JDOQL: The JDO Query Language

The Java Data Objects API and its query language overcome the deficiencies of other commonly used persistence mechanisms

by David Jordan

Go Online

Visit www.javapro.com for related resources. Simply type the Locator+ code into the field in the upper-right corner of the page.

Download

JP0207 Download all the code for this issue.

JP0207DJ Download the code for this article separately.

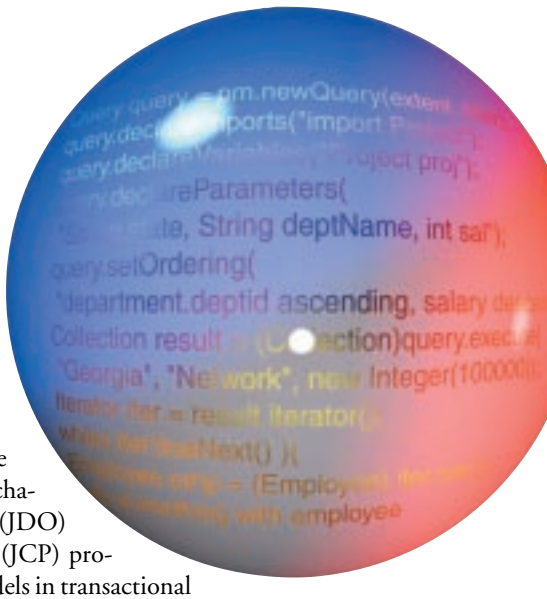
JP0207DJ_S Download a sidebar entitled "The JDO Story" that gives a few facts about JDO as well as provides additional links for more info.

Discuss

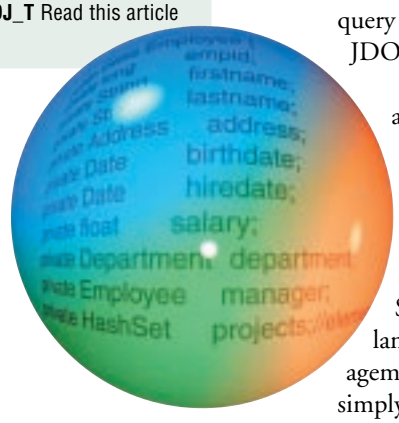
JPTalk Discuss this article in the "Talk to the Editors of Java Pro Magazine" discussion forum.

Read More

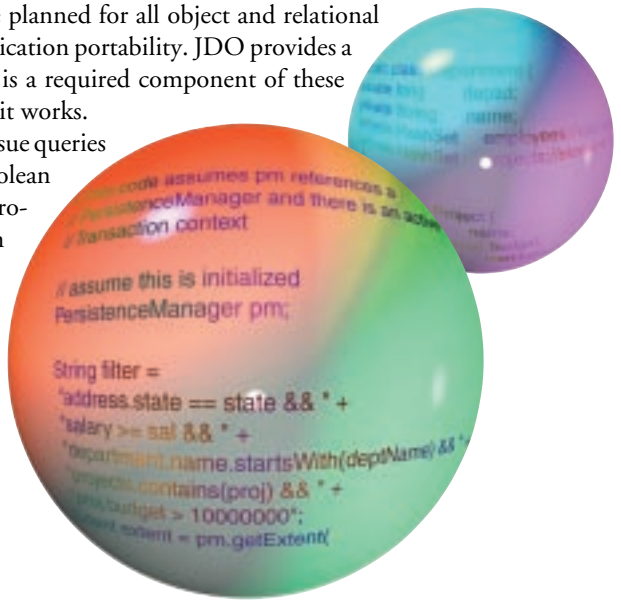
JP0207DJ_T Read this article online.



Java developers use serialization, JDBC, or EJB Container Managed Persistence (CMP) for the persistence of the data in their programs, but each of these commonly used persistence mechanisms has some drawbacks (see sidebar "Limitations of Current Persistence Mechanisms"). Now, however, the Java Data Objects (JDO) API defined in the Java Community Process (JCP) provides transparent persistence of Java object models in transactional datastores (see sidebar "The JDO Story" online at www.javapro.com). JDO has been adopted by the JCP and became a Java standard on March 26, 2002. Among other benefits JDO provides, instances of Java classes are directly persisted and the application does not have to handle any other data model, and JDO's architecture integrates well into EJB application servers and provides needed portability in such environments. JDO also provides transaction and query support, and binary compatibility is provided across all implementations. Implementations are planned for all object and relational databases, providing a high level of application portability. JDO provides a query language called JDOQL, which is a required component of these JDO implementations. Let's see how it works.



JDOQL consists of an API to issue queries and a query language in which Boolean filters are expressed. JDOQL provides a high degree of insolation from the underlying database architecture, which may support a relational query language like SQL, an object database query language such as Object Data Management Group's (ODMG) OQL, or simply be a set of library calls on which JDOQL is implemented.



JDO defines an interface called Query in package javax.jdo. In JDO, the primary interface an application uses to operate on instances in a transaction context is defined by the interface PersistenceManager, which includes methods to construct instances of Query. An application may construct and use multiple Query instances within the context of a particular PersistenceManager.

Components of a Query

Evaluating a query in JDOQL involves applying a Boolean filter to a collection of candidate instances and returning all candidate instances for which the filter evaluates to true. Scoping the names used in the query filter is necessary, so applications must specify the class of the candidate instances. All candidate instances must be of this class or a subclass. The collection of candidate instances passed to the query must either be a java.util.Collection or an Extent (defined in package javax.jdo). JDO defines the Extent interface; it represents all the instances of a particular class (and optionally its subclasses) in the database. The JDO interface PersistenceManager has the method getExtent(), which is used to get an Extent for a class. The Boolean filter is specified with a String that contains a JDOQL boolean query expression. If the filter is not specified, then the filter defaults to true, causing all the candidate instances to be in the query result.

The PersistenceManager interface has a set of methods called newQuery(), which allows you to create a Query instance from one or more of these query components:

```
Query newQuery();
Query newQuery(Class cls);
Query newQuery(Class cls,
               Collection c);
Query newQuery(Class cls,
               String filter);
Query newQuery(Class cls,
               Collection c,
               String filter);
Query newQuery(Extent e);
Query newQuery(Extent e,
               String filter);
```

In addition to specifying query components in the construction of a Query instance, the Query interface also provides these methods:

```
void setClass(
          Class candidateClass);
void setCandidates(
          Collection candidates);
void setCandidates(
          Extent candidates);
void setFilter(String filter);
```

A query can also have parameters. You would pass a parameter into a query to provide a value at run time used in a query constraint. The Query interface has the method declareParameters(), which is passed a String containing one or more parameter declarations separated with commas. A parameter declaration in JDOQL uses the same syntax as formal method parameters in Java, which consists of a type name and parameter name. For example, using the application schema provided in Listing 1, you may want to declare param-

Limitations of Current Persistence Mechanisms

The commonly used persistence mechanisms—serialization, JDBC, or EJB Container Managed Persistence (CMP)—all have disadvantages. These include:

- Serialization provides a tight integration with the Java language (allowing object models to be easily persisted), but it lacks support for robust database capabilities like transactions and queries.
- JDBC provides an interface between Java programs and SQL. It uses the SQL data model, which consists of rows and columns, but it does not provide support of storing Java object models. Although JDBC does provide database capabilities like transactions and queries, applications really communicate with a specific SQL implementation, and incompatibilities among SQL implementations can result in a loss of application portability.
- Developers use EJB so that they can leverage CMP's capability of allowing a more object-oriented model of their data than can be attained using JDBC. But EJB assumes a distributed model of computation, imposing performance degradations when one has objects of fine granularity.

Listing 1 Company Data

```
// Assume methods defined for these
// classes as well.
public class Address {
  private String      street;
  private String      city;
  private String      state;
  private String      zipcode;
}

public class Department {
  private long        deptid;
  private String      name;
  private HashSet     employees;//element:Employee
  private HashSet     projects;//element:Project
}
```

```
public class Employee {
  private long        empid;
  private String      firstname;
  private String      lastname;
  private Address     address;
  private Date        birthdate;
  private Date        hiredate;
  private float       salary;
  private Department  department;
  private Employee    manager;
  private HashSet     projects;//element:Project
}

public class Project {
  private String      name;
  private BigDecimal  budget;
  private HashSet     members;//element:Employee
}
```

Here is the data model of classes shown in the example.

eters to be used to constrain a person's name and birthdate:

```
query.declareParameters(
  "String lname, Date bdate");
```

Each parameter must be bound to a value when the query is executed. A value for a parameter is specified by a Java Object, which may be a simple wrapper type (for example, Float) or a more complex object type. Because a parameter is likely to have a different type than the class of the candidate instances, the types of the parameters need to be declared to avoid ambiguity.

It is necessary to import classes other than the candidate class. Import statements can be specified by calling the Query method declareImports(), which is passed a String containing semicolon-separated import statements (using the same syntax as the Java language import statement). For example:

```
query.declareImports(
  "import Project;" +"import
  Employee" );
```

A query has two namespaces. Type names live in one namespace; fields, variables, and parameters share another namespace. When a type is used, it must either be the name of the candidate class, the name of a class or interface imported by declareImports(), or a class or interface from the same package as the candidate class.

A JDOQL query can also have variables that are bound to values during the execution of the query. In particular, when a query filter involves iteration through a collection, a variable is used in conjunction with the contains() method to reference a collection element (more on this later). The name and type of each variable must be declared. The Query method declareVariables() can be called, passing a String that contains one or more variable declarations. They are separated by semicolons, using the same syntax as Java local variables. You may execute a query that uses the Department class for the candidate collection, but you can also navigate to Employee and Project instances to specify some query constraints (see Figure 1). You would need to declare variables like this:

```
query.declareVariables(
  "Employee emp; Project proj");
```

As you can see, JDOQL uses Java syntax conventions wherever possible.

Lastly, the order of the result set of a query can be specified. An ordering specification is a String that contains a list of comma-separated expressions, each with an ascending/descending indicator. This list of expressions is evaluated in sequence: Result instances are ordered according to the first expression, followed by the second expression (if it exists), followed by the third, and so on. This is the same behavior as the order by clause in SQL. The Query method setOrdering() is used to specify the ordering specification.

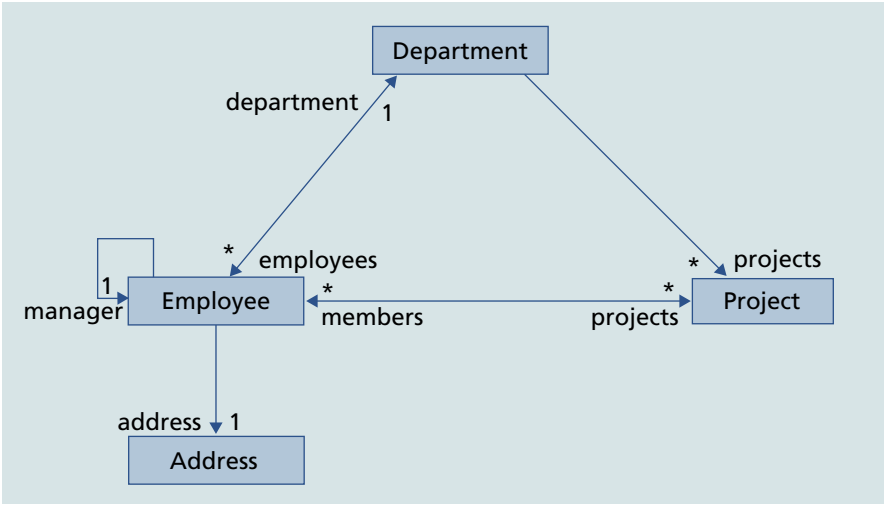


Figure 1 | Viewing Relationships Here is a visual representation of the example schema.

Description	JDOQL operator	ANSI SQL operator
Equals	==	==
Not equal	!=	<>
Greater than	>	>
Greater than or equal	>=	>=
Less than	<	<
Less than or equal	<=	<=
Boolean logical AND	&	AND
Conditional AND	&&	AND
Boolean logical OR	|	OR
Conditional OR	||	OR
Logical complement	!	NOT
Integral unary bitwise complement	~	(-expr) - 1
String concatenation	+	||
Binary or unary addition	+	+
Binary subtraction or numeric sign inversion	-	-
Multiplication	*	*
Division	/	/

Table 1 | Equivalent Operators Here are the JDOQL operators and ANSI SQL equivalents.

For example, if you want to have a query result of employees that are ordered based on their city, followed by their salary from highest to lowest, this ordering specification could be used:

```
query.setOrdering(
  "address.city ascending," +
  "salary descending");
```

Filter Specification

The query filter is a Boolean expression used to determine whether an instance in the candidate collection should be a member of the result. A candidate instance is returned in the result if it is assignment compatible with the candidate class of the Query, and if for all variables there exists a value for which the filter expression evaluates to true.

Table 1 lists the JDOQL query operators and their ANSI SQL equivalents. The mapping analysis work comparing JDOQL and SQL is provided courtesy of Michael Bouschen of Tech@Spree, a member of the JDO expert group. As you can see, the Boolean operators AND, OR, and NOT can be used to build up logical expressions of arbitrary complexity. Equality, comparison, and arithmetic operators are also supported. JDOQL is a left-associative language—operands are bound to operators in a left-to-right fashion. Parentheses can be used to explicitly mark operator precedence. White space is a separator and is ignored. One can build up expressions nested to arbitrary depths by using operator composition.

JDOQL operator	Supported types
==	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger Boolean, boolean, Date, any class instance or array
!=	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger Boolean, boolean, Date, any class instance or array
>	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String
>=	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String
<	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String
<=	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String
&	Boolean, boolean
&&	Boolean, boolean
|	Boolean, boolean
||	Boolean, boolean
!	Boolean, boolean
~	byte, short, int, long, char, Byte, Short, Integer, Long, Character
+	String
+	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger
-	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger
*	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger
/	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger

Table 2 | Good Support Here are the supported types for JDOQL operators.

Table 2 indicates which data types are supported for each operator. The operators apply to all types as they are defined in Java, except that String concatenation is supported only with String operands. The use of these operators is similar to their use in Java, but there are a few exceptions. One difference between Java and JDOQL is that in JDOQL it is valid to have equality and comparison operations between primitives (such as int, float) and instances of wrapper classes (such as Integer, Float). Equality, comparison, and arithmetic operations on object-valued fields of wrapper types (such as Integer and Long) and the numeric types (BigDecimal and BigInteger) use the wrapped values for comparisons. Numeric operands are promoted for comparisons. This allows a byte to be compared with a BigDecimal, to provide an extreme example. Use of the comparison operators on Strings is also supported. In JDOQL using the equality and comparison operators on two Date operands is possible—a capability not available in Java.

Equality of object-valued types depends on whether the type has been declared as a PersistenceCapable type in the JDO environment. Classes that will have instances stored in the database are enhanced to implement interface PersistenceCapable. Two instances of a PersistenceCapable type will be equal if they have the same JDO identity; in other words, if they refer to the same instance in the database. If equality is evaluated on instances of a non-PersistenceCapable type, the equals() method defined for the type is used.

Two String methods, startsWith() and endsWith(), are supported to perform wild card queries. The expression:

```
lastname.startsWith("Jo")
```

finds all employees whose name starts with "Jo", as in Jones, Johnson, and Joyner. JDO does not define any semantics associated with the String argument to these methods; special wild card characters are not specified.

Even though a JDOQL query evaluates a collection of instances of a single class, it is possible to navigate to related objects and apply additional query constraints. You can

Listing 2 Find What You Need

```
// This code assumes pm references a
// PersistenceManager and there is an active
// Transaction context

// assume this is initialized
PersistenceManager pm;

String filter =
  "address.state == state && " +
  "salary >= sal && " +
  "department.name.startsWith(deptName) && " +
  "projects.contains(proj) && " +
  "proj.budget > 10000000";
Extent extent = pm.getExtent(
  Employee.class, true);
Query query = pm.newQuery(extent, filter);
query.declareImports("import Project");
query.declareVariables("Project proj");
query.declareParameters(
  "String state, String deptName, int sal");
query.setOrdering(
  "department.deptid ascending, salary descending");
Collection result = (Collection)query.execute(
  "Georgia", "Network", new Integer(100000));
Iterator iter = result.iterator();
while( iter.hasNext() ){
  Employee emp = (Employee) iter.next();
  // do something with employee

}
query.close(result);
```

This is a rather complex query that finds all employees who live in the state of Georgia, earn a six-figure salary, work in a department dealing with networks, and work on a project with a budget that exceeds ten million dollars.

navigate through a reference by using the "." operator, using standard Java syntax. The "." operator can also be applied repeatedly in one expression (manager.manager.address.street) to navigate through multiple references in the model. Navigation through a null-valued field would throw a NullPointerException in Java. In JDOQL it is treated as if the filter expression evaluated to false for the current set of variable values.

If you have a collection in the model, the Boolean method isEmpty() can be called to determine if it contains any elements. The expression:

```
members.isEmpty()
```

on Project instances determines whether a project has any members assigned to it.

Navigating through a collection in the model is also possible. A variable is used to reference the current element of iteration in the collection, so declaring a variable for each collection that will be used in the query is necessary. The method contains() should be used in the query, with the variable passed to the method. This contains() method is really simply syntax that is used to associate the variable with each element in the collection. The contains() method returns true if the collection is non-null and contains at least one element. The contains() method must be used as the left operand of an AND expression, with the variable used in an expression in the right operand of the AND expression. In this example filter, the candidate class is Department and declareVariables() needs to have declared emp and proj:

```
employees.contains(emp) &
  emp.projects.contains(proj) &
  (proj.budget > 1000000.00 &
  proj.name.startsWith(
  "Software"))
```

Here, placing parentheses around the constraints on both the project budget and name is necessary, because the variable proj is valid only for the right operand of the AND expression in which proj is associated (in the left operand of the AND) with the elements of the collection emp.projects.

Compilation/Execution

The application can optionally call the Query method compile() to compile a query for subsequent execution. When this method is called, any elements bound to the Query instance are validated. Any inconsistencies are reported by the throwing of a JDOUserException. This method is a hint to the Query instance to prepare and optimize an execution plan for the query. This method may yield better performance if the query is used repeatedly.

The Query interface provides several methods to execute a query; the methods vary in their style of passing query parameters. Up to three query parameters can be passed directly to an execute() method:

```
Object execute(Object p1);
Object execute(Object p1,
               Object p2);
Object execute(Object p1,
               Object p2,
               Object p3);
```

If you need more than three parameters, you would use executeWithMap() or executeWithArray(), described in a moment.

The parameters passed to execute are associated according to the order they were declared in the parameter declaration. Each parameter is an Object that is either the necessary object value or an Object-wrapped value of a primitive. The parameters passed to an execute() method are used only for a single execution of a query, they are not remembered for future executions of the same query.

The execute() methods return an unmodifiable Collection. They are de-

> Developers use EJB so that they can leverage CMP's capability of allowing a more object-oriented model of their data than can be attained using JDBC. But EJB assumes a distributed model of computation, imposing performance degradations when one has objects of fine granularity

clared to return an Object instead of a Collection to allow for future extensions where a single instance could be returned. So an application must explicitly cast the result to a Collection. The application can then iterate the collection to retrieve the result. Attempts to change the collection will cause an UnsupportedOperationException to be thrown.

You can use two other methods to execute a query. Use these when the number of parameters is greater than three or if you prefer this alternate style of passing parameters:

```
Object executeWithMap(Map m);
Object executeWithArray(
                Object[] a);
```

The executeWithMap() method is passed a Map of key/value pairs where the key is the name of the declared parameter and the value is the value to use for that parameter in the query. The executeWithArray() method is similar to the execute() methods in that the position of a value in the array corresponds with the position of a parameter in the parameter declaration.

Putting It All Together

Listing 2 provides a complete example of the calls necessary to perform a rather complex query. It assumes that the application has acquired a PersistenceManager and begun a transaction. This query finds all employees who live in the state of Georgia, earn a six-figure salary, work in a department dealing with networks, and work on a project with a budget that exceeds ten million dollars. The result collection is returned with the employees grouped together according to their department's id, and within their department they are returned in order from highest to lowest salary.

JDOQL is a fairly simple language that should gain significant industry support over the next year. Its design, interface, and set of expressions largely emulate the Java language, which should make it fairly easy for Java developers to learn. In some cases, as with the promotion and conversion of operands, it provides a simpler syntax than Java when working with semantically similar yet distinct data types. JDO implementations are planned for a wide variety of datastores, including hierarchical, relational, and object datastores. JDOQL is a required component of these JDO implementations. JP

> JDOQL should gain significant industry support over the next year

David Jordan has been developing object-oriented software since 1981 and has been involved with object persistence and databases since 1985. He is an active member of the JDO expert group and founded Object Identity, Inc. to provide consulting services in support of JDO development. Reach him at **david.jordan@objectidentity.com**.