

The JDO object model

THE JAVA DATA Objects (JDO) API is being defined within Sun's Java Community Process. It is establishing a standard for transparent persistence of Java objects in transactional datastores. Commercial implementations are planned for both object and relational databases. Instead of managing rows and columns of primitive values with JDBC, applications will be able to work exclusively with Java object models. Sun is developing two reference implementations: one resides on top of the file system, while the other runs as a layer on top of JDBC. Work on the JDO Compatibility Test Suite is also under way, and the JDO API will be released soon. My column will now focus primarily on JDO, covering different aspects of the API.



The supported object model is very important when using a transparent persistence facility. It determines the modeling constructs available to you both in the database schema and in your in-memory classes. If there are limitations, they can

become quite frustrating and it can be difficult to work around them. Fortunately, the JDO object model has very few limitations. This article thoroughly explains JDO's object model.

The JDO object model of an application is determined by a set of Java classes and an XML metadata file. The XML metadata file contains modeling directives that either override the default semantics specified in Java or provide additional semantics not expressible in Java. Every application class that should be persistence-capable must be listed in this file. Listing 1 has the XML Document Type Descriptor (DTD) for the JDO metadata. The elements and attributes defined in this DTD will be described as we examine JDO's object model. The Java classes and XML metadata together determine an application's persistent object model.

Class Enhancement

For a Java class to be persistence-capable in a JDO environment, the class must be *enhanced*. These

enhancements are required for the class to support transparent persistence in a JDO runtime environment. They can be done with either a source file or a class file. The JDO specification details all the enhancements that must



be made to a class. The most common approach to enhancement will be to use an enhancer that reads a set of Java .class files and the JDO metadata file, and then generate a set of enhanced class files. One of the changes made to a class is that it is enhanced to implement the JDO interface PersistenceCapable. This interface provides the functionality required to manage the transparent persistence of instances in a JDO runtime environment. (All JDO interfaces and classes are defined in the Java package javax.jdo.)

The reference implementation provides a Reference Enhancer that produces class files enhanced according to the JDO specification. All JDO runtime implementations must support classes enhanced by this reference enhancer. An implementation can implement its own enhancer, which may provide additional capabilities. But these vendor-specific enhancements cannot conflict with the standard reference enhancements. This allows Java classes enhanced by either the reference enhancer or any specific vendor's enhancer to be usable in any JDO runtime environment. Class enhancement will be binary compatible across all JDO implementations.

Persistence-capable Classes

A class is either persistence-capable or not. A user-defined class will be persistence-capable as long as it avoids the few limitations JDO imposes. A user-defined class whose state depends on remote objects or is inaccessible in Java (due to the use of the Java Native Interface) will not be persistence-capable. The system-defined classes, in general, are not persistence-capable. This includes the classes defined in java.lang, java.io, java.net, etc. These classes also cannot serve as the type of persistent fields in a persistence-capable class.

A persistence-capable class can have instances that are persistent—i.e., stored in the database—but instances can also be transient (they only exist within the context of the JVM in which they were created). In fact, all instances start out transient and

David Jordan is a director at Trifolium Inc. in Raleigh, NC.

LISTING 1

JDO XML metadata

```

<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT jdo (package)+>
<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>

<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type
(application|datastore|none) 'datastore'>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass
CDATA #IMPLIED>

<!ELEMENT field ((collection|map|array)?, (extension)*)?>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier
(persistent|transactional|none) #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value
(exception|default|none) 'none'>
<!ATTLIST field default-fetch-group
(true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>

<!ELEMENT collection EMPTY>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element
(true|false) #IMPLIED>

<!ELEMENT map EMPTY>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>

<!ELEMENT array EMPTY>
<!ATTLIST array embedded-element (true|false) #IMPLIED>

<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>

```

some are converted to persistent instances either by explicit calls to the JDO interface (`makePersistent`) or implicitly via *persistence-by-reachability*. Persistence-by-reachability makes a transient instance persistent if it is persistence-capable and referenced by a persistent instance when a transaction commits. This capability is then propagated to all the persistence-capable instances

this newly persisted instance references. This allows a large collection of related instances to be persisted in the database merely by having a “root” object persisted or referenced by another persistent instance.

The class element in the XML metadata is used to specify each class that should be persistence-capable. These are defined within a package element that corresponds to the Java package of the class. In JDO, a class can optionally have an Extent that allows the application to iterate to every persistent instance of the class. The XML metadata has an attribute `requires-extent` on the class element to indicate whether a class has an Extent. If the application explicitly makes instances persistent by calling the `makePersistent` method, the class must have an Extent. A class that only has instances made persistent implicitly through persistence-by-reachability is not required to have an Extent. There is often overhead in an implementation to maintain an Extent. If an application never needs to directly access the instances of a class, but usually accesses them by navigating through other instances, then an Extent would not be necessary.

JDO also supports class inheritance. Within an inheritance hierarchy, each class can either be transient or persistence-capable. For a given class *C*, the subclasses can be transient or persistent, independent of how *C* is defined. In the XML metadata, it is necessary to inform the most immediate superclass that is persistence-capable for each persistence-capable class. This is specified with the `persistence-capable-superclass` attribute of the class element. Classes that do not have a class above them in the inheritance hierarchy that is persistence-capable do not use this attribute.

First-class and Second-class Objects

There are two categories of persistent-capable classes: First-Class Objects (FCO) and Second-Class Objects (SCO). An FCO has JDO identity, which is a unique identifier associated with each persistent instance in the datastore. JDO identity is discussed in detail in the next section of this column. There is a guarantee that a given FCO instance is only represented once in memory for a given `PersistenceManager`. (A `PersistenceManager` establishes a transaction and database connection context for an application.) An FCO instance can be referenced by any other FCO instance in the database, using its JDO identity. When an FCO is changed, all the other FCOs that reference it will immediately see the change. (Before the transaction commits, it will only be seen in the current transaction’s cache.) In contrast, an SCO does not have JDO identity. An SCO is either an instance of a `PersistenceCapable` class or a class provided by the JDO implementation that extends a system-defined class. Arrays are treated as SCOs in JDO.

You specify that a field references an SCO by using the attribute `embedded` of the field element with a value of `true` in the XML metadata. Note that this directive is placed on a field that references an instance, not on the

class itself. Therefore, in the case of a class that implements `PersistenceCapable`, in some situations a class can be an SCO, while in other situations it could be used as an FCO. The embedded directive also suggests to the implementation that the state of the SCO should be stored along with the state of the containing FCO. This is how objects can be clustered in a JDO implementation.

An SCO is either owned by a single FCO or it is unowned. Sharing of SCOs by multiple FCOs is not permitted in the database. In memory, references may be established from multiple FCOs to a single SCO. When the transaction commits, however, the fact that the references were to a common, single SCO is lost. Each FCO will have its own copy of the SCO state in the database. The next time these objects are brought into memory they will have their own copy of the SCO, each with its own distinct Java identity.

JDO Identity

Each FCO instance in the database has JDO identity. There are three types of JDO identity that are supported: application identity, datastore, and non-datastore. In *application identity*, the identity values are managed by the application but enforced by the JDO implementation. This is also commonly referred to as primary key identity and is the form used in relational databases. This identity uses the values of one or more fields in the class as the values of the identity. For a class A that has application identity, you must have a separate class that contains attributes with the same names and types as the attributes in the identity of A. With *datastore identity*, the identity value is generated by the datastore and is not tied to any values in an instance's fields. This form of identity is often used in object databases. *Non-datastore identity* is used with a datastore that does not have a unique identifier to reference an instance in the datastore. But a JDO implementation must still provide support for uniquely referencing an instance in memory. For datastores that do not provide a unique persistent identity, this third form of identity is used. Log files and text files are example datastores that do not have an identifier for their contents.

The application specifies the type of identity for a class in the XML metadata by using the `identity-type` attribute of the class element. If the value for the `identity-type` attribute is `application` for a class C, the application must also specify the class that serves as the identity class for class C. The `objectid-class` attribute of the class element is used in the XML to define this for class C.

Class and Field Modifiers

All of the class and field modifiers supported by the Java language are supported in JDO. This includes the following Java keywords: `public`, `private`, `protected`, `static`, `transient`, `abstract`, `final`, `synchronized`, and `volatile`. The first three

modifiers control access to the members of the class. The enhancer changes the direct use of each field to a call to a special accessor/mutator method, which has the access modifier originally declared for the field. It also changes all fields originally declared to be `public` or `protected` to be `private`. This is done to guarantee that all classes that access non-private fields are enhanced. The `synchronized` and `volatile` modifiers do not affect the JDO environment and can be used by the application.

If a field is declared `static` or `final`, it is always treated as a non-persistent field. The default policy is that fields declared `transient` in the Java source are not persistent. But this can be overridden in the XML metadata file by declaring the `persistence-modifier` attribute to be `persistent` for the field's element declaration. Similarly, this same attribute can be used with a value of `none` to state that a field that is persistent by default should be `transient`.

The XML metadata allows you to specify how null values are handled in the database. There is an attribute `null-value` defined for the field element. A value of `"none"` stores a null value as a null if the database supports them; otherwise, it throws a `JDODataStoreException`. If the attribute has the value `"exception,"` an exception is always thrown at runtime when an attempt is made to store a null value for the field. Finally, if the attribute has a value of `"default,"` then the null value is converted to whatever default value is used by the database for null values of the field's type.

Field Types

It is important to understand what types a field can have in a persistence-capable class. Table 1 lists the atomic (non-collection) data types that can be used in JDO—basically, all of the primitives, the primitive wrapper objects, and a few others like `Date` and `String`. None of the system-defined classes (those in packages `java.lang`, `java.io`, `java.net`, etc.) are capable of being persistence-capable or field types of persistence-capable classes unless explicitly stated in the specification. The application can have fields that are references to instances of `PersistenceCapable` classes. You can also declare fields for Java interfaces that have been defined by the application. But the referenced instances should be persistence-capable. One can also have arrays of any of the types listed in Table 1, as well as

Table 1. Atomic field types.

<code>boolean</code>	<code>Boolean</code>	<code>Locale</code>
<code>char</code>	<code>Character</code>	<code>String</code>
<code>byte</code>	<code>Byte</code>	<code>Date</code>
<code>short</code>	<code>Short</code>	<code>BigDecimal</code>
<code>int</code>	<code>Integer</code>	<code>BigInteger</code>
<code>long</code>	<code>Long</code>	
<code>float</code>	<code>Float</code>	
<code>double</code>	<code>Double</code>	

Table 2. Collection types.

COLLECTION INTERFACES	COLLECTION CLASSES	
Collection	Vector	Hashtable
List	ArrayList	LinkedList
Set	HashSet	TreeSet
Map	HashMap	TreeMap

arrays of references to interfaces and PersistenceCapable classes. However, arrays are optional in implementations.

JDO also supports collection field types. Table 2 lists all the collection interfaces and classes specified in the JDO specification. JDO will be used with a variety of database technologies—object and relational databases are just two examples. Furthermore, implementations are planned for relational databases where the schema cannot be altered to meet any requirements of an object model used by applications. To handle this restrictive environment, JDO only requires the HashSet collection to be supported. But object database implementations and relational database implementations that do underlying schema generation or enhancement will support a larger set of the collection types listed in Table 2.

As shown in the XML DTD in Listing 1, a collection field is described by a collection XML element nested within a field XML element. Within the XML description of a collection, it is necessary to specify the type

of the elements by using the element-type attribute. If the collection itself is to be embedded in the persistent object, the embedded attribute associated with its field element is used. The collection's elements themselves can also be embedded by using the embedded-element attribute within the collection specification.

The characteristics of a map differ from a collection and, therefore, a map has its own specification in the XML metadata. A map has a key and a value. It is necessary to identify the type of the key with the key-type attribute, and the type of the value with the value-type attribute associated with the XML map element. Similarly, you can specify that the key is to be embedded by using the embedded-key attribute, and that the value is embedded by using the embedded-value attribute of the map XML element.

I have summarized the object modeling capabilities possible within the Java Data Objects API. Essentially, you have access to all that Java has to offer—you only need to add a few additional modeling directives in an XML file. Some vendors are planning to provide a GUI tool to specify this information. The tool would manage the actual XML metadata file for you. The JDO API should be released as a standard this summer, while commercial implementations should be available before year's end. Applications will be able to directly persist Java classes in a wide variety of datastores. ■