delegate methods are actually sent from EODatabaseChannel, but they're defined in EODatabaseContext for ease of access:

– databaseContext:didSelectObjectsWithFetchSpecification:databaseChannel:

– databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:

– databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID:databaseChannel:

– databaseContext:shouldUsePessimisticLockWithFetchSpecification: databaseChannel:

You can use the EODatabaseContext delegate methods to intervene when objects are created and when they're fetched from the database. This gives you more fine-grained control over such issues as how an object's primary key is generated (**databaseContextNewPrimaryKeyForObjectdatabaseContext: newPrimaryKeyForObject:entity:**), how and if objects are locked (**databaseContextShouldLockObjectWithGlobalIDdatabaseContext: shouldLockObjectWithGlobalID:snapshot:**), what fetch specification is used to fetch objects (**databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:**), how batch faulting is performed (**databaseContext:shouldFetchArrayFault:** and **databaseContext: shouldFetchObjectFault:**), and so on. For more information, see the individual delegate method descriptions.

Instance Methods

databaseContext:didFetchObjects:fetchSpecification:editingContext:

– (void)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
didFetchObjects:(NSArray *)*objects*
fetchSpecification:(EOFetchSpecification *)*fetchSpecification* **editingContext:**
(EOEditingContext *)*anEditingContext*

Invoked from **objectsWithFetchSpecification:editingContext:** after *aDatabaseContext* fetches *objects* using the criteria defined in *fetchSpecification* on behalf of *anEditingContext*.

See also: – **databaseContext:shouldFetchObjectFault:**

databaseContext:didSelectObjectsWithFetchSpecification:databaseChannel:

– (void)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
didSelectObjectsWithFetchSpecification:(EOFetchSpecification *)*fetchSpecification*
databaseChannel:(EODatabaseChannel *)*channel*

Invoked from the EODatabaseChannel method **selectObjectsWithFetchSpecification:editingContext:** to tell the delegate that *channel* selected the objects on behalf of *aDatabaseContext* as specified by *fetchSpecification*.

See also: – **databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:**

databaseContext:failedToFetchObject:globalID:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
failedToFetchObject:(id)*object*
globalID:(EOGlobalID *)*globalID*

Sent when a to-one fault cannot find its data in the database. The *object* is a cleared fault identified by *globalID*. If this method returns YES, *aDatabaseContext* assumes that the delegate has handled the situation to its satisfaction, in whatever way it deemed appropriate (for example, by displaying an alert panel or initializing a fault object with new values). If it returns NO or if the delegate method is not implemented, *aDatabaseContext* raises an NSObjectNotAvailableException.

databaseContext:newPrimaryKeyForObject:entity:

– (NSDictionary *)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
newPrimaryKeyForObject:(id)*object*
entity:(EOEntity *)*entity*

Sent when a newly inserted enterprise *object* doesn't already have a primary key set. This delegate method can be used to implement custom primary key generation. If the delegate is not implemented or returns **nil**, then *aDatabaseContext* will send an EOAdaptorChannel a **primaryKeyForNewRowWithEntity:** message in an attempt to generate the key.

The dictionary you return from this delegate method contains the attribute or attributes (if *object* has a compound primary key) that make up *object*'s primary key.

databaseContext:shouldFetchArrayFault:

– (BOOL)**databaseContext:**(EODatabaseContext *)*databaseContext* **shouldFetchArrayFault:**
(id)*fault*

Invoked when a fault is fired, this delegate method lets you fine-tune the behavior of batch faulting. Delegates can fetch the array themselves (for example, by using the EODatabaseContext method

batchFetchRelationship:forSourceObjects:editingContext:) and return NO, or return YES to allow the *databaseContext* to do the fetch itself. If *databaseContext* performs the fetch it will batch fault according to the batch count on the relationship being fetched.

See also: – **databaseContext:shouldFetchObjectFault:**

databaseContext:shouldFetchObjectFault:

– (BOOL)**databaseContext:**(EODatabaseContext *)*databaseContext* **shouldFetchObjectFault:**(id)*fault*

Invoked when a fault is fired, this delegate method lets you fine-tune the behavior of batch faulting. Delegates can fetch the fault themselves (for example, by using the EODatabaseContext method **objectsWithFetchSpecification:editingContext:**) and return NO, or return YES to allow *databaseContext* to perform the fetch. If *databaseContext* performs the fetch, it will batch fault according to the batch count on the entity being fetched.

See also: – **databaseContext:shouldFetchArrayFault:**

databaseContext:shouldFetchObjectsWithFetchSpecification:editingContext:

– (NSArray *)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldFetchObjectsWithFetchSpecification:(EOFetchSpecification *)*fetchSpecification*
editingContext:(EOEditingContext *)*anEditingContext*

Invoked from **objectsWithFetchSpecification:editingContext:** to give the delegate the opportunity to satisfy *anEditingContext*'s fetch request (using the criteria specified in *fetchSpecification*) from a local cache. If the delegate returns **nil**, *aDatabaseContext* performs the fetch. Otherwise, the returned array is returned as the fetch result.

See also: **databaseContextDidFetchObjectsdatabaseContext:didFetchObjects:fetchSpecification:editingContext:**

databaseContext:shouldInvalidateObjectWithGlobalID:snapshot:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldInvalidateObjectWithGlobalID:(EOGlobalID *)*globalId*
snapshot:(NSDictionary *)*snapshot*

Invoked from **invalidateObjectsWithGlobalIDs:**. Delegate can cause *aDatabaseContext*'s object as identified by *globalID* to not be invalidated and that object's *snapshot* to not be cleared by returning NO.

databaseContext:shouldLockObjectWithGlobalID:snapshot:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldLockObjectWithGlobalID:(EOGlobalID *)*globalID*
snapshot:(NSDictionary *)*snapshot*

Invoked from **lockObjectWithGlobalID:editingContext:**. The delegate should return YES if it wants the operation to proceed or NO if it doesn't. Values from *snapshot* are used to create a qualifier from the attributes used for locking specified for the object's entity (that is, the object identified by *globalID*). Delegates can override the locking mechanism by implementing their own locking procedure and returning NO. Methods that override the locking mechanism should raise an exception on the failure to lock exactly one object.

databaseContext:shouldRaiseExceptionForLockFailure:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldRaiseExceptionForLockFailure:(NSException *)*exception*

Invoked from **lockObjectWithGlobalID:editingContext:**. This method allows the delegate to suppress an *exception* that has occurred during *aDatabaseContext*'s attempt to lock the object.

databaseContext:shouldSelectObjectsWithFetchSpecification:databaseChannel:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldSelectObjectsWithFetchSpecification:(EOFetchSpecification *)*fetchSpecification*
databaseChannel:(EODatabaseChannel *)*channel*

Invoked from the EODatabaseChannel method **selectObjectsWithFetchSpecification:editingContext:** to tell the delegate that *channel* will select objects on behalf of *aDatabaseContext* as specified by *fetchSpecification*. The delegate should not modify *fetchSpecification*'s qualifier or fetch order. If the delegate returns YES the channel will go ahead and select the object; if the delegate returns NO (possibly after issuing custom SQL against the adaptor) the *channel* will skip the select and return.

databaseContext:shouldUpdateCurrentSnapshot:newSnapshot:globalID: databaseChannel:

– (NSDictionary *)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
shouldUpdateCurrentSnapshot:(NSDictionary *)*currentSnapshot*
newSnapshot:(NSDictionary *)*newSnapshot*
globalID:(EOGlobalID *)*globalID*
databaseChannel:(EODatabaseChannel *)*channel*

Invoked from the EODatabaseChannel method **fetchObject** when *aDatabaseContext* already has a snapshot (*currentSnapshot*) for a row fetched from the database. This method is invoked without first

checking whether the snapshots are equivalent (the check would be too expensive to do in the common case), so the receiver may be passed equivalent snapshots. The default behavior is to not update an older snapshot with *newSnapshot*. The delegate can override this behavior by returning a dictionary (possibly *newSnapshot*) that will be recorded as the updated snapshot. This will result in *aDatabaseContext* broadcasting an EOObjectsChangedInStoreNotification, causing the object store hierarchy to invalidate existing objects (as identified by *globalID*) built from the obsolete snapshot. Returning **nil** raises an exception. You can use this method to achieve the same effect as using an EOFetchSpecification with **setRefreshesRefetchedObjects:** set to YES—that is, it allows you to overwrite in-memory object values with values from the database that may have been changed by someone else.

Returning *currentSnapshot* causes the *aDatabaseContext* to perform the default behavior (that is, not updating the older snapshot).

databaseContext:shouldUsePessimisticLockWithFetchSpecification: databaseChannel:

– (BOOL)**databaseContext:**(EODatabaseContext *)*databaseContext*
shouldUsePessimisticLockWithFetchSpecification:(EOFetchSpecification *)*fetchSpecification*
databaseChannel:(EODatabaseChannel *)*channel*

Invoked from the EODatabaseChannel method **selectObjectsWithFetchSpecification:editingContext:** regardless of the update strategy specified on *channel*'s *databaseContext*. The delegate should not modify the qualifier or fetch order contained in *fetchSpecification*. If the delegate returns YES the channel locks the rows being selected; if the delegate returns NO the channel selects the rows without locking.

databaseContext:willOrderAdaptorOperationsFromDatabaseOperations:

– (NSArray *)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
willOrderAdaptorOperationsFromDatabaseOperations:(NSArray *)*databaseOperations*

Sent from **performChanges**. If the delegate responds to this message, it must return an array of EOAdaptorOperations that *aDatabaseContext* can then submit to an EOAdaptorChannel for execution. The delegate can fabricate its own array by asking each of the *databaseOperations* for its list of EOAdaptorOperations, and adding them to the array which will eventually be returned by this method. The delegate is free to optimize, order, or transform the list in whatever way it deems necessary. This method is useful for applications that need a special ordering of the EOAdaptorOperations so as not to violate any database referential integrity constraints.

databaseContext:willPerformAdaptorOperations:adaptorChannel:

– (NSArray *)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
willPerformAdaptorOperations:(NSArray *)*adaptorOperations*
adaptorChannel:(EOAdaptorChannel *)*adaptorChannel*

Sent from **performChanges**. The delegate can return a new *adaptorOperations* array which *aDatabaseContext* will hand to *adaptorChannel* for execution in place of the old array of EOAdaptorOperations. This method is useful for applications that need a special ordering of the EOAdaptorOperations so as not to violate any database referential integrity constraints.

databaseContext:willRunLoginPanelToOpenDatabaseChannel:

– (BOOL)**databaseContext:**(EODatabaseContext *)*aDatabaseContext*
willRunLoginPanelToOpenDatabaseChannel:(EODatabaseChannel *)*channel*

When *aDatabaseContext* is about to use a *channel*, it checks to see if the *channel*'s corresponding EOAdaptorChannel is open. If it isn't, it attempts to open the EOAdaptorChannel by sending it an **openChannel** message. If that doesn't succeed, *aDatabaseContext* will ask the EOAdaptorChannel's adaptor to run the login panel and open the channel. *aDatabaseContext* gives the delegate a chance to intervene in this by invoking this delegate method. The delegate can return NO to stop *aDatabaseContext* from running the login panel. In this case, the delegate is responsible for opening the channel. If the delegate returns YES, *aDatabaseContext* runs the login panel.

EOEditingContext Additions

(informal protocol)

Category Of: EOEditingContext

Declared In: EOAccess/EOUtilities.h

Class Description

EOEditingContext Additions is a collection of convenience methods intended to make common operations with EOF easier. EOEditingContext Additions is a category on EOEditingContext provided in EOAccess.

Note: The Objective-C source code for EOUtilities is available as an example. On Mac OS X Server systems, see **/System/Developer/Examples/EnterpriseObjects/Sources/EOUtilities**. On NT, see $***NEXT_ROOT*\Developer\Examples\EnterpriseObjects\Sources\EOUtilities**.

Method Types

Fetching multiple objects

– objectsForEntityNamed:
– objectsForEntityNamed:qualifierFormat:
– objectsMatchingValue:forKey:entityNamed:
– objectsMatchingValues:entityNamed:
– objectsOfClass:
– objectsWithFetchSpecificationNamed:entityNamed:bindings:

Fetching single objects

– objectForEntityNamed:qualifierFormat:
– objectMatchingValue:forKey:entityNamed:
– objectMatchingValues:entityNamed:
– objectWithFetchSpecificationNamed:entityNamed:bindings:
– objectWithPrimaryKey:entityNamed:
– objectWithPrimaryKeyValue:entityNamed:

Fetching raw rows

– executeStoredProcedureNamed:arguments:
– objectFromRawRow:entityNamed:
– rawRowsForEntityNamed:qualifierFormat:
– rawRowsMatchingValue:forKey:entityNamed:
– rawRowsMatchingValues:entityNamed:
– rawRowsWithSQL:modelNamed:
– rawRowsWithStoredProcedureNamed:arguments:

Accessing the EOF stack

– connectWithModelNamed:connectionDictionaryOverrides:
– databaseContextForModelNamed:

Accessing object data

– destinationKeyForSourceObject:relationshipNamed:
– localInstanceOfObject:
– localInstancesOfObjects:
– primaryKeyForObject:

Accessing model information

– entityForClass:
– entityForObject:
– entityNamed:
– modelGroup

Instance Methods

connectWithModelNamed:connectionDictionaryOverrides:

– (void)**connectWithModelNamed:**(NSString *)*modelName*
connectionDictionaryOverrides:(NSDictionary *)*overrides*

Connects to the database using the connection information in the specified model and the provided overrides dictionary. This method facilitates per-session database logins in WebObjects applications. Typically, you'd put a login name and password in the overrides dictionary and otherwise use the values in the model's connection dictionary. Raises an exception if the connection failed.

databaseContextForModelNamed:

– (EODatabaseContext *)**databaseContextForModelNamed:**(NSString *)*entityName*

Returns the database context used to service the specified model.

destinationKeyForSourceObject:relationshipNamed:

– (NSDictionary *)**destinationKeyForSourceObject:**(id)*object*
relationshipNamed:(NSString *)*entityName*

Returns the foreign key for the rows at the destination entity of the specified relationship. As an example, given entities Department and Employee with a relationship called "department" joining

`Department.ID Employee.deptID`, invoking this method on a Department object with ID equal to 5 will return a dictionary with a value of 5 for the deptID key.

See also: – **primaryKeyForObject:**

entityForClass:

– (EOEntity *)**entityForClass:**(Class)*classObject*

Returns the entity associated with the specified class. Raises an exception if the specified entity can't be found or if more than one entity is associated with the class.

See also: – **entityForObject:**, – **entityNamed:**, – **objectsOfClass:**

entityForObject:

– (EOEntity *)**entityForObject:**(id)*object*

Returns the entity associated with the provided enterprise object. Raises an exception if the specified entity can't be found.

See also: – **entityForClass:**, – **entityNamed:**

entityNamed:

– (EOEntity *)**entityNamed:**(NSString *)*entityName*

Returns the entity with the specified name. Raises an exception if the specified entity can't be found.

See also: – **entityForClass:**, – **entityForObject:**

executeStoredProcedureNamed:arguments:

– (NSDictionary *)**executeStoredProcedureNamed:**(NSString *)*storedProcedureName*
arguments:(NSDictionary *)*arguments*

Executes the specified stored procedure with the provided arguments. Returns the stored procedure's return values (if any). Use only with stored procedures that don't return results rows.

See also: – **rawRowsWithStoredProcedureNamed:arguments:**

localInstanceOfObject:

– (id)**localInstanceOfObject:**(id)*object*

Translates the specified enterprise object from another editing context to the specified one.

See also: – **localInstancesOfObjects:**

localInstancesOfObjects:

– (NSArray *)**localInstancesOfObjects:**(NSArray *)*objects*

Translates the specified enterprise objects from another editing context to the specified one.

See also: – **localInstanceOfObject:**

modelGroup

– (EOModelGroup *)**modelGroup**

Returns the model group associated with the editing context's root object store, an EOObjectStoreCoordinator.

objectForEntityNamed:qualifierFormat:

– (id)**objectForEntityNamed:**(NSString *)*entityName* **qualifierFormat:**(NSString *)*format, ...*

Creates a qualifier with the provided format string and arguments, and returns matching enterprise objects. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectsForEntityNamed:qualifierFormat:**, – **rawRowsForEntityNamed:qualifierFormat:**

objectFromRawRow:entityNamed:

– (id)**objectFromRawRow:**(NSDictionary *)*row* **entityNamed:**(NSString *)*entityName*

Fetches and returns the object corresponding to the specified raw row (using EOEditingContext's **faultForRawRow:entityNamed:**). This method can only be used on raw rows that include the row's primary key.

objectMatchingValue:forKey:entityNamed:

– (id)**objectMatchingValue:**(id)*value* **forKey:**(NSString *)*key* **entityNamed:**(NSString *)*entityName*

Creates an EOKeyValueQualifier with the specified key and value and returns matching enterprise objects. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectMatchingValues:entityNamed:**, – **objectsMatchingValue:forKey:entityNamed:**

objectMatchingValues:entityNamed:

– (id)**objectMatchingValues:**(NSDictionary *)*values* **entityNamed:**(NSString *)*entityName*

Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching enterprise objects. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectMatchingValue:forKey:entityNamed:**, – **objectsMatchingValues:entityNamed:**

objectsForEntityNamed:

– (NSArray *)**objectsForEntityNamed:**(NSString *)*entityName*

Fetches and returns the enterprise objects associated with the specified entity.

See also: – **objectsForEntityNamed:qualifierFormat:**, – **objectsMatchingValue:forKey:entityNamed:**, – **objectsMatchingValues:entityNamed:**

objectsForEntityNamed:qualifierFormat:

– (NSArray *)**objectsForEntityNamed:**(NSString *)*entityName*
qualifierFormat:(NSString *)*format, ...*

Creates a qualifier with the provided format string and arguments, and returns matching enterprise objects.

See also: – **objectForEntityNamed:qualifierFormat:**, – **objectsForEntityNamed:**

objectsMatchingValue:forKey:entityNamed:

– (NSArray *)**objectsMatchingValue:**(id)*value*
forKey:(NSString *)*key*
entityNamed:(NSString *)*entityName*

Creates an EOKeyValueQualifier with the specified key and value and returns matching enterprise objects.

See also: – **objectMatchingValue:forKey:entityNamed:**, – **objectsForEntityNamed:**, – **objectsMatchingValues:entityNamed:**

objectsMatchingValues:entityNamed:

– (NSArray *)**objectsMatchingValues:**(NSDictionary *)*values* **entityNamed:**(NSString *)*entityName*

Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching enterprise objects.

See also: – **objectMatchingValues:entityNamed:**, – **objectsForEntityNamed:**, – **objectsMatchingValue:forKey:entityNamed:**

objectsOfClass:

– (NSArray *)**objectsOfClass:**(Class)*classObject*

Fetches and returns the enterprise objects associated with the specified class. Raises an EOMoreThanOneException if more than one entity for the class exists.

See also: – **entityForClass:**

objectsWithFetchSpecificationNamed:entityNamed:bindings:

– (NSArray *)**objectsWithFetchSpecificationNamed:**(NSString *)*fetchSpecName*
entityNamed:(NSString *)*entityName* **bindings:**(NSDictionary *)*bindings*

Fetches and returns the enterprise objects retrieved with the specified fetch specification and bindings.

See also: – **objectWithFetchSpecificationNamed:entityNamed:bindings:**

objectWithFetchSpecificationNamed:entityNamed:bindings:

– (id)**objectWithFetchSpecificationNamed:**(NSString *)*fetchSpecName*
entityNamed:(NSString *)*entityName*
bindings:(NSDictionary *)*bindings*

Fetches and returns the enterprise objects retrieved with the specified fetch specification and bindings. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectsWithFetchSpecificationNamed:entityNamed:bindings:**

objectWithPrimaryKey:entityNamed:

– (id)**objectWithPrimaryKey:**(NSDictionary *)*keyDictionary* **entityNamed:**(NSString *)*entityName*

Fetches and returns the enterprise object identified by the specified primary key dictionary. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectMatchingValue:forKey:entityNamed:**, – **objectWithPrimaryKeyValue:entityNamed:**, – **primaryKeyForObject:**

objectWithPrimaryKeyValue:entityNamed:

– (id)**objectWithPrimaryKeyValue:**(id)*value* **entityNamed:**(NSString *)*entityName*

Fetches and returns the enterprise object identified by the specified primary key value. For use only with enterprise objects that have non-compound primary keys. Raises an EOMoreThanOneException unless exactly one object is retrieved.

See also: – **objectsMatchingValues:entityNamed:**, – **objectWithPrimaryKey:entityNamed:**

primaryKeyForObject:

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

Returns the primary key dictionary for the specified enterprise object.

See also: – **objectWithPrimaryKey:entityNamed:**, – **objectWithPrimaryKeyValue:entityNamed:**

rawRowsForEntityNamed:qualifierFormat:

– (NSArray *)**rawRowsForEntityNamed:**(NSString *)*entityName*
qualifierFormat:(NSString *)*format, ...*;

Creates a qualifier for the specified entity and with the specified qualifier format and returns matching raw row dictionaries.

See also: – **objectsForEntityNamed:qualifierFormat:**, – **rawRowsWithSQL:modelNamed:**

rawRowsMatchingValue:forKey:entityNamed:

– (NSArray *)**rawRowsMatchingValue:**(id)*value*
forKey:(NSString *)*key*
entityNamed:(NSString *)*entityName*

Creates an EOKeyValueQualifier with the specified key and value and returns matching raw rows.

See also: – **objectMatchingValue:forKey:entityNamed:**, – **objectsMatchingValue:forKey:entityNamed:**, – **rawRowsMatchingValues:entityNamed:**

rawRowsMatchingValues:entityNamed:

– (NSArray *)**rawRowsMatchingValues:**(NSDictionary *)*values*
entityNamed:(NSString *)*entityName*

Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching raw rows.

See also: – **objectMatchingValues:entityNamed:**, – **objectsMatchingValues:entityNamed:**, – **rawRowsMatchingValue:forKey:entityNamed:**

rawRowsWithSQL:modelNamed:

– (NSArray *)**rawRowsWithSQL:**(NSString *)*sqlString* **modelNamed:**(NSString *)*modelName*

Evaluates the specified SQL and returns the resulting raw rows.

See also: – **rawRowsForEntityNamed:qualifierFormat:**, – **rawRowsWithStoredProcedureNamed:arguments:**

rawRowsWithStoredProcedureNamed:arguments:

– (NSArray *)**rawRowsWithStoredProcedureNamed:**(NSString *)*storedProcedureName*
arguments:(NSDictionary *)*arguments*

Executes the specified stored procedure with the provided arguments and returns the resulting raw rows.

See also: – **rawRowsWithSQL:modelNamed:**

EOModelGroupClassDelegation

Inherits From: NSObject

Declared In: EOAccess/EOModelGroup.h

Protocol Description

An EOModelGroup object should have a delegate which can influence how it finds and loads models. In addition to the delegates you assign to EOModelGroup instances, the EOModelGroup class itself can have a delegate. The class delegate implements a single method—**defaultModelGroup**.

For more information on EOModelGroup instance delegate methods, see the EOModelGroupDelegation specifications.

Instance Methods

defaultModelGroup

– (EOModelGroup *)**defaultModelGroup**

If implemented by the EOModelGroup class delegate, this method should return the EOModelGroup to be returned in response to the message **defaultModelGroup**. If this delegate method returns**nil**, EOModelGroup uses the default behavior of the **defaultModelGroup** class method.

Note: This method is implemented by the delegate assigned to the EOModelGroup class object.

See also: + **classDelegate** (EOModelGroup class), + **setClassDelegate:** (EOModelGroup class)

EOModelGroupDelegation

Inherits From: NSObject

Declared In: EOAccess/EOModelGroup.h

Protocol Description

An EOModelGroup object should have a delegate which can influence how it finds and loads models. The EOModelGroup instance delegate can implement the methods below:

- **entity:relationshipForRow:relationship:**
- **subEntityForEntity:primaryKey:isFinal:**
- **entity:failedToLookupClassNamed:**
- **entity:classForObjectWithGlobalID:**

In addition to the delegates you assign to EOModelGroup instances, the EOModelGroup class itself can have a delegate. The class delegate implements a single method—**defaultModelGroup**. For more information, see the EOModelGroupClassDelegation.

Instance Methods

entity:classForObjectWithGlobalID:

– (Class)**entity:**(EOEntity *)*entity* **classForObjectWithGlobalID:**(EOGlobalID *)*globalID*

Used to fine-tune inheritance. The delegate can use *globalID* to determine a subclass to be used in place of the one specified in *entity.*

entity:failedToLookupClassNamed:

– (Class)**entity:**(EOEntity *)*entity* **failedToLookupClassNamed:**(NSString *)*className*

Invoked when the class name specified for *entity* cannot be found at run-time. The delegate can take action (such as loading a bundle) to provide *entity* with a class corresponding to *className*. If the delegate cannot provide anything, or if there is no delegate, EOGenericRecord is used.

entity:relationshipForRow:relationship:

– (EORelationship *)**entity:**(EOEntity *)*entity* **relationshipForRow:**(NSDictionary *)*row*
relationship:(EORelationship *)*relationship*

Invoked when relationships are instantiated for a newly fetched object. The delegate can use the information in *row* to determine which entity the target enterprise object should be associated with, and replace the relationship appropriately.

modelGroup:entityNamed:

– (EOModel *)**modelGroup:**(EOModelGroup *)*group* **entityNamed:**(NSString *)*name*

If implemented by the delegate, this method should search the *group* for the entity named *name* and return the entity's EOModel. Return **nil** if *name* is not an entity in *group*.

relationship:failedToLookupDestinationNamed:

– (EOEntity *)**relationship:**(EORelationship *)*relationship* **failedToLookupDestinationNamed:**
(NSString *)*entityName*

Invoked when loading *relationship* and the destination *entityName* specified in the model file cannot be found in the model group. This most often occurs when a model references entities in another model file that can't be found. If the delegate doesn't implement this method, an exception is raised. If the delegate does implement this method, the method's return value is set as the destination entity. if the delegate returns **nil**, the destination entity is set to **nil**.

subEntityForEntity:primaryKey:isFinal:

– (EOEntity *)**subEntityForEntity:**(EOEntity *)*entity*
primaryKey:(NSDictionary *)*primaryKey*
isFinal:(BOOL *)*flag*

Allows the delegate to fine-tune inheritance by indicating from which sub-entity an object should be fetched based on its *primaryKey*. The entity returned must be a sub-entity of *entity*. If the delegate knows that the object should be fetched from the returned entity and not one of its sub-entities, it should set *flag* to YES.

EOPropertyListEncoding

Implemented By: EOAttribute
EOEntity
EORelationship
EOStoredProcedure

Interface Description

The EOPropertyListEncoding protocol declares methods that read and write objects to *property lists*—a dictionary containing only property list data types (that is, NSDictionary objects, NStrings, NSArray objects, and NSData objects).

Classes that implement this protocol must also initialize their instances with **initWithPropertyList:owner:**.

Objects initialized with **initWithPropertyList:owner:** are initialized from *propertyList*. The *owner* argument is optional and should be used only by objects requiring a reference to their owner. The newly created object isn't considered fully functional until it receives an **awakeWithPropertyList** message, which finishes initializing the object. The **awakeWithPropertyList** invocation should be deferred until after all of the objects identified in *propertyList* have been created.

The method **encodeIntoPropertyList:** is responsible for encoding the receiver into a property list for later restoration.

This interface is used to read and write modeling objects (EOModel, EOEntity, EOAttribute, and so on) to a model file.

Methods

awakeWithPropertyList

– (void)**awakeWithPropertyList:**(NSDictionary *)*propertyList*
Finishes initializing the receiver from *propertyList*, which must have been initialized with **initWithPropertyList:owner:**.

awakeWithPropertyList is responsible for restoring references to other objects. Consequently, it should not be invoked until all other objects that the receiver might reference have been initialized from *propertyList*.

encodeIntoPropertyList:

– (void)**encodeIntoPropertyList:**(NSMutableDictionary *)*propertyList*

Returns the receiver as a property list.

initWithPropertyList:owner:

– **initWithPropertyList:**(NSDictionary *)*propertyList* **owner:**(id)*owner*

Intializes a newly-allocated object from a property list. owner is optional, and should be used by objects requiring a back pointer to their owner. This method must be followed by a call to **awakeWithPropertyList** in order to create a fully-functional object. The call to **awakeWithPropertyList** should be deferred until after all other objects have been sent **init** messages.

EOQualifierSQLGeneration

Adopted By: EOAndQualifier, EOKeyComparisonQualifier, EOKeyValueQualifier, EONotQualifier, EOOrQualifier, EOSQLQualifier

Declared In: EOAccess/EOSQLQualifier.h

Protocol Description

The EOQualifierSQLGeneration protocol declares two methods that are adopted by qualifier classes to qualify fetches from a database. One of the methods, **schemaBasedQualifierWithRootEntity:**, is used to provide a qualifier suitable for evaluation by a database from a qualifier suitable for in-memory evaluation. The other method, **sqlStringForSQLExpression:**, is used by concrete subclasses of EOSQLExpression to generate WHERE clauses for SQL statements.

Instance Methods

sqlStringForSQLExpression:

– (NSString *)**sqlStringForSQLExpression:**(EOSQLExpression *)*sqlExpression*

Returns a SQL statement suitable for inclusion in a WHERE clause. Invoked from a concrete subclass of EOSQLExpression while it's preparing a SELECT, UPDATE, or DELETE statement.

See also: – **whereClauseString** (EOSQLExpression)

schemaBasedQualifierWithRootEntity:

– (EOQualifier *)**schemaBasedQualifierWithRootEntity:**(EOEntity *)*entity*

Returns a qualifier suitable for evaluation by a database (as opposed to in-memory evaluation). Invoked by an EODatabaseChannel object before it uses its EOAdaptorChannel to perform a database operation.

Whereas in-memory qualifier evaluation uses pointers to resolve relationships, a database qualifier must use foreign keys. For example, consider the qualifier below that is used to fetch all employees who work in a specified department:

```
Department *dept;    // Assume this exists.
EOQualifier *qualifer;

qualifier = [EOQualifier qualifierWithQualifierFormat:@"department = %@", dept];
```

For an in-memory search, the Framework queries employee objects for their department object and includes an employee in the result list if its department object is equal to **dept**. (See the EOQualifierEvaluation protocol description for more information on in-memory searching.)

For a database search, the Framework needs to qualify the fetch by specifying a foreign key value for **dept**. The Framework sends **qualifier** a **schemaBasedQualifierWithRootEntity:** message that creates and returns a new qualifier. Assume that the entity for employee objects has an attribute named **departmentID** and that the primary key value for **dept** is 459, the resulting qualifier specifies the search conditions as:

```
department.departmentID = 459
```

See also: – **selectObjectsWithFetchSpecification:editingContext:** (EODatabaseChannel)