Object Transaction Manager Tutorial

by Brian McCallister

1. The OTM API

1.1. Introduction

The Object Transaction Manager (OTM) is written as a tool on which to implement other high-level object persistence APIs. It is, however, very usable directly. It supports API's similar to the ODMG and PersistenceBroker API's in OJB. Several of its idioms are designed around the fact that it is meant to have additional, client-oriented, API's built on top of it, however.

The `OTMKit` is the initial access point to the OTM interfaces. The kit provides basic configuration information to the OTM components used in your system. This tutorial will use the `SimpleKit` which will work well under most circumstances for local transaction implementations.

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.

1.2. Persisting New Objects

The starting point for using the OTM directly is to look at making a transient object persistent. This code will use three things, an `OTMKit`, an `OTMConnection`, and a `Transaction`. The connection and transaction objects are obtained from the kit.

Initial access to the OTM client API's is through the `OTMKit` interface. We'll use the `SimpleKit`, an implementation of the `OTMkit` suitable for most circumstances using local transactions.

```
public static void storeProduct(Product product) throws LockingException
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
```

```
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        conn.makePersistent(product);
        tx.commit();
    }
    catch (LockingException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw e;
    }
    finally
    {
        conn.close();
    }
}
```

A kit is obtained and is used to obtain a connection. Connections are against a specific JCD alias. In this case we use the default, but a named datasource could also be used, as configured in the metadata repository. A transaction is obtained from the kit for the specific connection. Because multiple connections can be bound to the same transaction in the OTM, the transaction needs to be acquired from the kit instead of the connection itself. The `SimpleKit` uses the commonly seen transaction-per-thread idiom, but other kits do not need to do this.

Every persistence operation within the OTM needs to be executed within the context of a transaction. The JDBC concept of implicit transactions doesn't exist in the OTM -- transactions must be explicit.

Locks, on the other hand, are implicit in the OTM (though explicit locks are available). The `conn.makePersistent(..)` call obtains a write lock on `product` and will commit (insert) the object when the transaction is committed.

The `LockingException` will be thrown if the object cannot be write-locked in this transaction. As it is a transient object to begin with, this will probably only ever happen if it has been write-locked in another transaction already -- but this depends on the transaction semantics configured in the repository metadata.

Finally, connections maintain resources so it is important to make sure they are closed when no longer needed.

1.3. Deleting Persistent Objects

Deleting a persistent object from the backing store (making a persistent object transient) is almost identical to making it persistent -- the difference is just in the `conn.deletePersistent(product)` call instead of the `conn.makePersistent(product)` call. The same notes about transactions and resources apply here.

```
public static void storeProduct(Product product) throws LockingException
{
   OTMKit kit = SimpleKit.getInstance();
   OTMConnection conn = null;
   Transaction tx = null;
   try
   {
       conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
       tx = kit.getTransaction(conn);
       tx.begin();
       conn.deletePersistent(product);
       tx.commit();
   }
   catch (LockingException e)
   {
       if (tx.isInProgress()) tx.rollback();
       throw e;
   }
   finally
```

```
    {
        conn.close();
    }
}
```

1.4. Querying for Objects

The OTM implements a transaction system, not a new client API. As such it supports two styles of query at present -- an PersistenceBroker like query-by-criteria style querying system, and an ODMG OQL query system.

Information on constructing these types of queries is available in the PersistenceBroker and ODMG tutorials respectively. Using those queries with the OTM is examined here.

A PB style query can be handled as follows:

```
public Iterator findByCriteria(Query query)
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Iterator results = conn.getIteratorByQuery(query);
        tx.commit();
        return results;
    }
    finally
    {
        conn.close();
    }
}
```

Where, by default, a read lock is obtained on the returned objects. If a different lock is required it may be specified specifically:

```
public Iterator findByCriteriaWithLock(Query query, int lock)
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Iterator results = conn.getIteratorByQuery(query, lock);
        tx.commit();
        return results;
    }
    finally
    {
        conn.close();
    }
}
```

The int `lock` argument is one of the integer constants on `org.apache.ojb.otm.lock.LockType`:

```
LockType.NO_LOCK
LockType.READ_LOCK
LockType.WRITE_LOCK
```

OQL queries are also supported, as this somewhat more complex example demonstrates:

```
public Iterator findByOQL(String query, Object[] bindings) throws Exception
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        OQLQuery oql = conn.newOQLQuery();
        oql.create(query);
        for (int i = 0; i < bindings.length; ++i)
        {
            oql.bind(bindings[i]);
        }
        tx.begin();
        Iterator results = conn.getIteratorByOQLQuery(oql);
        tx.commit();
        return results;
    }
    catch (QueryInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Invalid OQL expression given", e);
    }
    catch (QueryParameterCountInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Incorrect number of bindings given", e);
    }
    catch (QueryParameterTypeInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Incorrect type of object given as binding", e);
    }
    finally
    {
        conn.close();
    }
}
```

This function is, at its core, doing the same thing as the PB style queries, except that it constructs the OQL query, which supports binding values in a manner similar to JDBC prepared statements.

The OQL style queries also support specifying the lock level the same way:

```
Iterator results = conn.getIteratorByOQLQuery(query, lock);
```

1.5. More Sophisticated Transaction Handling

These examples are a bit simplistic as they begin and commit their transactions all in one go -- they are only good for retrieving data. More often data will need to be retrieved, used, and committed back.

Only changes to persistent objects made within the bounds of a transaction are persisted. This means that frequently a query will be executed within the bounds of an already established transaction, data will be changed on the objects obtained, and the transaction will then be committed back.

A very convenient way to handle transactions in many applications is to start a transaction and then let any downstream code be executed within the bounds of the transaction automatically. This is straightforward to do with the OTM using the `SimpleKit`! Take a look at a very slightly modified version of the query by criteria function:

```
public Iterator moreRealisticQueryByCriteria(Query query, int lock)
{
     OTMKit kit = SimpleKit.getInstance();
```

```
     OTMConnection conn = null;
     Transaction tx = null;
     try
     {
         conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
         tx = kit.getTransaction(conn);
         boolean auto = ! tx.isInProgress();
         if (auto) tx.begin();
         Iterator results = conn.getIteratorByQuery(query, lock);
         if (auto) tx.commit();
         return results;
     }
     finally
     {
         conn.close();
     }
}
```

In this case the function looks to see if a transaction is already in progress and sets a boolean flag if it is, `auto`. It then handles transactions itself, or allows the already opened transaction to maintain control.

Because connections can be attached to existing transactions the `SimpleKit` can attach the new connection to the already established transaction, allowing this function to work as expected whether there is a transaction in progress or not!

Client code using this function could then open a transaction, query for products, change them, and commit the changes back. For example:

```
public void renameWidgetExample()
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Product sample = new Product();
        sample.setName("Wonder Widget");
        Query query = QueryFactory.newQueryByExample(sample);
        Iterator wonderWidgets
                    = moreRealisticQueryByCriteria(query, LockType.WRITE_LOCK);
        while (wonderWidgets.hasNext())
        {
            Product widget = (Product) wonderWidgets.next();
            widget.setName("Improved Wonder Widget");
        }
        tx.commit();
    }
    finally
    {
        conn.close();
    }
}
```

This sample renames a whole bunch of products from "Wonder Widget" to "Improved Wonder Widget" and stores them back. It must makes the changes within the context of the transaction it obtained for those changes to be stored back to the database. If the same iterator were obtained outside of a transaction, and the changes made, the changes would be made on the objects in memory, but not in the database. You can think of non-transaction objects as free immutable transfer objects.

This example also demonstrates two connections bound to the same transaction, as the `renameWidgetExample(...)` function obtains a connection, and the `moreRealisticQueryByCriteria(...)` function obtains an additional connection to the same transaction!

2. Notes on the Object Transaction Manager

2.1. Transactions

The Object Transaction Manager (OTM) is a transaction management layer for Java objects. It typically maps 1:1 to database transactions behind the scenes, but this is not actually required for the OTM to work correctly.

The OTM supports a wide range of transactional options, delimited in the LockManager documentation. While the lock manager is writte to the ODMG API, the same locking rules apply at the OTM layer.