Expression Tree Interface

This is informal user documentation for the Objectivity expression tree interface.

This feature has been updated for the 11.2 release.

Contents

Purpose of Objectivity Expression Tree Interface

This feature provides a public interface for accessing the Objectivity expression tree, which enables users to qualify persistent objects according to the values of one or more of their attributes.

In pre-Release 10.0 versions, applications could use ooQuery to qualify individual objects. However, an aspect of the ooQuery mechanism was problematic—it did not expose its internal predicate representation (or *expression tree*). Accordingly, the only way to use ooQuery was to translate to and from the Objectivity/DB predicate query language (PQL). Users had to create their own predicate expression structures and create the ability to convert to and from PQL.

Full access to the underlying predicate-query expression-tree representation is now provided, so a non-Objectivity/DB query front end may be adapted to perform an Objectivity/DB object-qualification operation. A predicate expression tree may be programmatically validated in an iterative query-building approach, and the result saved in portable form—either for re-use or for parallelized use (with the parallel query server).

Access to the expression tree interface provides support for:

- External search agents and user-defined indexes. These tools need the ability to examine the predicate expression so that they can interface with query systems unfamiliar with PQL.
- Query builders and parsers. These tools need the ability to update predicate expressions so they can build a predicate expression directly.

The point of entry for using the expression tree is through the ObjectQualifier class, which supersedes the ooQuery class. The ObjectQualifier class can be used qualifiy objects according to a provided predicate string or a provided predicate expression.

The ObjectQualifier class can be used to qualify objects one at a time, or it can be used in a scan operation that searches the storage hierarchy of a federated database for objects of interest.

Objectivity Expression Tree Users

The Objectivity expression tree interface is intended for predicate expression tree tools, such as:

- Parsers
- Query builders
- External search agents
- User-defined indexes
- Custom splitters

Conceptual Model

Overview

Note: This document uses C++ as the binding language for all diagrams and examples.

Starting with release 10.0, Objectivity uses a predicate expression tree paired with a result type to represent a predicate used for object qualification. The expression-tree interface provides a set of classes that let you represent predicate expression trees, and provides a mechanism for validating expression trees and using them for object qualification.

A set of new class structures and public APIs are provided for:

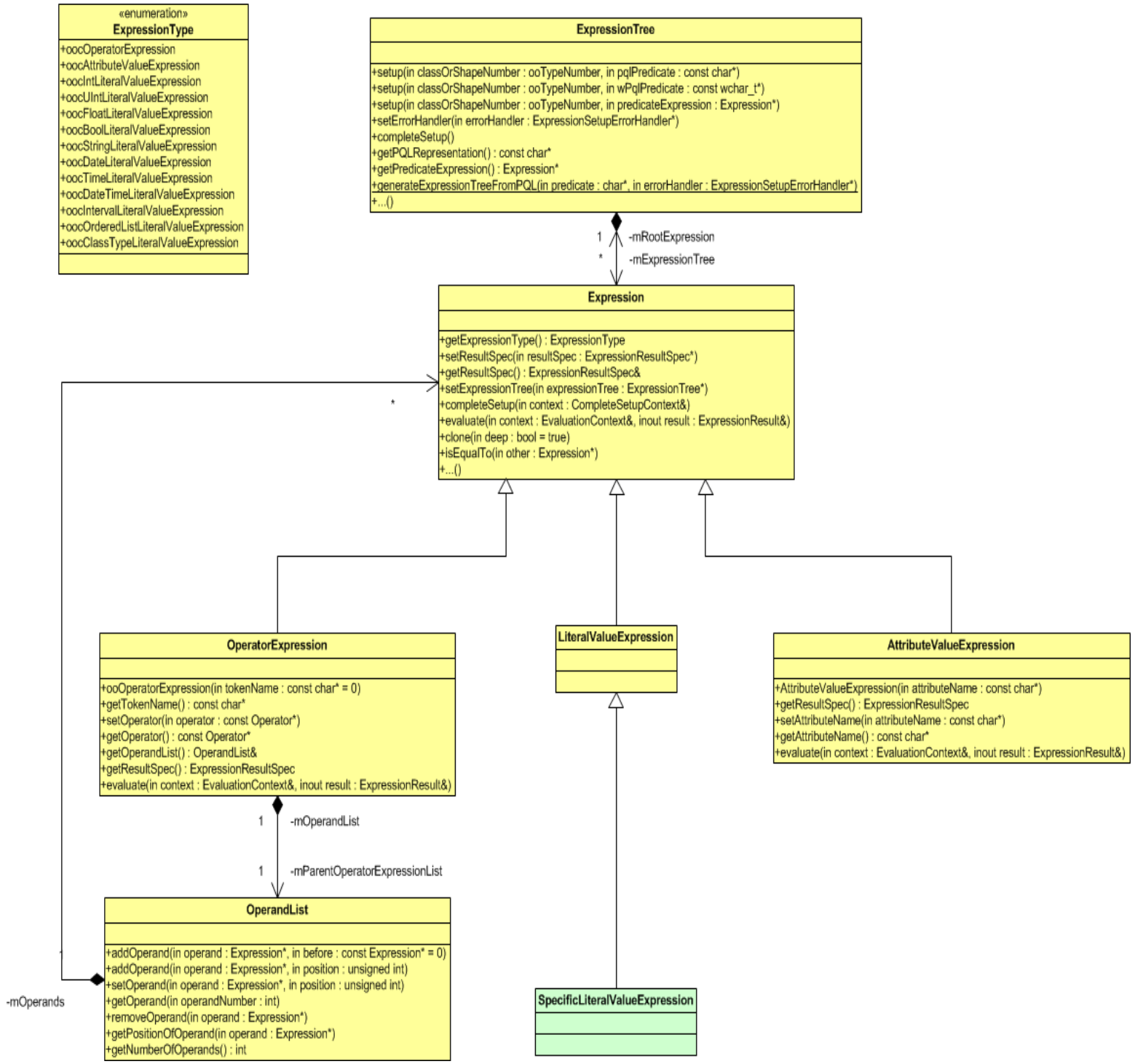
- Constructing a predicate expression tree.
- Examining and manipulating a predicate expression tree.

- Validating a predicate expression tree.
- Extending a predicate expression tree by adding user-defined data or user-defined behavior to the expression tree's nodes.

The main abstractions for representing the expression tree and performing object qualification are described in the following sections.

Expression Tree

The following diagram shows the main classes that represent the expression tree interface.



The **ExpressionTree** class holds a handle to the expression tree's head node, which is an instance of the Expression class. The ExpressionTree class provides methods to set up an expression tree, validate it, and subsequently access it.

The **Expression** class is the base class for the three types of expression nodes in the expression tree interface:

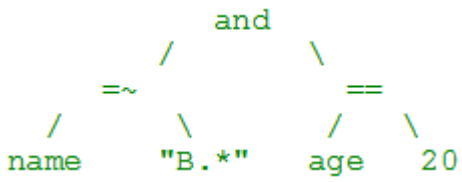
- **OperatorExpression**: represents an operator expression node.
- **AttributeValueExpression**: represents an attribute-value expression node of either a basic type member (Boolean, int, unsigned int, float, string, date, time, datetime, interval), a reference or an embedded member, or a member of query-collection type (fixed or variable array, fixed array of variable arrays, to-many relationship, or persistent collection).
- **LiteralValueExpression**: base class for all literal value expression classes to represent all supported types of literal value nodes.
    - **BoolLiteralValueExpression**: represents a Boolean literal value expression node.
    - **StringLiteralValueExpression**: represents a string literal value expression node.
    - **IntLiteralValueExpression**: represents an integer literal value expression node.
    - **UIntLiteralValueExpression**: represents an unsigned integer literal value expression node.
    - **FloatLiteralValueExpression** represents a float literal value expression node.
    - **DateLiteralValueExpression** represents a date literal value expression node.
    - **TimeLiteralValueExpression** represents a time literal value expression node.
    - **DateTimeLiteralValueExpression** represents a datetime literal value expression node.
    - **IntervalLiteralValueExpression** represents an interval literal value expression node.
    - **OrderedListLiteralValueExpression** represents an ordered-list literal value node node.
    - **ClassTypeLiteralValueExpression** represents a class-type literal value expression node.
    - **OidLiteralValueExpression** represents an object's OID literal value expression node.
    - **ObjectLiteralValueExpression** represents a referenced or embedded object literal value expression node.

All expression types are captured by the **ExpressionType** enumeration. The Expression::getExpressionType() virtual function returns the ExpressionType value.

An expression tree is made of expression nodes such that:

- The head node's result type is Boolean.
- Intermediate level nodes (non leaves) are OperatorExpression nodes, which can have a number of other expression nodes as their operands.
- Leaf nodes are value expression nodes of type AttributeValueExpression or LiteralValueExpression.

For example:



OperatorExpression

The OperatorExpression class defines an operator expression that is associated with an operator and holds a list of operands.

An OperatorExpression object can be constructed with or without a specified token name. If constructed with a token name, as is typically done by parsers, the actual operator used by the operator expression is selected later during the setup process. The specified token name can be retrieved with the getTokenName() method. If not constructed with a token name, the operator must be specified using the setOperator(const Operator*) method. In either case, the selected operator can be obtained using getOperator().

An OperatorExpression has one or more operands, which can be of any type of Expression. The list of operands for an OperatorExpression is stored and managed by an OperandList.

The **OperandList** class provides public APIs to add, access or manipulate the operands:

- `addOperand()`
- `setOperand()`
- `getOperand()`
- `removeOperand()`
- `getNumberOfOperands()`

You can navigate the whole expression tree starting from the head node. To do this, use a combination of function calls:

- getOperandList() on each OperatorExpression node.
- getNumberOfOperands() and getOperand() on each returned OperandList.

Note: See the *Custom Operator Support* topic on this website for details about the Operator class.

AttributeValueExpression

The AttributeValueExpression class defines an attribute value of the target object or of a related object. The following data types are supported.

- A basic type, such as the name or age for an instance of an *employee* class.
- A reference or embedded object type, such as the employer for an instance of the *employee* class.
- An attribute value of a query collection type, such as the children of an instance of the *employee* class.

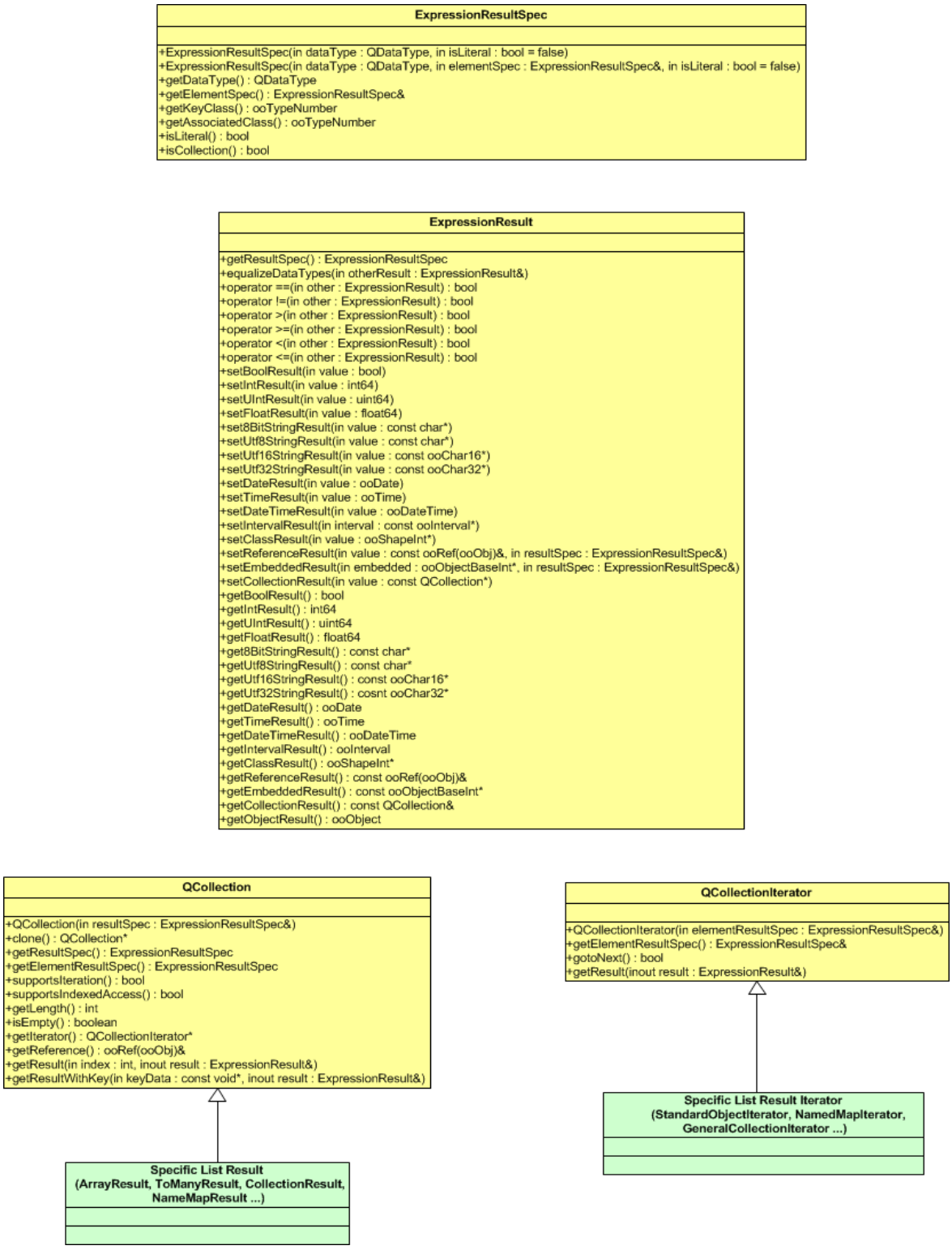
The AttributeValueExpression constructor takes the attribute name as an argument. If the same attribute name is defined in both the base class and the class being qualified (or is referenced from the class being qualified), the following syntax must be used to express the attribute of the base class: *baseClassName::AttributeName.*

LiteralValueExpression

The LiteralValueExpression class is an abstract base class for all literal value expression classes. Use the constructors of its derived classes to create literal value instances of particular types. When creating a particular type of literal value object, supply the literal value to the constructor. The literal value can be reset with setValue() or retrieved with getValue().

Expression Result

The following diagram shows the classes representing the expression result and the supported expression result types.



QDataType

QDataType is an enumeration type that captures all data types that can be queried. It is used as part of an ExpressionResultSpec to describe the types of results produced by literals, attributes, and operator expressions.

```
enum QDataType
{
  oocQNoDataType = 0,
  oocQUInt = 1,
  oocQInt = 2,
  oocQWholeNumber = oocQUInt | oocQInt,
  oocQFloat32 = 4,
  oocQFloat64 = 8,
  oocQFloat = oocQFloat32 | oocQFloat64,
  oocQNumeric = oocQUInt | oocQInt | oocQFloat,
  oocQBool = 16,
  oocQ8BitString = 32,
  oocQUtf8String = 64,
  oocQUtf16String = 128,
  oocQUtf32String = 256,
  oocQUtfString = oocQUtf8String | oocQUtf16String | oocQUtf32String,
  oocQString = oocQ8BitString | oocQUtfString,
  oocQDate = 512,
  oocQTime = 1024,
  oocQDateTime = 2048,
  oocQAllDate = oocQDate | oocQDateTime,
  oocQInterval = 4096,
  oocQSimple = oocQNumeric | oocQBool | oocQString
             | oocQDate | oocQTime | oocQDateTime | oocQInterval,
  oocQClass = 8192,
  oocQReference = 16384,
  oocQEmbedded = 32768,
  oocQObject = oocQReference | oocQEmbedded,
  oocQOrderedArray = 65536,
  oocQOrderedList = 131072,
  oocQOrderedSet = 262144,
  oocQOrderedMap = 524288,
  oocQOldOrderedCollection = 1048576,
  oocQOrderedCollection = oocQOrderedArray | oocQOrderedList |
                          oocQOrderedSet | oocQOrderedMap|
                          oocQOldOrderedCollection,
  oocQUnorderedArray = 2097152,
  oocQUnorderedSet = 4194304,
  oocQUnorderedMap = 8388608,
  oocQOldUnorderedCollection = 16777216,
  oocQUnorderedCollection = oocQUnorderedArray | oocQUnorderedSet |
                           oocQUnorderedMap|oocQOldUnorderedCollection,
  oocQKeyedCollection = oocQOrderedSet | oocQUnorderedSet |
                        oocQOrderedMap | oocQUnorderedMap,
  oocQReferencedCollection = oocQOrderedList | oocQKeyedCollection
                | oocQOldOrderedCollection|oocQOldUnorderedCollection,
  oocQReferenceAccessible = oocQReference | oocQReferencedCollection,
  oocQCollection = oocQOrderedCollection | oocQUnorderedCollection,
  oocQNonCollection = oocQSimple | oocQObject | oocQClass,
  oocQAllDataTypes = oocQNonCollection | oocQCollection
};
```

ExpressionResultSpec

The ExpressionResultSpec class represents the result type of an Expression node in an expression tree. ExpressionResultSpec has one member for the data type and one member for the element type. Note that the element type is only valid if the data type is a query collection (oocQCollection in QDataType).

ExpressionResultSpec is also used to represent the valid operand types of an operator. The QDataType enum field values correspond to bit positions so that they can be combined to represent types of a certain category or grouping. For example, the following shows a combined numerical type for an operand and result type of an arithmetic operator such as multiplication.

oocQNumeric = oocQUInt | oocQUInt |oocFloat

ExpressionResult

The evaluation result of an expression is captured by an ExpressionResult, which has a member attribute for each type of expression result. To set expression result data to an ExpressionResult, call one of the set*xxx()* methods for the corresponding result type. To get the expression result data from an ExpressionResult, find the result type by first calling the getResultType() method, then call one of the get*xxx()* methods for the corresponding result type.

The overloaded operators (=, and ,!=) apply to all supported result types; while other overloaded operators (>, >=, <, <=) apply only to oocQSimple types (oocQNumeric |oocQBool |oocQString |oocQDateTime |oocQInterval).

The enum values for the numerical types (oocQUInt, oocQInt, oocQFloat) also reflect the data type precedence among numerical results. When different numeric result types are involved in an operation evaluation, the result with a lower data type precedence will be cast to the result type with the higher data type precedence.

The equalizeDataTypes(ExpressionResult& otherResult) method is provided to synchronize an ExpressionResult with another ExpressionResult passed as an argument; the result with a lower data-type precedence is upgraded to the result type with the higher data-type precedence.

For those result types that involve memory allocation, such as strings, the evaluate method must allocate the needed memory for the result and set the result to the passed ExpressionResult argument. The ExpressionResult is then the owner of this data and its destructor will free all the memory allocated for it when the expression result goes out of scope.

In some cases, the expression result is unknown. For example, the attribute value might not be set, the index might be out of bounds, or there might be a non-existing reference.

These cases are represented with "null" value results. The isNull()/setIsNull() methods can check if an expression result is null, or set an expression result to null.

QCollection

QCollection is the base class to represent an expression result that has multiple elements—for example, the result of a class member of a query collection type for an attribute value expression.

This class is used to represent both index-accessible collections and iteration-based collections. The supportsIndexedAccess() and supportsIteration() methods are provided to determine the access type of the query collection result. For index-based collections, use getResult(unsigned index, ExpressionResult& result) to get the collection element result. For iteration-based collections, use getIterator() to get the collection iterator in order to walk through each element of the collection.

QCollection derived classes are implemented for the following built-in query-collection attribute types:

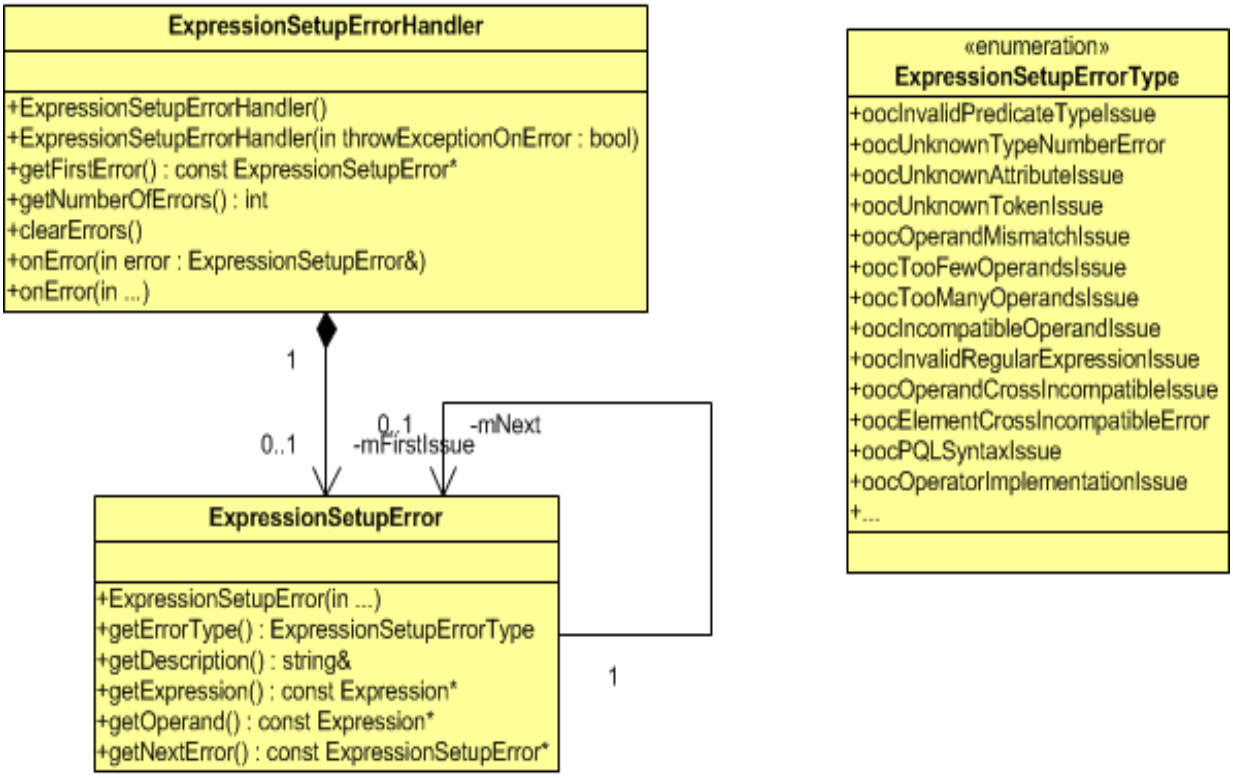
- Fixed arrays
- Variable arrays
- Fixed array of variable arrays
- To-many relationships
- Persistent collections
- Name maps
- Java compatibility arrays

QCollectionIterator

QCollectionIterator is the base class for the collection result iterator for iteration-based QCollection types. Its derived classes are implemented for all iteration-based built-in query collection attribute types.

Expression-Setup Error Handling

The following diagram shows the helper classes used for handling errors that occur during expression setup, which occurs when the ExpressionTree's completeSetup method is called.



ExpressionSetupErrorHandler

Error checking is performed by implicitly or explicitly invoking completeSetup() from the ExpressionTree, which in turn calls completeSetup() from the head node, which then propagates to the whole expression tree.

The completeSetup methods are defined for the following classes:

- AttributeValueExpression
- OperatorExpression
- Certain operators (subscript operators, regular expression operators)

The completeSetup methods report errors to the ExpressionSetupErrorHandler instance, which is responsible for handling setup errors.

The ExpressionSetupErrorHandler class provides two modes for handling errors. It can throw an exception upon the first issue encountered, or it can collect all errors in a list. To use the mode that immediately throws an exception, you need not do anything.

To use the mode that creates the error list:

1. Construct an ExpressionSetupErrorHandler, setting the constructor's Boolean argument to false. Alternatively, you can use the default constructor.
2. Designate your ExpressionSetupErrorHandler instance using setErrorHandler on the ExpressionTree.

When using the error list, the following methods are available:

- `getNumberOfErrors()`
  Gets the total number of errors.
- `ExpressionSetupError* getFirstError()`
  Gets the first expression error from the list. Call getNextError() from the first ExpressionSetupError instance to get the next error.

ExpressionSetupError

The ExpressionSetupError class represents a semantic error found in the expression during complete setup. The following public APIs provide details about the error:

- `getErrorType()`
  Gets the error type. Possible error types are listed by the ooExpressionSetupErrorType enumeration.
- `getDescription()`
  Gets the description of the error.
- `Expression* getExpression()`
  Gets the expression node that has the error or null if there is no associated expression node.
- `Expression* getOperand()`
  Gets the operand that has the error if applicable.

ExpressionSetupErrorType

This enumeration includes the following error types:

oocPQLSyntaxError
: The provided PQL predicate string has syntax errors.

oocInvalidPredicateTypeError
: The result type of the PQL predicate string must be a Boolean type.

oocUnknownTypeError
: The provided type number is not found in the schema.

oocUnknownAttributeError
: The specified attribute name is not found in the schema.

oocUnknownTokenError
: An operator for the specified token name cannot be found in the operator registry.

oocOperandMismatchError
: More than one operator for the specified token name has been found, but none of them match the given operands.

oocTooFewOperandsError
: Too few operands were provided for the found operator.

oocTooManyOperandsError
: Too many operands were provided for the found operator.

oocIncompatibleOperandError
: The result type of an operand does not meet the input requirements of the found operator.

oocInvalidRegularExpressionError
: The expected regular expression operand for the regular expression operator must be a string literal or a valid regular expression.

oocInvalidClassTypeExpressionError
: The expected class type operand for this operator must be a valid class type literal.

oocOperandCrossIncompatibleError
: The result types of operands are not compatible with each other as required by the found operator. Some operators, such as ==, require that all operands have the same or compatible types.

oocElementCrossIncompatibleError
: The provided elements are not compatible with each other as required by the ordered-list literal.

oocOperatorImplementationError
: Operator implementation error. For example, the operator has set an unknown type number on one of its operands.

oocObjectLitValueIncompatibleError
: One of the provided values does not match the type of the attribute it is paired with.

oocUnknownVariableTypeError
: The specified PQL variable type is not supported. The supported types are: bool, int, uint, float, string, date, time, datetime, interval, class, and oid.

oocUndefinedVariableError
: Cannot set the value of an undefined PQL variable.

oocVariableValueNotSetError
: The value for a defined PQL variable was not set before calling CompleteSetup().

oocIncompatibleVariableValueError
: The variable value you attempted to set is not compatible with the variable type that was specified in PQL.

oocCreateLookupFieldError
: An error occurred during index lookup field construction.

ExpressionException

The ExpressionException class, which is derived from ooException, represents expression exceptions thrown from an expression tree during construction, setup or evaluation.

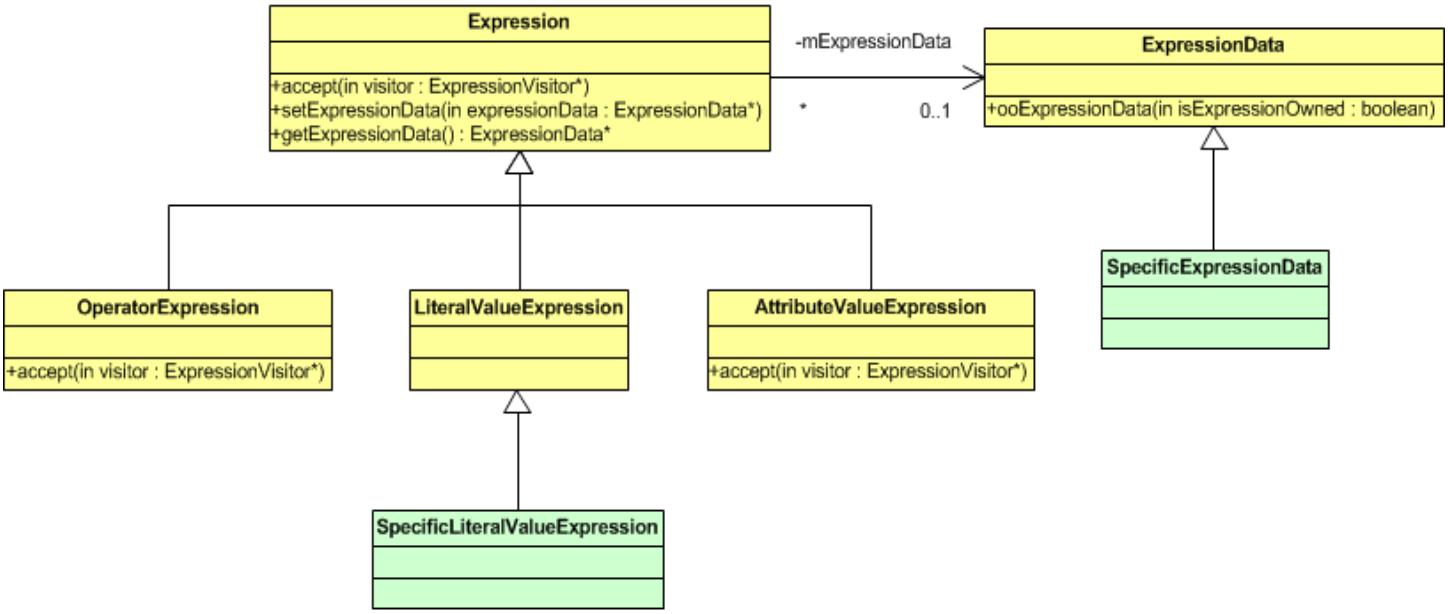
The following subclasses are derived from ExpressionException:

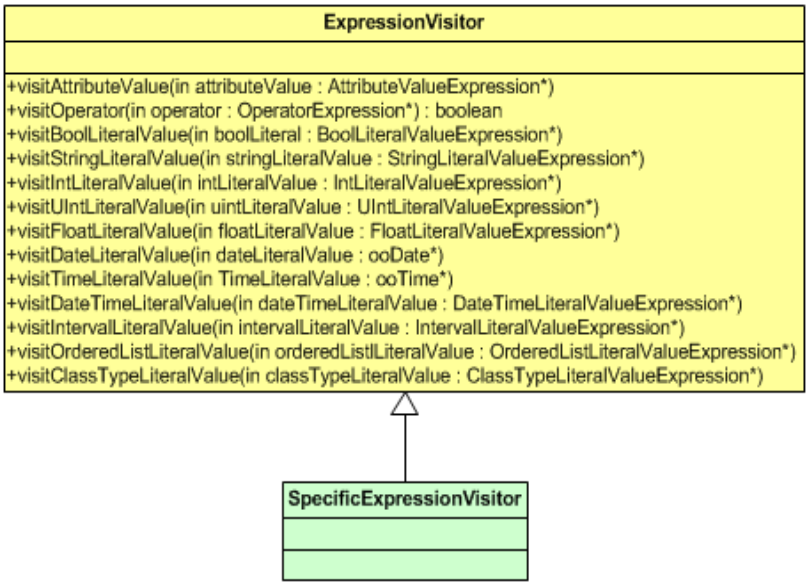
- ExpressionConstructionException
- ExpressionSetupException
- ExpressionEvaluationException

Performing Object Qualification, such as through a scan method or by using the doesQualify method on an ObjectQualifier instance, can result in an ooKernelException or an ExpressionException. An ooKernelException represents a kernel problem such as a failure to obtain a lock. ExpressionExceptions can represent problems that occurred while constructing an expression tree (ExpressionConstructionExceptions), during the complete setup of an expression tree (ExpressionSetupExceptions), or during expression evaluation (ExpressionEvaluationExceptions).

Expression Extensibility

Extensibility helper classes are provided so that end users can add their own data and methods to the expression node classes





ExpressionData

The ExpressionData class is provided as the base class for user-defined expression data classes.

The ExpressionData(Boolean isExpressionOwned) constructor takes an argument to specify whether the user-defined expression data is owned by the Expression or owned by the user application. If the user data is owned by the expression, it is the responsibility of the Expression instance to delete it when the expression is deleted.

To define your own data to be associated with an Expression node, derive a new class from ExpressionData, then define the data and access methods in this class.

The following methods set or retrieve the ExpressionData object associated with an Expression node.

- `setExpressionData(ExpressionData* expressionData)`
- `getExpressionData()`

ExpressionVisitor

The visitor pattern[1] is used to provide the extensibility for adding new operations on the expression tree structure. The visitor pattern allows you to define a new operation to be performed on elements in a class structure without changing and recompiling the classes.

The ExpressionVisitor class is the base class for user-defined expression visitor classes that add operations to be performed on the expression nodes.

In order for the expression classes to perform the user-added operation, the accept(ExpressionVisitor* visitor) method, which takes a visitor (a concrete ExpressionVisitor instance) as an argument, is provided on all expression node classes.

To add and perform a new operation on all nodes in the expression tree structure:

- Define a concrete expression-visitor class derived from ExpressionVisitor (called *specificExpressionVisitor* from here on). Implement the new operation on each type of expression by implementing all the virtual visit methods defined in ExpressionVisitor:
    - visitOperator()
    - visitAtributeValue()
    - visitIntLiteralValue()
    - visitUIntLiteralValue()
    - visitFloatLiteralValue()

[1] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series

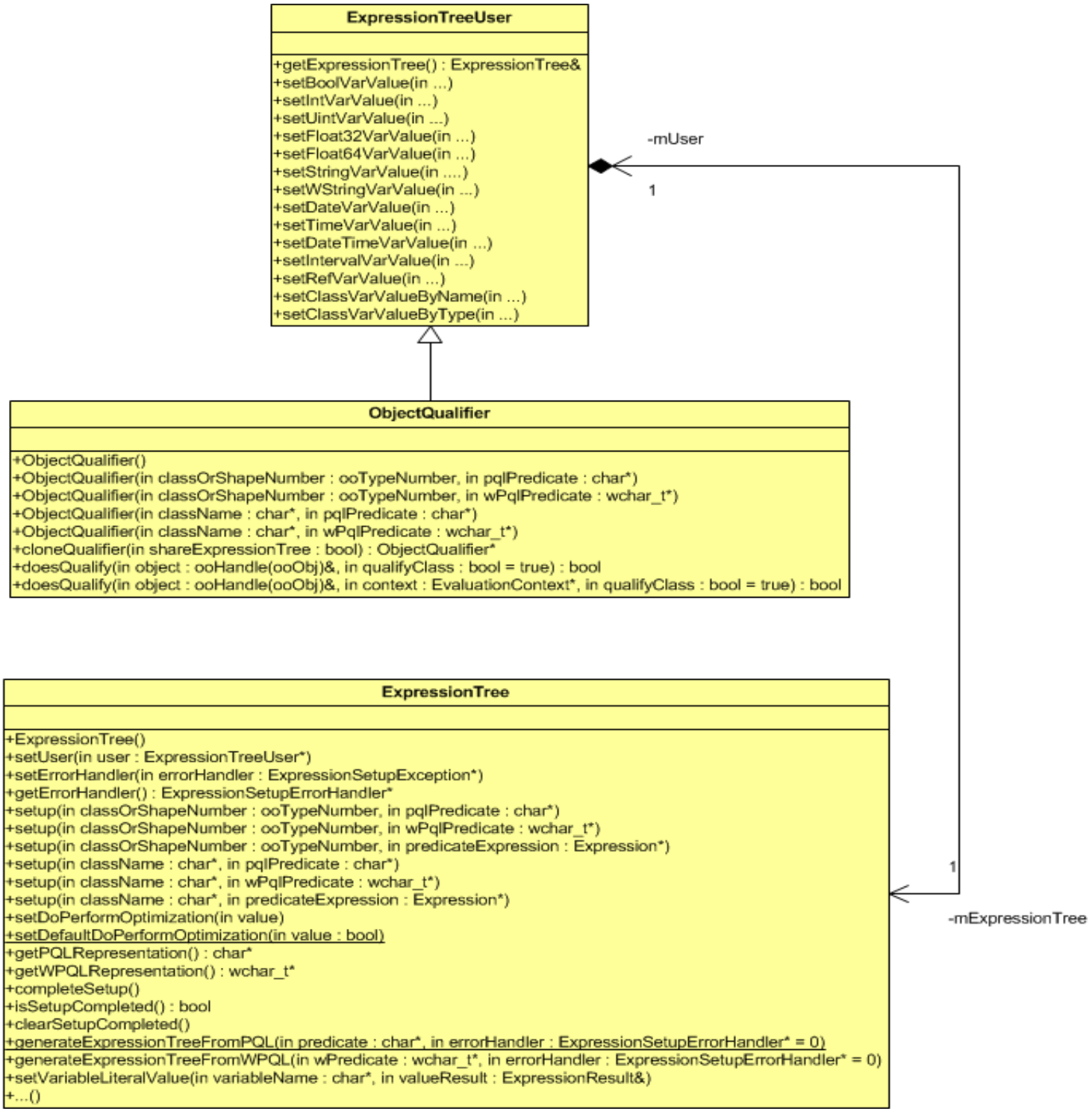
- visitStringLiteralValue()
- visitBoolLiteralValue(), and so forth.

The visitOperator() method returns a Boolean type to indicate whether or not the operation needs to be populated to its operands (to call the accept() method of its operands from the accept() method of the operator expression).

- Create an instance of the specificExpressonVisitor (referred to as *specificVisitorPtr*).
- Perform the operation implemented by specificExpressionVisitor on the expression tree nodes by calling the function accept(specificVisitorPtr) from the head node, and passing in the reference to the specific visitor instance as an argument.
  The operation will be performed on the head node first, then populated to all its operands as the operator expression's accept() method calls its operands' accept() method (if visitOperator() returns true). The whole tree will be traversed, and the new defined operation will be performed on each expression node automatically as it is visited.

Object Qualification

The following diagram shows the main abstractions for object qualification, which include the ObjectQualifier class, its base class ExpressionTreeUser, and the ExpressionTree class.



ExpressionTreeUser

ExpressionTreeUser is the base class for all classes that use an expression tree, which includes the ObjectQualifier class.

ExpressionTreeUser provides a method to get the expression tree, and provides methods to set values for any PQL variables that were defined in predicate strings.

- `ExpressionTree& getExpressionTree();`
- `void setBoolVarValue(const char* variableName, bool value);`
- `void setIntVarValue(const char* variableName, int64 value);`
- `void setUintVarValue(const char* variableName, uint64 value);`
- `void setFloat32VarValue(const char* variableName, float32 value);`
- `void setFloat64VarValue(const char* variableName, float64 value);`
- `void setStringVarValue(const char* variableName, const char* value);`
- `void setWStringVarValue(const char* variableName, const wchar_t* value);`
- `void setDateVarValue(const char* variableName, ooDate value);`
- `void setTimeVarValue(const char* variableName, ooTime value);`
- `void setDateTimeVarValue(const char* variableName, ooDateTime value);`
- `void setIntervalVarValue(const char* variableName, ooInterval value);`
- `void setRefVarValue(const char* variableName, const ooRef(ooObj)& value);`
- `void setClassVarValueByName(const char* variableName, const char* className);`
- `void setClassVarValueByType(const char* variableName, ooTypeNumber classType);`

ObjectQualifier

The ObjectQualifier class is derived from the ExpressionTreeUser class, and it is used to perform object qualification.

When you construct an object qualifier, you provide:

- The class information for the objects to be qualified.
- The predicate string for the qualifying condition, written in PQL.

A set of constructors taking a PQL predicate string either as a regular char* or a wide char* let you easily construct and set up an `ObjectQualifier`.

- `ObjectQualifier(ooTypeNumber classNumber, const char* pqlPredicate)`
- `ObjectQualifier(ooTypeNumber classNumber, const wchar_t* wPqlPredicate)`
- `ObjectQualifier(char* className, const char* pqlPredicate)`
- `ObjectQualifier(char* className, const wchar_t* wPqlPredicate)`

Each of these constructors creates an underlying predicate expression tree, which can be retrieved using the `getExpressionTree` method on the base class.

Note: You cannot construct a new object qualifier by providing a predicate expression tree. However, you can construct an empty, default object qualifier, call the above `getExpressionTree()` method to get the underlying expression tree, then set the underlying expression tree to your own, pre-built expression tree; see "Object Qualification Process" for details.

Once setup is complete, an ObjectQualifier can qualify objects one at a time. The following method performs qualification for the specified object.

```
Bool doesQualify(const ooHandle(ooObj)& object,
                 bool qualifyClass = true)
```

You can also associate an ObjectQualifier with a scan operation to perform object qualification for all objects in the federated database, for example:

```
ooStatus scan(const ooRefHandle(ooObj) &scope,
              const objy::query::ObjectQualifier& objQlfier)
```

For more information about PQL and how to perform object qualification using ObjectQualifiers, refer to the documentation for your Objectivity programming language interface.

ExpressionTree

The ExpressionTree class encapsulates a raw expression tree by holding a handle to the root expression node, which is an instance of the Expression class. The ExpressionTree class provides APIs to construct, set up, and access the expression tree.

Setup Methods

The following methods set up an expression tree with the qualifying class, specified either by class type number or class name. The predicate is specified either by the PQL predicate string or the expression tree instance.

- `void setup(ooTypeNumber classNumber, const char* pqlPredicate)`
- `void setup(ooTypeNumber classNumber, const wchar_t* wPqlPredicate)`

- `void setup(ooTypeNumber classNumber,`
  `Expression* predicateExpression)`
- `void setup(char* className,`
  `const char* pqlPredicate)`
- `void setup(char* className,`
  `const wchar_t* wPqlPredicate)`
- `void setup(char* className,`
  `Expression* predicateExpression)`

Complete Setup Methods

The setup methods implicitly call the completeSetup methods, which complete predicate expression setup by performing:

- Operator selection (if needed)
- Error detection
- Optimization

The completeSetup method can also be called explicitly after the initial setup of the expression tree. This might be needed if the expression tree has been changed or if the values for any PQL variables have been set.

Following are the completeSetup and related methods:

- `void completeSetup()`
- `bool isSetupCompleted()`
- `void clearSetupCompleted()`

Error Handler Methods

The completeSetup methods report errors to an ExpressionSetupErrorHandler instance, which is responsible for handling setup errors. The ExpressionSetupErrorHandler can either throw an exception upon the first issue encountered, or collect all errors in a list. By default, an error handler that uses the behavior of throwing an exception is used. If collecting errors into a list is the prefered behavior, you need to create an error handler (as described in ExpressionSetupErrorHandler) and set it by calling setErrorHandler().

Following are the methods related to error handling:

- `void setErrorHandler(ExpressionSetupErrorHandler* errorHandler)`
- `ExpressionSetupErrorHandler* getErrorHandler() const`

Additional Settings

The following method explicitly sets the user/owner of the expression tree (for example, an ObjectQualifier instance). The user/owner is usually set implicitly by one of the ExpressionTreeUser constructors.

- `void setUser(ExpressionTreeUser* user)`

The following methods provide additional settings that affect the behavior for expression trees. As these settings affect the functionality of completeSetup, they need to be called before setup() or completeSetup().

- `void setRequiredResult(const ExpressionResultSpec& required,`
  `const char* requiredResultTypeMessage)`
  Specifies the result that is required from the head node. For ObjectQualifiers, the required result type is set to oocQBool.

- `void setDoPerformOptimization(bool value)`
  Specifies whether or not to perform predicate optimization. The default is true, but this can be changed with the setDefaultDoPerformOptimization method.

- `static void setDefaultDoPerformOptimization(bool value)`
  Specifies whether or not to perform predicate optimization by default, which can always be overridden by calling setDoPerformOptimization on specific ExpressionTree instances. The initial value of this default is true.

Additional Accessor Methods

The following methods get information related to the class being qualified.

- `const char* getClassName() const;`
- `ooTypeNumber getShapeNumber() const;`

The following methods get the original predicate string, which was set with the setup method.

- `const char* getPredicateString() const;`
- `const wchar_t* getWPredicateString() const;`

Methods to Generate Predicate Expressions From Predicate Strings

The following static methods generate predicate expressions from predicate strings.

- ```
  static Expression*  generateExpressionTreeFromPQL(
      const char* predicate,
      ExpressionSetupErrorHandler* errorHandler=0);
  ```
  ```

- `static Expression*  generateExpressionTreeFromWPQL(`
  `const wchar_t* wPredicate,`
  `ExpressionSetupErrorHandler* errorHandler=0);`

Methods to Get Predicate Strings From Predicate Expressions

The following methods get the equivalent predicate string for the predicate expression associated with this ExpressionTree instance.

- `const char* getPQLRepresentation();`
- `const wchar_t* getWPQLRepresentation();`

The returned predicate string is generated from the ExpressionTree using the default presentation format, which might be syntactically slightly different than the original predicate string (set with the setup method). However, the original and generated predicate strings are semantically equal and correspond to the same expression tree.

Object Qualification Process

An object qualifier is used to evaluate whether a persistent object satisfies conditions specified by a given predicate. That predicate can be provided as a predicate string or as a predicate expression that you constructed programmatically.

An object qualifier can be used to test against objects of a specified class or its derived classes. The class can be specified by the type number/shape number or the class name.

The following outlines the object qualification process.

1. **Construction and Set Up**

   The first step is to construct an ObjectQualifier. You can either use an ObjectQualifier constructor that takes a predicate string, or construct an empty ObjectQualifier and subsequently provide a predicate expression.

   Provide predicate string
   When you construct an ObjectQualifier with a predicate string, the ExpressionTree setup method automatically generates the underlying expression tree.

   ```
   ObjectQualifier objQlfier(("Employee", "name == 'Bob'");
   ```

   Provide predicate expression
   In order to provide the predicate as an expression, you first construct an empty object qualifier using the default constructor. Next, get the object qualifier's ExpressionTree object and call its setup method, providing a predicate expression you previously created.

```
Expression* predicateExpression;
// Build the expression tree programmatically or assign it from an
// existing one
…

ObjectQualifier objQlfier;
ExpressionTree expTreeObj = objQlfier.getExpressionTree();
expTreeObj.setup("Employee", predicateExpression);
```

See Constructing an Expression Tree for information about building an expression tree.

2. **Expression Validation/Complete Setup**

   A predicate expression must be validated and prepared before it can be used for qualifying objects. This step is called the complete setup.

   During complete setup, the predicate expression tree is checked for errors, operators are selected for operator expression nodes (if needed), and optimization is performed (if enabled).

   The ExpressionTree's completeSetup() method performs the complete setup. Typically, you don't need to call completeSetup() explicitly because the ExpressionTree's setup() methods invoke completeSetup() implicitly.

   If the expression tree has been changed or the values for any PQL variables have been set, the completeSetup status is reset to false. If you want to revalidate the expression tree, call completeSetup() explicitly to get the error report.

   To check the setup-completion status of an ObjectQualifier object, call the isSetupCompleted() method, which returns true if the setup is complete and ready for object qualification; false otherwise.

   The setup-completed status of an ObjectQualifier object is set and maintained by its owned ExpressionTree object. The value of the status is true if the entire predicate expression is valid and complete; false otherwise. The status is initially false and only becomes true upon a successful completeSetup call. If completeSetup finishes successfully, and the predicate expression tree is modified in any way, the setup-completion status is set to false again.

   The setup-completion status is used to ensure that setup has been completed prior to use of the object qualifier for object qualification. If setup has not been completed prior to a call to doesQualify() or prior to a scan operation that uses the object qualifier, that call will implicitly call completeSetup() again.

3. **Object Qualification**

Once an object qualifier is instantiated, its underlying predicate expression tree is constructed, and setup is complete, that object qualifier is ready for use. The object qualifier can be associated with a scan operation that performs object qualification for all objects in the federated database, or it can be used to qualify individual objects using its doesQualify method.

The `doesQualify(ooHandle(ooObj)& object, bool qualifyClass = false)` method qualifies each passed-in object against the expression tree.

This method takes the following arguments:

- A reference to the object as the context in which the expression is being evaluated
- A Boolean indicating whether the object's class should be qualified.

The doesQualify() method internally calls evaluate() on the head node. If the head node is an operator expression, it will obtain its result from its operator, which bases its result on its algorithm and the evaluate() calls on its operands. The evaluate() method qualifies each passed-in object against the operator expression for that operator.

This way the expression tree is evaluated as necessary to obtain a result. Note that because some operators may employ optimization techniques such as short-circuit Boolean, the entire tree may or may not be visited. The doesQualify() method will return true if the passed-in object qualifies according to the object's class and optional predicate expression tree; false otherwise.

If setup was not completed successfully, the doesQualify() method will implicitly call completeSetup() and (in the default mode) throw an exception if an error occurs. Accordingly, it is recommend that you always put doesQualify() inside a try/catch block.

Note: Any scan operation that uses an object qualifier employs calls to doesQualify in its implementation, so the same evaluation of the expression tree occurs in this context as well.

Error Handling

The completeSetup method reports errors to the ExpressionSetupErrorHandler instance associated with the ExpressionTree.

By default, an internal ExpressionSetupErrorHandler that throws an exception on encountering the first error is used.

If you want to get a list of all errors instead, you must construct an error handler (the default constructor chooses the error list mode) and designate this handler by calling setErrorHandler() from the ExpressionTree before setup is performed.

For example:

```
ObjectQualifier objQlfier;
ExpressionSetupErrorHandler errorHdlr;

ExpressionTree expTreeObj = objQlfier.getExpressionTree();
expTreeObj.setErrorHandler(&errorHdlr);

expTreeObj.setup("Employee", "age > 40");
```

Tasks

This section describes the high level tasks needed to create and use an expression tree for object qualification. These tasks include:

- Constructing an Expression Tree
- Setting Up an Object Qualifier with a Predicate String
- Performing Complete Setup and Error Handling

Constructing an Expression Tree

This section explains how to construct an expression tree programmatically using the provided API. The expression tree is built up by constructing the expression nodes. There are three categories of expression nodes: literal value expression, attribute value expression, and operator expression.

Create a Literal Value Expression

- **Create a Boolean literal value expression:**

```
BoolLiteralValueExpression* boolLitValueExp =
   new BoolLiteralValueExpression(true);
```

- **Create a string literal value expression:**

```
StringLiteralValueExpression* stringLitValueExp =
   new StringLiteralValueExpression("Mark");
```

Note: A string literal can be initialized with a const char* or a const wchar_t*. The latter is required if the literal is to be used to compare to a Unicode value, otherwise, an oocStringMismatchError will be reported to the error handler during complete setup.

- **Create an integer literal value expression:**

```
IntLiteralValueExpression* intLitValueExp =
   new IntLiteralValueExpression(-20);
```

- **Create an unsigned integer literal value expression:**

```
UIntLiteralValueExpression* uintLitValueExp =
   new UIntLiteralValueExpression(20);
```

- **Create a float literal value expression:**

```
FloatLiteralValueExpression* floatLitValueExp =
   new FloatLiteralValueExpression(6.5);
```

- **Create a date literal value expression:**

```
ooDate today = ooDate::today();
DateLiteralValueExpression* dateLitValueExp =
   new DateLiteralValueExpression(today);
```

- **Create a time literal value expression:**

```
ooTime timeOfDay = ooTime::utcNow();
TimeLiteralValueExpression* timeLitValueExp =
   new TimeLiteralValueExpression(timeOfDay);
```

- **Create a datetime literal value expression:**

```
ooDateTime now = ooDateTime::utcNow();
DateTimeLiteralValueExpression* dateTimeLitValueExp =
   new DateTimeLiteralValueExpression(now);
```

- **Create an interval literal value expression:**

```
ooInterval oneHour(1, 0, 0, 0);
IntervalLiteralValueExpression* intervalLitValueExp =
   new IntervalLiteralValueExpression(oneHour);
```

- **Create an ordered list literal value expression:**

```
OrderedListLiteralValueExpression* primaryColors =
   new OrderedListLiteralValueExpression();
ExpressionResult red;
red.set8BitStringResult("Red");
primaryColors->addValue(red);
ExpressionResult green;
green.set8BitStringResult("Green");
primaryColors->addValue(green);
ExpressionResult blue;
blue.set8BitStringResult("Blue");
primaryColors->addValue(blue);
```

Create an AttributeValueExpression

An AttributeValueExpression can be used to express member attributes of basic types, reference, or query collection types.

```
AttributeValueExpression* employer =
   new AttributeValueExpression("employer");
AttributeValueExpression* employees =
   new AttributeValueExpression("employees");
```

Create an Operator Expression

- **Create an operator expression with a token name:**

  ```
  OperatorExpression* andOperatorExp =
     new OperatorExpression("and");
  ```

  Once the token name is provided to an OperatorExpression instance, the corresponding operator reference will be found from the registered operator groups and set automatically by the completeSetup() method.

- **Create an operator expression without a token name**:

  ```
  OperatorExpression* operatorExp = new OperatorExpression();
  ```

  In this case, the operator reference associated with this OperatorExpression instance must be set manually.

  ```
  // Find all matching operators in the global operator registry
  OperatorGroup* matching =
     OperatorGroupRegistry::getMatchingOperators("AND");

  if(matching != 0)
  {
     // Further select the right operator if more than one is returned
     // (here just select the first matching one)
     const Operator* myOperator = matching->getOperator(0);
     // Set the operator reference to the ooOperatorExpression
     operatorExp->setOperator(myOperator);
  }
  ```

Add operands to an Operator Expression

Any expression node can serve as an operand for an operator expression node. Each operator expression has one or more operands maintained by OperandList.

To add an operand to an OperandList, use the following API:

```
OperandList::addOperand(Expression* operand,
   const Expression* before = 0)

OperandList::addOperand(Expression* operand, unsigned position)
```

The operands list maintained by ooOperandList is implemented as an array. You can specify that an operand should be placed before another operand in the list, otherwise it is added at the end of the list. You can also specify the exact position in the list in which to add the new operand.

An operator expression is considered completed when the operator expression instance is created and all its operands have been added. The expression tree is completely constructed when all its operator expression nodes are completed.

Note: Once an expression node instance has been added to the expression tree, it cannot be shared by adding it again to the same or to another expression tree. To move an attached expression node to a different position in the tree, detach it by calling Expression::setAttached(false) first. Then re-add it to the expression tree in the new position.

The following examples use addOperand() to construct an operator expression.

Example 1:

```
//================= Construct the expression tree ========================//
//                     (name=~"B.*") and (age==20)                        //
//                                                                        //
//                                and                                     //
//                              /       \                                 //
//                           =~           ==                            //
//                         /      \       /   \                          //
//                      name    "B.*"   age   20                          //
//========================================================================//


// Create the head node: AND operator
OperatorExpression* headExp = new OperatorExpression("AND");

//Construct operand1 for "AND": name =~ "B.*"
OperatorExpression* regExEqOprtExp = new OperatorExpression("=~");
headExp->getOperandList().addOperand(regExEqOprtExp);

AttributeValueExpression* name = new AttributeValueExpression("name");
StringLiteralValueExpression* pattern =
   new StringLiteralValueExpression("B.*");
regExEqOprtExp->getOperandList().addOperand(name);
regExEqOprtExp->getOperandList().addOperand(pattern);

// Construct operand2 for "AND": age == 20
OperatorExpression* equalOprtExp = new OperatorExpression("==");
headExp->getOperandList().addOperand(equalOprtExp);

AttributeValueExpression* age = new AttributeValueExpression("age");
UIntLiteralValueExpression* ageValue =
   new UIntLiteralValueExpression(20);
equalOprtExp->getOperandList().addOperand(age);
equalOprtExp->getOperandList().addOperand(ageValue);
```

Example 2:

```
//================= Construct the expression tree ========================//
//                    employees[3].name == "Mary"                         //
//                                                                        //
//                                 ==                                     //
//                               /    \                                   //
//                             .       "Mary"                             //
//                           /   \                                        //
//                        []     name                                     //
//                      /   \                                             //
//             employees    3                                             //
//========================================================================//


// Create the head node: "==" operator
OperatorExpression* headExp = new OperatorExpression("==");

// Construct the operand1 for "==": employees[3].name
OperatorExpression* dotOprtExp = new OperatorExpression(".");
headExp->getOperandList().addOperand(dotOprtExp);

// Construct the operand1 for . operator: employees[3]
OperatorExpression* indexValueOprtExp = new OperatorExpression("[]");
dotOprtExp->getOperandList().addOperand(indexValueOprtExp);

AttributeValueExpression* employees = new
   AttributeValueExpression("employees");
UIntLiteralValueExpression* indexValue = new
   UIntLiteralValueExpression(3);
indexValueOprtExp->getOperandList().addOperand(employees);
indexValueOprtExp->getOperandList().addOperand(indexValue);

// Construct the operand2 for . operator: name
AttributeValueExpression* name = new AttributeValueExpression("name");
dotOprtExp->getOperandList().addOperand(name);

// Construct the operand2 for "==": string literal for "Mary"
StringLiteralValueExpression*  strLitValue = new
   StringLiteralValueExpression("Mary");
headExp->getOperandList().addOperand(strLitValue);
```

Example 3:

```
//=================== Construct the expression tree ======================//
//                        all(employees, age > 20)                        //
//                                                                        //
//                                all                                     //
//                              /      \                                  //
//                        employees     >                                 //
//                                    /   \                               //
//                                  age    20                             //
//========================================================================//

// Create the head node: all operator
OperatorExpression* headExp = new OperatorExpression("all");

// Construct operand1 for all operator: employees
AttributeValueExpression*  employees =
   new AttributeValueExpression("employees");
headExp->getOperandList().addOperand(employees);

// Construct operand2 for all operator: predicate condition "age > 20"
OperatorExpression* GTOprtExp = new OperatorExpression(">");
headExp->getOperandList().addOperand(GTOprtExp);

AttributeValueExpression* age = new AttributeValueExpression("age");
UIntLiteralValueExpression* ageValue = new
   UIntLiteralValueExpression(20);
GTOprtExp->getOperandList().addOperand(age);
GTOprtExp->getOperandList().addOperand(ageValue);
```

Example 4:

```
//=================== Construct the expression tree ======================//
//  any(schools, (name == "Jefferson") && any(students, name == "kathy")) //
//                                                                        //
//             any                                                        //
//          /       \                                                     //
//      schools       &&                                                  //
//                 /          \                                           //
//              ==              any                                       //
//           /      \         /      \                                    //
//       name "Jefferson"  students   ==                                  //
//                                  /    \                                //
//                               name   "kathy"                           //
//========================================================================//

// Create the head node: any operator for "schools"
OperatorExpression* headExp = new OperatorExpression("any");

// Construct operand1 for headExp: schools (under class context: City)
AttributeValueExpression*  schoolList = new
   AttributeValueExpression("schools");
headExp->getOperandList().addOperand(schoolList);

// Construct operand2 for headExp: (name=="Jefferson") &&
// any(students,name=="kathy")(under class context: School)

OperatorExpression* andOptrExp = new OperatorExpression("&&");
headExp->getOperandList().addOperand(andOptrExp);
```

```
// Construct operand1 for andOprtExp: name == "Jefferson"
OperatorExpression* EQOprtExp = new OperatorExpression("==");
andOptrExp->getOperandList().addOperand(EQOprtExp);

AttributeValueExpression* schoolName = new
   AttributeValueExpression("name");
StringLiteralValueExpression*  schoolNameValue = new
   StringLiteralValueExpression("Jefferson");
EQOprtExp->getOperandList().addOperand(schoolName);
EQOprtExp->getOperandList().addOperand(schoolNameValue);

// Construct operand2 for andOprtExp: any(students, name=="kathy")
// (under class context: School)
OperatorExpression* anyStudentsOprtExp = new OperatorExpression("any");
andOptrExp->getOperandList().addOperand(anyStudentsOprtExp);

// Construct operand1 for anyStudentsOprtExp: students
// (under class context: School)
AttributeValueExpression*  studentList = new
   AttributeValueExpression("students");
anyStudentsOprtExp->getOperandList().addOperand(studentList);

// Construct operand2 for anyStudentOprtExp: "name == "kathy"
// (under class context: Student)
EQOprtExp = new OperatorExpression("==");
anyStudentsOprtExp->getOperandList().addOperand(EQOprtExp);

AttributeValueExpression* studentName = new
   AttributeValueExpression("name");
StringLiteralValueExpression*  studentNameValue = new
   StringLiteralValueExpression("kathy");
EQOprtExp->getOperandList().addOperand(studentName);
EQOprtExp->getOperandList().addOperand(studentNameValue);
```

Modify the Expression Tree

The expression tree can be modified with the following ooOperandList APIs:

- `addOperand (Expression* operand, const Expression* before = 0)`

  Adds the specified operand to the enclosing operator expression before the specified operand or at the end if *before* is 0.

- `addOperand(Expression* operand, unsigned position)`

  Adds the specified operand to the enclosing operator expression at the specified position. The list is expanded if needed. If an operand exists at the specified position, it and all subsequent operands are moved down in the list by one.

- `setOperand(Expression* operand, unsigned position)`

  Sets the operand on the specified position. The list is expanded if needed. If an operand exists at the specified position, it is replaced.

- `removeOperand(Expression* operand)`

  Removes an operand from the operands list.

  Note: The caller is responsible for deleting the removed operand.

Setting Up an Object Qualifier with the Predicate Expression

There is no ObjectQualifier constructor that takes a predicate expression directly. To set up a predicate expression for an object qualifier, you need to first get the object qualifier's ExpressionTree. You then call the ExpressionTree's setup method and provide your pre-built predicate expression.

For example,

```
ooTypeNumber shapeNumber = ooTypeN(Person);
ooOperatorExpression* headExp;

// Construct the expression tree
...

try {
   ObjectQualifier objectQualifier;
   ExpressionTree& expressionTree = objectQualifier.getExpressionTree();
   expressionTree.setup(shapeNumber, headExp);
 }
 catch (ExpressionException &ex)
 {
   ex.reportErrors();
 }
```

Performing Complete Setup and Error Handling

Recall that expression tree set-up errors can result from an explicit call to the ExpressionTree completeSetup method, or from implicit calls to completeSetup triggered by a call to one of the setup methods.

There are two ways to access expression tree set-up errors. By default, an exception is thrown on encountering the first error. If you want to receive a list of errors, construct an ExpressionSetupErrorHandler and set this handler on the ExpressionTree before calling setup.

The following example shows how to get an error list and access information about the errors.

```
// Define an error handler
ExpressionSetupErrorHandler errorHandler;

ObjectQualifier objectQualifier;
ExpressionTree& expressionTree = objectQualifier.getExpressionTree();

// Set the error handler
expressionTree.setErrorHandler(&errorHandler);
expressionTree.setup(shapeNumber, headExp);
```

```
// Iterate the errors list
if (errorHandler.getNumberOfErrors() != 0)
{
    const ExpressionSetupError* expError = errorHandler.getFirstError();

    while (expError != 0)
    {
      // get the error description
      const char* errorDescrip = expError->getDescription().c_str();
      cout << "Expression Error:" << errorDescrip << endl;

      // get the next error
      expError = expError->getNextError();
    }
 }
```

Reference Specifications

The following classes provide the expression tree infrastructure.

- ExpressionTreeUser
- ObjectQualifier
- ExpressionTree
- DataType
- ExpressionResultSpec
- ExpressionResult
- QCollection
- QCollectionIterator
- ExpressionType
- Expression
- OperatorExpression
- NameList
- OperandList
- AttributeValueExpression
- ooLiteralValueExpression
- BoolLiteralValueExpression
- StringLiteralValueExpression
- IntLiteralValueExpression
- UIntLiteralValueExpression
- FloatLiteralValueExpression
- DateLiteralValueExpression
- TimeLiteralValueExpression
- DateTimeLiteralValueExpression
- IntervalLiteralValueExpression
- OrderedListLiteralValueExpression
- ClassTypeLiteralValueExpression
- CompleteSetupContext
- Operator

- OperatorGroup
- OperatorGroupRegistry
- ExpressionData
- ExpressionVisitor
- ExpressionSetupErrorType
- ExpressionSetupError
- ExpressionSetupErrorHandler
- ExpressionException
- ExpressionConstructionException
- ExpressionSetupException
- ExpressionEvaluationException

Refer to the public header files in *installDir*/include/obj/query for details.

Packaging and Deployment

Include Files:

- obj/query/ExpressionTree.h,
- obj/query/ObjectQualifier.h

For applications using the expression tree interface, the new ObjectQualifier.h header file must be included. On UNIX, applications need to link to libooObjectQualification.so (or the library with the appropriate extension for the particular UNIX platform) using the -looObjectQualification link option. On Windows, the ooObjectQualification.dll is linked automatically when you include the ObjectQualifier.h header file.

For deploying an application that uses the expression tree interface, the following libraries must be packaged and deployed:

On Windows
- ooObjectQualification.dll
- ooObjectModel.dll
- ooSchemaModelInt.dll
- oopcre.dll
- ooas100.dll

On UNIX
- libooObjectQualification.so
- libooObjectModel.so
- libooSchemaModelInt.so
- libPCRE.so
- liboo_as.so

Glossary

- Complete setup
  Phase in which the predicate expression is checked for errors, operators are selected for operator expression nodes, and optimization is performed (if enabled). Complete setup can be implicitly triggered by calling setup() methods on the expression tree instance, or can be explicitly called using completeSetup() on the expression tree instance.
- Predicate expression tree
  A tree data structure whose components and structure represent a predicate used to qualify persistent objects.
- Object qualifier
  Instance of the ObjectQualifier class, which is used to find persistent objects that meet some condition according to the values of one or more of their attributes.
- Object qualification
  A means of modifying a search operation to find persistent objects that meet some condition according to the values of one or more of their attributes.
- PQL
  Objectivity query language.
- PQL variable
  A variable expression that lets you substitute different literal values into a predicate string used in an object qualifier.
- Predicate expression
  A tree data structure that expresses a condition for object qualification. When tested against a candidate object and its attribute values, the predicate expression evaluates to true or false.
- Predicate string
  A string in the Objectivity/DB predicate-query language that expresses a condition for object qualification. When tested against a candidate object and its attribute values, the predicate string evaluates to true or false.

Related Documentation

- The *Objectivity/DB Predicate Query Language* manual in your Objectivity documentation.
- The *Custom Operator Support* advanced topic, which can be found on this website.