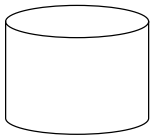
ΗΥ351:
Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
Information Systems Analysis and Design

Persistent Data Management Layer Design (III) and Java-related technologies

Γιάννης Τζίτζικας

Διάλεξη/Φροντιστήριο: 16



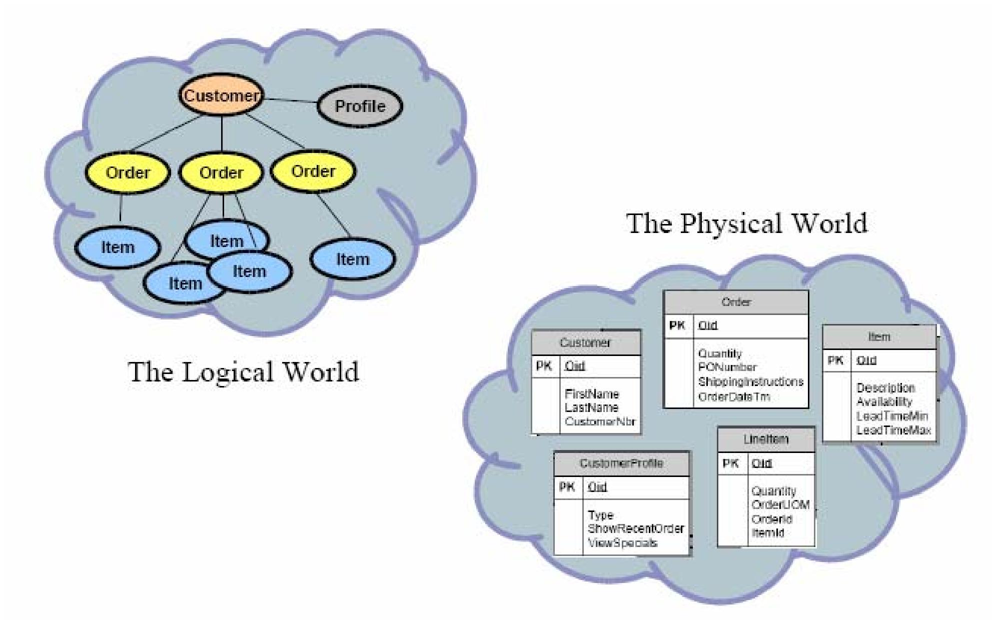
JDO

Η ενότητα αυτή περιλαμβάνει παρουσίαση τεχνολογιών που μπορείτε να χρησιμοποιήσετε στην 3η φάση της εργασίας (project) του μαθήματος.
Η ύλη όμως δεν θα εξεταστεί.

The Landscape

- Persistence
  - Saving your (persistent) application data
  - Mapping your component/object model to the persistence store, typically referred to as O/R (Object relational) mapping
- Data consistency and concurrent access
- Transactional semantics
- Managing your persistent state is non-trivial and complex

Object-Relational Impedance Mismatch



Java Data Object (JDO)

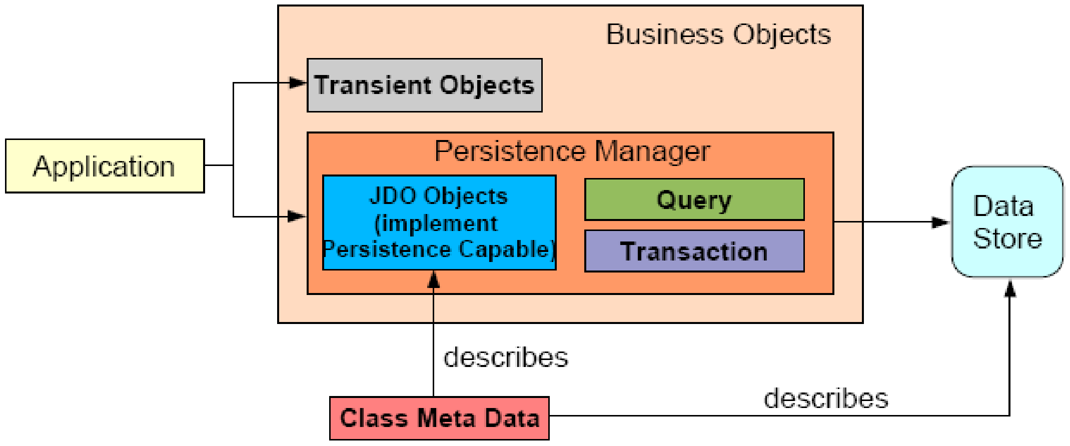
What is **JDO** (Java Data Object) ?

- Standard for generic/transparent Java object persistence
- Provides developers with a Java-centric and object view of persistence and data store access
- Designed to allow pluggable vendor “drivers” for accessing any database/data store
- *Connector Architecture* used to specify the contract between JDO Vendor and Application Server for instance, connection, and transaction management

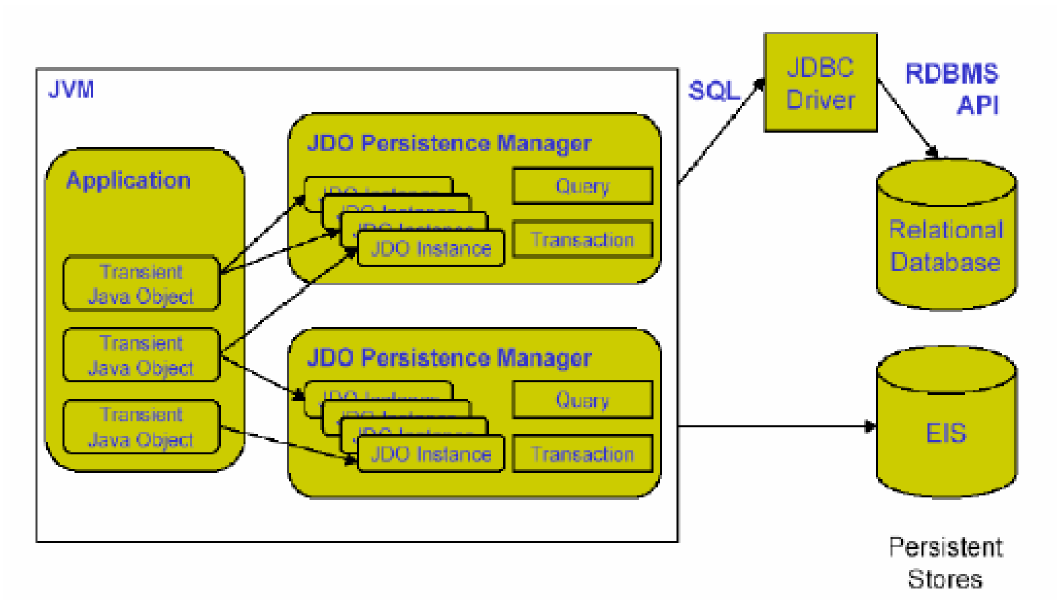
JDO Architecture

Persistence By Reachability

- Any object loaded directly or indirectly (by reference) from a JDO loaded object is automatically persisted if the enclosing transaction commits

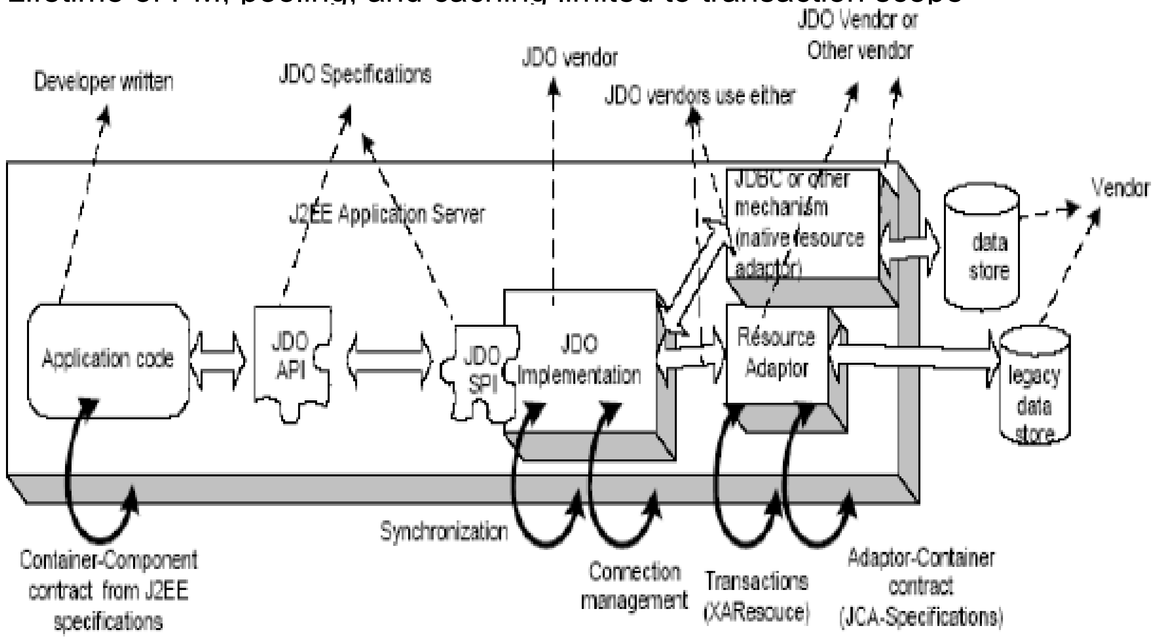


JDO Non-Managed Runtime



Managed Environment

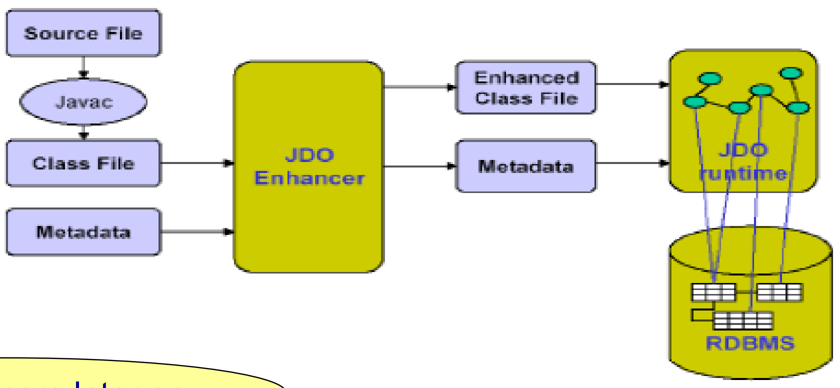
- J2EE-based, multi-tier
  - Lifetime of PM, pooling, and caching limited to transaction scope



Byte Code Enhancement

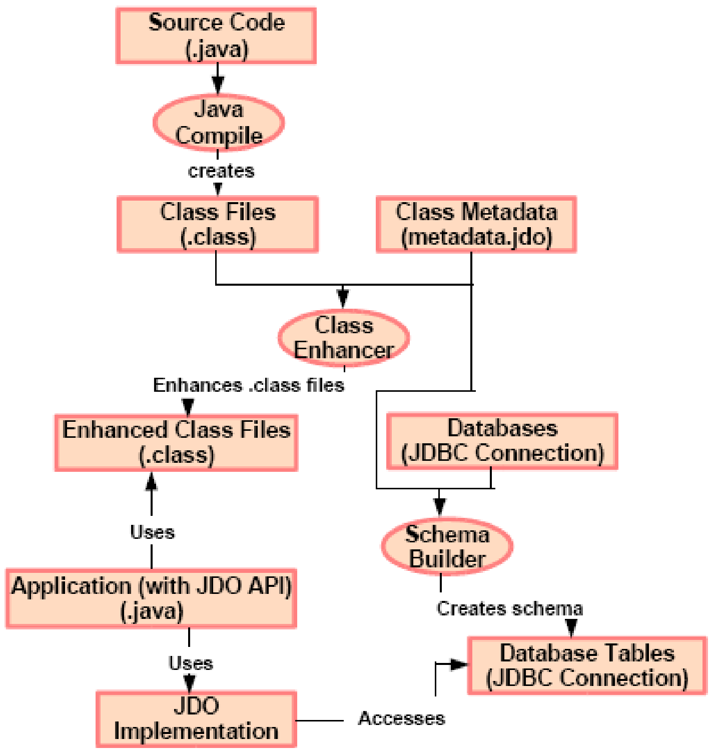
Most JDO vendors use the **bytecode modification** for the following reasons:

- Avoid potentially messy source code modification
- Allow persistence to be hidden from the programmer. The programmer is database unaware



we will see more later on

JDO Deployment Process



Working with JDO Interfaces and Classes

A typical way to use JDO is the following:

- *Use **PersistenceManagerFactory** to get a **PersistenceManager***
  - *PersistenceManager embodies a database connection*
- *Use a **PersistenceManager** to create a **Transaction** or a **Query***
- *Use a **Transaction** to control transaction boundaries*
- *Use a **Query** to find objects*
- *Enhanced classes implicitly implement **PersistenceCapable***
- *PersistenceCapable classes can implement InstanceCallbacks*

JDO API for Persistence

```
public static void main(String[] args) {
    PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(System.getProperties());
    PersistenceManager pm = pmf.getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    tx.begin();
        Author author = new Author(“Mr. Author");
        Book book = new Book("Java Book", "0-11-570731-7");
        author.addBook(book);
        // do some other work with books, publishers etc
        pm.makePersistent(author);
    tx.commit();
    pm.close();
}
```

JPOX
JAVA PERSISTENT OBJECTS

JPOX: an implementation of JDO
http://www.jpox.org/docs/jdo/jdo_overview.html

JPOX: A reference implementation of JDO

- Recall what JDO is:
  - JDO defines an interface (or API) to persist normal Java objects (or POJO's in some peoples terminology) to a datastore. JDO doesn't define the type of datastore. It is datastore-agnostic. You would use the same interface to persist your Java object to RDBMS, or OODBMS, or XML, or whatever form of data storage. JDO is a *standard*. JDO 1.0 has been in existence since 2002, whilst JDO 2.0 was approved in early 2005. JDO defines the interface that an implementation has to implement.
- JPOX is *an implementation* of the JDO interface specification (JDO 1.0, JDO 2.0, JDO 2.1).
  - There are also other implementations. The whole point of having a *standard* interface is that users can, in principle, swap between implementations of JDO without changing their code.

JDO categorises classes into **3 types**

The type of your class defines how it interacts with JDO. Some classes have no interaction with JDO, whilst others require you to define their behaviour under JDO.

- **Persistence Capable** classes
  - are classes whose instances can be persisted to a datastore. JDO provide the mechanism for persisting these instances, and they are core to JDO. These classes need to be *enhanced* according to a JDO Meta-Data specification before use within a JDO environment.
- **Persistence Aware** classes
  - are classes that manipulate Persistence Capable instances through direct attribute manipulation. These classes are typically enhanced with very minimal JDO Meta-Data. The enhancement process performs very little changes to these classes.
- **Normal** classes
  - are classes that aren't themselves persistable, and have no knowledge of persistence either. These classes are totally unchanged in JDO, and require no JDO Meta-Data whatsoever.

Defining **PersistenceCapable** classes

Classes are defined as **PersistenceCapable** either by XML MetaData, like this

```
<class name="MyClass">
...
</class>
```

or, in JDO2.1, using Annotations. Like this :

```
@PersistenceCapable
public class MyClass
{ ...
}
```

Defining **PersistenceAware** classes

Classes are defined as **PersistenceAware** either by XML MetaData, like this

```
<class name="MyClass“ persistence-modifier="persistence-aware"/> >
...
</class>
```

or, in JDO2.1, using Annotations. Like this :

```
@PersistenceAware
public class MyClass
{ ...
}
```

Controlling the persistence of your objects.

- This is performed using a PersistenceManagerFactory/ PersistenceManager. The persistence of Java objects results in changes to the lifecycle state of the objects.

Persistence Manager Factory

- Any JDO-enabled application will require at least one *PersistenceManagerFactory*. Typically applications create one per datastore being utilised. A *PersistenceManagerFactory* provides access to *PersistenceManager*s which allow objects to be persisted, and retrieved. The *PersistenceManagerFactory* can be configured to provide particular behaviour.
- The simplest way of creating a *PersistenceManagerFactory* is as follows

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass", "{my_implementation_pmf_class}");
properties.setProperty("javax.jdo.option.ConnectionDriverName","com.mysql.jdbc.Driver");
properties.setProperty("javax.jdo.option.ConnectionURL","jdbc:mysql://localhost/myDB");
properties.setProperty("javax.jdo.option.ConnectionUserName","login");
properties.setProperty("javax.jdo.option.ConnectionPassword","password");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

Persistence Manager Factory (cont)

- A slight variation on this, is to use a file ("jdo.properties" for example) to specify these properties like this

```
javax.jdo.PersistenceManagerFactoryClass={my_implementation_pmf_class}
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/myDB
javax.jdo.option.ConnectionUserName=login
javax.jdo.option.ConnectionPassword=password
```

and then to create the *PersistenceManagerFactory* using this file

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("jdo.properties");
```

A final alternative would be to call *JDOHelper.getPersistenceManagerFactory(jndiLocation, context);*, hence accessing the properties via JNDI. Whichever way we wish to obtain the *PersistenceManagerFactory*

Regarding the Properties of *PersistenceManagerFactory*, the first property specifies to use PMF of the implementation required to be used, and the following 4 properties define the datastore that it should connect to.

For more about the properties see http://db.apache.org/jdo/pmf.html

Persistence Manager

- Any JDO-enabled application will require at least one *PersistenceManager* (PM). This is obtained from the PersistenceManagerFactory for the datastore. The simplest way of creating a *PersistenceManager* is as follows

```
PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(props);
PersistenceManager pm = pmf.getPersistenceManager();
```

A *PersistenceManager* is the key to all persistence operations in JDO. With it you can *persist*, *update*, *delete*, and *retrieve* objects from the datastore.

A *PersistenceManager* has a single transaction.

How to **persist** objects

- To persist an object, the object must first be marked as persistable using MetaData (XML/Annotations). Then you would start the PM transaction, and use *makePersistent* as follows

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try{
   tx.begin(); // starts the transaction
  MyClass obj = new MyClass(); // create the object to persist
   pm.makePersistent(obj); // persist is to the datastore
   tx.commit(); //  Commit the transaction, flushing the object to the datastore
} catch (Exception e) {
   ... handle exceptions
}
finally {
   if (tx.isActive())    {
      // Error occurred so rollback the transaction
      tx.rollback();
   }
   pm.close();
}
```

How to **persist** objects (cont)

- The *makePersistent* method of **PersistenceManager** makes the object persistent in the datastore, and updates the 'state' of the object from *Transient* (at the start) to *Hollow* (after commit() of the transaction).
- When an object is persisted, if it has any other objects referenced from that object they also will be made persistent. This is referred to as **persistence-by-reachability**. The main benefit of this is that if you have an object graph to persist, then you don't need to call *makePersistent()* on all objects, instead just using one that can be used to find all of the others.
- **persistence-by-reachability** is also run at the time of calling *commit()* on the transaction. This has the effect that if you had called *makePersistent()* on an object and that had persisted another object, and before commit you had removed the relation to this other object, then at *commit()* the reachability algorithm will find that this other object is no longer reachable and will remove it from persistence.

How to **retrieve** objects

- So we've made some of our objects persistent, and now we want to retrieve them in our application. Here's one way of retrieving objects of a particular type.

```
tx = pm.currentTransaction();
try {
    tx.begin();
    Extent e = pm.getExtent(mydomain.MyClass.class, true);
    Iterator iter=e.iterator();
    while (iter.hasNext())    {
        MyClass my_obj=(MyClass)iter.next();
        ...
    }
    tx.commit();
} catch (Exception e){
    if (tx.isActive())   {
        tx.rollback();
    }
}
```

The **Extent** interface is one of the ways to retrieve your objects. The others use the **Query** interface, allowing more precise filtering over the objects returned.

How to **update** objects

- To update an object we firstly retrieve it, as above, and then we call any of its mutator methods. For example

```
tx = pm.currentTransaction();
try {
    tx.begin();
    Extent e = pm.getExtent(mydomain.MyClass.class, true);
    Iterator iter=e.iterator();
    while (iter.hasNext())  {
        MyClass my_obj=(MyClass)iter.next();
        my_obj.setValue(25.0); // Change the value
        ...
    }
    tx.commit();
} catch (Exception e) {
    if (tx.isActive())   {
        tx.rollback();
    }
}
```

When *setValue()* is called on the persistent object this change is intercepted by JDO and the value change will be automatically sent to the datastore ... transparently!

How to **delete** objects

- So we can persist objects, and retrieve them. Now we want to remove one from persistence.

```
try{
    tx = pm.currentTransaction();
    tx.begin();
    ... (code to retrieve object in question) ...
    pm.deletePersistent(my_obj);
    tx.commit();
} catch (Exception e) {
    if (tx.isActive())  {
        tx.rollback();
    }
}
```

How to make an object **transient**

- As we have seen in the JDO States guide, an object can have many possible states. When we want to take an object and work on it, but removing its identity we can make it **transient**. This means that it will retain the values of its fields, yet will no longer be associated with the object in the datastore. We do this as follows

```
try {
    tx = pm.currentTransaction();
    tx.begin();
    ... (code to retrieve object in question) ...
    pm.makeTransient(my_obj);
    tx.commit();
}
catch (Exception e){
    if (tx.isActive())   {
        tx.rollback();
    }
}
... (code to work on "my_obj")
```

How to **query** your objects to the datastore

- Once you have persisted objects you need to query them.
  - For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products.
- The JDO specifications (ver 1.0.1 and ver 2.0) require that implementations provide a Query capability using its own query language (**JDOQL**).
  - JDOQL is oriented around the objects that are persisted, and provides an interface for selecting these objects within the framework of a query.
  - JPOX provides such a JDOQL Query mechanism. The JDO 2.0 specification requires that implementations provide a **SQL** query mechanism (for datastores that support SQL). JPOX provides this. In addition, JPOX provides a query language that is positioned between JDOQL and SQL that has SQL-like syntax yet allows access to the field names of classes - **JPOXSQL** .
- Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer **JDOQL**. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of **SQL**.
  - This is the power of an implementation like JPOX in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

How to **query** your objects to the datastore

- Let's now try to understand the Query interface in JDO , We firstly need to look at a typical Query

```
Query query = pm.newQuery("javax.jdo.query.JDOQL",
                "SELECT FROM org.jpox.MyClass WHERE param2 < threshold");
query.declareImports("import java.util.Date");
query.declareParameters("Date threshold");
query.setOrdering("param1 ascending");
List results = (List)query.execute(my_threshold);
```

In this Query, we select our query language (**JDOQL** in this case), and the query is specified to return all objects of type *org.jpox.MyClass* (or subclasses) which have the field *param2* less than some threshold value. We've specified the query like this because we want to pass the threshold value in dynamically. We then import the type of our *threshold* parameter, and the parameter itself, and set the *ordering* of the results from the Query to be in ascending order of some field *param1*. The Query is then executed, passing in the threshold value. The example is to highlight the typical methods specified for a Query. Clearly you may only specify the Query line if you wanted something very simple. The result of the Query is cast to a List since in this case it returns a List of results.

Query Language differences

JPOX provides 3 query languages for the user.

- The most portable, provided across all datastores, is **JDOQL**.
- The is another one using the RDBMS query language **SQL**.
- In addition JPOX provides its own extension to **SQL** called **JPOXSQL**. The latter is non-portable across JDO implementations, and so by using it you would be reducing the portability of your JDO application.

The 3 languages have clear differences in their syntax, but also in the application of the various methods on the Query. This table attempts to highlight the differences.

- For more details see: http://www.jpox.org/docs/1_1/query.html

Named Queries

- The query described above is constructed dynamically. Queries of that form are perfect for situations like a web system where the user selects something and you want to present them with particular information based on their selections. There do however exist other types of situation where you know a particular query will be needed. In this case **it isn't desirable to have to construct it at runtime**. This functionality is added in the JDO 2.0 specification, <u>allowing the user to specify queries in the JDO Meta-Data</u>.
- To highlight how to do this, lets say we have a class called *Product* (something to sell in a store). We define the JDO Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

Named Queries (cont)

```
<jdo>
   <package name="org.jpox.example">
      <class name="Product">
         ...
         <query name="SoldOut" language="javax.jdo.query.JDOQL">
        <![CDATA[
         SELECT FROM org.jpox.example.Product WHERE status == "Sold Out"
         ]]></query>
      </class>
   </package>
</jdo>
```

- So we have a query called "SoldOut" defined for the class *org.jpox.example.Product* that returns all Product (and subclasses) that have a *status* of "Sold Out". So in our application all we need to do now is

```
Query query = pm.newNamedQuery(org.jpox.example.Product.class,"SoldOut");
List results = (List)query.execute();
```

Note that this syntax is based around the single-string form of JDOQL and applies to JPOX 1.1.0-beta-1 onwards. You now have the means to use the 2 principal types of queries in JDO.

Result Class

- When you perform a query, using JDOQL or SQL the query will, in general, return a List of objects. These objects are by default of the same type as the candidate class. This is good for the majority of situations but there are some situations where you would like to control the output object. This can be achieved by specifying the *Result Class*.

```
query.setResultClass(myResultClass);
```

The *Result Class* has to meet certain requirements. These are

•Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]

•Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

Result Class (cont)

Where you have a query returning a single field, you could specify the *Result Class* to be one of the first group for example. Where your query returns multiple fields then you can set the *Result Class* to be your own class. So we could have a query like this

```
Query query = pm.newQuery(pm.getExtent(org.jpox.samples.Payment.class,false));
query.setFilter("amount > 10.0");
query.setResultClass(Price.class);
query.setResult("amount, currency");
List results = (List)query.execute();
```

and we define our *Result Class* **Price** as follows

```
public class Price {
    protected double amount = 0.0;
    protected String currency = null;
    public Price(double amount, String currency)  {
        this.amount = amount;
        this.currency = currency;
 }    ...
}
```

In this case our query is returning 2 fields (a Double and a String), and these map onto the constructor arguments, so JPOX will create objects of the **Price** class using that constructor. We could have provided a class with public fields instead, or provided *setXXX* methods or a *put* method. They all work in the same way.

Result Fetching

When a Query is executed it executes SQL in the datastore, which returns a ResultSet. JPOX could clearly read all results from this ResultSet in one go and return them all to the user, or could allow control over this fetching process. JDO2 provides a *fetch size* on the Fetch Plan to allow this control. You would set this as follows

```
Query q = pm.newQuery(...);
q.getFetchPlan().setFetchSize(FetchPlan.FETCH_SIZE_OPTIMAL)
```

fetch size has 3 possible values.

•**FETCH_SIZE_OPTIMAL** - allows JPOX full control over the fetching. In this case JPOX will fetch each object when they are requested, and then when the owning transaction is committed will retrieve all remaining rows (so that the Query is still usable after the close of the transaction).

•**FETCH_SIZE_GREEDY** - JPOX will read all objects in at query execution. This can be efficient for queries with few results, and very inefficient for queries returning large result sets.

•**A positive value** - JPOX will read this number of objects at query execution. Thereafter it will read the objects when requested.

Result Fetching (cont)

In addition to the number of objects fetched, you can also control which fields are fetched for each object of the candidate type. This is controlled via the *FetchPlan*. See also Fetch Groups.

- JPOX also allows an extension to give further control. As mentioned above, when the transaction containing the Query is committed, all remaining results are read so that they can then be accessed later (meaning that the query is still usable). Where you have a large result set and you don't want this behaviour you can turn it off by specifying a Query extension

```
q.addExtension("org.jpox.query.loadResultsAtCommit", "false");
```

- so when the transaction is committed, no more results will be available from the query.

Query Timeouts

- JPOX provides a useful extension to JDO queries by allowing control over the timeout of the query. So, for example, if you have a query that can cause problems in terms of the time taken, you can set a timeout on the query to retain the usability of your application.
- A JDO2 standard way of doing this (from 1.1.0-beta-6) for each query is to do

```
query.addExtension("org.jpox.query.timeout","20");
```

- The value passed in is in seconds. You can also specify this for all queries using a PMF property "org.jpox.query.timeout".

Result Set Control

- JPOX provides a useful extension to JDO by allowing control over the ResultSet's that are created by queries. You have at your convenience 4 properties that give you the power to control whether the result set is read only, whether it can be read forward only, the number of rows to be fetched etc. You can specify these on a per-Query basis (from version 1.1.0-beta-6) as follows

```
query.addExtension("org.jpox.query.fetchSize", "20");
query.addExtension("org.jpox.query.fetchDirection", "forward");
query.addExtension("org.jpox.query.resultSetType", "scroll-insensitive");
query.addExtension("org.jpox.query.resultSetConcurrency", "read-only");
```

Alternatively you can specify these on the PersistenceManagerFactory so that they apply to all queries for that PMF. Again, the properties are

•**org.jpox.query.fetchSize** - controls the number of records fetched from the datastore each time more are required.

•**org.jpox.query.fetchDirection** - controls the direction that the ResultSet is navigated. By default this is forwards only. Use this property to change that.

•**org.jpox.query.resultSetType** - controls the type of ResultSet.

•**org.jpox.query.resultSetConcurrency** - controls whether the ResultSet is read only or updateable.

Bear in mind that not all JDBC drivers support all of the possible values for these options. That said, they do add a degree of control that is often useful.

Transactions

- When managing the persistence of objects using a PersistenceManager it is normal to handle all datastore operations in a transaction. For this reason each *PersistenceManager* has its own transaction. Consequently a typical JDO persistence method will look something like this

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
   tx.begin(); // Start the PM transaction
   ... perform some persistence operations
   tx.commit(); // Commit the PM transaction
}
finally {
   if (tx.isActive())   {
      tx.rollback(); // Error occurred so rollback the PM transaction
 }}
```

Transactions (cont)

JDO supports the two main forms of transaction

- Transactions can **lock all** records in a datastore and keep them locked until they are ready to commit their changes. These are known as **Pessimistic** (or datastore) Transactions
- Transactions can simply assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. These are known as **Optimistic** Transactions.

You select the type of transaction to be used by a *PersistenceManager* (PM) either by setting the PMF property **javax.jdo.option.Optimistic**, or on the transaction you call

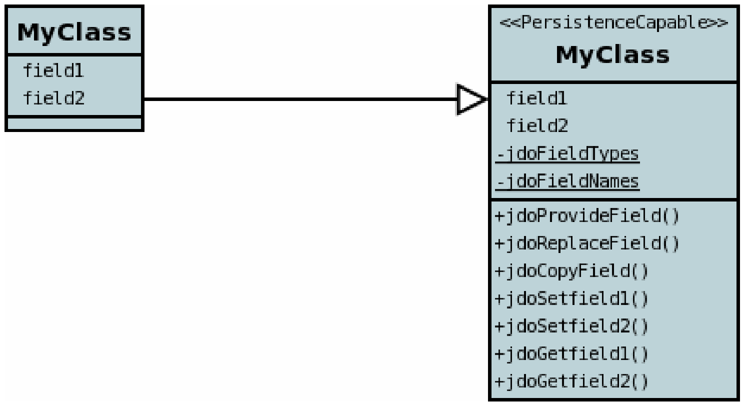
```
pm.currentTransaction().setOptimistic(true);
```

Some internals of JPOX: JPOX bytecode enhancer

- JDO requires that all classes to be persisted must implement the *PersistenceCapable* interface . Users could manually do this themselves but this would impose work on them. JDO permits the use of a **byte-code enhancer** that converts the users normal classes to implement this interface. JPOX provides its own byte-code enhancer (this can be found in the *jpox-enhancer.jar*). The enhancement process adds the necessary methods to the users class in order to implement *PersistenceCapable*.

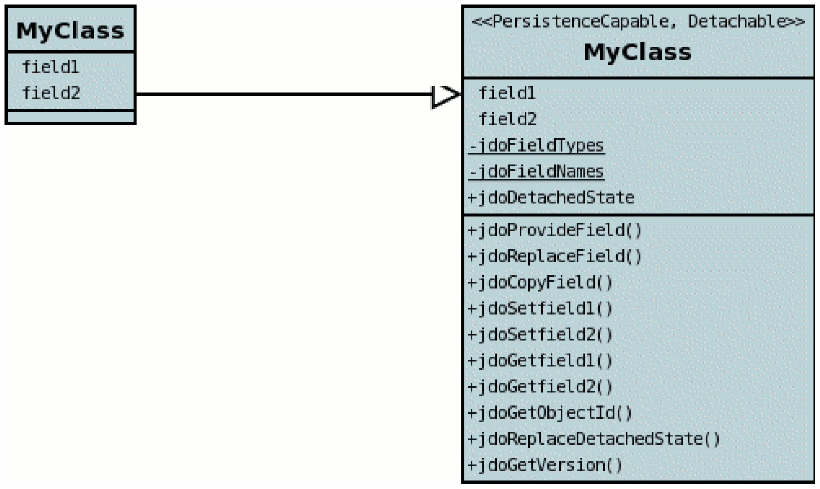
JPOX bytecode enhancer (cont)

- This example doesn't show all *PersistenceCapable* methods, but demonstrates that all added methods and fields are prefixed with "jdo" to distinguish them from the users own methods and fields. Also each persistent field of the class will be given a jdoGetXXX, jdoSetXXX method so that accesses of these fields are intercepted so that JDO can manage their "dirty" state.



JPOX bytecode enhancer (cont)

- The MetaData defines which classes are required to be persisted, and also defines which aspects of persistence each class requires. For example if a class has the *detachable* attribute set to *true*, then that class will be enhanced to also implement *Detachable*



(for more info about JDO and JPOX)

For more about JDO

- Apache JDO (http://db.apache.org/jdo) is the project controlling the direction of the JDO standard and, as such, is the place to go for information specific to the standard API.
- You could also download a Free JDO1 book (http://www.orientechnologies.com/docs/JavaDataObjects-RobinRoos-1.0.pdf) and do some reading.

For more about JPOX

- See the tutorial http://www.jpox.org/docs/1_1/tutorials/tutorial.html
- Eclipse Plugin
  - http://www.jpox.org/docs/1_1/tutorials/eclipse.html