JDOcentral.com
Developer's Community for Java Data Objects

A Comparison Between Java Data Objects (JDO), Serialization and JDBC for Java Persistence

David Jordan
Object Identity, Inc.
www.objectidentity.com

Java Data Objects (JDO) is a new industry standard data management interface for Java applications. JDO was developed through the Java Community Process (JCP) under Java Specification Request (JSR) 12. Development of the JDO specification, reference implementation, and compatibility test suite was led by Craig Russell of Sun Microsystems, Inc. Contributors to the JDO specification included an industry expert group of representatives from companies providing relational database, object database, and object-relational mapping products as well as vendor-neutral consultants expert in persistent object technologies.

A Java application defines its data model in terms of a set of interrelated classes. During the application's execution, instances of these classes are instantiated and associated with other instances. The data for these instances usually needs to be stored so it can be shared and accessed by other applications. Serialization and Java DataBase Connectivity (JDBC) are the two existing standard application program interfaces (APIs) for persisting data in the Java community. After reviewing the capabilities and fundamental differences of these two technologies, we will examine a new standard for persisting data called Java Data Objects (JDO). JDO provides many of the benefits of serialization and JDBC without their corresponding limitations.

Serialization provides direct support for storing instances of an application's classes in a file but lacks important facilities like transactions and queries. JDBC provides Java applications with an interface to issue SQL commands, usually to a relational database system (RDBMS). JDBC does provide transactions and queries but it lacks support for the application's object model. Though serialization and JDBC both support the storage of data, they are substantially different technologies. Each presents developers with mixed offerings of advantages and disadvantages for building robust, object-oriented Java applications.

The Java Data Objects (JDO) industry standard provides persistence for an application's object model in an environment that supports transactions and queries. JDO provides a binary compatible interface for storing Java objects across a broad range of data management implementations including relational databases, object databases, flat files and mainframe connectivity and transaction systems. JDO is aptly being characterized by the phrase "write once, persist everywhere." JDO is set to become the dominant standard for persisting Java data due to its direct support of the Java object model, robust offering of transactions and queries and vast portability among data management options.

Serialization

Standard in every Java environment

Serialization is a standard component of every Java Virtual Machine (JVM), making it free and ubiquitous in all Java environments. It allows an object graph to be serialized into a stream (specifically an `ObjectOutputStream`), which can be associated with a file. An instance is serialized by passing it as a parameter to the `writeObject` method of `ObjectOutputStream`. The entire graph of objects reachable from the instance is then serialized into the stream. The object graph is reconstructed later by deserializing the data from an `ObjectInputStream`.

Easy to Use

Serialization is very easy to use and easily stores the object models defined in the Java language. A class to be serialized declares that it implements `Serializable` or `Externalizable`. This allows the Java runtime environment to perform default serialization of the class without any other programming by the application. If the default serialization provided by the JVM is insufficient, the application can implement methods `writeObject` and `readObject` to define the serialized representation of an instance. If fields in a class should not be stored, the application declares them as transient. The application can also identify fields to be serialized by including a static array field in the class definition of type `ObjectStreamField` named `serialPersistentFields`.

Uses Java classes as the data model

Serialization uses Java classes as its data model. The relationships among classes are represented by references and collections of references. Entire object graphs are serialized such that the relationships among reachable instances are preserved in the serialized form of the object graph.

Lacks features of a robust database environment Serialization lacks many of the features found in a robust database environment. There is no support for transactions. Without concurrency control, nothing prevents multiple applications from serializing to the same file and corrupting the data. Serialization also lacks the ability to perform queries against the data. The granularity of access is an entire object graph making it impossible to access a single instance or subset of the serialized data.

Lacks scalability required by most applications It is rare that a single serialization can store all the data needed by an application. Applications must manage multiple serializations, either in the same file or different files. Serialization lacks support for identity and the coordinated management of the instances in storage and memory. This means that developers must take extreme care to avoid storing and loading redundant instances. If different parts of a large application read the same serialization more than once, multiple copies of this instance will reside in memory. Redundant copies would make coordinating separate updates extremely difficult. These issues collectively result in a lack of scalability needed by most applications.

JDBC

Provides access to SQL JDBC provides a standardized interface for a Java application to issue SQL statements to a relational database. By leveraging the popularity of SQL and relational database technology, JDBC has gained broad acceptance in the market. SQL is the standard query language for relational databases. The transactions and queries of SQL are directly supported in JDBC.

Uses SQL's relational data model JDBC uses SQL's relational data model as its representation of data. This data model consists of a set of tables that have rows and columns. As an entity in the application domain is mapped to a table, its attributes are mapped to the columns of the table. A specific entity instance is represented by a row in the table. To uniquely identify a specific instance, the table uses a primary key that consists of one or more columns of the table whose values uniquely identify the instance. Relationships among entities are represented by column(s) containing a foreign key that matches a row's primary key. These relationships are established in a SQL query by expressing join conditions.

Operations are expressed relative to tables, rows and columns All operations used to manage persistent data with JDBC are expressed relative to the relational model of tables, rows and columns. To add a row to a table, the JDBC application issues a SQL INSERT statement which provides values for the columns of the table. A SQL UPDATE statement is used to modify a row. The application must provide the values for the primary key and the modified columns. A row is deleted by issuing a SQL DELETE statement. A SQL SELECT statement is issued to query the database. The WHERE clause of SQL is used to express join conditions and other query constraints. JDBC returns a ResultSet object, which is used to iterate over the rows of the result, with methods provided to extract individual column values. The granularity of data passed between the Java application and the SQL implementation is at the level of a table cell for each of these operations.

Rarely portable across products Though there is commonality among SQL implementations, there are many different dialects of SQL. SQL code written for one RDBMS will not always work with another. So even though SQL is a standard, SQL queries are rarely portable across products.

Does not support Java classes JDBC applications must work directly with the SQL data model, because JDBC does not support Java classes. An application is forced to deal with two very different data models: the Java object model and the relational data model. Developers must decide whether the stored entities of the application domain will be represented as Java objects. Any book on object-oriented design makes it very clear that the real advantages of object-oriented development are only achieved by representing your application domain entities as objects.

Sometimes developers choose not to represent their data as objects. Without direct support for the mapping between the object and relational data model, and with tight schedule constraints, Java developers working independently often develop JDBC code that is very procedural in nature, making little use of objects.

Procedural instead of Object Oriented

Consequently, their application code attains very little reusability. The developers must have a thorough understanding of the relational schema to interact with the data model. It is rare to have a development staff that is proficient in Java and also highly skilled in database and SQL development. This can often result in applications getting written with inefficient SQL expressions and an inability for developers to attain most of the benefits of object-oriented development.

Processing data through SQL instead of Java

Some applications manage data that does not require the processing capabilities of Java. SQL has very useful features for dynamically associating information and performing complex operations like GROUP BY. Some applications can directly express the necessary computations very succinctly in SQL. In these circumstances there may not be any benefit to representing the data in Java. Developers should do some data analysis to determine whether their application fits this situation.

Proprietary Object-Relational Mapping

It may be decided that a Java object model should be defined which corresponds to the data stored in the relational database. Once the object model, relational model and a mapping between them are defined, most of the application developers are freed from the need to understand the relational schema and can focus on the Java object model. But a decision needs to be made as to whether the development staff should build their own object layer or buy an off-the-shelf object-relational mapping (ORM) product.

Significant risks in home-grown object-relational mapping

Some organizations choose to implement their own object mapping layer. They justify this by reasoning that they will have complete control over all the software in their system and do not need to incur additional software licensing costs. There are many negative consequences with this approach. There are significant costs and risks in developing such an object mapping layer. First, it must be developed and tested before the application objects can be defined. This can cause a substantial delay in completion of the application. Second, a staff often does not have the expertise required to properly develop an effective mapping layer. The result ends up being deficient in functionality. The costs of development, documentation, training and maintenance of a home-grown object mapping layer becomes a significant portion of the overall cost of engineering the product.

Application development is quicker if a commercial object-relational mapping product is purchased. Prior to JDO, this meant committing to a proprietary API supported by a single vendor. To minimize risks, a lengthy evaluation process was performed to assure the right choice was made.

Limitations of proprietary object-relational mapping APIs. Adopting a proprietary API is risky

Many existing mapping products impose modeling constraints on the Java application. These include:

- Loss of encapsulation caused by the need to define public getter and setter methods for each persistent field
- Limited extensibility caused by lack of support for inheritance
- Lack of support for polymorphic references
- Overdependence on a single vendor because of the required use of vendor-specific class libraries

The lack of a standard API supported by multiple providers makes the selection process difficult and risky. Every application developed with JDBC has been forced to make the decision whether to define an object model and, if so, which proprietary object-relational mapping product to use. The time necessary to make these decisions can cause a substantial delay in application implementation.

Java Data Objects

JDO: database facilities of JDBC and Java integration of Serialization

JDO is changing all this. JDO allows developers to directly persist Java object models into a data store. It supports transactions, queries and the management of an object cache. Multiple vendors provide implementations for relational databases, object databases, file stores and mainframe connectivity environments. Applications

based on JDO will be binary compatible across all implementations. For Java applications that want to represent their persistent data in an object model, JDO combines the database facilities found in JDBC with the level of Java integration found in serialization. The following figure illustrates where the three APIs are positioned relative to their support for Java objects models and database facilities.

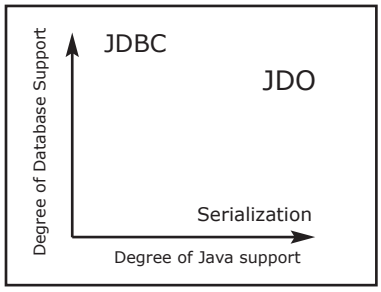

Figure 1: API Support of Java and Database

Data Model

JDO uses the Java object model

An application's Java classes serve as the data model for JDO. All of Java's class and field modifiers are supported. Fields types, that can be used, include the primitive types, wrapper classes, references, interfaces, `String, Date, BigDecimal` and `BigInteger`. JDO also supports the standard Java collections defined in the java.util package. Relationships among instances are represented using the Java convention of a reference when the cardinality is one and by a collection when the cardinality is greater than one. Applications apply encapsulation by declaring data members private and providing only the methods necessary to represent the abstraction being modeled. Applications can also use inheritance, polymorphism and interfaces.

There are no JDO-specific types that must be used for defining persistent classes. In fact, the persistent classes do not need to import anything from JDO. In many cases, existing Java classes in their compiled form can be used in a JDO environment, allowing the application to store instances of classes acquired from a third party.

Metadata

JDO metadata specifies additional persistence information

Though the data model is specified via a set of Java classes, it is necessary to provide the JDO implementation with some additional information related to persistence, that is not directly expressible in Java. A metadata file written in XML is used to specify additional persistence-related information. At a minimum, it is necessary to specify which classes are persistent. It is also necessary to specify the element type of the collections stored in the database and whether JDO should maintain an extent for a class. An extent is the mechanism by which an application can access all the instances of a class.

Supports transient fields in persistent classes

Just like Serialization, fields declared as transient are, by default, not stored in the database. You can override the default behavior in the metadata by declaring that a transient field should be stored. Likewise, you can also specify that a field which had not been declared transient in Java should not be stored. Thus JDO allows the persistence of a field in the database to be different from how the field is handled when serialized. You have complete control over which fields are stored.

Class enhancement

Adding JDO to a class requires that it be enhanced

Adding JDO behavior to a persistent class requires that it be 'enhanced.' The JDO specification defines these required enhancements to classes in detail. Enhancement adds two data members and a set of methods to each persistent class. It also alters field accesses to ensure the data is in memory. Enhancing classes provides transparent access to the objects in the database, so the state of objects can be mapped between memory and the database.

Enhanced classes work with any JDO implementation

Classes can be enhanced at the source level by hand or by using a tool, referred to as the primary key. A more common approach, however, is to use a tool which adds the necessary enhancements directly to the .class files, such as the Reference Enhancer developed by Sun Microsystems. A JDO implementation may provide its own class

enhancer, but all implementations must support the standard reference enhancement defined in the JDO specification. A class enhanced by any vendor's enhancer must work with any other JDO implementation. Enhancement has been defined such that a single Java class can even be used concurrently with multiple JDO implementations in the same JVM.

The following figure illustrates the enhancement process. A class file produced by a Java compiler and a JDO metadata file specifying the additional persistence information are used as input into an enhancer. The enhancer produces a new class file with the enhancements necessary for the class to be managed in a JDO environment.

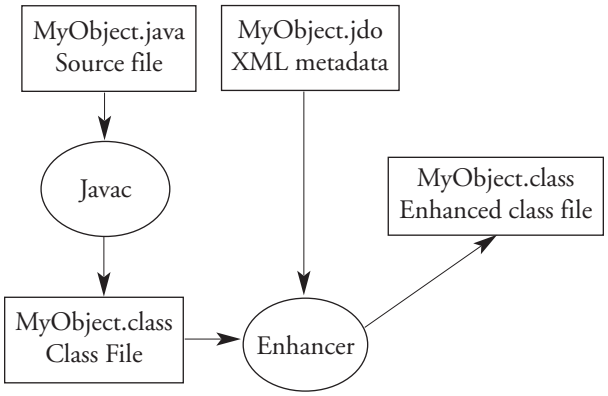

Figure 2: Class enhancement process

Identity

Instances have unique identity

Each persistent instance has a unique object identifier used to reference it in a database. JDO defines different forms of identity: datastore, application and nondurable identity. The form of identity used for a class is specified in the metadata. Nondurable identity is used when the datastore does not have a means of uniquely identifying and accessing an instance. A datastore identifier is a unique identifier generated by the implementation that is not dependent on the state of the object.

An application identifier does depend on the state in an object, it is based on one or more fields in the class. In the metadata for a persistent class you need to specify which fields are part of the primary key. The application must also define an application identity class that contains fields with the same name and type as the fields of the primary key. This application identity class is also specified in the metadata.

Transactions

JDO supports transactions

JDO's primary means for managing persistent instances is the `PersistenceManager` interface. A `PersistenceManager` instance manages a cache of objects associated with a transaction. Its method `currentTransaction` returns an instance of the `Transaction` interface which is used to begin, commit and rollback a transaction. The `Transaction` interface is used for all transaction management operations.

All JDO implementations provide transactions that have a pessimistic concurrency control policy. These transactions acquire locks as objects are accessed. Support of optimistic transactions is an optional JDO feature. Optimistic transactions do not acquire locks and check for concurrent update activity until a transaction is committed. This reduces the resources necessary for locking and provides performance advantages when concurrent access to the same objects is rare.

JDO supports the `Synchronization` interface specified in the Java Transaction API (JTA). The application registers an object with the `Transaction` instance that implements the `Synchronization` interface. This allows the application to be notified when the transaction is about to perform its commit phase and also at the completion of commit processing, indicating whether or not it succeeded with the transaction interface.

Object Access

Making instances persistent

Instances of a persistent class are either persistent or transient. An application explicitly makes an instance persistent by calling the `PersistenceManager` method `makePersistent`. JDO also supports persistence-by-reachability, which persists all instances that are reachable via a reference contained in another persistent instance. Persistence-by-reachability is performed without any explicit calls by the application.

This provides transparent persistence of object graphs, similar to serialization. However, JDO also allows the instances to be accessed and managed at an object-level of granularity.

Accessing stored objects is transparent An application does not need to make explicit calls to access a related object. It simply traverses a reference or iterates through a collection and the JDO runtime returns the objects from the database. The application does not need to know whether or not the objects have already been loaded into memory. JDO brings the objects in memory on demand and ensures that only a single copy of the object is in memory for the transaction. Every attempt by the application to access a particular persistent instance returns the same instance in memory, regardless of how the instance is accessed. This capability is provided by JDO's cache management facilities. Developing with the JDO object cache is a major paradigm shift in data access for applications. Developers who have never worked with such an environment get very excited when they discover this capability. Once they become accustomed to using it, they are reluctant to revert to previous approaches.

Updating stored objects is transparent The application does not need to make an explicit call to mark an instance as updated. When a field in a persistent instance is modified, a state variable maintained by the `PersistenceManager` is set to indicate that it has changed. The `PersistenceManager` tracks all updates made by the application, which includes making instances persistent or deleting them. When the associated transaction commits, all updates are propagated to the database without requiring any programming by the application.

The `PersistenceManager` method `getExtent` is called to get an Extent, which is used to access all the instances of a class. Methods named `newQuery` are used to construct Query instances for performing queries against the database. An instance can be accessed via its identifier by calling the method `getObjectById`. Instances are normally accessed by iterating an extent, issuing a query, or simply navigating to related instances.

JDO Supports concurrent transactions An application can create multiple `PersistenceManager` instances to support concurrent transactions in the same JVM. Each `PersistenceManager` maintains its own cache of persistent objects. The `PersistenceManager` instances can be from the same or different JDO implementations. An application server environment typically maintains a pool of `PersistenceManager` instances.

JDO works with JCA and EJB JDO has been defined to use the Java Connector API (JCA) when used in a managed environment so that JDO implementations and applications can easily integrate into an application server environment. The JDO specification also details the conventions for using JDO as the persistence mechanism in Enterprise Java Beans (EJB) environments.

Query Support

JDOQL is independent of underlying implementations Queries in JDO applications are expressed using the Java Data Objects Query Language (JDOQL). JDOQL's syntax and expressions are based largely on Java. The goal of JDOQL is to provide a uniform query syntax that is independent of underlying datastore query languages. JDOQL has been defined so that its queries can be optimized to specific underlying query languages, in particular SQL. An implementation of JDO layered on top of JDBC would need to translate JDOQL queries into the underlying SQL used by the RDBMS. The JDOQL language and syntax is consistent across all JDO implementations making JDOQL queries portable across all implementations. This insulates the application from portability problems such as those caused by the different SQL dialects found among RDBMSs.

JD0QL uses Java syntax In JDOQL a query is a Boolean filter that is applied to either a collection or an extent. The result is a collection containing instances for which the filter expression evaluated as true. Parameters can provide values to be used in query expressions for constraining the result. They are declared using the same syntax Java uses for declaring method parameters. Variables are used to iterate collections and extents. They are declared using Java's syntax for local variables. Types used in the query are imported with the same import syntax found in Java.

Filters in JDOQL contain many of the operators found in Java, including the equality, comparison, logical and arithmetic operators. JDOQL goes beyond Java by promoting numeric operands of primitive and wrapper classes for the comparison and arithmetic operators. Though the initial input for the query is a collection or extent of objects of a single type, you can still navigate in the filter to related objects. References are traversed in a query by using the same dot operator used in Java. You can iterate over the elements of a collection by associating a variable with a collection via the contains method.

JDOQL provides a filtering mechanism that understands and can navigate the Java object model. Its syntax and operators are based on Java. Most importantly, it provides Java developers with a familiar and consistent language environment that will be portable and work across all implementations.

Summary: What a JDO Application Looks Like

Minimal intrusion

JDO lets application developers take full advantage of the object paradigm. In fact, JDO has a minimal intrusion on how Java software normally accesses related objects in memory. Most of the application does not need to perform any explicit database operations. A typical application first begins a transaction and then performs a query to get some specific "starting point" objects of interest. The remainder of the application flow simply involves navigating the object model, accessing and updating instances as necessary, without any explicit calls to the database. At some point the application commits the transaction and the JDO implementation propagates all the updates made by the application to the database.

JDO's transparency means few explicit database calls are needed

The application does not need to provide any mapping between data models. It also does not need to track what has changed and remap those changes back into the data model of the underlying database. The application just traverses its object model in memory and the JDO runtime takes care of providing the instances from the database on demand. JDO provides significant development productivity advantages over other approaches. Using JDO will dramatically reduce the costs of developing an application.

Conclusion

Until recently, serialization and JDBC were the only two persistence APIs considered standard by the Java community. Serialization can directly persist an application's object model but it lacks the database features that are essential for building reliable and scalable applications. JDBC provides reliability and scalability by interfacing with SQL-based datastores but its use of the relational model makes it difficult to integrate with an application's object model. JDO combines the best features of the previous persistence APIs, without their associated deficiencies. By combining the Java object model support and portability of Serialization with the reliability and scalability of JDBC, JDO will compel application developers to "write once, persist everywhere!"

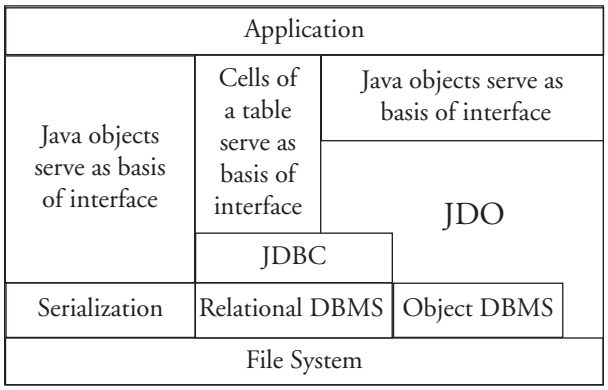

Figure 3: Alternative persistent API choices shown from a layered architecture perspective.

Conclusion continued on page 8...

Table 1: Summary of High-Level Differences Among Persistence APIs

Feature	Serialization	JDBC	JDO
Data model	Java	Relational table model	Java
Support of Java classes	Yes	No	Yes
Access granularity	Object graph	Table cell	Object
Support of inheritance and polymorphism	Yes	No	Yes
Support of references and collections	Yes	No	Yes
Unique identity	No	Primary key	Primary key or Datastore identifier
Automatic management of cache	No	No	Yes
Transactions	No	Yes	Yes
Concurrency	No	Yes	Yes
Query Language	None	SQL, each vendor has a different dialect (non-portable)	JDOQL, standard language neutral of underlying language of database vendor (portable)
Object model supported in queries	No	No	Yes
Java Connector API compatibility for application server integration	No	Planned for JDBC 3.0	Yes

URLs to visit

JDO community website: http://www.jdocentral.com

JDO within the Java Community Process: http://www.jcp.org/jsr/detail/12.jsp.

JDO web site maintained by JDO Specification Lead: http://access1.sun.com/jdo.

About the author:

David Jordan founded Object Identity, Inc. to provide JDO consulting, training and custom software development services. He has been an active member of the JDO expert group since its inception. He has been a software developer and architect involved in object persistence technology since 1985 when he initiated the development of the first C++ object database. He has had columns in C++ Report and Java Report covering object persistence technologies. He has served as C++ and Java editor for the Object Data Management Group. David is also author of the book titled "C++ Object Databases". He was also a reviewer of JDBC prior to its release and was acknowledged in Sun's JDBC 1.0 specification book published in their Addison-Wesley Java series.