David Jordan / djordan@buildnet.com

An overview of Sun's Java Data Objects specification

THE JAVA DATA Objects (JDO) specification is under development within the Sun Community Process under JSR-000012. The main objective of JDO is to provide support for transparent object-level persistence of Java objects, so that Java class developers need not provide their own persistence support.

Prior to JDO, there was not a Java platform specification that provided a standard architecture for storing Java objects in a transactional data store.

The JDO API is defined such that applications are independent of the particular data store being used by a JDO implementation. Implementations are planned for file systems, hierarchical, relational, and object databases. These will be available in the following Java environments.

- Java 2 Micro Edition (J2ME) Connected Device Configuration
- Java 2 Standard Edition (J2SE)
- Java 2 Enterprise Edition (J2EE) and EJB environments

Companies with representatives on the expert group include those listed in Table 1. You can expect to see these vendors offer JDO technology over the next year. This list includes implementations for both object and relational databases. I encourage you to contact these vendors to get more specific information about their JDO product offerings.

The JDO JSR was approved in July 1999 and the expert group was formed and met in August 1999. A public draft specification is scheduled for release by June 2000. I am constrained in what I can publish now about the specification; once it has been released I will cover it in more detail.

David Jordan is a Systems Architect with BuildNet Inc. based in Research Triangle Park,NC, which provides builder and supplier management software to the residential construction industry.

Persistent Classes

Java applications are usually represented by a set of classes with interrelationships among instances of the classes. Relationships are represented by either a single reference or a collection of references to related objects. Almost any user-defined class can be made persistent. Excluded are those classes that use native methods (Java Native Interface; JNI) or that are subclasses of Java system classes. The types of the data members of a persistent class can be any of the primitive types, interface types, and references. The `java.lang.String` class is supported. The JDO specification supports the `java.util.Date` class and `java.util.Collection` interfaces and classes. Java arrays are also supported with primitive types, interface types, or persistence-capable classes as the element types.

Each persistent instance of a class has a unique JDO identifier. The Java language defines *identity* in terms of two references being equal, i.e., that they refer to the same object in memory. This concept isn't adequate for JDO because the same database object may have multiple JVM instances in different transactions. Java also defines object *equality* via the `equals` method defined in `Object`, which can be overridden by a class. The JDO identifier handles the *persistent identification* of an object. There are several forms of identification, depending on the application. The identification could be based on a *primary key*, which is defined by the application and enforced by the database. This is the form most often used with a relational database. Another form is *database identification*, where the database itself manages the value of the identifier. This form may be used with object databases. Finally, a *nonmanaged identifier* occurs with a relational table that does not have a primary key.

The JDO Object Model distinguishes between First Class and Second Class objects. A First Class object is a persistence-capable class that has JDO identity. A Second Class object does not have its own JDO identifier and therefore cannot be referenced by multiple objects in the data store (more on identity later). It is always associated with one containing a First Class object. Sharing of a Second Class object by more than one First Class object is not supported. A First Class

Table 1. **Member companies of the JDO Expert Group.**

Advanced Language Technologies	www.alt1.com
Computer Associates Inc.	www.cai.com
GemStone Systems Inc.	www.gemstone.com
Informix Software	www.informix.com
POET Software	www.poet.com
Secant Technologies Inc.	www.secant.com
Sun Microsystems Inc.	www.sun.com
Tech@Spree Software Technology GmbH	www.tech.spree.de
Versant Corp.	www.versant.com
eXcelon Corp.	www.exceloncorp.com

object would be stored in a data store together with its primitive fields and associated Second Class objects. Second Class objects are stored as values along with the First Class object that refers to them. Second Class objects must track change to themselves and notify their containing First Class object that they have been changed so that their new state gets propagated back to the database. This is done by calling the method jdoMakeDirty on the First Class object.

Whenever a reference is followed from one persistent object to another, the JDO implementation transparently instantiates the instance in memory, unless it has already done so. When an object is first brought into memory from the database, the JDO implementation takes care of mapping between the database and in-memory representation for the object. JDO provides the illusion that the network of objects traversed by the application all reside in memory, when, in reality, they are only activated as needed by the application. This capability provided by JDO is known as transparent data access, transparent persistence, or database transparency.

Class Enhancer

To allow classes to be persistent in a transparent manner, they need to be enhanced. JDO introduces an Enhancer that will process the .class file of a Java class and create a new .class file with the necessary enhancements. In addition to the Java class definitions, a property file in XML format will define which classes will be persistent and various persistence properties of the classes.

The Enhancer needs to be run before the class can be used in a JVM. Some implementations will support enhancement at development time, others may support the dynamic enhancement of a class when it is loaded into the JVM. Each persistent class is changed to implement the interface PersistenceCapable. The Enhancer also adds method implementations for the methods defined by PersistenceCapable, which are the following:

```
public boolean jdoIsPersistent();
public boolean jdoIsNew();
public boolean jdoIsDeleted();
public boolean jdoIsTransactional();
public boolean jdoIsDirty();
public void    jdoMakeDirty();
public PersistenceManager jdoGetPersistenceManager();
public Object getObjectId();
```

Notice that each method has a prefix of jdo so that it won't conflict with method names defined by the application.

JDO defines an interface called PersistenceManager that serves as the application's primary interface to the persistence services provided by the JDO implementation. The goal is to provide application portability across different JDO vendor implementations. A PersistenceManager is used for managing the identity and lifecycle of instances. A PersistenceManager maintains a transactional cache of objects for a particular data store. The PersistenceManager only needs to be visible to those application components that perform queries or manage the life cycle of JDO instances. The objects persisted in a JDO implementation do not need to directly use (and depend) on the PersistenceManager.

JDO allows multiple PersistenceManager instances to be active in a JVM, and they can be from the same or a different vendor. Thus, an application running in a single JVM can access both a relational and object database, using the same API to manage objects in the two databases. A PersistenceManager supports one transaction at a time, using one connection to a data source. To support multiple concurrent connection-oriented data sources in an application, multiple PersistenceManager instances are required.

A PersistenceManagerFactory is used as a standard mechanism for creating PersistenceManager instances. It uses JavaBeans conventions for getting and setting properties, which include database user name, password, and connection URL. The factory object may implement pooling of PersistenceManager instances and also pooling of database connections among multiple PersistenceManager instances. The PersistenceManagerFactory is serializable and also supports the Java Naming and Directory Interface (JNDI).

An instance of a class can be either transient or persistent; the method jdoIsPersistent is used to determine this. The PersistenceManager interface has a method public void makePersistent(Object pc);,which is used to make persistent a transient instance of an enhanced class. To remove an instance from the database, a call is made to the PersistenceManager method public void deletePersistent(Object pc);.

The PersistenceManager has two methods that deal with the mapping between an instance and its JDO identifier: public Object getObjectId(Object pc); and public Object getObjectById(Object oid);.

A JDO instance, representing a specific object in the data store, will only exist once within a particular PersistenceManager cache. An application may query or navigate to the object through different references, but the cache management facilities ensure that just one copy is in the cache. As previously noted, multiple PersistenceManager instances can be active in a JVM. Each PersistenceManager instance may have its own copy of an object with the same ObjectId. The PersistenceManager method public Object getTransactionalInstance(Object pc); allows the application to obtain a copy of

the object referenced by pc from another PersistenceManager context.

Transactions

JDO interfaces support both local and distributed transactions. The transaction will provide the transaction ACID properties of atomicity, consistency, isolation, and durability. These properties will scale from embedded to enterprise-level environments. A PersistenceManager is a Transaction factory. The following methods are supported in the Transaction interface:

```
public boolean isActive();
public void begin();
public void commit();
public void rollback();
```

The JDO architecture is defined such that it can be employed in embedded environments, two-tier client-server environments, or application-server environments. In the case of an application-server environment, JDO uses the J2EE Connector architecture, making it applicable in all J2EE platform-compliant application servers from multiple vendors. The J2EE Connector facility, being developed as JSR 000016, is used for the application server interface for distributed transactions. With the Connector implementation, XAResource is used for distributed transactions and ManagedConnection is used for connection pooling and security. Developers of application components will have a standard object-persistence mechanism that will be portable across all application-server and data-storage implementations. An application server will be able to connect to multiple types of data stores in a transparent fashion. Use of the J2EE Connector mechanism is not required in a JDO implementation.

Enterprise JavaBeans

JDO has been designed to work in an EJB environment. Representatives from Sun and other companies involved with EJB participated in the design of JDO. JDO provides transparent persistence for entity beans; the class developers do not need to provide the persistence support. EJB containers manage the life cycle of beans. The JDO PersistenceManager manages the life cycle of persistent instances stored in a JDO data store. EJB containers manage distributed transactions via Connectors used by JDO transactions.

In the development of EJB entity beans, tool-generated entity beans will be used for some or all of the JDO PersistenceCapable classes. Briefly, the method ejbLoad associates the bean with a transactional instance of a JDO application class. Flushing to the database will be done during the SynchronizationbeforeCompletion callback. Business methods will be delegated to the JDO instance.

With EJB session beans, the developer implements beans by explicitly using JDO APIs. A PersistenceManager is instantiated when the EJB session bean is activated. The demarcation of transactions can be managed by either the session bean or EJB container.

Extents

The set of all instances of a class in the database is called an *extent*. This is similar to a table in a relational database. A PersistenceManager is a factory for extents and has the following method:

```
public Collection getExtent(Class pc, boolean subclasses);
```

The argument pc should be the Class object of a class that implements PersistenceCapable. The subclasses argument is used to indicate whether the collection should also contain instances of classes that extend the class referenced by pc.

Queries

A PersistenceManager is also a factory for Query objects. The query constructs are intended to be query language neutral, not tied to a particular query language such as SQL. Though neutral, it has been designed to allow optimizations for specific query languages (including SQL). This includes support for compiled queries. The Query architecture has also been designed to work well in multitier architectures and handle large result sets well.

A Query performs a filtering operation: It takes a Collection as input and produces a new Collection as output. A query requires a collection of candidate instances as input, which could either be an extent or simply a collection in the JVM. The query also requires the class of the candidates and the filter to apply. The filter has a syntax similar to a Java boolean expression; the intent is to have Java syntax as opposed to the syntax found in declarative query languages such as SQL.

The query examples below use the following application classes:

```
class Department {
    Collection emps;
}
class Employee {
    String name;
    Float salary;
    Employee boss;
}
```

(At the time of this writing, the query facilities of JDO were still being developed. The examples described here do not give comprehensive coverage of all the facilities that will be provided in the final specification.)

The identifiers used in a filter are in the scope of the candidate class. The filter String filter = "salary > 100000"; can be used with a Query where the candidate class is Employee. The Query interface has a method called setFilter to set the filter to use when the query is executed. Each employee with a salary higher than "100000" will be in the result collection. You can also use navigation; the identifiers used with a reference are in the scope of the reference type:

```
String filter = "salary > boss.salary"
```

In this case, the boss reference refers to another Employee instance. If it had referred to a different class, the member would need to be associated with the referenced class.

Query parameters can be used to substitute values during query execution. A parameter has a name and type. The following line declares a parameter:

```
query.declareParameters("float sal");
query.setFilter("salary > sal");
```

The parameter declaration is a String containing one or more parameter type declarations separated by commas, similar to Java formal parameters. When the query is executed, a value must be provided for each parameter:

```
result = query.execute(new Float(50000));
```

Note that primitive values must be passed as wrapper objects, and the filter can compare primitive values and wrapped numeric Objects, performing the appropriate unwrapping and numeric promotions.

It is also possible to iterate over elements of a collection and express query constraints involving the objects referenced in the collection. A method called contains is defined on collections in a query to associate an object reference with each element of the collection. Assume we are filtering a collection of Department objects and declare the following variable:

```
query.declareVariables("Employee well_comp");
```

The values of parameters are set in the execute call; the values of variables are dynamic and vary during the execution of the filter. Here is the call to set the filter that uses the variable:

```
query.setFilter(
"emps.contains(well_comp) && well_comp.salary > sal");
```

While the filter is executing, for each Department object in the collection being queried, each element of the emps collection will be assigned to well_comp. This syntax allows you to navigate through multiple levels of an object hierarchy by using multiple variables. Note that this is only one strategy; an equivalent strategy with an Extent in a relational JDO implementation might involve construction of an SQL statement with joins to be executed in the backend database.

Reference Implementation

All specifications developed within the Java Community Process must have a reference implementation and test suite before they are considered complete. A JDO reference implementation will be developed and provided along with the specification. Described here are the current plans for the reference implementation. This will also provide you with a better understanding of how implementations provide transparent object persistence. Some of the information presented here will be common across all JDO implementations that get developed.

Each class of an application fits into one of three categories. A class can be *persistence-capable*, which means it is able to have instances stored in the database. Instances can be either transient or persistent. There are also classes that will never have instances stored in the database, these are referred to as *transient* classes. Many of the Java system classes—such as File, Socket, Thread, etc.—are transient and can never have instances stored in the database. A third category is *persistence-aware* classes. A class that is persistence-aware is not persistent-capable, as no instances of the class can be stored in the database. However, the class accesses the public data members of a persistent class. In most implementations, if the developer practices encapsulation and only has private data members in each persistent class, there will not need to be any classes that are persistence-aware.

For each field in a class, it is necessary to declare whether it

continued on page 144

is persistent or not. The transient designation used to indicate whether a field is serialized is an independent concept. A field might be transient for serialization purposes but persistent for JDO purposes. A field may also be derived, which means that it is a transient field, but its value is derived from the values of other fields that are persistent. If the JDO implementation is being used with a database that uses primary keys (such as a relational database), it is necessary to declare which fields of the class are components of the primary key.

The JDO reference implementation is defined to use a Class Enhancer, which post-processes Java byte code to enhance it with the code necessary to provide transparent persistence. The Enhancer will change a persistent class to declare that it implements the PersistenceCapable interface defined in the javax.jdo package. The methods defined in this interface are used for querying and managing the life cycle of an instance. A public field named jdoFlags is added to the class to indicate whether it is OK to read or write the object. A public field called jdoStateManager is also added to reference a StateManager object, which handles the transfer of the object's data between memory and the implementation's data-store buffers. Methods are provided for loading and storing groups of persistent fields. Both get and set methods are provided for each field type (e.g., getIntField and setIntField). These methods use the jdoStateManager to perform functions. Some of the methods include:

- jdoLoad: copy values from the StateManager to fields in the object.
- jdoStore: copy values from the object's fields to the StateManager.
- jdoCompare: compare two objects, field by field.
- jdoCopy: copy one object to another, field by field.

The StateManager manages the transfer of data between the objects and the database, but how this is done will differ across implementations. The bulleted methods are meant to be used by the JDO implementation to support transparent persistence. These are not considered part of the interface that the application normally uses, but have been described to provide some understanding of how the implementation would support persistence.

The reference implementation has the notion of a *default fetch group*—a set of fields that are copied from the StateManager as a group. They are often, though not always, read from and written to the database as a group. These fields are directly accessible by the application once they have been read from the database. The jdoFlags field added by the Enhancer indicates the status of all the fields in the default fetch group. There are also fields that are usually not in the default fetch group. These fields are intermediated by the StateManager individually each time they are used by the application. There is additional processing that occurs every time the application reads or writes these fields, with calls made to the StateManager. Field types that are often not in the default fetch group include all object references and primary key fields.

The JDO specification should be released for public review by the time this article is published. I encourage you to obtain a copy of the specification and learn more about it. Over the next year we will see JDO implementations become available in the market, providing a standard API for transparent object persistence supported across object and relational databases. ■

Acknowledgment

Many thanks to Craig Russell at Sun, the specification lead for JDO; he provided assistance in preparing this article and approved the early publication of this material.