Java Data Objects (JDO)

Java Data Object Expert Group

Vinay Sampath Kumar

30 Oct, 2003

Introduction

- Standard for *transparent* Java object persistence.
- Developed through the Java Community Process (JCP).
- JDO became a standard in March,2002.
- Designed to allow "pluggable" vendor drivers.

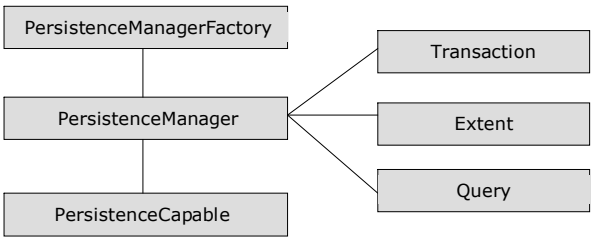
Goals

- Transparent object persistence
  - Minimal constraints on building classes
  - No new data access language
- Use in a range of implementations
  - J2SE (client-server), J2EE (Enterprise Java Beans)
- Data store independence
  - Relational, object, object relational, file system…

Why JDO?

- From an Application developer's perspective:
  - No need to write persistent management code
  - Applications view data and relationships as a class hierarchy
  - Data store independence
    - No vendor lock-in
    - Portability between relational and object data stores
  - Object oriented features are supported
  - No coding using SQL

JDO Classes and Interfaces



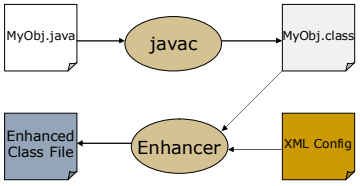
JDO Interfaces and Classes

- PersistenceCapable
  - Interface implemented by User defined persistence classes
  - System-type classes such as Thread, Socket cannot be made persistent
- PersistenceManager
  - Manages PersistenceCapable objects.
    - Identity management
    - Life-cycle management
  - Has Query creation methods
  - Has Transaction creation methods
- PersistenceManagerFactory
  - Creates PersistenceManager instances.

Using JDO

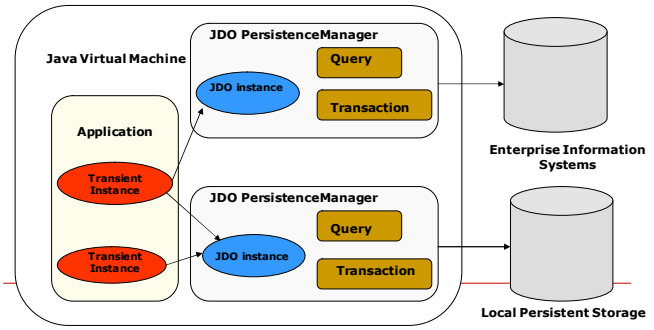
- Write your classes and describe persistence needs in a XML file.
- Use a JDO enhancer to add hooks
- Use PersistentManagerFactory to get a PersistentManager
- Use the PersistentManager to create a Transaction or a Query
- Use Transaction to control transaction boundaries
- Use a Query to find objects

JDO Development Life Cycle



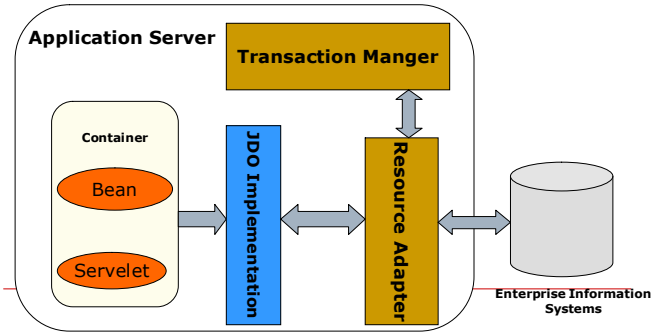
JDO Architecture

- Non-managed JDO architecture–Client/Server
  - Explicit connection and transaction management.



JDO Architecture

- Managed JDO architecture - EJB
  - Implicit connection and transaction management.



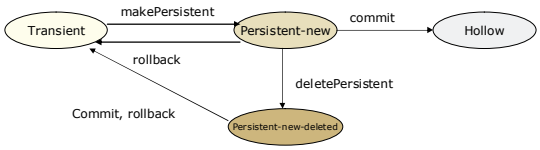
JDO Instances

- Transient
  - Non-transactional - *Unmanaged*
  - Transactional - *Optional*
- Persistent
  - Non-transactional - *Optional*
  - Transactional - *Required*
    - JDO implementation tracks the changes made to the instances.
    - JDO implementation refreshes and saves values to/from the datastore to maintain transactional integrity

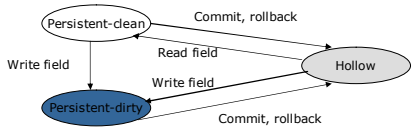
Life Cycle states for JDO instance

- States of *JDO instance*
  - Transient
  - Persistent-new
  - Hollow
  - Persistent-dirty
  - Persistent-clean
  - Persistent-deleted
  - Persistent-new-deleted

Life Cycle of JDO Instances



Life Cycle of New-Persistent Instances



Life Cycle of Datastore Transactions

JDO Identity

- **JDO Identity**
  - Tests whether two in-memory JDO instances represent the same state in the datastore.
- **JDO identifies three types of JDO identity**
  - Application identity (primary key)
    - values in the instance determines the identity in the datastore.
  - Datastore identity
    - managed by the datastore without being tied to any field values.
  - Nondurable identity
    - managed by the JDO implementation to guarantee uniqueness in the JVM
- Type of identity used is fixed at enhancement time.

JDO Persistence Model

- JDO provides transparent data access.
- JDO provides the illusion that the application can access the entire graph of connected instances.
- **Goals**
  - All field type in Java are supported
  - All class and field modifiers are supported.
  - Some system defined classes, those used for modeling state should be persistent capable.

JDO Persistence model

- **First class objects**
  - has a JDO identity.
  - independently stored, queried and deleted from the datastore.
- **Second class objects**
  - stored only as a part of a first class object
  - has no JDO identity
  - some instances do not have a literal datastore representation. Eg- Collection
- A class can be persistence capable independently of the inheritance hierarchy.
- PersistanceCapable classes are indicated in the XML metadata.

Query Facility in JDO

- Application can get access to a JDO instance
  - Constructing a valid ObjectId
  - By iterating over a class extent
  - Using the **JDO Query** interface to acquire a JDO instance based on a search criteria
- **Goals:**
  - Query language neutrality
  - Optimization to specific query languages
  - Accommodation of multi-tier architectures
    - In memory and server side
  - Large result set support
  - Compiled query support

JDO Query Facility

- Query has three required elements
  - class of the candidate instances
  - collection of candidate JDO instances
  - query filter
- **Query execution**
  - Query interface provides methods that execute the query based on the parameters given.
  - Returns an un-modifiable Collection which the user can iterate to get results.

JDO Query Facility

- Filter Specification
  - is a string containing a boolean expression to be evaluated for each of the instance in the candidate collection.
    - boolean expression is expressed using the java language
- Ordering Statement
  - is a string containing one or more ordering declarations followed by 'ascending' or 'descending'

Parameters and Variables

- Support for parameterized queries
  - Parameters are substituted at execution time
  - Parameters are typed
  - Parameters are specified using Java syntax
- Support for use of variables in queries
  - Variables are typed
  - Variables specified using Java syntax

JDO Query Example

```
Class Employee
{
    String name;
    Float salary;
    Department dept;
    Employee boss;
}

Class Department
{
    String name;
    Collection emps;
}
```

- Selects all 'Employee' instances from the candidate collection whose 'salary' is greater than 3000.

```
Class eClass = Employee.class;
Extent cEmp= pm.getExtent(eClass, false);
String fil = "salary > 3000";
Query q = pm.newQuery(eClass, cEmp, fil);
Collection emps = (Collection) q.execute();
```

- The salary comparison value is parameterized.

```
String param = "float sal";
q.declareParameters(param);
Collection emps = (Collection) q.execute(new
                                    Float(30000));
```


Transactions

- Persistent objects always work within the context of a Transaction
- Transaction and PersistentManager – one-to-one relationship
- Methods available in an Unmanaged environment
  - begin()
  - commit()
  - rollback()

Issues to be resolved

- Some of the features to be added in future releases:
  - Nested Transactions
  - API for enhancer invocation
  - API for prefetching
  - Support for BLOB/CLOB data-type
  - Support for projections in queries

JDO vs. JDBC

- JDBC provides an interface for applications to issue SQL statements.
- JDO provides a transparent persistence model for java classes.
- JDBC is a cause for unsafe programming as queries are specified as Strings
- JDO has some amount of type checking built into JDO-QL
- JDBC has no client caching.
- JDO supports caching objects at the clients for scalability
- JDO-QL is not as powerful as SQL.

JDO vs. CMP

- JDO and CMP were developed concurrently
- JDO works in any tier of an enterprise application.
- CMP is container-bound
- JDO development is inexpensive
- CMP development is costly due to conformity to EJB design
- JDO adopts a language transparent approach (byte-code enhancement)
- CMP advocates a functional approach (source-code enhancement)

JDO vs. OPJ

- Very Similar!
- Persistence by reachability
- System classes like Threads are not made persistent.
- In OPJ, all fields of all instances which can be referenced from a root are persistent
- In JDO, one can specify the fields that need to persist.
- OPJ implemented by changing a class loader
- JDO implemented by changing the byte code to add hooks

Conclusion

- JDO – a transparent persistence model
- Similar to OPJ and other orthogonally persistent systems
- Is JDO the *"right"* kind of persistence?
  - No orthogonal persistence but Selective persistence
  - Standardized O/R mapper?
  - JDO-QL – API for subset of SQL
- Is JDO going to replace JBDC ?
- Is JDO going to replace CMP ?