EOAttribute

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOAccess/EOAttribute.h

Class Description

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

EOAttributes are also used to represent arguments for EOStoredProcedures.

You usually define attributes in your EOModel with the EOModeler application, which is documented in the *Enterprise Objects Framework Developer's Guide*. Your code probably won't need to programmatically interact with EOAttribute unless you're working at the adaptor level.

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

A flattened attribute of an entity is actually an attribute of some other entity reached through a relationship. 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".

Like the other major modeling classes, EOAttribute provides a user dictionary for the application to store any auxiliary information it needs.

Mapping from Database to Objects

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

- NSString
- NSNumber
- NSDecimalNumber
- NSData
- NSDate

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

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

Working with Custom Data Types

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

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

- Set the attributes's value class (using the EOAttribute method **setValueClassName:**).
- Set the factory method that will be used to create instances of your class from raw data (using the EOAttribute method **setValueFactoryMethodName:**).
- Set the type of the argument that should be passed to the factory method (using the EOAttribute method **setFactoryMethodArgumentType:**).

- Set the conversion method that will be used to convert your data back into one of the primitive data types the adaptor can work with; this enables the data to be stored in the database (using the EOAttribute method **setAdaptorValueConversionMethodName:**).

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

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

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

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

You can also define a custom data type using the Attributes Inspector in EOModeler. For more information, see the chapter "Advanced Modeling Techniques" in the *Enterprise Objects Framework Developer's Guide*.

Fetching Custom Values

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

newValueForBytes:length: can handle NSData and raw bytes (**void ***). It converts the raw bytes into an NSData if the custom value argument type is EOFactoryMethodArgumentIsNSData, then invokes the custom value factory method with the NSData or bytes. If the EOAttribute has no custom value factory method, this method simply returns an NSData object containing the bytes.

newValueForBytes:length:encoding: can handle NSString and raw bytes. It converts the raw bytes into an NSString if the custom value argument type is EOFactoryMethodArgumentIsNSString, then invokes the

custom value factory method with the string or bytes. If the EOAttribute has no custom value factory method, this method simply returns an NSString object created from the bytes.

Converting Custom Values

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

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 external name)
- A declaration of the type of that column as defined by the database and adaptor (the external type)
- A declaration of the Objective-C class used to represent values in the application
- For Objective-C value classes that require it, a subtype for such distinctions as between numeric types

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

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

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

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

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

Creating a Derived Attribute

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

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

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

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

Creating a Flattened Attribute

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

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

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

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

Method Types

Getting the entity	– entity – parent
Setting the name	– beautifyName – name – setName: – validateName:
Setting date information	– serverTimeZone – setServerTimeZone:

Setting external definitions	– columnName – definition – externalType – setColumnName: – setDefinition: – setExternalType:
Setting value type information	– allowsNull – precision – scale – setAdaptorValueConversionMethodName: – setAllowsNull: – setFactoryMethodArgumentType: – setPrecision: – setScale: – setValueClassName: – setValueFactoryMethodName: – setValueType: – setWidth: – validateValue: – valueClassName – valueType – width
Checking type information	– isDerived – isFlattened – isReadOnly – setReadOnly:
Setting SQL statement formats	– setWriteFormat: – setReadFormat: – readFormat – writeFormat
Setting the user dictionary	– setUserInfo: – userInfo

Methods used by the adaptor	– adaptorValueByConvertingAttributeValue: – adaptorValueConversionMethod – adaptorValueConversionMethodName – adaptorValueType – factoryMethodArgumentType – newDateForYear:month:day:hour:minute:second:millisecond: timezone:zone: – newValueForBytes:length: – newValueForBytes:length:encoding: – valueFactoryMethod – valueFactoryMethodName
Working with stored procedures	– parameterDirection – storedProcedure – setParameterDirection:

Instance Methods

adaptorValueByConvertingAttributeValue:

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

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

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

adaptorValueConversionMethod

– (SEL)**adaptorValueConversionMethod**

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

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

adaptorValueConversionMethodName

– (NSString *)**adaptorValueConversionMethodName**

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

See also: – **adaptorValueByConvertingAttributeValue:**

adaptorValueType

– (EOAdaptorValueType)**adaptorValueType**

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

EOAdaptorNumberType
EOAdaptorCharactersType
EOAdaptorBytesType
EOAdaptorDateType

See also: – **factoryMethodArgumentType:**

allowsNull

– (BOOL)**allowsNull**

Returns whether or not the attribute can have a **nil** value. 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.

beautifyName

– (void)**beautifyName**

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

See also: – **validateName:**

columnName

– (NSString *)**columnName**

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

See also: – **definition**, – **externalType**

definition

– (NSString *)**definition**

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

See also: – **columnName**, – **externalType**

entity

– (EOEntity *)**entity**

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

See also: – **storedProcedure**

externalType

– (NSString *)**externalType**

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

See also: – **columnName**

factoryMethodArgumentType

– (EOFactoryMethodArgumentType)**factoryMethodArgumentType**

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

Return Value	Argument Type
EOFactoryMethodArgumentIsNSData	an NSData
EOFactoryMethodArgumentIsNSString	an NSString
EOFactoryMethodArgumentIsBytes	raw bytes

See also: – **valueFactoryMethod**

isDerived

– (BOOL)**isDerived**

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

Note that flattened attributes are also considered as derived attributes.

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

isFlattened

– (BOOL)**isFlattened**

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

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

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

isReadOnly

– (BOOL)**isReadOnly**

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

name

– (NSString *)**name**

Returns the attribute’s name.

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

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

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

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

newValueForBytes:length:

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

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

newValueForBytes:length:encoding:

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

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

parameterDirection

– (EOParameterDirection)**parameterDirection**

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

EOVoid
EOInParameter
EOOutParameter
EOInOutParameter

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

parent

– (id)**parent**

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

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

precision

– (unsigned)**precision**

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

See also: – **scale**

readFormat

– (NSString *)**readFormat**

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

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

scale

– (int)**scale**

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

See also: – **precision**

serverTimeZone

– (NSTimeZone *)**serverTimeZone**

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

See also: + **localTimeZone** (NSTimeZone)

setAdaptorValueConversionMethodName:

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

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

setAllowsNull:

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

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

setColumnName:

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

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

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

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

See also: – **setDefinition:**

setDefinition:

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

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

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

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

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

See also: – **setColumnName:**

setExternalType:

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

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

See also: – **setDefinition:**

setFactoryMethodArgumentType:

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

Sets 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. The factory method accepts NSStrings, NSDatas, and raw bytes; specify an *argumentType* of EOFactoryMethodArgumentIsNSString, EOFactoryMethodArgumentIsNSData, or EOFactoryMethodArgumentIsBytes as appropriate.

See also: – **setValueFactoryMethodName:**

setName:

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

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

See also: – **validateName:**

setParameterDirection:

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

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

EOVoid
EOInParameter

EOOutParameter
EOInOutParameter

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

setPrecision:

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

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

See also: – **setScale:**

setReadFormat:

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

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

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

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

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

setReadOnly:

– (void)**setReadOnly:**(BOOL)*flag*

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

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

setScale:

– (void)**setScale:**(int)*scale*

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

See also: – **setPrecision:**

setServerTimeZone:

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

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

setUserInfo:

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

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types (that is, NSDictionaries, NSStrings, NSArrays, and NSDatas).

setValueClassName:

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

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

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

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

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

See also: – **setValueType:**

setValueFactoryMethodName:

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

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

setValueType:

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

Sets to *typeName* the format type for custom value classes, such as “TIFF” or “RTF”.

See also: – **setValueClassName:**

setWidth:

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

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

See also: – **width**

setWriteFormat:

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

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

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

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

storedProcedure

– (EOStoredProcedure *)**storedProcedure**

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

See also: – **entity**

userInfo

– (NSDictionary *)**userInfo**

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

See also: – **setUserInfo:**

validateName:

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

Validates *name* and returns **nil** if it is a valid name, or an exception if it isn't. A name is invalid if it has zero length; starts with a character other than a letter, a number, or "@", "#", or "_"; or contains a character other than a letter, a number, "@", "#", "_", or "$".

setName: uses this method to validate its argument.

validateValue:

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

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

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

valueClassName

– (NSString *)**valueClassName**

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

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

See also: – **valueType**

valueFactoryMethod

– (SEL)**valueFactoryMethod**

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

valueFactoryMethodName

– (NSString *)**valueFactoryMethodName**

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

See also: – **valueFactoryMethod**

valueType

– (NSString *)**valueType**

Returns the format type for custom-value classes, such as "TIFF" or "RTF".

See also: – **valueClassName**

width

– (unsigned)**width**

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

See also: – **setWidth:**

writeFormat

– (NSString *)**writeFormat**

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

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