|  |
| --- |
| **JDO Named Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

With the JDO2 API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called *Product*(something to sell in a store). We define the JDO Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

<jdo>

<package name="org.datanucleus.example">

<class name="Product">

...

<query name="SoldOut" language="javax.jdo.query.JDOQL"><![CDATA[

SELECT FROM org.datanucleus.example.Product WHERE status == "Sold Out"

]]></query>

</class>

</package>

</jdo>

So we have a JDOQL query called "SoldOut" defined for the class *Product*that returns all Products (and subclasses) that have a *status*of "Sold Out". Out of interest, what we would then do in our application to execute this query woule be

Query query=pm.newNamedQuery(org.datanucleus.example.Product.class,"SoldOut");

Collection results=(Collection)query.execute();

The above example was for the JDOQL object-based query language. We can do a similar thing using SQL, so we define the following in our MetaData for our *Product*class

<jdo>

<package name="org.datanucleus.example">

<class name="Product">

...

<query name="PriceBelowValue" language="javax.jdo.query.SQL"><![CDATA[

SELECT NAME FROM PRODUCT WHERE PRICE < ?

]]></query>

</class>

</package>

</jdo>

So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named query, asking for the names of all Products with price below 20 euros.

Query query=pm.newNamedQuery(org.datanucleus.example.Product.class,"PriceBelowValue");

Collection results=(Collection)query.execute(20.0);

All of the examples above have been specifed within the <class> element of the MetaData. You can, however, specify queries below <jdo> in which case the query is not scoped by a particular candidate class. In this case you must put your queries in any of the following MetaData files

/META-INF/package.jdo

/WEB-INF/package.jdo

/package.jdo

/META-INF/package-{mapping}.orm

/WEB-INF/package-{mapping}.orm

/package-{mapping}.orm

/META-INF/package.jdoquery

/WEB-INF/package.jdoquery

/package.jdoquery

Please proceed to the sections specific to [**JDOQL**](http://www.datanucleus.org/products/accessplatform_1_1/jdo/jdoql.html)and [**SQL**](http://www.datanucleus.org/products/accessplatform_1_1/jdo/sql.html)for details on the precise nature of the query for the querying languages supported by DataNucleus.

|  |
| --- |
| **JDOQL Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

JDO defines ways of querying objects persisted into the datastore. It provides its own object-based query language (JDOQL). JDOQL is designed as the Java developers way of having the power of SQL queries, yet retaining the Java object relationship that exist in their application model. DataNucleus provides all functionality required by the JDO 1.0 and 2.0 specifications as well as providing a series of vendor extensions. A typical JDOQL query may be set up in one of 2 ways. Here's an example

Declarative JDOQL :

Query q = pm.newQuery(org.datanucleus.Person.class, "lastName == \"Jones\" && age < age\_limit");

q.declareParameters("double age\_limit");

List results = (List)q.execute(20.0);

Single-String JDOQL :

Query q = pm.newQuery("SELECT FROM org.datanucleus.Person WHERE lastName == \"Jones\"" +

" && age < :age\_limit PARAMETERS double age\_limit");

List results = (List)q.execute(20.0);

So here in our example we select all "Person" objects with surname of "Jones" and where the persons age is below 20. The language is intuitive for Java developers, and is intended as their interface to accessing the persisted data model. As can be seen above, the query is made up of distinct parts. The class being selected (the SELECT clause in SQL), the filter (which equates to the WHERE clause in SQL), together with any sorting (the ORDER BY clause in SQL), etc.

|  |
| --- |
| **Single-String JDOQL** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

In traditional (declarative) JDOQL (JDO 1.0) it was necessary to specify the component parts (filter, candidate class, ordering, etc) of the query using the mutator methods on the Query. In JDO 2 you can now specify it all in a single string. This string has to follow a particular pattern, but provides the convenience that many people have been asking for. The pattern to use is as follows

SELECT [UNIQUE] [<result>] [INTO <result-class>]

[FROM <candidate-class> [EXCLUDE SUBCLASSES]]

[WHERE <filter>]

[VARIABLES <variable declarations>]

[PARAMETERS <parameter declarations>]

[<import declarations>]

[GROUP BY <grouping>]

[ORDER BY <ordering>]

[RANGE <start>, <end>]

The "keywords" in the query are shown in UPPER CASE but can be in *UPPER*or *lower*case.

Lets give an example of a query using this syntax

SELECT UNIQUE FROM org.datanucleus.samples.Employee ORDER BY departmentNumber

so we form the parts of the query as before, yet here we just specify it all in a single call.

|  |
| --- |
| **Accessing Fields** |

|  |
| --- |
|  |

In JDOQL you access fields in the query by referring to the field name. For example, if you are querying a class called *Product*and it has a field "price", then you access it like this

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class, "price < 150.0");

In addition to the persistent fields, you can also access "public static final" fields of any class. You can do this as follows

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,

"taxPercent < org.datanucleus.samples.store.Product.TAX\_BAND\_A");

So this will find all products that include a tax percentage less than some "BAND A" level. Where you are using "public static final" fields you can either fully-qualify the class name or you can include it in the "imports" section of the query (see later).

|  |
| --- |
| **Data types : literals** |

|  |
| --- |
|  |

JDOQL supports the following literals: IntegerLiteral, FloatingPointLiteral, BooleanLiteral, CharacterLiteral, StringLiteral, and NullLiteral.

|  |
| --- |
| **Operators precedence** |

|  |
| --- |
|  |

The following list describes the operator precedence in JDOQL.

1. Cast
2. Unary ("~") ("!")
3. Unary ("+") ("-")
4. Multiplicative ("\*") ("/") ("%")
5. Additive ("+") ("-")
6. Relational (">=") (">") ("<=") ("<") ("instanceof")
7. Equality ("==") ("=!")
8. Boolean logical AND ("&")
9. Boolean logical OR ("|")
10. Conditional AND ("&&")
11. Conditional OR ("||")

|  |
| --- |
| **Concatenation Expressions** |

|  |
| --- |
|  |

The concatenation operator(+) concatenates a String to either another String or Number. Concatenations of String or Numbers to null results in null.

|  |
| --- |
| **Example 1 - Use of Explicit Parameters** |

|  |
| --- |
|  |

Here's a simple example for finding the elements of a class with a field below a particular threshold level. Here we pass in the threshold value ( *limit*), and sort the output in order of ascending price.

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,"price < limit");

query.declareParameters("double limit");

query.setOrdering("price ascending");

List results = (List)query.execute(150.00);

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE " +

"price < limit PARAMETERS double limit ORDER BY price ASCENDING");

List results = (List)query.execute(150.00);

For completeness, the class is shown here

class Product

{

String name;

double price;

java.util.Date endDate;

...

}

<jdo>

<package name="org.datanucleus.samples.store">

<class name="Product">

<field name="name">

<column length="100" jdbc-type="VARCHAR"/>

</field>

<field name="abreviation">

<column length="20" jdbc-type="VARCHAR"/>

</field>

<field name="price"/>

<field name="endDate"/>

</class>

</package>

</jdo>

|  |
| --- |
| **Example 2 - Use of Implicit Parameters** |

|  |
| --- |
|  |

Let's repeat the previous query but this time using *implicit*parameters.

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,"price < :limit");

query.setOrdering("price ascending");

List results = (List)query.execute(150.00);

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE " +

"price < :limit ORDER BY price ASCENDING");

List results = (List)query.execute(150.00);

So we omitted the declaration of the parameter and just prefixed it with a colon (:)

|  |
| --- |
| **Example 3 - Comparison against Dates** |

|  |
| --- |
|  |

Here's another example using the same Product class as above, but this time comparing to a Date field. Because we are using a type in our query, we need to *import*it ... just like you would in a Java class if you were using it there.

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class,

"endDate > best\_before\_limit");

query.declareImports("import java.util.Date");

query.declareParameters("Date best\_before\_limit");

query.setOrdering("endDate descending");

Collection results = (Collection)query.execute(my\_date\_limit);

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product " +

"WHERE endDate > best\_before\_limit " +

"PARAMETERS Date best\_before\_limit " +

"import java.util.Date ORDER BY endDate DESC");

List results = (List)query.execute(my\_date\_limit);

|  |
| --- |
| **Example 4 - Instanceof** |

|  |
| --- |
|  |

This example demonstrates use of the "instanceof" operator. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. Clearly the field "b" of A can be of type B, B1, B2, B3 etc, and we want to find all objects of type A that have the field "b" that is of type B2. We do it like this

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.A.class);

query.setFilter("b instanceof org.datanucleus.samples.B2");

List results = (List)query.execute();

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.A WHERE b instanceof org.datanucleus.samples.B2");

List results = (List)query.execute();

* ariable
* A parameter (though why you would want a parameter returning is hard to see since you input the value in the first place)
* An aggregate (count(), avg(), sum(), min(), max())
* An expression involving a field (e.g "field1 + 1")
* A navigational expression (navigating from one field to another ... e.g "field1.field4")

The result is specified in JDOQL like this

query.setResult("count(field1), field2");

In **Single-String JDOQL**you would specify it directly.

|  |
| --- |
| **Result type** |

|  |
| --- |
|  |

What you specify in the *result*defines what form of result you get back.

* **Object**- this is returned if you have only a single row in the results and a single column. This is achived when you specified either UNIQUE, or just an aggregate (e.g "max(field2)")
* **Object[]**- this is returned if you have only a single row in the results, but more than 1 column (e.g "max(field1), avg(field2)")
* **List<Object>**- this is returned if you have only a single column in the result, and you don't have only aggregates in the result (e.g "field2")
* **List<Object[]>**- this is returned if you have more than 1 column in the result, and you don't have only aggregates in the result (e.g "field2, avg(field3)")

|  |
| --- |
| **Aggregates** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

There are situations when you want to return a single number for a column, representing an aggregate of the values of all records. There are 5 standard JDO2 aggregate functions available. These are

* **avg(val)**- returns the average of "val". "val" can be a field, numeric field expression or "distinct field".
* **sum(val)**- returns the sum of "val". "val" can be a field, numeric field expression, or "distinct field".
* **count(val)**- returns the count of records of "val". "val" can be a field, or can be "this", or "distinct field".
* **min(val)**- returns the minimum of "val". "val" can be a field
* **max(val)**- returns the maximum of "val". "val" can be a field

So to utilise these you could specify something like

Query q = pm.newQuery("SELECT max(price), min(price) FROM org.datanucleus.samples.store.Product WHERE status == 1");

This will return a single row of results with 2 values, the maximum price and the minimum price of all products that have status code of 1.

|  |
| --- |
| **Example 1 - Use of aggregates** |

|  |
| --- |
|  |

JDO 2 introduces the ability to use aggregates in queries. Here's another example using the same Product class as above, but this time looking for the maximum price of products that are CD Players. Note that the result for this particular query will be of type Double since there is a single double precision value being returned via the "result".

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);

query.setFilter("name == \"CD Player\"");

query.setResult("max(this.price)");

List results = (List)query.execute();

Iterator iter = c.iterator();

Double max\_price = (Double)iter.next();

Single-String JDOQL :

Query query = pm.newQuery("SELECT max(price) FROM org.datanucleus.samples.store.Product WHERE name == \"CD Player\"");

List results = (List)query.execute();

Iterator iter = c.iterator();

Double max\_price = (Double)iter.next();

|  |
| --- |
| **JDOQL : Methods** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. This document defines the standard methods available in JDO2. If you look at the datastore-specific implementations you can find some extensions to this list (particularly for [RDBMS](http://www.datanucleus.org/products/accessplatform_1_1/rdbms/jdoql_methods.html)).

|  |  |  |  |
| --- | --- | --- | --- |
| **Java Type** | **Method** | **Description** | **Specification** |
| String | startsWith(String) | Returns if the string starts with the passed string | JDO 1.0, JDO 2.0 |
| String | endsWith(String) | Returns if the string ends with the passed string | JDO 1.0, JDO 2.0 |
| String | indexOf(String) | Returns the first position of the passed string | JDO 2.0 |
| String | indexOf(String,int) | Returns the position of the passed string, after the passed position | JDO 2.0 |
| String | substring(int) | Returns the substring starting from the passed position | JDO 2.0 |
| String | substring(int,int) | Returns the substring between the passed positions | JDO 2.0 |
| String | toLowerCase() | Returns the string in lowercase | JDO 2.0 |
| String | toUpperCase() | Retuns the string in UPPERCASE | JDO 2.0 |
| String | matches(String pattern) | Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method. | JDO 2.0 |
| Collection | isEmpty() | Returns whether the collection is empty | JDO 1.0, JDO 2.0 |
| Collection | contains(value) | Returns whether the collection contains the passed element | JDO 1.0, JDO 2.0 |
| Collection | size() | Returns the number of elements in the collection | JDO 2.0 |
| Map | isEmpty() | Returns whether the map is empty | JDO 1.0, JDO 2.0 |
| Map | containsKey(key) | Returns whether the map contains the passed key | JDO 2.0 |
| Map | containsValue(value) | Returns whether the map contains the passed value | JDO 2.0 |
| Map | get(key) | Returns the value from the map with the passed key | JDO 2.0 |
| Map | size() | Returns the number of entries in the map | JDO 2.0 |
| Math | abs(number) | Returns the absolute value of the passed number | JDO 2.0 |
| Math | sqrt(number) | Returns the square root of the passed number | JDO 2.0 |
| JDOHelper | getObjectId(object) | Returns the object identity of the passed persistent object | JDO 2.0 |

The following sections provide some examples of what can be done using JDOQL methods.

|  |
| --- |
| **Example 1 - Map methods (I)** |

|  |
| --- |
|  |

Here's another example using the same Product class as a value in a Map. This introduces how you query Collection and Map fields using the operations available. Collections and Maps act very similarly. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with the name "product 1".

Declarative JDOQL :

Extent e=pm.getExtent(org.datanucleus.samples.store.Inventory.class,false);

Query query = pm.newQuery(e,"products.containsKey(\"product 1\")");

List results = (List)query.execute();

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Inventory EXCLUDE SUBCLASSES " +

"WHERE products.containsKey(\"product 1\")");

List results = (List)query.execute();

Here's the source code for reference

class Inventory

{

Map products;

...

}

class Product

{

String name;

double price;

double salePrice;

java.util.Date endDate;

String[] composition;

...

}

<jdo>

<package name="org.datanucleus.samples.store">

<class name="Inventory">

<field name="products">

<map key-type="java.lang.String" value-type="org.datanucleus.samples.store.Product"/>

<key mapped-by="name"/>

</field>

</class>

<class name="Product">

<field name="name">

<column length="100" jdbc-type="VARCHAR"/>

</field>

<field name="price"/>

<field name="endDate"/>

</class>

</package>

</jdo>

|  |
| --- |
| **Example 2 - Map methods (II)** |

|  |
| --- |
|  |

We might want to to check if a Collection field contains one or other elements. We extend the previous example that is using the Product class as a value in a Map. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with the name "product 1" or "product 2".

Declarative JDOQL :

Extent e=pm.getExtent(org.datanucleus.samples.store.Inventory.class,false);

Query query = pm.newQuery(e);

query.declareVariables("String productName");

query.setFilter("products.containsKey(productName) && (productName==\"product 1\" || productName==\"product 2\")");

List results = (List)query.execute();

Single-String JDOQL:

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Inventory EXCLUDE SUBCLASSES " +

"WHERE products.containsKey(productName) && (productName==\"product 1\" || productName==\"product 2\") " +

"VARIABLES String productName");

List results = (List)query.execute();

|  |
| --- |
| **Example 3 - String.startsWith() method** |

|  |
| --- |
|  |

Here's another example using the same Product class as above, but this time looking for objects which their abreviation is the begin of a trade name. The trade name is provided as parameter.

Declarative JDOQL :

Query query = pm.newQuery(org.datanucleus.samples.store.Product.class);

query.declareImports("import java.lang.String");

query.declareParameters("java.lang.String tradeName");

query.setFilter("tradeName.startsWith(this.abbreviation)");

List results = (List)query.execute("Workbook Advanced");

Single-String JDOQL :

Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product " +

"WHERE tradeName.startsWith(this.abbreviation) " +

"PARAMETERS java.lang.String tradeName import java.lang.String");

List results = (List)query.execute("Workbook Advanced");

|  |
| --- |
| **JDOQL Subqueries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2_1.gif

With JDOQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JDO 2.1 provides a very useful addition to JDOQL adding on subqueries, so that both calculations can be performed in one query. Here's an example, using single-string JDOQL

SELECT FROM org.datanucleus.Employee WHERE salary >

(SELECT avg(salary) FROM org.datanucleus.Employee e)

So we want to find all Employees that have a salary greater than the average salary. In single-string JDOQL the subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "e", whereas in the outer query the alias is "this".

We can specify the same query using the JDOQL API, like this

Query averageSalaryQuery = pm.newQuery(Employee.class);

averageSalaryQuery.setResult("avg(this.salary)");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");

q.declareVariables("double averageSalary");

q.addSubquery(averageSalaryQuery, "double averageSalary", null, null);

List results = (List)q.execute();

So we define a subquery as its own Query (that could be executed just like any query if so desired), and the in the main query have an implicit variable that we define as being represented by the subquery.

|  |
| --- |
| **Referring to the outer query in the subquery** |

|  |
| --- |
|  |

JDOQL subqueries allows use of the outer query fields within the subquery if so desired. Taking the above example and extending it, here is how we do it in single-string JDOQL

SELECT FROM org.datanucleus.Employee WHERE salary >

(SELECT avg(salary) FROM org.datanucleus.Employee e WHERE e.lastName == this.lastName)

So with single-string JDOQL we make use of the alias identifier "this" to link back to the outer query.

Using the JDOQL API, to achieve the same thing we would do

Query averageSalaryQuery = pm.newQuery(Employee.class);

averageSalaryQuery.setResult("avg(this.salary)");

averageSalaryQuery.setFilter("this.lastName == :lastNameParam");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");

q.declareVariables("double averageSalary");

q.addSubquery(averageSalaryQuery, "double averageSalary", null, "this.lastName");

List results = (List)q.execute();

So with the JDOQL API we make use of parameters, and the last argument to *addSubquery*is the value of the parameter *lastNameParam*.

|  |
| --- |
| **Candidate of the subquery being part of the outer query** |

|  |
| --- |
|  |

There are occasions where we want the candidate of the subquery to be part of the outer query, so JDOQL subqueries has the notion of a *candidate expression*. This is an expression relative to the candidate of the outer query. An example

SELECT FROM org.datanucleus.Employee WHERE this.weeklyhours >

(SELECT AVG(e.weeklyhours) FROM this.department.employees e)

so the candidate of the subquery is *this.department.employees*. If using a candidate expression we must provide an alias.

You can do the same with the JDOQL API. Like this

Query averageHoursQuery = pm.newQuery(Employee.class);

averageHoursQuery.setResult("avg(this.weeklyhours)");

Query q = pm.newQuery(Employee.class);

q.setFilter("this.weeklyhours > averageWeeklyhours");

q.addSubquery(averageHoursQuery, "double averageWeeklyhours", "this.department.employees", null);

so now our subquery has a candidate related to the outer query candidate.

|  |
| --- |
| **JDOQL : In-Memory queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/nucleus_extension.gif

The typical use of a JDOQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JDOQL syntax in the datastore *native query language*and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of *in-memory evaluation*in DataNucleus.

* Query methods **Collection.contains**, **Map.containsKey**, **Map.containsValue**and **Map.get**are not supported currently
* Only simple subqueries are supported currently

To enable evaluation in memory you specify the query extension **datanucleus.query.evaluateInMemory**to *true*as follows

query.addExtension("datanucleus.query.evaluateInMemory","true");

This is also useful where you have a Collection of (persisted) objects and want to run a query over the Collection. Simply turn on in-memory evaluation, and supply the candidate collection to the query, and no communication with the datastore will be needed.

|  |
| --- |
| **SQL Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jdo2.gif

The ability to query the datastore is an essential part of any system that persists data. Sometimes an object-based query language (such as JDOQL) is considered not suitable, maybe due to the lack of familiarity of the application developer with such a query language. In this case it is desirable to query using **SQL**. JDO 2 standardises this as a valid query mechanism, and DataNucleus supports this. **Please be aware that the SQL query that you invoke has to be valid for your RDBMS, and that the SQL syntax differs across almost all RDBMS.**

To utilise **SQL**syntax in queries, you create a Query as follows

Query q = pm.newQuery("javax.jdo.query.SQL",the\_query);

You have several forms of SQL queries, depending on what form of output you require.

* **No candidate class and no result class**- the result will be a List of Objects (when there is a single column in the query), or a List of Object[]s (when there are multiple columns in the query)
* **Candidate class specified, no result class**- the result will be a List of candidate class objects, or will be a single candidate class object (when you have specified "unique"). The columns of the querys result set are matched up to the fields of the candidate class by name. You need to select a minimum of the PK columns in the SQL statement.
* **No candidate class, result class specified**- the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). Your result class has to abide by the rules of JDO2 result classes (see [Result Class specification](http://www.datanucleus.org/products/accessplatform_1_1/jdo/query.html#resultclass)) - this typically means either providing public fields matching the columns of the result, or providing setters/getters for the columns of the result.
* **Candidate class and result class specified**- the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). The result class has to abide by the rules of JDO2 result classes (see [Result Class specification](http://www.datanucleus.org/products/accessplatform_1_1/jdo/query.html#resultclass)).

|  |
| --- |
| **Stored Procedures** |

|  |
| --- |
|  |

In JDO2 all SQL queries must begin "SELECT ...", and consequently it is not possible to execute stored procedures. In DataNucleus we have an extension that allows this to be overridden. To enable this you should pass the property **datanucleus.rdbms.sql.allowAllSQLStaments**as true when creating the PersistenceManagerFactory. Thereafter you just invoke your stored procedures like this

Query q = pm.newQuery("javax.jdo.query.SQL", "EXECUTE sp\_who");

Where "sp\_who" is the stored procedure being invoked. Clearly the same rules will apply regarding the results of the stored procedure and mapping them to any result class. The syntax of calling a stored procedure differs across RDBMS. Some require "CALL ..." and some "EXECUTE ...". Go consult your manual.

|  |
| --- |
| **Inserting/Updating/Deleting** |

|  |
| --- |
|  |

In JDO2 all SQL queries must begin "SELECT ...", and consequently it is not possible to execute queries that change data. In DataNucleus we have an extension that allows this to be overridden. To enable this you should pass the property **datanucleus.rdbms.sql.allowAllSQLStaments**as true when creating the PersistenceManagerFactory. Thereafter you just invoke your statements like this

Query q = pm.newQuery("javax.jdo.query.SQL", "UPDATE MY\_TABLE SET MY\_COLUMN = ? WHERE MY\_ID = ?");

you then pass any parameters in as normal for an SQL query.

|  |
| --- |
| **Parameters** |

|  |
| --- |
|  |

In JDO2 SQL queries can have parameters but must be *positional*. This means that you do as follows

Query q = pm.newQuery("javax.jdo.query.SQL",

"SELECT col1, col2 FROM MYTABLE WHERE col3 = ? AND col4 = ? and col5 = ?");

List results = (List) q.execute(val1, val2, val3);

So we used traditional JDBC form of parametrisation, using "?".

http://www.datanucleus.org/products/accessplatform_1_1/images/nucleus_extension.gif

DataNucleus also supports two further variations. The first is called *numbered*parameters where we assign numbers to them, so the previous example could have been written like this

Query q = pm.newQuery("javax.jdo.query.SQL",

"SELECT col1, col2 FROM MYTABLE WHERE col3 = ?1 AND col4 = ?2 and col5 = ?1");

List results = (List) q.execute(val1, val2);

so we can reuse parameters in this variation. The second variation is called *named*parameters where we assign names to them, and so the example can be furtehr rewritten like this

Query q = pm.newQuery("javax.jdo.query.SQL",

"SELECT col1, col2 FROM MYTABLE WHERE col3 = :firstVal AND col4 = :secondVal and col5 = :firstVal");

Map params = new HashMap();

params.put("firstVal", val1);

params.put("secondVal", val1);

List results = (List) q.executeWithMap(params);

|  |
| --- |
| **Example 1 - Using SQL aggregate functions, without candidate class** |

|  |
| --- |
|  |

Here's an example for getting the size of a table without a candidate class.

Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(\*) FROM MYTABLE");

List results = (List) query.execute();

Integer tableSize = (Integer) result.iterator().next();

Here's an example for getting the maximum and miminum of a parameter without a candidate class.

Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT max(PARAM1), min(PARAM1) FROM MYTABLE");

List results = (List) query.execute();

Object[] measures = (Object[])result.iterator().next();

Double maximum = (Double)measures[0];

Double minimum = (Double)measures[1];

|  |
| --- |
| **Example 2 - Using SQL aggregate functions, with result class** |

|  |
| --- |
|  |

Here's an example for getting the size of a table with a result class. So we have a result class of

public class TableStatistics

{

private int total;

public setTotal(int total);

}

So we define our query to populate this class

Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(\*) AS total FROM MYTABLE");

query.setResultClass(TableStatistics.class);

List results = (List) query.execute();

TableStatistics tableStats = (TableStatistics) result.iterator().next();

Each row of the results is of the type of our result class. Since our query is for an aggregate, there is actually only 1 row.

|  |
| --- |
| **Example 3 - Retrieval using candidate class** |

|  |
| --- |
|  |

When we want to retrieve objects of a particular PersistenceCapable class we specify the candidate class. Here we need to select, as a minimum, the identity columns for the class.

Query query = pm.newQuery("javax.jdo.query.SQL",

"SELECT MY\_ID, MY\_NAME FROM MYTABLE");

query.setClass(MyClass.class);

List results = (List) query.execute();

Iterator resultsIter = results.iterator();

while (resultsIter.hasNext())

{

MyClass obj = (MyClass)resultsIter.next();

}

class MyClass

{

String name;

...

}

<jdo>

<package name="org.datanucleus.samples.sql">

<class name="MyClass" identity-type="datastore" table="MYTABLE">

<datastore-identity strategy="identity">

<column name="MY\_ID"/>

</datastore-identity>

<field name="name" persistence-modifier="persistent">

<column name="MY\_NAME"/>

</field>

</class>

</package>

</jdo>

|  |
| --- |
| **Example 4 - Using parameters, without candidate class** |

|  |
| --- |
|  |

Here's an example for getting the number of people with a particular email address. You simply add a "?" for all parameters that are passed in, and these are subsitituted at execution time.

Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(\*) FROM PERSON WHERE EMAIL\_ADDRESS = ?");

List results = (List) query.execute("nobody@datanucleus.org");

Integer tableSize = (Integer) result.iterator().next();

|  |
| --- |
| **Example 5 - Named Query** |

|  |
| --- |
|  |

While "named" queries were introduced primarily for JDOQL queries, we can define "named" queries for SQL also. So let's take a *Product*class, and we want to define a query for all products that are "sold out". We firstly add this to our MetaData

<jdo>

<package name="org.datanucleus.samples.store">

<class name="Product" identity-type="datastore" table="PRODUCT">

<datastore-identity strategy="identity">

<column name="PRODUCT\_ID"/>

</datastore-identity>

<field name="name" persistence-modifier="persistent">

<column name="NAME"/>

</field>

<field name="status" persistence-modifier="persistent">

<column name="STATUS"/>

</field>

<query name="SoldOut" language="javax.jdo.query.SQL"><![CDATA[

SELECT PRODUCT\_ID FROM PRODUCT WHERE STATUS == "Sold Out"

]]></query>

</class>

</package>

</jdo>

And then in our application code we utilise the query

Query q = pm.newNamedQuery(Product.class, "SoldOut");

List results = (List)q.execute();

JDO provides a flexible API for use of query languages. DataNucleus makes use of this to allow use of the query language defined in the JPA1 specification (JPQL) with JDO persistence. To provide a simple example, this is what you would do

Query q = pm.newQuery("javax.jdo.query.JPQL", "SELECT p FROM Person p WHERE p.lastName = 'Jones'");

List results = (List)q.execute();

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

|  |
| --- |
| **JPQL Syntax** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

In traditional (declarative) JDOQL (JDO 1.0) it was necessary to specify the component parts (filter, candidate class, ordering, etc) of the query using the mutator methods on the Query. In JDO 2 you can now specify it all in a single string. This string has to follow a particular pattern, but provides the convenience that many people have been asking for. The pattern to use is as follows

SELECT [<result>]

[FROM <candidate-class(es)>]

[WHERE <filter>]

[GROUP BY <grouping>]

[HAVING <having>]

[ORDER BY <ordering>]

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

|  |
| --- |
| **Entity Name** |

|  |
| --- |
|  |

In the example shown you note that we did not specify the full class name. We used *Person p*and thereafter could refer to *p*as the alias. The *Person*is called the **entity name**and in JPA MetaData this can be defined against each class in its definition. With JDO we dont have this MetaData attribute so we simply define the **entity name**as *the name of the class omitting the package name*. So *org.datanucleus.test.samples.Person*will have an entity name of *Person*.

|  |
| --- |
| **Input Parameters** |

|  |
| --- |
|  |

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Let's take two examples

Named Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :surname AND o.firstName = :forename");

q.setParameter("surname", theSurname);

q.setParameter("forename", theForename");

Numbered Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName = ?2");

q.setParameter(1, theSurname);

q.setParameter(2, theForename);

So in the first case we have parameters that are prefixed by **:**(colon) to identify them as a parameter and we use that name when calling *Query.setParameter()*. In the second case we have parameters that are prefixed by **?**(question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*.

|  |
| --- |
| **Unique Results** |

|  |
| --- |
|  |

Sometimes you know that the query can only every return 0 or 1 objects. In this case you can simplify your job by adding

query.setUnique(true);

In this case the return from the execution of the Query will be a single Object rather than a List, so you've no need to use iterators, just cast it to your candidate class type.