The ODMG API

by Brian McCallister

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 query language based interface over the PersistenceBroker API.

This tutorial operates on a simple example class:

```
package org.apache.ojb.tutorials;

public class Product
{
/* Instance Properties */

private Double price;
private Integer stock;
private String name;

/* artificial property used as primary key */

private Integer id;

/* Getters and Setters */
...
}
```

The metadata descriptor for mapping this class is described in the mapping tutorial

The source code for this tutorial is available with the source distribution of OJB in the `src/test/org/apache/ojb/tutorials/` directory.

2. Initializing ODMG

The ODMG implementation needs to have a database opened for it to access. This is accomplished via the following code:

```
Implementation odmg = OJB.getInstance();
Database db = odmg.newDatabase();
db.open("default", Database.OPEN_READ_WRITE);

/* ... use the database ... */

db.close();
```

This opens an ODMG `Database` using the name specified in metadata for the database -- "default" in this case. Notice the `Database` is opened in read/write mode. It is possible to open it in read-only or write-only modes as well.

Once a `Database` has been opened it is available for use. Unlike `PersistenceBroker` instances, ODMG `Database` instances are threadsafe and can typically be used for the entire lifecycle of an application. There is no need to call the `Database.close()` method until the database is truly no longer needed.

3. Persisting New Objects

Persisting an object via the ODMG API is handled by writing it to the peristence store within the context of a transaction:

```
public static void storeProduct(Product product)
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();
tx.lock(product, Transaction.WRITE);
tx.commit();
}
```

The `OJB.getInstance()` function provides the ODMG `Implementation` instance required for using the ODMG API. From here on out it is straight ODMG code that should work against any compliant ODMG implementation.

Once the ODMG implementation has been obtained it is used to begin a transaction, obtain a write lock on the `Product`, and commit the transaction. It is very important to note that all changes need to be made within transactions in the ODMG API. When the transaction is committed the changes are made to the database. Until the transaction is committed the database is unaware of any changes -- they exist solely in the object model.

4. Querying Persistent Objects

The ODMG API uses the OQL query language for obtaining references to persistent objects. OQL is very similar to SQL, and using it is very similar to use JDBC. The ODMG implementation is used to create a query, the query is specifed, executed, and a list fo results is returned:

```
public static Product findProductByName(String name) throws Exception
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();

OQLQuery query = impl.newOQLQuery();
query.create("select products from "
             + Product.class.getName()
             + " where name = $1");
query.bind(name);
DList results = (DList) query.execute();
Product product = (Product) results.iterator().next();

tx.commit();
return product;
}
```

5. Updating Persistent Objects

Updating a persistent object is done by modifying it in the context of a transaction, and then committing the transaction:

```
public static void sellProduct(Product product, int number)
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();

tx.lock(product, Transaction.WRITE);
product.setStock(new Integer(product.getStock().intValue() -  number));

tx.commit();
}
```

The sample code obtains a write lock on the object, binding it to the transaction, changes the object, and commits the transaction. The newly modified `Product` now has a new `stock` value.

6. Deleting Persistent Objects

Deleting persistent objects requires directly addressing the `Database` which contains the persistent object. This can be obtained from the ODMG `Implementation` by asking for it. Once retrieved, just ask the `Database` to delete the object. Once again, this is all done in the context of a transaction.

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

tx.begin();
Database db = impl.getDatabase(product);
db.deletePersistent(product);
tx.commit();
}
```

It is important to note that the `Database.deletePerstient()` call does not delete the object itself, just the persistent representation of it. The transient object still exists and can be used however desired -- it is simply no longer persistent.

7. Notes on Using the ODMG API

7.1. 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 to ensure atomicity of its changes.

7.2. Locks

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

In the ODMG API, locks obtained on objects are locked within the context of a thread. Any object modified within the context of a transaction will be stored with the transaction, however changes made to the same object by other threads will also be stored. The ODMG locking conventions ensure that an object can only be modified within the transaction on the locking thread.

7.3. 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.