ODMG-api Guide

by Armin Waibel

Table of contents

1. Introduction

The *ODMG API* is an implementation of the ODMG 3.0 Object Persistence API. The ODMG API provides a higher-level API and OQL query language based interface over the PersistenceBroker API.

This document is not a ODMG tutorial (newbies please read the tutorial first) rather than a guide showing the specific usage and possible pitfalls in handling the ODMG-api and the proprietary extensions by OJB.

If you don't find an answer for a specific question, please have a look at the FAQ and the other reference guides.

Additionaly the OJB's ODMG implementation has several extensions described below.

2. Specific Metadata Settings

To make OJB's *ODMG-api* implementation proper work, some specific metadata settings needed in the repository mapping files.

All defined reference-descriptor and collection-descriptor need specific *auto-xxx* settings:

- auto-retrieve="true"
- auto_update="none"
- auto-delete="none" or auto-delete="object" (to enable cascading delete, since OJB 1.0.4!)

> **Note:**
> These settings are mandatory for proper work of the odmg-api!

So an example object mapping class-descriptor look like:

```
<class-descriptor
    class="org.apache.ojb.odmg.shared.Master"
    table="MDTEST_MASTER"
    >
    <field-descriptor
        name="masterId"
        column="MASTERID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="masterText"
        column="MASTER_TEXT"
        jdbc-type="VARCHAR"
        />
    <collection-descriptor
        name="collDetailFKinPK"
        element-class-ref="org.apache.ojb.odmg.shared.DetailFKinPK"
        proxy="false"
        auto-retrieve="true"
        auto-update="none"
        auto-delete="none"
        >
        <inverse-foreignkey field-ref="masterId"/>
    </collection-descriptor>
...
</class-descriptor>
```

A lot of mapping samples can be found in mappings for the OJB test suite. All mappings for the ODMG unit test are in `repository_junit_odmg.xml` file, which can be found under the

src/test directory.

3. How to access ODMG-api

Obtain a `org.odmg.Implementation` instance first, then create a new `org.odmg.Database` instance and open this instance by setting the used jcd-alias name:

```
Implementation odmg = OJB.getInstance();
Database database = odmg.newDatabase();
database.open("jcdAliasName#user#password", Database.OPEN_READ_WRITE);
```

The *user* and *password* separated by *#* hash only needed, when the user/passwd is not specified in the connection metadata (jdbc-connection-descriptor).

The jdbc-connection-descriptor may look like:

```
<jdbc-connection-descriptor
                  jcd-alias="jcdAliasName"
                  ...
                  username="user"
                  password="password"
                  ...
    >
    ...
</jdbc-connection-descriptor>
```

With method call `OJB.getInstance()` always a **new** org.odmg.Implementation instance will be created and `odmg.newDatabase()` returns a new `Database` instance.

For best performance it's recommended to share the `Implementation` instance across the application. To get the current open database from the `Implementation` instance, use method `Implementation.getDatabase(null)`

```
Implementation odmg = ....
// get current used database
Database database = odmg.getDatabase(null);
```

Or share the open `Database` instance as well.

See further in FAQ "Needed to put user/password of database connection in repository file?".

4. Configuration Properties

The OJB *ODMG-api* implementation has some adjustable properties and pluggable components. All configuration properties can be set in the OJB.properties file.

Here are all properties used by OJB's *ODMG-api* implementation:

Property Name	Description
OqlCollectionClass	This entry defines the collection type returned from OQL queries. By default this value is set to a List implementation. This will be suffice in most situations. If you want to use the additional features of the DList interface (DList itself is persistable, support of *predicates*) directly on query results, change setting to the DList implementation (See also property 'DListClass' entry). But this will affect the performance - especially for large result sets. So recommended way is create `DCollection` instances only when needed (e.g. by converting a List result set to a

	DList). Important note: The collection class to be used MUST implement the interface `org.apache.ojb.broker.ManageableCollection`. More info about implementing OJB collection types here.
ImplementationClass	Specifies the used base class for the *ODMG API* implementation. In managed environments a specific class is needed to potentiate JTA integration of OJB's ODMG implementation.
OJBTxManagerClass	Specifies the class for transaction management. In managed environments a specific class is needed to potentiate JTA integration of OJB's ODMG implementation.
ImplicitLocking	This property defines the *implicit locking* behavior. If set to *true* OJB implicitely locks objects to ODMG transactions after performing OQL queries or when do a single lock on an object using `Transaction#lock(...)` method. If implicit locking is used locking objects is recursive, that is associated objects are also locked. If ImplicitLocking is set to *false*, no locks are obtained in OQL queries and there is also no recursive locking when do single lock on an object.
LockAssociations	This property was only used when *ImplicitLocking* is enabled. It defines the behaviour for the OJB implicit locking feature. If set to *true* acquiring a write-lock on a given object x implies write locks on all objects associated to x. If set to *false*, in any case implicit read-locks are acquired. Acquiring a read- or write lock on x thus allways results in implicit read-locks on all associated objects.
Ordering	Enable/Disable OJB's persistent object ordering algorithm on commit of a transaction. If enabled OJB try to calculate a valid order for all new/modified objects (and referenced objects). If the used databases support 'deferred checks' it's recommended to use this feature and to disable OJB's object ordering. **Note:** This setting can be changed at runtime using OJB's ODMG extensions.
ImplicitLockingBackward	A **@deprecated** property only for backward compatibility with older versions (before 1.0.4). If set *true* the behavior of method ImplementationImpl#setImplicitLocking(...) will be the same as in OJB in 1.0.3 or earlier (set the implicit locking behavior of the current used

	transaction) and disable the new possibility of global 'implicit locking' setting at runtime with ImplementationExt#setImplicitLocking. This is only for backward compatibility and will be removed at a later date.
DListClass	The used `org.odmg.DList` implementation class.
DArrayClass	The used `org.odmg.DArray` implementation class.
DMapClass	The used `org.odmg.DMap` implementation class.
DBagClass	The used `org.odmg.DBag` implementation class.
DSetClass	The used `org.odmg.DSet` implementation class.

5. OJB Extensions of ODMG

This section describes the propietary extension of the *ODMG-api* provided by OJB.

5.1. The ImplementationExt Interface

The OJB extension of the odmg Implementation interface is called ImplementationExt and provide additional methods missed in the standard class definition.

- get/setOqlCollectionClass
  Use this methods to change the used OQL query result class at runtime. Description can be found in *Configuration Properties* section and in javadoc of ImplementationExt.
- is/setImpliciteWriteLocks
  Use this methods to global change the associated locking type at runtime when implicit locking is used. Description can be found in *Configuration Properties* section and in javadoc of ImplementationExt.
- is/setOrdering
  Use this methods to global enable/disable OJB's object ordering algorithm. Description can be found in *Configuration Properties* section and in javadoc of ImplementationExt.

5.2. The TransactionExt Interface

The OJB extension of the odmg Transaction interface is called TransactionExt and provide additional methods missed in the standard class definition.

- markDelete
  Description can be found in javadoc of TransactionExt.
- markDirty
  Description can be found in javadoc of TransactionExt.
- flush
  Description can be found in javadoc of TransactionExt.
- is/setImplicitLocking
  Description can be found in javadoc of TransactionExt.
- is/setOrdering
  Description can be found in javadoc of TransactionExt.
- setCascadingDelete
  Description can be found in javadoc of TransactionExt.

- getBroker()
  Returns the current used broker instance. Usage example is here.

5.3. The EnhancedOQLQuery Interface

The OJB extension of the odmg OQLQuery interface is called EnhancedOQLQuery and provide additional methods missed in the standard class definition.

- create(String queryString, int startAtIndex, int endAtIndex)
  Description can be found in javadoc of EnhancedOQLQuery.

5.4. Access the PB-api within ODMG

As the PB-api was used by OJB's *ODMG-api* implementation, thus it is possible to get access of the used `PersistenceBroker` instance using the extended Transaction interface class TransactionExt:

```
Implementation odmg = ...;
TransactionExt tx = (TransactionExt) odmg.newTransaction();
tx.begin();
...
PersistenceBroker broker = tx.getBroker();
// do work with broker
...
tx.commit();
```

It's mandatory that the used *PersistenceBroker* instance **never** be closed with a `PersistenceBroker.close()` call or be committed with `PersistenceBroker.commitTransaction()`, this will be done internally by the ODMG implementation.

6. Notes on Using the ODMG API

6.1. Transactions

The ODMG API uses *object-level transactions*, compared to the PersistenceBroker *database-level transactions*. An ODMG `Transaction` instance contains all of the changes made to the object model within the context of that transaction, and will not commit them to the database until the ODMG `Transaction` is committed. At that point it will use a database transaction (the underlying PB-api) to ensure atomicity of its changes.

6.2. Locks

The ODMG specification includes several levels of locks and isolation. These are explained in much more detail in the Locking documentation.

In the ODMG API, locks obtained on objects are locked within the context of a transaction. Any object modified within the context of a transaction will be stored with the transaction, other changes made to the same object instance by other threads, ignoring the lock state of the object, will also be stored - so take care of locking conventions.
The ODMG locking conventions (obtain a write lock before do any modifications on an object) ensure that an object can only be modified within the transaction.

It's possible to configure OJB's ODMG implementation to support implicit locking with *WRITE* locks. Then a write lock on an object forces OJB to obtain implicit write locks on all referenced objects. See configuration properties.

6.3. Persisting Non-Transactional Objects

Frequently, objects will be modified outside of the context of an ODMG transaction, such as a data access object in a web application. In those cases a persistent object can still be modified, but not directly through the *OMG ODMG specification*. OJB provides an extension to the ODMG specification for instances such as this. Examine this code:

```
public static void persistChanges(Product product)
{
    Implementation impl = OJB.getInstance();
    TransactionExt tx = (TransactionExt) impl.newTransaction();

    tx.begin();
    tx.markDirty(product);
    tx.commit();
}
```

In this function the product is modified outside the context of the transaction, and is then the changes are persisted within a transaction. The `TransactionExt.markDirty()` method indicates to the Transaction that the passed object has been modified, even if the Transaction itself sees no changes to the object.

7. ODMG Named Objects

Using *named objects* allows to persist all serializable objects under a specified name. The methods to handle *named objects* are:

```
/**
 * Associate a name with an object and make it persistent.
 * An object instance may be bound to more than one name.
 * Binding a previously transient object to a name makes that object persistent.
 * @param object The object to be named.
 * @param name The name to be given to the object.
 * @exception org.odmg.ObjectNameNotUniqueException
 * If an attempt is made to bind a name to an object and that name is already
bound
 * to an object.
 */
public void bind(Object object, String name) throws
ObjectNameNotUniqueException;

/**
 * Lookup an object via its name.
 * @param name The name of an object.
 * @return The object with that name.
 * @exception ObjectNameNotFoundException There is no object with the specified
name.
 * @see ObjectNameNotFoundException
 */
public Object lookup(String name) throws ObjectNameNotFoundException;

/**
 * Disassociate a name with an object
 * @param name The name of an object.
 * @exception ObjectNameNotFoundException No object exists in the database with
that name.
 */
public void unbind(String name) throws ObjectNameNotFoundException;
```

To use this feature a internal table and metadata mapping is madatory (by default these settings are enabled in OJB). More information about the needed internal tables see in Platform Guide.

If the object to `bind` is a persistence capable object (the object class is declared in OJB metadata mapping), then the object will be persisted (if needed) dependent on the declared metadata mapping

and the *named object* will be a **link** to the real persisted object.
On `unbind` of the *named object* only the link of the persistent object will be removed, the persistent object itself will be untouched.

If the object to `bind` is a serializable non-persistence cacpable object, the object will be serialized and persisted under the specified name.
On `unbind` the serialized object will be removed.

7.1. Examples

In OJB test-suite is a test case called `org.apache.ojb.odmg.NamedRootsTest` which shown similar examples as below, but more detailed.

1. Persist a serializable object as named object

We want to persist a name list of all planets:

```
Transaction tx = odmg.newTransaction();
tx.begin();
List planets = new ArrayList();
example.add("Mercury");
example.add("Venus");
example.add("Earth");
...
database.bind(planets, "planet-list");
tx.commit();
```

The specified `List` with all planet names will be serialized and persisted as VARBINARY object.

To lookup the persisted list of the solar system planets:

```
Transaction tx = odmg.newTransaction();
tx.begin();
List planets = (List) database.lookup("planet-list");
tx.commit();
```

To remove the persistent list do:

```
Transaction tx = odmg.newTransaction();
tx.begin();
database.unbind("planet-list");
tx.commit();
```

2. Persist a persistence capable object as named object

We want to create a *named object* representing a persistence capable `Article` object (`Article` class is declared in OJB metadata mapping):

```
Transaction tx = odmg.newTransaction();
tx.begin();
// get existing or a new Article object
Article article = ....
database.bind(article, "my-article");
tx.commit();
```

OJB first checks if the specified `Article` object is already persisted - if not it will be persisted. Then based on the `Article` object `Identity` the *named object* will be persisted. So the persistent *named object* is a link to the persistent real `Article` object.

On lookup of the *named object* the real `Article` instance will be returned:

```
Transaction tx = odmg.newTransaction();
tx.begin();
Article article = (Article) database.lookup("my-article");
tx.commit();
```

On unbind of the *named object* only the link to the real `Article` object will be removed, the `Article` itself will not be touched.
To remove the `named object` and the `Article` instance do:

```
tx.begin();
// this only remove the named object link, the Article object
// itself will not be touched
database.unbind("my-article");
// thus delete the object itself too
database.deletePersistent(article);
tx.commit();
```

3. Persist a collection of persistence capable object as named object

We want to persist a list of the last shown `Article` objects. The `Article` class is a persistence capable object (declared in OJB metadata mapping). Thus we don't want to persist a serialized List of `Article` objects (because the real `Article` object may change), as shown in example 1, rather we want to persist a List that links to the real persistent `Article` objects.
This is possible when the ODMG `DCollections` are used:

```
// get the list with last shown Article objects
List lastArticles = ...
Transaction tx = odmg.newTransaction();
tx.begin();
// obtain new DList instance from Implementation class
DList namedArticles = odmg.newDList();
// push Articles to DList
namedArticles.addAll(lastArticles);
database.bind(namedArticles, "last-shown");
tx.commit();
```

In this case OJB first checks for transient `Article` objects and make these new objects persistent, then based on the `Article` object `Identity` the *named object* will be persisted. So the persistent *named object* is in this case a list of links to persistent `Article` objects.

On `database.lookup("last-shown")` the `DList` will be returned and when access the list entries the `Article` objects will be materialized.

To remove the *named object* some more attention is needed:

```
tx.begin();
DList namedArticles = ...
// we want to completely remove the named object
// the persisted DList with all DList entries,
// but the Article objects itself shouldn't be deleted:
// 1. mandatory, clear the list to remove all entries
namedArticles.clear();
// 2. unbind named object
database.unbind("last-shown");
tx.commit();
```

After this the *named object* will be completely removed, but all `Article` object will be untouched.

8. ODMG's DCollections

The ODMG api declare some specific extensions of the `java.util.Collection` interface:

- `org.odmg.DList`
- `org.odmg.DSet`
- `org.odmg.DBag`
- `org.odmg.DMap`

- `org.odmg.DArray`

The ODMG `Implementation` class provide methods to get new instances of these classes.

In OJB all associations between persistence capable classes are declared in the mapping files and *1:n and m:n* relations can use any collection type class which implement the specific interface `ManageableCollection`.
So there is no need to use the ODMG specific collection classes in object relations or when oql-queries are performed (more detailed info see 'oql collection class setting').

One difference to *normal* collection classes is that `DCollection` implementation classes are persistence capable classes itself. This means that they can be persisted - e.g. see named objects example. Mandatory is that all containing objects are persistence capable itself.

When persisting a `DCollection` object OJB first lock the collection entries, then the collection itself was locked. On commit the collection entries will be handled in a *normal* way and for each entry a *link object* (containing the `Identity` of the persistence capable object) is persisted.

When lookup the persisted `DCollection` object the *link objects* are materialized and on access the collection entry will be materialized by the identity.

9. Foreign Keys Constraints and ODMG-api

If cross-referenced database tables are used it's recommended to set *foreign key constraints* to guarantee database consistency. The consequence of using *foreign key constraints* is that the order of the persistence capable objects on *insert* and *delete* operations will become cruical.

Some databases support *deferred constraint checks*, this can help to avoid foreign key issues.

On transaction commit (using standard settings) OJB try to order the objects by itself. If this doesn't suffice it's possible to determine the object order "by hand".

If foreign key constraint violations arise when using 1:1 references and circular/bidirectional 1:1 references it's possible to use a workaround introduced in version 1.0.4 to specify the database FK constraint in OJB using a custom attribute named *'constraint'*:

```
<reference-descriptor name="refAA"
    class-ref="org.apache.ojb.odmg.CircularTest$ObjectAA"
    proxy="false"
    auto-retrieve="true"
    auto-update="none"
    auto-delete="none"
>
    <foreignkey field-ref="fkId"/>
    <attribute attribute-name="constraint" attribute-value="true"/>
</reference-descriptor>
```

10. Questions and Tips

10.1. Disable OJB's object ordering, determine object order "by hand"

By default OJB try to order all persistent objects on transaction commit call to avoid ordering problems. If this is not needed or helpful it can be disabled in two ways.
In most cases it's needed to disable *implicite locking* too, because it will lock/register dependend objects (e.g. 1:n references) automatically. First in OJB.properties file:

```
# Enable/Disable OJB's persistent object ordering algorithm on commit
# of a transaction. If enabled OJB try to calculate a valid order for
# all new/modified objects (and referenced objects).
```

```
# If the used databases support 'deferred checks' it's recommended to use this
# feature and to disable OJB's object ordering.
# This setting can be changed at runtime using OJB's ODMG extensions.
Ordering=false
```

Second at runtime, using OJB's ODMG extension classes ImplementationExt (global setting) and TransactionExt (per tx setting).

```
TransactionExt tx = (TransactionExt) odmg.newTransaction();
tx.begin();
...
/*
we want to manually insert new object, so we disable
OJB's ordering and implicit object locking
*/
tx.setOrdering(false);
tx.setImplicitLocking(false);
...
tx.commit();
```

10.2. Circular- and Bidirectional References

The good news, OJB can handle *bidirectional*- and *circular*- references. When using foreign key constraints for referential integrety in these cases you have to pay attention.

In OJB test-suite a unit test called `org.apache.ojb.odmg.CircularTest` can be found. The tests show the handling of circular/bidirectional references and the possibilities how to handle object insert/update/delete ordering on transaction commit.

10.3. I don't like OQL, can I use the PersistenceBroker Queries within ODMG

Yes you can! The ODMG implementation relies on PB Queries internally! Several users (including myself) are doing this.

If you have a look at the simple example below you will see how OJB Query objects can be used withing ODMG transactions.
The most important thing is to lock all objects returned by a query to the current transaction before starting manipulating these objects.
Further on do not commit or close the obtained PB-instance, this will be done by the ODMG transaction on `tx.commit() / tx.rollback()`.

```
TransactionExt tx = (TransactionExt) odmg.newTransaction();
tx.begin();
....
// cast to get intern used PB instance
PersistenceBroker broker = tx.getBroker();
...
// build query
QueryByCriteria query = ...
// perform PB-query
Collection result = broker.getCollectionByQuery(query);
// use result
...

tx.commit();
...
```

Note: Don't close or commit the used broker instance, this will be done by the odmg-api.

10.4. How to use multiple Databases

For each database define a jdbc-connection-descriptor same way as described in the FAQ.

Now it is possible to

- access the databases one after another, by closing the current used `Database` instance and by open a new one.

```
// get current used database instance
Database database = ...;
// close it
database.close();
// open a new one
database = odmg.newDatabase();
database.open("jcdAliasName#user#password", Database.OPEN_READ_WRITE);
...
```

  The `Database.close()` call close the current used `Database` instance, after this it is possible to open a new database instance.

- use multiple databases in parallel, by creating a separate `Implementation` and `Database` instance for each jdbc-connection-descriptor defined in the mapping metadata.

```
Implementation odmg_1 = OJB.getInstance();
Database database_1 = odmg.newDatabase();
database.open("db_1#user#password", Database.OPEN_READ_WRITE);

Implementation odmg_2 = OJB.getInstance();
Database database_2 = odmg.newDatabase();
database.open("db_2#user#password", Database.OPEN_READ_WRITE);
```

  Now it's possible to use both databases in parallel.

Note:

OJB does not provide distributed transactions by itself. To use distributed transactions, OJB have to be integrated in an j2ee conform environment (or made work with an JTA/JTS implementation).