Objectivity/DB
Predicate Query Language

Release 12.0

Objectivity/DB Predicate Query Language

Part Number: 12.0-QL-0

Release 12.0, August 22, 2016

The information in this document is subject to change without notice. Objectivity, Inc. assumes no responsibility for any errors that may appear in this document.

Contents

About This Book

This book describes how to use the predicate query language (PQL) to create strings that can be used to:

- Qualify persistent objects according to the values of one or more of their attributes.
- Qualify paths and perform filtering in the context of a navigation query across a graph of related objects.

Audience

This book is for Objectivity for Java or Objectivity/C++ application developers.

Organization

- Chapter 1 defines the uses for PQL and discusses the basic building blocks for PQL expressions.
- Chapter 2 explains how to qualify persistent objects based on their attribute values and relationships.
- Chapter 3 explains how to use PQL in a navigation query across a graph of related objects.
- Chapter 4 includes the complete reference documentation for PQL and its operators.
- Appendix A provides supplemental information for using PQL with the Objectivity/C++ programming interface.
- Appendix B provides supplemental information for using PQL with the Objectivity for Java programming interface.

Conventions and Abbreviations

Navigation

In the online version of this book, table of contents entries, index entries, cross-references, and underlined text are hypertext links.

Typographical Conventions

`cd`	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
`installDir`	Variable element (such as a filename or a parameter) for which you must substitute a value
Browse FD	Graphical user-interface label for a menu item or button
lock server	New term, book title, or emphasized word

Abbreviations

(administration)	Feature intended for database administration tasks
(HA)	Feature of the Objectivity/DB High Availability product
(*ODMG*)	Feature conforming to the Object Database Management Group interface

Command Syntax Symbols

[...]	Optional item. You may either enter or omit the enclosed item.
{…}	Item that can be repeated.
...|...	Alternative items. You should enter only one of the items separated by this symbol.
(…)	Logical group of items. The parentheses themselves are not part of the command syntax; do not type them.

Command and Code Conventions

In code examples or commands, the continuation of a long line is indented. Omitted code is indicated with the ellipsis (…) symbol. "Enter" refers to the standard key (labeled either Enter or Return) for terminating a line of input.

Getting Help

The Objectivity Developer Network provides technical information and resources, such as tutorials, documentation, FAQs, code examples, and sample applications. You can also access information about supported platforms and compilers. The Developer Network is found at:

http://support.objectivity.com

You can log onto your existing customer account from this location. Your customer account gives you access to product downloads and information about known bugs and bug fixes. Contact Customer Support if you need a login.

How to Reach Objectivity Customer Support

You can contact Customer Support by:

- **Telephone:** Call 1.408.992.7100 *or* 1.800.SOS.OBJY (1.800.767.6259) Monday through Friday between 6:00 A.M. and 6:00 P.M. Pacific Time, and ask for Customer Support.

  The toll-free 800 number can be dialed *only* within the 48 contiguous states of the United States and Canada.
- **Fax:** Send a fax to Objectivity at 1.408.992.7171.
- **Email:** Send electronic mail to *help@objectivity.com*.

Before You Call

Please be ready to submit the following information:

- Your name, company name, address, telephone number, fax number, and email address
- Detailed description of the problem
- Information about your workstation, including the type of workstation, its operating system version, and compiler or interpreter
- Information about your Objectivity products, including the version of the Objectivity/DB libraries

You can use the Objectivity/DB `oosupportinfo` tool to obtain information about your workstation and your Objectivity products.

1

Getting Started

The *predicate query language* (PQL) is useful for qualifying objects or qualifying the paths that connect related objects in your federated database.

PQL provides a set of built-in *operators* that perform arithmetic, relational, logical, path, and other comparison operations. Together these operators and their *operands* are used to form *operator expressions*.

The operands can be:

- *Attribute expressions*, which refer to attributes of persistent objects; see "Attribute Expressions" on page 82.
- *Literal expressions*, which define constants; see "Literal Expressions" on page 86.
- *Variable expressions*, which let you substitute different literal values into a predicate string used in an object qualifier; see "Variable Expressions" on page 87.

PQL's operators and expressions can be combined and nested to create complex expressions that offer a wide range of functionality. For detailed information about PQL and its syntax, see Chapter 4, "PQL Reference."

NOTE As a query language, PQL can only refer to attributes of persistent objects in an Objectivity/DB federated database. It does not provide the ability to modify the state of objects or call the methods of the objects being qualified.

Use Models

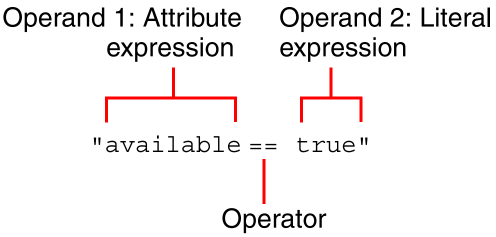
PQL currently supports two use models:

- *Object qualification*, which is the process of determining whether a persistent object satisfies a set of conditions expressed in a *predicate string*.
- (C++ only) *Navigation path qualification*, which is the process of determining whether a path connecting related objects satisfies a set of conditions, also expressed in a predicate string.

Object Qualification

PQL is most commonly used for object qualification, which is typically performed as part of a search operation that finds a group of persistent objects based on the values of one or more of their attributes.

For example, suppose you have a user-defined class called Vehicle, and each Vehicle instance has a Boolean attribute that indicates whether the vehicle is currently available. The following predicate string qualifies an available vehicle instance.


Operand 1: Attribute expression
Operand 2: Literal expression
"available == true"
Operator

For more information; see Chapter 2, "Object Qualification."

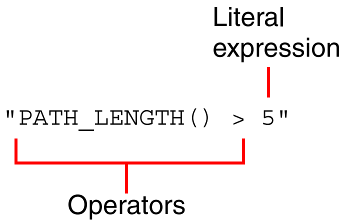
Navigation-Path Qualification

PQL can be used in the context of a *navigation query*, in which you traverse a *graph* of related objects (also known as *vertices*), searching for paths to any number of target vertices that meet certain criteria. The relationships in the graph (also known as edges) can be represented by reference attributes, collections, variable-size arrays (VArrays), name maps, and associations.

NOTE Navigation queries are available only in Objectivity/C++.

When working with navigation queries, you can use *navigation path qualification* to discontinue traversal on encountering a path with a certain composition or length. You can also perform filtering so that paths that must cross over a certain type of vertex or edge are excluded from the traversal.

Finally, you can further qualify a found target object by some characteristics of the path that led to up to it. For example, the following predicate string can be used as part of a larger navigation query that qualifies paths whose length is greater than five:



```
"PATH_LENGTH() > 5"
```

For more information; see Chapter 3, “Navigation-Path Qualification.”

2

Object Qualification

Object qualification is the process of determining whether a persistent object satisfies a set of specified conditions, and is typically performed as part of a search operation that finds a group of persistent objects based on the values of their attributes and relationships.

This chapter describes:

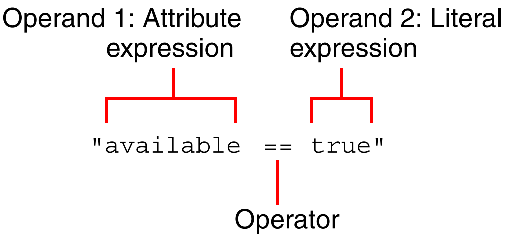
- Understanding Object Qualification
- Example: Qualifying User-Defined Objects
- Example: Object Qualification Code

NOTE For information about using indexes to optimize certain predicate scans, see the indexing chapter in the documentation for your programming interface.

Understanding Object Qualification

You use *object qualification* to determine whether a persistent object satisfies a set of specified conditions, which are expressed as a *predicate string*. When tested against a candidate object and its attribute values, the predicate string evaluates to `true` or `false`. If the predicate string evaluates to `true`, the object is considered qualified.

For example, suppose you have a user-defined Vehicle class with a Boolean attribute that indicates whether the vehicle is currently available. The following predicate string qualifies an available vehicle.


Operand 1: Attribute expression
Operand 2: Literal expression
"available == true"
Operator

NOTE As a query language, PQL can only refer to attributes of persistent objects. It does not provide the ability to modify the state of objects or call the methods of the objects being qualified.

For detailed information about PQL and its syntax, see Chapter 4, "PQL Reference."

Predicate Queries

Object qualification is typically performed as part of a search operation that finds a group of persistent objects based on the values of one or more of their attributes. Object qualification evaluates each candidate persistent object against the predicate string, and selects only the objects with attribute *values* and *relationships* that meet those conditions.

You can use predicate strings in the following search operations, which are collectively called *predicate queries*:

Search Operation	For More information
Scanning a federated database	See "Scanning a Federated Database" on page 26.
Testing a single object with an *object qualifier*	See "Qualifying Objects One at a Time" on page 27.
Finding destination objects linked by a to-many relationship	See the chapter about creating and following links in the documentation for your programming interface.
Finding destination objects using a *parallel query*	See the chapter about parallel queries in the documentation for your programming interface.
Finding destination objects in a graph of related objects using a *navigation query* (C++ only)	See Chapter 19, "Navigation Queries," in the *Objectivity/C++ Programmer's Guide*.

NOTE In the documentation, associations in Objectivity/C++ and relationships in Objectivity for Java are often referred to generically as *relationship*s.

Example: Qualifying User-Defined Objects

This section provides an example that shows how to use PQL to qualify objects in a federated database with user-defined classes. The example is based on a rental car company with vehicles for rent.

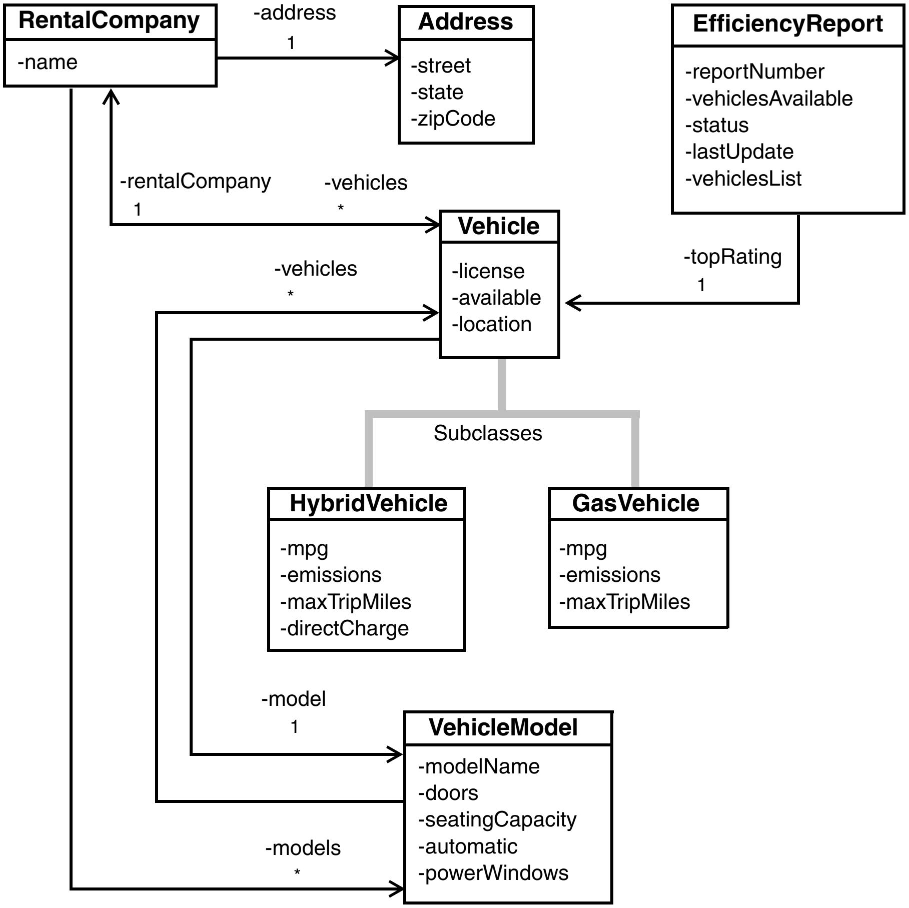
The example demonstrates how to qualify an object based on the value of an attribute, or how to qualify an object based on the values of the attributes of its related objects. Various other query techniques are also demonstrated.

Schema Model

The schema model for the example specifies that:

- A car rental company has a name, an address, a bidirectional to-many relationship to vehicles, and an array of references to vehicle models.
- Two types of vehicles are available for rent, hybrid and standard gas vehicles.
- A vehicle has a license string, an indicator of its availability, a location code, a reference to a model type, and an inverse relationship to a rental company.
- A vehicle model has several attributes including an array of references to vehicles of its model type.
- An efficiency report has several attributes including a report number and the date it was last updated. The report also has a reference to the top-rated vehicle.

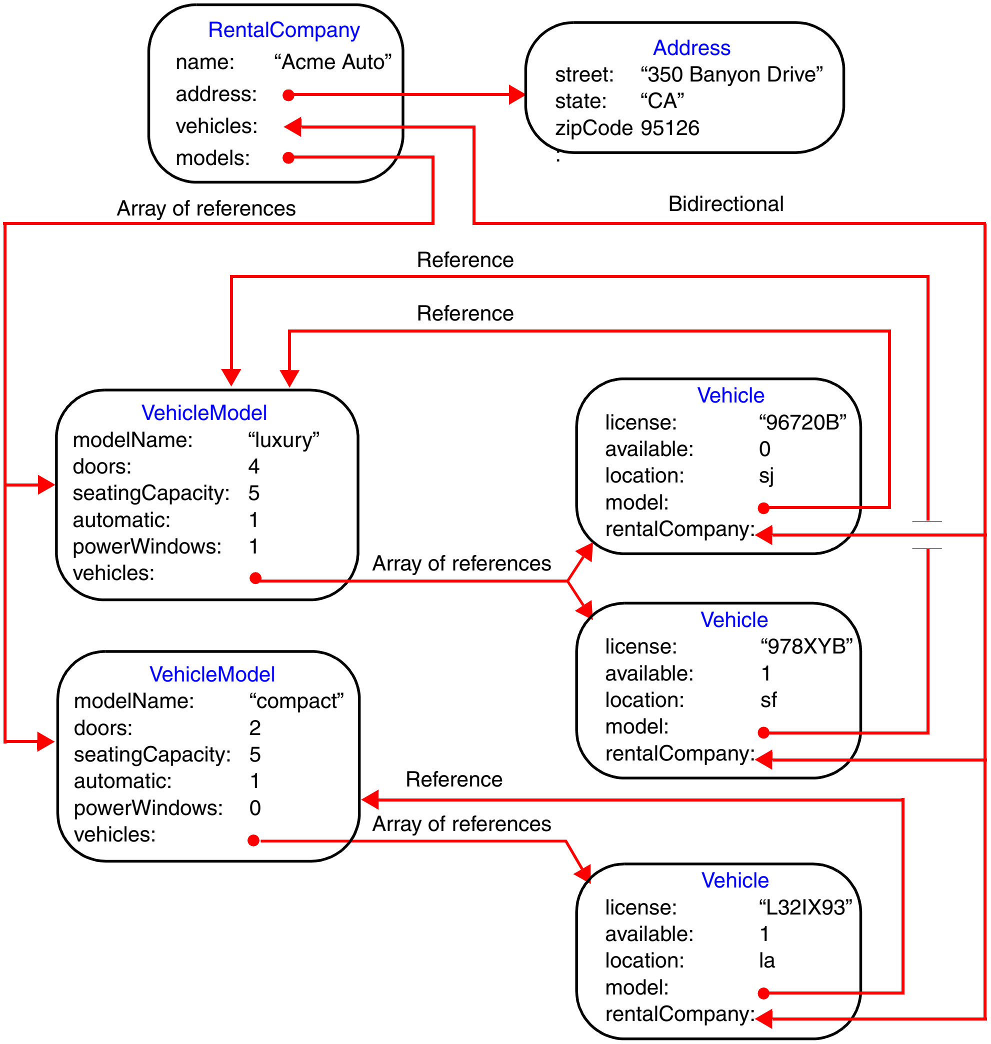
The following shows the schema model for the classes used in the example.



The data types for the attributes depends on your programming interface; see Appendix A, "C++ Examples" and Appendix B, "Java Examples" for details.

Sample Data

Given the schema model, an application could create a rental company called Acme Auto that carries two models of cars and a fleet of vehicles, each of which is either a luxury or a compact model.

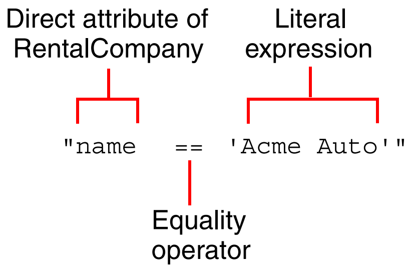


Qualifying Objects

This section shows how to qualify various objects in the sample data. The examples qualify objects using *direct attributes* of the object being qualified, and *indirect attributes*, which are attributes on objects related to the object being qualified.

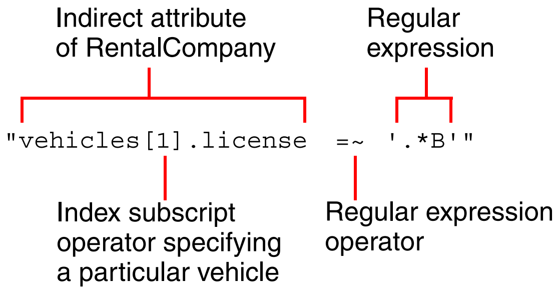
Using Direct Attributes

The following predicate string qualifies the rental company instance with the name `Acme Auto`.



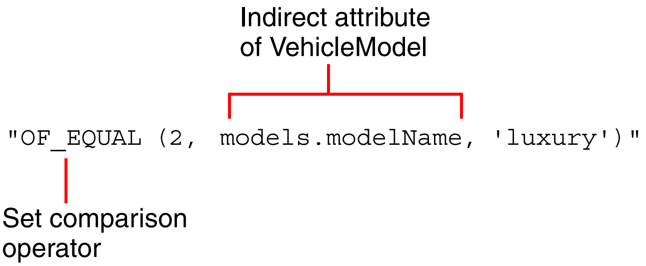
Using Indirect Attributes

The following predicate string qualifies a rental company that has a particular vehicle whose license string ends with the letter `B`.


Indirect attribute of RentalCompany
Regular expression
"vehicles[1].license =~ '.*B'"
Index subscript operator specifying a particular vehicle
Regular expression operator

In this predicate string, `license` belongs to a `Vehicle` object to which `RentalCompany` has a bidirectional relationship.

The following string qualifies a rental company that has at least two models whose model name is `luxury`. Accordingly, there is no qualifying object.

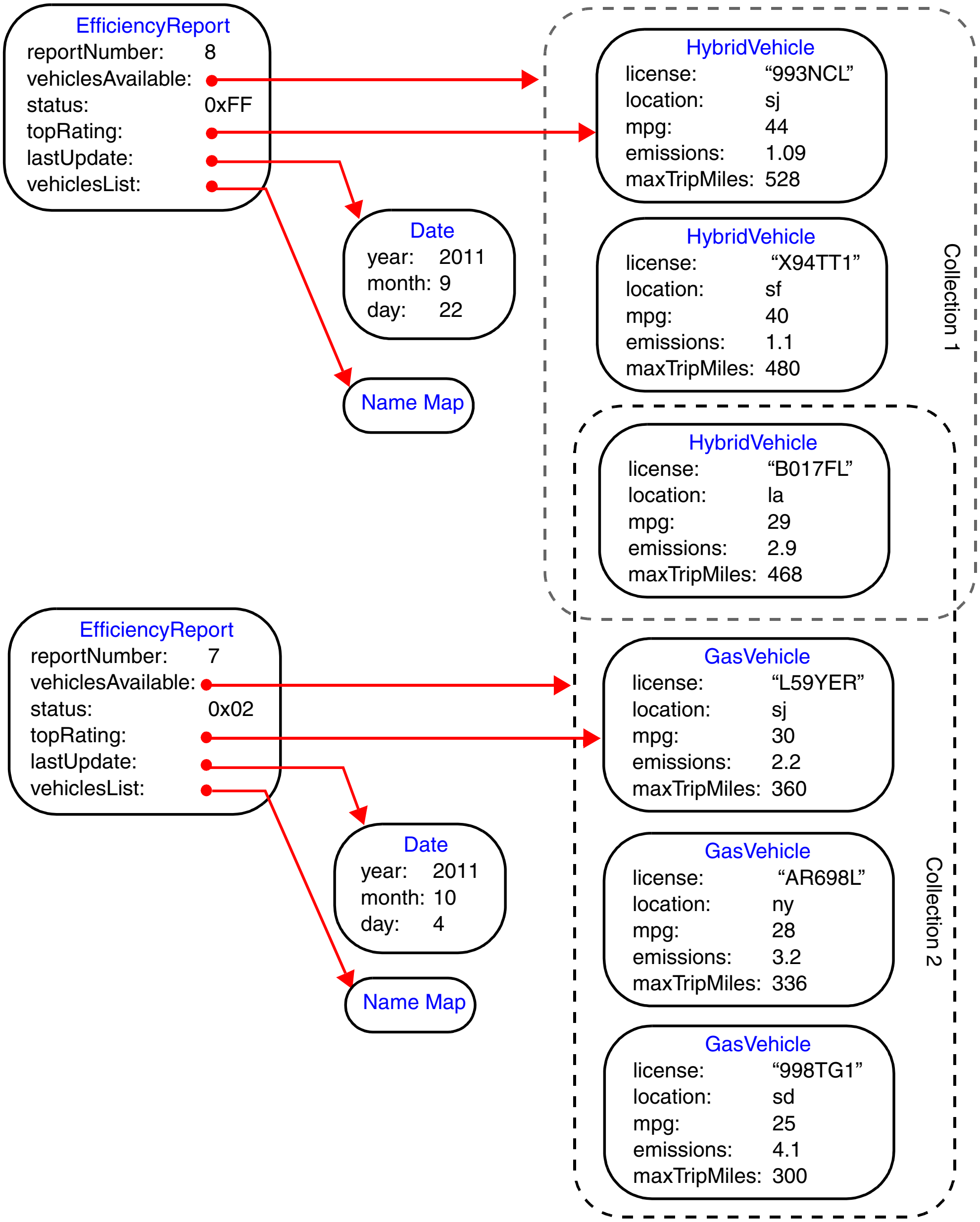


```
"OF_EQUAL (2, models.modelName, 'luxury')"
```

The expression `models.modelName` evaluates to a temporary, internal *multi-element* object holding all the model names for every model instance found. The `OF_EQUAL` operator compares those names to the literal string `luxury`.

Extended Sample Data

This section provides more sample data in order to demonstrate the use of type-evaluation and type-access operators. Given the same schema model, an application could create two efficiency reports, each with a top-rated vehicle and a collection of available vehicles.

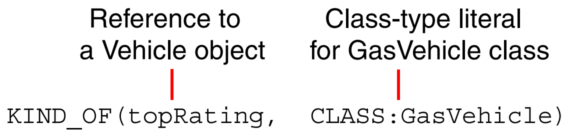


Qualifying Objects Using Type Evaluation and Casting

This section shows how to qualify objects based on their class type, and also shows how to cast a referenced object or a referenced collection of objects to a given class type in order to access attributes of the casted objects.

Qualifying by Class Type

When qualifying an `EfficiencyReport` object, the following expression evaluates to `true` if the `topRating` attribute references an object of the `GasVehicle` class type:

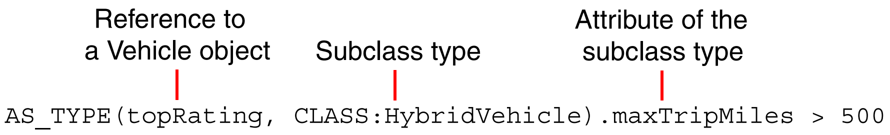

Reference to a Vehicle object
Class-type literal for GasVehicle class

```
KIND_OF(topRating,  CLASS:GasVehicle)
```

The `CLASS:`*`classType`* syntax creates a literal indicating the type of a class.

Qualifying by Attributes of a Subclass

The following expression qualifies an `EfficiencyReport` object whose `topRating` attribute references a hybrid vehicle whose `maxTripMiles` attribute is greater than 500.

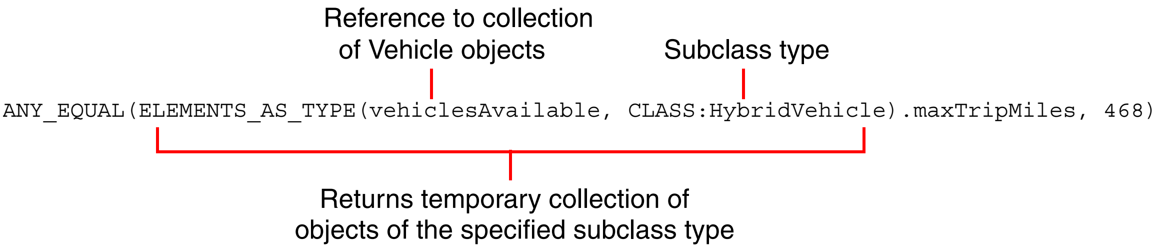


```
AS_TYPE(topRating, CLASS:HybridVehicle).maxTripMiles > 500
```

The `AS_TYPE` operator specifies that the `Vehicle` object referenced by the `topRating` attribute is to be cast to a particular class type, namely, `HybridVehicle`. When combined with the path operator, this allows access to the subclass attribute, `maxTripMiles`. If the first operand of the `AS_TYPE` operator is not a *kind of* class of the second operand, the result of the expression is a null value.

Qualifying by Attributes of Objects in a Referenced Collection

The following expression qualifies an `EfficiencyReport` object whose `vehiclesAvailable` attribute references a collection of concrete vehicle references that contains at least one hybrid vehicle whose `maxTripMiles` attribute is set to 468.



```
ANY_EQUAL(ELEMENTS_AS_TYPE(vehiclesAvailable, CLASS:HybridVehicle).maxTripMiles, 468)
```

Reference to collection of Vehicle objects
Subclass type
Returns temporary collection of objects of the specified subclass type

The `ELEMENTS_AS_TYPE` operator works like the `AS_TYPE` operator except that it operates on each element in a persistent collection. If the referenced collection of available vehicles includes two hybrid vehicles and one gas vehicle, the result of the `ELEMENTS_AS_TYPE` expression is a temporary collection populated with two hybrid vehicles and one null value.

The `ELEMENTS_OF_TYPE` operator is similar to the `ELEMENTS_AS_TYPE` operator except that it filters out elements that are not of the specified subclass type. So, for the previous example, the resulting temporary collection would include only the two hybrid vehicles.

See “Type-Evaluation Operators” on page 62 and “Type-Access Operators” on page 64 for more information.

Example: Object Qualification Code

You can use predicate strings to qualify objects in any of the kinds of operations listed in “Predicate Queries” on page 17. The following sections show several examples.

Scanning a Federated Database

You can use a predicate string in a *predicate scan*, which searches a federated database for qualified objects.

The code segments that follow iterate over vehicles in a federated database, scan for a vehicle whose license ends with the letter B, and print the complete license string for each qualified vehicle.

C++ Example

```
...
ooHandle(ooFDObj) fdH = ... // Federated-database handle
char* pql = "license =~ '.*B'";
ooItr(Vehicle) nextVehicle;
nextVehicle.scan(fdH, pql);
  while(nextVehicle.next()) {
    cout << "Matching license: " << nextVehicle->getLicense() << endl;
  }
```

Java Example

```
...
ooFDObj fd = ... // Federated database
String pql = "license =~ '.*B'";
Iterator vehicleList = fd.scan("Vehicle", pql);
      while(vehicleList.hasNext()){
      Vehicle myVehicle = (Vehicle)vehicleList.next();
      System.out.println("Matching license: " + myVehicle.getLicense);
}
```

For a complete description of scanning, see the chapter about scanning for qualified objects in the documentation for your programming interface.

NOTE An object qualifier can be used instead of a predicate string when performing a scan operation; see the section that follows.

Qualifying Objects One at a Time

You can use an object qualifier to examine each object encountered while iterating across a collection of objects. You can also use an object qualifier to verify the characteristics of an object passed in from a calling function.

An object qualifier is an instance of the non-persistence-capable class `ObjectQualifier`; it contains a predicate string to be tested against objects of a specified class or its derived classes.

To qualify objects one at a time:

1. Construct an object qualifier, passing in:
   - The type number or the class name of the persistence-capable class of objects to be tested
   - The predicate string
2. Call the object qualifier's `doesQualify` method, passing a handle to the object to be qualified.

NOTE An object qualifier can also be used when performing a scan operation on a federated database.

Qualifying a Passed-in Object

The following examples create an `objectTester` function that qualifies a rental company object if it has a related vehicle with a specified license value. Note the use of try/catch blocks.

C++ EXAMPLE

```
// Application code file

#include <ooObjy.h>
#include <objy/query/ObjectQualifier.h>
...
using namespace objy::query;
...
bool objectTester(ooHandle(RentalCompany) objH)
{
  try{
    ObjectQualifier* objQ = new ObjectQualifier(
      ooTypeN(RentalCompany), "vehicles ANY(license == 'L32IX93')");
    if(objQ->doesQualify(objH))
    {
      return true;
    }
    else
      return false;
  }
  catch(ooException &expEx){
      expEx.reportErrors();
  }
  return false;
}
```

JAVA EXAMPLE

```
// Application code file

import com.objy.db.app.ooObj;
import com.objy.query.ObjectQualifier;
...

public static boolean objectTester(RentalCompany obj)
{
  try{
    ObjectQualifier objQ = new ObjectQualifier(
      obj.getTypeNumber(), "vehicles any(license != 'L32IX93')");
  if (objQ.doesQualify(obj)) {
    return true;
  }
```

```
        else
          return false;
        }
        catch (ObjyRuntimeException e){
            e.reportErrors();
        }
          return false;
      }
```

Qualifying Objects in an Arbitrary Group

You can use an object qualifier to qualify each object in a persistent collection against a predicate string. The following examples iterate through collections and attempt to qualify the included objects.

The constructor for the `ObjectQualifier` is surrounded by a `try/catch` block that will catch validation errors for the object qualifier. The `doesQualify` method is surrounded by a `try/catch` block that will catch runtime errors during the evaluation of the predicate string.

C++ Example This example uses the DDL shown on page 95 and creates an object qualifier to qualify vehicles with a particular license plate number.

```
// Application code file
...
#include <ooCollectionBase.h>
#include <objy/query/ObjectQualifier.h>
using namespace objy::query;
…
ooHandle(ooHashSetX) setH;
ooHandle(ooObj) objH;
ObjectQualifier* objQ;
… // Set setH to reference a set of objects

// Establish the type number of the class
ooTypeNumber typeN = ooTypeN(Vehicle);

// Create an object qualifier
try {
   objQ = new ObjectQualifier(typeN, "license == '998TG1'");
} catch (ooException &expEx){
   cout << "PQL Exception: " << expEx.what() << endl;
}
```

```
// Create and initialize a scalable-collection iterator
ooCollectionIterator *setI = setH->iterator();

// Iterate through the Vehicle objects in the set
while(setI->hasNext()){
   objH = setI->next();
   // Qualify each Vehicle object
   try {
      if(objQ->doesQualify(objH)){
      … // Do something with this Vehicle object
      }
   } catch(ooException &expEx){
      cout << "PQL Exception: " << expEx.what() << endl;
   }
}
delete setI; // Delete the scalable-collection iterator
```

JAVA EXAMPLE The following example uses the Java class definitions shown on page page 103 and creates an object qualifier that attempts to qualify vehicle models that have four or more doors and seat five people.

```
// Application code file
...
import com.objy.db.util.ooCollectionIterator;
import com.objy.db.util.ooTreeSetX;
import com.objy.query.ObjectQualifier;
...
ooObj obj = new ooObj();
ObjectQualifier objQ = null;
ooTreeSetX set;
// Set 'set' to reference a collection of objects;

// Create an object qualifier.
try{
  objQ = new ObjectQualifier("Vehicle", "license == '993NCL'");
}
catch (ObjyRuntimeException e){
  System.out.println("Runtime exception caught.");
  e.reportErrors();
}
```

```
// Create a scalable-collection iterator.
ooCollectionIterator setI = (ooCollectionIterator) set.iterator();
while(setI.hasNext()){
  obj = (ooObj)setI.next();
  try{
    if(objQ.doesQualify(obj)){
      // Do something with this Vehicle object.
    }
  }
  catch (ObjyRuntimeException e){
    e.reportErrors();
  }
}
```

Using PQL Variables

You can use a *PQL variable* inside a predicate string in an object qualifier. This lets you reuse the same object qualifier after substituting different literal values for the variable.

Variable names are prefixed with a dollar sign (`$`) and suffixed with a colon followed by the type for the variable. You set the value of a variable in an object qualifier's predicate string using a method on `ObjectQualifier`, such as `setStringVarValue`.

The following examples create an object qualifier that uses a PQL variable that matches a vehicle with a given license.

C++ Example

```
...
// Create the PQL string with the PQL variable.
char* pql = "license == $licenseVar:STRING";

// Create the object qualifier.
ObjectQualifier objQ = ObjectQualifier(typeNumber, pql);

// Set the value of the variable in the object qualifier.
objQ.setStringVarValue("licenseVar", "L32IX93");

// Use qualifier to find matching object.
...

// Set a different value for the variable in the object qualifier.
objQ.setStringVarValue("licenseVar", "X94TT1");
```

```
// Use qualifier to find matching object
...
```

JAVA EXAMPLE

```
...
// Create the PQL string with the PQL variable.
String pql = "license == $licenseVar:STRING)"

// Create the object qualifier.
ObjectQualifier objQ = new ObjectQualifier(typeNumber, pql);

// Set the value of the variable in the object qualifier.
objQ.setStringVarValue("licenseVar" , "L32IX93");

// Use qualifier to find matching object.
...

// Set a different value for the variable in the object qualifier.
objQ.setStringVarValue("licenseVar" , "X94TT1");

// Use qualifier to find matching object.
...
```

See “Variable Expressions” on page 87 for more information about using PQL variables.

3

Navigation-Path Qualification

Navigation-path qualification is the process of determining whether the path that connects a series of related objects in a graph satisfies a set of conditions, typically expressed as a predicate string. A *navigation path* is traversed during the course of a *navigation query* across a graph.

This chapter describes:

- Understanding Navigation-Path Qualification
- Example: Qualifying Navigation Paths
- Qualifying Navigation Paths in a Result Qualifier
- Qualifying (Filtering) Navigation Paths in a Graph View

NOTE Navigation queries are only supported in Objectivity/C++.

Understanding Navigation-Path Qualification

Objectivity/C++ supports the concept of a graph database, in which querying according to the relationships in the data is vital. In particular, the path that connects a source object to a related target object might be meaningful because of some aspect of its composition.

When you perform a navigation query, you have the opportunity to qualify paths encountered during the traversal. You do this through the navigator components that you supply when you create a navigator instance. In particular, the following navigator components support path qualification:

- A *result qualifier* identifies the targets of the navigation query using object qualification, navigation-path qualification, or a combination of both, expressed with a predicate string.
- A *graph view* provides a mechanism for eliminating uninteresting paths from your navigation query, optionally using a predicate string.

- An optional *custom path qualifier* can perform advanced path qualification based on your own code. (This approach does not support the use of predicate strings.)

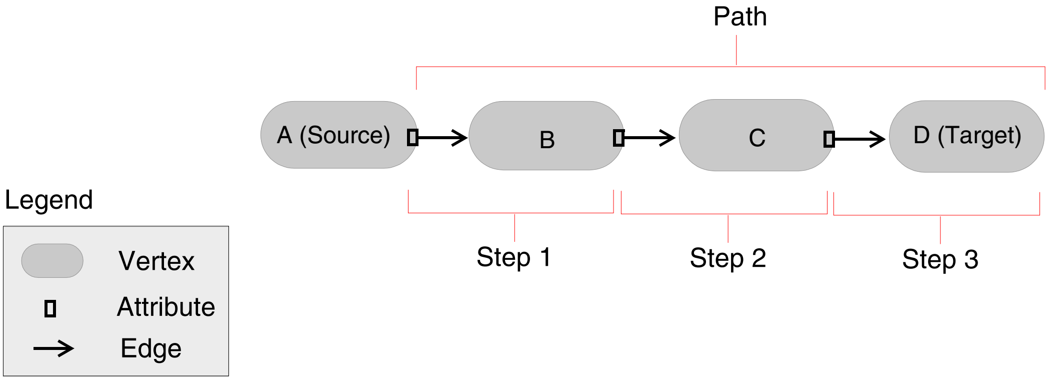
The predicate strings that you provide for result qualifiers and graph views can make use of a set of *navigation-path operators* specifically designed for path qualification.

NOTE For information about performing navigation queries, see Chapter 19, "Navigation Queries" in the *Objectivity/C++ Programmer's Guide*.

Paths and Steps

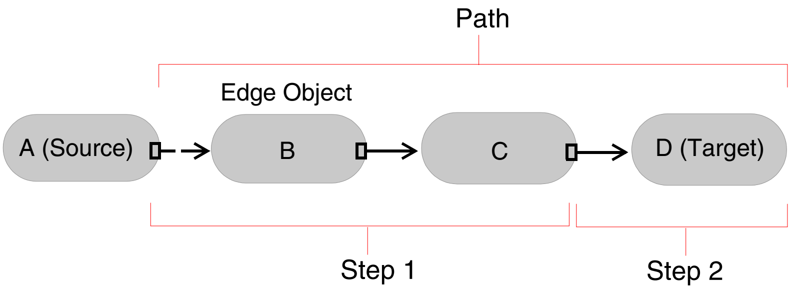
Before delving further into navigation-path qualification, it's helpful to review what constitutes a *path*. A path is a series of linked objects (also known as *vertices*) that connect a source object to a target object. The links (also known as *edges*) can be reference attributes, collections, variable-size arrays, name maps, and associations.

Each individual *step* in a path includes information about the attribute on the previous vertex that led to the current vertex, a reference to the current vertex, and information about the class type of the current vertex.



When a navigation query finds a target vertex, it returns a *result path*, which is the series of steps up to and including the target.

A graph can include *edge objects*, which are objects that themselves represent edges. Edge objects do not contribute to the step count for a path. For example, with object B from above designated as an edge object, the step count is reduced to two.



Edge objects can be defined as such after their creation; see Chapter 19, “Navigation Queries” in the *Objectivity/C++ Programmer’s Guide* for information about how to designate edge objects in your graph.

The rest of this chapter provides examples of predicates that qualify result paths and shows the context in which they are used; see “Navigation Path Operators” on page 79 for a list of all the navigation path operators and their syntax.

Example: Qualifying Navigation Paths

This section provides a sample data set and shows how to use navigation-path operators to qualify paths in that set. The example is based on a group of people that are connected through their involvement in organizations and via their email communication with each other.

The focus of this section is to demonstrate the use of predicate strings with navigation-path operators, but other code is shown to provide context (blue text is used to easily differentiate the predicate strings from other code).

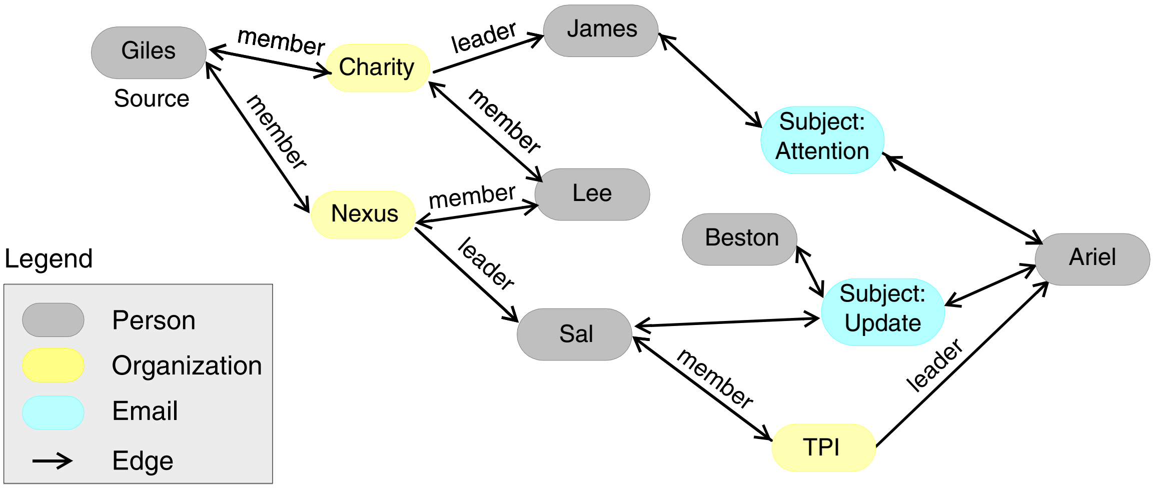
NOTE The navigation-path operators are a subset of the operators available in the PQL language, and the examples in this section often use complex predicates that involve a variety of PQL operators; see Chapter 4, "PQL Reference," for information about all available operators.

Schema Model

The data model specifies that a person can reference a collection of incoming email objects and a collection of outgoing email objects, and an organization has a bidirectional association to any number of members and a single reference to a leader.

Sample Data

Given the schema model, you might have a set of connected data as follows.



Qualifying Navigation Paths in a Result Qualifier

This section shows how to use navigation-path operators (and other PQL operators) in a result qualifier that defines the target vertices of the navigation query.

The targets themselves can be defined using any of the object qualification techniques available in PQL; see Chapter 2, "Object Qualification." After a target vertex is encountered, the path leading up to that target can be additionally qualified, and this is where the navigation path operators are used.

By Length

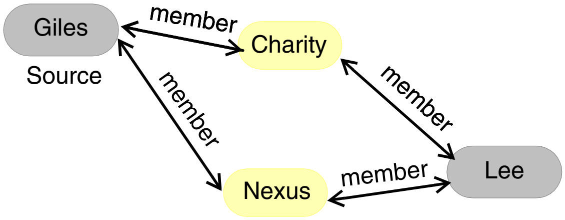
You can qualify a path based on the number of its steps. For example, the following predicate qualifies a path with two or fewer steps:

```
"PATH_LENGTH() <= 2"
```

This can be used in a result qualifier that specifies the type of the target, optionally further qualifies the target, then species a path-length predicate.

```
qualifiers::PredicateQualifier myResultQualifer(ooTypeN(Person),
   "name == 'Lee' && PATH_LENGTH() <= 2");
```

Given our sample data, the following two paths qualify (recall that the source vertex is not included in the step count for a path).



NOTE You can use a *navigation policy* to globally limit the depth of navigations independent of the result qualifier in use; see "Customizing a Navigator's Behavior Using Policies" in Chapter 19 of the *Objectivity/C++ Programmer's Guide*.

By Previous Edge or Vertex

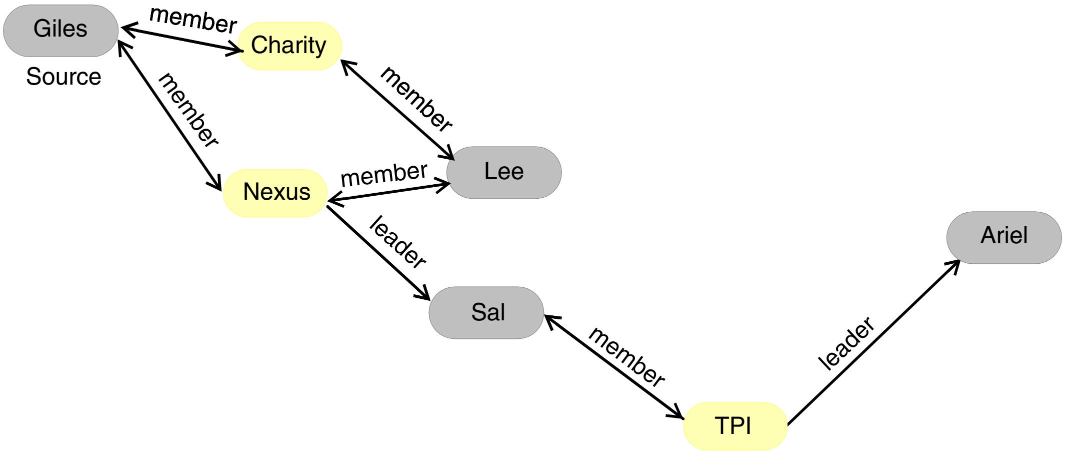
You can qualify paths based on aspects of the edge or vertex previous to the target vertex. For example, the following tests whether the previous vertex is an `Organization` named TPI:

```
"QUALIFY(PREV_VERTEX(), CLASS:Organization, name == 'TPI')"
```

This predicate can be used in a result qualifier that qualifies a `Person` vertex whose previous vertex is an `Organization` with the name `TPI`:

```
qualifiers::PredicateQualifier myResultQualifier(ooTypeN(Person),
   "QUALIFY(PREV_VERTEX(), CLASS:Organization, name == 'TPI')");
```

Given our sample data, the following two paths qualify.



NOTE You can use a *navigation policy* to prevent the revisiting repeat paths; see "Customizing a Navigator's Behavior Using Policies" in Chapter 19 of the *Objectivity/C++ Programmer's Guide*.

You can use the `PREV_EDGE()` operator in a similar fashion if you have designated edge classes in your graph view. You can also create predicates that combine the use of `PREV_VERTEX()` and `PREV_EDGE()`. See "Designating Edge Classes" in Chapter 19 of the *Objectivity/C++ Programmer's Guide* for information on setting up edge classes with a graph view.

For information about the other operators (such as `QUALIFY` and `CLASS`) used in the predicate, refer to Chapter 4, "PQL Reference."

According to Composition

You can qualify paths according to their composition by specifying a particular sequence of steps or a pattern.

The `VERTICES[]` and `EDGES[]` navigation-path operators return an array of all the vertices or edges in a path, which can then be individually qualified. To access elements from the beginning of the array, use zero or positive numbers. To access elements counting back from the end of the array, use negative numbers.

NOTE The array of vertices returned by VERTICES[] includes the source vertex but not the target vertex.

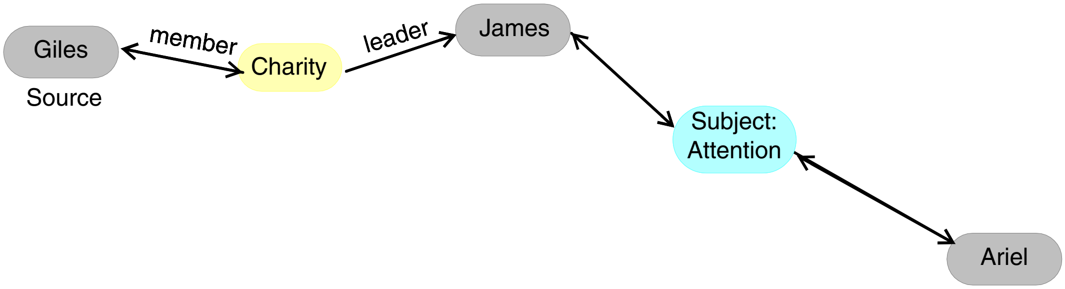
The following example tests whether the last vertex in the array (the vertex prior to the target vertex) is preceded by an Email vertex with the given subject line.

```
"QUALIFY(VERTICES()[-1], CLASS:Email, subject == 'Attention')"
```

This predicate can be used in a result qualifier that qualifies any Person vertex whose previous vertex is an Email with the subject Attention:

```
qualifiers::PredicateQualifier myResultQualifier(ooTypeN(Person),
  "QUALIFY(VERTICES()[-1], CLASS:Email, subject == 'Attention') &&
  PATH_LENGTH() < 5");
```

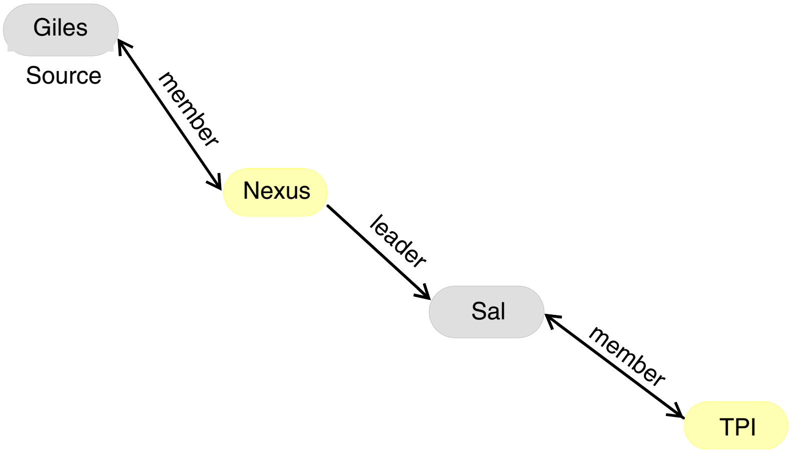
With the path length limited to four, only one path in our sample data qualifies:



You can also qualify a path by indexing into the array of vertices or edges starting at the source object. For example, the following qualifies a path that leads to an Organization vertex if the third vertex in the array is a Person named Sal.

```
qualifiers::PredicateQualifier
  matchPattern(ooTypeN(Organization),"QUALIFY(VERTICES()[2],
  CLASS:Person, name = 'Sal')");
```

Given our sample data, the following path qualifies.



If your data model includes edge classes, you can use the `EDGES[]` operator to return the array of edges in a path. You can also create predicates that combine the use of `EDGES[]` and `VERTICES[]` to match a pattern based on a sequence of edges and vertices. See "Designating Edge Classes" in Chapter 19 of the *Objectivity/C++ Programmer's Guide* for information about setting up edge classes with a graph view.

Qualifying (Filtering) Navigation Paths in a Graph View

This section shows how to eliminate uninteresting paths from your navigation query by using navigation-path operators in a graph view. With a graph view, you specify a particular type, then perform path qualification around that type. The type can be that of a vertex or an edge object in your graph.

Because a graph view's main purpose is filtering, a qualified path is eliminated from eligibility for traversal.

By Length

You can specify that certain paths in the graph are eliminated from eligibility for traversal based on their length. This is accomplished by specifying a type of object with the graph view, then specifying a path-length predicate.

```
myGraphView.excludeClass(ooTypeN(Organization), "PATH_LENGTH() > 2");
```

When an `Organization` vertex is encountered during the traversal, this graph view prevents further traversal down that path if the maximum path length of two has already been reached.

NOTE You can use a *navigation policy* to globally limit the depth of navigations independent of the result qualifier in use; see "Customizing a Navigator's Behavior Using Policies" in Chapter 19 of the *Objectivity/C++ Programmer's Guide*.

By Previous Edge or Vertex

You can specify that certain paths in the graph are eliminated from eligibility for traversal based on their previous vertex or edge. This is accomplished by specifying a type of object with the graph view, then using the PREV_EDGE() or PREV_VERTEX() operator in a predicate that qualifies that vertex or edge, for example:

```
myGraphView.excludeClass(ooTypeN(Person), "QUALIFY(PREV_VERTEX(),
   CLASS:Organization, name == 'Nexus')");
```

When a Person vertex is encountered during a traversal, this graph view prevents further traversal down this path if the previous vertex is an Organization with the given name.

You can use the PREV_EDGE operator in a similar fashion if you have designated edge classes in your graph view; see "Designating Edge Classes" in Chapter 19 of the *Objectivity/C++ Programmer's Guide*.

By Composition

You can specify that certain paths in the graph are eliminated from eligibility for traversal based on their composition. This is accomplished by specifying a type of object with the graph view, then using the VERTICES[] and/or EDGES[] operators to identify a pattern or sequence leading up to that type, for example:

```
myGraphView.excludeClass(ooTypeN(Organization),
   "QUALIFY(VERTICES()[-1], CLASS:Person, name == 'Sal') &&
    QUALIFY(VERTICES()[-2], CLASS:Organization, name == 'Nexus')");
```

When an Organization vertex is encountered during a traversal, this graph view prevents further traversal down this path if the previous vertex was a Person named Sal and the vertex before that was an Organization named Nexus.

If your data model includes edge classes, you can use the EDGES[] operator to return the array of edges in a path. You can also create predicates that combine the use of EDGES[] and VERTICES[], to match a pattern based on a sequence of edges and vertices. See "Designating Edge Classes" in Chapter 19 of the *Objectivity/C++ Programmer's Guide* for information on setting up edge classes with a graph view.

4

PQL Reference

This chapter describes the kinds of expressions you can write in PQL and provides detailed reference information for each; covered are:

- Format of Operator Expressions
- PQL Operands
- Operand Data Types
- PQL Operators
- Attribute Expressions
- Literal Expressions
- Variable Expressions
- Complex PQL Expressions
- Checking for Errors in the Predicate String

PQL ignores any whitespace or new lines that separate standard tokens.

Format of Operator Expressions

PQL operators are specified by tokens and perform arithmetic, relational, logical, and other comparison operations.

The formats for operator expressions are as follows:

Format	Description
Unary	The PQL operator precedes a single operand: **`OPERATOR`** *`operand`*
Binary	The PQL operator is between two operands: *`operand`* ***`OPERATOR`*** *`operand`*
Functional	The PQL operator precedes the operands, which are enclosed in parentheses and separated by commas: ***`OPERATOR`***`(`*`operand`*`, `*`operand`*`, `*`operand...`*`)` **Note:** Some operators use the functional format but take no operands.

Remember the following rules when working with operator expressions:

- If an operator supports interchangeable token synonyms (such as `AND` and `&&`), use the token name and not the token symbol in the functional format.

Use	Do Not Use
`AND(married, hasChild)`	`&&(married, hasChild)`

- Use unary format for token symbols.

Use	Do Not Use
`!married`	`NOT married`

- Use the functional format for unary operators specified by token name, for example, reference and count operators.
- Use the functional format for operators with more than two operands, for example, `OF` and `OF_EQUAL`.
- Use either binary or functional format for operators such as `AND`, `OR`, `XOR`, `EQ`, `PLUS`, and `MULTIPLY` that can have an arbitrary number of operands. The

functional format for specifying multiple operands is prefix notation—for example:

Functional Format	Binary Format
`AND(a, b, c)`	`a AND b AND c`
`PLUS(1, 2, 3)`	`1 + 2 + 3`
`EQ(5, 5.0, +5)`	`5 == 5.0 == +5`

Operator expressions can be nested and are evaluated in the precedence order defined in Table 4-4.

PQL Operands

PQL supports the following kinds of operands:

- Attribute expressions, which evaluate to named values from the object being qualified.
- Literal expressions, which remain constant over all objects being qualified.
- Variable expressions, which let you substitute different literal values in place of a PQL variable.
- Nested operator expressions, which pass the resulting value from one operation to another; see "Complex PQL Expressions" on page 89 for information about nested operator expressions.

Operand Data Types

Every operand has a data type, which corresponds to one or more data types recognized by the Objectivity/DB schema. When qualifying objects, you must use PQL operators that support the schema data types of that object's attributes.

The following table maps the PQL operand and result types to the corresponding schema data types, depending on your Objectivity programming interface.

All the operand types are valid for attribute expressions and the results of nested operator expressions. A subset of the operand data types is supported for literal expressions (Table 4-2) and variable expressions (Table 4-3).

NOTE In the documentation, associations in Objectivity/C++ and relationships in Objectivity for Java are often referred to generically as *relationship*s.

Table 4-1: Operand and Result Types Mapped to Schema Data Types

Operand or Result	Description		Data Types

Integer	**C++**	Signed or unsigned integer type.	`oooUInt8 ooUInt16 ooUInt32 ooUInt64`
			`ooInt8 ooInt16 ooInt32 ooInt64`

	Java	Integer type.	`int`

Numeric	**C++**	Signed or unsigned integer type, or a floating-point type.	`ooUInt8 ooUInt16 ooUInt32 ooUInt64`
			`ooInt8 ooInt16 ooInt32 ooInt64`
			`ooFloat32 ooFloat64`

	Java	Integer or floating-point type	`byte short int long float double char`
Boolean	**C++**	Value that is `true` or `false`.	`ooBoolean`
	Java	Value that is `true` or `false`.	`boolean`

String[1]	**C++**	A fixed-size array of characters, a variable-length string including Unicode strings, or an optimized string. Character types are considered strings of length 1.	`char ooChar ooChar[n] ooVArrayT<ooChar>`
			`ooChar16 ooChar16[n] ooVArrayT<ooChar16>`
			`ooChar32 ooChar32[n] ooVArrayT<ooChar32>`
			`ooUtf8String ooUtf16String ooUtf32String`
			`ooStringT<N> ooVString`
			`oojString oojArrayOfCharacter`

	Java	A variable-length string. Character types are considered strings of length 1.	`java.lang.String` `java.lang.StringBuffer`

Table 4-1: Operand and Result Types Mapped to Schema Data Types (Continued)

Operand or Result	Description		Data Types

Datetime	**C++**	An object of an Objectivity/C++ system class representing an instant in time, typically expressed as a date and time of day.	`ooRef(oojTimestamp)`
			`ooDateTime`
			`ooSQLtimestamp`
			`ooSQLnull_timestamp`
			`d_Timestamp`

	Java	An object of an Objectivity/DB system class representing an instant in time, typically expressed as a date and time of day.	`java.sql.Timestamp`

Date	**C++**	An object of an Objectivity/C++ system class representing a calendar date.	`ooDate`
			`ooRef(oojDate)`
			`ooSQLdate`
			`ooSQLnull_date`
			`d_Date`

	Java	An object of an Objectivity/DB system class representing a calendar date.	`java.util.Date` `java.sql.Date`

Time	**C++**	An object of an Objectivity/C++ system class representing a time of day.	`ooTime`
			`ooRef(oojTime)`
			`ooSQLTime`
			`ooSQLnull_time`
			`d_Time`

	Java	An object of an Objectivity/DB system class representing a time of day.	`java.sql.Time`
Interval	**C++**	An Objectivity/C++ system class representing an interval of time, measured in days, hours, minutes, and seconds.	`ooInterval`
	Java	Not available.	
Embedded-class[2]	**C++**	An embedded object of an application-defined non-persistence-capable class (*`NPCclass`*).	*`NPCclass`*
	Java	Not available.	

Table 4-1: Operand and Result Types Mapped to Schema Data Types (Continued)

Operand or Result	Description		Data Types
Reference	**C++**	An object reference to an object of a persistence-capable class (*PCclass*) or the OID of an object of a *PCclass*. *PCclass* may be an application-defined class or an Objectivity/C++ system class.	`ooRef(`*`PCclass`*`)` to-one association
	Java	A reference to an object of a persistence-capable class (*PCclass*) or the OID of an object of a *PCclass*. *PCclass* may be an application-defined class or an Objectivity for Java system class.	*`PCclass`* to-one relationship

Multi-element[3]	**C++**	Elements held in a fixed-size array, variable-size array, to-many association, persistent collection (list, set, or map), persistent oojArray, or fixed-size array of strings.	*`FixedArray`*`[`*`n`*`]`
			`ooVArrayT<`*`Type`*`>`
			to-many association
			persistent collection
			`ooRef(oojArrayOf`*`Type`*`)`
			`Type``[`*`n`*`]`

	Java	Elements held in a variable-size array, to-many relationship, persistent collection (list, set, or map), or array of strings.	*`Type[]`* to-many relationship persistent collection
Class Type	**C++**	Value that is the class type (type number) of a persistence-capable class.	`ooTypeNumber`
	Java	Value that is the class type (type number) of a persistence-capable class	`int`

1. A string can also be thought of as *multi-element* in that it is an array of characters.
2. May *not* be a string or a variable-size array.
3. Collections created both pre- and post-Release 9.3 are supported.

PQL Operators

Objectivity/DB supports the following kinds of PQL operators.

- Arithmetic Operators
- Math Operators
- Relational Operators
- Equality Operators
- Regular Expression Operators
- String Operators
- Logical Operators
- Path Operators
- Type-Evaluation Operators
- Type-Access Operators
- Reference Operators
- Count Operators
- Index Subscript Operator
- Predicate Subscript Operator
- Set Comparison Operators Based on a Boolean Expression
- Set Comparison Operators Based on Equality
- Name Map Operator
- Bitwise Operators
- Floating-Point Operators
- Date and Time Operators
- Context Operator
- Qualify Operator
- Navigation Path Operators

Naming Conventions for Operators

In some cases, a single operator may have multiple token synonyms which are interchangeable—for example:

- You can specify a path operator using either `.` or `->`
- The token `ANY_EQUAL` is a synonym and is interchangeable with the token `CONTAINS`

Some operators are overloaded such that their functionality depends on the operands that you supply. For example, the `CONTAINS` operator can be used in string comparison (looking for a substring) or in set comparison (looking for an element of a set).

You can use all capital letters, all small letters, or an initial capital letter for token names—for example, you can specify the conjunction operator using any of the following:

- `AND`
- `and`
- `And`
- `&&`

Token names with multiple words have underscores:

- `IS_VALID`
- `ANY_EQUAL`

In the sections that follow, the sample predicate strings qualify the objects introduced in "Example: Qualifying User-Defined Objects" on page 18.

NOTE See the Objectivity Developer Network for information about creating custom operators.

Arithmetic Operators

Arithmetic operators produce values as the result of mathematical functions on numeric operands. The numeric operands can be attribute expressions, literal expressions, or nested operator expressions.

Operator	Usage	Description		First Operand (op1)	Second Operand (op2)	Result Type[1]
`+`	`+op1`	Unary plus	**C++**	Numeric	—	Numeric
			Java	Numeric	—	Numeric
`-`	`-op1`	Unary minus	**C++**	Numeric	—	Numeric
			Java	Numeric	—	Numeric

`*`
`MULTIPLY`
	`op1 * op2`
	`op1 MULTIPLY op2`
		Multiplication	**C++**	Numeric	Numeric	Numeric

			Java	Numeric	Numeric	Numeric

`/`
`DIVIDE`
	`op1 / op2`
	`op1 DIVIDE op2`
		Division	**C++**	Numeric	Numeric	Numeric

			Java	Numeric	Numeric	Numeric

`%`
`MODULO`
	`op1 % op2`
	`op1 MODULO op2`
		Modulus (remainder)	**C++**	Numeric	Numeric	Numeric

			Java	Numeric	Numeric	Numeric

`+`
`PLUS`
	`op1 + op2`
	`op1 PLUS op2`
		Addition	**C++**	Numeric, Datetime, Date, Time, Interval	Interval	Same as op1

			Java	Numeric	Numeric	Numeric

`-`
`MINUS`
	`op1 - op2`
	`op1 MINUS op2`
		Subtraction	**C++**	Datetime, Date, Time, Interval	Interval	Same as op1

				Datetime, Date, Time	Same as op1	Interval
			Java	Numeric	Numeric	Numeric

1. The arithmetic operators return null if either operand is a null value.

Implicit Type Coercion

For numeric arithmetic operators, *implicit type coercion* is performed on the operands. In a mixed-type numeric expression, the following precedence of data types is used (lowest to highest):

integer → unsigned integer → floating point

The value of the operand of the lower type is promoted to that of the higher type and the result is an expression of the higher type. For example an integer multiplied by a floating-point number returns a floating-point number.

Math Operators

Math operators provide support for mathematical calculations.

Operator	Description	Usage	First Operand (op1)	Result Type[1]
`ABS`	Produces the absolute value of a number.	`ABS(op1)`	Numeric	Unsigned integer

[1] The math operators return null if the operand is a null value

Example. The ABS operator can be useful if you need to perform equality comparisons with floating-point numbers:

```
"ABS(myFloatNumber - 78.01) < 0.001"
```

Relational Operators

Relational operators produce Boolean values based on the comparison of two operands of the same operand type.

Operator	Description	Usage	First Operand (op1)		Second Operand (op2)	Result Type[1]

`<`
`LT`
	Less than	`op1 < op2`
		`op1 LT op2`
			C++	Numeric, Boolean, String, Datetime, Date, Time, Interval	Same as op1	Boolean

			Java	Numeric, Boolean, String, Datetime, Date, Time	Same as op1	Boolean

`<=`
`LE`
	Less than or equal to	`op1 <= op2`
		`op1 LE op2`
			C++	Numeric, Boolean, String, Datetime, Date, Time, Interval	Same as op1	Boolean

			Java	Numeric, Boolean, String, Datetime, Date, Time	Same as op1	Boolean

`>`
`GT`
	Greater than	`op1 > op2`
		`op1 GT op2`
			C++	Numeric, Boolean, String, Datetime, Date, Time, Interval	Same as op1	Boolean

			Java	Numeric, Boolean, String, Datetime, Date, Time	Same as op1	Boolean

`>=`
`GE`
	Greater than or equal to	`op1 >= op2`
		`op1 GE op2`
			C++	Numeric, Boolean, String, Datetime, Date, Time Interval	Same as op1	Boolean

			Java	Numeric, Boolean, String, Datetime, Date, Time	Same as op1	Boolean

[1] The relational operators return null if either operand is a null value.

For relational operators with Boolean operands, the Boolean value `true` is greater than the Boolean value `false`. For relational operators of mixed-type numeric expressions, implicit type coercion is performed as described in “Implicit Type Coercion” in the previous section.

Equality Operators

Equality operators produce Boolean values based on the equality of two operands of the same operand type. The equality operator supports an unlimited number of operands when expressed in the functional format.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Result Type[1]

`=`
`==`
`EQ`
	Equality	`op1 = op2`
		`op1 == op2`
		`op1 EQ op2`
			Any type in Table 4-1	Same as op1	Boolean

`!=`
`<>`
`NE`
	Inequality	`op1 != op2`
		`op1 <> op2`
		`op1 NE op2`			

[1] The equality operators return null if either operand is a null value.

Equality is determined based on the operand type:

- For **numeric operands**, equality is determined by value. For equality operators of mixed-type numeric expressions, implicit type coercion is performed as described in “Implicit Type Coercion” on page 52.
- For **Boolean**, **datetime, date, time, and** (C++ only) **interval operands**, both operands need to be the same operand type for value comparison.
- For **string operands**, the string lengths must be the same and equality is determined by character-by-character comparison.
- For **embedded-class operands** (C++ only), the operands must be of the same class and equality is determined by an attribute-by-attribute value comparison based on these equality rules.
- For **reference operands**, the two referenced objects are considered equal if they have the same object identifier (OID).
- For **multi-element operands**, the operands are considered equal if:
  - ❐ Both operands have the same number of elements.
  - ❐ The elements in both operands are in the same order. (For unordered multi-element operands, the order of the elements is determined by an internal iterator.)
  - ❐ The corresponding elements from each operand are equal.

Regular Expression Operators

Regular expression operators produce Boolean values based on the comparison of a string expression to a pattern. The left operand is a string attribute expression and the right operand is a string literal containing a regular expression; see "Regular Expressions" on page 80.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Result Type[1]
`=~`	Matches, case sensitive	`op1 =~ op2`	String	Regular expression	Boolean
`!~`	Does not match, case sensitive	`op1 !~ op2`			
`=~~`	Matches, case insensitive	`op1 =~~ op2`			
`!~~`	Does not match, case insensitive	`op1 !~~ op2`			

[1] The regular expression operators return null if the first operand is a null value.

All regular expression operators match the entire string in the left operand against the regular expression in the right operand. To match a prefix, suffix, or substring, the pattern must explicitly include wildcard characters at the beginning or end; see "Regular Expressions" on page 80.

String Operators

String operators provide standard capabilities for working with strings. The first operand is always a string expression. Additional operands are as described in the table.

You can qualify a string based on whether it contains a substring, or you can extract a specific substring for comparison. You can also convert strings to all uppercase or all lowercase letters for comparison.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Third Operand (op3)	Result Type[1]
`CONTAINS`	Search for substring	`CONTAINS(op1, op2)`	String	String	—	Boolean

`SUBSTR`
`SUBSTRING`
	Extract substring	`SUBSTRING(op1, op2, op3)`
		`SUBSTRING(op1, op2)`
			String	Integer	Integer	String

`UPPER`	Convert to uppercase	`UPPER(op1)`	String	—	—	String
`LOWER`	Convert to lowercase	`LOWER(op1)`	String	—	—	String

[1] The string operators return null if either operand is a null value.

The `CONTAINS` operator returns true if the value of the first operand contains the string literal specified with the second operand.

Example. When qualifying a `RentalCompany` object (whose name is `"Acme Auto"`), the following expression evaluates to true.

```
CONTAINS(name, 'cm')
```

The `SUBSTRING` operator extracts a substring starting at the specified index position and of the specified length. If length is not specified, the rest of the string is extracted.

Example. When qualifying a `RentalCompany` object, the following expression extracts a substring from the `name` attribute, starting at position zero with a length of two characters:

```
SUBSTRING(name, 0, 2)
```

Your application can use this expression in a query that finds a rental company whose name starts with the specified characters:

```
SUBSTRING(name, 0, 2) == 'Ac'
```

The `UPPER` and `LOWER` operators convert a string’s characters to all uppercase or all lowercase letters. These operators are applicable only for characters that represent ASCII alphabet letters, including UTF encodings that represent ASCII characters.

Example. When qualifying a `RentalCompany` object, the following qualifies the name regardless of the casing of letters used.

```
UPPER(name) == 'ACME AUTO'
```

Note: When using these operators on UTF strings, every encoding must represent an ASCII character.

Logical Operators

Logical operators produce Boolean values from the negation, conjunction, or disjunction of one or more Boolean expressions. Typical operands are nested operator expressions that use relational or regular expression operators.

Operator	Description	Usage[1]	First Operand (op1)	Second Operand (op2)	Result Type[2]

`!`
`NOT`
	Unary negation	`!op1`
		`NOT op1`
			Boolean	—	Boolean

`&&`
`AND`
	Conjunction	`op1 && op2`
		`op1 AND op2`
			Boolean	Boolean	Boolean

`||`
`OR`
	Disjunction	`op1 || op2`
		`op1 OR op2`
			Boolean	Boolean	Boolean

`^^`
`XOR`
	Exclusive disjunction	`op1 ^^ op2`
		`op1 XOR op2`
			Boolean	Boolean	Boolean

[1] The `NOT` operator requires functional format.
[2] The NOT operators return null for a null operand.

The following table shows the results for the conjunction, disjunction, and exclusive disjunction operators given the possible combinations of operands.

Operands		Conjunction	Disjunction	Exclusive Disjunction
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false
null	null	null	null	null
null	false	false	null	null
null	true	null	true	null
false	null	false	null	null
true	null	null	true	null

Path Operators

Path operators produce the value of an attribute of an object that is embedded into, related to, or referenced by, an attribute of the object being qualified.

Operator	Description	Usage	First Operand (op1)		Second Operand (op2)	Result Type[1]

`.`
`->`
	Accesses an attribute value of a referenced object or an embedded object	`op1.op2`
		`op1->op2`
			C++	Reference, Embedded-class	Attribute of any type in Table 4-1	Same as op2

			Java	Reference	Attribute of any type in Table 4-1	Same as op2

`.`
`->`
	Returns the attribute values of the elements of a multi-element object	`op1.op2`
		`op1->op2`
			C++	Multi-element	Attribute of any type in Table 4-1	Multi-element of element type of op2

			Java	Multi-element	Attribute of any type in Table 4-1	Multi-element of element type of op2

[1] Path operators return null if the first operand is a null value, such as a null object or a null reference. If the first operand is a multi-element of objects, any null objects in the multi-element are excluded from the evaluation.

When the object being qualified has a multi-element attribute, path operators can reference the attribute values of each element and produce a multi-element result. A path operator that produces a multi-element result cannot be the sole operator in a predicate string, but must be nested and combined with other operators.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to the values for all `license` attributes of all `Vehicle` objects that are elements of the multi-element attribute named `vehicles`:

```
vehicles.license
```

Your application can use this expression in a query that finds a rental company that has a vehicle with a particular `license`:

```
vehicles.license ANY_EQUAL ‘L321X93’
```

NOTE If the first operand is a persistent collection or an array of references, the path operator must be combined with type-access operators; see the examples in “Type-Access Operators” on page 64.

Type-Evaluation Operators

The *type-evaluation operators* let you qualify objects based on their class.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Result Type[1]
`CLASS_TYPE`	Returns class type of the referenced object	`CLASS_TYPE(op1)`	Reference	—	Class type

`KIND_OF`
`IS_TYPE`
`IS`
	Returns true if the referenced object or class type of the first operand matches the *kind of* class type of the second operand, or, when only one operand is supplied, returns true if the current object matches the provided class type.	`op1 KIND_OF op2`
		`KIND_OF(op2)`
			Reference, Class type	Class type	Boolean

`ELEMENTS_OF_TYPE`	Filters the collection of references to include only those that match the kind of class type of the second operand.	`ELEMENTS_OF_TYPE (op1, op2)`	Reference to persistent collection of references	Class type	Collection of references

[1] The `CLASS_TYPE`, `KIND_OF`, `IS_TYPE`, and `IS` operators return null if a reference operand is null. The `ELEMENTS_OF_TYPE` operand returns null if the reference to the collection of references is null.

The `CLASS_TYPE` operator returns the class type of the referenced object.

The `KIND_OF` operators that accept two arguments return Boolean values based on whether or not the referenced object is a *kind of* a given class type. One object is a kind of a class type if it is an instance of the same class or if it is an instance of a direct or indirect subclass of that class.

The `KIND_OF` operators that accept a single argument return Boolean values based on whether or not the current object is a kind of a given class type, where the current object can be one element in a multi-element or one element in an array of objects returned by a navigation path operator; see “Navigation Path Operators” on page 79.

The `ELEMENTS_OF_TYPE` operator filters a collection of referenced objects with potentially mixed class types such that the resulting temporary collection includes only elements that are of the same kind as the specified type. If any of the elements in the input referenced collection are null references, they are excluded from the resulting temporary collection.

A CLASS_TYPE or ELEMENTS_OF_TYPE operator cannot be the sole operator in a predicate string. These operators must be combined and nested with other operators.

Example. When qualifying an EfficiencyReport object, the following expression evaluates to true if the topRating attribute references an object of the GasVehicle class type:

```
CLASS_TYPE(topRating) == CLASS:GasVehicle
```

Example. When qualifying an EfficiencyReport object, the following expression evaluates to true if the topRating attribute references an object of the GasVehicle class type or of a parent class type:

```
KIND_OF(topRating, CLASS:GasVehicle)
```

Example. When qualifying an EfficiencyReport object, the following expression evaluates to true if the vehiclesAvailable attribute references a collection of objects with three objects of the GasVehicle class type:

```
LENGTH(ELEMENTS_OF_TYPE(vehiclesAvailable, CLASS:GasVehicle))
      == 3
```

Example. When qualifying a navigation path, the following expression evaluates to true if any vertex in the path is an object of the Email class type; see Chapter 3, “Navigation-Path Qualification.”

```
ANY(VERTICES(), IS_TYPE(CLASS:Email))
```

Type-Access Operators

The *type-access operators* let you cast a referenced object or a referenced collection of referenced objects to a given class type, which can provide access to attributes of the casted objects in expressions with multiple or nested operators.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Result Type[1]
`AS_TYPE` `AS`	Casts the referenced object to the specified class type.	`AS_TYPE(op1, op2)`	Reference	Class type	Casted reference
`ELEMENTS_AS_TYPE`	Casts the given collection of referenced objects to the specified type.	`ELEMENTS_AS_TYPE (op1, op2)`	Multi-element of references	Class type	Collection of casted references

[1] The `AS_TYPE` and `AS` operators return null if the first operand is not of the same kind of class as the second operand. The `ELEMENTS_AS_TYPE` operator returns null if the reference to the persistent collection is a null reference.

The `ELEMENTS_AS_TYPE` operator returns a temporary collection of references of the specified subclass type. The returned collection includes null values for elements in the input collection that were not of the kind of the specified type or for elements that were themselves null references.

A type-access operator cannot be the sole operator in a predicate string. It must be combined and nested with other operators.

Example. The following expression qualifies an `EfficiencyReport` object whose `topRating` attribute references a hybrid vehicle whose `maxTripMiles` attribute is greater than 500.

```
AS(topRating, CLASS:HybridVehicle).maxTripMiles > 500
```

Example. The following expression qualifies an `EfficiencyReport` object whose `vehiclesAvailable` attribute references a collection that includes a reference to a hybrid vehicle whose `maxTripMiles` attribute is set to 468.

```
ANY_EQUAL(ELEMENTS_AS_TYPE(vehiclesAvailable,
        CLASS:HybridVehicle).maxTripMiles, 468)
```

Reference Operators

Reference operators produce Boolean values when testing a reference attribute of the object being qualified. The reference attribute refers to another object and could be a to-one relationship.

Operator	Description	Usage[1]	Unary Operand (op1)	Result Type
`IS_NULL`	Checks for a null value; returns `true` if the attribute is not set	`IS_NULL(op1)`	Reference	Boolean
`IS_VALID`	Checks for a valid object reference; returns `true` if the attribute references an existing object	`IS_VALID(op1)`	Reference	Boolean

[1] Reference operators require functional format.

Count Operators

Count operators evaluate a multi-element attribute and produce information about the number of elements, if any.

Operator	Description	Usage[1]	Unary Operand (op1)		Result Type[2]
`IS_EMPTY`	Returns `true` if the value of the multi-element attribute has zero elements	`IS_EMPTY(op1)`	**C++**	Multi-element	Boolean
			Java	Multi-element	Boolean

`COUNT`
`LENGTH`
	Returns the number of elements in the value of the multi-element attribute	`COUNT(op1)`
		`LENGTH(op1)`
			C++	Multi-element	Unsigned integer

			Java	Multi-element	Integer

1. Count operators require functional format.
2. The count operators return null if the operand is a null value.

Index Subscript Operator

The *index subscript operator* evaluates a multi-element attribute of an object being qualified, and returns the element at the specified index location `n`, where `n` is an integer. If `n` is greater than the number of elements, a null value is returned.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)[1]	Result Type[2]
`[]`	Returns an element at an index location	`op1[op2]`	Multi-element	Integer	Element type of op1

[1] The index subscript operator can accept a negative number.

[2] The index subscript operator returns null if either operand is a null value., or if the specified index is out of bounds.

When supplying an integer value to the index subscript operator:

- Zero accesses the first element in the index.
- One accesses the second element in the index, two accesses the third element, and so on.
- Negative one accesses the last element in the index, negative two accesses the element prior to that, and so on.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to the specified `Vehicle` object at index *n* of the multi-element attribute named `vehicles`:

```
vehicles[n]
```

Your application can use this expression in a query that finds a rental company with a particular vehicle.

(C++ only) For a fixed-size array of variable-size arrays, the following expression evaluates to the *m*th element of the variable-size array at the *n*th index of the fixed-size array named *`fixedArrayAttributeName`*:

```
fixedArrayAttributeName[n][m]
```

Predicate Subscript Operator

The *predicate subscript operator* evaluates a Boolean expression for every element in a multi-element attribute, and creates an internal iterator that tracks the elements that qualify. Because it does not return a Boolean value, the predicate subscript operator cannot be the sole operator in a predicate string, but must be nested and combined with other operators.

Operator	Description	Usage	First Operand (op1)		Second Operand (op2)	Result Type[1]
`[]`	Returns elements that match the expression op2	`op1[op2]`	**C++**	Multi-element of reference or embedded class	Boolean	Multi-element of element type of op1
			Java	Multi-element of reference	Boolean	Multi-element of element type of op1

[1] The predicate subscript operator returns null if the first operand is a null value.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to all `Vehicle` objects that are elements of the multi-element attribute named `vehicles`, where each `Vehicle` satisfies the Boolean expression `available == true`:

```
vehicles[available == true]
```

Your application can use this expression in a query that finds a rental company with a particular number of vehicles available for rental:

```
COUNT(vehicles[available == true]) > 2
```

Set Comparison Operators Based on a Boolean Expression

The *set comparison operators* `ANY`, `ALL`, and `OF` evaluate to `true` if *at least one, all,* or *`n`* elements satisfy a given Boolean expression.

Operator	Description	Usage[1]	First Operand (op1)		Second Operand (op2)	Third Operand (op3)	Result Type[2]
`ANY`	Returns `true` if at least one element of op1 satisfies the expression op2	`op1 ANY op2`	**C++**	Multi-element of reference, or of embedded class	Boolean	—	Boolean
			Java	Multi-element of reference	Boolean	—	Boolean
`ALL`	Returns `true` if all elements in op1 satisfy the expression op2	`op1 ALL op2`	**C++**	Multi-element of reference, or of embedded class	Boolean	—	Boolean
			Java	Multi-element of reference	Boolean	—	Boolean
`OF` `SOME`	Returns `true` if at least op1 elements of op2 satisfy the expression op3	`OF(op1,op2,op3)`	**C++**	Unsigned integer	Multi-element of reference, or of embedded class	Boolean	Boolean
			Java	Integer	Multi-element of reference	Boolean	Boolean

1. The `OF` operator requires functional format.
2. The set comparison operators return null if the multi-element operand is a null value.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to `true` if at least one `Vehicle` object (of the multi-element attribute named `vehicles`) has a reference attribute `model` to a `VehicleModel` object with 4 doors:

```
vehicles ANY(model.doors == 4)
```

Your application can use this expression in a query that finds a rental company that has four-door vehicles.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to `true` if at least two of the `VehicleModel` objects referenced by its `models` attribute have more than two doors:

```
of(3, models, doors > 2)
```

Set Comparison Operators Based on Equality

The set comparison operators `ANY_EQUAL`, `ALL_EQUAL`, and `OF_EQUAL` evaluate to `true` if *at least one*, *all*, or *`n`* elements of a multi-element operand are equal to a comparison value. The elements of the multi-element operand and the comparison value must be the same Objectivity/DB schema data type.

Operator	Description	Usage[1]	First Operand (op1)		Second Operand (op2)	Third Operand (op3)	Result Type

`ANY_EQUAL`
`CONTAINS`
	Evaluates to `true` if at least one element of op1 is equal to the comparison value op2	`op1 ANY_EQUAL op2`
		`op1 CONTAINS op2`
			C++	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Interval, Embedded-class, Reference	Element type of op1	—	Boolean

			Java	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Reference	Element type of op1	—	Boolean
`ALL_EQUAL`	Evaluates to `true` if all elements of op1 are equal to the comparison value op2	`op1 ALL_EQUAL op2`	**C++**	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Interval, Embedded-class, Reference	Element type of op1	—	Boolean
			Java	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Reference	Element type of op1	—	Boolean
`OF_EQUAL` `SOME_EQUAL`	Evaluates to `true` if at least op1 elements in op2 are equal to the comparison value op3	`OF_EQUAL(op1,op2, op3)`	**C++**	Unsigned Integer	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Interval, Embedded-class, Reference	Element type of op2	Boolean
			Java	Integer	Multi-element of: Numeric, Boolean, String, Datetime, Date, Time, Reference	Element type of op2	Boolean

[1] The `OF_EQUAL` operator requires functional format.

The set comparison operators return null if the multi-element operand is a null value. If the multi-element operand contains null elements, those elements are treated as non-qualified. In the case of ALL_EQUAL, a null element causes the operator to return false.

(Java only) These operators must be combined with type-access operators when used with multi-element reference operands; see "Type-Access Operators" on page 64 for more information.

C++ Example. When qualifying a RentalCompany object, the following expression evaluates to true if at least one VehicleModel object of the multi-element attribute named models, has a numeric attribute doors equal to 4:

```
models.doors ANY_EQUAL 4
```

Java Example. When qualifying a RentalCompany object, the following expression evaluates to true if at least one VehicleModel object of the multi-element reference attribute named models, has a numeric attribute doors equal to 4:

```
ELEMENTS_AS_TYPE(models, class:VehicleModel).doors ANY_EQUAL 4
```

Name Map Operator

The *name map* operator produces the value corresponding to the specified string key of a name map, which is an object reference associated with the key.

Operator	Description	Usage	First Operand (op1)	Second Operand (op2)	Result Type
`KEY`	Returns the value corresponding to the `op2` key in the name map.	`op1[KEY == op2]`	Reference to an ooMap	String	Reference

A name map operator cannot be the sole operator in a predicate string, but must be combined and nested with other operators.

Example. When qualifying an `EfficiencyReport` object, the following expression evaluates to `true` if the `vehiclesList` attribute references a name map that includes a valid value for the key `vehicle1`.

```
IS_VALID(vehiclesList[KEY == 'vehicle1'])
```

Example. The following expression evaluates to `true` if the `vehiclesList` attribute references a name map that includes a vehicle with the key `vehicle1` whose license plate number is `993NCL`.

```
AS_TYPE(vehiclesList[KEY =='vehicle1'], CLASS:Vehicle).license
      == '993NCL'
```

Bitwise Operators

In some cases, an integer attribute is used to express a combination of Boolean values as a single integer value. The *bitwise operators* are useful in qualifying objects with attributes used this way.

Operator	Description	Usage	First Operand (op1)		Second Operand (op2)	Result Type
`&` `BIT_AND`	Bitwise conjunction	`op1 & op2`	**C++**	Unsigned integer	Unsigned integer	Unsigned integer
			Java	Integer	Integer	Integer
`|` `BIT_OR`	Bitwise disjunction	`op1 | op2`	**C++**	Unsigned integer	Unsigned integer	Unsigned integer
			Java	Integer	Integer	Integer
`^` `BIT_XOR`	Bitwise exclusive disjunction	`op1 ^ op2`	**C++**	Unsigned integer	Unsigned integer	Unsigned integer
			Java	Integer	Integer	Integer
`~` `BIT_COMP`	Bitwise complement	`~op1`	**C++**	Unsigned integer	—	Unsigned integer
			Java	Integer	—	Integer
`>>`	Shift right	`op1 >> op2`	**C++**	Unsigned integer	Unsigned integer	Unsigned integer
			Java	Integer	Integer	Integer
`<<`	Shift left	`op1 << op2`	**C++**	Unsigned integer	Unsigned integer	Unsigned integer
			Java	Integer	Integer	Integer

Example. When qualifying an `EfficiencyReport` object, the following expression evaluates to `true` if all of the bits in `status` are on.

```
(status & 0xFF) == 0xFF
```

Floating-Point Operators

The *floating-point operators* return true if the value of a floating-point numeric attribute represents an undefined number or an infinite number.

Operator	Description	Usage[1]	Unary Operand (op1)	Result Type
`IS_NAN`	Checks for an undefined number; returns `true` if found.	`IS_NAN(op1)`	Floating-point number	Boolean
`IS_INF`	Checks for an infinite number; returns `true` if found	`IS_INF(op1)`		

[1] Floating-point operators require functional format.

Example. The following evaluates to true if the value of the `myFloat` attribute is not a number.

```
IS_NAN(myFloat)
```

Date and Time Operators

The *date and time operators* provide information about dates, times, and datetimes.

Operator	Description	Usage	Unary Operand (op1)	Result Type
`NOW`	Returns the current date and time (local time)	`NOW()`	—	DateTime
`CUR_TIME`	Returns the current time (local time)	`CUR_TIME()`	—	Time

`TODAY`
`CUR_DATE`
	Returns the current date	`TODAY()`
		`CUR_DATE()`
			—	Date

`DAY_NAME`	Returns the name of the day in all uppercase letters in English	`DAY_NAME(`*op1*`)`	Datetime, Date	String
`MONTH_NAME`	Returns the name of the month in all uppercase letters in English	`MONTH_NAME(`*op1*`)`	Datetime, Date	String
`DAY_OF_WEEK`	Returns the number of the day within the week	`DAY_OF_WEEK(`*op1*`)`	Datetime, Date	Integer
`DAY_OF_MONTH`	Returns the number of the day within the month	`DAY_OF_MONTH(`*op1*`)`	Datetime, Date	Integer
`WEEK`	Returns the number of the week within the year	`WEEK(`*op1*`)`	Datetime, Date	Integer
`MONTH`	Returns the number of the month within the year	`MONTH(`*op1*`)`	Datetime, Date	Integer
`YEAR`	Returns the year	`YEAR(`*op1*`)`	Datetime, Date	Integer

The `NOW` operator evaluates to the current date and time. The `CUR_TIME` and `CUR_DATE` operators evaluate to the current time and the current date, respectively.

The `DAY_NAME` and `MONTH_NAME` operators accept a data or a date time and return the full name of the day or the month, respectively, in all uppercase letters in English.

The `DAY_OF_WEEK` operator returns the number of the day within the week, ranging from zero to six where Sunday is zero, Monday is one, and so forth.

The `DAY_OF_MONTH` operator returns the number of the day within the month.

The `WEEK`, `MONTH`, and `YEAR` operators return integers representing the number of a week within a year (based on the ISO 8601 standard), the number of a month within a year, or the year itself.

Example. When qualifying an `EfficiencyReport` object, the following expression evaluates to true if the report was updated before the current date and time.

```
lastUpdate < NOW()
```

Example. When qualifying an `EfficiencyReport` object, the following expression evaluates to true if the report was last updated on a Wednesday.

```
DAY_NAME(lastUpdate) == 'WEDNESDAY'
```

Example. When qualifying an `EfficiencyReport` object, the following expression evaluates to true if the last update was subsequent to 2009.

```
YEAR(lastUpdate) > 2009
```

Context Operator

The *context operator* returns a reference to the object being qualified in the current context. You can use this operator to qualify an object according to its object identifier.

Operator	Description	Usage	Result Type
`THIS`	Reference to the current object being qualified	`THIS()`	Reference

Example. When qualifying a `RentalCompany` object, the following expression evaluates to true if the rental company being qualified has the given object identifier (OID).

```
THIS() == #3-2-1-2
```

You can qualify a vehicle in the rental company’s fleet according to its OID as follows:

```
ANY(vehicles, THIS() == #2-3-1-23)
```

You can also check for a valid object reference:

```
IS_VALID(THIS())
```

Qualify Operator

The *qualify* operator tests whether a given object matches the specified type and predicate, or determines whether the current object being qualified matches the given type and predicate.

Operator	Description	Usage	First Operand (op1)		Second Operand (op2)	Third Operand (op3)	Result Type[2]

`QUALIFY`	Returns true if the given object qualifies, or, when only one operand is supplied, returns true if the current object (in an array of references) qualifies.	`QUALIFY(op1, op2, op3)`
		`QUALIFY(op2, op3)`
			C++	Reference or embedded class	Class	Boolean	Boolean

			Java	Reference	Class	Boolean	Boolean

The qualify operator can be used in object qualification or can be used when performing navigation queries.

Example. When performing object qualification on a rental company object, the following evaluates to true if a gas vehicle with the given license is available.

```
ANY(vehiclesAvailable, QUALIFY(class:GasVehicle, license ==
'AR698L'))
```

In a navigation query, the qualify operator can be used to qualify the vertex or edge objects returned by a navigation path operator; see “Navigation Path Operators” on page 79. The following examples qualify data introduced in “Example: Qualifying Navigation Paths” on page 36.

Example. When qualifying a navigation path, the following evaluates to true if the vertex previous to a designated object (identified elsewhere in the navigation query) is an organization vertex with the name `Charity`.

```
QUALIFY(PREV_VERTEX(), CLASS:Organization, name == 'Charity')
```

Example. When qualifying a navigation path, the following evaluates to true if any vertex in the navigation path is a person with the name `Ariel`.

```
ANY(VERTICES(), QUALIFY(CLASS:Person, name == 'Ariel'))
```

Navigation Path Operators

(C++ only) The *navigation path operators* produce information about a path between related vertices in a graph and are used when performing *navigation queries*; see Chapter 19, "Navigation Queries" in the *Objectivity/C++ Programmer's Guide*.

Operator	Description	Usage	Result Type

`DEPTH`
`PATH_LENGTH`
	Returns the number of steps in the path.	`DEPTH()`
		`PATH_LENGTH()`
			Unsigned integer

`PREV_EDGE`	Returns the previous edge in the path.	`PREV_EDGE()`	Reference
`PREV_VERTEX`	Returns the previous vertex in the path.	`PREV_VERTEX()`	Reference
`EDGES`	Returns a collection of all edges in the path.	`EDGES()`	Multi-element of reference
`VERTICES`	Returns a collection of all vertices in the path including the starting vertex, but excluding the target vertex.	`VERTICES()`	Multi-element of reference

The navigation-path operators can be used in result qualifiers and graph views, which are two of the components of a navigator instance that is used to perform a navigation query.

The following examples qualify the paths introduced in "Example: Qualifying Navigation Paths" on page 36.

Example. The following qualifies a path with fewer than three steps.

```
PATH_LENGTH() < 3
```

Example. The following qualifies a path where the vertex prior to a designated object (identified elsewhere in the navigation query) is an organization vertex with the name `TPI`.

```
QUALIFY(PREV_VERTEX(), CLASS:Organization, name == 'TPI')
```

Example. The following qualifies a path if any of its vertices is an organization with the name `Nexus`.

```
ANY(VERTICES(), QUALIFY(CLASS:Organization, name == 'Nexus'))
```

Example. The following qualifies a path where the third vertex from the end is an email vertex with the given subject line.

```
QUALIFY(VERTICES()[-2], CLASS:Email, subject == 'Attention')
```

Refer to Chapter 3, "Navigation-Path Qualification" for more information about using navigation-path qualifiers.

Regular Expressions

Objectivity/DB regular expression operators (page 56) test whether a string matches a pattern. A pattern is specified as a *regular expression*. Objectivity/DB implements its regular expressions based on the PCRE (Perl Compatible Regular Expression) library using the POSIX-style API. The regular-expression metacharacters in the following table are a subset of the most commonly used ones.

Metacharacter	Description
`.`	Matches any single character. Loses its special meaning when used within `[]`.
`\`	Used as a prefix operator to override any special meaning of the following character. Loses its special meaning when used within `[]`. **Note:** Within a string in your program, you must enter `\\` to produce a single `\` character in your predicate string.
`[]`	Used to bracket a sequence of characters or character ranges; matches any single character in the sequence or in one of the specified ranges. If the first character in the sequence is `^`, this pattern matches any character *except* the characters in the sequence and the specified ranges. **Note:** Within `[]`, you can use `[` to match the character `[`, but you must use `\]` to match the character `]`.
`-`	When used within `[]`, indicates a range of consecutive ASCII characters. For example, `[0-5]` is equivalent to `[012345]`. Loses its special meaning if it is the first or last character within `[]`, or the first character after an initial `^`. No special meaning when used outside `[]`.
`*`	Used as a postfix operator to cause the preceding pattern to be matched zero or more times. Loses its special meaning when used within `[]`.
`+`	Used as a postfix operator to cause the preceding pattern to be matched one or more times. Loses its special meaning when used within `[]`.
`^`	When used as the first character within `[]`, causes the bracketed pattern to match any character not specified within `[]`. When used as the first character of a regular expression, matches the beginning of the string; this use is redundant in PQL because a regular expression always matches the entire string from beginning to end. No special meaning in other locations in a regular expression.

<table>
<tr><th>Metacharacter</th><th>Description</th></tr>
<tr><td>$</td><td>When used as the last character of a regular expression, matches the end of the string; this use is redundant in PQL because a regular expression always matches the entire string from beginning to end.
No special meaning in other locations in a regular expression.</td></tr>
<tr><td>()</td><td>Used to group patterns into a single pattern (often used with the | operator).</td></tr>
<tr><td>|</td><td>OR operator in regular expressions; when used between two patterns, matches either one of the patterns.</td></tr>
<tr><td>?</td><td>? Matches the preceding element zero or one time. For example, ba? matches 'b' or 'ba'. Loses its special meaning when used within [].</td></tr>
</table>

All characters not listed in the table are literals that match themselves. For example, the comparison character A in a regular expression matches the character A in a string; in a case-insensitive comparison, it also matches the character a.

Newline characters are matched according to the POSIX default behavior.

Unlike other languages that match strings against regular expressions, the Objectivity/DB predicate-query language matches a regular expression against the *entire* string—as if the regular expression had a ^ inserted at the beginning and a $ at the end. For example, the following patterns are equivalent. They all match strings that begin with the characters 'Re' and end with the characters 'tal':

```
'Re.*tal'
'^Re.*tal'
'^Re.*tal$'
'Re.*tal$'
```

To match a prefix, suffix, or substring of the left operand, the regular expression must explicitly include wildcard characters.

- To match a prefix, end the pattern with the .* characters. For example, the following pattern matches any string that begins with the characters 'Ren':

  ```
  'Ren.*'
  ```

- To match a suffix, begin the pattern with the .* characters. For example, the following pattern matches any string that ends with the characters 'tal':

  ```
  '.*tal'
  ```

- To match a substring, begin and end the pattern with the .* characters. For example, the following pattern matches any string that contains the characters 'ent':

  ```
  '.*ent.*'
  ```

Attribute Expressions

An attribute expression evaluates to an attribute value of the object being qualified or an attribute value of a related object. The data type of the attribute determines the format of the attribute expression.

- Direct attributes of the object being qualified are referred to by name.
- Indirect attributes of the object being qualified are accessed using PQL path operators or subscript operators; see “Path Operators” on page 61 and “Index Subscript Operator” on page 67. Indirect attributes include the following:
  - (C++ only) Attributes of an embedded object
  - Attributes of a destination object of a relationship
  - Attributes of a referenced object
  - Attributes of an array element
  - Attributes of a referenced collection of references (within a limited context; see “Attribute of a Persistent-Collection Element” on page 85)

Objectivity/DB supports the attribute expressions listed in the following sections.

Note The sample predicate strings that follow qualify objects introduced in “Example: Qualifying User-Defined Objects” on page 18.

Direct Attribute of the Object Being Qualified

Within a predicate string, an unquoted sequence of alphanumeric characters is interpreted as an attribute name.

Example. When qualifying a `Vehicle` object, the following expression evaluates to the value of the string `license`:

```
license
```

Your application can use this expression in a query that finds a vehicle with a particular license:

```
license == 'L321X93'
```

A scoped syntax is needed if the attribute name is ambiguous (for example, the same name is defined in both the base class and the class of the object being tested) or if the attribute name is not visible to the object being tested due to access control. The following expressions evaluate to the value of the attribute *inheritedAttribute*, which is inherited from the base class *baseClassName*:

C++ Example

```
baseClassName::inheritedAttribute
```

Java Example

```
baseClassName.inheritedAttribute
```

NOTE Specify the base class by name only; namespace-qualified class names (C++) or package-qualified class names (Java) are not supported.

Attribute of an Embedded Object

(C++ only) A predicate string can test an attribute of an embedded object of an application-defined non-persistence-capable class.

Example. When qualifying a RentalCompany object, the following expression evaluates to the value of the string street of the embedded object address:

```
address.street
```

Your application can use this expression in a query that finds a rental company on a particular street:

```
address.street == '350 Banyon Drive'
```

A predicate string can also test based on multiple attributes of an embedded object of an application-defined non-persistence-capable class using an object literal; see Table 4-2, “Literal Expression Example Syntax,” on page 86 for more information about object literals.

Example. When qualifying a RentalCompany object, the following expression finds a rental company with a particular street, state, and zip code.

```
address == object:Address(street:'350 Banyon Drive',
           state:'CA', zipCode:95126)
```

Attribute of Destination Object of a Relationship

A predicate string can test an attribute of a destination object linked by a to-one or to-many association (C++) or relationship (Java) to the object being qualified.

Example. When qualifying a Vehicle object, the following expression evaluates to the name of the destination object RentalCompany of the relationship rentalCompany:

```
rentalCompany.name
```

Your application can use this expression in a query that finds a vehicle that is associated with a particular rental company:

```
rentalCompany.name == 'Acme Auto'
```

Example. When qualifying a `RentalCompany` object, the following expression evaluates to the value of the `license` of the specified `vehicle` object at index *n* of the relationship `vehicles`:

```
vehicles[n].license
```

Attribute of a Referenced Object

A predicate string can test an attribute of a destination object linked by a reference attribute of the object being qualified. The value of a reference attribute is an object reference to an object of a persistence-capable class.

Example. When qualifying a `RentalCompany` object, the following expression evaluates to the value of the numeric `seatingCapacity` of the `VehicleModel` object accessed through the reference attribute `model`.

```
model.seatingCapacity
```

Your application can use this expression in a query that finds vehicles with a particular seating capacity.

You can also qualify a destination object linked by a reference attribute according to its OID.

Example. When qualifying a `Vehicle` object, the following expression evaluates to true if the OID of the referenced `VehicleModel` object is `#2-2-1-11`.

```
model == #2-2-1-11
```

Your application can use this expression in a query that finds a vehicle that has a vehicle model with a particular OID.

Attribute of an Object Array Element

A predicate string can test an attribute of an object element of a fixed-size array (Java or C++) or a variable-size array (C++ only).

C++ Example. When qualifying a `RentalCompany` object, the following expression evaluates to the value of the numeric `doors` of the specified `VehicleModel` object at index *n* of the variable-size array `models`:

```
models[n].doors
```

Your application can use this expression in a query that finds a rental company that has a particular vehicle model identified by the number of doors.

Java Example. When qualifying a `RentalCompany` object, the following expression casts each element in the array of referenced `VehicleModels` to the `VehicleModel` class type, then evaluates to the value of the numeric `doors` of the specified `VehicleModel` object at index *n* of the array.

```
ELEMENTS_AS_TYPE(models, class:VehicleModel)[n].doors
```

Attribute of a Persistent-Collection Element

A predicate string can test an attribute of an object in a persistent collection.

Example. The following expression casts `Vehicle` objects in the persistent collection to the class type `HybridVehicle`.

```
ELEMENTS_AS_TYPE(vehiclesAvailable, CLASS:HybridVehicle)
```

You can use this expression with other PQL operators to test an attribute of an element of a persistent collection. For example, the following expression qualifies any `EfficiencyReport` object whose persistent collection attribute `vehiclesAvailable` includes a hybrid vehicle whose `maxTripMiles` attribute is set to 468:

```
ANY_EQUAL(ELEMENTS_AS_TYPE(vehiclesAvailable,
      CLASS:HybridVehicle).maxTripMiles, 468)
```

Example. The following expression qualifies an `EfficiencyReport` object whose `vehiclesList` attribute references a name map that includes a vehicle with the key `vehicle1` whose license plate number is `993NCL`.

```
AS_TYPE(vehiclesList[KEY == 'vehicle1'], CLASS:Vehicle).license
      =='993NCL'
```

Literal Expressions

Literal expressions are specified values that remain constant over all objects being qualified. You can specify literal expressions of the operand types shown in Table 4-2 for comparison to an attribute expression of a corresponding operand type.

Table 4-2: Literal Expression Example Syntax

Operand Type	Description	Example Syntax

Numeric	*Numeric literals* have the same syntax as in C++ and Java.	`123`
		`-77`
		`-98.765`
		`88.3e-9`
		`0xFF`

Boolean	A *Boolean literal* has the value `true` or `false`.	`true` `false`

String[1]	A *string literal* is a single-quoted or double-quoted sequence of characters.	`'555-1212'`
		`\"John Doe\"`

Datetime[2]	A *datetime literal* is a representation of month, day, year, hours, minutes, seconds, and optionally milliseconds, in that order.	`1/1/2009 11:52:30 pm`
		`1/1/2009 1:10:50:40 pm`

Date	A *date literal* is a representation of month, day, and, year, in that order.	`2-16-2007`
		`3/15/2008`

Time[2]	A *time literal* is a representation of hours, minutes, seconds, and optionally milliseconds, in that order.	`1:10:30 pm`
		`1:10:50:40 pm`

Interval (C++ only)	An *interval literal* is a duration of elapsed time.	`10:55:30`
		`46732:33:12`
		`65:4:12:40:888`

Reference	A *reference literal* or *OID literal* is an OID for a particular object.	`#3-2-9-11`
Object (C++ only)	An *object literal* specifies an object or a set of objects according to a set of attribute values. The number of the attribute name value pairs and their order is not important. The object may *not* be a string or a variable-size array.	`OBJECT:Address(street:'350 Banyon Drive', state:'CA', zipcode:95126)`

Table 4-2: Literal Expression Example Syntax (Continued)

Operand Type	Description	Example Syntax

Ordered multi-element	An *ordered multi-element literal* is enclosed in parentheses with elements separated by commas; it can contain numeric or string element types.	`(1,4,75)`
		`(4,1,75)`
		`('Bob', 'Mary', 'Jane')`

Class type	A *class-type literal* specifies a class type. It can include the namespace (C++) or package qualification (Java) as shown in the last two examples. The two forms of syntax are interchangeable.	`CLASS:A`
		`CLASS:X::Y::Z`
		`CLASS:X.Y.Z`

1. You can use both single quotes and double quotes in string literals. When you use double quotes, precede each double quote with a backslash.
2. Times are assumed to be of the same kind, there is no way to differentiate between Coordinated Universal Time (UTC) standard or in the local time standard.

Named Constants

Named constants are not supported in predicate strings.

Variable Expressions

Variable expressions let you substitute different literal values into a predicate string used in an object qualifier.

You can use PQL variables to represent the operand types shown in Table 4-2.

Table 4-3 shows the syntax for specifying variables for each literal type, where *myVar* can be any variable name you choose.

Table 4-3: Variable Expression Syntax

Operand Type	PQL Variable Syntax
Numeric	$*myVar*:INT $*myVar*:UINT $*myVar*:FLOAT
Boolean	$*myVar*:BOOL
String	$*myVar*:STRING
Datetime	$*myVar*:DATETIME
Date	$*myVar*:DATE

Table 4-3: Variable Expression Syntax (Continued)

Operand Type	PQL Variable Syntax
Time	`$myVar:TIME`
Interval (C++ only)	`$myVar:INTERVAL`
Reference	`$myVar:OID`
Class type	`$myVar:CLASS`

Variable expressions can only be used within an object qualifier's predicate string. Multiple variables can be used in the same predicate string.

Example. The following expression creates a PQL string variable called `licenseVar` that is compared against a vehicle's license attribute. The expression evaluates to true if the value supplied for the `licenseVar` variable matches the value of the vehicle's `license` attribute.

```
license == $licenseVar:STRING
```

You can use this expression as part of the predicate string in an object qualifier that qualifies a `RentalCompany` that has a vehicle with a particular license string. You use one of the set methods on `ObjectQualifier` to set the value of the PQL variable; see "Using PQL Variables" on page 31 for an example.

For more information about object qualifiers and the methods for setting variable values, see the `ObjectQualifier` class documentation for your programming interface.

NOTE For applications that perform large numbers of queries where the predicate need only differ by the values of literals, using an object qualifier with a PQL variable is more efficient than repeated scan operations with different PQL expressions; see "Using an Object Qualifier with a PQL Variable" on page 99 for an Objectivity/C++ example showing both approaches.

Complex PQL Expressions

PQL supports complex expressions that combine and nest multiple operator expressions.

Precedence

PQL operators are evaluated in precedence order. Parentheses can be used to override the precedence order. For example, the multiplication operator has higher precedence than the addition operator, so the numeric expression `3+2*5` is evaluated as `3+(2*5)`. To override this, use `(3+2)*5`.

The following table lists the PQL operators in precedence order. The operators at the top of the list have higher precedence and are evaluated first; operators in the same associativity grouping have equivalent precedence.

Table 4-4: PQL Operator Precedence

Operator		Description	Associativity
`::`	`()`	Class scope, grouping	Left to right
operator`()`		Functional format	
`[n]`	`[predicate]`	Index subscript, predicate subscript	
`KEY`		Name map lookup	
`->`	`.`	Path	
`IS_NULL`	`IS_VALID`	Reference operators	
`IS_EMPTY`	`LENGTH, COUNT`	Count operators	
`ANY` `OF`	`ALL`	Set comparison (boolean)	

`ANY_EQUAL,CONTAINS`
`OF_EQUAL`
	`ALL_EQUAL`	Set comparison (equality)	

`CLASS_TYPE`
`IS`
	`KIND_OF, IS_TYPE`
	`ELEMENTS_OF_TYPE`
		Type evaluation	

`AS_TYPE` `ELEMENTS_AS_TYPE`	`AS`	Type access	
`NOW` `TODAY`	`CUR_TIME`	(C++ only) Current date and time	

Table 4-4: PQL Operator Precedence (Continued)

Operator		Description	Associativity
IS_INF	IS_NAN	Floating-point operators	
CONTAINS UPPER	SUBSTR, SUBSTRING LOWER	String operators	
THIS		Context operator	
ABS		Absolute value	
DEPTH,PATH_LENGTH EDGES VERTICES	PREV_EDGE PREV_VERTEX	Navigation-path operators	
QUALIFY		Qualify operator	
!		Logical negation	Right to left
~		Bitwise complement	
+	-	Unary plus, minus	
*	/	Multiplication, division	Left to right
%		Modulus	
+	-	Addition, subtraction	Left to right
<<	>>	Bitwise shift left, right	Left to right
< >	<= >=	Less than, greater than	Left to right
==	!=	Equality, inequality	Left to right
=~ =~~	!~ !~~	Regular expression	
&		Bitwise AND	Left to right
^		Bitwise exclusive OR	Left to right
|		Bitwise inclusive OR	Left to right
&&		Logical AND	Left to right
^^		Logical exclusive OR	Left to right
||		Logical inclusive OR	Left to right

Checking for Errors in the Predicate String

If there is an error in the PQL syntax, or if there is a runtime error during the evaluation of the predicate string, an exception is thrown:

- For Objectivity/C++, an `ooException` is thrown.
- For Objectivity for Java, an `ObjyRuntimeException` is thrown.

After catching an exception in a `try/catch` block, you can call the `reportErrors` method to determine the specific error.

The following examples demonstrate *invalid* predicate strings.

Predicate String	Description of Error
`"name == RENTAL"`	The schema attribute for the attribute name specified in the predicate string cannot be found. Use `'RENTAL'` to designate the string literal.
`"doors >* 2"`	The specified operator token name is unknown. Use `>=` to designate the token.
`"(doors + seatingCapacity >= 9 AND automatic"`	The provided predicate string has syntax errors. Add the missing closing parenthesis.
`"vehicles.license == 'L32IX93'"`	The operands are incompatible. The `vehicles.license` operand returns an array of strings, which cannot be tested for equivalency against a single string. The equality operator `==` requires that both operands be of the same type. A sample correct usage is: `vehicles.license ANY_EQUAL 'L321X93'`

The following table lists different kinds of errors that result in exceptions when working with predicate strings:

Type of Error	Description
Syntax Error	The predicate string has syntax errors.
Invalid Predicate	The result type of the predicate string must be a Boolean type.

Type of Error	Description
Unknown Type Number	The type number used to initialize an object qualifier is not known.
Unknown Attribute	The attribute name specified in the predicate string cannot be found in the schema model.
Unknown Token	The operator token name specified in the predicate string is unknown.
Operand Mismatch	The operator requires different operands than were provided.
Too Few Operands	The operator requires more operands than were provided.
Too Many Operands	The operator requires fewer operands than were provided.
Incompatible Operand	The result type of an operand is not compatible with the result type required by the operator.
Invalid Regular Expression	The second operand for the regular expression operator must be a string literal or a valid regular expression.
Operand Types Mutually Incompatible	The operands are not compatible with each other given the specified operator. Some operators, such as ==, require that all operands have the same or compatible types.
Element Types Mutually Incompatible	The elements are not compatible with each other given the specified ordered multi-element literal.
Operator Implementation Error	The operator has set an unknown type number on one of its operands.
(C++ only) Object Literal Value Incompatible	One of the object literal's attribute types does not match the attribute it is paired with.
Variable Type Not Supported	The specified variable type is not supported; see "Variable Expressions" on page 87 for the supported types.
Variable Not Defined	The variable name specified with the `ObjectQualifier`'s `set`*`Type`*`VarValue` method is not defined for the PQL string.
Variable Value Not Set	The value of a variable used in the PQL string provided to the `ObjectQualifier` has not been set.
Variable Value Incompatible With Variable Type	The variable value is not compatible with the defined variable type.

A

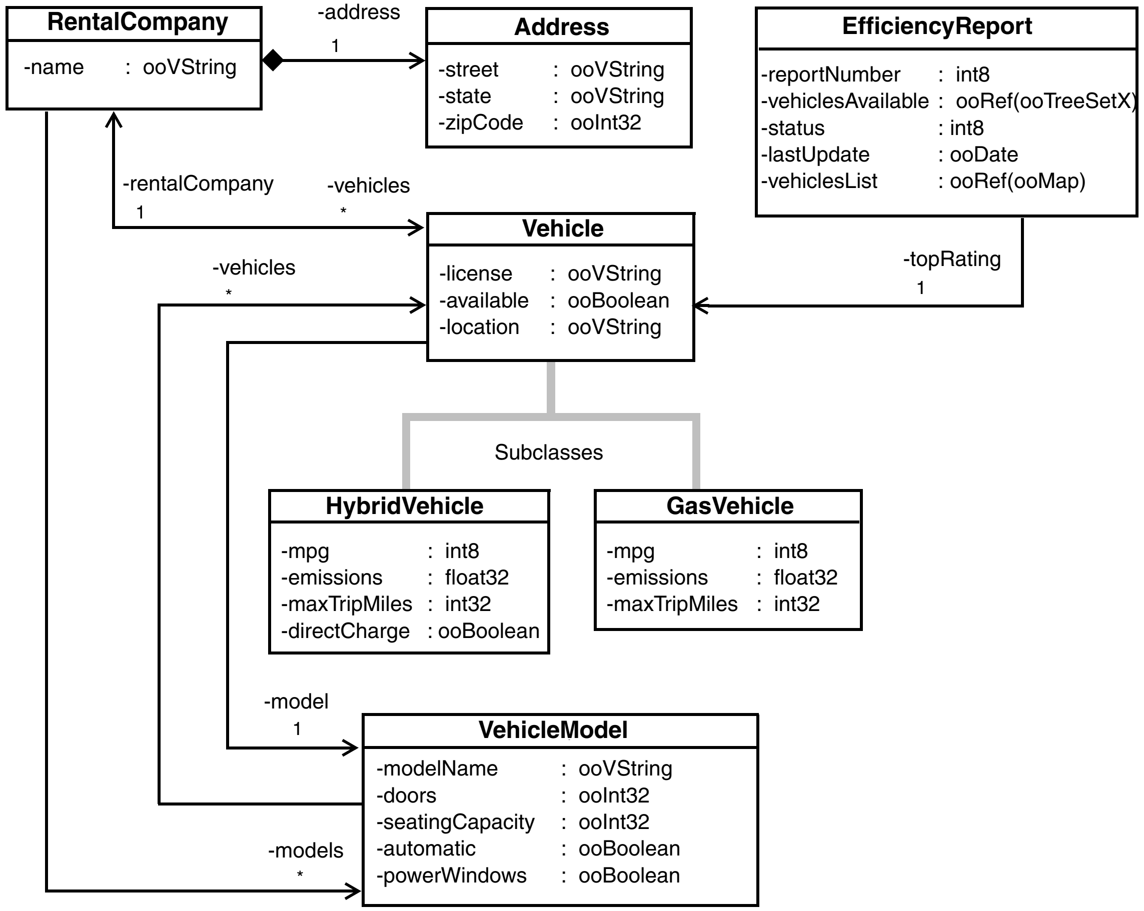
C++ Examples

This appendix provides supplemental information for using PQL with the Objectivity/C++ programming interface. Included are:

- Details about the schema model and the DDL files for the rental company example discussed in Chapter 2, “Object Qualification.”
- Examples of complex PQL expressions in the context of Objectivity/C++ code.
- An example that uses an object qualifier with a PQL variable.

Rental Company Example

The following shows the schema model for the user-defined classes in the car rental company example discussed in Chapter 2, “Object Qualification.”



DDL Class Definitions

The DDL for the schema model is as follows.

```
// DDL file rentalCompany.ddl
class Address {
public:
   ooVString street;
   ooVString state;
   ooInt32 zipCode;
   …
};

class RentalCompany : public ooObj {
public:
   ooVString name;
   Address address;      // Embedded attribute
   // Bidirectional association
   ooRef(Vehicle) vehicles[] <-> rentalCompany
         : copy(drop);
   ooVArrayT<ooRef(VehicleModel)> models;
   …
};

class VehicleModel;            // Forward reference

class Vehicle : public ooObj {
public:
   ooVString license;
   ooBoolean available;
   ooVString location;
   ooRef(VehicleModel) model;
   // Bidirectional association
   ooRef(RentalCompany) rentalCompany <-> vehicles[]
         : copy(drop);
   …
};

class GasVehicle : public Vehicle {
public:
   int32 mpg;
   float32 emissions;
   int32 maxTripMiles;
   …
};
```

```
class HybridVehicle : public Vehicle {
public:
   int32 mpg;
   float32 emissions;
   int32 maxTripMiles;
   ooBoolean directCharge;
   …
};

class VehicleModel : public ooObj {
public:
   ooVString modelName;
   ooInt32 doors;
   ooInt32 seatingCapacity;
   ooBoolean automatic;
   ooBoolean powerWindows;
   // Variable-size array of references
   ooVArrayT<ooRef(Vehicle)> vehicles;
   …
};

class EfficiencyReport : public ooObj {
public:
   int32 reportNumber;
   ooRef(ooTreeSetX) vehiclesAvailable;
   int32 status;
   ooDate lastUpdate;
   ooRef(ooMap) vehiclesList;
   ooRef(Vehicle) topRating;
   …
};
```

Examples of Complex PQL Expressions

This section presents examples of complex PQL expressions that an application could use to qualify objects that were introduced in the car rental company example. The examples assume that the application creates a session, opens it for read, and uses it to obtain a handle called `fdH` to the federated database.

The following query finds four-door compact vehicle models without power windows:

```
// Application code file
...
ooHandle(ooFDObj) fdH = ... // Set the federated-database handle
ooItr(VehicleModel) vmItr1; // Create a VehicleModel iterator
char* pql =
   "AND(modelName == 'compact', doors == 4, !powerWindows)";
try {
   vmItr1.scan(fdH, pql);
} catch (ooException expEx){
   cout << "PQL Exception: " << expEx.what() << endl;
}
... // Advance the iterator and process each object
```

The following query finds large capacity vehicle models where at least ten vehicles are available.

```
ooItr(VehicleModel) vmItr2; // Create a VehicleModel iterator
char* pql2 = "((seatingCapacity)>=5) &&
                                    COUNT(vehicles[available])>10";

... // Initialize the iterator and process each object
```

The following query finds rental companies with available vehicles that have a license starting with `'CA'`.

```
ooItr(RentalCompany) rcItr1; // Create a RentalCompany iterator
char* pql3 = "ANY(vehicles[license =~ 'CA.*'],available)";

... // Initialize the iterator and process each object
```

The following query finds rental companies that have five or more models, each with ten or more vehicles, such that each model has four doors.

```
ooItr(RentalCompany) rcItr2; // Create a RentalCompany iterator
char* pql4 = "OF(5,models[COUNT(vehicles) >= 10], doors == 4)";

... // Initialize the iterator and process each object
```

The following query finds rental companies using an object literal that specifies the state and zip code of the address.

```
ooItr(RentalCompany) rcItr1; // Create a RentalCompany iterator
char* pql3 = "OBJECT:Address(state:'CA', zipcode:95126)";

... // Initialize the iterator and process each object
```

The following query finds rental companies that have at least two luxury vehicle models.

```
ooItr(RentalCompany) rcItr3; // Create a RentalCompany iterator
char* pql5 = "OF_EQUAL(2, models.modelName, 'luxury')";

... // Initialize the iterator and process each object
```

The following query qualifies an efficiency report object that has a gas vehicle with a particular license.

```
ooItr(EfficiencyReport) repItr1; // RentalCompany iterator
char* pql6 = "ANY(ELEMENTS_AS_TYPE(vehiclesAvailable,
                      CLASS:GasVehicle), license == 'AR698L')";

... // Initialize the iterator and process each object
```

The following query qualifies an efficiency report object updated today whose `status` bits are all on.

```
ooItr(EfficiencyReport) repItr2; // Efficiency report iterator
char* pql7 = "((status & 0xFF) == 0xFF) ||
                    (lastUpdated == TODAY())";

... // Initialize the iterator and process each object
```

Using an Object Qualifier with a PQL Variable

For applications that perform large numbers of queries where the predicate need only differ by the values of literals, using an object qualifier with a PQL variable is more efficient than repeated scan operations with different PQL expressions.

C++ EXAMPLE The following examples show the repeated scan approach, followed by the more efficient approach that uses an object qualifier with a PQL variable.

```
// Less efficient - repeating scans with different PQL strings
#include <ooObjy.h>
...
ooItr(EfficiencyReport) reportItr;
int reportNumber;
int reportCount = 0;
int reportsFound = 0;
int i;
...
for(i = 0; i < reportCount; i++)
   {
      reportNumber = i + 1;
      char predicate[50];
      sprintf(predicate, "reportNumber == %d", reportNumber);
      reportItr.scan(db, predicate);
      if(reportItr.next())
         reportsFound++;
   }
...
```

```
// More efficient - using an object qualifier with a PQL
// variable
#include <ooObjy.h>
#include <objy/query/ObjectQualifier.h>
...
using namespace objy::query;
...
ooItr(EfficiencyReport) reportItr;
int reportNumber;
int reportCount = 0;
int reportsFound = 0;
int i;
```

```
ObjectQualifier objQ =
            ObjectQualifier(ooTypeN(EfficiencyReport),
               "reportNumber == $reportNumVar:UINT");
...
for(i = 0; i < reportCount; i++)
   {
      reportNumber = i + 1;
      objQ.setUintVarValue("reportNumVar", reportNumber);
      reportItr.scan(db, objQ);
      if(reportItr.next())
         reportsFound++;
   }
...
```

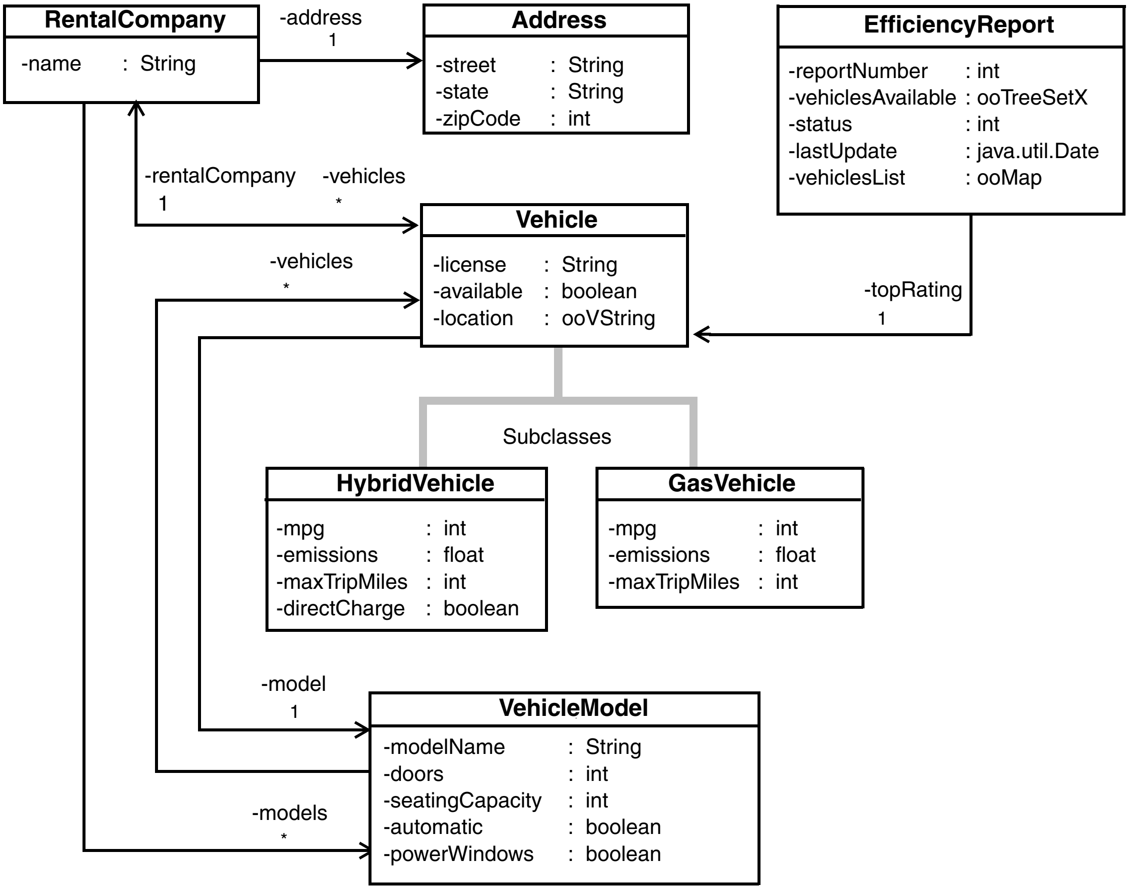
Java Examples

This appendix provides supplemental information for using PQL with the Objectivity for Java programming interface. Included are:

- Details about the schema model and the Java class files for the rental company example discussed in Chapter 2, "Object Qualification."
- Examples of complex PQL expressions in the context of Objectivity for Java code.

Rental Company Example

The following shows the schema model for the user-defined classes in the car rental company example discussed in Chapter 2, "Object Qualification."



Java Class Definitions

The Java class definitions for the schema model are as follows.

Address.java

```
…
public class Address extends ooObj {
   protected String street;
   …
}
```

RentalCompany.java

```
…
public class RentalCompany extends ooObj {
   protected String name;
   protected Address address;
   protected VehicleModel[] models;
   protected ToManyRelationship vehicles;
   …
   public static OneToMany vehicles_Relationship(){
      return new OneToMany(
      "vehicles", // field name
      "Vehicle",    // related class
      "rentalCompany", // inverse relationship field name
      Relationship.COPY_DELETE,
      Relationship.VERSION_DELETE, false, false,
      Relationship.INLINE_NONE);
   }
   …
}
```

Vehicle.java

```
…
public class Vehicle extends ooObj {
   protected String license;
   protected boolean available;
   protected String location;
   protected VehicleModel model;
   protected ToOneRelationship rentalCompany;
   …
   public static ManyToOne rentalCompany_Relationship(){
      return new ManyToOne(
      "rentalCompany", // field name
      "RentalCompany",    // related class
      "vehicles", // inverse relationship field name
      Relationship.COPY_DELETE,
      Relationship.VERSION_DELETE, false, false,
      Relationship.INLINE_NONE);
   }
   …
}
```

GasVehicle.java

```
…
public class GasVehicle extends Vehicle {
   private int mpg;
   private float emissions;
   private int maxTripMiles;
   …
}
```

HybridVehicle.java

```
…
public class GasVehicle extends Vehicle {
   private int mpg;
   private float emissions;
   private int maxTripMiles;
   private boolean directCharge;
   …
}
```

VehicleModel.java

```
...
public class VehicleModel extends ooObj {
   protected String modelName;
   protected int doors;
   protected int seatingCapacity;
   protected boolean automatic;
   protected boolean powerWindows;
   protected Vehicle[] vehicles;
   ...
}
```

EfficiencyReport.java

```
...
public class EfficiencyReport extends ooObj {
   public int reportNumber;
   public ooTreeSetX vehiclesAvailable;
   public int status;
   public java.util.Date lastUpdate;
   public ooMap vehiclesList;
   public Vehicle topRating;
   ...
```

Examples of Complex PQL Expressions

This section presents examples of complex PQL expressions that an application could use to qualify objects that were introduced in the car rental company example. The examples assume that the application creates a session called `session`, opens it for read, and sets the variable `fdH` to the database to be scanned.

The following query finds four-door compact vehicle models with power windows:

```
String pql =
  "AND(modelName == 'compact', doors == 4, powerWindows == true)";
Iterator vmIter = fdH.scan("VehicleModel", pql); // Create iterator
  while(vmIter.hasNext()){
    VehicleModel myVehicleModel = (VehicleModel)vmIter.next();
    myVehicle.fetch();
    System.out.println("Model name: " + myVehicleModel.modelName);
    ... // Process object
    myVehicleModel.delete();
  }
```

The following query finds large capacity vehicle models where at least ten vehicles are available.

```
String pql2 =
  "((seatingCapacity)>=5) && COUNT(ELEMENTS_AS(vehicles,
        CLASS:Vehicle)[available])>10";
Iterator vmIter2 = dbH.scan("VehicleModel", pql2); // Create iterator
while(vmIter2.hasNext()){
  ... // Process object
}
```

The following query finds rental companies with available vehicles that have a license starting with `'CA'`.

```
String pql3 =
  "ANY(vehicles[license =~ 'CA.*'],available)";
Iterator rcIter = dbH.scan("RentalCompany", pql3); // Create iterator
while(rcIter.hasNext()){
  ... // Process object
}
```

The following query finds rental companies with at least two vehicles that are available.

```
String pql4 = "OF_EQUAL(2, vehicles.available, true)";
Iterator rcIter = dbH.scan("RentalCompany", pql4); // Create iterator
while(rcIter.hasNext()){
  ... // Process object
}
```

The following query finds rental companies that have an available vehicle whose model name is luxury.

```
String pql5 = "AN(vehicles[available], model.modelName == 'luxury')";
Iterator rcIter = dbH.scan("RentalCompany", pql5); // Create iterator
while(rcIter.hasNext()){
  ... // Process object
}
```

The following query qualifies an efficiency report object that has a gas vehicle with a particular license.

```
char* pql6 = "ANY(ELEMENTS_AS_TYPE(vehiclesAvailable,
  CLASS:GasVehicle), license == 'AR698L')";
Iterator erIter1 = fd.scan("EfficiencyReport", EfficiencyReportPql);

while(erIter1.hasNext()){
      ... // Process object
      }
```

Index