EOAccess Framework

Java API Reference

The EOAccess Framework

Package: com.apple.yellow.eoaccess

Introduction

The EOAccess framework is one of a group of frameworks known collectively as the Enterprise Objects Framework. The classes and interfaces 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. The remainder of the EOAccess Framework consists of several related groups of classes, a few miscellaneous classes, and a number of interfaces.

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.

- 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.

The Database Level

The database level is where enterprise objects are created from the dictionaries retrieved 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 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.

- 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.

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.

Extensions of EOControl 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.

- EOSQLQualifier is a subclass of EOQualifier that contains unstructured text that can be transformed into an SQL expression.

Delegates

A number of EOAccess classes delegate behavior. The delegate methods are defined in these interfaces:

■ 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 EOModelGroup class delegate implements a method that returns the default model group.

■ An EOModelGroup delegate influences how the model group finds and loads models.

Miscellaneous Classes and Interfaces

■ EOUtilities 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.

EOAccessArrayFaultHandler

Inherits from:	EOAccessGenericFaultHandler :
	EOFaultHandler (EOControl) :
	NSObject
Package:	com.apple.yellow.eoaccess

Class Description

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

Constructors

EOAccessArrayFaultHandler

```
public EOAccessArrayFaultHandler(
    com.apple.yellow.eocontrol.EOKeyGlobalID sourceGID,
    String relationshipName,
    EODatabaseContext databaseContext,
    com.apple.yellow.eocontrol.EOEditingContext editingContext)
```

Returns a handler initialized 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.

Instance Methods

completeInitializationOfObject

```
public void completeInitializationOfObject(Object anObject)
```

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

databaseContext

```
public EODatabaseContext databaseContext()
```

Returns the receiver's database context.

editingContext

`public com.apple.yellow.eocontrol.EOEditingContext editingContext()`

Returns the receiver's editing context.

relationshipName

`public String relationshipName()`

Returns the receiver's relationship name.

sourceGlobalID

`public com.apple.yellow.eocontrol.EOKeyGlobalID sourceGlobalID()`

Returns the receiver's source global ID.

EOAccessFaultHandler

Inherits from:	EOAccessGenericFaultHandler :
	EOFaultHandler (EOControl) :
	NSObject
Package:	com.apple.yellow.eoaccess

Class Description

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

Constructors

EOAccessFaultHandler

```
public EOAccessFaultHandler(
    com.apple.yellow.eocontrol.EOKeyGlobalID globalID,
    EODatabaseContext databaseContext,
    com.apple.yellow.eocontrol.EOEditingContext editingContext)
```

Returns a handler initialized with all of the information necessary to fetch the object when the fault is fired.

Instance Methods

completeInitializationOfObject

```
public void completeInitializationOfObject(Object anObject)
```

Asks the receiver's database context to fetch *anObject* 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 receiver's editing context.

databaseContext

```
public EODatabaseContext databaseContext()
```

Returns the receiver's database context.

editingContext

```
public com.apple.yellow.eocontrol.EOEditingContext editingContext()
```

Returns the receiver's editing context.

globalID

```
public com.apple.yellow.eocontrol.EOKeyGlobalID globalID()
```

Returns the receiver's global ID.

EOAccessGenericFaultHandler

Inherits from:	EOFaultHandler (EOControl) : NSObject
Package:	com.apple.yellow.eoaccess

Class Description

EOAccessGenericFaultHandler is an abstract class that helps an EOAccessFault to fire by fetching data using an EODatabaseContext. Don't use EOAccessGenericFaultHandler 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 `linkAfterHandler` to link one fault after another. Use `next` and `previous` to traverse the chain.

Instance Methods

generation

`public int generation()`

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

linkAfterHandler

```
public void linkAfterHandler(
    EOAccessGenericFaultHandler faultHandler,
    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

```
public EOAccessGenericFaultHandler next()
```

Returns the next fault in the chain.

previous

```
public EOAccessGenericFaultHandler previous()
```

Returns the previous fault in the chain.

EOAdaptor

Inherits from:	NSObject
Package:	com.apple.yellow.eoaccess

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 static 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.

Note: EOAdaptor isn't declared to be abstract, but conceptually it is abstract. Never create instances of the EOAdaptor class.

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:

Method	Description
`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 <u>"Creating an EOAdaptor Subclass"</u> (page 35).

Constants

EOAdaptor defines one constant, a String, as described below:

Constant	Description
`GeneralAdaptorException`	The name of exceptions raised by adaptors when errors occur during interactions with their database servers.

Method Types

Creating an EOAdaptor

```
adaptorWithName
```

```
adaptorWithModel
```

Accessing an adaptor's name

```
name
```

Accessing the names of all available adaptors

```
availableAdaptorNames
```

Connecting to a database server

```
assertConnectionDictionaryIsValid
```

```
connectionDictionary
```

```
setConnectionDictionary
```

```
runLoginPanelAndValidateConnectionDictionary
```

```
runLoginPanel
```

```
isDroppedConnectionException
```

```
handleDroppedConnection
```

Encoding database strings

```
databaseEncoding
```

Performing database-specific transformations on values

```
fetchedValueForValue
```

```
fetchedValueForDataValue
```

```
fetchedValueForDateValue
```

```
fetchedValueForNumberValue
```

```
fetchedValueForStringValue
```

Servicing models

```
canServiceModel
internalTypeForExternalType
externalTypesWithModel:
assignExternalInfoForEntireModel:
assignExternalInfoForEntity
assignExternalInfoForAttribute
isValidQualifierType
```

Creating adaptor contexts

```
createAdaptorContext
contexts
```

Checking connection status

```
hasOpenChannels
```

Accessing a default expression class

```
setExpressionClassName
expressionClass
defaultExpressionClass
```

Accessing an adaptor's login panel

```
sharedLoginPanelInstance
runLoginPanelAndValidateConnectionDictionary
runLoginPanel
```

Accessing the delegate

```
delegate
setDelegate
setDefaultDelegate
defaultDelegate
```

Creating and dropping databases

`createDatabaseWithAdministrativeConnectionDictionary`

`dropDatabaseWithAdministrativeConnectionDictionary`

Providing prototype attributes

`prototypeAttributes`

Synchronizing the database with a model

`objectStoreChangesFromAttributeToAttribute`

Constructors

EOAdaptor

`public EOAdaptor(String `*`name`*`)`

Creates and returns a new EOAdaptor 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, the Oracle adaptor is loaded from the framework `OracleEOAdaptor.framework`. When you create an adaptor subclass, override this method to create a new adaptor with *name*.

Never use this constructor directly. It is invoked automatically from `adaptorWithName` and `adaptorWithModel`—EOAdaptor static methods you use to create a new adaptor.

Static Methods

adaptorWithModel

`public static Object 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. Throws an exception if `model` is `null`, if `model`'s adaptor name is `null`, or if the adaptor named in `model` can't be loaded.

A subclass of EOAdaptor doesn't need to implement this method. A subclass that does implement this method must incorporate the superclass's version.

See Also: `adaptorName` (EOModel), `setConnectionDictionary`

adaptorWithName

`public static Object adaptorWithName(String name)`

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

`EOAdaptor myAdaptor = (EOAdaptor)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`. Throws an exception if `name` is `null` 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

`public static void assignExternalInfoForAttribute(EOAttribute `*`attribute`*`)`

Implemented by adaptor subclasses to assign database-specific characteristics to *`attribute`*. EOAdaptor's implementation invokes `assignExternalTypeForAttribute` to assign an external type, and then it 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.

A subclass of EOAdaptor doesn't need to implement this method. A subclass that does implement this method must incorporate the superclass's version.

See Also: `assignExternalInfoForEntireModel:`

assignExternalInfoForEntireModel:

`public static 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 assigns external types for all the model's attributes.

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

assignExternalInfoForEntity

`public static void assignExternalInfoForEntity(EOEntity `*`entity`*`)`

Implemented 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 implement this method to assign additional database-specific characteristics, if any. A subclass that does implement this method must incorporate the superclass's version.

See Also: `assignExternalInfoForEntireModel:`

assignExternalTypeForAttribute

`public static void assignExternalTypeForAttribute(EOAttribute attribute)`

Overridden by adaptor subclasses to assign the external type to `attribute`. EOAdaptor's implementation does nothing.

An adaptor subclass should implement this method to assign an external type using `attribute`'s internal type, precision, and length information. A subclass that does implement this method should incorporate the superclass's version.

See Also: `assignExternalInfoForEntireModel:`

availableAdaptorNames

`public static NSArray availableAdaptorNames()`

Returns an array containing the names of all available adaptors. If no adaptors are found, this method returns an empty array.

defaultDelegate

`public static Object defaultDelegate()`

Returns the default delegate—the object that is assigned to new adaptor instances as their delegate.

externalTypesWithModel:

```
public static 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*.

An adaptor subclass should implement this method.

internalTypeForExternalType

```
public static String internalTypeForExternalType(
    String extType,
    EOModel model)
```

Implemented by subclasses to return the name of the 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 null if no mapping for *extType* is found.

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

setDefaultDelegate

```
public static void setDefaultDelegate(Object anObject)
```

Sets the default delegate—the object assigned as delegate to all newly created EOAdaptor instances. By default, there is no default delegate.

setExpressionClassName

```
public static void setExpressionClassName(
    String sqlExpressionClassName,
    String adaptorClassName)
```

Sets the expression class for instances of the class named adaptorClassName to `sqlExpressionClassName`. If `sqlExpressionClassName` is `null`, restores the expression class to the default. Throws an exception if `adaptorClassName` is `null` or the empty string.

Use this method to substitute a subclass of EOSQLExpression for the expression class provided by the adaptor.

A subclass of EOAdaptor doesn't need to implement this method. A subclass that does implement this method must incorporate the superclass's version.

See Also: `defaultExpressionClass`

sharedLoginPanelInstance

```
public static EOLoginPanel sharedLoginPanelInstance()
```

Returns the receiver's login panel in applications that have a graphical user interface. Returns `null` 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`.

Instance Methods

assertConnectionDictionaryIsValid

`public 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. Throws an exception if the connection dictionary contains invalid information.

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

See Also: `setConnectionDictionary`

canServiceModel

`public boolean canServiceModel(EOModel `*`model`*`)`

Returns `true` if the receiver can service *model*, `false` otherwise. EOAdaptor's implementation returns `true` 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

`public NSDictionary connectionDictionary()`

Returns the receiver's connection dictionary, or `null` 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.

contexts

```
public NSArray contexts()
```

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

See Also: `createAdaptorContext`

createAdaptorContext

```
public EOAdaptorContext createAdaptorContext()
```

Implemented by subclasses to create and return a new EOAdaptorContext, or `null` if a new context can't be created. A newly created EOAdaptor has no contexts.

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

See Also: `contexts`

createDatabaseWithAdministrativeConnectionDictionary

```
public void createDatabaseWithAdministrativeConnectionDictionary(
   NSDictionary connectionDictionary)
```

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

See Also: `dropDatabaseWithAdministrativeConnectionDictionary`

databaseEncoding

```
public int databaseEncoding()
```

Returns the string encoding used to encode and decode database strings. 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.

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 throws an exception if the receiver's database encoding isn't valid.

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

defaultExpressionClass

```
public 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

delegate

```
public Object delegate()
```

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

dropDatabaseWithAdministrativeConnectionDictionary

```
public 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

```
public 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`. If no class has been set for the receiver's class, this method determines the expression class by sending `defaultExpressionClass` to `this`.

You rarely need to invoke this method yourself. It's invoked by the Framework to determine the class to use to represent query language expressions. You should, however, use this method if you explicitly create EOSQLExpression instances. To be sure you're using the correct expression class, create instances of the class returned from 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`.

fetchedValueForDataValue

```
public NSData fetchedValueForDataValue(
    NSData value,
    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` 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

```
public NSGregorianDate fetchedValueForDateValue(
    NSGregorianDate value,
    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` 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

```
public Number fetchedValueForNumberValue(
    Number value,
    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` 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

```
public String fetchedValueForStringValue(
    String value,
    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` when the value argument is a string.

EOAdaptor's implementation trims trailing spaces and returns `null` for zero-length strings. An adaptor subclass should override this method to perform any additional conversion or formatting on string values.

fetchedValueForValue

```
public Object fetchedValueForValue(
    Object value,
    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` 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` 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 String, Number, NSGregorianDate, NSData, or any of their subclasses), `fetchedValueForValue` returns *value* unchanged.

This method invokes the EOAdaptor.Delegate method `adaptorFetchedValueForValue` which can override the adaptor's default behavior.

See Also: `fetchedValueForDataValue`, `fetchedValueForDateValue`, `fetchedValueForNumberValue`, `fetchedValueForStringValue`, `valueFactoryMethod` **(EOAttribute)**

handleDroppedConnection

`public void handleDroppedConnection()`

Invoked when necessary to clean up after a dropped connection. Sends `handleDroppedConnection` to all of its adaptor contexts and then clears its array of contexts. If the delegate implements `reconnectionDictionaryForAdaptor`, that method is invoked, and the return value is assigned to the adaptor as its new connection dictionary.

You should never invoke this method; it is invoked automatically by the Framework. Subclasses don't normally need to override the superclass implementation.

hasOpenChannels

`public boolean hasOpenChannels()`

Returns `true` if any of the receiver's contexts have open channels, `false` otherwise. A subclass of EOAdaptor doesn't need to override this method.

See Also: `hasOpenChannels` (EOAdaptorContext)

isDroppedConnectionException

```
public boolean isDroppedConnectionException(Throwable aThrowable)
```

Returns `true` if the exception is one that the adaptor can attempt to recover from by reconnecting to the database, `false` otherwise.

Invoked if an exception is raised during fetching or saving. If the adaptor returns `true`, then the adaptor attempts to reconnect to the database and retries the operation. If the reconnection attempt fails, the exception from the failure is raised as usual. If the adaptor returns `false`, reconnection isn't attempted and the exception is raised.

The default implementation of `isDroppedConnectionException:` returns `false`. Subclasses that support database reconnection should implement this method to allow for automatic database reconnection.

See Also: `handleDroppedConnection`, `reconnectionDictionaryForAdaptor` (EOAdaptor.Delegate)

isValidQualifierType

```
public boolean isValidQualifierType(
    String typeName,
    EOModel model)
```

Implemented by subclasses to return `true` if an attribute of type *typeName* can be used in a qualifier (a SQL WHERE clause) sent to the database server, or `false` 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

```
public String 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`

objectStoreChangesFromAttributeToAttribute

```
public NSDictionary objectStoreChangesFromAttributeToAttribute(
    EOAttribute schemaAttribute,
    EOAttribute modelAttribute)
```

Returns a dictionary describing the changes to synchronize *schemaAttribute* (the attribute reflecting the definition of a column in the database) with *modelAttribute* (the attribute as it's defined in the model).

prototypeAttributes

```
public NSArray prototypeAttributes()
```

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

runLoginPanel

```
public NSDictionary runLoginPanel()
```

Runs the adaptor's login panel by sending a `runLoginPanelAndValidateConnectionDictionary` message to the adaptor's login panel object with the validate flag `false`. 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: `setConnectionDictionary`, `assertConnectionDictionaryIsValid`, `sharedLoginPanelInstance`

runLoginPanelAndValidateConnectionDictionary

`public boolean runLoginPanelAndValidateConnectionDictionary()`

Runs the adaptor's login panel by sending a `runPanelForAdaptor` message to the adaptor's login panel object with the validate flag `true`. Returns `true` if the user enters valid connection information, or `false` 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

`public void setConnectionDictionary(NSDictionary dictionary)`

Sets the adaptor's connection dictionary to `dictionary`, which must only contain String, NSData, NSDictionary, and NSArray objects. Throws an exception 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` **(EOLoginPanel)**

setDelegate

`public void setDelegate(Object delegate)`

Sets the receiver's delegate to `delegate`, or removes its delegate if `delegate` is `null`. 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`.

EOAdaptor

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`

- `setConnectionDictionary`

- `setDelegate`

If you override `setConnectionDictionary` or `setDelegate`, 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`

- `assertConnectionDictionaryIsValid`

- `createAdaptorContext`

- `fetchedValueForDataValue`

- `fetchedValueForDateValue`

- `fetchedValueForNumberValue`

- `fetchedValueForStringValue`

- `fetchedValueForValue`

- `isValidQualifierType`

EOAdaptorChannel

Inherits from:	NSObject
Package:	com.apple.yellow.webobjects

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.

Note: EOAdaptorChannel isn't declared to be abstract, but conceptually it is abstract. Never create instances of the EOAdaptorChannel class.

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 `adaptorChannelShouldEvaluateExpression`, let the delegate determine whether the channel should perform an operation. Others, such as `adaptorChannelDidEvaluateExpression`, 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 `adaptorChannelShouldEvaluateExpression`. If it simply wants notification when a transaction has begun, it implements `adaptorChannelDidEvaluateExpression`.

The principal attributes of the EOAdaptorChannel class are:

■ Adaptor context

■ Delegate

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.

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

Method	Description
openChannel	Opens the channel so it can perform database operations.
closeChannel	Close the channel.
selectAttributes	Selects rows matching the specified qualifier.
fetchRow	Fetches a row resulting from the last selectAttributes, executeStoredProcedure, or evaluateExpression.
insertRow	Inserts the specified row.

Method	Description
updateValuesInRowsDescribedByQualifier	Updates the row described by the specified qualifier.
deleteRowDescribedByQualifier	Deletes the row described by the specified qualifier.
executeStoredProcedure	Performs the specified stored procedure.
evaluateExpression	Sends the specified expression to the database.

Method	Description
openChannel	Opens the channel so it can perform database operations.
closeChannel	Close the channel.
selectAttributes	Selects rows matching the specified qualifier.
fetchRow	Fetches a row resulting from the last selectAttributes, executeStoredProcedure, or evaluateExpression.
insertRow	Inserts the specified row.
updateValuesInRowDescribedByQualifier	Updates the row described by the specified qualifier.
deleteRowDescribedByQualifier	Deletes the row described by the specified qualifier.
executeStoredProcedure	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" (page 57).

Constants

EOAdaptorChannel defines several String constants for use as keys and values in an exception's userInfo dictionary (see `performAdaptorOperations`).

Constant	Description
AdaptorOperationsKey	A userInfo dictionary key.
FailedAdaptorOperationKey	A userInfo dictionary key.
AdaptorFailureKey	A userInfo dictionary key.
AdaptorOptimisticLockingFailure	A userInfo dictionary value.

Method Types

Accessing the adaptor context

 adaptorContext

Opening and closing a channel

 openChannel

 closeChannel

 isOpen

Modifying rows

 insertRow

 updateValuesInRowDescribedByQualifier

 updateValuesInRowsDescribedByQualifier

 deleteRowDescribedByQualifier

deleteRowsDescribedByQualifier

lockRowComparingAttributes

Fetching rows

selectAttributes

describeResults

setAttributesToFetch

attributesToFetch

fetchRow

dictionaryWithObjectsForAttributes

cancelFetch

isFetchInProgress

Invoking stored procedures

executeStoredProcedure

returnValuesForLastStoredProcedureInvocation

Assigning primary keys

primaryKeyForNewRowWithEntity

Sending SQL to the server

evaluateExpression

Batch processing operations

performAdaptorOperation

performAdaptorOperations

Accessing schema information

describeTableNames

describeStoredProcedureNames

addStoredProceduresNamed

describeModelWithTableNames

Debugging

    setDebugEnabled

    isDebugEnabled

Accessing the delegate

    delegate

    setDelegate

Constructors

EOAdaptorChannel

`public EOAdaptorChannel(EOAdaptorContext adaptorContext)`

Creates and returns an EOAdaptorChannel, with *adaptorContext*. When you create an adaptor channel subclass, override this method.

Don't 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.

See Also: `adaptorContext`

Instance Methods

adaptorContext

`public EOAdaptorContext adaptorContext()`

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

See Also: `EOAdaptorChannel` **constructor**

addStoredProceduresNamed

```
public void addStoredProceduresNamed(
    NSArray storedProcedureNames,
    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. Throws an exception if an error occurs.

attributesToFetch

```
public NSArray attributesToFetch()
```

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

cancelFetch

```
public void cancelFetch()
```

Implemented by subclasses to clear all result sets established by the last selectAttributes, executeStoredProcedure, or evaluateExpression message and terminate the current fetch, so that isFetchInProgress returns false.

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

closeChannel

```
public 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`, `hasOpenTransaction` **(EOAdaptorContext)**

delegate

```
public Object delegate()
```

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

deleteRowDescribedByQualifier

```
public void deleteRowDescribedByQualifier(
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    EOEntity entity)
```

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

deleteRowsDescribedByQualifier

```
public int deleteRowsDescribedByQualifier(
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    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. Throws 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`, `isOpen`, `isFetchInProgress`, `hasOpenTransaction`
(EOAdaptorContext)

describeModelWithTableNames

```
public 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

```
public NSArray describeResults()
```

Implemented by subclasses to return an array of EOAttributes describing the properties
available in the current result set, as determined by `selectAttributes`, `executeStoredProcedure`,
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

```
public 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` to build a default
model in EOModeler. Throws an exception if an error occurs.

describeTableNames

```
public 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 `null`. An adaptor channel subclass should override this method to construct an array of table names from database metadata.

dictionaryWithObjectsForAttributes

```
public NSMutableDictionary dictionaryWithObjectsForAttributes(
    Object[] objects,
    NSArray attributes)
```

Used by EOAdaptorChannel subclasses to create dictionaries that can be returned from `fetchRow`. 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*.

A subclass of EOAdaptorChannel shouldn't override this method.

evaluateExpression

```
public 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. Throws 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`. 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 `adaptorChannelShouldEvaluateExpression` and `adaptorChannelDidEvaluateExpression`.

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: `fetchRow`

executeStoredProcedure

```
public void executeStoredProcedure(
   EOStoredProcedure storedProcedure,
   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 `fetchRow` to get result rows and `returnValuesForLastStoredProcedureInvocation` to get return arguments and result status, if any. Throws 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` 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

fetchRow

```
public NSMutableDictionary fetchRow()
```

Implemented by subclasses to fetch the next row from the result set of the last `selectAttributes`, `executeStoredProcedure`, 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 `null`, and invokes the delegate method `adaptorChannelDidChangeResultSet` if there are more results sets. When there are no more rows or result sets, this method returns `null`, ends the fetch, and invokes `adaptorChannelDidFinishFetching`. `isFetchInProgress` returns `true` until the fetch is canceled or until this method exhausts all result sets and returns `null`. This method also invoke the delegate methods `adaptorChannelWillFetchRow` and `adaptorChannelDidFetchRow`.Throws an exception if an error occurs.

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

See Also: `setAttributesToFetch`

insertRow

```
public void insertRow(
    NSDictionary row,
    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. Throws 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

```
public boolean isDebugEnabled()
```

Returns `true` if the adaptor channel logs evaluated SQL and other useful information to the console (or to the standard error stream), `false` if not. A subclass of EOAdaptorChannel doesn't need to override this method.

See Also: `setDebugEnabled` **(EOAdaptorContext)**

isFetchInProgress

```
public boolean isFetchInProgress()
```

Implemented by subclasses to return `true` if the receiver is fetching, `false` otherwise. An adaptor channel is fetching if:

■ It's been sent a successful `selectAttributes` message.

- A stored procedure that returns rows has been successfully executed using `executeStoredProcedure`.

- 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: `fetchRow`

isOpen

```
public boolean isOpen()
```

Implemented by subclasses to return `true` if the channel has been opened with `openChannel`, `false` if not. An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See Also: `closeChannel`

lockRowComparingAttributes

```
public void lockRowComparingAttributes(
    NSArray attributes,
    EOEntity entity,
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    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` invokes `selectAttributes` 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 throws an exception. It also throws an exception 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 `null`.) 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 doesn't need to override this method.

openChannel

```
public 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. Throws an exception if an error occurs. An adaptor channel subclass should override this method without invoking EOAdaptorChannel's implementation.

See Also: `isOpen`, `closeChannel`

performAdaptorOperation

```
public 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` using information in `adaptorOperation` to supply the arguments. Throws an exception if an error occurs.

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

See Also: `performAdaptorOperations`

performAdaptorOperations

`public 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 `adaptorChannelWillPerformOperations` and `adaptorChannelDidPerformOperations`.

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

Constant	The corresponding value in the exception's userInfo dictionary
`AdaptorOperationsKey`	An array of the EOAdaptorOperations being executed.
`FailedAdaptorOperationKey`	The particular EOAdaptorOperation that failed.
`AdaptorFailureKey`	If present, offers additional information on the type of error that occurred. Currently, the only possible value for this key is `AdaptorOptimisticLockingFailure`, 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

`public 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 `null` 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 `null`.

EOAdaptorChannel's implementation simply returns `null`. 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

`public NSDictionary returnValuesForLastStoredProcedureInvocation()`

Implemented by subclasses to return stored procedure parameter and return values. Used in conjunction with `executeStoredProcedure`. 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 `null` if the stored procedure has results to fetch. In this case, you must use `fetchRow` 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

```
public void selectAttributes(
    NSArray attributes,
    EOFetchSpecification fetchSpecification,
    boolean flag,
    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 `fetchRow` messages according to fetchSpecification's sort orderings. If flag is `true`, the rows are locked if possible so that no other user can modify them (the lock specification in `fetchSpecification` is ignored). Throws 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

```
public 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 `fetchRow`. This method throws an exception if invoked when there is no fetch in progress.

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

See Also: `selectAttributes`

setDebugEnabled

```
public void setDebugEnabled(boolean flag)
```

Enables debugging in the receiver and all its channels. If `flag` is `true`, 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: `setDebugEnabled` **(EOAdaptorContext)**

setDelegate

```
public void setDelegate(Object delegate)
```

Sets the receiver's delegate to `delegate`, or removes its delegate if `delegate` is `null`. 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`.

updateValuesInRowDescribedByQualifier

```
public void updateValuesInRowDescribedByQualifier(
   NSDictionary values,
   com.apple.yellow.eocontrol.EOQualifier qualifier,
   EOEntity entity)
```

Updates the row described by `qualifier`. Invokes `updateValuesInRowsDescribedByQualifier` and raises an exception unless exactly one row is updated.

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

updateValuesInRowsDescribedByQualifier

```
public int updateValuesInRowsDescribedByQualifier(
    NSDictionary values,
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    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. Throws 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: updateValuesInRowDescribedByQualifier

55

EOAdaptorChannel

Creating an EOAdaptorChannel Subclass

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

■   adaptorContext

■   delegate

■   deleteRowDescribedByQualifier

■   isDebugEnabled

■   lockRowComparingAttributes

■   performAdaptorOperation

■   performAdaptorOperations

■   updateValuesInRowDescribedByQualifier

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

■   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`
- `describeModelWithTableNames`
- `describeResults`
- `describeStoredProcedureNames`
- `describeTableNames`
- `evaluateExpression`
- `executeStoredProcedure`
- `fetchRow`
- `insertRow`
- `isFetchInProgress`
- `isOpen`
- `openChannel`
- `primaryKeyForNewRowWithEntity`
- `returnValuesForLastStoredProcedureInvocation`
- `selectAttributes`
- `setAttributesToFetch`
- `updateValuesInRowsDescribedByQualifier`

EOAdaptorContext

Inherits from:	NSObject
Package:	com.apple.yellow.webobjects

Class Description

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.

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.

Note: EOAdaptorContext isn't declared to be abstract, but conceptually it is abstract. Never create instances of the EOAdaptorContext class.

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.

The EOAdaptorContext class has the following principal attributes:

■ Array of adaptor channels

■ Delegate

■ Adaptor

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.

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 following table lists the most commonly-used EOAdaptorContext methods:

Method	Description
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.

Method Types

Constructors

EOAdaptorContext

Accessing the adaptor

adaptor

Creating adaptor channels

createAdaptorChannel

channels

Accessing and managing connection status

hasOpenChannels

hasBusyChannels

handleDroppedConnection

Controlling transactions

beginTransaction

commitTransaction

rollbackTransaction

transactionDidBegin

transactionDidCommit

transactionDidRollback

hasOpenTransaction

Debugging

setDebugEnabledDefault

debugEnabledDefault

setDebugEnabled

```
    isDebugEnabled
```

Accessing the delegate

```
    delegate

    setDelegate

    defaultDelegate

    setDefaultDelegate
```

Constructors

EOAdaptorContext

```
public EOAdaptorContext(EOAdaptor anAdaptor)
```

Returns a new EOAdaptorContext. You never invoke this constructor directly. You must use the Adaptor method `createAdaptorContext` to create a new adaptor context.

See Also: `adaptor`

Static Methods

debugEnabledDefault

```
public static boolean debugEnabledDefault()
```

Returns `true` if new adaptor context instances have debugging enabled by default, `false` otherwise. By default, adaptor contexts have debugging enabled if the user default EOAdaptorDebugEnabled is `true`. (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`.

defaultDelegate

```
public static Object defaultDelegate()
```

Returns the default delegate—the object that is assigned as delegate to new adaptor context instances (and their channels).

setDebugEnabledDefault

```
public static void setDebugEnabledDefault(boolean flag)
```

Sets default debugging behavior for new instances of EOAdaptorContext. If `flag` is `true`, debugging is enabled for new instances. If `flag` is `false`, debugging is disabled. Use the instance method `setDebugEnabled` to enable debugging for a specific adaptor context.

See Also: `isDebugEnabled`

setDefaultDelegate

```
public static void setDefaultDelegate(Object anObject)
```

Sets the default delegate—the object assigned as delegate to all newly created EOAdaptorContext instances (and their EOAdaptorChannels). By default there is no default delegate.

Instance Methods

adaptor

```
public EOAdaptor adaptor()
```

Returns the receiver's EOAdaptor.

See Also: `EOAdaptorContext` constructor

beginTransaction

```
public void beginTransaction()
```

Implemented by subclasses to attempt to begin a new transaction. A 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, the method sends `this` a `transactionDidBegin` message and invokes the delegate method `adaptorContextDidBegin`. Throws an exception if the attempt is unsuccessful. Some possible reasons for failure are:

- A connection to the database hasn't been established.
- 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: `hasOpenTransaction`

channels

```
public NSArray channels()
```

Returns an array of channels created by the receiver.

See Also: `createAdaptorChannel`

commitTransaction

```
public 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, the method sends `this` a `transactionDidCommit` message and invokes the delegate method `adaptorContextDidCommit`. Throws an exception 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`, `rollbackTransaction`, `hasBusyChannels`

createAdaptorChannel

```
public EOAdaptorChannel createAdaptorChannel()
```

Implemented by subclasses to create and return a new AdaptorChannel, or `null` if a new channel cannot be created. Sets the new channel's `adaptorContext` to this. 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

```
public Object delegate()
```

Returns the receiver's delegate, or `null` if the receiver doesn't have a delegate.

handleDroppedConnection

`public void handleDroppedConnection()`

Implemented by subclasses to clean up after the receiver's adaptor lost its connection to its database server. Invoked from EOAdaptor's `handleDroppedConnection`, this method cleans up the state of its adaptor channels and of itself so the receiver and its channels can be safely disposed of without any errors.

You should never invoke this method; it is invoked automatically by the Framework. Subclasses must implement this method, without invoking super, if the adaptor supports automatic database reconnection.

hasBusyChannels

`public boolean hasBusyChannels()`

Returns `true` if any of the receiver's channels have outstanding operations (that is, have a fetch in progress), `false` otherwise.

See Also: `isFetchInProgress` **(EOAdaptorChannel)**

hasOpenChannels

`public boolean hasOpenChannels()`

Returns `true` if any of the receiver's channels are open, `false` otherwise.

See Also: `openChannel` (EOAdaptorChannel), `isOpen` **(EOAdaptorChannel)**

hasOpenTransaction

`public boolean hasOpenTransaction()`

Returns `true` if a transaction is open (begun but not yet committed or rolled back), `false` otherwise.

isDebugEnabled

```
public boolean isDebugEnabled()
```

Returns `true` if debugging is enabled in the receiver, `false` otherwise.

See Also: `debugEnabledDefault`, `setDebugEnabledDefault`

rollbackTransaction

```
public 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, the method sends `this` a `transactionDidRollback` message and invokes the delegate method `adaptorContextDidRollback`. Throws an exception 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

```
public void setDebugEnabled(boolean flag)
```

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

See Also: `setDebugEnabled` (EOAdaptorChannel), `isDebugEnabled`, `setDebugEnabledDefault`, `channels`

setDelegate

```
public void setDelegate(Object delegate)
```

Sets the receiver's delegate and the delegate of all the receiver's channels to `delegate`, or removes their delegates if `delegate` is `null`.

See Also: `channels`

transactionDidBegin

```
public 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 `AdaptorContextBeginTransactionNotification`. 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`.

transactionDidCommit

```
public 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 `AdaptorContextCommitTransactionNotification`. 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`.

transactionDidRollback

```
public 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 `AdaptorContextRollbackTransactionNotification`. 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`.

Notifications

AdaptorContextBeginTransactionNotification

```
public static final String AdaptorContextBeginTransactionNotification
```

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

Notification Object	The notifying EOAdaptorContext object
Userinfo	None

AdaptorContextCommitTransactionNotification

`public static final String AdaptorContextCommitTransactionNotification`

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

Notification Object	The notifying EOAdaptorContext object
Userinfo	None

AdaptorContextRollbackTransactionNotification

`public static final String AdaptorContextRollbackTransactionNotification`

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

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 registering as an observer with 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 be overridden with caution:

- `setDebugEnabledDefault`
- `transactionDidBegin`
- `transactionDidCommit`
- `transactionDidRollback`
- `hasOpenTransaction`

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

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`
- `commitTransaction`
- `createAdaptorChannel`
- `rollbackTransaction`

EOAdaptorOperation

Inherits from: NSObject

Package: com.apple.yellow.webobjects

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 (`AdaptorLockOperator`, `AdaptorInsertOperator`, `AdaptorUpdateOperator`, `AdaptorDeleteOperator`, or `AdaptorStoredProcedureOperator`) 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

Constructors

    `EOAdaptorOperation`

Accessing the entity

    `entity`

Accessing the operator

    `setAdaptorOperator`

    `adaptorOperator`

Accessing the qualifier

    `setQualifier`

    `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
```

Constructors

EOAdaptorOperation

```
public EOAdaptorOperation(EOEntity entity)
```

Creates and returns a new EOAdaptorOperation, with *entity* as the entity to which the operation will be applied.

See Also: entity

Instance Methods

adaptorOperator

```
public int 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 AdaptorInsertOperator uses changedValues, **but not** attributes, qualifier, or storedProcedure.

attributes

```
public 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 AdaptorLockOperator.

changedValues

```
public NSDictionary changedValues()
```

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

compareAdaptorOperation

```
public int 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:

- `AdaptorLockOperator`

- `AdaptorInsertOperator`

- `AdaptorUpdateOperator`

- `AdaptorDeleteOperator`

- `AdaptorStoredProcedureOperator`

`AdaptorLockOperator` **precedes** `AdaptorInsertOperator`, `AdaptorInsertOperator` **precedes** `AdaptorUpdateOperator`, **and so on.**

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

entity

```
public EOEntity entity()
```

Returns the entity to which the operation will be applied.

exception

```
public Throwable exception()
```

Returns the exception that was thrown when an adaptor channel attempted to process the receiver. Returns `null` if no exception was thrown or if the receiver hasn't been processed yet.

qualifier

```
public com.apple.yellow.eocontrol.EOQualifier qualifier()
```

Returns the qualifier that identifies the specific row to which the operation applies. Not valid with adaptor operations with the operators `AdaptorInsertOperator` and `AdaptorStoredProcedureOperator`.

setAdaptorOperator

```
public void setAdaptorOperator(int adaptorOperator)
```

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

- `AdaptorLockOperator`
- `AdaptorInsertOperator`
- `AdaptorUpdateOperator`
- `AdaptorDeleteOperator`
- `AdaptorStoredProcedureOperator`

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

setAttributes

```
public 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 `null` 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 `AdaptorLockOperator`.

setChangedValues

```
public 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
AdaptorLockOperator	Snapshot values used to verify that the database row hasn't changed since this application last fetched it
AdaptorInsertOperator	The values to insert
AdaptorUpdateOperator	The new values for the columns to update
AdaptorDeleteOperator	Snapshot values (*changedValues* is only valid for AdaptorDeleteOperator if the receiver's entity uses a stored procedure to perform delete operations.)
AdaptorStoredProcedureOperator	Snapshot values

setException

```
public void setException(Throwable 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.

setQualifier

```
public void setQualifier(com.apple.yellow.eocontrol.EOQualifier qualifier)
```

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

setStoredProcedure

```
public void setStoredProcedure(EOStoredProcedure storedProcedure)
```

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

storedProcedure

```
public EOStoredProcedure storedProcedure()
```

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

EOAttribute

Inherits from:	NSObject
Implements:	EOPropertyListEncoding
Package:	com.apple.yellow.eoaccess

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 *Enterprise Objects Framework 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" (page 103) for information on creating your own attribute objects.

Fore detailed discussion of using attribute objects to map database data types to Java objects, see "Mapping from Database to Objects" (page 105) and "Working with Custom Data Types" (page 105). EOAttributes can also alter the way values are selected, inserted, and updated in the database by defining special format strings; see "SQL Statement Formats" (page 107) for more information.

Constants

EOAttribute defines the following `int` constants as possible return values for the EOAttribute method `adaptorValueType`. The return value indicates the data type that will be fetched from the database for the receiving attribute.

- `AdaptorNumberType`

- `AdaptorCharactersType`

- `AdaptorBytesType`

- `AdaptorDateType`

EOAttribute defines the additional `int` constants to be used by `factoryMethodArgumentType` and `setFactoryMethodArgumentType` to specify the type of argument that should be passed to the attribute's "factory method". For more information, see <u>"Working with Custom Data Types"</u> (page 105).

- `FactoryMethodArgumentIsData`

- `FactoryMethodArgumentIsString`

- `FactoryMethodArgumentIsBytes`

EOAttribute also defines `int` constants to represent parameter direction for EOAttributes that represent arguments to a stored procedure. For more information, see the `parameterDirection` method description.

- `Void`

- `InParameter`

- `OutParameter`

- `InOutParameter`

Interfaces Implemented

EOPropertyListEncoding

awakeWithPropertyList

encodeIntoPropertyList

Method Types

Constructors

EOAttribute

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
```

CLASS EOAttribute

```
factoryMethodArgumentType

setAdaptorValueConversionMethodName

adaptorValueConversionMethod

adaptorValueConversionMethodName

archiveDataForObject
```

Accessing attribute characteristics

```
setReadOnly

isReadOnly

isDerived

isFlattened
```

Accessing SQL statement formats

```
setReadFormat

readFormat

setWriteFormat

writeFormat
```

Accessing the user dictionary

```
setUserInfo

userInfo
```

Working with stored procedures

```
setParameterDirection

parameterDirection

storedProcedure
```

Working with prototypes

```
overridesPrototypeDefinitionForKey

prototype

prototypeName

setPrototype
```

Constructors

EOAttribute

```
public EOAttribute(
    NSDictionary propertyList,
    Object owner)
```

Creates a new EOAttribute initialized from *propertyList*—a dictionary containing only property list data types (that is, String, NSDictionary, NSArray, and NSData objects). This constructor is used by EOModeler when it reads in a Model from a file, for example. The *owner* argument should be the EOAttribute's EOEntity or EOStoredProcedure. EOAttributes created from a property list must receive an `awakeWithPropertyList` message immediately after creation before they are fully functional, but the `awake...` message should be deferred until the all of the other objects in the model have also been created.

Static Methods

archiveDataForObject

```
public static NSData archiveDataForObject(NSObject anObject)
```

Return *anObject*'s value as a NSData object whose bytes can be stored in an external repository.

Instance Methods

adaptorValueByConvertingAttributeValue

`public Object adaptorValueByConvertingAttributeValue(Object `*`value`*`)`

Ensures that *value* is either a String, Number, 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 `AdaptorBytesType`, `archiveDataForObject` will be invoked to convert the attribute value.

See Also: `adaptorValueConversionMethod`, `adaptorValueType`

adaptorValueConversionMethod

`public NSSelector adaptorValueConversionMethod()`

Returns the method used to convert a custom class into one of the primitive types that the adaptor knows how to manipulate: String, Number, 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 Java run-time, `null` is returned.

See Also: `adaptorValueByConvertingAttributeValue`, `adaptorValueConversionMethodName`

adaptorValueConversionMethodName

`public String 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: String, Number, NSData, or NSDate.

See Also: `adaptorValueByConvertingAttributeValue`

adaptorValueType

```
public int adaptorValueType()
```

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

Constant	Description
AdaptorNumberType	A number value
AdaptorCharactersType	A string of characters
AdaptorBytesType	Raw bytes
AdaptorDateType	A date

See Also: factoryMethodArgumentType

allowsNull

```
public boolean allowsNull()
```

Returns true to indicate that the attribute can have a null value, false 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

```
public 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: nameForExternalName (EOEntity), validateName, beautifyNames **(EOModel)**

columnName

```
public String columnName()
```

Returns the name of the column in the database that corresponds to this attribute, or `null` 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 `null`.

See Also: , `externalType`

definition

```
public String definition()
```

Returns a derived or flattened attribute's definition, or `null` 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 `null`.

See Also: `externalType`, `setDefinition`

entity

```
public EOEntity entity()
```

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

See Also: `storedProcedure`

externalType

```
public String 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

```
public int 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
FactoryMethodArgumentIsData	NSData
FactoryMethodArgumentIsString	String
FactoryMethodArgumentIsBytes	raw bytes

See Also: valueFactoryMethod, setFactoryMethodArgumentType

isDerived

```
public boolean isDerived()
```

Returns false if the attribute corresponds exactly to one column in the table associated with its entity, and true 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

```
public boolean isFlattened()
```

Returns true if the attribute is flattened, false 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

`public boolean isReadOnly()`

Returns `true` if the value of the attribute can't be modified, `false` if it can.

See Also: `setReadOnly`

name

`public String name()`

Returns the attribute's name.

See Also: `columnName, definition, setName`

overridesPrototypeDefinitionForKey

`public boolean overridesPrototypeDefinitionForKey(String ` *`key`*`)`

Returns `false` if the requested key gets its value from the prototype attribute. If the attribute has an override, then this method returns `true`. Valid values for key include "columnName," "valueClass," and so on.

See Also: `prototype`

parameterDirection

```
public int parameterDirection()
```

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

Constant	Description
Void	No parameters
InParameter	Input only parameters
OutParameter	Output only parameters
InOutParameter	Bidirectional parameters (input and output)

See Also: `storedProcedure`, `storedProcedureForOperation` (**EOEntity**), `setParameterDirection`

parent

```
public Object 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.

precision

```
public int precision()
```

Returns the precision of the database representation for attributes with a value class of Number or java.math.BigDecimal.

See Also: `scale`

prototype

```
public EOAttribute prototype()
```

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

See Also: `overridesPrototypeDefinitionForKey`

prototypeName

`public String prototypeName()`

Returns the name of the prototype attribute of the receiver.

See Also: `prototype`

readFormat

`public String 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

`public int scale()`

Returns the scale of the database representation for attributes with a value class of Number or java.math.BigDecimal. The returned value can be negative.

See Also: `precision`, `setScale`

serverTimeZone

`public 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: `setServerTimeZone`

setAdaptorValueConversionMethodName

public void setAdaptorValueConversionMethodName(String *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: String, Number, NSData, or NSDate.

See Also: adaptorValueConversionMethodName

setAllowsNull

public void setAllowsNull(boolean *allowsNull*)

Sets according to *allowsNull* whether or not the attribute can have a null 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

public void setColumnName(String *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 null

See Also: columnName

setDefinition

```
public void setDefinition(String 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 removed 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 `null`.

See Also: definition

setExternalType

```
public void setExternalType(String 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

```
public void setFactoryMethodArgumentType(int 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 Strings, NSDatas, or raw bytes; specify an *argumentType* as `FactoryMethodArgumentIsString`, `FactoryMethodArgumentIsData`, or `FactoryMethodArgumentIsBytes` as appropriate.

See Also: setValueFactoryMethodName, factoryMethodArgumentType

setName

```
public void setName(String name)
```

Sets the attribute's name to *name*. Throws an exception 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

```
public void setParameterDirection(int parameterDirection)
```

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

- Void

- InParameter

- OutParameter

- InOutParameter

See Also: setStoredProcedure (**EOEntity**), parameterDirection

setPrecision

```
public void setPrecision(int precision)
```

Sets to *precision* the precision of the database representation for attributes with a value class of Number or java.math.BigDecimal.

See Also: setScale, precision

setPrototype

```
public void setPrototype(EOAttribute prototype)
```

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

See Also: prototype

setReadFormat

```
public void setReadFormat(String 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.

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

See Also: `setWriteFormat`, `readFormat`

setReadOnly

```
public void setReadOnly(boolean flag)
```

Sets whether the value of the attribute can be modified according to *flag*. Throws an exception if *flag* is `false` and the argument is derived but not flattened.

See Also: `isDerived`, `isFlattened`, `isReadOnly`

setScale

```
public void setScale(int scale)
```

Sets to *scale* the scale of the database representation for attributes with a value class of Number or java.math.BigDecimal. *scale* can be negative.

See Also: `setPrecision`, `scale`

setServerTimeZone

```
public void setServerTimeZone(NSTimeZone aTimeZone)
```

Sets to *aTimeZone* the time zone used for NSDates in the database server. If *aTimeZone* is `null` 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

`public 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 String).

See Also: userInfo

setValueClassName

`public void setValueClassName(String `*`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.

See Also: setValueType, valueClassName

setValueFactoryMethodName

`public void setValueFactoryMethodName(String `*`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 static method returning an 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

```
public void setValueType(String typeName)
```

Sets to *typeName* the conversion character (such as "i" or "d") for the data type a Number 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

```
public void setWidth(int 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

```
public void setWriteFormat(String 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.

See Also: setReadFormat, writeFormat

storedProcedure

```
public 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 `null`.

See Also: entity

userInfo

`public NSDictionary userInfo()`

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

See Also: `setUserInfo`

validateName

`public void validateName(String name)`

Validates *name* and returns `null` 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:

`public Object validateValue(Object value)`

Validates value by converting it to the attribute's value type and by testing other attribute validation constraints (such as `allowsNull`, `width`, and so on). If, during the validation process, any coercion was performed, the converted value is returned.

See Also: `adaptorValueByConvertingAttributeValue`, `allowsNull`, `valueType`, `valueClassName`, `width`

valueClassName

`public String 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.

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

`public NSSelector 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 method in the Java run-time, this method returns `null`.

valueFactoryMethodName

`public String valueFactoryMethodName()`

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

See Also: `valueFactoryMethod`, `setValueFactoryMethodName`

valueType

`public String valueType()`

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

See Also: `valueClassName`, `setValueType`

width

`public int 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

`public String 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`

EOAttribute

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 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 Java class used to represent values outside the context of an enterprise object

- For Java 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.

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.

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.

Instead of flattening attributes in your model, a better approach is often to directly traverse the object graph through relationships.

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 a Java 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:

- String
- Number
- java.math.BigDecimal
- 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. 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: String, Number, java.math.BigDecimal, 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 com.apple.yellow.application.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 `FactoryMethodArgumentIsData` or `FactoryMethodArgumentIsBytes`, 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 `FactoryMethodArgumentIsString` or `FactoryMethodArgumentIsBytes`, indicating that the method takes a String object or raw bytes as an argument. These types apply when fetching custom values.

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 Enterprise Objects Modeling" in the *Enterprise Objects Framework Developer's Guide*.

Fetching Custom Values

Custom values are created during fetching in EOAdaptorChannel's `fetchRow` method. This method fetches data in the external (server) type and converts it to a value object, applying the custom value factory method (`valueFactoryMethod`) to convert a value into the custom class if necessary. Once the value is converted, the EOAdaptorChannel puts it into the dictionary for the row being fetched.

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` 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 *Enterprise Objects Framework Tools and Techniques.*

107

EOAutoMorphedObject

Inherits from:	NSObject
Implements:	com.apple.yellow.eocontrol.EOEnterpriseObject
Package:	com.apple.yellow.eoaccess

Class Description

The EOAutoMorphedObject class is available in Yellow Box applications only; these is not an equivalent in Java Client.

EOAutoMorphedObject allows you to use Objective–C enterprise object classes in Java code without wrapping them. When you use an unwrapped Objective–C enterprise object in Java code, the Java bridge creates EOAutoMorphedObjects to represent them. The drawback to using this feature is that you must use key-value coding rather than invoke methods directly.

You never instantiate or subclass EOAutoMorphedObject. Rather, it's something that the Java bridge creates automatically.

EODatabase

Inherits from: NSObject

Package: com.apple.yellow.eoaccess

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.

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.

For more information on the EODatabase class, see the sections:

- "The Database Level" (page 125)
- "Snapshots" (page 127)

Constants

EODatabase declares the following constants.

Constant	Type	Description
DistantPastTimeInterval	double	The lower bound on timestamps.
GeneralDatabaseException	String	The name of exceptions raised by the database sublayer when errors occur during interactions with a database server.

Method Types

Constructors

EODatabase

Adding and removing models

addModel

addModelIfCompatible

removeModel

models

Accessing entities

entityForObject

entityNamed

Recording snapshots

recordSnapshotForGlobalID

```
recordSnapshotForSourceGlobalID

recordSnapshots

recordToManySnapshots:
```

Forgetting snapshots

```
forgetSnapshotForGlobalID

forgetSnapshotsForGlobalIDs

forgetAllSnapshots
```

Accessing snapshots and snapshot timestamps

```
snapshotForGlobalID

snapshotForSourceGlobalID

snapshots

timestampForGlobalID

timestampForSourceGlobalID
```

Snapshot reference counting

```
incrementSnapshotCountForGlobalID

decrementSnapshotCountForGlobalID

disableSnapshotRefcounting
```

Registering database contexts

```
registerContext

unregisterContext

registeredContexts
```

Accessing the adaptor

```
adaptor
```

Managing the database connection

```
handleDroppedConnection
```

Managing the result cache

```
invalidateResultCache
```

```
invalidateResultCacheForEntityNamed

resultCacheForEntityNamed

setResultCacheForEntityWithName
```

Constructors

EODatabase

```
public EODatabase(EOAdaptor anAdaptor)
```

Creates and returns a new EODatabase object, specifying *anAdaptor* as the new EODatabase's adaptor. 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, create an EODatabase with an EOModel instead of an EOAdaptor.

```
public EODatabase(EOModel aModel)
```

Creates and returns a new EODatabase object, also creating an instance of the EOAdaptor named in *aModel* and assigning that EOAdaptor object as the new EODatabase's adaptor.

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: addModel, adaptor, adaptorName **(EOModel)**

Static Methods

disableSnapshotRefcounting

```
public static void disableSnapshotRefcounting()
```

Configures EODatabase instances not to release unreferenced snapshots.

Instance Methods

adaptor

`public 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).

See Also: `EODatabase` constructors

addModel

`public 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

`public boolean 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 `true` if `aModel` is already in the list or if it's successfully added. Returns `false` if `aModel`'s adaptor name differs from that of the receivers or if the receiver's `adaptor` returns `false` to a `canServiceModel:` message.

See Also: `addModel`, `models`, `removeModel`

115

decrementSnapshotCountForGlobalID

```
public void
    decrementSnapshotCountForGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalId)
```

If the receiver releases unreferenced snapshots, decrements the reference count for the shared snapshot associated with *globalID*; and if no more objects refer to the snapshot, removes it from the snapshot table. (If the receiver doesn't release snapshots, this method does nothing.)

entityForObject

```
public EOEntity entityForObject(Object anObject)
```

Returns the EOEntity from one of the receiver's Models that's mapped to *anObject*, or null 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

```
public EOEntity entityNamed(String entityName)
```

Returns the EOEntity from one of the receiver's Models that's named *entityName*, or null 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

```
public void forgetAllSnapshots()
```

Clears all of the receiver's snapshots and posts an `ObjectsChangedInStoreNotification` (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, recordSnapshotForGlobalID, recordSnapshots, recordSnapshotForSourceGlobalID, recordToManySnapshots:`

forgetSnapshotForGlobalID

`public void forgetSnapshotForGlobalID(com.apple.yellow.eocontrol.EOGlobalID `*`globalID`*`)`

Clears the snapshot made for the enterprise object identified by *`globalID`* and posts an `ObjectsChangedInStoreNotification` (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`, `recordSnapshotForGlobalID`

forgetSnapshotsForGlobalIDs

`public void forgetSnapshotsForGlobalIDs(NSArray `*`globalIDs`*`)`

Clears the snapshots made for the enterprise objects identified by each of the EOGlobalIDs in *`globalIDs`* and posts an ObjectsChangedInStoreNotification (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`

handleDroppedConnection

`public void handleDroppedConnection()`

Invoked to initiate clean up when the Framework detects a dropped database connection. The receiver cleans up by sending `handleDroppedConnection` to its adaptor, and then sending `handleDroppedConnection` to all of its registered database contexts. When the cleanup procedure is complete, the Framework can automatically reconnect to the database.

You should never invoke this method; it's invoked automatically by the Framework.

incrementSnapshotCountForGlobalID

`public void`
   `incrementSnapshotCountForGlobalID(com.apple.yellow.eocontrol.EOGlobalID `*`globalID`*`)`

Increments the reference count for the shared snapshot associated with *`globalID`* if the receiver releases unreferenced snapshots. (Does nothing if the receiver doesn't release snapshots.)

117

invalidateResultCache

```
public void invalidateResultCache()
```

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

See Also: `invalidateResultCacheForEntityNamed`, `resultCacheForEntityNamed`

invalidateResultCacheForEntityNamed

```
public void invalidateResultCacheForEntityNamed(String 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

```
public NSArray models()
```

Returns the receiver's EOModels.

See Also: EODatabase **constructor**, `addModel`, `addModelIfCompatible`, `removeModel`

recordSnapshotForGlobalID

```
public void recordSnapshotForGlobalID(
    NSDictionary aSnapshot,
    com.apple.yellow.eocontrol.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`

recordSnapshotForSourceGlobalID

```
public void recordSnapshotForSourceGlobalID(
    NSArray globalIDs,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String 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: recordSnapshotForGlobalID, recordSnapshots, recordSnapshotForGlobalID, snapshotForSourceGlobalID:relationshipName:

recordSnapshots

```
public 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: recordSnapshotForGlobalID, forgetSnapshotsForGlobalIDs

recordToManySnapshots:

```
public 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: recordSnapshotForSourceGlobalID, recordSnapshotForGlobalID, snapshotForSourceGlobalID:relationshipName:

registerContext

`public void registerContext(EODatabaseContext *aContext*)`

Records *aContext* as one of the receiver's EODatabaseContexts. The receiver must have been specified as aContext's EODatabase in the EODatabaseContext constructor (which invokes this method automatically). You should never need to invoke this method directly.

See Also: `unregisterContext, registeredContexts`

registeredContexts

`public 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

`public void removeModel(EOModel *aModel*)`

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

See Also: `addModel, addModelIfCompatible, models`

resultCacheForEntityNamed

`public NSArray resultCacheForEntityNamed(String *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`

setResultCacheForEntityWithName

```
public void setResultCacheForEntityWithName(
    NSArray cache,
    String 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`

setTimestampToNow

```
public void setTimestampToNow()
```

Sets the internal timestamp to the value returned by NSDate's `timeIntervalSinceReferenceDate` method. Used for recording subsequent snapshots.

snapshotForGlobalID

```
public NSDictionary snapshotForGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    double timestamp)
```

```
public NSDictionary snapshotForGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalID)
```

Returns the snapshot associated with *globalID*. Returns `null` if there isn't a snapshot for the globalID or if the corresponding timestamp is less than timestamp. If timestamp isn't provided, `DistantPastTimeInterval` is used. For a description of snapshots and their role in an application, see the class description.

See Also: `recordSnapshotForGlobalID`, `forgetSnapshotForGlobalID`

snapshotForSourceGlobalID

```
public NSArray snapshotForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String name,
    double timestamp)
```

```
public NSArray snapshotForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String name)
```

Returns the to-many snapshot for `globalId` and `name`. A to-many snapshot is 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`. Returns `null` if there isn't a to-many snapshot for `globalId` or if the timestamp is less than timestamp. If timestamp isn't provided, `DistantPastTimeInterval` is used. For a description of snapshots and their role in an application, see the class description.

snapshots

```
public NSDictionary snapshots()
```

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

See Also: `recordSnapshotForSourceGlobalID`, `recordToManySnapshots:`

timestampForGlobalID

```
public double timestampForGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalId)
```

Returns the timestamp of the snapshot for `globalID`. Returns `EODistantPastTimeInterval` if there isn't a snapshot.

timestampForSourceGlobalID

```
public double timestampForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalId,
    String relationshipName)
```

Returns the timestamp of the to-many snapshot for the relationship specified by `relationshipName` and the object specified by `globalID`. Returns `EODistantPastTimeInterval` if there isn't a snapshot.

unregisterContext

`public void unregisterContext(EODatabaseContext `*`aContext`*`)`

Removes *aContext* as one of the receiver's EODatabaseContexts. An EODatabaseContext automatically invokes this method when it's finalized; you should never need to invoke it directly.

See Also: `registerContext, registeredContexts`

EODatabase

The Database Level

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 0-1   The Access Layer
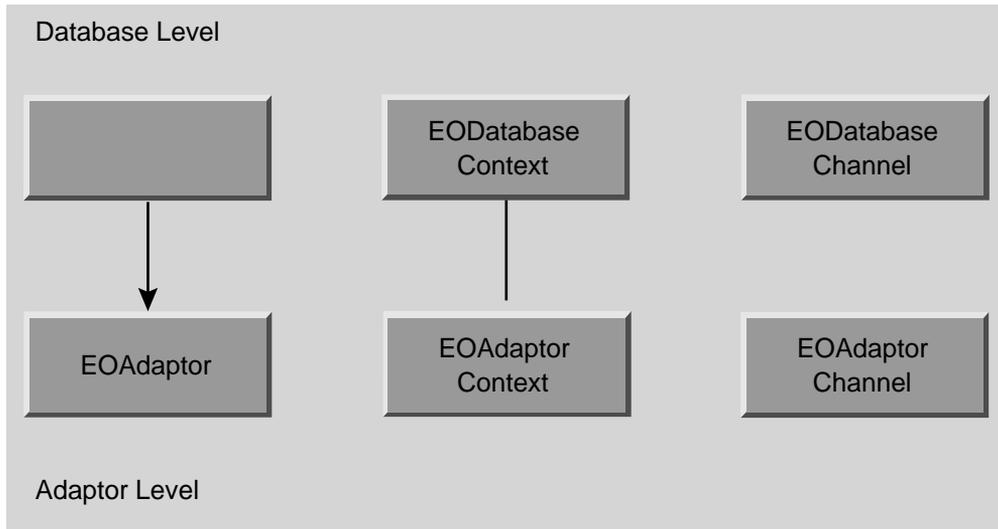
The database level acts as an intermediary between the adaptor level and the **control layer** , which includes an EOObjectStoreCoordinator and an EOEditingContext (Figure 0-2). The control layer operates in terms of enterprise objects, while the adaptor level operates in terms of database rows packaged 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 0-2   The EODatabase Level as an Intermediary Between the Adaptor Level and the Control
Layer
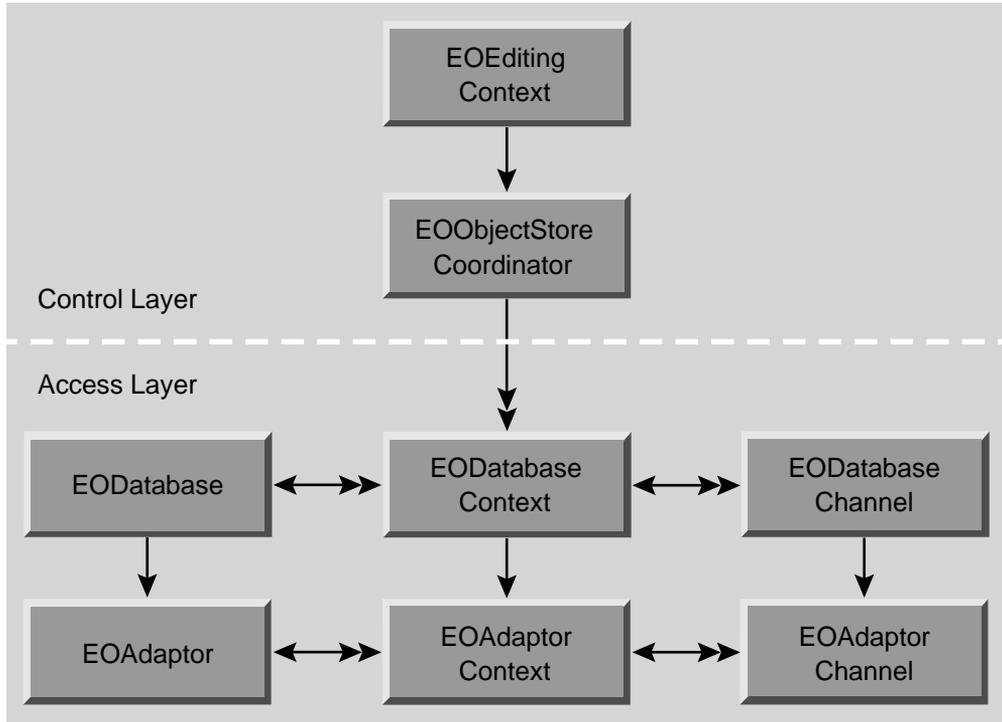
Snapshots

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 globalID 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.

Entity Caching

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 entity 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 entity 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. Since the ratings in the Rating entity never change, the 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 entity 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
Package:	com.apple.yellow.eoaccess

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

Constructors

EODatabaseChannel

Accessing cooperating objects

adaptorChannel

databaseContext

Fetching objects

selectObjectsWithFetchSpecification

isFetchInProgress

fetchObject

cancelFetch

Accessing internal fetch state

setCurrentEntity

setCurrentEditingContext

setIsLocking

isLocking

setIsRefreshingObjects

isRefreshingObjects

Accessing the delegate

setDelegate

delegate

Constructors

EODatabaseChannel

`public EODatabaseChannel(EODatabaseContext `*`aDatabaseContext`*`)`

Creates and returns a new EODatabaseChannel. 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.

`aDatabaseContext` is assigned to the new EODatabaseChannel as the DatabaseContext in which the channel works. The new EODatabaseChannel creates an AdaptorChannel with which to communicate with the database server. The constructor throws an exception if the underlying adaptor context can't create a corresponding adaptor channel.

See Also: `databaseContext`, `adaptorChannel`

Instance Methods

adaptorChannel

`public EOAdaptorChannel adaptorChannel()`

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

See Also: `EODatabaseChannel` **constructor**

cancelFetch

`public void cancelFetch()`

Cancels any fetch in progress.

See Also: `isFetchInProgress`, `selectObjectsWithFetchSpecification`, `fetchObject`

databaseContext

`public EODatabaseContext databaseContext()`

Returns the EODatabaseContext that controls transactions for the receiver.

See Also: `EODatabaseChannel` constructor

delegate

`public Object 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

`public Object fetchObject()`

Fetches and returns the next object in the result set produced by a `selectObjectsWithFetchSpecification` message; returns `null` 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 `awakeFromFetch` message to allow it to finish setting up its state.

If no snapshot exists for the fetched object, the receiver sends its EODatabase a `recordSnapshotForGlobalID` 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 `databaseContextShouldUpdateCurrentSnapshot` 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 ObjectsChangedInStoreNotification 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`

isFetchInProgress

Returns `true` if the receiver is fetching, `false` otherwise. An EODatabaseChannel is fetching if it's been sent a successful `selectObjectsWithFetchSpecification` message. An EODatabaseChannel stops fetching when there are no more objects to fetch or when it is sent a `cancelFetch` message.

isLocking

`public boolean isLocking()`

Returns `true` 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 `false` otherwise. This method always returns `false` when no fetch is in progress.

See Also: `locksObjects` **(EOFetchSpecification)**

isRefreshingObjects

`public boolean isRefreshingObjects()`

Returns `true` 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 `false` otherwise. This behavior is controlled by the EOFetchSpecification used in a `selectObjectsWithFetchSpecification` message.

See Also: `refreshesRefetchedObjects` (EOFetchSpecification), `fetchObject`

selectObjectsWithFetchSpecification

```
public void selectObjectsWithFetchSpecification(
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    com.apple.yellow.eocontrol.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.

Throws 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 `databaseContextShouldSelectObjects`, `databaseContextShouldUsePessimisticLock`, and `databaseContextDidSelectObjects`. 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

```
public void
    setCurrentEditingContext(com.apple.yellow.eocontrol.EOEditingContext anEditingContex
    t)
```

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

See Also: `setCurrentEntity`

setCurrentEntity

`public 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`.You should never need to invoke it directly.

See Also: `setCurrentEditingContext`

setDelegate

`public void setDelegate(Object `*`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

`public void setIsLocking(boolean `*`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 `true` the EODatabaseChannel modifies its fetching behavior to lock objects; if *flag* is `false` it simply fetches them.

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

See Also: `locksObjects` **(EOFetchSpecification)**

setIsRefreshingObjects

```
public void setIsRefreshingObjects(boolean flag)
```

Records whether the receiver causes existing snapshots and enterprise objects to be overwritten with fetched values. If *flag* is true the receiver overwrites existing snapshots with fetched values and posts an ObjectsChangedInStoreNotification 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 false, 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 message. You might invoke this method directly if evaluating SQL directly with EOAdaptorChannel's evaluateExpression: method.

See Also: refreshesRefetchedObjects **(EOFetchSpecification)**

EODatabaseContext

Inherits from:	EOCooperatingObjectStore (EOControl) :
	EOObjectStore (EOControl) :
	NSObject
Package:	com.apple.yellow.eoaccess

Class Description

An EODatabaseContext object is an EOObjectStore (EOControl) 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 on EODatabaseContext, see the sections:

- "EODatabaseContext's Interaction with Other Classes" (page 165)
- "Creating and Using an EODatabaseContext" (page 167)
- "Fetching and Saving Objects" (page 168)
- "Using a Custom Query" (page 169)
- "Faulting" (page 169)

Constants

EODatabaseContext defines the following constants:

Constant	Type	Description
UpdateWithOptimisticLocking	int	Identifies the locking strategy as optimistic.
UpdateWithPessimisticLocking	int	Identifies the locking strategy as pessimistic
UpdateWithNoLocking	int	Identifies the locking strategy as no locking
CustomQueryExpressionHintKey	String	A key in an EOFetchSpecification's hint dictionary
StoredProcedureNameHintKey	String	A key in an EOFetchSpecification's hint dictionary
DatabaseContextKey	String	A key in an GenericAdaptorException's userInfo dictionary
DatabaseOperationsKey	String	A key in an GenericAdaptorException's userInfo dictionary
FailedDatabaseOperationKey	String	A key in an GenericAdaptorException's userInfo dictionary

In addition, EODatabaseContext defines a constant for the name of the notification it posts. For more information on the notification, see "Notifications" (page 164).

Method Types

Constructors

    `EODatabaseContext`

Fetching objects

    `objectsWithFetchSpecification`

    `objectsForSourceGlobalID`

    `arrayFaultWithSourceGlobalID`

    `faultForGlobalID`

    `faultForRawRow`

    `batchFetchRelationship`

    `missingObjectGlobalIDs`

Enabling shared object loading

    `setSharedObjectLoadingEnabled`

    `isSharedObjectLoadingEnabled`

Accessing the adaptor context

    `adaptorContext`

Managing the database connection

    `forceConnectionWithModel`

    `handleDroppedConnection`

Accessing the database object

    `database`

Accessing the coordinator

    `coordinator`

Managing channels

`availableChannel`

`registerChannel`

`registeredChannels`

`unregisterChannel`

`hasBusyChannels`

Accessing the delegate

`setDelegate`

`delegate`

`setDefaultDelegate`

`defaultDelegate`

Committing or discarding changes

`saveChangesInEditingContext`

`invalidateAllObjects`

`invalidateObjectsWithGlobalIDs`

`rollbackChanges`

`commitChanges`

`prepareForSaveWithCoordinator`

`recordUpdateForObject`

`recordChangesInEditingContext`

`refaultObject`

Determining if the EODatabaseContext is responsible for a particular operation

`ownsObject`

`ownsGlobalID`

`handlesFetchSpecification`

Recording snapshots

`recordSnapshotForGlobalID`

recordSnapshots

recordSnapshotForSourceGlobalID

recordToManySnapshots

Forgetting snapshots

forgetSnapshotForGlobalID

forgetSnapshotsForGlobalIDs

editingContextDidForgetObjectWithGlobalID

Accessing snapshots

localSnapshotForGlobalID

localSnapshotForSourceGlobalID

snapshotForGlobalID

snapshotForSourceGlobalID

Initializing objects

initializeObject

Obtaining an EODatabaseContext

registeredDatabaseContextForModel

Locking objects

setUpdateStrategy

updateStrategy

registerLockedObjectWithGlobalID

isObjectLockedWithGlobalID

isObjectLockedWithGlobalID

forgetAllLocks

forgetLocksForObjectsWithGlobalIDs

lockObjectWithGlobalID

Returning information about objects

valuesForKeys

Setting the context class

    contextClassToRegister

    setContextClassToRegister

Thread safety

    lock

    unlock

Constructors

EODatabaseContext

public EODatabaseContext(EODatabase *aDatabase*)

Creates and returns a new EODatabaseContext. Typically, you don't need to programmatically create database contexts. Rather, they are created automatically by the control layer. See "Creating and Using an EODatabaseContext" for more information.

aDatabase is assigned to the new database as the EODatabase object with which the new context works. The new database context creates an EOAdaptorContext with which to communicate with the database server. Throws an exception if the underlying adaptor context can't create a corresponding adaptor channel.

See Also: database

Static Methods

contextClassToRegister

`public static Class contextClassToRegister()`

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

When an EOObjectStoreCoordinator sends an `CooperatingObjectStoreNeeded` notification for an EOEntity in the default model group, if `contextClassToRegister` is non-`null` (and it should be— it makes no sense to set `contextClassToRegister` to `null`), 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.

defaultDelegate

`public static Object defaultDelegate()`

Returns the default delegate—the object assigned as delegate to new EODatabaseContext instances.

forceConnectionWithModel

```
public static EODatabaseContext forceConnectionWithModel(
    EOModel aModel,
    NSDictionary overrides,
    com.apple.yellow.eocontrol.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*.

isSharedObjectLoadingEnabled

```
public static boolean isSharedObjectLoadingEnabled()
```

Returns `true` if database contexts automatically load enterprise objects into the default shared editing context when they load models; `false` otherwise. The objects loaded into the shared editing context are those identified by entities' shared fetch specifications.

See Also: `sharedObjectFetchSpecificationNames` (EOEntity)

registeredDatabaseContextForModel

```
public static EODatabaseContext registeredDatabaseContextForModel(
    EOModel aModel,
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext)
```

Finds the EOObjectStoreCoordinator (EOControl) 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

```
public static void setContextClassToRegister(Class contextClass)
```

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

setDefaultDelegate

```
public static void setDefaultDelegate(Object defaultDelegate)
```

Sets the default delegate to *defaultDelegate*—the object assigned as delegate to new instances of EODatabaseContext.

setSharedObjectLoadingEnabled

`public static void setSharedObjectLoadingEnabled(boolean `*`flag`*`)`

Sets according to *flag* whether database contexts automatically load enterprise objects into the default shared editing context when they load models. The default is `true` (the database automatically loads shared objects). The objects loaded into the shared editing context are those identified by entities' shared fetch specifications.

See Also: `sharedObjectFetchSpecificationNames` (**EOEntity**)

Instance Methods

adaptorContext

`public EOAdaptorContext adaptorContext()`

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

arrayFaultWithSourceGlobalID

```
public NSArray arrayFaultWithSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String name,
    com.apple.yellow.eocontrol.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`

availableChannel

`public 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 `DatabaseChannelNeededNotification` 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 `null`.

See Also: `registerChannel`, `registeredChannels`, `unregisterChannel`

batchFetchRelationship

```
public void batchFetchRelationship(
   EORelationship relationship,
   NSArray objects,
   com.apple.yellow.eocontrol.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

```
public 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 `CooperatingObjectStoreNeeded` (defined in EOControl's 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: `ownsGlobalID`, `rollbackChanges`

coordinator

```
public com.apple.yellow.eocontrol.EOObjectStoreCoordinator coordinator()
```

Returns the receiver's EOObjectStoreCoordinator (EOControl) or `null` if there is none. This method is only valid during a save operation.

database

```
public EODatabase database()
```

Returns the receiver's EODatabase.

See Also: EODatabaseContext

delegate

```
public Object delegate()
```

Returns the receiver's delegate.

See Also: setDelegate

editingContextDidForgetObjectWithGlobalID

```
public void editingContextDidForgetObjectWithGlobalID(
    com.apple.yellow.eocontrol.EOEditingContext context,
    com.apple.yellow.eocontrol.EOGlobalID gid)
```

Overrides the inherited implementation. Invoked when context is no longer using the object corresponding to *gid*. EODatabaseContext's implementation destroys related data (such as snapshots) if no other objects are using it. Don't invoke this method; it is invoked automatically by the Framework.

See Also: decrementSnapshotCountForGlobalID (EODatabase),
incrementSnapshotCountForGlobalID **(EODatabase)**

faultForGlobalID

```
public com.apple.yellow.eocontrol.EOEnterpriseObject faultForGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.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

faultForRawRow

```
public com.apple.yellow.eocontrol.EOEnterpriseObject faultForRawRow(
    Object row,
    String entityName,
    com.apple.yellow.eocontrol.EOEditingContext editingContext)
```

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

```
public 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, forgetLocksForObjectsWithGlobalIDs, lockObjectWithGlobalID, lockObject **(EOEditingContext)**

forgetLocksForObjectsWithGlobalIDs

```
public 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, forgetAllLocks, lockObjectWithGlobalID, lockObject **(EOEditingContext)**

forgetSnapshotForGlobalID

`public void forgetSnapshotForGlobalID(com.apple.yellow.eocontrol.EOGlobalID` *`globalID`*`)`

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

See Also: `recordSnapshotForGlobalID`, `localSnapshotForGlobalID`, `recordSnapshots`, `snapshotForGlobalID:`, `forgetSnapshotsForGlobalIDs`

forgetSnapshotsForGlobalIDs

`public void forgetSnapshotsForGlobalIDs(NSArray` *`globalIDs`*`)`

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

See Also: `recordSnapshotForGlobalID`, `localSnapshotForGlobalID`, `recordSnapshots`, `snapshotForGlobalID:`

handleDroppedConnection

`public void handleDroppedConnection()`

Cleans up after a database connection is dropped by releasing the receiver's adaptor context and database channels, and then creating a new adaptor context. Don't invoke this method; it's invoked automatically by the Framework.

handlesFetchSpecification

```
public boolean
   handlesFetchSpecification(com.apple.yellow.eocontrol.EOFetchSpecification fetchSpec)
```

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

See Also: `ownsObject`, `ownsGlobalID`

hasBusyChannels

```
public boolean hasBusyChannels()
```

Returns `true` if the receiver's EOAdaptorContext has channels that have outstanding operations (that is, have a fetch in progress), `false` otherwise.

initializeObject

```
public void initializeObject(
    com.apple.yellow.eocontrol.EOEnterpriseObject object,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.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.

invalidateAllObjects

```
public void invalidateAllObjects()
```

Overrides the inherited implementation to discard all snapshots in the receiver's EODatabase, forget all locks, and post an `InvalidatedAllObjectsInStoreNotification`, as well as an `ObjectsChangedInStoreNotification` with the invalidated global IDs in the `userInfo` dictionary. Both of these notifications are defined in EOObjectStore (EOControl). This method works by invoking `invalidateObjectsWithGlobalIDs` for all of the snapshots in the receiver's EODatabase.

invalidateObjectsWithGlobalIDs

```
public void invalidateObjectsWithGlobalIDs(NSArray globalIDs)
```

Overrides the inherited implementation to discard the snapshots for the objects identified by the EOGlobalIDs in *globalIDs* and broadcasts an `ObjectsChangedInStoreNotification` (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

```
public boolean
    isObjectLockedWithGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalID)
```

Returns `true` if the enterprise object identified by *globalID* is locked, `false` otherwise.

See Also: `registerLockedObjectWithGlobalID`, `forgetAllLocks`, `isObjectLockedWithGlobalID`, `forgetLocksForObjectsWithGlobalIDs`, `lockObjectWithGlobalID`, `lockObject` **(EOEditingContext)**

isObjectLockedWithGlobalID

```
public boolean isObjectLockedWithGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext)
```

Overrides the EOObjectStore method `isObjectLockedWithGlobalID:editingContext:` to return `true` 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`, `lockObject` **(EOEditingContext)**

localSnapshotForGlobalID

```
public NSDictionary
    localSnapshotForGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalID)
```

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

See Also: `recordSnapshotForGlobalID`, `forgetSnapshotForGlobalID`, `recordSnapshots`, `snapshotForGlobalID:`

localSnapshotForSourceGlobalID

```
public NSArray localSnapshotForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String 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 null. Only searches locally (in the transaction scope), not in the EODatabase.

See Also: recordSnapshotForSourceGlobalID, snapshotForSourceGlobalID:relationshipName:

lock

```
public 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

```
public void lockObjectWithGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.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. Throws an exception if unable to obtain the lock.

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

missingObjectGlobalIDs

```
public NSArray missingObjectGlobalIDs()
```

Returns the globalIDs of any "missing" enterprise objects, or an empty array if no missing objects are known to the receiver. An object is "missing" when a fault fires and the corresponding row for the fault isn't found in the database.

To be notified when a missing object is discovered, implement the delegate method `databaseContextFailedToFetchObject`.

If an application tries to save a missing object, an exception is raised.

objectsForSourceGlobalID

```
public NSArray objectsForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String name,
    com.apple.yellow.eocontrol.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`.

objectsWithFetchSpecification

```
public NSArray objectsWithFetchSpecification(
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    com.apple.yellow.eocontrol.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, EODatabaseContext defines the following keys:

Constant	Corresponding value in the hints dictionary
CustomQueryExpressionHintKey	A String specifying raw SQL with which to perform the fetch. There is no way to pass down parameters with this hint.
StoredProcedureNameHintKey	A String 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 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

public boolean ownsGlobalID(com.apple.yellow.eocontrol.EOGlobalID *globalID*)

Overrides the inherited implementation to return true if the receiver is responsible for fetching and saving the object identified by *globalID*, false 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

`public boolean ownsObject(com.apple.yellow.eocontrol.EOEnterpriseObject` *`object`*`)`

Overrides the inherited implementation to return `true` if the receiver is responsible for fetching and saving *`object`*, `false` 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

`public void performChanges()`

Overrides the inherited implementation to construct EOAdaptorOperations from the EODatabaseOperations produced during `recordChangesInEditingContext` and `recordUpdateForObject`. Invokes the delegate method `databaseContextWillOrderAdaptorOperations` to give the delegate an opportunity to construct alternative EOAdaptorOperations from the EODatabaseOperations. Then invokes the delegate method `databaseContextWillPerformAdaptorOperations` 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:

Key (String Constant)	Value
`DatabaseContextKey`	The EODatabaseContext object that was trying to save to its underlying repository when the exception was raised.
`DatabaseOperationsKey`	The list of database operations the EODatabaseContext was trying to perform when the failure occurred.
`FailedDatabaseOperationKey`	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

```
public void prepareForSaveWithCoordinator(
    com.apple.yellow.eocontrol.EOObjectStoreCoordinator coordinator,
    com.apple.yellow.eocontrol.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

```
public 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` and before `ownsGlobalID`. You should never need to invoke this method directly.

recordSnapshotForGlobalID

```
public void recordSnapshotForGlobalID(
    NSDictionary aSnapshot,
    com.apple.yellow.eocontrol.EOGlobalID aGlobalID)
```

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:`

recordSnapshotForSourceGlobalID

```
public void recordSnapshotForSourceGlobalID(
    NSArray globalIDs,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    String 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`, `recordToManySnapshots`

recordSnapshots

```
public void recordSnapshots(NSDictionary snapshots)
```

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

See Also: `recordSnapshotForGlobalID`, `localSnapshotForGlobalID`, `forgetSnapshotForGlobalID`, `snapshotForGlobalID:`

recordToManySnapshots

```
public 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: `recordSnapshotForSourceGlobalID`, `snapshotForSourceGlobalID:relationshipName:`, `localSnapshotForSourceGlobalID`

159

recordUpdateForObject

```
public void recordUpdateForObject(
    com.apple.yellow.eocontrol.EOEnterpriseObject object,
    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` and before `ownsGlobalID`.

refaultObject

```
public void refaultObject(
    com.apple.yellow.eocontrol.EOEnterpriseObject object,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext)
```

Overrides the inherited implementation to refault the enterprise 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

```
public 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. You use this method if you need to perform more than one fetch simultaneously.

See Also: availableChannel, registeredChannels, unregisterChannel

registeredChannels

```
public NSArray registeredChannels()
```

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

See Also: `registerChannel`, `availableChannel`, `unregisterChannel`

registerLockedObjectWithGlobalID

```
public void
    registerLockedObjectWithGlobalID(com.apple.yellow.eocontrol.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`, `lockObject` **(EOEditingContext)**

rollbackChanges

```
public 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: `ownsGlobalID`, `commitChanges`

saveChangesInEditingContext

```
public void
    saveChangesInEditingContext(com.apple.yellow.eocontrol.EOEditingContext context)
```

Overrides the inherited implementation to save the changes made in *context*. 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

```
public void setDelegate(Object 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

```
public void setUpdateStrategy(int strategy)
```

Sets the update strategy used by the EODatabaseContext to *strategy*. See "Updating And Locking Strategies" (page 172) for information on the update strategies:

■   UpdateWithOptimisticLocking

■   UpdateWithPessimisticLocking

■   UpdateWithNoLocking

Throws an exception if the receiver has any transactions in progress or if you try to set *strategy* to UpdateWithPessimisticLocking and the receiver's EODatabase already has snapshots.

See Also: updateStrategy

snapshotForGlobalID

```
public NSDictionary snapshotForGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    double timestamp)
```

```
public NSDictionary snapshotForGlobalID(com.apple.yellow.eocontrol.EOGlobalID globalID)
```

Returns the snapshot associated with *globalID*. Returns null if there isn't a snapshot for the globalID or if the corresponding timestamp is less than *timestamp*. Searches first locally (in the transaction scope) and then in the EODatabase. If *timestamp* is not provided, DistantPastTimeInterval is used.

snapshotForSourceGlobalID

```
public NSArray snapshotForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalId,
    String name,
    double timestamp)
```

```
public NSArray snapshotForSourceGlobalID(
    com.apple.yellow.eocontrol.EOGlobalID globalId,
    String name)
```

Returns the to-many snapshot for `globalId` and `name`. A to-many snapshot is 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.` Returns `null` if there isn't a to-many snapshot for `globalId` or if the timestamp is less than `timestamp`. Searches first locally (in the transaction scope) and then in the EODatabase. If `timestamp` isn't provided, `DistantPastTimeInterval` is used.

unlock

```
public void unlock()
```

Used internally to release the lock that protects access to the receiver in a multi-threaded environment.

See Also: `lock`

unregisterChannel

```
public 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

```
public int updateStrategy()
```

Returns the update strategy used by the receiver, one of:

- UpdateWithOptimisticLocking

- UpdateWithPessimisticLocking

- UpdateWithNoLocking

The default strategy is UpdateWithOptimisticLocking.

See Also: setUpdateStrategy

valuesForKeys

```
public NSDictionary valuesForKeys(
    NSArray keys,
    com.apple.yellow.eocontrol.EOEnterpriseObject 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

DatabaseChannelNeededNotification
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

EODatabaseContext's Interaction with Other Classes

The relationship between EODatabaseContext and other classes in the control and access layers is illustrated in Figure 0-3.

Figure 0-3   The Role of an EODatabaseContext
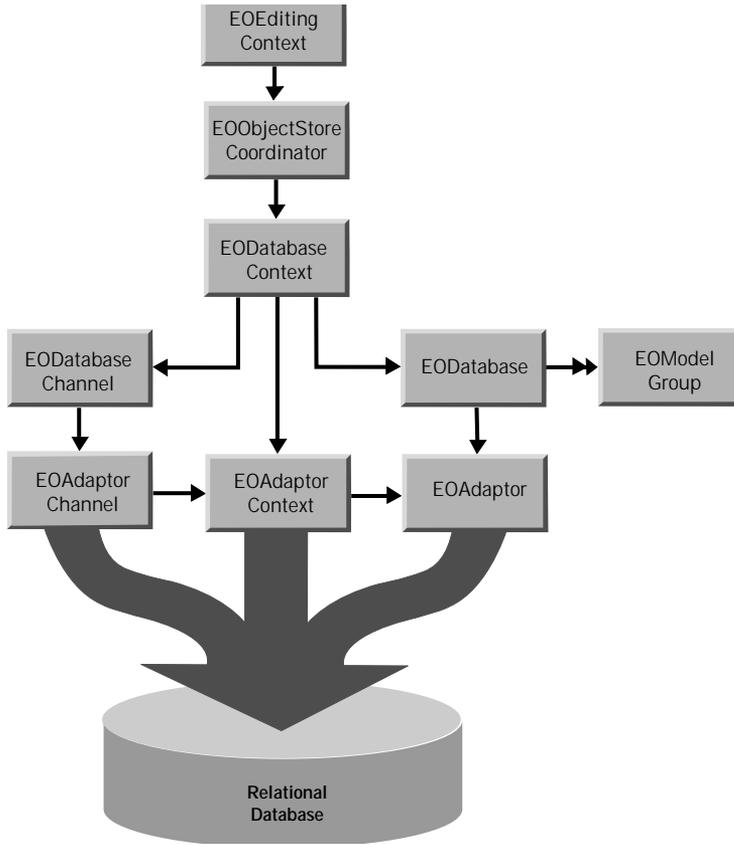
As a subclass of EOCooperatingObjectStore, EODatabaseContext acts as one of possibly several EOCooperatingObjectStores for an EOObjectStoreCoordinator, which mediates between EOEditingContexts and EOCooperatingObjectStores. (EOObjectStore, EOCooperatingObjectStore, and EOObjectStoreCoordinator are provided by the control layer.)

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 static method `registeredDatabaseContextForModel`, you should rarely need to do so. If you're using the "higher-level" objects EOEditingContexts (EOControl) 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—the interface layer's EODisplayGroup or WebObject's 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 (EOControl) posts an `CooperatingObjectStoreNeeded` 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" (page 172).

- To do performance tuning, which is described in the section "Faulting" (page 169).

- To intervene when objects are created and fetched to provide custom behavior. This is described in the section "Delegate Methods" (page 170), and in the individual delegate method descriptions in the section "Instance Methods" (page 368).

Fetching and Saving Objects

Conceptually, an EODatabaseContext fetches and saves objects on behalf of an EOEditingContext (EOControl). However, the two objects don't interact with each other directly—an EOObjectStoreCoordinator (EOControl) acts as a mediator between them. The relationship between EOEditingContext, EOObjectStoreCoordinator, and EODatabaseContext is illustrated in Figure 0-4. This configuration includes one EOObjectStoreCoordinator, and can include one or more EOEditingContexts, and one or more EODatabaseContexts.

Figure 0-4   EOEditingContexts, EOObjectStoreCoordinators, and EODatabaseContexts
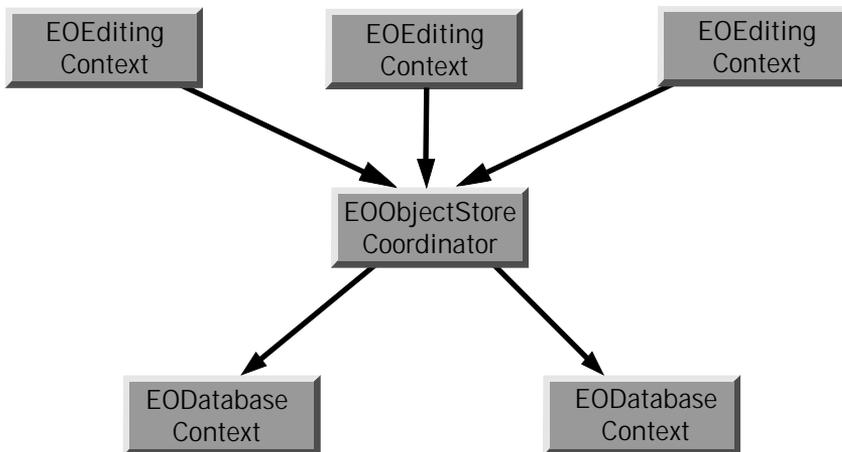
When an editing context fetches objects, the request is passed through the coordinator, which forwards it to the appropriate database context based on the fetch specification or global ID. When the database context receives a request to fetch or write information to the database, it tries to use one of its EODatabaseChannels. If all of its channels are busy, it broadcasts an `DatabaseChannelNeededNotification` 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.

Using a Custom Query

EODatabaseContext defines a hint for use with an EOFetchSpecification (EOControl) in the `objectsWithFetchSpecification` method. Named by the key `CustomQueryExpressionHintKey`, 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`.

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 aren't initially filled with fetched data. Instead, they exist without any of their values until those values are actually needed. When the "empty" destination object (called a *fault*) is accessed (sent a message), the object triggers its EODatabaseContext to fetch its data.

Faults come in two varieties: single object faults for to-one relationships, and array faults for to-many relationships. When an array fault is accessed, it fetches all of the destination objects and replaces itself with an array of those objects.

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` 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 and 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`. For more information on setting batch faulting in an EOModel, see the book *Enterprise Objects Framework Tools and Techniques.*

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:

■   `databaseContextDidSelectObjects`

■   `databaseContextShouldSelectObjects`

■   `databaseContextShouldUpdateCurrentSnapshot`

■   `databaseContextShouldUsePessimisticLock`

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 (`databaseContextNewPrimaryKey`), how and if objects are locked (`databaseContextShouldLockObjectWithGlobalID`), what fetch specification is used to fetch objects (`databaseContextShouldSelectObjects`), how batch faulting is performed

(`databaseContextShouldFetchArrayFault` and `databaseContextShouldFetchObjectFault`), and so on. For more information, see the individual delegate method descriptions in the section <u>"Instance Methods"</u> (page 368).

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 three updating strategies defined in the EODatabaseContext class as integer values:

Constant	Description
UpdateWithOptimisticLocking	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.
UpdateWithPessimisticLocking	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.
UpdateWithNoLocking	Objects are never locked. No comparisons are made between the snapshot and the row to ensure that the values in the corresponding database row haven't changed since the object was fetched.

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 to lock a database row for a particular object, or objectsWithFetchSpecification 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:	com.apple.yellow.eocontrol.EODataSource : NSObject
Package:	com.apple.yellow.eoaccess

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 identified by the 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

Constructors

    EODatabaseDataSource

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

Other

    deleteObject

    insertObject

```
dataSourceQualifiedByKey

qualifyWithRelationshipKey
```

Constructors

EODatabaseDataSource

```
public EODatabaseDataSource(
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext,
    String anEntityName)
```

Creates and returns a new EODatabaseDataSource object. The new EODatabaseDataSource
fetches objects into *anEditingContext* for the EOEntity named by *anEntityName*. If
anEditingContext's EOObjectStoreCoordinator (EOControl) doesn't have an
EODatabaseContext to service the model containing the named entity, one is created. The new
data source uses a fetch specification that fetches all the entity's objects.

```
public EODatabaseDataSource(
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext,
    String anEntityName,
    String fetchSpecificationName)
```

Creates and returns a new EODatabaseDataSource object. The new EODatabaseDataSource
fetches objects into *anEditingContext* for the EOEntity named by *anEntityName*. If
anEditingContext's EOObjectStoreCoordinator (EOControl) doesn't have an
EODatabaseContext to service the model containing the named entity, one is created. The
fetchSpecificationName argument is used to find the named fetch specification in the entity. If
the *fetchSpecificationName* is not included or is null, a new fetch specification that fetches all
the entity's objects is created.

Instance Methods

auxiliaryQualifier

`public com.apple.yellow.eocontrol.EOQualifier auxiliaryQualifier()`

Returns the EOQualifier used to further filter the objects fetched by the receiver's EOFetchSpecification (in EOControl).

See Also: `fetchSpecificationForFetch`, `fetchSpecification`

databaseContext

`public 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.)

dataSourceQualifiedByKey

`public com.apple.yellow.eocontrol.EODataSource dataSourceQualifiedByKey(String key)`

Returns a detail data source that provides the destination objects of the relationship named by *key*. The returned detail data source can be qualified by using `qualifierWithKey` to set a specific master object or to change the relationship key.

deleteObject

`public void deleteObject(Object anObject)`

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

```
public EOEntity entity()
```

Returns the EOEntity from which the receiver fetches objects.

fetchSpecification

```
public com.apple.yellow.eocontrol.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

```
public com.apple.yellow.eocontrol.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

```
public String fetchSpecificationName()
```

Returns the name of the fetch specification (or `null` if there is no name).

See Also: `setFetchSpecificationByName`

insertObject

```
public void insertObject(Object anObject)
```

Inserts *anObject* into the data source.

isFetchEnabled

`public boolean isFetchEnabled()`

Returns `true` if the receiver's `fetchObjects` method actually fetches objects, `false` 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

`public 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

`public 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.

qualifyWithRelationshipKey

```
public void qualifyWithRelationshipKey(
    String key,
    Object sourceObject)
```

Displays destination objects for the relationship named *key* belonging to *sourceObject*. *key* should be the same as the key specified in the `dataSourceQualifiedByKey` message that created the receiver. If *sourceObject* is `null`, the receiver qualifies itself to provide no objects.

setAuxiliaryQualifier

```
public void setAuxiliaryQualifier(com.apple.yellow.eocontrol.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`

setFetchEnabled

```
public void setFetchEnabled(boolean flag)
```

Controls whether the receiver can fetch. If `flag` is `true` the receiver's `fetchObjects` method actually fetches objects, if `false` 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` invokes this method to enable or disable fetching based on whether a master object is provided.

See Also: `isFetchEnabled`

setFetchSpecification

```
public void
    setFetchSpecification(com.apple.yellow.eocontrol.EOFetchSpecification fetchSpec)
```

Sets the receiver's basic EOFetchSpecification to `fetchSpec`. 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 `null`.

See Also: `setAuxiliaryQualifier`, `fetchSpecificationForFetch`, `setFetchSpecificationByName`

setFetchSpecificationByName

```
public void setFetchSpecificationByName(String 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: fetchSpecification, fetchSpecificationName

setQualifierBindings

```
public void 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

EODatabaseOperation

Inherits from:	NSObject
Package:	com.apple.yellow.eoaccess

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 `databaseContextWillOrderAdaptorOperations`).

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.

Constants

EODatabaseOperation defines the following `int` constants to identify the primitive database operation represented by an EOAdaptorOperation object or an EODatabaseOperation object:

EOAdaptorOperation Operators	EODatabaseOperation Operators
AdaptorLockOperator	DatabaseNothingOperator
AdaptorInsertOperator	DatabaseInsertOperator
AdaptorUpdateOperator	DatabaseUpdateOperator
AdaptorDeleteOperator	DatabaseDeleteOperator
AdaptorStoredProcedureOperator	

Method Types

Constructors

    EODatabaseOperation

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`

`toManySnapshots`

Constructors

EODatabaseOperation

```
public EODatabaseOperation(
    com.apple.yellow.eocontrol.EOGlobalID aGlobalID,
    Object anObject,
    EOEntity anEntity)
```

Creates and returns a new EODatabaseOperation object, setting the `object` to which the operation will be applied to *anObject*, `globalID` to *aGlobalID*, and `entity` to *anEntity*.

Instance Methods

adaptorOperations

`public 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

`public void addAdaptorOperation(EOAdaptorOperation adaptorOperation)`

Adds `adaptorOperation` to the receiver's list of adaptor operations. Throws an exception if `adaptorOperation` is `null`.

See Also: `adaptorOperations`, `removeAdaptorOperation`

databaseOperator

`public int databaseOperator()`

Returns the receiver's database operator.

dbSnapshot

`public 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: `setDatabaseOperator`

entity

```
public EOEntity entity()
```

Returns the entity that corresponds to the receiver's enterprise object.

See Also: EODatabaseOperation

globalID

```
public com.apple.yellow.eocontrol.EOGlobalID globalID()
```

Returns the EOGlobalID object that corresponds to the receiver's enterprise object.

See Also: EODatabaseOperation

newRow

```
public 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.

object

```
public Object object()
```

Returns the receiver's enterprise object.

See Also: EODatabaseOperation

primaryKeyDiffs

```
public NSDictionary primaryKeyDiffs()
```

Returns a dictionary that contains any primary key values in `newRow` that are different from those in the `dbSnapshot`. Returns `null` if the receiver doesn't have EODatabaseUpdateOperator set as its database operator.

See Also: `setDatabaseOperator`, `newRow`

recordToManySnapshot

```
public void recordToManySnapshot(
    NSArray globalIDs,
    String 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

```
public void removeAdaptorOperation(EOAdaptorOperation adaptorOperation)
```

Removes *adaptorOperation* from the receiver's list of adaptor operations.

See Also: `adaptorOperations`, `addAdaptorOperation`

rowDiffs

```
public 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

`public 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

`public void setDatabaseOperator(int` *databaseOperator*`)`

Sets the receiver's database operator. *databaseOperator* can be one of the following:

- DatabaseNothingOperator
- DatabaseInsertOperator
- DatabaseUpdateOperator
- DatabaseDeleteOperator

setDBSnapshot

`public void setDBSnapshot(NSDictionary` *dbSnapshot*`)`

Sets the snapshot for the receiver's enterprise object. If the object has just been inserted into an an EOEditingContext (EOControl), it won't have a snapshot. In this case, *dbSnapshot* should be an empty dictionary.

setNewRow

`public 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: `databaseOperator`

toManySnapshots

```
public NSDictionary toManySnapshots()
```

Returns the NSDictionary containing the snapshots for the to-many relationships of the receiver's enterprise object.

See Also: recordToManySnapshot

EOEntity

Inherits from:	NSObject
Implements:	EOPropertyListEncoding
Package:	com.apple.yellow.eoaccess

Class Description

An EOEntity 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. 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 *Enterprise Objects 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` 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 .

Constants

EOEntity defines the following String constants:

Constant	Description
`FetchAllProcedureOperation`	A stored procedure to fetch all records
`FetchWithPrimaryKeyProcedureOperation`	A stored procedure to fetch by primary key
`InsertProcedureOperation`	A stored procedure to insert a row
`DeleteProcedureOperation`	A stored procedure to delete a row
`NextPrimaryKeyProcedureOperation`	A stored procedure to generate a new primary key

Interfaces Implemented

EOPropertyListEncoding

    awakeWithPropertyList

    encodeIntoPropertyList

Method Types

Constructors

    EOEntity

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 a schema-based qualifier from a qualifier for in-memory evaluation

```
schemaBasedQualifier
```

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

externalNameForInternalName

nameForExternalName
```

Accessing whether an entity is read only

```
setReadOnly

isReadOnly
```

Accessing the user dictionary

```
setUserInfo
```

```
userInfo
```

Working with stored procedures

```
setStoredProcedure
```

```
storedProcedureForOperation
```

Working with fetch specifications

```
addFetchSpecification
```

```
fetchSpecificationNamed
```

```
fetchSpecificationNames
```

```
removeFetchSpecificationNamed
```

```
addSharedObjectFetchSpecificationByName
```

```
sharedObjectFetchSpecificationNames
```

```
setSharedObjectFetchSpecificationsByName
```

```
removeSharedObjectFetchSpecificationByName
```

Working with entity inheritance hierarchies

```
parentEntity
```

```
subEntities
```

```
addSubEntity
```

```
removeSubEntity
```

```
setIsAbstractEntity
```

```
isAbstractEntity
```

Specifying fault behavior

```
setMaxNumberOfInstancesToBatchFetch
```

```
maxNumberOfInstancesToBatchFetch
```

Caching objects

```
setCachesObjects
```

cachesObjects

Constructors

EOEntity

```
public EOEntity(
    NSDictionary propertyList,
    Object owner)
```

Creates a new EOEntity initialized from *propertyList*—a dictionary containing only property list data types (that is, NSDictionary, NSArray, NSData, and String). This constructor is used by EOModeler when it reads in an EOModel from a file, for example. The *owner* argument should be the EOEntity's EOModel. Entities created from a property list must receive an awakeWithPropertyList (EOPropertyListEncoding) message after creation before they are fully functional, but the awake... message should be deferred until the all of the other objects in the model have also been created.

See Also: encodeIntoPropertyList **(EOPropertyListEncoding)**

Static Methods

externalNameForInternalName

```
public static String externalNameForInternalName(
    String name,
    String separatorString,
    boolean 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 `false` converts "firstName" to "first_name".

nameForExternalName

```
public static String nameForExternalName(
    String name,
    String separatorString,
    boolean 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, which is upper case. 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 `true` for entities and `false` for everything else.

See Also: `beautifyNames` (EOModel), `beautifyName`, – `beautifyName` (EOAttribute, EORelationship, EOStoredProcedure)

Instance Methods

addAttribute

```
public void addAttribute(EOAttribute anAttribute)
```

Adds *anAttribute* to the receiver. Throws an exception if *anAttribute*'s name is already in use by another attribute or relationship. Sets *anAttribute*'s entity to `this`.

See Also: `removeAttribute`, `attributes`, `attributeNamed`

addFetchSpecification

```
public void addFetchSpecification(
   com.apple.yellow.eocontrol.EOFetchSpecification fetchSpec,
   String fetchSpecName)
```

Adds the fetch specification and associates `fetchSpecName` with it.

See Also: `fetchSpecificationNamed`, `fetchSpecificationNames`, `removeFetchSpecificationNamed`

addRelationship

```
public void addRelationship(EORelationship aRelationship)
```

Adds `aRelationship` to the receiver. Throws an exception if `aRelationship`'s name is already in use by another attribute or relationship. Sets `aRelationship`'s entity to `this`.

See Also: `removeRelationship`, `relationships`, `relationshipNamed`

addSharedObjectFetchSpecificationByName

```
public void addSharedObjectFetchSpecificationByName(String name)
```

Adds the fetch specification identified by `name` to the set of fetch specifications used to load objects into a shared editing context.

addSubEntity

```
public 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

`public EOAttribute anyAttributeNamed(String `*`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, `null` is returned.

See Also: `attributeNamed`, `attributes`

anyRelationshipNamed

`public EORelationship anyRelationshipNamed(String `*`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, `null` is returned.

See Also: `relationshipNamed`, `relationships`

attributeNamed

`public EOAttribute attributeNamed(String `*`attributeName`*`)`

Returns the attribute named *attributeName*, or `null` if no such attribute exists.

See Also: `anyAttributeNamed`, `attributes`, `relationshipNamed`

attributes

`public NSArray attributes()`

Returns all of the receiver's attributes, or `null` if the receiver has none.

See Also: `anyAttributeNamed`, `attributeNamed`

attributesToFetch

```
public 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

```
public 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

```
public 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

```
public boolean cachesObjects()
```

Returns `true` 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

```
public com.apple.yellow.eocontrol.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

```
public String 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

```
public 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

```
public 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

```
public 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

```
public String externalName()
```

Returns the name of the receiver as understood by the database server.

externalQuery

```
public String externalQuery()
```

Returns a query statement that's used by an EOAdaptorChannel to select rows for the receiver when a qualifier is empty, or `null` 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

```
public com.apple.yellow.eocontrol.EOFetchSpecification
   fetchSpecificationNamed(String fetchSpecName)
```

Returns the fetch specification associated with `fetchSpecName`.

See Also: `addFetchSpecification`, `fetchSpecificationNames`, `removeFetchSpecificationNamed`

fetchSpecificationNames

```
public NSArray fetchSpecificationNames()
```

Returns an alphabetically sorted array of names of the entity's fetch specifications.

See Also: `addFetchSpecification`, `fetchSpecificationNamed`, `removeFetchSpecificationNamed`

globalIDForRow

```
public com.apple.yellow.eocontrol.EOGlobalID globalIDForRow(NSDictionary row)
```

Constructs a global identifier from the specified row for the receiver.

See Also: `primaryKeyForGlobalID`

isAbstractEntity

```
public boolean isAbstractEntity()
```

Returns `true` to indicate that the receiver is abstract, `false` 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

```
public boolean isPrimaryKeyValidInObject(Object anObject)
```

Returns `true` if every key attribute is present in *anObject* and has a value that is not `null`. Returns `false` otherwise. This method uses the EOKeyValueCoding interface so a dictionary may be provided instead of an enterprise object.

See Also: `primaryKeyForRow`

isQualifierForPrimaryKey

```
public boolean
    isQualifierForPrimaryKey(com.apple.yellow.eocontrol.EOQualifier aQualifier)
```

Returns `true` if *aQualifier* describes the primary key and nothing but the primary key, `false` otherwise.

isReadOnly

```
public boolean isReadOnly()
```

Returns `true` if the receiver can't be modified, `false` 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

`public boolean isValidAttributeUsedForLocking(EOAttribute `*`anAttribute`*`)`

Returns `false` if *anAttribute* isn't an EOAttribute, if the EOAttribute doesn't belong to the receiver, or if *anAttribute* is derived. Otherwise returns `true`. An attribute that isn't valid for locking will cause `setAttributesUsedForLocking` to fail.

See Also: `attributesUsedForLocking`

isValidClassProperty

`public boolean isValidClassProperty(Object `*`aProperty`*`)`

Returns `false` if either *aProperty* isn't an EOAttribute or EORelationship, or if *aProperty* doesn't belong to the receiver. Otherwise returns `true`. 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

`public boolean isValidPrimaryKeyAttribute(EOAttribute `*`anAttribute`*`)`

Returns `false` if *anAttribute* isn't an EOAttribute, doesn't belong to the receiver, or is derived. Otherwise returns `true`.

See Also: `setPrimaryKeyAttributes`

maxNumberOfInstancesToBatchFetch

`public int maxNumberOfInstancesToBatchFetch()`

Returns the maximum number of to-one faults from the receiver to fire at one time. See the method description for `setMaxNumberOfInstancesToBatchFetch` for more explanation of what this means.

model

`public EOModel model()`

Returns the model that contains the receiver.

See Also: `addEntity` **(EOModel)**

name

`public String name()`

Returns the receiver's name.

parentEntity

`public EOEntity parentEntity()`

Returns the entity from which the receiver inherits.

See Also: `subEntities`

primaryKeyAttributeNames

`public NSArray primaryKeyAttributeNames()`

Returns an array containing the names of the attributes that make up the receiver's primary key.

See Also: `primaryKeyAttributes`

primaryKeyAttributes

`public NSArray primaryKeyAttributes()`

Returns an array of those attributes that make up the receiver's primary key.

See Also: `primaryKeyAttributeNames`

primaryKeyForGlobalID

```
public NSDictionary
    primaryKeyForGlobalID(com.apple.yellow.eocontrol.EOKeyGlobalID globalID)
```

Returns the primary key for the object identified by *globalID*.

See Also: globalIDForRow

primaryKeyForRow

```
public NSDictionary primaryKeyForRow(NSDictionary aRow)
```

Returns the primary key for *aRow*, or null if the primary key can't be computed. The primary key is a dictionary whose keys are attribute names and whose values are values for those attributes.

See Also: primaryKeyForGlobalID

primaryKeyRootName

```
public String 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

```
public com.apple.yellow.eocontrol.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

```
public boolean referencesProperty(Object aProperty)
```

Returns `true` if any of the receiver's attributes or relationships reference *aProperty*, `false` 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

```
public EORelationship relationshipNamed(String name)
```

Returns the relationship named *name*, or `null` if the receiver has no such relationship.

See Also: `anyRelationshipNamed`, `attributeNamed`, `relationships`

relationships

```
public NSArray relationships()
```

Returns all of the receiver's relationships, or `null` if the receiver has none.

See Also: `attributes`

removeAttribute

```
public 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

`public void removeFetchSpecificationNamed(String `*`fetchSpecName`*`)`

Removes the fetch specification referred to by *fetchSpecName*.

See Also: `addFetchSpecification, fetchSpecificationNamed, fetchSpecificationNames`

removeRelationship

`public 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`

removeSharedObjectFetchSpecificationByName

`public void removeSharedObjectFetchSpecificationByName(String `*`name`*`)`

Removes the fetch specification identified by *name* from the set of fetch specifications used to load objects into a shared editing context.

removeSubEntity

`public void removeSubEntity(EOEntity `*`child`*`)`

Removes *child* from the receiver's list of sub-entities.

See Also: `addSubEntity, subEntities`

restrictingQualifier

`public com.apple.yellow.eocontrol.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

schemaBasedQualifier

```
public com.apple.yellow.eocontrol.EOQualifier
    schemaBasedQualifier(com.apple.yellow.eocontrol.EOQualifier aQualifier)
```

Returns a qualifier based on *aQualifier* 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 object instance variables to resolve relationships, a database qualifier must use foreign keys. For example, consider a qualifier that is used to fetch all employees who work in a specified department:

```
Department dept;    // Assume this exists.
EOQualifier qualifier;
NSMutableArray qualArgs = new NSMutableArray();
qualArgs.addObject(dept);

qualifier = EOQualifier.qualifierWithQualifierFormat("department = %@", qualArgs);
```

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 EOControl's EOQualifierEvaluation interface 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 the EOEntity class a schemaBasedQualifier message that creates a new EOQualifier object from 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 **(EODatabaseChannel)**

setAttributesUsedForLocking

```
public boolean setAttributesUsedForLocking(NSArray attributes)
```

Sets *attributes* as the attributes used when an EODatabaseChannel locks enterprise objects for updates. Returns `false` and doesn't set the attributes used for locking if any of the attributes in *attributes* responds `false` to isValidAttributeUsedForLocking; returns `true` otherwise. See the EODatabase, EODatabaseContext, and EODatabaseChannel class specifications for information on locking.

setCachesObjects

```
public void setCachesObjects(boolean 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

```
public void setClassName(String name)
```

Assigns *name* as the name of the class associated with the receiver or "EOGenericRecord" if *name* is `null`. The specified 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.

An enterprise object class other than EOGenericRecord can be mapped to only one entity.

See Also: className

setClassProperties

```
public boolean setClassProperties(NSArray properties)
```

Sets the receiver's class properties to the EOAttributes and EORelationships in *properties* and returns `true`, unless the receiver responds `false` to isValidClassProperty for any of the objects in the array. In this event, the receiver's class properties aren't changed and `false` is returned.

setExternalName

```
public void setExternalName(String 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

```
public void setExternalQuery(String 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

```
public void setIsAbstractEntity(boolean 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

`public void setMaxNumberOfInstancesToBatchFetch(int `*`size`*`)`

Sets the maximum number of faults from the receiver to trigger at one time. By default, only one object is fetched from the database when you trigger a fault. You can optionally use this method to set to size the number of faults 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

`public void setName(String `*`name`*`)`

Sets the receiver's name to *name*. Throws an exception 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

`public boolean setPrimaryKeyAttributes(NSArray `*`keys`*`)`

If the receiver responds `false` to `isValidPrimaryKeyAttribute` for any of the objects in *keys*, this method returns `false`. Otherwise, this method sets the primary key attributes to the attributes in *keys* and returns `true`.

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. BigDecimal objects will work, but they'll entail more overhead than integers.

See Also: `isValidPrimaryKeyAttribute`

setReadOnly

```
public void setReadOnly(boolean flag)
```

Sets according to *flag* whether the database rows for the receiver can be modified by the database level objects.

See Also: isReadOnly

setRestrictingQualifier

```
public void setRestrictingQualifier(com.apple.yellow.eocontrol.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.

setSharedObjectFetchSpecificationsByName

```
public void setSharedObjectFetchSpecificationsByName(NSArray names)
```

Sets the fetch specifications used to load objects into a shared editing context to the fetch specifications identified by name in the provided array, *names*.

setStoredProcedure

```
public void setStoredProcedure(
    EOStoredProcedure storedProcedure,
    String operation)
```

Sets *storedProcedure* for *operation*. *operation* can be one of the following:

■ FetchAllProcedureOperation

■ FetchWithPrimaryKeyProcedureOperation

■ InsertProcedureOperation

■ DeleteProcedureOperation

■ NextPrimaryKeyProcedureOperation

213

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

`public 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, String, NSDictionary, NSArray, and NSData.

sharedObjectFetchSpecificationNames

`public NSArray sharedObjectFetchSpecificationNames()`

Returns an array of strings, which are the names of the fetch specifications used to load objects into a shared editing context.

storedProcedureForOperation

`public EOStoredProcedure storedProcedureForOperation(String operation)`

Returns the stored procedure for the specified `operation`, if one has been set. Otherwise, returns `null`. `operation` can be one of the following:

■ `FetchAllProcedureOperation`

■ `FetchWithPrimaryKeyProcedureOperation`

■ `InsertProcedureOperation`

■ `DeleteProcedureOperation`

■ `NextPrimaryKeyProcedureOperation`

See Also: `setStoredProcedure`, `parameterDirection` **(EOAttribute)**, `storedProcedure` **(EOAttribute)**

subEntities

`public NSArray subEntities()`

Returns a list of those entities which inherit from the receiver.

See Also: `addSubEntity, parentEntity, removeSubEntity`

userInfo

`public NSDictionary userInfo()`

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

See Also: `setUserInfo`

validateName

`public Throwable validateName(String name)`

Validates `name` and returns `null` 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.

EOEntity

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.

EOEntityClassDescription

Inherits from:	EOClassDescription (EOControl) : NSObject
Package:	com.apple.yellow.eoaccess

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. For detailed information on the methods, see the EOClassDescription class specification.

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 `ClassDescriptionNeeded...` notification (either an `ClassDescriptionNeededForClassNotification` or an `ClassDescriptionNeededForEntityNameNotification`) 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.

Constructors

EOEntityClassDescription

```
public EOEntityClassDescription(EOEntity entity)
```

Creates a new EOEntityClassDescription and assigns *entity* to it.

See Also: entity

Instance Methods

entity

```
public EOEntity entity()
```

Returns the entity associated with the receiver.

See Also: EOEntityClassDescription

EOJoin

Inherits from: NSObject

Package: com.apple.yellow.eoaccess

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

Constructors

    EOJoin

Querying the join

    destinationAttribute

    isReciprocalToJoin

    sourceAttribute

Constructors

EOJoin

```
public EOJoin(
    EOAttribute source,
    EOAttribute destination)
```

Creates and returns a new EOJoin with the given source and destination attributes. See the EORelationship class specification for an example of creating a relationship using EOJoins.

See Also: addJoin (EORelationship)

Instance Methods

destinationAttribute

```
public EOAttribute destinationAttribute()
```

Returns the destination ("right") attribute used by the join.

See Also: destinationAttributes (EORelationship)

isReciprocalToJoin

```
public boolean isReciprocalToJoin(EOJoin otherJoin)
```

Returns true 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

```
public EOAttribute sourceAttribute()
```

Returns the source ("left") attribute used by the join.

See Also: `sourceAttributes` (EORelationship**)**

EOLoginPanel

Inherits from:	NSObject
Package:	com.apple.yellow.eoaccess

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

`public NSDictionary administrativeConnectionDictionaryForAdaptor(EOAdaptor adaptor)`

Adaptor subclass should implement a subclass that implements this method. Returns `null` if the user cancels the panel.

runPanelForAdaptor

```
public NSDictionary runPanelForAdaptor(
    EOAdaptor adaptor,
    boolean flag,
    boolean allowsCreation)
```

Implemented by subclasses to run the login panel, allowing a user to enter new connection information. Returns the new connection information or `null` if the user cancels the panel. If *flag* is `true`, this method runs the login panel until the user enters valid connection information or cancels the panel. If *allowsCreation* is `true`, the panel will have an additional button that allows the user to create 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 `false`, 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

Package: com.apple.yellow.eoaccess

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 section "Loading a Model File" (page 239).

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.

Constants

EOModel defines a String constant for the name of the notification it posts. For more information, see "Notifications" (page 238).

Method Types

Constructors

    EOModel

Saving a model

    encodeTableOfContentsIntoPropertyList

    writeToFile

Loading a model's objects

    loadAllModelObjects

Working with entities

```
addEntity

removeEntity

removeEntityAndReferences

entityNames

entityNamed

entities

entitiesWithSharedObjects
```

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

Accessing prototype attributes

availablePrototypeAttributeNames

prototypeAttributeNamed

Constructors

EOModel

```
public EOModel(String path)
```

Creates a new EOModel object by reading the contents of the file identified by `path` as a model archive. Sets the EOModel's name and path from the context of the model archive. Throws an exception if for any reason it cannot initialize the model from the file specified by `path`.

```
public EOModel(
    NSDictionary tableOfContents,
    String path)
```

Creates a new EOModel object from `tableOfContents,` which is the property list representation of a EOModel). Sets the EOModel's name and path using `path`.

See Also: name, path, encodeTableOfContentsIntoPropertyList

Instance Methods

adaptorName

```
public String adaptorName()
```

Returns the name of the adaptor for the receiver. This name can be used with EOAdaptor's `adaptorWithName` static method to create an adaptor.

addEntity

```
public void addEntity(EOEntity anEntity)
```

Adds *anEntity* to the receiver. Throws an `exception` 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

```
public void addStoredProcedure(EOStoredProcedure storedProcedure)
```

Adds *storedProcedure* to the receiver. Throws an exception if an error occurs (for example, if a stored procedure of the same name is already in the receiver).

See Also: `removeStoredProcedure`, `storedProcedures`, `storedProcedureNamed`

availablePrototypeAttributeNames

```
public NSArray availablePrototypeAttributeNames()
```

Returns a list of available prototype names.

See Also: `prototypeAttributeNamed`

beautifyNames

```
public 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: `nameForExternalName` (EOEntity), `beautifyName` (EOEntity, EOAttribute, EORelationship, EOStoredProcedure), `name`

connectionDictionary

```
public 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

```
public 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: `EOModel` constructors

entities

```
public 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`

entitiesWithSharedObjects

```
public NSArray entitiesWithSharedObjects()
```

Returns an array of entities that have objects to load into a shared editing context.

entityForObject

```
public EOEntity entityForObject(Object anEO)
```

Returns the entity associated with *anEO*, whether *anEO* is an instance of an enterprise object class, an instance of EOGenericRecord, or a fault . Returns `null` if *anEO* has no associated entity.

entityNamed

```
public EOEntity entityNamed(String name)
```

Returns the entity named *name*, or `null` if no such entity exists. Posts an `EntityLoadedNotification` when the entity is loaded.

See Also: `entityNames`, `entities`

entityNames

```
public NSArray entityNames()
```

Returns an array containing the names of the EOModel's entities.

See Also: `entities`, `entityNamed`

externalModelsReferenced

```
public NSArray externalModelsReferenced()
```

Returns an array containing those models that are referenced by this model.

See Also: `referencesToProperty`

loadAllModelObjects

```
public 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

```
public EOModelGroup modelGroup()
```

Returns the model group of which the receiver is a part.

See Also: `setModelGroup`

name

```
public String name()
```

Returns the receiver's name.

See Also: `path`, `EOModel` constructors

path

```
public String path()
```

Returns the name of the EOModel file used to create the receiver, or `null` if the model wasn't initialized from a file.

See Also: `name`, `EOModel` constructors

prototypeAttributeNamed

```
public EOAttribute prototypeAttributeNamed(String attributeName)
```

Returns the prototype attribute for the given *attributeName*. It first looks for the prototype in an entity named EO*adaptorName*Prototypes (which can be in any model in the receiver's model group). If the prototype isn't found there or if the EO*adaptorName*Prototypes entity doesn't exist,

it then looks in an entity named EOPrototypes (in any model in the model group). 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

`public NSArray referencesToProperty(Object` *`aProperty`*`)`

Returns an array of all properties in the receiver that reference *`aProperty`*, whether derived attributes, relationships that reference *`aProperty`*, and so on. Returns `null` if *`aProperty`* isn't referenced by any of the properties in the model.

See Also: `externalModelsReferenced`

removeEntity

`public void removeEntity(EOEntity` *`name`*`)`

Removes the entity with the given *`name`* without performing any referential integrity checking.

See Also: `addEntity`, `removeEntityAndReferences`

removeEntityAndReferences

`public void removeEntityAndReferences(EOEntity` *`entity`*`)`

Removes *`entity`* and any attributes or relationships in other entities that reference *`entity`*.

See Also: `removeEntity`, `addEntity`

removeStoredProcedure

`public void removeStoredProcedure(EOStoredProcedure` *`storedProcedure`*`)`

Removes *`storedProcedure`* without checking to see if an entity uses it.

See Also: `addStoredProcedure`, `storedProcedures`

setAdaptorName

```
public void setAdaptorName(String adaptorName)
```

Sets the name of the receiver's adaptor to *adaptorName*.

See Also: availableAdaptorNames **(EOAdaptor)**

setConnectionDictionary

```
public 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

```
public void setModelGroup(EOModelGroup group)
```

Sets the model group of which the receiver should be a part. Note that you shouldn't change an EOModel's model group after it has been bound to other models in its group.

See Also: modelGroup

setName

```
public void setName(String name)
```

Sets the name of the receiver to *name*.

setUserInfo

```
public 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, String, NSDictionary, NSArray, and NSData.

storedProcedureNamed

`public EOStoredProcedure storedProcedureNamed(String `*`name`*`)`

Returns the stored procedure named *name*, or `null` if the model doesn't contain a stored procedure with the given name.

See Also: `storedProcedureNames, storedProcedures`

storedProcedureNames

`public NSArray storedProcedureNames()`

Returns an array containing the names of all of the model's stored procedures.

See Also: `storedProcedureNamed, storedProcedures`

storedProcedures

`public 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

`public NSDictionary userInfo()`

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

See Also: `setUserInfo`

writeToFile

`public void writeToFile(String `*`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.

Throws an exception on any error which prevents the file from being written.

See Also: `path`

Notifications

EOModel declares and posts the following notification.

EntityLoadedNotification
Posted after an EOEntity is loaded into memory. The notification contains:

Notification Object	The entity that was loaded.
Userinfo	None

EOModel

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.

To load an EOModel, provide a model file's path to the constructor. 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 create a new model from a path`initWithContentsOfFile:`. When an entity is loaded, EOModel posts an `EntityLoadedNotification`. 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
Package:	com.apple.yellow.eoaccess

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" (page 251).

Accessing Models Within a Model Group

Each model lives within a group and can form connections to other models in its group. A model can find a related model using the statement:

```
this.modelGroup().modelNamed(name);
```

A data source can locate a model using the statement:

```
EOModelGroup.defaultGroup().modelNamed(name);
```

EOModeler puts models with identical names in separate groups to allow you to load two models with the same name at the same time.

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 EOModelGroup.Delegate interface. For more information on EOModelGroup class delegate and instance delegate methods, see the EOModelGroup.ClassDelegate and EOModelGroup.Delegate interface 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:

■ classForObjectWithGlobalID

■ entityFailedToLookupClassNamed

■ entityRelationshipForRow

■ subEntityForEntity

Constants

EOModelGroup defines String constants for the names of the notifications it posts. For more information on these notifications, see "Notifications" (page 249).

Method Types

Accessing models

addModelWithPath

modelNamed

modelNames

models

modelWithPath

removeModel

Accessing model groups

defaultGroup

setDefaultGroup

globalModelGroup

modelGroupForObjectStoreCoordinator

setModelGroup

Searching a group

entityNamed

entityForObject

entitiesWithSharedObjects

fetchSpecificationNamed

storedProcedureNamed

Loading all of a group's objects

loadAllModelObjects

Assigning a delegate

classDelegate

```
delegate
setClassDelegate
setClassDelegate
```

Static Methods

classDelegate

```
public static Object classDelegate()
```

Returns the EOModelGroup's class delegate. This delegate optionally implements the `defaultModelGroup` method (see the EOModelGroup.ClassDelegate interface specification for more information).

See Also: `setClassDelegate`

defaultGroup

```
public static 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-`null` value, this method is equivalent to `globalModelGroup`.

See Also: `classDelegate`

globalModelGroup

```
public static 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`

modelGroupForObjectStoreCoordinator

```
public static EOModelGroup
   modelGroupForObjectStoreCoordinator(com.apple.yellow.eocontrol.EOObjectStoreCoordina
   tor anObjectStoreCoordinator)
```

Returns the EOModelGroup used by *anObjectStoreCoordinator*.

See Also: setModelGroup

setClassDelegate

```
public static void setClassDelegate(Object 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 EOModelGroup.ClassDelegate interface specification for more information).

See Also: classDelegate, defaultModelGroup

setDefaultGroup

```
public static void setDefaultGroup(EOModelGroup group)
```

Sets the default model group to *group*. If you've implemented the defaultModelGroup class delegate method to return a non-null value, the delegate's return value overrides *group* as the default model group.

See Also: defaultGroup,setClassDelegate

setModelGroup

```
public static void setModelGroup(
   EOModelGroup group,
   com.apple.yellow.eocontrol.EOObjectStoreCoordinator anObjectStoreCoordinator)
```

Assigns *group* to *anObjectStoreCoordinator*. By default, an EOObjectStoreCoordinator uses the defaultGroup. You might want to assign a different group to an EOObjectStoreCoordinator if you need to scope models to particular coordinators—if different models have the same name, or if different entities in different models have the same name.

See Also: modelGroupForObjectStoreCoordinator

Instance Methods

addModel

```
public void addModel(EOModel model)
```

Adds a *model* to the receiver, sets the *model*'s model group to the receiver, and posts ModelAddedNotification. Throws an exception if the receiver already contains an EOModel with the same name as the specified *model*.

addModelWithPath

```
public EOModel addModelWithPath(String path)
```

Creates an EOModel object with the contents of the file identified by *path*, and adds the newly created model to the receiver. Adds the new model to the receiver. Throws an exception if for any reason it cannot create the model from the file specified by *path*.

delegate

```
public Object 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 EOModelGroup.Delegate interface specification for more information.

See Also: setClassDelegate, classDelegate

entitiesWithSharedObjects

```
public NSArray entitiesWithSharedObjects()
```

Returns an array of entities that have objects to load into a shared editing context.

entityForObject

```
public EOEntity entityForObject(Object object)
```

Returns the EOEntity associated with *object* from any of the models in the receiver that handle *object*, or `null` if none of the entities in the receiver handles *object*.

See Also: entityForObject **(EOModel)**

entityNamed

```
public EOEntity entityNamed(String entityName)
```

Searches each of the EOModels in the receiver for the entity specified by *entityName*, and returns the entity if found. Returns `null` if it is unable to find the specified entity.

See Also: entityNamed **(EOModel)**

fetchSpecificationNamed

```
public com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecificationNamed(
    String fetchSpecName,
    String entityName)
```

Returns the named fetch specification from the entity specified by *entityName* in the receiving model group.

loadAllModelObjects

```
public 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

```
public EOModel modelNamed(String modelName)
```

Returns the EOModel named *modelName* if it's part of the receiver, or `null` if the receiver doesn't contain an EOModel with the specified name.

modelNames

```
public 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.

models

```
public 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.

modelWithPath

```
public EOModel modelWithPath(String 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 `null`. String's equals method is used to compare the paths, and each path is standardized before comparison.

See Also: `path` **(EOModel)**

removeModel

```
public void removeModel(EOModel aModel)
```

Removes *aModel* from the receiver, and unbinds any connections to *aModel* from other EOModels in the receiver. Posts `ModelInvalidatedNotification` to the default notification center after removing *aModel* from the receiver.

See Also: EOModelGroup, `models`

setDelegate

`public void setDelegate(Object `*`anObject`*`)`

Sets the receiver's delegate to *anObject*. See the EOModelGroup.Delegate interface specification for more information.

See Also: `delegate`

storedProcedureNamed

`public EOStoredProcedure storedProcedureNamed(String `*`aName`*`)`

Returns the stored procedure in the receiving model group having the given name.

Notifications

EOModelGroup declares and posts the following notifications.

ModelAddedNotification
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

ModelInvalidatedNotification

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

EOModelGroup

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:

```
String modelPath;  // Assume this exists
EOModelGroup group = new EOModelGroup();

group.addModelWithPath(modelPath);
EOModelGroup.setDefaultGroup(group);
```

EORelationship

Inherits from:	NSObject
Implements:	EOPropertyListEncoding
Package:	com.apple.yellow.eoaccess

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 *Enterprise Objects Framework 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 the sections "Creating a Simple Relationship" (page 269) and "Creating a Flattened Relationship" (page 271).

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" (page 271) 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: `InnerJoin`, `FullOuterJoin`, `LeftOuterJoin`, and `RightOuterJoin`. 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 that not all join semantics are supported by all database servers.

Constants

EORelationship defines the following `int` constants to specify the manner in which a join should be made:

- `InnerJoin`
- `FullOuterJoin`
- `LeftOuterJoin`
- `RightOuterJoin`

Interfaces Implemented

EOPropertyListEncoding

    `awakeWithPropertyList`

    `encodeIntoPropertyList`

Method Types

Constructors

    `EORelationship`

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
```

Constructors

EORelationship

```
public EORelationship(
    NSDictionary propertyList,
    Object owner)
```

Creates and returns a new EORelationship is initialized from *propertyList*—a dictionary containing only property list data types (that is, NSDictionaries, Strings, NSArrays, and next.util.ImmutableBytes). This constructor is used by EOModeler when it reads in a Model from a file, for example. The *owner* argument should be the EORelationship's Entity. EORelationships created from a property list must receive an awakeWithPropertyList message immediately after creation before they are fully functional, but the awake... message should be deferred until the all of the other objects in the model have also been created.

See Also: encodeIntoPropertyList **(EOPropertyListEncoding interface)**

Instance Methods

addJoin

```
public void addJoin(EOJoin aJoin)
```

Adds a source-destination attribute pair to the relationship. Throws an exception 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

`public EORelationship anyInverseRelationship()`

Searches the relationship's destination entity for a user-created, back-referencing 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

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

`public NSArray componentRelationships()`

Returns an array of base relationships making up a flattened relationship, or `null` if the relationship isn't flattened.

See Also: `definition`

definition

`public String definition()`

Returns the data path of a flattened relationship; for example "department.facility". If the relationship isn't flattened, `definition` returns `null`.

See Also: `componentRelationships`

259

deleteRule

```
public int deleteRule()
```

Returns a rule that describes the action to take when an object is being deleted. The returned rule is one of the following integers (defined in the control layer's EOClassDescription class):

Value	Description
EOClassDescription. DeleteRuleNullify	Delete the department and remove any back reference the employee has to the department.
EOClassDescription. DeleteRuleCascade	Delete the department and all of the employees it contains.
EOClassDescription. DeleteRuleDeny	Refuse the deletion if the department contains employees.
EOClassDescription. DeleteRuleNoAction	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

```
public NSArray destinationAttributes()
```

Returns the destination attributes of the relationship. These correspond one-to-one with the attributes returned by `sourceAttributes`. Returns `null` if the relationship is flattened.

See Also: `joins`, `destinationAttribute` **(EOJoin)**

destinationEntity

```
public 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

```
public EOEntity entity()
```

Returns the relationship's source entity.

See Also: `destinationEntity`, `addRelationship` **(EOEntity)**

inverseRelationship

```
public EORelationship inverseRelationship()
```

Searches the relationship's destination entity for a user-created, back-referencing relationship joining on the same keys. Returns the inverse relationship if one is found, `null` otherwise.

See Also: `anyInverseRelationship`

isCompound

```
public boolean isCompound()
```

Returns `true` if the relationship contains more than one join (that is, if it joins more than one pair of attributes), `false` if it has only one join. See <u>"Creating a Simple Relationship"</u> (page 269) for information on compound relationships.

See Also: `joins`, `joinSemantic`

isFlattened

```
public boolean isFlattened()
```

Returns `true` if the relationship traverses more than two entities, `false` otherwise. See <u>"Creating a Flattened Relationship"</u> (page 271) for an example of a flattened relationship.

isMandatory

```
public boolean isMandatory()
```

Returns `true` if the target of the relationship is required, `false` if it can be `null`.

See Also: `setIsMandatory`

isToMany

```
public boolean isToMany()
```

Returns `true` if the relationship is to-many, `false` if it's to-one.

See Also: `setToMany`

joinSemantic

```
public int joinSemantic()
```

Returns the semantic used to create SQL expressions for this relationship. The returned join semantic is one of the following:

Constant	Description
InnerJoin	Produces results only for destinations of the join relationship that have non-NULL values.
FullOuterJoin	Produces results for all source records, regardless of the values of the relationships.
LeftOuterJoin	Preserves rows in the left (source) table, keeping them even if there's no corresponding row in the right table.
RightOuterJoin	Preserves rows in the right (destination) table, keeping them even if there's no corresponding row in the left table.

See Also: `joins`

joins

```
public NSArray joins()
```

Returns all joins used by relationship.

See Also: `destinationAttributes`, `joinSemantic`, `sourceAttributes`

name

```
public String name()
```

Returns the relationship's name.

numberOfToManyFaultsToBatchFetch

```
public int numberOfToManyFaultsToBatchFetch()
```

Returns the number of to-many faults that are triggered at one time.

ownsDestination

```
public boolean ownsDestination()
```

Returns `true` if the receiver's source object owns its destination objects, `false` otherwise. See the method description for `setOwnsDestination` for more discussion of this topic.

See Also: `destinationAttributes`

propagatesPrimaryKey

```
public boolean propagatesPrimaryKey()
```

Returns `true` 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

```
public EOQualifier qualifierWithSourceRow(NSDictionary sourceRow)
```

Returns a qualifier that can be used to fetch the destination of the receiving relationship, given *sourceRow*.

referencesProperty

`public boolean referencesProperty(Object `*`aProperty`*`)`

Returns `true` if *`aProperty`* is in the relationship's data path or is an attribute belonging to one of the relationship's joins; otherwise, it returns `false`. See the class description for information on how relationships reference properties.

See Also: `referencesProperty` **(EOEntity)**

removeJoin

`public void removeJoin(EOJoin `*`aJoin`*`)`

Deletes *`aJoin`* from the relationship. Does nothing if the relationship is flattened.

See Also: `addJoin`

setDefinition

`public void setDefinition(String `*`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" (page 271) for more information on flattened relationships.

See Also: `addJoin`, `setEntity`

setDeleteRule

`public void setDeleteRule(int `*`deleteRule`*`)`

Set a rule describing the action to take when object is being deleted. *`deleteRule`* can be one of the following (defined in the control layer's EOClassDescription):

■ `EOClassDescription.DeleteRuleNullify`

■ `EOClassDescription.DeleteRuleCascade`

■   `EOClassDescription.DeleteRuleDeny`

■   `EOClassDescription.DeleteRuleNoAction`

For more discussion of what these rules mean, see the method description for `deleteRule`.

setEntity

`public 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

`public void setIsMandatory(boolean `*`flag`*`)`

Specifies according to *`flag`* whether the target of the relationship must be supplied or can be `null`.

setJoinSemantic

`public void setJoinSemantic(int `*`joinSemantic`*`)`

Sets the semantic used to create SQL expressions for this relationship. *`joinSemantic`* should be one of the following:

■   `InnerJoin`

■   `FullOuterJoin`

■   `LeftOuterJoin`

■   `RightOuterJoin`

See Also: `addJoin`, `joinSemantic`

setName

```
public void setName(String name)
```

Sets the relationship's name to *name*. Throws a verification exception if *name* is not a valid relationship name, and an invalid argument exception 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

```
public void setNumberOfToManyFaultsToBatchFetch(int size)
```

Sets the number of "toMany" faults that are fired at one time to *size*.

See Also: `isToMany`, `numberOfToManyFaultsToBatchFetch`

setOwnsDestination

```
public void setOwnsDestination(boolean flag)
```

Sets according to *flag* whether a receiver's source object owns its destination objects. The default is `false`. 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

```
public void setPropagatesPrimaryKey(boolean 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

```
public void setToMany(boolean flag)
```

Sets a simple relationship as to-many according to *flag*. Throws an exception if the receiver is flattened. See the class description for considerations in setting this flag.

See Also: isFlattened

setUserInfo

```
public 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, String, NSArray, and NSData).

sourceAttributes

```
public NSArray sourceAttributes()
```

Returns the source attributes of a simple (non-flattened) relationship. These correspond one-to-one with the attributes returned by destinationAttributes. Returns null if the relationship is flattened.

See Also: joins, sourceAttribute **(EOJoin)**

userInfo

```
public NSDictionary userInfo()
```

Returns a dictionary of user data. Your application can use this data for whatever it needs.

validateName

```
public void validateName(String name)
```

Validates name and returns null 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

```
public Object validateValue(Object value)
```

For relationships marked as mandatory, throws a validation exception if the receiver is to-one and value is null, or if the receiver is to-many an value has a count of 0. A mandatory relationship is one in which the target of the relationship is required. Returns null to indicate success.

See Also: isMandatory, setIsMandatory

EORelationship

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 = new EOJoin(movieStudioIDAttribute, studioIDAttribute);

toStudioRelationship = new EORelationship();
```

```
toStudioRelationship.setName("studio");
movieEntity.addRelationship(toStudioRelationship);
toStudioRelationship.addJoin(toStudioJoin);
toStudioRelationship.setToMany(false);
toStudioRelationship.setJoinSemantic(EORelationship.InnerJoin);
```

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, `InnerJoin` 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 = new EOJoin(studioIDAttribute, movieStudioIDAttribute);
toMoviesRelationship = new EORelationship();

toMoviesRelationship.setName("movies");
studioEntity.addRelationship(toMoviesRelationship);
toMoviesRelationship.addJoin(toMoviesJoin);
toMoviesRelationship.setToMany(true);
toMoviesRelationship.setJoinSemantic(EORelationship.InnerJoin);
```

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 = new EORelationship();

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
Package:	com.apple.yellow.eoaccess

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 static method `expressionForString`, and send the expression object to an adaptor channel using EOAdaptorChannel's `evaluateExpression` method.

For more information on using EOSQLExpressions, see the following sections:

Constants

EOSQLExpression defines the following String constants.

Constant	Description
BindVariableNameKey	The key for the name of a bind variable in a bind variable dictionary.
BindVariablePlaceHolderKey	A key for use in bind variable dictionaries. The corresponding value is the placeholder string to be used in SQL.
BindVariableAttributeKey	A key for use in bind variable dictionaries. The corresponding value is the attribute that uses the bind variable.
BindVariableValueKey	A key for use in bind variable dictionaries. The corresponding value is the value for the bind variable.
CreateTablesKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to create tables.
DropTablesKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to drop tables.
CreatePrimaryKeySupportKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to create primary key support.

Constant	Description
DropPrimaryKeySupportKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to drop primary key support.
PrimaryKeyConstraintsKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to create primary key constraints.
ForeignKeyConstraintsKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to create foreign key constraints.
CreateDatabaseKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to create a database.
DropDatabaseKey	Key for use in options dictionaries. A corresponding value of "YES" indicates that the EOSQLExpression should generate SQL to drop a database.
AllowsNullKey	Key for use in change dictionaries. A corresponding value indicates that the column's allows NULL value should be changed from.
ColumnNameKey	Key for use in change dictionaries. A corresponding value indicates that the column's allows NULL value should be changed from.
ExternalNameKey	Key for use in change dictionaries. A corresponding value indicates that the column's allows NULL value should be changed from.
ExternalTypeKey	Key for use in change dictionaries. A corresponding value indicates that the column's allows NULL value should be changed from.

Constant	Description
NameKey	Key for use in change dictionaries. A corresponding value indicates the old value of the table or column.
PrecisionKey	Key for use in change dictionaries. A corresponding value indicates the value a column's precision should be changed from.
RelationshipsKey	Key for use in change dictionaries. The corresponding value is a dictionary of relationships which have been modified since the last time the model and schema were sychronized. For more information see "The Change Dictionary" (page 328).
ScaleKey	Key for use in change dictionaries. A corresponding value indicates the value the column's scale should be changed from.
WidthKey	Key for use in change dictionaries. A corresponding value indicates the value the column's width should be changed from.

Method Types

Constructors

EOSQLExpression

Creating an EOSQLExpression object

selectStatementForAttributes

insertStatementForRow

updateStatementForRow

deleteStatementWithQualifier

`expressionForString`

Building SQL Expressions

`prepareSelectExpressionWithAttributes`

`prepareInsertExpressionWithRow`

`prepareUpdateExpressionWithRow`

`prepareDeleteExpressionForQualifier`

`setStatement`

`statement`

Generating SQL for attributes and values

`formatSQLString`

`formatValueForAttribute`

`formatStringValue`

`sqlStringForValue`

`sqlStringForAttributeNamed`

`sqlStringForAttribute`

`sqlStringForAttributePath`

`sqlStringForNumber`

`sqlStringForString`

Generating SQL for names of database objects

`sqlStringForSchemaObjectName`

`setUseQuotedExternalNames`

`useQuotedExternalNames`

`externalNameQuoteCharacter`

Generating an attribute list

`addSelectListAttribute`

`addInsertListAttribute`

`addUpdateListAttribute`

```
appendItemToListString
```

```
listString
```

Generating a value list

```
addInsertListAttribute
```

```
addUpdateListAttribute
```

```
valueList
```

Generating a table list

```
tableListWithRootEntity
```

```
aliasesByRelationshipPath
```

Generating the join clause

```
joinExpression
```

```
addJoinClause
```

```
assembleJoinClause
```

```
joinClauseString
```

Generating a search pattern

```
sqlPatternFromShellPattern
```

```
sqlPatternFromShellPatternWithEscapeCharacter
```

Generating a relational operator

```
sqlStringForSelector
```

Accessing the where clause

```
whereClauseString
```

Generating an order by clause

```
addOrderByAttributeOrdering
```

```
orderByString
```

Accessing the lock clause

```
lockClause
```

Assembling a statement

    assembleSelectStatementWithAttributes

    assembleInsertStatementWithRow

    assembleUpdateStatementWithRow

    assembleDeleteStatementWithQualifier

Generating SQL for qualifiers

    sqlStringForQualifier

    sqlStringForConjoinedQualifiers

    sqlStringForDisjoinedQualifiers

    sqlStringForKeyComparisonQualifier

    sqlStringForKeyValueQualifier

    sqlStringForNegatedQualifier

Managing bind variables

    setUseBindVariables

    useBindVariables

    addBindVariableDictionary

    bindVariableDictionaries

    bindVariableDictionaryForAttribute

    mustUseBindVariableForAttribute

    shouldUseBindVariableForAttribute

Using table aliases

    setUseAliases

    useAliases

Accessing the entity

    entity

Creating a schema generation script

    schemaCreationScriptForEntities

schemaCreationStatementsForEntities

appendExpression

createTableStatementsForEntityGroup

createTableStatementsForEntityGroups

dropTableStatementsForEntityGroup

dropTableStatementsForEntityGroups

primaryKeyConstraintStatementsForEntityGroup

primaryKeyConstraintStatementsForEntityGroups

primaryKeySupportStatementsForEntityGroup

primaryKeySupportStatementsForEntityGroups

dropPrimaryKeySupportStatementsForEntityGroup

dropPrimaryKeySupportStatementsForEntityGroups

addCreateClauseForAttribute

columnTypeStringForAttribute

allowsNullClauseForConstraint

foreignKeyConstraintStatementsForRelationship

prepareConstraintStatementForRelationship

createDatabaseStatementsForConnectionDictionary

dropDatabaseStatementsForConnectionDictionary

Synchronizing the database with a model

statementsToUpdateObjectStoreForModel

statementsToUpdateObjectStoreForEntityGroup

statementsToCopyTableNamed

phraseCastingColumnNamed

statementsToRenameTableNamed

statementsToInsertColumnForAttribute

statementsToDeleteColumnNamed

statementsToRenameColumnNamed

statementsToModifyColumnNullRule

statementsToConvertColumnType

isColumnTypeEquivalentToColumnType

statementsToDropForeignKeyConstraintsOnEntityGroup

statementsToDropPrimaryKeyConstraintsOnEntityGroup

statementsToDropPrimaryKeySupportForEntityGroup

statementsToImplementForeignKeyConstraintsOnEntityGroup

statementsToImplementPrimaryKeyConstraintsOnEntityGroup

statementsToImplementPrimaryKeySupportForEntityGroup

Querying about database synchronization support

supportsSchemaSynchronization

supportsDirectColumnCoercion

supportsDirectColumnDeletion

supportsDirectColumnInsertion

supportsDirectColumnNullRuleModification

supportsDirectColumnRenaming

Constructors

EOSQLExpression

public EOSQLExpression(EOEntity *anEntity*)

Creates a new EOSQLExpression rooted to *anEntity*.

See Also: entity

Static Methods

appendExpression

```
public static void appendExpression(
    EOSQLExpression anSQLExpression,
    String script)
```

Append's *anSQLExpression*'s statement to *script* along with any necessary delimiter. EOSQLExpression's implementation appends the SQL statement for *anSQLExpression* to *script* followed by a semicolon and a newline. A subclass of EOSQLExpression only needs to override this method if the delimiter for its database server is different. For example, the Oracle and Informix use the default implementation, whereas the Sybase adaptor appends the word "go" instead of a semicolon.

See Also: createTableStatementsForEntityGroup

createDatabaseStatementsForConnectionDictionary

```
public static NSArray createDatabaseStatementsForConnectionDictionary(
    NSDictionary connectionDictionary,
    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

createTableStatementsForEntityGroup

```
public static NSArray createTableStatementsForEntityGroup(NSArray entityGroup)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create a table for *entityGroup*, an array of EOEntity objects that have the same externalName. Returns an empty array if *entityGroup* is null or empty.

EOSQLExpression's implementation does the following:

1. Creates an EOSQLExpression object.

2. Sets the expression's `entity` to the first entity in *entityGroup*.

3. Adds a create clause for each Attribute in *entityGroup*'s Entities.

4. Sets the expression's `statement` to CREATE TABLE *TABLE_NAME* (*LIST_STRING*), where *TABLE_NAME* is the `externalName` of the Entity objects in *entityGroup* and *LIST_STRING* is the expression's `listString`.

5. Adds the expression to an array.

6. Returns the array.

The following is an example of a CREATE TABLE statement produced by the default implementation:

```
create table EMPLOYEE (
    EMP_ID      int not null,
    DEPT_ID     int null,
    LAST_NAME   varchar(40) not null,
    PHONE       char(12) null,
    HIRE_DATE   date null,
    SALARY      number(7, 2) null
)
```

If a subclass's database server's table creation semantics are different, the subclass should override this method or one or more of the following methods as appropriate:

■   `addCreateClauseForAttribute`

■   `columnTypeStringForAttribute`

■   `allowsNullClauseForConstraint`

See Also: `createTableStatementsForEntityGroup`, `dropTableStatementsForEntityGroup`

createTableStatementsForEntityGroups

```
public static NSArray createTableStatementsForEntityGroups(NSArray entityGroups)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create the tables specified in *entityGroups*. An entity group is an array of Entity objects that have the same `externalName`, and *entityGroups* is an array of entity groups. Returns an empty array if

entityGroups is `null` or empty. EOSQLExpression's implementation invokes `createTableStatementsForEntityGroup` for each entity group in *entityGroups* and returns an array of all the resulting EOSQLExpressions.

See Also: `schemaCreationStatementsForEntities`

deleteStatementWithQualifier

```
public static EOSQLExpression deleteStatementWithQualifier(
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    Object 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. Throws an `exception` if qualifier is `null`.

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(true)` message prior to invoking `super`'s version.

dropDatabaseStatementsForConnectionDictionary

```
public static NSArray dropDatabaseStatementsForConnectionDictionary(
    NSDictionary connectionDictionary,
    NSDictionary adminDictionary)
```

Generates the SQL statements to drop a database (or user, for Oracle).

See Also: `createDatabaseStatementsForConnectionDictionary`

dropPrimaryKeySupportStatementsForEntityGroup

`public static NSArray dropPrimaryKeySupportStatementsForEntityGroup(NSArray entityGroup)`

Returns an array of EOSQLExpression objects that define the SQL necessary to drop the primary key generation support for `entityGroup`, an array of Entity objects that have the same `externalName`. The drop statement generated by this method should be sufficient to remove the primary key support created by `primaryKeySupportStatementsForEntityGroup`'s statements.

EOSQLExpression's implementation creates a statement of the following form:

`drop sequence SEQUENCE_NAME`

Where `SEQUENCE_NAME` is the `primaryKeyRootName` for the first entity in `entityGroup` concatenated with "_SEQ" (EMP_ID_SEQ, for example).

If a subclass uses a different primary key generation mechanism or if the subclass's database server's drop semantics are different, the subclass should override this method.

dropPrimaryKeySupportStatementsForEntityGroups

`public static NSArray`
`    dropPrimaryKeySupportStatementsForEntityGroups(NSArray entityGroups)`

Returns an array of EOSQLExpression objects that define the SQL necessary to drop the primary key generation support for the entities specified in `entityGroups`. An entity group is an array of EOEntity objects that have the same `externalName`, and `entityGroups` is an array of entity groups. EOSQLExpression's implementation invokes `dropPrimaryKeySupportStatementsForEntityGroup` for each entity group in `entityGroups` and returns an array of all the resulting EOSQLExpressions.

See Also: `schemaCreationStatementsForEntities`

dropTableStatementsForEntityGroup

`public static NSArray dropTableStatementsForEntityGroup(NSArray entityGroup)`

Returns an array of EOSQLExpression objects that define the SQL necessary to drop the table identified by `entityGroup`, an array of Entity objects that have the same `externalName`. The drop statement generated by this method should be sufficient to remove the table created by `createTableStatementsForEntityGroup`'s statements.

EOSQLExpression's implementation creates a statement of the following form:

```
DROP TABLE TABLE_NAME
```

Where `TABLE_NAME` is the `externalName` of the first entity in `entityGroup`.

If a subclass's database server's drop semantics are different, the subclass should override this method.

dropTableStatementsForEntityGroups

```
public static NSArray dropTableStatementsForEntityGroups(NSArray entityGroups)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to drop the tables for `entityGroups`. An entity group is an array of Entity objects that have the same `externalName`, and `entityGroups` is an array of entity groups. EOSQLExpression's implementation invokes `dropTableStatementsForEntityGroup` for each entity group in `entityGroups` and returns an array of all the resulting EOSQLExpressions.

See Also: schemaCreationStatementsForEntities

expressionForString

```
public static EOSQLExpression expressionForString(String 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

foreignKeyConstraintStatementsForRelationship

```
public static NSArray
    foreignKeyConstraintStatementsForRelationship(EORelationship aRelationship)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create foreign key constraints for `aRelationship`. EOSQLExpression's implementation generates statements such as the following:

```
ALTER TABLE EMPLOYEE ADD CONSTRAINT TO_DEPARTMENT FOREIGN KEY (DEPT_ID)
        REFERENCES DEPARTMENT(DEPT_ID)
```

It returns an empty array if either of the following are true:

■ *aRelationship* spans models (if *aRelationship*'s destinationEntity is in a different model than *aRelationship*'s source entity)

■ *aRelationship* is a to-many relationship, or if the inverse relationship of *aRelationship* is not a to-many. In other words, foreign key constraint statements are only created for to-one relationships whose inverse is a to-many.

If neither of the above are true, this method creates a new EOSQLExpression, assigns its entity to *aRelationship*'s entity, invokes prepareConstraintStatementForRelationship, and returns an array containing the expression.

If a subclass's database server's foreign key constraint semantics are different, the subclass should override this method or override the method prepareConstraintStatementForRelationship.

See Also: schemaCreationStatementsForEntities

formatSQLString

```
public static String formatSQLString(
    String sqlString,
    String format)
```

Applies *format* (an EOAttribute object's "read" or "write" format) to *sqlString* (a value for the attribute). If *format* is null, this method returns *sqlString* unchanged.

See Also: readFormat (EOAttribute), writeFormat **(EOAttribute)**

formatStringValue

```
public static String formatStringValue(String 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*. Throws an exception if *string* is null.

formatValueForAttribute

```
public static String formatValueForAttribute(
   Object value,
   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

```
public static EOSQLExpression insertStatementForRow(
   NSDictionary row,
   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`. Throws an exception if entity is `null`.

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(true)` message prior to invoking `super`'s version.

isColumnTypeEquivalentToColumnType

```
public static boolean isColumnTypeEquivalentToColumnType(
   EOSQLExpression.EOColumnTypes columnTypeA,
   EOSQLExpression.EOColumnTypes columnTypeB,
   NSDictionary options)
```

Returns `true` if values in a column of `columnTypeA` can be copied into a column of `columnTypeB` without the use of a casting phrase, `false` otherwise. The `options` dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

phraseCastingColumnNamed

```
public static String phraseCastingColumnNamed(
    String columnName,
    EOSQLExpression.EOColumnTypes fromType,
    EOSQLExpression.EOColumnTypes castType
    NSDictionary options)
```

Returns an SQL string to cast the values in the column specified by *columnName* to a new type. This method is used when the adaptor doesn't support in-place column type coercion, and the table has to be recreated. To move data from the old table to the new table, sometimes a conversion statement is needed (for example, to convert strings in a VARCHAR column to numbers). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see <u>"The Options Dictionary"</u> (page 326).

primaryKeyConstraintStatementsForEntityGroup

```
public static NSArray primaryKeyConstraintStatementsForEntityGroup(NSArray entityGroup)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create the primary key constraints for *entityGroup*, an array of EOEntity objects that have the same externalName. Returns an empty array if any of the primary key attributes in *entityGroup* don't have a columnName.

EOSQLExpression's implementation creates a statement of the following form:

```
ALTER TABLE TABLE_NAME ADD PRIMARY KEY (PRIMARY_KEY_COLUMN_NAMES)
```

Where *TABLE_NAME* is the externalName for the first entity in *entityGroup* and *PRIMARY_KEY_COLUMN_NAMES* is a comma-separated list of the columnNames of the first entity's primaryKeyAttributes.

If the subclass's database server's primary key constraint semantics are different, the subclass should override this method.

primaryKeyConstraintStatementsForEntityGroups

```
public static NSArray
   primaryKeyConstraintStatementsForEntityGroups(NSArray entityGroups)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create the primary key constraints for the Entities specified in *entityGroups*. An entity group is an array of Entity objects that have the same externalName, and *entityGroups* is an array of entity groups. EOSQLExpression's implementation invokes primaryKeySupportStatementsForEntityGroup for each entity group in *entityGroups* and returns an array of all the resulting EOSQLExpressions.

primaryKeySupportStatementsForEntityGroup

```
public static NSArray primaryKeySupportStatementsForEntityGroup(NSArray entityGroup)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create the primary key generation support for *entityGroup*, an array of EOEntity objects that have the same externalName. EOSQLExpression's implementation creates a statement of the following form:

```
create sequence SEQUENCE_NAME
```

Where *SEQUENCE_NAME* is the primaryKeyRootName for the first entity in *entityGroup* concatenated with "_SEQ" (EMP_ID_SEQ, for example).

If a subclass uses a different primary key generation mechanism or if the subclass's database server's drop semantics are different, the subclass should override this method.

See Also: dropPrimaryKeySupportStatementsForEntityGroup, primaryKeyForNewRowWithEntity **(EOAdaptorChannel)**

primaryKeySupportStatementsForEntityGroups

```
public static NSArray primaryKeySupportStatementsForEntityGroups(NSArray entityGroups)
```

Returns an array of EOSQLExpression objects that define the SQL necessary to create the primary key generation support for the Entities specified in *entityGroups*. An entity group is an array of Entity objects that have the same externalName, and *entityGroups* is an array of entity groups. EOSQLExpression's implementation invokes primaryKeySupportStatementsForEntityGroup for each entity group in *entityGroups* and returns an array of all the resulting EOSQLExpressions.

schemaCreationScriptForEntities

```
public static String schemaCreationScriptForEntities(
    NSArray entities,
    NSDictionary options)
```

Returns a script of SQL statements suitable to create the schema for the EOEntity objects in *entities*. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326). EOSQLExpression's implementation invokes schemaCreationStatementsForEntities with *entities* and *options* and then uses appendExpression to generate the script from the EOSQLExpressions generated by schemaCreationStatementsForEntities.

schemaCreationStatementsForEntities

```
public static NSArray schemaCreationStatementsForEntities(
    NSArray entities,
    NSDictionary options)
```

Returns an array of EOSQLExpressions suitable to create the schema for the Entity objects in *entities*. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

EOSQLExpression's implementation uses the following methods:

- createTableStatementsForEntityGroups

- dropTableStatementsForEntityGroups

- primaryKeySupportStatementsForEntityGroups

- dropPrimaryKeySupportStatementsForEntityGroups

- primaryKeyConstraintStatementsForEntityGroups

- foreignKeyConstraintStatementsForRelationship

to generate EOSQLExpressions for the support identified in *options*.

See Also: schemaCreationScriptForEntities

selectStatementForAttributes

```
public static EOSQLExpression selectStatementForAttributes(
    NSArray attributes,
    boolean flag,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    EOEntity entity)
```

Creates and returns an SQL SELECT expression. Creates an instance of EOSQLExpression, initializes it with `entity`, and sends it `prepareSelectExpressionWithAttributes`. The expression created with this method uses table aliases. Throws an exception if attributes is `null` or empty, `fetchSpecification` is `null`, or `entity` is `null`.

The expression created with this method uses table aliases. To generate SELECT statements that don't use them, you must override `prepareSelectExpressionWithAttributes` to send a `setUseAliases(false)` message prior to invoking `super`'s version.

setUseBindVariables

```
public static void setUseBindVariables(boolean 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 `true`, though, instances do use them. For more information on bind variables, see the discussion in the class description.

See Also: `useBindVariables`

setUseQuotedExternalNames

```
public static void setUseQuotedExternalNames(boolean flag)
```

Sets whether all instances of EOSQLExpression subclasses quote external names when they are referenced in SQL statements. By setting `flag` to `true`, 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 `true`, though, instances do use quotes.

See Also: `useQuotedExternalNames`, `sqlStringForSchemaObjectName`, `externalNameQuoteCharacter`

sqlPatternFromShellPattern

```
public static String sqlPatternFromShellPattern(String pattern)
```

Translates a "like" qualifier to an SQL "like" expression. Invoked from `sqlStringForKeyValueQualifier` when the qualifier argument is an EOKeyValueQualifier object whose selector is `EOQualifier.QualifierOperatorLike`. 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: `sqlPatternFromShellPatternWithEscapeCharacter`

sqlPatternFromShellPatternWithEscapeCharacter

```
public static String sqlPatternFromShellPatternWithEscapeCharacter(
    String pattern,
    char escapeCharacter)
```

Like `sqlPatternFromShellPattern` except the argument `escapeCharacter` allows you to specify a character for escaping the wild card characters "%" and "_".

statementsToConvertColumnType

```
public static NSArray statementsToConvertColumnType(
    String columnName,
    String tableName,
    EOSQLExpression.EOColumnTypes type,
    EOSQLExpression.EOColumnTypes newType,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to convert in place the type of the specified column. The `options` dictionary describes the aspects of the schema for which to create SQL statements; for more information, see <u>"The Options Dictionary"</u> (page 326).

statementsToCopyTableNamed

```
public static NSArray statementsToCopyTableNamed(
    String tableName,
    NSArray entityGroup,
    NSDictionary changes,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to copy the specified table into a new table, whose definition is provided by *entityGroup*—an array of EOEntity objects rooted to the table named *tableName*. This method is used when the adaptor doesn't support the in-place table modifications required to synchronize the database to a model.

The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToDeleteColumnNamed

```
public static NSArray statementsToDeleteColumnNamed(
    String columnName,
    String tableName,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to delete in place the specified column from the specified table. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToDropForeignKeyConstraintsOnEntityGroup

```
public static NSArray statementsToDropForeignKeyConstraintsOnEntityGroup(
    NSArray entityGroup,
    NSDictionary changes,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to drop foreign key constraints for the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToDropPrimaryKeyConstraintsOnEntityGroup

```
public static NSArray statementsToDropPrimaryKeyConstraintsOnEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to drop primary key constraints for the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToDropPrimaryKeySupportForEntityGroup

```
public static NSArray statementsToDropPrimaryKeySupportForEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to drop the primary key support mechanism for the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToImplementForeignKeyConstraintsOnEntityGroup

```
public static NSArray statementsToImplementForeignKeyConstraintsOnEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to implement foreign key constraints on the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToImplementPrimaryKeyConstraintsOnEntityGroup

```
public static NSArray statementsToImplementPrimaryKeyConstraintsOnEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to implement primary key constraints on the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToImplementPrimaryKeySupportForEntityGroup

```
public static NSArray statementsToImplementPrimaryKeySupportForEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to implement support mechanisms for primary key generation for the table corresponding to *entityGroup*—an array of EOEntity objects rooted to the same table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToInsertColumnForAttribute

```
public static NSArray statementsToInsertColumnForAttribute(
   EOAttribute attribute,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to insert in place a column for the specified attribute. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToModifyColumnNullRule

```
public static NSArray statementsToModifyColumnNullRule(
    String columnName,
    String tableName,
    boolean allowsNull,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to modify in place the specified column to either allow or not allow NULL values as specified by *allowsNull*. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToRenameColumnNamed

```
public static NSArray statementsToRenameColumnNamed(
    String columnName,
    String tableName,
    String newName,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to rename in place the specified column. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToRenameTableNamed

```
public static NSArray statementsToRenameTableNamed(
    String tableName,
    String newName,
    NSDictionary options)
```

Returns an array of EOSQLExpressions to rename in place the specified table. The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToUpdateObjectStoreForEntityGroup

```
public static NSArray statementsToUpdateObjectStoreForEntityGroup(
   NSArray entityGroup,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to update the table that corresponds to *entityGroup*—an array of EOEntity objects rooted to the same table. Inserts and deletes columns, and updates modified columns. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

statementsToUpdateObjectStoreForModel

```
public static NSArray statementsToUpdateObjectStoreForModel(
   EOModel model,
   NSDictionary changes,
   NSDictionary options)
```

Returns an array of EOSQLExpressions to synchronize the database with *model*. Prepares the statements to insert and delete new and deleted tables before invoking `statementsToUpdateObjectStoreForEntityGroup` for each modified table. The *changes* dictionary identifies the changes to make to the database schema; for more information, see "The Change Dictionary" (page 328). The *options* dictionary describes the aspects of the schema for which to create SQL statements; for more information, see "The Options Dictionary" (page 326).

sqlStringForNumber

```
public static String sqlStringForNumber(Number aNumber)
```

Returns the SQL string for *aNumber*.

sqlStringForString

```
public static String sqlStringForString(String aString)
```

Returns the SQL string for *aString*.

supportsDirectColumnCoercion

`public static boolean supportsDirectColumnCoercion()`

Returns `true` if the adaptor can change the type of an existing column in place, `false` otherwise.

supportsDirectColumnDeletion

`public static boolean supportsDirectColumnDeletion()`

Returns `true` if the adaptor can delete columns, `false` otherwise.

supportsDirectColumnInsertion

`public static boolean supportsDirectColumnInsertion()`

Returns `true` if the adaptor can add columns to a table, `false` otherwise.

supportsDirectColumnNullRuleModification

`public static boolean supportsDirectColumnNullRuleModification()`

Returns `true` if the adaptor can modify the null rule of an existing column in place, `false` otherwise.

supportsDirectColumnRenaming

`public static boolean supportsDirectColumnRenaming()`

Returns `true` if the adaptor can rename table columns, `false` otherwise.

supportsSchemaSynchronization

`public static boolean supportsSchemaSynchronization()`

Returns `true` if the adaptor can update the database to reflect changes in a model, `false` otherwise.

updateStatementForRow

```
public static EOSQLExpression updateStatementForRow(
    NSDictionary row,
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    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`.

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` to send a `setUseAliases(true)` message prior to invoking `super`'s version.

See Also: `setUseAliases`

useBindVariables

```
public static boolean useBindVariables()
```

Returns `true` if instances use bind variables, `false` otherwise. For more information on bind variables, see the discussion in the class description.

See Also: `setUseBindVariables`

useQuotedExternalNames

```
public static boolean useQuotedExternalNames()
```

Returns `true` if instances use quoted external names, `false` otherwise.

See Also: `setUseQuotedExternalNames`, `sqlStringForSchemaObjectName`, `externalNameQuoteCharacter`

Instance Methods

addBindVariableDictionary

`public void addBindVariableDictionary(NSMutableDictionary `*`binding`*`)`

Adds *binding* to the receiver's array of bind variable dictionaries. *binding* is generally created using the method `bindVariableDictionaryForAttribute` and is added to the receiver's bind variable dictionaries in `sqlStringForValue` when the receiver uses a bind variable for the specified attribute. See the method description for `bindVariableDictionaryForAttribute` 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`

addCreateClauseForAttribute

`public void addCreateClauseForAttribute(EOAttribute `*`attribute`*`)`

Adds the SQL string for creating *attribute* to a comma-separated list of attribute creation clauses. The list is constructed for use in a CREATE TABLE statement produced by `createTableStatementsForEntityGroup`. Use the method `listString` to access creation clauses.

EOSQLExpression's implementation creates clauses in the following form:

`COLUMN_NAME COLUMN_TYPE ALLOWS_NULL_CLAUSE`

Where

■ *COLUMN_TYPE* is the string returned from `columnTypeStringForAttribute` for *anAttribute.*

■ *ALLOWS_NULL_CLAUSE* is the string returned from `allowsNullClauseForConstraint` with `true` if *anAttribute* `allowsNull` or with `false` if *anAttribute* doesn't.

addInsertListAttribute

```
public void addInsertListAttribute(
    EOAttribute attribute,
    String 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 `appendItemToListString` 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`, `formatValueForAttribute`

addJoinClause

```
public void addJoinClause(
    String leftName,
    String rightName,
    int semantic)
```

Creates a new join clause by invoking `assembleJoinClause` 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

```
public void
    addOrderByAttributeOrdering(com.apple.yellow.eocontrol.EOSortOrdering sortOrdering)
```

Adds an attribute-direction pair ("LAST_NAME asc", for example) to the receiver's ORDER BY string. If `sortOrdering`'s selector is `EOSortOrdering.CompareCaseInsensitiveAscending` or `EOSortOrdering.CompareCaseInsensitiveAscending`, the string generated has the format "upper(attribute) direction". Use the method `orderByString` to access the ORDER BY string. `addOrderByAttributeOrdering` invokes `appendItemToListString` to add the attribute-direction pair.

See Also: `sqlStringForAttributeNamed`

addSelectListAttribute

```
public 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 `appendItemToListString` to add the attribute name.

See Also: `sqlStringForAttribute`, `formatSQLString`, `readFormat` **(EOAttribute)**

addUpdateListAttribute

```
public void addUpdateListAttribute(
    EOAttribute attribute,
    String value)
```

Adds an 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` invokes `appendItemToListString` to add the attribute-value assignment.

See Also: `formatSQLString`

aliasesByRelationshipPath

```
public 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`

allowsNullClauseForConstraint

```
public String allowsNullClauseForConstraint(boolean flag)
```

Returns according to *flag* an adaptor specific string for use in a CREATE TABLE statement. The returned string indicates whether a column allows null values. EOSQLExpression's implementation returns the empty string if *flag* is true, "NOT NULL" otherwise. A subclass should override this if its database server's semantics are different. For example, the SybaseSLQExpression returns "null" if *flag* is true, the empty string otherwise.

See Also: addCreateClauseForAttribute

appendItemToListString

```
public void appendItemToListString(
    String itemString,
    String 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, addUpdateListAttribute, and addOrderByAttributeOrdering

assembleDeleteStatementWithQualifier

```
public String assembleDeleteStatementWithQualifier(
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    String tableList,
    String 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

```
public String assembleInsertStatementWithRow(
    NSDictionary row,
    String tableList,
    String columnList,
    String valueList)
```

Invoked from `prepareInsertExpressionWithRow` to return an SQL INSERT statement of the form:

```
INSERT INTO tableList (columnList)
VALUES valueList
```

or, if `columnList` is null:

```
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.

assembleJoinClause

```
public String assembleJoinClause(
    String leftName,
    String rightName,
    int 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

```
public String assembleSelectStatementWithAttributes(
    NSArray attributes,
    boolean lock,
    com.apple.yellow.eocontrol.EOQualifier qualifier,
    NSArray fetchOrder,
```

```
   String selectString,
   String columnList,
   String tableList,
   String whereClause,
   String joinClause,
   String orderByClause,
   String lockClause)
```

Invoked from `prepareSelectExpressionWithAttributes` to return an SQL SELECT statement of the form:

```
SELECT columnList
FROM tableList lockClause
WHERE whereClause AND joinClause
ORDER BY orderByClause
```

If *lockClause* is `null`, it is omitted from the statement. Similarly, if *orderByClause* is `null`, the "ORDER BY *orderByClause*" is omitted. If either *whereClause* or *joinClause* is `null`, the "AND" and `null`-valued argument are omitted. If both are `null`, the entire WHERE clause is omitted.

attributes, *lock*, *qualifier*, and *fetchOrder* are the arguments to `prepareSelectExpressionWithAttributes` 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

```
public String assembleUpdateStatementWithRow(
   NSDictionary row,
   com.apple.yellow.eocontrol.EOQualifier qualifier,
   String tableList,
   String updateList,
   String whereClause)
```

Invoked from `prepareUpdateExpressionWithRow` to return an SQL UPDATE statement of the form:

```
UPDATE tableList
SET updateList
WHERE whereClause
```

row and *qualifier* are the arguments to `prepareUpdateExpressionWithRow` 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

```
public 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

```
public NSMutableDictionary bindVariableDictionaryForAttribute(
    EOAttribute attribute,
    Object value)
```

Implemented by subclasses to create and return the bind variable dictionary for *attribute* and *value*. The dictionary returned from this method must contain the following key-value pairs:

Key	Corresponding Value
BindVariableNameKey	Name of the bind variable for *attribute*
BindVariablePlaceHolderKey	Placeholder string used in the SQL statement
BindVariableAttributeKey	*attribute*
BindVariableValueKey	*value*

An adaptor subclass may define additional entries as required by its RDBMS.

Invoked from `sqlStringForValue` when the message `mustUseBindVariableForAttribute(attribute)` returns `true` or when the receiver's class uses bind variables and the message `shouldUseBindVariableForAttribute(attribute)` returns `true`. 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`, `useBindVariables`

columnTypeStringForAttribute

`public String columnTypeStringForAttribute(EOAttribute anAttribute)`

Returns an adaptor specific type string for `anAttribute` that's suitable for use in a CREATE TABLE statement. EOSQLExpression's implementation creates a string based on `anAttribute`'s `externalType`, `precision`, and `width` as follows:

If Condition	Generated String
precision is non-zero	externalType(precision, scale)
precision is zero and width is non-zero	externalType(scale)
precision and width are zero	externalType

A subclass should override the default implementation if its database server requires column types in a different format.

See Also: `addCreateClauseForAttribute`

entity

`public EOEntity entity()`

Returns the receiver's entity.

See Also: `EOSQLExpression` constructor

externalNameQuoteCharacter

`public String externalNameQuoteCharacter()`

Returns the string '\"' (an escaped quote character) if the receiver uses quoted external names, or the empty string ("") otherwise.

See Also: `useQuotedExternalNames`, `sqlStringForSchemaObjectName`

joinClauseString

`public String joinClauseString()`

Returns the part of the receiver's WHERE clause that specifies join conditions. Together, the `joinExpression` 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

`public 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(this)` to the qualifier for the SELECT statement. (These methods build up the `aliasesByRelationshipPath` dictionary by invoking `sqlStringForAttributePath`.)

See Also: `whereClauseString`

listString

```
public String listString()
```

Returns a comma-separated list of attributes or "attribute = value" assignments. `listString` is built up with successive invocations of `addInsertListAttribute`, `addSelectListAttribute`, or `addUpdateListAttribute` 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

```
public String 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

```
public boolean mustUseBindVariableForAttribute(EOAttribute attribute)
```

Returns `true` if the receiver must use bind variables for `attribute`, `false` otherwise. EOSQLExpression's implementation returns `false`. An SQL expression subclass that uses bind variables should override this method to return `true` if the underlying RDBMS requires that bind variables be used for attributes with `attribute`'s external type.

See Also: `shouldUseBindVariableForAttribute`, `bindVariableDictionaryForAttribute`

orderByString

```
public String 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

prepareConstraintStatementForRelationship

```
public void prepareConstraintStatementForRelationship(
    EORelationship relationship,
    NSArray sourceColumns,
    NSArray destinationColumns)
```

Sets the receiver's statement to an adaptor specific constraint for relationship. EOSQLExpression's implementation generates statements of the form:

```
ALTER TABLE TABLE_NAME ADD CONSTRAINT CONSTRAINT_NAME
    FOREIGN KEY (SOURCE_KEY_LIST)
    REFERENCES DESTINATION_TABLE_NAME (DESTINATION_KEY_LIST)
```

Where

- TABLE_NAME is the external name of the receiver's entity.

- CONSTRAINT_NAME is the external name of the receiver's entity, relationship's name, and the string "FK", concatenated with underbars between them (EMPLOYEE_MANAGER_FK, for example),

- SOURCE_KEY_LIST is a comma-separated list of the source columns in sourceColumns.

- DESTINATION_TABLE_NAME is the external name of relationship's destination entity.

- DESTINATION_KEY_LIST is a comma-separated list of the destination columns in destinationColumns

See Also: foreignKeyConstraintStatementsForRelationship

prepareDeleteExpressionForQualifier

```
public void prepareDeleteExpressionForQualifier(
    com.apple.yellow.eocontrol.EOQualifier qualifier)
```

Generates a DELETE statement by performing the following steps:

1. Sends an `sqlStringForSQLExpression(this)` message to *qualifier* to generate the receiver's `whereClauseString`.

2. Invokes `tableListWithRootEntity` to get the table name for the FROM clause.

3. Invokes `assembleDeleteStatementWithQualifier`.

See Also: `deleteStatementWithQualifier`

prepareInsertExpressionWithRow

```
public void prepareInsertExpressionWithRow(NSDictionary row)
```

Generates an INSERT statement by performing the following steps:

1. Invokes `addInsertListAttribute` 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`.

See Also: `insertStatementForRow`

prepareSelectExpressionWithAttributes

```
public void prepareSelectExpressionWithAttributes(
    NSArray attributes,
    boolean flag,
    com.apple.yellow.eocontrol.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(this)` 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 `true`, invokes `lockClause` to get the SQL string to lock selected rows.

7. Invokes `assembleSelectStatementWithAttributes`.

See Also: `selectStatementForAttributes`

prepareUpdateExpressionWithRow

```
public void prepareUpdateExpressionWithRow(
    NSDictionary row,
    com.apple.yellow.eocontrol.EOQualifier qualifier)
```

Generates an UPDATE statement by performing the following steps:

1. Invokes `addUpdateListAttribute` for each entry in *`row`* to prepare the comma-separated list of "attribute = value" assignments.

2. Sends an `sqlStringForSQLExpression(this)` message to *`qualifier`* to generate the receiver's `whereClauseString`.

3. Invokes `tableListWithRootEntity` to get the table name for the FROM clause.

4. Invokes `assembleUpdateStatementWithRow`.

See Also: `updateStatementForRow`

setStatement

```
public void setStatement(String 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

public void setUseAliases(boolean *flag*)

Tells the receiver whether or not to use table aliases.

See Also: useAliases

shouldUseBindVariableForAttribute

public boolean shouldUseBindVariableForAttribute(EOAttribute *attribute*)

Returns true if the receiver can provide a bind variable dictionary for *attribute*, false otherwise. Bind variables aren't used for values associated with this attribute when the static method useBindVariables returns false. EOSQLExpression's implementation returns false. An SQL expression subclass should override this method to return true if the receiver should use bind variables for attributes with *attribute*'s external type. It should also return true for any attribute for which the receiver must use bind variables.

See Also: mustUseBindVariableForAttribute

sqlStringForAttribute

public String 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

public String sqlStringForAttributeNamed(String *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

`public String 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 `aliasesByRelationshipPath` dictionary.

See Also: `sqlStringForAttribute`, `aliasesByRelationshipPath`

sqlStringForCaseInsensitiveLike

```
public String sqlStringForCaseInsensitiveLike(
    String valueString,
    String keyString)
```

Overridden by subclasses to return a case insensitive comparison of *valueString* and *keyString*. For example, a subclass implementation might return the string "UPPER(*keyString*) LIKE UPPER(*valueString*)".

sqlStringForConjoinedQualifiers

```
public String 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 `this` 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

```
public String 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 `this` 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

```
public String sqlStringForKeyComparisonQualifier(
    com.apple.yellow.eocontrol.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` with *qualifier*'s selector and `null` 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` with the corresponding attributes' "read" formats.

sqlStringForKeyValueQualifier

```
public String sqlStringForKeyValueQualifier(
    com.apple.yellow.eocontrol.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` with *qualifier*'s selector and value. Generates an SQL string for *qualifier*'s key by invoking `sqlStringForAttributeNamed` to get an SQL string and `formatSQLString` with the corresponding attribute's "read" format. Similarly, generates an SQL string for qualifier's value by invoking `sqlStringForValue` to get an SQL string and `formatValueForAttribute` to format it. (First invokes `sqlPatternFromShellPattern` for the value if *qualifier*'s selector is `EOQualifier.QualifierOperatorLike`.)

sqlStringForNegatedQualifier

```
public String
    sqlStringForNegatedQualifier(com.apple.yellow.eocontrol.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 `this` as the argument. If the SQL string for *qualifier* contains only white space, this method returns `null`.

sqlStringForQualifier

```
public String sqlStringForQualifier(com.apple.yellow.eocontrol.EOQualifier aQualifier)
```

Returns a SQL statement for *aQualifier* suitable for inclusion in a WHERE clause. Invoked from an EOSQLExpression while it's preparing a SELECT, UPDATE, or DELETE statement.

See Also: `whereClauseString`

sqlStringForSchemaObjectName

```
public String sqlStringForSchemaObjectName(String 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

```
public String sqlStringForSelector(
    NSSelector selector,
    Object value)
```

Returns an SQL operator for *selector* and *value*. The possible values for *selector* are defined as constants (in EOControl). The following table summarizes EOSQLExpression's default mapping:

Selector (Constant)	SQL Operator
EOQualifier.QualifierOperatorIsEqual	"is" if value is an EONullValue, "=" otherwise
EOQualifier.QualifierOperatorNotEqual	"is not" if *value* is an EONullValue, "<>" otherwise
EOQualifier.QualifierOperatorLessThan	"<"
EOQualifier.QualifierOperatorGreaterThan	">"
EOQualifier.QualifierOperatorLessThanOrEqualTo	"<="
EOQualifier.QualifierOperatorGreaterThanOrEqualTo	">="
EOQualifier.QualifierOperatorLike	"like"

Throws an exception if *selector* is an unknown operator.

See Also: sqlStringForKeyComparisonQualifier, sqlStringForKeyValueQualifier

sqlStringForValue

```
public String sqlStringForValue(
    Object value,
    String 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` 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 `BindVariablePlaceHolderKey`. Otherwise, this method invokes `formatValueForAttribute` and returns the formatted string for `value`.

See Also: `mustUseBindVariableForAttribute`, `shouldUseBindVariableForAttribute`, `useBindVariables`, `bindVariableDictionaries`, `addBindVariableDictionary`

statement

```
public String statement()
```

Returns the complete SQL statement for the receiver. An SQL statement can be assigned to an EOSQLExpression object directly using the static method `expressionForString` or using the instance method `setStatement`. Generally, however, an EOSQLExpression's statement is built up using one of the following methods:

■   `prepareSelectExpressionWithAttributes`

■   `prepareInsertExpressionWithRow`

■   `prepareUpdateExpressionWithRow`

■   `prepareDeleteExpressionForQualifier`

tableListWithRootEntity

```
public String 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

`public boolean useAliases()`

Returns `true` if the receiver generates statements with table aliases, `false` 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

`public String 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` messages.

whereClauseString

```
public String 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`

■ `insertStatementForRow`

■ `updateStatementForRow`

■ `deleteStatementWithQualifier`

Unless you're implementing an EOSQLExpression subclass, these and the static 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`

■ `prepareInsertExpressionWithRow`

■ `prepareUpdateExpressionWithRow`

■ `prepareDeleteExpressionForQualifier`

The `prepare...` methods, in turn, invoke a corresponding `assemble...` method, first generating values for the `assemble...` method's arguments. The `assemble...` methods:

- `assembleSelectStatementWithAttributes`

- `assembleInsertStatementWithRow`

- `assembleUpdateStatementWithRow`

- `assembleDeleteStatementWithQualifier`

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`. If necessary, it also formats the SQL strings according to an EOAttribute's "read" or "write" format with the static method `formatSQLString`.

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 methods:

- `bindVariableDictionaryForAttribute`

- `mustUseBindVariableForAttribute`

■ shouldUseBindVariableForAttribute

If your adaptor doesn't need to use bind variables, the default implementations of the bind variable methods are sufficient.

Schema Generation

EOSQLExpression provides API to generate SQL that can be used to create a database. EOModeler uses these methods to generate scripts that you can execute from within EOModeler to create a database or that you can copy and paste into an interactive SQL shell for your database. It's rare that you'd ever invoke this API programmatically. You won't need to know anything about it unless you're implementing it for a custom adaptor. If you are writing an adaptor, you must ensure that EOSQLExpression's implementation of the schema generation API is sufficient to support EOModeler's schema generation.

The entry point into the schema generation code is the method schemaCreationScriptForEntities, which returns a script of SQL statements suitable to create the schema for the EOEntity objects in the entities argument. The options dictionary specifies the aspects of the schema for which to create SQL statements. EOSQLExpression's implementation invokes schemaCreationStatementsForEntities and then uses appendExpression to generate the script.

The Options Dictionary

The options dictionary specifies the aspects of the schema for which to create SQL statements. It's contents are described in the following table:

Dictionary Key	Acceptable Values (Strings)	Default
CreateTablesKey	"YES" or "NO"	YES
DropTablesKey	"YES" or "NO"	YES
CreatePrimaryKeySupportKey	"YES" or "NO"	YES
DropPrimaryKeySupportKey	"YES" or "NO"	YES

Dictionary Key	Acceptable Values (Strings)	Default
PrimaryKeyConstraintsKey	"YES" or "NO"	YES
ForeignKeyConstraintsKey	"YES" or "NO"	NO
CreateDatabaseKey	"YES" or "NO"	NO
DropDatabaseKey	"YES" or "NO"	NO

Note: If you specify entries for CreateDatabaseKey or DropDatabaseKey, the SQL for those statements must be executed by an administrative user.

Schema Synchronization

EOSQLExpression provides API to generate SQL that can be used to synchronize a database with a corresponding model. As with the schema generation API, EOModeler makes use of the schema synchronization API, and it's rare that you'd ever invoke it programmatically. You won't need to know anything about the API yourself unless you're implementing the API for a custom adaptor. This section describes what the API is an how it works in the event that you need to implement it.

The entry point into the schema synchronization code is the static method statementsToUpdateObjectStoreForModel. The change dictionary argument identifies the changes to make to the database schema to synchronize it with the specified model, and the options dictionary identifies aspects of the schema for which to create SQL statements. For more information on the changes dictionary, see "The Change Dictionary" (page 328). For more information on the options dictionary, see "The Options Dictionary" (page 326). Using the change dictionary, statementsToUpdateObjectStoreForModel identifies the database tables that need to be updated, finds the entities that correspond to those tables, and invokes the static method statementsToUpdateObjectStoreForEntityGroup for each table. This method determines which, if any, of the synchronization operations can be performed in place (without creating a new table, moving the data, and dropping the old table). Depending on the features supported by the adaptor, the method invokes an operation specific method to make the changes in place, or it invokes statementsToCopyTableNamed to create a new table with an updated definition, copy

data from the old table to the new table, and drop the old table. An adaptor advertises what kind of synchronization operations it supports with supports*Operation* methods that return `true` if they support a feature, or `false` otherwise.

The Change Dictionary

The change dictionary argument in the schema synchronization API contains information about tables to insert, tables to delete, tables to update, and information about how to update them.

The change dictionary can have any of the three keys `EOEditingContext.InsertedKey`, `EOEditingContext.DeletedKey`, and `EOEditingContext.UpdatedKey` (defined in EOControl). The values for the `InsertedKey` and `DeletedKey` are arrays of table names to insert or delete, respectively. The value for the `UpdatedKey` is a subdictionary.

The subdictionary for the `UpdatedKey` has keys that are the names of the tables to update. The corresponding values are additional dictionaries that describe how to modify the tables. The keys of these dictionaries can be any of:

`ExternalNameKey`
> The name of the table before the change.

`RelationshipsKey`
> A dictionary of relationships which have been modified since the last time the model and schema were sychronized. The keys of the dictionary are relationship names, and the values are subdictionaries with one entry. A subdictionary's key is `NameKey` (defined as "name"), and it's value is the old name of the corresponding relationship.
>
> For example, suppose the Movie entity has a relationship named "movieRoles" to the MovieRole entity, and suppose that you change that relationship's name to "roles". The `RelationshipsKey`'s dictionary for that change looks like this has an entry with the key "Movie.roles". (The relationship name must be prefixed with the name of the relationship's source entity because more than one entity can refer to the updated table.) The corresponding value is a subdictionary whose key is `NameKey` and whose value is the string "movieRoles"—the old name of the changed relationship.
>
> The old name of the relationship is needed because relationship names are used to define foreign key constraints. In order to drop the old constraints, the schema synchronization methods need to have the old relationship names. In the movieRoles example, the schema synchronization methods must drop the foreign key constraint based on the old relationship name:
>
> ```
> alter table MOVIE drop constraint MOVIE_movieRoles_FK cascade
> ```

After any old constraints are dropped, new ones are created based on the new relationship names.

InsertedKey

An array of column names to insert into the table

DeletedKey

An array of column names to delete from the table

UpdatedKey

A dictionary containing information on columns to update and how to update them. The keys are the names of the columns to update, and the values are subdictionaries on how to update the corresponding columns. A subdictionary key identifies an updated property of the column: name, type, and so on. The corresponding value is the old property value. The keys are:

- AllowsNullKey

- ColumnNameKey

- ExternalNameKey

- ExternalTypeKey

- NameKey

- PrecisionKey

- RelationshipsKey

- ScaleKey

- WidthKey

EOSQLQualifier

Inherits from:	EOQualifier : NSObject
Package:	com.apple.yellow.eoaccess

Class Description

EOSQLQualifier is a subclass of EOQualifier that contains unstructured text that can be transformed into an SQL expression. EOSQLQualifier provides 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.

To create an EOSQLQualifier, provide to the constructor a root entity for the qualifier and a format string like that used with the EOQualifier creation method `qualifierWithQualifierFormat`. (You can't use the `qualifierWithQualifierFormat` method because it doesn't take an entity as an argument and an SQL qualifier must be rooted to an entity.)

Constructors

EOSQLQualifier

```
public EOSQLQualifier(
    EOEntity entity,
    String qualifierFormat)
```

Creates and returns a newly allocated EOSQLQualifier rooted in *entity* and built from a format string *qualifierFormat*. *qualifierFormat* is a `printf()`-style format string like that used with EOQualifier's `qualifierWithQualifierFormat` method. Returns a new EOSQLQualifier if it can parse *qualifierFormat* successfully, `null` otherwise.

Static Methods

qualifierMigratedFromEntity

```
public static com.apple.yellow.eocontrol.EOQualifier qualifierMigratedFromEntity(
    com.apple.yellow.eocontrol.EOQualifier aQualifier,
    EOEntity entity,
    String relationshipPath)
```

Creates a copy of *aQualifier*, 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 named "department" to a Department entity. You could create 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, send a `qualifierMigratedFromEntity` message with the Employee entity as the entity and "department" as the relationship path.

qualifierWithQualifierFormat

```
public static com.apple.yellow.eocontrol.EOQualifier
    qualifierWithQualifierFormat(String format)
```

Throws an exception. An EOSQLQualifier must be created with an entity, and this method does not provide one. Use a constructor and provide an entity to create an EOSQLQualifier.

Instance Methods

qualifierWithBindings

```
public com.apple.yellow.eocontrol.EOQualifier qualifierWithBindings(
    NSDictionary aDictionary,
    boolean flag)
```

Returns a new qualifier created by substituting all EOQualifierVariables with the values contained in *aDictionary*. If *flag* is true, then the new qualifier requires all its variables. If *flag* is false, then the new qualifier doesn't require all its variables; and if any variable is not found in *aDictionary*, the node containing that variable is simply pruned from the qualifier tree. Note that `null` and EONull are not the same in this context. If a value in *aDictionary* is `null`, this method prunes it from the qualifier tree. If a value is EONull, this method assumes that you are looking for an object with a `null` value.

validateKeysWithRootClassDescription

```
public Throwable validateKeysWithRootClassDescription(
    com.apple.yellow.eocontrol.EOClassDescription classDesc)
```

Validates that a qualifier contains keys and key paths that belong to or originate from classDesc. This method returns an `NSInternalInconsistencyException` if an unknown key is found, otherwise it returns null to indicate that the keys contained by the qualifier are valid.

EOStoredProcedure

Inherits from:	NSObject
Implements:	EOPropertyListEncoding
Package:	com.apple.yellow.eoaccess

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` 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.

Interfaces Implemented

EOPropertyListEncoding

    awakeWithPropertyList

    encodeIntoPropertyList

Method Types

Constructors

    EOStoredProcedure

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

Constructors

EOStoredProcedure

```
public EOEOStoredProcedure(String name)
```

Creates and returns a new EOStoredProcedure named *name*.

```
public EOStoredProcedure(
   NSDictionary propertyList,
   Object owner)
```

Creates and returns a new EOStoredProcedure initialized from *propertyList*—a dictionary containing only property list data types (that is, String, NSDictionary, NSArray, and NSData). This constructor is used by EOModeler when it reads in an EOModel object from a file, for example. The *owner* argument should be the EOStoredProcedure's EOModel. EOStoredProcedures created from a property list must receive an awakeWithPropertyList message immediately after creation before they are fully functional, but the awake... message should be deferred until the all of the other objects in the model have also been created.

See Also: encodeIntoPropertyList (**EOPropertyListEncoding**) setName, name

Instance Methods

arguments

```
public NSArray arguments()
```

Returns the EOAttribute objects that describe the stored procedure's arguments or `null` if the stored procedure has no arguments.

beautifyName

```
public 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". This method is used in reverse-engineering a model.

See Also: `setArguments`, `beautifyNames` **(EOModel)**

externalName

```
public String externalName()
```

Returns the name of the stored procedure as it is defined in the database, or `null` if the receiver doesn't have an external name.

See Also: `setExternalName`

model

```
public EOModel model()
```

Returns the model to which the receiver belongs.

See Also: `addStoredProcedure` **(EOModel)**

name

```
public String name()
```

Returns the name of the receiver.

See Also: setName, EOStoredProcedure **constructor**

setArguments

```
public 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

```
public void setExternalName(String 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

```
public void setName(String name)
```

Sets the name of the receiver.

See Also: name, EOStoredProcedure **constructor**

setUserInfo

```
public 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, String, NSDictionary, NSArray, and NSData).

See Also: userInfo

userInfo

```
public NSDictionary userInfo()
```

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

See Also: setUserInfo

EOUtilities

Inherits from: Object

Package: com.apple.yellow.webobjects

Class Description

EOUtilities is a collection of convenience methods intended to make common operations with EOF easier. EOUtilities is an EOAccess class that consists entirely of static methods—you never instantiate an EOUtilities object.

Each method requires an editing context into which the objects should be fetched; this editing context is passed as the first argument to each method in EOUtilities.

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

Creating new objects

createAndInsertInstance

Fetching multiple objects

objectsForEntityNamed

objectsWithQualifierFormat

objectsMatchingKeyAndValue

objectsMatchingValues

objectsOfClass

objectsWithFetchSpecificationAndBindings

Fetching single objects

objectWithQualifierFormat

objectMatchingKeyAndValue

objectMatchingValues

objectWithFetchSpecificationAndBindings

objectWithPrimaryKey

objectWithPrimaryKeyValue

Fetching raw rows

executeStoredProcedureNamed

objectFromRawRow

rawRowsWithQualifierFormat

rawRowsMatchingKeyAndValue

rawRowsMatchingValues

rawRowsForSQL

rawRowsForStoredProcedureNamed

Accessing the EOF stack

connectWithModelNamed

databaseContextForModelNamed

Accessing object data

destinationKeyForSourceObject

localInstanceOfObject

localInstancesOfObjects

primaryKeyForObject

Accessing model information

entityForClass

entityForObject

entityNamed

modelGroup

Instance Methods

connectWithModelNamed

```
public static void connectWithModelNamed(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String modelName,
    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. Throws an exception if the connection failed.

createAndInsertInstance

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject createAndInsertInstance(
   com.apple.yellow.eocontrol.EOEditingContext editingContext,
   String entityName)
```

Creates a new enterprise object for the specified entity, inserts it into `editingContext`, and returns the new object.

databaseContextForModelNamed

```
public static EODatabaseContext databaseContextForModelNamed(
   com.apple.yellow.eocontrol.EOEditingContext editingContext,
   String modelName)
```

Returns the database context used to service the specified model.

destinationKeyForSourceObject

```
public static NSDictionary destinationKeyForSourceObject(
   com.apple.yellow.eocontrol.EOEditingContext editingContext,
   com.apple.yellow.eocontrol.EOEnterpriseObject object,
   String relationshipName)
```

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` and `Employee.deptID`, invoking this method on a Department object with ID equal to 5 returns a dictionary with a value of 5 for the `deptID` key.

See Also: `primaryKeyForObject`

entityForClass

```
public static EOEntity entityForClass(
   com.apple.yellow.eocontrol.EOEditingContext editingContext,
   Class classObject)
```

Returns the entity associated with the specified class. Throws 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

```
public static EOEntity entityForObject(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    com.apple.yellow.eocontrol.EOEnterpriseObject object)
```

Returns the entity associated with the provided enterprise object. Throws an exception if the specified entity can't be found.

See Also: entityForClass, entityNamed

entityNamed

```
public static EOEntity entityNamed(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName)
```

Returns the entity with the specified name. Throws an exception if the specified entity can't be found.

See Also: entityForClass, entityForObject

executeStoredProcedureNamed

```
public static NSDictionary executeStoredProcedureNamed(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String storedProcedureName,
    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: rawRowsForStoredProcedureNamed

localInstanceOfObject

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject localInstanceOfObject(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    com.apple.yellow.eocontrol.EOEnterpriseObject object)
```

Translates the specified enterprise object from another editing context to the specified one.

See Also: `localInstancesOfObjects`

localInstancesOfObjects

```
public static NSArray localInstancesOfObjects(
    com.apple.yellow.eocontrol.EOEditingContext,
    editingContext,
    NSArray objects)
```

Translates the specified enterprise objects from another editing context to the specified one.

See Also: `localInstanceOfObject`

modelGroup

```
public static EOModelGroup
    modelGroup(com.apple.yellow.eocontrol.EOEditingContext editingContext)
```

Returns the model group associated with the editing context's root object store, an EOObjectStoreCoordinator.

objectFromRawRow

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectFromRawRow(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    NSDictionary row)
```

Fetches and returns the object corresponding to the specified raw row (using EOEditingContext's `faultForRawRow`). This method can only be used on raw rows that include the row's primary key.

objectMatchingKeyAndValue

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectMatchingKeyAndValue(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String key,
    Object value)
```

Creates an EOKeyValueQualifier with the specified key and value and returns matching enterprise objects. Throws an exception unless exactly one object is retrieved.

See Also: `objectMatchingValues`, `objectsMatchingKeyAndValue`

objectMatchingValues

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectMatchingValues(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    NSDictionary values)
```

Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching enterprise objects. Throws an exception unless exactly one object is retrieved.

See Also: `objectMatchingKeyAndValue`, `objectsMatchingValues`

objectsForEntityNamed

```
public static NSArray objectsForEntityNamed(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName)
```

Fetches and returns the enterprise objects associated with the specified entity.

See Also: `objectsWithQualifierFormat`, `objectsMatchingKeyAndValue`, `objectsMatchingValues`

objectsMatchingKeyAndValue

```
public static NSArray objectsMatchingKeyAndValue(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String key,
    Object value)
```

Creates an EOKeyValueQualifier with the specified key and value and returns matching enterprise objects.

See Also: objectMatchingKeyAndValue, objectsForEntityNamed, objectsMatchingValues

objectsMatchingValues

```
public static NSArray objectsMatchingValues(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    NSDictionary values)
```

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, objectsForEntityNamed, objectsMatchingKeyAndValue

objectsOfClass

```
public static NSArray objectsOfClass(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    Class classObject)
```

Fetches and returns the enterprise objects associated with the specified class. Throws an exception if more than one entity for the class exists.

See Also: entityForClass

objectsWithFetchSpecificationAndBindings

```
public static NSArray objectsWithFetchSpecificationAndBindings(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String fetchSpecName,
    NSDictionary bindings)
```

Fetches and returns the enterprise objects retrieved with the specified fetch specification and bindings.

See Also: objectWithFetchSpecificationAndBindings

objectsWithQualifierFormat

```
public static NSArray objectsWithQualifierFormat(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String format,
    NSArray arguments)
```

Creates a qualifier with the provided format string and arguments, and returns matching enterprise objects.

See Also: objectWithQualifierFormat, objectsForEntityNamed

objectWithFetchSpecificationAndBindings

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject
    objectWithFetchSpecificationAndBindings(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String fetchSpecName,
    NSDictionary bindings)
```

Fetches and returns the enterprise objects retrieved with the specified fetch specification and bindings. Throws an exception unless exactly one object is retrieved.

See Also: objectsWithFetchSpecificationAndBindings

objectWithPrimaryKey

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectWithPrimaryKey(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    NSDictionary keyDictionary)
```

Fetches and returns the enterprise object identified by the specified primary key dictionary. Throws an exception unless exactly one object is retrieved.

See Also: objectMatchingKeyAndValue, objectWithPrimaryKeyValue, primaryKeyForObject

objectWithPrimaryKeyValue

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectWithPrimaryKeyValue(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    Object value)
```

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. Throws an exception unless exactly one object is retrieved.

See Also: objectsMatchingValues, objectWithPrimaryKey

objectWithQualifierFormat

```
public static com.apple.yellow.eocontrol.EOEnterpriseObject objectWithQualifierFormat(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String format,
    NSArray arguments)
```

Creates a qualifier with the provided format string and arguments, and returns matching enterprise objects. Throws an exception unless exactly one object is retrieved.

See Also: objectsWithQualifierFormat, rawRowsWithQualifierFormat

primaryKeyForObject

```
public static NSDictionary primaryKeyForObject(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    com.apple.yellow.eocontrol.EOEnterpriseObject object)
```

Returns the primary key dictionary for the specified enterprise object.

See Also: objectWithPrimaryKey, objectWithPrimaryKeyValue

rawRowsForSQL

```
public static NSArray rawRowsForSQL(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String modelName,
    String sqlString)
```

Evaluates the specified SQL and returns the resulting raw rows.

See Also: rawRowsWithQualifierFormat, rawRowsForStoredProcedureNamed

rawRowsForStoredProcedureNamed

```
public static NSArray rawRowsForStoredProcedureNamed(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String storedProcedureName,
    NSDictionary arguments)
```

Executes the specified stored procedure with the provided arguments and returns the resulting raw rows.

See Also: rawRowsForSQL

rawRowsMatchingKeyAndValue

```
public static NSArray rawRowsMatchingKeyAndValue(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String key,
    Object value)
```

Creates an EOKeyValueQualifier with the specified key and value and returns matching raw rows.

See Also: objectMatchingKeyAndValue, objectsMatchingKeyAndValue, rawRowsMatchingValues

rawRowsMatchingValues

```
public static NSArray rawRowsMatchingValues(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    NSDictionary values)
```

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, objectsMatchingValues, rawRowsMatchingKeyAndValue

rawRowsWithQualifierFormat

```
public static NSArray rawRowsWithQualifierFormat(
    com.apple.yellow.eocontrol.EOEditingContext editingContext,
    String entityName,
    String format,
    NSArray arguments)
```

Creates a qualifier for the specified entity and with the specified qualifier format and returns matching raw row dictionaries.

See Also: objectsWithQualifierFormat, rawRowsForSQL

EOAdaptorChannel.Delegate

(informal interface)

Package: com.apple.yellow.eoaccess

Interface 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

`public abstract void adaptorChannelDidChangeResultSet(Object channel)`

Invoked from `fetchRow` when a select operation resulted in multiple result sets. This method tells the delegate that the next invocation of `fetchRow` will fetch from the next result set. This method is invoked when `fetchRow` returns `null` and there are still result sets left to fetch. The delegate can invoke `setAttributesToFetch` to prepare for fetching the new rows.

adaptorChannelDidEvaluateExpression

```
public abstract void adaptorChannelDidEvaluateExpression(
    Object channel,
    EOSQLExpression expression)
```

Invoked from `evaluateExpression` to tell the delegate that a query language expression has been evaluated by the database server.

adaptorChannelDidExecuteStoredProcedure

```
public abstract void adaptorChannelDidExecuteStoredProcedure(
    Object channel,
    EOStoredProcedure procedure,
    NSDictionary values)
```

Invoked from `executeStoredProcedure` after *procedure* is executed successfully.

adaptorChannelDidFetchRow

```
public abstract void adaptorChannelDidFetchRow(
    Object channel,
    NSMutableDictionary row)
```

Invoked from `fetchRow` after a row is fetched successfully. This method is not invoked if an exception occurs during the fetch or if the same returns `null` because there are no more rows in the current result set. The delegate may modify *row*, which will be returned from `fetchRow`.

adaptorChannelDidFinishFetching

```
public abstract void adaptorChannelDidFinishFetching(Object channel)
```

Invoked from `fetchRow` to tell the delegate that fetching is finished for the current select operation. This method is invoked when a fetch ends in `fetchRow` because there are no more result sets.

adaptorChannelDidPerformOperations

```
public abstract Throwable adaptorChannelDidPerformOperations(
    Object channel,
    NSArray operations,
    Throwable exception)
```

Invoked from `performAdaptorOperations`. `exception` is `null` 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 `null` to prevent the adaptor channel from raising an exception.

adaptorChannelDidSelectAttributes

```
public abstract void adaptorChannelDidSelectAttributes(
    Object channel,
    NSArray attributes,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    boolean flag,
    EOEntity entity)
```

Invoked from `selectAttributes` to tell the delegate that rows have been selected in the database server.

adaptorChannelShouldConstructStoredProcedureReturnValues

```
public abstract NSDictionary
    adaptorChannelShouldConstructStoredProcedureReturnValues(Object 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 `null`, that value will be returned immediately from `returnValuesForLastStoredProcedureInvocation`.

adaptorChannelShouldEvaluateExpression

```
public abstract boolean adaptorChannelShouldEvaluateExpression(
    Object channel,
    EOSQLExpression expression)
```

Invoked from `evaluateExpression` to tell the delegate that `channel` is sending an expression to the database server. The delegate returns `true` to permit the adaptor channel to send `expression` to the server. If the delegate returns `false`, the adaptor channel does not send the expression and returns immediately. When the delegate returns `false`, 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 `false`.

adaptorChannelShouldExecuteStoredProcedure

```
public abstract NSDictionary adaptorChannelShouldExecuteStoredProcedure(
    Object channel,
    EOStoredProcedure procedure,
    NSDictionary values)
```

Invoked from `executeStoredProcedure` to tell the delegate that `channel` is executing a stored procedure. If the delegate returns a value other than `null`, that value is used as the arguments to the stored procedure instead of `values`.

adaptorChannelShouldReturnValuesForStoredProcedure

```
public abstract NSDictionary adaptorChannelShouldReturnValuesForStoredProcedure(
    Object channel,
    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 `null`, that value is returned from `returnValuesForLastStoredProcedureInvocation` instead of `returnValues`.

adaptorChannelShouldSelectAttributes

```
public abstract boolean adaptorChannelShouldSelectAttributes(
    Object channel,
    NSArray attributes,
```

```
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    boolean flag,
    EOEntity entity)
```

Invoked from `selectAttributes` 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` itself with a new fetch specification, and return `false` (indicating that the adaptor channel should not perform the select itself).

adaptorChannelWillFetchRow

```
public abstract void adaptorChannelWillFetchRow(Object channel)
```

Invoked from `fetchRow` 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.

adaptorChannelWillPerformOperations

```
public abstract NSArray adaptorChannelWillPerformOperations(
    Object channel,
    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 `null`, the adaptor channel does not perform the operations and returns from the method immediately.

EOAdaptorContext.Delegate

(informal interface)

Package: com.apple.yellow.eoaccess

Interface 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

```
public abstract void adaptorContextDidBegin(Object context)
```

Invoked from `beginTransaction` to tell the delegate that a transaction has begun.

adaptorContextDidCommit

`public abstract void adaptorContextDidCommit(Object` *`context`*`)`

Invoked from `commitTransaction` to tell the delegate that a transaction has been committed.

adaptorContextDidRollback

`public abstract void adaptorContextDidRollback(Object` *`context`*`)`

Invoked from `rollbackTransaction` to tell the delegate that a transaction has been rolled back.

adaptorContextShouldBegin

`public abstract boolean adaptorContextShouldBegin(Object` *`context`*`)`

Invoked from `beginTransaction` to tell the delegate that *context* is beginning a transaction. If this method returns `false`, the adaptor context does not begin a transaction. Return `true` to allow the adaptor context to begin a transaction.

adaptorContextShouldCommit

`public abstract boolean adaptorContextShouldCommit(Object` *`context`*`)`

Invoked from `commitTransaction` to tell the delegate that *context* is committing a transaction. If this method returns `false`, the adaptor context does not commit the transaction. Return `true` to allow the adaptor context to commit.

Note that if you implement this delegate method to return `false`, 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

`public abstract boolean adaptorContextShouldConnect(Object` *`context`*`)`

Invoked before the adaptor attempts to connect. The delegate can return `false` if it wants to override the connect, `true` if it wants the adaptor to attempt to connect in the usual way. The delegate should throw an exception if it fails to connect.

adaptorContextShouldRollback

`public abstract boolean adaptorContextShouldRollback(Object context)`

Invoked from `rollbackTransaction` to tell the delegate that *context* is rolling back a transaction. If this method returns `false`, the adaptor context does not roll back the transaction. Return `true` to allow the adaptor context to roll back.

EOAdaptor.Delegate

(informal interface)

Package: com.apple.yellow.eoaccess

Interface Description

The delegate for EOAdaptor can implement the method `adaptorFetchedValueForValue` to perform a database-specific transformations on a value.

Instance Methods

adaptorFetchedValueForValue

```
public abstract Object adaptorFetchedValueForValue(
    EOAdaptor adaptor,
    Object value,
    EOAttribute attribute)
```

Invoked from `fetchedValueForValue` 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` invokes one of the type-specific `fetchedValue...` methods depending on the type of *value*. If you implement this delegate method, `fetchedValueForValue` does not invoke the other `fetchedValue...` methods. It simply invokes your delegate method and returns the value returned from it. Therefore, an implementation of `adaptorFetchedValueForValue` must handle values of all types.

reconnectionDictionaryForAdaptor

`public abstract NSDictionary reconnectionDictionaryForAdaptor(EOAdaptor` *anEOAdaptor*`)`

Invoked from `handleDroppedConnection` to provides a new connection dictionary for reconnection attempts. If the adaptor's database connection is dropped (and the adaptor supports database reconnection), the adaptor attempts to recover by reconnecting. By default, the adaptor attempts to connect using its original connection dictionary. If you want it to connect to a different database, implement this method to return a connection dictionary for the secondary database. (Note that the secondary database should have the same data as the original.) If the delegate method is not implemented, the adaptor uses its existing connection dictionary to reconnect to the server.

See Also: `isDroppedConnectionException`

EOSQLExpression.ColumnTypes

Package: com.apple.yellow.eoaccess

Interface Description

This interface describes the API for interacting with objects passed as arguments to the following EOSQLExpression methods:

```
isColumnTypeEquivalentToColumnType
phraseCastingColumnNamed
statementsToConvertColumnType
```

You only need to know about this interface if you are implementing schema synchronization API for a custom adaptor. In that case, you don't have to implement a class that implements this interface; EOSQLExpression's implementation of the schema synchronization API uses a private class that implements it. You only need to know about the interface because your method implementations of the above methods needs to compare two objects that implement the protocol.

Instance Methods

name

```
public abstract String name()
```

Returns the receiver's name.

precision

```
public abstract int precision()
```

Returns the receiver's precision.

scale

```
public abstract int scale()
```

Returns the receiver's scale.

width

```
public abstract int width()
```

Returns the receiver's width.

EODatabaseContext.Delegate

(informal interface)

Package: com.apple.yellow.eoaccess

Interface 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:

    databaseContextDidSelectObjects

    databaseContextShouldSelectObjects

    databaseContextShouldUpdateCurrentSnapshot

    databaseContextShouldUsePessimisticLock

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 (`databaseContextNewPrimaryKey`), how and if objects are locked (`databaseContextShouldLockObjectWithGlobalID`), what fetch specification is used to fetch objects (`databaseContextShouldSelectObjects`), how batch faulting is performed (`databaseContextShouldFetchArrayFault` and `databaseContextShouldFetchObjectFault`), and so on. For more information, see the individual delegate method descriptions.

Instance Methods

databaseContextDidFetchObjects

```
public abstract void databaseContextDidFetchObjects(
    EODatabaseContext aDatabaseContext,
    NSArray objects,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext)
```

Invoked from `objectsWithFetchSpecification` after *aDatabaseContext* fetches *objects* using the criteria defined in *fetchSpecification* on behalf of *anEditingContext*.

See Also: databaseContextShouldFetchObjectFault

databaseContextDidSelectObjects

```
public abstract void databaseContextDidSelectObjects(
    EODatabaseContext aDatabaseContext,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    EODatabaseChannel channel)
```

Invoked from the EODatabaseChannel method `selectObjectsWithFetchSpecification` to tell the delegate that *channel* selected the objects on behalf of *aDatabaseContext* as specified by *fetchSpecification*.

See Also: databaseContextShouldSelectObjects

databaseContextFailedToFetchObject

```
public abstract boolean databaseContextFailedToFetchObject(
    EODatabaseContext aDatabaseContext,
    Object object,
    com.apple.yellow.eocontrol.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 `true`, *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

`false` or if the delegate method is not implemented, *aDatabaseContext* tracks the globalID of the offending object. If the tracked globalID is in the list of updated objects when `prepareForSaveWithCoordinator` is invoked, *aDatabaseContext* throws an exception.

To get a list of the objects that failed to fetch, see the method `missingObjectGlobalIDs`.

databaseContextNewPrimaryKey

```
public abstract NSDictionary databaseContextNewPrimaryKey(
    EODatabaseContext aDatabaseContext,
    Object object,
    EOEntity anEntity)
```

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 `null`, 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.

databaseContextShouldFetchArrayFault

```
public abstract boolean databaseContextShouldFetchArrayFault(
    EODatabaseContext databaseContext,
    Object 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`) and return `false`, or return `true` 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: `databaseContextShouldFetchObjectFault`

databaseContextShouldFetchObjectFault

```
public abstract boolean databaseContextShouldFetchObjectFault(
    EODatabaseContext databaseContext,
    Object 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`) and return `false`, or return `true` 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: databaseContextShouldFetchArrayFault

databaseContextShouldFetchObjects

```
public abstract NSArray databaseContextShouldFetchObjects(
    EODatabaseContext aDatabaseContext,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    com.apple.yellow.eocontrol.EOEditingContext anEditingContext)
```

Invoked from `objectsWithFetchSpecification` 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 `null`, `aDatabaseContext` performs the fetch. Otherwise, the returned array is returned as the fetch result.

See Also: databaseContextDidFetchObjects

databaseContextShouldInvalidateObjectWithGlobalID

```
public abstract boolean databaseContextShouldInvalidateObjectWithGlobalID(
    EODatabaseContext aDatabaseContext,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    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 `false`.

databaseContextShouldLockObjectWithGlobalID

```
public abstract boolean databaseContextShouldLockObjectWithGlobalID(
    EODatabaseContext aDatabaseContext,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    NSDictionary snapshot)
```

Invoked from `lockObjectWithGlobalID`. The delegate should return `true` if it wants the operation to proceed or `false` 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 `false`. Methods that override the locking mechanism should throw an exception on the failure to lock exactly one object.

databaseContextShouldRaiseExceptionForLockFailure

```
public abstract boolean databaseContextShouldRaiseExceptionForLockFailure(
    EODatabaseContext aDatabaseContext,
    Throwable exception)
```

Invoked from `lockObjectWithGlobalID`. This method allows the delegate to suppress an *exception* that has occurred during *aDatabaseContext*'s attempt to lock the object.

databaseContextShouldSelectObjects

```
public abstract boolean databaseContextShouldSelectObjects(
    EODatabaseContext aDatabaseContext,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    EODatabaseChannel channel)
```

Invoked from the EODatabaseChannel method `selectObjectsWithFetchSpecification` 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 `true` the channel will go ahead and select the object; if the delegate returns `false` (possibly after issuing custom SQL against the adaptor) the *channel* will skip the select and return.

databaseContextShouldUpdateCurrentSnapshot

```
public abstract NSDictionary databaseContextShouldUpdateCurrentSnapshot(
    EODatabaseContext aDatabaseContext,
    NSDictionary currentSnapshot,
```

```
    NSDictionary newSnapshot,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    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 results in *aDatabaseContext* broadcasting an EOObjectsChangedInStoreNotification, causing the object store hierarchy to invalidate existing objects (as identified by *globalID*) built from the obsolete snapshot. You can use this method to achieve the same effect as using a EOFetchSpecification with setRefreshesRefetchedObjects: set to true—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* (or null) causes the *aDatabaseContext* to perform the default behavior (that is, not updating the older snapshot).

databaseContextShouldUsePessimisticLock

```
public abstract boolean databaseContextShouldUsePessimisticLock(
    EODatabaseContext databaseContext,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpecification,
    EODatabaseChannel channel)
```

Invoked from the EODatabaseChannel method selectObjectsWithFetchSpecification 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 true the channel locks the rows being selected; if the delegate returns false the channel selects the rows without locking.

databaseContextWillFireArrayFaultForGlobalID

```
public abstract void databaseContextWillFireArrayFaultForGlobalID(
    EODatabaseContext dbContext,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
```

```
EORelationship relationship,
com.apple.yellow.eocontrol.EOFetchSpecification fetchSpec,
com.apple.yellow.eocontrol.EOEditingContext edContext)
```

Invoked just before the Framework-generated fetch specification, *fetchSpec*, is used to clear the fault for the specified globalID and relationship.

Note: It is very dangerous to modify the fetch specification.

databaseContextWillFireObjectFaultForGlobalID

```
public abstract void databaseContextWillFireObjectFaultForGlobalID(
    EODatabaseContext dbContext,
    com.apple.yellow.eocontrol.EOGlobalID globalID,
    com.apple.yellow.eocontrol.EOFetchSpecification fetchSpec,
    com.apple.yellow.eocontrol.EOEditingContext edContext)
```

Invoked just before the Framework-generated fetch specification, *fetchSpec*, is used to clear the fault for the specified globalID.

Note: It is very dangerous to modify the fetch specification.

databaseContextWillOrderAdaptorOperations

```
public abstract NSArray databaseContextWillOrderAdaptorOperations(
    EODatabaseContext aDatabaseContext,
    NSArray databaseOperations)
```

Sent from ownsGlobalID. 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.

databaseContextWillPerformAdaptorOperations

```
public abstract NSArray databaseContextWillPerformAdaptorOperations(
    EODatabaseContext aDatabaseContext,
    NSArray adaptorOperations,
    EOAdaptorChannel adaptorChannel)
```

Sent from `ownsGlobalID`. 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.

databaseContextWillRunLoginPanelToOpenDatabaseChannel

```
public abstract boolean databaseContextWillRunLoginPanelToOpenDatabaseChannel(
    EODatabaseContext aDatabaseContext,
    NSArray adaptorOperations,
    EOAdaptorChannel adaptorChannel)
```

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` asks 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 `false` to stop `aDatabaseContext` from running the login panel. In this case, the delegate is responsible for opening the channel. If the delegate returns `true`, `aDatabaseContext` runs the login panel.

EOModelGroup.ClassDelegate

(informal interface)

Package: com.apple.yellow.eoaccess

Interface 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 EOModelGroup.Delegate interface specification.

Instance Methods

defaultModelGroup

`public abstract 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 `null`, EOModelGroup uses the default behavior of the `defaultModelGroup` class method. Note that this method is implemented by the delegate assigned to the EOModelGroup class object.

See Also: `classDelegate` (EOModelGroup class), `setClassDelegate` **(EOModelGroup class)**

EOModelGroup.Delegate

Package: com.apple.yellow.eoaccess

Interface 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:

- `entityRelationshipForRow`

- `subEntityForEntity`

- `entityFailedToLookupClassNamed`

- `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 EOModelGroup.ClassDelegate interface description.

Instance Methods

classForObjectWithGlobalID

```
public abstract Class classForObjectWithGlobalID(
    EOEntity entity,
    com.apple.yellow.eocontrol.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.

entityFailedToLookupClassNamed

```
public abstract Class entityFailedToLookupClassNamed(
    EOEntity entity,
    String 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.

entityRelationshipForRow

```
public abstract EORelationship entityRelationshipForRow(
    EOEntity entity,
    NSDictionary row,
    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.

modelGroupEntityWithName

```
public abstract EOModel modelGroupEntityWithName(
    EOModelGroup group,
    String name)
```

If implemented by the delegate, this method should search the *group* for the entity named *name* and return the entity's EOModel. Return null if *name* is not an entity in *group*.

relationshipFailedToLookupDestinationWithName

```
public abstract EOEntity relationshipFailedToLookupDestinationWithName(
    EORelationship relationship,
    String 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 null, the destination entity is set to null.

subEntityForEntity

```
public abstract EOEntity subEntityForEntity(
    EOEntity entity,
    NSDictionary primaryKey)
```

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*.

EOPropertyListEncoding

Implemented by: EOAttribute
                           EOEntity
                           EORelationship
                           EOStoredProcedure

Package: com.apple.yellow.eoaccess

Interface Description

The EOPropertyListEncoding interface declares methods that read and write objects to **property lists** —a dictionary containing only property list data types (that is, NSDictionary objects, Strings, NSArray objects, and NSData objects).

Classes that implement this interface must also provide a constructor that creates objects from a property list and an owner:

```
public ClassName(NSDictionary propertyList, Object owner)
```

Objects created with a constructor of this type are initialized from the provided property list. 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 the property list 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.

Instance Methods

awakeWithPropertyList

```
public abstract void awakeWithPropertyList(NSDictionary propertyList)
```

Finishes initializing the receiver from *propertyList*, which must have been created with a constructor of the form:

```
public ClassName(NSDictionary propertyList, Object 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 created from *propertyList*.

encodeIntoPropertyList

```
public abstract void encodeIntoPropertyList(NSMutableDictionary propertyList)
```

Encodes the receiver as a property list.

Deprecated API

This file enumerates those EOAccess Framework classes and methods that have been deprecated and should no longer be used. Wherever possible, notes have been included to indicate what API should be used in place of the deprecated class or method.

EOAdaptorContext

Nested transactions are no longer supported. Enterprise Objects Framework never actually used nested transactions. Furthermore, the concrete adaptors were not guaranteed to support them, especially since the SQL/92 standard doesn't allow nested transactions. New features in Enterprise Objects Framework 4.5 make nested transactions impossible to support. Consequently, the methods supporting nested transactions have been depreted.

canNestTransactions

```
public boolean canNestTransactions()
```

Deprecated in Enterprise Objects Framework 4.5. Don't use this method. There is no new API or workaround; no adaptor supports nested transactions.

Implemented by subclasses to return `true` if the database server and adaptor context could nest transactions, `false` otherwise.

transactionNestingLevel

```
public int transactionNestingLevel()
```

Deprecated in Enterprise Objects Framework 4.5. Use `hasOpenTransaction` instead.

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

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software.

Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.