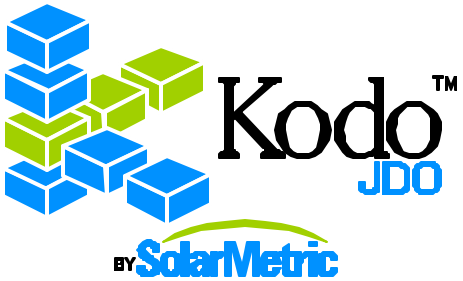
JDOQL 2.0 Quick Reference Guide



Query Structure

Items in square brackets are optional.
Query clauses must be in the specified order.

select [**unique**] [<result>] [**into** <result-class-name>]
[**from** <candidate-class-name> [**exclude subclasses**]]
[**where** <filter>]
[**variables** <variable-list>]
[**parameters** <parameter-list>]
[**imports** <import-list>]
[**group by** <grouping-clause>]
[**order by** <ordering-clause>]
[**range** <from-range> **,**<to-range>]

Query Examples

The following examples use this sample class.

```
public class Person
{
   private int age;
   private String firstName;
   private String lastName;
   private Address address;
   private Set<Person> children;
}
```

Basic Example

Print last name of all with the first name "John".

```
Query q = pm.newQuery ("select from "
  + "com.example.Person "
  + "where firstName == 'John'");
List<Person> people = (List<Person>)
    q.execute();
for (Person person : people)
    print(person.getLastName());
q.closeAll(); // Close resources
```

Parameters

Parameters can be specified in query strings by placing a colon in front of the identifier (i.e. :param). Parameters can help memory utilization and performance.

Simple parameters:

Find all people named "John".

```
Query q = pm.newQuery("select from "
  + "com.example.Person where "
  + "firstName == :param");
List<Person> people = (List<Person>)
  q.execute ("John");
```

Using persistent instances as parameters:

Find all people over 21 with a given address.

```
Address a = (Address) pm.getObjectById
  (Address.class, id);
Query q = pm.newQuery("select from "
  + "com.example.Person "
  + "where address == :param1 "
  + "&& age > :param2");
List<Person> people = (List<Person>)
  q.execute (a, new Integer (21));
```

Variables

Variables allow queries on multi-value relationships or on unrelated classes.

Querying on a related instance:

Find all parents of a Person named "John".

```
Query q = pm.newQuery("select from "
  + "com.example.Person "
  + "where children.contains(p)"
  + "&& p.firstName == :name");
List<Person> parents = (List<Person>)
  q.execute("John");
```

Ordering Results

Ordering on a single field:

Order query results by age, oldest first.

```
select from com.example.Person
  order by age desc
```

Ordering on multiple fields:

Order by name, A first, then by age, oldest first.

```
select from com.example.Person
  order by firstName asc, age desc
```

Keywords

Keywords must appear in either all upper-case or all lower-case characters.

```
as, asc, ascending, avg, by, count, desc,
descending, distinct, exclude, from, group,
having, imports, into, max, min, order,
parameters, range, select, subclasses, sum,
to, unique, variables, where
```

Optimizations

These represent a few of the available methods to speed up JDO queries.

Limiting and paging query results:

The query can be configured to only return a subset of the results so that unused elements will not be instantiated. The start point is included, while the element at the limit is not.

```
select from com.example.Person range 10, 20
```

Ignore PersistenceManager cache:

Setting this parameter to true can speed up queries because changes made during the transaction do not need to be included in the results.

```
query.setIgnoreCache(true);
```

Indicate unique result:

Specifies that only one result is expected and to return only the single instance instead of a List.

```
Query q = pm.newQuery ("select unique "
  + "from com.example.Person where "
  + "firstName == :name");
Person john = (Person) q.execute("John");
```

Aggregates, Projections, and Grouping

Grouping allows aggregates and projections to be grouped by a given field and optionally limited using"having". Available aggregates are min, max, sum, avg, and count.

Simple grouping:

```
Query q = pm.newQuery("select avg(age) "
  + "from com.example.Person "
  + "group by firstName");
```

Limiting grouping with "having" expression:

Group by firstName where the firstName starts with "J".

```
Query q = pm.newQuery("select count(this) "
  + "from com.example.Person "
  + "group by firstName "
  + "having firstName.startsWith(:string)");
q.execute("J");
```

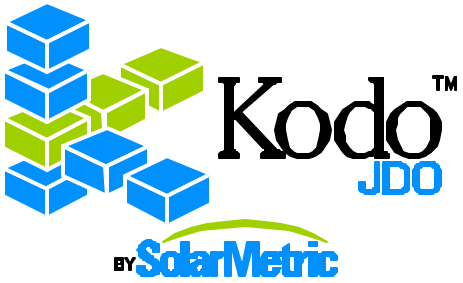
SQL Queries

Queries can use SQL when accessing a relational database. Find people whose first name is "John".

```
Query q = pm.newQuery(Query.SQL, "SELECT * "
    +"FROM PERSON WHERE FIRSTNAME = ?");
q.setClass (Person.class);
List<Person> people = (List<Person>)
    q.execute("John");
```


Content based upon JDO 2.0 public draft http://solarmetric.com/resources/jdoql-quickref.pdf

JDOQL 2.0 Quick Reference Guide



Kodo Extensions

Fully-qualified Column

Places the correct alias for the column into the query.

```
query.setFilter("address.ext:getColumn "
  + "('ZIP_PLUS_4') == '78751-1228'");
```

SQL Snippets

Embeds the result of an SQL query as an argument.

```
query.setFilter(
  "ext:sql('ROUND(SALARY,0)') > 100000");
```

You can also build your own extension by implementing kodo.jdbc.query.JDBCFilterListener.

JDOQL Subqueries

Find the oldest people in the company.

```
Query q = pm.newQuery ("select from "
  + "com.example.Person where age == "
  + "(select max(p.age) from Person p)");
```

Find those whose last name is part of a street address.

```
Query q = pm.newQuery("select from "
  + "com.example.Person where (select "
  + "a.streetAddress from Address a)."
  + "contains (lastName)");
```

Execute a JDOQL query via the Kodo EJB Query interface.

```
KodoEntityManager kem =
  KodoPersistence.cast(em);
Query q = kem.createQuery(
  kodo.query.QueryLanguages.LANG_JDOQL,
  "select from com.example.Person where "
    + "address.state == :stateCode");
q.setParameter("stateCode", "MA");
List<Person> people = (List<Person>)
  q.getResultList();
```

Where Clause Methods

The following methods may be used in a where clause. For example, "where Math.abs (balance) > 500".

Collection methods	`contains(Object), isEmpty(), size()`
Map methods	`containsKey(Object), size(Object), containsValue(Object), isEmpty(), get(Object)`
String methods	`startsWith(String), endsWith(String), matches(String), toLowerCase(), toUpperCase(), indexOf(String), indexOf(String, int), substring(int), substring(int, int)`
Math methods	`Math.abs(numeric), Math.sqrt(numeric)`
JDOHelper methods	`getObjectId(Object)`

Regular Expressions

The matches(String) method accepts limited regular expression syntax. The following wildcards can be used.

.	represents a single character
.*	represents multiple characters
(?i)	represents case-insensitive matching

All people with 'rick' in their name – Patrick, Rick, etc.

```
select from com.example.Person
  where firstName.matches('.*rick.*(?i)')
```

Result Classes and Aliases

You can have query results placed directly into a custom class. This example uses the custom class Name below.

```
public class Name {
  public String first;
  public String last;
}
Query q = pm.newQuery ("select "
  + "firstName as first, lastName as last "
  + "into com.example.Name "
  + "from com.example.Person "
  + "where age > :param");
List<Name> names = (List<Name>)
  q.execute(30);
for(Name name : names)
  printLabel(name);
query.close(names);
```

Where Clause Operators

==	equal (*note: can be used with Strings*)
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
&&	conditional AND
&	boolean logical AND
||	conditional OR
|	boolean logical OR
-	subtract or invert sign
+	add or concatenate strings
*	multiply
/	divide
%	modulo
!	logical complement
~	bitwise complement
instanceof	instance of a class

Named Queries

Named queries are defined in XML and consist of a name, query language, unmodifiable attribute, and the query itself.

```
<class name="Person">
  <query name="adultsByFirstName"
      unmodifiable="false">
    select where age > 18 group by firstName
  </query>
</class>
```

```
List<Person> adults = (List<Person>)
  pm.newNamedQuery (Person.class,
  "adultsByFirstName").execute ();
```

In-Memory Queries

JDOQL queries can be evaluated against an in-memory collection of persistent or transactional types.

```
Query q = pm.newQuery ("select from "
  + "com.example.Person "
  + "where firstName == 'John'");
q.setCandidates (allPeople)
filteredPeople = (List<Person>) q.execute();
```

For additional detail about JDOQL see the full Kodo documentation at http://docs.solarmetric.com.

Content based upon JDO 2.0 public draft http://solarmetric.com/resources/jdoql-quickref.pdf