Introduction

The MOIN Query Language (aka MQL) is a small query language and API to access the MOIN repository. Unlike JMI (the Java Metadata Interfaces, described elsewhere) or OCL (the Object Constraint Language, also described elsewhere), MQL is more restrictive and declarative, which makes it possible to more efficiently retrieve data from the repository (both in terms of memory and performance). The price for this is that it is somewhat less expressive than either JMI or OCL. MQL queries are akin to SQL, but instead of being defined over tables in a schema, an MQL query is defined over types in a MOF-based metamodel. As in SQL, it is possible to constrain the Cartesian product of provided types via attribute value constraints as well as joining constraints over both associations and attributes. Based on this filtered Cartesian product, selections can be made to compose the result table.

This user guide explains the different features of MQL and also provides hints on how to optimally exploit MQL for efficient querying. We focus on the concrete syntax as well as how to integrate MQL queries in MOIN client programs. The code examples in this user guide should be runnable with slight adaptations and assuming the setup is there. E.g. we assume a basic understanding of MOIN connections, MRIs, PRIs, and CRIs. Also, we do not explain peculiarities of different MOIN deployments such as the Eclipse integration. Instead, we assume that the reader knows how to obtain a connection and understand what parts of the repository become visible because of it. We will refer to the visibility of one connection as the *client-scope*. This is important because MQL queries by default are executed over this client-scope unless additional scope restrictions are provided (as we will explain below).

Small introductory example

Before getting into the nitty-gritty of MQL queries, we provide a simple example and show how the query can be expressed with the standard MOIN API and JMI; with OCL; and with MQL. Assume we have a metamodel with one meta-class called **`Department`** whose **`name`** attribute is of type **`String`**. The query we are interested in wants to retrieve all names of all departments whose name starts with **`s`**.

Example in JMI

With the standard MOIN API and JMI, the following code solves our problem:

```
        DepartmentClass departmentClassExtent = myConnection.getClass( DepartmentClass.CLASS_DESCRIPTOR );
        Collection<RefObject> departments = departmentClassExtent.refAllOfType( );
        List<String> departmentNamesStartingWithS = new ArrayList<String>( );
        for ( RefObject department : departments ) {
            String depName = ( (Department) department ).getName( );
            if ( depName.startsWith( "s" ) ) {
                departmentNamesStartingWithS.add( depName );
            }
        }
```

Via a MOIN connection, we are able to obtain the JMI class extent for departments (**`DepartmentClass`**). A JMI class extent is both a factory and holder for already created instances. A class extent provides a method **`refAllOfType()`** which returns all instances of **`Department`** and all its sub-classes as a collection of JMI objects. From there it is easy to obtain the requested names because attributes on JMI objects can be accessed in the usual JavaBeans style way.

Example in OCL

OCL, the Object Constraint Language, is primarily intended to describe constraints (i.e. predicates) on a MOF metamodel. In this capacity, it has to be able to query the repository. MOIN offers an OCL freestyle registry which can be used to emulate this. Here is the code for our problem:

```
        OclFreestyleRegistry ocl = myConnection.getOclRegistryService( ).getFreestyleRegistry( );
        String oclQuery = "Department.allInstances().name->select(depName | depName.subString(1, 1) = 's')";
        RefClass context = myConnection.getClass( DepartmentClass.CLASS_DESCRIPTOR );
        RefPackage[] packages = new RefPackage[] { myConnection.getPackage( CompanyPackage.PACKAGE_DESCRIPTOR ) };
        OclExpressionRegistration oclExpression =
              ocl.createExpressionRegistration(
              "myQuery", oclQuery, OclRegistrationSeverity.Warning, new String[] { "test" }, context, packages );
        Collection<String> departmentNamesStartingWithS = (Collection<String>) (oclExpression.evaluateExpression(null));
```

This code is admittedly somewhat cumbersome because MOIN's OCL offering has not been optimized for this use-case. Instead, let us turn to the MQL way of doing this.

Example in MQL

In MQL, the query can be formulated, executed, and accessed as follows:

```
        String query = "select dep.name from Company::Department as dep where for dep(name like 's*')";
        MQLResultSet resultSet = this.conn.getMQLProcessor().execute(query);
        List<String> departmentNamesStartingWithS = new ArrayList<String>(resultSet.getSize());
        for (int i = 0; i < resultSet.getSize(); i++) {
            departmentNamesStartingWithS.add((String)resultSet.getAttribute(i, "dep", "name"));
        }
```

The critical difference between the MQL solution and the OCL or JMI solution is the *way* the query is executed. Although it highly depends on the capabilities of the current MOIN configuration in which the query is executed (more accurately, the strength of the underlying index), it is the goal of MQL to execute the query without burdening MOIN's runtime system. In both the JMI and OCL scenarios, MOIN will always load all partitions in which instances of **`Department`** can be found and execute the additional filtering on the attribute inside MOIN's runtime environment. In contrast, an MQL result set will not resolve any objects unless explicitly asked for. In the above example, the MQL code will not at all address MOIN's runtime if it is equiped with a sufficiently powerful index.

JMI's refAllOfType() and refAllOfClass() in MQL

Before we delve into the details and features of MQL, we describe how to handle the 2 standard JMI queries **`refAllOfClass()`** (all instances of a given type) and **`refAllOfType()`** (all instances of a given type and all its sub classes). They are really only a simplification of the example query given above. Observe however that, generally, it is advisable to constrain the sought instances more by specifying additional filters on the attribute values and perhaps reducing the scope from where instances should be taken (see later sections for more information on scope restrictions). The inability to provide additional filters with JMI's **`refAllOfClass()`**and **`refAllOfType()`**is its principal weakness because, as we mentioned before, there is no way for JMI to optimize the query with respect to the underlying indexing infrastructure.

If **`myConnection`** is your MOIN connection, you obtain an MQL processor as follows:

```
        MQLProcessor mql = myConnection.getMQLProcessor( );
```

In original JMI code, you may have written something like this:

```
        DepartmentClass departmentClassExtent = myConnection.getClass( DepartmentClass.CLASS_DESCRIPTOR );
        Collection<RefObject> departments = departmentClassExtent.refAllOfType( );
```

This is equivalent to the following MQL code:

```
        String refAllOfTypeQuery = "select dep from Company::Department as dep";
        RefObject[] departments = mql.execute( refAllOfTypeQuery ).getRefObjects( "dep" );
```

Alternatively, when you already have a department then the following code will do as well:

```
        String refAllOfTypeQuery = "select dep from type " +
                                   ( (Partitionable) someDep.refMetaObject( ) ).get___Mri( ) + " as dep";
        RefObject[] departments = mql.execute( refAllOfTypeQuery ).getRefObjects( "dep" );
```

The case for **`refAllOfClass()`** is almost identical:

```
        Collection<RefObject> departments = departmentClassExtent.refAllOfClass( ); // without subtypes
```

turns into:

```
        String refAllOfClassQuery = "select dep from Company::Department withoutsubtypes as dep";
        RefObject[] departments = mql.execute( refAllOfClassQuery ).getRefObjects( "dep" );
```

As you can see, the keyword **`withoutsubtypes`** indicates that subtypes should not be included. It's that simple!

Overview

Key Characteristics

The simple example above already illustrates the key ideas behind MOIN's query language. Here we summarize its key characteristics as a query language

- MQL is an SQL-like language designed for querying instances of MOF-based metamodels.
- MQL queries are always strongly typed against classes (or structure types) in the metamodel. This implies that MQL is not appropriate for untyped queries like "return all contained elements of a given element".
- Because MQL queries are strongly typed, it is possible to first prepare a query (which performs all type and consistency checks) before executing it. A prepared query can be executed multiple times.
- MQL result sets are table-like structures that are fully calculated upon execution and returned for further processing. In other words, it is not possible to keep a stream of results open like in a regular database.
- MQL queries come both in a concrete syntax form and abstract syntax form. We will not describe the abstract syntax here because parsing a concrete syntax query is very fast and *much* more convenient. However, if you feel the need to encode your query in the provided AST, please consult the JavaDoc, which is provided with each element of the abstract syntax (The entire MQL Java Docs are centered around package **`com.sap.tc.moin.repository.mql`**). The start element for an AST-based query is MQLQuery

Query Structure

An MQL query essentially consists of 3 main blocks: the *selection clause*, the *from clause*, and the optional *where clauses*. In the EBNF for the concrete syntax, this looks like this (the complete EBNF of MQL can be found here):

```
<MQLquery> ::= select <select-clause>
               from <from-clause>
               (where <where-clause>)*
```

- *Selection clause*: The selection clause allows you to select either model elements (MRIs as we will see below) and/or (primitive-typed) attributes of model elements
- *From clause*: In the from clause, you list all types (which are MOF classes or structure types) of one ore more metamodels from which we you

want to obtain instances. If no additional constraints are provided, MQL will produce a result set by building the Cartesian product of all provided types. Optionally, a type in the from clause can be annotated with additional restrictions such as scope. Also, instead of providing a metamodel type, it is also possible to provide a number of fixed instance elements. Examples of different from clauses are discussed below

- *Where clauses*: A where clause can be used to further filter the desired tuples of the result set. Where clauses can put local constraints on attributes of types in the from clause. Additionally, it is possible to formulate join-constraints on multiple types in the from clause. Such join-constraints can be formulated over metamodel associations or attribute equality. Details of the different types of where clauses are also provided later. All where-clauses are always logically AND-connected. In other words, each extra constraint added as a where-clause *additionally* filters the result set.

MQL in Java

In Java, your MQL code will mostly have the following format:

```
        String yourQuery = "..."; // here you formulate your MQL query in concrete syntax
        MQLProcessor mql = myConnection.getMQLProcessor( ); // obtain the mql processor
        MQLPreparedQuery preparedQuery = mql.prepare( query ); // prepare the query
        MQLResultSet resultSet = mql.execute( preparedQuery ); // execute it
        RefObject[] resultElements = resultSet.getRefObjects( "myAlias" ); // dissect the result set
```

You first obtain an **`MQLProcessor`**, then you encode your query in the concrete syntax as a String, prepare the query to obtain an **`MQLPreparedQuery`** and finally execute it to obtain an **`MQLResultSet`**. This result set can then be further dissected to process the query results.

Preparation is optional, i.e. you can also write the following:

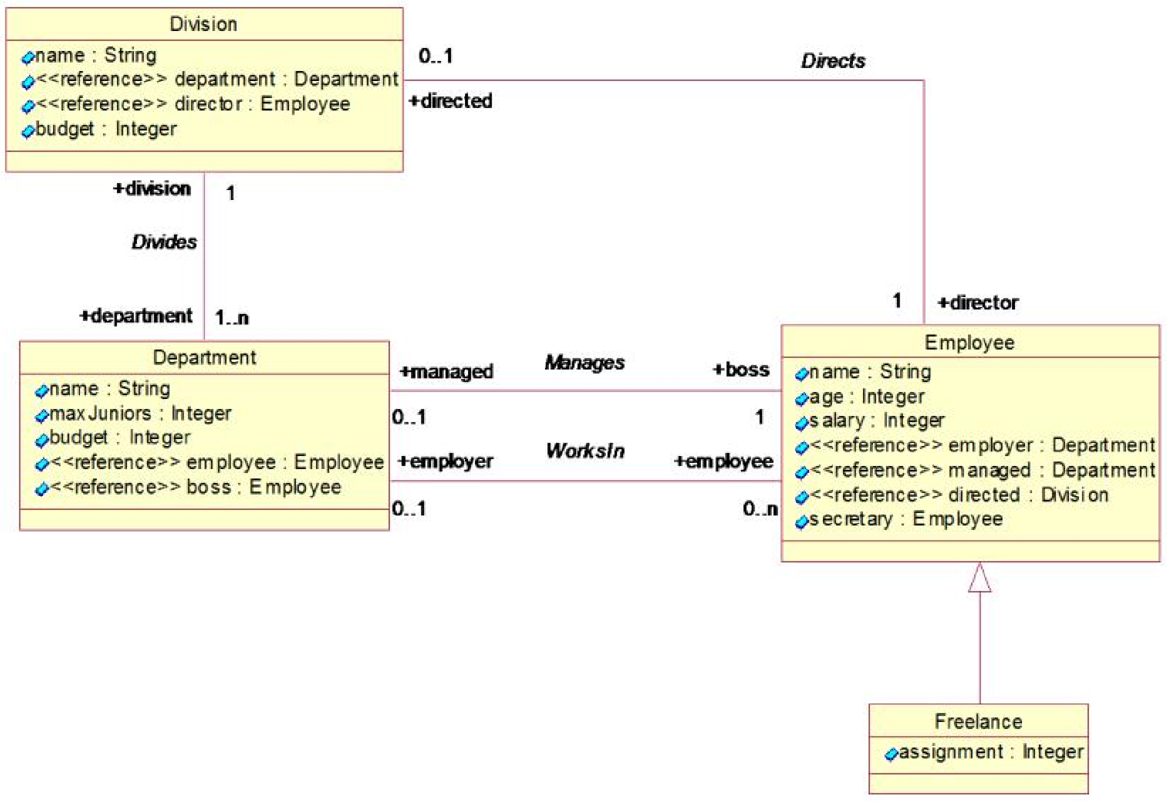
```
        ...
        String yourQuery = "..."; // here you formulate your MQL query in concrete syntax
        MQLResultSet resultSet = mql.execute( yourQuery ); // immediately execute a concrete syntax query
        ...
```

However, if one and the same query is executed many times, it is worthwhile to cache it as a prepared query because the preparation checks and formats the query according to the deployed metamodels. Note however that MQL currently does not cache prepared queries itself.

Further details on the nature of the possible queries, additional scope restrictions, type and execution problems are detailed in the sections to come.

Example Metamodel

MQL can be used as soon as a connection is available because a connection defines the client-scope on which a query will be executed. Explanations on how to obtain a connections are given elsewhere. However, before you can query the repository, you have to have metamodels which describe the content in the repository. All further examples in this user guide are based on the *Company* metamodel.



These classes are all contained in the top-level MOF package **`Company`**. For the sake of the example, we will also assume that the metamodel is deployed in the container **`sap.com/tc/moin/test/company`**. Note that in an NWDI environment, container's coincide with DCs (Development Components). In the NWDI-Eclipse integration of MOIN, there is a *deployed metamodel viewer* which lists the DC names of all available metamodels. You should use this if you are unsure in what container your metamodel is deployed.

Example Queries and Main Features

In this section, we show the most important MQL features by means of series of examples in MQL's concrete syntax. For each example, we provide code on how to implement them in Java with the MOIN API. All examples produce result sets, but we will not show how a result set is deconstructed until later. We always assume that some connection with the right client-scope is available.

Elements and Attributes

The simplest MQL query is one which obtains the instances of a type and perhaps one or more attributes on top of that.

E.g. Return all employees and their names, but not those who freelance.

```
select em, em.name
from “sap.com/moin/test/company”#Company::Employee withoutsubtypes as em
```

In Java, this is achieved as follows:

```
        String query = "select em, em.name " +
                       "from \"sap.com/tc/moin/test/company\"#Company::Employee withoutsubtypes as em";
        MQLProcessor mql = myConnection.getMQLProcessor();
        MQLPreparedQuery preparedQuery = mql.prepare(query);
        MQLResultSet resultSet = mql.execute(preparedQuery);
```

Comments

- This query produces a table with two columns. In the first column, the actual employee elements are stored. In the second column, their associated names are stored as strings.
- Elements in a result set are always encoded via their MRIs. However, it is also possible to retrieve them as RefObjects as we shall see in the section on result sets.
- The from clause illustrates several important aspects:
  - The type, whose instances are taken, is always given by its fully qualified name (in this case **`Company::Employee`**) and optionally also the container name of the metamodel (in this case **`sap.com/tc/moin/test/company`**). If the metamodel container name is not given, you have to be sure that the fully qualified name is unique over all deployed metamodels.
  - The keyword **`withoutsubtypes`** avoids that instances of sub classes are also taken into the result set. This is akin to JMI's **`refAllOfClass()`**. Alternatively, if you want those instances as well, which is the default, simply drop the keyword. This is akin to JMI's **`refAllOfType()`**.
  - The alias (here **`em`**) stands for the type in the from-clause and is referenced by name in other parts of the query, for instance in the selection. Unlike SQL, aliases are mandatory in MQL.
- A container-name rarely follows the standard identifier rules because of punctuation and odd symbols (MQL identifier rules are almost identical to those of Java). Therefore, such identifiers have to be escaped by means of the double quote ("). Observe that the double quote itself has to be escaped in Java strings leading to the preceding backslashes. Of course any other identifier in an MQL query can be escaped in the same way. This is necessary when, e.g. a meta-class name or attribute name accidentally contains odd punctuations (such as spaces) or coincides with an MQL keyword.
- It is possible to be more specific about which subtypes should be excluded when using the **`withoutsubtypes`** keyword. For example,

  ```
  select em, em.name
  from “sap.com/moin/test/company”#Company::Employee withoutsubtypes {Company::Freelance} as em
  ```

  considers all employees which are not freelance (in this example, this happens to coincide with just using the **`withoutsubtypes`** keyword). The set accompanying the **`withoutsubtypes`** keyword cannot be empty, but may contain more than one type. . Observe that only instances of the provided types in the set are excluded, not its subtypes (which you have to add yourself if you want to exclude those as well). It is not permitted to exclude the top-level type of the from-clause itself. So,

  ```
  select em, em.name
  from “sap.com/moin/test/company”#Company::Employee withoutsubtypes {Company::Employee} as em
  ```

  is illegal.

Local Attribute Constraints

Usually, queries are further constrained by all sorts of conditions on the provided types in the from clause. A common type of constraint is local on the value of one of the attributes.

E.g. Return all the names of employees who are younger than 40 or whose names do not start with s

```
select em.name
from Company::Employee as em
```

```
where for em(age < 40 or not(name like 's*'))
```

In Java, this is achieved as follows:

```
        String query = "select em.name from " +
                       "Company::Employee as em where for em(age < 40 or not(name like 's*'))";
        MQLResultSet resultSet = myConnection.getMQLProcessor().execute(query);
```

Comments

- In this example, we have left out the container name for the Company metamodel
- The syntax for local attributes is somewhat unusual because of the **`for`** keyword, but reflects the fact that an arbitrary logical conjunction (using **`not`**, **`and`**, and **`or`**) is only permitted for one type (or alias) at a time.
    - If a query does not **`or`**-connect local attribute constraints (or negates them), a slightly more convenient syntax can be used:

      ```
      where em.age < 40
      ```

    - No connectors (such as **`or`** and **`and`**) can be used with this convenience syntax.
    - In the rest of the guide, we will use this more convenient form where possible.
- Strings are single-quoted, not double-quoted (those are reserved for escaping identifiers). These lexical rules are identical to those of OCL!
- The **`like`** operation is only permitted on strings. It is reminiscent of the **`like`** operation provided by SQL. There are two wildcards which can occur arbitrary often within one pattern:
    - **`?`** stands for exactly one character
    - **`*`** stands for any (possibly empty) sequence of characters
- Attributes can be compared to literals of the 6 standard MOF types (Boolean, String, Integer, Long, Float, Double):
    - For Boolean literals, only the = operation is permitted
    - For String literals, only the operations =, <>, and **`like`** and **`not like`** are permitted
    - For the other types, the operations `=, <>, > >=, <,`and `<=` are permitted
- Attributes of type String can also be compared for equality (or inequality) with **`null`**, i.e. whether the value is assigned or not (all other primitive types have default values, even if the attribute was not explicitly set).

Joining Association Constraints

Generally, queries are formulated over more than one type, where there is a constraint between the two types via some association.

E.g. Return all the company's divisions and its departments where the division's name is 'NetWeaver' and the department's name does not start with 'N'

```
select div, dep
from Company::Division as div,
     Company::Department as dep
where div.department = dep
where div.name = 'NetWeaver'
where dep.name not like 'N*'
```

In Java, this achieved as follows:

```
        String query = "select div, dep " +
                       "from \"sap.com/tc/moin/test/company\"#Company::Division as div, " +
                       "\"sap.com/tc/moin/test/company\"#Company::Department as dep " +
                       "where div.department = dep " +
                       "where div.name = 'NetWeaver' " +
                       "where dep.name not like 'N*'";
        MQLResultSet resultSet = myConnection.getMQLProcessor().execute(query);
```

Comments

- Multiple from clause entries are comma-separated.
- Multiple where clauses repeat the **`where`** keyword and are logically AND-connected.
- Local constraints on different types (or aliases) have to be provided in different where-clauses.
- The first where-constraint above is a joining association constraint. It says that for each instance assigned to **`div`**, by navigating via the reference **`department`**, has to connect to an instance of **`dep`**.
- A joining association constraint permits navigation over references and association ends, even if they are ambiguous. In this case, the association also has to be provided. The following sample where-clauses are valid alternatives for the where-clause above:
    - `where dep.division = div`
    - `where dep.division[Company::Divides] = div`
    - `where div.department[Company::Divides] = dep`
- The alternatives with the qualifying association would be mandatory if more than one association connects the **`Department`** and **`Division`** types via association ends with identical names (which is generally permitted in MOF metamodels).
- It is also possible to check if a reference (or link) exists at all by comparing with **`null`**. This is formulated as follows:
    - `where div.department = null`
- However, this only works if department is a reference. If you want to use an association-end (i.e. without an existing reference), you have to fully qualify the navigation with the association name. In the example, if you want to know if a department is connected to a division, you have to write the following:
    - `where dep.division[Company::Divides] = null`

- It is worth mentioning that the multiplicity of an association end or attribute does not affect the syntax of a joining association constraint. In the above example, there are many departments for a given division. Thus, the clause **`where`** `div.department = dep` should be read as follows: "for a given division (`div`) and department (`dep`) in the Cartesian product, there has to exist a link between them via the association end department on the side of the division".

Joining Attribute Constraints

MQL does not only support joining constraints via associations, but also permits the comparison of primitive-typed attributes over different types.

E.g. Return the budgets of all departments and divisions where the department has a larger budget than the division

```
select div.budget, dep.budget
from Company::Division as div,
     Company::Department as dep
where div.budget < dep.budget
```

In Java, this is achieved as follows:

```
        String query = "select div.budget, dep.budget " + like
                       " from Company::Division as div, Company::Department as dep " +
                       "where div.budget < dep.budget";
        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

Comments

- Joining attribute constraints can appear in conjunction with any other number of constraints, but it is also possible to have them just by themselves.
- Joining attribute constraints can be formulated on any of the 6 primitive types (Boolean, String, Integer, Long, Float, Double).
- Comparisons on the scalar types Integer, Long, Float and Double permit all standard comparison operations (`=, <>, > >=, <, <=`).
- Comparisons on the types String and Boolean only permit the operations `=` and `<>`.

Nested Association Constraints

In MQL, it is also possible to *nest* queries inside constraints. This is the only MQL-mechanism for negative joining constraints.

E.g. Return the names of departments, which do not belong to a division whose budget is more than 1.000.000 euros.

```
select dep.name
from Company::Department as dep
where dep.division[Company::Divides]
      not in (select div
              from Company::Division as div
              where div.budget > 1000000)
```

In Java, this is achieved as follows:

```
        String query = "select dep.name from Company::Department as dep " +
                       "where dep.division[Company::Divides] not in " +
                       "(select div from Company::Division as div where div.budget > 1000000)";
        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

Comments

- The syntax for nested association constraints is akin to that of joining association constraints, except that the right-hand side is not an alias, but the keyword **`in`** followed by the nested query (which has to be surrounded by parentheses).
- Optionally, as shown in the example, the keyword **`in`** can be preceded by the keyword **`not`** to indicate the negation in the constraint.
- The association, as with joining association constraints, can optionally be provided with a qualified association in order to disambiguate (as we did in this example).
- It is *not* possible to access the aliases of nested query in the surrounding query.
- Moreover, nested queries do not form a namespace, so aliases have to be unique across the entire query, including all other nested queries.

One or More Fixed Elements

Sometimes, it is useful to formulate a query where one of the from-clause entries is just one or more fixed elements. We provide two examples.

E.g. For a given department, provide its employees (including freelance workers).

```
select em
from Company::Employee as em,
     "PF.LocalDevelopment[local]:DCs/test.sap.com/tc/moin/test/_comp/src/test
/Company.mointest#47A333DA4F626722D0D611DCC1848000600FE800" as dep
where dep.employee = em
```

In Java, this is achieved as follows (where we assume **`myDepartment`** is a valid instance of **`Department`**):

```
        String query = "select em.name from \"sap.com/tc/moin/test/company\"#Company::Employee as em, " +
                       "\"" + ( (Partitionable) myDepartment ).get___Mri( ) + "\" as dep " +
                       " where dep.employee = em";

        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

Comments

- Although it is possible to actually write out the MRI of the given department in the query (by using the double quote the escapes), this is not very useful in practice. As shown in the Java example, the MRI is obtained by casting the RefObject to a Partitionable and, after fetching the actual MRI, inserting it in the String with **`toString()`**.
- Note that you still have to escape the result of **`toString()`**, which may be a somewhat cumbersome.
- The element has to be of the correct type as usual or the query will be invalid.

E.g. Return the employees (including freelance workers) of a set of provided departments.

```
select em
from Company::Employee as em,
     Company::Department as dep
     in elements
         {"PF.LocalDevelopment[local]:DCs/test.sap.com/tc/moin/test/_comp/src/test
/Company.mointest#47A333DA4F626722D0D611DCC1848000600FE800",
          "PF.LocalDevelopment[local]:DCs/test.sap.com/tc/moin/test/_comp/src/test
/Company.mointest#47A333DA4F626723D0D611DC8BD68000600FE800"}
where dep.employee = em
```

In Java, this is achieved as follows (where we assume **`myDepartment`** and **`mySecondDepartment`** are valid instances of **`Department`**):

```
        String query = "select em.name from \"sap.com/tc/moin/test/company\"#Company::Employee as em, " +
                       "Company::Department as dep in elements{\"" +
                       ( (Partitionable) myDepartment ).get___Mri( ) + "\", \"" +
                       ( (Partitionable) mySecondDepartment ).get___Mri( ) + "\"} " +
                       "where dep.employee = em";

        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

Comments

- The main difference between this example and the previous example is that if more than one element is provided, the user has to specify the common supertype.
- If only one fixed element is given, MQL implicitly derives the immediate type of the given element.
- Observe that the syntax demands a comma-separated list of elements enclosed in curly braces and preceded by the keywords **`in elements`**.
- MQL does not support the negative statement **`not in elements {...}`**. However, it is possible to use a negated nested query which returns a fixed set elements. This covers some of the use-cases for **`not in elements {...}`**.

Result Set Operations

So far, we have discussed how to formulate and execute different types of queries. The result of query execution is an object of type **`MQLResultSet`**. In this section, we discuss how to unpack such a result set. In order to explain this, we assume the following query:

```
        String query = "select div, dep, div.name, div.budget, dep.name " +
                   "from \"sap.com/tc/moin/test/company\"#Company::Division as div, " +
                   "\"sap.com/tc/moin/test/company\"#Company::Department as dep " +
                   "where div.department = dep " +
                   "where div.name = 'NetWeaver'";

        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

Size and Result Table Order

Result sets generally do not have an order. However, to guarantee determinism, MQL results are lexicographically sorted from left-to-right as dictated by the columns of the result table. The columns of the result table are exactly in the same order as the selection of the query. So, in the above example, the first sort order criteria is determined by the elements assigned to **`div`**, then by the elements assigned to **`dep`**, etc. When selecting primitive types, sorting is obvious. In the case of elements, it is sufficient to know that they will always be returned in the same order, where the nature of that order is unspecified.

MQL always completely calculates the result set before returning from its execution. It is always possible to query the size from a result set as follows:

```
        int sizeOfResultSet = resultSet.getSize( );
```

Obtaining Elements

Assuming a non-empty result set, the question is now how we can retrieve individual elements from the result set. In the above example, we may want to obtain the divisions and departments. There are two ways to do this:

```
        MRI divisionAsMRI = resultSet.getMri( 0, "div" );
        RefObject division = resultSet.getRefObject( 0, "div" );
```

In both cases, we obtain the first element in the result set (as in Java, we start counting at 0) and identify the desired element by using the alias. However, in the first case, we only retrieve the MRI, whereas in the second case, we obtain a full-blown RefObject. Although client-code will eventually work with RefObjects, it is important to note that MQL itself only retrieves the MRI and (as a convenience service) resolves the MRI to an actual runtime element. The important aspect to understand is that a MRI is only a cheap element reference whereas a RefObject is a full-blown, more expensive, MOIN runtime object, which affects the current state of MOIN's runtime (e.g. by possibly loading a partition).

In practice, you'd probably loop over the result set and obtain the entire result tuple (more on attribute retrieval in a moment). However, sometimes it is useful to immediately obtain the entire column of elements. For instance, assume we immediately want to obtain all divisions in the above example, we could write something like this:

```
        MRI[] divisionsAsMRIs = resultSet.getMris( "div" );
        RefObject[] divisions = resultSet.getRefObjects( "div" );
```

Again, there is a choice between obtaining all MRIs or all RefObjects. Be aware that obtaining all RefObjects loads all relevant partitions in which these objects reside into MOIN's runtime. You may consider allowing selections on returned attribute values before resolving one of the returned MRIs into a RefObject. Also note that this operation does not remove duplicates, but returns an array of exactly the size of the result set. So, if, in the example a division has multiple departments, these arrays will contain duplicate MRI references of duplicate RefObjects (of course, pointing to the same runtime instance).

Obtaining Attributes

In our example query, we have also selected attributes. If you wonder why that is useful - after all, we could obtain attribute values via JMI APIs on the returned RefObject - it is essentially the same reason as to why we permit the retrieval of MRI's instead of full-blown RefObjects. In the example, for instance, we could display the names of the divisions and departments with the associated budget in a user-dialog and only retrieve the RefObject once the user has made a selection. Client-code could then continue to display more attributes or permit its alteration. But the key point is that no MOIN partitions are loaded into the runtime until the RefObject is requested. Loading a partition into memory is more flexible and even required if you have edit the model data, but always more expensive in terms of resources.

Obtaining attributes is analogous to obtaining elements, except that you also have to provide the attribute name. You can only retrieve attributes whose type is one of the 6 primitive types, which the API returns as **`java.lang.Object`**. So you have to downcast to the appropriate (boxed) Java primitive type.

```
        String departmentName = (String) resultSet.getAttribute( 0, "dep", "name" );
        Integer budget = (Integer) resultSet.getAttribute( 0, "div", "budget" );
```

For convenience, it is also possible to immediately obtain all attributes of an alias in a result row. The attributes are always returned in the order in which they appear in the result selection.

```
        Object[] divisionAttributes = (String[]) resultSet.getAttributes( 0, "div" );
        String divisionName = (String) divisionAttributes[0];
        Integer divisionBudget = (Integer) divisionAttributes[1];
```

It is also possible to obtain multi-valued attributes. Our metamodel does not have an example for that, but the type of such an attribute value is **`java.lang.Object[]`**.

Reflection on Result Set Structure

If for some reason you do not have static control over the exact format of the selection of your query, for instance, because you automatically generated your query, or your query was entered by an interactive dialog, you can still obtain the structure of the result set by invoking a reflective method:

```
        MQLColumnType[] columnTypes = resultSet.getQueryColumnTypes( );
        MQLColumnType firstColumn = columnTypes[0];
        String aliasName = firstColumn.alias;
        String attributeNameOrNull = firstColumn.attribute;
        String primitiveTypeName = firstColumn.typeName;
        boolean isMultivalued = firstColumn.multiValued;
        boolean isOrderedMultivalued = firstColumn.isOrdered;
        boolean isUniqueMultiValued = firstColumn.isUnique;
```

As can be seen from this sample code, each column is represented by an **`MQLColumnType`**. This column type contains information about the nature of the column, for instance, what is the involved alias and optionally what is the attribute name. This information is required to retrieve results from the result set, as explained in the previous paragraphs.

If the column contains primitive typed attributes, you can obtain this type as a String and if a column contains multi-valued attributes, the column type allows you to query if this is the case and whether it is ordered and/or unique.

Printing and Exporting

Finally, an MQL result set implements a **`toString()`** method which produces a pretty-printed table of the result set. This is particularly helpful during debugging. In fact, the **`toString()`** method is a convenient default for the more general **`asCSV(Writer writer)`** method, which produces a comma-separated value output to the provided Java **`Writer`**. This is useful to export result sets and read them, for instance with Excel.

Scope Providers

By default, MQL queries are always executed over the entire client-scope of a connection. Although in many scenarios, this may be sufficient, there is often a need to reduce the scope of a query (or some of its from-clause parts) over a fixed number of partitions or containers. MQL provides a flexible mechanism to define such scopes by allowing clients to implement a so-called **`QueryScopeProvider`** interface. The most common query scope providers are available via the MQL processor API (see below).

The `QueryScopeProvider` Interface

The **`QueryScopeProvider`** is a public API interface, which MOIN clients can implement themselves to provide for custom scopes. It has the following form:

```
        public interface QueryScopeProvider {
            boolean isInclusiveScope( );
            PRI[] getPartitionScope( );
            CRI[] getContainerScope( );
        }
```

An implementation of this interface has to decide whether the produced scope provider is *inclusive*, i.e. the scope should be restricted to the partitions and containers provided by the other two methods, or, whether a scope is *exclusive*, i.e. the scope excludes the partitions and containers provided by the other two methods. The scope (be it inclusive or exclusive) is then specified by the union of partitions and containers (identified by PRIs and CRIs respectively) as produced by the methods **`getPartitionScope()`** and **`getContainerScope()`**. Internally, the MQL execution engine will convert each CRI of a scope into a set of PRIs just before the query is executed. If a particular scope provider implementation only wants to provide CRIs, the **`getPartitionScope()`** method should return an empty array (**`null`** is not a valid return value here). Analogously for the other scenario where only PRIs are provided.

Consider the following example method, which returns a query scope provider defining an inclusive container scope:

```
    public QueryScopeProvider getInclusiveCriScopeProvider( final CRI... containerScope ) {

        QueryScopeProvider queryScopeProvider = new QueryScopeProvider( ) {

            public boolean isInclusiveScope( ) {
                return true;
            }

            public CRI[] getContainerScope( ) {
                if ( containerScope == null ) {
                    return new CRI[0];
                } else {
                    return containerScope;
                }
            }

            public PRI[] getPartitionScope( ) {
                return new PRI[0];
            }
        };

        return queryScopeProvider;
    }
```

Standard Scope Providers

MQL provides a set of standard scope provider via its main API. These are sufficient for the most common use cases and avoids that clients have to keep re-inventing the wheel. Here is a brief summary of the most important ones. For more variations, please consult the JavaDoc of the **`MQLProcessor`**.

```
    QueryScopeProvider getQueryScopeProvider( boolean scopeInclusive, PRI[] partitionScope, CRI[] containerScope );
```

This is the most general scope provider because it directly maps on the **`QueryScopeProvider`** methods. You can use it to be flexible with inclusion/exclusion and provide both PRIs and CRIs to delimit the scope.

```
    QueryScopeProvider getInclusivePartitionScopeProvider( PRI... partitionScope );
```

As natural simplification, this method returns a query scope provider which delimits the execution of a query within the given partitions.

```
    QueryScopeProvider getInclusiveCriScopeProvider( CRI... containerScope );
```

Analogously, this method returns a query scope provider which delimits the execution of a query within the given CRIs (i.e. containers). Keep in mind that the MQL processor only determines the partitions for the given CRIs just before each execution.

```
    QueryScopeProvider getInclusiveVisibleCriScopeProvider( CRI cri );
```

This is a somewhat more sophisticated query scope provider method. It calculates the scope by combining all partitions in the provided CRI with all partitions in public parts of used containers (in NWDI, this would be *used DCs*). Note that the scope also includes the null-partition. This query scope provider is commonly needed in an NWDI-deployment landscape.

Scope upon Execution

So, far, we have seen how scopes can be defined and obtained. It is now straightforward to add a concrete scope provider when executing a query. MQL provides the following three methods to do this:

```
    MQLResultSet execute( MQLPreparedQuery query,
                          QueryScopeProvider scopeProvider ) throws MQLExecutionException;

    MQLResultSet execute( MQLQuery query,
                          QueryScopeProvider scopeProvider ) throws MQLExecutionException, MQLFormatException;

    MQLResultSet execute( String query,
                          QueryScopeProvider scopeProvider ) throws MQLExecutionException, MQLFormatException;
```

These variants permit the execution of a prepared query (**`MQLPreparedQuery`**), an abstract syntax query (**`MQLQuery`**), and a concrete syntax query (**`String`**).

Scoping Individual Types

Although in almost all scenarios, you will want to provide the query scope upon execution, there are some use-cases where individual entries in the from-clause have to be provided with individual (and probably different) scopes. The following example illustrates how to do this in the concrete syntax (we assume a given **`myCri`** and **`myPri`**):

```
        String query = "select dep, em " +
                       "from Company::Department as dep in containers{\"" + myCri + "\"}, " +
                       "Company::Employee as em not in partitions{\"" + myPri + "\" }" +
                       "where em.employer = dep";
```

Comments

- The syntax is almost identical to that of a fixed element set, except that you use the key-words **`partitions`** and **`containers`**
- Also, instead of inserting MRIs, you insert PRIs or CRIs (depending on what kind of scope you want to insert).
- Unlike query scope providers, in the concrete syntax, there is no provision to mix PRIs and CRIs in the scope definition.
- It is still possible to define an exclusive scope by writing **`not in partitions`** instead of **`in partitions`** (and analogous for container scopes)
- The concrete syntax does not permit the usage of generic query scope providers. If you still need this, and individually different for parts of the from-clause, you have to use the abstract syntax to specify your MQL query.

Other MQL Features

There are several MQL features which fall outside the example, but are useful in practice. We list the most significant here.

Enumerations and Enumeration Labels

MOF, and thus MOIN, supports enumeration types, which have a fixed number of labels. MQL treats enumeration-typed attributes as if they were strings.

Structure Types and Structure Fields

From MQL's perspective MOF structure types are almost identical to MOF classes. The only difference is that structure types cannot exist by themselves, i.e. there is no such thing a structure type instance. This has the following consequences:

- It is illegal to put the alias bound to a structure type in the selection clause.
- Structure fields are treated like attributes. Therefore, if they are of primitive type, they can be selected in a selection-clause and they can be used to expression a condition in where-clause, etc.
- Because values of structure types are not really instances, MOIN decides itself how to manage these values within one ore more partitions. Therefore, it is pointless to provide a scope for a from-clause entry of a structure type (MQL will simply ignore it)
- It is not possible to define a query with one structure type in the from-clause, or, if the from-clause also contains a regular MOF-class, without having the structure type constrained to the MOF-class in where-clause. This is obvious as long as you remember that there is no such thing as a structure type instance.
- Because we do not permit the alias to be bound to a structure type in a selection clause, it is not possible to have a structure type as the connecting type in a joining where-constraint with a nested query.
- It is also not possible to check if a value of structure type is not set (by comparing it to **`null`**).

Conditions on Multi-valued Attributes

In MOF, and thus in MOIN, primitive-typed attributes can be multi-valued. When you define a where-clause on a multi-valued attribute, the condition is true as soon as one of the attribute values satisfies the comparison. Suppose we have a multi-valued attribute **`multiAttribute`** of type Integer.

```
        where myClass.multiAttribute = 5
```

An instance for **myClass** will be considered as soon as one of the attribute values is equal to 5. Note that this behavior applies to all operations, also **>**, etc.

Similarly, it is also possible to compare a multi-valued attribute to another attribute. If that other attribute is single-valued, then the behavior is analogous. But if the other attribute is also multi-valued, the where-clause is positive as soon as one element in one set compares positively to one element in the other set. For instance, in

```
        where myClass.multiAttribute > otherClass.otherMultiAttribute
```

it is sufficient that one element's **multiAttribute** is greater than another element's **otherMultiAttribute**!

Reflect::Element

MOIN supports a special custom type **Reflect::Element** which is an implicit super-type of all elements. This type is useful for tools who link data from arbitrary metamodels. MQL supports queries which include **Reflect::Element** or subtypes thereof.

However, this feature has to be used with caution. If the following query is executed without a scope provider, it would return the entire repository!

```
        select entireRepository
        from Reflect::Element as entireRepository
```

Because this query is so problematic, it is actually forbidden to write it like that without providing an inclusive scope over some containers or partitions. In practice, queries involving **Reflect::Element** usually have an association navigation from a regular type to it, automatically constraining the number of returned elements.

From-clause entries as MRIs instead of Qualified Names

In some rare cases, a client-tool may not have the qualified name of a type it wants to use in a from-clause. This might be the case, for instance, when the type is the result of a query at a higher meta-level. In such scenarios, it is also possible to use the MRI of the type in the from-clause. In order to avoid confusion with a from-clause entry for one fixed element, the **type** keyword has to precede the MRI. Here is an example:

```
        String query = "select em.name from \"sap.com/tc/moin/test/company\"#Company::Employee as em, " +
                       " type \"" + ( (Partitionable) myDepartment.refMetaObject() ).get___Mri( ) +
                       "\" as dep where dep.employee = em";

        MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query );
```

In this case, **myDepartment** is an M1 object. Its type (obtained by calling **refMetaObject()**) is again a **partitionable** from which we can obtain the MRI. By preceding the entry with **type**, we indicate that we are interested in all instances of the type of **myDepartment** (which in this case would be **Department**).

Reduce the number of requested result-set rows

Generally, the MQL processor will try to obtain all valid result-set rows, which fulfill the constraints of the provided query. Sometimes, it is useful to only obtain a certain number of result set rows. For example, when a client is only interested in the existence of a result(independent of its value or values), it is sufficient to ask for one result set row.

This can be achieved by adding one extra parameter to the `execute` method. For example:

```
MQLResultSet resultSet = myConnection.getMQLProcessor( ).execute( query, 1 );

boolean hasAtLeastOneResult = !resultSet.isEmpty();
```

This is more efficient than obtaining all results because MQL will stop its calculations as soon as one result is found and the size of the result-set will never exceed 1. Observe that MQL makes no guarantees about *which* element is returned, so use this feature with caution.

Problem Resolution

When writing MQL queries, there are two possible error classes which you have to consider (see below). They come up either as **MQLFormatException** or **MQLExecutionException**. These are Java runtime exceptions and thus not checked. If you get any other kind of exception, you've probably encountered a bug and you should report it per CSN.

Syntax and Type Errors

Syntax and type errors are the most common kind of error, especially when you have not actually tested your queries. They manifest themselves in an **MQLFormatException**. Such an exception can only occur during the preparation of a query (or execution if you do not first prepare your query). The list of possible format exceptions is too long to describe here. However, the exception message will show a preparation report indicating one or more problems that occurred while trying to prepare the query. If you use the concrete syntax, the report will also provide line and column information, which permits a more accurate identification of the problem at hand. The preparation report can also be obtained explicitly:

```
        try {
```

```
        ...
    } catch (MQLFormatException e) {
        ProcessReport report = e.getPreparationReport();
    }
```

Consult the JavaDoc for further details.

Execution Problems and Result Set Sizes

In most cases, a prepared MQL query is executable without problems. The only exception is when the result set is too large. MQL demands that you put a limit on the size of the result set, which by default is 10000. This is to avoid a JVM crash because of an out-of-heap memory message when you (usually accidentally) produce an extremely large result set. You can adjust and query this maximum result set size with the following API on `MQLProcessor`:

```
    int getMaxResultSetSize( );

    void setMaxResultSetSize( int maxResultSetSize );
```

The most typical scenario in which an unintentional result set explosion happens is when you provide a type in the from-clause which is not linked via an association constraint to any other type in the from-clause. If the unlinked type in the from-clause has many instances, the product set of all entries in the from-clause will be too large.

MQL Concrete Syntax as EBNF

This is the extended BNF for the MOIN Query Language.

```
<MQLquery>                ::=   select <select-clause>
                                from <from-clause>
                                (where <where-entry>)*

<select-clause>           ::=   <select-entry> {, <select-entry>}

<select-entry>            ::=   <alias-identifier> [. <attr-identifier>]

<from-clause>             ::=   <from-entry> (, <from-entry>)*

<from-entry>              ::=   <type-clause> as <alias-identifier> [<scope-clause>]

<type-clause>             ::=   <type-clause-qname> | <type-clause-mri> | <mri-identifier>

<type-clause-qname>       ::=   <fqualified-type-name> [ withoutsubtypes ]

<fqualified-type-name>    ::=   [<container-identifier> #] <qualified-type-name>

<qualified-type-name>     ::=   <type-path-identifier> (:: <type-path-identifier>)*

<type-clause-mri>         ::=   type <mri-identifier> [ withoutsubtypes ]

<scope-clause>            ::=   [ not ] in ( partitions { [<partition-scope-list>] }
                           |                 containers { [<container-scope-list>] } )
                           |    in elements { [<mri-element-list>] }

<where-entry>             ::=   <local-where-entry> | <join-where-entry>

<local-where-entry>       ::=   for <alias-identifier> ( <local-where-condition> )

                           |    <alias-identifier> . <attr-identifier> <operation-part>

<local-where-condition>   ::=   not <local-where-condition>
                           |    <local-where-condition> or <local-where-condition> (or <local-where-condition>)*
                           |    <local-where-condition> and <local-where-condition> (and <local-where-condition>)*
                           |    <attr-identifier> <operation-part>
                           |    ( <local-where-condition> )

<operation-part>          ::=   <numeric-operation-part>
                           |    <boolean-operation-part>
                           |    <string-operation-part>

<numeric-operation-part>  ::=   <primitive-operation> <numeric-literal>

<primitive-operation>     ::=   < | > | <= | >= | = | <>

<boolean-operation-part>  ::=   = true | = false

<string-operation-part>   ::=   <string-operation> <string-literal>

                           |    ( = | <> ) null

<string-operation>        ::=   = | <> | like | not like

<join-where-entry>        ::=   <assoc-predicate>
                           |    <link-predicate>
                           |    <comparison-predicate>

<assoc-predicate>        ::=   <alias-identifier> . <navigation-clause> ( = <alias-identifier> | = null | <> null )

<link-predicate>         ::=   <alias-identifier> . <navigation-clause> [ not ] in ( <MQLquery> )
```

```
<navigation-clause>     ::=   <assoc-end-identifier> [ [ <fqualified-assoc-name> ] ]
                         |    <reference-identifier>
                         |    <attr-identifier>

<fqualified-assoc-name> ::=   [<container-identifier> #] <qualified-assoc-name>

<qualified-assoc-name>  ::=   <assoc-path-identifier> [:: <assoc-path-identifier>]

<comparison-predicate>  ::=   <alias-identifier> . <attr-identifier> <primitive-operator> <alias-identifier> . <attr-identifier>
                         |    <alias-identifier> = <alias-identifier>
```