NeoDatis ODB

Object Oriented Database For Java and C#

V 1.9 (beta 3)
07/04/2008

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.

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

1 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.

1.1 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.

1.2 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.

1.3 Fast

ODB can store more than 20000 objects per second.

1.4 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.

1.5 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.

1.6 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).

1.7 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)).

1.8 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.

1.9 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.

1.10 Easy to integrate

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

1.11 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)

1.12 License

ODB is distributed under the LGPL license.

2 Wiki

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

3 Download

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

4 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.

5 Distributions

Jar	Size	description
neodatis-odb.jar	520kb	The complete **ODB**
neodatis-odb-rt.jar	420kb	same as neodatis-odb.jar, without **ODBExplorer**
neodatis-odb-rt-lite.jar	327kb	The **ODB** local runtime without **ODBExplorer**, XML import/export, Client/server mode
neodatis-odb-gui.jar	100kb	The **ODBExplorer**

6 How to execute ODB

A single jar (neodatis-odb.jar) is needed to run the **ODB** database.

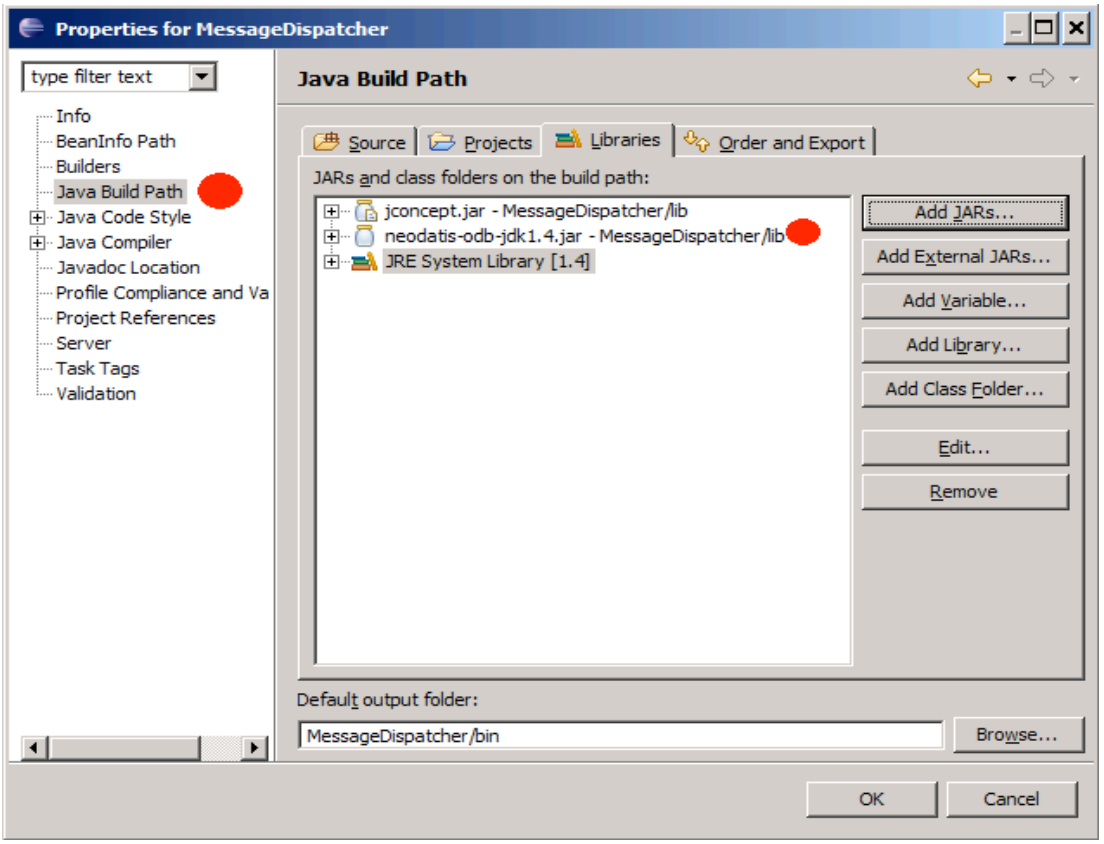
To execute a class that use **ODB** to persist objects, just add the **ODB** runtime to the classpath:

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

7 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:



8 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.

See **Reconnecting Objects to Session** to see how to simply update objects for web application.

9 Migration from previous releases to 1.8

Previous releases include all inclusive 1.8 beta releases.

9.1 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.

9.2 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 Resctrictions class remains available for compatibility issues but it is marked as deprecated.

10 Storing Objects

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:

```
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:

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

And then

```
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:

```
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

```
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

```
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

```
public class Game {
    private Date when;
    private Sport sport;
    private Team team1;
    private Team team2;
    private String result;
}
```

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**

Lets check this, here is 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.

11 Object retrieving

In the previous example, we learned how to store objects. Now, obviously, we need to get theses 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.

11.1 Retrieving all objects of a specific class

The **ODB** interface has a method **getObjects** that receives 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 **Player** and displays each one.

11.2 CriteriaQuery

CriteriaQuery let's you specify 'Where conditions' on objects that the query result must contain. The **ODB CriteriaQuery** API is very close to the **Hibernate** Criteria API.

Here is a simple example of CriteriaQuery:

```
public void step3() throws Exception {
        ODB odb = null;

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

                IQuery query = new CriteriaQuery(Player.class,
                        Where.equal("name", "olivier"));

                Objects players = odb.getObjects(query);

                System.out.println("\nStep 3 : Players with name olivier");

                // display each object
                while(players.hasNext()) {
                        System.out.println((i + 1) + "\t: " + players.next());
                        }
        } finally {
                if (odb != null) {
                        // Close the database
                        odb.close();
                }
        }
}
```

The following code creates a query on objects of type **Player** where the name is equal to *olivier.*

IQuery query = new CriteriaQuery(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:

```
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:

```
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):

```
// 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" works with objects too.

Other functions are available while working with **CriteriaQuery**. Here is a list of functions available in the Where Factory:

equality	
public static ICriterion equal(String attributeName,boolean value)	for primitive boolean value
public static ICriterion equal(String attributeName,int value)	for primitive int value
public static ICriterion equal(String attributeName,short value)	for primitive short value
public static ICriterion equal(String attributeName,byte value)	for primitive byte value
public static ICriterion equal(String attributeName,float value)	for primitive float value

public static ICriterion equal(String attributeName,double value)	for primitive double value
public static ICriterion equal(String attributeName,long value)	for primitive long value
public static ICriterion equal(String attributeName,char value)	for primitive char value
public static ICriterion equal(String attributeName,Object value)	for object
public static ICriterion equal(String attributeName,Object value)	Case insensitive equal.
like	
public static ICriterion like(String attributeName,String value)	for patterns like 'name="pet %"'
public static ICriterion ilike(String attributeName,String value)	with case insensitive option
greater than (gt)	
public static ICriterion gt(String attributeName,Comparable value)	for Comparable objects
public static ICriterion gt(String attributeName,int value)	for primitive int
public static ICriterion gt(String attributeName,short value)	for primitive short
public static ICriterion gt(String attributeName,byte value)	for primitive byte
public static ICriterion gt(String attributeName,float value)	for primitve float
public static ICriterion gt(String attributeName,double value)	for primitive double
public static ICriterion gt(String attributeName,long value)	for primitive long
public static ICriterion gt(String attributeName,char value)	for primitive char
greater or equal (ge)	
public static ICriterion ge(String attributeName,Comparable value)	for Comparable objects
public static ICriterion ge(String attributeName,int value)	for primitive int
public static ICriterion ge(String attributeName,short value)	for primitive short
public static ICriterion ge(String attributeName,byte value)	for primitive byte
public static ICriterion ge(String attributeName,float value)	for primitve float
public static ICriterion ge(String attributeName,double value)	for primitive double
public static ICriterion ge(String attributeName,long value)	for primitive long
public static ICriterion ge(String attributeName,char value)	for primitive char
less than (lt)	
public static ICriterion lt(String attributeName,Comparable value)	for Comparable objects
public static ICriterion lt(String attributeName,int value)	for primitive int
public static ICriterion lt(String attributeName,short value)	for primitive short
public static ICriterion lt(String attributeName,byte value)	for primitive byte
public static ICriterion lt(String attributeName,float value)	for primitve float
public static ICriterion lt(String attributeName,double value)	for primitive double
public static ICriterion lt(String attributeName,long value)	for primitive long
public static ICriterion lt(String attributeName,long value)	for primitive char
less or equal (le)	
public static ICriterion le(String attributeName,Comparable value)	for Comparable objects
public static ICriterion le(String attributeName,int value)	for primitive int
public static ICriterion le(String attributeName,short value)	for primitive short
public static ICriterion le(String attributeName,byte value)	for primitive byte
public static ICriterion le(String attributeName,float value)	for primitve float

public static ICriterion le(String attributeName,double value)	for primitive double
public static ICriterion le(String attributeName,long value)	for primitive long
public static ICriterion le(String attributeName,char value)	for primitive char
contain -To test if an array or a collection contain a specific value	
public static ICriterion contain(String attributeName,boolean value)	for primitive boolean value
public static ICriterion contain(String attributeName,int value)	for primitive int value
public static ICriterion contain(String attributeName,short value)	for primitive short value
public static ICriterion contain(String attributeName,byte value)	for primitive byte value
public static ICriterion contain(String attributeName,float value)	for primitive float value
public static ICriterion contain(String attributeName,double value)	for primitive double value
public static ICriterion contain(String attributeName,long value)	for primitive long value
public static ICriterion contain(String attributeName,char value)	for primitive char value
public static ICriterion contain(String attributeName,Object value)	for object
Null Objects	
public static ICriterion isNull(String attributeName)	only null objects
public static ICriterion isNotNull(String attributeName)	only not null objects
Where on collection or array size	
public static ICriterion sizeEq(String attributeName,int size)	with a size equal to
public static ICriterion sizeNe(String attributeName,int size)	with a size not equal to
public static ICriterion sizeGt(String attributeName,int size)	with a size greater than
public static ICriterion sizeGe(String attributeName,int size)	with a size greater or equal
public static ICriterion sizeLt(String attributeName,int size)	with a size lesser than
public static ICriterion sizeLe(String attributeName,int size)	with a size lesser or equal
Logical	
public static ComposedExpression or()	a OR expression
public static ComposedExpression and()	a AND expression
public static IExpression not(ICriterion criterion)	negate a restriction

More Criteria Query Examples:

```
// 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 doas 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);
aq.orderByDesc("boolean1,int1");
```

11.3 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 peace 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 does 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:

```
public boolean match(Player player)
```

A native query that return all players whose favorite sport's name(transformed to lower case) starts with 'volley':

```
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
		whil (players.hasNext()) {
			System.out.println((i + 1) + "\t: " + players.next());
			}
		} finally {
			if (odb != null) {
				// Close the database
				odb.close();
			}
		}
	}
```

Warning: *In client/Server mode, native query are executed on the server. So the ODB server must be started with the Native Query (and its dependencies) in its classpath and the Native Query class must implement the Serializable interface as ODB use serialization to send the Native Query object to the server.*

11.4 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!

11.5 Query tuning

There exist 3 signatures of the **getObjects** method:

- 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).

11.5.1 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':

```
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.

11.6 Object Values API

Object Values API breaks the object paradigm providing direct access to the values of the attributes of the objects and aggregate functions like Sum,Average, Min, Max, Count and Group by. While it is still a work in progress (API is available in the current development release) the Object Values API already demonstrated its flexibility and performance versus pure object query languages.

The NeoDatis ODB Object Values API provides:

- Direct access to the values of an object
- Dynamic Views : Navigation through relations capability
- Aggregate functions (Sum, Average, Min, Max , Group by and Count)
- Custom functions

This leverages the flexibility of SQL language to an Object Oriented Database. Object Values API is a query layer and does not change anything to the object model nor impose restrictions on Objects. It does not require any specific mapping.

In the following sections, we will be using three classes as examples:
1.Class Function : with a name attribute (String)
2.Class Profile : with a name attribute (String) and a list of Functions
3.Class User : with a name(String), an email(String), number of login(integer) and profile (Profile)

11.6.1 AGGREGATE FUNCTIONS

An aggregate function is a function that performs a computation on a set of values rather than on a single value. Object Values API currently supports the following functions:

Method	Description
IValuesQuery.**sum**(String fieldName)	Calculates the sum of all the fields <fieldName> that satisfy the query
IvaluesQuery.**avg**(String fieldName)	Calculates the average of all the fields <fieldName> that satisfy the query
IvaluesQuery.**count**(String alias)	Counts the number of object that satisfy the query
IvaluesQuery.**min**(String fieldName)	Retrieves the minimum value of the specific field that satisfy the query
IvaluesQuery.**max**(String fieldName)	Retrieves the maximum value of the specific field that satisfy the query
IvaluesQuery.**groupBy**(String fieldNames)	Execute the query group by the fields
IValuesQuery.(String fieldName, int fromIndex, int size)	The sublist Object Values API method returns a sublist of a list attribute. The sublist returned is a lazy loading list.
IValuesQuery.**sublist**(String fieldName, int fromIndex, int toIndex)	Returns a sublist of a list attribute
IValuesQuery.**size**(String fieldName)	The size method is used to retrieve the size of a collection. It is only applicable to collection attributes. This is done without actually loading all the objects of the list

The following paragraph demonstrates the use of aggregate functions by example and comparing with the standard Sql version of the query

11.6.1.1 Sum function

Retrieving the total sum of logins

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class).**sum**("nbLogins")
SQL	select sum(nbLogins) from User

11.6.1.2 Average function

Retrieving the average number of logins

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class).**avg**("nbLogins")
SQL	select avg(nbLogins) from User

11.6.1.3 Minimum and Maximum

Retrieving the minimum and maximum number of logins

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class) .**min**("nbLogins","min of nbLogins") .**max**("nbLogins","max of nbLogins");
SQL	select min(nbLogins) , max(nbLogins) from User

11.6.1.4 Counting values

O b j e c t Values	odb.getValues(new ValuesCriteriaQuery(User.class).**count**("nb users");
SQL	select count(*) from User

11.6.1.5 Group by

Retrieving the name of the profile, the number of user for that profile and their average login number grouped by the name of the profile

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class) .field("profile.name") .count("count") .avg("nbLogins","avg") .**groupBy**("profile.name"));
SQL	select p.name, count(u.*), avg(u.nbLogins) from User u, Profile p where u.profile_id = p.id group by p.name

11.6.1.6 Sublist & iist size

```
IValuesQuery q = new ValuesCriteriaQuery(Profile.class)
.field("name")
.sublist("functions", 1, 2, false)
.size("functions", "fsize");
```

Beyond providing aggregate functions, Object Values API also provides direct access to the attributes of the objects. It may be very useful when only partial data are needed (to build report for example) or when a very high volume of data is expected.

This returns result that have three elements:

- element 1: the name of the profile

- element 2 : a sublist of the list starting at index 1 and size 2

- element 3 : The size of the whole list.

The returned sublist is a lazy-loading list.

The API does not return objects, it returns a list of Map : each map contains the requested values.

Example: Getting only the names of the users

```
Values values = odb.getValues(
        new ValuesCriteriaQuery(User.class).field("name"));
```

SQL Equivalent: select name from User

Dynamic Views

A very interesting part of the Object Values API is the capability to directly navigate into object relations to get the necessary information. This feature is called **Dynamic Views**.

To use Dynamic Views, instead of specifying the field name, the complete relation name to reach the attribute is required:For example, to get the name of the profile of a User, the complete relation name would be : profile.name.

Example: getting the name of the users and their profile names:

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class) .field("name"). **field**("profile.name"));
SQL	select u.name, p.name from User u, Profile p where u.profile_Id = p.id

The relation navigation capability can also be used to restrict query results:

Object Values	odb.getValues(new ValuesCriteriaQuery(User.class,Where.equals("profile.name","profile 1")) .field("name") .field("profile.name"));
SQL	select u.name, p.name from User u, Profile p where u.profileId = p.id and p.name='profile 1'

Dynamic views provide the same facility as SQL Joins for semantic relations.

11.6.2 Custom Functions

When a specific function is not available on the current Object Values API, it is possible to implement it to compute the necessary information. This is done by extending the **CustomQueryFieldAction** class. This implementation is done using the native programming language (Java or C#.net).

The class CustomQueryFieldAction has four methods that need to be implemented:

void execute(final OID oid, final AttributeValuesMap values)	The main method that will do the calculations. It receives the Object ID and the requested attribute values
public Object getValue()	A method to get the result of the calculation
public boolean isMultiRow()	To indicate if the calculation return a single value or one value per object
public void start()	A method called by the query processor at the beginning of the query execution
public void end()	A method called by the query processor at the end of the query execution

Here is a simple implementation example that computes the number of login of the user that are currently logged in. To demonstrate the power of custom actions, the logged user information is not retrieved from database but from an external java class.

```
public class TestCustomQueryFieldAction2 extends CustomQueryFieldAction {

	/** The number of logins */
	private long nbLoggedUsers;

	public TestCustomQueryFieldAction2() {
		this.nbLoggedUsers = 0;
	}

	/** The method that actually computes the logins */
	public void execute(final OID oid, final AttributeValuesMap values) {
		// Gets the name of the user
		String userName = (String) values.get("name");

		// Call an external class (Users) to check if the user is logged in
		if (Sessions.isLogged(userName)) {
			nbLoggedUsers++;
		}
	}

	public Object getValue() {
		return new Long(nbLoggedUsers);
	}

	public boolean isMultiRow() {
		return false;
	}

	public void start() {
		// Nothing to do
	}

	public void end() {
		// Nothing to do
	}
}
```

And here is how you use this custom function:

```
CustomQueryFieldAction customAction = new TestCustomQueryFieldAction();

Values values = odb.getValues(new ValuesCriteriaQuery(Users.class)
		.custom("nbLogins", "nb logged users",customAction)
		.field("name"));
```

This is a simple way to build a custom query in which it is possible to interact with any other systems.

12 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 then store it back into **ODB**.

```
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 databas eto check the change
		sports = odb.getObjects(Sport.class);

		System.out.println("\nStep 12 : Updating sport");

		// display each object
		while (sports.hasNext()) {
			System.out.println((i + 1) + "\t: " + sports.next());
		}
	} finally {
		if (odb != null) {
			// Close the database
			odb.close();
		}
	}
}
```

The output of Sport listing is:

```
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!**

13 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

```
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 databas eto 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

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

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

		// Firts 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 databas eto check the change
		Objects players = odb.getObjects(Player.class);

		System.out.println("\nStep 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();
		}
	}
}
```

14 Using ODB as Client/Server

ODB can also be used as a client/server database and has two different Client/Server modes:

- Traditional Client/Server where Clients and Server run in a different Virtual machine
- Optimized Client/Server mode where Clients and Server run in the same Virtual Machine (version 1.9+). Useful when using **ODB** in a Web application. This mode is much more faster.

The first thing 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 that 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:

```
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. There are two ways to create a client. If the client will run in the same virtual machine than the server (if your developing a web application, for example), you can create a client from the server instance like this:

```
// Open the database client
ODB odb = server.openClient("base1");
```

If the client will run in another virtual machine, then the following API must be used:

```
// 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**.

The API to interact with the two types of clients is exactly the same.

The first one, using a same virtual machine, is faster because the communication between the client server is optimized: it does not use net IO.

Complete example:

```
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();
		}
	}
}
```

15 Reconnecting Objects to Session

When opening a NeoDatis base, all objects that are stored or selected from the ODB instance are connected to the session. Sometimes, you may need to reconnect object (loaded in a previous session) to a newly opened session. This is very common when using NeoDatis in a web application for example. You may keep the object in your session, and then when you want to update it, you should reload the object first to be able to update it. A shortcut to this is to reconnect the object to the session by calling odb.reconnect(object). This will automatically connect the object the new session enabling a direct store.

Warning: This feature uses byte code instrumentation. To enable it, you have to start you application using an agent :

```
java -javaagent:neodatis-odb.jar=<packages> ...
```

The <packages> tag is the list of packages of the classes that will be stored in ODB. This is a way to tell NeoDatis which classes must be instrumented.

Example:

```
java -javaagent:neodatis-odb.jar=org.neodatis.vo1,org.neodatis.vo2 org.neodatis.test.MyTest
```

In this example, we tell NeoDatis to instrument all objects that are in the packages org.neodatis.vo1 and org.neodatis.vo2.

16 ODBExplorer

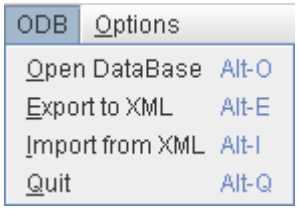
ODBExplorer is a tool to :

- Browse objects
- Query objects

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

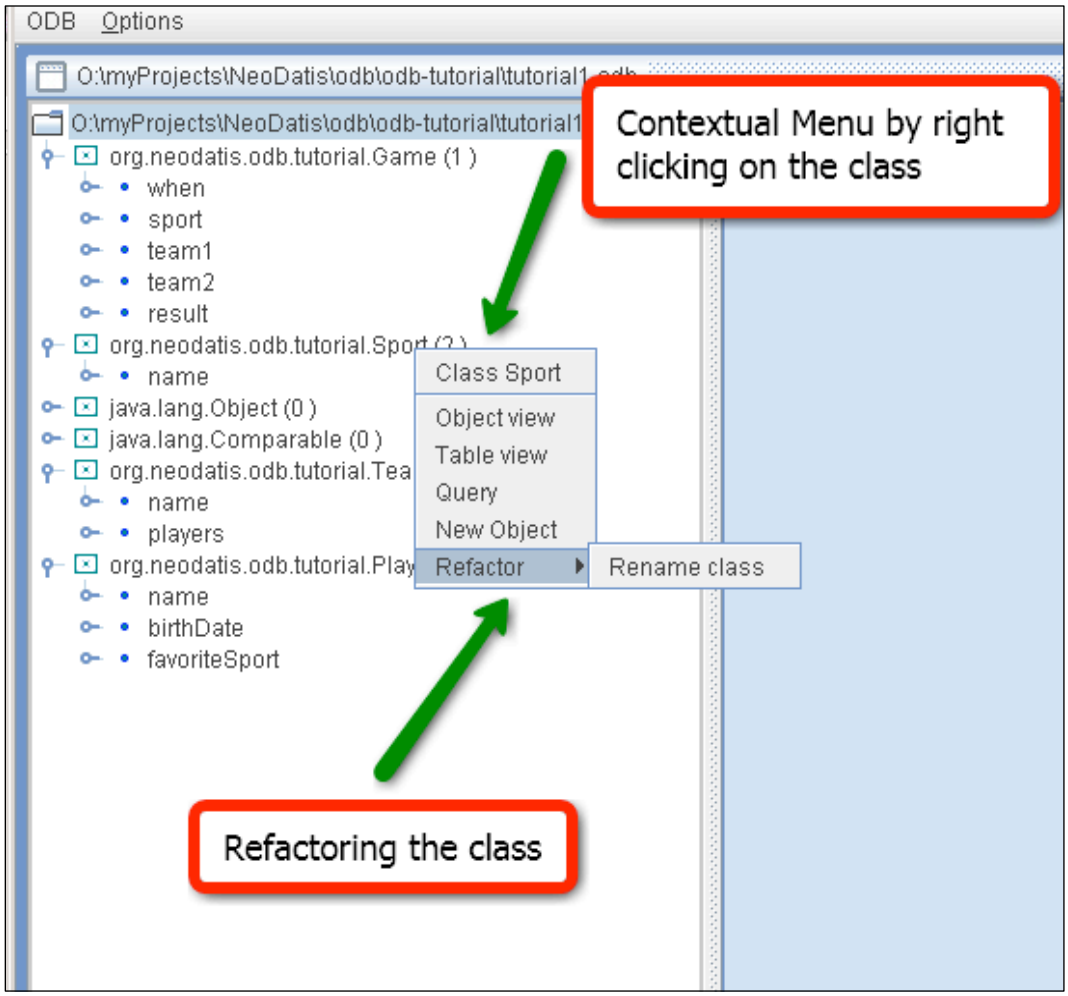
To start ODB Explorer just execute the neodatis-odb.jar (java –jar neodatis-odb.jar).

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:



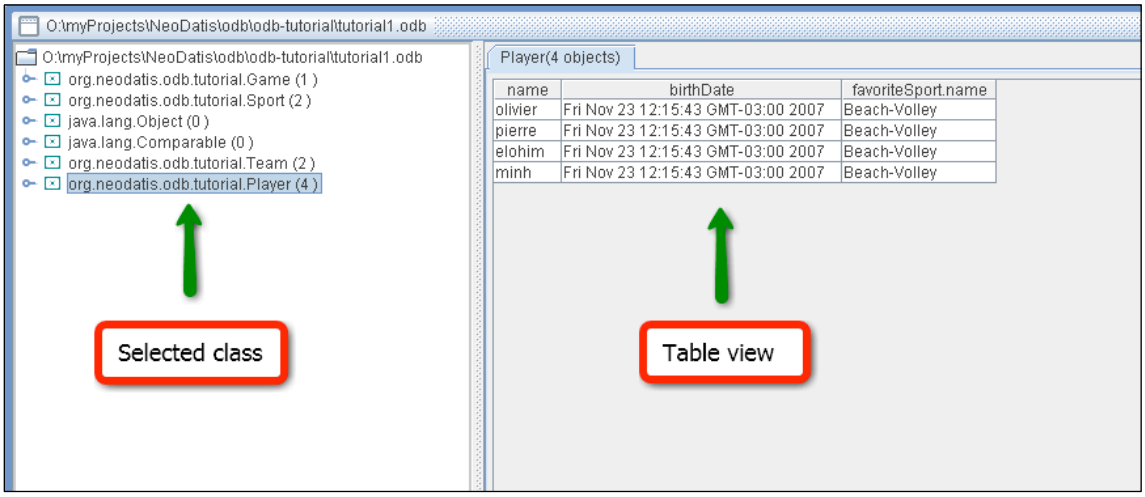
- The 'Table View' item, displays data in an sql-like query result
- The 'Object View' item displays all objects in a hierarchy mode
- The 'Query' item opens a graphical wizard to build a CriteriaQuery
- The 'New Object' item opens a window to create a new instance of the specific class
- The Refactoring – Raname class allows renaming the class in the database.

16.1 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.

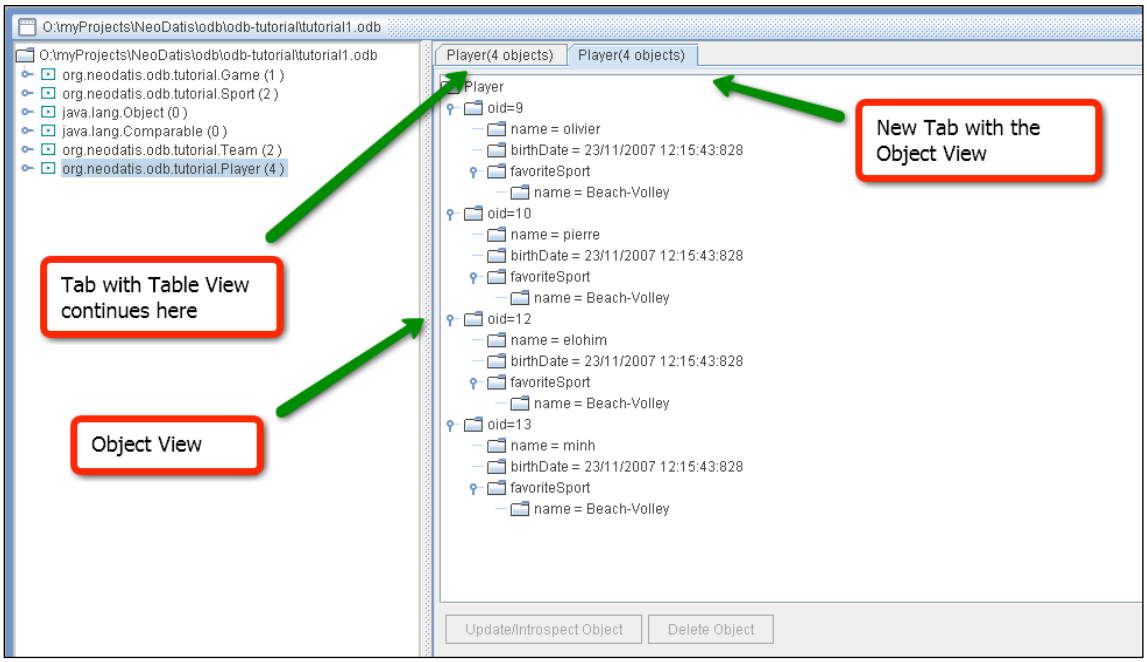
Choose a class and a way to display data:

- Table View: display the result as a SQL query result.
- Object View: display objects as a tree respecting the object model.



ODBExplorer using Table View query result display

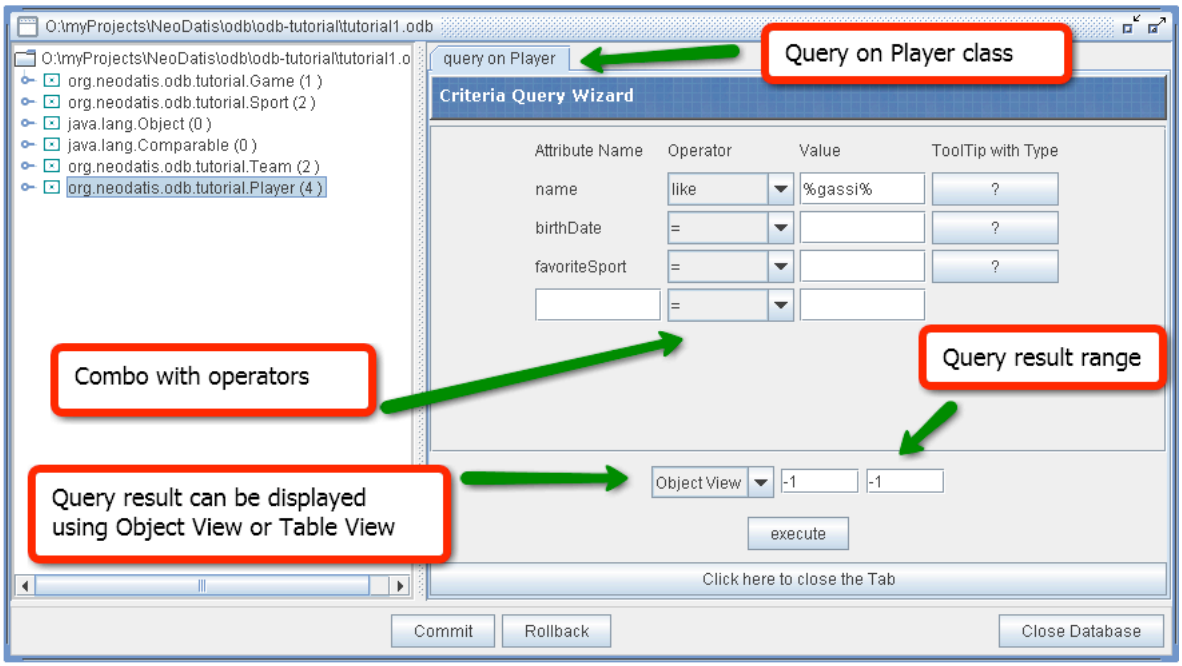
If you prefer to see the objects with their recursive structure, then choose the Object Browse mode:



ODBExplorer using Object View query result display

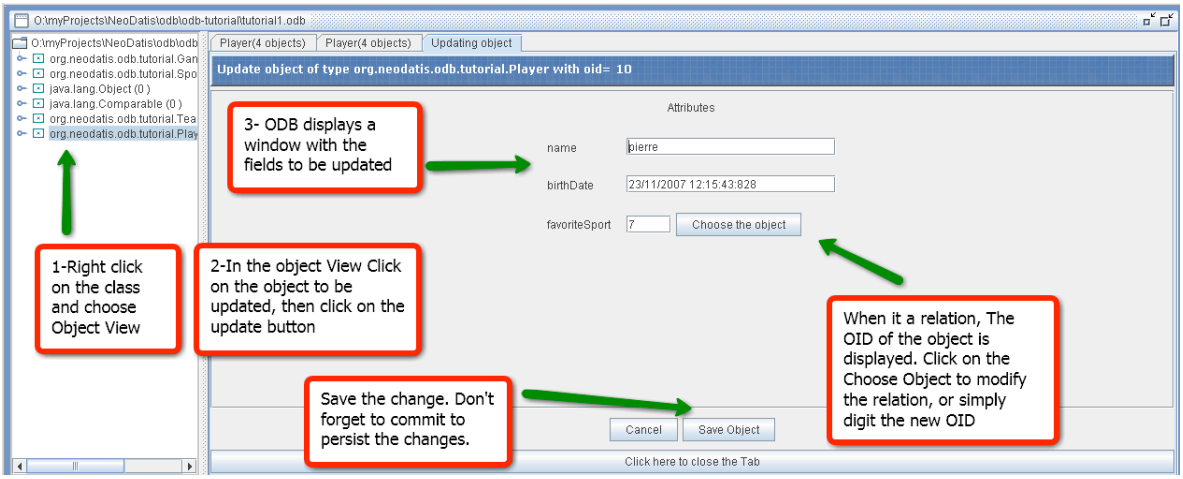
16.2 Query

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

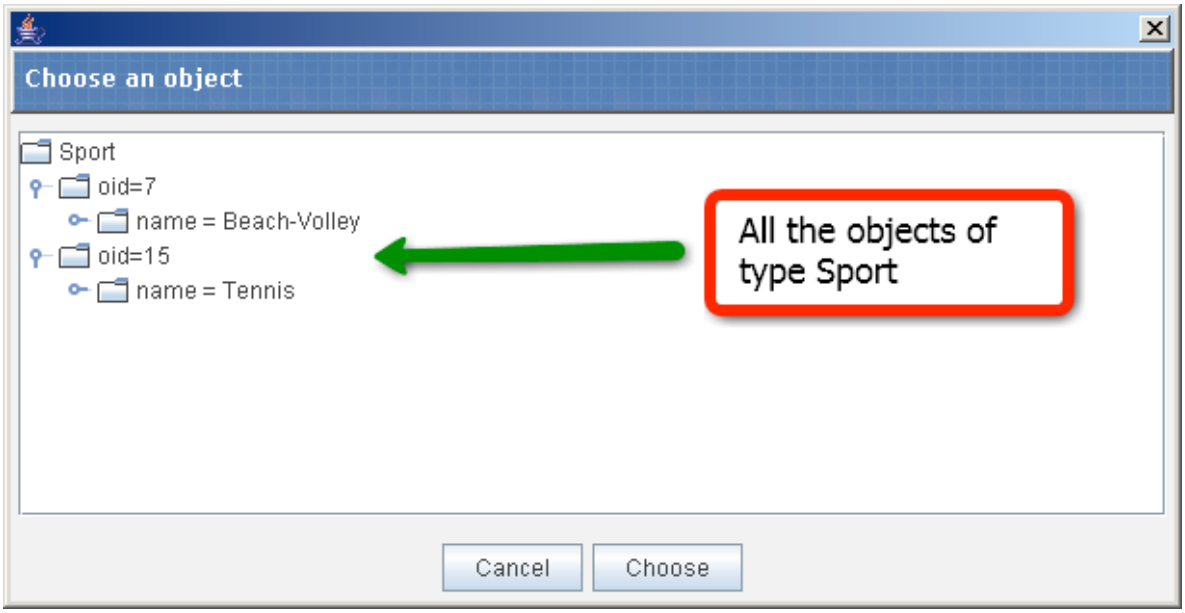


16.3 Updating

It is possible to update objects 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!:

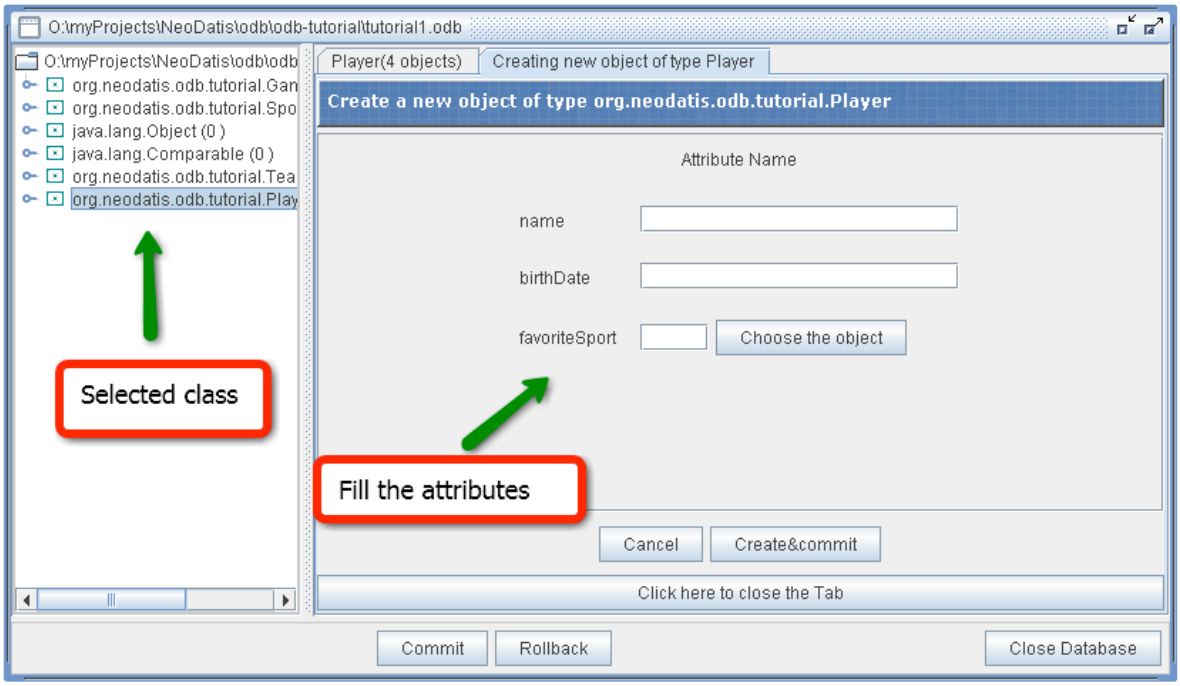


When an object has an attribute that is another object, click on the 'Choose the object' o browse and choose the desired object:'



16.4 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:



17 XML

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

17.1 Export

```
        ODB odb = null;

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

                // work with odb
                // …

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

Choose the export option on the **ODB** menu, choose the name of the **ODB** database to export and then point to the xml file that must be udes to sae the export.

17.2 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.

17.3 Via API

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

17.3.1 Exporting data XML using the XMLExporter

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

17.3.2 Importing data from 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 databas eto 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) + "\t: " + players.next());
		}
	} finally {
		if (odb != null) {
			// Close the database
			odb.close();
		}
	}
}
```

18 NeoDatis Extended API

The extends API provides some advanced functions like:

- getLastTransactionId() : To get the last transaction id of the database
- getObjectVersion(OID oid) : to get the version of an object with the specific OID
- getObjectCreationDate(OID oid) : to get the creation date of an object with the specified OID
-

19 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:

```
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
			odb.close();
		}
	}
}
```

20 Best Practices

20.1 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:

```
        ODB odb = null;

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

                // work with odb
                // …

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

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.

20.2 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.

21 Advanced Features

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

21.1 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>).

21.2 Automatic close of ODB Database

Default value is false.

21.3 Defragmentation

<to do>

21.4 Classes without default constructor

ODB uses Reflection to create objects. Sometimes some classes may not have empty constructors. In this case, when ODB creates an instance, it tries to instantiate without calling any constructor.

In other cases, a default constructor exists but it may need some specific data to be executed successfully. To resolve this, ODB has 2 interfaces that may be implemented to help ODB instantiate objects:

- ParameterHelper
- InstanceHelper

The parameterHelper interface may be used to help ODB with the right data to execute a constructor.

The InstanceHelper interface may be used to help ODB create the instance.

Example:

The "No Calling Constructor" feature is enabled by default. To enable/disable, use Configuration.setEnableEmptyConstructorCreation(true/false);

22 Supported Types

ODB can persist the following language native types:

22.1 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

23 Annexes

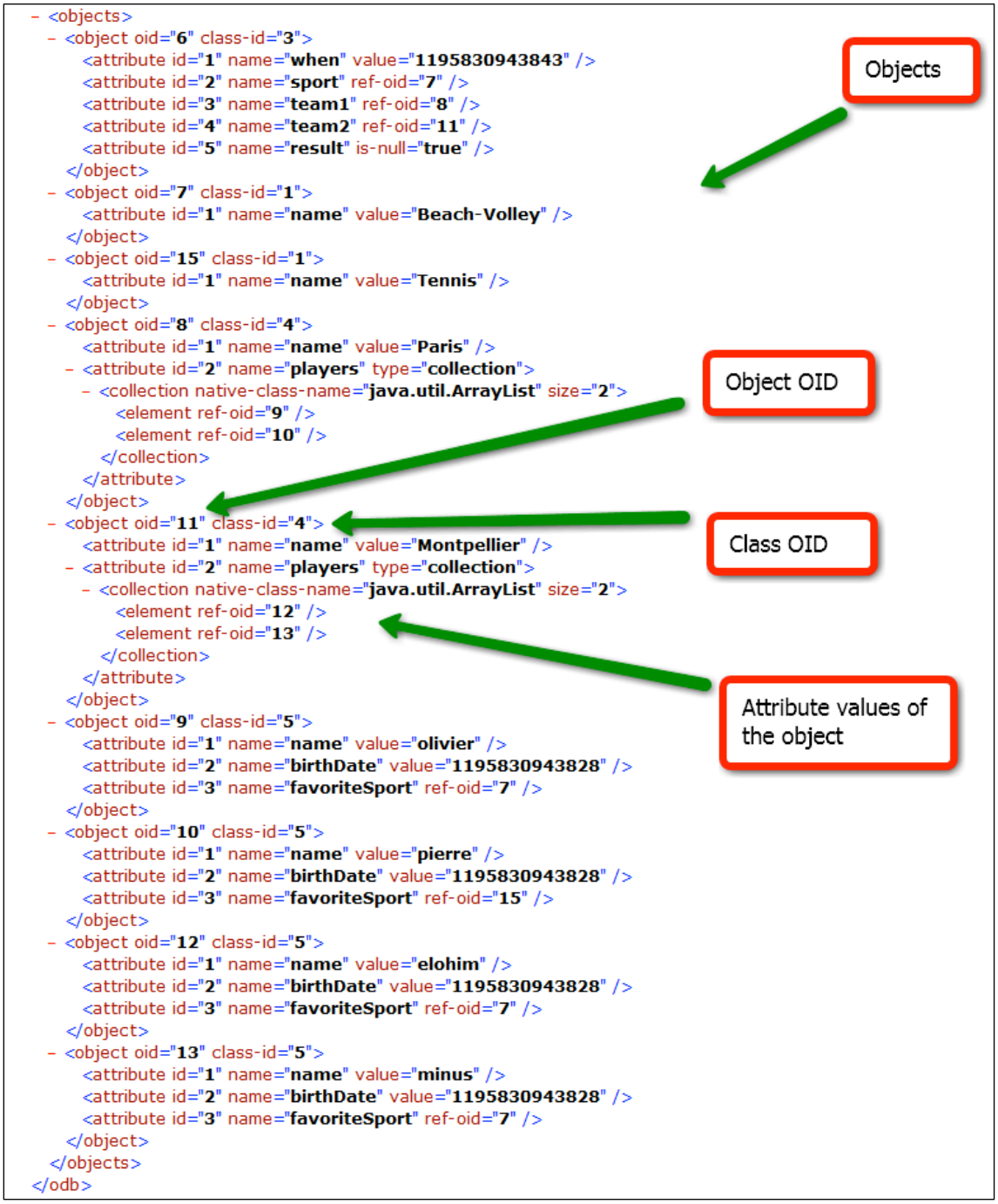
23.1 Annex 1 : Xml Exported file of the tutorial ODB base

Meta-model

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <odb name="tutorial1.odb" export-date-time="1195836303468" max-oid="40">
  - <meta-model>
    - <class id="3" name="Game" package="org.neodatis.odb.tutorial">
        <attribute id="1" name="when" type="java.util.Date" />
        <attribute id="2" name="sport" type="org.neodatis.odb.tutorial.Sport" />
        <attribute id="3" name="team1" type="org.neodatis.odb.tutorial.Team" />
        <attribute id="4" name="team2" type="org.neodatis.odb.tutorial.Team" />
        <attribute id="5" name="result" type="java.lang.String" />
      </class>
    - <class id="1" name="Sport" package="org.neodatis.odb.tutorial">
        <attribute id="1" name="name" type="java.lang.String" />
      </class>
      <class id="18" name="Object" package="java.lang" />
      <class id="19" name="Comparable" package="java.lang" />
    - <class id="4" name="Team" package="org.neodatis.odb.tutorial">
        <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.neodatis.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.neodatis.odb.tutorial.Sport" />
      </class>
    </meta-model>
```

Meta model

Objects




```
- <objects>
  - <object oid="6" class-id="3">
      <attribute id="1" name="when" value="1195830943843" />
      <attribute id="2" name="sport" ref-oid="7" />
      <attribute id="3" name="team1" ref-oid="8" />
      <attribute id="4" name="team2" ref-oid="11" />
      <attribute id="5" name="result" is-null="true" />
    </object>
  - <object oid="7" class-id="1">
      <attribute id="1" name="name" value="Beach-Volley" />
    </object>
  - <object oid="15" class-id="1">
      <attribute id="1" name="name" value="Tennis" />
    </object>
  - <object oid="8" class-id="4">
      <attribute id="1" name="name" value="Paris" />
    - <attribute id="2" name="players" type="collection">
      - <collection native-class-name="java.util.ArrayList" size="2">
          <element ref-oid="9" />
          <element ref-oid="10" />
        </collection>
      </attribute>
    </object>
  - <object oid="11" class-id="4">
      <attribute id="1" name="name" value="Montpellier" />
    - <attribute id="2" name="players" type="collection">
      - <collection native-class-name="java.util.ArrayList" size="2">
          <element ref-oid="12" />
          <element ref-oid="13" />
        </collection>
      </attribute>
    </object>
  - <object oid="9" class-id="5">
      <attribute id="1" name="name" value="olivier" />
      <attribute id="2" name="birthDate" value="1195830943828" />
      <attribute id="3" name="favoriteSport" ref-oid="7" />
    </object>
  - <object oid="10" class-id="5">
      <attribute id="1" name="name" value="pierre" />
      <attribute id="2" name="birthDate" value="1195830943828" />
      <attribute id="3" name="favoriteSport" ref-oid="15" />
    </object>
  - <object oid="12" class-id="5">
      <attribute id="1" name="name" value="elohim" />
      <attribute id="2" name="birthDate" value="1195830943828" />
      <attribute id="3" name="favoriteSport" ref-oid="7" />
    </object>
  - <object oid="13" class-id="5">
      <attribute id="1" name="name" value="minus" />
      <attribute id="2" name="birthDate" value="1195830943828" />
      <attribute id="3" name="favoriteSport" ref-oid="7" />
    </object>
  </objects>
</odb>
```