### Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>Simple</td>
<td>4</td>
</tr>
<tr>
<td>Small</td>
<td>4</td>
</tr>
<tr>
<td>Fast</td>
<td>4</td>
</tr>
<tr>
<td>Safe and robust</td>
<td>4</td>
</tr>
<tr>
<td>One single database file</td>
<td>4</td>
</tr>
<tr>
<td>Multiplatform</td>
<td>5</td>
</tr>
<tr>
<td>Thread safety</td>
<td>5</td>
</tr>
<tr>
<td>Data are always available</td>
<td>5</td>
</tr>
<tr>
<td>Productivity</td>
<td>5</td>
</tr>
<tr>
<td>Easy to integrate</td>
<td>5</td>
</tr>
<tr>
<td>Refactoring</td>
<td>6</td>
</tr>
<tr>
<td>License</td>
<td>6</td>
</tr>
<tr>
<td>Wiki</td>
<td>7</td>
</tr>
<tr>
<td>Download</td>
<td>7</td>
</tr>
<tr>
<td>Content of the download</td>
<td>7</td>
</tr>
<tr>
<td>Distributions</td>
<td>7</td>
</tr>
<tr>
<td>How to execute ODB</td>
<td>7</td>
</tr>
<tr>
<td>Integrating ODB into your IDE</td>
<td>8</td>
</tr>
<tr>
<td>Using ODB in Web Applications</td>
<td>9</td>
</tr>
<tr>
<td>Migration from previous releases to 1.8</td>
<td>9</td>
</tr>
<tr>
<td>Database</td>
<td>9</td>
</tr>
<tr>
<td>API</td>
<td>9</td>
</tr>
<tr>
<td>Storing Objects</td>
<td>9</td>
</tr>
<tr>
<td>Player Class</td>
<td>12</td>
</tr>
<tr>
<td>Team class</td>
<td>12</td>
</tr>
<tr>
<td>Game Class</td>
<td>13</td>
</tr>
<tr>
<td>Object retrieving</td>
<td>19</td>
</tr>
<tr>
<td>Retrieving all objects of a specific class</td>
<td>19</td>
</tr>
<tr>
<td>CriteriaQuery</td>
<td>20</td>
</tr>
<tr>
<td>Native Query</td>
<td>25</td>
</tr>
<tr>
<td>Retrieving an object by its OID</td>
<td>26</td>
</tr>
<tr>
<td>Query tuning</td>
<td>27</td>
</tr>
<tr>
<td>Indexes</td>
<td>27</td>
</tr>
<tr>
<td>Updating Objects</td>
<td>28</td>
</tr>
<tr>
<td>Deleting objects</td>
<td>30</td>
</tr>
<tr>
<td>Example 2: deleting an object using its internal id</td>
<td>31</td>
</tr>
<tr>
<td>Using ODB as Client/Server</td>
<td>32</td>
</tr>
<tr>
<td>ODBExplorer</td>
<td>35</td>
</tr>
</tbody>
</table>
The aim of the tutorial is to introduce the basic concepts of **ODB**.

You will learn to store, retrieve, update and delete objects. More advanced concepts like XML importation/Exportation, Defragmentation and tuning will also be presented.

For any question please access the site [odb.neodatis.org](http://odb.neodatis.org).

**Warning**: If you are migrating from **NeoDatis ODB 1.5** to **1.8** see the migration section.

**Overview**

**NeoDatis ODB** is a new generation Object Oriented Database. **ODB** is a real transparent persistence layer that allows anyone to persist native objects with a single line of code.

**ODB** can be used as an embedded database engine that can be seamlessly integrated to any product without requiring any specific installation or in client/server mode.
**ODB** simplifies software development by turning totally transparent the persisting layer.

**Simple**

**ODB** is very simple and intuitive: the learning time is very short. Have a look at the [**ODB one minute tutorial**](#) to check this. The API is simple and does not require learning any mapping technique. There is no need for mapping between the native objects and the persistence store. **ODB simply stores the objects the way they are.** **ODB** requires zero administration and zero installation.

**Small**

The **ODB** runtime is less than 450k and is distributed as a single jar/dll that can easily be packaged in any type of application.

**Fast**

**ODB** can store more than 20000 objects per second.

**Safe and robust**

**ODB** supports ACID transactions to guarantee data integrity of the database. All committed work will be applied to the database even in case of hardware failure. This is done by automatic transaction recovery on the next startup.

**One single database file**

**ODB** uses a single file to store all data:
- The Meta-model
- The objects
- The indexes

For better performance, **ODB** can be configured to use more than one file.
**Multiplatform**

**ODB** runs on Java and .Net (Microsoft and Mono) platform*

*: It currently works on Java platform and is being ported to .Net platform (Mono and MS .Net version can be downloaded from cvs).

**Thread safety**

**ODB** can be used in a multi-threaded environment. **ODB** only needs to be informed about the thread pool size (calling Configuration.useMultiThread(true, pool size)).

**Data are always available**

**ODB** lets you export all data to a standard XML Format (Annex 1: Xml Exported file of the tutorial ODB base) which guarantee that data are always available. **ODB** can also import data from the same XML format. Import and Export features are available via API or via the **ODB** Object Explorer.

**Productivity**

**ODB** lets you persist data with a very few lines of code. There is no need to modify the classes that must be persisted and no mapping is needed. So developers can concentrate on business logic implementation instead of wasting time with the persistence layer.

**Easy to integrate**

The only requirement to use **ODB** is to have a single jar/dll on the application classpath/path.
Refactoring

**ODB** currently supports 5 types of refactoring:

- Renaming a class
- Renaming a Field
- Changing the type of a field (respect the **ODB** Matrix Type Compatibility) (not yet implemented)
- Adding a new Field (automatically detected)
- Removing a field (automatically detected)

License

**ODB** is distributed under the LGPL license.
Wiki

Check the wiki.neodatis.org site to access to new Wiki that contains much more documentation!

Download

The last ODB distribution can be downloaded at http://www.neodatis.org

Content of the download

The ODB Distribution contains:

- Root directory:
  - The ODB runtime: neodatis-odb.jar
- Doc directory:
  - NeoDatisODB.pdf : The ODB documentation
  - tutorial.jar, a jar with the classes used in the documentation
  - run-tutorial.bat: a batch to execute the tutorial
  - build.xml: An ant script to build the tutorial jar
  - src: the source code of all the classes used in the tutorial.

Distributions

<table>
<thead>
<tr>
<th>Jar</th>
<th>Size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>neodatis-odb.jar</td>
<td>520kb</td>
<td>The complete ODB</td>
</tr>
<tr>
<td>neodatis-odb-rt.jar</td>
<td>420kb</td>
<td>same as neodatis-odb.jar, without ODBExplorer</td>
</tr>
<tr>
<td>neodatis-odb-rt-lite.jar</td>
<td>327kb</td>
<td>The ODB local runtime without ODBExplorer, XML import/export, Client/server mode</td>
</tr>
<tr>
<td>neodatis-odb-gui.jar</td>
<td>100kb</td>
<td>The ODBExplorer</td>
</tr>
</tbody>
</table>

How to execute ODB

A single jar (neodatis-odb.jar) is needed to run the ODB database.
To execute a class that use **ODB** to persist objects, just add the **ODB** runtime to the classpath:

```java
java –cp neodatis-odb.jar <your-class-name>
```

**Integrating ODB into your IDE**

To use **ODB** in your favorite IDE, the classpath of your project must be updated to contain the odb runtime jar.

**Using ODB in an Eclipse Project:** Select your project, right-click on the project root (In the Navigator view or in the package explorer) choose Properties and then click on the ‘Java Build Path’ item. In the library tab add the odb runtime jar:
Using ODB in Web Applications

To use ODB in WEB application, you just need put the ODB jar in the WEB-INF/lib of the war. The default place of the ODB database file (if not specified when opening the ODB file) will be the execution directory of the web container. For example, if your use Tomcat, the ODB database file will be created in the $TOMCAT/bin directory.

Migration from previous releases to 1.8

Previous releases include all inclusive 1.8 beta releases.

Database

The database file format has changed so it is necessary to export database to XML file using previous version and import the xml in a new database using the 1.8 beta 4 version. This can be done easily with the ODBExplorer graphical application.

API

Some changes have been done to the ODB interface:

- The Object ID is now an object, it is not a long anymore. So the methods `getObjectId` and `getObjectById` now work with OID interface instead of long.

- The store method now returns the ODB OID of the new created object.

- The delete method on interface ODB now throw an Exception instead of IOException

- The getObjects method now return an object of type Objects (which implements Collection) instead of a list. So the iterator pattern should be used to get objects from the return collection.

- The Class Restrictions used in CriteriaQuery has been replaced by a class called Where. The Restrictions class remains available for compatibility issues but it is marked as deprecated.

Storing Objects

11/02/2008  www.neodatis.org  9/52
For this Tutorial, we will create some data objects related with the Sport domain: Sport, Player, Team, Game... To simplify, we only describe class attributes in code sections, getters, setters and toString methods will be omitted.

Let’s start creating a class **Sport** with a single name attribute:

```java
package org.neodatis.odb.tutorial;

public class Sport {
    private String name;

    public Sport(String name) {
        this.name = name;
    }
}
```

To store an object, we need to create a **Sport** instance, open the database and store the object.

To simplify the source code, we use a Constant to define the name of the **ODB** base:

```java
public static final String ODB_NAME = "tutorial1.odb";
```

And then

```java
public void step1() throws Exception{
    // Create instance
    Sport sport = new Sport("volley-ball");

    // Open the database
    ODB odb = ODBFactory.open(ODB_NAME);

    // Store the object
    odb.store(sport);

    // Close the database
    odb.close();
}
```
After this first step, our database already contains an instance of Sport. Let’s execute the following code to display the instances of Sport of our database:

```java
public void displaySports() throws Exception{
// Open the database
    ODB odb = ODBFactory.open(ODB_NAME);

    // Get all object of type Sport
    Objects sports = odb.getObjects(Sport.class);

    // display each object
    Sport sport = null;
    while(sports.hasNext()){
        sport = (Sport) sports.next();
        System.out.println(sport.getName());
    }

    // Closes the database
    odb.close();
}
```

This code should produce the following output:

```
1 sport(s):
1    : volley-ball
```

The most important point here is that the only thing you have to do to store an object is to call the `store` method.

Let’s create more classes to increase the complexity of our model.

A **Sport** needs one or two teams of players. So let’s create a **Player** class and a **Team** class. The **Player** has a name, a date of birth and a favorite sport. A **Team** has a name and a list of players. Then we can create the class **Game** that has a sport and two teams.
**Player Class**

```java
package org.neodatis.odb.tutorial;

import java.util.Date;

public class Player {
    private String name;
    private Date birthDate;
    private Sport favoriteSport;

    public Player(String name, Date birthDate, Sport favoriteSport) {
        this.name = name;
        this.birthDate = birthDate;
        this.favoriteSport = favoriteSport;
    }
}
```

**Team class**

```java
package org.neodatis.odb.tutorial;

import java.util.List;

public class Team {
    private String name;
    private List players;

    public Team(String name) {
        this.name = name;
        players = new ArrayList();
    }
}
```
Game Class

```java
package org.neodatis.odb.tutorial;

public class Game {
    private Sport sport;
    private Team team1;
    private Team team2;
    private String result;

    public Game(Sport sport, Team team1, Team team2) {
        this.sport = sport;
        this.team1 = team1;
        this.team2 = team2;
    }
}
```

Now, we can create a more complex scenario storing a bigger object structure:
We first create an instance of Sport(Volley-ball), create 4 Players, then two Teams with 2 players each and finally a Game of Volley-ball with the two teams.

After this, to persist all the objects, you only need to persist the game instance. ODB will traverse the instance and store all objects it references:
public void step2() throws Exception {

    // Create instance
    Sport volleyball = new Sport("volley-ball");

    // Create 4 players
    Player player1 = new Player("olivier", new Date(), volleyball);
    Player player2 = new Player("pierre", new Date(), volleyball);
    Player player3 = new Player("elohim", new Date(), volleyball);
    Player player4 = new Player("minh", new Date(), volleyball);

    // Create two teams
    Team team1 = new Team("Paris");
    Team team2 = new Team("Montpellier");

    // Set players for team1
    team1.addPlayer(player1);
    team1.addPlayer(player2);

    // Set players for team2
    team2.addPlayer(player3);
    team2.addPlayer(player4);

    // Then create a volley ball game for the two teams
    Game game = new Game(new Date(), volleyball, team1, team2);

    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);

        // Store the object
        odb.store(game);
    } finally {
        if (odb != null) {
            // Close the database
            odb.close();
        }
    }
}
After this execution of the step 2, ODB should contain:

- 1 instance of **Game**
- 2 instances of **Team**
- 4 instances of **Player**
- 1 instance of **Sport**

Let's check this, here the output of querying objects of each type:

```
Step 2 : 1 games(s):
  1     : Thu Jun 22 06:29:13 BRT 2006 : Game of volley-ball between Paris and Montpellier

Step 2 : 2 team(s):
  1     : Team Paris [olivier, pierre]
  2     : Team Montpellier [elohim, minh]

Step 2 : 4 player(s):
  1     : olivier
  2     : pierre
  3     : elohim
  4     : minh

Step 2 : 1 sport(s):
  1     : volley-ball
```

This example shows how it is simple to store complex objects, as you don’t need to worry in storing each single objects, storing the top-level object will resolve.
Object retrieving

In the previous example, we learned how to store objects. Now, obviously, we need to get these objects back. For instance, ODB has four ways to retrieve objects:

- Retrieving all objects of a specific class
- Retrieving a subset of objects of a specific class using CriteriaQuery
- Retrieving a subset of objects of a specific class using NativeQuery
- Retrieving object by OID

It will soon support Query By example and SQL-Like queries.

Retrieving all objects of a specific class

The ODB interface has a method getObjects that receive a class and returns an object of type Objects (that implements Collection). This method is used to obtain all objects of a specific class:

```
ODB odb = null;
try{
    // Open the database
    ODBFactory.open(ODB_NAME);

    // Get all object of type clazz
    Objects objects = odb.getObjects(Player.class);

    System.out.println(objects.size() + " player(s)");

    // display each object
    While(objects.hasNext()){
        System.out.println((i+1) + "\t: " + objects.next());
    }
}finally{
    // Closes the database
    If(odb!=null){
        odb.close();
    }
}
```
This code opens the database, retrieve a list of all objects of type \texttt{Player} and displays each one.

\textbf{CriteriaQuery}

CriteriaQuery let’s you specify Where on objects that the query result must contain. The \texttt{ODB CriteriaQuery} API is very close to the \texttt{Hibernate Criteria API}.

Here is a simple example of CriteriaQuery:

```java
public void step3() throws Exception {
    ODB odb = null;
    
    try {
        // Open the database
        odb = ODBFactory.open(\texttt{ODB_NAME});
        
        IQery query = \texttt{new CriteriaQuery(\texttt{Player.class},
            Where.equal("name", "olivier"));}
        
        Objects players = odb.getObjects(query);
        System.out.println("\nStep 3 : Players with name olivier";
        
        // display each object
        \texttt{while(players.hasNext()) { \
            System.out.println((i + 1) + "\t: " + players.next());
        
        } finally {
            if (odb != null) {
                // Close the database
                \texttt{// Close the database
                odb.close();
            }
        }
    
    }
```

The following code creates a query on objects of type \texttt{Player} where the name is equal to \texttt{Olivier}.

\texttt{IQuery query = new CriteriaQuery(\texttt{Player.class, Where.equal("name", "olivier"));}
A powerful feature of **ODB CriteriaQuery** is the capability of navigating through object relations. The following example demonstrates this feature by retrieving all players whose favorite sport is Volley-ball:

```java
IQuery query = new CriteriaQuery(
    Player.class,
    Where.equal("favoriteSport.name", "volley-ball"));
```

Another way to get the result is to get the object volley-ball:

```java
IQuery query = new CriteriaQuery(
    Sport.class,
    Where.equal("name", "volley-ball"));

Sport volleyBall = (Sport) odb.getObjects(query).getFirst();
```

And then use the following criteria query to get all players that play volley-ball(using the volley-ball object previously retrieved):

```java
// Now build a query to get all players that play volley ball, using the volley ball object
IQuery query = new CriteriaQuery(
    Player.class,
    Where.equal("favoriteSport", volleyBall));

Objects players = odb.getObjects(query);
```

So, as you can see, **CriteriaQuery** Where work with objects too.
Other functions are available while working with **CriteriaQuery**. Here is a list of functions available in the Where Factory:

<table>
<thead>
<tr>
<th>equality</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>public static ICriterion equal(String attributeName, boolean value)</td>
<td>for primitive boolean value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, int value)</td>
<td>for primitive int value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, short value)</td>
<td>for primitive short value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, byte value)</td>
<td>for primitive byte value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, float value)</td>
<td>for primitive float value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, double value)</td>
<td>for primitive double value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, long value)</td>
<td>for primitive long value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, char value)</td>
<td>for primitive char value</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, Object value)</td>
<td>for object</td>
</tr>
<tr>
<td>public static ICriterion equal(String attributeName, Object value) Case insensitive equal.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>like</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>public static ICriterion like(String attributeName, String value)</td>
<td>for patterns like 'name=&quot;pet %&quot;'</td>
</tr>
<tr>
<td>public static ICriterion ilike(String attributeName, String value)</td>
<td>with case insensitive option</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>greater than (gt)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>public static ICriterion gt(String attributeName, Comparable value)</td>
<td>for Comparable objects</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, int value)</td>
<td>for primitive int</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, short value)</td>
<td>for primitive short</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, byte value)</td>
<td>for primitive byte</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, float value)</td>
<td>for primitive float</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, double value)</td>
<td>for primitive double</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, long value)</td>
<td>for primitive long</td>
</tr>
<tr>
<td>public static ICriterion gt(String attributeName, char value)</td>
<td>for primitive char</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>greater or equal (ge)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>public static ICriterion ge(String attributeName, Comparable value)</td>
<td>for Comparable objects</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, int value)</td>
<td>for primitive int</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, short value)</td>
<td>for primitive short</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, byte value)</td>
<td>for primitive byte</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, float value)</td>
<td>for primitive float</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, double value)</td>
<td>for primitive double</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, long value)</td>
<td>for primitive long</td>
</tr>
<tr>
<td>public static ICriterion ge(String attributeName, char value)</td>
<td>for primitive char</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>less than (lt)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>public static ICriterion lt(String attributeName, Comparable value)</td>
<td>for Comparable objects</td>
</tr>
<tr>
<td>public static ICriterion lt(String attributeName, int value)</td>
<td>for primitive int</td>
</tr>
<tr>
<td>public static ICriterion lt(String attributeName, short value)</td>
<td>for primitive short</td>
</tr>
<tr>
<td>public static ICriterion lt(String attributeName, byte value)</td>
<td>for primitive byte</td>
</tr>
<tr>
<td>public static ICriterion lt(String attributeName, float value)</td>
<td>for primitive float</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>public static ICriterion lt(String attributeName, double value)</code></td>
<td>for primitive double</td>
</tr>
<tr>
<td><code>public static ICriterion lt(String attributeName, long value)</code></td>
<td>for primitive long</td>
</tr>
<tr>
<td><code>public static ICriterion lt(String attributeName, long value)</code></td>
<td>for primitive char</td>
</tr>
<tr>
<td><strong>less or equal (le)</strong></td>
<td></td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, Comparable value)</code></td>
<td>for Comparable objects</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, int value)</code></td>
<td>for primitive int</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, short value)</code></td>
<td>for primitive short</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, byte value)</code></td>
<td>for primitive byte</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, float value)</code></td>
<td>for primitive float</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, double value)</code></td>
<td>for primitive double</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, long value)</code></td>
<td>for primitive long</td>
</tr>
<tr>
<td><code>public static ICriterion le(String attributeName, char value)</code></td>
<td>for primitive char</td>
</tr>
<tr>
<td><strong>contain</strong> - To test if a array or a collection contain a specific value</td>
<td></td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, boolean value)</code></td>
<td>for primitive boolean value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, int value)</code></td>
<td>for primitive int value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, short value)</code></td>
<td>for primitive short value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, byte value)</code></td>
<td>for primitive byte value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, float value)</code></td>
<td>for primitive float value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, double value)</code></td>
<td>for primitive double value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, long value)</code></td>
<td>for primitive long value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, char value)</code></td>
<td>for primitive char value</td>
</tr>
<tr>
<td><code>public static ICriterion contain(String attributeName, Object value)</code></td>
<td>for object</td>
</tr>
<tr>
<td><strong>Null Objects</strong></td>
<td></td>
</tr>
<tr>
<td><code>public static ICriterion isNull(String attributeName)</code></td>
<td>only null objects</td>
</tr>
<tr>
<td><code>public static ICriterion isNotNull(String attributeName)</code></td>
<td>only not null objects</td>
</tr>
<tr>
<td><strong>Where on collection or array size</strong></td>
<td></td>
</tr>
<tr>
<td><code>public static ICriterion sizeEq(String attributeName, int size)</code></td>
<td>with a size equal to</td>
</tr>
<tr>
<td><code>public static ICriterion sizeNe(String attributeName, int size)</code></td>
<td>with a size not equal to</td>
</tr>
<tr>
<td><code>public static ICriterion sizeGt(String attributeName, int size)</code></td>
<td>with a size greater than</td>
</tr>
<tr>
<td><code>public static ICriterion sizeGe(String attributeName, int size)</code></td>
<td>with a size greater or equal</td>
</tr>
<tr>
<td><code>public static ICriterion sizeLt(String attributeName, int size)</code></td>
<td>with a size lesser than</td>
</tr>
<tr>
<td><code>public static ICriterion sizeLe(String attributeName, int size)</code></td>
<td>with a size lesser or equal</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td></td>
</tr>
<tr>
<td><code>public static ComposedExpression or()</code></td>
<td>a OR expression</td>
</tr>
<tr>
<td><code>public static ComposedExpression and()</code></td>
<td>a AND expression</td>
</tr>
<tr>
<td><code>public static IExpression not(ICriterion criterion)</code></td>
<td>negate a restriction</td>
</tr>
</tbody>
</table>
More examples of Criteria query:

```java
// users that have a profile which name is 'profile2'
CriteriaQuery query = new CriteriaQuery(User.class, Where.equal("profile.name", "profile2"));

// users that have a specific profile p0
query = new CriteriaQuery(User.class, Where.equal("profile", p0));

// users with a specific function in their profile
query = new CriteriaQuery(User.class, Where.contain("profile.functions", f2bis));

// users with a profile that contain no function
query = new CriteriaQuery(User.class, Where.sizeEq("profile.functions", 0));

// users with a profile that have 4 functions
query = new CriteriaQuery(User.class, Where.sizeEq("profile.functions", 4));

// users with a profile that have 1 function
query = new CriteriaQuery(User.class, Where.sizeEq("profile.functions", 1));

// users with a profile that have more than 2 functions
query = new CriteriaQuery(User.class, Where.sizeGt("profile.functions", 2));

// users with a profile that does not have 1 function
query = new CriteriaQuery(User.class, Where.sizeNe("profile.functions", 1));

// TestClass objects where the attribute 'bigDecimal1' is null
query = new CriteriaQuery(TestClass.class, Where.isNull("bigDecimal1"));

// TestClass objects where the attribute 'string1' is equal to 'test class 1' or 'test class 2'
aq = new CriteriaQuery(TestClass.class,
    Where.or()
    .add(Where.equal("string1", "test class 1"));
   .add(Where.equal("string1", "test class 3"));

// TestClass objects where the attribute 'string1' is not equal to 'test class 2'
aq = new CriteriaQuery(TestClass.class, Where.not(Where.equal("string1", "test class 2")));

// TestClass objects where the condition 'string1' is equal to 'test class 0' or the attribute 'bigDecimal1' is equal to 5 is not matched
aq = new CriteriaQuery(TestClass.class,
    Where.not(
    Where.or()
    .add(Where.equal("string1", "test class 0"))
    .add(Where.equal("bigDecimal1", new BigDecimal("5"))));

// TestClass object where the attribute 'string1' is equal to 'test class 2' or 'test class 3' or 'test class 4' or 'test class 5'
// The query result will be ordered by the fields 'boolean1' and 'int1'
ICriterion c = Where.or()
    .add(Where.equal("string1", "test class 2"))
    .add(Where.equal("string1", "test class 3"))
    .add(Where.equal("string1", "test class 4"))
    .add(Where.equal("string1", "test class 5"));
aq = new CriteriaQuery(TestClass.class, c);
```
**Native Query**

Native queries (NQ) were introduced by Prof. William Cook at the 27th International Conference on Software Engineering (ICSE) in May of 2005 (They were first implemented by Db4O – www.db4o.com). NQs are queries written in native language. A native query is a piece of code that receives an object of the database and returns a Boolean value to indicate the query manager if the object must be included in the query result set.

Native Queries advantages are:
- No need to learn another query language
- As NQs are written in native language (Java or .net):
  - NQs are ‘refactorable’
  - No more problems with string based queries, NQs are checked in compile time

To implement a Native query in **ODB**, you must implement the interface `SimpleNativeQuery`. This interface do not have contract, but you must implement the following method `boolean match(ObjectType object)`

Where **ObjectType** must be the class of the objects that must be queried. For example, to execute the query on Player objects, the method signature should be:

```java
public boolean match(Player player)
```
A native query that return all players whose favorite sport’s name(transformed to lower case) starts with ‘volley’:

```java
public void step8() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);
        IQuery query = new SimpleNativeQuery() {
            public boolean match(Player player) {
                return player.getFavoriteSport()
                    .getName().toLowerCase()
                    .startsWith("volley");
            }
        };

        Objects players = odb.getObjects(query);

        System.out.println("\nStep 8 bis: Players that play Volley-ball");
        // display each object
        while (players.hasNext()) {
            System.out.println((i + 1) + "\t: " + players.next());
        }
    } finally {
        if (odb != null) {
            // Close the database
            // Close the database
            odb.close();
        }
    }
}
```

**Retrieving an object by its OID**

If you have the OID of an object, you can use the getObjectFromId to directly retrieve it.

The OID is returned by the ODB.store(Object) and ODB.getObjectId(Object) methods.

**Warning:** The method getObjectId can only be called for objects stored or retrieved in the current open ODB session!
Query tuning

There exist 3 signatures of the `getObjects` method:

```java
getObjects(IQuery query)
getObjects(IQuery query, boolean inMemory)
getObjects(IQuery query, boolean inMemory, int startIndex, int endIndex)
```

The boolean `inMemory` is used by ODB if all objects must created at query time or in a lazy load fashion. If `true`, a collection with all objects already created will be returned.

If `false`, the collection will contain ids of objects: each time you get an object from the list, ODB will create it on the fly.

The default value is true. This option is faster but uses more memory. If you know that a query may return a lot of objects and that you won't need to get all of them, it is a good practice to use `inMemory=false`.

The startIndex and endIndex are used to specify a range of objects that are to be returned. It can be used to cut a query result into various pages. If a query result should return 20000 objects, you can use the `getObjects(query,true,0,10000)` to get the first 10000 objects and `getObjects(query,true,10000,20000)` to get the next 10000. Default values are -1 (which disables query result paging).

Indexes

To speedup the ODB queries, you can declare indexes. Indexes can be declared on various field of a class. Here is an example of index declaration on the class `Sport` for the field `name`:

```java
ODB odb = ODBFactory.open(ODB_NAME);
String [] fieldNames = {"name"};
odb.getClassRepresentation(Sport.class)
    .addUniqueIndexOn("sport-index", fieldNames,true);
```
For instance, **ODB** only supports unique indexes.

**Updating Objects**

**To update an object in ODB, it is necessary to load it first.** This is necessary to let **ODB** know that the object already exists. So the process is to get the object, modify it and the store it back into **ODB**.

```java
public void step12() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);

        IQuery query = new CriteriaQuery(Sport.class, Where.equal("name", "volley-ball"));
        Objects sports = odb.getObjects(query);

        // Gets the first sport (there is only one!) Sport volley = (Sport) sports.getFirst();

        // Changes the name volley.setName("Beach-Volley");

        // Actually updates the object odb.store(volley);

        // Commits the changes odb.close();

        odb = ODBFactory.open(ODB_NAME);

        // Now query the database to check the change sports = odb.getObjects(Sport.class);

        System.out.println("\nStep 12 : Updating sport");
        System.out.println((i + 1) + "	: "+ sports.next());
    }
    finally {
        if (odb != null) {
            // Close the database odb.close();
        }
    }
}
```

Step 8 : Updating sport  
1 : Beach-Volley  
2 : Tennis
Warning: Always remember to retrieve the object before updating it. If an object is not previously loaded from ODB, calling the store method will create a new one!
Deleting objects

There are two ways to delete an object:
1. Getting the object and ask ODB to delete it
2. If you have the id of the object, ask ODB to delete the object with this specific id

Example 1: deleting an object

```java
public void step13() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);
        IQuery query = new CriteriaQuery(Player.class,
                                          Where.like("name", "%Agassi"));

        Objects players = odb.getObjects(query);

        // Gets the first player (there is only one!)
        Player agassi = (Player) players.getFirst();

        odb.delete(agassi);
        odb.close();

        odb = ODBFactory.open(ODB_NAME);
        // Now query the database to check the change
        players = odb.getObjects(Player.class);

        System.out.println("\nStep 13: Deleting players");
        // display each object
        while (players.hasNext()) {
            System.out.println((i + 1) + "\t: " + players.next());
        }
    } finally {
        if (odb != null) {
            // Close the database
            odb.close();
        }
    }
}
```
Example 2: deleting an object using its internal id

```java
public void step14() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);

        // First re-create Agassi player - it has been deleted in step 13
        Player agassi = new Player("André Agassi",
                                 new Date(),
                                 new Sport("Tennis"));
        OID agassiId = odb.store(agassi);
        odb.commit();
        odb.deleteObjectWithId(agassiId);
        odb.close();
        odb = ODBFactory.open(ODB_NAME);

        // Now query the database to check the change
        Objects players = odb.getObjects(Player.class);
        System.out.println("Step 14 : Deleting players");
        // display each object
        while (players.hasNext()) {
            System.out.println((i + 1) + "\t: " + players.next());
        }
    }
    finally {
        if (odb != null) {
            // Close the database
            odb.close();
        }
    }
}
```
Using ODB as Client/Server

ODB can also be used as a client/server database. The first step to do this is to start the ODB server. A server needs some parameters to be created:

- The port on which it must be executed: port that will receive client connections
- The database(s) that must be managed by the server: a server can ‘serve’ more than one database. This is done by using the ‘addBase’ method in which you specify the name of the base and its database file. The name of the base will be used by clients to tell to which base they must be connected.
- The server can be started in the current thread (startServer(false)) or in a background thread (startServer(true))

Here is how to create a Server:

```java
ODBServer server = null;

// Creates the server on port 8000
server = ODBFactory.openServer(8000);

// Tells the server to manage base 'base1' that points to the file tutorial2.odb
server.addBase("base1", ODB_NAME);

// Then starts the server to run in background
server.startServer(true);
```

Then a client must be created. The client must specify the following informations:
- The server host name
- The server port
- The name of the base to connect to

```java
// Open the database client on the localhost on port 8000 and specify which database // instance
odb = ODBFactory.openClient("localhost", 8000, "base1");
```
Here, the client will access the base 'base1' on the server localhost on the port 8000.
Complete example:

```java
public void step20() throws Exception {
    // Create instance
    Sport sport = new Sport("volley-ball");

    ODB odb = null;
    ODBServer server = null;
    try {
        // Creates the server on port 8000
        server = ODBFactory.openServer(8000);
        // Tells the server to manage base 'base1' that points to the file tutorial2.odb
        server.addBase("base1", ODB_NAME);
        // Then starts the server to run in background
        server.startServer(true);

        // Open the database client on the localhost on port 8000 and specify which
        // database instance
        odb = ODBFactory.openClient("localhost", 8000, "base1");

        // Store the object
        odb.store(sport);
    } finally {
        if (odb != null) {
            // First close the client
            odb.close();
        }
        if (server != null) {
            // Then close the database server
            server.close();
        }
    }
}
```

The Client/server mode is in Beta release.
ODBExplorer

ODBExplorer is a tool to:

- Browse objects
- Query objects
- Create objects
- Update objects
- Delete objects
- export/import an ODB Database.
- Refactor the database

To open a database, click on the ODB menu and choose the ‘Open Database’ item then point to the database file you want to open.

Then ODB Explorer displays the meta-model of the database of the left of the window.

Clicking on a class, ODBExplorer displays a contextual menu:
- The ‘Table View’ item, displays data in an sql-like query result
- The ‘Object View’ item displays all objects in a hierarchy mode
- The ‘Query’ item opens a graphical wizard to build a CriteriaQuery
- The ‘New Object’ item opens a window to create a new instance of the specific class
- The Refactoring – Rename class allows renaming the class in the database.
**Browsing data**

To browse a database, simply open the database file. On the left of the screen, the meta-model of the database will be displayed.

Choose a class and a way to display data:
- **Table View**: display the result as a SQL query result.
- **Object View**: display objects as a tree respecting the object model.
If you prefer to see the objects with their recursive structure, then choose the Object Browse mode:

ODBExplorer using **Object View** query result display
Query

The ODBExplorer offers a graphic interface to build a **CriteriaQuery** to query a subset of objects:
**Updating**

It is possible to update data using the ODBExplorer. This can be done only in the ‘Object View mode. When clicking on an objects, the update button will be enabled. Remember to commit or rollback your changes:

1. Right click on the class and choose Object View
2. In the Object View Click on the object to be updated, then click on the update button
3. ODB displays a window with the fields to be updated

When saving the change, Don’t forget to commit to persist the changes.

When it is a relation, The OID of the object is displayed. Click on the Choose Object to modify the relation, or simply digit the new OID.

**Choose an object**

All the objects of type Sport

When an object has an attribute that is another object, click on the ‘Choose the object’ o browse and choose the desired object:'
Creating new objects
It is also possible to create objects using ODBExplorer. Just select a Class on the left panel and click the `New Object` button:

XML

Using ODBExplorer, an entire ODB database can be exported to XML and later imported back.

Export

Open a database, click on the export button and choose the name of the destination XML file.
Import

Choose the import option on the ODB menu, choose the name of the ODB database to import into and then point to the xml file that must be imported.

Via API

Import and export features are also available via API using the XMLExporter and XMLImporter classes:

Exporting data XML using the XMLExporter

```java
public void step15() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open(ODB_NAME);
        // Creates the exporter
        XMLExporter exporter = new XMLExporter(odb);
        // Actually export to current directory into the sports.xml file
        exporter.export(".", "sports.xml");
    } finally {
        if (odb != null) {
            // Close the database
            odb.close();
        }
    }
    System.out.println("\nStep 15 : exporting database to sports.xml");
}
```
public void step16() throws Exception {
    ODB odb = null;

    try {
        // Open a database to receive imported data
        odb = ODBFactory.open("imported-" + ODB_NAME);
        // Creates the exporter
        XMLImporter importer = new XMLImporter(odb);

        // Actually import data from sports.xml file
        importer.importFile(".", "sports.xml");

        // Closes the database
        odb.close();

        // Re open the database
        odb = ODBFactory.open("imported-" + ODB_NAME);
        // Now query the database to check the change
        Objects players = odb.getObjects(Player.class);

        System.out.println("\nStep 16: getting players of imported database");
        // display each object
        while (players.hasNext()) {
            System.out.println((i + 1) + "	: " + players.next());
        }
    }

    } finally {
        if (odb != null) {
            // Close the database
            odb.close();
        }
    }
}
User/Password protection

If you need to protect the access of the database, you can open/create it with a user/password. Once created with a user, it will always be necessary to pass the correct user and password to open the database:

```java
public void step17() throws Exception {
    ODB odb = null;

    try {
        // Open the database
        odb = ODBFactory.open("ODB_NAME_2", "user", "password");

        odb.store(new Sport("Tennis"));
        // Commits the changes
        odb.close();
        try {
            // try to open the database without user/password
            odb = ODBFactory.open("ODB_NAME_2");
        } catch (ODBAuthenticationRuntimeException e) {
            System.out.println("\nStep 17 : invalid user/password : database could not be opened");
        }
        // then open the database with correct user/password
        odb = ODBFactory.open("ODB_NAME_2", "user", "password");
        System.out.println("\nStep 17 : user/password : database opened");
    }
    finally {
        if (odb != null) {
            // Close the database
            // Close the database
            odb.close();
        }
    }
}
```

Best Practices

Open/Close Database

When working with ODB, it is important to call the close method to commit changes. To be sure to do this, it is a good practice to use a try/finally block:
It is also a good practice to put the `ODBFactory.open(ODB_NAME);` code line in a separated class to isolate the opening of the database.

**Transient fields**

Sometimes, classes have fields that are used for processing but do not need to be persisted with the objects. Such fields should be declared as transient to tell `ODB` that they do not need to be persisted.
Advanced Features

All Configuration and tuning are done using the class org.neodatis.odb.core.Configuration.

Multi-thread

For instance, ODB does not support concurrent access yet. But there is a way to use it in multi-thread runtime environment. To do so, it is necessary to inform ODB that you are using multi-thread and specify the thread pool size. This can be done using:

Configuration.useMultiThread(true, <thread pool size>).

Automatic close of ODB Database

Default value is false.

Defragmentation

<to do>

Supported Types

ODB can persist the following language native types:

Java

all primitive types (byte, short, int, long, float, double, char, boolean) and their respective wrapper types (Byte, Short, Integer, Long, Double, Character, Boolean).

And

• BigDecimal
• BigInteger
• java.util.Date
• String
- Collections
- array of primitive types
- array of Objects
- Maps
Annexes

Annex 1 : Xml Exported file of the tutorial ODB base

Meta-model

```xml
<xml version="1.0" encoding="UTF-8"?>
  <odb name="tutorial1.odb" export-date-time="119536303468" max-oid="400">
    <meta-model>
      <class id="3" name="Game" package="org.neo4j.odb.tutorial">
        <attribute id="1" name="when" type="java.util.Date" />
      </class>
      <class id="2" name="Sport" package="org.neo4j.odb.tutorial.Sport" />
      <class id="3" name="Team1" package="org.neo4j.odb.tutorial.Team1" />
      <class id="4" name="Team2" package="org.neo4j.odb.tutorial.Team2" />
      <class id="5" name="result" type="java.lang.String" />
    </class>
    <class id="1" name="Sport" package="org.neo4j.odb.tutorial">
      <attribute id="1" name="name" type="java.lang.String" />
    </class>
    <class id="10" name="Object" package="java.lang" />
    <class id="19" name="Comparable" package="java.lang" />
    <class id="4" name="Team" package="org.neo4j.odb.tutorial.Team">
      <attribute id="1" name="name" type="java.lang.String" />
      <attribute id="2" name="players" type="java.util.Collection" />
    </class>
    <class id="5" name="Player" package="org.neo4j.odb.tutorial">
      <attribute id="1" name="name" type="java.lang.String" />
      <attribute id="2" name="birthdate" type="java.util.Date" />
      <attribute id="3" name="favoriteSport" type="org.neo4j.odb.tutorial.Sport" />
    </class>
  </meta-model>
</odb>
```