EOQualifier

Inherits From:	NSObject
Conforms To:	NSCopying
Declared In:	EOControl/EOQualifier.h

Class Description

EOQualifier is an abstract class for objects that hold information used to restrict selections on objects or database rows according to specified criteria. With the exception of EOSQLQualifier, qualifiers aren't based on SQL and they don't rely upon an EOModel. Thus, the same qualifier can be used both to perform in-memory searches and to fetch from the database.

The subclasses of EOQualifier are as follows:

EOKeyValueQualifier	Compares the named property of an object to a supplied value, for example, "weight > 150".
EOKeyComparisonQualifier	Compares the named property of one object with the named property of another, for example "name = wife.name".
EOAndQualifier	Contains multiple qualifiers, which it conjoins. For example, "name = 'Fred' AND age < 20".
EOOrQualifier	Contains multiple qualifiers, which it disjoins. For example, "name = 'Fred' OR name = 'Ethel'".
EONotQualifier	Contains a single qualifier, which it negates. For example, "NOT (name = 'Fred')".
EOSQLQualifier	Contains unstructured text that can be transformed into a SQL expression. EOSQLQualifier is provided for backward compatibility with pre-2.0 Enterprise Objects Framework releases and to provide a way to create SQL expressions with any arbitrary SQL. Because EOSQLQualifiers can't be evaluated against objects in memory, you should use EOQualifier whenever possible and only use EOSQLQualifier in cases that absolutely require it.

With the exception of EOSQLQualifier, all of the subclasses are essentially key-value based and implement the protocol EOQualifierEvaluation, which means that their objects can be evaluated in memory. The subclasses also implement the protocol EOQualifierSQLGeneration, which allows them to generate SQL. Note that all of the SQL generation functionality is contained in the access layer. For more information, see the specifications for EOQualifierEvaluation, EOQualifierSQLGeneration, and the individual subclasses.

Creating a Qualifier

As described above, there are several EOQualifier subclasses, each of which represents a different semantic. However, in most cases you simply create a qualifier using the EOQualifier class method **qualifierWithQualifierFormat:**, as follows:

```
EOQualifier *qual = [EOQualifier
    qualifierWithQualifierFormat:@"lastName = 'Smith'"];
```

The qualifier or group of qualifiers that result from such a statement is based on the contents of the format string you provide. For example, giving the format string "lastName = 'Smith'" as an argument to **qualifierWithQualifierFormat:** returns an EOKeyValueQualifier object. But you don't normally need to be concerned with this level of detail.

The format strings you use to create a qualifier can be compound logical expressions, such as "firstName = 'Fred' AND age < 20". When you create a qualifier, compound logical expressions are translated into a tree of EOQualifier nodes. Logical operators such as AND and OR become EOAndQualifiers and EOOrQualifiers, respectively. These qualifiers conjoin (AND) or disjoin (OR) a group of sub-qualifiers. This is illustrated in Figure 1, in which the format string "salary > 300 AND firstName = 'Angela' AND manager.name = 'Fred'" has been translated into a tree of qualifiers.

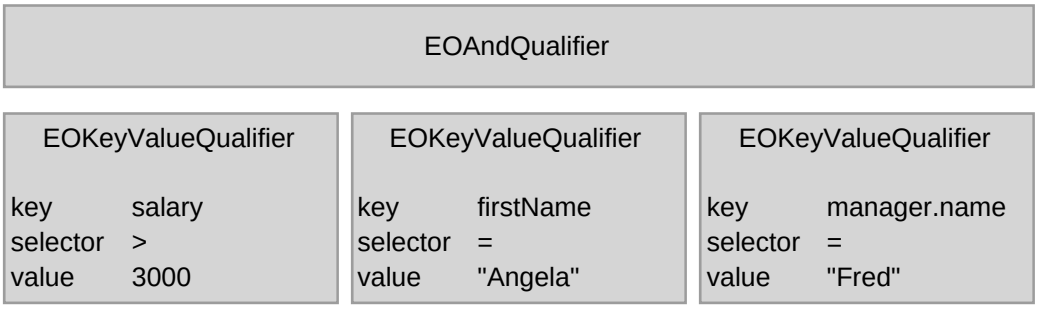

Figure 1 EOQualifier Tree for **salary > 300 AND firstName = "Angela" AND manager.name = "Fred"**

Note: The **qualifierWithQualifierFormat:** method can't be used to create an instance of EOSQLQualifier. This is because EOSQLQualifier uses a non-structured syntax to provide backward compatibility with pre-2.0 Enterprise Objects Framework releases. It also requires an entity. To create an instance of EOSQLQualifier, you'd use a statement such as the following:

```
myQual = [[EOSQLQualifier alloc] initWithEntity:myEntity format:myFormatString];
```

Constructing Format Strings

As described above, you typically create a qualifier from a format string by using **qualifierWithQualifierFormat:**. This method takes as an argument a format string somewhat like that used with the standard C **printf()** function. The format string can embed strings, numbers, and objects using

the conversion specifications listed below. This allows qualifiers to be built dynamically. The following table lists the conversion specifications you can use in a format string and their corresponding data types.

Conversion Specification	Expected Value or Result
%s	A constant C string (**const char ***).
%d	An **int**.
%f	A **float** or **double**.
%@	An **id** argument—valid objects are EOAttribute, NSString, or anything that responds to **expressionValueForContext:**. The behavior of this conversion specification depends on its position. It can either be an object whose description method returns a key (in other words, an NSString), or a value such as an NSString, NSNumber, NSCalendarDate, and so on.
%%	Results in a literal **%** character.

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

For example, suppose you have an Employee entity with the properties **empID**, **firstName**, **lastName**, **salary**, and **department** (representing a to-one relationship to the employee's department), and a Department entity with properties **deptID**, and **name**. You could construct simple qualifier strings like the following:

```
lastName = 'Smith'
salary > 2500
department.name = 'Personnel'
```

The following examples build qualifiers similar to the qualifier strings described above, but take the specific values from an already-fetched enterprise object:

```
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ = %@",
   @"lastName", [anEmployee lastName]];
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ > %f",
   @"salary", [anEmployee salary]];
myQualifier = [EOQualifier qualifierWithQualifierFormat:@"%@ = %@",
   @"department.name", [aDept name]];
```

The enterprise objects here implement convenience methods for directly accessing the given attributes: **lastName** and **salary** for Employee objects, and **name** for Department objects.

Note: Unlike a string literal, the %@ conversion specification is never surrounded by single quotes:

```
// For a literal string value such as Smith, you use single quotes.
[EOQualifier qualifierWithQualifierFormat:@"lastName = 'Smith'"];

// For the conversion specification %@, you don’t use quotes
[EOQualifier qualifierWithQualifierFormat:@"lastName = %@", @"Jones"];
```

Typically format strings include only two data types: strings and numbers. Single-quoted or double-quoted strings are NSStrings, non-quoted numbers are NSNumber, and non-quoted strings are keys. You can get around this limitation by performing explicit casting, as described in the section “Using Different Data Types.”

The operators you can use in constructing qualifiers are =, ==, !=, <, >, <=, >=, and “like”. The **like** operator can be used with wildcards to perform pattern matching, as described in the following section.

Checking for NULL Values

To construct a qualifier that fetches rows with null values, use any of the following approaches:

```
[EOQualifier qualifierWithQualifierFormat:@"bonus = nil"];
[EOQualifier qualifierWithQualifierFormat:@"bonus = %@", [EONull null]];
[EOQualifier qualifierWithQualifierFormat:@"bonus = %@", nil];
```

Using Wildcards and the like Operator

When you use the **like** operator in a qualifier expression, you can use the * and ? wildcard characters to perform pattern matching, for example:

```
@"lastName like ’Jo*’"
```

matches Jones, Johnson, Jolsen, Josephs, and so on.

The ? character just matches a single character, for example:

```
@"lastName like ’Jone?’"
```

matches Jones.

The asterisk character (*) is only interpreted as a wildcard in expressions that use the **like** operator. For example, in the following statement, the character * is treated as a literal value, not as a wildcard:

```
@"lastName = ’Jo*’"// The * character doesn’t act as a wildcard in this statement.
```

Using Selectors in Qualifier Expressions

The format strings you use to initialize a qualifier can include selectors. The parser recognizes a selector as an unquoted string followed by a colon, such as **myMethod:**. For example:

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

Selectors in a qualifier are parsed and applied in memory; that is, they can't be used in SQL generation.

Using Different Data Types in Format Strings

As stated in the section "Constructing Format Strings," format strings normally include only two data types: strings and numbers. To get around this limitation, you can perform explicit casting.

For example, NSCalendarDate and NSDecimalNumber are two classes that are likely to be used in qualifiers. You can construct format strings that for objects of these classes as follows:

```
hireDate = (NSCalendarDate)'1990-03-16 00:00:00 +0000'
salary = (NSDecimalNumber)'15000.02'
```

When you use this approach, qualifiers are constructed by looking up the class and invoking `[[`*`class`* `alloc] initWithString:stringValue]`. Therefore, this technique only works for classes that implement **initWithString:**.

Note that to construct a date qualifier using a format string, you must use the default NSCalendarDate format, which is %Y-%m-%d %H:%M:%S %z—for example:

```
EOQualifier *qual = [EOQualifier qualifierWithQualifierFormat:
   @"dateReleased < (NSCalendarDate)'1990-01-26 00:00:00 +0000'"];
```

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

```
EOQualifier *qual = [EOQualifier qualifierWithQualifierFormat:
   @"dateReleased > %@", [NSCalendarDate calendarDate]];
```

Using EOQualifier's Subclasses

You rarely need to explicitly create an instance of EOAndQualifier, EOOrQualifier, or EONotQualifier. However, you may want to create instances of EOKeyValueQualifier and EOKeyComparisionQualifier. The primary advantage of this is that it lets you exercise more control over how the qualifier is constructed, which is desirable in some cases.

The following code excerpt uses EOKeyValueQualifier to select all objects whose "isOut" key is equal to YES. Once constructed, the qualifier is used to filter an in-memory array.

```
// Create the qualifier
EOQualifier *qual = [[EOKeyValueQualifier alloc] initWithKey:@"isOut"
       operatorSelector:EOQualifierOperatorEqual
       value:[NSNumber numberWithBool:YES]];

// Filter an array and return it
return [[self allRentals] filteredArrayUsingQualifier:qual];
```

filteredArrayUsingQualifier: is a method that Enterprise Objects Framework adds to NSArray. It's used for filtering in-memory arrays.

Creating Subclasses

EOQualifier offers extensibility across two dimensions: new classes can be added to extend qualifier semantics, and categories can be added to extend functionality (for example, to provide in-memory evaluation).

Subclasses used to evaluate objects in memory must implement the EOQualifierEvaluation protocol. Subclasses used to generate SQL queries must conform to the EOQualifierSQLGeneration protocol.

Adopted Protocols

NSCopying

Method Types

Create a qualifier	+ qualifierWithQualifierFormat: + qualifierWithQualifierFormat:arguments:
Convert strings and operators	+ operatorSelectorForString: + stringForOperatorSelector:
Validate a qualifier's keys	– validateKeysWithRootClassDescription:

Class Methods

operatorSelectorForString:

+ (SEL)**operatorSelectorForString:**(NSString *)*aString*

Returns an operator selector based on the string *aString*. This method is used in parsing a qualifier. For example, the following statement returns the selector **isNotEqualTo:**.

```
selector = [EOQualifier operatorSelectorForString:@"!="];
```

The possible values of *string* are =, ==, !=, <, >, <=, >=, and "like".

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

See also: + **stringForOperatorSelector:**

qualifierWithQualifierFormat:

+ (EOQualifier *)**qualifierWithQualifierFormat:**(NSString *)*qualifierFormat, ...*

Parses the format string *qualifierFormat*, uses it to create an EOQualifier, and returns the EOQualifier. Based on the content of *qualifierFormat*, generates a tree of the basic qualifier types. For example, the format string “firstName = 'Joe' AND department = ’Facilities’" generates an EOAndQualifier that contains two “sub” EOKeyValueQualifiers. The following code excerpt shows a typical way to use the **qualifierWithQualifierFormat:** method. The excerpt constructs an EOFetchSpecification, which includes an entity name and a qualifier. It then applies the EOFetchSpecification to the EODisplayGroup’s data source and tells the EODisplayGroup to fetch.

```
EODisplayGroup *displayGroup;     /* Assume this exists.*/
EOFetchSpecification *fetchSpec;
EODatabaseDataSource *dataSource;

dataSource = [displayGroup dataSource];
fetchSpec = [EOFetchSpecification
   fetchSpecificationWithEntityName:@"Member"
   qualifier:[EOQualifier qualifierWithQualifierFormat:
   @"cardType = 'Visa' "]
   sortOrderings:nil];
[dataSource setFetchSpecification:fetchSpec];
[displayGroup fetch];
```

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

qualifierWithQualifierFormat:arguments:

+ (EOQualifier *)**qualifierWithQualifierFormat:**(NSString *)*qualifierFormat*
  arguments:(NSArray *)*arguments*

Parses the format string *qualifierFormat* and the specified *arguments*, uses them to create an EOQualifier, and returns the EOQualifier. This method is equivalent to **qualifierWithQualifierFormat:** except that format characters (for example, %@, %d, %f) in *qualifierFormat* cause the method to search in the *arguments* array for values rather than in a variable argument list.

stringForOperatorSelector:

+ (NSString *)**stringForOperatorSelector:**(SEL)*aSelector*

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

```
operator = [EOQualifier stringForOperatorSelector:EOQualifierOperatorNotEqual];
```

The possible values for *selector* are as follows:

EOQualifierOperatorEqual
EOQualifierOperatorNotEqual
EOQualifierOperatorLessThan
EOQualifierOperatorGreaterThan
EOQualifierOperatorLessThanOrEqualTo
EOQualifierOperatorGreaterThanOrEqualTo
EOQualifierOperatorLike

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

See also: + **operatorSelectorForString:**

Instance Methods

validateKeysWithRootClassDescription:

- (NSException *)**validateKeysWithRootClassDescription:**(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 **nil** to indicate that the keys contained by the qualifier are valid.