
LIBeLIS
White paper
Why JDO is a critical components of j2ee ?

Why JDO is a critical component of J2EE ?

White paper

Table of content

1 Objectives

The aim of this document is to explain why JDO, the new standard for persistence of Java objects, is so important for the software industry, and why we strongly believe that it will soon become a mainstream component of the J2EE platform.

This document assumes that readers already know Java, J2EE and databases/transactions issues.

The first part briefly introduces the JDO standard, while the second part focuses on the J2EE integration.

Reference documents, links:

- Sun JDO Specification: Java Data Object 0.98 public final draft
- Sun JDO sites: http://access1.sun.com/jdo/ , http://jcp.org/jsr/detail/012.jsp
- LIBeLIS JDO site: www.libelis.com/jdo.jsp

2 JDO Overview

JDO is a new specification from the Sun JCP (Java Community Process) which deals with persistence of Java objects.

2.1 History

JDO is a synthesis of works on object persistence, and tries to propose a complete vision of object persistence. JDO inherits from both ODMG (object data management group, an independent committee that standardizes object databases) and object-relational mapping tool vendors.

- JSR #000012 approved in July 1999
- Specification Lead (Craig Russell) selected in July 1999
- Expert group formed in August 1999:
    - Apple
    - BEA
    - Ericsson
    - Excelon (Object Design)
    - Forté (Sun)
    - IBM
    - Informix
    - Lawson
    - LIBeLIS
    - Objectivity
    - Oracle
    - Orient Technology
    - Poet
    - Rational
    - SAP
    - Secant
    - Silverstream
    - Software AG
    - Sun Microsystems
    - Tech@Spree
    - Versant
    - WebGain (Object People)
    - Suad Alagic, Martin McClure, Constantine Plotnikov
- Public review draft completed in May 2000
- Introduced at JavaOne in June 2000
- Final release draft 0.93 in March 2001
- Public Final Draft 0.96 in May 2001
- Launch during JavaOne in June 2001

- First implementations:
    - Sun Forté already supports JDO 0.80 since march 2001 JDO release (no inheritance support)
    - PrismTechnology supports JDO 0.93 for RDBMS in June 2001 (no collection support), next release due August 2001
    - TechTrader Kodo supports JDO 0.95 for RDBMS in July 2001, next release due August 2001
    - LIBeLIS LiDO support JDO 0.96 for Versant in July 2001
    - Sun Reference Implementation in September 2001
        - JDBC and C-ISAM Flat Files
        - Test and certification suite
    - LIBeLIS Orcas J2EE application server with JDO container support in December 2001Sun iPlanet to be released with JDO container support in 2002

2.2 Goals

- Define persistence at object level
    - Full object model support, including references, collections, interfaces, inheritance, …
    - Fully transparent persistence: this allows to make business object totally independent from any database technology
        - Reduces development cycle (no more mapping)
        - Clearly isolates business and database expertizes in development teams
    - Universal persistence. While JDBC is limited to RDBMS, JDO is potentially able to cope with any kind of data source, including RDBMS, ODBMS, TP monitors transactions, ASCII flat files, XML files, properties files, Cobol databases on mainframes , … JDO is a clear solution for large information systems where information is stored in a wide range of heterogeneous data sources
    - Wide range of implementations covering J2EE, j2se & J2ME
    - Strong transactional model
    - Both client-server and multi-tiers architectures support

2.3 Architecture

2.3.1 JDO Packages

PersistentCapable	A class that may have persistent instances must implement this interface. Manages object life-cycle.
PersistenceManager	Represents connections to data sources. An application can open one or more PersistenceManagers.
PersistenceManagerFactory	Allows to get new instances of PersistenceManager from data sources. This factory may also acts as a connection pool.
Transaction	Allows to set transactions boundaries.
Query	Allows to explicitly and declaratively get objects from data sources using the JDO query language. **NB**: objects might also be implicitly and transparently fetched from data sources using basic navigation between references.
InstanceCallback	Defines some hooks that allows to do "special things" (like initialisations of transient attributes) during database operations (like before/after read, before/after write, …).
JDOException	Exceptions raised during JDO operations.

JDO also defines Helper classes, object identity (managed by application or by data sources).
JDO implementations may support compatible PersistenceManagers or not (when PersistenceManagers are compatible you can have references between objects stored in different databases.)
NB: in that first release JDO does not firmly defines locks and locking strategies.

2.3.2 *JDO object model*

The JDO object model is basically the Java object model, including all basic types, references, collections and even interfaces:

- All field types (primitives, immutable and mutable object types, user-defined classes, arrays, collections, interfaces)are supported but references to system-defined classes.
- All field modifiers (private, public, protected, static, transient, abstract, final, synchronized, volatile) are supported.
- All user-defined classes can be PersistentCapable except when objects state might depend on the state of inaccessible or remote objects, eg., extend Java.net.SocketImpl, native methods, …

2.3.3 *JDO object life cycle*

In order to be able to access and store objects in data source application must first get a connection to one or several data sources. A JDO PersistenceManager object represents such a connection. It can be obtained using the JDO PersistenceManagerFactory class.

Persistent objects must be instances of classes that implement the JDO PersistentCapable interface. Such classes might have both persistent and transient instances. To make an instance persistent, programmers must call the makePersistent method from the PersistentManager.

It is important to notice that JDO object be persistent or transient but even if they are transient you have JDO behaviour available, such as transaction management or object identity.

Object identity may be managed either by the application or may be delegated to the data source (which is the case with most ODBMS for instance, because the notion of ObjectID itself is part of the ODMG model).

JDO enforces a model, where persistence is automatically propagated to referenced objects. This mechanism is often called “persistence by reachability” or “transitive persistence”. It means that as soon as a transient object is referenced by an already persistent one, it automatically becomes persistent. This model might seem odd to JDBC programmers, but they will find with experience that in most cases it represents what programmers expect from a persistence framework.

Example

```
pmf = (PersistenceManagerFactory)
(Class.forName("com.libelis.lido.PersistenceManagerFactory")
.newInstance());
pmf.setDriverName("versant");
pmf.setConnectionURL(dbName);
pm = pmf.getPersistenceManager();
tx = pm.currentTransaction();

tx.begin();
Provider aProvider = new Provider(“LIBeLIS”);
pm.makerPersistent(aProvider);  // aProvider now persists
Address anAddress = new Address(“25 rue Paul Barruel”,
“France”, “Paris”);
aProvider.address = anAddress ;  // anAddress now persists

tx.commit();
pm.close();
```

Objects are brought in memory either as the result of an explicit JDO Query or during standard navigation between Java objects.
This last mechanism is a very powerful one.
You can imagine that persistent objects are located in a special part of the JVM heap that we can call the “client cache”.
Each time, you try to navigate from an object to another one, if it’s not already on memory, it will be automatically fetched from the data source, and brought into the cache in memory.

Example
Let’s suppose that the object aProvider is already loaded in memory and it’s address is not. When you write the following code:

```
System.out.println(aProvider.address.city);
```

the Address object will be loaded automatically for you.

The cache management is directly linked with transaction boundaries, it means that this cache is flushed at the end of each transaction and all entries are marked as invalid. The next time objects will be accessed their state will be automatically and transparently reloaded from the data source.

Example

```
pmf = (PersistenceManagerFactory)
(Class.forName("com.libelis.lido.PersistenceManagerFactory")
.newInstance());
pmf.setDriverName("versant");
pmf.setConnectionURL(dbName);
pm = pmf.getPersistenceManager();
tx = pm.currentTransaction();

tx.begin();
Provider aProvider = new Provider(“LIBeLIS”);
pm.makerPersistent(aProvider);
Address anAddress = new Address(“25 rue Paul Barruel”,
“France”, “Paris”);
aProvider.address = anAddress ;
tx.commit(); // objects are stored into the data source
// client cache is discarded, references are invalid

tx.begin();
System.out.println(aProvider);
// aProvider is refreshed from the data source
tx.commit();


pm.close();
```

Each time you modify an object it’s entry in the JDO cache is transparently marked as ‘dirty”.

Example

```
tx.begin();
aProvider.address = aNewAdr; // aProvider is marked as dirty
tx.commit(); // aProvider will updated in the data source
```

At the end of a transaction, the PersistentManager will update in the data source all JDO objects marked as modified (“dirtied” objects).

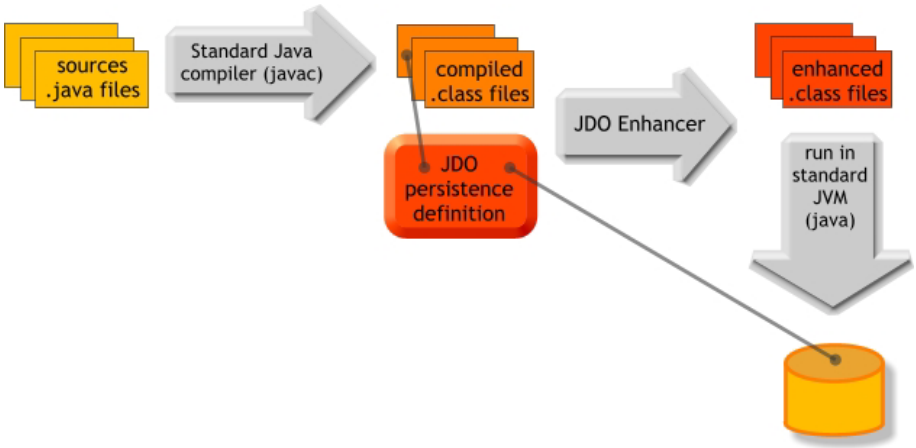
Each object in the JDO cache have a state (in fact they have a reference on one associated StateManager object), and JDO specifies a large set of states and transitions among them. Please directly refers to the JDO specification to see all these states if you are interested.
NB: These states mostly interest JDO vendors, not JDO programmers.

2.3.4 *Development cycle*

In order to achieve the fully transparent persistence described in the previous section JDO defines a byte-code instrumentation mechanism called “Enhancement”.

The idea is to remove any explicit database dependent code from business classes. The mapping with existing or new data sources is then defined using external metadata XML files.

The JDO enhancer takes compiled Java files (.class files) and apply persistence rules as defined in the metadata file, as in the following diagram:



Enhancement will add in the byte-code of classes described in the metadata file:

- declaration of implementing PersistentCapable
- byte-code of the methods declared in that interface and that must be implemented
- code to mark objects as “dirty” each time one of their attribute is modified
- code to fetch objects from data source when necessary (transparent navigation)
- code to map raw data from data source into Java objects according to the mapping specified in the metadata file

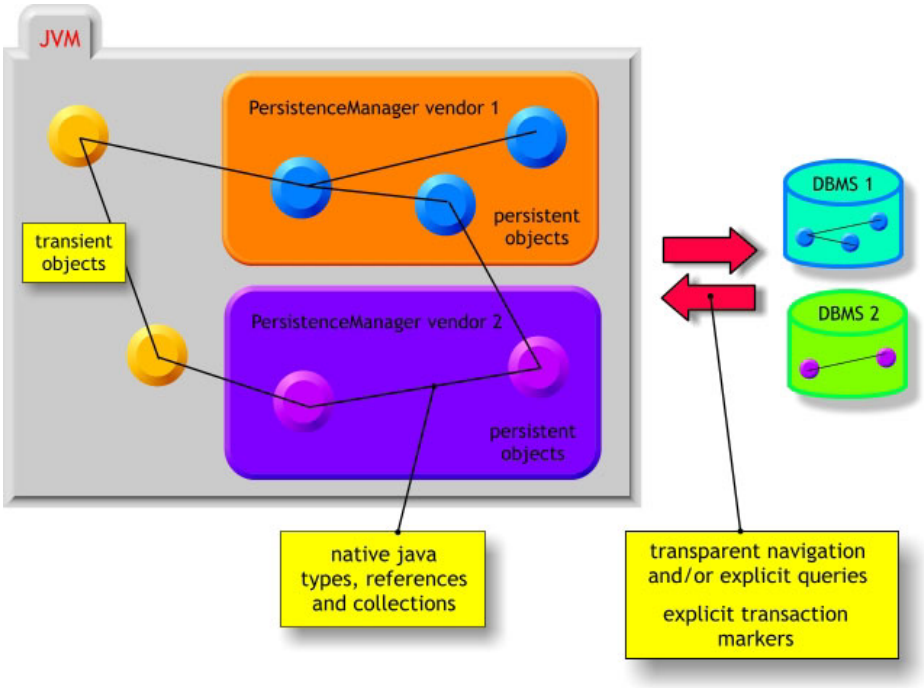
NB: it has been a large and passionate debate within the expert group whether or not enhancement should be part of the JDO specification. Some experts thought that developers could be threatened by enhancement technology.

It’s a fact that developers unfamiliar with this technology might be astonished by byte-code enhancement. But enhancement is a very common and robust development practice, that may be used with benefits in very different cases. At the beginning, developers tend to accuse the enhancer each time they face a bug.

We strongly recommend “newbies” to enhancement to have a close look on the BCEL site (hosted by the Source Forge Open Source community at http://bcel.sourceforge.net). A lot of useful information and tools about byte-code instrumentation might be found there.

Versant among other vendors is using this mechanism for years with success in it’s Java interface. All Versant users can attest of it’s interest.

2.3.5 *Integration in 2-tiers (client/server) architectures*



In traditional client-server architecture, the JDO application must connect itself to one or many data sources using PersistenceManagers, provided by JDO implementations.

Persistent objects can have references on transient ones.
Persistent objects can have references that spread over different PersistenceManagers, but this is not a mandatory feature.

Most of database related source code (like JDBC) is removed from business objects, but transactions boundaries must still be explicitly set by programmers (using traditional begin, commit and rollback methods of the Transaction class).

The PersistentManager‘s role is to manage the mapping between the in-memory Java model and the on-disk data source physical model. This is quite direct when using ODBMS but becomes much more complex when using RDBMS or even simpler data storages.

2.3.6 *JDO Queries*

Goals

- Query language neutral: JDO QL is portable on any data source
- Optimisations are possible for specific query language implementations: SQL, OQL, EJB QL, …
- Multi-tier architecture: entirely in memory, back-end (data-store query engine) execution
- Support for large result sets (cursors, …)
- Compiled query support
- Support queries on references and collections
- Support parameters

Usage

- PersistenceManager is the Query factory
- Query filters Collections and/or Extents and returns Collections
- Required elements in Query
  - Collection of candidate instances
    - May be an Extent (proper or with subclasses)
    - May be a Collection in JVM
  - Class of results
  - Filter (Java boolean expression)
    - Identifiers are in scope of candidate class
    - Numeric promotion for operators

```
filter = "salary > 100000";
filter = "salary > boss.salary";
```

- Query execution

```
Class empClass = Class.forName(« Employee »);
Collection ext = pm.getExtent(empClass);
Query q = pm.newQuery(empClass, ext, filter);
Collection emps = q.execute();
```

- Parameters and variables

```
query.declareParameters ("float sal");
query.declareVariables ("Employee well_comp");
```

- Collection method supported

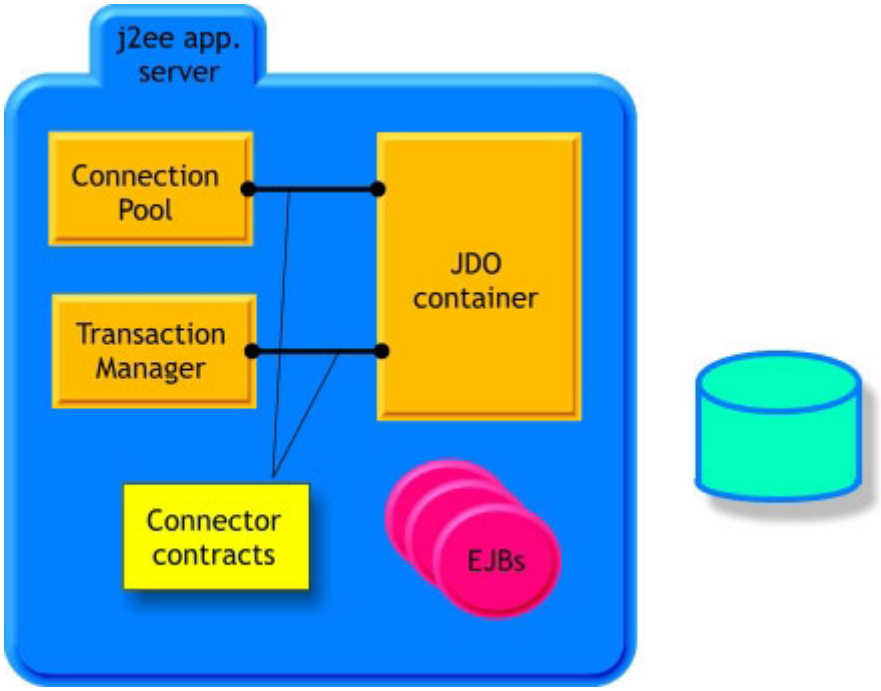
```
contains (Object o)
```

  - Example: Find Departments with at least one well-compensated Employee

```
query.setFilter ("emps.contains (well_comp) &&
                                well_comp.salary > sal");
result = query.execute (new Float (150000));
```


3 JDO and J2EE

JDO has been designed to be integrated in J2EE architectures.
JDO relies on the new Connector specification (JCA) to manage the interactions between a J2EE application server and a JDO container. Within the JDO specification this is described as managed environments (in the sense that transactions and connections are managed by the application server itself).
The JDO container interacts with the J2EE application server to get connections to the data sources (as the application has its own connection pool) and to execute transactions as specified by the application server's JTA compliant Transaction Manager.

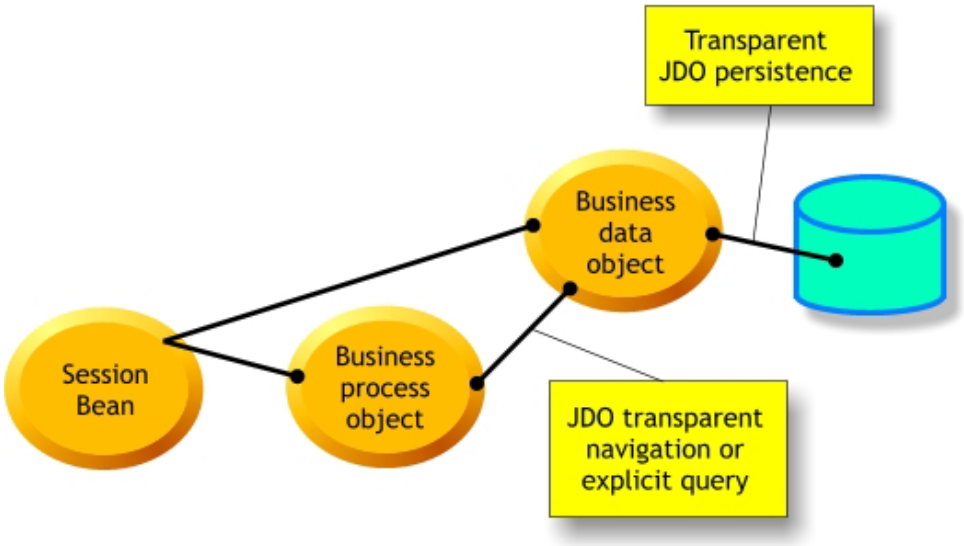


JDO clearly addresses one of the major drawback of the J2EE architecture, which is that the EJB component model closely couples persistence and distribution.
These two concepts are orthogonal, but EJB specification, trying to simplify things, obliges applications to apply the same granularity level for persistence and distribution. This is not clean from a software engineering point of view, and moreover this is not scalable.

Furthermore, the EJB persistent models (CMP and BMP) are too simplistic and can only cope with very limited, not realistic applications. The

interested reader can easily find on J2EE portals large discussion threads about EJB persistence and JDO.

With JDO, you can use a very elegant and generic design where Session Beans, access business process objects, themselves accessing business data objects.



Doing like that, applications still have all the advantages of the J2EE architecture (distribution, transaction, connections, …), and persistence is transparently and efficiently managed through JDO.

JDO can support very complex mapping mechanism, in heterogeneous data sources, while EJB/CMP mode is limited to simple JDBC model. With JDO you have no limitation on the business object model complexity (while EJB/CMP does not support inheritance).

With JDO there is absolutely no database code in your business data object (while with EJB/BMP your business code is infected by JDBC code). Business process objects provide methods that deal with several business data objects. Business process objects are commonly not persistent and they generally get business data object mixing JDO queries and navigation.

It remains important to isolate business process methods from Session Beans, thus allowing your business model to be used in any infrastructure from batch applications to J2EE ones.

Feedback

Feel free to contact should you see any error in this white paper or should you think something is not clear or should be improved.

Also, please participate in the LIBeLIS forums on JDO (www.libelis.com).

LIBeLIS

LIBeLIS
13 rue Camille Desmoulins
92310 Issy les Moulineaux
France

✆+33(0) 158 042 615

sales@libelis.com
support@libelis.com
info@libelis.com

www.libelis.com

About LIBeLIS

LIBeLIS provides solutions for organizations that want to transform their Information Systems using e-Business technologies. We believe that, quite soon, all Information Systems (and not only e-Commerce ones) will be built by assembling distributed components running in *Open Source J2EE* platforms.
LIBeLIS added-value is in highly scalable and adaptable information systems.

LIBeLIS participates in the Java Community Process to define the new standard for object persistence. We believe that JDO will soon become a key component of the Sun J2EE architecture. JDO can be seen as a universal and transparent way to store and access Java objects.

Why JDO is a critical component of J2EE ?
Published by LIBeLIS, january 2001