

Frequently Asked Questions

Table of contents

1 Questions.....	3
1.1 1. General.....	3
1.1.1 1.1. Why OJB? Why do we need another O/R mapping tool?	3
1.1.2 1.2. How is OJB related to ODMG and JDO?	3
1.1.3 1.3. What are the OJB design principals?	4
1.1.4 1.4. Where can I learn more about Object/Relational mapping in general?	4
1.1.5 1.5. How OJB performance compares to native JDBC programming?	4
1.1.6 1.6. How OJB performance compares to other O/R mapping tools?	4
1.1.7 1.7. Is OJB ready for production environments?	4
1.1.8 1.8. Does OJB supports caching?	5
1.2 2. Getting Started.....	5
1.2.1 2.1. Help! I'm having problems installing and using OJB!	5
1.2.2 2.2. Help! I still have serious problems installing OJB!	5
1.2.3 2.3. OJB does not start?	6
1.2.4 2.4. Does OJB support my RDBMS?	6
1.2.5 2.5. What are the OJB internal tables for?	6
1.2.6 2.6. What does the exception Could not borrow connection from pool mean?	6
1.2.7 2.7. Any tools help to generate the metadata files?	6
1.3 3. OJB APIs.....	6
1.3.1 3.1. What are the differences between the different OJB APIs? Which one should I use in my applications?	7
1.3.2 3.2. I don't like OQL, can I use the PersistenceBroker Queries within ODMG?	7
1.3.3 3.3. The OJB JDO implementation is not finished, how can I start using OJB?	7
1.4 4. Howto.....	8
1.4.1 4.1. How to use OJB with my RDBMS?	8
1.4.2 4.2. How to use OJB in an web app?	8
1.4.3 4.3. What are the best settings for maximal performance?	8
1.4.4 4.4. How to page and sort?	8
1.4.5 4.5. What about performance and memory usage if thousands of objects matching a query are returned as a Collection?	9
1.4.6 4.6. When is it helpful to use Proxy Classes?	9
1.4.7 4.7. How can I convert data between RDBMS and OJB?	9
1.4.8 4.8. How can I trace and/or profile SQL statements executed by OJB?	9

1.4.9 4.9. How does OJB manage foreign keys?	10
1.4.10 4.10. How does OJB manage 'null' for primitive primary key?	10
1.4.11 4.11. How to lookup object by primary key?	10
1.4.12 4.12. Difference between getIteratorByQuery() and getCollectionByQuery()?	10
1.4.13 4.13. How can Collections of primitive typed elements be mapped?	11
1.4.14 4.14. How could class 'myClass' represent a collection of 'myClass' objects	11
1.4.15 4.15. How to lookup PersistenceBroker instances?	11
1.4.16 4.16. How to access ODMG?	11
1.4.17 4.17. Needed to put user/password of database connection in repository file?	11
1.4.18 4.18. Many different database user - How do they login?	12
1.4.19 4.19. How do I use multiple databases within OJB?	12
1.4.20 4.20. How does OJB handle connection pooling?	13
1.4.21 4.21. Can I directly obtain a java.sql.Connection within OJB?	13
1.4.22 4.22. Is it possible to perform my own sql-queries in OJB?	13
1.4.23 4.23. When does OJB open/close a connection?	13
1.4.24 4.24. Start OJB without a repository file?	13
1.4.25 4.25. Connect to database at runtime?	13
1.4.26 4.26. Hook into OJB - How to add Listener, callback interface?	13
1.4.27 4.27. Add new persistent objects metadata (class-descriptor) at runtime?	13
1.4.28 4.28. Global metadata changes at runtime?	13
1.4.29 4.29. Per thread metadata changes at runtime?	14
1.4.30 4.30. Is it possible to use OJB within EJB's?	14
1.4.31 4.31. Can OJB handle ternary (or higher) associations?	14
1.4.32 4.32. How to map a list of Strings	16
1.4.33 4.33. How to set up Optimistic Locking	16
1.4.34 4.34. How to use OJB in a cluster	16
1.4.35 4.35. How to turn of caching?	17
1.4.36 4.36. JDO - Why must my persisten class implement javax.jdo.spi.PersistenceCapable?	17

Questions

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 highly 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 easily 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 use 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 lifecycle 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](#).

1.8. Does OJB supports caching?

Short answer is yes. To get a detailed answer, please read the [caching guide](#).

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 `{ $OBJ_base_dir }/target/test/obj`. 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/obj/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.obj.broker.metadata.RepositoryXmlHandler` and `org.apache.obj.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 APIs

3.1. What are the differences between the different OJB APIs? 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?

Please refer to the [ODMG-guide](#).

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 respective 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. How to use OJB in an web app?

If you follow these rules, then OJB works fine in web apps:

- Don't put OJB's jars into one of the servers directories but rather put them into the WEB-INF/lib folder of your web app.
- OJB searches for its configuration files (OJB.properties, repository.xml) in the classpath. Therefore, it is easiest if you put them in the WEB-INF/classes folder which is automatically in the classpath of the web app
- Don't hold onto the PersistenceBroker instances, rather get one whenever you want to do something, and close it once you're done.

See [deployment doc](#) for more information.

4.3. What are the best settings for maximal performance?

See [performance section](#).

4.4. 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 `objb.broker.PersistenceBroker::getIteratorByQuery(...)`.

4.5. 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 lightweight 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 lightweight 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 Performance 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.

4.6. 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 unnecessary 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](#).

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

For Example I have a DB column of type `INTEGER` but a class attribute 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](#).

4.8. 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 which delegates all JDBC calls to the real JDBC driver and traces all calls to a log file.

P6Spy is contained in the `p6spy.jar`, which you'll find in the `lib` folder of your OJB distribution.

Add this to the classpath of your app (if you're using the [ojb-blank](#) project, then simply copy the jar into the `lib` folder of the project and if you're using Eclipse then also add it to the project build path).

Now the only other thing left is to configure OJB to use P6Spy, and P6Spy to use your database's driver. To achieve this, change the database driver in your `jdbc-connection-descriptor` (in your repository file) to

```
<jdbc-connection-descriptor
  ...
  driver="com.p6spy.engine.spy.P6SpyDriver"
  ...
/>
```

In `ojb-blank` this setting is changed in the `build.properties` instead.

Also copy the file `spy.properties` which can be found in the `src/test/org/apache/ojb` folder into your classpath (e.g. in the same place where your `OJB.properties` file is). In this file you'll find a line starting with `realdriver` where you should put the name of the jdbc driver of your database, e.g.

```
realdriver=org.hsqldb.jdbcDriver
```

Also, here you can influence to where P6Spy will output the SQL statements. The appender defines how the logging is performed, e.g. to the console or to a file. The `logfile` setting defines into which file the statements will be printed (when a file appender is used). For instance, these settings will write to a file `spy.log`:

```
logfile = spy.log
appender = com.p6spy.engine.logging.appender.FileLogger

# This would be logging to the console
#appender = com.p6spy.engine.logging.appender.StdoutLogger
```

That's all there is to it, no recompile or other change of your app is necessary. Btw, P6Spy also measures the time needed to execute each statement!

4.9. 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.

4.10. 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.

4.11. How to lookup object by primary key?

Please see [PB tutorial section](#).

4.12. 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.obj.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.obj.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()`.

4.13. 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.

4.14. 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 association 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.

4.15. How to lookup PersistenceBroker instances?

Please refer to [PB-guide](#).

4.16. How to access ODMG?

Please refer to [ODMG-guide](#).

4.17. 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();
```

4.18. 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?](#).

4.19. How do I use multiple databases within OJB?

Define for each database a jdbc-connection-descriptor, use the different jcdAlias names in the [repository file](#) to match the according database.

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

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

Specific notes related to the [PB-api here](#).

Specific notes related to the [ODMG-api here](#).

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).

4.20. How does OJB handle connection pooling?

Please have a look in section [Connection Handling](#).

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

Please have a look in section [Connection Handling](#).

4.22. 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);
```

4.23. When does OJB open/close a connection?

Please see [Connection handling guide](#).

4.24. Start OJB without a repository file?

See section [Metadata Handling](#).

4.25. Connect to database at runtime?

See section [Metadata Handling](#).

4.26. Hook into OJB - How to add Listener, callback interface?

See *Listener/Callback* section in [PB-Guide](#).

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

See section [Metadata Handling](#).

4.28. Global metadata changes at runtime?

Please see section [Metadata Handling](#).

4.29. Per thread metadata changes at runtime?

Please see section [Metadata Handling](#).

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

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

4.31. 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 language;
    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");
criteria.addEqualTo("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");
criteria.addEqualTo("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>

```

4.32. 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 a list 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 column `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.obj.broker.accesslayer.conversions.StringVector2VarcharFieldConversion"
/>

```

4.33. How to set up Optimistic Locking

Please see [locking section](#).

4.34. 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.

4.35. How to turn of caching?

Declare an no-op implementation of the ObjectCache interface as cache. See [detailed description here](#).

4.36. 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 transparent 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 implementation 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 documented in the first section of tutorial4.html.