Frequently Asked Questions

Questions

1. **General**
   - Why OJB? Why do we need another O/R mapping tool?
   - How is OJB related to ODMG and JDO?
   - What are the OJB design principals?
   - Where can I learn more about Object/Relational mapping in general?
   - How OJB performance compares to native JDBC programming?
   - How OJB performance compares to other O/R mapping tools?
   - Is OJB ready for production environments?
2. **Getting Started**
   - Help! I'm having problems installing and using OJB!
   - Help! I still have serious problems installing OJB!
   - OJB does not start?
   - Does OJB support my RDBMS?
   - What are the OJB internal tables for?
   - What does the exception *Could not borrow connection from pool* mean?
   - Any tools help to generate the metadata files?
3. **OJB api's**
   - What are the differences between the PersistenceBroker API and the ODMG API? Which one should I use in my applications?
   - I don't like OQL, can I use the PersistenceBroker Queries within ODMG?
   - The OJB JDO implementation is not finished, how can I start using OJB?
4. **Howto**
   - How to use OJB with my RDBMS?
   - What are the best settings for maximal performance?
   - How to page and sort?
   - What about performance and memory usage if thousands of objects matching a query are returned as a Collection?
   - When is it helpful to use Proxy Classes?
   - How can I convert data between RDBMS and OJB?
   - How can I trace and/or profile SQL statements executed by OJB?
   - How does OJB manage foreign keys?
   - How does OJB manage 'null' for primitive primary key?
   - How to lookup object by primary key?
   - Difference between getIteratorByQuery() and getCollectionByQuery()?
   - How can Collections of primitive typed elements be mapped?
   - How could class 'myClass' represent a collection of 'myClass' objects
   - How to lookup `PersistenceBroker` instances?
   - How to access ODMG?
   - Needed to put user/password of database connection in repository file?
   - Many different database user - How do they login?
   - How do I use multiple databases within OJB?
   - How does OJB handle connection pooling?
   - Can I directly obtain a `java.sql.Connection` within OJB?
   - Is it possible to perform my own sql-queries in OJB?

- Start OJB without a repository file?
- Connect to database at runtime?
- Add new persistent objects metadata (`class-descriptor`) at runtime?
- Global metadata changes at runtime?
- Per thread metadata changes at runtime?
- Is it possible to use OJB within EJB's?
- Can OJB handle ternary (or higher) associations?
- How to map a list of Strings
- How to set up Optimistic Locking
- How to use OJB in a cluster
- How to work with the ObjectCacheEmptyImpl
- JDO - Why must my persisten class implement `javax.jdo.spi.PersistenceCapable`?

Answers

1. General

1.1. Why OJB? Why do we need another O/R mapping tool?

here are some outstanding OJB features:

- It's fully ODMG 3.0 compliant
- It will have a full JDO implementation
- It's higly scalable (Loadbalanced Multiserver scenario)
- It provides multiple APIs:
  - The full fledged ODMG-API,
  - The JDO API (planned)
  - and the PersistenceBroker API. This API provides a O/R persistence kernel which can be used to build higher level APIs (like the ODMG and JDO Implementations)
- It's able to handle multiple RDBMS simultaneously.
- it has a slick MetaLevel Architecture: By changing the MetaData at runtime you can change the O/R mapping behaviour. (E.G. turning on/off usage of Proxies.)
- It has a simple CacheMechanisms that is fully garbage collectable by usage of weak references.
- It has a simple and clean pattern based design.
- It uses a configurable plugin concept. This allows to replace components (e.g. the ObjectCache) by user defined Replacements.
- It has a modular architecture (you can quite easily reuse some components in your own applications if you don't want to use the whole thing:
  - The PersistenceBroker (e.g. to build your own PersistenceManager)
  - The Query Interface as an abstract query syntax
  - The OQL Parser
  - The MetaData Layer
  - The JDBC Accesslayer
- It has a very sharp focus: It's concerned with O/R mapping and nothing else.

Before making OJB an OpenSource project I had a look around at the emerging OpenSource O/R scene and was asking myself if there is really a need for yet another O/R tool. I came to the conclusion that there was a need for OJB because:

- There was no ODMG/JDO compliant opensource tool available
- There was no scalable opensource O/R tool available
- there was no tool available with the idea of a PersistenceBroker Kernel that could be easiliy extended
- The tools available had no dynamic MetaData architectures.
- The tools available were not as clearly designed as I hoped, thus extending one of them would have been very difficult.

1.2. How is OJB related to ODMG and JDO?

ODMG is a standard API for Object Persistence specified by the ODMG consortium (www.odmg.org). JDO is Sun's API specification for Object Persistence. ODMG may well be regarded as a Precursor to JDO. In fact JDO incorporates many ideas from ODMG and several people who have been involved in the ODMG spec are now in the JDO team.
I assume JDO will have tremendous influence on OODBMS-, RDBMS-, J2EE-server and O/R-tool-vendors to provide compliant products.
OJB wants to provide first class support for JDO and ODMG APIs.

OJB currently contains of four main layers, each with its own API:

1. A low-level **PersistenceBroker** API which serves as the OJB persistence kernel. The PersistenceBroker also provides a scalable multi-server architecture that allows to used it in heavy-duty app-server scenarios.
   This API can also be used directly by applications that don't need full fledged object level transactions (see PB tutorial for details).
2. An Object Transaction Manager (OTM) layer that contains all features that JDO and ODMG have in common as Object level transactions, lock-management, instance lifecyle etc. (See OTM tutorial for details.) The OTM is work in progress.
3. A full featured **ODMG 3.0** compliant API. (See ODMG tutorial for an introduction.)
   Currently this API is implemented on top the PersistenceBroker. Once the OTM layer is finished ODMG will be implemented on top of OTM.
4. A **JDO** compliant API. This is work in progress. (See JDO tutorial for an introduction.)
   Currently this API is implemented on top the PersistenceBroker. Once the OTM layer is finished JDO will be implemented on top of OTM.

The following graphics shows the layering of these APIs. Please note that the layers coloured in yellow are not yet implemented.

OJB Layer

1.3. What are the OJB design principals?

OJB has a "pattern driven" design. Please refer to this document for more details

1.4. Where can I learn more about Object/Relational mapping in general?

We have a link list pointing to further readings.

1.5. How OJB performance compares to native JDBC programming?

See page Performance.

1.6. How OJB performance compares to other O/R mapping tools?

See page Performance.

1.7. Is OJB ready for production environments?

Depends on your production environment. If you want to program an aeroplane autopilot system you should not use Java at all. (according to the official disclaimer).
But I assume we are talking about enterprise business applications, aren't we? And for such applications it's a clear **yes**. OJB is used in production application since version 0.5. We have about 6.000 downloads each month (and growing) and a large user base using it in a wide spectrum of production scenarios.
We provide a regression test suite for Quality Assurance. You can use this testsuite to check if OJB works smoothly in your target environment. (see supported platforms documentation)

We also provide a performance testsuite that compares OJB performance against native JDBC. This test will give you an impression of the performance impact OJB will have in your target environment. (see Performance testsuite documentation)
OJB is also the persistence layer of choice in several books on programming J2EE based enterprise business systems. (see our links and references section)
Reference projects and user testimonials are listed here.

2. Getting Started

2.1. Help! I'm having problems installing and using OJB!

Please read the Getting Started document. OJB is a powerful and complex system - installing and configuring OJB is not a trivial task. Be sure to follow **all the steps** mentioned in that document - don't skip any steps when first installing OJB on your systems.

If you are having problems running OJB against your target database, read the respective platform documentation. Before you try to deploy OJB to your environment, read the deployment guide.

2.2. Help! I still have serious problems installing OJB!

The following answer is quoted from the OJB user-list. It is from a reply to a user who had serious problems getting started with OJB.

I would say it was stupid not to understand OJB. How can you know what another programmer wrote. I've been a Java programmer for quite some time and I could show you stuff I wrote that I know you wouldn't understand. I'll just break it down the best I can on what, where and why.

OJB is a data persistence layer for Java. I'll just use an example of how I use it. I have an RDMS. I would like to save Java object states to this database and I would like to be able to search this information as well. If you serialize objects it's hard to search and if you use SQL it won't work with any different database. Plus it's a mess having to work with all that SQL in your code. And by using SQL you don't get to work with just Java objects. But, with OJB your separated from having to work outside the object world and unlike serialization you can preform SQL like searches on your data. Also, there's things like caching and connection pooling in OJB that help with performance. After setting up OJB you will use either PB-API or ODMG or JDO to access your information in a object centric manner. PB API is a non-standard O/R mapping API with many features and great flexibility. All top-level API's like ODMG or JDO build on top of the PB-api. ODMG is a standard for the api for accessing your data. That means you can use any ODMG compliant api if you don't want to use OJB. The JDO part is like ODMG except it's the SUN JDO standard. I use ODMG because the JDO interface is not ready yet.

OJB is easy to use. I'll just break it down into two sides. There's the side your writing your code for your application and there's the side that you configure to make OJB connect to your database. Starting with your application side, all that is needed is to use the interface you wish. I use ODMG because JDO is not complete yet. Here's a link to the ODMG part with some code for examples.
That's all you need on the application side. Next there's the configuration side. This is the one your fighting with. Here you need to setup the core tables for OJB and you will define the classes you wish to store in your database.

First thing to do is to build the cvs's with the default database HSQL, because you know it will work. If you get past this point you should have a working OJB compiled. Now if your using JDK 1.4 you will need to set in build.properties JDBC=+JDBC30 and do a *ant preprocess* first. Next you will do a *ant junit* and this will build OJB and test everything for you. If you get a build successful then your in business. Then you will want to run *ant jar* to create the OJB jar to put in your /lib. You will need a couple other jars in you /lib directory to make it all work. See this page for those. http://jakarta.apache.org/ojb/deployment.html

Next you will need some xml and configuration files in your class path for OJB. You will find those files under {$OJB_base_dir}/target/test/ojb. All the repository.xml's and OJB.properties for sure. With all these files in place with your application you should be ready to use OJB and start writing your application.

Finally you will want to setup your connection to your database and define your classes you will be storing in your database. In the repository.xml file you can configure your JDBC parameters so OJB can connect to your database. You will also need your JDBC jar somewhere in your class path. Then you will define your classes in the repository_user.xml file. Look here for examples. http://jakarta.apache.org/ojb/tutorial1.html Note you will want to comment out the junit part in repository.xml because it's just for testing.

The final thing to do is to make sure the OJB core tables are in your database. Look on this page for the core tables. These core tables are used by OJB to store internal data while it's running. It needs these. Then there's the tables you define. The ones you mapped in the repository_user.xml file.

Sorry if any of this is off. OJB is growing so fast that it's hard to keep up with all changes. The order I gave the steps in is just how I would think it's understood better. You can go in any order you want. The steps I've shown are mostly for deployment. Hope this helps you understand OJB a little better. I'm not sure if this is what your wanting or not.

2.3. OJB does not start?

If you carefully attended the installing hints there may be something wrong with your metadata mapping defined in the repository file or one the included sub files.

- Are you included all configuration files in classpath?
- On update to a new release, make sure you replaced all configuration files
- Check your metadata mapping - typos,... ?

If something going wrong while OJB read the metadata files you can enable *debug* log level for `org.apache.ojb.broker.metadata.RepositoryXmlHandler` and `org.apache.ojb.broker.metadata.ConnectionDescriptorXmlHandler` to get more detailed information.

Note:
If OJB default logging was used, change entries for these classes in OJB.properties file (this may change in future).

2.4. Does OJB support my RDBMS?

please refer to this document.

2.5. What are the OJB internal tables for?

Please refer to this document.

2.6. What does the exception Could not borrow connection from pool mean?

There can be several reasons

2.7. Any tools help to generate the metadata files?

Please refer to this document.

3. OJB api's

3.1. What are the differences between the PersistenceBroker API and the ODMG API? Which one should I use in my applications?

The PersistenceBroker (PB) provides a minimal API for transparent persistence:

- O/R mapping

- Retrieval of objects with a simple query interface from RDBMS
- storing (insert, update) of objects to RDBMS
- deleting of objects from RDBMS

This is all you need for simple applications as in tutorial1.

The OJB ODMG implementation uses the PB as its persistence kernel. But it provides much more functionality to the application developer. ODMG is a full fledged API for Object Persistence, including:

- OQL Query interface
- real Object Transactions
- A Locking Mechanism for management of concurrent threads (apps) accessing the same objects
- predefined persistent capable Collections and Hashtables

Some examples explaining the implications of these functional differences:

1. Say you use the PB to query an object O that has a collection attribute col with five elements a,b,c,d,e. Next you delete Objects d and e from col and store O again with PersistenceBroker.store(O);
   PB will store the remaining objects a,b,c. But it will not delete d and e ! If you then requery object O it will again contain a,b,c,d,e !!!
   The PB keeps no transactional state of the persistent Objects, thus it does not know that d and e have to be deleted. (as a side note: deletion of d and e could also be an error, as there might be references to them from other objects !!!)
   Using ODMG for the above scenario will eliminate all trouble: Objects are registered to a transaction so that on commit of the transaction it knows that d and e do not longer belong to the collection. the ODMG collection will not delete the objects d and e but only the REFERENCES from the collection to those objects!
2. Say you have two threads (applications) that try to access and modify the same object O. The PB has no means to check whether objects are used by concurrent threads. Thus it has no locking facilities. You can get all kind of trouble by this situation. The ODMG implementation has a Lockmanager that is capable of synchronizing concurrent threads. You can even use four transaction isolation levels:
   read-uncommitted, read-committed, repeatable-read, serializable.

In my eyes the PB is a persistence kernel that can be used to build high-level PersistenceManagers like an ODMG or JDO implementation. It can also be used to write simple applications, but you have to do all management things (locking, tracking objects state, object transactions) on your own.

3.2. 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()`.

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

tx.commit();
```

. . .

3.3. The OJB JDO implementation is not finished, how can I start using OJB?

I recommend to not use JDO now, but to use the existing ODMG api for the time being.

Migrating to JDO later will be smooth if you follow the following steps. I recommend to first divide your model layer into Activity- (or Process-) classes and Entity classes.

Entity classes represent classes that must be made persistent at some point in time, say a "Customer" or a "Order" object. These persistent classes and the repsective O/R mapping in repository.xml will remain unchanged.

Activities are classes that perform business tasks and work upon entities, e.g. "edit a Customer entry", "enter a new Order"... They implement (parts of) use cases.

Activities are driving transactions against the persistent storage.

I recommend to have a Transaction interface that your Activities can use. This Transaction interface can be implemented by ODMG or by JDO Transactions (which are quite similar). The implementation should be made configurable to allow to switch from ODMG to JDO later.

The most obvious difference between ODMG and JDO are the query languages: ODMG uses OQL, JDO define JDOQL. As an OO developer you won't like both of them. I recommend to use the ojb Query objects that allow an abstract syntax representation of queries. It is possible to use these queries within ODMG transactions and it will also be possible to use them within JDO Transactions. (this is contained in the FAQ too).

Using your own Transaction interface in conjunction with the OJB query api will provide a simple but powerful abstraction of the underlying persistence layer.

We are using this concept to provide an abstract layer above OJB-ODMG, TopLink and LDAP servers in my company. Making it work with OJB-JDO will be easy!

4. Howto

4.1. How to use OJB with my RDBMS?

please refer to this document.

4.2. What are the best settings for maximal performance?

See performance section.

4.3. How to page and sort?

Sorting can be configured by `org.apache.ojb.broker.query.Criteria::orderBy(column_name)`.

There is no paging support in OJB. OJB is concerned with Object/Relational mapping and not with application specific presentation details like presenting a scrollable page of items.

OJB returns query results as Collections or Iterators.

You can easily implement your partial display of result data by using an Iterator as returned by `ojb.broker.PersistenceBroker::getIteratorByQuery(...)`.

4.4. What about performance and memory usage if thousands of objects matching a query are returned as a Collection?

You can do two things to enhance performance if you have to process queries that produce thousands of result objects:

1. Use getIteratorByQuery() rather than getCollectionByQuery(). The returned Iterator is lazy and does not materialize Objects in advance. Objects are only materialized if you call the Iterators next() method. Thus you have total control about when and how many Objects get materialized! Please see here for proper handling.
2. You can define Proxy Objects as placeholder for your persistent business objects. Proxys are lighweight objects that contain only primary key information. Thus their materialization is not as expensive as a full object materialization. In your case this would result in a collection containing 1000 lighweight proxies. Materialization of the full objects does only occur if the objects are accessed directly. Thus you can build similar lazy paging as with the Iterator. You will find examples in the OJB test suite (src-distribution only: [db-ojb]/src/test). More info about Proxy object here.

The Perfomance of 1. will be better than 2. This approach will also work for VERY large resultsets, as there are no references to result objects that would prevent their garbage collectability.

4.5. When is it helpful to use Proxy Classes?

Proxy classes can be used for "lazy loading" aka "lazy materialization". Using Proxy classes can help you in reducing unneccessary db lookups. Example:

Say you load a ProductGroup object from the db which contains a collection of 15 Article objects.

Without proxies all 15 Article objects are immediately loaded from the db, even if you are not interested in them but just want to lookup the description-attribute of the ProductGroup object.

With a proxy class, the collection is filled with 15 proxy objects, that implement the same interface as the "real objects" but contain only an OID and a void reference.

Once you access such a proxy object it loads its "real subject" by OID and delegates the method call to it.

have a look at section proxy usage of page basic technique.

4.6. How can I convert data between RDBMS and OJB?

For Example I have a DB column of type INTEGER but a class atribute of type boolean. How can I provide an automatic mapping with OJB?

OJB provides a concept of ConversionStrategies that can be used for such conversion tasks. Have a look at the respective document.

4.7. How can I trace and/or profile SQL statements executed by OJB?

OJB ships with out of the box support for P6Spy. P6Spy is a JDBC proxy. It delegates all JDBC calls to the real JDBC driver and traces all calls to a log file.

In the file `build.properties` you have to set the switch `useP6Spy` to `true` in order to activate it:

```
# The useP6Spy switch determines if the tracing JDBC driver P6Spy
# is used.
# If you enable this switch, you must also edit the file
# jakarta-ojb/src/test/org/apache/ojb/spy.properties
# to tell P6Spy which JDBC driver to use and where to write the log.
# By default the HSQLDB driver is used.
useP6Spy=true
```

This setup uses P6Spy to trace and profile all executed SQL to a file `target/test/ojb/spy.log`. It also measures the time needed to execute each statement!

4.8. How does OJB manage foreign keys?

Automatically! you just define 1:1, 1:n or m:n associations in the repository_user.xml file. OJB does the rest!

Please refer to basic technique and xml-metadata repository for details.

4.9. How does OJB manage 'null' for primitive primary key?

Primitive values (int, long, ...) can't be `null`, so OJB interpret '0' as `null` for primitive PK/FK fields in persistent objects. Thus primitive PK fields of persistent objects should never be represented by a '0' value in DB and never used as a sequence key value.
This is only true for primitive PK/FK fields (e.g. `Integer(0)` is allowed). All other fields have 'normal' behavior.

4.10. How to lookup object by primary key?

Please see PB tutorial section.

4.11. Difference between getIteratorByQuery() and getCollectionByQuery()?

The first one returns an `org.apache.ojb.broker.OJBIterator` instance. The returned Iterator instance is lazy and does not materialize Objects in advance. Objects are only materialized from the underlying query result set if you call the Iterators next() method. If all objects materialized or the calling `org.apache.ojb.broker.PersistenceBroker` instance was closed or transaction demarcations ends the Iterator instance release all used resources (e.g. used Statement and ResultSet instances).

Method `getCollectionByQuery()` use an Iterator to materialize all objects first and then return the materialized objects within the `java.util.Collection` instance.

Note:

If method `getIteratorByQuery()` was used keep in mind that the used Iterator instance is only valid as long as the used `org.apache.ojb.broker.PersistenceBroker` instance ends transaction or be closed. So it is NOT possible to get an Iterator, close the PersistenceBroker and pass the Iterator instance to a servlet or client. In that case use `getCollectionByQuery()`.

4.12. How can Collections of primitive typed elements be mapped?

The first thing to ask is: How are these primitive typed elements (Strings are also treated as primitive types here) stored in the database.
1) are they treated as ordinary domain objects and stored in a separate table?
2) are they serialized into a Varchar field?
3) are they stored as a comma separated varchar field?
4) is each element of the vector or array stored in a separate column? (this solution does only work for a fixed number of elements!)
Follow these steps for solution 3):
a) simply define ordinary collection-descriptors as for every other collection of domain objects.
b) use the Object2ByteArrFieldConversion. See jdbc-types.html for details on conversion strategies.
c) use the StringVector2VarcharFieldConversion. See jdbc-types.html for details on conversion strategies.
d) provide a field-descriptor for each element.

4.13. How could class 'myClass' represent a collection of 'myClass' objects

OJB can handle such recursive associations without problems.

- add a collection attribute 'myClasses' to the class `myClass` this collection will hold the associated `myClass` objects.

- you have to decide wether this assosciation is 1:n or m:n.
  for 1:n you just need an additional foreignkey attribute in the MY_CLASS table. Of course you'll also need a matching attribute in the class `myClass`.
  For a m:n association you'll have to define a intermediary table to hold the mapping entries.
- define a `collection-descriptor` tag in the `class-descriptor` of `myClass` in repository.xml. Follow the steps in basic technique on 1:n and m:n.

4.14. How to lookup PersistenceBroker instances?

The `org.apache.ojb.broker.PersistenceBrokerFactory` make several methods available:

```
public PersistenceBroker createPersistenceBroker(PBKey key) throws PBFactoryException;

public PersistenceBroker createPersistenceBroker(String jcdAlias, String user, String password)
        throws PBFactoryException;

public PersistenceBroker defaultPersistenceBroker() throws PBFactoryException;
```

Method `defaultPersistenceBroker()` can be used if the attribute *default-connection* is set *true* in *jdbc-connection-descriptor*. It's a convenience method, useful when only one database is used.

The standard way to lookup a broker instance is via `org.apache.ojb.broker.PBKey` by specify *jcdAlias* (specified in the repository file (or sub file)), *user* and *passwd*. If the user and password is already set in *jdbc-connection-descriptor* it is possible to lookup the broker instance only be specify the *jcdAlias* in PBKey:

```
PBKey pbKey = new PBKey("myJcdAliasName");
PersistenceBroker broker = PersitenceBrokerFactory.createPersistenceBroker(pbKey);
```

See here too.

4.15. How to access ODMG?

Obtain a `org.odmg.Implementation` instance first, then create 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 not specified in the connection metadata (jdbc-connection-descriptor).

4.16. Needed to put user/password of database connection in repository file?

There is no need to put user/password in the repository file (more exact in the `jdbc-connection-descriptor`). You can pass this information at runtime. See Many different database user - How do they login?.

Only if you want to use convenience `PersistenceBroker` lookup method of `PersistenceBrokerFactory`, OJB needs all database connection information in the configuration files. More details see repository file doc - section jdbc-connection-descriptor `default-connection` attribute

See lookup PB api.
See lookup ODMG api.

```
PBKey pbKey = new PBKey(jcdAlias, user, passwd);
PersistenceBroker broker = PersistenceBrokerFactory.createPersistenceBroker(pbKey);
// or using a convenience (when default-connection was set in jdbc-connection-descriptor)
PersistenceBroker broker = PersistenceBrokerFactory.defaultPersistenceBroker();
```

4.17. Many different database user - How do they login?

There are two ways to do that. Define for each user a `jdbc-connection-descriptor` (unattractive way, because we have to add each new user to repository file), or let OJB handle this for you.
For it define **one** `jdbc-connection-descriptor`, now you can use the same `jcdAlias` name with different `User/Password`. OJB **copy** the defined `jdbc-connection-descriptor` and replace the `username` and `password` with the given `User/Password`.

PersistenceBroker-api example:

```
PBKey user_1 = new PBKey(jcdAlias,username, passwd);
PersistenceBroker broker =
PersistenceBrokerFactory.createPersistenceBroker(user_1);
...
```

ODMG-api example:

```
Implementation odmg = OJB.getInstance();
Database db = odmg.newDatabase();
db.open("jcdAlias#username#passwd", Database.OPEN_READ_WRITE);
...
```

Keep in mind, when the `connection-pool` element enables connection pooling, every user get its separate pool. See How does OJB handle connection pooling?.

4.18. How do I use multiple databases within OJB?

Define for each database a `jdbc-connection-descriptor`, use the different `jcdAlias` names in the repositry file to match the according database.

```
<jdbc-connection-descriptor
    jcd-alias="myFirstDb"
    ...
>
    ...
</jdbc-connection-descriptor>

<jdbc-connection-descriptor
    jcd-alias="mySecondDb"
    ...
>
    ...
</jdbc-connection-descriptor>
```

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).

4.19. How does OJB handle connection pooling?

Please have a look in section Connection Handling.

4.20. Can I directly obtain a java.sql.Connection within OJB?

Please have a look in section Connection Handling.

4.21. Is it possible to perform my own sql-queries in OJB?

There are serveral ways in OJB to do that.

If you completely want to bypass the OJBquery-api see Can I directly obtain a java.sql.Connection within OJB?.
A more elegant way is to use a `QueryBySQL` object:

```
String sql =
"SELECT A.Artikel_Nr FROM Artikel A, Kategorien PG"
+ " WHERE A.Kategorie_Nr = PG.Kategorie_Nr"
+ " AND PG.Kategorie_Nr = 2";
// get the QueryBySQL
Query q2 = QueryFactory.newQuery(Article.class, sql);

Iterator iter2 = broker.getIteratorByQuery(q2);
// or
Collection col2 = broker.getCollectionByQuery(q2);
```

4.22. Start OJB without a repository file?

See section Metadata Handling.

4.23. Connect to database at runtime?

See section Metadata Handling.

4.24. Add new persistent objects metadata (class-descriptor) at runtime?

See section Metadata Handling.

4.25. Global metadata changes at runtime?

Please see section Metadata Handling.

4.26. Per thread metadata changes at runtime?

Please see section Metadata Handling.

4.27. Is it possible to use OJB within EJB's?

Yes, see deployment instructions in the docs. Additional you can find some EJB example beans in package `org.apache.ojb.ejb` under `[jakarta-ojb]/src/ejb`.

4.28. Can OJB handle ternary (or higher) associations?

Yes, that's possible. Here is an example. With a ternary relationship there are three (or more) entities 'related' to each other. An example would be `Developer`, `Language` and `Project`.

Each entity is mapped to one table (`DEVELOPER`, `LANGUAGE` and `PROJECT`). To represent the combinations of these entities we need an additional bridge table (`PROJECTRELATIONSHIP`)with three columns holding the foreign keys to the other three tables (just like an m:n association is represented by an intermediary table with 2 columns).

To handle this table with OJB we have to define a class that is mapped on it. This Relationship class can then be used to perform queries/updates as with any other persistent class. Here is the layout of this class:

```
public class ProjectRelationship {
  Integer developerId;
  Integer languageId;
  Integer projectId;

  Developer developer;
  Language lanuage;
```

```
  Project project;

  /** setters and getters not shown for brevity**/
}
```

Here is the respective extract from the repository :

```
<class-descriptor
    class="ProjectRelationship"
    table="PROJECTRELATIONSHIP"
>
    <field-descriptor
        name="developerId"
        column="DEVELOPER_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="languageId"
          column="LANGUAGE_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="projectId"
          column="PROJECT_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <reference-descriptor
          name="developer"
          class-ref="Developer"
      >
        <foreignkey field-id-ref="developerId" />
      </reference-descriptor>
      <reference-descriptor
          name="language"
          class-ref="Language"
      >
        <foreignkey field-id-ref="languageId" />
      </reference-descriptor>
      <reference-descriptor
          name="project"
          class-ref="Project"
      >
        <foreignkey field-ref="projectId" />
      </reference-descriptor>
</class-descriptor>
```

Here is some sample code for storing a relationship :

```
Developer dev = .... ; // create or retrieve
Project  proj = .... ; // create or retrieve
Language lang = .... ; // create or retrieve

ProjectRelationship rel = new ProjectRelationship();
rel.setDeveloper(dev);
rel.setLanguage(lang);
rel.setProject(proj);

broker.store(r);
```

In the next code sample we are looking up all Projects that Developer "Bob" has done in "Java".

```
Criteria criteria = new Criteria();
criteria.addEqualTo("developer.name","Bob");
cirteria.addEquatTo("language.name","Java");
```

```
Query q = new QueryByCriteria(ProjectRelationship.class, criteria, true);
Iterator iter = Broker.getIteratorByQuery(q);

// now iterate over the collection and retrieve all projects:
while (iter.hasNext())
{
    ProjectRelationship rel = (ProjectRelationship) iter.next();
    System.out.println(rel.getProject().toString());
}
```

You could also have on the Project class-descriptor a `collection-descriptor` that returns all relationships associated with the Project. If it was call "projectRelationships" the following would give you all projects that have a relationship with "bob" and the language "java".

```
Criteria criteria = new Criteria();
criteria.addEqualTo("projectRelationships.developer.name","bob");
cirteria.addEquatTo("projectRelationships.language.name","java");

Query q = new QueryByCriteria(Project.class, criteria, true);
Collection projects = Broker.getCollectionByQuery(q);
```

This is the layout of the Project class:

```
public class Project {
  Integer id;
  String name;
  Collection projectRelationships;

  /** setters and getters not shown for brevity**/
}
```

This is the class-descriptor of the Project class:

```
<class-descriptor
    class="Project"
    table="PROJECT"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="name"
          column="NAME"
        jdbc-type="VARCHAR"
    />
      <collection-descriptor
          name="projectRelationships"
          element-class-ref="ProjectRelationship"
      >
        <inverse-foreignkey field-ref="projectId" />
      </collection-descriptor>
</class-descriptor>
```

4.29. How to map a list of Strings

You can not map a list of Strings with a collection descriptor. A collection descriptor can only be used if the element class is a persistent class too. But element-class-ref="java.lang.String" won't work, because it's no persistent entity class!
Follow these steps to provide a mapping for an attribute holding alist of Strings. Let's assume your persistent class has an attribute `listOfStrings` holding a list of Strings:

```
protected Collection listOfStrings;
```

The database table mapped to the persistent class has a colum `LIST_OF_STRINGS` of type `VARCHAR` that is used to hold all

strings.

```
<field-descriptor
    name="listOfStrings"
    column="LIST_OF_STRINGS"
    jdbc-type="VARCHAR"
    conversion=
"o.a.ojb.broker.accesslayer.conversions.StringVector2VarcharFieldConversion"
/>
```

4.30. How to set up Optimistic Locking

Optimistic locking use an additional column (*Timestamp* or *Integer*) which is incremented each time changes are committed to the object, and is utilizied to determine whether an optimistic transaction should succeed or fail. Optimistic locking is fast, because it checks data integrity only at update time.

1. In your table you need a dedicated column of type `INTEGER` or `TIMESTAMP`. Say the column is typed as `INTEGER` and named `VERSION_MAINTAINED_BY_OJB`.
2. You then need a (possibly private) attribute in your java class corresponding to the column. Say the attribute is defined as:

   ```
   private int versionMaintainedByOjb;
   ```
3. in repository.xml you need a field-descriptor for this attribute. this field-descriptor must specify `locking="true"`
4. The resulting field-descriptor will look as follows:

   ```
   <field-descriptor
       name="versionMaintainedByOjb"
       column="VERSION_MAINTAINED_BY_OJB"
       jdbc-type="INTEGER"
       locking="true"
   />
   ```

For further reference see also the repository documentation.

4.31. How to use OJB in a cluster

Q: I'm running a web site in a load-balanced/cluster environment. Multiple servlet engines (different VMs/HTTP sessions), each running an OJB instance, against a single shared database. How should OJB be configured to get the concurrent servlet engines synchronized properly?

transactional isolation and locking
If you are using the PersistenceBroker API use optimistic locking (OL) to let OJB handle write conflicts. To use OL define a TIMESTAMP or INTEGER column and the respective Java attribute for it. In the field-descriptor of this attribute set the attribute *locking="true"*.
If you are working with the ODMG API distributed pessemistic locking should be used, by setting the respective flag in OJB.properties.

sequence numbers
Use a SequenceManager that is safe across multiple JVMs. The NextVal based SequenceManagers or any other SequenceManager based on database mechanisms will be fine.

caching
You could use different caching implementations

1. Use the EmptyCacheImpl to avoid any dirty reads. (But: The EmptyCache cannot handle cyclic structures on load!)
2. Use the PerBrokerCache Implementation to avoid dirty reads.
3. Use the OSCache cache implementation as distributed object cache.

There is also a complete howto document available that covers these topics.

4.32. How to work with the ObjectCacheEmptyImpl

Q: I just tried to turn caching off by using ObjectCacheEmptyImpl setting in ObjectCacheClass, and it seems to continuously loop through the SQL statement infinitely. The default works fine though.
Any ideas why this might be?

A: The Problem you see is due to circular references in your data. Say A references B and B has a backreference to A.
Now we load A from the DB. If autoretrieve="true" for the reference-descriptor defining the reference to B, OJB will also load B. If autoretrieve="true" for the B-reference-descriptor describing the back-reference to A, OJB must retrieve A. And here is the key point.
If we use the defaultcache A will be in the cache already, as it was loaded first. So OJB will simply lookup A from the cache. No endless recursion!
But if we use the emptycache, A will not be cached. So OJB must load A from the DB. And then again B is retrieved, etc., etc. There's you endless recursion.
In other words: A non-empty cache is needed to allow proper loading of circular references. (Other O/R tools like TopLink work the same way).
If you still want to use the EmptyCacheImpl you should set autoretrieve="false" and load references explicitely by broker.retrieveReference(...).

4.33. JDO - Why must my persisten class implement javax.jdo.spi.PersistenceCapable?

As specified by JDO all persistent classe must implement the interface `javax.jdo.spi.PersistenceCapable`. If a class does not implement this interface a JDO implementation does not know how to handle it.
On the other hand the JDO spec claims to provide transaparent persistence. That is no persistence class is required to implement a specific interface or to be derived from a special base class.
Sounds like a contradiction? It is! The JDO spec resolves this contradiction by stating that a JDO implemention is responsible to add the methods required by `javax.jdo.spi.PersistenceCapable` to the the user classes. This "injection" could be achieved by Pre- or Post-processing. The strategy most implementations use is called "bytecode-enhancement". This is a postprocesing step that adds the required methods to the .class files of the persistent user classes.
The JDO Reference implementation also uses bytecode-enhancement. In order to enhance the Product class to implement the `javax.jdo.spi.PersistenceCapable` interface use the ant target "enhance-jdori" before launching the tutorial5 application. This is documentated in the first section of tutorial4.html.